Stateful application migration in geo-distributed systems

Paulo Ricardo Rodrigues de Souza Junior

DOCTORAT BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Paulo Ricardo RODRIGUES DE SOUZA JUNIOR

« Stateful Application Migration in Geo-Distributed Systems »

Thèse présentée et soutenue à Rennes, le 23 Novembre 2022
Unité de recherche : IRISA (UMR 6074)

Rapporteurs avant soutenance :

Pierre SENS, Professeur des Universités, LIP6 / Sorbonne Université
Gaël THOMAS, Professeur, Telecom SudParis

Composition du Jury :

Président :	Anne-Cécile ORGERIE	Directrice de Recherche, CNRS
Examinateurs :	Pierre SENS	Professeur des Universités, LIP6 / Sorbonne Université
	Gaël THOMAS	Professeur, Telecom SudParis
	Françoise SAILHAN	Professeure, IMT Atlantique
	Federico FACCA	Chief Technology Officer, Martel Lab
Dir. de thèse :	Guillaume PIERRE	Professeur des Universités, Université de Rennes 1
Co-dir. de thèse :	Daniele MIORANDI	Chief Executive Officer, U-Hopper srl

This work is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 765452. The information and views set out in this publication are those of the author(s) and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein.

Acknowledgement

Firstly, I would like to express my sincerest gratitute to my advisors Guillaume Pierre and Daniele Miorandi, for their continuous guidance and encouragement throughout my Ph.D. journey. For the long discussion sessions and the most diverse point of views that broaden my way of seeing the world. I am profoundly grateful to all of you for your patience, advice and immense support in my personal and professional growth and in the development of this thesis. It was such a honor to work with you.

Many thanks to all the jury members: Anne-Cécile Orgerie, Pierre Sens, Gaël Thomas, Françoise Sailhan and Federico Facca. For accepting to review my thesis and providing valuable discussions and meaningful feedback. I would like to extend gratitute to the members of my CSID committee: Olivier Barais and Subarna Chatterjee who followed the development of this work and provided extremely important discussions and suggestions to the accomplishment of this thesis.

I would like to thank all my colleagues from the FogGuru project and the Myriads Team, for the amazing memories, the help with technical and non-technical challenges I faced during my whole journey. It was such a pleasure to work with you, learn from you and with you. Especially to Dimi, Nena, Ali, Mehdi, Hamid, Julie, Genc, Cristhyne, Felipe and Mozhdeh, the friends I made for life and I will not ever forget.

I have a deep sense of gratitude as well to the organizations that received me during this journey. To Las Naves and U-hopper for being so welcoming, for sharing and contributing to my professional and personal career, and easing my stay in each of these foreign countries.

I am thankful as well to all the friends I have made during this moment of my life, moving between three different countries is not easy and I am eternefully grateful to the very good friends and their support and the amazing moments they shared with me during this journey. A todos mis amigos de España que compartieron las mas increíbles aventuras que he tenido en Europa, principalmente a Mau, una de las personas mas importantes de mi vida, que llegó de manera inesperada y me apoyó durante los momentos que mas necesité. Anche a tutti i miei amici italiani, mes amis français et a todos meus amigos brasileiros, eu sou eternamente grato.

Most importantly, this whole dream would not be possible without the support of my family. For sharing this great dream with me and giving the whole support I needed throughout this process. For being so understanding and caring with my goals, even though, I was so far from home. Especially to my parents, Paulo Ricardo Rodrigues de Souza and Luciana Alves da Silva, for their unconditional love and support. In memoria of my father, who taught and inspired me to chase all my dreams and not be afraid of the unexpected. I know, if he was here now, he would have been the proudest. Without them I would not have been able to accomplish all this.

Résumé

Ces dernières années ont vu une prolifération de données produites par de nombreuses sources, telles que des capteurs ubiquitaires. Ces données doivent être traitées pour générer de la valeur et, dans de nombreux cas, il est essentiel de réduire le délai avant que des informations exploitables soient générées à partir de données d'entrée en temps réel. Cela a conduit au développement d'une large gamme d'applications sensibles au délai couvrant de nombreux secteurs économiques tels que la gestion des villes intelligentes, l'éducation, l'armée ou le commerce de détail.

Les fournisseurs cloud facilitent le développement et l'opération d'applications en ligne complexes en mettant à disposition des ressources publiques pour une acquisition de données à la demande. Le cloud computing se compose de serveurs nombreux et puissants, interconnectés par des liaisons réseau à haut débit. Il permet le traitement des données dans une infrastructure stable et garantit la qualité de prestation de services.

Cependant, les ressources des fournisseurs cloud sont souvent centralisées dans un petit nombre de centres de données et garantissent uniquement une communication à faible latence en interne. Par conséquent, il peut y avoir un manque dans la communication longue distance des utilisateurs avec le centre de données de leur fournisseur cloud. Cela implique souvent une plus grande latence de communication pour les applications dans le cloud, ne répondant par conséquent pas aux exigences des applications exigeantes sensibles à la latence.

En réponse à cette lacune, le fog computing vise à réduire la distance et la latence réseau entre les instances d'application et les utilisateurs qui les utilisent. Dans le fog computing, les ressources sont géo-distribuées et localisées à proximité des utilisateurs. Par conséquence, le fog computing fournit une communication à faible latence, permettant aux demandes d'être entièrement ou partiellement traitées localement plutôt que dans un centre de données lointain. D'autre part, les ressources de fog computing sont souvent composées de machines moins puissantes que celles présentes dans le cloud, ce qui signifie qu'il y a une limite à leur capacité de traitement et de stockage.

Le modèle décentralisé du fog ne vise pas à remplacer entièrement l'usage du cloud. Au contraire, il complète l'offre du cloud et multiplie les avantages de son utilisation.

Par exemple, il peut réduire la charge arrivant sur un service hébergé dans le cloud en prétraitant les demandes entrantes avant qu'elles n'arrivent dans le cloud.

Pour répondre aux exigences de faible latence, les instances d'application doivent être placées sur un nœud proche des utilisateurs cibles. L'idée est de réduire le nombre de sauts de réseau nécessaires pour accéder au service d'application afin de réduire la latence de bout en bout au maximum. Toutefois, assurer la proximité lors de la création des ressources est insuffisant car il est possible que les utilisateurs se déplacent ensuite vers différents endroits. Si aucune mesure n'est prise, la mobilité des utilisateurs pourrait entraîner, une fois de plus, une augmentation de la longueur du chemin du réseau pour accéder aux services, ce qui pourrait contrecarrer l'objectif initial de colocaliser l'application dans le voisinage des utilisateurs.

La mobilité des utilisateurs présente un défi pour le fog car la mobilité peut compromettre l'un de ses objectifs fondamentaux, la faible latence. Afin de surmonter ce défi, la migration d'applications se présente comme un candidat puissant pour déplacer les applications en cours d'exécution à proximité de leurs utilisateurs. Ce processus doit se dérouler de manière transparente sans compromettre l'exécution de l'application. Il doit conserver l'intégralité de l'état de l'application (en mémoire et sur disque), maintenir les connexions réseau ouvertes et réduire autant que possible la durée d'interruption de service pendant la migration.

La migration est une opération courante dans les environnements cloud et est souvent effectuée sur des machines virtuelles. Elle accorde de nombreux avantages à l'environnement cloud en améliorant l'utilisation, la disponibilité et la résilience des ressources. Cependant, cela se fait généralement à l'intérieur d'un même centre de données en s'appuyant sur des ressources puissantes et des réseaux dédiés.

De son côté, le fog repose sur un ensemble limité de ressources et de bande passante réseau. Par conséquent, la migration à l'intérieur d'une plate-forme fog ne peut pas compter sur des ressources abondantes. Il est donc nécessaire de concevoir un nouvel ensemble d'approches pour cet environnement et de reconsidérer la conception des techniques de migration géo-distribuée.

La technologie des conteneurs est devenue populaire pour faciliter et gérer le déploiement des applications et des services. Le conditionnement des logiciels dans des conteneurs légers et isolés présente des avantages significatifs du point de vue de la gestion des opérations informatiques. Pour cette raison, les conteneurs sont devenus un bon candidat pour effectuer la gestion et le déploiement des services dans les infrastructures

fog. Néanmoins, lorsqu'il s'agit de déploiements complexes, couvrant des dizaines voire des centaines de machines différentes exécutant des milliers de conteneurs, la conteneurisation à elle seule peut ne pas suffire. C'est ce qui a conduit à la création de systèmes d'orchestration de conteneurs tels que Kubernetes.

Les systèmes d'orchestration de conteneurs peuvent automatiser, dans une large mesure, le déploiement, la mise à l'échelle et les opérations des conteneurs sur des ensembles de serveurs, réduisant ainsi les erreurs humaines et économisant du temps et de l'argent. Cependant, conçus pour des environnements cloud "traditionnels" (c'est-à-dire de grands centres de données avec des machines proches connectées par des réseaux à haut débit), des systèmes tels que Kubernetes présentent certaines limites dans les environnements géo-distribués car les conteneurs ne sont pas destinés à migrer et sauter de serveur en serveur.

Cette thèse explore la migration de pods à grande échelle, en déplaçant les conteneurs avec leurs volumes de données, grâce à un ensemble d'outils pour atteindre cet objectif. La combinaison de ces outils permet la migration transparente des services d'un point A à un point B sans encourir de perte de données ni interférer avec l'accès de l'utilisateur à l'application. Dans le même temps, l'outil de migration gère les connexions réseau et redirige de manière transparente les requêtes des applications vers le nouvel ensemble de ressources sans interrompre les connexions existantes. Toutes les contributions sont implémentées sur Kubernetes et sont évaluées dans un banc d'essai fog réaliste.

Première contribution: migration de la mémoire des pods Kubernetes

Cette contribution présente MyceDrive, une solution transparente de migration de pods pour Kubernetes géo-distribué. Un pod Kubernetes est une unité de déploiement contenant un ou plusieurs conteneurs. Afin de déplacer un pod Kubernetes vers un nouveau serveur, MyceDrive effectue une migration à état en récupérant l'intégralité de l'état en mémoire ainsi que les images de conteneur.

Mycedrive crée un cliché des processus en cours d'exécution à l'intérieur des conteneurs et gère la reconfiguration du réseau en configurant une route temporaire pour rediriger les requêtes entrantes vers le conteneur migré jusqu'à ce que le conteneur soit correctement établi.

MyceDrive évite de mettre fin à un pod en cours d'exécution et d'attendre que Kubernetes en redémarre un nouveau. Par conséquent, l'application s'exécute à l'intérieur du pod et les utilisateurs ne sont pas au courant de la migration.

Nous montrons que la migration géo-distribuée des pods Kubernetes est réalisable tout en restant totalement transparente pour l'application migrée ainsi que pour ses clients, tout en réduisant le temps d'arrêt jusqu'à 7 fois par rapport aux solutions actuelles.

Deuxième contribution : Migration incrémentale des volumes

Dans la deuxième contribution, nous proposons un outil de migration de volumes Kubernetes. Cette contribution complète la première en permettant la migration de volumes dans les pods.

Le déploiement d'un service comprend à la fois un pod et les volumes qui stockent ses données. Lors de la migration d'un pod, ses conteneurs peuvent être déplacés vers un autre emplacement. Cependant, les volumes sont généralement liés à un nœud spécifique où ils ont été créés. Par conséquent, lorsque le pod en cours d'exécution accède aux volumes, il peut se trouver dans un emplacement distant, nécéssitant beaucoup de bande passante et augmentant la latence et le nombre de sauts réseau.

La deuxième contribution garantit que les données persistantes seront relocalisées lors du maintien à proximité du conteneur de ses volumes. Il propose une migration incrémentale en exploitant soigneusement le système de fichiers Overlay et en générant des points de contrôle avec uniquement le delta de chaque étape de migration. Avant la migration du conteneur, il migre initialement les fichiers en lecture seule et les fichiers qui ne sont pas activement utilisés. Plus tard, dans les toutes dernières étapes, il crée des points de contrôle comprenant uniquement les différences restantes (c'est-à-dire les fichiers mis à jour et nouveaux).

Les évaluations sont basées sur un banc d'essai fog réaliste montrent que notre technique réduit le temps d'arrêt du conteneur pendant la migration par un facteur de 4 par rapport à une référence où aucune migration incrémentale n'est appliquée.

Abstract

Recent years have witnessed a proliferation of data produced by numerous sources, such as ubiquitous sensors. These data must be processed to generate value and, in many cases, it is essential to reduce the delays before usable insight is generated out of real-time input data. This has led to the development of a wide range of delay-sensitive applications covering numerous economic sectors such as smart city management, education, military, and retail.

Cloud providers facilitate the development and operation of complex online applications by making public resources available for on-demand acquisition. Cloud computing consists of abundant and capable servers inter-connected using high-speed network links. It allows data processing in a stable infrastructure and guarantees service delivery.

However, cloud providers' resources are often centralized in a handful of data centers and only ensure low-latency communication internally. Therefore, there can be a gap in the long-distance communication of the users with the cloud provider's data center. This frequently implies greater communication latency for the hosted applications, consequently not meeting the requirements of demanding latency-sensitive applications.

In response to this gap, fog computing aims to reduce the distance and network latency between application instances and the users using them. In fog computing, the resources are geo-distributed and located close to the users. As a result, fog computing provides low latency communication, allowing requests to be entirely or partially processed locally rather than in a faraway data center. On the other hand, fog computing resources are often composed of devices that are not as powerful as the ones present in the cloud, meaning there is a limit on their processing and storage capacity.

The decentralized model of fog computing does not aim to replace the usage of the cloud. On the contrary, it complements the cloud's offering and increases the range of advantages of using it. For example, it may reduce the load arriving at a cloud-hosted service by pre-processing incoming requests before they arrive at the cloud.

To meet low-latency requirements, application instances should be placed in a node close to the target users. The idea is to reduce the number of network hops that are necessary to access the application service so the end-to-end latency can be kept at the

minimum. However, ensuring proximity when resources are being created is insufficient because there is the possibility of the users moving to different locations afterward. If no measures are taken, the users' mobility could cause, once again, an increase in network path length to access the services, possibly defeating the purpose of co-locating the application in the same vicinity as the users.

User mobility presents a challenge for fog computing as it may compromise one of its core objectives, the low latency. In order to overcome this challenge, application migration comes as a powerful candidate to relocate running applications close to their users. This process should occur transparently without compromising the application execution. It should keep the entire application state (in memory and on disk), maintain network connections open, and reduce as much as possible the downtime of service interruption during migration.

Migration is a common task in cloud environments and is often performed over virtual machines. It grants many advantages to the cloud environment improving resource usage, availability, and resilience. However, it is usually done on the same premises relying upon powerful resources and dedicated network channels.

In contrast, fog computing relies on a limited set of resources and available network bandwidth. Therefore, migration inside a fog platform cannot count on abundant resources. Hence, it is necessary to design a new set of approaches for this new environment and reconsider the design of geo-distributed migration techniques.

Container technology has become popular for easing and managing applications and service deployment. Packaging software into lightweight, isolated containers presents significant advantages from an IT operations management standpoint. Because of this, containers have become a good candidate for performing service management and deployment in fog computing infrastructures. Nevertheless, when it comes to complex deployments, spanning tens or possibly hundreds of different machines running thousands of containers, containerization by itself may not suffice. This is what led to the creation of container orchestration systems such as Kubernetes.

Container orchestration systems can automate, to a large extent, the deployment, scaling, and operation of containers across clusters of nodes, reducing human errors and saving cost and time. However, designed with "traditional" cloud environments in mind (i.e., large data centers with close-by machines connected by high-speed networks), systems like Kubernetes present some limitations in geo-distributed environments because containers are not intended to migrate and jump from host to host.

This thesis explores wide-area pod migration, moving containers together with their data volumes, by presenting a set of tools to accomplish this goal. The combination of these tools enables the transparent migration of services from point A to point B without incurring data loss or interfering with the user's access to the application. At the same time, the migration tool manages network connections and transparently redirects application requests to the new set of resources without breaking the existing connections. All the contributions are implemented on top of Kubernetes and are evaluated in a real fog-based testbed.

First contribution: Kubernetes Pod Memory Migration

This contribution presents MyceDrive, a seamless pod migration solution for geo-distributed Kubernetes. A Kubernetes pod is a unit of deployment containing one or more containers.

In order to relocate a Kubernetes pod to a new server, MyceDrive performs a stateful migration by retrieving the entire in-memory state alongside the container images. Mycedrive creates a snapshot of the running processes inside the containers, and it manages network reconfiguration by setting up a temporary route to redirect incoming requests to the migrated container until the container is appropriately established.

MyceDrive avoids terminating a healthy running pod and waiting for Kubernetes to restart a new one. As a result, the application executing inside the pod, and the users remain unaware of the migration.

We show that geo-distributed Kubernetes pod migration is feasible while remaining fully transparent to the migrated application as well as its clients while reducing downtimes up to 7x compared to state-of-the-art solutions.

Second contribution: Incremental Volume Migration

In the second contribution, we propose a tool for migrating Kubernetes volumes. This contribution complements the first one by enabling volume migration in pods.

A service's deployment includes both a pod and the volumes that store its data. Whenever migrating a pod, its containers may be relocated to another location. However, the volumes are usually bound to a specific node where they were first created. Therefore,

when the running pod accesses the volumes, it may be in a remote location, overflowing bandwidth and increasing the latency and the number of hops.

The second contribution ensures that the persisted data will be relocated when maintaining the container's close vicinity to its volumes. It proposes an incremental migration by carefully exploiting the Overlay file system and generating checkpoints with only the delta of each migration step. Before the container migration, it initially migrates the read-only files and the files that are not being actively used. Later, in the very last steps, it creates checkpoints comprising only the remaining differences (i.e., updated and new files).

The evaluations are based on a real fog computing test-bed and show that our technique reduces the container's downtime during migration by a factor of 4 compared to a baseline where no incremental migration is applied.

Table of Contents

List of Tables

List of Figures

Acronyms

AuFS *Advanced multi layered Unification FileSystem*. 46

BTRFS *B-tree File System*. 46

CLI *Command-line Interface*. 43

CoW *Copy-on-Write*. 44, 46

CRIU *Checkpoint/Restore in Userspace*. 48

DFS *Distributed File System*. 46

DMTCP *Distributed MultiThreaded CheckPointing*. 64, 68

DNS *Domain Name System*. 50

EA *Execution Agent*. 71, 72, 86

ext4 *Fourth Extended Filesystem*. 46

HPC *High Performance Computing*. 63

IaaS *Infrastructure-as-a-Service*. 34

IDC *International Data Corporation*. 23

IoT *Internet of Things*. 23–25

ISR *Interrupt Service Routine*. 59

LXC *Linux Container*. 62

LXD *Linux Container Daemon*. 62

MC *Migration Controller*. 71, 74, 75, 79, 80, 86

MEC *Mobile Edge Computing*. 39

MPTCP *MultiPath Transport Communication Protocol*. 62

NAS *Network-attached Storage*. 65

NFS *Network File System*. 65

OS *Operation System*. 35, 57

PaaS *Platform-as-a-Service*. 34

PoP *Points of Presence*. 40

SaaS *Software-as-a-Service*. 34

SAN *Storage Area Network*. 65

VM *Virtual Machine*. 19, 27, 28, 35, 36, 58–60, 67, 68

Chapter 1

Introduction

Companies continuously require additional computational power to deal with the growing workloads produced by new clients and products [1]. Parts of this trend derives from the ever-increasing popularity of IoT, which causes a immense rise in data production by new intelligent and interconnected devices. From 2015 to 2020, the number of IoT devices grew globally by 33% with a total of 12.3 billion connected devices [2]. The overwhelming rate of data production from these devices challenges companies which need to store and process these data. In particular, low-latency applications domains such as intelligent transportation, entertainment and health care often require real-time responses to effectively take actions in a timely fashion [3].

To address these needs, cloud computing infrastructures emerge as an opportunity to provide companies with a wide range of on-demand computing, storage, and communication services, with a pay-as-you-go model [4]. Using the cloud, companies can dynamically build up their computing infrastructure with easily customizable computational capacity and pay only for the resources they use.

Cloud resources are available from numerous providers such as Google Cloud Platform [5], Amazon Web Services [6] and Microsoft Azure [7]. These cloud providers manage large infrastructures and are in charge of hiding all the complexity and ensuring properties such as data security, resilience, scalability, disaster recovery, and high availability [8].

To cope with the growing demand for resources, cloud providers have been expanding their infrastructures with larger and larger data centers. According to to IDC, from 2013 to 2018, there was a growth of 34 billion square feet of data center occupied area [9]. However, instead of relying upon a single ever-bigger gigantic data center, all the major cloud providers now propose multiple data centers in various locations. A data center's location determines, among others, the quality of communication between the running applications, the users, and third-party services [10].

According to Gartner, 81% of companies make use of two or more cloud providers [11]. One of the motivations to do so is to gain access to a larger number of data center

locations. Many companies further expand their possibilities by leveraging hybrid cloud models, which means using public and private clouds together [12]. When the company itself owns a set of resources as a cloud, this defines the private cloud, which is usually located in a safe and controlled premise with fully isolated access granting extra security. Meanwhile, the public cloud is located off-premises and shared with many other customer applications.

Distributed cloud infrastructures are particularly useful when processing IoT data produced in a large variety of locations. To deliver real-time or close to real-time responses the distance, the availability of resources, and the quality of the connection with the cloud servers matter the most [13]. A straightforward solution to reduce distance would be to place the services close to the location where the data were originally produced. However, using a handful of data centers does not solve the entire problem. There is a mass data production with ever-increasing frequency coming from new devices, and they need to provide fast answers [14]. For instance, there are about 1.35 million tech startups around the world proposing new applications using the most diverse type of sensors or data collectors [15]. These applications often rely on high throughput, high availability, and low latency to guarantee fast responses. Placing those applications in a nearby data center is not sufficient to cope with these requirements.

Fog computing emerges to close the existing gap left by the limitations of traditional distributed cloud platforms [16]. Fog computing extends cloud platforms with additional computing resources located close to the main data sources. It, therefore, constitutes a decentralized computing infrastructure where data processing, storing, and applications can be hosted between the data sources and the cloud. In addition, fog nodes are often used to filter and reduce the data close to the the point of creation, improving communication by reducing the volume of data transfers with the cloud.

The general reference of fog computing architecture is defined by the IEEE 1934-2018 standard [17]. Fog nodes compose this architecture, and they are small units capable of processing and storing data. Fog nodes are connected to IoT devices using short-distance wireless "access networks" such as LPWAN, cellular networks like 4G and 5G, Wi-Fi, and Bluetooth [18], [19]. They are connected to one another and to the cloud using medium and long-distance "backhaul networks" [20]. To provide optimal proximity to the users and the IoT devices, fog nodes are often organized into several small fog clusters that withhold multiple fog nodes and provide coverage for a particular area using wireless networks.

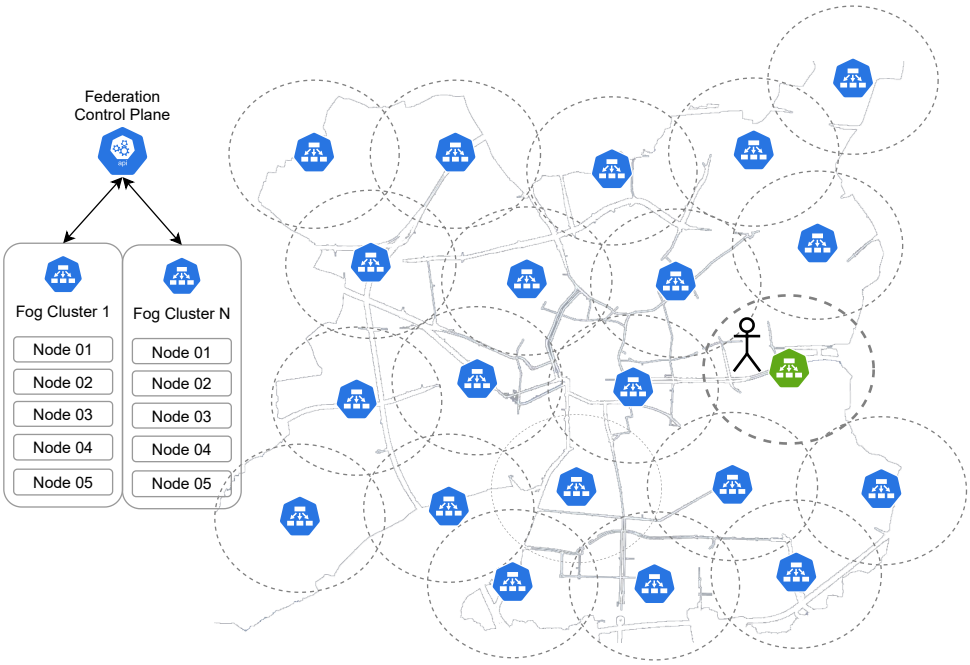


Figure 1.1 – Example of a fog infrastructure federation in a city managed by a single control plane.

To unite numerous fog clusters in a single infrastructure, a useful abstraction is a cluster federation [21]. A federation is essentially a distributed cluster of fog nodes clusters. It consists of a central control plane in charge of commissioning a number of member clusters, in which applications may be deployed.

Figure 1.1 illustrates a fog infrastructure and the deployment of an application inside a city. Each fog cluster contains one or more nodes with limited computational power [19]. It is obviously possible to increase the capacity of the fog infrastructure by deploying extra resources to the same cluster. However, external factors such as space limitation, limited network links, and maintenance may negatively impact the fog infrastructure and should be taken into account.

Because of the limited range of wireless access networks, each fog cluster should ideally be reached from the end users and their IoT devices within a single network hop. Therefore, a natural decision is to place the application in the closest fog cluster to the device which will produce or consume the data. However, not all devices or users remain at a fixed location for extended time periods. Therefore, when a device or user moves, it soon becomes necessary to migrate the application to a different cluster to maintain proximity between them and the application.

Fog infrastructures often rely on container technologies for deploying applications in the chosen fog node [22]. Containers' popularity is growing [23]. Although early container-based platforms relied on low-level container technologies such as LXC, the increased popularity of container frameworks has resulted from the introduction of high-level tools such as Docker [1].

Containers are interesting in this context because, for example, Docker[1] provides a standalone, executable package of software that includes everything needed to run an application as lightweight container virtualization, which is developer-friendly to the development of workflows [24]. Containers are sufficiently lightweight for exploiting limited machines such as Raspberry Pis [25].

The trending of container orchestration frameworks such as Kubernetes [26] has been increasing because of the popularity and the employ of containers for the deployment of applications. Kubernetes is a container orchestration platform that automates the deployment, scaling, and management of containerized applications in large-scale computing infrastructures such as a cluster and a datacenter. It relies on container runtime systems such as `Docker`[1], `cri-o` [2] and `containerd` [3], and it is in charge of creating, deploying, and running containers within a group of server machines. Kubernetes is also a good basis for designing fog computing environments capable of extending traditional cloud data centers with additional resources located close to the end users [27]–[31].

The geographical resource distribution of a fog computing platform is particularly important to address scenarios with end-user mobility. For example, Figure 1.2 illustrates a situation where a client has moved inside a city from location A to location B. The application is represented with green color, and it is no longer in the same location as the client. The application becomes increasingly distant as the client moves, forcing an additional number of network hops inside the fog infrastructure to reach the application. Therefore, the communication latency increases, creating a need to migrate the application from one fog cluster to another in order to maintain proximity to the user.

