PATRICK LINDNER

LAYERED TRANSFER: MAINTAINING SERVICE AVAILABILITY DURING CONTAINER VOLUME MIGRATIONS

[October 1, 2024 at 12:46]

LAYERED TRANSFER: MAINTAINING SERVICE AVAILABILITY DURING CONTAINER VOLUME MIGRATIONS

PATRICK LINDNER

Layered Transfer: Maintaining Service Availability during Container Volume Migrations

Distributed Computing
Faculty of Science and Engineering
University of Groningen

August 2024

ABSTRACT

Cloud computing is a great tool for delivering computing power as a utility. However, choosing for a specific cloud provider, might narrow down the deployment to the geographical locations and the servers of the cloud provider company. A migration to a different cloud provider mostly comes with great cost and downtime [15]. Technologies like Liqo provide a way of adding existing Kubernetes clusters from potentially different cloud providers to a single multi cluster, where stateless services can freely be migrated between clusters and therefore cloud providers [3]. However, the problem of live migrating stateful containers along with their attached volumes is still challenging. The aim of this thesis is to explore and improve current stateful-live-container-migration processes in order to provide the best quality of service and resource efficiency during a migration. The approaches are evaluated based on a collection of key performance indicators. This research focuses on two stateful container migration approaches from the literature and combines them into a novel method, seeking better performance. The combined approach is evaluated to be up to 30% less CPU consuming than the state of the art of live stateful container migration. In addition to that, for read heavy applications, the service latency can be reduced by up to 50%. This comes with the cost of more disk space consumption and for some scenarios a longer migration time. This extra disk space consumption and the additional migration time is attributed to the fact that the approach utilizes Copy-on-Write operations. However, these operations have potential to be replaced in future research.

ACKNOWLEDGMENTS

Personally, I would like to thank Edwin Harmsma and Coen van Leeuwen, which supported me throughout the journey of this research project. They provided valuable feedback and helped me in finding a relevant topic. In addition to that, I would like to thank my supervisors from the University of Groningen, Alexander Lazovik, and Vasilios Andrikopoulos for their feedback on this work. Furthermore, I would like to thank the TNO for providing office space, technical equipment and financial support for conducting the research. The opportunity to contribute with this research to the TNOs ECOFED Cloud Federation project provided me with extra motivation.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

SSH Secure Shell

SSHFS Secure Shell File System

CoW Copy-on-Write

HTTP Hyper Text Transfer Protocol

JSON JavaScript Object Notation

KPI Key Performance Indicator

AUFS Advanced Multi-layered Unification File System

UUIDv4 Universal Unique Identifyer Version 4

VM Virtual Machines

NIST National Institute of Standards and Technology

1 INTRODUCTION

Cloud computing is a great tool for delivering computing power as a utility. In comparison to traditional hosting, cloud computing providers only bill the time of actual processing. No large upfront investment for server and networking hardware is required. This potentially provides small companies and startups, with the equal amount of computing power as large companies with proprietary self-hosted hardware have. However, for most companies receiving cloud computing services from a cloud provider, their initial decision locks them in the environment of that provider and limits the location of data processing and storage to the physical servers of that provider. A migration to a different cloud provider is mostly connected to great cost and downtime [15].

The concept of a federated cloud tackles this problem. It introduces cloud interoperability features, that enable the application operator to move their service between different clouds for long and short term purpose [8]. The American National Institute of Standards and Technology (NIST) published a Cloud Federation Architecture Reference documentation, that defines the guiding principles for a cloud federated environment and its components [10]. According to this documentation, among others, the principle of data portability needs to be addressed in a federated cloud. This principle is defined as "The ability of customers to move their data or applications across multiple cloud environments at low cost and with minimal disruption." [10]. This provides the customer of a cloud provider with a minimal invasive approach to move their running application, including its data, to a different cloud provider or a different geographical locations, without having to interrupt their service. This is desirable for the customer in various ways. By loosening the coupling between the customers application and the cloud provider, the customers cost of a cloud-provider-change decreases, which increases the competition among cloud providers in favor to their customers. Furthermore, a follow the moon scenario can be implemented on container level and independent to the cloud provider. In such a scenario, the container of the application is moved to a data center at which it is nighttime, in order to use the decreased temperature for more efficient cooling [21]. Vice versa, this scenario can also be utilized to deploy the container to a datacenter with most daylight, in order to make use of as much solar energy as possible and therefore reduce the applications carbon

footprint [9].

The NIST architecture document introduces the adopted tools Docker and Kubernetes for the purpose of code portability. However, no technology for the purpose of data portability is listed in the document. The existing Kubernetes multi cluster technology Liqo [11] provides an implementation to tackle the code portability problem between multiple Kubernetes clusters. This technology allows for dynamically adding and removing clusters during runtime [3]. These clusters could potentially be managed by different cloud providers. For instance, an application operator could add an Amazon AWS Kubernetes cluster to their already existing Microsoft Azure Kubernetes cluster and move their application containers between them. This makes the management of the application more dynamic and approaches a federated cloud environment. However, this technology only aims to migrate the container without its attaches volume. Therefore, the requirement of data portability in a federated cloud is not fulfilled using this technology.

