Efficient Cloud Application Deployment in Distributed Fog Infrastructures

Arif Ahmed

UNIVERSITE BRETAGNE LOIRE MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITE DE RENNES 1
COMUE **U**NIVERSITE **B**RETAGNE **L**OIRE

Ecole Doctorale N°601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : Informatique

Par

« **Arif AHMED** »

« Efficient Cloud Application Deployment in Distributed Fog Infrastructures »

Thèse présentée et soutenue à RENNES, le 20 janvier 2020
Unité de recherche : IRISA (UMR 6074)
Thèse N° :

Rapporteurs avant soutenance :

Paolo BELLAVISTA, professeur, Université de Bologne (Italie)
Frédéric DESPREZ, directeur de recherche, Inria

Composition du jury :

Président : Paolo BELLAVISTA, professeur, Université de Bologne (Italie)

Examinateurs :

Lydia CHEN, professeure associée, TU Delft (Pays-Bas)
Gene COOPERMAN, professeur, Northeastern University (Etats-U7nis)
Frédéric DESPREZ, directeur de recherche, Inria
Shadi IBRAHIM, chargé de recherche, Inria
Cédric TEDESCHI, maître de conférence (HdR), Université de Rennes 1

Dir. de thèse : Guillaume PIERRE, professeur des universités, Université de Rennes 1

ACKNOWLEDGEMENT

I would like to sincerely express my deepest gratitude to my supervisor, Professor Guillaume Pierre, for his immense support, invaluable guidance, supervision and encouragement throughout this research. His dynamism, vision and motivation have deeply inspired me. He has taught me the methodology to carry out the research and to present the research work as clearly as possible. It was a great privilege and honor to work under his guidance. Thereafter, I am deeply grateful and obligated to Professor Gene Cooperman, Northeastern University, US, for inviting me to go through an internship under his supervision and for guiding me with his invaluable knowledge.

I would also like to thank Christine Morin, Inria, Senior Scientist, and Shadi Ibrahim, Inria, Research Scientist for accepting to be the CSID Committee Members of my thesis. Their thorough evaluation, suggestions and comments with encouraging words has assisted me to improve and complete the research report.

My heartfelt gratitude to all the jury members of the thesis for taking interest on my research work, especially, Professor Paolo Bellavista, University of Bologna, Italy and Frédéric Desprez, Inria, Senior Researcher, for reviewing the thesis and giving invaluable comments, which has helped me to improve the research report.

I thank my fellow and former members of MYRIADS team, who have always supported me and boosted up my confidence anytime. I also want to thank Apoorve Mohan, PhD Student, Northeastern University, US for his support while moving to Northeastern University, US and the technical support I received during my Internship work.

Special thanks to my wife, Firoza Choudhury, for being the pillar of strength, for always believing in me and motivating me with love and support, and my family and friends who have always stood beside me with their endless love and support during my PhD journey.

RESUMÉ

Les architectures de cloud computing sont composées d'un grand nombre de serveurs puissants connectés les uns aux autres et au reste de l'Internet avec liens réseau à grande vitesse. La latence entre un utilisateur final et le centre de données cloud le plus proche se situe dans une plage de 20 à 40 ms sur les réseaux filaires, et jusqu'à 150 ms sur les réseaux mobiles 4G. Bien que cette latence soit acceptable pour de nombreuses applications, elle crée de nombreux défis pour certains types d'applications comme par exemple les applications sensibles à la latence telles que les applications de réalité augmentée. Ces applications exigent une latence de bout en bout, y compris le délai de traitement et de réseau, de moins de 10-20 ms. Un autre exemple de telles applications est l'analyse de données IoT. Le nombre croissant de dispositifs IoT produit chaque jour de grandes quantités de données. Les données collectées sont généralement envoyées au cloud pour analyse ultérieure, ce qui consomme une grande quantité de trafic Internet mondial. Une solution possible pour relever ces défis consiste à héberger les applications à proximité des utilisateurs finaux. Les infrastructures de type Fog computing étendent donc les ressources du cloud (calcul, stockage et réseau) en distribuant largement un grand nombre de nœuds à proximité des utilisateurs finaux. Par conséquence, la capacité de calcul est toujours disponible à proximité des utilisateurs.

Les architectures informatiques de Fog computing sont composées d'un grand nombre de nœuds informatiques dispersés dans une zone géographique telle qu'une ville, une région ou même un pays entier afin de maintenir la proximité avec un grand nombre d'utilisateurs. En conséquence, les ressources Fog sont souvent organisées en un grand nombre de points de présence (PoP), où chaque PoP est composé d'un petit nombre de machines faibles, telles que des nano-ordinateurs connectés les uns aux autres et au reste de l'Internet avec des réseaux hétérogènes. Un utilisateur final accède toujours aux applications depuis le point de présence le plus proche pour maintenir une latence minimale.

Nous prévoyons que les applications Fog seront déployées à plusieurs reprises dans différents PoPs : pour maintenir une latence minimale entre les applications

hébergées dans le Fog et leurs utilisateurs finaux, les applications peuvent avoir besoin de se déplacer fréquemment d'un PoP à un autre. La mobilité humaine est loin d'être aléatoire, et il a été prouvé qu'elle est prévisible malgré des différences importantes entre les habitudes de déplacement individuelles. Les applications Fog telles que l'assistance cognitive portable qui vise à servir un seul utilisateur avec une latence ultra-faible peuvent donc être déployées de manière répétée dans le même PoP que l'utilisateur visite souvent (à la maison, au travail, etc.). Dans un autre exemple, des applications de calcul intensif telles que l'analyse de flux vidéo en direct peuvent avoir besoin de déployer plusieurs instances identiques dans le même PoP afin de passer horizontalement leur capacité de traitement à l'échelle. Dans ces scénarios, le processus de déploiement d'application ne peut pas être considéré comme une opération unique qui n'affecte pas la qualité d'expérience de l'utilisateur final. Au contraire, il devient une partie intégrante du chemin critique vers la fourniture du service attendu par les utilisateurs.

Le déploiement lent d'applications est donc un problème difficile dans les infrastructures Fog. Tout retard dans le déploiement de l'application peut forcer l'utilisateur à attendre que l'application ait été entièrement déployée et soit prête à servir ses utilisateurs. Lorsque l'utilisateur passe d'un PoP à un autre, il peut être nécessaire de redéployer l'application pour maintenir une faible latence d'accès et réduire le trafic réseau longue distance. Dans ces cas, tout retard dans le déploiement de l'application peut interrompre le service en cours d'exécution, conduisant à une dégradation de la qualité d'expérience (QoE) de l'utilisateur. Dans les deux scénarios, un temps de déploiement d'applications minimal est essentiel pour fournir des services cloud transparents aux utilisateurs finaux. Cette thèse vise donc à réduire autant que possible le temps de déploiement des applications des applications Fog.

Nous avons étudié les raisons de la lenteur de déploiement des conteneurs Docker dans des environnements Fog distribués, et identifié trois opportunités susceptibles de réduire le temps de déploiement des conteneurs: (1) améliorer le taux de réussite du cache Docker, ce qui réduit les chances d'avoir à instalelr une nouvelle image; (2) accélérer l'opération d'installation d'une image; et (3) accélérer le processus de démarrage après la création d'un conteneur. Nous avons donc proposé trois solutions différentes pour optimiser le temps de déploiement global des applications. Chaque solution vise à résoudre l'un des problèmes ci-dessus dans le processus de déploiement.

Contribution 1: Améliorer le taux de réussite du cache Docker

La première contribution de la thèse est d'améliorer le temps de déploiement des applications en réduisant la probabilité d'avoir à installer de nouvelles images lors d'une demande de déploiement d'un conteneur. Les serveurs Docker téléchargent une image depuis un registre distant chaque fois qu'ils constatent que le l'image requise n'est pas disponible dans le cache local. Docker stocke toutes les images téléchargées dans son cache local et ne les supprime jamais jusqu'à ce que soit explicitement demandé. C'est une stratégie judicieuse pour les serveurs puissants car la même image de conteneur ne devra pas être téléchargée à nouveau lors d'un futur déploiement du même conteneur. Cependant, ce scénario ne convient pas dans les environnements Fog, car les serveurs Fog ont une capacité de stockage limitée. En conséquence, l'ensemble de travail des images fréquemment utilisées peut dépasser la capacité de stockage totale du serveur. Un autre problème est que les caches Docker des nœuds co-localisés peuvent contenir des copies redondantes des mêmes images.

Nous proposons un nouveau système de partage d'images Docker qui regroupe les caches des serveurs Fog co-localisés grâce à un système de fichiers partagé. Le résultat final est un cache d'image Docker beaucoup plus volumineux qui peut donc partager plus d'images, ce qui réduit la probabilité de déploiement d'un nouvelle image lors du déploiement d'un conteneur. Notre évaluation de ce système basé sur des traces réelles de registres Docker montre que le partage des images Docker peut considérablement améliorer le taux de réussite du cache et, par conséquent, réduire le temps de déploiement des conteneurs entre 37% et 78% selon le scénario.

Contribution 2: Amélioration du déploiement des images Docker

Le partage d'images Docker entre des serveurs co-localisés améliore le taux de réussite du cache d'images Docker et réduit la probabilité que l'image doive être installée lors d'une demande de déploiement d'un conteneur. Cependant, Docker doit encore déployer les images quand leur déploiement est demandé pour la première fois dans un POP, ou lors d'un défaut du cache. Le déploiement d'images Docker peut être très lent, dans l'ordre de plusieurs minutes dans des nœuds Fog aux ressources limitées tels que les nano-ordinateurs Raspberry Pi. Nous avons étudié la raison de cette lenteur de déploiement en analysant la consommation de ressources de Docker lors d'un déploiement. Nous avons constaté que cette lenteur est en grande partie due au

fait que Docker sous-utilise les ressources matérielles disponibles: Docker télécharge d'abord les différentes couches d'image simultanément, ce qui est très gourmand en ressources réseau. Docker lance ensuite un cycle de décompression de l'image, qui est gourmande en ressources processeur. Enfin l'extraction de l'image est gourmande en ressources d'entrées/sorties disque. En d'autres termes, il y a peu ou pas de chevauchement entre l'utilisation des différences ressources matérielles durant le déploiement de l'image.

Nous avons proposé trois optimisations pour améliorer l'utilisation de ressources lors du déploiement d'images: (1) télécharger les couches d'images séquentiellement pour optimiser le temps de téléchargement; (2) décompression multi-thread pour réduire le temps de décompression des couches; et (3) organiser le processus en pipeline d'entrées/sorties pour commencer à décompresser les couches immédiatement après le téléchargement des premiers octets. Docker-pi combine toutes ces solutions et par conséquent parallélise l'utilisation des trois ressources matérielles (réseau, processeur et disque), ce qui permet de réduire le temps de déploiement des images de 25% à 75% dans des Raspberry Pis en fonction de la capacité du réseau et de la taille de l'image.

Contribution 3: éviter la phase de démarrage du conteneur

Après avoir créé un conteneur, Docker lance la phase de démarrage en lançant le processus initial de l'application. Le démarrage se termine lorsque le conteneur est prêt à accepter les requêtes de l'utilisateur final. Cette phase peut avoir un impact significatif dans des environnements Fog lorsque la même image de conteneur est lancée à plusieurs reprises sur plusieurs serveurs d'un point d'accès. La phase de démarrage des conteneurs reste cependant identique à chaque démarrage. On peut donc sauvegarder l'état d'un conteneur après qu'il ait terminé sa phase de démarrage, et redémarrer le conteneur à partir de l'état sauvegardé lors des déploiements suivants.

Nous avons proposé un nouveau concept de déploiement de conteneurs utilisant DMTCP qui permet de déployer un conteneur à partir d'une image sauvegardée d'un conteneur démarré, ce qui permet d'éviter la phase de démarrage du conteneur. Ce système utilise Ceph afin the partager efficacement les images entre plusieurs serveurs Fog d'un même PoP. Notre évaluation montre que cette technique améliore

la durée de la phase de démarrage d'un conteneur jusqu'à 60x selon le type de conteneur. Les surcoûts d'exécution de ce système restent raisonnables.

ABSTRACT

Cloud computing architectures consist of large number of powerful servers connected to each other and to the rest of the Internet with high-speed network links. The latency between a typical end user and the closest cloud data center comes in the range of 20-40 ms over wired networks, and up to 150 ms over 4G mobile networks. Although this latency is acceptable for many applications, it creates many challenges for certain types of applications: for example, latency-sensitive applications like augmented reality games require an end-to-end latency including network and processing delay under 10-20 ms. Another example of such applications is IoT data analysis. The growing number of IoT devices produces large amounts of data every day. The collected data is typically sent to the core-cloud for further analysis. which consumes large amount of global Internet traffic. An obvious solution to address these challenges is to host applications near the end users. Fog computing therefore extends the cloud resources (compute, storage and network) by broadly distributing large numbers of compute nodes near the end users. Therefore, computational capacity is always available in the vicinity of the users.

In contrast, Fog computing architectures consist of large number of computing nodes dispersed across a geographical area such as a city, a region or even a country to maintain proximity with a large number of users. As a consequence, fog resources are often organized in a large number of Point-of-Presence (PoP), where each PoP is composed of a small number of weak machines such as single-board computers connected to each other and to the rest of the Internet using heterogeneous networks. An end user always accesses the applications from the closest PoP to maintain minimal latency.

We expect that fog applications will be repeatedly deployed in different PoPs: to maintain minimum latency between the applications hosted in the fog and their end users, applications may need to roam frequently from one PoP to another. Human mobility remains far from being random, and it has been proven to be predictable despite important differences between individual travel patterns. Fog applications such as wearable cognitive assistance which aims at serving a single user with ultra-low

latency may therefore be repeatedly deployed in the same PoP the user visits often (home, work, etc.). In another example, compute-intensive applications such as live video feed analysis may need to deploy multiple identical instances in the same PoP in order to horizontally scale their processing capacity. In these scenarios, the application deployment process cannot be considered as a one-time operation which does not affect the end-user's quality of experience. Rather, it becomes an integral part of the critical path towards providing the expected service to its end users.

Slow application deployment is therefore a challenging issue in fog infrastructures. Any delay in the application deployment may force the user to wait until the application has been fully deployed and is ready to serve users. When the user moves from one PoP to another, the application may have to be re-deployed to maintain proximity, low latency, and reduce long-distance traffic. In such cases, any delay in the application deployment may interrupt the already-running service, leading to a degradation of the user's Quality-of-Experience (QoE). In both scenarios, a minimal application deployment time is essential to provide seamless cloud services to the end-users. This thesis therefore aims to reduce the application deployment time of fog applications as much as possible.

We studied the reasons behind the slow deployment time of Docker containers in distributed fog infrastructures, and identified three opportunities that are likely to speed up the container deployment time: (1) Improving the hit ratio of the Docker cache, which reduces the chances of having to pull a new image; (2) Speeding up the image pull operation itself; and (3) Speeding up the boot process after a container has been started. We therefore proposed three different solutions to optimize the overall application deployment time. Each solution aims to address one of the above issues within the deployment process.

Contribution 1: Improving the Docker cache hit ratio

The first contribution of the thesis is to improve the application deployment time by reducing the probability of having to deploy new images upon container deployment requests. Docker servers download an image from a registry whenever they find that the required image is missing in the local cache. Docker stores all the downloaded images in its local cache and never removes them until explicitly asked for. This is a sensible strategy in powerful servers as the same container image will not need to be

downloaded again in future deployment of the same container. However, this scenario is not suitable in fog environments, as fog servers have limited storage capacity. As a consequence, working set of images may grow larger than the total storage capacity of the server. Another issue is that the image caches of the co-located nodes may contain redundant copies of the same images.

We proposed a new Docker image sharing framework which aggregates the image caches of co-located fog servers using a distributed file system. The end result is a much larger Docker image cache that can share more images, which reduces the probability of deploying a new image upon a container deployment request. Our performance evaluation of the proposed framework using a real-world Docker registry workload shows that sharing the Docker images can significantly improve the hit ratio and, as a result, reduce container deployment time between 37% and 78% depending on the scenario.

Contribution 2: Improving the Docker image deployment

Sharing Docker images among co-located servers enhances the Docker cache hit ratio and reduces the probability of image pull upon a container deployment request. However, Docker still needs to deploy an image when it is requested for the first time in a PoP or upon a cache miss. Docker image deployment can be very slow, in the order of a couple of minutes in resource-constrained fog nodes such as single-board Raspberry Pi. We investigated the reason behind this slow deployment by analyzing the resource consumption of Docker upon a image deployment. We found that this slow deployment time is largely due to the fact that Docker under-utilizes the available hardware resources during deployment: Docker first downloads the different image layers simultaneously which is very network intensive, followed by a cycle of CPU-intensive decompression and then disk-intensive extraction. In other words, there is little or no overlapping among the usage of different hardware resources during the image deployment.

We proposed three optimizations to improve the resource utilization of Docker during image deployment: (1) Sequentially downloading the image layers to optimize layer download time; (2) Multi-threaded decompression to reduce the decompression time of layers; and (3) I/O pipelining to decompress the layers immediately after the first few bytes have been downloaded. Docker-pi combines all these solutions and therefore

parallelizes the usage of the three hardware resource (network, CPU and disk), resulting in reducing the image deployment time by 25% to 75% in Raspberry Pis depending on the network capacity and the image size.

Contribution 3: Avoiding the container boot phase

After creating a container, Docker starts the boot phase by launching the starting process of the application. Booting terminates when the container is ready to accept end user requests. This phase may have a significant impact in fog environments when the same container image is being repeatedly launched, created, and booted in multiple servers of a PoP. The boot phase of containers however remains the same every time. We can therefore save the state of a container after completing its boot phase and then later restart the container from the saved state in the subsequent deployments.

We proposed a new container deployment design which uses DMTCP to deploy the container from a booted checkpoint image, therefore skipping the container boot phase. The design uses Ceph distributed storage to store container environments and checkpoint images to efficiently share them across fog servers. Our evaluation shows that this technique improves the container boot phase time up to 60x depending on the type of container. The checkpointing overhead of the proposed system remains reasonable.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Cloud computing relies on large numbers of powerful computing nodes connected to each other and to the rest of the Internet with reliable high-capacity networks. The combination of flexibility, scalability and manageable cost of cloud infrastructures dictated the immense popularity of this new computing paradigm. However, cloud resources are concentrated in a small number of data centers, usually far from the end-users they are serving. The latency between an end user and the closest available cloud data center is typically in the range of 20-40 ms over wired networks, and up to 150 ms over 4G mobile networks [44]. Although this is perfectly acceptable for a wide range of useful applications, a number of latency-sensitive applications such as augmented reality games require end-to-end latency including network and processing delay under 10-20 ms [1, 30]. These constraints make it impossible to host such latency-critical application backends in the cloud. In another use case, the growing number of Internet of Things (IoT) devices produces a large volume of sensor data every day [42]. Sending all the collected data to the core cloud using long-distance Wide Area Networks (WAN) for further processing would consume enormous amount of network resources [15]. An obvious solution to address these problems is to place cloud server nodes extremely close to the users, within a couple of network hops. In fog computing, computational nodes are broadly distributed in a large number of geographical locations so computation capacity is always available in immediate proximity of any end user. Fog computing promises to deliver low latency between the end users and their application and to reduce the usage of long-distance networks. Some other examples of using fog computing for different purposes include: privacy and security [159], service management [135], computational offloading [139], service monitoring [126] and content caching [133].

Fog computing architectures are fundamentally different from traditional cloud architecture: to maintain proximity with a large number of users, fog resources must necessarily be dispersed across a large geographical area such as a city, a region or even

an entire country [32]. In contrast, clouds are typically organized with a handful of extremely powerful data centers connected to each other by dedicated ultra-high-speed networks. As a consequence, fog resources are often organized in a large number of Points-of-Presence (PoPs) dispersed across the covered area. Each PoP may be composed not of datacenter-grade servers but rather of a small number of resource-limited nodes such as single-board computers which are connected to each other and with the rest of the Internet using heterogeneous commodity networks [75, 153]. Users usually connect to the closest PoP in order to access the services offered by the fog platform.

Fog applications must often be repeatedly deployed in different fog servers: in particular, to maintain proximity between the applications deployed in the fog and their end users, applications may need to roam frequently from one PoP to another, whereas cloud applications are usually placed in one or more dedicated machines irrespective of users' mobility [28]. The mobility of human beings is far from being random, and it has been shown that despite significant differences between individual travel patterns, user mobility remains remarkably repetitive and predictable [25, 175]. A fog application such as wearable cognitive assistance which aims at serving a single user with ultra-low latency may therefore repeatedly deploy the application in the same server locations the user visits often (home, work, etc.) [74] . In another case, compute-intensive applications such as live video feed analysis may need to deploy multiple identical instances in the same PoP in order to horizontally scale its processing capacity [208]. In these scenarios, the application deployment process cannot be considered as a one-time operation which does not affect the end-user's quality of experience. Rather, it becomes an integral part of the critical path towards providing the expected service to its end users.

Slow application deployment is therefore a challenging issue in fog infrastructures. Any delay in the application deployment may force the user to wait until the application is being fully deployed and ready to serve users. When the user moves, the application may have to be re-deployed in multiple fog PoPs to maintain proximity, low latency, and reduce long-distance traffic. In such cases, any delay in the application deployment may interrupt the already-running service, leading to a degradation of the user Quality-of-Experience (QoE). In both scenarios, a minimal application deployment time is essential to provide seamless cloud services to the end-users. This thesis therefore aims to reduce the application deployment time of fog applications as much as possible. We define application deployment time as:

"The time elapsed after giving the application deployment instruction until the application is ready to serve users."

Docker is by far the most popular application deployment tool in fog environment [7]. It is widely used to deploy containers, either directly or via the use of distributed container orchestration frameworks such as Kubernetes [164]. Docker virtualizes hardware resources such as compute, network, storage resources with the help of special Linux kernel features [91]. The primary reasons for the increasing popularity of Docker containers are their lightweight nature, and the ease of encapsulating, deploying, and running applications. Instead of installing a full operating system inside a virtual machine, all Docker containers in a single host machine share the underlying Linux kernel which makes container images much smaller and faster to deploy compared to virtual machine images [170]. In fog, which is often made of weak machines such as Raspberry Pis, resources have very limited processing, storage and I/O throughput [153]. In such environment, containers are considered the best tool for cloud application deployment that give better performance over traditional virtual machines [123]. We therefore choose to use Docker as our basis for studying the cloud application deployment in fog computing environment.

Figure 1.1 shows a flowchart of the Docker application deployment process. Application deployment starts by giving the container deployment instruction with the application image name and tag and other container configuration. Docker maintains an image cache in the local disk of the server where images and other configurations are stored. Upon receiving the deployment command, Docker first checks whether the image of the application is already present in the local cache. If the image is not cached, Docker triggers the image pull command with the name and tag of the image. The deployment starts by downloading all the layers of the image from a registry server and finally building the image. Once the image is available in the local cache, Docker then creates a container file system on top of the image and creates the container with the given configuration. The application finally needs to start before being able to serve its users. Some applications such as *mysql* require significant amount of time to boot before being ready to server their end-users.

Within this application deployment process, we identify three opportunities that are likely to speed up the application deployment: (a) Improving the hit ratio of the Docker cache, which reduces the chances of having to pull a new image; (b) Speeding up the image pull operation itself; (c) Speeding up the boot process of the container. We

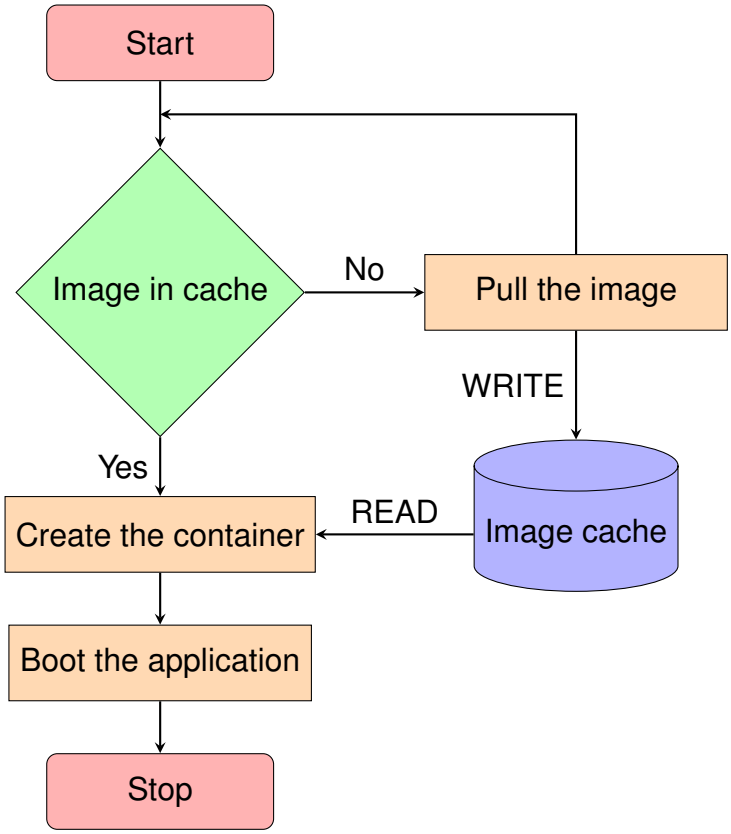


Figure 1.1 – Flowchart of Docker application deployment process.

therefore propose three different solutions to optimize the overall application deployment time. Each solution aims to address one the above issues within the deployment process. We now discuss each proposed contribution of the thesis.

1.1 Contributions

The main contributions of this thesis are as follows:

— *Contribution #1: Improving Docker's cache hit ratio.*
Docker was designed with the assumption it would mostly run in large-scale powerful machines. Docker stores all the running container images in the Docker cache of the local disk to avoid downloading the same image again in case the same container is deployed in the future. Because disk space is not expected to be an issue, Docker never removes the images from the cache unless explicitly asked to do so. This design choice however creates an important storage

issue in fog infrastructures where the servers have limited storage capacity and containers are frequently started and stopped. If the size of the working set of container images is greater than the server's storage capacity then the same image may need to be repeatedly downloaded, utilized and deleted. Another effect of keeping separate Docker cache in each node is that the caches of multiple fog nodes in the same PoP may contain highly redundant content due to some popular images being deployed multiple times in different machines.

We therefore propose a new Docker image sharing framework that allows multiple fog nodes in the same PoP to share the content of their Docker images. Instead of keeping a separate local Docker cache in each node, we aggregate the storage of co-located fog nodes in a PoP. The end result is a much larger cache for each PoP that can store more images, significantly reducing the chances of downloading the images from the long-distance network upon container deployment. Our evaluation based on a real-world Docker registry deployment trace workload shows that sharing the images delivers significant cache hit ratio improvements, leading to a reduction of deployment time between 37% and 78% depending on the scenario.

— *Contribution #2: Speeding up the Docker image pull process.*

Although sharing caches significantly improves the cache hit ratio, Docker still needs to download the images when applications are deployed for the first time in a PoP or upon a cache miss. Docker takes several minutes to deploy an image in a resource-constrained fog server such as single-board Raspberry Pi. This slow deployment however is not only due to the fact that these resource-constrained machines have limited processing, storage, I/O and network throughput. We show that Docker implementation inefficiencies creates unnecessary delay. The standard image deployment process starts by downloading image layers from a registry server in parallel (with a default parallelism degree of 3) and then goes through multiple decompression and disk write cycles to extract the layers sequentially beginning from the first layer. The above process leads to three important issues in the image deployment: (a) Downloading multiple layers in parallel delays the download process of the first layer and therefore, postpones the moment its decompression and extraction phase can start. Therefore, delaying the downloading of the first layer ultimately leads to slowing down the extraction phase; (b) Docker image layers are shipped as

compressed tar files. Upon downloading an image file, Docker decompresses it using single-threaded decompression *gzip* which account only for ∼37% CPU utilization of all the machine's cores. A significant amount of deployment time is spent in decompressing the layers; (c) Each image layer is sequentially downloaded, decompressed and extracted to disk. In other words, there is very little overlapping between the three activities of the different hardware resources (i.e. network, CPU and disk) while deploying the image.

To address the above issues, we propose three optimization solutions which address these issues in the standard Docker image deployment process. We then present Docker-pi which combines these optimizations together. We show that Docker-pi reduces the image deployment time by a factor of up to 4 depending on the size of the image and the available network bandwidth. Docker-pi also reduces the image deployment time by 23–36% in powerful data-center grade servers.

— *Contribution #3: Reducing the container boot time.*

Once an image is available in the local cache, Docker creates a container file system on top of the image and then creates the container itself with the given configuration. The application then needs to boot inside the container before being ready for usage. We found that creating a Docker container takes a negligible amount of time *(less than 1 s)*. However, the application boot process may take significant time for some applications: for example, the popular *mysql* database application takes about 10 s to boot the application before accepting user's commands [142]. The significant boot time leads to slow down the application deployment.

To address the above issue, we propose a container deployment model that uses process checkpoint/restart to launch the application inside the container. The checkpoint/restart allows one to save the state of a running application by storing the process information of the application such as memory pages, open sockets and open files in a file or checkpoint image [129]. The application can then be restarted from the checkpoint image file and continue its execution from there on. We propose to use DMTCP to start the application upon creating the container and checkpoint it after completing the boot phase [10]. The resulting checkpoint image contains a full snapshot of the application after the boot phase. In later deployments DMTCP is instructed to launch the application from the

checkpoint image, and therefore, skip the application boot phase. The evaluation of this proposed container deployment model based on *Edge-sharelatex* [182] has shown that it can reduce the boot time upon deployment which results in the improvement of container boot time by up to 60x times over the standard Docker with reasonable checkpoint overhead.

1.2 Published papers related to the thesis

The following manuscripts are currently published or under review:

Journal articles

1. *Docker-pi : Docker container deployment in Fog Computing Infrastructures*, **Arif Ahmed** and Guillaume Pierre, Inderscience International Journal of Cloud Computing, In press.

Conference papers

1. *Docker Container Deployment in Distributed Fog Infrastructures with Checkpoint/Restart*, **Arif Ahmed**, Apoorve Mohan, Gene Cooperman and Guillaume Pierre, The 8th IEEE International Conference on Mobile Cloud (IEEE Mobile Cloud), Apr 2020, Oxford, UK.
2. *Docker Image Sharing in Distributed Fog Infrastructures*, **Arif Ahmed** and Guillaume Pierre, The 11th IEEE International Conference on Cloud Computing Technology and Science (IEEE CloudCom), Dec 2019, Sydney, Australia.
3. *Docker Container Deployment in Fog Computing Infrastructures*, **Arif Ahmed** and Guillaume Pierre, IEEE International Conference on Edge Computing (IEEE EDGE), Jul 2018, San Francisco, CA, United States.

Posters

1. *Efficient Container Deployment in Edge Computing Platforms* , **Arif Ahmed** and Guillaume Pierre, RESCOM 2017 summer school – Le Croisic, France. Jun 2017.

1.3 Organization of the thesis

This thesis is organized in 7 chapters:

Chapter 2 presents the technical background of the thesis. First, we present an overview of cloud computing and identify some of its limitations. We then define fog computing and explain how it addresses the limitations of cloud computing. Finally, we present an overview of Docker, describe its important components, and how applications are deployed inside Docker containers.

Chapter 3 presents the state-of-the-art of the thesis. We first present the different opportunities to improve deployment process within the Docker container deployment processes. We then describe each of the proposed optimization solutions and also shows how our contributions complement them. Finally, we conclude the chapter by presenting a taxonomy of the state of the art of Docker container deployment optimizations.

Chapter 4 starts by demonstrating the potential benefits of sharing individual Docker cache of fog servers in a PoP. We then identified different issues to build a shareable Docker image cache in fog computing environments. We then show how the proposed Docker image sharing framework addresses each issue. The chapter is concluded with the performance evaluation of the image cache sharing framework with two benchmarks: micro-benchmarks and macro-benchmarks in a fog environment testbed.

Chapter 5 illustrates the experimental study of Docker image deployment and then presents an analysis of resource utilization of Docker while an image is being deployed. We then show the incumbencies in Docker image deployment process which lead to slow image deployment. We then present the three optimization solutions and their performance improvement over the standard Docker. The chapter is concluded with an interesting discussion of Docker-pi in various aspects.

Chapter 6 presents the scope of checkpoint/restart tools in particular DMTCP to improve Docker container deployment. We then identify the different challenges while integrating DMTCP with Docker for container deployment. We present the proposed container deployment design. Finally the chapter is concluded by showing the performance evaluation of the our design with a use case.

Chapter 7 presents the conclusion of the thesis. We briefly remind the importance of application deployment time in distributed fog infrastructures. We then summarize the different contributions of the thesis to improve the application deployment time. Finally, we highlight the number of directions that we may study in the future to further reduce the application deployment time in distributed fog infrastructures.