Not only does mobility affect the performance and user experience of an application, but the high concentration of users at the same location may create congestion in the network links and nearby resources. Therefore, moving the application towards neighbors' resources or back to the cloud might help, depending on the latency requirements of each application.

1. `https://www.docker.com`
2. `https://cri-o.io`
3. `https://containerd.io`

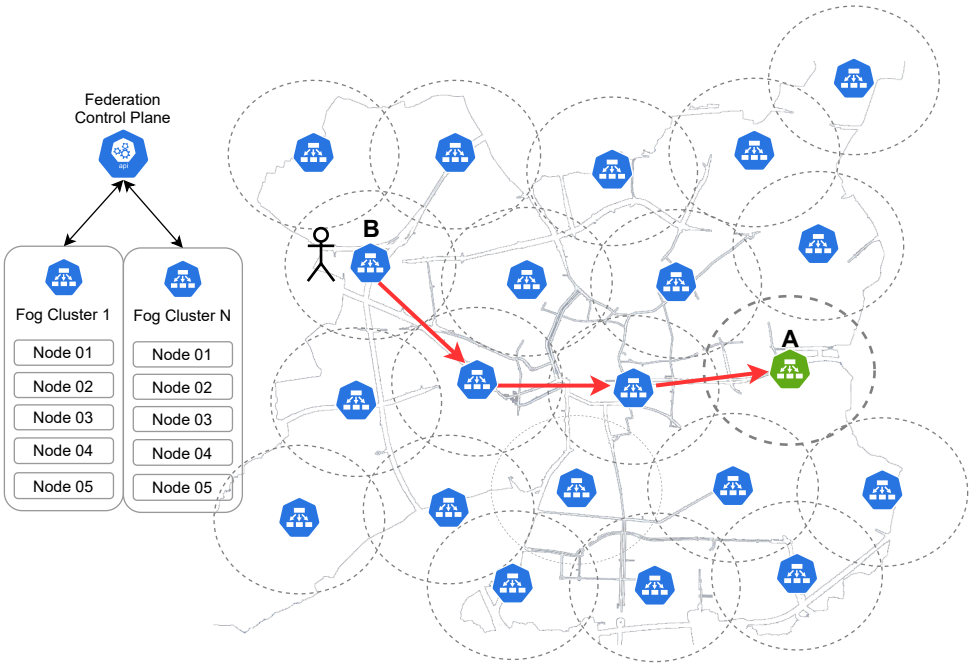


Figure 1.2 – Example of a fog infrastructure deployment in a city and its actors with mobility.

Independently on the placement decision, it becomes necessary to migrate the application. By doing this, the users will be able to move freely as the application will follow accordingly. Thus, any adversity in the system will trigger the application's movement without negatively affecting the user experience

Migration is an essential functionality in large-scale virtualized computing platforms. Migration stands for the capacity of moving assets inside or outside the computing infrastructure. For example, in cloud computing, migration is commonly applied to move VM instances between servers in the same data center. Migrating virtual machines is used in cloud infrastructures by data center managers as an enabler for scheduled server decommission, resource consolidation, disaster recovery, vertical scaling, etc [32]. As many applications are moving from VM-based technologies to container-based infrastructures similar techniques are becoming necessary in new environments like fog computing.

There are many types of migrations. For example, stateless migration consists of migrating only on application's code and configuration files. After the migration, it is therefore only possible to start the application without any previous memory state [33]. On the other hand, stateful migration stands for the capacity to migrate an entire application while maintaining all the memory footprint and persisted data. The application is thus

able to transparently restart in another location, in the exact same state as it was before being relocated.

Migrating a container or a VM sometimes requires moving the data storage that is bound to them. When migration takes place inside a single cloud data center, the data storage usually does not need to be moved because data volumes stored in a network-attached storage can be remounted from the new location without negatively impacting performance.

However, whenever we need to migrate to another data center, as is often the case in fog computing scenarios. As the data location impacts the quality of the provided service and the application's overall performance, making it extremely necessary to move the data. Moving data volumes together with the VM or container may become necessary.

There are two main techniques to perform a migration: live and cold migration. Cold migration stops the application entirely before the migration tool creates a snapshot, copies it to the destination, and restarts the application in the destination node. Conversely, live migration performs as many migration operations as possible without stopping the application runtime. The migration tool then performs all its tasks of snapshotting, remote copying, and restarting the application on the fly. Live migration significantly reduces the migration downtime during which the application is not executing. This type of migration constitutes an important technique for moving an application across servers [34].

One of the remaining challenges of geo-distributed migration is maintaining these relocations transparent for the application itself and the clients using it. Therefore, it is important to maintain active network connections during and after the migration and to transparently update the communication routes.

Migration has been deeply explored in the context of VMs [35]. Conceptually, similar techniques could be applied to migrate containers. However, container migration techniques are currently much less advanced [36] and often experience significant downtimes. This is because they were designed at the operating system level only without integrating them with higher-level container orchestration environments like Kubernetes. Therefore, new techniques and approaches should be created considering the higher-level orchestrator systems and the minimal requirements of a migration technique. In this context, container migration may exceed the domain of system management tools and additionally become a primary platform feature allowing operators to migrate a fog computing application from one location to the next to follow the mobility of humans and devices.

In Kubernetes, the smallest scheduling unit which may contain one or more containers and data volumes. Moreover, performing stateful migration of an entire pod is no easy task [36]. Besides the obvious challenges of snapshotting and restoring the containers' memory states and the data volumes' content, there is also a need to take special measures to maintain other resources such as the open network connections between the pod and external devices or users. These needs are currently not filled by existing systems, which strongly limits the development of migration technologies for fog computing platforms.

This thesis addresses the problem of geo-distributed pod migration inside a high-level distributed fog orchestration platform. In this context, there are a number of important properties that migration must support:

— **Migrating high-level Kubernetes Pods rather than VMs or single containers**: orchestrator platforms usually handle containers not in isolation from each other, but rather in pods, which are comprised of multiples containers and data volumes. When migrating applications, it is therefore important to migrate entire pods rather than the individual elements they are made of.

— **Migrating memory state** requires maintaining the current execution state of a container and the process inside, and being able to restart them in a different location. This requires snapshotting the current memory footprint that the application, process, or container. Not maintaining the current memory compromises the application workflow and user experience.

— **Migrating transparently and seamlessly**: making the migration completely transparent for the clients and sensors requires to maintain open network connections between the containers and the outside world without interrupting the communication. This ensures that the application does not need to be aware if it is being migrated, or it will proceed with its execution without suffering external interruptions such as having to re-establish network connection.

— **Migrating the application storage**: Most applications use persistent storage to maintain parts of their state. When migrating within a single data center, it is possible to reattach the same storage to multiple deployments without having to migrate the data themselves. However, in a fog infrastructure, data locality impacts the application performance affecting the latency and the available bandwidth. Therefore, being able to migrate data storage, data volumes, or disk state together with the application itself is a critical feature that has to be supported by the migration system.

1.1 Contributions

This thesis proposes techniques to perform migration of Kubernetes pods that provide full stateful migration which maintains all state from running sessions, user persisted data and information; a seamless and transparent migration for the users or connected devices; disk state migration without interrupting disk access in writing or reading; and finally on the fly route updates without breaking any connection and increasing the downtime.

The first contribution proposes a technique to perform stateful pod migration in geo-distributed. The solution requires a consistent and efficient method to checkpoint and migrate applications without interrupting the communication with connected services and users. The application memory copy should be consistent to not negatively affect the execution, with no corrupted or incorrect information. The process should also be appropriately integrated with Kubernetes and any of its container's systems. This contribution proposes MyceDrive: a mechanism integrated with Kubernetes that performs stateful pod checkpointing migration. It comprises the migration of pods consisting of one or many containers and its in-memory state. This contribution does not address disk state migration. The evaluation compares MyceDrive with existing approaches from the state-of-the-art, showing 7x shorter downtime. The experiments also indicate that the low overhead generated to employ stateful migration is reasonable compared with stateless migration.

The second contribution proposes a technique for incremental data volume migration. In the same cloud data center, performing migration is relatively simple by attaching a volume and reattaching to another running instance. However, moving a volume in a widely distributed system is an onerous action that brings much cost in terms of networking, infrastructure, and application uptime. This contribution proposes a method to incrementally migrate container volumes using checkpoints in the form of layers. By exploiting the OverlayFS, a volume is built using the concept of layers. It is thus possible to perform migration over multiple steps, layer by layer, before the application is stopped. The downtime can be reduced considerably by simply migrating the "biggest" layers before the migration occurs. The evaluation compares the usage of a pre-deployment approach and the employment of 0 to n incremental checkpoints during migration, showing that it is possible to reduce the application's downtime by using checkpoints in a factor of 4.

1.2 Published papers

The following papers have been published as part of this thesis:

1. *"Stateful Container Migration in Geo-Distributed Environments"*, **P. Souza Junior**, D. Miorandi, and G. Pierre, in IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Bangkok, Thailand. December 2020. [37]
2. *"Good Shepherds Care For Their Cattle: Seamless Pod Migration in Geo-Distributed Kubernetes"*, **P. Souza Junior**, D. Miorandi, and G. Pierre, in IEEE 6th International Conference on Fog and Edge Computing (ICFEC), Taormina, Italy. May 2022. [38]

1.3 Thesis organization

The remainder of this thesis is organized in five chapters:

Chapter 2 provides a theoretical and practical background of cloud and fog computing platforms and the systems employed to accomplish the thesis' goals. This chapter covers the architecture and limitations of cloud computing and the opportunities for the emergence of fog computing. The design of orchestration and containerized systems are then described in detail, focusing on the practical aspects emphasizing migration and the opportunities to meet application requirements. Alongside is presented a detailed description of the tools that constitute a fog platform which serves as the basis for this thesis.

Chapter 3 highlights the state-of-the-art on resource migration. The discussion starts by presenting the most common virtual machine migration techniques and how new technologies can rely upon similar approaches to perform container migration. Next, we summarize the existing container and volume migration techniques indicating their respective strong and weak points. Finally, we present a comparison of the techniques with the contributions of this thesis as we discuss the relevant aspects of each of them.

In Chapter 4, we introduce MyceDrive, a seamless pod migration tool for geo-distributed Kubernetes environments. MyceDrive performs a stateful migration by checkpointing the in-memory state alongside the container image and placing it in a new location. This chapter describes the system design of MyceDrive, how to integrate the migration to Kubernetes, and evaluates and compares with existing solutions that migrate containers.

Chapter 5 introduces incremental volume migration, as it is an essential functionality in large-scale geo-distributed platforms. It complements the prior contribution by enabling data migration as part of a pod deployment. Incremental volume migration is possible by exploiting the OverlayFS as we implement this mechanism to migrate Kubernetes volumes. This chapter presents the system design indicating how the incremental checkpoints are created, and the migration is coordinated. Finally, we evaluate performance based on a real fog computing test-bed.

Chapter 6 presents the conclusion of this thesis. We recap the importance of migration and how it can overcome challenges inside new paradigms such as fog computing, with a summary of the contributions that this thesis accomplishes. This final chapter concludes the thesis with some future directions to broaden the range of the proposed techniques aiming to improve pod migration over a geo-distributed system such as fog computing.

Chapter 2

Background

This chapter presents an overview of the concepts and technologies employed in this thesis. Section 2.1 discusses cloud computing concepts, virtualization techniques, the opportunities in the cloud, and the key differences with fog computing. Section 2.2 extends the discussion with details about fog computing, providing its concepts, opportunities, and remaining challenges. Section 2.3 shows why container systems are being employed in many cloud systems, their main leverages, and why they are particularly suitable for fog environments. Besides, it presents the migration aspects of these containerized systems. Finally, Section 2.4 presents container orchestration systems, how they are composed, and why they help to create and manage a fog infrastructure.

2.1 Cloud Computing

The computational world is growing larger and becoming more complex every day. Many human tasks are being adapted into applications for computational systems. Cloud computing comes as a prevalent model to support the execution and host of these applications. RightScale indicates in their State of the Cloud Report for 2019 that 91% of companies adopted the public cloud and 72% used a private one. Thus, most of the enterprises utilize both options — with 69% opting for a hybrid cloud solution [39].

The private and the public cloud have fundamental differences. The main difference is the location and the number of resources of both cloud models. Figure 2.1 illustrates how the cloud models are organized. The private cloud is usually located on-premise, providing additional security to the data within. However, it might have a limited number of physical resources. On the other hand, the public cloud is located off-premises. It relies on resources from a cloud provider company that are often located in different areas. Thus, the public cloud resources are not limited as in the private cloud since it is possible to contract extra resources accordingly as they are needed.

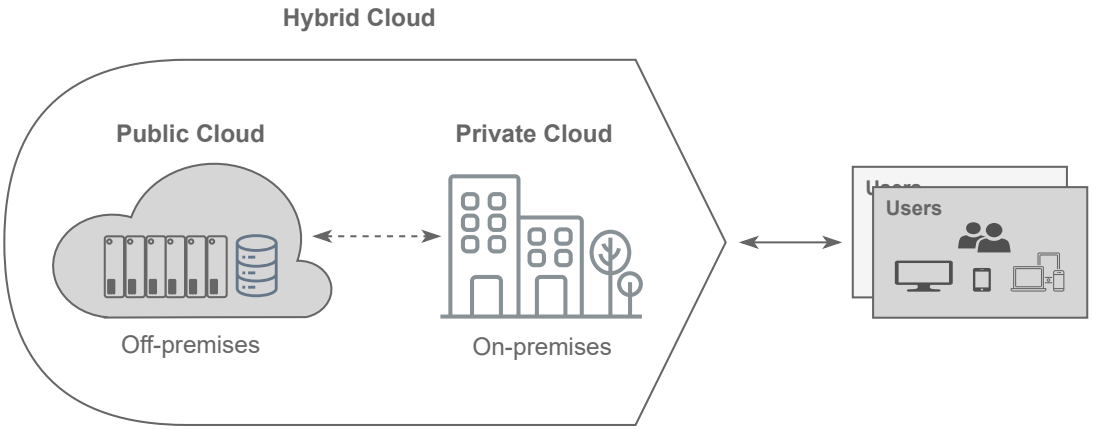


Figure 2.1 – Cloud computing models.

Many companies rely upon a hybrid cloud because it enables them to take advantage of both models simultaneously. Furthermore, it allows one to scale services through a public cloud quickly. This is convenient because there is no need to extend the local physical resources and the premises, which can be very expensive [4]. Therefore, maintaining only the sensitive and essential data in the private cloud premises.

The adoption of the cloud model reduces costs and operational complexity. Companies often rely upon these conveniences because they can leave the management, configuration, and maintenance of physical resources to the cloud provider. As a result, the costs (e.g., the total cost of ownership, running costs, upfront costs, and operational expenses) to use cloud resources are significantly lower than if the company deploys its resources on-premises.

The cloud model offers its solutions in the form of services. Among them, *Software-as-a-Service* (SaaS) is also known as cloud-based software and stands for providing software access over the internet without needing to install, configure or maintain it. *Platform-as-a-Service* (PaaS) delivers hardware and software on its infrastructure, giving freedom for developers to create, develop and release new software. Finally, *Infrastructure-as-a-Service* (IaaS) provides resources as servers for computation, storage, and networking. It is the lowest level of access and control over the resources granting the responsibility of configuring and maintaining resources in the platform. These services enable access to on-demand software, storage, infrastructures, and computing capabilities at affordable prices [8].

2.1.1 Cloud architecture

The cloud architecture consists of combined software technologies and physical resources that build up the cloud. Regardless of the type of cloud, they all follow the same principles of having multiple back-end servers capable of storing, communicating, and processing data. In addition, every cloud has a front-end platform that allows access to the control plane of the cloud, with a delivery methodology in the form of services — lastly, a network to connects it all [4].

The providers maintain all the cloud technology and place it in one or more data centers. For example, Google has 23 data centers distributed all over the world [40]. A data center consists mainly of many interconnected servers with persistent storage and low-latency internal communication. Communication inside a data center is typically stable and has high bandwidth. In addition, the communication is usually virtualized, and it does not interfere with or generate overhead inside the data center [41].

Figure 2.2 shows a cloud architecture and its users. Instead of using the physical computing resources directly, users get access to virtualized environments in the form of VMs or containers. Virtualization allows the easy management of the cloud, as discussed in the next section. Hypervisor supervises and manages the VMs, which allows the usage of different types of OS over the same physical resource.

2.1.2 Virtualization

The cloud infrastructure uses virtualization to enable the deployment of multiple resources such as servers, storage, and network. Virtualization allows creating a new layer of abstraction using software to represent and separate multiple virtualized resources. In addition, virtualization enables applications to share the same physical resources over different scopes and operating systems [42].

Server virtualization systems using a hypervisor enable the creation of multiple virtual machines over the same host. The most known server virtualization vendors are VMWare with vSphere, Microsoft with Hyper-V, as well as KVM and Xen in the open-source world [43].

Figure 2.2 also shows how hypervisors are organized inside the cloud. The hypervisor is placed over the hardware layer as it interfaces with it. The VMs are placed over the hypervisor layer as they have their own isolated guest operating system. Each VM may run one or more applications that are isolated from other VMs by the virtualization system.

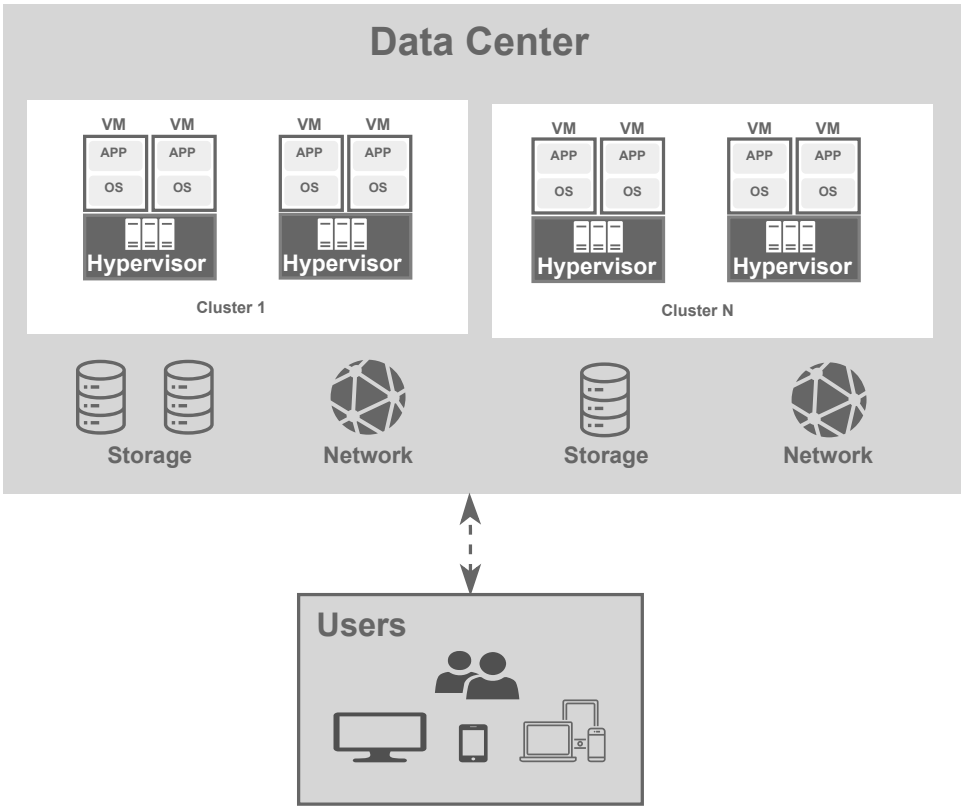


Figure 2.2 – Cloud computing architecture.

Virtualization grants benefits such as resource consolidation and migration, allowing one to place and move resources at specific physical resources. For example, a virtual machine can be provisioned in a physical resource alongside many others according to a particular set of options such as resource and network availability or energy consumption [42], [44].

Resource consolidation also avoids resource underutilization. For example, a company might use 60% of a server's physical resource capacity, leaving the remaining 40% untouched. Then, another VM can be placed in these 40%, if it meets the requirements of its deployment, taking advantage of most of the available physical resources [45].

Migrating applications inside the cloud is one of the essential functionalities present for resource consolidation, as will be discussed in Section 3.1. Taking advantage of the availability of physical resources may require the applications to be moved inside the infrastructure. The migration of virtual machines enables this.

2.1.3 Cloud limitations and Fog opportunities

Cloud computing provides many advantages to many enterprises, and it accelerates the growth of new businesses. However, cloud computing still features several limitations, especially for some types of applications demanding specific requirements. For example, coping with the real-time requirements of latency-sensitive applications remains a big challenge for the cloud. The most important limitations within the cloud and potential ways to overcome them are:

— **Long-distance communication**: the cloud model relies on a centralized architecture where all the provided services are usually located in a handful of data centers. Regardless of the location of the data centers, they cannot be close to all the end-users, devices, and/or third-party services. Furthermore, even using modern optic fiber links, long-distance communication will still generate significant network latency. Therefore, the latency may not meet the expected optimal requirements for latency-sensitive applications. On the other hand, the fog computing model relies on deploying computational resources near the users and the devices to provide low-latency communication.

— **Robustness and Connectivity**: having a centralized cloud model means all services and data will be placed in the same location. In case any problem arises with the connectivity with the cloud, the services will be interrupted, and the user will no longer be able to access its data and application. In fog computing, the fog nodes are densely distributed. They are reachable by different types of wired and wireless connections. The services are distributed, and they might be reachable from another fog node in case of failure. Depending on the implementation of the application, communication with the cloud can be interrupted. In contrast, the fog node may keep working and providing access to data.

— **Closed systems**: each cloud provider uses a specific platform to provide its services, which is essentially a black box to those hiring the cloud resources. Therefore, the control over this platform is limited, and migration between clouds can be complex because of implicit dependencies and standardization between technlogies [46]. As a counterpoint, fog computing relies on open platforms and systems, giving additional control over the deployment of an application.

By all means, fog computing does not aim to replace cloud computing fully. Instead, the fog extends the cloud functionalities with a widely distributed organization in nearby

vicinities. The main idea is to tackle the existing constraints from the cloud paradigm as they can conjunct in harmony to overcome its limitations and meet the requirements of the new applications.

2.2 Fog Computing

Cisco initially proposed the concept of fog computing in 2012, introduced it as a widely distributed cloud infrastructure. It inherits many ideas from the *cloudlets* proposed by Satyanarayanan, Bahl, Caceres, *et al.* back in 2009 [47]. The idea was to use distributed internet components to stash chunks of cached data.

Later on, fog computing became more consolidated as the OpenFog Consortium created an IEEE standard. The OpenFog Consortium defined fog computing as: *A horizontal, system-level architecture that distributes computing, storage, control and networking functions closer to the users along a cloud-to-thing continuum* [17]. Therefore, the fog is defined as an additional layer extending cloud capability that lies between the point where the data are produced or consumed (i.e., the edge devices) and the cloud. The definition proposed by the OpenFog Consortium is broadly accepted as it converges with other proposals [18], [48], [49].

Fog computing has a similar concept to edge computing, and they are often mistaken as a single thing, however, they are not [50]. This thesis focuses on the fog computing layer. For a better understanding, the key differences between the edge and the fog layer are:

— Fog computing extends and includes cloud computing, whereas edge computing uses a self-contained architecture.

— Fog computing follows a hierarchical structure (cloud, middle, edge). In contrast, the edge is represented as only the peripheral part of the network (i.e., a single layer).

The contributions in this thesis assume the devices in the edge will only produce and consume data, as they are part of the fog architecture. Thus, the devices at the network's edge do not perform processing operations themselves. Instead, the devices leave the complexity of processing and providing results to the fog nodes (placed close to the edge) and the cloud infrastructure.

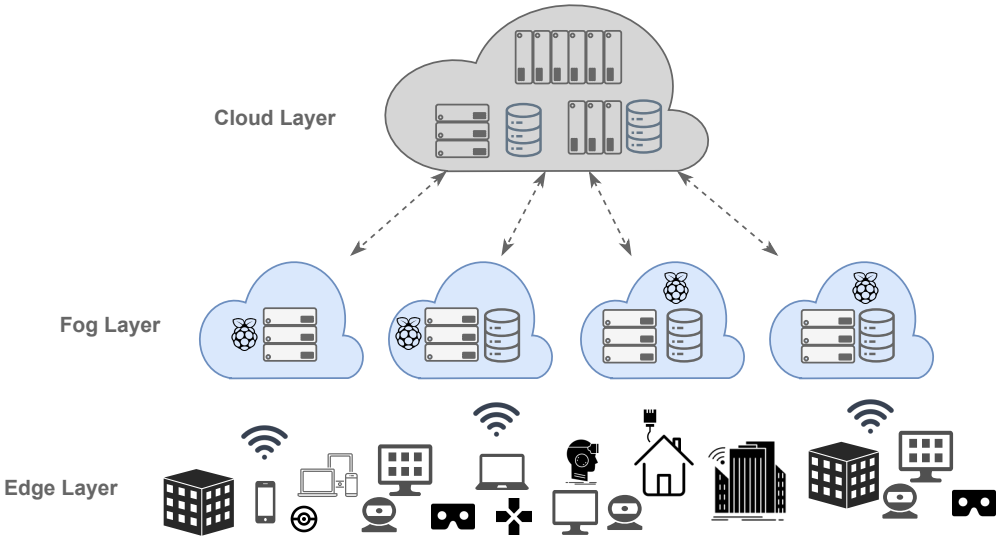


Figure 2.3 – Fog Computing layer is placed between the cloud and edge layers.

2.2.1 Fog Architecture

Figure 2.3 illustrates the fog computing architecture. Three layers compose it: the cloud layer, the fog layer, and the edge layer. The cloud layer contains one or many cloud data centers that can rely on different types of cloud architectures. The fog layer is the primary layer of the fog architecture, upholding resources and extending the cloud capability, consisting of fog clusters and nodes. Finally, the edge layer represents the network's border, being composed of many devices often identified as intelligent devices.

A fog node, as previously described, is the unit of computational power capable of processing complex operations. The fog nodes are widely distributed over a fog infrastructure, covering regions such as cities or neighborhoods. Fog nodes often rely upon small devices with limited computational power. Many proposals employ the usage of Raspberry Pi boards since they are capable nodes that can be organized as a small cluster and provide sufficient computation power [18], [28], [50], [51].

There are different types of fog nodes coming from the industry. For example, telecommunication companies rely upon small data centers deployed at the base of their cell towers [52], [53]. The cell towers antennas cover a wide range of mobile devices. *Mobile Edge Computing* (MEC) provides access to nearby powerful computational resources in the same vicinity as they are well connected inside the same infrastructure.

Another approach from telecommunication companies is the usage of fog nodes to increase the coverage range of their network. Antennas have limited coverage, especially

with new technologies like 5G having weak obstacle penetration rates and reduced range of access [54]. Using fog nodes inside infrastructures is becoming an ordinary reality [55], [56]. However, the fog and edge devices are placed in public spaces, limiting the number of physical resources.

Not only are telecommunication companies investing in the development of fog technology but also cloud giants. For example, Google has been investing in the concept of *Points of Presence* (PoP) being over 146 regions around the world [57]. PoP are a concept where the fog nodes are placed in micro datacenters, having small units of computational capacity that are distributed in a region and interconnected using public networks [58].