A potential solution to satisfy the requirement of data portability is designing a container migration approach, that migrates the attached volume along the container from a source machine in one cloud, to a target machine in a different cloud. This can be done using a cold migration, which interrupts the container on the source machine and copies its state to the target machine, where the container is resumed after the copying has finished. However, this impacts the quality of service of the application by making it unavailable during the time of the migration, which is not desirable for the application user. In order to keep the impact on the quality of service as small as possible, the application should be available with full reading and writing access during the whole migration process. In this document, this kind of migration is referred to as live container volume migration. The previous works, conducted by Nadgowda et al. [14] and Junior et al. [4] have been focused on creating live container volume migration approaches. These are based on union mount file systems, in order to manage file changes during the migration process. The approaches aim to migrate a container and its attached volume between two machines, with the least amount of application downtime.

In order to contribute to the research field, the aim of this work is to design a live container volume migration approach using ideas from previous works and union mount file systems. Hereby, the main focus is laid upon maximizing the service availability and performance of the application during the migration, while ensuring a complete and consistent transfer of the volume. Therefore, it needs to be made sure that especially data changes that happen during the migration, are

Migration Time
Request Latency
CPU Utilization
Memory Usage
Network Traffic
Additional Disk Space Consumption
Downtime

Table 1.1: Considered KPIs in Evaluation

transferred to the target machine. In order to measure the impact of the migration on the performance of the application, and its impact on the host machines, the migration approach is evaluated by comparing it to both a cold migration and the state-of-the-art in live container volume migration. The impact is expressed using the Key Performance Indicator (KPI)s listed in Table 1.1. The approach will be referred to as Layered Transfer migration throughout this document. In order to conduct this work, the following three research questions are posed:

RQ1: What is the current state of the art in live container volume migration, and how can it be improved?

RQ2: How does the Layered Transfer approach compare to a cold container volume migration in the given KPIs in Table 1.1?

RQ3: How does the Layered Transfer approach compare to the state of the art of live container volume migration in the given KPIs in Table 1.1?

The chapters of this document are structured as follows: Chapter 2 introduces related work that has been done on the topic of live container migration, and positions this work in the field. Chapter 3 introduces the design of the Layered Transfer migration approach. Chapter 4 describes the technical details of the implementation from this research and Chapter 5 explain the evaluation methods and experiments. Chapter 6 gives insight into the measured results, retrieved from the experiments. In Chapter 7 these measurements are discussed while Chapter 8 concludes this work.

2 RELATED WORK

This chapter introduces related work and positions the current research in the field.

Virtual Machines (VM) migration is a well researched topic, where numerous research paper have been published on in the past. Zanhg et al. [21] conducted a survey in 2018, that compiles a large amount of this work, which gives a broad overview on the research field. In their research, the authors review roughly 185 VM migration related research papers, to give a broad overview on the topic. Furthermore, the researched VM migration techniques are categorizes in a taxonomy. In addition to that, the authors identify three main challenges of VM migrations: memory data migration, storage data migration, and network continuity for both live and non-live (cold) migrations. In contrast to VM migrations, the current research focuses on migrating a container only. The VM, the container is potentially running on, is viewed as a black box.

In 2022, Kaur et al. [6] conducted a survey on container placement and migration techniques for cloud edge and fog computing. In their survey, the authors compiled roughly 100 studies in order to provide a broad overview of the state of the art of container placement and migration. The paper gives a classification of container placement and migration algorithms, and proposes a taxonomy on migrations. Here the concept of a cold migration is defined as: suspending the service on the source machine, migrating it to the target machine and starting it up. In addition to that, the concept of a live migration is defined as: the migration of a running service without service interruption. These definitions for cold and live migration are used throughout this thesis document.

2.1 STATELESS CONTAINER MIGRATION

Some work on the topic of live container migration has been done in the past. However, most of it focuses on migrating stateless containers without their attached volumes.

In their study, Tobias Kurze et al. [8] introduces the concept of data and service migration in a federated cloud environment. This concept incorporates the movement of a data and services from one

cluster to another. They distinguish between two types of migration: Shadowed/redundant migration and Non-redundant migration. In a scenario of shadowed/redundant migration, a service is duplicated into a new cluster. The new service will shadow the old service for a defined amount of time. After this time has passed, the services switch places and the old one shadow the new one. In this period, it can be validated that the new service is fully operational. When the validation period has been completed successfully, the old service can be discarded and the migration is completed. The non-redundant migration simply switches the traffic over to the duplicated service and discards the old one, without any shadowing or validation phase. Furthermore, they distinguish full- and partial migration. While a full migration migrates the whole application stack e.g. web servers and database, a partial migration, only migrates selected components.

In [7], the authors managed to migrate a state less service from a Kubernetes cluster on the 5G edge, to a Kubernetes cluster in the cloud. This migration is performed without shutting down the application and is therefore considered a live migration. This migration is executed with plain Kubernetes tool, without the usage of external multi cluster management tool like Liqo.

In 2019, Ma et al. [12] aimed to speed up live container migration between edge cluster in order to maintain a good quality of service for the user. They utilize the layered architecture of Advanced Multi-layered Unification File System (AUFS), in order to transfer only the required layer to the target host instead of the complete storage like similar technologies as P.Haul [17]. In contrast to the current research, the study of Ma et al. does only migrate the storage of the internal docker container without its volume.