CHAPTER 2

BACKGROUND

This chapter presents the technical background of the thesis. First, we present an overview of cloud computing and identify some of its limitations. We then define fog computing and explain how it addresses the limitations of cloud computing. Finally, we present an overview of Docker, describe its important components, and how applications are deployed inside Docker containers.

2.1 Cloud computing

Cloud computing is an IT organization paradigm that aims to provide resources (infrastructure, platform, software) on-demand to customers [36, 72]. Traditionally, small and medium-sized enterprises had to own IT infrastructures and hire software developers and system administrators to deploy services, which resulted in large costs of ownership. Cloud computing offers to deliver virtual resources (both hardware and software) that can be accessed from anywhere through Internet at a cost depending on the usage of the resources. With the advent of cloud computing, enterprises no longer have to own their IT infrastructures and other resources to deploy services, and may instead exploit the resources provided by the cloud. Therefore, those enterprises may reduce their infrastructure cost and instead invest solely on application-level innovation [62, 155]. The profit brought by the cloud computing attracts many companies to migrate applications from on-premise to cloud infrastructures. As a result, it is estimated that in 2020, 83% of enterprise workloads will be running in the cloud [45].

2.1.1 Cloud computing architecture

Figure 2.1 presents a general architecture of cloud computing. Cloud computing is based on a centralized architecture and is composed of many elements (servers,

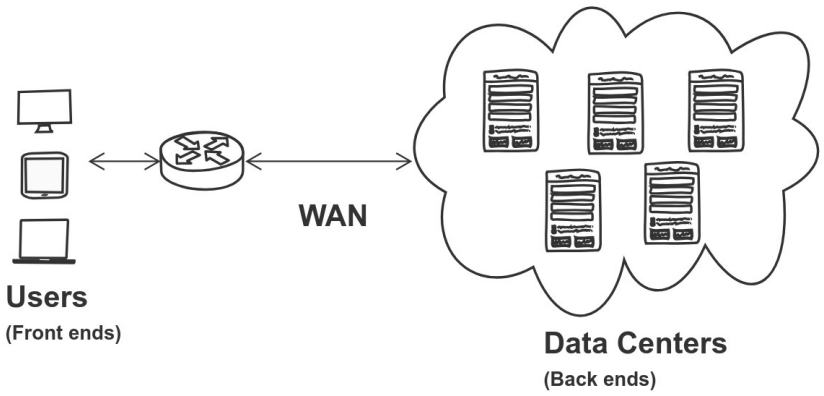


Figure 2.1 – Cloud computing architecture.

switches, firewalls etc.). They are loosely coupled with each other and with the Internet. The architecture has two main parts [178]:

— Front-end: The front-end refers to the client part of services or applications deployed in the cloud platform. It consists of application interfaces such as command lines or Graphical User Interfaces (GUI) that are required to access the cloud computing platforms. The front-end part connects to the cloud services through the Internet.

— Back-end: The back-end parts are mainly composed of resources i.e. compute, storage, network as well as software. They are deployed in a handful of datacenters which are connected to each other and to the rest of the Internet with ultra high-speed backbone networks [67]. Some of the popular cloud service providers are Amazon Web Services [84], Google cloud engine [106], and Microsoft Azure [110].

2.1.2 Limitations of cloud computing

The combination of flexibility, scalibility and manageable cost of cloud infrastructures dictated the immense popularity of this new computing paradigm. However, cloud platforms also exhibit limitations. Cloud computing platforms are consist of a handful of powerful datacenters which are connected with high speed networks. However, the small number of datacenters implies that they are deployed very far away from end-users. Users therefore usually use Wide-Area Networks (WAN) to access the services

deployed in a cloud platform. This architecture leads to important issues which limit the performance of some applications:

a) Latency-sensitive applications such as augmented-reality games require a maximum end-to-end latency in the order of 20 ms (including network and processing delays) [44]. However, the latencies between an end user and their closest data center come in the range of 20-40 ms (in wired networks) and 40-150 ms (in 4G mobile networks) [1, 30]. Such network delays make it impossible to run the server side of latency-sensitive applications in cloud datacenters [38].

b) A growing number of IoT devices produce large volume of sensor data every day [42]. The server side of IoT data analysis applications is usually deployed in a cloud data center in order to process and analyze the collected data. However, sending such enormous amounts of data to the cloud over long-distance WAN consumes large amount of unnecessary resources and energy [15].

2.2 Approaches to address cloud computing limitations

A number of computing paradigms have been proposed in recent years to address the limitations of cloud computing [136]. For examples, *Edge Computing* enables computational capacity at the edge of the network through small data centers that are placed close to end-users (within 1 or 2 hops away from the users) [111]. However, due to the low processing capacity of small datacenters deployed in Edge computing, the total end-to-end latency (including network and processing delay) may end up being actually greater than using simple cloud computing [171]. A closely related paradigm is *Mobile-Edge Computing* which focuses on delivering cloud services with minimum latency by deploying computing resources in mobile phone base stations [64]. However, *mobile edge computing* mainly serves applications which are accessible from mobile clients using cellular network therefore, typical applications are limited to use cases such as content delivery network [66, 172], computational offloading [39], health monitoring [37] etc.

Fog computing was then introduced with the aim of addressing the limitations of both cloud and edge computing. It aims to extend cloud computing datacenters resources by bringing additional compute, storage and networking resources in the close

proximity of its end users [34, 53]. By deploying the server part of applications between the cloud and its end user, fog computing promises to enhance performance of applications that need extremely low latency or that process data locally where it is produced, while retaining large amounts of resources in the cloud for non-critical parts.

2.2.1 Fog computing definitions

Defining fog computing precisely is still an ongoing discussion topic, and many slightly different definitions have been proposed. They however all share a common characteristic: resources are available between the cloud and its end users in order to minimize the end-to-end latency of the applications. We present these proposed definitions and highlight the particularity in each individual definition:

1. In 2012, CISCO proposed the first concept of fog computing [32, 85]. IoT deployment requires mobility support, location awareness, geo-distribution and low latency. The authors argue that fog computing can provide all these requirements by extending datacenters with additional resources located close to end users.

   "Fog computing is a highly virtualized platform that provides compute, storage, and networking services between IoT devices and traditional cloud computing data centers, typically, but not exclusively located at the edge of network."

2. In 2014, Vaquero *et al* defined Fog computing as a computing paradigm that can provide network functionality, service management, with a particular focus on privacy [188]:

   "A large amount of heterogeneous, ubiquitous, and decentralized devices that can cooperate to form a network for storage and processing without third-party intervention."

3. In 2017, the OpenFog Consortium was established to standardize Fog architecture and protocols to support cloud computing services in IoT devices and edge ecosystem [149]. OpenFog emphases the "horizontal" aspect which means fog infrastructure consist of a large number of fog nodes that are distributed across a large geographical location. The same standard was later adopted by IEEE [14].

   "A horizontal, system-level architecture that distributes computing, storage, control and networking functions closer to the users along a cloud-to-thing continuum."

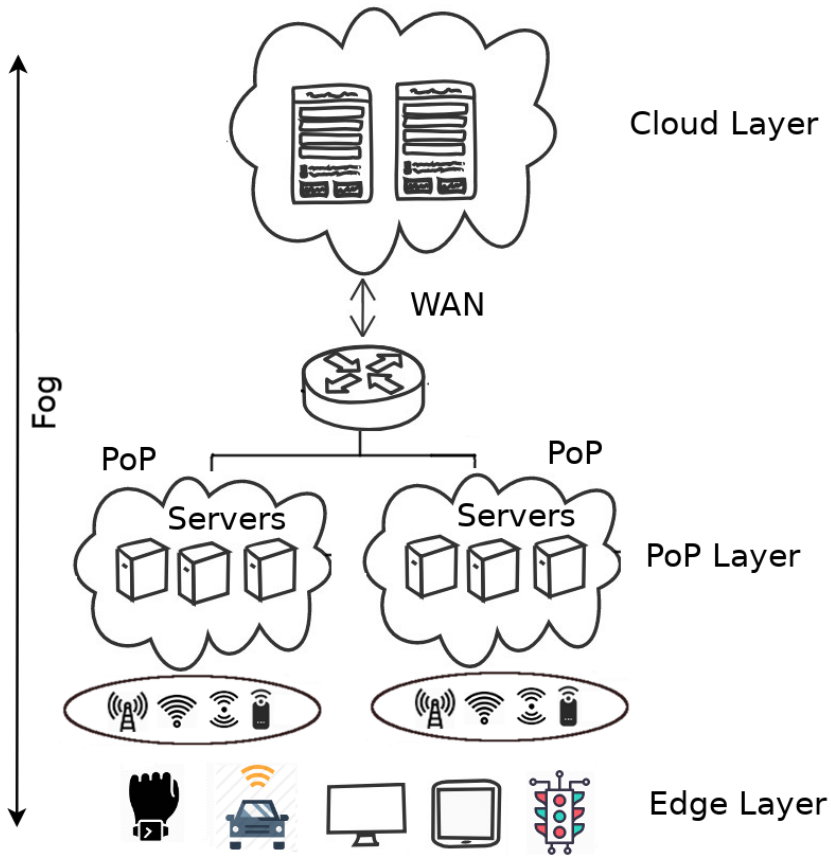


Figure 2.2 – Distributed fog computing architecture.

Since the OpenFog Consortium was established to standardize fog protocols and their definition was accepted by IEEE, we expect this definition will be used in future fog computing research. Therefore, we adopt the same definition given by OpenFog Consortium in this work.

2.2.2 Fog computing architecture

Figure 2.2 presents a distributed fog computing architecture. The architecture consists of 3 layers.

1. ***Edge Layer:*** The bottom layer is the edge layer which is closest to the end users and their physical environment. It mainly consists of IoT sensors, mobile phones, smart vehicles, wearable devices, street cameras etc. The devices belonging to this layer collect data from their surroundings and send them to the upper layers for further processing and storage. Depending on the applications, they

may also receive results back from the fog that allows them to actuate their environments. The edge devices usually use available access network (such as cellular network, WiFi and LoRa) to connect to the upper layers.

2. ***PoP Layer:*** The intermediate layer between end devices and the cloud is the PoP layer. A fog infrastructure aims to bring compute, storage and networking resources in the immediate proximity of its end users. It is therefore composed of widely distributed small groups of servers also known as Points of Presence (PoP) placed in strategic locations such as shopping malls, bus stations, streets, stadiums etc. across a potentially large geographical area. Each PoP contains a small number of devices such as single-board computers [200], drones [141], vehicles [208] etc. with limited compute and storage capacity. These devices can be either *static* or *mobile* [137, 185]. The servers which belong to the same PoP are collocated with each other which implies that they may easily be connected to each other using a fast local-area network. The devices in a PoP layer are equipped with IP networking and thus able to communicate with rest of the Internet and the cloud generally using commodity networks. Fog applications that need compute, storage and network resources close to the end users are deployed in this layer.
3. ***Cloud layer:*** The top most layer is the cloud layer. It mainly consists of powerful datacenters connected to each other and to the rest of the Internet with high-speed networks. It contains powerful computing and storage capacity to support applications which need extensive computational analysis and back end storage, can support being deployed far from the end users.

As the architecture of fog computing is different from cloud computing, we present a comparison of important characteristics of the two computing paradigms in Table 2.1.

2.2.3 Application deployment frameworks in fog computing

Many research efforts have been made for building a highly scalable, flexible, efficient fog application deployment framework that can support cloud-like workloads. Bonomi *et al.* suggested that fog devices should be configured either as virtualized resources as in traditional cloud, or offered as bare metal servers [31]. GigaSight uses virtual machine (VM)-based cloudlets to deploy privacy-aware video analytic applications in three-tier architecture [167]. VM-based virtualization is considered well suited

Table 2.1 – Different characteristics of fog computing and cloud computing.

Characteristics	Fog computing	Cloud computing
Latency from end user to the closest server	Low	High
Distance from users	Close	Far
Architecture	Distributed	Centralized
Processing capacity	Moderate	High
Access networks	LAN, WiFi, Cellular	WAN
Storage capacity	Moderate	Large
Application types	Latency sensitive, IoT analytic	General applications

Table 2.2 – Fog application deployment frameworks

Reference	Platform	Application type	Fog node	Year
Mobile Fog [79]	Not specified	IoT	Not specified	2013
Satyanarayanan *et al* [167]	VM	Analytic	Clusters	2015
LEONORE [190]	Docker	IoT	Clusters	2015
Claus [153, 154]	Docker	General	RPI	2015
Foglets [169]	Docker	General	Clusters	2016
Geelytics [40]	Not specified	Analytic	RPI	2016
MEC-conpaas [123]	LXC	General	RPI	2017
Foggy [166, 205]	Docker	IoT	RPI	2017
Bellavista *et al* [24]	Docker	IoT	RPI	2017
Fogernetes [202]	Docker	General	RPI	2018

in cloudlet environments [168] but it performs poorly in resource-constraint fog devices such as routers, gateways and single-board machines that have significantly low memory, bandwidth and processing capacity [78].

Another way to virtualize fog nodes is using containers [80]. Containers, and particularly Docker containers have important advantages over VMs in fog environments: they are lightweight, portable and easy to deploy and orchestrate in resource-constrained fog nodes. A number of IoT-based application deployment frameworks that rely on Docker containers has been proposed for application deployment in resource-constrained IoT gateways [27, 61, 166, 190, 202]. Kempen *et al* showed that single-board machines have the potential to run real edge cloud applications [23, 123]. Even extremely resource-constrained devices such as Raspberry PIs may be successfully used to build IoT cloud gateways [24]. With proper configuration, these devices can make up scalable fog platforms with minimal overhead.

Table 2.2 presents a comparison of the different proposed fog application deployment frameworks. We observe that the majority of the frameworks, and in particular the most recent ones, are using Docker containers [91] or Kubernetes [19] for application deployment in single-board machines such as Raspberry Pis. This shows the potential of such devices as fog nodes in the near future.

2.3 Docker

Containers are self-contained software packages that encapsulate everything as a software needs to run: executable binaries, libraries, dependencies, settings etc. They are different from software programs mainly because containers are isolated from other software running in the same host machine and underlying operating system [9, 41, 153]. There are different containerization tools available such as Docker [91], LXC [108], OpenVZ [189], and rtk [119]. Among them Docker is certainly the most popular [140]. Docker is portable, operable, lightweight, and its container images are easily shareable [11]. Popular container orchestration tools such as Docker swarm [102], Kubernetes [19], Mesos [109] heavily rely on Docker to create, deploy, and manage the container life cycle [65]. Docker is implemented in Go language [18] and its source code of the project is freely available online [89, 99].

Figure 2.3 shows a host running a set of Docker containers on top of the host operating system. Docker uses special Linux kernel features such as namespace [187] and cgroups [186] to virtualize hardware resources such as compute, network and storage. In the earlier versions of Docker, this was done with the help of LXC containers [108]. Since Docker version 0.9, the libcontainer library [97] is used to integrate low-level Kernel namespace [187] and cgroups [186] features directly [90, 180]. Applications running inside Docker containers are packaged in the form of *images* which contain a part of the container file system with the required libraries, executables, configuration files etc [87]. All the containers running in the same host share the underlying Linux kernel of the host, which makes the size of the Docker images much smaller compared to virtual machine images [170].

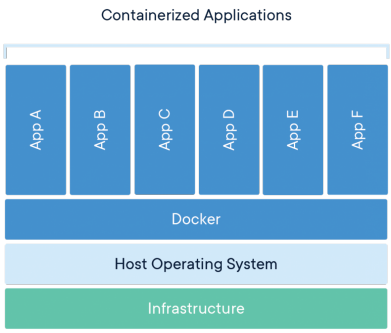


Figure 2.3 – Multiple Docker containers running in a same host machine (adapted from [91]).

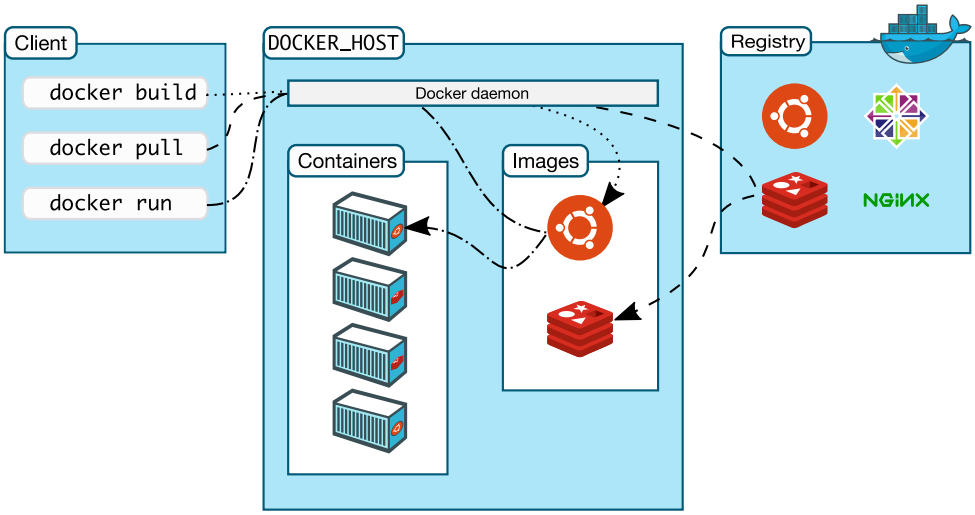


Figure 2.4 – Docker architecture (adapted from [93]).

2.3.1 Docker architecture

Figure 2.4 depicts the Docker architecture. It is composed of three main components: the Docker client, the Docker server and the Docker registry. The architecture utilizes a client-server model and a remote API to create and manage Docker containers [93]. The Docker client and daemon may be deployed on the same host. Alternatively the Docker client can connect to Docker daemon running in a remote machine. The Docker client and the daemon communicate using a REST API, or over UNIX sockets or a network interface [93].

— **Docker server:** The Docker server (also called the Docker daemon) is in charge of the main functionalities of Docker such as creating containers, images, networks, and volumes.

— **Docker client:** The Docker client allows one to interact with Docker servers. The command line is the primary way to interact with Docker server. Upon receiving an instruction, the client sends the request to the selected Docker server using the communication interface. For example, the following command instructs Docker to download, setup and start a containerized web server.

```
docker run nginx:latest
```

A full list of Docker client commands is provided in [104].

— **Docker registry:** A Docker registry server is a repository which stores Docker images [94]. Docker registries can be of two types: *public* or and *private*. A *public registry* is deployed in a secure environment and publicly accessible to upload or download images. For instance, Docker Hub is the most popular public registry. It hosts nearly 2 millions images and is still growing [56]. On the other hand, a *private registry* allows only authorized users to upload or download images.

2.3.2 Docker images

Docker images are consist of multiple layers stacked upon one another: every layer may add, remove, or overwrite files present in the layers below itself. This enables developers to build new images very easily by specializing pre-existing images.

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 layer which stores all file system updates. The container's image layers themselves remain read-only.

A Docker image can be build from a Dockerfile. A Dockerfile is a human-readable file which contains list of instructions to build an image [88]. Docker provides a standard `docker build` command which reads the supplied Dockerfile and creates image layers sequentially starting from the first instruction [81].