2.2.2 Fog Applications

The fog computing concept has mainly been developed to overcome the constraints of the application hosted in the cloud. It, therefore, also creates the opportunity to develop new applications that before were not practicable because of latency, locality, and connectivity issues [18].

Latency-sensitive applications greatly benefit from the fog infrastructure. For example, augmented reality applications provide users with an enhanced worldview. These types of applications rely upon very low latency on the scale of at most 10–20 ms [59], [60]. Typically, data center cable communication using optical fiber experience 40–80 ms [61] latency, whereas for not cable connections, latency can increase considerably and compromise user experience ultimately.

Fog applications also take advantage of the improved usage of available bandwidth. The fog nodes can reduce the amount of data to be transferred from video surveillance cameras or other stream processing applications. Therefore, the communication with the cloud may decrease, and the amount of data to be processed in the cloud.

Using fog computing, the video analysis, broadcasting, stream processing tasks, and analytical applications are no longer entirely dependent on the cloud's connectivity. Instead, the fog nodes perform the processing and decision-making in-site overseeing any disconnectivity with the cloud. The same applies to the data placed locally, providing high availability and low latency access.

The set of advantages brought by the fog computing model allows the development of new use cases and business sectors as companies propose innovative ideas for new applications exploiting these advantages. The following sectors are already taking advantage

of fog: Transportation [62]–[65], HealthCare [66], [67], Entertainment [68], [69], SmartCity [64], [70], Supply Chain [71], [72], Agriculture [73], and Security [66], [74].

2.2.3 Fog computing challenges

The fog computing model grants numerous advantages in tackling cloud limitations. However, it also brings new challenges and requires innovative ideas. The following challenges require special attention to oversee fog limitations:

— **Platform management**: the fog computing resources are widely distributed with a high-level granularity and heterogeneity; resources in the exact same location are stacked as multiple machines in the form of a cluster. These resources can be used for multiple purposes. The challenge behind this situation is appropriately managing this vast amount of connected machines. The fog platform needs to rely on a powerful orchestrator (e.g., Kubernetes), capable of handling multiple machines at many levels of abstraction and integration. Thus, it should be able to efficiently deploy an application by using lightweight technologies such as containers. The orchestrators should be aware of everything happening inside the platform to take decisions autonomously and in time, following the set of rules predefined by the platform manager. Furthermore, it has to handle all the communication inside and outside the platform by properly creating routes, connecting machines, and exposing them efficiently.

— **Service Proximity**: the reponsable platform orchestrator should be able to provide functionalities such as location-awareness and lightweight controlled deployment, allowing one to place applications within specific clusters and/or regions. The platform manager should consider the location of the users, peers, and sensors when placing the application. Being able to understand where the resources are placed and how to properly expose the services are crucial to proximity, providing low latencies. However, measuring the proximity can be challenging because the application might be relatively close to the users and the sensors while having high latencies because of networking problems or unoptimized routes.

— **Data Placement**: another fundamental challenge of fog computing is the locality of data and the capacity to decide where to place the application data. It is essential to place the application around its data. It enables fast access without requiring to reach the cloud and considerably improves its service quality. At the

same time, any applications that do not store data in memory only should maintain proximity to their data volumes not to compromise their performance. Therefore, the orchestration system should be responsible for meeting all deployed applications' requirements and maintaining its volumes near.

— **Mobility**: Keeping applications close to where the data is being produced and stored is an important part of fog computing challenges. Usually, the initial idea of placement considers just the application's initial state, resources, and users to make this decision. It does not mean that the environment will not change because the placement decision will also change in the case of mobility. For example, users often move as they have a business to attend or even the nature of the application might require sensors or users to move from location A to location B. Therefore, the platform should be able to define a method for migrating the application and its data from location A to location B accordingly.

The fog computing infrastructure already takes advantage of popular orchestration systems like Kubernetes because they provide most of the requirements to set up a fog computing platform [75]. In addition, Kubernetes can use a lightweight container system such as Docker, which has been deeply explored in the field [22], [25], [75]–[77]. The remaining challenges come as a way to improve the current state of these tools, as new solutions need to be proposed to fulfill the existing gaps created by those challenges.

However, there is a lack of support for application and data migration on containerized systems. It broadly limits the capacity to move containers and data volumes within the most accepted fog computing platform. Therefore, this thesis oversees the mobility challenge by setting up strategies to migrate containers and volumes inside the fog platform and its tools.

The following section explicitly describes the tools employed in the fog platform, their limitations, advantages, and how this thesis will address the challenge of mobility.

2.3 Containerized Systems

Container technology is a lightweight option to achieve server virtualization and obtain application isolation, cost-effective scalability, portability, resource disposability, greater resource efficiency, and improved developer productivity [78]. Because of their lightweight nature, most fog computing platforms chose to rely upon containers rather than VMs.

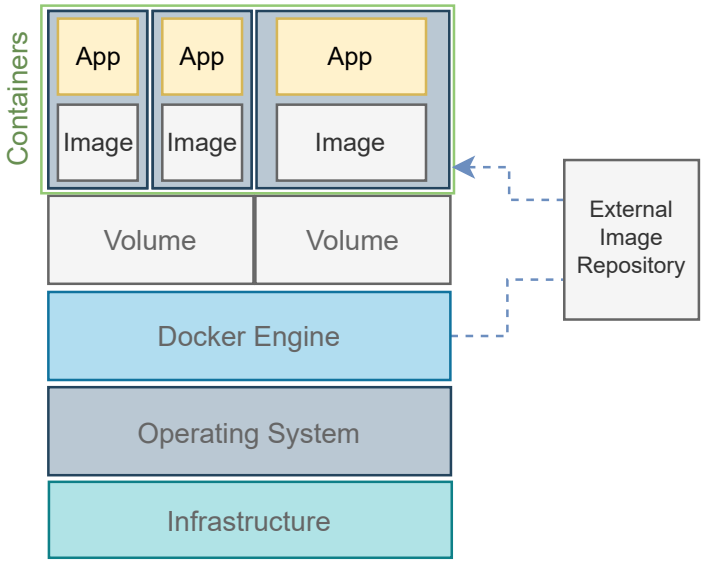


Figure 2.4 – Docker architecture.

Containers directly exploit the process isolation and virtualization capabilities built into the Linux kernel without a need for hypervisors nor guest operating systems. Containers use *Cgroups* for allocating resources among processes and *namespaces* to restrict access.

Docker is an open-source platform for building, deploying, and managing containerized applications. It is by far the most popular container framework in cluster, cloud, and fog environments [24].

Docker's architecture is shown in Figure 2.4. From bottom to top, it is composed of an infrastructure made of physical resources with an operating system (over different architectures such as x86-64 and ARM). The Docker Engine constitutes the core of Docker technology, providing a long-running daemon process (*dockerd*), an API interfacing with the Docker daemon, and a *Command-line Interface* (CLI) for the developer to execute on-flight commands. A Docker container may use one or more Docker volumes to store data. Volumes are the preferred mechanism to persist data inside Docker containers because they are overall easier to back up, migrate and have superior performance than using regular filesystem mounts and drivers from the Linux toolset [79]. Finally, on top of all that, one or more containerized applications are running isolated from the other processes.

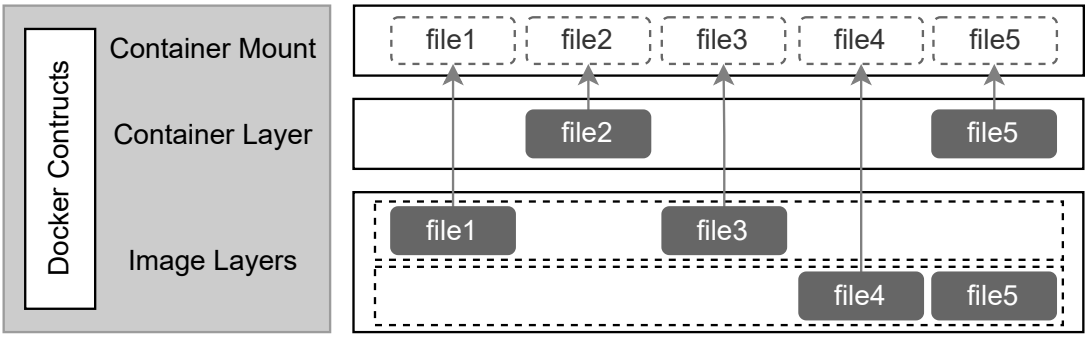


Figure 2.5 – Layered file system structure.

2.3.1 Container Images

Docker applications are packaged in the form of *images* which consist of multiple *layers*, as shown in Figure 2.5, where each layer contains a part of the container's file system that contains application binaries, libraries, data, etc. This structure makes it very easy for developers to define new container images incrementally by adding a specialization layer on top of an existing image. Docker supports multiple layered file systems, the default one being OverlayFS [79].

An external image repository known as Docker Registery stores layers that the images are mounted from. This approach reduces the number of redundant files and significantly improves the storage and deployment of containers.

The same layering strategy is also used to store file system updates performed by the applications after a container has started. Upon every container deployment, Docker creates an additional writable top-level "container layer" that stores all updates following a *Copy-on-Write* (CoW) policy. The container's image layers themselves remain read-only. As shown in Figure 2.5, although each file system layer is stored separately on disk, the layered file system exposes a single merged "container mount" view of all the layers to the container. The top "container layer" contains every file which was created or updated by the container since its creation. Under it, one or more read-only "image layers" typically store the container's programs, libraries, and configuration files. Image layers can usually be fetched from public repositories such as the Docker Hub [1].

Container images repositories can also be private providing image layers nearby the fog platform resources [80]. Private repositories help during migration since public ones

1. https://hub.docker.com/

might be distant from the platform or not reachable at all because of disconnections or service unavailability.

During any container migration, it is easy to re-create containers images at the destination node by fetching the entire read-only content from the closest Docker repository. It makes even more sense for heavier container layers that may require longer data transfers. Data transfers may also be done directly from the origin node in several rounds by fetching container layers suffered significant changes during the old container execution. The process of pre-fetching the read-only container layers before migration is called pre-deployment, and it should be applied during any type of migration and deployment [22]. On the other hand, migrating the top-level read-write container layer requires more attention as the running container's content may be constantly updated.

2.3.2 Volumes and Storage Drivers

As previously mentioned, volumes are the units inside Docker to persist data. They have better performance than using the regular *bind mounts* because *bind mounts* depend on the directory structure in the OS, whereas volumes are entirely managed by Docker and take advantage of diverse storage drivers.

Volumes provide several advantages: a multi-architecture sharing interface, safety locks for sharing content between multiple containers, and remote access to cloud providers' storage services with encryption and security customizations. In addition, volumes are independent from the container lifecycle so volumes can be reattached to a new container without losing any data whenever the lifecycle of an old container ends.

Multiple Docker containers may access the same volume, building fault-tolerant applications by configuring multiple replicas from different machines of the same service accessing the same files [81]. In order to make this possible, the docker engine uses different types of storage drivers.

Docker storage drivers control how the images and the container's persisted volumes are managed inside the docker host. They use a pluggable architecture to provide different types of storage drivers, extending the potential of a container file system. However, every storage driver provides different advantages and disadvantages. Furthermore, Docker storage drivers are available only for a specific set of operating systems and architectures. Therefore, choosing the appropriate driver depends on the use-case scenario, and it should be considered at an early stage.

The most interesting storage drivers in the context of this thesis are:

— **Overlay2**: is the most recent and stable version of OverlayFS [79]. It is a modern union filesystem very similar to the *Advanced multi layered Unification FileSystem* (AuFS). Docker natively uses an overlay to manage container images. OverlayFS layers at most two directories on a single Linux host and presents them as a single directory. They are known as layers, and the unification process is a union mount. OverlayFS refers to the lower directory as *lowerdir* and the upper directory a *upperdir*. The unified view is exposed through its own directory called *merged*. It works with the following backing filesystem: xfs and ext4. Overlay2 is an improved version of OverlayFS and it supports the *Copy-on-Write* (CoW) up to 2 levels at the same time, for example, by stacking the *upperdir* into the *lowerdir* whenever a new layer level is created. Differently from OverlayFS, Overlay2 allows users to perform changes and create new layers while the file system is being used using CoW. Overlay2 supports the creation of at most 128 lower layers.

— **BTRFS**: stands for B-tree FileSystem and is based on CoW allowing one to create clones of entire file systems. It has been widely employed in mainstream Linux distributions [82]. It requires extensive configuration, maintenance, and setup because it is a robust and complex file system and provides further advanced functionalities like snapshotting. It has a higher complexity and inferior performance because it attends to the requirements of robust systems when compared against Overlay2. Besides, it only works with its own backing filesystem (i.e., btrfs) and does not support regular ext4 and others. Btrfs uses CoW as well, as it allows multiple layers levels because of its robustness [79].

— **DFS**: is a distributed filesystem, and it allows the data to be stored not in the same machine the container is running. DFS can also replicate the data in multiple places to be accessible locally. It is ideal for sharing data within the same fog cluster as the containers might simultaneously use the same volume to read and perform changes. However, it does not support the layering system since it synchronizes the data as a single layer in one or more nodes.

DFS is the simplest way to make volume data accessible after the migration in the case of docker container migration. Data access is possible from different locations because DFS creates volume replicas at the same location or allows remote access. However, this is not ideal as this solution does not scale well, mainly because fog nodes are not as powerful as the nodes in the cloud. Furthermore, it does not fix the problem regarding

latency because keeping the data synchronized from a distant location requires constant communication with multiple scattered locations.

Overlay2 and BTRFS offer more opportunities to support containers migration. With them, it is possible to create on-the-fly layers and move them between machines as necessary. Overlay2 is relatively limited in the number of layers it can handle. However, it provides a very lightweight and efficient solution. In contrast, BTRFS offers a more robust solution capable of handling more layers and levels of abstraction since it is applied to Linux storage systems.

Every storage driver grants a specific set of advantages and may provide improvement for different scenarios. Volumes can rely upon the storage drivers during and to perform migration. However, the concept of volume itself already facilitates migration, not depending on a complex storage driver.

Volumes are easily attached and unattached to containers since they have a separate lifecycle, not depending on the containers' existence. A volume is not necessarily replaced or recreated during migration but it can be reattached to a new container. However, a regular volume without a complex storage driver will place the volume content in the same machine as the original container and assume that the initial container will cease to exist whenever migration happens. The volume will remain in the same location, and the new container may reattach to this volume from another location facing higher latency communication, limiting the throughput and possibly saturating the network bandwidth. It is therefore necessary to migrate volumes upon container migration. We return to this topic in Chapter 5.

2.3.3 Container Migration

Docker natively supports the migration of containers by using the commit operation available in its API [83]. However, it allows only the cold migration of the modifications performed in the container's image. *docker commit* essentially compresses the container image into a single file. Later, Docker can start the container again with a *docker load* operation from the generated file. The commit file is a pack of code, libraries, and configuration files that constitute an image's set of layers. It is impossible to restart the same container in a node with different architecture. Furthermore, it does not maintain any memory or session state of the running container unless the data has been dumped into the top layer of the container image. In contrast, in this thesis, we aim to provide a

consistent method to migrate containers while keeping the full memory state, and data volumes without interrupting any open network connection with the running container.

The latest API version 1.25+ of Docker provides a new functionality called Checkpoint and Restore (*docker checkpoint*) to create checkpoints of running containers. Although this is an experimental functionality, it has proved an interesting approach to migrate containers. Checkpoint and Restore allows one to freeze in time a running container by checkpointing it, which turns its state into a collection of files. Later, the container can be restored from the point in time it was frozen. This is possible thanks to a tool named *Checkpoint/Restore in Userspace* (CRIU) [84]. The checkpoint feature in Docker is often employed for restarting the running node without breaking container execution, speeding up the start-up time or a rewinding a container to an earlier point in time (in case of failures, debugging or rollout) [85].

Checkpointing a container in one node and restarting it in another one is the definition of migration. As the maintainers of the CRIU tool and Docker community claim: it is possible to checkpoint and restore from one machine to another. However, this is not their priority, and the workflow is not optimized for this task [85].

Even when using Docker's checkpoint/restore, it is currently impossible to perform live migration in Docker. By all means, it is necessary to stop the running container entirely, and the open connections with external services, devices, or users are closed. Thus, the migration process incurs a significant downtime since it is necessary to wrap up the container image and memory state to create a checkpoint file. This may result in a long waiting times to generate the file. In addition, the file is not optimized in size because it contains the whole unmodified image, file updates, and memory content. In contrast, in this thesis we aim to minimize the user-perceived downtime by transfering a majority of the data out of the critical path.

Another important aspect is that when Docker containers or images are migrated from one host to another using standard export, commit tools, or checkpoints, the underlying data volumes are not. In such scenarios, the directory containing application data must be manually moved to the new host node. Then, new containers are created with reference to that directory from the same data volume.

When the containers to be migrated are managed by an orchestration system such as Kubernetes, it is also important to integrate the migration within the orchestrator. We discuss orchestrators in the following section.

2.4 Orchestration Systems

Fog computing platforms can grow considerably in terms of the number of resources, creating a need for complex abstractions between all the components inside the platform. Furthermore, the deployment of multiples resources and the high granularity in terms of distribution bring a challenge to manage and take better advantage of the available resources.

Orchestration systems grant many essential advantages for large and complex platforms providing declarative configuration, automation, resources management, and coordination. They reduce the complexity of manual manipulations to deploy applications and optimize the usage of resources. Therefore, orchestration systems are very important when building a fog computing platform. Popular orchestration systems are Kubernetes, Docker Swarm, Fleet, and Apache Mesos.

This thesis focuses on Kubernetes specifically because it has already been widely accepted as part of a fog platform [30], [31], [75]. The solution developed during the execution of this thesis is however also applicable for any container orchestration platform that relies in containers such as Docker as its container system.

2.4.1 Docker Swarm

Docker has a swarm mode, commonly called Docker Swarm, which natively manages a cluster of Docker Engines. It uses the Docker CLI to control a swarm (i.e., a pool of resources), allowing cluster management, decentralized design, multi-host networking, load balancing, and many other orchestration functionalities [86]. It is tightly integrated with the Docker Engine, making it suitable for Docker containerized systems. However, fog infrastructures that rely on a different container engine cannot exploit Docker Swarm to manage their resources.

Docker Swarm has decentralized mechanisms without requiring a dedicated set of discovery services. It uses multiple masters applying a leadership library (Etcd, Consul, or Zookeper) between the swarm nodes. It uses a set of basic scheduling configurations to decide where to place containers within the available resource pool. The scheduling might be customized using constraints and affinities by relying upon a labeling system to attend to different rules of placement considering geographical location [87].

The Docker Engine already provides automation to manage containers using regular Docker operations. The greatest advantage behind Docker Swarm is a whole new layer of

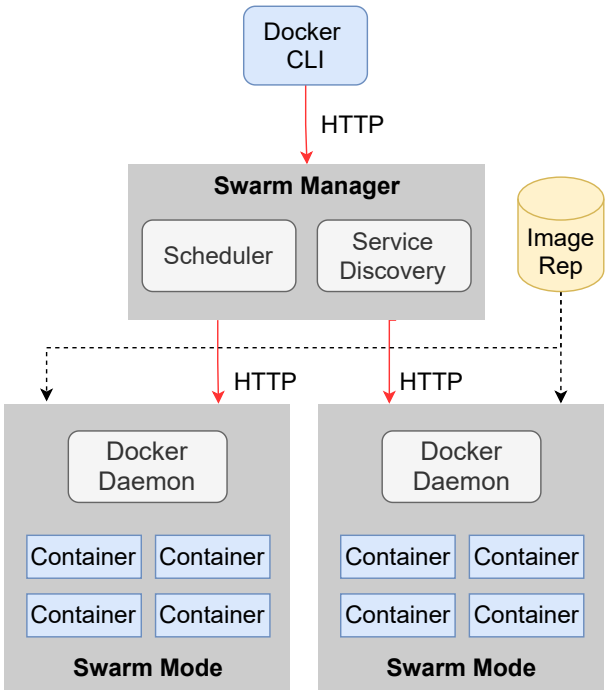


Figure 2.6 – Docker Swarm architecture.

management capable of abstracting a pool of resources as a single one. Components in this new layer are responsible for interfacing the communication with the Docker Engine and taking advantage of its functionalities.

Figure 2.6 shows the overall architecture of Docker Swarm. It uses the Docker CLI to translate the commands to the Swarm Manager through an HTTP interface. The Swarm Manager consists of a single scheduler, responsible for commissioning nodes to deploy containers. It also has a Service Discovery that creates a DNS server and assigns each service in the swarm a unique network name, load balances, and exposes running containers.

Every node runs a Docker Daemon in Swarm Mode. Therefore, the Swarm Manager is able to manage the Docker Daemon and deploy containers within the nodes. The Service Discovery manages the communication and the routes between containers, as well as exposes the services. All the container images necessary to run the containers are obtained using a Docker Repository, commonly using the Docker Hub repository if not using a private repository.

Docker Swarm Migration

Docker Swarm follows the premise of *do no harm*; it sees a service as a group of containers and not as a single instance. Therefore, it makes sure the existence of at least one running container while performing any deployment, deletion, or changes inside the infrastructure.

Docker Swarm may delete containers if necessary, as it may create new containers autonomously, not negatively affecting the service. For example, in the case of migration from Node A to Node B, there is no straightforward way to tell Docker Swarm that migration should occur, meaning that Container 1 needs to move from Node A to B. The only possible way is to change the container's deployment and indicate the container's new location, not as a migration but as a reconfiguration to the deployment inside the environment. Then, Docker Swarm will create a new container in Node B, and delete the older container (at the same moment) regardless of the running state. This type of migration is stateless, as it stops the running service. There will also be a period of downtime. The downtime period will mainly occur during the service reconfiguration and redeployment since it will need to update the routes to reach the new container in a different node. Docker Swarm abstracts the communication behind all this.

Although Docker Swarm relies on the Docker Engine technology, it still has limitations. Docker Swarm does not take advantage of the entire functionalities present in the Docker Engine. During this type of migration, the container is simply deleted and recreated in another node. Docker normally pulls the image from a repository and does not use any functionalities such as *docker commit* and *docker load* to maintain changes in the top layer of an image. There is no support for performing stateful migration using *docker checkpoint*.

2.4.2 Kubernetes

Kubernetes is a container orchestration platform which automates the deployment, scaling, and management of containerized applications in large-scale computing infrastructures such as a cluster and a datacenter [88]. It relies on container runtime systems such as Docker [2], cri-o [3] and containerd [4], and it is in charge of creating, deploying, and running containers within a group of server machines.

2. `https://www.docker.com`
3. `https://cri-o.io`
4. `https://containerd.io`

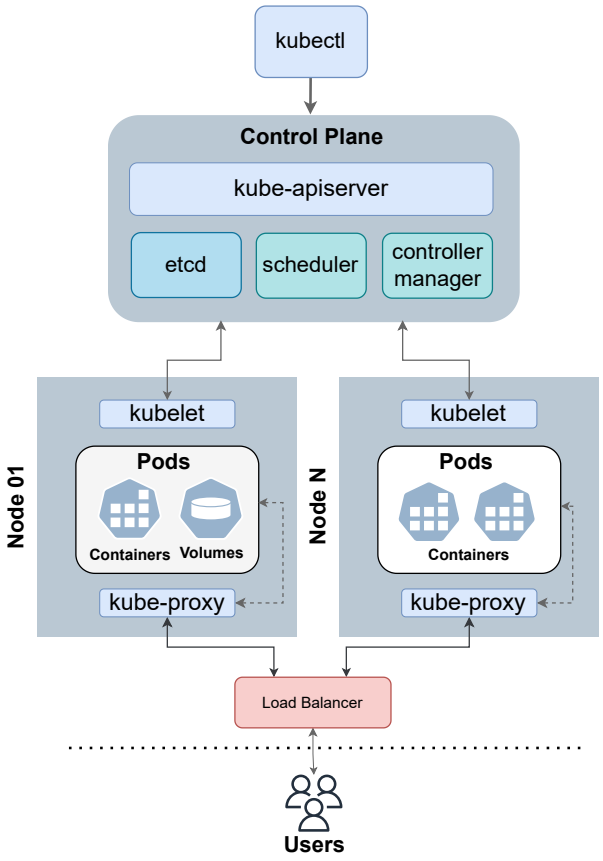


Figure 2.7 – Simplified Kubernetes Architecture.

Kubernetes was initially developed by Google, and, nowadays, it is the most popular open-source orchestration systems [89]. Many cloud providers and enterprises rely on Kubernetes as the core system to orchestrate, manage, and maintain services running [5].

Kubernetes provides functionalities that are highly applicable to a fog computing infrastructure, and many proposals are adopting Kubernetes and its lightweight versions to build fog platforms [31], [75], [90]. Furthermore, the easy management, automation of deployment, and self-adaptability are some of the main reasons we chose in this thesis to use Kubernetes as the base for our contributions.

Kubernetes' scheduler provides the desired services and deployments within the available resources. Similar to Docker Swarm, the system manager can customize the deployment by setting up rules of resources utilization and using a labeling system granting extra control over the location of applications.

Figure 2.7 shows a simplified view of the Kubernetes Architecture [91]. The Control Plane manages the work nodes and contains four components: etcd, scheduler, controller

manager, and the kube-apiserver. The components define the cluster state (e.g., creating or deleting container deployments, reconfiguring a service, and resetting or erasing network routes).

The *etcd* is designed to be a strongly consistent and highly-available key-value keeper, backing up all the cluster data. It stores the cluster configuration for the internal components of Kubernetes, as for the rules, deployments, and configurations from all the users of the Kubernetes deployment.

The *kube-scheduler* watches over the cluster resources. It makes decisions to place pods (a unit with one or more containers) according to the work nodes' deployment requirements. The scheduler considers a series of factors for scheduling decisions, such as individual and collective resource requirements, policy constraints, affinity and anti-affinity specifications, and data locality.

The *controller manager* runs all the controller processes: the node controller, the job controller, the endpoints controller, and the service account controller. Each controller has an individual responsibility: the node controller reacts whenever a node goes down; the job controller handles the job objects and creates pods to run the tasks within these objects; the endpoints controller populates the Endpoint objects by joining services and pods together; finally, the service account controller handles the accounts and access with the API using access tokens for different namespaces.

The *kube-apiserver* exposes the Kubernetes API, allowing the system administrator to define and execute tasks in the form of files or commands, such as deployment files for containers, services, volumes and networking. Kube-apiserver scales itself by deploying its replicas and balancing the traffic between them.

Typically, Kubernetes reserves a node specifically to hold the control plane (i.e., a master node). The master node does not accept any container that is not part of the Kubernetes management tools. The system administrator may also deploy the control plane across multiple nodes in a production environment, providing fault tolerance and high availability.

The worker nodes run two components: kubelet and kube-proxy. They are responsible for maintaining the pods running and the runtime environment for Kubernetes.

Kubelet is an agent that controls the running containers, and it interfaces the communication between the worker node with the master by registering the worker node with the kube-apiserver in the master. Thus, it is responsible for keeping the pods running and healthy.

The kube-proxy is a network configuration daemon that runs on each cluster node. It maintains the network rules of in and out communication with the pods, i.e., defining how the pods communicate externally and internally. Kube-proxy uses the operating system packet filtering layer if there is one. Otherwise, kube-proxy forwards the traffic itself.

This architecture allows an easy setup of services and applications by defining the desired availability of resources, application images, exposure, etc. Then, the components provide accordingly the specifications described by the deployment files. Finally, all the tasks to achieve the desired state occur following the routine inside the scheduler, the communication and synchronization of components.