In 2020, Benjaponpitak et al. introduces the container migration tool CloudHopper [2]. This tool facilitates the live multi container application migration from one cloud provider to another. In their evaluation, they tested the tool across the three mayor cloud providers by the companies Amazon, Google and Microsoft. The tool is intended to facilitate a seamless interoperable cloud with non-perceivable downtime, without being tied down to one single cloud provider. However, only the container without potentially attached volumes is migrated.

In 2022, Junior et al. created a full Kubernetes pod migration tool in [5]. The researcher approach the problem of migrating internal container state by exploiting distributed multithreaded checkpointing in order to dump multiple checkpoints of the memory. However, the study does not include migrating the stateful volumes of attached to

the container.

Additional to the problem of migrating the container itself without downtime, there is the problem of migrating live TCP connections on running containers. In 2023 Yu et al. [20] introduced a networking architecture, that can handle such requirements.

2.2 STATEFUL LIVE MIGRATION

This section summarizes the researches, conducted on stateful live container migration. Two interesting previous works have designed steteful live container migration approaches. Both of those approaches utilize a union mount file system, in order to manage file changes during the migration. A union mount file system provides the ability to present a file system as a result of union mounting different directories over each other. It usually consists of one or more lower layers and one single upper layer. The lower layers are considered to be read-only, while the upper layer is read-write. When reading a file from an mounted representation, the file will be read from the highest layer possible, meaning when a file exists in both the upper and a lower layer, it will be read from the upper layer. All new files that are written to the mount, will be written to the upper layer. Changing a file from a lower layer will result in a Copy-on-Write (CoW) operation. This means that the file will be copied from its lower layer to the upper layer, before it is changed. All consecutive reads will then read the changed file from the upper layer [16].

In 2017, Nadgowda et al. [14] conducted a research, where they created a complete container migration tool which works with zero downtime. The tool is called Voyager. It supports memory state migration, local file storage migration and network file system migration. Voyager migrates a full container including its local file system to a different host. The relevant part of this research is the local file system migration. Hereby, the authors introduce the concept of remotely mounting the file system of the source machine to the file system of the target machine in a clever way. Additionally, AUFS [1] is utilized to keep track of changing files during the migration. The general idea of that approach is, to make all files from the source available at the target instantaneously. This is done by creating an AUFS union mount on the target machine, where the lower layer is redirected to the file system of the source machine. Using this setup, the container can freely be moved to a different host machine without losing the connection to any data persisted in the file system. During this setup, the files on the target machine can be copied to the source machine, which finishes the migration. Thanks to the AUFS mount, all new incoming changes

from the service, will be persisted in the upper layer on the target machine. This process renders the connected containerized service fully available with access to all initial files during the migration. According to the research paper, the project has been open sources. Unfortunately, the link provided in the paper results in a 404 error and the implementation could not be found using a web search.

A similar study has been conducted by Junior et al. [4]. In that study, the authors created a migration tool, that exploits the OverlayFS structure by integrating it into a Kubernetes volume, in order to migrate the local file system to a remote host. The tool seeks to reduce the downtime of the application during the migration phase, in comparison to a cold migration. This is done by copying all not actively modified files prior to the migration. This way, only a few "hot-files" need to be transferred during the downtime of the application. The initial step is to mount an OverlayFS as the container volume. Afterward, a new layer in the OverlayFS structure is created, which makes all initial files, read only. The author refers to this process as "creating a checkpoint" and "checkpointing". Before files from the lower layer can be changed, they are copied to the upper layer of the OverlayFS mount by a CoW operation. This process makes it straight forward to identify files that have been changed after the checkpoint is created, since all of them are located in the upper layer. Afterward, all files in the lower layer can be safely copied to the target machine. Files that change during the copying phase, are written to the upper layer and have to be copied after copying the lower layer. The process of creating and transmitting a checkpoint can be performed iteratively in order to reduce the amount of data in the upper layer as much as possible. After an arbitrary number of iterations, the container is stopped on the source machine and the last changes in the upper layer can be copied to the target machine. After that, the container can be started on the target machine, with access to all data that has been written before, and during the migration. However, this type of migration is not considered to be without downtime, since a minimal downtime is induced before copying the last upper layer. In their research, the authors evaluated the tool on a Kubernetes cluster. In this setup, a reduction of downtime by a factor of 4 could be measured compared to their baseline migration where the full file system is transferred during the migration. The experiments, conducted in this study, are focused on a fog computing environment, with one single Kubernetes cluster with two nodes. The persistent data used in the experiments are generated by a RabbitMQ service pod. In contrast to that research, the current research focuses on live migration of long term, and potentially, large scale persistent data in cloud federated environments. Moreover, the process defined by Junior et al. in [4] is used in the current research to improve the voyager migration approach by Nad-

gowda et al. [14].

The two researches, enumerated in this section, are the only researches found that are concerned about live container migration, that include their attached volume. Since the approach of Junior et al. [5] has at least some downtime, it is considered a semi live migration, and not a completely live migration. Therefore, the Voyager migration approach from Nadgowda et al. is considered the state of the art in container live volume migration.