Figure 2.5 shows a typical example of a Dockerfile for building a Python-based Docker image. This Dockerfile contains four instructions, and which creates a new layer in the image. The first instruction is the FROM statement which indicates the image is built from a pre-built ubuntu:15.04 image. The COPY command adds files from the

```
FROM ubuntu:15.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

Figure 2.5 – Dockerfile for creating “stream:1.0” Docker image.

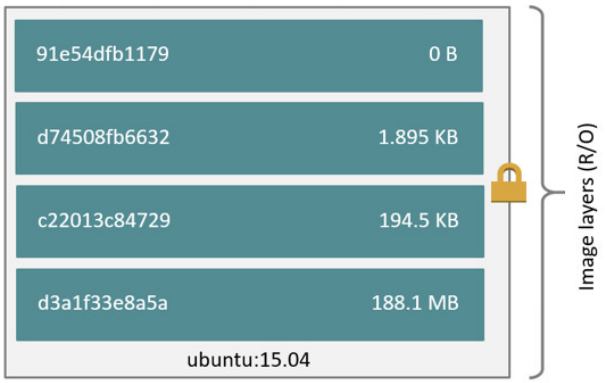


Figure 2.6 – Structure of “stream:1.0” Docker image.

current working directory to the container file system. The third RUN instruction builds the application using the make command. Finally, the last instruction specifies which command to run when a container of the image is deployed. Figure 2.6 shows the resulting image built from the above Dockerfile. A full list of instructions to build a Docker image is given here [96].

Docker encourages layer reusability so it is frequent that different images would share the same bottom-level layers and differ only by their top-level ones [76]. To implement layer reusability Docker applies Copy-On-Write(CoW) strategy while creating the image layers [197]. If a file or directory already exists in a lower layer of an image and another layer needs to read the file or the directory, then Docker simply reads from the lower layer. However, when the file or the directory is modified for the first time, Docker copies the modified file or directory in the top layer.

Docker storage drivers

Docker stores each image layer separately in the local file system. It then exposes a unified view of a set of layers to the running containers, thanks to a storage driver whose main purpose is to handle the different layers (i.e., mutable and immutable lay-

ers) in the container image [181]. A storage driver also handles details about the way different layers interact with each other. Multiple storage drivers are available, including AUFS [193], Overlay and Overlay2 [199], Devicemapper [198], Slacker [76] and btrfs [194].

— **AUFS:** AUFS is a union filesystem [201]. The main principle of a union file system is that it layers multiple directories or branches on a single Linux host and presents them as unified single directory. The branches in AUFS drivers are used to represent different Docker image layers. AUFS storage driver unifies all the layers of the image and exposes them as a single file system. AUFS also implements the Copy-on-Write (CoW) strategy in order to maximize storage efficiency (i.e., re-usability of image layers).

— **OverlayFS:** OverlayFS is a union file system similar to AUFS, but faster and based on a simpler implementation [101]. It layers multiple directories on a single Linux host and presents them as a single directory. OverlayFS refers to the lower read-only directories as *lowerdir* and the upper read-write directory as *upperdir*. The unified view is exposed through its own directory called *merged* which is the containers' mount point.

— **Devicemapper:** Unlike the previous drivers, Devicemapper works at the block level rather than the file system level. It relies on Linux's device-mapper subsystem to create a set of thin-provisioned block devices. Firstly, Docker creates a pool, which typically sits on top of two physical devices— one for user data and one for device-mapper metadata (e.g., block mappings). Secondly, when Docker creates a container, the Devicemapper driver allocates an individual volume for the container from the pool. Devicemapper implements CoW by creating new volumes from writable snapshots of previously created volumes.
  However, Docker containers operate on file systems rather than a provided raw block device. Therefore, a third step is to format the volumes with a configurable file system (either Ext4 or XFS). The big advantage of Devicemapper over the union file systems (AUFS or Overlay2) is that it can perform CoW at a block level granularity (512kB by default) rather than a single file as in union file systems. On the other hand, Devicemapper is completely file system-oblivious and therefore cannot benefit from using any file system information during snapshot creation.

— **Btrfs:** Btrfs [194] is a CoW file system based on B-tree structure [165]. Compared to unifon file systems, Btrfs natively supports CoW and does not require

an underlying file system. Btrfs works with subvolumes which are directory trees, represented in their own B-trees. Subvolumes can be snapshot by adding a new root which points to the children of the existing root.
In terms of storage driver, Btrfs stores the base layer of an image as a separate subvolume and consecutive images are snapshots of their parent layer. Similarly to Devicemapper, Btrfs also performs CoW at the block-level granularity which is more efficient in terms of performance and space utilization compared to file-based CoW. However, Btrfs can experience higher fragmentation due to the finer-grained CoW [181].

Docker originally used *AUFS* by default. However, in recent versions, the use of *Overlay2* is encouraged for performance reasons [103, 193]. The performance of a storage driver depends on the platform and type of application [60, 181]. A set of instructions for selecting an appropriate storage driver is discussed in the Docker documentation [101].

Docker image metadata

Docker organizes the images as a set of layers that are stacked upon one another. It also maintains metadata of the images and layers. The main purpose of the image metadata is to simplify operations on the image layers. Docker image metadata contains three stores:

— **The reference store** contains the manifests of all the images present in the local image cache. Before creating a container, Docker inspects the contents of the reference store to check if the image is already present in the local cache.
— **The layer store** contains all the locally-available layers, identified by their sha256 ID as well as metadata such as the layer's size, parent layer ID etc.
— **The image store** contains image configuration information such as the CPU architecture it relies on, the default exposed ports, attached volumes, etc. It also stores the image history, with a list of layers identified by their with sha256 ID.

Figure 2.7 shows how Docker stores the above information in persistent files inside the Docker storage directory. As these metadata are frequently accessed, Docker keeps a copy in memory to speed-up container operations. When the Docker server is initialized, it copies the metadata to cache in memory and relies on the in-memory version from there on.

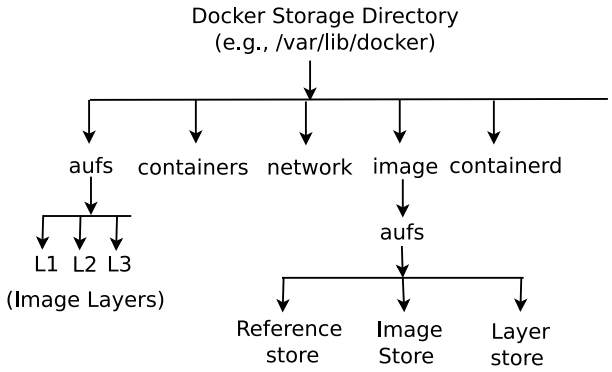


Figure 2.7 – Docker storage directory.

Docker image sharing

The standard mechanism by which Docker supports image sharing between multiple servers is based on a centralized registry where the full set of available images and layers is stored [56]. A Docker registry may be either public (e.g., the public Docker Hub currently contains more than two million images) or private. In essence, a registry supports two main operations: *docker push* and *docker pull*.

— **docker push** uploads a new set of image layers and their manifest file to the registry server;

— **docker pull** downloads a container image from the registry to the docker server where the pull command was issued. The Docker server first downloads the manifest file which contains the list of required layers, then it downloads the missing layers from the same repository. These layers are then kept in the local cache for potential future reuse.

Docker image cache

Docker uses a single directory in the local file system (by default `/var/lib/docker`) to store all cached data such as image layers, metadata and Docker server configurations. Docker uses a "delete-nothing" policy, which means that the cached images are never deleted from the storage directory unless explicitly asked for it [95]. This allows one to avoid redeploying the same image if the application is deployed again in the future. This may however create storage capacity issues in fog environment where the servers have limited storage capacity. Also within a fog computing where each PoP is composed of multiple servers, these cached images may be stored redundantly if

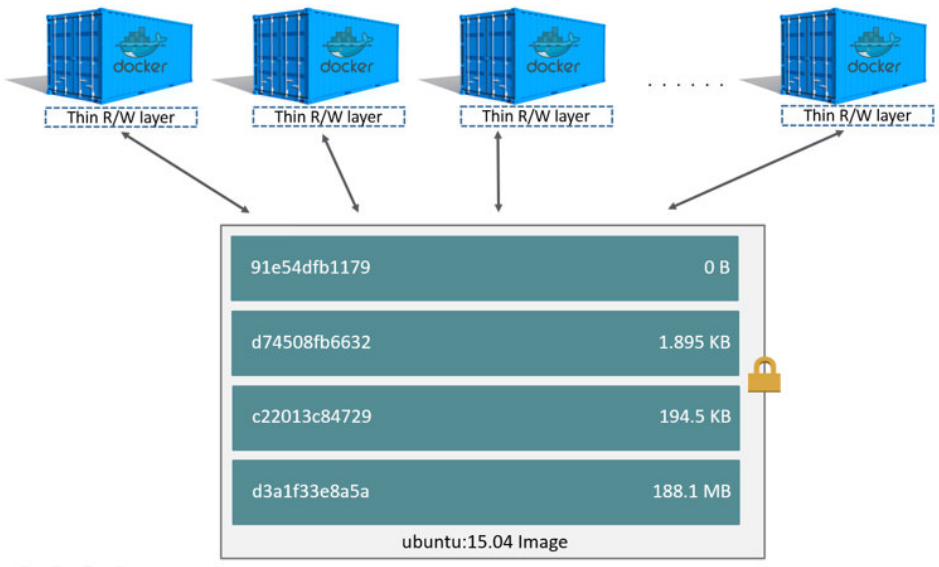


Figure 2.8 – Multiple container images sharing the same underlying layers (adapted from [87]).

the same image has been deployed in multiple servers. This shows that Docker's design for image management was developed for servers that have nearly infinite storage capacity, but it creates strong storage issues in resource-constrained servers.

2.4 Application deployment inside Docker containers

Application deployment in Docker containers starts when the container deployment command `docker run IMAGE:TAG [parameters]` is issued with the name of the image, its version tag, and container configuration. Docker first checks whether the container image is already available in the Docker image cache. This is done by reading the Reference store where all the cached images are listed. If the image is not present in the cache then Docker postpones the container deployment process and triggers on image deployment command `docker pull IMAGE:TAG [parameters]` with the name of the image and its version tag.

Figure 2.9 depicts the flowchart of Docker image deployment in the case of cache miss. Docker needs to deploy the image from a registry server before the actual container deployment operation. The image deployment command involves a series of communications between the Docker server and the registry server. Firstly, Docker lo-

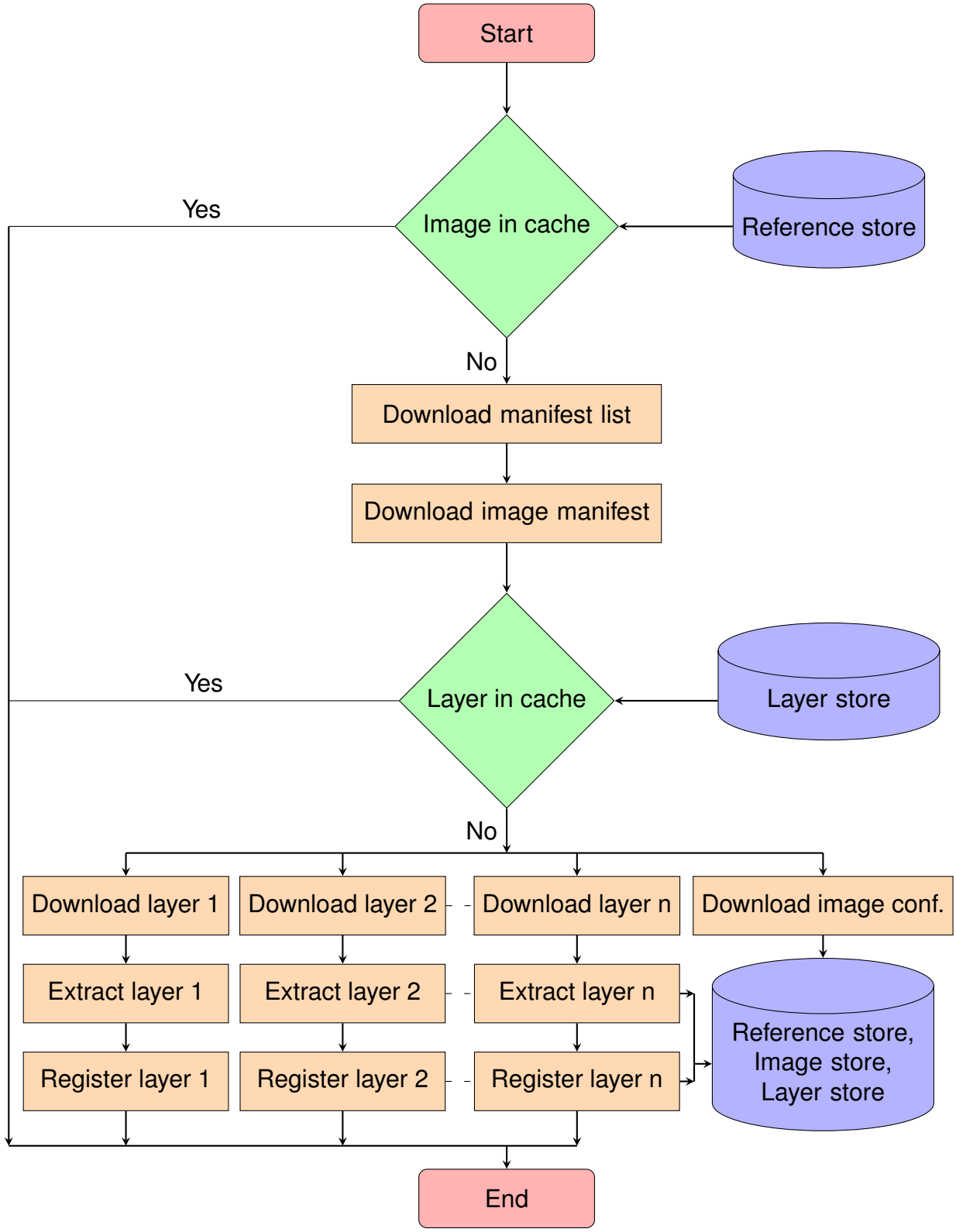


Figure 2.9 – Flowchart of the Docker `docker pull` image deployment process.

cally checks if the image is already present in its cache by reading the content of the Reference store. If the image is not present then Docker connects to the registry and downloads the *manifest list* of the image [98]. This file contains a list of images ID with the requested image name, and tags and platform architectures. The Docker server selects the appropriate image ID and then requests its *image manifest*. This file contains the list of layers present in the image with their corresponding layer ID. The server checks the contents of this file against the Layer store of image metadata to identify missing image layers in the local cache. It then creates multiple threads to download the missing layers and image configuration. The missing layers are downloaded starting from the first layer with a default parallelism degree of 3. Image layers are shipped in the form of a compressed tar file. Every downloaded layer is then separately extracted in the local disk starting from the first layer to preserve the consistency of the file systems. Successfully extracted image layers are then checked for the integrity and registered in the image metadata of the Layer store. Once all the image layers are extracted to disk, Docker writes the image configuration in the Image store and updates the Reference store accordingly.

Once the image is available in the local cache, Docker resumes the container deployment process. It first creates a Read-write container layer on top of the image. The writable top-level layer stores all file system updates performed by the applications after a container has started. The same layering strategy is used to write any file updates on the container layer following Copy-on-Write (CoW) policy. Once the container is created, Docker boots the application by starting the init process of the application. After the application has started, it becomes ready to fulfill its intended usage, and the deployment process terminates.

CHAPTER 3

STATE OF THE ART

This chapter presents the state-of-the-art of this thesis. We present different proposed design approaches to improve the application deployment time in container-based distributed fog environments. The approaches are broadly classified into: (1) speeding up Docker image deployment upon a container creation request; (2) avoiding Docker image deployment; (3) speeding up container creation time; and (4) speeding up the container boot phase.

3.1 Introduction

Within the application deployment process, there are multiple opportunities to improve the end-to-end container deployment time. Many optimizations have been proposed to improve the container deployment time, addressing various limitations in the deployment process. We broadly classify the proposed approaches in the following categories:

1. Speeding up the image deployment: the first phase of application deployment is to download the container image from a registry server. This process takes a significant faction of the application deployment time. Reducing the image deployment time can therefore considerably speed up the end-to-end container deployment.
2. Avoiding the deployment of Docker images: Docker keeps the downloaded images in its local cache for subsequent deployments of the same application. However, in resource-constrained fog servers cache space is limited, and it is fragmented between multiple servers in the same PoP. Improving the hit rate reduces the chances of image deployment upon container creation and therefore reduces the overall average application deployment time.

3. Speeding up the container creation: Docker container creation involves creating a container file system on top of the unified image layers and then isolating the container resources. This process may be slow in scenarios where containers are frequently created, deployed and deleted. Speeding up the container creation reduces the overall application deployment time.
4. Speeding up the container boot phase: The final phase of the deployment is to boot the container by starting the init process of the application. The boot phase may take significant amounts of time depending on the application's set of instructions before it is ready to serve user requests. Reducing the boot phase time or totally eliminating it eventually reduces the overall application deployment time.

3.2 Speeding up the Docker image deployment

Docker container deployment time is largely dictated by the image deployment operation, especially on single-board computers: on average it is estimated that Docker spends nearly 76% of container deployment time in pulling the images from a registry server [76]. Docker image deployment involves a series of communications between the Docker engine in the host machines and the remote registry server. The image deployment time is mainly due to the fact that Docker must download the image layers from the registry server, decompress them and extract them to disk. The performance of image deployment can be improved with efficient design either at the *server side* or at the *client side* [12]. A *server-side* approach involves re-designing the Docker registry server, while a *client-side* approach requires redesigning the Docker engine itself.

3.2.1 Server-side approaches

Distributed registries

A number of efforts have been made to improve Docker image deployment by changing the design of Docker registry [12, 122, 144, 179]. The registry is a centralized, data-intensive component. As the number of stored images in a registry grows and the number of concurrent image deployment requests from multiple clients increases, the Docker registry may become an important performance bottleneck which impacts the

performance of container deployment. To address this issue, CoMICon proposes using a distributed Docker registry instead. It distributes the layers of an image among multiple nodes to increase their availability and reduce the container provisioning time. The distribution allows one to pull an image from multiple registries simultaneously, which reduces the average layer's download times [144]. A similar concept has been used in the "Faster Image Distribution System" for Docker Platform (FID) [122]. FID uses the standard Bit-torrent protocol to distribute image layers in a single data-center setup. Similar approaches that rely on other peer-to-peer protocols have been proposed [21, 134]. Distributed downloading relies on the assumption that multiple powerful servers are interconnected with a high-speed local-area network, and therefore the main performance bottleneck is the long-distance network to a remote centralized repository. However, in the case of fog computing platforms, servers will be geographically distributed to maximize proximity to the end users, and they will rarely be connected to one another using high-capacity networks. This limits the efficiency of distributed downloading in such environments.

In order to address the issue above, Darrous *et al* study the role of image placement across edge servers in situations where the network bandwidths between edge servers are not homogeneous [52]. The authors propose two heuristic algorithms based on the K-center optimization problem that place the image layers on a set of nodes such that the maximum container image deployment time to any server is reduced. The first algorithm is more generic and aims to place the image layers and replicas across the nodes so that the distance from any node to the servers is minimized. The second algorithm avoids placing multiple layers of an image in the same node since this will degrade the advantage of image distribution. Extensive simulations of the proposed algorithms with real-world registry workload shows that the proposed solutions improve the overall image deployment time by 13% to 18% compare to standard Best-fit and random placement techniques. This solution is designed for fog environments where the nodes are *static* and network bandwidth among the servers is relatively stable. On the contrary, in some systems the fog nodes are considered to be *mobile*, and therefore the constantly changing bandwidth among different nodes needs to be taken into account while placing the image layers.

Redesigning individual registries

Another way to improve image deployment time is by improving the performance of individual Docker registry servers. Anwar *et al* studied the workload of Docker registry servers in real-world conditions at IBM where images are frequently deployed to support cloud-like workloads [12]. The analysis revealed that access to the images in the registry is highly skewed: 90% of the pull requests account for only 10% of the images. The second conclusion is that there is a strong correlation between the upload of an image (i.e., push operation) and subsequent download requests (i.e., pull operation) for the manifest files and layers of the image. Based on the above conclusions, the authors propose two registry design approaches. First, two-level caching focuses on the most popular images, it stores the frequently-accessed image layers in fast access storage such as main-memory and SSD. It therefore avoids fetching the image layers from the back-end storage while deploying the images, and thereby significantly reduces deployment time. The second design exploits the relationship between push and subsequent pull requests. When a new image is pushed to the repository, its layers are immediately pre-loaded into main-memory.

3.2.2 Client-side approaches

Image size optimization

The deployment time of Docker images largely depends on the image size and the available network bandwidth. Certainly, a small image can download faster and requires less disk space than a large one. The author of [55] suggests good practices to build small size Docker images. In particular, a small base layer significantly reduces the size of the image. The author also emphasizes the need for reducing the number of image layers which mechanically reduces the overall size of the image as well. In this proposal, the process however remains manual.

DockerSlim [158] is an open-source tool to reduce the size of Docker images. It takes an input image and uses static and dynamic analysis to identify the files in the image layers that are not used by the application. These files can thus be excluded from the container image which significantly reduces original size of the image. The main drawback of Dockerslim is that it also potentially removes important tools such as debugging and profiling tools from the container file system.

While creating a Docker image from a Dockerfile, it is not uncommon that temporary files such as application data are downloaded or generated, utilized and deleted. However, due to the design of UnionFS, Docker creates a new layer for the Dockerfile instructions that delete temporary files which increases the size of the image. Xu *et al* study how to detect such temporary files which are deleted during the image development process but removed logically [131, 204]. The authors propose dynamic analysis of I/O operations which injects code in the Overlay file system kernel module to log temporary file creation and deletion during the image build process. The logs are analyzed and then identified the temporary files which are never removed physically from the image layers. The authors conclude that prior knowledge of such temporary files may reduce the size of Docker images. However, injecting additional code in the kernel to trace I/O operations impacts on performance of the whole system.

Docker engine redesign

Docker image deployment can be improved by optimizing the Docker image build procedure. FastBuild speeds up the Docker image building phase upon a container deployment request [81]. In DevOps environments, container images are frequently changed, built and deployed during the application development and testing phase. Therefore building new images is a frequent activity. FastBuild keeps a record of the frequently accessed files from the Internet during the image build process, and buffers them in the local file system. When building a new image, FastBuild intercepts the requested files to download and supplies them from the buffer. Caching the frequently accessed files avoids remote file access and reduces the image build time. FastBuild targets scenarios where images are frequently modified, built and deployed. In contrast, our goal is to improve the deployment time of production images independently from the way they were built.

Another way to reduce the deployment time is by improving the Docker image deployment process itself. An interesting finding is that Docker needs only about 6% of an image to start a new container [76]. However, upon a container image deployment, Docker has to wait until the whole image has been fully downloaded from the registry before starting the container, which significantly slows down the container deployment. Slacker proposes to rely on an NFS file system to share the images between all the nodes within a datacenter [76]. The proposed storage driver allows lazy pulling of the accessed parts of the container image, which significantly reduces the overall container

deployment time. However, the new storage driver expects that the container image is already present in the local multi-server cluster environment. In contrast, a fog computing environment is made of large numbers of nodes located far from each other, and the limited storage capacity of each node implies that few images can be stored locally for future re-use. Besides, Slacker requires flattening the Docker images in a single layer. This makes it easier to support snapshot and clone operations, but it deviates from the standard Docker philosophy which promotes the layering system as a way to simplify image creation and updates.

Civolani proposes to extends the lazy pulling and redesigns the Docker deployment process so that it can start a container after downloading the image only partially [43]. To implement this, the new model restructures Docker images with a new layer added at the bottom of the image which contains all the required files to start the container. Upon a container deployment request, Docker first downloads and extracts the first layer and creates dummy empty layers for the remaining layers. The container is then started while the remaining image layers are being downloaded and filled asynchronously to the dummy layers. This proposed deployment model however works only with the OverlayFS storage driver. The lazy pulling of the image layers significantly improves the container deployment time, however, creating the dummy layers does not allow one to check the layers' integrity which may leads to undetected image corruption.

In this thesis, we show that Docker does not utilize all the hardware resources of the host machine while deploying the images. Fog nodes such as Raspberry Pis have very limited resources such as processing, storage and network capacity. Any inefficient usage of the available hardware resources therefore has in important impact on performance. We analyze the resource consumption of the host machine while deploying an image and propose optimization solutions to speed up the Docker image deployment. This contribution is presented in Chapter 5.

3.3 Avoiding image deployment

The deployment time of Docker containers can be improved by reducing the probability that an image deployment operation must be carried on upon a container creation. Many solutions have been proposed to avoid redundant deployment of an image in a cluster by sharing Docker images across different servers [35, 57, 76, 206]. For example, the Slacker storage driver was introduced the first to use a shared (NFS) server for

sharing Docker images across multiple servers (see Section 3.2.2). Another solution is to use a distributed file system to share Docker images.

Wharf proposes to share the caches of multiple Docker servers using a distributed file system to reduce storage utilization and the number of redundant image retrievals [206]. It is designed for powerful server clusters where network bandwidth and data storage are cheap. In consequence the authors mostly discuss container startup times and do not address the issues related to limited cache size in fog computing servers. Wharf also relies on the ability of Docker servers to download multiple image layers simultaneously, which was shown to perform poorly in the context of single-board fog servers [4].

Contrary to high-performance datacenters environments, fog servers have very limited storage, compute and networking resources. In this thesis, we study how to increase the Docker image cache hit in fog infrastructures that are made of large numbers of resource-constrained nodes. Our propose Docker image sharing framework addresses the different challenges to share Docker images across multiple fog servers. This contribution is presented in Chapter 4.

3.4 Speeding up container creation