Kubernetes manages many scattered resources using broadly distributed system concepts that are being improved nowadays. For example, the concept of federation allows managing multiple Kubernetes clusters by translating the deployment and configuration of applications between multiple Kubernetes platforms.

Deployment

A Deployment describes the desired state of an application. The Deployment Controller is in charge of monitoring the actual application state and of resolving any discrepancy between the desired and the observed state, for example, by adding or removing application pods [92]. Deployments may be updated by their administrators at any time, which typically triggers the Deployment Controller to create, delete or modify pods.

A deployment defines as well the minimal conditions to run a pod. The scheduler is responsible for maintaining the current desired state in the system by considering the external deployment files (usually in the form of YAML or JSON) received by the kube-apiserver.

The scheduler follows the premises within the deployment and observes the current state of the cluster in order to make a decision of placement, creation, deletion and updates.

Pod scheduling

In Kubernetes, the smallest software deployment unit is a *pod*, defined as a tight set of logically-related containers and data volumes to be deployed on a single machine. Kubernetes assigns each pod with CPU, memory, and disk resources in one available worker node in the system. The choice of which node should host which pod is made by the Kubernetes scheduler [93] by relying in a score system and availability of resources.

Kubernetes scheduler also guarantees that the containers which belong to a pod execute in the same machine and share the same set of resources, such as a single private IP address within the cluster. Besides, the scheduler is responsable of the whole lifecycle of a pod, from it very creation to the end of its existence. If there is a failure or an deployment reconfiguration (e.g., changes on the minimum required resources, rollback or new version rollout), the scheduler will delete or create new pods, attempting to meet the updates on deployment state.

Services

Kubernetes defines a *service* to provide a group of pods with a stable network address with a stable IP address. The service can be exposed to external users, which acts as a load balancer between a set of related pods. This means that, upon a pod's migration from one Kubernetes worker node to another, the pod may remain reachable by external end users by simply exposing a stable service IP address which gets dynamically rerouted to the new pod's location.

A Service in Kubernetes is an abstraction that allows accessing pods. A pod usually has a unique IP address and is only exposed outside by a Service. Services are exposed in different ways; the common and interesting one here is the load balancer, which enables routing the traffic across a set of pods or a single pod.

The Services rely upon the controller component, which often scans all the pods, and based on the type of service if any change occurs in the pods or the service, the controller updates the kube-proxy, changing to the expected configuration. This process typically runs asynchronously and may variate over time to input a new modification.

During the migration process, it is necessary to update the routes inside a Service. Because upon a pod's migration from one Kubernetes worker node to another, the pod needs to remain reachable by external users using the same service and IP address. It is possible to achieve this by relying on a load balancer service that consistently provides the same IP and dynamically updates the routes to the new pod's location.

Volumes

In Kubernetes, the Docker read-write container layers are explicitly assumed to be ephemeral. When a pod is stopped (a frequent operation in Kubernetes), its container layers are simply discarded.

Kubernetes provides persistent storage in the form of *volumes*, which are virtual disks available to the containers within a pod [94]. Kubernetes provides the same volumes and storage drivers as the ones in Docker, and more.

Volumes remain persistent even when a pod is stopped, and they can be re-attached when the pod is restarted. Multiple containers inside a single pod may share the same volume and perform any read/write operations simultaneously; this pattern is commonly used in Kubernetes. Upon a geo-distributed pod's migration, it is therefore essential to also migrate the volume's content, so the pod's data remains co-located with the containers which access them.

Kubernetes Migration

Although Kubernetes does not officially support pod migration, enabling migration is possible by manipulating the container engine. This thesis does not need extensive changes in the Kubernetes system but rather reconfiguration inside the deployment files, services, and volumes. Chapter 4 and 5 extend this discussion by providing more specific details regarding the changes and customizations necessary to enable migration in a fog platform.

The biggest challenge in performing migration inside Kubernetes is synchronizing every migration step's execution. Usually, a pod contains one or many containers, and moving a pod consists of several steps. The initial step is to place the new pod, this is a relatively simple task, and it can be easily achieved by relying on the Kubernetes labeling system and scheduler. With the labels, the scheduler only considers putting a pod in a specific machine or set of machines with the exact same labels.

The scaling-up functionality allows the creation of a new pod in a different location. Therefore, asking the Kubernetes API to scale up deployment and setting up this deployment with the configuration of the desired location will make the scheduler create a new pod of the same deployment in the new location. The second step is to delete the ancient pod, assuming there is no need for it. By using scale down, the scheduler will handle the older pod and finalize its execution because the older pod does not meet the deployment requirements, and it will keep only the pod that follows the desired state.

Finally, whenever the old container dies, there is a need to handle open connections. The Kubernetes service normally will update the routes to the new container, however, there is a significant downtime necessary to update these routes that will impact the application execution.

Chapter 3

State of the art

The migration of applications between infrastructures and machines has become feasible with the popularity of VMs. Virtualization allows one to simply wrap up an operating system into a single file and, later on, resume from it in a different machine. Virtual Machines allow this process by creating snapshots or checkpoints from their current state. Migration has been deeply explored in recent years, and many types of migrations have been proposed.

As container systems are becoming more popular every day, the requirements for an efficient type of migration are becoming decisive in this context as well. Containers are an excellent alternative to VMs for fog computing infrastructures because of their lightweight and low complexity employment. Migration is essential to maintain proximity between the applications and the devices using them.

This chapter studies virtual machine and container migration techniques by considering their strong and weak points. We discuss these approaches' applicability, advantages, and disadvantages for containerized systems. Thus, we compare a limited number of approaches to container migration and indicate the key differences with the contributions within this thesis.

3.1 Migration Techniques

Migration enables virtualized systems consolidation, which is a key functionality inside Cloud Computing. It provides flexibility, cost-saving, and increased uptime. With virtualization, it is possible to share the computational resources among multiple OS deployments. However, the way the computational resources are shared is not always optimized, and changes or interruptions may occur, requiring extra management and updates in the application deployment [95]. Migration helps in these situations by moving applications or processes between machines using different sets of techniques depending on the situation and the requirements. Nowadays, the same concepts apply to containers, which justifies

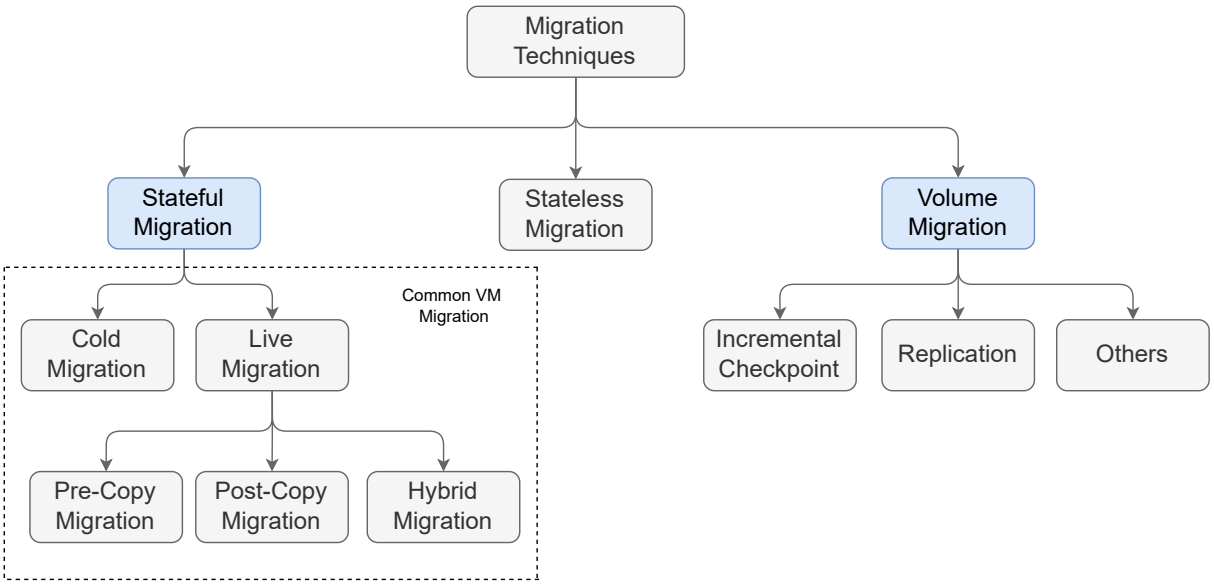


Figure 3.1 – Taxonomy of *Virtual Machine* (VM) Migration Techniques.

developing new techniques to migrate containers. This section presents details regarding the existing VM migration techniques and discusses how to apply the same concepts to containers.

Virtual machine migration consists of schemes to migrate the runtime state of virtual deployments (e.g., in-memory processes, persisted data) among multiple physical nodes. Migration aims to increase fault tolerance, manageability, cost-savings and applications' performance [34]. For example, power management provides cost savings by turning off under-loaded servers and migrating VMs to a reduced number of servers. The same applies to not interrupting the application during routine system maintenance, which requires the machines to be turned off. Therefore, relying upon migration is possible to preserve the current state of a running process or a job avoiding losing any computed data.

Figure 3.1 shows a taxonomy of container migration techniques organized along into three main branches. Stateful migration consists on keeping the running application's in-memory state across migration whereas stateless migration keeps no state. Finally, volume migration moves the application's persisted data. Common VM migration approaches provide a conceptual base to design and consolidate container migration. Furthermore, it is possible to apply VM concepts to help develop new approaches aimed at container environments.

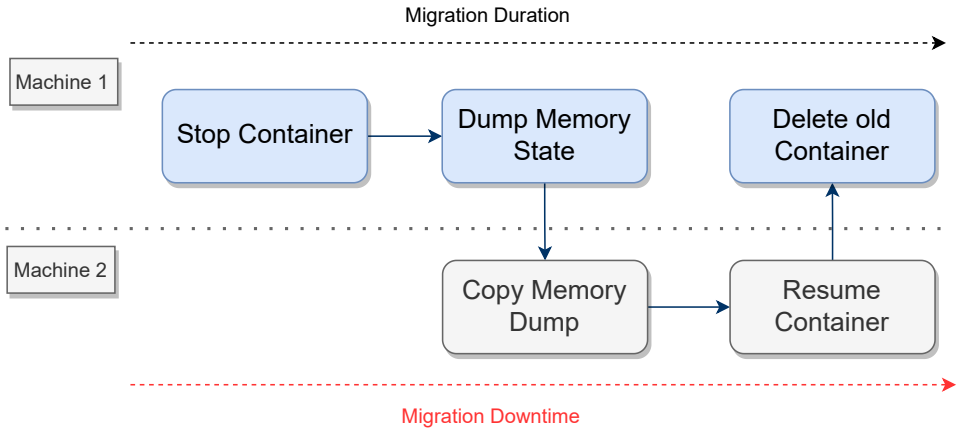


Figure 3.2 – Cold Migration workflow.

Commonly, whenever performing a stateful type of migration, there are two main ways of classifying this migration, and that is live and cold migration. Every type of migration has its pros and cons, as discussed in the following.

3.1.1 Cold Migration

Cold migration (i.e., non-live migration, offline migration) normally has a downtime equal to the migration time, meaning that the VM is offline during the entire migration. The cold migration scheme in Figure 3.2 follows the ISR pattern of *suspend* and *resume* the processes [34], [96]. The first step during cold migration is suspending the VM and freezing its execution. The most complex tasks happen at that moment because snapshotting generates a copy of the entire state of the running VM. Generating this file sometimes takes time since it dumps memory state, processes, persists data, and writes them into disk [34]. This infers considerably large snapshot files that later on need to be transferred over the network. Once the target node receives the copy from the source VM, the system is able to resume the VM using the snapshot and proceed with normal execution.

3.1.2 Live Migration

Live migration means that migration occurs while the VM is running. The overall idea is that the main running processes are not interrupted whenever the migration happens. However, depending on the type of live migration, the application may be interrupted for a

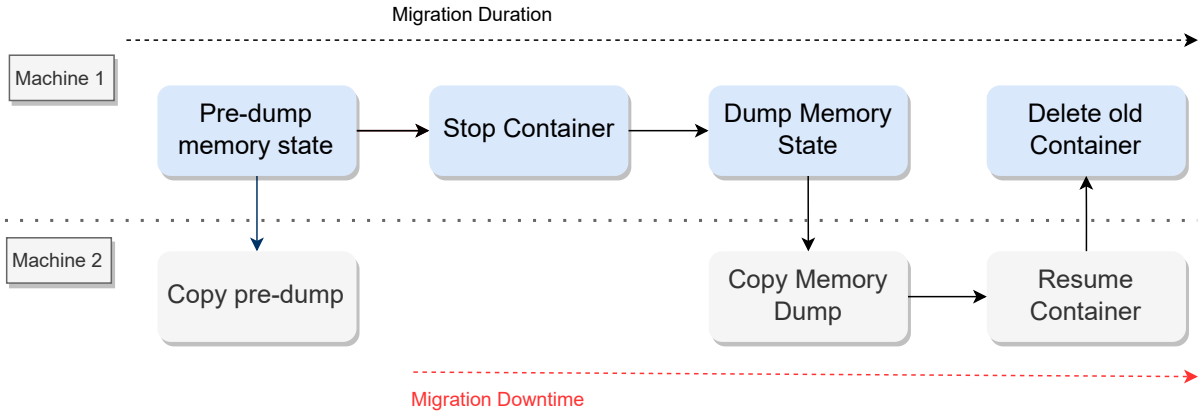


Figure 3.3 – Pre-copy Live Migration workflow.

short period before or after migration using the following approaches: *pre-copy*, *post-copy* and *hybrid* [34], [96].

— The *pre-copy* approach is illustrated in Figure 3.3. It consists of iteratively copying VM memory pages, reducing the size of memory content to be transferred between machines before actually stopping the VM. The migration tool uses a set of rules to decide when to stop the VM (for example, when the initial dump has been fully sent). Then, the remaining memory delta (i.e., dirty memory, memory difference) can be dumped and transferred, so the VM can properly be resumed in the target node. All the steps to perform migration run while the VM executes normally. This process takes longer to execute than cold migration, increasing the migration time but reducing the downtime [96].

— The *post-copy* strategy is illustrated in Figure 3.4. It consists in first stopping the source VM. It generates an initial memory dump with only the minimum content necessary for the VM to start its execution. Afterward, this initial dump is used in the target node to let the VM reestablish execution. The remaining VM source memory is sent in the background. This approach reduces the downtime and migration time compared to the rest of the strategies. However, it implies generating page faults which may slow down the application for a period of time after migrating the VM. In the source VM, the migration tool and the application are responsible for maintaining in-memory consistency.

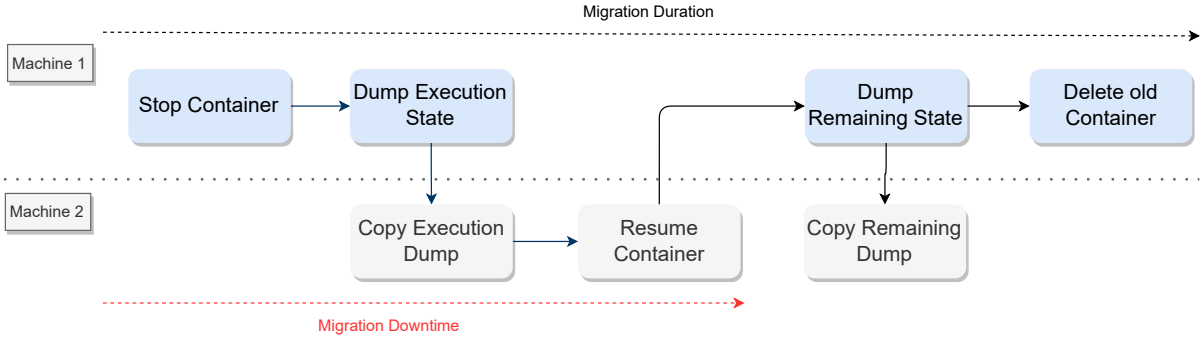


Figure 3.4 – Post-copy Live Migration workflow.

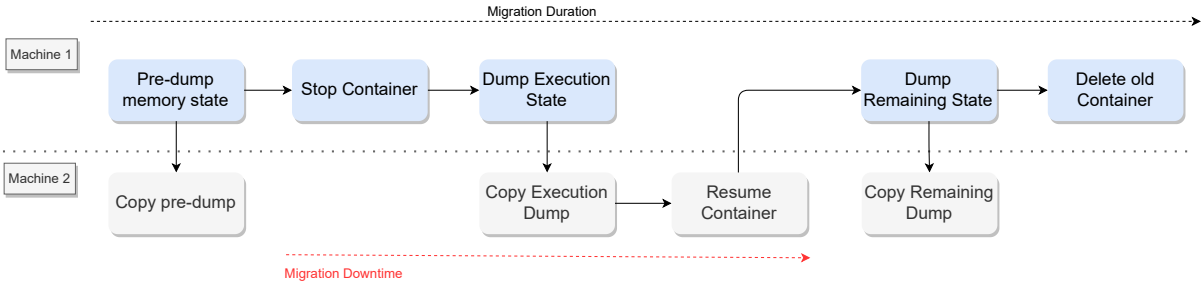


Figure 3.5 – Hybrid Live Migration workflow.

— The *hybrid* method is illustrated in Figure 3.5. It consists in applying both aspects of the *pre-copy* and the *post-copy* approach. By applying an initial step using the *pre-copy* and later on another step with the *post-copy* approach.

The key objectives of these types of migrations are: optimizing the application performance during migration, improving bandwidth utilization, minimizing downtime, and increasing service availability [34], [97]. On the other hand, there is a downside because an additional overhead created to the system during migration generates extra resource utilization [34].

3.2 Container Migration

All VM migration types discussed so far consist in maintaining state, therefore, defining themselves as stateful types. Stateless migration is not as popular since it does not keep any application state or data after migration. The stateless type of migration consists of only using the same image with the original or unchanged source files.

In the world of container migration, a stateless migration is very common, especially in large infrastructures that rely on Kubernetes as their orchestration system. A container individually is not as important as the complete application deployment composed of multiple replicas. Therefore, losing a container for a limited amount of time is often bearable and expected by the system. However, it all depends on the type of the application because when an application that relies upon in-memory is lost, it means that the data should be processed all over again. The uniformity and order of the data might be compromised, resulting in application errors or unexpected results.

The forthcoming urgency for containerized solutions and systems has created opportunities to develop new technologies and tools for migrating containers. However, container migration is often not exploited because of low-level implementation, architecture challenges, and unforeseen requirements [98].

Normally, in the container world, the migration of a containerized application is managed by the maintainers or developers of the application themselves. It does not depend precisely on container technology to provide such functionality. A small number of technologies exist and provide only cold migration for containers. Thus, these solutions are onerous and may generate extended downtime.

3.2.1 LXD container migration

Linux Container (LXC) is the native container runtime system in Linux environments. It is a lightweight mechanism that allows the creation of containers by sharing kernel and characteristics from the host OS with all the containers. It relies on CGroups (control groups) to manage and share resources and POSIX file capabilities to implement processes and network isolation [99].

Linux Container Daemon (LXD) is the newest version of Linux containers, and it supports container migration either by using built-in libraries [100], [101] or by relying on CRIU (Checkpoint/Restore In Userspace) [102], [103] which is discussed in the following subsection.

Qiu, in his master thesis, attempts to improve the current state of LXC migration with CRIU [104]. Qiu proposes a migration using MPTCP claiming that migration may experience network failures and with *MultiPath Transport Communication Protocol* (MPTCP) is possible to establish multiple network subflows granting resilience during migration.

3.2.2 CRIU-based migration

CRIU is a Linux kernel module to snapshot and later restart the contents of a container's memory pages, open files, etc [84]. CRIU freezes the running container and creates a snapshot from its current state. This operation potentially requires significant time and space as the snapshot contains a full dump of the memory state as well as all open files. Furthermore, the snapshot does not contain the entire container image but only the modifications within it. It is, therefore, necessary to deploy the same image in the destination node to restart from the snapshot. The container remains unavailable while being snapshot and during state transfer to another node until a new container is created.

Multiple migration systems exploit CRIU. For instance, in HPC environments, it is used to migrate MPI applications to improve workload resilience to anticipated hardware failures [105]. In Docker-based edge computing environments, it may also be used for checkpointing, suspending and potentially migrating long-running blocking FaaS functions [106]. The authors show that checkpoint files are small and therefore suitable to implement live migration. Finally, H-Container migrates containerized applications across computing nodes of different ISA architectures by adapting LLVM using CRIU [107].

Some other works aim at improving CRIU to address specific situations. For example, VAS-CRIU is a variant of CRIU which saves the snapshot in memory rather than on disk [108]. This avoids disk contention problems and significantly speeds up the process of creating and transferring a snapshot. It is most suitable for being used in powerful servers equipped with large DRAM. On the other hand, fog infrastructures are often built from weak single-board computers where DRAM is scarce [25], [28], [109], [110].

Redundancy Migration proposes to reduce the downtime due to migration by buffering incoming network packets during migration and replaying them on the migrated container [111]. This allows one to avoid the necessary time for client machines to detect packet loss and retransmit. This work is interesting because it shows that container migration does not necessarily imply breaking open TCP connections at the time of the migration. However, it relies on the active participation of client nodes to update their routing rules during migration. On the other hand, we aim at making migration fully transparent for the client nodes.

Finally, some works exploit CRIU to checkpoint and migrate containers in ARM-based machines. This is not easy due to compatibility limitations regarding architecture, cross migration, high downtime, wrong-order reception, and lack of kernel support to migrate containers [112].

3.2.3 Kubernetes-based Migration

In Kubernetes, the simplest way to migrate a running pod is to stop it, then redeploy a new one in a different server. The disk state of the deleted pod may be preserved by declaring the pod as a StatefulSet and then by re-attaching the preserved data volume in the newly-created pod [26]. A StatefulSet requires the declaration of at least one PersistentVolume that is used to keep the disk state. The new pod may be accessed by its users using the same IP address as the old one by exposing the pod using a Kubernetes Service.

From an application's point of view, this migration procedure presents two major weaknesses. First, it requires the application to be designed in such a way that it immediately dumps its entire runtime state to disk upon receiving the SIGTERM signal, which indicates that the pod is being stopped. Any unsaved state (e.g., a variable maintained in memory) cannot be recovered after the migration procedure. This is a significant issue considering that most workloads in Kubernetes exploit standard third-party software such as Redis, Postgres and ElasticSearch [113] which may or may not have this capability.

Second, stopping a running pod at a random time implies breaking the open TCP connections between the pod and its end-users at the time of the migration. This means that pod migration is visible from the external world, and possibly creates inconsistencies between the clients and the server pod because the clients have no way of determining whether their latest request was executed or not before the pod failure [114].

3.2.4 DMTCP-based migration

Distributed MultiThreaded CheckPointing (DMTCP) is a user-level checkpointing package for distributed applications [115]. With DMTCP, it is possible to checkpoint a group of processes, and later recreate processes from the checkpoint. DMTCP works in user space, and requires no modification to the processes' binary nor the Linux kernel.

The original motivation behind DMTCP was to provide fault-tolerance properties to running processes. Although DMTCP is mostly used in x86-64 architectures, it has also been used in ARM-based machines. For example, [116] uses DMTCP to checkpoint and clone processes in Raspberry PIs. This process turns out to be 500 times faster than spawning or forking new processes. This work also demonstrates the usage of DMTCP in machines with limited memory space.

With DMTCP, it is possible to snapshot a group of processes. However, there are a few requirements to make this possible. First, one has to start a DMTCP Coordinator on the host machine. Secondly, we must launch the concerned processes by replacing their startup command with as "`dmtcp_launch [command]`". The `dmtcp_launch` prefix registers the started process with the coordinator and wraps some of the application's library and system calls to track the creation of new threads or processes, operating system resources such as locks and open files and network sockets.

A DMTCP coordinator is used to manage any process that the user wants to checkpoint. Each process only needs to start using a DMTCP launcher that will connect to the coordinator and keep track of any spawned thread, forked processes, or remote processes. DMTCP creates consistent checkpoints of the processes' memory state as well as their open operating system resources such as open files and network sockets.

The pod migration technique proposed in Chapter 4 of this thesis relies on DMTCP for snapshooting and restarting the pod's containers.

3.3 Volume Migration

Migrating containers and services means bringing not only the service nearby but also the service's data.

One crucial aspect of widely distributed environments is the requirements to access or perform operations over a substantial amount of data. The concept of spreading resources often means using serverless, stateless, and in most recent cases, stateful services that rely on nearby databases or storage systems. Having persistent storage close to the application is therefore key to improving responsiveness and service quality.

Relocating storage is no novelty, VM migration has it as a feature [117]. It is often performed as part of a live migration without interrupting the access to the storage [118]. Usually, the source and the destination machine share the storage using mechanisms such as NFS, NAS, and *Storage Area Network* (SAN)s [119] so some data may be re-attached by the new VM without needing to migrate the volume. However, in geo-distributed environments, simply re-attaching a logical disk unit after a migration would imply long-distance remote access to a migrated container's data, which may negate the benefits of the migration.

Cloud services commonly provide storage migration functionality. However, it mainly focuses on storage services or geo-distributed storage systems [120]. At the same time,

the existing techniques attempt to improve the current state of storage migration by proposing innovative approaches relying upon replication, prediction, overlays, and placement [121]–[124]. However, these approaches often consider only the migration between robust data centers with private communication links and powerful resources. In contrast, in fog platforms, the resources and the network bandwidth are often limited, presenting new challenges and lacking efficient alternatives.

Inside Kubernetes, the unit of storage is called a volume. It serves as a type of file system for containers and provides data access. In this sense, a volume can be thought of as a logical disk rather than a physical one [125]. The notion of volume actually comes from Docker. It is generally represented by a driver or a simple folder placed somewhere in a disk, as discussed in the previous chapter.

The most popular techniques to perform container migration do not consider the migration of the data volumes and simply re-attach them after migration [103], [126], [127]. This implies potential latency and performance problems inside fog platforms. There are solutions for VM storage migration, and they may propose an opportunity to reimplement the same concept for container volume [128], [129]. Volume checkpoint techniques are necessary for fog platforms as they may be used to improve the speed of geo-distributed data volume migration [37].

3.4 Container Migration techniques comparison

As previously mentioned, container migration is an essential functionality in large distributed systems such as Fog Computing. However, the state-of-the-art does not provide many approaches that support this functionality. This is partially due to the fact that containers are often expected to remain stateless [130].

Initially, the cloud computing community realized the necessity of migrating containers between public clouds or over public and private cloud [36]. However, most of the proposed solutions rely on performing migration at the application level. Therefore, it would be able only to migrate a specific set of applications, relying on ad hoc strategies such as using an ephemerous console to buffer new requests while the container application itself writes its current memory state on disk [131]–[133].