3 THE LAYERED TRANSFER MIGRATION

The idea behind the Layered Transfer migration approach is to combine the volume migration approaches introduced by Nadgowda et al. [14] (Voyager) and Junior et al. [4]. The goal is to shorten the time, the Voyager migration relies on remote reading files from the source machine. This is achieved by copying the initial files prior to switching the user traffic to the target machine. This means that after switching, only some files, which have been changed during the initial copy phase, have to be copied during the remote mount phase. This essentially shortens the remote read phase.

In the following, a high level overview of the migration is given. Technical details are reduced to a minimum in order to give a general overview. The specific implementation details are discussed in Chapter 4.

3.1 INFRASTRUCTURE

The Layered Transfer migration process moves all files from inside a volume on a source machine to a volume on a target machine. Figure 3.1 provides an overview over the required components. A reverse proxy is used to control the request flow from the service user to either the source or the target machine. The coordination client is required in order to coordinate the migration between the target and the host machine. It starts containers on both machines via SSH, triggers the different steps of the migration via HTTP, and sets the route of the reverse proxy to either the source or the target machine. It does not matter on which machine this client is running as long as it can reach the source, the target and the reverse proxy machine. Since all communication is done via network protocols, all components can run on different machines in different datacenters.

3.2 MIGRATION STAGES

The process of the migration is divided in 4 different stages. These stages are displayed in Figure 3.2.

The first stage, Stage 0, is about preparing the target machine for the migration. At this stage, the container and the volume are located at the source machine and the reverse proxy sends all incoming requests to it. In order to transit the system to the next stage, Stage 1, the following steps are executed:

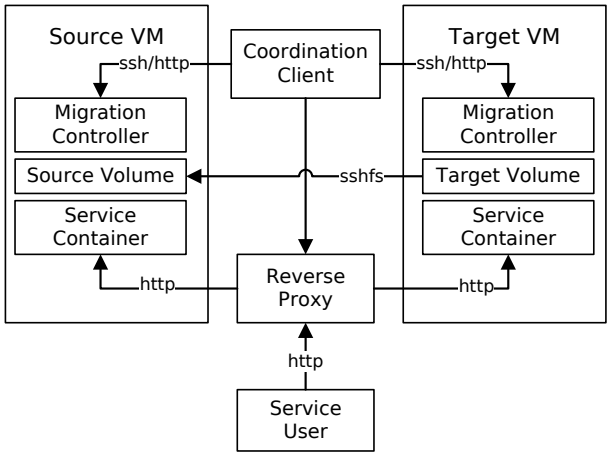

Figure 3.1: Component Setup

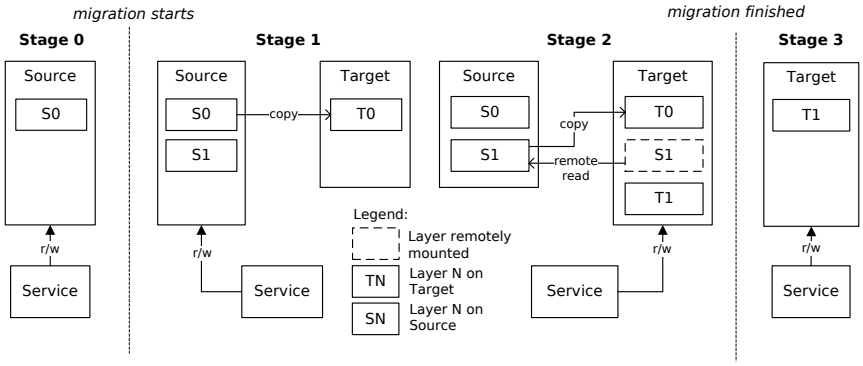

Figure 3.2: Migration Stages

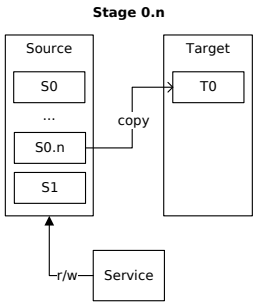

Figure 3.3: Checkpointing

1. *start preparation*: a new layered volume is created on the target machine, which is used to store the files from the volume that is to be migrated. Then, all layers from the source machine are remotely mounted on the target machine for easy and clear access of the files. In the implementation, the described actions are invoked by sending a GET request to the /prepare endpoint.

2. *iterative checkpointing*: the source machine creates a new layer on the volume, that is used to write all new incoming changes to. The lower layer on the source is now read only (as seen in Figure 3.2 Stage 1) and can be copied to the target machine, safely. This step can be repeated iteratively. For the n^{th} iteration, the migration is considered to be in Stage 0.n. This process is illustrated in Figure 3.3.

After performing these steps, the migration is considered to be in Stage 1, the initial copying stage. In this stage, still, all incoming changes are written to the source volume. However, the initial amount of data from the source volume, has already been transferred to the target volume. In order to transit the migration to stage 2, the following steps are executed:

1. The layered volume, created in the start preparation phase, is mounted as union mount on the target machine. The lower layer contains the data that has been copied during stage 0. The middle layer contains the remotely mounted upper layer from the source, which contains all changes that happened during stage 0 - stage 1. The upper layer of the mount will be empty, in order to receive new changes. The layered volume now has reading and writing access to all files of the volume.