The next way to improve container deployment time is to speed up the container creation time. Oakes *et al* studied the slow container creation time in the OpenLambda [68] serverless framework where Docker containers are frequently created, stopped and deleted [147]. The authors find that the main bottleneck of Docker's slow container creation time is that it uses expensive resource isolation techniques for creating separate cgroups, unifying the image layers with a union file system, and creating a namespace for each container. In order to address these issues, the authors propose the SOCK serverless framework which is based on three optimizations: (1) SOCK relies on a set of lean containers that use lightweight isolation; (2) instead of importing new python packages, SOCK uses Zygot-provisioning to import pre-imported packages at runtime [33]; and (3) three-tier caching creates multiple containers which avoids python packaging and importing costs. The SOCK framework shows how one may reduce the container creation time in a serverless framework where containers are frequently created, deployed and deleted. However, it sacrifices many of Docker's features such as strict container isolation and re-usability of image layers.

Container creation time can be reduced by reducing the necessary I/O operations while loading the container image's file system. Docker reads the essential part of the images from the local disk. However, generic disks are often relatively slow [4]. YOLO (You Only Load Once) uses caching techniques by pre-loading the essential part of the image in main memory or fast disk storage like SSD [145]. To implement this, YOLO creates a subset of the Docker image that contains the essential parts for container deployment. It transparently loads this subset to fast access storage upon the first container request. In subsequent deployments, Docker can then read the image subset from fast-access storage instead of local disk, and thereby, improve container deployment up to 2 times depending on system load conditions. However, this subset image requires on average about 5MB of main memory per container image. The proposed solution therefore is suitable for high performance servers where the main memory capacity is not an issue. However, fog servers are typically limited in memory space. For example, a RPI has in total only 1 GB of main memory. Caching image subsets for all the running containers in the already over-loaded main memory may result in additional swapping which will impact the run-time performance of the running containers.

3.5 Speeding up the container boot phase

The last way to reduce the end-to-end container deployment time is by reducing the time to boot the application. After a container has been created, some applications takes significant time to boot. For example, the popular *Mysql* require about 10 s on a fast server being available to serve end-user commands [142]. In the world of virtual machines, a popular solution to reduce boot time of VMs is by using snapshot and clone [73, 160]. Similarly, with the same goal, process checkpoint and restart with DMTCP has been used to reduce VM boot time [71].

There have not been many studies on the use of process checkpoint/restart in container deployment optimization. Nadgowda *et al* proposed a container auto-scaling method which is based on CRIU [151] to checkpoint a running container and efficiently replicate them in other nodes [142]. However, the proposed solution only aims at horizontal scalability scenarios and does not aim to minimize the container initialization latency in the standalone application deployment.

In this thesis, we use process checkpoint and restart techniques to eliminate the container initialization phase upon an application deployment request. We particularly

rely on DMTCP [10] to checkpoint and restore applications inside Docker container and distribute the checkpoint image with distributed storage. This contribution is presented in Chapter 6.

3.6 Conclusion

Figure 3.1 summarizes the proposed approaches that aim to reduce end-to-end container deployment time. Most of these approaches were proposed in the context of high performance servers. In consequence, they do not consider the performance implication that are specifically present in resource-constrained fog environments. Although the deployment time can be improved either at the server or the registry side, in this thesis, we solely focus on the client side optimizations and consider that the fog resources are limited.

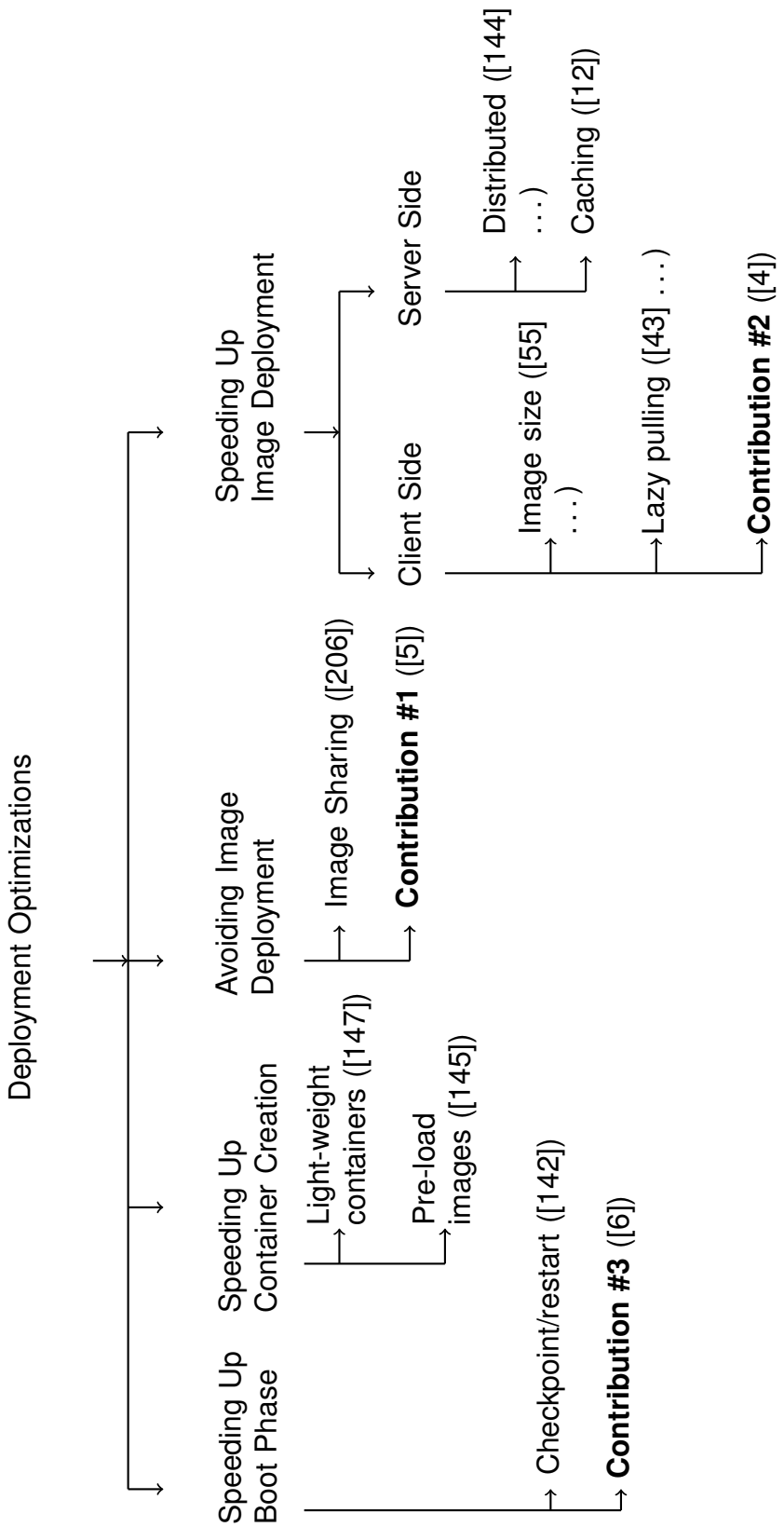


Figure 3.1 – Docker container deployment optimizations.

CHAPTER 4

IMPROVING THE DOCKER CACHE HIT RATIO

This chapter presents the first contribution of the thesis. The first opportunity to reduce container deployment time is by improving the hit ratio of the Docker image cache. The hit ratio of a local image cache present in every fog node may be low due to the fact that the nodes have very limited storage capacity. We propose to aggregate the image caches of multiple co-located fog nodes, so the end result is a large image cache where the images are shared with a distributed file system.

4.1 Introduction

The users of fog applications are usually mobile, which implies that the applications running in the fog may need to be frequently re-deployed in different PoPs to maintain proximity, low latency, and reduce long-distance traffic [29]. However, software deployment in fog infrastructures can be painfully slow when the fog node needs to download a full container image of the deployed application before starting the container itself [4]. Having their fog-hosted application freeze frequently while new containers get started would clearly be a source of frustration for most end users. Reducing the probability of such image cache misses, and the performance impact of their occurrence when they cannot be avoided, is therefore of crucial importance for providing the end users with a satisfactory quality of experience.

Docker, which is by far the most popular container deployment engine [164], was originally designed for powerful server machines. It therefore keeps a copy of every container image in each server's local cache so it does not need to be downloaded again in case the same image is deployed in the future. Docker also never removes content from its caches unless explicitly requested by their user to do so [100]. Although

this strategy makes perfect sense in powerful server machines where disk space is rarely an issue, it creates important storage capacity problems in an environment composed of many weak machines with limited storage space and where containers are frequently started and stopped. If the working set of frequently-deployed images is larger than the storage capacity of a fog computing node, then the same image may need to be repeatedly downloaded, utilized and deleted, creating unnecessary delays and network transfers when re-deploying a container after its image had to be removed from the local node. Another effect of keeping separate image caches in each node is that these caches are likely to contain highly redundant content due to the fact that the same popular images may have been deployed multiple times in different nodes.

We propose to transform these issues in opportunities by allowing multiple fog nodes within a PoP to share the content of their Docker image caches. Instead of using the fog nodes' storage capacity as a set of limited and isolated caches, we propose to aggregate the storage capacity of clusters of co-located fog nodes using a distributed file system. The end result is a single sizable Docker image cache per PoP where large numbers of images can be stored, thereby significantly reducing the probability that images need to be downloaded over a long-distance network upon container deployment.

We analyze a large Docker registry workload and demonstrate the potential for deployment time improvements of Docker image sharing. We survey distributed file systems (which were typically designed for HPC environments) and discuss their suitability in fog computing environments. We present the design of our Docker image sharing framework which supports image cache sharing between multiple co-located fog nodes. Our trace-based evaluations show that sharing caches between multiple fog nodes delivers significant cache hit rate improvements, and leads to reductions of the average container deployment times between 37% and 78% depending on the scenario.

The chapter is organized as follow. Section 4.2 analyses the workload of large docker registries and demonstrates the potential of image sharing in fog infrastructures. Then, Section 4.3 presents our cooperative Docker framework and Section 4.4 evaluates its performance. Finally Section 4.5 concludes.

Table 4.1 – Registries used in the simulations.

Availability Zone (AZ) name	# of pull requests (k)	Total image size (GB)
Frankfurt (fra)	149	86
Sydney (syd)	55	92
London (lon)	349	1719
Dallas (dal)	937	6789
Prestaging (prs)	43	213
Staging (stg)	301	1181
Development (dev)	22	283

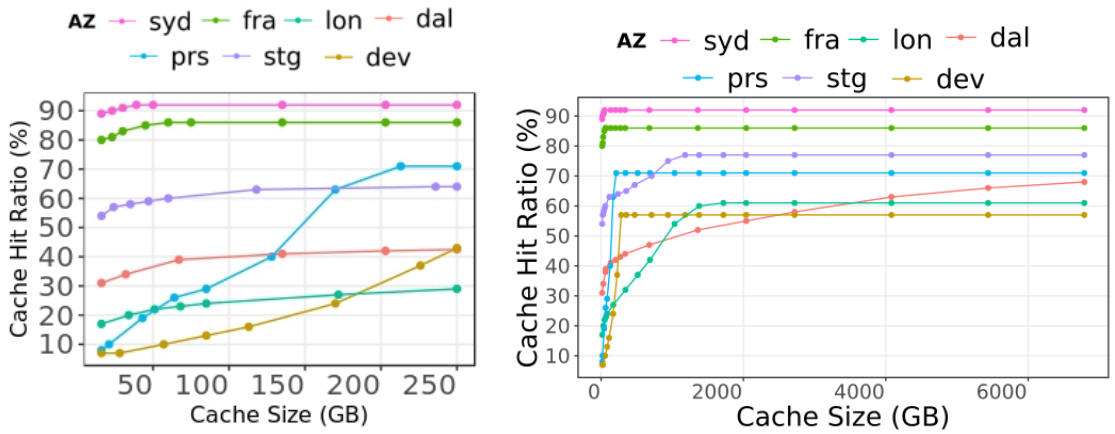
4.2 Potential benefit of cache sharing

To evaluate the potential benefits of sharing Docker caches among fog servers, we analyze the workload of actual Docker registry servers. In the absence of publicly-available fog computing workloads, we instead analyzed a production workload of Docker container deployments in a cloud computing context. As previously discussed, we expect fog computing platforms to experience more frequent re-deployments of the same images than in a normal cloud platform. The results presented here therefore represent a worst-case analysis in terms of cache hit rates.

4.2.1 Simulation setup

We simulate the behavior of Docker image caches under a real registry workload composed of a collection of HTTP-logs generated from 36 IBM Docker registry servers [12]. Each entry in this trace contains the signature of an HTTP request made by a Docker server to the registry for operations such as *docker pull* and *docker push*. The signature provides information about the request such as name of the image, the type of request, and timestamp.

Table 4.1 shows how these registries are classified in 7 Availability Zones (AZ) based on their geographical location and the type of workload they serve. Four AZs (*fra, syd, dal* and *lon*) are dedicated to serve production workloads: *fra* and *syd* are relatively new and have fairly small workloads whereas *dal* and *lon* serve a much larger working set of images. Two AZs (*prs* and *stg*) are used for staging (pre-production) purpose, and finally *dev* is dedicated for development purpose.



(a) Small cache sizes.

(b) Large cache sizes.

Figure 4.1 – Cache hit ratios of different AZs vs. shared cache size.

Note that every request present in this trace corresponds to a cache miss that was incurred at one of the Docker servers, and that consequently triggered a *docker pull* operation. The trace does not contain sufficient information to reliably detect container deployments which resulted in a cache hit in the Docker servers. Due to the fact that Docker never deletes cached image layers unless explicitly requested to do so, we can see this trace as the residual cache miss traffic from a large set of isolated, infinite-sized image caches. Any temporal locality found in this trace therefore highlights an opportunity for cache sharing between multiple servers.

The trace also contains no indication about the containers' lifetime after they are started. However, this lifetime has no influence over the Docker image cache hit rate. In our simulations we therefore assume that containers are stopped immediately after having been started.

To study the benefit of Docker image sharing across fog computing servers, we replay the container deployment logs in a simulator which reproduces the behavior of a Docker image cache: when deploying a new container, the server first checks if the image is available in the (shared or non-shared) cache storage and, if found, immediately starts the container. Otherwise, if downloads the missing layer(s) in the image cache before starting the container.

4.2.2 Cache hit ratio analysis

Figure 4.1 depicts the cache hit ratio that a shared cache for each AZ would have with different storage capacity. Unlike the standard Docker cache management policy, in this study we assume that the shared cache can decide to delete unused cached image layers to make space for new ones that are being requested. In our simulation we use the well-known Least-Recently Used (LRU) policy to decide which layer should be removed when the cache does not have sufficient available capacity to store a newly-requested image.

We can see in Figure 4.1(a) that, even with very small shared cache sizes, several AZs exhibit significant cache hit ratios. This is particularly true for the *syd* and *fra* AZs which respectively exhibit 89% and 80% hit rates with a shared cache size of 32 GB. This is hardly surprising: as shown in Table 4.1 these AZs handle a very small working set of images which can fit even in a very small shared cache.

However, even AZs with a much larger image working set exhibit respectable performance with a very small shared cache. For example *stg* has a hit rate in the order of 55%, and *dal* (the AZ with the largest working set in this trace) around 31%. This indicates that a small number of highly popular container images is repeatedly deployed in different servers from the same zone. For these scenarios, sharing even an extremely small cache delivers significant performance improvement compared with no sharing.

On the other hand, *prs* and *dev* observe almost no cache hits with a very small cache size. This is probably due to the fact that, during development and pre-staging phases, each container image is deployed only a small number of times before either being replaced with an updated version (in case a bug was detected) or being moved to staging or production.

When looking at slightly larger shared cache sizes, we observe that shared cache sizes in the order of a few hundred GBs are usually sufficient to exploit the temporal locality and reach hit rates of 50-95%. Sharing image caches would clearly provide important benefits for these AZs.

Figure 4.1(b) extends these curves until cache sizes of several TBs. We can observe that such cache sizes do not deliver additional benefits compared to much smaller sizes. The only exception is *dal* here every cache size increase (up to the size of its total working set) delivers performance improvement.

We conclude the cache sharing between multiple Docker servers would clearly deliver significant benefits in almost all considered scenarios, even with limited size for

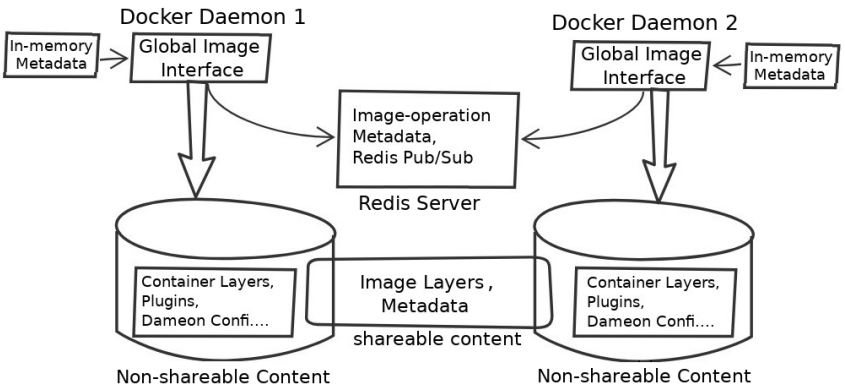


Figure 4.2 – Docker shared image cache architecture.

the shared caches. This is a good news for us considering that in a fog computing platform each PoP would probably have limited storage capacity. The only exceptions where the benefits of sharing caches are limited derive from scenarios where each image is deployed only a couple of times (e.g., during the development phase) or the overall image working set is extremely large.

4.3 System design

The general framework for image cache sharing among a group of Docker servers is depicted in Figure 4.2. In this design, multiple Docker servers use a shared file system to store the (immutable) image manifest and layer files. On the other hand, each Docker server keeps in their local storage the containers' read-write layers, plugins, configuration files etc. However, realizing this design forces us to address a number of difficult challenges.

1. Most distributed file systems designed to aggregate the storage capacity of multiple servers were designed in the context of powerful clusters, and tailored to the needs of high-performance computing applications. However, a fog computing PoP is composed of a small number of relatively weak machines. We therefore need to select a distributed file system which best fits our particular set of constraints. We discuss the choice of a distributed file system in Section 4.3.1.
2. We need to reorganize the directories in which Docker stores its different files such that the immutable image manifests and layers can be mounted from the

distributed file system while the other files remain in a non-shared local storage. We discuss this topic in Section 4.3.2.

3. Docker servers are implemented with the assumption that the image cache is entirely controlled by a single server. Consequently, it maintains parts of its meta-data in memory. When one Docker server modifies the content of the shared image cache, it must therefore notify the other servers to maintain their in-memory metadata consistent with the content of the shared store. We discuss this topic in Section 4.3.3.
4. When multiple Docker servers are requested to deploy the same container image concurrently, we must ensure that each container image is downloaded only once by one of the servers while the others wait for the image to become globally available. We discuss the synchronization of multiple image downloads in Section 4.3.4.
5. In a fog computing PoP we expect that the shared file system will have a relatively limited capacity. We must therefore implement a cache replacement policy so unused image layers can be evicted from the cache when the available storage is not sufficient to hold new layers to be downloaded. We discuss the cache replacement policy in Section 4.3.5.

4.3.1 Choice of distributed file system

Table 4.2 – Comparison of popular distributed file systems.

	HDFS	**CephFS**	**MooseFS**	**GlusterFS**	**iRods**	**Lustre**
Metadata Server	Centralized	**Distributed**	Centralized	**Decentralized**	Centralized	Centralized
Metadata storage	Memory	**Disk**	Memory	**Algorithm**	Disk	Disk
Placement policy	Auto	**User controlled [114]**	No	Random	Admin	No
Striping	Yes	Yes	Yes	Yes	No	Yes
Replication	Yes	Yes	Yes	Yes	Yes	No
Suitable For files	Big	**Big and small**	Big	**Big and small**	Big	Big
Caching	Yes	Yes	Yes	No	Yes	Yes
Memory usage (GB)	8	**1**	20	**1**	2	1
API access	FUSE	**FUSE, ceph, librados**	FUSE	**FUSE**	FUSE	FUSE
POSIX compliance	No	**Yes**	Yes	**Yes**	No	Yes

A distributed file system (DFS) is defined as any file system that allows access to files from multiple hosts sharing via a computer network [196]. We specifically focus here on distributed file systems which aggregate the storage capacity of multiple machines to provide a single unified view of the shared file system to every server within a PoP.

Distributed file systems were mostly designed for high-performance computing environments where the servers are computationally powerful and connected by a high-speed network. In contrast, we aim to use them in severely resource-constrained fog computing environments. Our choice of DFS is therefore largely guided by an analysis of the resource requirements of different DFS.

Distributed file systems typically store file content in blocks that are located in one or several object servers. They therefore need to separately maintain metadata such as the location of these blocks within the distributed file system. This is the task of the Metadata Server (MDS). Upon any file access within the DFS, client machines first need to query metadata before accessing the file content itself.

The design of the MDS is an important differentiating factor between the many available distributed file systems. Depending on the DFS implementations the DFS may be centralized in a single machine or decentralized. In the centralized case, the machine which holds the MDS may incur a significant extra load and potentially become a performance bottleneck. A distributed MDS would share this load among the available servers and arguably exhibit better scalability and fault-tolerance properties.

Distributed file systems also differ in the storage medium used to keep the metadata during its operation. Some file systems load the metadata in memory (which promises fast metadata access) while others keep them on disk. In a resource-limited environment such as a fog computing PoP, memory must be considered as a scarce resource. Although keeping metadata in memory may remain affordable if the number of shared files was small, any container image would contain a large number of (usually small) files, and therefore require significant memory resources to maintain their metadata.

Table 4.2 presents a comparison of six popular file systems: HDFS [13], CephFS [162], MooseFS [49], GlusterFS [115], iRods [47] and Lustre [150] based on information found on the respective file systems' web sites as well as a survey on distributed file systems [54]. Out of all studied file systems, CephFS and GlusterFS stand out because they rely on a distributed metadata server.

CephFS is a fully scalable distributed file system [191]. A CephFS cluster must include one monitor (which maintains a master copy of the cluster map), one manager daemon (in charge of monitoring the file system cluster), at least three Object Storage Devices (OSD) and at least one metadata server (MDS). The monitor and manager are lightweight processes which can easily run in a specific node from a fog computing PoP. the OSDs are in charge of storing all objects from the file system. Finally, the metadata servers share the metadata workload with one another. CephFS splits every file into one or multiple blocks and store them as objects in the OSDs. The client can define so-called CRUSH rules to determine the object placement across the different OSDs. By default ceph replicates every object on two OSDs, which allows the system to survive the crash of any single server within the cluster. Since CephFS stores metadata in disk instead of main memory, it can easily be deployed in resource-constrained compute nodes [113].

GlusterFS is a fully decentralized file system. Unlike the others in this list, it does make use of any metadata server. Instead, it uses an Elastic Hash Algorithm to deterministically choose in which location each file must be stored [105]. GlusterFS can also stripe files in multiple chunks distributed across all the cluster nodes. This allows one to parallelize most I/O operations, and therefore improve the I/O performance when reading or writing large files. Finally, GlusterFS supports RAID1 replication to tolerate node failures. Similar to CephFS, GlusterFS is sufficiently lightweight (expecially thanks to its absence of metadata servers) to be deployed in a fog computing PoP.

We therefore consider CephFS and GlusterFS as the best two contenders for being used in a fog computing scenario. We evaluate and compare their performance experimentally in more details in Section 4.4.1.

4.3.2 Sharing Docker images

Docker uses a single local directory (e.g., `/var/lib/docker/aufs`) to keep all cached data such as image manifests, layer metadata, and the layers themselves. To implement image sharing between multiple docker servers it is important to distinguish the cached content which should be shared from the one which should not.

Shareable content: the shareable content consists of image layers data and the metadata files. To allow multiple Docker servers to access the same layers we mount the distributed file system over the directories which contain these files.

Non-shareable content: some other content such as server-specific configurations and plugins should not be shared. Also, the read-write layers which are dynamically created when starting every new container are meant to be used by a single container, and can therefore be considered as non-shareable content. Although Docker stores these read-write layers in the same directory as the read-only layers, we configured Docker to create the read-write layers in a separate directory out of the mounted distributed file system.

Note that, although sharing the image and layer files across multiple Docker servers is necessary for our approach, it is by no means sufficient. Docker keeps a copy of the cache metadata in memory, and it does not systematically check the consistency of the in-memory data with the persistent ones before using them. We therefore need to design additional mechanisms to maintain these data consistent, as we discuss next.

4.3.3 Consistency maintenance of in-memory metadata

Sharing Docker images through a distributed file system is not sufficient to guarantee the in-memory metadata of the image cache remains consistent with the shared content over time. For instance, when a Docker server executes an image operation such as adding an image in the shared image cache, the updates in the image cache are reflected in the shared file system and the in-memory metadata present in the concerned machine itself, but they are not propagated to the other Docker servers. As a result, in case another Docker server wants to deploy the same image, it will not find it and download it unnecessarily.

To maintain the consistency of the in-memory metadata across all servers within a PoP, we use the popular Redis system and create a publish-subscribe channel to disseminate any update to the in-memory metadata [127, 163]. Whenever one server incurs an update in its in-memory metadata after adding or removing an image, it sends a message (in our case update in-memory metadata) in this dissemination channel. When a server receives this message from the above channel, it discards its in-memory metadata and re-reads the image metadata from the shared file system. With this sim-

ple mechanism, the in-memory metadata remain consistent across all the servers of PoP.

4.3.4 Preventing concurrent deployments of the same image

In a Docker cluster it is not unusual to start multiple instances of the same image simultaneously, for instance in order to aggregate the processing capacity of multiple servers. In our case, if multiple servers from the same PoP attempted to concurrently deploy the same image, they may all notice that the image is not present in cache and redundantly download the same image. It is therefore necessary to allow multiple servers to coordinate with each other and download each image layer only once.

We propose to let a single Docker server download all the required layers. Other servers simply block when they discover the image they want to deploy is being downloaded, and resume the normal deployment process after the image download has completed. To allow each Docker server to reliably detect if an image is already being downloaded by another server we store locks in the Redis key-value store: each image is controlled by a separate lock identified by the sha256 ID of the image.

Figure 4.3 illustrates the updated workflow of the *docker pull* operation. When pulling an image, the Docker server first checks in the Redis database whether a lock for the same image has been created by another server. If the image is not already being downloaded, the server creates a lock in the Redis database under the ID of the image, then pulls the image normally. When the download is completed, it removes the lock and sends a notification to a Redis channel with the same ID. The lock test and set operations are executed within a transaction [128] to ensure atomicity and avoid race conditions.

If a server discovers that the image it needs to pull is already being downloaded by another server, it simply subscribes to the Redis channel (again within a transaction) and waits until it receives a notification that the image download has completed. It can then update its in-memory metadata as discussed in the previous section and terminate the deployment normally.

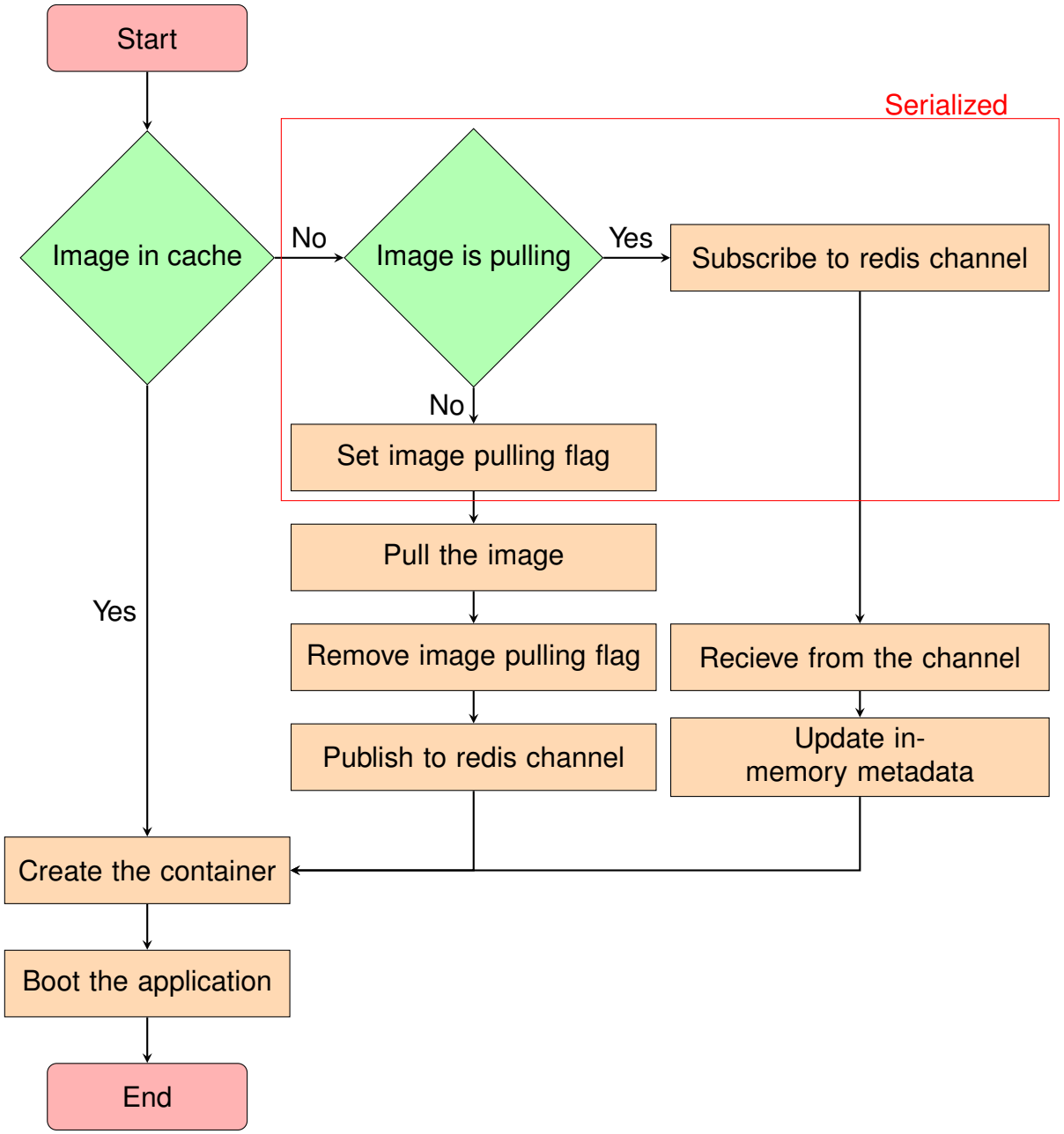


Figure 4.3 – Flowchart of the proposed `docker pull` command.

Algorithm 1: Image replacement algorithm.