Table 3.1 classifies the proposals of migration by comparing parameters such as the types of migration, relying on or not on the creation of snapshots, and the support of network reconfiguration (i.e., capacity of updating the routes and maintaining an open

Type	Ref.	Stateful	Live	Pre-copy / Incremental	Snapshot / Layered	Network / Bandwidth
VM	[34], [96]	✔	✘	✘	✔	✘
	[134], [135]	✔	✔	✘	✔	✘
	[136]	✔	✘	✘	✘	✔
	[137]–[139]	✔	✔	✔	✔	✘
Container	[84], [102], [126], [127], [131], [140]	✔	✘	✘	✔	✘
	[133], [141]	✔	✘	✔	✔	✘
	[26], [142], [143]	✘	✘	✔	✘	✘
	This thesis work	✔	✔	✘	✔	✔
Data	[117]	✔	✘	✔	✘	✘
	[118], [120]	✔	✔	✘	✘	✔
	[119], [144], [145]	✔	✔	✘	✘	✔
	[128], [129]	✔	✔	✘	✔	✘
	This thesis work	✔	✔	✘	✔	✔

Table 3.1 – Comparative table of migration approaches for VMs, Containers and Data.

connection with the clients during and after migration). VM migration approaches are presented since they are similar to container migration techniques. They also constitute a basis for new migration techniques for containers. All the VM migration techniques in the table rely upon use *pre-copy* to perform a migration, and we do not present approaches that rely upon *post-copy*. This is due to the fact that no container migration technique uses *post-copy*.

Most container migration techniques are grouped into a single line in Table 3.1. These methods for repositioning containers use the same CRIU-based technique to achieve stateful migration, therefore, having the same parameters in the table. The migration itself is a stateful type of migration. It is a cold migration, so there is no usage of *pre-copy*. However, all of them allow the creation of a snapshot. Besides, no proposal considers network reconfiguration necessary to maintain open connections across multiposition.

Although CRIU has been used in multiple container migration systems, it features two significant limitations which impact all migration techniques based on it. First, CRIU is a Linux kernel module that requires modifying the OS of the cluster's server machines. It supports only a few kernel versions, particularly on ARM processors, which may create conflicts with other platform requirements. Second, CRIU is currently unable to checkpoint and recreate open network connections transparently to the clients. This implies that open network connections to client machines are necessarily broken upon container migration.

Unlike migration within a local area network, crossing network boundaries results in network reconfiguration. Moving into a new subnet forcees the machine to get a new IP address which, as a result, breaks existing network connections. Either the network addresses must be preserved or applications must be made aware of the network reconfiguration semantics.

In contrast, this thesis work performs a stateful and live type of migration. It performs a similar step of *pre-copy* named container pre-deployment [22] where the container's read-only parts can be firstly moved without any requirement of stopping the running application. Thus, it achieves network reconfiguration to avoid connection reset or complete loss of running sockets. This is possible because we exploit DMTCP which offers this possibility.

Compared to CRIU, implementing migration using DMTCP has two main advantages. First, it runs entirely in userspace and is therefore agnostic to the Linux kernel version. Second, it can checkpoint and recreate network socket state, which give us the opportunity to maintain open connections with the clients machine during migration.

The remaining works in Table 3.1 propose alternatives to migrate data volumes over different contexts (for VMs and containers). Most of the approaches perform the migration without stopping access to the volume. However, not all of them are optimized and consider the bandwidth contention. The primary technique applied here is replication, as every change performed in a volume is broadcasted with an entire file to the rest of the volumes over-flooding the network. On the other hand, this thesis employs the use of the layered file systems allowing the creation of checkpoints in the form of layers with only the changes performed on disk. Therefore, the layer sizes are comprehensible smaller in size, and the amount of data sent over the network is lower.

Chapter 4

Stateful Pod Migration

This thesis chapter presents MyceDrive, a stateful resource migration solution natively integrated with Kubernetes. In geo-distributed environments, moving running applications, following moving data sources/sinks or unpredictable changes in the network substrate, is a relatively common operation. With MyceDrive, geo-distributed Kubernetes pod migration is feasible while remaining fully transparent to the running application and its clients. The evaluations show that it is possible to perform a fully stateful migration without compromising the application execution, the network connections, nor the data correctness.

4.1 Introduction

A leading design principle of modern cloud computing systems is that cloud resources should be treated as cattle rather than pets. In other words, cloud resources such as virtual machines and containers (the cattle), and the applications they contain should be designed so they may fail and be easily replaceable with other ones, without impacting the overall system (the herd). This design principle has proven to be extremely successful for guaranteeing the robustness and stability of many cloud platforms such as the popular Kubernetes container orchestration platform [130].

However, sensible as it is, this principle is often interpreted in an excessive fashion such as "containers must remain totally stateless" or "any management action performed on a container essentially turns it into a pet" [146]. On the contrary, an arguably rational management of "cattle" resources may be to care for them and try to maximize their usefulness within the platform, while keeping in mind that they should not create service disruption in case they die – for instance, because of a crash of the server which runs them.

One topic in which "rational" management of container resources may provide tangible benefits without violating the pet/cattle principle is the choice of which server should be used to execute a given container. In geo-distributed environments such as fog computing

platforms, every server may be installed in a different location, and each container may need to run in a specific server [147]. When runtime conditions such as user location change, it may become useful to migrate the concerned container to another location, for example, to maintain network proximity between the container and the user(s) making use of it.

For example, Augmented Reality based on mobile devices enables the creation of games such as escape rooms and seeking treasures within an entire city [148]. These kinds of games require very tight interactions between the players' devices and the game application. As the players move within the city, it becomes desirable to migrate the application with its full state to closeby servers ideally reachable from the user device through a single network hop. However, to maintain game continuity, it is important to reduce the downtime during which the game is unresponsive while being migrated.

This chapter introduces MyceDrive, a seamless pod migration technique integrated within the Kubernetes container orchestration system. Following the pet/cattle analogy, MyceDrive avoids slaughtering a healthy running pod and waiting for Kubernetes to restart a new one when it is, instead, possible to migrate it to a different server. Migration is totally transparent to the application running inside the pod as well as the clients having open TCP connections to the migrated pod. MyceDrive relies on DMTCP [115] to checkpoint the container's memory state and open network connections at the migration time, and to resume this saved state in the restarted pod. MyceDrive is easily integrated with Kubernetes deployments. It is fully compatible with ARM and x86 CPU architectures, with no need for any specific OS, hypervisor, or kernel modules. This chapter focuses on migrating pods without considering the migration of data volumes within them. We address migration in Chapter 5.

MyceDrive is designed to operate in geo-distributed environments where each server is placed in a different location, and network links between servers may be slow. Even in such difficult operating conditions, we show in our evaluations that it introduces low service downtimes during pod migration, up to 7x faster than state-of-the-art technologies.

4.2 System Design

Migrating a Kubernetes pod from one server to another is conceptually very simple. In principle, one simply needs to stop and checkpoint the memory state and system-level resources of the "source" pod, transfer the checkpoint data to the destination server, then

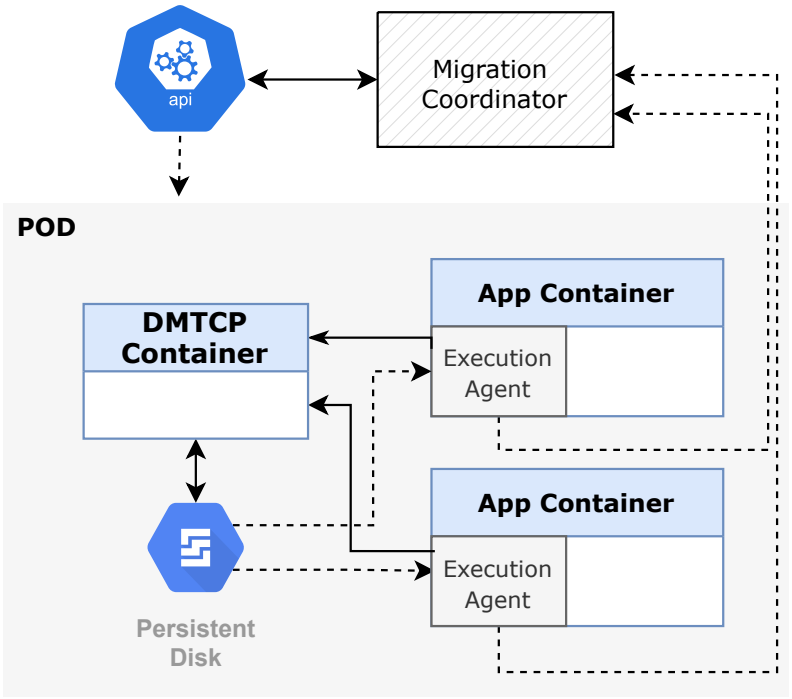


Figure 4.1 – MyceDrive System architecture.

restart a new pod from this checkpoint. We do not address the migration of the pod's disk volumes in this paper as this was the topic of a separate paper [37].

We address the migration of pods within Kubernetes deployment systems because it is the most popular orchestration tool and fairly accepted for Fog Infrastructures. Our model is capable of migrating a pod and all its containers. The pod can be moved inside a Federation as long as the application images and the deployment files are the same.

As illustrated in Figure 4.1, MyceDrive's architecture is composed of two elements. First, an EA is integrated into every application container. It is in charge of controlling the container's lifecycle such as triggering a checkpoint and restarting from a saved checkpoint. Second, a MC is deployed out of the application pod (for example, in the server running the Kubernetes Control Plane). It is in charge of interacting with the Kubernetes API to start or finish pods and coordinating the migration by sending requests to the EA. To issue checkpoint and restart commands, the EA interfaces with a lightweight DMTCP container which runs the `dmtcp_coordinator` in charge of managing every application process and creating the checkpoint.

4.2.1 Execution Agent

We enrich pods to enable migration, they include an EA for every application and an additional DMTCP container per pod. This way, every process gets started using the `dmtcp_launch` command. This can be done with a few simple modifications in the pod specification file. Listing 1 shows the modified specification of a simple pod running the nginx [1] web server.

When a container starts, it executes an "entry point" script in charge of spawning one or more processes within the container. This entry point is included in the container image together with all the files needed to execute it. To allow DMTCP to checkpoint these processes, we need to start them using the DMTCP wrappers. Lines 2-5 of the pod specification request the creation of a container running an enriched version of the standard nginx image. This can be done without modifying the container's implementation itself by simply making use of the layered structure of Docker container images [149]. Instead of modifying the container image, we create an additional layer containing the `dmtcp_launch` entry point script that overrides the original one.

To execute DMTCP in the application containers, we need to make DMTCP's binaries and libraries available in these containers. This applies to every application container within the pod, as well as the additional DMTCP container. Unfortunately, the DMTCP tool includes a large library with all required versions of the system call wrappers, as well as a number of dependencies to other software packages, ending up with a total size of about 200 MB. We prefer not adding these files in the additional image layer itself as this would unnecessarily increase the image size and potentially slow down the pod deployment process. Instead, we group the necessary files in a read-only data volume which is pre-staged in the worker node and mounted by every concerned process. Lines 7-9 in the pod spec request this volume mount in the `dmtcpcontainer` container.

The new container entry point is a generic script that must be able to wrap any created process. It, therefore, needs configuration containing the list of processes it should start and the address of the Migration Controller where these processes should connect. Lines 11-16 of the pod spec provide these configurations. The EA uses line 14 to start the container application process. It then remains inactive until requested to checkpoint the pod.

The application containers need to issue the checkpointing command when requested to do so. To maintain the consistency of the pod state before and after migration, the

1. `https://www.nginx.com/`

```yaml
containers:
  - name: nginxcontainer
    image: enrichednginx
    ports:
    - containerPort: 80
    volumeMounts: # Shared volume with
    #the necessary binaries
    - name: dmtcp-shared
      mountPath: /dmtcp
    env: # Environment variables with
    #EA's configuration
    - name: MIGR_COOR
      value: coord.api  #hostname to
      #reach the coordinator API
    - name: START_UP
      value: "/usr/sbin/nginx -g 'daemon off;'"
      # Application startup
      # command to be wrapped by DMTCP
    lifecycle:
    # An end script is injected in the lifecycle
    # to make sure the checkpoints are created
      preStop:
        exec:
          command: [ "./end_container" ]
  - name: dmtcpcontainer
  # Container running dmtcp_coordinator
    image: dmtcp:dev
    env:
    - name: DMTCP_CHECKPOINT_DIR
      value: /dmtcp/checkpoints
    volumeMounts:
    # Shared volume with the necessary binaries
    - name: dmtcp-shared
      mountPath: /share
```

Listing 1 – Pod specification file to enable MyceDrive.

checkpointing operation and the container shutdown must be issued atomically. This is done by triggering the `end_container` command as the last operation to be executed before the container stops. This command creates the checkpoint and informs the MC that it can be transferred to the destination node. Lines 18-22 specify this.

The final part of the pod spec (lines 24-31) requests the creation of the container running the DMTCP coordinator. Similar to the application container, this DMTCP container mounts the read-only volume with access to the DMTCP binaries and libraries.

When the pod starts, it creates the nginx and the dmtcp containers with their respective volume mounts and entry points. The entry point of the nginx container starts nginx using the `dmtcp_launch` wrappers which register details about the process with the MC such as the process name, status, and meta-data. Any further process forked by the application process automatically inherits the same wrappers.

4.2.2 Migration Controller

Migrating a pod requires one to coordinate multiple actions to be performed in the source and destination nodes. We also need to preserve the information about the migration, pod status, and meta-data despite the fact that the pod is about to be deleted. This is the role of the Migration Controller (MC).

The MC is a REST API implemented in Python, which needs to run in one node of the Kubernetes cluster. A single MC deployment is able to manage an entire Kubernetes cluster regardless of the number of pods. It provides the following methods:

register: this method registers a new application container. The MC distinguishes normal containers being started within a pod from migrated containers by comparing the container's labels and whether the container is marked for migration with a list of already-registered containers.

remove: this method removes a registered container, receiving as argument the container name and labels. It returns if this container is used in migration or not, informing if it should create a checkpoint or proceed with termination.

migrate: this methods initiates a pod migration and returns a boolean indicating if the migration succeeded or not. It receives as parameters the source node which currently runs the pod, the destination node where the pod should be migrated, a Kubernetes label used to constrain the choice of node where the new pod must be started, and the labels of the application deployment and names of its containers.

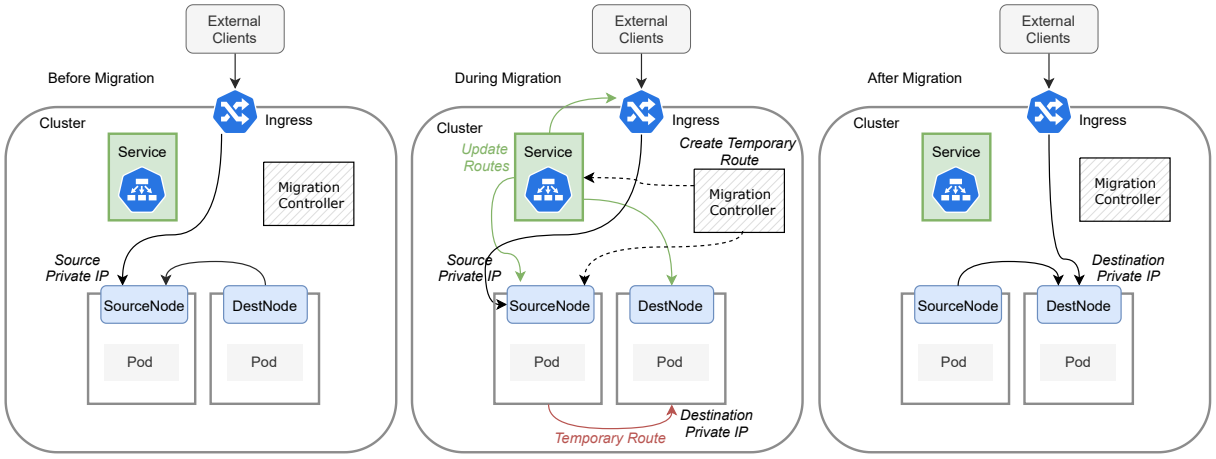


Figure 4.2 – Internal networking routes before, during and after migration.

copy: this method notifies the MC that a checkpoint is ready to be moved. The MC then uses the Kubernetes copy routine to move the checkpoint between containers. It returns a confirmation that the checkpoint was correctly transmitted.

Pod migration may be requested by the users or administrators by calling the `migrate` method of the MC.

4.2.3 Keeping network connections open

Making pod migration transparent to the client processes requires maintaining open network connections across the migration operation. This essentially requires three properties: (i) preserving the TCP socket state such as port numbers, TCP sequence numbers and buffered incoming/outgoing packets; (ii) routing packets from/to the client machines without changing the pod's public IP address; and (iii) reducing the migration downtime as much as possible to avoid connection timeouts.

Preserving socket state is ensured by DMTCP during its checkpoint/restart operations. DMTCP was designed to checkpoint not only individual processes but also entire distributed applications such as MPI, which maintain long-lived network connections between the processes. Contrary to CRIU, which currently does not support this feature, DMTCP, therefore, checkpoints and recreates socket state in the same way it checkpoints file descriptors, pipes, signal handlers and semaphores.

Maintaining network routes: Kubernetes assigns a unique private IP address to every running pod. These addresses are assigned dynamically. As discussed in the next section,

the source pod is still running when creating the destination pod. It is therefore impossible to give the new pod the same private IP address as the old one. On the other hand, Kubernetes Services enable users to provide a stable public IP address that acts as a load-balancer for a group of pods. Services are not implemented using a proxy process but as a set of network routing rules injected in the kernel of all worker nodes in the cluster. When a Service detects a change in the set of pods it acts as a frontend for, it triggers a corresponding reconfiguration of these internal routes. As illustrated in Figure 4.2, we , therefore, place the pods to be migrated behind a Service that ensures that client machines can keep communicating with the new pod using the same public IP address as the old pod.

Keeping the downtime as low as possible: Kubernetes is organized as a set of controllers which periodically observe the system state using a monitoring service, compare the observed state with the desired one, and issue corrective actions when a discrepancy is detected. This implies that a Kubernetes service must first *detect* the creation of the new pod before notification can be issued to daemons running in all worker nodes to request the creation of new internal network routes. Depending on the frequency at which the service probes the monitoring service. It may therefore take up to 10 to 30 seconds before new networking routes are created. This is problematic because it creates long downtimes as perceived by the clients and it possibly breaks networking connections due to TCP timeouts.

As shown in Figure 4.2, we significantly reduce the downtime until network routes to the new pods are re-established in two different ways. First, instead of waiting until the Service discovers the creation of the new pod, we explicitly notify the Service via the Kubernetes API by updating its labels, forcing an immediate update of the entire Service. However, this is not sufficient because there remains a period of time when both pods co-exist, and incoming network traffic may be routed to one or another. Therefore, the EA in the source node also injects an additional temporary route so any traffic addressed to the source pod gets re-routed to the new pod, as indicated in the figure by the red arrow. This temporary route is removed after the source pod has been deleted and internal routes have been re-established.

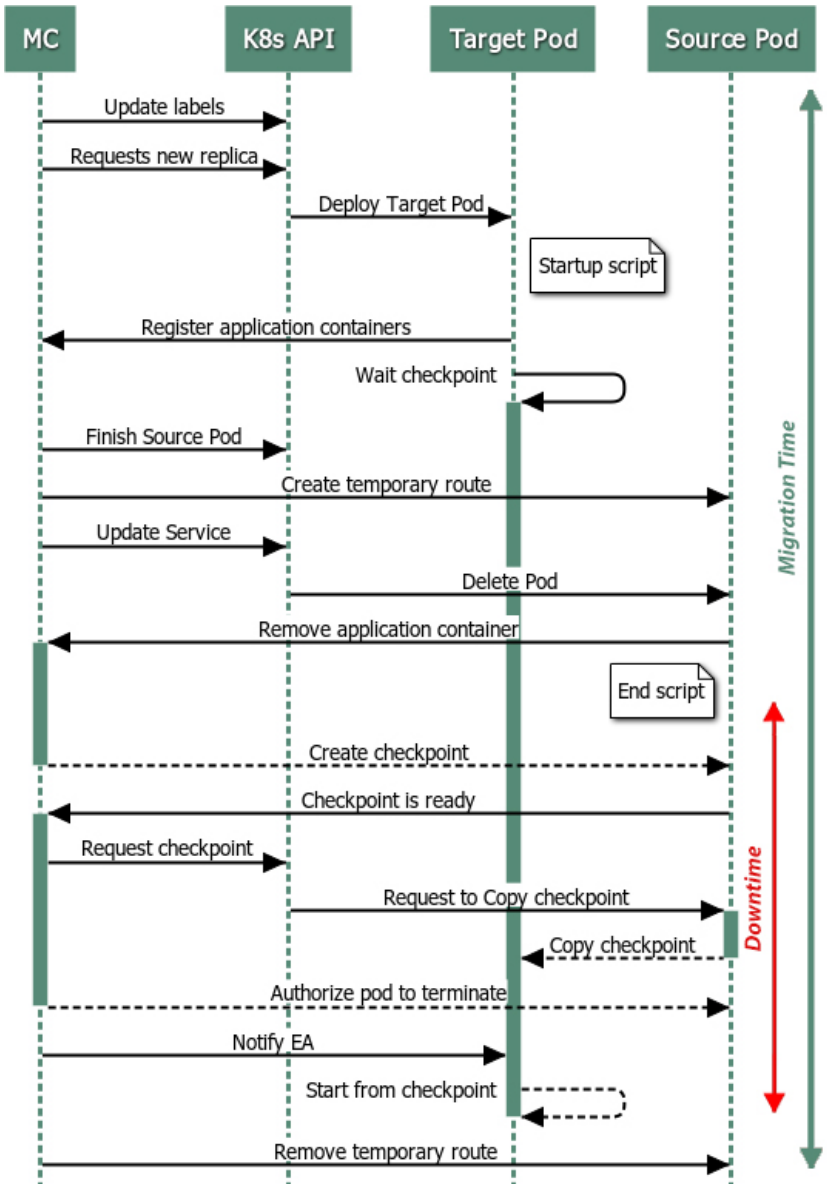


Figure 4.3 – Pod migration process.

4.2.4 Migration coordination

Figure 4.3 illustrates the different actions that are required to migrate a pod after the `migrate` method of the MC is called.

The first step of the migration procedure is dedicated to preparing the migration as well as ensuring that Kubernetes creates the destination pod in the chosen destination node. Pod placement in Kubernetes is chosen in two steps [150]: first, a set of "possible" nodes is determined based on nodes' resource availability as well as user-defined constraints. Second, the possible nodes are ranked, and the highest-ranked one is selected to start the new pod. We use this mechanism by updating the labels attached to the Kubernetes Deployment to specify that only two nodes (the source and destination nodes of the migration) are acceptable to run pods of this application, and by adding an anti-affinity rule which states that the pods should be placed in different machines.

In the second step, we update the Deployment a second time to request that it creates a new pod replica for this application. Because of the placement constraints, Kubernetes can only choose to deploy this new pod in the chosen destination node. The new pod downloads the necessary container image(s) if they are not already available in the local image cache, then it starts its DMTCP container as well as its application container(s). The entry point of the application container(s) registers the containers with the MC. The `register` method identifies that this is a migrated container because it already has a registered container under the same ID. It then blocks the call until a snapshot has been created and copied to the destination node, which effectively delays the launch of the application container.

In the third step, the MC triggers the network route updates by requesting the source node to inject a temporary route to the destination node and by updating the Service, so it immediately notices the new pod. Up until this point, the source pod keeps running normally.

In the fourth step, the MC requests the K8s API to terminate the source pod. This triggers the end script to be executed. The script calls the `remove` method in the MC, and detects if it should snapshot the pod before terminating or if this is a normal pod termination that does not require a snapshot. Once the snapshot has been created and compacted, the EA notifies the MC, which then initiates the copy of this checkpoint from the source to the destination node via the K8s API.

Finally, the MC's `register` method returns the checkpoint name and authorizes the target pod's entry point to restart the application containers from the checkpoint.

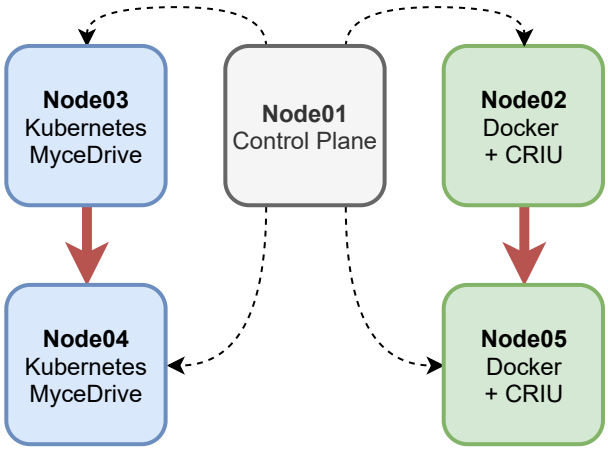


Figure 4.4 – Testbed organization.

4.3 Evaluation

4.3.1 Experimental setup

We evaluate this work using a fog computing testbed composed of five Raspberry Pi (RPI) single-board computers model 3 B+ with quad-core 1.2 GHz CPU, 1 GB of RAM and a 32 GB micro-SD storage card. This type of machine is frequently used to prototype fog computing platforms [25], [28], [109], [110]. The RPIs use the HypriotOS Linux distribution, Kubernetes 1.16 and Docker 19.03.5.

To enable a performance comparison between MyceDrive-driven and CRIU-driven migration, we organize the infrastructure with four workers nodes and a control plane node. As illustrated in Figure 4.4, two workers run Kubernetes while two others have only Docker and CRIU. The control plane node is used to coordinate migration either between the two Kubernetes nodes using MyceDrive, or between the two Docker nodes using CRIU. The two sets of worker nodes use slightly different Linux kernel configurations because CRIU and Kubernetes require the usage of mutually-incompatible kernel modules.

We perform the migration between different nodes, as shown in Figure 4.4. K8s and MyceDrive migrations are performed from Node03 to Node04 whereas CRIU migrations are performed from Node02 to Node05.

We perform a single pod unit migration in our evaluation, however, it does not mean it is only possible to perform a single pod migration at the same time. On the contrary, MyceDrive MC allows multiple migration tasks of different pods simultaneously. As the

Table 4.1 – Evaluation parameters.

Migration technique	K8s, CRIU, MyceDrive
Bandwidth	500 kbps, 1000 kbps, 3000 kbps
Application	MongoDB, Mosquitto, Redis

Table 4.2 – Functional differences between the evaluated container migration approaches.

	K8s	**CRIU**	**Myce-Drive**
Stateful migration	✘	✔	✔
Supports any kernel version	✔	✘	✔
Maintains open network connections	✘	✘	✔
Transparent migration for the application	✘	✔	✔
Transparent migration for the clients	✘	✘	✔
Uses unmodified application images	✔	✔	✘

MC is a service with multiple scopes separated by tasks and it has no dependability between different migrations.

4.3.2 Comparison and metrics

As presented in Table 4.1 we compare three different migration techniques using three real-world applications and various network bandwidth limitations.

Migration techniques We contrast MyceDrive with two other techniques for migrating containers. First, K8s migration consists of simply requesting the Kubernetes API to terminate the running pod and to create a new one in the selected node using the same labeling technique as described in Section 4.2.4.

The second migration technique relies on CRIU to migrate Docker containers. Although this technique is not integrated into Kubernetes, we used the exact same hardware, system-level, and application-level configurations as for the experiments based on Kubernetes. The main exception is that CRIU requires a different set of Linux modules that had to be built separately. The other difference is that we do not include the additional image layer required by MyceDrive, considering that CRIU migration does not require this additional layer. Finally, considering that DMTCP compresses its checkpoints before transmitting them over long-distance network links, we compress CRIU checkpoints as well.