2. The container is started on the target machine, with the layered volume attached.

3. The migration waits for the container on the target machine to be booted up completely. When the container is ready to handle requests, the reverse proxy changes its route from the source machine to the target machine.

4. The remaining changes, located in the upper layer of the source machine, are transferred to the lower layer of the target volume.

5. The middle layer, which contains the remotely mounted upper layer from the source volume, is removed from the mount.

After performing these steps, the migration is considered to be in stage 2. Now, all new changes will be written to the upper layer of the target volume, and the container on the target machine is in a fully operable state. All files that have existed before the migration,

and those that have been written during the migration, are located on the target machine. The source machine can now be cleaned and shut down.

Due to the CoW mechanism of a union mount file system, the service performance of the container on the target machine is now worse than before the migration on the source machine. In order to prevent this, all files from the lower layer are merged to a single layer. Now, the service performance of the target container matches the performance of the source container, since no CoW operations are necessary. At this point, the migration has reached Stage 3 in which it is considered to be completed.

4 IMPLEMENTATION AND TECHNICAL DETAILS

This section sheds a light on the implementation and the technical details of the Layered Transfer migration process and of the evaluation of it.

4.1 VOYAGER MIGRATION

In order to compare the Layered Transfer migration process from this study, to the state of the art of live volume migration, an implementation of the Voyager, introduced in Chapter 2, is required. Unfortunately, its source code is not published. Therefore, an implementation of the Voyager process is provided. The implementation of the voyager in this research only focuses on migrating the volume of the container from the source to the target machine. This is in contrast to the original implementation, introduced in [14], which additionally migrates the internal memory of the container. The main difference between the Layered Transfer migration process and the Voyager implementation, is the iterative checkpointing before remote mounting the source volume layers to the target machine. Speaking in terms of stages: the Voyager implementation skips stage 1 completely and transits from stage 0 directly to stage 2. Hence, the Voyager process is implemented, by reusing the already existing code from the Layered Transfer process.

4.2 COLD MIGRATION

A cold migration is considered to be the most straight forward form of migration [6]. Here, the following steps are undertaken:

1. The service on the source machine is stopped, preventing any new changes to be written to the disk.
2. All files to be migrated are transferred to the target machine.
3. The system is started up on the target machine, having access to all migrated files.

This migration process is implemented using *Python* and Secure Shell File System (SSHFS). Due to its simplicity, the cold migration is expected to perform best in all KPIs with the cost of downtime. Therefore, it established the baseline for the investigated KPIs.

4.3 TECHNOLOGICAL STACK

The migration process is coordinated by a python based application. On both the source and the target machine runs a flask server that accepts Hyper Text Transfer Protocol (HTTP) requests for controlling the phases of the migration. The union mount file system, used by the migration approach, is implemented by the OverlayFS kernel module, which is preinstalled on most Linux distributions. In order to communicate with the kernel module, the python application executes the *mount*[23] by creating a command line interface string. Mounting the layers from the source machine to the target machine is done using the SSHFS command also by creating a command line interface string. In order to remotely communicate between the coordination client and the source and target machine, the Secure Shell (SSH) client python library Paramiko [18] is used.

4.4 LAYERED VOLUME

In order to create a layered volume, and checkpointable volume OverlayFS [16] is used. Unfortunately, OverlayFS does not provide an off-the-shelf checkpoint function. OverlayFS merely provides a way to statically union mount multiple layers as a merged representation. Therefore, checkpointing has to be manually implemented.
In order to create a checkpoint, first a new directory is created which represents the new layer. Afterward, all layers which have already been mounted to the overlay mount will be mounted as lower layers, with the newly created layer as upper layer. In order to have full availability of all files during checkpointing, the new mount can be mounted "over" the existing OverlayFS mount. This means that the existing mount does not have to be unmounted first. A caveat of this approach is, that files that have been opened for writing in the former upper layer will still be written to the former upper layer while the layer is considered to be read only. By waiting for all open files in the former upper layer to be closed, it can be made sure that the former upper layer is read-only. All new writes to the mount will result in opened files in the new upper layer. Then, the former upper layer can be copied without the danger of data loss.

4.5 CHECKING FOR OPEN FILES

Before a layer can be copied, it has to be made sure that no changes will happen during or after the copying process. As described in Section 4.4, it is possible that files in lower layers of an overlay mount are changing. Therefore, checking for open files in a layer is essential before copying, in order to prevent data loss. In the migration process, checking for open files is done by the management client. Therefore,

it executes the *lsof* [22] command in the running docker container in order to check the files, that are currently open for writing. Since the container itself does not know that the volume it is attached to is layered, the *device number* from the upper layer needs to be used on the upper layer of the volume. It can be read using the *stat* command on the upper layer directory from outside the container. Using this number, the management client can find all open files on the volume from inside the container that match the mount point of the volume and do not match the device number of the upper layer. Necessarily, these files are open in a lower layer. By periodically repeating this process, the migration can wait until all files in lower layers are closed before copying them to the target machine.