```
Input:
S = size of the new image,
{I} = list of unused images,
i = image to be deleted
1 while (available_space < S) do {
2       i = least_recently_used(I)
3       docker rmi i
4       Redis_Publish(Update_Metadata)
5       {I} = {I} - i
6 }
```

4.3.5 Cache replacement

In a resource-limited environment such as a fog computing PoP, it is important to carefully manage resource scarcity. In particular, in a fog computing platform where we expect a large variety of applications to be deployed over time, it is important to ensure that only the most relevant container images are kept in cache, and that the less frequently-deployed ones get discarded to save space. This significantly deviates from the standard Docker policy of never removing any image automatically and of rather relying on human administrators to remove unnecessary images manually [100].

To automatically handle the removal of unused container images, we implemented a cache replacement mechanism within Docker [195]. This mechanism get triggered when the available shared disk space is not sufficient to store a new image which is being downloaded. While deploying an image, if the storage capacity is insufficient then Docker pauses the deployment process and calls the *Image Replacement Interface* to remove one or more unused images.

It would obviously be incorrect to remove an image from the cache while it is being used by any of the PoP's Docker servers. To inform other servers about the images they are currently using, Docker servers register the image name and current number of instances in the Redis database.

Algorithm 1 depicts the replacement mechanism. Whenever the available storage capacity is insufficient to store a new image, the concerned Docker server first builds a list of the currently unused images and the date they were last accessed. We use the popular Least Recently Used policy which evicts the image that was unused for the longest period of time. Once the image has been removed from the shared image

cache, the in-memory metadata of all Docker servers is updated using the same mechanism as discussed in Section 4.3.3. This removal process is repeated until the system has sufficient storage to store the newly downloaded image.

4.4 Evaluation

We evaluate our shared image cache design using a combination of micro- and macro-benchmarks. Micro-benchmarks highlight the performance of a single container deployment using various shared file system configurations, whereas macro-benchmarks show the system's performance under an actual scenario with multiple deployments.

We perform all evaluations on a set of 10 virtual machines representing the nodes of a fog computing point-of-presence. These VMs are created using KVM on a Dell PowerEdge R430 server equipped with two Intel Xeon E5-2620 v4 processors running at 2.10GHz, with 8 hyperthreaded cores each, and 64 GB of RAM. Each VM is configured with 2 vCPUs, 1GB RAM and 32 GB disk and runs Ubuntu 18.04 server with Linux kernel 4.15.0-47-generic. We based ourselves on Docker-pi 18.04, which already contains a number of optimizations designed for Fog computing infrastructures [4]. To avoid interferences from the long-distance network capacity or the Docker hub server, we deployed a private Docker registry in a separate VM in the testbed.

4.4.1 Micro-benchmarks

We first evaluate the performance of our system with different distributed file system configurations. Table 4.3 depicts the three experimental scenarios used in this study. Scenarios 1 and 2 rely on CephFS (Ceph version 13.2.4) with either its user-level FUSE client [124] of the kernel-based one. Conversely, Scenario 3 relies on Gluster (Glusterfs 4.0.2) with its native FUSE client (GlusterFS does not provide a kernel-level client).

All configurations are using three nodes and a replication degree of 1, in order to maximize file system write performance while downloading a new image.

Table 4.3 – Distributed file system configurations.

Scenario	File system	Client	Number of nodes	Replication degree
1	CephFS	Kernel	3	1
2	CephFS	FUSE	3	1
3	GlusterFS	FUSE	3	1

Table 4.4 – Deployment times of an *ubuntu:latest* container.

	File system configuration			
	No cache sharing	**Ceph kernel**	**Ceph FUSE**	**Gluster FUSE**
Cache miss	5.2 s	7.01 s	27.01 s	32.1 s
Cache hit	0.99 s	1.23 s	2.43 s	2.54 s

Deployment time

In this experiment, we deploy the popular *Ubuntu:latest* image using either regular Docker with no shared image cache, or one of the three shared cache scenarios listed in Table 4.3. All machines are kept otherwise idle while deploying the image.

Table 4.4 compares the container's deployment time with shared and non-shared storage, measured from the time the *docker run* command is issued to the moment the container has started. In the case of a cache miss we observe that the configuration with no cache sharing requires 5.2 s to deploy the image, whereas in the distributed file system cases deployment times range from 7 s to 32 s. It is not surprising that shared file system scenarios are slower than regular Docker, as writing the image on a distributed file system creates additional tasks for the Docker server compared to simply writing it on the local drive. We however notice large performance variations depending on the client being used to access the distributed file system: although the kernel-based Ceph driver delivers similar performance to a native local drive, the FUSE-based clients suffer from considerable overhead.

In the case of a cache hit, results are similar although the difference between kernel-based and FUSE clients is less important. This is probably due to the fact that it is not necessary to read the entire image content to start a container so the impact of file

system performance is lower compared to the other operations that must be conveyed upon container creation.

Resource utilization

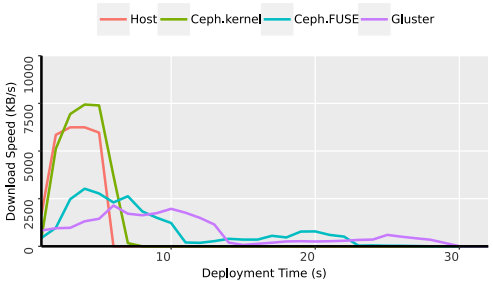
To better understand the relative performance of different distributed file system configurations we monitored the utilization of critical resources while containers are being deployed. We instrumented the testbed machines to trace the overall deployment time as well as the node's resource consumption:

1. Memory usage: the memory consumption of the client machine is monitored using the standard *free -m* Linux command.
2. CPU usage: the CPU consumption (in %) is traced using the *top* Linux command.
3. Network throughput: we measure the upload and download network throughput using the *nethogs* utility [63].

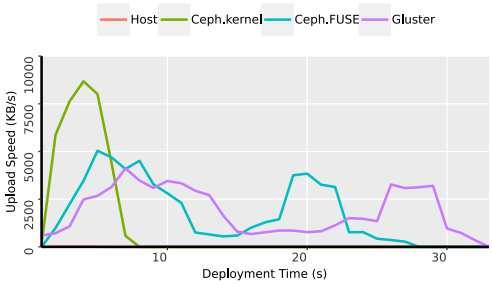
Figures 4.4(a) and 4.4(b) respectively depict the download and upload bandwidth of the Docker server machine while the container image is being deployed upon a cache miss. For obvious reasons the native Docker server does not upload any significant amount of data during the image pull operation, whereas the distributed file systems scenarios see both download (from the image registry to the Docker server) and upload (from the Docker server to the other nodes which participate in the distributed file system). We can also see that the Ceph+kernel configuration can upload data to the distributed file server at a similar rate as the image is being fetched from the registry, whereas the FUSE-based Ceph and Gluster configurations achieve a much lower transfer rate. This is probably due to the fact that FUSE works in user space, which generate large numbers of context switches upon any I/O operation.

Figures 4.4(c) presents the CPU utilization and memory footprint of the Docker server upon a cache miss. The CPU utilization is very comparable in all cases, except that it logically returns sooner to very low values when the container deployment operation is quick (native Docker and Ceph+kernel) compared to the FUSE-based configurations.

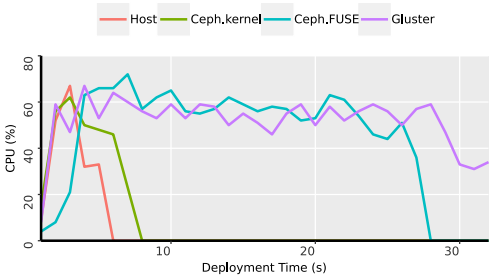
Finally, the memory consumption during a cache miss is shown in Figure 4.4(d). We can see that the distributed file systems impose an additional memory overhead compared to the native Docker. Ceph requires more memory than Gluster, which can



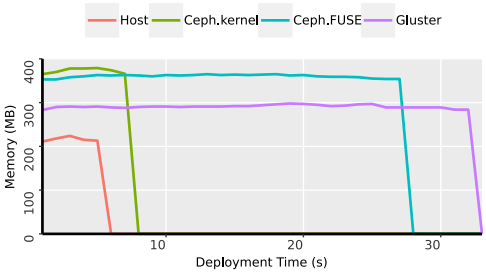
(a) Download speed (kB/s).



(b) Upload speed (kB/s).

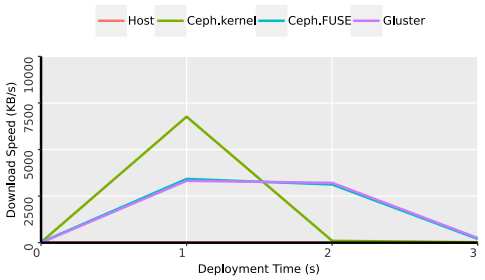


(c) CPU usage (%).

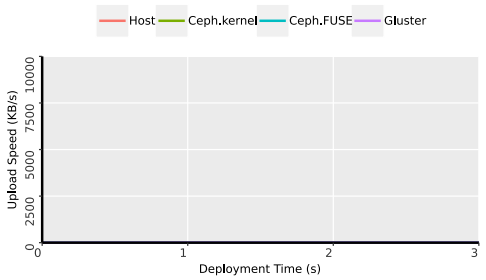


(d) Memory usage (MB).

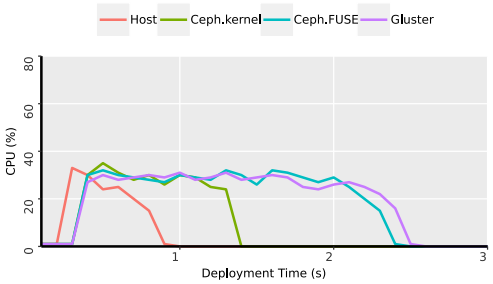
Figure 4.4 – Resource utilization upon a cache miss.



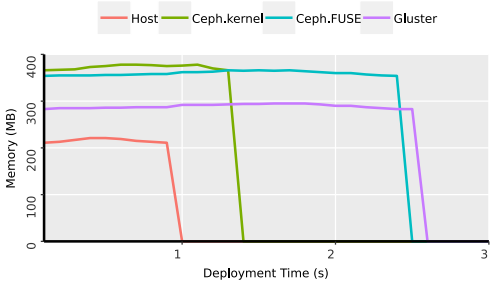
(a) Download speed (kB/s).



(b) Upload speed (kB/s).



(c) CPU usage (%).



(d) Memory usage (MB).

Figure 4.5 – Resource utilization upon a cache hit.

possibly be explained by the fact that Gluster does not need any metadata servers. Interestingly, Ceph's memory footprint decreases to very low values after the container deployment is finished whereas Gluster keeps the same memory footprint during and after the deployment.

Figure 4.5 shows the resource utilization of the Docker server while a container is being deployed from an already cached image. The upload speed is negligible because no content needs to be written to disk. We however observe some download traffic which corresponds to the read operation from the distributed file system. We observe the same phenomenon as in the cache miss scenario, where Ceph+kernel is the only configuration capable of reaching significant throughput in this operation. Here as well, the memory footprint of Ceph is greater than that of Gluster, but it remains only during the container deployment operation.

We conclude that the Ceph+kernel configuration is the only one which can deliver deployment performance similar to that of the native Docker, both in the *cache hit* and *cache miss* scenarios. It requires slightly more memory than Gluster but only during the deployment operation. Based on these findings, in the next sections we focus on the CephFS+kernel configuration only.

4.4.2 Simultaneous Application Deployment

In any cloud-like system, it is frequent that multiple identical VMs or containers get deployed at the same time, for instance to execute a horizontally-scalable application over a significant number of resources. Without cache sharing, each physical server involved in this operation simply deploys a subset of these containers independently from the other servers. When sharing the Docker image cache, all servers read the same image content in the shared file system at the same time. To evaluate whether this operation may constitute a performance bottleneck, we deployed multiple instances of the same *ubuntu:latest* container image in a PoP composed of 10 machines. The image was previously downloaded in the cache, so all deployments result in a cache hit.

Figure 4.6 depicts the average deployment time when varying the number of containers being simultaneously deployed. When deploying a single container the deployment time is 1.2 s. This deployment time grows until 1.5 s when deploying 10 containers simultaneously (one on each PoP node). The overhead of simultaneous container de-

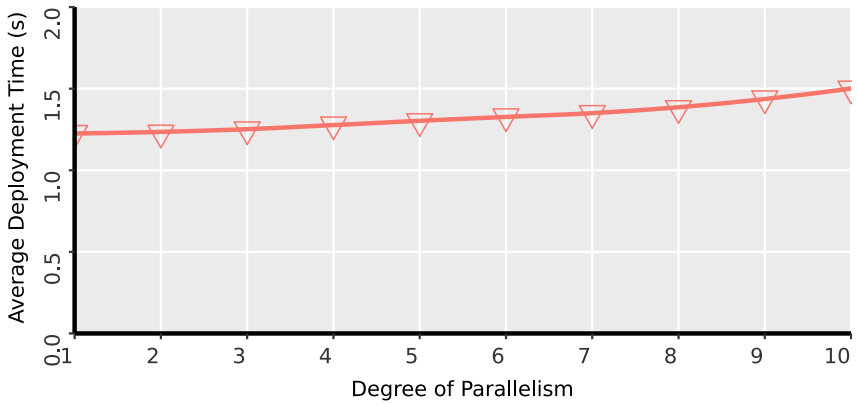


Figure 4.6 – Overhead of simultaneous container deployments.

ployments is therefore real, but sufficiently limited to retain all the other benefits of shared image caches.

4.4.3 Macro-benchmarks

The purpose of sharing Docker image caches is to allow multiple resource-limited PoP servers to increase their cache hit rate by gaining access to a large image cache with a good probability that an image is already present at the time it must be deployed. We therefore evaluate the respective performance of non-shared and shared caches under the same container deployment workload as discussed in Section 4.2.

We created a PoP composed of 5 machines with the same configuration as in the previous sections. When using the shared cache configuration, every machine from the PoP dedicates 10 GB of its disk space to the Ceph distributed file system, and keeps the rest for its local usage. Ceph reserves 1.5 GB space of each disk to store the underlying file system journal, so the total shared storage capacity is 43 GB.

We replayed two traces of container deployments:

— The *fra* availability zone has a total working set of 31 GB. This means it is too large to fit in a single PoP node's local cache but it can entirely fit in the shared cache whose aggregate capacity is 43 GB.

— The *dal* availability zone has a total working set of 54 GB. In this case, even the shared image cache is too small to store all downloaded images. It therefore applies the cache replacement mechanisms described in Section 4.3.5 to keep only the most recently used images. As discussed in Section 4.2.2, this workload

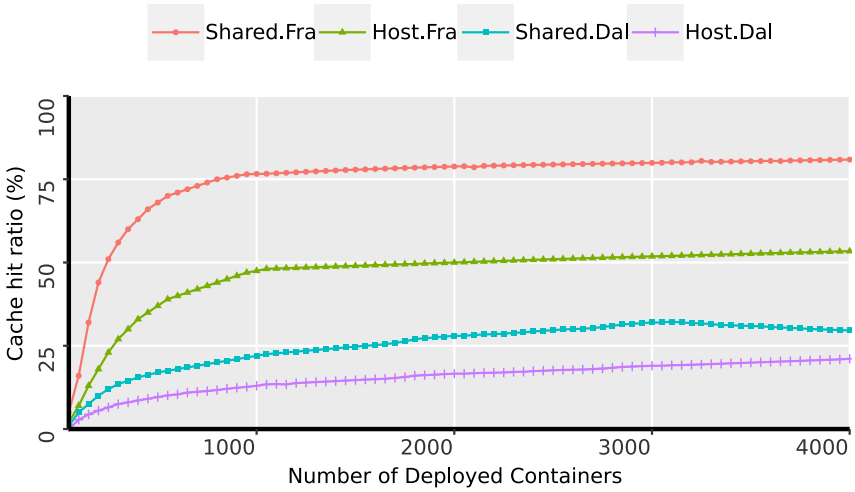


Figure 4.7 – Hit ratio of shared vs. non-shared image caches under a workload of 4000 container deployments.

has limited temporal locality properties and is expected to deliver modest cache hit rates.

Image deployments are issued to the different nodes of the PoP following a round-robin policy. Since we are only interested in the container deployment times, we stop every container immediately after the end of its deployment.

Figure 4.7 depicts the evolution of the cache hit ratio during the execution of the two traces, with and without shared cache, binned by groups of 50 consecutive deployments. For both workloads, the shared cache clearly delivers a much greater cache hit rate than the non-shared caches. More precisely, during the first few hundred deployments, the shared cache hit rate grows much faster that the non-shared caches. This can be explained by the fact that the most popular images need to be downloaded only once in the case of a shared cache whereas in the non-shared case the same image must be downloaded separately by multiple fog nodes. In the end of the curve we observe the effect of a increasing the cache size available to any of the Docker servers: the cache hit rate of the *fra* zone stabilizes around 82% using the shared cache whereas the non-shared caches deliver only 52% hit rate. In the case of the *dal* zone the cache hit rates are more modest (as expected from the study in Section 4.2.2) but there as well the shared cache delivers a significant cache hit rate improvement compared to non-shared caches.

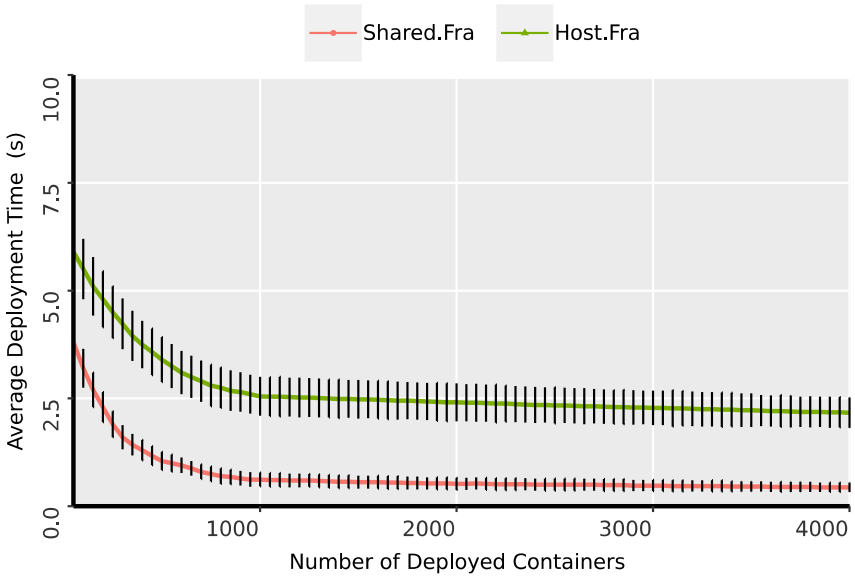
Figure 4.8 compares the average and standard deviation of deployment times during the same experiment. After deploying the first dozen deployments, in the *fra* trace

the average shared-cache deployment time is approximately 4 s whereas the non-shared cache observes deployment times close to 6 s. This difference persists through the entire workload: the mean deployment time of shared caches stabilizes to a value 78% lower than non-shared caches. In the case of the *dal* trace we observe a similar behavior. The lower cache hit rates of this difficult workload imply that the average deployment times remain above 2 s. However, here as well the shared cache delivers significantly lower deployment times than the non-shared scenario (37% reduction of the stabilized mean response time). We also observe smaller standard deviations of the deployment times in the shared-cache scenarios, which indicates that deployment times are more predictable than in the case of non-shared caches.

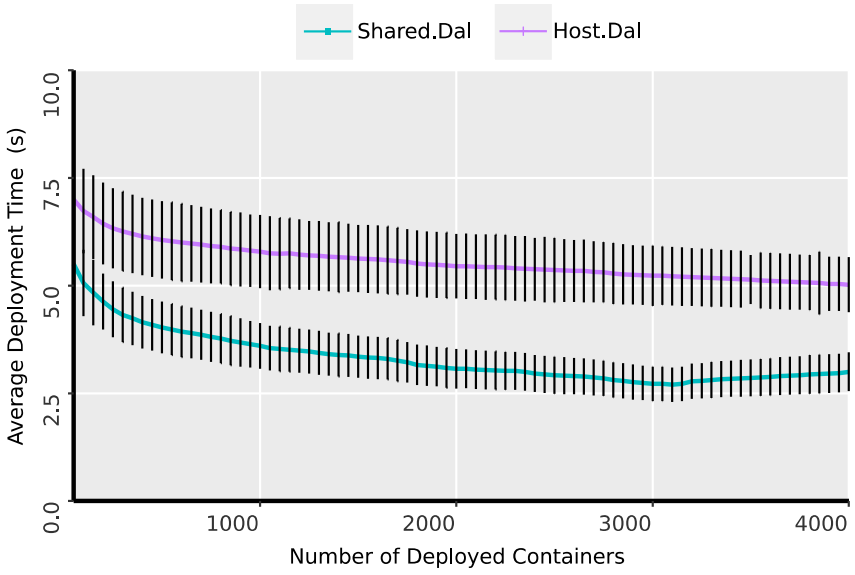
These significant differences in container deployment times may significantly impact the perception that fog applications' end users have about the performance of the overall system.

4.5 Conclusion

Docker was implemented with the assumption that every server's local cache would be large enough to store all the relevant container images after they are first downloaded. This assumption is however not true in fog computing environment where the compute resources are split between a large number of relatively weak machines. In such environments we extended Docker with a cache replacement policy which evicts unused images and maintains an acceptable image cache size. Splitting the available cache size also negatively impacts the cache hit rates because the same popular images must be downloaded and stored separately in multiple disjoint caches. We therefore proposed sharing caches between multiple co-located fog nodes. Our trace-based evaluations show that the proposed design achieves significant cache hit improvements, leading to reductions of average container deployment times between 37% and 78% depending on the scenarios. In fog computing environments container deployment must be assumed to be a frequent operation. Reducing container deployment times will therefore directly benefit the end users and help provide them with agile and responsive applications and services.



(a) *Fra* availability zone.



(b) *Dal* availability zone.

Figure 4.8 – Deployment time of shared vs. non-shared image caches under a workload of 4000 container deployments.

CHAPTER 5

SPEEDING UP THE DOCKER IMAGE DEPLOYMENT

This chapter presents the second contribution of the thesis. The second opportunity to reduce container deployment time is by improving the Docker image deployment process itself. Docker image deployment in resource-constrained fog nodes such as Raspberry Pis can be painfully slow: deploying one image before starting the container may take multiple minutes. We show that this slow deployment time is not only due to the resource-constrained nodes but also to Docker's inefficient usage of the hardware resources while deploying an image. We therefore propose Docker-pi which is an amalgamation of three optimization solutions which together reduce image deployment times significantly.

5.1 Introduction

Sharing Docker image cache improves cache hit ratio and reduce container deployment time between 37% and 78% depending on the scenarios, however, Docker still needs to download the container image while the container deployment request is launched for the first time. Deploying container images can be painfully slow, in the order of multiple minutes depending on the container's image size and network condition. However, such delays are unacceptable in scenarios such as a fog-assisted augmented reality application where the end users are mobile and new containers must be dynamically created when a user enters a new geographical area. Reducing deployment times as much as possible is therefore instrumental in providing a satisfactory user experience.

We show that this poor performance is not only due to hardware limitations. In fact it largely results from the way Docker implements the container's image download op-

eration: Docker exploits different hardware subsystems (network bandwidth, CPU, disk I/O) sequentially rather than simultaneously. We therefore propose three optimization techniques which aim to improve the level of parallelism of the deployment process. Each technique reduces deployment times by 10-50% depending on the content and structure of the container's image and the available network bandwidth. When combined together, the resulting "Docker-pi" implementation makes container deployment up to 4 times faster than the vanilla Docker implementation, while remaining totally compatible with unmodified Docker images.

Interestingly, although we designed Docker-pi in the context of single-board computers, it also provides 23–36% performance improvements on high-end servers as well, depending on the image size and organization.

The chapter is organized as follows. Section 5.2 analyzes the deployment process and points out its inefficiencies. Section 5.3 proposes and evaluates three optimizations. Finally, Section 5.4 discusses practicalities, and Section 5.5 concludes.

5.2 Understanding the Docker container deployment process

To understand the Docker container deployment process in full details we analyzed the hardware resource usage during the download, installation and deployment of a number of Docker images on a Raspberry PI-based infrastructure. The limited hardware capabilities were instrumental in highlighting the inefficiencies of this process, which would be more difficult to pinpoint using faster server machines.

5.2.1 Experimental setup

We monitored the Docker deployment process on a testbed which consists of three Raspberry Pi 3 machines connected to each other and to the rest of the Internet with 10 Gbps Ethernet [183]. The testbed was installed with the latest Docker version (17.06) This setup also allowed us to emulate slower network connections — which are arguably representative of real fog computing scenarios — by throttling network traffic at the network interface level. We used the `tc` (Traffic Control) command to run experiments either with unlimited bandwidth, or with limits of 1 Mbps, 512 kbps or 256 kbps.

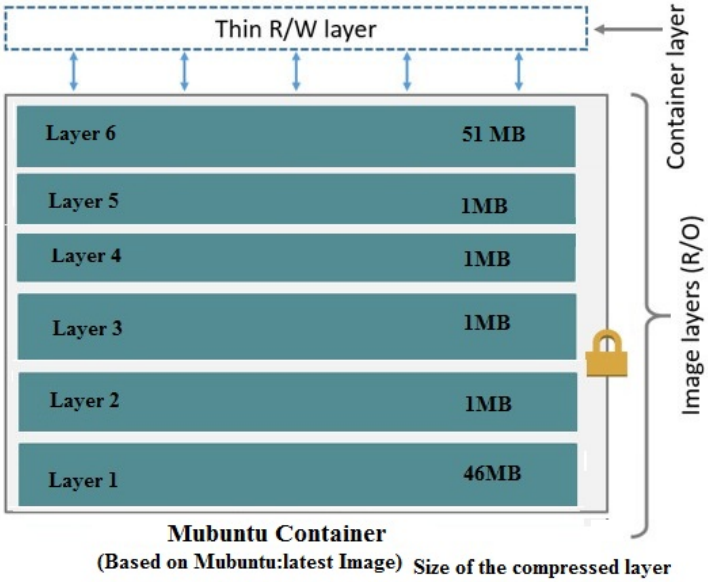


Figure 5.1 – Structure of the “Mubuntu” container image.

Table 5.1 – Structure of the Docker images.

	Ubuntu image	**Mubuntu image**	**Biglayers image**
6th layer	–	51 MB	-
5th layer	<1 MB	<1 MB	-
4th layer	<1 MB	<1 MB	62 MB
3rd layer	<1 MB	<1 MB	54 MB
2nd layer	<1 MB	<1 MB	64 MB
1st layer	46 MB	46 MB	52 MB
Total size	50 MB	101 MB	232 MB

Table 5.1 depicts the three Docker images we used for this study. The first image simply conveys a standard *Ubuntu* operating system: it is composed of one layer containing most of the content, and four small additional layers which contain various updates of the base layer. The second is the *Mubuntu* image already presented in Figure 5.1. Finally, as the name suggests, the *BigLayers* image is composed of four big layers which allow us to highlight the effect of the layering system on container deployment performance.

We instrumented the testbed nodes to monitor the overall deployment time as well as the utilization of important resources during the container deployment process:

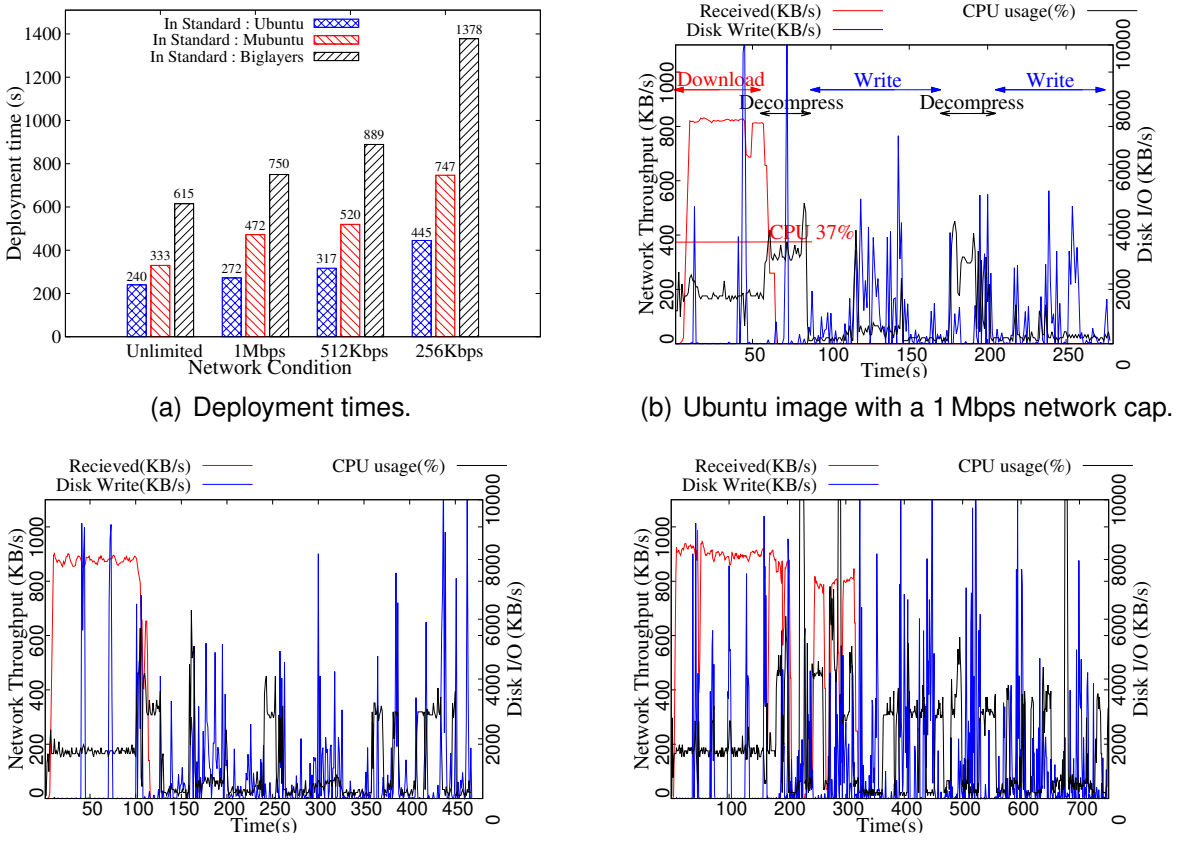
— **Deployment time:** We measured the deployment time from the moment the deployment command is issued, until the time when Docker reports that the container is started.
— **Network activities:** Network activities include incoming data and outgoing data during deployment. We used the *nethogs* tool to monitor the network activities during the whole deployment processes at a 1-second granularity[63]. The script traces the specific network activity of the Docker daemon, and therefore does not take other sources of background traffic into account.
— **Disk throughput:** We monitored the disk activity with the *iostat* Linux command which monitors the number of bytes written to or read from disk at a 1-second granularity.
— **CPU usage:** We monitored CPU utilization by watching the */proc/stat* file at a 1-second granularity.

Unless otherwise stated, every container deployment experiment was issued on an idle node, and with an empty image cache.

5.2.2 Monitoring the Docker container deployment process

Figure 5.2 depicts the results when deploying the three images using regular Docker. Figure 5.2(a) shows the deployment time of our three images in different network conditions: deploying the Ubuntu, Mubuntu and Biglayers images with unlimited network bandwidth respectively takes 240, 333 and 615 seconds. Clearly, the overall container deployment time is roughly proportional to the size of the image. When throttling the network capacity, deployment times grow steadily as the network capacity is reduced to 1 Mbps, 512 kbps, and 256 kbps. For instance, deploying the Ubuntu container takes 6 minutes when the network capacity is reduced to 512 kbps. This is considerable with regards to the deployment efficiency one would expect from a container-based infrastructure. However, the interesting information for us is the *reason* why deployment takes so long, as we discuss next.

Figure 5.2(b) depicts the utilization of different hardware resources from the host machine during the deployment of the standard Ubuntu image. The red line shows incoming network bandwidth utilization, while the blue curve represents the number of bytes written to the disk and the black line shows the CPU utilization. The first phase after the container creation command is issued involves intensive network activities,



(a) Deployment times.

(b) Ubuntu image with a 1 Mbps network cap.

(c) Mubuntu image with a 1 Mbps network cap.

(d) BigLayers image with a 1 Mbps network cap.

Figure 5.2 – Deployment times and resource usage using standard Docker.

which indicates that Docker is downloading the image layers from the remote image registry. By default Docker downloads up to three image layers in parallel. The duration of downloads clearly depend on the image size and the available network capacity: between 55 s and 200 s for the Ubuntu and Mubuntu images. During this phase, we observe no significant disk activity in the host machine, which indicates that the downloaded file is kept in main memory.

After the download phase, Docker extracts the downloaded image layers to the disk before building the final image of the application. The extraction of a layer involves two operations: decompression (which is CPU-intensive) and writing to the disk (which is disk-intensive). We observe that the resource utilization alternates between periods during which the CPU is busy ($\sim$40% utilization) while few disk activities are performed, and periods during which disk writes are the only notable activity of the system. We conclude that, after the image layers have been downloaded, Docker sequentially decompresses the image and writes the decompressed data to disk. When the image data is big, Docker alternates between partial decompressions and disk writes, while maintaining the same sequential behavior.

We see a very similar phenomenon in Figures 5.2(c) and 5.2(d). However, in here the downloading of the first layer terminates before the other layers have finished downloading. The extraction of the first layer can therefore start before the end of the download phase, creating a small overlap between the downloading and extraction phases.

5.2.3 Critical observations

From the previous experiments we derive a few important observations.

Overall deployment time

The overall deployment of a new container mainly involves three operations: searching for the cached image, pulling the image from the registry and starting the container. Our work assumes that the image is not cached on the machine, so every container deployment involves pulling the image from the registry. As we have seen, Docker takes a significant amount of time for pulling the image from the registry while the other two operations take a negligible amount of time. In this paper, we therefore mainly focus on optimizing the Docker image pull operation.

Pulling image layers in parallel

By default, Docker downloads image layers in parallel with a maximum parallelism level of three. These layers are then decompressed and extracted to disk sequentially starting from the first layer. However, when the available network bandwidth is limited, downloading multiple layers in parallel will delay the download completion of the first layer, and therefore will postpone the moment when the decompression and extraction process can start. Therefore, delaying the downloading of the first layer ultimately leads to slowing down the extraction phase.

Single-threaded decompression

Docker always ships the image layers in compressed form, usually implemented as a *gzipped* tar file. This reduces the transmission cost of the image layers but it increases the CPU demand on the server node to decompress the images before extracting the image to disk. Docker decompresses the images via a call to the standard *gunzip.go* function, which happens to be single-threaded. However, even very limited machines usually have several CPU cores available (4 cores in the case of a Raspberry Pi 3). The whole process is therefore bottlenecked by the single-threaded decompression. As a result the CPU utilization never grows beyond $\sim$40% of the four cores of the machine, wasting precious computation resources which may be exploited to speed up image decompression.

Resource under-utilization

The standard Docker container deployment process under-utilizes the available hardware resources. Essentially, deploying a container begins with a network-intensive phase during which the CPU and disk are mostly idle. It then alternates between CPU-intensive decompression operations (during which the network and disk are mostly idle) and I/O-intensive image extraction operations (during which the network and CPU are mostly idle). The only case where these operations slightly overlap are images such as Mubuntu and BigLayers when the decompress and extraction process of the first layer can start while the last images are still being downloaded.

This resource under-utilization is one of the main reason for the poor performance of the overall container deployment process. The main contribution of this paper is to

show how one may reorganize the Docker deployment process to maximize resource utilization during deployment.

5.3 Optimizing the container deployment process

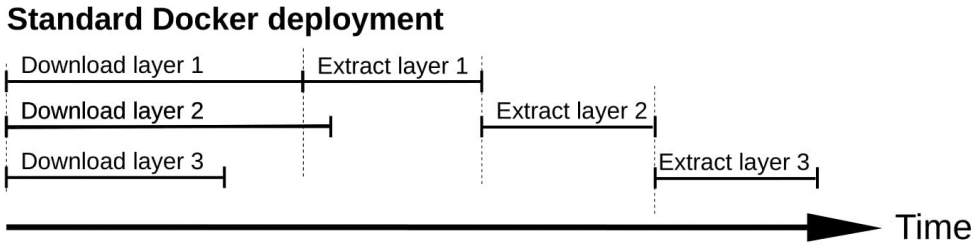
To address the inefficiencies presented in the previous section we propose and evaluate three optimization techniques to speed up the container provisioning time. Each optimization addresses a different issue in the standard Docker container deployment. We can therefore combine them all together, which brings significant performance improvement.

5.3.1 Sequential image layer downloading

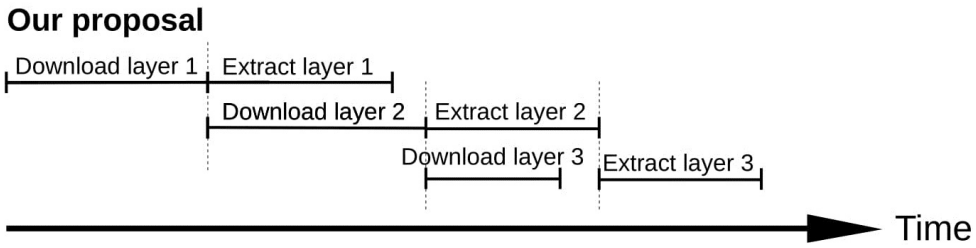
As previously discussed, Docker parallelizes image downloads from the central repository to the local node, with a default concurrency degree of 3. This is a classical technique to maximize the overall network throughput. However, in the specific case of Docker image downloads this strategy has a negative effect because the next phases of the container deployment, namely the decompress and extraction phases, must take place sequentially to preserve the Copy-on-Write policy of Docker storage drivers. The decompress & extract phase can start only after the first layer has been downloaded. Downloading multiple image layers in parallel will delay the download completion time of the first layer because this download must share network resources with other less-urgent image downloads, and will therefore also delay the moment when the first layer can start its decompress & extract phase.

The only cases where the decompression and extraction of one layer overlaps with the download of another image can be seen in Figure 5.2(c) and 5.2(d). They result from the fact that the download concurrency degree delays the downloading of the last images. We therefore propose to extend this phenomenon, and to reduce the download concurrency degree to one, essentially reverting to a sequential download of the image layers one after another.

Figure 5.3 illustrates the effect of downloading multiple layers sequentially rather than in parallel, in an example with an image made of three layers. In both cases, three threads are created to handle the three image layers. However, in the first option the downloads take place in parallel whereas the only required inter-thread synchroniza-



(a) Standard Docker pull with parallel layer download.



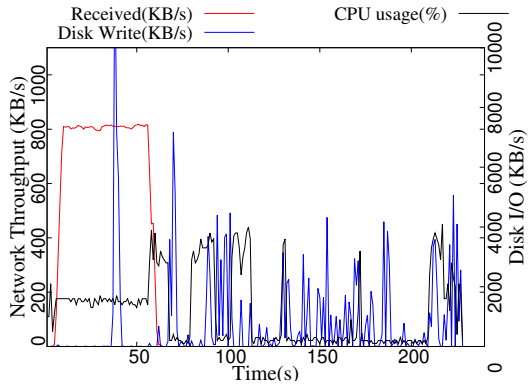
(b) Docker pull with sequential layer download.

Figure 5.3 – Standard and sequential layer pull operations.

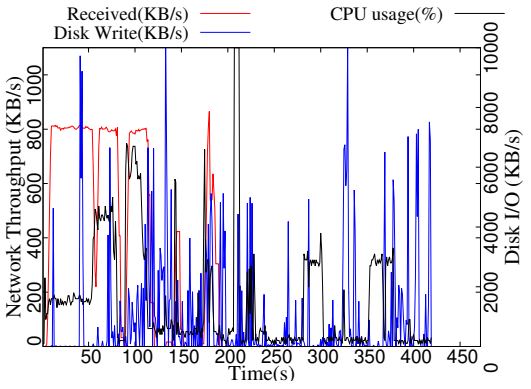
tion requires that the decompression and extraction of layer n can start only after the decompression and extraction of layer $n-1$ has completed. In sequential downloading, the second layer starts downloading only when the first download has completed, which means that it takes place while the first layer is being decompressed and extracted to disk. This allows the first-layer extraction to start sooner and it also increases resource utilization because the download and the decompress & extract operations make intensive use of different part of the machine's hardware.

Implementing sequential image downloading requires additional inter-thread synchronization: in this new model the downloading of layer n can start only after the end of the layer $n-1$ download, whereas the decompress & extract of layer n can start only after layer n has been downloaded *and* the layer $n-1$ has finished its extraction. A simple way to implement this is to set the "max-concurrent-downloads" parameter to 1 in the `/etc/docker/daemon.json` configuration file.

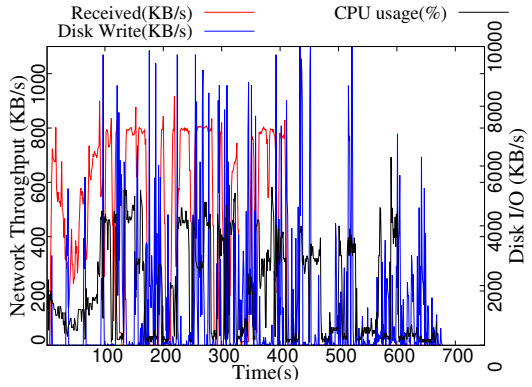
Figure 5.4 depicts the host machine's resource usage when using sequential downloading of our reference images, and compares the overall deployment times with various network bandwidth limitations. Figure 5.4(a) shows the resource usage when deploying the Ubuntu image with sequential downloading and a 1 Mbps network capacity. The figure is not much different from Figure 5.2(a) where the image downloads were



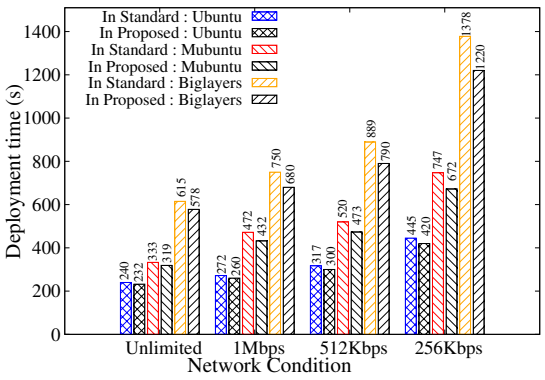
(a) Ubuntu image with a 1 Mbps network cap.



(b) Mubuntu image with a 1 Mbps network cap.



(c) BigLayers image with a 1 Mbps network cap.



(d) Deployment times.

Figure 5.4 – Resource usage and deployment time with sequential layers downloading.

done in parallel. The reason is that the Ubuntu image contains only one layer with a significant size, so the downloading and extraction of layers 2–5 is very short compared to layer 1. However, in Figures 5.4(b) and 5.4(c), we observe that after the downloading of layer 1 has completed, the utilization of hardware resources is much greater, with in particular a clear overlap between periods of intensive network, CPU and I/O resources. Also we can observe that the decompression of the first layer (visible as the first spike of CPU utilization) takes place sooner than in Figure 5.2.

Figure 5.4(d) compares the overall container deployment times with parallel and sequential downloads in various network conditions. When the network capacity is unlimited the performance gains in the deployment of the Ubuntu, Mubuntu and BigLayers images are 3%, 4.2% and 6% respectively.