Note that these three container migration techniques are not functionally equivalent. As illustrated in Table 4.2, K8s migration is stateless as it stops the application in the source pod before restarting it in the destination. The only way to preserve the state is to request the application to save its state to the disk when being stopped. Migration is therefore not transparent for the application. This technique does not allow one to maintain open network connections during migration, and therefore, it is not transparent for the clients. On the other hand, CRIU-based migration checkpoints the application's memory state. CRIU imposes strong constraints on the choice of Linux kernel and modules. It also does not maintain open network connections. Its migration is therefore transparent for the application but not for the clients. Finally, MyceDrive is the only migration scheme that combines all these good properties at once. On the other hand, contrary to the other two techniques, it requires creating modified container images with the additional DMTCP layer.

Available bandwidth In a geo-distributed environment, the servers are located close to the end-users but necessarily far from one another. Therefore, the network link between the source and destination servers may rely on commodity networking technologies such as DSL, 4G, and 5G. We, therefore, evaluate the migration approaches in an environment with limited available bandwidth between the nodes. We use typical values found in real-world edge computing environments [151] and reshape the network using the linux tc [2] (traffic control) command.

Migrated applications We compare the different migration techniques using three mainstream applications. The first application is a robust non-relational database based on MongoDB [3]. The storage engine is hybrid since it focuses on in-memory storage besides only using the disk. We exercise this application using a single client that acts as a sensor producing temperature readings in Celsius with a timestamp of when the message was generated. The randomly generated data are produced every 1 ms, and then inserted into the database.

The second application is a Redis [4] in-memory data structure store that can be used as a database, cache, and as message broker. We exercise it with the same workload as for MongoDB.

2. `https://linux.die.net/man/8/tc`
3. `https://www.mongodb.com`
4. `https://redis.io`

Lastly, we use the Mosquitto [5] MQTT message broker. We exercise this application using a producer and a consumer script that simulates a sensor producing data and an application processing them. We use the same data generated for the other applications. At the time the consumer collects the data, it shows the average temperature received and the timestamp delay from the time the message was generated to the time it was received. The biggest gap created by the difference of the timestamps demonstrates the time it was necessary to reconnect both producer and consumer.

All three applications maintain and frequently update a majority of their state in main memory, which constitutes the most difficult scenario for stateful container migration. They also maintain long-lived open network connections to their clients, which implies that breaking network connection is treated as a server failure and has a strong negative impact on client-perceived QoS.

Evaluation metrics Migration is typically evaluated using two main metrics. *Migration time* is defined as the entire duration of the migration operations from the moment migration is initiated by the administrator until the moment when the last operation has completed. Migration time includes the duration of operations such as pulling container images, exposing services, etc. This metric is important for the administrators as it defines the duration of a complex reconfiguration which may require additional resources compared to simply running containers.

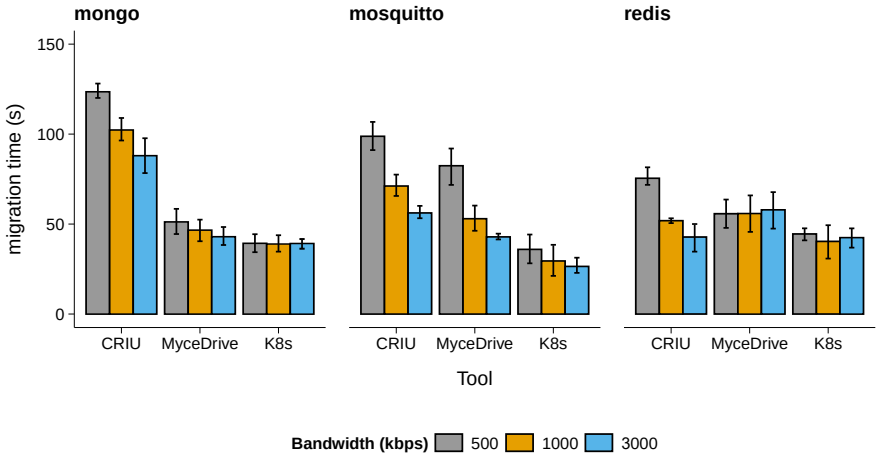
The second important metric is *downtime*, which measures the time during which the container is not running and does not serve any client workload. We measure downtime from the clients' point of view by checking if the application is reachable and whether it answers client requests. Downtime is important for the application and its clients as it defines the time during which the application is not performing its normal operations.

Finally, we evaluate the size of the transferred checkpoints, and the CPU and memory usage of different nodes during migration.

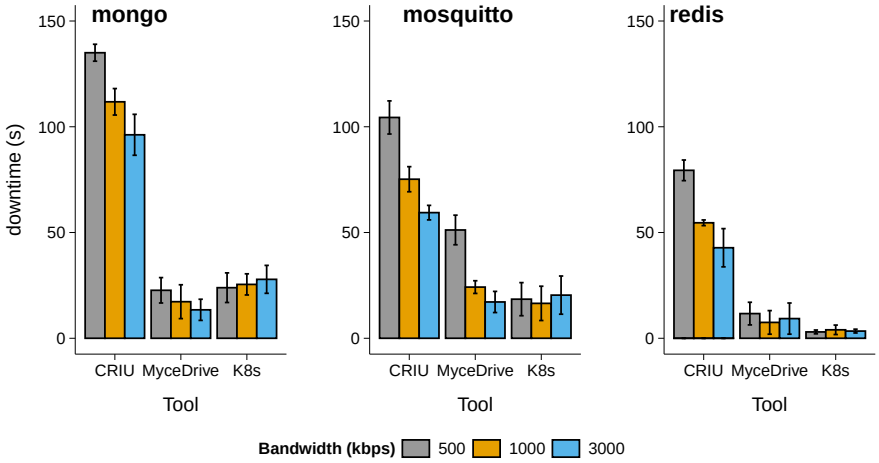
4.3.3 Migration performance

Figure 4.5a shows the migration time for every application and tool over different bandwidth conditions. CRIU has the longest migration time when running MongoDB and Mosquitto applications, followed by MyceDrive and then K8s. In all these cases, migration time decreases when more bandwidth is available between the nodes. For Redis,

5. `https://mosquitto.org`



(a) Total migration duration.



(b) Migration downtime.

Figure 4.5 – Migration times.

the migration times follow the same pattern using CRIU-based migration, but they remain mostly constant for the other two migration techniques, with large standard deviations in the case of MyceDrive. This indicates that migration time is dominated by other factors than the transfer of the memory snapshot. We believe this is due to the time to deploy the target pod before transferring the snapshot.

Figure 4.5b presents the migration downtimes as perceived by the clients. In CRIU-based migration, the downtimes are mostly equal to the migration times because all migration operations take place while the migrated container is stopped. On the other hand, we observe that downtimes are significantly smaller in the case of MyceDrive. This is due to the fact that MyceDrive delays the termination of the source pod as long as

Table 4.3 – Checkpoint sizes in original and compressed size.

Migration technique	**CRIU**	**MyceDrive**	**K8s**
MongoDB	41 MB (6.9 MB)	3.1 MB (2.1 MB)	–
Mosquitto	38.2 MB (5.9 MB)	3.2 MB (3 MB)	–
Redis	18.5 MB (5.1 MB)	1.8 MB (1.7 MB)	–

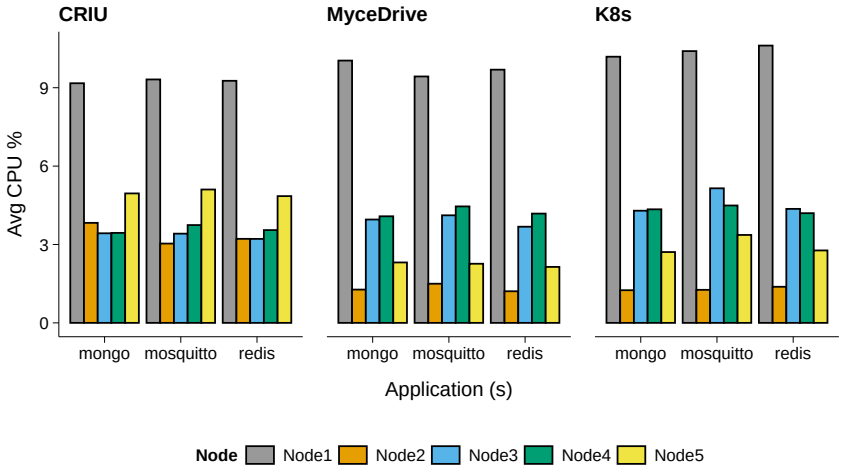
possible until the snapshot has been created. Also, the strategies for keeping network connections open discussed in Section 4.2.3 reduce the downtime as perceived by the clients.

In the case of Redis pod migration, MyceDrive observes a very low downtime compared to the corresponding migration time. This is consistent with the observation that migration time is dominated by the image download and starting operations rather than snapshot transfer. There as well the available network bandwidth does not have a significant influence on downtimes.

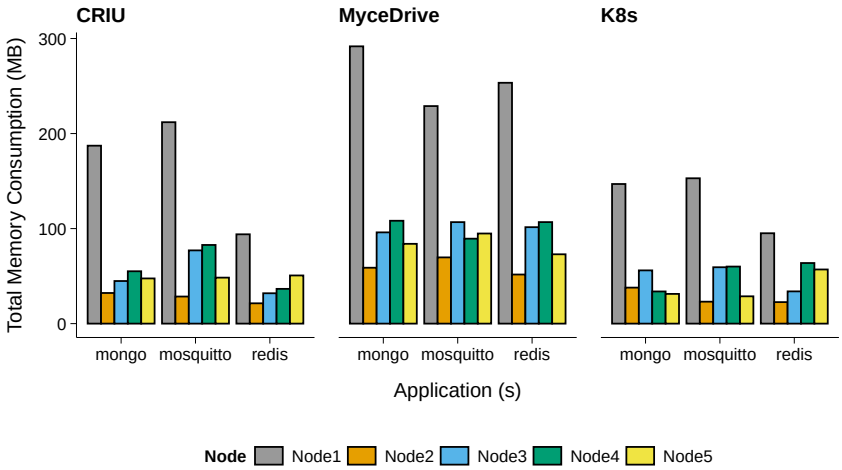
Finally, K8s migration produces the shortest downtimes of all techniques. However, remember that K8s migration is stateless, so no memory snapshot has to be copied during migration. K8s migration, therefore, requires developers to redesign their applications to be able to lose their memory state with no ill effect, which can often be a difficult operation **k8slegacy**.

The two stateful migration techniques experience significant downtime reduction when more network bandwidth is available. Greater bandwidth ensures faster interactions between the servers and reduces the time needed to copy the snapshots from the source to the destination server. Table 4.3 shows the respective uncompressed and compressed checkpoint sizes. We can see that DMTCP generates much more compact snapshots, even though CRIU snapshots seem to obtain better compression ratios.

MyceDrive's downtimes vary largely from one application to another. The main factor determining downtime is the compressed snapshot's size, combined with the available network bandwidth to transfer the snapshot. The largest DMTCP snapshot belongs to Mosquitto, which also happens to have the largest downtimes combined with the greatest variability due to different available bandwidths. This suggests the usage of additional strategies to reduce the size of the snapshot that must be transferred during downtime. In particular, DMTCP includes experimental support for incremental checkpoints **dmtcp-incremental**. Future versions of MyceDrive may therefore consider snapshotting the pod more than once so the initial checkpoint(s) could be transferred before



(a) Average CPU usage during migration.



(b) Total memory usage during migration.

Figure 4.6 – Resource usage during migration using 3000 kbps bandwidth.

stopping the source pod, and only the final checkpoint transfer would incur a downtime. This strategy is known as "iterative pre-copy" in the domain of virtual machine migration **vm-migration**.

4.3.4 Resource usage

Pod migration is a complex operation that requires the system to convey additional tasks while continuing to process its normal workload. We now evaluate the additional resource usage caused by migration, compared to K8S as the baseline of resource usage

for managing pods and containers. We measure CPU and memory usage using the `dstat` [6] tool. In this section, we report only the resource usage measured during migration with 3000 kbps available bandwidth, as there was no meaningful difference with the other evaluated bandwidths.

Figure 4.6a shows the average CPU usage of each node during migration, in every possible scenario with the three migration techniques and the three evaluated applications. Node01 has the highest CPU utilization in all scenarios, as it runs the control plane responsible for managing the migration. CRIU performs container migration from Node02 to Node05. We can see an increase of about 2.5% CPU usage in those nodes whenever the migration is triggered. On the other hand, K8s and MyceDrive migrate pods from Node03 to Node04. We can see an increase of CPU usage in the order of 4% to 5% during migration, with no significant difference between K8s and MyceDrive. We conclude that this is the normal CPU cost of starting and stopping Kubernetes pods, and that MyceDrive does not generate any significant increase in CPU usage compared to K8s. Finally, we note that the CPU usage remains identical regardless of the application.

Figure 4.6b shows the additional memory usage compared to running regular containers or pods without migration. We can see that Node01 bears the greatest cost, with for example an additional 94 MB of memory consumption when migrating the Mongo application using MyceDrive. Other applications observe similar numbers. This is the memory footprint incurred by the MC to manage migration within a Kubernetes cluster. On the other hand, CRIU-based migration does not require a complex migration controller and can be scripted instead, resulting in a lower memory footprint in Node01. Finally, K8s migration incurs the lowest memory consumption in Node01 as no additional software needs to be deployed in the control plane node to organize migration. MyceDrive also generates a slightly greater memory usage in the worker nodes involved in the migration, which corresponds to the memory footprint of the EA which must be attached to every pod.

We conclude that MyceDrive incurs almost no additional CPU costs compared to CRIU and K8s migration, and that the memory footprint of the MC and the EAs remain reasonable.

6. `https://github.com/dstat-real/dstat`

4.4 Conclusion

Migration is an essential functionality in large-scale virtualized platforms as it allows system administrators and application providers to revisit the choice of server which was initially made to run an application when runtime conditions change. Migrating containers is difficult in particular in geo-distributed environments because limited network capacity between the nodes slow down the migration operations and potentially impose unacceptably long downtimes.

We proposed MyceDrive, which implements stateful and fully transparent migration in geo-distributed environments, where neither the migrated application nor the external clients communicating with it need to be aware of the migration operation. MyceDrive is integrated with Kubernetes and, therefore, can be used to migrate entire Kubernetes pods rather than single containers. We showed that, although the total pod migration times are similar to those achieved by CRIU-based migration, it exhibits downtimes up to 7x shorter than those from CRIU.

This chapter demonstrates that the pet/cattle principle which was instrumental in making Kubernetes currently the most popular container orchestration platform does not necessarily impose slaughtering a perfectly running pod and restarting a new one from scratch elsewhere when workload relocation becomes necessary. Instead, it is perfectly possible to migrate pods from one node to another without requiring the application to shut down, store its entire state to disk, and restart from scratch after breaking all the open network connections.

Chapter 5

Incremental Volume Migration

Migration is an essential functionality in large-scale geo-distributed platforms. Contrary to migration within a single data center, long-distance migration requires that the container's disk state should be migrated together with the container itself. However, this state may be arbitrarily large, so its transfer may create long periods of unavailability for the container. In this chapter, we propose to exploit the layered structure provided by the OverlayFS file system to transparently snapshot the volumes' contents and transfer them prior to the actual container migration. We implemented this mechanism within Kubernetes. Our evaluations are based on a real fog computing test-bed and show that our techniques decrease the container's downtime by a factor of 4 when comparing with a baseline that does not use any volume checkpoint.

5.1 Introduction

Migration is an essential functionality in large-scale virtualized computing platforms. Migrating virtual machines is commonly used by data center managers as an enabler for scheduled server decommission, resource consolidation, disaster recovery, vertical scaling, etc [32]. As many applications are moving from VM-based to container-based infrastructures for reasons of simplicity, performance and cost, similar techniques are becoming necessary in container environments as a fundamental system management tool.

Although early container-based platforms relied on the low-level container technologies such as LXC, the increased popularity of container frameworks have resulted from the introduction of high-level tools such as Docker (which provides developer-friendly software development workflows) and Kubernetes (which orchestrates Docker containers at the scale of a cluster or a data center). However, the current container migration techniques are designed at the lowest level only, making them unsuitable for usage in higher-level container orchestration environments.

Container migration is made even more important by the trending usage of container orchestration frameworks such as Kubernetes as a basis for designing "fog computing" environments capable of extending traditional cloud data centers with additional resources located close to the end users [27]–[31]. In this context, container migration may exceed the domain of the system management tools and additionally become a primary platform feature allowing operators to migrate a fog computing application from one location to the next to follow the mobility of human end users.

Pod migration in geo-distributed fog computing environments creates new challenges [98]. In particular, Kubernetes pods may use data volumes to store their application-level files and databases. In single-datacenter environments, these volumes are typically stored in a separate network-attached storage (NAS) [152]. This chapter focuses on the specific question of migrating the data volumes of migrated Kubernetes pods efficiently.

Upon any VM or container migration within the same data center, storage volumes do not need to be migrated as the new VM or container may simply re-attach the same volume. However, using the same technique upon a geo-distributed container migration would imply that the new container should issue long-distance remote data access, which is likely to negate the performance benefits of any such migration. When migrating a container from one fog computing location to another, it is therefore essential to seamlessly migrate its data volumes as well. On the other hand, this operation must be carefully organized in order to minimize the downtime witnessed by external users who may be accessing the container during its migration.

In this chapter, we exploit the OverlayFS layered structure of Docker container volumes to migrate the files that are not being actively modified prior to the actual container migration. This significantly reduces the migration downtime as only a small number of "hot" files must be transferred during the downtime. We integrate this migration technique within Kubernetes and ensure that container migration remains transparent to external users. Our evaluation results based on a real fog computing testbed show that this technique reduces at the best-case scenario, the user-perceived downtime during migration by a factor of 4 compared to a baseline migration process.

5.2 System Design

Migrating a pod from one Kubernetes node to another is in principle very simple: one essentially needs to stop the pod, capture and transfer its state to the destination node, restart a new pod based on the transferred state, and adjust the network routing rules to make the new pod available under the same IP address as the old one. However, implementing this in a geo-distributed Kubernetes setup creates a number of challenges. First, if the state is large this simple strategy may imply substantial downtime while it is being transferred from one fog computing location to another. In this work we specifically focus on the disk state, which may grow arbitrary large over time depending on the pod's activity. Second, one needs to coordinate the different operations to snapshot the pod's state, stop and restart the pod in the correct locations, and re-establish correct routing. We discuss these two challenges in turn.

Fog infrastructure resources are geo-distributed, clusters in the same location provide a limited amount of resources. Regularly the migration occurs between different areas, which are impacted by the available bandwidth and the latency. During a migration, one crucial step is to move the persisted data, usually stored in a volume from the Kubernetes API. Volumes may grow more significantly, and it depends on the running application. Simultaneously, volumes with persisted data have parts that will not change (or not in a substantial time interval since the data is persisted). Commonly, this part can be read-only, which facilitates the migration process. On the counterpart, networking applies constraints in the migration process. Therefore, it is necessary to evaluate the tradeoff between moving a significantly more significant volume in a minimal bandwidth or just reprocessing and recreating the current state of a container in another location.

5.2.1 Volume checkpointing

Kubernetes pods may be composed of one or more Docker containers, which have access to two distinct forms of disk storage. First, each container has access to its "virtual local disk" in the form of a container image. In this image, the bottom layers are read-only through the lifetime of the container. They can thus be copied to the destination nodes prior to the container migration without incurring any downtime. Since these layers constitute the container's image which can be fetched from a Docker repository, we do not need to copy them explicitly from the source to the destination node but rather rely

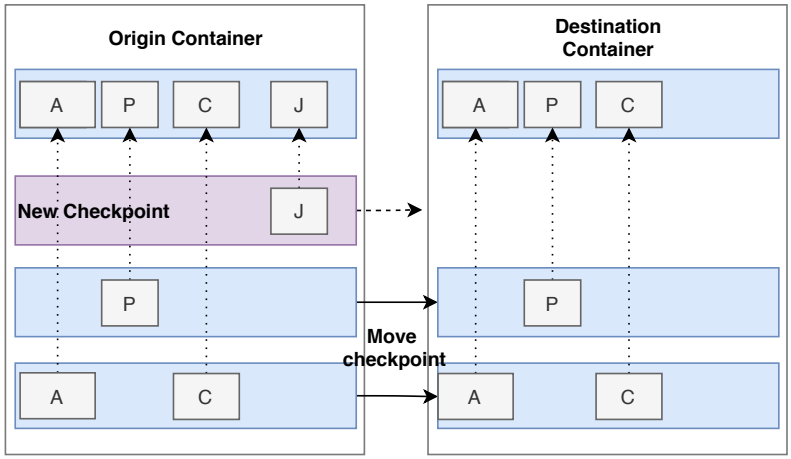


Figure 5.1 – Checkpointing disk volume checkpoints.

on Docker's default behavior which fetches these layers from the repository upon any container deployment, unless they are already available in the local image cache.

The top-level read-write container layer captures all the file system updates issued by the container since its creation. Depending on the container's activity it may contain updated versions of the files from the underlying layers (following a copy-on-write policy), and any new file that the application inside the container may have created. However, in Kubernetes, this container layer is explicitly assumed to be ephemeral. When a pod is stopped, its container layers are simply discarded. When migrating a pod it is therefore not necessary for us to transfer this part of the state.

Persistent storage in Kubernetes is provided in the form of *volumes*, which are virtual disks available to the containers within a pod. Volumes remain persistent even when a pod is stopped, and they are re-attached when the pod is restarted. Kubernetes supports many types of volumes stored either in the same node as the pod or in a local Network Attached Storage (NAS). In our implementation, we use local volumes to maintain proximity between the containers and their disk storage in a geo-distributed environment. However, the same concept also applies to volumes stored in a NAS, which must also be migrated from one NAS to another.

Upon a pod migration, we therefore need to snapshot the state of the pod's volume(s) precisely at the time when the pod has been stopped, and to transfer its content to the destination node before restarting a new pod-based on this disk state. A naïve baseline strategy would simply transfer the container layer's content after the old pod stops. How-

ever, we show in Section 5.3 that this may require long transfer times before the new pod can be restarted.

A better strategy consists of identifying the parts of the container layer that are not currently being modified and transferring them before stopping the old container. To do this, we require that the pod's volume should be formatted using the same layered OverlayFS file system as is used for the container's image. Kubernetes volumes are usually formatted using a non-layered file system such as `ext4`. Note that OverlayFS does not introduce any significant performance overhead compared to `ext4` [81].

When migrating a Kubernetes pod, we use OverlayFS to create a snapshot of the pod's volume(s). Upon this operation, the snapshot content becomes read-only, and a new empty read/write layer is added on top. This operation is transparent for the container's processes which only see the merged file system created by OverlayFS. Transferring the newly created volume snapshot may take time during which new file system updates are issued on the container layer. It is therefore possible to create and transfer more than one snapshot before stopping the container and transferring the last remaining file system updates. Figure 5.1 illustrates this process.

The different operations required to migrate a container are issued by a migration tool that must be included in the container when it is created in the first place. This migration tool implements the following API:

Init Volume: When a pod starts, Kubernetes mounts its volume(s) as a regular non-layered file system. The Init Volume method checks if the volume already contains the "lowerdir" and "upperdir" directories required by OverlayFS, creates them if necessary, and remounts the volume in the pod's containers using the OverlayFS file system.

Create Checkpoint: This method checkpoints the pod's volume. As a result the current top-level layer of the volume becomes read-only (with the content of the checkpoint), and a new empty read-write layer is created on top to capture any future file system updates. The method then remounts the volume on the fly with the new layer structure.

Copy Checkpoint: This method is called at the origin node upon a pod migration to copy one or more checkpoint layers to the destination node. We assume that the network bandwidth is the main bottleneck in a geo-distributed environment, therefore this method compresses the layers before transferring them.

Receive Checkpoint: This method is issued at the destination node to identify the checkpoint that has been copied, and mount it in the new pod following the same strategy as the Init Volume method.

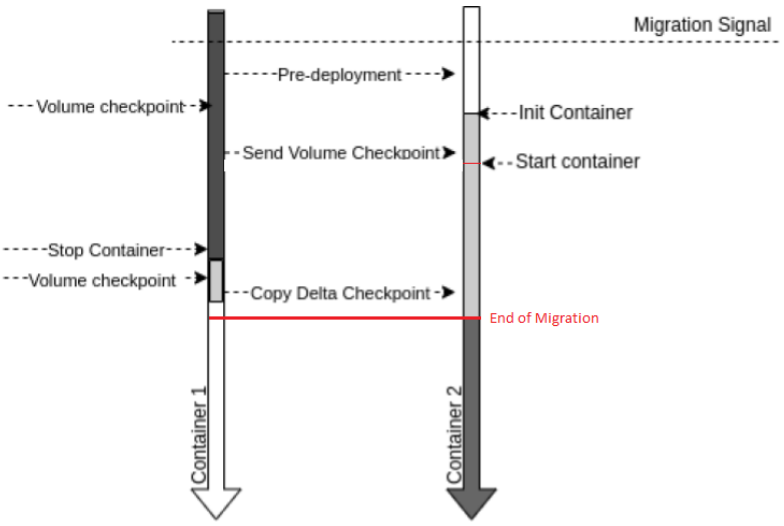


Figure 5.2 – Pod migration process.

End Volume: Whenever a pod is stopped because it is being migrated to another node, this method unmounts the volume and migrates the last file system updates that were not yet transferred. It finally releases the volume to be recycled by Kubernetes.

During a pod migration, the first snapshot captures the entire state of the volume at the time the migration is initiated. After being snapshotted, this content becomes read-only so it can be transferred to the destination node prior to stopping the old container. One or more additional snapshots may be created so that file system updates that took place while transferring the first can be migrated out of the critical path of the container's downtime. The only content which must be transferred during the downtime, therefore, remains small. Transferring this last file system layer takes place during the downtime after the old pod has been stopped, and before the next one can be started. The different operations must be carefully coordinated to minimize downtime, as we discuss next.

5.2.2 Migration Coordination

When a pod migration request is issued, an ephemeral "coordinator" container is created in the origin node, in a new pod, to coordinate the migration. The coordinator has full access rights to the Kubernetes and volume management APIs.

Figure 5.2 depicts the migration process. Migration begins with an initial request from the user or application manager. When the coordinator starts, it receives the migration parameters, the pod name, the destination node, and the number of requested checkpoints.

It then instructs Kubernetes to create a new pod in the destination node by attaching its request for a new pod with Kubernetes labels which constrain its placement on the desired node. The new pod is created with a thin additional image layer that contains an *initializer* script in charge of receiving the checkpoints and setting up the new pod.

The coordinator then creates the first volume checkpoint. This process dumps all the content of the upper layer of the volume to a read-only layer, and creates a new top-level empty layer. The coordinator invokes the Copy Checkpoint function to copy this layer to the destination container. The process may be repeated in case additional checkpoints are required. During this time, the origin container keeps running and potentially issuing file system updates in its new top-level layer. The final steps consist of stopping the old container, invoking the End Volume to migrate the last remaining file system updates, and deleting the volume so the Kubernetes system can recycle it.

At the destination node, every time a new snapshot arrives, the initializer script is in charge of storing it in a new file system layer and remount the volume in the container. After receiving the last file system updates, the initializer invokes the command to start the container's services. It finally updates the pod's labels to indicate to the Kubernetes that it needs to update its load-balancing rules and redirect the network traffic to the new pod instead of the old pod.