4.6 MERGING LAYERS

As described in the transition process from stage 2 to stage 3, all files from the lower layer need to be merged to the upper layer in order to prevent CoW operations and therefore increase the service performance. In order to do so, all files in the lower layer that are not present in the upper are hard linked to the upper layer. Hard linking is preferred over copying since it is much faster and does not produce duplicate data. Unfortunately, due to the *dentry* cache of the file system, OverlayFS will still expect these files in the lower layer and will try to perform CoW operations on them. This results in a *FileExistsException*, since the file is already present in the upper layer. In order to prevent this, the *dentry* cache is reset after moving the files. Now, overlay will have to update the *dentry* cache on new read and write operations.

A small caveat of this approach is that it is not atomic, and OverlayFS updates the *dentry* cache on every read and write action. This means that writes to lower layer files during the linking will result in the *FileExistsException* since the cache still expects the file to be in the lower layer. This caveat can be mitigated, by cleaning the cache before, and after linking the files. However, a tiny chance for *FileExistsException* to occur persist, for files that are written after the linking phase, but before cleaning the cache. This chance can be completely reduced to zero by prohibiting OverlayFS from writing into the *dentry* cache on reads at the time of merging. However, this alteration of OverlayFS is considered out of scope for this research.

4.7 LAZY FILE TRANSFER

As an enhancement to the migration time, Nadgowda et al. [14] proposes to implement a lazy file transfer. This means, that in stage 2, not all files from the upper source layer have to be copied to the lower

layer of the target. Since all writing access to files in lower layers trigger a CoW operation, they are necessarily copied to the upper layer of the target machine. Therefore, they can be skipped when the upper source layer is copied from the source to the target machine during Stage 2.

4.8 REQUIREMENTS FOR THE MACHINES

In order to create a remote mount and transfer data between the source and the target machine, the target machine requires the SSHFS software to be installed. Consequentially, on the source machine, a running SSH client is required. Additionally, the target machine needs the credentials of a user that can access the layers of volumes on the source machine. Before a layer is copied from the source to the target, it needs to be verified, that there are no open files in that layer, in order to prevent data loss. Otherwise, a change could be written to the layer after it has been copied. In order to check for open files, the management client needs access to the docker daemon on both the source and the target machine. Naturally, OverlayFS is required on both the source and the target machine, however it is part of the Linux kernel and will therefore be preinstalled on Linux distributions. In contrast to the migration approach by Junior et al. [4], no additional migration tool has to be installed in the container itself.

4.9 IMPROVEMENTS

Keeping track of chaining files in the volume is implemented using OverlayFS. However, since it employs CoW operations, changing a small amount of data in a large file, results in that large file being duplicated to the upper layer. With the proposed process, it is copied twice to the target. Therefore, the process can be improved by moving away from checkpointing the file system using CoW operation, to keeping track of the changes them self. This way, a small change in a large file remains a small change to be persisted and transmitted via the network. An additional improvement is to compress layers before copying them to the target machine, in order to save data.

5 EVALUATION DESIGN

In order to evaluate the Layered Transfer migration process, it is compared to a cold migration and to the Voyager implementation. The measurements retrieved by executing the cold migration establish a theoretical baseline, since the cold migration is supposed to be the most naive migration approach. The measurements retrieved by experiments using the Voyager migration are compared to the measurements of the Layered Transfer migration in order to explore the amount of improvements.

5.1 FILE BASED DUMMY APPLICATION

In order to validate the implementation of the migration process, a file based dummy application is created. Its purpose is to simulate a real world application as close as possible. In order to do so, it exposes files on the volume via an HTTP interface. The application is containerized and shipped to both the target and the source machine. There it is connected to a layered volume. For the evaluation, the following three HTTP methods are used:

1. $POST/init?chars =< x > \&files =< y >$: In order to populate the volume to be migrated with initial files, the init method is used. In this method, the number of files (y) and the number of characters per file (x) can be specified. The service will create one file with $x - 1$ "I" characters and one "E" character and duplicate it y times. In ASCII encoding, one character is represented by one, byte. Therefore, we can control the file size of the newly created file. Every file will receive a new Universal Unique Identifyer Version 4 (UUIDv4) as name in order to store them in the same directory.

2. $POST/write - new - file?chars =< x >$: This method writes a new file to the volume into the target directory. For the name of the files, an UUIDv4 is used. As content of the file, the number of $x - 1$ "I" characters with a trailing new line character are used. The filename will be returned after the operation is completed.

3. $POST/change - current - file$: This method changes the last file that has been written to the disk using the *write-new-file* method. A single "E" character is appended to that file. During the migration, this method does mostly not trigger a Copy on Write operation.

4. *POST/change − random − file*: This method, changes a random file that has been written using the *write-new-file* method. Therefore, a random file will be chosen from the target directory and an "E" character is appended. The filename of the changed file is returned after the operation is completed. During the migration, this method mostly triggers a Copy on Write operation.

5. *GET/file*: This method reads a random file from the target directory. As a result, the filename and the complete content of the file will be returned in JavaScript Object Notation (JSON) format.

The application is implemented to be a lightweight, easy to evaluate application, that can simulate a broad range of real world file based application types. Its methods are created to make the evaluation adaptable to different application types.