However, the performance gains grow steadily as the available network bandwidth gets reduced. With a bandwidth cap of 256 kbps, sequential downloading brings improvements of 6% for the Ubuntu image, 10% for Mubuntu and 12% for BigLayers. This is due to the fact that slower network capacities exacerbate the duration of the download phases and increases the delaying effect of parallel layer downloading.

Sequential downloading therefore provides modest yet non-negligible performance gains. This technique works best when the deployed image contains multiple large layers, and when the network capacity is very limited. These conditions happen to match the properties of a container deployment in fog computing environments where non-trivial applications will be deployed in extremely distributed infrastructures which necessarily rely on limited commodity networks.

5.3.2 Multi-threaded layer decompression

Docker image layers are stored and downloaded in the form of a gzipped tar file. After downloading the files from the registry, the compute node therefore needs to decompress every layer before building the image on disk. In our experiments based on Raspberry PI and an unlimited network capacity, the duration of the decompression phase is greater than that of the image download. Increasing the speed of file decompression therefore has the potential to significantly reduce the overall container deployment time.

By default, Docker compresses image layers using gzip. Decompression is implemented entirely in the Go language using the standard *gunzip.go* library [16]. However,

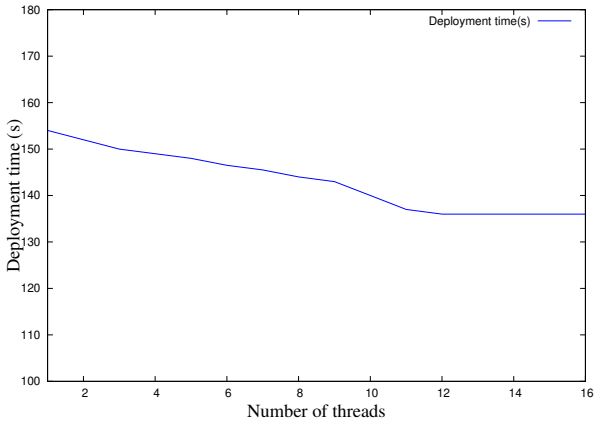


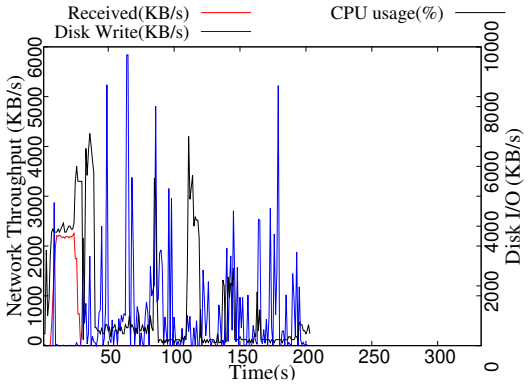
Figure 5.5 – Impact of the number of *pgzip* threads on the deployment time.

this function is single-threaded which means that it is unable to exploit multiple cores to speed up decompression. As a result, the CPU utilization during decompression never exceeds 40% of the four available cores in the Raspberry Pi machine.

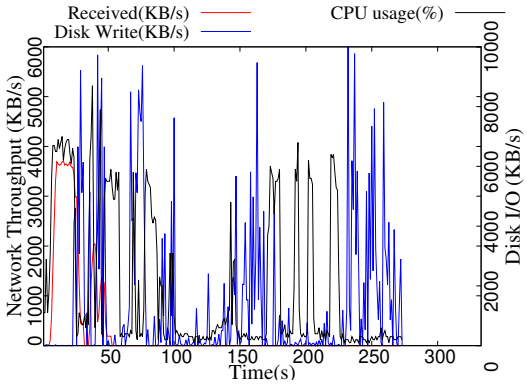
We therefore propose to replace the single-threaded *gunzip.go* library with a multi-threaded implementation so that all the available CPU resources may be used to speed up this part of the container deployment process. We use *pgzip*, which is a multi-threaded implementation of the standard gzip/gunzip functions [157]. Its functionalities are exactly the same as those of the standard *gzip*, however it splits the work between multiple independent threads. When applied to large files of at least 1 MB, this can significantly speed up decompression.

To determine the appropriate number of threads we should allow *pgzip* to use, we deployed a custom container image while varying the available number of threads. We used a very simple image which consists of a single layer of 20 MB in compressed form. This allows us to better isolate the performance gains of the parallel decompression from other effects such as the possible overlap of the decompression with the download of other layers. The results are depicted in Figure 5.5.

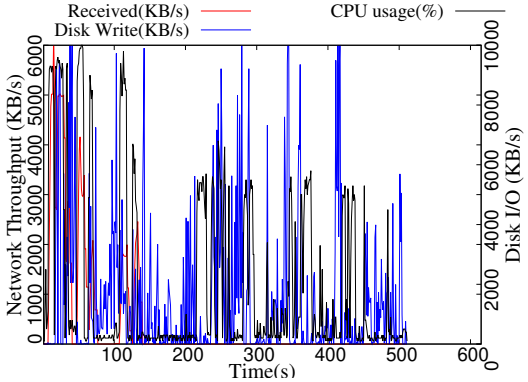
When *pgzip* uses a single thread, the performance and CPU utilization during decompression are very similar to the standard *gunzip* implementation. However, when we increase the number of threads from 1 to 12, the overall container deployment time decreases from 154 s to 136 s. At the same time, the CPU utilization during decompression steadily increases from 40% to 71% of the four available CPU cores. If we push beyond 12 threads, no additional gains are observed. We clearly see that the parallel decompression does not scale linearly, as it is not able to exploit the full ca-



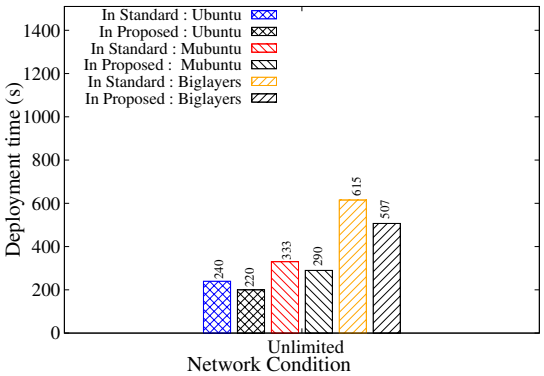
(a) Ubuntu image with multi-threaded decompression.



(b) Mubuntu image with multi-threaded decompression.



(c) BigLayers image with multi-threaded decompression.



(d) Deployment times.

Figure 5.6 – Resource usage and deployment time with multi-threaded image layer decompression.

pacity of the overall CPU: this is due to the fact that gzip decompression must process data blocks of variable size so the decompression operation itself is inherently single-threaded [173]. The benefit of multi-threading decompression is that other necessary operations during decompression (essentially data buffering and CRC verification) can be delegated to other threads and moved out of the critical path.

Figure 5.6 shows the effect of using parallel decompression when deploying our three standard container images with 12 threads. We observe in Figures 5.6(a), 5.6(b) and 5.6(c) that the CPU utilization is greater during the decompression phases than with standard Docker, in the order of 70% utilization instead of 40%. Also, the decompression phase is notably shorter. The same phenomenon is visible in all three images.

We also notice that the parallel image download phase is fairly CPU-intensive: in the examples of the Mubuntu and the BigLayers images which both have several large layers to decompress, the CPU utilization during downloading grows up to 70% for Mubuntu and even 90% for BigLayers. This clearly indicates that it would be pointless to attempt parallel image layer decompression while simultaneously downloading multiple image layers.

Finally, Figure 5.6(d) compares the overall container deployment times with parallel decompression against that of the standard Docker. The network performance does not influence the decompression time so we conducted the evaluation only with an unlimited network capacity. The performance gain from multi-threaded decompression is similar for all three images, in the order of 17% of the overall deployment time.

The standard single-threaded *gzip* implementation creates an unnecessary performance bottleneck because it under-utilizes the CPU resources during the decompression of the image layer. Although parallelizing data decompression is very hard and it cannot offer linear speedup, parallel decompression allows one to move the tasks not directly related to decompression to helper threads, which still provides interesting performance benefits. Multi-threading decompression increases the CPU usage, and reduces the overall Docker container deployment times.

5.3.3 I/O pipelining

Despite the sequential downloading and the multi-threaded decompression techniques, the container deployment process still under-utilizes the hardware resources. The reason is due to the sequential nature of the workflow which is applied to each

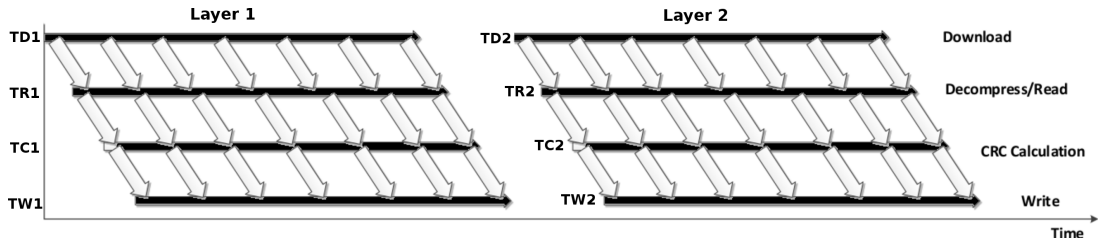


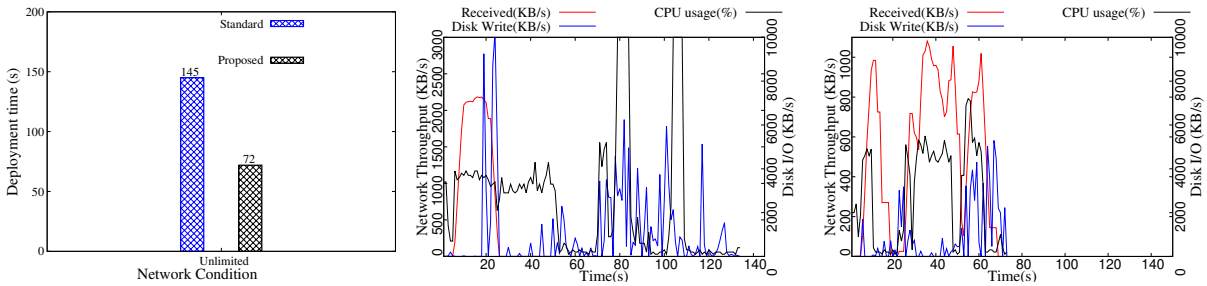
Figure 5.7 – Docker pull operation with I/O pipelining.

individual layer. Each layer is first downloaded in its entirety, then it is decompressed entirely, then it is extracted to disk. This requires Docker to keep the entire decompressed layer in memory, which can be significant considering that a Raspberry Pi 3 has only 1 GB of main memory [50]. Also, it means that the first significant disk activity can start only after the first layer has been fully downloaded and decompressed. Similarly, Docker necessarily decompresses and extracts the last layer to disk while the networking device is mostly inactive.

However, there is no strict requirement for the download, decompress and extraction of a single layer to take place sequentially. For example, decompression may start right after the first bytes of the compressed layer have been downloaded. Similarly, extracting the layer may start immediately after the beginning of the layer image has been decompressed.

We therefore reorganize the download, decompression and extraction of a single layer in three separate threads where each thread pipelines data to the next as soon as some data is available. In Unix shell syntax this essentially replaces the sequential "`download; decompress; crc-check; extract`" command with the concurrent "`download | decompress | crc-check | extract`" command. Figure 5.7 illustrates this technique with four threads responsible for downloading and decompression a Docker image layer. The thread TD1 downloads the image layer while TR1 performs the decompression, TC1 calculates the CRC, and finally TW1 writes the decompressed data to the disk. Since we stream the incoming downloaded data without buffering the entire layer, the thread TW1 can start writing content to disk long before the download process has completed.

We implemented pipelining using the *io.pipe()* GO API, which creates a synchronized in-memory pipe between an *Input(writer)* and an *Output(reader)* [17]. However, we must be careful about synchronizing this process between multiple image layers: for example, if we created an independent pipeline for each layer separately, the result



(a) Deployment time comparison. (b) Resource utilization with standard Docker. (c) Resource utilization with I/O pipelining.

Figure 5.8 – Deployment time and resource usage with I/O pipelining.

would violate the Docker policy that layers must be extracted to disk sequentially, as one layer may overwrite a file which is present in a lower layer. If we extracted multiple layers simultaneously we could end up with the wrong version of the file being given to the container. Rather than building complex synchronization mechanisms, we instead decided to rely on Docker's sequential downloading feature already discussed in Section 5.3.1. When a multi-layer image is deployed, this imposes that layers are downloaded and extracted one after the other, while using the I/O pipelining technique within each layer.

We now evaluate the I/O pipelining technique using a single-layer image only. In the next section we combine all three optimization techniques and therefore show the combined effect of the sequential download and the I/O pipelining. A single-layered image can be created using a so-called "flatten" operation which creates a single tar file out of a deployed image. Since the flattened version contains a single copy of every file (even though it may have been overwritten multiple times by different layers), the flattened image is usually slightly smaller than the sum of all the initial layers' sizes.

Figure 5.8 compares the deployment of a flattened image between standard Docker and the I/O pipelining technique. We can see in Figure 5.8(a) that the pipelined version is roughly 50% faster than its standard counterpart. The reason can be found in Figures 5.8(b) and 5.8(c). In the standard deployment, resources are used one after the other: first network-intensive download, then CPU-intensive decompression, then finally disk-intensive image creation. In the pipelined version all operations take place simultaneously, which better utilizes the available hardware and significantly reduces the container deployment time.

5.3.4 Docker-pi

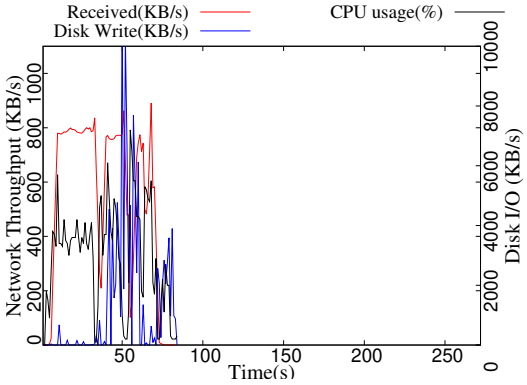
The three techniques presented here address different issues. Sequential downloading of the image layers speeds up the downloading of the first layer in slow network environments. Multi-threaded decompression speeds up the layer decompression by utilizing multiple CPU cores. Finally, I/O pipelining speeds up the deployment of each layer by conducting the download, decompress and extraction processes simultaneously, while avoiding having to keep large amounts of data in memory during the deployment process. We therefore propose *Docker-pi*, an optimized version of Docker which combines the three techniques to optimize container deployment on single-board machines such as Raspberry PIs. The implementation of Docker-pi is available online in the gitlab repository [3].

Deployment Time

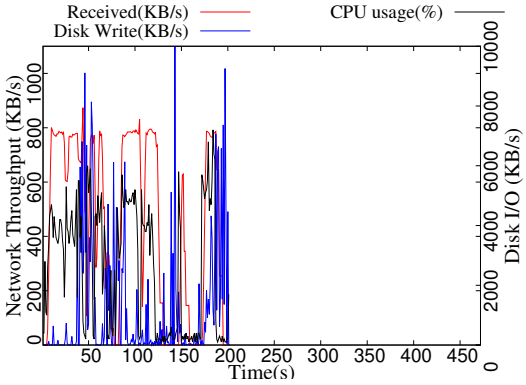
Figure 5.9 depicts the resource usage and deployment time of our three standard images using Docker-pi. We can clearly see in Figures 5.9(a), 5.9(b) and 5.9(c) that the networking, CPU and disk resources are used simultaneously and have a much greater utilization than with the standard Docker implementation. In particular, the CPU and disk activities start very early after the first few bytes of data have been downloaded.

Finally, Figure 5.9(d) highlights significant speedups compared to vanilla Docker: with no network cap, Docker-pi is 73% faster than Docker for the Ubuntu image, 65% faster for Mubuntu and 58% faster for BigLayer. When we impose bandwidth caps the overall deployment time becomes constraint by the download times, while the decompression and extraction operations take place while the download is taking place. In such bandwidth-limited environments the deployment time therefore cannot be reduced any further other than by pre-fetching images before the container deployment command is issued.

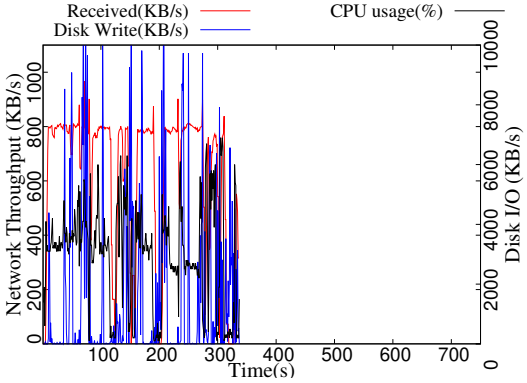
The reason why the gains are lower for the Mubuntu and BigLayers images is that the default download concurrency degree of 3 in vanilla Docker already makes them benefit from some of the improvements that we generalized in Docker-pi. If we increase the concurrency degree of vanilla Docker to 4, the BigLayers image deploys in 644 s whereas Docker-pi needs only 207 s, which represents 68% improvement.



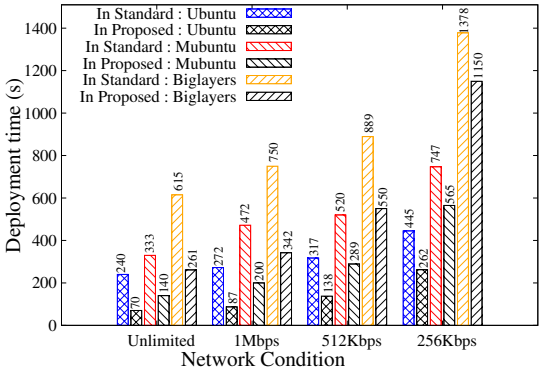
(a) Ubuntu image with a 1 Mbps network cap.



(b) Mubuntu image with a 1 Mbps network cap.



(c) BigLayers image with a 1 Mbps network cap.



(d) Deployment times.

Figure 5.9 – Resource usage and deployment time with Docker-pi.

Memory usage

We now evaluate the memory footprint of Docker and Docker-pi during the deployment process. For simplicity, we deployed a single layer image and extracted memory usage of the node by watching */proc/meminfo* file at a 1-second granularity. Figure 5.10 clearly shows the effect of pipelining on the memory footprint. Docker-pi starts extracting the layer to local disk immediately after the first few blocks of the layer are downloaded, therefore, memory footprint never exceeds 10 MB while deploying the image. In contrast, standard docker's memory usage is nearly 70 MB. The reason is that it keeps the complete compressed layer in memory, decompresses it entirely in memory again, before writing to disk and releasing both files from memory.

The memory footprint of standard Docker may vary depending on the image size and concurrency degree of parallel download. In a memory-constrained device like

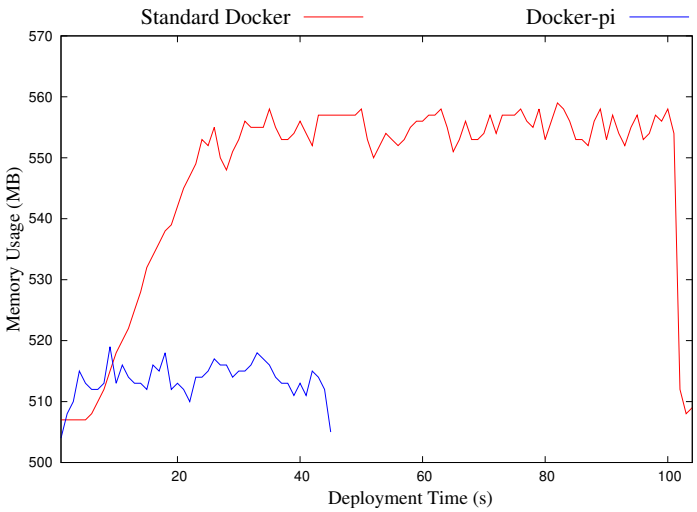


Figure 5.10 – Memory footprint of Docker and Docker-pi during container deployment.

Raspberry PI which has only 1GB of RAM, this may create significant bottlenecks. On the other hand, Docker-pi uses less memory and has a fairly constant footprint during the image pull operation irrespective of image structure, which makes it a better choice in environments such as memory-constraint fog computing devices.

Performance interference with already-running containers

In all experiments presented so far, container deployment took place in an otherwise idle machine. However, this scenario is unlikely in a busy fog computing environment where numerous independent applications share a limited number of physical resources. We therefore now evaluate the impact that container deployment has on already-running containers in the same machine.

We use an Apache Web server container [69] as the already-running container so we can observe its performance while another container with the single-layer image is being deployed. The Apache server serves a constant request workload produced by the http_load HTTP benchmarking tool [125]. We configured it to generate a constant load of 300 requests/second which fetch a single 5kB file. We monitor the Web server's network throughput using *nethogs* before, during and after the single-layer container is being deployed.

Figure 5.11 compares the upload throughput of the Web server while standard Docker or Docker-pi are deploying a new container image. We observe that in both

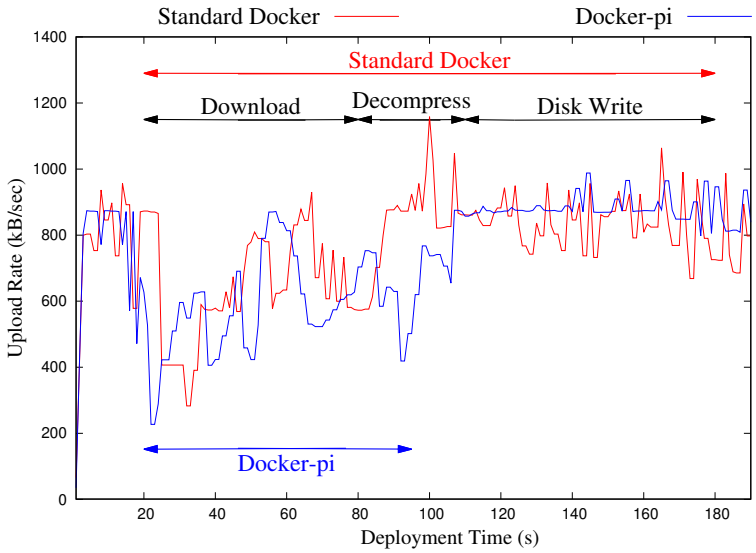


Figure 5.11 – Upload throughput of the Apache web server.

cases the Web server performance is affected by the simultaneous container deployment taking place in the same machine. This is largely due to the fact that the web server and the container deployment processes need to compete for limited resources such as available network bandwidth. Interestingly, both versions of Docker impose a similar performance reduction to the web server during the time of container deployment. However, Docker-pi deploys the new container faster so the duration of the interference it creates is shorter than using regular Docker. The fact that Docker-pi generates high resource utilization during container deployment does not seem to affect other containers in greater proportions than regular Docker.

5.4 Discussion

5.4.1 Should we flatten all Docker images?

Flattening all Docker images may arguably provide performance improvement in the deployment process. Indeed, multiple image layers may contain successive versions of the same file whereas a flattened image contains only the final version of every file. A flattened image is therefore always a little smaller than its multi-layered counterpart. Systems like Slacker actually rely on the fact that images have been flattened [76]. On the other hand, Docker-pi supports both flattened images and unmodified multi-layer

images. We however do not believe that flattening all images would bring significant benefits.

Docker does not provide any standard tool to flatten images. This operation must be done manually by first exporting an image with all its layers, then re-importing the result as a single layer while re-introducing the startup commands from all the initial layers. The operation must be redone every time any update is made in any of the layers. Although this process could be integrated in a standard image build workflow, it contradicts the Docker philosophy which promotes incremental development based on image layer reusability.

In a system where many applications execute concurrently, one may reasonably expect many images to share at least the same base layers (e.g., Ubuntu) which produce a standard execution environment. If all images were flattened this would create large amounts of redundancy between different images, creating the need for sophisticated de-duplication techniques [76]. On the other hand, we believe that the layering system can be seen as a domain-specific form of de-duplication which naturally integrates in a developer's devops workflow. We therefore prefer keeping docker images unmodified, and demonstrated that container deployment can be made extremely efficient without the need for flattening images.

5.4.2 Does Docker-pi work also for powerful server machines?

Although we designed Docker-pi for single-board machines, the inefficiencies of vanilla Docker also exist in powerful server environments. We therefore evaluate the respective performance of Docker and Docker-pi in the Grid'5000 testbed which is commonly used for research on parallel and distributed computing including Cloud, HPC and Big Data [22]. We specifically use a Dell PowerEdge C6220 server equipped with two 10-core Intel Xeon E5-2660v2 processors running at 2.2GHz, 128 GB of main memory and two 10 Gbps network connections, and do not cap the network bandwidth.

Figure 5.12 compares the deployment times of Docker and Docker-pi with our three standard images. Obviously container deployment is much faster in this environment than in Raspberry PIs. However, here as well Docker-pi provides respectable performance improvement in the order of 23% (Ubuntu), 29% (Mubuntu) and 36% (BigLayers). In this powerful server the network and CPU resources cannot be considered as bottlenecks so the sequential layer downloading and multi-threaded decompression

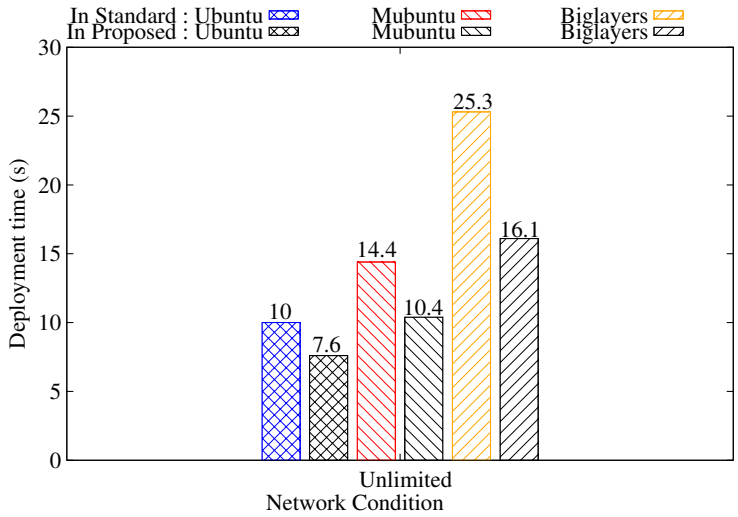


Figure 5.12 – Deployment time of Docker and Docker-pi in Grid'5000.

techniques bring almost no improvement compared to the standard Docker. On the other hand, the sequential nature of the download/decompress/extract process is still present regardless of the hardware architecture, so the I/O pipelining technique brings similar performance gains as with the Raspberry PI.

5.5 Conclusion

The transition from virtual machine-based infrastructures to container-based ones brings the promise of swift and efficient software deployment in large-scale computing infrastructures. However, this promise is not being held in fog computing platforms which are often made of very small computers such as Raspberry PIs. In such environments, deploying even a very simple Docker container may take multiple minutes.

We studied the Docker container deployment process in details and identified three sources of inefficiency: (1) Docker downloads multiple layers in parallel; (2) it uses single-threaded decompression; and (3) it sequentially downloads, decompresses and extracts any given image layer. We proposed three optimization techniques which, once combined together, speed up container deployment roughly by a factor 4. Last but not least, we demonstrated that these optimizations also bring significant benefits in regular server environments.

This work eliminates the unnecessary delays that take place during container deployment. Depending on the hardware, deployment time is now basically dictated only

by the slowest of the three main resources: network bandwidth, CPU, or disk I/O. As hardware will evolve in the next years the bottleneck may shift from one to the other. But, regardless of the specificities of any particular machine, Docker-pi will exploit the available hardware to its fullest extent.

CHAPTER 6

AVOIDING THE CONTAINER BOOT PHASE

This chapter presents the third contribution of the thesis. The final opportunity to improve container deployment is to optimize the boot phase of a container. The boot phase starts when creating the first process of the application and terminates when the application is ready to serve user requests. Some applications take a significant amount of time to boot the container. We propose to refactor the container boot phase using process checkpoint/restart, which allows one to deploy containers from a checkpoint image of an already started-container. This skips the container boot phase and significantly reduces the overall deployment time.

6.1 Introduction

Upon a container deployment request, Docker first creates the container with the given configuration. It then starts the application inside the container, which is also known as the container's boot phase. The boot phase starts by launching the main process of the application and terminates when the application is ready to accept user requests. This phase may take significant amount of time depending on the set of of instructions that the application must execute before being ready to serve user requests. For example, booting the popular *mysql* database requires reading configuration data as well as internal tables from disk, then warming up the data in memory, and finally carrying out initialization steps. Booting *mysql* requires about 10 s on a fast server before being available to serve end-user commands. This delay is significant compared to the container creation time which less than 1 s on the same machine [142].

As previously discussed, we expect that fog applications will need to be launched frequently in the same Point-of-Presence of fog infrastructures. Every time the application is launched in the fog servers, the same boot phase must be performed. In

this work, we aim to avoid repeating this boot phase when a container is repeatedly deployed in a PoP, which may in turn reduce the average container deployment time.

One way to avoid the container boot phase during the Docker container deployment process is to save the state of the application at the end of its boot phase. Since many applications always follow the same boot sequence during their deployment, we can save the state of a booted application and later re-start the container from this state in subsequent deployments. We propose to use checkpoint/restart techniques to create a snapshot of an application during container deployment, such that it can be used to restart the application in the later deployments.

Process checkpoint/restart is a technique where the state of a running application is saved for later reuse. It mainly involves two operations: the first one is checkpoint, where the state of a running process is captured, including the CPU registers, memory pages, open sockets, open files etc. The captured data are stored in a persistent file also known as checkpoint image [174]. The second operation is restart, where the process is restarted from the checkpoint state by reading the contents of the checkpoint image.

Process checkpoint/restart assumes that the application will be restarted in the same environment where it was checkpointed, so it does not save the full environment of a running application during checkpoint. An application environment contains the application's executable, shared libraries and data files, which are stored on disk. During restart of the application from the checkpoint image, the system expects exactly the same environment to be present in the system to smoothly restart the application. Hence, in order to enable checkpoint/restart in container systems, capturing the whole container environment is necessary during the checkpoint operation. During restoration, the container environment and checkpoint image must be present in the new system.

We propose to incorporate process checkpoint/restart in Docker container deployment. During the first deployment of a container in the PoP, after completing the boot phase and when the application is ready to accept user requests, we use the DMTCP tool to checkpoint the application, and we save the checkpoint image and the container environment [121]. In every subsequent deployment of the application, Docker creates the container but instead of bootstrapping the application normally, it calls DMTCP to restart application from the checkpoint image. Therefore, the container skips the boot phase which may reduce the container deployment time.

Integrating process checkpoint/restart in the Docker container deployment process requires one to address a number of issues: (1) Docker does not provide any standard API to checkpoint/restart containers; (2) As we intend to checkpoint a container in one system and restart in another to deploy new containers, the complete container environment needs to be preserved during checkpoint; and (3) Docker containers are expected to deploy frequently in different fog servers of a PoP, so an efficient mechanism is necessary to share the checkpoint images and the application environments across the servers of a PoP.

In this chapter, we present a Docker container deployment design that integrates process checkpoint/restart. The system design has two main components: (1) A thin DMTCP container that enables Docker to checkpoint applications running inside the containers and to capture their container environment. It can also restart an application inside a container from its checkpoint image and container environment; and (2) a mechanism which leverages Ceph RADOS block devices to share the container environments and checkpoint images across servers of a PoP [112]. The performance evaluation of the proposed design shows that it can deploy containers and skip the boot phase which results in 1.0x up to 60.0x times improvement in the container boot phase time depending on the type of the container.

This chapter is structured as follows: Section 6.2 presents the state of the art of checkpoint/restart. Section 6.3 discusses the different issues in developing container deployment with DMTCP; Section 6.4 presents the proposed Docker container deployment using DMTCP. Finally the performance evaluation of the proposed container deployment is presented in Section 6.5 and Section 6.6 concludes the chapter.

6.2 State of the art

Checkpoint/restart is a technique to save the current state of a single process for later restart [174]. It is primarily used to achieve fault-tolerance of an application where the application can be restored to a previous stable state after a crash [83]. It is also useful for many other purposes such as application debugging [82], process migration [176], application scaling [77], and virtual machine deployment [71]. Checkpoint/restart can be implemented at different levels of the software stack.

Operating System level checkpointing: The operating system level implementation of checkpoint/restart saves the state of all the running processes of the OS at periodic intervals and allows the operating system to restart from the last checkpoint state [48]. A popular example is virtual machine snapshot and clone [146]. However, in a large scale system that runs several applications in parallel, this type of checkpoint/restart requires large space to store the checkpoint images.

Application-level checkpointing: The implementation of checkpoint/restart is done in the application code. It therefore only checkpoints a single target application instead of saving the state of all the applications running in the system. Developers may insert checkpoint/restart in the application source code to trigger a snapshot of the application state periodically [177]. This process reduces the size of the checkpoint image as it targets only one specific application and not the whole OS. However, application-level checkpoint/restart creates additional complexity for the developers as the same checkpoint/restart code must be updated in every new release of the application source code. It is also not transparent to users and applications [130].

System-level checkpointing: system-level checkpointing addresses the transparency issue from application-level checkpointing [10]. Similar to application-level checkpointing, it only checkpoints a single target application, however checkpoint/restart is implemented in a separate library. This makes checkpointing completely transparent to the application. It also allows one to trigger the checkpoint process at any arbitrary time or upon any specific event. Another advantage of using system-level checkpoint is that it does not need to modify the application source code. For these reasons, in this work we will adopt the system-level approach to checkpoint/restart containers.

System-level checkpointing can be implemented in many ways, and many tools are available [20]. Some notable ones are BLCR (Berkley Lab's Checkpoint/Restart) [59], CRIU (Checkpoint and Restart In User-space) [151] and DMTCP (Distributed Multi-Threaded CheckPointing) [10]. All these tools allow one to save and restore a running application; however they differ in many ways: for example how the state of a process is preserved, which information about a process state is preserved in the checkpoint image, how the preserved process information is stored (compressed or uncompressed), APIs, and command-line interfaces.

BLCR is the most used library to checkpoint and restart a process [59]. TODAY, CRIU is the most used for single-host checkpointing. BLCR is now very poorly maintained, (and so not used very much at all any more). See, for example, the frequency of maintenance release updates on its web site. It captures the state of a process such as its context and allocated memory regions, and saves the checkpoint data in a file. To implement checkpoint/restart, BLCR modifies the Linux kernel (as a patch or a loadable module). As a result it needs to update its kernel module frequently to support new kernel versions. Another shortcoming of BLCR is that it does not checkpoint open files and communication channels (i.e., open sockets) which are an essential part of many fog applications. And in addition, it doesn't handle SysV shared memory. SysV shared memory among processes is become very common, for the sake of efficiency. The alternative, BSD shared memory, relies purely on parent-child relationships.

CRIU employs a hybrid approach which combines kernel-space and user-space to checkpoint and restart applications [151]. It is primarily used to support checkpoint and restart of Linux containers. It started with checkpointing of servers outside any container. During the checkpoint operation, CRIU copies the contents of memory pages, open sockets, open files etc. of a process in a file. Although Docker in its experimental mode supports CRIU-based container checkpointing, it has a number of limitations. In particular, restoration of network connections (both TCP and UDP) is currently not functional [2]. Also, although Docker supports many storage drivers for container file system management, container restoration with CRIU is only possible using AUFS or OverlayFS [152].

DMTCP is a user-level checkpointing tool which requires no system privileges and does not require to modify the library or user source code or any kernel module [10, 121]. When checkpointing an application, DMTCP relies on the `/proc` Linux file system to capture a map of memory pages, open file descriptors, open sockets etc. of each process of the application. While restoring the application, DMTCP reads the checkpoint image and forks all the processes which are immediately restored to their previous state.

Multiple checkpoint approaches rely on customizing kernel modules (BLCR) and exporting kernel internals to the proc interface (CRIU). This type of implementation limits the checkpointing features since the system may restore applications only in machines with exactly the same kernel version (for BLCR) and with a compatible kernel version

(for CRIU) [207]. Another disadvantage of such type of checkpointing techniques is that any change in the kernel may require updating the checkpointing system as well. Because of these reasons most of the kernel-based checkpoint/restart approaches do not work with recent kernel versions (for example BLCR was not updated since 2013) [58]. Moreover, CRIU and BLCR, cannot fully checkpoint network communications. Checkpointing network communications is also vital in modern applications. Therefore, in our work, we choose to adopt DMTCP which works entirely in user space and can checkpoint and restart network sockets. In the next section, we discuss different challenges which arise when integrating DMTCP with Docker containers.