As soon as this reconfiguration is completed, the pod migration is finished. The pod's migration remains transparent to its users, which keep addressing the new pod using the same IP address as they were using before the migration. They may observe a downtime period during which the pod remains unresponsive. However, they need not be aware of any detail about the migration.

5.3 Evaluation

In this section, we present the evaluation methodology of our proposal. We provide details regarding the experimental setup and the analysis of the impact of checkpoints alongside the downtime of the migrated services in a fog environment.

5.3.1 Experimental setup

We evaluate this work using a fog computing testbed composed of five Raspberry Pi (RPI) single-board computers model 3 B+ with quad-core 1.2 GHz CPU, 1GB of RAM

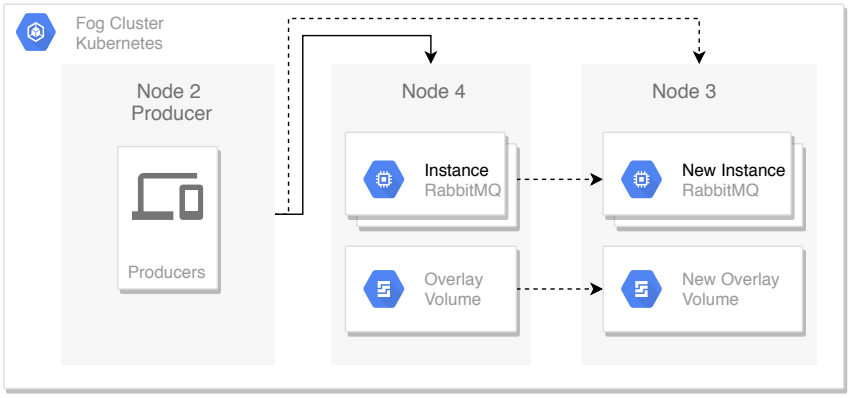


Figure 5.3 – Experimental setup.

Table 5.1 – Evaluation parameters.

# of checkpoints	0, 1, 2
Bandwidth (kbps)	500, 1000, 2000
Pre-pull strategy	Yes, No

and a 32 GB micro-SD storage device. This type of machine is frequently used to prototype fog computing infrastructures [25], [28], [51], [110]. The RPIs use the HypriotOS Linux distribution, Kubernetes 1.16 and Docker 19.03.5. The experimental setup is illustrated in Figure 5.3: the first RPI acts as the Kubernetes master node while the other four can run application pods.

We exercise the system using a RabbitMQ 3.7 persistent MQTT service [153] running inside a pod which is migrated during its execution between different machines. This type of queuing service is frequently used in fog computing platforms [18]. RabbitMQ uses 70 MB of storage for its image content, and an extra 400 MB of state in its disk volume to persist messages before they have been fully delivered. RabbitMQ is therefore representative of a demanding IO-intensive fog application.

We generate a message load using RabbitMQ-PerfTest [154] with one message producer which produces random data in JSON format based on a fixed seed, with 100 messages/s of 128 kB each distributed between two different queues. Messages are sent to RabbitMQ with no consumer to receive them.

In our experiments, we first start the RabbitMQ pod with one container and one 20 GB data volume on Node 4, generate workload with the producer's pod running in Node 2, and then migrate the RabbitMQ pod to Node 3. Each experiment lasts 150 s and the pod migration is issued 60 s after starting the workload.

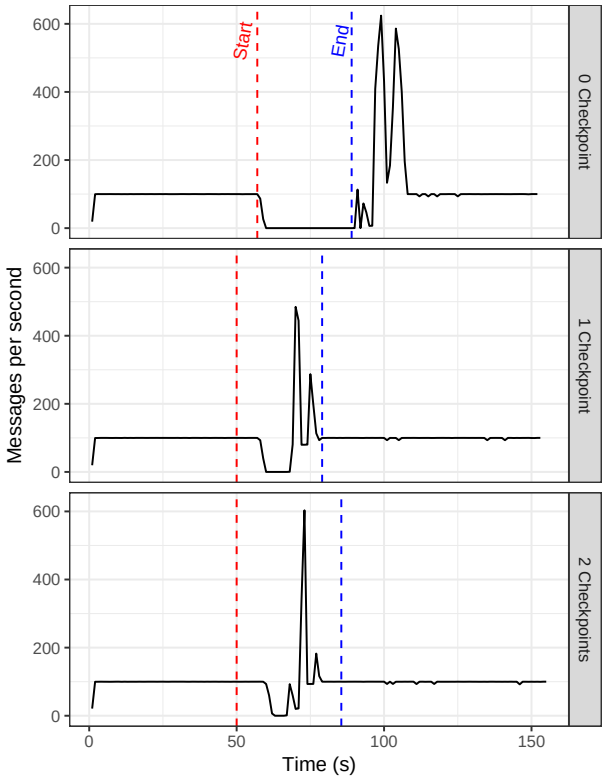


Figure 5.4 – RabbitMQ's message throughput during migration with 2000 kbps bandwidth, pre-pull strategy, and different numbers of checkpoints.

We compare the performance of our pod migration mechanism with that of a baseline strategy which does not checkpoint the pod's volume. The baseline (thereafter named "0 checkpoint") simply stops the old pod, migrates its entire state to the destination node, and restarts the pod in the new location. Table 5.1 shows the tested configurations for the pod migration system. We vary the number of checkpoints from 0 to 1 and 2 checkpoints. We control the available network bandwidth between the worker nodes using the "traffic control" (`tc`) tool available in Linux systems, with 500 kbps, 1000 kbps and 2000 kbps. These values are based on a recent study that highlights the characteristics of today's networking technologies used in fog computing settings [151]. Finally, we test two variants of the migration algorithm, which respectively pull and do not pull the container image layers in the destination node prior to the migration.

5.3.2 Message throughput during migration

Figure 5.4 shows the RabbitMQ message throughput before, during, and after migration while processing a workload of 100 messages/s. In this experiment, we use a network bandwidth of 2000 kbps, and the pre-pull strategy, and vary the number of checkpoints from 0 (baseline algorithm) to 1 and 2. The vertical lines depict the time at which the migration is initiated and completed.

In the baseline algorithm with no volume checkpoint before the migration, the message throughput drops to zero immediately after the start of the migration procedure. We observe a downtime of about 40 seconds until the volume has been transferred and the new container is operational again. Immediately after the restart, the message throughput increases to large values because the message producer delivers all the produced messages accumulated during the downtime. Finally, the throughput returns to its initial value.

In the next experiment with one checkpoint prior to migration, we observe that the downtime starts several seconds after the migration was initiated. The duration of the downtime is also noticeably shorter. This is due to the fact that a smaller amount of data needs to be transferred between the concerned nodes during the downtime. Also, we observe that the spike of messages to be delivered after the migration is smaller than using the baseline algorithm. This is due to the fact that a smaller number of undelivered messages were buffered during the container downtime. A longer downtime results in a higher number of buffered messages sent when the service becomes available again. In busy fog computing environments, this provides an additional benefit to our approach as a shorter downtime creates a smaller backlog of operations that need to be processed after the migration. Finally, when using two checkpoints before migrating the container, we observe that the downtime is further slightly reduced.

The performance gains of increasing the number of checkpoints create a tradeoff between the volume of data to be transferred (which is greater with more significant numbers of checkpoints) and the observed container downtime.

We also notice that the presence of the migration tool does not impact the running application's performance, as no interference is visible in the throughput of the application prior to migration. The agent inside the ephemeral container creates the checkpoints using lazy mounting in the filesystem, with minimum impact on the running application container(s).

5.3.3 Container downtime

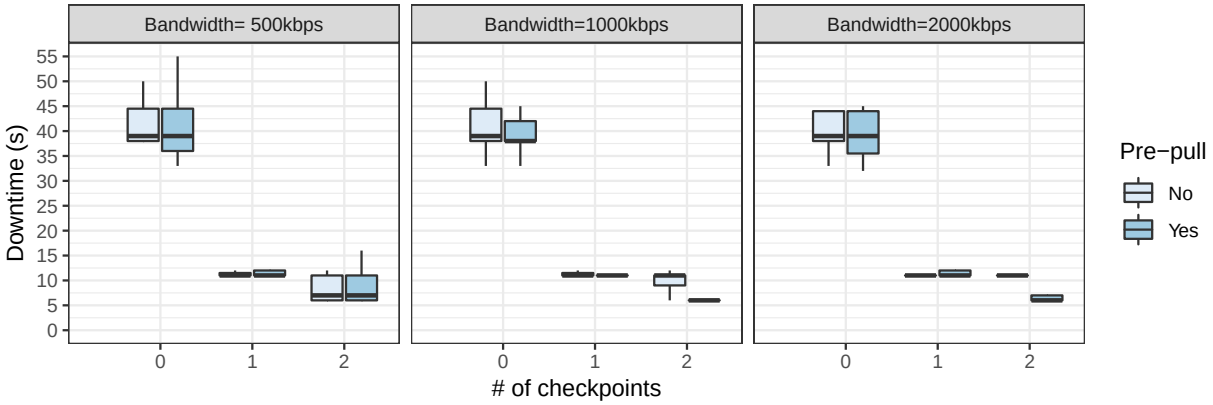


Figure 5.5 – Migration downtime.

Figure 5.5 shows the overall container downtime across the complete set of experimentation scenarios. We run each experiment 15 times, and report the mean, minimum, maximum, 25th, and 75th percentile values. Thus, the upper whisker extends from the hinge to the largest value as the lower whisker goes to the smallest value, and both do not go further than 1.5 * IQR.

In all experiments, the performance benefits of checkpointing the pod's volume before migration are evident. We observe a reduction of the downtime by a factor of 4 between the baseline with no checkpoint and our migration technique with one checkpoint. A second checkpoint further reduces the downtime a little, yet at the cost of increased variability between runs. This is due to the fact that RabbitMQ does not constantly write to disk, so the volume of the second checkpoint varies from one run to the next.

We observe that the available network bandwidth does not significantly influence the downtimes. This is due to the fact that the main bottleneck is created by slow disk I/O (the RPIs use micro-SD cards as their only stable storage).

Finally, we observe that pre-pulling the container layers improves performance a little, especially for situations with low bandwidth. Notwithstanding, migration with more checkpoints provides extra time to pre-download the content of a container image, directly impacting the low bandwidth network. However, it does not provide further improvements in a scenario where just one checkpoint is used, especially with a larger bandwidth. On the other hand, when two checkpoints are used, more time is provided to pre-download the images since the service stays up for more time before it is stopped to be replaced by a new one. It happens because there is a fixed interval between creating one check-

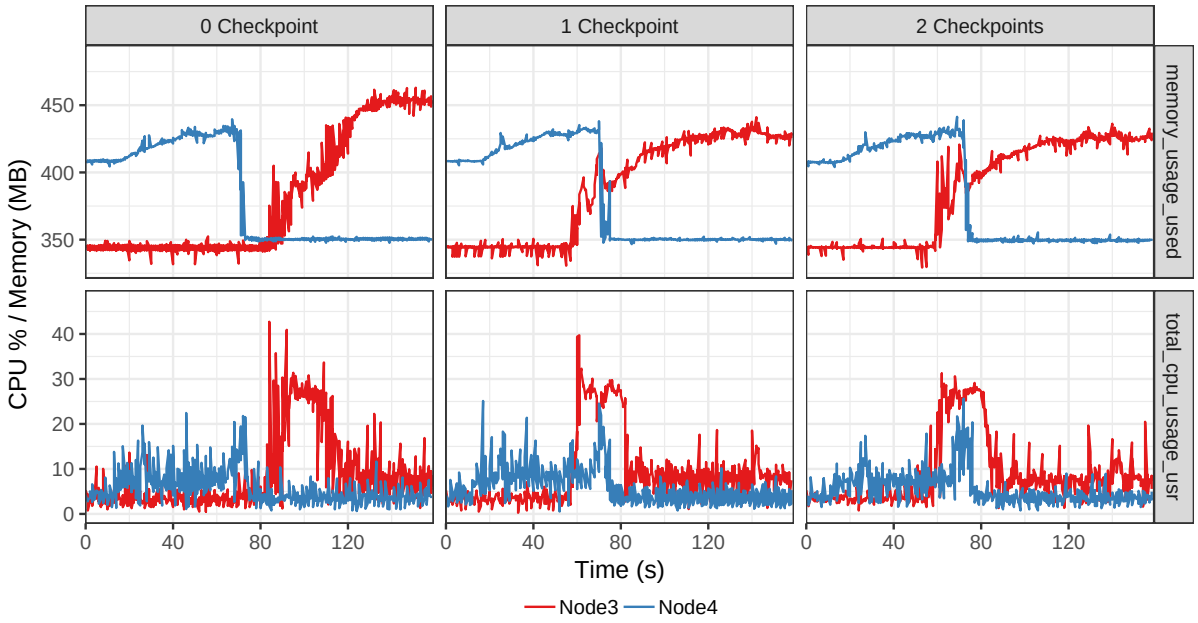


Figure 5.6 – CPU and memory usage during container migration.

point after another, which is not arbitrarily defined and can be changed under different circumstances.

5.3.4 Resource usage

In a fog computing infrastructure, resources may be scarce. It is therefore important to use them with caution. On the other hand, container migration necessarily creates an additional burden on the infrastructure.

Figure 5.6 shows the CPU and memory usage of the two worker nodes respectively hosting the old and the new container during migration, when using zero, one, or two checkpoints. These experiments exploit the pre-pull strategy as we know that it provides better performance than its no pre-pull counterpart.

Before migration, we observe that Node 4 has more significant CPU and memory usage than Node 3. This is expected, as Node 4 runs the RabbitMQ pod, whereas Node 3 remains idle. When the migration is initiated, the memory usage of Node 3 increases as it receives the checkpoints and writes them to the disk. Once this is finished, the memory usage in Node 3 matches that of Node 4 before migration. Conversely, as soon as the old pod gets deleted in Node 4, its memory usage drops quickly.

We observe that migration is more CPU-intensive in the destination node than in the origin node. This is due to the fact that the destination node must start a new container, create a local layered file system, receive checkpoints, and remount the file system. Also, as soon as the migration is finished, the RabbitMQ application needs to process all the delayed messages during migration, which generates additional load. This increased CPU usage is more important in the scenario with no checkpoint compared to one and two checkpoints, as more messages have been delayed due to a more extended container downtime.

At the beginning of the experiment, node 4 has a higher utilization of CPU and memory since the service is currently running. As the migration request arrives, node 3 starts to download and prepare the images. Afterward, it receives the first checkpoint, which is the bigger one in size. At this point, we can see a pick of CPU utilization to mount and prepare the persistent volumes. Later on, the CPU utilization keeps higher than usual until it receives the last checkpoints, then remounts and loads into memory. Thus during this process, it is noticeable the increase in the memory utilization of node 3, which commences loading the persisted data into memory. On the other side, node 4 starts to reduce its memory allocation; however, CPU utilization is still higher since it has to checkpoint the current state. As long as the last checkpoint is created, its CPU and memory utilization go down.

For the scenario where checkpoints are not used, we see higher usage of CPU, and the memory stays idle longer, around 16% usage. It happens since all tasks are run at once, as soon as the service goes down, the migration is performed synchronously. Download the image, build it, receive the persistent data to resume the service, and expose the new service. Besides, it is crucial to pinpoint at the end of the experiment, where the checkpoints are equal to 1 and 2, after execution time 150s, the experiment is done and the services are stopped, and the logs are collected. We do not see the same behavior for a configuration with no checkpoints, this happens since the experimentation is yet not done because it has a more extended downtime, and the perf application is still running until all its data is consumed.

In all scenarios, we notice that resource usage remains reasonable and not significantly different from that of a running container with no migration.

5.4 Conclusion

Container migration is an essential functionality in large-scale geo-distributed platforms such as fog computing infrastructures. Contrary to migration within a single data center, long-distance migration requires that the container's disk state should be migrated together with the container itself. However, this state may be arbitrarily large, so its transfer may create long periods of unavailability for the container.

We proposed to exploit the layered structure provided by the OverlayFS file system to transparently snapshot the volumes' contents and transfer them prior to the actual container migration. We implemented this mechanism within Kubernetes and demonstrated that it reduces the container's downtime during migration at most by a factor 4 compared to a baseline with no volume checkpoint.

Finally, it is possible to achieve a complete solution for geo-distributed pod migration by combining the techniques present in Chapter 4 with the ones in this Chapter. Moving the in-memory content and the attached volumes is vital to migrate Kubernetes deployments effectively.

Chapter 6

Conclusion

This final chapter presents the conclusions of this thesis and indicates a number of future directions to broaden the range of the proposed techniques aiming to improve pod migration.

6.1 Summary

The outstanding growth of data production caused by the adoption of new devices and sensors grants new opportunities for innovation. For example, many companies are proposing real-time or near real-time applications, creating a whole new family of applications that require low latency between the users and the application instances serving them.

Cloud providers grant many advantages thanks to the usage of sophisticated cloud infrastructures. However, such centralized infrastructures are not always able to cope with the requirement of these types of applications (e.g., augmented reality, outdoor games, and health tracking). This creates an excellent opportunity for new paradigms such as Fog and Edge computing. Nevertheless, fog/edge computing are recent concepts that are still being matured, promising to meet the expected requirements of future low-latency applications.

A latency-sensitive application requires on-time responses with a very tight round-trip time. The end-to-end latency depends on multiple elements such as the geo-location of the servers and users, the physical and virtual infrastructure, and routing. With fog computing, the providers can place their resources in many locations and strategically cover multiple points of interest. This allows users to reach nearby servers and access the application's services quickly by the simple fact of being close to each other. However, simply placing application instances as close as possible to their end users at creation time is not enough to meet the requirements and convey an acceptable service quality in the long term. People are constantly moving, so even if the resources are created nearby

clients, the clients might end up moving far their initial location. This means that the initial low-latency guarantee may not be sustained over time.

In this thesis, to address this problem, we explored wide-area application migration with its containers and volumes by proposing a set of tools to achieve this goal. These tools allow transparently migrating services from location A to location B without compromising the user's connections with the application or incurring data loss. At the same time, the migration tool handles the network routes and transparently redirects the application requests to the new set of resources.

The **first contribution** proposes MyceDrive, a seamless pod migration tool for geo-distributed Kubernetes. MyceDrive performs stateful migrations because it grabs the whole container image and in-memory state to transparently migrate them to another location. Therefore, it avoids killing a healthy running pod and waiting for Kubernetes to restart a new one when it is, instead, possible to migrate the running pod to a different server. Migration is transparent to the application running inside the pod as well as to the clients. This is possible since MyceDrive handles the network reconfiguration by creating a temporary route to redirect incoming requests addressed to the old pod to the migrated one until Kubernetes has created a new route. The evaluation in a real fog computing testbed shows that, even with poor network conditions, MyceDrive achieves lower downtimes when compared to the state-of-the-art technologies, reestablishing the container execution up to seven times faster.

The **second contribution** proposes a tool that migrates data volumes. Widespread deployment of an application comprises not only the running application (a set of containers) but also the volume(s) that store data. This contribution complements the first contribution by enabling volume migration, which means that the persisted data will also be migrated after keeping the proximity of the pod to its volumes. It performs incremental migration by exploiting the Overlay file system to create incremental checkpoints. The first checkpoint migrates the files that are not being actively modified before the container migration. Later, the system creates additional checkpoints with only the remaining differences (i.e., files that changed and new files) until the migration is fully concluded. The evaluation compares with a baseline where there is no usage of checkpoints, in a real fog computing testbed. Experiments show that this technique reduces the user-perceived downtime during migration by a factor up to 4.

The two contributions were designed considering the existence of a widely geo-distributed Kubernetes, which uses as resources RPi boards and have more limited band-

width when compared to cloud servers. The contributions of this thesis grant a complete migration approach. Although this work was realized in the context of a geo-distributed Kubernetes cluster, we believe similar techniques may be applied using different orchestrators.

6.2 Future Directions

The set of contributions in this thesis enables a complete migration approach by identifying every component that constitutes a Kubernetes deployment and proposing, comparing, and creating alternatives to perform a stateful pod and volume migration. However, even though it performs better than the state-of-the-art approaches, it may still use further improvements. The following section discusses the remaining opportunities to improve the techniques proposed in this thesis.

6.2.1 Incremental DMTCP Migration

MyceDrive relies upon the DMTCP tool to create a complete snapshot of the current in-memory state of a pod. MyceDrive creates a single snapshot with the entire pod state, which may take time to copy into the destination server. It is, therefore, a limitation that does not allow the possibility of a live migration using incremental pod checkpoints.

However, DMTCP provides the option of creating incremental checkpoints by using HBICT (Hash-Based Incremental Checkpointing Tool) [155]. HBICT gives support for delta-compression relative to the previous checkpoint, granting the opportunity for the development of an incremental in-memory migration.

With an incremental migration, the application downtime would reduce significantly because most of the data transfers could be performed before stopping and migrating the pod, thereby providing lower delay and performance variation for the application.

6.2.2 Migration within a fog federation

The geo-distribution of computing resources requires an advanced management approach to optimize the usage of resources. Fog computing environments must address an extra level of complexity to manage resources compared to a cloud environment since they are spread over multiple locations. A common approach is to aggregate resources in

different locations as separate Kubernetes clusters [49]. However, these Kubernetes clusters require global management. The standard solution organizes multiple clusters in a federation [21]. This brings a new layer of complexity to the management of containers and affects how the migration should perform.

Migration within a single geo-distributed cluster happens typically inside the same private network and does not require the creation of a complex temporary route to the second cluster. However, a migration that happens externally from one cluster to another requires an extra level of reconfiguration to create a temporary route. It means redirecting requests that arrive in one cluster to the other, avoiding losing messages, and reaching the new container internally, inside a new pod over a different region/location. In the long run, an external load balancer should direct user requests directly to the destination cluster.

All this creates the opportunity to develop a load balancer that is migration-aware and can create temporary routes as a migration happens to avoid any possible disconnection or losing any message. The temporary route would still imply a higher latency during migration. Thus, the load balancer should be aware of the migration steps to reconsolidate the best route (the one with the lowest latency), after the migration happens.

6.2.3 Migration as a integrated Kubernetes component

Kubernetes is a widely adopted open source orchestration platform. It consists of many components to manage containers, workloads, and networking, that are maintained and updated daily. The techniques in this thesis work inside of Kubernetes, and they are expected to be easily unified with the Kubernetes platform. However, the current prototype is not a fully integrated Kubernetes component, and it will require extra effort to enable migration as a native functionality of this platform.

One opportunity is the adaptation of both contributions into a single component inside of Kubernetes, improving the engineering aspects and reducing the extra complexity to make an application deployment 'migration-ready'.

Another alternative would be the creation of a plugin for Kubernetes containing the solutions proposed in this thesis. The plugins would bring both the stateful pod migration and the incremental volume migration and could be attached to a Kubernetes deployment according to the requirements of every environment.

6.3 Closing Statement

Mobility is part of humankind as people move for many reasons such as entertainment, work, socialization, food, and so on. Currently, network channels are not able to cope with the application's requirements of very low-latency communication, and even if the computational resources are brought nearby to the users, peers, or any mobile device, the tendency is that they will move and, once again, increase the communication latency. Therefore, it is necessary to maintain the applications nearby users by exploring the opportunity of application migration. Migration extends the potential of fog platforms and builds a future where applications can seamlessly move alongside their users and maintain excellent application performance across a wide variety of mobility patterns.

BIBLIOGRAPHY

[1] O. D. Steven and Vidya Sargur, « The Changing Landscape of Data Ingest, Continuous Delivery and Streaming Applications », https://community.cloudera.com/t5/Community-Articles/The-Changing-Landscape-of-Data-Ingest-Continuous-Delivery/ta-p/288341, 2020.

[2] S. Sinha, « State of IoT 2021 Number of connected IoT devices growing 9% to 12.3 billion globally, cellular IoT now surpassing 2 billion », https://iot-analytics.com/number-connected-iot-devices/, 2021.

[3] M. M. Rathore, A. Paul, W.-H. Hong, H. Seo, I. Awan, and S. Saeed, « Exploiting IoT and big data analytics: Defining Smart Digital City using real-time urban data », *Sustainable Cities and Society*, vol. 40, 2018.

[4] G. Suciu, E. G. Ularu, and R. Craciunescu, « Public versus private cloud adoption — A case study based on open source cloud platforms », *in Proc. TELFOR*, 2012.

[5] GCP, « Google Cloud Platform », https://cloud.google.com/gcp/.

[6] AWS, « Amazon Web Services, Inc. », https://aws.amazon.com/.

[7] Microsoft, « Azure », https://azure.microsoft.com/.

[8] GlobalDots, *13 Key Cloud Computing Benefits for Your Business*, https://www.globaldots.com/resources/blog/cloud-computing-benefits-7-key-advantages-for-your-business/, 2018.

[9] Infotechlead, *Datacenter space to grow to 1.94 bn square ft in 2018: IDC*, https://infotechlead.com/it-statistics/datacenter-space-grow-1-94-bn-square-ft-2018-idc-26513, 2014.

[10] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan, « Volley: Automated Data Placement for Geo-Distributed Cloud Services », *in Proc. NSDI*, 2010.

[11] Gartner, *Why Organizations Choose a Multicloud Strategy*, https://www.gartner.com/en/conferences/apac/infrastructure-operations-cloud-india/gartner-insights/swg-why-organizations-choose-a-multicloud-strategy, 2020.

[12] J. C. dos Anjos, M. D. Assunção, J. Bez, C. Geyer, E. P. de Freitas, A. Carissimi, J. P. C. L. Costa, G. Fedak, F. Freitag, V. Markl, P. Fergus, and R. Pereira, « SMART: An Application Framework for Real Time Big Data Analysis on Heterogeneous Cloud Environments », *in Proc. CIT*, 2015.

[13] S. Abolfazli, Z. Sanaei, M. Alizadeh, A. Gani, and F. Xia, « An experimental analysis on cloud-based mobile augmentation in mobile cloud computing », *IEEE Transactions on Consumer Electronics*, vol. 60, *1*, 2014.

[14] B. N. Silva, M. Khan, C. Jung, J. Seo, D. Muhammad, J. Han, Y. Yoon, and K. Han, « Urban Planning and Smart City Decision Management Empowered by Real-Time Data Processing Using Big Data Analytics », *Sensors*, vol. 18, *9*, 2018.

[15] J. Bulao, *How Fast Is Technology Advancing in 2021*, Techjury, https://techjury.net/blog/how-fast-is-technology-growing, 2021.

[16] R. Mahmud, R. Kotagiri, and R. Buyya, « Fog Computing: A Taxonomy, Survey and Future Directions », *in Internet of Everything: Algorithms, Methodologies, Technologies and Perspectives*. Springer Singapore, 2018.

[17] « IEEE Standard for Adoption of OpenFog Reference Architecture for Fog Computing », *IEEE Std 1934-2018*, 2018.

[18] A. Ahmed, H. Arkian, D. Battulga, A. J. Fahs, M. Farhadi, D. Giouroukis, A. Gougeon, F. O. Gutierrez, G. Pierre, P. Souza Junior, M. Ayalew Tamiru, and L. Wu, *Fog Computing Applications: Taxonomy and Requirements*, http://arxiv.org/abs/1907.11621, 2019. arXiv: 1907.11621.

[19] R. K. Naha, S. Garg, D. Georgakopoulos, P. P. Jayaraman, L. Gao, Y. Xiang, and R. Ranjan, « Fog Computing: Survey of Trends, Architectures, Requirements, and Research Directions », *IEEE Access*, vol. 6, 2018.