After a migration of the filedummy application, it can easily be tested for consistency. Therefore, every */write-new-file*, *write-current-file* and */write-random-file* call are stored, together with the expected number of "E" characters that should be in the file when the operation is completed. After a migration of the filedummy application, all files are read and the number of expected "E" characters per file can be compared to the number of actual new lines per file. This way, it can be checked whether all files that have been added or alternated during the migration are stored on the target volume.

5.2 MACHINE SETUP

For the *target* and the *source* machine, a virtual machine with identical specifications from the TNO datacenter is used. Additional to the two VMs, a third VM is used in order to run the coordination client, the reverse proxy and the service user. The following table summarizes the specifications and relevant installed software packages.

Memory	4 GB
CPU	4 Core
Network UP/DOWN	10 Gigabit
Operating System	Debian Linux 12
Linux Kernel	6.1.0-21-amd64
Python Version	3.11.2
SSHFS	3.14.0
nginx	1.22.1

5.3 REQUEST LATENCY

Concerning the measurement of the *read-* and *write latency* during the migration, an HTTP client is constructed. This client records the used endpoint, the start time, the end time and the response status of a request. Since the volume is a union mount, some types of writing requests trigger a copy on write operation. Therefore, writing requests are split in three different categories:

New Write: Requests of this type trigger the creation of a new file on the server. In terms of a layered volume, the new file is written to the upper layer without any added actions. This request is simulated by the HTTP method *POST /write-new-file* from the filedummy service.

Sequential Write: Requests of this type trigger a change of the file that is currently processed by the service. When the volume is layered, most of the time, the current file is located in the upper layer of the union mount file system. Therefore, it is altered and persisted to the disk. This type of request is simulated by the HTTP method *POST /change-current-file*.

Random Write: Requests of this type trigger a change of a randomly chosen file. During a migration, such file will mostly be located in a lower layer of the union mount. This strongly depends on the number of initial files, that have to be migrated. More files, means a higher chance of writing a file, that has not been written before, and is therefore located in the lower layer. Therefore, before changing the file, a CoW operation has to be performed. This leads to a worse response time. This type of request is simulated by the *POST /change-random-file*

As a configuration parameter, before starting the client, for every of the four request types *read*, *sequential-write*, *random-write* and *new-write* a probability can be defined. When the migration starts, the client starts to send non-blocking requests to the file dummy application. The number of concurrent requests is neither limited by the client, nor by the service. Before sending a request, the type is determined by its probability. This way, different application types with different use cases can be simulated during the migration.

5.4 MACHINE STATISTICS

In order to measure the KPIs *CPU Utilization*, *Memory Usage* and *Network Traffic*, the HTTP interface of the VM manager in the TNO datacenter is used. Therefore, the statistics endpoint of the API is called once every second during the migration for both the source and the target machine. The timestamp, at which the request is sent, can

be used to map the response data to the collected request latencies of the migration. The additional disk space consumption is measured by checking the folder size of the layers of a volume and dividing this value by the folder size of the volume size itself. Therefore, the additional space consumption due to copy on writing in proportional to the physically required space is retrieved.

5.5 CONSISTENCY

In order to check, whether the implemented process is consistent, every conducted experiment checks the consistency of the files in the following way: Since the filedummy application, returns the ID of an altered file as response, it is straight forward to memorize which files have changed. Additionally, since every writing request appends exactly one single "E" character to the altered file, it can also be memorized how often a file has been changed. After running an evaluation experiment, the number of received writing requests (*POST /write-new-file*, *POST /change-current-file* and *POST /change-random-file*) are counted per file. If the number of correctly processed write requests matches the number of "E" characters in the corresponding file after the migration, the file is considered to be migrated consistently.

5.6 EXPERIMENTS AND PARAMETERS

In order to answer the research questions *RQ2* and *RQ3*, the experiments listed Table 5.1 in are conducted. For every listed experiment, a cold migration, a voyager based migration and a Layered Transfer migration are executed. The first four experiment configurations (*Only Read*, *Only Write*, *Only Sequential-Write* and *Only New-Write*) have been chosen in order to explore the dependency of that specific request type on the KPIs of interest. Naturally, no real world application would only perform one single type of operation, nevertheless in the experiment the other request types are eliminated as confounding factors. Therefore, the result can be used to make recommendations for the choice of a migration method for different application types. The experiment configurations *Read Heavy* and *Write Heavy* are conduced in order to produce a more realistic result, since real world services mostly execute read and write operations. All experiments are conducted with an initial amount of data of 1 GB in total. This 1 GB of data is split in 3 different ways in the number of initial files and the initial file size, as displayed in Table 5.1. The combination of 1000 initial files with a size of 1 MB each is used to establish a baseline. In order to discover the dependence of the file size on the KPIs one combination with smaller files (2000 x 0.5 MB) and one combination with larger files (500 x 2 MB) are constructed. All migrations using the Layered

Experiment	Number of Initial Files	Initial File Size (MB)	R/SW/RW/NW [1]
Only Read	2000	0.5	100 / 0 / 0 / 0
	1000	1	
	500	2	
Only Sequential-Write	2000	0.5	0 / 100 / 0 / 0
	1000	1	
	500	2	
Only Random-Write	2000	0.5	0 / 0 / 100 / 0
	1000	1	
	500	2	
Only New-Write	2000	0.5	0 / 0 / 0 / 100
	1000	1	
	500	2	
Read Heavy	2000	0.5	80 / 10 / 5 / 5
	1000	1	
	500	2	
Write Heavy	2000	0.5	10 / 70 / 10 / 10
	1000	1	
	500	2	

Table 5.1: Experiments with Parameters

Transfer approach, create one single checkpoint during the migration in order to keep the resulting data clear.