6.3 Design issues

The goal in this chapter is to enable Docker: (1) to checkpoint a running container after completing its boot phase; and (2) to deploy containers from an already checkpointed image. This may remove the container boot phase while deploying the container. However, to successfully implement above goals, we need to address three challenges which are illustrated in the next section.

6.3.1 Integration of DMTCP with Docker

When deploying a container, Docker pulls the container image if necessary, then creates the container with the given configuration and immediately starts the boot phase. This is done by launching the first process of the application. The boot phase terminates when the application is ready to serve user requests. Our proposed design rather intends to start application inside the container from a checkpoint image. First, the system should enable Docker to use DMTCP to checkpoint an application inside the container in the first deployment. Second, while deploying containers in subsequent deployments, Docker should restart the application from the checkpoint image instead of starting the boot phase normally. We discuss how Docker and DMTCP are integrated to implement these two operations in Section 6.4.1.

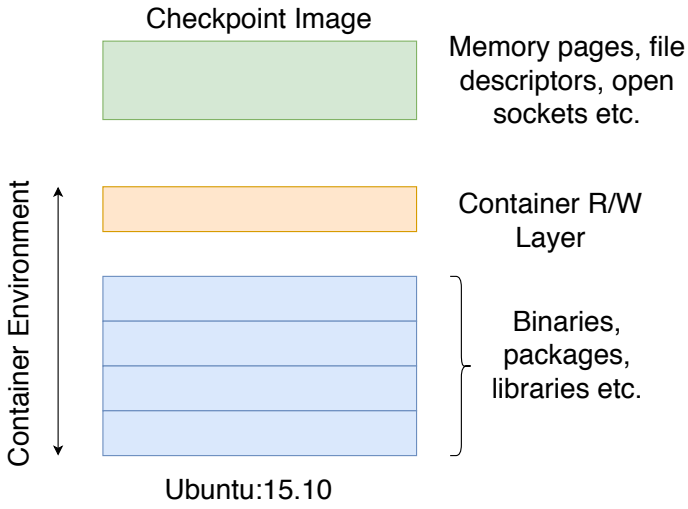


Figure 6.1 – Contents of a container environment and a DMTCP checkpoint image

6.3.2 Sharing container environments

The components of a container include all the binaries, packages, libraries and essential data of the application running inside the container. Figure 6.1 depicts these components. The read-only Docker image layers contain all the necessary files to start the application such as binaries, packages and shared libraries. All the modified files from the image layers, including application data, are stored in the container read/write layer. During checkpoint, DMTCP captures the memory pages, file descriptors and open sockets of the application, which are essential to save the state of the application.

During restart, DMTCP reads the checkpoint image and restarts the application from the checkpoint state. It is usual that at run time an application may access the application data, and shared libraries which are stored in the container environment. Therefore, the container environment captured during the checkpoint must be recreated before the application is restored. We discuss in Section 6.4.2 how container environments (Docker image layers and container R/W layers) are captured during checkpoint and shared among the servers of a PoP.

6.3.3 Sharing the checkpoint images

Fog applications are expected to deploy frequently across several servers of a PoP. It is likely that a checkpoint image and its associated container environment will be accessed in several servers when the same container is being deployed repeatedly.

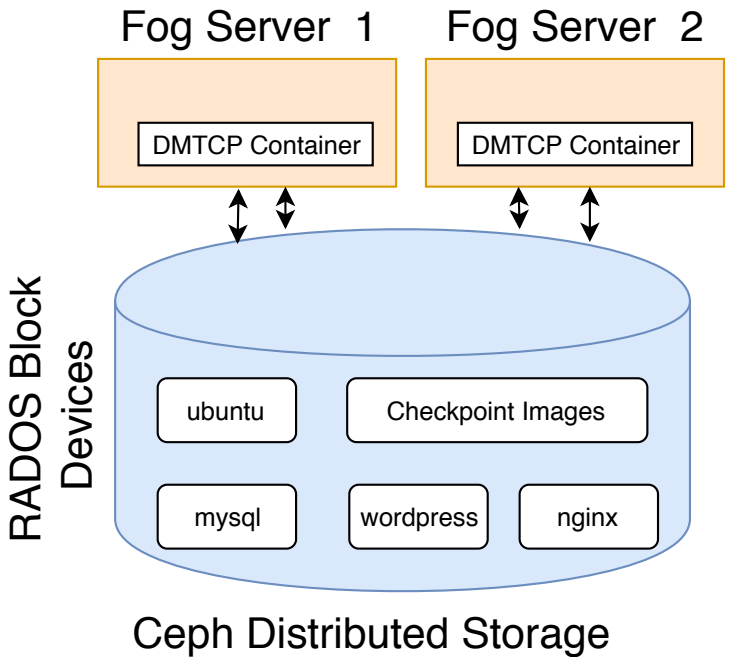


Figure 6.2 – Proposed container deployment with DMTCP

Therefore, an efficient mechanism is required to share the checkpoint images and container environments across the servers of a PoP. We discuss how we address this issue in Section 6.4.3.

6.4 Proposed container deployment design

Figure 6.2 illustrates the proposed Docker container deployment with DMTCP checkpoint and restart. In this design, Docker can use DMTCP to checkpoint an application after the container boot phase. Later it can deploy any number of containers from the checkpoint image, which skips the boot phase. To implement this, the proposed design has two main components:

— A DMTCP lightweight container which is deployed in all the servers of a PoP. It is responsible for checkpointing and restarting applications inside the containers.
— Ceph Rados Block Devices (RBD) use the Ceph distributed storage to share container environments and checkpoint images across the servers of a PoP.

6.4.1 DMTCP lightweight containers

Docker implements many APIs which handle various functionalities of a container life cycle. For example, `docker ps` shows all the containers currently running in the host machine. However, Docker does not provide any API to integrate external tools such as DMTCP which can checkpoint/restart applications running inside containers.

We therefore create a lightweight container image which contains the binaries of DMTCP and all the required libraries to perform checkpoint and restart. This allows us to avoid having to modify the Docker source code. Another advantage of the lightweight container is that the same DMTCP container can be used multiple times to checkpoint and restore different applications. Finally, the lightweight container is easy to maintain and distribute across the servers of a PoP with the help of a registry server.

The two fundamental purposes of the DMTCP lightweight container are: (1) to checkpoint application which is running inside a container and (2) to restore application from a checkpoint image. Although this lightweight container can perform these operations, it does not contain the application environments in the container file systems. We discuss how application environments are stored, managed and made available to the lightweight container's file system in the next section.

6.4.2 Ceph block devices

Snapshotting container environments

When checkpointing a container with DMTCP, Docker needs to save the container's read/write layer to capture its environment. This can be implemented using the standard `docker commit` command which creates a new image by adding the container's read/write layer on the top of the image [92]. The resulting image thus contains the container environment and can be shared across multiple servers through the registry. Although this simple mechanism can preserve and share a container's environment, it also presents number of challenges: Firstly, during `docker commit`, Docker momentarily pauses the container which generates application down-time [26]; Secondly, this operation is not transparent to the application, as the commit operation has to be performed externally; Thirdly, in order to distribute the container's environment, the image has to be pushed first to a registry server and then pulled multiple times in different nodes. Both operations take significant network resources and require one to store the

container environment redundantly into multiple servers. Due to the drawbacks listed above, we propose to store the container environment in the Ceph distributed storage system [162]. We discuss how Docker image layers and container read/write layers are organized, stored, snapshot and shared across the servers of a PoP in the following sections.

Ceph distributed storage

Ceph is a highly scalable distributed storage system, and it is considered suitable for distributed fog infrastructures (as discussed in Section 4.3.1). It can provision storage using different models: object storage [46], block storage [112] and POSIX-compatible Ceph file systems [161]. Ceph RADOS Block Devices store fixed-size blocks of data (for example 1 MB blocks). The block devices are thin-provisioned and resizable, and the stored data can be striped over multiple object storage drives (OSD). Ceph provides a kernel module and the *librbd* library to create, manage and control the block devices [107, 116].

Ceph provides many features to manipulate block devices such as *snapshot* and *clone* [120], persistent cache (for caching) [118], and RBD Mirroring (for replication) [117]. The *snapshot* and *clone* feature allows users to create multiple children of a block device using Copy-on-Write (CoW). Once a block device has been snapshot, it becomes read-only. Multiple block devices can then be cloned from the snapshot. All further updates are written in the cloned block devices following Copy-on-Write (CoW). Ceph implements Copy-on-Write (CoW) at block-level granularity.

Ceph RBD to store container environment

We propose to store the full container environment in the Ceph distributed storage system. A container environment contains the Docker image's read-only layers and the container read/write layer. Instead of keeping the Docker images and the container read/write layers in the local file systems which is the standard option, we propose to store the environment in the shared Ceph RADOS Block Devices (RBD). To implement this, a pool of RBDs is created and a RBD is assigned to each container environment.

Storing container environments in Ceph distributed storage brings many advantages: (1) multiple Docker servers running in different nodes can share the same environment, which gives storage efficiency; (2) the container environment can be shared

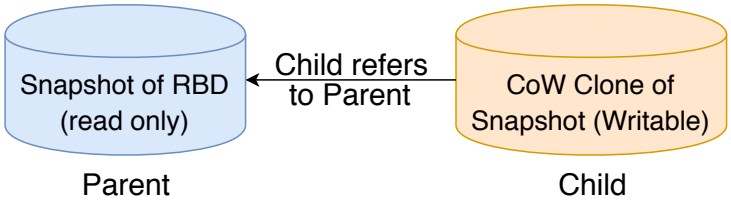


Figure 6.3 – Snapshot and clone feature of Ceph

without the use of a registry server; (3) the system can efficiently snapshot a container environment and share across servers of a PoP; and (4) Ceph performs CoW at block-level granularity which is more efficient in terms of performance and space utilization compared to file-based CoW [181].

This simple mechanism allows us to share container environments across many co-located servers. However, we also need to make it available to the DMTCP lightweight container. We therefore need to configure the system such that the Ceph block storage is accessible to the DMTCP lightweight containers.

Container environment layering with *snaphshot* and *cloning*

Ceph supports a feature to save the state of a block device using snapshotting [120]. The outcome of a snapshot operation is that the block device becomes read-only. Ceph can then create multiple cloned block devices from any snapshot. All further modifications over the snapshot are performed in the cloned devices following copy-on-write (CoW). The obvious advantage of *snaphshot* and *clone* is that it can re-use the snapshot block devices for multiple purposes. Figure 6.3 illustrates the working principle of snapshot and clone of a Ceph RBD. The left-side RBD becomes read-only once the snapshot is done, and it is generally referred to as the parent. The right-side RBD is created with a clone operation from the parent and it is referred to as the child. Now all the modifications in the snapshot are done in the child RBD.

We use the Ceph *snapshot* and *clone* feature to implement container environment layering. The container environment layers are read-only except the top layer which is read/write layer. This is similar to Docker image layering which allows one to re-use image layers, where all the layers remains remain read-only except the top layer where the file system updates are performed. To implement the layering, we assign a snapshot ID to each new snapshot created from a RBD. This snapshot ID is used to create multiple cloned RBDs where the modification of the snapshot can be performed.

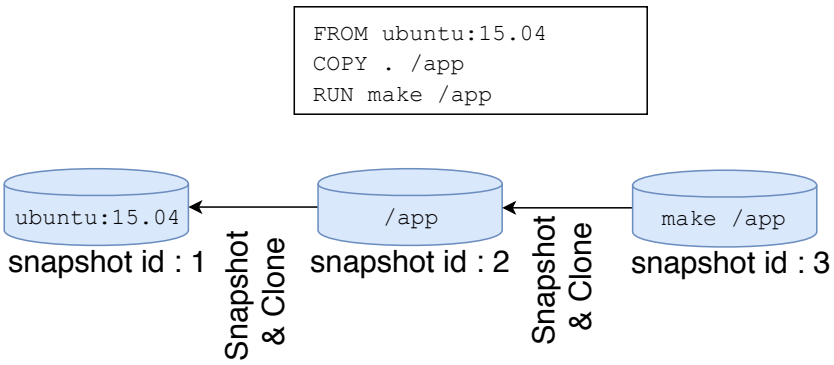


Figure 6.4 – Container environment layering with snapshot and clone

Figure 6.4 shows how the container environment layering works in the proposed system using Ceph RBDs with a Dockerfile which contains three instructions. The first instruction creates a Ceph RBD, copies the contents of Ubuntu:15.04 in the Ceph RBD, and takes a snapshot. With the second instruction, first a clone RBD is created from the parent snapshot, and then the current directory is copied to the clone block device. The third instruction compiles the content of a specific directory. Finally, a snapshot ID is generated for each container environment which can be used for further modification in the environment.

How is a container environment available to the DMTCP lightweight container?

Our design stores the container environments in Ceph RBDs which helps to share the environments across co-located servers of a PoP. However, in order to enable DMTCP to checkpoint/restart an application, the DMTCP lightweight container must access the container environment. Figure 6.5 illustrates the set of instructions that the system must perform on the block devices to make the container environment fully available in the DMTCP lightweight container's file system.

1. **Creating a container read/write layer:** we use the snapshot ID of the container environment to create a clone RBD. The clone RBD is therefore a read/write layer where any update in the container environment will be performed.
2. **Mapping the RBD:** This operation registers the clone RBD to the local kernel block device. With this operation, the kernel assigns a block device identifier such as /dev/rbd* to the RBD. The map operation (e.g., `rbd map`) is implemented in the librbd Ceph kernel module.

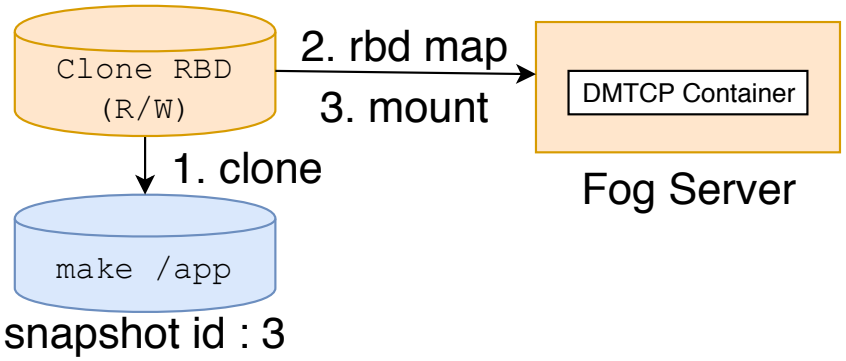


Figure 6.5 – How block device is mounted in container file system

2. **Mounting the RBD to the container file system:** When creating a container with `docker run`, the system needs to assign two flags in the container configuration: the `-device` to enable the device inside the container and the `-privileged` flags to mount the block device inside the container file system. The container now can access the block device locally. It then mounts the block device in the local directory of the file system using the standard *mount* command.

6.4.3 Container deployment with checkpoint/restart

Figure 6.6 shows the container deployment design flowchart. Our system relies on two Ceph RBDs to store the container environment and the checkpoint image respectively. The system first creates the two RBDs by cloning from their respective snapshots: (1) the container RBD is created from the snapshot of the container environment. This RBD is used to store the changes in the container environment; and (2) the checkpoint RBD is cloned from the snapshot of the checkpoint RBD.

Docker then uses the DMTCP lightweight image to deploy the container. When creating the container, the system follows the procedure described in 6.4.2 to mount the two cloned RBDs in the local container file system. Once the container is created, the system then launches the application based on the type of deployment:

— Creating a checkpoint image: This type of deployment is done to create snapshot of the checkpoint image and the container environment. The system uses DMTCP to launch the application and boot the container. When the boot phase is completed, DMTCP checkpoints the application and stops the container. The

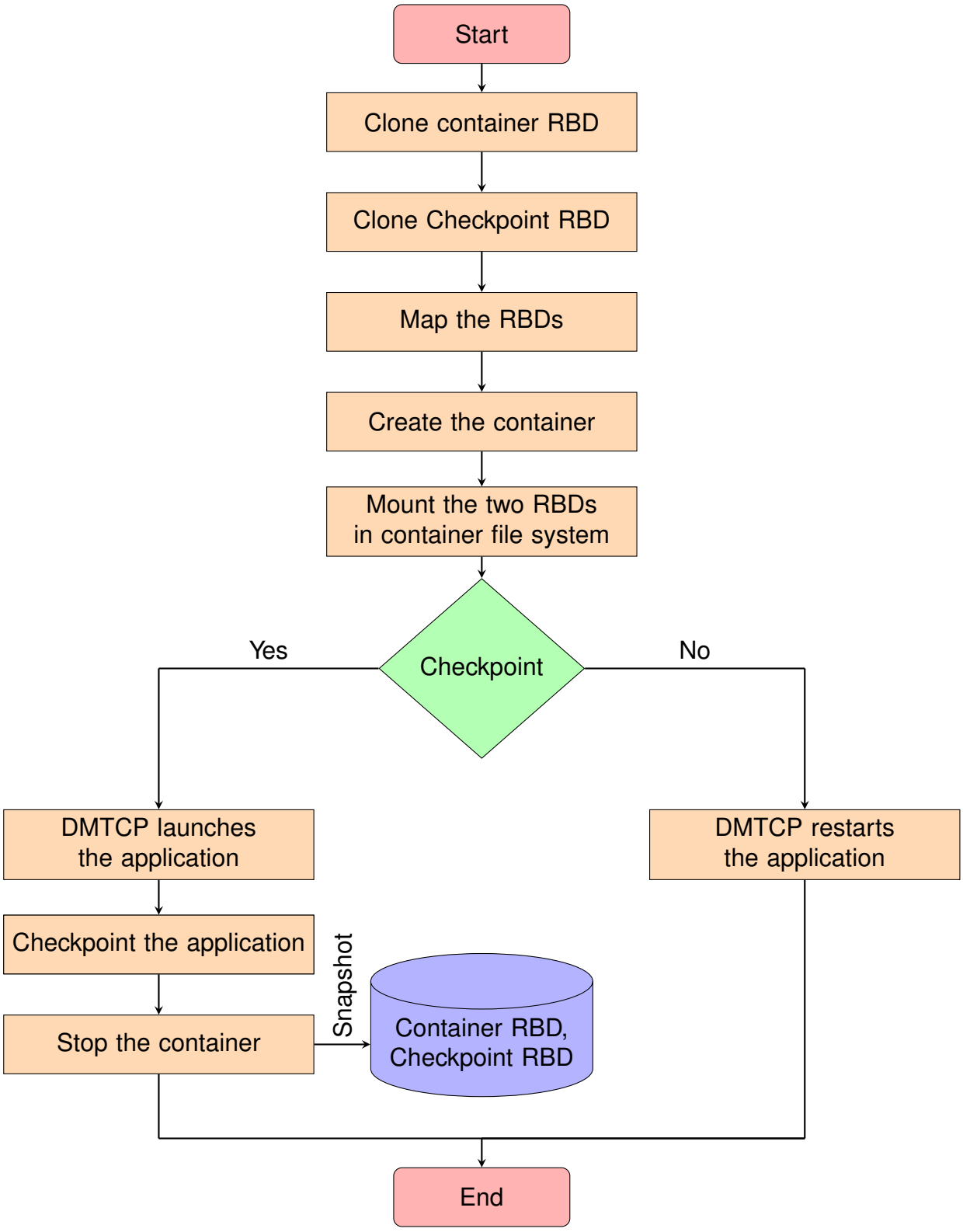


Figure 6.6 – Flowchart of the Docker container deployment with DMTCP.

resulting DMTCP checkpoint image is saved in the checkpoint RBD. Finally, the checkpoint RBD and container RBD are snapshot. The two snapshot RBDs can be shared across co-located servers of a PoP. In the subsequent deployment, those two RBDs are used to deploy the container.

— Deploying from a checkpoint image: in this case, the system deploys the application from a checkpoint image. The cloned checkpoint RBD and container RBD already have booted the checkpoint image and the container environment respectively. Therefore, the system uses DMTCP to restart the application from the checkpoint image.

6.5 Evaluation

We evaluate the performance of our container deployment design in a distributed fog environment. The experimental testbed is composed of five virtual machines representing the servers of a fog point-of-presence. These VMs are created using KVM on a Dell PowerEdge R430 server with two Intel Xeon E5-2620 v4 processors running at 2.10GHz, with 8 hyperthreaded cores each, and 64 GB of RAM. Each VM has 2 vCPUs, 1 GB RAM and 32 GB disk, and runs Ubuntu 18.04 server with 4.15.0-47-generic Linux kernel.

Building a Ceph cluster requires at least three Object Storage Daemons (OSD) where the objects are stored, plus one monitor and manager which are responsible for controlling, monitoring and managing the cluster. We set up the Ceph cluster using Ceph version 12.0.4 over the 5 servers of the testbed. The cluster has one monitor running in one machine, and five OSDs running in separate fog node. We also deploy the Ceph client in every machine.

We use Docker version 18.04 to deploy containers throughout the experiments. Finally, we build DMTCP Docker image based on DMTCP version 2.5.2 [121]. While deploying the containers, we make sure all the systems are idle to avoid any interference from other applications.

6.5.1 A use-case: Edge-sharelatex

We carried out the performance evaluation by deploying the *Edge-sharelatex* application [182]. *Edge-sharelatex* is a web-based application that allows users to edit

Table 6.1 – Name of the services and their purpose.

Service	**Purpose**	**Service**	**Purpose**
web	user-interface	clsi	latex compiler
filestore	store binary files	docstore	store texfile
tags	manage tags	notifications	notify users
contacts	manage contacts	spelling	check spelling
chat	manage chats	tracker	manage changes
real-time	state synchronization	updater	update document
Mongo	mongo database	redis	redis database

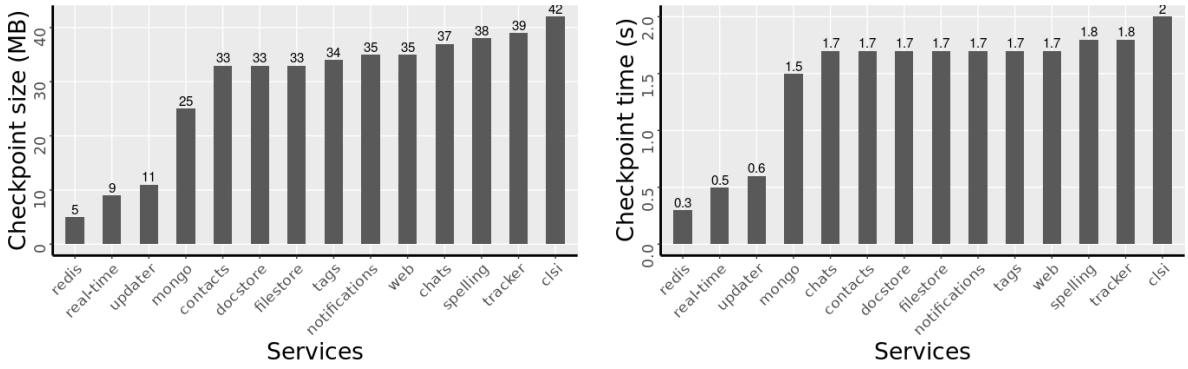
latex projects collaboratively, compile them and generate output. It is composed of 14 micro-services dedicated to different tasks. Table 6.1 shows the list of micro-services and their purposes. This gives us a set of 14 independent applications with different characteristics to evaluate our system.

Before the experimentation, we configure the Ceph RBDs with the container environment and checkpoint the image of each micro-service. When running the experiment, we deploy all the services in a single fog server to simplify the experiments. Furthermore, we make sure that each micro-service is running in a separate container. During the deployment, the system is kept idle to avoid any interference from other applications.

6.5.2 Checkpointing overhead

Our system checkpoints containers the first time they are deployed in a PoP. When checkpointing, the system momentarily stops the container and the resulting checkpoint image is stored in the Ceph distributed storage. In this section, we analyze the system overhead while checkpointing the Edge-sharelarex services. We particularly trace the checkpointing time and the size of the checkpoint image for each service.

Figure 6.7(a) depicts the checkpoint image size of the Edge-sharelatex services. The size of the checkpoint images varies from 5 MB to 42 MB depending on the service. We can clearly differentiate the services based on their checkpoint image size: for example, lightweight services such as redis, real-time, updater their have image size in the range from 5 MB to 11 MB. The checkpoint image size of the Mongo container



(a) Checkpoint image size (MB).

(b) Checkpointing time (s).

Figure 6.7 – Checkpoint image size and checkpointing time of the services.

is 25 MB. For the other containers which run a script inside the container and Mongo database, the checkpoint image size varies from 33 MB to 42 MB.

Figure 6.7(b) shows the checkpointing time of the services. The checkpointing time varies from 0.3 s to 2.0 s depending on the service. We observe that the checkpointing time is largely proportional to the size of the checkpoint image.

6.5.3 Boot phase time

Table 6.2 compares the boot phase time of the Edge-sharelatex services while deploying with standard Docker and with container restart. We observe that the service boot time with standard Docker varies from 0.1 s to 109.0 s, whereas with the proposed model, this range is from 0.1 s to 1.8 s. The gains of eliminating the boot phase in the proposed model however depend on the type of the containers and the complexity of their boot phase. We also observe similarities among the services: for example, the lightweight services such as redis, real-time, updater take negligible amount of time (less than 1 s) to boot. Deploying such containers with DMTCP takes almost the same amount of time. However, other services such as notifications, chats and filestore takes significant time to boot, between 5 and 20 s. The significant boot time delay of such services is due to the fact that they deploy a Mongo database inside the container in addition to their other software. Deploying such services with the proposed model takes only about 1 s, which brings a speedup in the range of 5x to 8x. Finally, the *web* service takes nearly 109 s to boot its container. This is mainly due to the fact that it compiles some of its scripts before becoming ready to serve end users. In this case, our system

Table 6.2 – Boot phase time of the services with standard Docker and the proposed model.

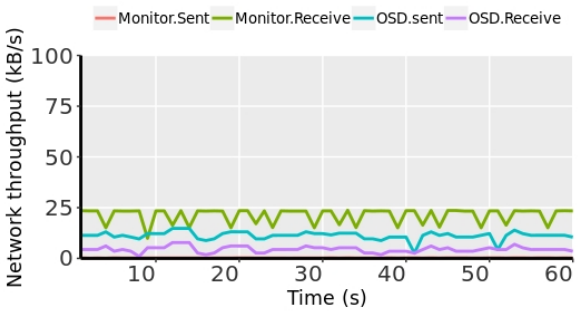
Service	Boot phase time			Snapshot size
	Docker	Proposed	Gains	
redis	0.1	0.1	1x	5
real-time	0.4	0.4	1x	9
updater	0.6	0.5	1.05x	11
notifications	5.2	0.8	7x	35
chats	5.2	0.9	6x	37
filestore	5.3	0.9	6x	33
spelling	5.7	1.0	6x	38
docstore	6.6	0.9	7x	33
tags	6.9	0.8	9x	34
contacts	7.3	1.0	7x	33
tracker	8.3	1.1	8x	39
Mongo	12.1	1.5	8x	25
clsi	20.2	1.5	13x	42
web	109.0	1.8	60x	35

delivers 60x improvement. We can conclude from this experiment that the gain brought by the proposed model is largely proportional to the time the container takes to boot. The deployment time from a container snapshot depends on the size of its data rather than the procedure to boot it.

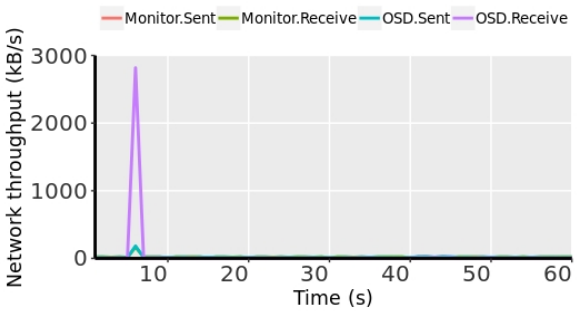
6.5.4 Communication within the Ceph cluster

Our system stores the container environments and checkpoint images in Ceph block devices. The system's various Ceph components such as OSDs, monitor and manager running on different nodes constantly communicate with each other to keep the cluster working. To evaluate the impact of transferred data between the Ceph components, we trace the download and upload network throughput of the cluster nodes.

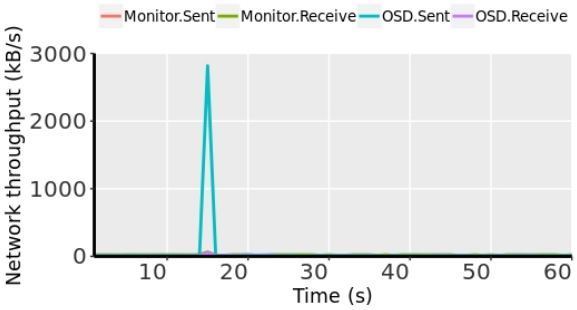
We use *nethogs* utility to capture the network throughput for duration of 60 s in 1 s granularity [63]. We capture the network throughput of the machine hosting the monitor+manager, and one which is running an OSD. We did the experiment in three scenarios: first, when the system is in idle condition; and second, when DMTCP is checkpointing the *redis* container; and third, when DMTCP is restarting a container.



(a) System is idle.



(b) System is checkpointing (with large scale Y-axis).



(c) System is restarting (with large scale Y-axis).

Figure 6.8 – Network throughput of the nodes when the system is: (a) idle; (b) checkpointing and (c) restarting.

Figure 6.8(a) shows the network throughput of the target machines when the system is in idle state. We observe that both machines (monitor+manager and OSD) exhibit constant low-bandwidth network activities throughout the trace. This may be due to the fact that the monitor+manager periodically communicates with its OSDs to control and monitor the cluster state. The upload throughput of the monitor+manager node is negligible, whereas the download throughput reaches 24 kB/s. The OSD node exhibits both download and upload activities, though the download throughput is always lower than the upload throughput. The reason for this is that each OSD sends own information and about the stored objects to other OSDs and to the monitor+manager.

Figure 6.8(b) represents the network throughput of the target machines when the system is checkpointing a container. We observe the same phenomenon except at the time when the system checkpoints the container. During the checkpoint (at t = 6 s), we observe an intense network activity, in particular with the download throughput of the OSD. The download throughput of the OSD goes nearly up to 2900 kB/s. This is due to the fact during the checkpoint, the system writes large chunks of the checkpoint image in that OSD.

Figure 6.8(c) represents the network throughput of the target machines when the system is restarting a container from the checkpoint image. During the restart (at t = 15 s), we observe an intense upload throughout in the OSD node. This reflects the system reads the checkpoint image from the Ceph OSD while restarting the container.

Finally, we can conclude from this study that Ceph cluster components (i.e., monitor+manager and OSDs) perform very little network activities to keep the cluster alive. But we observe intense network activity in the OSD nodes when the system checkpoints (to write the checkpoint image) and restarting a container(to read the checkpoint image).

6.5.5 Interference with other applications

Sharing container environments in the Ceph distributed storage clearly has many advantages. However, Docker needs to fetch the application files remotely from the distributed storage, which may in turn impact the performance of other running applications. We therefore study the runtime performance of the applications when they are deployed with the standard Docker and the proposed system.

Table 6.3 – Throughput of the HTTP service in the standard Docker and proposed system.

Concurrency level	10	50	100	200	300	400	500	1000
Standard (req/s)	3668	3703	3834	3905	3973	4057	4117	4234
Proposed (req/s)	3537	3569	3685	3743	3791	3856	3891	3954
Overhead (%)	3.6	3.7	3.8	4.1	4.6	5.0	5.5	6.7

We deploy an Apache HTTP server in one cluster node [69]. The server hosts a web page of 5 kB. We then generate an artificial HTTP load from one of the cluster node using the standard `ab` benchmarking tool with varying concurrency requests [70]. The load generates intense I/O activities in the server (as no file system cache was employed). At the end of each experiment, we measure the throughput of the HTTP server i.e., number of requests served per second. Table 6.3 compares the throughput of the HTTP server in both scenarios with different concurrency levels. We observe a small overhead of 3.6% with a concurrency level 10. This is expected due to the fact the HTTP load generates intensive I/O operations and as a result the proposed system may need to fetch the files from the distributed storage. When we increase the concurrency level, the overhead due to remote access grows gradually. Finally, at concurrency level 1000, this overhead reaches 6.7%. This shows that the communication between Docker and Ceph clusters creates small overhead in the application runtime. We can conclude from this study that the proposed system may exhibit an improved boot phase time with lower overhead. This is due to the distributed storage used to store the container environments.

6.6 Conclusion

Booting a Docker container after it has been started requires significant time during the container deployment process. This delay may have important impact in fog computing environments, since the same container may be repeatedly launched, created, booted. The boot sequence of most containers however always remain the same. This gives us an opportunity to eliminate the boot phase by saving the state of a fully-booted container. In subsequent deployments, the container can be restarted from the saved state, which wholly skips the boot phase in the deployment process.

We proposed a Docker container deployment system that uses DMTCP checkpoint/restart to deploy containers from a checkpoint image. The design also stores the container environments and checkpoint images in Ceph distributed storage to efficiently share them across the servers of a PoP. The performance evaluation shows it can bring up to 60x speedup in the container boot phase time depending on the type of the container. The overhead of creating checkpoints and storing them across the PoP remain reasonable.

CHAPTER 7

CONCLUSION AND FUTURE WORK

This chapter presents the conclusion of the thesis. We briefly remind the importance of application deployment time in distributed fog infrastructures. We then summarize the different contributions of the thesis to improve the application deployment time. Finally, we highlight the number of directions that we may study in the future to further reduce the application deployment time in distributed fog infrastructures.

7.1 Conclusion

Cloud computing architectures consist of large number of powerful servers connected to each other and to the rest of the Internet with high-speed network links. The latency between a typical end user and the closest cloud data center comes in the range of 20-40 ms over wired networks, and up to 150 ms over 4G mobile networks. Although this latency is acceptable for many applications, it creates many challenges for certain types of applications: for example, latency-sensitive applications like augmented reality games require an end-to-end latency including network and processing delay under 10-20 ms. Another example of such applications is IoT data analysis. The growing number of IoT devices produce large amounts of data every day. The collected data is typically sent to the core cloud for further analysis, which consumes large amount of global Internet traffic. An obvious solution to address these challenges is to host applications closer to the end users. Fog computing therefore extends the cloud resources (compute, storage and network) by broadly distributing large numbers of compute nodes near the end users. Therefore, computational capacity is always available in the vicinity of the users.

Fog computing architectures consist of large number of computing nodes dispersed across a geographical area such as a city, a region or even a country to maintain proximity with a large number of users. As a consequence, fog resources are often orga-

nized in a large number of Point-of-Presence (PoPs), where each PoP is composed of a small number of weak machines such as single-board computers connected to each other and to the rest of the Internet using heterogeneous networks. An end user always accesses the applications from the closest PoP to maintain minimal latency.

We expect that fog applications will be repeatedly deployed in different PoPs: to maintain minimum latency between the applications hosted in the fog and their end users, applications may need to roam frequently from one PoP to another. Human mobility remains far from being random, and it has been proven to be predictable despite important differences between individual travel patterns. Fog applications that aims at serving a single user with ultra-low latency, such as wearable cognitive assistance, may therefore be repeatedly deployed in the same PoP the user visits often (home, work, etc.). In another example, compute-intensive applications such as live video feed analysis may need to deploy multiple identical instances in the same PoP in order to horizontally scale their processing capacity. In these scenarios, the application deployment process cannot be considered as a one-time operation that does not affect the end-user's quality of experience. Rather, it becomes an integral part of the critical path towards providing the expected service to its end users.

Slow application deployment is therefore a challenging issue in fog infrastructures. Any delay in the application deployment may force the user to wait until the application has been fully deployed and is ready to serve users. When the user moves from one PoP to another, the application may have to be re-deployed to maintain proximity, low latency, and reduce long-distance traffic. In such cases, any delay in the application deployment may interrupt the already-running service, leading to a degradation of the user's Quality-of-Experience (QoE). In both scenarios, a minimal application deployment time is essential to provide seamless cloud services to the end-users. This thesis therefore aims to reduce the application deployment time of fog applications as much as possible.

We studied the reasons behind the slow deployment time of Docker containers in distributed fog infrastructures, and identified three opportunities that are likely to speed up the container deployment time: (1) improving the hit ratio of the Docker cache, which reduces the chances of having to pull a new image; (2) speeding up the image pull operation itself; and (3) speeding up the boot process after a container has been started. We therefore proposed three different solutions to optimize the overall application de-

ployment time. Each solution aims to address one of the above issues within the deployment process.

7.1.1 Contribution 1: Improving the Docker cache hit ratio

The first contribution of the thesis is to improve the application deployment time by reducing the probability of having to deploy new images upon container deployment requests. Docker servers download an image from a registry whenever they find that the required image is missing in the local cache. Docker stores all the downloaded images in its local cache and never removes them until explicitly asked to do so. This is a sensible strategy in powerful servers as the same container image will not need to be downloaded again in future deployment of the same container. However, this scenario is not suitable in fog environments, as fog servers have limited storage capacity. As a consequence, the working set of images may grow larger than the total storage capacity of the server. Another issue is that the image caches of the co-located nodes may contain redundant copies of the same image.

We proposed a new Docker image-sharing framework which aggregates the image caches of co-located fog servers using a distributed file system. The end result is a much larger Docker image cache that can share more images, which reduces the probability of deploying a new image upon a new container deployment request. Our performance evaluation of the proposed framework using a real-world Docker registry workload shows that sharing the Docker images can significantly improve the hit ratio and, as a result, reduce container deployment time between 37% and 78% depending on the scenario.

7.1.2 Contribution 2: Improving the Docker image deployment