[20] J. Hasenburg, M. Grambow, and D. Bermbach, « Towards a Replication Service for Data-Intensive Fog Applications », *in Proc. ACM SAC*, 2020.

[21] F. Faticanti, D. Santoro, S. Cretti, and D. Siracusa, « An Application of Kubernetes Cluster Federation in Fog Computing », *in Proc. ICIN*, 2021.

[22] A. Ahmed and G. Pierre, « Docker Container Deployment in Fog Computing Infrastructures », *in Proc. IEEE EDGE*, 2018.

[23] B. Burns and D. Oppenheimer, « Design Patterns for Container-based Distributed Systems », *in Proc. HotCloud 16*, 2016.

[24] S. J. Vaughan-Nichols, *What is Docker and why is it so darn popular?*, ZDNet, `https://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/`, 2018.

[25] P. Bellavista and A. Zanni, « Feasibility of Fog Computing Deployment based on Docker Containerization over RaspberryPi », *in Proc. ACM ICDCN*, 2017.

[26] K. Hightower, B. Burns, and J. Beda, *Kubernetes: Up and Running Dive into the Future of Infrastructure*, 1st edition. O'Reilly Media, Inc., 2017.

[27] A. Fahs and G. Pierre, « Proximity-Aware Traffic Routing in Distributed Fog Computing Platforms », *in Proc. ACM/IEEE CCGrid*, 2019.

[28] C. Wöbker, A. Seitz, H. Mueller, and B. Bruegge, « Fogernetes: Deployment and management of fog computing applications », *in Proc. IEEE/IFIP NOMS*, 2018.

[29] M. Chima Ogbuachi, A. Reale, P. Suskovics, and B. Kovacs, « Context-Aware Kubernetes Scheduler for Edge-native Applications on 5G », *Journal of Communications Software and Systems*, vol. 16, *1*, 2020.

[30] J. Santos, T. Wauters, B. Volckaert, and P. De Turck, « Towards Network-Aware Resource Provisioning in Kubernetes for Fog Computing applications », *in Proc. NetSoft*, 2019.

[31] W.-S. Zheng and L.-H. Yen, « Auto-scaling in Kubernetes-Based Fog Computing Platform », *in Proc. ICS*, 2018.

[32] Rajesh K, *Importance of Virtual Machine Migration in Server Virtualization*, excITingIP.com, `https://bit.ly/3ih5lQR`, 2011.

[33] S. Oh and J. Kim, « Stateful Container Migration employing Checkpoint-based Restoration for Orchestrated Container Clusters », *in Proc. ICTC*, 2018.

[34] R. Sturm, C. Pollard, and J. Craig, « Chapter 5 - Application Management in Virtualized Systems », *in Application Performance Management (APM) in the Digital Enterprise*, R. Sturm, C. Pollard, and J. Craig, Eds., 2017.

[35] W. Hu, A. Hicks, L. Zhang, E. M. Dow, V. Soni, H. Jiang, R. Bull, and J. N. Matthews, « A Quantitative Study of Virtual Machine Live Migration », *in Proc. ACM CAC*, 2013.

[36] D. S. Linthicum, « Moving to Autonomous and Self-Migrating Containers for Cloud Applications », *IEEE Cloud Computing*, vol. 3, *6*, 2016.

[37] P. Souza Junior, D. Miorandi, and G. Pierre, « Stateful Container Migration in Geo-Distributed Environments », *in Proc. IEEE CloudCom*, Dec. 2020.

[38] P. Souza Junior, D. Miorandi, and G. Pierre, « Good Shepherds Care For Their Cattle: Seamless Pod Migration in Geo-Distributed Kubernetes », *in Proc. IEEE ICFEC*, May 2022.

[39] RightScale, *State of the Cloud*, Flexera, `https://resources.flexera.com/web/media/documents/rightscale-2019-state-of-the-cloud-report-from-flexera.pdf`, 2019.

[40] Google, « Data centers », `https://cloud.google.com/about/locations`.

[41] T. Voith, K. Oberle, and M. Stein, « Quality of service provisioning for distributed data center inter-connectivity enabled by network virtualization », *Future Generation Computer Systems*, vol. 28, *3*, 2012.

[42] S. Wang, A. Zhou, C.-H. Hsu, X. Xiao, and F. Yang, « Provision of Data-Intensive Services Through Energy- and QoS-Aware Virtual Machine Placement in National Cloud Data Centers », *IEEE Transactions on Emerging Topics in Computing*, vol. 4, *2*, 2016.

[43] SpiceWorks, *The 2020 State of Virtualization Technology*, Spiceworks, `https://www.spiceworks.com/marketing/reports/state-of-virtualization/`, 2021.

[44] F. Liu, Z. Ma, B. Wang, and W. Lin, « A Virtual Machine Consolidation Algorithm Based on Ant Colony System and Extreme Learning Machine for Cloud Data Center », *IEEE Access*, vol. 8, 2020.

[45] M. Chebiyyam, R. Malviya, S. K. Bose, and S. Sundarrajan, « Server consolidation: Leveraging the benefits of virtualization », *Infosys Research, SETLabs Briefings*, vol. 7, *1*, 2009.

[46] S. Chauhan and S. Vermani, « Cloud Computing to Fog Computing: A Paradigm Shift », *Journal of Applied Computing*, vol. 1, *1*, 2016.

[47] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, « The Case for VM-Based Cloudlets in Mobile Computing », *IEEE Pervasive Computing*, vol. 8, *4*, 2009.

[48] M. Iorga, L. Feldman, R. Barton, M. Martin, N. Goren, and C. Mahmoudi, *Fog Computing Conceptual Model*, Mar. 2018.

[49] A. Yousefpour, C. Fung, T. Nguyen, K. Kadiyala, F. Jalali, A. Niakanlahiji, J. Kong, and J. P.Jue, « All one needs to know about fog computing and related edge computing paradigms: a complete survey », *Journal of Systems Architecture*, vol. 98, 2019.

[50] E. Marín-Tordera, X. Masip-Bruin, J. García-Almiñana, A. Jukan, G.-J. Ren, and J. Zhu, « Do we all really know what a fog node is? Current trends towards an open definition », *Computer Communications*, vol. 109, 2017.

[51] H. Arkian, D. Giouroukis, P. Souza Junior, and G. Pierre, « Potable Water Management with integrated Fog computing and LoRaWAN technologies », *IEEE IoT Newsletter*, 2020.

[52] Tom Banta, « How Mobile Edge Computing Transforms the Telecom Industry », `https://www.vxchnge.com/blog/reasons-mobile-edge-computing-is-important`.

[53] Ronan Mevel, « Edge computing: Redistribute computing power and extend the frontiers of the Cloud », `https://www.orange-business.com/sites/default/files/livre-blanc-edge-computing-etendre-les-frontieres-du-cloud.pdf`.

[54] S. Singh, Y.-C. Chiu, Y.-H. Tsai, and J.-S. Yang, « Mobile Edge Fog Computing in 5G Era: Architecture and Implementation », *in Proc. ICS*, 2016.

[55] Telefonica, *Telefonica Open Access and Edge Computing*, `https://www.telefonica.com/es/wp-content/uploads/sites/4/2021/02/whitepaper-telefonica-opa-mec-feb-2019.pdf`, 2019.

[56] Nearbycomputing, *Satellite Edge Computing*, `https://www.nearbycomputing.com/solutions/satellite-edge/`, 2020.

[57] Google, « Network edge locations », `https://cloud.google.com/vpc/docs/edge-locations`.

[58] K. Dolui and S. K. Datta, « Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing », *in Proc. GIoTS*, 2017.

[59] OculusRift, *Delivers Some Home Truths On Latency*, https://oculusrift-blog.com/john-carmacks-message-of-latency/682/.

[60] J. McChesney, N. Wang, A. Tanwer, E. de Lara, and B. Varghese, « DeFog: Fog Computing Benchmarks », Proc. SEC, 2019.

[61] S. Choy, B. Wong, G. Simon, and C. Rosenberg, « The brewing storm in cloud gaming: A measurement study on cloud to end-user latency », *in Proc. NetGames*, 2012.

[62] OpenFog Consortium, *Transportation Scenario: Smart Cars and Traffic Control (3.1)*, https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf, 2017.

[63] ——, *Autonomous Driving*, http://www.fogguru.eu/tmp/OpenFog-Use-Cases.zip, 2018.

[64] C. Zhu, G. Pastor, Y. Xiao, and A. Ylajaaski, « Vehicular Fog Computing for Video Crowdsourcing: Applications, Feasibility, and Challenges », *in IEEE Communications Magazine*, vol. 56, Oct. 2018.

[65] R. Rajesh and V. Shijimol, « Vehicular Pollution Monitoring and Controlling using Fog Computing and Clustering Algorithm », *in International Journal of New Innovations in Engineering and Technology*, vol. 4, Mar. 2016.

[66] H. A. A. Hamid, S. M. M. Rahman, M. S. Hossain, A. Almogren, and A. Alamri, « A Security Model for Preserving the Privacy of Medical Big Data in a Healthcare Cloud Using a Fog Computing Facility With Pairing-Based Cryptography », *IEEE Access*, vol. 5, 2017.

[67] OpenFog Consortium, *Patient Monitoring*, http://www.fogguru.eu/tmp/OpenFog-Use-Cases.zip, 2018.

[68] Y. Lin and H. Shen, « CloudFog: Leveraging Fog to Extend Cloud Gaming for Thin-Client MMOG with High Quality of Service », *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, *9*, 2017.

[69] G. Ma, Z. Wang, M. Zhang, J. Ye, M. Chen, and W. Zhu, « Understanding performance of edge content caching for mobile video streaming », *IEEE Journal on Selected Areas in Communications*, vol. 35, *5*, 2017.

[70] OpenFog Consortium, *Smart Cities Scenario (3.3)*, https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf, 2017.

[71] ——, *Process Manufacturing – Beverage Industry*, http://www.fogguru.eu/tmp/OpenFog-Use-Cases.zip, 2018.

[72] ——, *Out of the Fog: Use Case Scenarios (High-Scale Drone Package Delivery)*, http://www.fogguru.eu/tmp/OpenFog-Use-Cases.zip, 2018.

[73] ——, *Real-time Subsurface Imaging*, http://www.fogguru.eu/tmp/OpenFog-Use-Cases.zip, 2018.

[74] ——, *Visual Security and Surveillance Scenario (3.2)*, https://www.iiconsortium.org/pdf/OpenFog_Reference_Architecture_2_09_17.pdf, 2017.

[75] D. Battulga, M. Farhadi, M. A. Tamiru, L. Wu, and G. Pierre, « LivingFog: Leveraging fog computing and LoRaWAN technologies for smart marina management (experience paper) », *in Proc. ICIN*, 2022.

[76] L. Ma, S. Yi, and Q. Li, « Efficient Service Handoff across Edge Servers via Docker Container Migration », *in Proc. ACM/IEEE SEC*, 2017.

[77] A. Ahmed, A. Mohan, G. Cooperman, and G. Pierre, « Docker Container Deployment in Distributed Fog Infrastructures with Checkpoint/Restart », *in Proc. IEEE Mobile Cloud*, 2020.

[78] C. Anderson, « Docker [Software engineering] », *IEEE Software*, vol. 32, *3*, 2015.

[79] Docker Documentation, *Docker Volumes*, https://docs.docker.com/storage/volumes/.

[80] J. Darrous, S. Ibrahim, A. C. Zhou, and C. Perez, « Nitro: Network-Aware Virtual Machine Image Management in Geo-Distributed Clouds », Proc. CCGrid, 2018.

[81] Q. Xu, M. Awasthi, K. Malladi, J. Bhimani, J. Yang, M. Annavaram, and M. Hsieh, « Performance Analysis of Containerized Applications on Local and Remote Storage », *in Proc. MSST*, 2017.

[82] O. Rodeh, J. Bacik, and C. Mason, « BTRFS: The Linux B-Tree Filesystem », *ACM Trans. Storage*, vol. 9, 2013.

[83] Docker Documentation, *Docker Commit*, https://docs.docker.com/engine/reference/commandline/commit/.

[84] OpenVZ team, « CRIU », https://criu.org/.

[85] Docker Documentation, *Docker Checkpoints*, https://docs.docker.com/engine/reference/commandline/checkpoint/.

[86] ——, *Docker Swarm mode overview*, https://docs.docker.com/engine/swarm/.

[87] F. Soppelsa and C. Kaewkasi, *Native Docker Clustering with Swarm*. Packt Publishing Ltd, 2016.

[88] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, « Borg, Omega, and Kubernetes », *ACM Queue*, vol. 14, *1*, 2016.

[89] Y. Pan, I. Chen, F. Brasileiro, G. Jayaputera, and R. Sinnott, « A Performance Comparison of Cloud-Based Container Orchestration Tools », *in Proc. ICBK*, 2019.

[90] M. Fogli, T. Kudla, B. Musters, G. Pingen, C. Van den Broek, H. Bastiaansen, N. Suri, and S. Webb, « Performance Evaluation of Kubernetes Distributions (K8s, K3s, KubeEdge) in an Adaptive and Federated Cloud Infrastructure for Disadvantaged Tactical Networks », *in Proc. ICMCIS*, 2021.

[91] The Kubernetes Authors, *Kubernetes Components*, https://kubernetes.io/docs/concepts/overview/components/.

[92] V. Chemitiganti, *Kubernetes Concepts and Architecture*, Platform9 blog, https://platform9.com/blog/kubernetes-enterprise-chapter-2-kubernetes-architecture-concepts/, May 2019.

[93] L. Abdollahi Vayghan, M. Aymen Saied, M. Toeroe, and F. Khendek, « Deploying Microservice Based Applications with Kubernetes: Experiments and Lessons Learned », *in Proc. IEEE CLOUD*, 2018.

[94] The Kubernetes authors, « Kubernetes: Volumes », https://kubernetes.io/docs/concepts/storage/volumes/.

[95] M. R. Desai and H. B. Patel, « Efficient Virtual Machine Migration in Cloud Computing », *in in Proc. International Conference on Communication Systems and Network Technologies*, 2015.

[96] A. Shribman and B. Hudzia, « Pre-Copy and Post-Copy VM Live Migration for Memory Intensive Applications », Proc. Euro-Par, 2012.

[97] P. Svärd, B. Hudzia, J. Tordsson, and E. Elmroth, « Evaluation of Delta Compression Techniques for Efficient Live Migration of Large Virtual Machines », *in Proc. VEE of the 7th ACM SIGPLAN/SIGOPS*, 2011.

[98] C. Puliafito, C. Vallati, E. Mingozzi, G. Merlino, F. Longo, and A. Puliafito, « Container Migration in the Fog: A Performance Evaluation », *Sensors*, vol. 19, 7, 2019.

[99] D. Beserra, E. D. Moreno, P. T. Endo, J. Barreto, D. Sadok, and S. Fernandes, « Performance Analysis of LXC for HPC Environments », *in Proc. CISIS*, 2015.

[100] V. Gite, *How to move/migrate LXD VM to another host on Linux*, `https://bit.ly/3aXIbwB`, Apr. 2021.

[101] R. de Jesus Martins, C. B. Both, J. A. Wickboldt, and L. Z. Granville, « Virtual Network Functions Migration Cost: from Identification to Prediction », *Computer Networks*, vol. 181, 2020.

[102] Y. Qiu, C.-H. Lung, S. Ajila, and P. Srivastava, « LXC Container Migration in Cloudlets under Multipath TCP », *in Proc. COMPSAC*, 2017.

[103] S. Pickartz, N. Eiling, S. Lankes, L. Razik, and A. Monti, « Migrating LinuX Containers Using CRIU », *in Proc. HiPC*, 2016.

[104] Y. Qiu, C.-H. Lung, S. Ajila, and P. Srivastava, « Experimental evaluation of LXC container migration for cloudlets using multipath TCP », vol. 164, 2019.

[105] M. Sindi and J. R. Williams, « Using Container Migration for HPC Workloads Resilience », *in Proc. HPEC*, 2019.

[106] P. Karhula, J. Janak, and H. Schulzrinne, « Checkpointing and Migration of IoT Edge Functions », *in Proc. EdgeSys*, 2019.

[107] A. Barbalace, M. L. Karaoui, W. Wang, T. Xing, P. Olivier, and B. Ravindran, « Edge Computing: The Case for Heterogeneous-ISA Container Migration », *in Proc. ACM VEE*, 2020.

[108] R. S. Venkatesh, T. Smejkal, D. S. Milojicic, and A. Gavrilovska, « Fast In-Memory CRIU for Docker Containers », *in Proc. MEMSYS*, 2019.

[109] M. A. Tamiru, G. Pierre, J. Tordsson, and E. Elmroth, « mck8s: An orchestration platform for geo-distributed multi-cluster environments », *in Proc. IEEE ICCCN*, Jul. 2021.

[110] A. van Kempen, T. Crivat, B. Trubert, D. Roy, and G. Pierre, « MEC-ConPaaS: An experimental single-board based mobile edge cloud », *in Proc. IEEE Mobile Cloud*, Apr. 2017.

[111] K. Govindaraj and A. Artemenko, « Container Live Migration for Latency Critical Industrial Applications on Edge Computing », *in Proc. ETFA*, 2018.

[112] S. R. U. Kakakhel, L. Mukkala, T. Westerlund, and J. Plosila, « Virtualization at the network edge: A technology perspective », *in Proc. FMEC*, 2018.

[113] Datadog, *11 Facts about Real-World Container Use*, `https://www.datadoghq.com/container-report/`, Nov. 2020.

[114] M. van Steen and A. S. Tanenbaum. Distributed Systems, 3rd edition, 2017, ch. 8.3.

[115] J. Ansel, K. Arya, and G. Cooperman, « DMTCP: Transparent checkpointing for cluster computations and the desktop », *in Proc. IEEE PDP*, 2009.

[116] F. Aïssaoui, G. Cooperman, T. Monteil, and S. Tazi, « Smart scene management for IoT-based constrained devices using checkpointing », *in Proc. NCA*, 2016.

[117] A. J. Mashtizadeh, E. Celebi, T. Garfinkel, and M. Cai, « The Design and Evolution of Live Storage Migration in VMware ESX », *in Proc. USENIX ATC*, 2011.

[118] T. Hirofuchi, H. Nakada, H. Ogawa, S. Itoh, and S. Sekiguchi, « A Live Storage Migration Mechanism over Wan and Its Performance Evaluation », *in Proc. 3rd International Workshop on Virtualization Technologies in Distributed Computing*, Barcelona, Spain, 2009.

[119] M. Mishra, A. Das, P. Kulkarni, and A. Sahoo, « Dynamic resource management using virtual machine migrations », *IEEE Communications Magazine*, vol. 50, *9*, 2012.

[120] C. Li, J. Zhang, T. Ma, H. Tang, L. Zhang, and Y. Luo, « Data locality optimization based on data migration and hotspots prediction in geo-distributed cloud environment », *Knowledge-Based Systems*, vol. 165, pp. 321–334, 2019.

[121] N. Tran, M. K. Aguilera, and M. Balakrishnan, « Online Migration for Geo-distributed Storage Systems », *in Proc. USENIX ATC*, 2011.

[122] J. Zhou, J. Fan, J. Jia, B. Cheng, and Z. Liu, « Optimizing cost for geo-distributed storage systems in online social networks », *Journal of Computational Science*, vol. 26, pp. 363–374, 2018.

[123] Y. Wu, C. Wu, B. Li, L. Zhang, Z. Li, and F. C. M. Lau, « Scaling Social Media Applications Into Geo-Distributed Clouds », *IEEE/ACM Transactions on Networking*, vol. 23, *3*, 2015.

[124] E. Saurez, K. Hong, D. Lillethun, U. Ramachandran, and B. Ottenwälder, « Incremental Deployment and Migration of Geo-Distributed Situation Awareness Applications in the Fog », in Proc. ACM DEBS, 2016.

[125] R. Sheldon and P. Crocetti, *Storage and Volume*, `https://www.techtarget.com/searchstorage/definition/volume`, 2021.

[126] J. Schrettenbrunner, « Migrating Pods in Kubernetes », Master's thesis, University of Applied Sciences Hochschule Darmstadt, 2020.

[127] S. Nadgowda, S. Suneja, N. Bila, and C. Isci, « Voyager: Complete Container State Migration », *in Proc. IEEE ICDCS*, 2017.

[128] A. Celesti, F. Tusa, M. Villari, and A. Puliafito, « Improving Virtual Machine Migration in Federated Cloud Environments », *in Proc. INTERNET*, 2010.

[129] F. Zhang, X. Fu, and R. Yahyapour, « LayerMover: Storage Migration of Virtual Machine across Data Centers Based on Three-Layer Image Structure », *in Proc. IEEE MASCOTS*, 2016.

[130] Atomist blog, *Kubernetes Clusters: Pets or Cattle?*, `https://blog.atomist.com/kubernetes-clusters-pets-or-cattle/`, Aug. 2019.

[131] D. Elliott, C. Otero, M. Ridley, and X. Merino, « A Cloud-Agnostic Container Orchestrator for Improving Interoperability », *in Proc. IEEE CLOUD*, 2018.

[132] W. Bao, D. Yuan, Z. Yang, S. Wang, W. Li, B. B. Zhou, and A. Y. Zomaya, « Follow Me Fog: Toward Seamless Handover Timing Schemes in a Fog Computing Environment », *IEEE Communications Magazine*, vol. 55, *11*, 2017.

[133] T. Fernandez, « Continuous Blue-Green Deployments With Kubernetes », `https://semaphoreci.com/blog/continuous-blue-green-deployments-with-kubernetes`.

[134] K. Z. Ibrahim, S. Hofmeyr, C. Iancu, and E. Roman, « Optimized pre-copy live migration for memory intensive applications », *in Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011.

[135] J. Xia, D. Pang, Z. Cai, M. Xu, and G. Hu, « Reasonably Migrating Virtual Machine in NFV-Featured Networks », *in Proc. IEEE CIT*, 2016.

[136] W. Huang, Q. Gao, J. Liu, and D. K. Panda, « High performance virtual machine migration with RDMA over modern interconnects », *in Proc. IEEE Cluster*, 2007.

[137] U. Deshpande and K. Keahey, « Traffic-Sensitive Live Migration of Virtual Machines », *in Proc. IEEE/ACM CCGrid*, 2015.

[138] T. S. Kang, M. Tsugawa, A. Matsunaga, T. Hirofuchi, and J. A. Fortes, « Design and Implementation of Middleware for Cloud Disaster Recovery via Virtual Machine Migration Management », *in Proc. IEEE/ACM UCC*, 2014.

[139] K. Ye, X. Jiang, D. Huang, J. Chen, and B. Wang, « Live Migration of Multiple Virtual Machines with Resource Reservation in Cloud Computing Environments », *in Proc. IEEE CLOUD*, 2011.

[140] M. Sindi and J. R. Williams, « Using Container Migration for HPC Workloads Resilience », *in Proc. IEEE HPEC*, 2019.

[141] « Velero », `https://velero.io/`, 2021.

[142] C. Dupont, R. Giaffreda, and L. Capra, « Edge computing in IoT context: Horizontal and vertical Linux container migration », *in Proc. GIoTS*, 2017.

[143] D. Pizzolli, G. Cossu, D. Santoro, L. Capra, C. Dupont, D. Charalampos, F. D. Pellegrini, F. Antonelli, and S. Cretti, « Cloud4IoT: A Heterogeneous, Distributed and Autonomic Cloud Platform for the IoT », *Proc. IEEE CloudCom*, 2016.

[144] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg, « Live Wide-Area Migration of Virtual Machines Including Local Persistent State », Proc. ACM VEE, 2007.

[145] D. Darsena, G. Gelli, A. Manzalini, F. Melito, and F. Verde, « Live migration of virtual machines among edge networks via WAN links », *in Proc. IEEE Future Network and Mobile Summit*, Jan. 2013, pp. 1–10.

[146] R. Bias, *The History of Pets vs Cattle and How to Use the Analogy Properly*, Cloudscaling, `https://bit.ly/3BaROTm`, Sep. 2016.

[147] A. Fahs, G. Pierre, and E. Elmroth, « Voilà: Tail-Latency-Aware Fog Application Replicas Autoscaler », *in Proc. IEEE MASCOTS*, 2020.

[148] B. Sumner, *City-Wide Augmented Reality Gaming*, `https://gtc.inf.ethz.ch/research/city-wide-ar-gaming.html`, 2016.

[149] G. Rotsaert, *Docker Layers Explained*, `https://dzone.com/articles/docker-layers-explained`, Mar. 2019.

[150] The Kubernetes Authors, *Assigning Pods to Nodes*, https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/, Jun. 2021.

[151] S. Noghabi, L. Cox, S. Agarwal, and G. Ananthanarayanan, « The emerging landscape of edge computing », *GetMobile: Mobile Computing and Communications*, vol. 23, *4*, 2020.

[152] B. Salmon, *Understanding cloud storage models*, InfoWorld, https://bit.ly/2VCLt0V, 2015.

[153] Pivotal, « RabbitMQ – RabbitMQ is the most widely deployed open source message broker. », https://www.rabbitmq.com/.

[154] ——, « RabbitMQ PerfTest », https://rabbitmq.github.io/rabbitmq-perf-test/stable/htmlsingle/.

[155] « HBICT – Hash Based Incremental Checkpointing Tool », http://hbict.sourceforge.net/, 2014.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Migration à état d'applications dans les systèmes géo-distribués

Mot clés : Migration de ressources, Géodistribution, Kubernetes, Conteneurs, Fog Computing, Migration de mémoire, Migration de volume

Résumé : La production massive de données est un phénomène de en plus exploré, notamment pour extraire de la valeur des données en temps réel. Dans ce contexte, il est nécessaire d'assurer la continuité et la qualité des services dans les applications sensibles à la latence. Le Fog Computing offre une communication à faible latence et répond aux exigences de ces applications émergentes. En revanche, il ne peut pas faire face à la mobilité des utilisateurs. La mobilité représente un défi car il est nécessaire de fournir des alternatives pour réduire la distance entre l'application et les utilisateurs. La migration est une excellente occasion de déplacer les applications vers des utilisateurs en déplacement perpétuel. La migration doit se faire de manière transparente pour l'application et les utilisateurs sans compromettre son exécution. Dans cette thèse, nous abordons ce problème avec les contributions suivantes : la première propose MyceDrive, un outil de migration d'applications pour les systèmes géo-distribués, la seconde propose un outil de migration de volumes de conteneurs exploitant OverlayFS et effectuant une migration incrémentale. Nous évaluons les deux contributions dans un environnement fog réaliste et les comparons à l'état de l'art.

Title: Stateful Application Migration in Geo-Distributed Systems

Keywords: Resource Migration, Geo-Distributed, Kubernetes, Containers, Fog Computing, Memory Migration, Volume Migration

Abstract: Mass production of data is increasingly explored, particularly for extracting value from real-time data. With this, there is a need for continuity and quality of services in latency-sensitive applications. Fog computing delivers low-latency communication and attends to the requirements of these emerging applications. However, mobility presents a challenge as it is necessary to provide techniques to reduce the distance between the application and its users. Migration comes as a powerful opportunity to relocate applications to perpetually moving users. Migration should occur transparently to the application and the users without compromising its execution. In this thesis, we address it with the following contributions: the first proposes MyceDrive, a seamless application migration tool for geo-distributed systems; the second proposes a tool for migrating container volumes exploiting OverlayFS and performing incremental migration. We evaluate both contributions in a real fog environment and compare them with state-of-the-art techniques.