Sharing Docker images among co-located servers enhances the Docker cache hit ratio and reduces the probability of pulling an image upon a container deployment request. However, Docker still needs to deploy an image when it is requested for the first time in a PoP or upon a cache miss. Docker image deployment can be very slow, on the order of a couple of minutes in resource-constrained fog nodes such as single-board Raspberry Pi. We investigated the reason behind this slow deployment by analyzing the resource consumption of Docker upon image deployment. We found that this slow

deployment time is largely due to the fact that Docker under-utilizes the available hardware resources during deployment: Docker first downloads the different image layers simultaneously which is very network-intensive, followed by a cycle of CPU-intensive decompression and then disk-intensive extraction. In other words, there is little or no overlapping among the usage of different hardware resources during the image deployment.

We proposed three optimizations to improve the resource utilization of Docker during image deployment: (1) sequentially downloading the image layers to optimize layer download time; (2) multi-threaded decompression to reduce the decompression time of layers; and (3) I/O pipelining to decompress the layers immediately after the first few bytes have been downloaded. Docker-pi combines all these solutions and therefore parallelizes the usage of the three hardware resource (network, CPU and disk), resulting in reducing the image deployment time by 25% to 75% in Raspberry Pis depending on the network capacity and the image size.

7.1.3 Contribution 3: Avoiding the container boot phase

After creating a container, Docker starts the boot phase by launching the starting process of the application. Booting terminates when the container is ready to accept end user requests. This phase may have a significant impact in fog environments when the same container image is being repeatedly launched, created, and booted in multiple servers of a PoP. The boot phase of containers however remains the same every time. We can therefore save the state of a container after completing its boot phase and then later restart the container from the saved state in the subsequent deployments.

We proposed a new container deployment design which uses DMTCP to deploy the container from a booted checkpoint image, therefore skipping the container boot phase. The design uses Ceph distributed storage to store container environments and checkpoint images to efficiently share them across fog servers. Our evaluation shows that this technique improves the container boot phase time up to 60x depending on the type of container. The checkpointing overhead of the proposed system remains reasonable.

7.2 Future directions

We presented several optimization solutions to address optimization opportunities in the Docker container deployment in distributed fog infrastructures. Our solutions create further opportunities that can help to reduce the container deployment time. In this section, we discuss several potential research directions brought forth by this thesis.

7.2.1 Finding a better cache replacement algorithm

The proposed Docker image sharing framework incorporates an image replacement algorithm to remove unused Docker images when the size of the working set images is larger than the storage capacity. In our contribution, we used the well-known least recently used (LRU) algorithm which replaces the least recently used image from the working set of images [195]. However LRU relies on the assumption that all cached objects have the same size (and therefore generates the same storage costs), and that all objects incur the same download delay in case of a cache miss. These two hypotheses are clearly not necessarily correct in the context of Docker images. We may therefore study how the choice of a better cache replacement algorithm may further reduce the average container deployment times. The field of Web caching has produced numerous such algorithms which may be used as a starting point [156].

7.2.2 Image layer placement in the distributed file system

The proposed Docker image sharing framework uses CephFS to store the image layers and metadata. While deploying an image, CephFS stores the image layers and metadata in the available OSDs. Like most other systems, Ceph distributes the objects and workload in its different OSDs to efficiently utilize all the available resources, while facilitating the system scale and managing hardware failures. In particular, Ceph uses a user-defined algorithm or CRUSH to distribute its objects and replication into multiple OSDs [192]. The write throughput of Ceph is however determined by how the CRUSH algorithm is configured. In fog environment, it is very likely that computation and communication capacity of the servers are heterogeneous. Therefore, a new CRUSH algorithm may need to adapt the fog characteristics, i.e., incorporate the location of the servers and their communication so that the write throughput of the objects

is maximized [203]. This may eventually improve the overall disk write throughput of the Ceph cluster, and as a consequence, speed up Docker image deployment.

7.2.3 Pre-fetching Docker images

Fog users are often mobile, and their mobility pattern are far from being random. They are often repetitive and might be predictable as most people often visit the same place every day [25, 175]. For example, researchers often take the same route while coming to the office every day. Similarly, users in fog environments are expected to move from one fog PoP to another, while accessing their favorite fog application. Therefore, the applications deployment would be repetitive and predictable.

Docker container deployment time could be improved if we can predict the user mobility. We may use machine learning and stochastic process to predict a human behavior and in particular the mobility pattern of the fog users [138, 184]. If we can predict the probability of approaching a fog user in a PoP, then the container image can be pre-fetched before the user even launches the application in the PoP. This will avoid pulling the image from the registry after its container deployment request.

REFERENCES

[1] M. Abrash, *Latency – The Sine Qua Non of AR and VR*, http://blogs.valvesoftware.com/abrash/latency-the-sine-qua-non-of-ar-and-vr/, 2012.

[2] X. M. Aguilera et al., « Managed Containers: A Framework for Resilient Ccontainerized Mission Critical Systems », *in*: *Proc. IEEE CLOUD*, 2018.

[3] A. Ahmed, *Docker-pi*, https://gitlab.com/aahmed/docker-pi-v18.06.git, 2018.

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

[5] A. Ahmed and G. Pierre, « Docker Image Sharing in Distributed Fog Infrastructures », *in*: *Proc. IEEE CloudCom*, 2019.

[6] A. Ahmed et al., « Docker Container Deployment in Distributed Fog Infrastructures with Checkpoint/Restart », *in*: *Proc. IEEE Mobile Cloud*, 2020.

[7] A. Ahmed et al., « Fog Computing Applications: Taxonomy and Requirements », *in*: *CoRR* abs/1907.11621 (2019), URL: http://arxiv.org/abs/1907.11621.

[8] C. D. Alfonso et al., « Container-based virtual elastic clusters », *in*: *Journal of Systems and Software* 127 (2017).

[9] C. Anderson, « Docker [Software engineering] », *in*: *IEEE Software* 32.*3* (2015).

[10] J. Ansel et al., « DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop », *in*: *Proc. IEEE IPDPS*, 2009.

[11] A. Antonio et al., « Exploring Container Virtualization in IoT Clouds », *in*: *Proc. IEEE SMARTCOMP*, 2016.

[12] A. Anwar et al., « Improving Docker Registry Design Based on Production Workload Analysis », *in*: *Proc. Usenix FAST*, 2018.

[13] Apache Software Foundation, *Apache Hadoop*, http://hadoop.apache.org/, 2019.

[14] Standard Association, *IEEE 1934-2018 -IEEE Standard for Adoption of Openfog Reference Architecture for Fog Computing*, https://standards.ieee.org/standard/1934-2018.html, 2017.

[15] H. Atlam et al., « Fog Computing and the Internet of Things: a Review », *in*: *Big Data and Cognitive Computing* 2.*2* (2018).

[16] The Go Authors, *Source File src/compress/gzip/gunzip.go*, https://golang.org/src/compress/gzip/gunzip.go, 2017.

[17] The Go Authors, *Source File src/io/pipe.go*, https://golang.org/pkg/io/Pipe, 2018.

[18] The Go Authors, *The Go Programming Language*, https://golang.org/, 2019.

[19] The Kubernetes Authors, *Production-Grade Container Orchestration - Kubernetes*, https://kubernetes.io, 2019.

[20] Authors of Checkpointing.org, *Checkpointing.org - The home to checkpointing packages*, https://checkpointing.org/, 2019.

[21] I. Babrou, *Docker registry bay*, https://github.com/bobrik/bay, 2016.

[22] D. Balouek et al., « Adding Virtualization Capabilities to the Grid'5000 Testbed », *in*: *Cloud Computing and Services Science*, vol. 367, Springer, 2013.

[23] P. Basford et al., « Performance Analysis of Single Board Computer Clusters », *in*: *Future Generation Systems* (2019).

[24] P. Bellavista et al., « Feasibility of Fog Computing Deployment based on Docker Containerization over RaspberryPi », *in*: *Proc. ACM ICDCN*, 2019.

[25] P. Bellavista et al., « Human-enabled Edge Computing: Exploiting the Crowd as a Dynamic Extension of Mobile Edge Computing », *in*: *IEEE Communications Magazine* 56.*1* (2018).

[26] R. Benevides, *10 things to avoid in docker containers - Red Hat Developer*, https://blog.codeship.com/using-docker-commit-to-create-and-change-an-image/, 2019.

[27] T. Binz et al., « Portable Cloud Services Using Tosca », *in*: *IEEE Internet Computing* 16.*3* (2012).

[28] R. Birke et al., « State-of-the-practice in Data Center Virtualization: Toward a Better Understanding of VM usage », *in*: *Proc. IEEE DSN*, 2013.

[29] L.F. Bittencourt and othersdi, « Mobility-Aware Application Scheduling in Fog Computing », *in*: *IEEE Cloud Computing* 4.*2* (2017).

[30] OculusRift Blog, *Delivers Some Home Truths On Latency | Oculus Rift Blog*, `https://oculusrift-blog.com/john-carmacks-message-of-latency/682/`.

[31] F. Bonomi et al., « Fog computing: A platform for internet of things and analytics », *in*: *Big data and internet of things: A roadmap for smart environments*, Springer, 2014.

[32] F. Bonomi et al., « Fog Computing and its Role in the Internet of things », *in*: *Proc. ACM MCC*, 2012.

[33] D. Bornstein, « Dalvik VM Internals », *in*: *Google I/O Developer Conference*, 2008.

[34] A. Brogi et al., « How to Best Deploy your Fog Applications, probably », *in*: *Proc. IEEE ICFEC*, 2017.

[35] B. Caldwell, *Running Docker on Lustre*, `http://wiki.lustre.org/images/f/ff/LUG2016D2_An-Architechture-on-Docker_Caldwell.pdf`, 2016.

[36] D. Catteddu, « Cloud Computing: Benefits, Risks and Recommendations for Information Security », *in*: *Proc. Iberic Web Application Security Conference*, Springer, 2009.

[37] M. Chen et al., « Edge Cognitive Computing Based Smart Healthcare System », *in*: *Future Generation Computer Systems* 86 (2018).

[38] X. Chen, « Decentralized Computation Offloading Game for Mobile Cloud Computing », *in*: *IEEE Transactions on Parallel and Distributed Systems* 26.*4* (2014).

[39] X. Chen et al., « Efficient Multi-user Computation Offloading For Mobile-edge Cloud Computing », *in*: *IEEE/ACM Transactions on Networking* 24.*5* (2015).

[40] B. Cheng et al., « Geelytics: Enabling On-demand Edge Analytics Over Scoped Data Sources », *in*: *Proc. IEEE BigData*, 2016.

[41] P. Christensson, *Container Definition*, `https://techterms.com/definition/container`, 2017.

[42] Cisco Inc., *Internet of Things (IoT) Data Continues to Explode Exponentially. Who Is Using That Data and How? - Cisco Blog*, https://blogs.cisco.com/datacenter/internet-of-things-iot-data-continues-to-explode-exponentially-who-is-using-that-data-and-how, 2019.

[43] L. Civolani et al., « FogDocker: Start Container Now, Fetch Image Later », *in*: *Proc. IEEE/ACM UCC*, 2019.

[44] CLAudit Project, *Planetary-scale Cloud Latency Auditing Platform*, http://bit.do/bS4js, 2016.

[45] L. Columbus, *83% Of Enterprise Workloads Will Be In The Cloud By 2020*, https://www.forbes.com/sites/louiscolumbus/2018/01/07/83-of-enterprise-workloads-will-be-in-the-cloud-by-2020/, 2017.

[46] B. Confais et al., « Performance Analysis of Object Store Systems in a Fog/Edge Computing Infrastructures », *in*: *Proc. IEEE CloudCom*, 2016.

[47] iRODS Consortium, *iRODS*, https://irods.org/, 2019.

[48] L. Copeland, *Checkpoint and Restart | Computerworld*, https://www.computerworld.com/article/2588055/checkpoint-and-restart.html, 2002.

[49] Core Technology, *MooseFS Distributed File System*, https://moosefs.com/, 2019.

[50] S. J. Cox et al., « Iridis-pi: a Low-cost, Compact Demonstration cluster », *in*: *luster Computing* 17.*2* (2014).

[51] J. Darrous et al., « Nitro: Network-Aware Virtual Machine Image Management in Geo-Distributed Clouds », *in*: *Proc. IEEE/ACM CCGrid*, 2018.

[52] J. Darrous et al., « On the Importance of Container Image Placement for Service Provisioning in the Edge », *in*: *Proc. ICCCN*, 2019.

[53] A. V. Dastjerdi et al., « Fog computing: Principles, Architectures, and Applications », *in*: *Internet of things*, Elsevier, 2016.

[54] B. Depardon et al., *Analysis of Six Distributed File Systems*, Research report, https://hal.inria.fr/hal-00789086, 2016.

[55] G. Diener, *How to Build a Smaller Docker Image – Gabriele Diener – Medium*, https://medium.com/@gdiener/how-to-build-a-smaller-docker-image-76779e18d48a, 2019.

[56] Docker Inc., *Docker Hub*, https://hub.docker.com/, 2019.

[57] L. Du et al., « Cider: A Rapid Docker Container Deployment System Through Sharing Network Storage », *in*: *Proc. IEEE HPCC*, 2017.

[58] J. C. Duell, *Berkeley Lab Checkpoint/Restart (BLCR) for LINUX*, https://crd.lbl.gov/departments/computer-science/CLaSS/research/BLCR/, 2019.

[59] J. C. Duell et al., *Berkeley Lab for Linux*, tech. rep. BLCR; 002078WKSTN00, Lawrence Berkeley National Laboratory, 2003.

[60] J. Eder, *Comprehensive Overview of Storage Scalability in Docker - Red Hat Developer Blog*, https://developers.redhat.com/blog/2014/09/30/overview-storage-scalability-docker/, 2014.

[61] Y. Elkhatib et al., « On Using Micro-Clouds to Deliver the Fog », *in*: *IEEE Internet Computing* 21.*2* (2017).

[62] J. Elson and J. Howell, « Handling Flash Crowds from Your Garage. », *in*: *Proc. USENIX ATC*, 2008.

[63] A. Engelen, *Raboof/nethogs: Linux 'net top' Tool*, https://github.com/raboof/nethogs, 2017.

[64] ETSI, *Mobile-Edge Computing - ETSI Portal*, https://portal.etsi.org/, 2019.

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

[66] J. O. Fajardo et al., « Improving Content Delivery Efficiency Through Multi-layer Mobile Edge Adaptation », *in*: *IEEE Network* 29.*6* (2015).

[67] I. Foster et al., « Cloud Computing and Grid Computing 360-Degree Compared », *in*: *Proc. Grid Computing Environments Workshop*, 2008.

[68] Apache Software Foundation, *OpenLambda · GitHub*, https://github.com/open-lambda, 2019.

[69] Apache Software Foundation, *Welcome! - The Apache HTTP Server Project*, https://httpd.apache.org/, 2018.

[70] The Apache Software Foundation, *ab - Apache HTTP server benchmarking tool - Apache HTTP Server Version 2.4*, https://httpd.apache.org/docs/2.4/programs/ab.html, 2019.

[71] R. Garg et al., « Checkpoint-restart for a Network of Virtual Machines », *in*: *Proc. IEEE CLUSTER*, 2013.

[72] S. Gargand et al., « A Framework for Ranking of Cloud Computing Services », *in*: *Future Generation Computer Systems* 29.*4* (2013).

[73] G. Goodson et al., *System and Method for Fast Restart of a Guest Operating System in a Virtual Machine Environment*, US Patent 8.006.079, 2011.

[74] K. Ha et al., « Towards Wearable Cognitive Assistance », *in*: *Proc. ACM MobiSys*, 2014.

[75] W. Hajji and F. P. Tso, « Understanding the Performance of Low Power Raspberry Pi Cloud for Big Data », *in*: *Electronics* 5.*2* (2016).

[76] T. Harter et al., « Slacker: Fast Distribution with Lazy Docker Containers », *in*: *Proc. Usenix FAST*, 2016.

[77] J. C. Ho et al., « Scalable Group-based Checkpoint/restart for Large-scale Message-passing Systems », *in*: *Proc. IEEE ISPDP*, 2008.

[78] C. Hong and B. Varghese, « Resource Management in Fog/Edge Computing: A Survey on Architectures, Infrastructure, and Algorithms », *in*: *ACM Computing Surveys* 52.*5* (2019).

[79] K. Hong et al., « Mobile Fog: A Programming Model for Large-scale Applications on the Internet of Things », *in*: *Proc. ACM SIGCOMM workshop on Mobile Cloud Computing*, 2013.

[80] S. Hoque et al., « Towards Container Orchestration in Fog Computing Infrastructures », *in*: *Proc. IEEE COMPSAC*, 2019.

[81] Z. Huang et al., « FastBuild: Accelerating Docker Image Building for Efficient Development and Deployment of Container », *in*: *Proc. MSST*, 2019.

[82] J. Hursey et al., « Checkpoint/restart-enabled Parallel Debugging », *in*: *Proc. European MPI Users' Group Meeting*, 2010.

[83] J. Hursey et al., « The Design and Implementation of Checkpoint/restart Process Fault Tolerance for Open MPI », *in*: *Proc. IEEE IPDPS*, 2007.

[84] Amazon Inc., *Amazon Web Services (AWS) - Cloud Computing Services*, `https://aws.amazon.com/`, 2019.

[85] Cisco Inc., *Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are*, https://www.cisco.com/c/dam/en_us/solutions/trends/iot/docs/computing-overview.pdf, 2015.

[86] Docker Inc., *| Docker Documentation*, https://docs.docker.com/engine/reference/builder/, 2019.

[87] Docker Inc., *About Images, Containers, and Storage Drivers | Docker Documentation*, https://docs.docker.com/v17.09/engine/userguide/storagedriver/imagesandcontainers/, 2019.

[88] Docker Inc., *Best Practices for Writing Dockerfiles | Docker Documentation*, https://docs.docker.com/develop/develop-images/dockerfile_best-practices/, 2019.

[89] Docker Inc., *Develop with Docker Engine SDKs and API | Docker Documentation*, https://docs.docker.com/develop/sdk/, 2019.

[90] Docker Inc., *Docker 0.9: Introducing Execution Drivers and Libcontainer - Docker Blog*, https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/, 2019.

[91] Docker Inc., *Docker: Build, Ship, and Run Any App, Anywhere*, https://www.docker.com/, 2019.

[92] Docker Inc., *Docker Commit | Docker Documentation*, https://docs.docker.com/engine/reference/commandline/commit/, 2019.

[93] Docker Inc., *Docker Overview | Docker Documentation*, https://docs.docker.com/engine/docker-overview/, 2019.

[94] Docker Inc., *Docker Registry | Docker Documentation*, https://docs.docker.com/registry/, 2019.

[95] Docker Inc., *Docker System Prune | Docker Documentation*, https://docs.docker.com/engine/reference/commandline/system_prune/, 2019.

[96] Docker Inc., *Dockerfile Reference*, https://docs.docker.com/engine/reference/builder/dockerignore-file, 2019.

[97] Docker Inc., *GitHub - Docker/libcontainer*, https://github.com/docker/libcontainer, 2019.

[98] Docker Inc., *Image Manifest V 2, Schema 2 | Docker Documentation*, https://docs.docker.com/registry/spec/manifest-v2-2/, 2019.

[99] Docker Inc., *Moby - Github*, https://github.com/moby/moby, 2019.

[100] Docker Inc., *Prune Unused Docker Objects*, https://docs.docker.com/config/pruning/, 2019.

[101] Docker Inc., *Select a Storage Driver | Docker Documentation*, https://docs.docker.com/v17.09/engine/userguide/storagedriver/selectadriver/, 2019.

[102] Docker Inc., *Swarm Mode Overview | Docker Documentation*, https://docs.docker.com/engine/swarm/, 2019.

[103] Docker Inc., *Use the AUFS Storage Sriver | Docker Documentation*, https://docs.docker.com/storage/storagedriver/aufs-driver/, 2019.

[104] Docker Inc., *Use the Docker Command Line | Docker Documentation*, https://docs.docker.com/engine/reference/commandline/cli/, 2019.

[105] Gluster Inc., « Cloud Storage for the Modern Data Center – An Introduction to Gluster Architecture », http://moo.nac.uci.edu/~hjm/fs/An_Introduction_To_Gluster_ArchitectureV7_110708.pdf, 2011.

[106] Google Inc., *Google Cloud: Cloud Computing Services*, https://cloud.google.com/, 2019.

[107] Inktank Storage Inc., *Rbd(8): Manage Rados Block Device Images - Linux Man Page*, https://linux.die.net/man/8/rbdc, 2019.

[108] Linux Foundation Inc., *Linux Containers*, https://linuxcontainers.org/, 2019.

[109] Mesosphere Inc., *Marathon: A Container Orchestration Platform for Mesos and DC/OS*, https://mesosphere.github.io/marathon/, 2019.

[110] Microsoft Inc., *Microsoft Azure Cloud Computing Platform & Services*, https://azure.microsoft.com/, 2019.

[111] Microsoft Inc., *Open Edge Computing Initiative*, https://www.openedgecomputing.org, 2019.

[112] Red Hat Inc., *Ceph Block Device —- Ceph Documentation*, https://docs.ceph.com/docs/master/rbd/, 2019.

[113] Red Hat Inc, *CephFS Best Practices — Ceph Documentation*, http://docs.ceph.com/docs/mimic/cephfs/best-practices/, 2019.

[114] Red Hat Inc., *CRUSH Maps - Ceph Documentation*, http://docs.ceph.com/docs/jewel/rados/operations/crush-map/, 2019.

[115] Red Hat Inc., *Gluster | Storage for Your Cloud*, https://www.gluster.org/, 2019.

[116] Red Hat Inc., *Librbd (Python) —- Ceph Documentation*, https://docs.ceph.com/docs/jewel/rbd/librbdpy/, 2019.

[117] Red Hat Inc., *RBD Mirroring —- Ceph Documentation*, https://docs.ceph.com/docs/master/rbd/rbd-mirroring/, 2019.

[118] Red Hat Inc., *RBD Persistent Cache —- Ceph Documentation*, https://docs.ceph.com/docs/master/rbd/rbd-persistent-cache/, 2019.

[119] Red Hat Inc., *Rkt, A Security-minded, Standards-based Container Engine*, https://coreos.com/rkt/, 2019.

[120] Red Hat Inc., *Snapshots —- Ceph Documentation*, https://docs.ceph.com/docs/master/rbd/rbd-snapshot/, 2019.

[121] K. Arya and others, *DMTCP : Distributed MultiThreaded Checkpointing*, http://dmtcp.sourceforge.net/, 2019.

[122] W. Kangjin et al., « FID: A Faster Image Distribution System for Docker Platform », *in*: *Proc. IEEE FAS* W*, 2017.

[123] A. van Kempen et al., « MEC-ConPaaS: An Experimental Single-board Based Mobile Edge Cloud », *in*: *Proc. IEEE Mobile Cloud*, 2017.

[124] M. Kerrisk, *Fuse(4) - Linux Manual Page*, http://man7.org/linux/man-pages/man4/fuse.4.html, 2019.

[125] ACME Laboratories, *Http-load*, https://acme.com/software/http_load/, 2018.

[126] Open Fog Consortium Laboratories, *Patient Monitoring*, https://www.openfogconsortium.org/wp-content/uploads/Patient-Mo, 2019.

[127] Redis Labs, *Redis Pub/Sub*, https://redis.io/topics/pubsub, 2019.

[128] Redis Labs, *Redis Transactions*, https://redis.io/topics/transactions, 2019.

[129] M. Litzkow, *Checkpoint and Migration of UNIX Processes in the Condor Distributed Processing System*, Computer Sciences Department, University of Wisconsin, 1997, URL: https://books.google.fr/books?id=6jGnPgAACAAJ.

[130] Th. Louati et al., « Lxcloud-cr: Towards Linux Containers Distributed Hash Table Based Checkpoint-restart », *in*: *Journal of Parallel and Distributed Computing* 111 (2018).

[131] Z. Lu et al., « An Empirical Case Study on the Temporary File Smell in Dockerfiles », *in*: *IEEE Access* 7 (2019).

[132] T. H. Luan et al., « Fog Computing: Focusing on Mobile Users at the Edge », *in*: *CoRR* abs/1502.01815 (2015).

[133] G. Ma et al., « Understanding Performance of Edge content Caching for Mobile Video Streaming », *in*: *Selected Areas in Communications* 35.*5* (2017).

[134] M. Ma et al., *Dockyard – Container And Artifact Repository*, https://github.com/Huawei/dockyard, 2017.

[135] B. Maharjan et al., « A Human-Centric Cloud and Fog-Assisted IoT Platform for Disaster Management », *in*: *Proc. IEEE MobileCloud*, 2019.

[136] R. Mahmud et al., « Fog Computing: A Taxonomy, Survey and Future Directions », *in*: *Internet of everything*, Springer, 2018.

[137] E. Marín-Tordera et al., « Do We All Really Know What a Fog Node is? Current Trends Towards an Open Definition », *in*: *Computer Communications* 109 (2017).

[138] W. Mathew et al., « Predicting Future Locations with Hidden Markov Models », *in*: *Proc. ACM conference on ubiquitous computing*, 2012.

[139] X. Meng et al., « Delay-Constrained Hybrid Computation Offloading With Cloud and Fog Computing », *in*: *IEEE Access* 5 (2017).

[140] D. Merkel, « Docker: Lightweight Linux Containers for Consistent Development and Deployment », *in*: *Linux Journal* 2014.*239* (2014).

[141] N. Mohamed et al., « UAVFog: A UAV-based Fog Computing for Internet of Things », *in*: *Proc. IEEE SmartWorld*, 2017.

[142] S. Nadgowda et al., « Comparing Scaling Methods for Linux Containers », *in*: *Proc. IEEE IC2E*, 2017.

[143] M. Nardelli et al., « Multi-Level Elastic Deployment of Containerized Applications in Geo-Distributed Environments », *in*: *Proc. IEEE FiCloud*, 2018.

[144] S. Nathan et al., « CoMICon: A Co-Operative Management System for Docker Container Images », *in*: *Proc. IEEE IC2E*, 2017.

[145] T. L. Nguyen et al., « YOLO: Speeding Up VM and Docker Boot Time by Reducing I/O Operations », *in*: *Proc. Europar*, 2019.

[146] B. Nicolae and F. Cappello, « BlobCR: Virtual Disk Based Checkpoint-restart for HPC Applications on IaaS Clouds », *in*: *Journal of Parallel and Distributed Computing* 73.*5* (2013).

[147] E. Oakes et al., « SOCK: Rapid Task Provisioning with Serverless-Optimized Containers », *in*: *Proc. USENIX ATC*, 2018.

[148] Open Fog Consortium, *Fog Computing Use Cases*, `https://www.openfogconsortium.org/resources/`, 2019.

[149] OpenFog Consortium, *OpenFog Reference Architecture for Fog Computing*, `https://www.openfogconsortium.org/ra/`, 2017.

[150] OpenSFS, *Lustre*, `http://lustre.org/`, 2019.

[151] OpenVZ team, *CRIU*, `https://www.criu.org/Main_Page`, 2019.

[152] OpenVZ team, *Docker External - CRIU*, `https://criu.org/Docker_External`, 2019.

[153] C. Pahl et al., « A Container-based Edge Cloud Paas Architecture Based on Raspberry Pi Clusters », *in*: *Proc. IEEE FiCloudW*, 2016.

[154] C. Pahl et al., « Containers and Clusters for Edge Cloud Architectures–A Technology Review », *in*: *Proc. IEEE FiCloud*, 2015.

[155] K. Parbat, *Cloud Computing Help Reduce Costs*, `https://economictimes.indiatimes.com/opinion/interviews/cloud-computing-help-reduce-costs/articleshow/4699359.cms`, 2009.

[156] S. Podlipnig and L. Böszörmenyi, « A Survey of Web Cache Replacement Strategies », *in*: *ACM Computing Surveys* 35.*4* (2003).

[157] K. Post, *klauspost/pgzip: Go parallel gzip (de)compression*, `https://github.com/klauspost/pgzip`, 2017.

[158] K. Quest, *DockerSlim - Minify Docker Image and Generate Security Profiles. Frictionless!*, https://dockersl.im/, 2019.

[159] A. Rauf et al., « Security and Privacy for IoT and Fog Computing Paradigm », *in*: *Proc. Learning and Technology Conference*, 2018.

[160] K. Razavi and T. Kielmann, « Scalable Virtual Machine Deployment Using VM Image Caches », *in*: *Proc. ACM/IEEE SC*, 2013.

[161] Inc Red Hat, *Chapter 1. What is the Ceph File System (CephFS)? Red Hat Ceph Storage 2 | Red Hat Customer Portal*, https://access.redhat.com/documentation/en-us/red_hat_ceph_storage/2/html/ceph_file_system_guide_technology_preview/what_is_the_ceph_file_system_cephfs, 2019.

[162] Red Hat Inc., *Ceph*, https://ceph.com/, 2019.

[163] Redis Labs, *Redis*, https://redis.io/, 2019.

[164] RightScale, *State of the Cloud Report*, https://www.rightscale.com/lp/state-of-the-cloud, 2019.

[165] O. Rodeh et al., « Btrfs: The Linux B-tree Filesystem », *in*: *ACM Transactions on Storage* 9.*3* (2013).

[166] D. Santoro et al., « Foggy: A Platform for Workload Orchestration in a Fog Computing Environment », *in*: *Proc. IEEE CloudCom*, 2017.

[167] M. Satyanarayanan et al., « Edge Analytics in the Internet of Things », *in*: *IEEE Pervasive Computing* 14.*2* (2015).

[168] M. Satyanarayanan et al., « The Case for VM-based Cloudlets in Mobile Computing », *in*: *Pervasive Computing 4* (2009).

[169] E. Saurez et al., « Incremental Deployment and Migration of Geo-distributed Situation Awareness Applications in the Fog », *in*: *Proc. ACM DEBS*, 2016.

[170] P. Sharma et al., « Containers and Virtual Machines at Scale: A Comparative Study », *in*: *Proc. ACM Middleware*, 2016.

[171] W. Shi et al., « Edge Computing: Vision and Challenges », *in*: *IEEE Internet of Things Journal* 3.*5* (2016).

[172] M. Shojafar et al., « FLAPS: Bandwidth and Delay-efficient Distributed Data Searching in Fog-supported P2P Content Delivery Networks », *in*: *The journal of supercomputing* 73.*12* (2017).

[173] E. Sitaridi et al., « Massively-Parallel Lossless Data Decompression », *in*: *Proc. ICPP*, 2016.

[174] J. M. Smith and J. Ioannidis, *Implementing Remote Fork () with Checkpoint/restart*, Department of Computer Science, Columbia University, 1987.

[175] C. Song et al., « Limits of Predictability in Human Mobility », *in*: *Science* 327 (2010).

[176] G. Stellner, « CoCheck: Checkpointing and Process Migration for MPI », *in*: *Proc. ICPP*, 1996.

[177] T. Sterling et al., *High Performance Computing: Modern Systems and Practices*, Morgan Kaufmann, 2017.

[178] J. Strickland, *Cloud Computing Architecture | HowStuffWorks*, https://computer.howstuffworks.com/cloud-computing/cloud-computing1.htm, 2019.

[179] A. Sullivan, *ThePub NetApp: Building A Distributed Docker Registry At Scale*, https://netapp.github.io/blog/2015/12/15/building-a-distributed-docker-registry-at-scale/, 2015.

[180] Ch. Swan, *Docker Drops LXC as Default Execution Environment*, https://www.infoq.com/news/2014/03/docker-0-9/, 2019.

[181] V. Tarasov et al., « Evaluating Docker Storage Performance: from Workloads to Graph Drivers », *in*: *Cluster Computing* (2019), ISSN: 1573-7543.

[182] G. Tato et al., « ShareLatex on the Edge: Evaluation of the Hybrid Core/Edge Deployment of a Microservices-based Application », *in*: *Proc. MECC*, 2018.

[183] *The Mobile Edge Cloud Testbed at IRISA Myriads team*, YouTube video, https://www.youtube.com/watch?v=7uLkLitiSPo, 2017.

[184] J. Tkačík and P. Kordík, « Neural Turing Machine for Sequential Learning of Human Mobility Patterns », *in*: *Proc. IEEE IJCNN*, 2016.

[185] K. Toczé and S. Nadjm-Tehrani, « ORCH: Distributed Orchestration Framework using Mobile Edge Devices », *in*: *Proc. IEEE ICFEC*, 2019.

[186] man7.org Training and Consulting, *Cgroups(7) - Linux manual page*, http://man7.org/linux/man-pages/man7/cgroups.7.html, 2019.

[187] man7.org Training and Consulting, *Namespaces(7) - Linux manual page*, http://man7.org/linux/man-pages/man7/namespaces.7.html, 2019.

[188] L. Vaquero, « Finding Your Way in the Fog: Towards a Comprehensive Definition of Fog Computing », *in*: *ACM Computer Communication Review* 44.*5* (2014).

[189] Virtuozzo and OpenVZ Community, *OpenVZ*, https://openvz.org/, 2019.

[190] M. Vögler et al., « LEONORE–Large-scale Provisioning of Resource-constrained IoT Deployments », *in*: *Proc. IEEE SOSE*, 2015.

[191] S. A. Weil et al., « Ceph: A Scalable, High-performance Distributed File System », *in*: *Proc. Usenix*, 2016.

[192] S. A. Weil et al., « CRUSH: Controlled, scalable, decentralized placement of replicated data », *in*: *Proc. ACM/IEEE SC*, 2006.

[193] Wikipedia, *aufs*, https://en.wikipedia.org/wiki/aufs, 2019.

[194] Wikipedia, *Btrfs*, https://en.wikipedia.org/wiki/Btrfs, 2019.

[195] Wikipedia, *Cache Replacement Policies*, https://en.wikipedia.org/wiki/Cache_replacement_policies, 2019.

[196] Wikipedia, *Comparison of Distributed File Systems*, https://en.wikipedia.org/wiki/Comparison_of_distributed_file_systems, 2019.

[197] Wikipedia, *Copy-on-Write*, https://en.wikipedia.org/wiki/Copy-on-write, 2019.

[198] Wikipedia, *Device Mapper*, https://en.wikipedia.org/wiki/Device_mapper, 2019.

[199] Wikipedia, *OverlayFS*, https://en.wikipedia.org/wiki/overlayfs, 2019.

[200] Wikipedia, *Single Board Computers*, https://en.wikipedia.org/wiki/Single-board_computer, 2019.

[201] Wikipedia, *UnionFS - Wikipedia*, https://en.wikipedia.org/wiki/UnionFS, 2019.

[202] C. Wöbker et al., « Fogernetes: Deployment and Management of Fog Computing Applications », *in*: *Proc. IEEE NOMS*, 2018.

[203] Ch. Wu et al., « File Placement Mechanisms for Improving Write Throughputs of Cloud Storage Services Based on Ceph and HDFS », *in*: *Proc. IEEE ICASI*, 2017.

[204] J. Xu et al., « Dockerfile TF Smell Detection Based on Dynamic and Static Analysis Methods », *in*: *Proc. IEEE COMPSAC*, 2019.

[205] E. Yigitoglu et al., « Foggy: A Framework for Continuous Automated IoT Application Deployment in Fog Computing », *in*: *Proc. IEEE AIMS*, 2017.

[206] C. Zheng et al., « Wharf: Sharing Docker Images in a Distributed File System », *in*: *Proc. ACM SOCC*, 2018.

[207] H. Zhong and J. Nieh, *CRAK: Linux Checkpoint/restart as a Kernel Module*, tech. rep. CUCS-014-01, Department of Computer Science, Columbia University, 2001.

[208] C. Zhu et al., « Vehicular Fog Computing for Video Crowdsourcing: Applications, Feasibility, and Challenges », *in*: *IEEE Communications Magazine*, vol. 56, 10, 2018.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre: Déploiement efficace d'applications cloud dans les infrastructures fog distribuées

Mot clés : fog computing; containers; Docker

Resumé : Les architectures Fog computing sont composées d'un grand nombre de machines dispersées dans une zone géographique telle qu'une ville ou une région. Dans ce contexte il est important de permettre un démarrage rapide des applications déployées sous forme de containers Docker. Cette thèse étudie les raisons de la lenteur de déploiement, et identifie trois opportunités susceptibles de réduire le temps de déploiement des conteneurs: (1) améliorer le taux de réussite du cache Docker; (2) accélérer l'opération d'installation d'une image; et (3) accélérer le processus de démarrage après la création d'un conteneur.

Title: Efficient Cloud Application Deployment in Distributed Fog Infrastructures

Keywords : fog computing; containers; Docker

Abstract : Fog computing architectures are composed of a large number of machines distributed across a geographical area such as a city or a region. In this context it is important to support a quick startup of applications deployed in the for of docker containers. This thesis explores the reasons for slow deployment and identifies three improvement opportunities: (1) improving the Docker cache hit rate; (2) speed-up the image installation operation; and (3) accelerate the application boot phase after the creation of a container.