5.7 DATA COLLECTION

This section describes on which way the result data is retrieved. The logging of the request latency (Section 5.3) and the logging of the host statistics (Section 5.4) are both written to a separate file, named after the volume name that is migrated in that experiment. The metadata of an experiment (migration time, additional disk space consumption target, additional disk space consumption source) is collected in a single file. Using the name of the volume that is migrated in an experiment, the metadata can be mapped to the measured latency and the host statistics. Every experiment is executed 5 times with the

1 The Ratio of Reads (*R*), Sequential-Writes (*SW*), Random-Writes (*RW*) and New-Writes *NW*

exact same parameters. Afterward, the mean of all 5 experiment runs is calculated in order to create a more truthful result, where outliers are mitigated. For every experiment, the KPIs are measured from 5 seconds before the migration starts, until 5 seconds after the migration is finished. This way, the impact of the migration on the KPI of interest can be detected and visualized.

The data is analyzed in python using the *pandas*[24] library and plotted using the *matplotlib*[13] library. Both frameworks are combined in a *jupyter notebook*[19] in order to explore the data.

6 EVALUATION RESULTS

This chapter summarizes the results of the conducted experiments, listed in Table 5.1 based on the KPIs from *RQ1* and *RQ2*. During the execution of the listed experiments, the following KPIs, starting at Section 6.2 are measured and compared. Every experiment configuration, listed in Table 5.1 is conducted five times and a mean value is build for each measured KPI. A complete overview of all collected data can be found in the appendix in Section A.1.

6.1 PLOT CLARIFICATION

In the following sections, various KPI values are plotted in graphs. Most of the plots include vertical red dotted lines. These lines specify the point in time where the migration has completed a stage, as described in Section 3.2. Naturally, the number of stages differ between the used migration methods. The cold migration will be divided in three stages. These stages correspond to the steps explained in the listing from Section 4.2. The Layered Transfer plots, will have four different stages, which correspond to the stages explained in Section 3.2. The Voyager migration will have three stages that correspond to the stages explained in Section 4.1.

6.2 MIGRATION TIME

In order to measure and compare the duration of a migration the four experiments **Only Read**, **Only Sequential**, **Only Random** and **Only New** are taken into account. Table 6.1 provides the migration durations for the different experiments. Each experiment is conducted 5 times and the mean result is displayed.

	Cold	Layered Transfer	Voyager
only-new	26.8	29.7	27.2
only-random	24.3	33.8	29.3
only-read	25.8	29.9	29.1
only-sequential	28.5	29.9	26.1

Table 6.1: Migration Time in Seconds for 1000 initial Files of 1MB Size

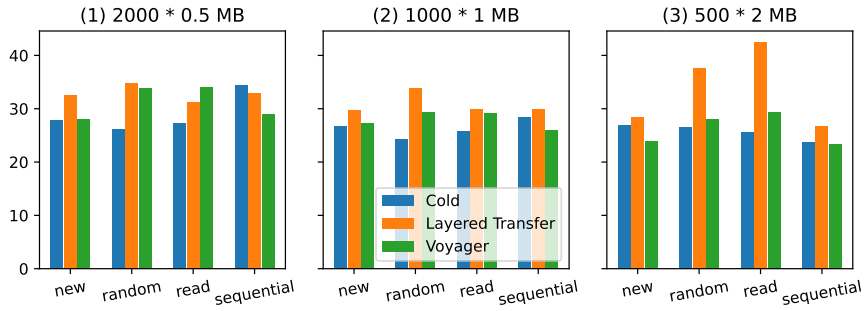

Figure 6.1: Migration Time Comparison

When comparing the Layered Transfer to the Cold Migration, it can be observed that in all experiments the Layered Transfer approach, takes slightly more time to be completed. The highest difference is observed in the only-random experiment. Here, the Layered Transfer approach takes 9 longer than the cold migration. This amounts to 37% additional time consumed.

In comparison to the Voyager implementation, a similar result can be observed. The Layered Transfer method finishes later than the voyager migration in every experiment.

Figure 6.1 compares the three different migration methods with different combinations of initial file sizes and number of initial files. When adjusting the initial file size to 0.5 MB (Figure 6.1 (1)), some small differences can be observed. For the experiment only-read, the Layered Transfer migration is slightly faster than the voyager migration. For the only sequential experiment, the same difference can be observed. In this experiment, the Layered Transfer migration is even faster than the cold migration.

In subfigure (3) of Figure 6.1, it can be observed, that the migration time raises for the experiments only random and only read. Here it is approximately 42% slower than the cold migration for the only random experiment and even 68% slower for the only read experiment. In contrary to that, the migration time of the voyager approach, does not change significantly in comparison to the experiment with a filesize of 1 MB (2).

6.3 REQUEST LATENCY

Since the request latency depends on the type of operation the service has to execute, the requests are split in the following 4 categories:

- Reading

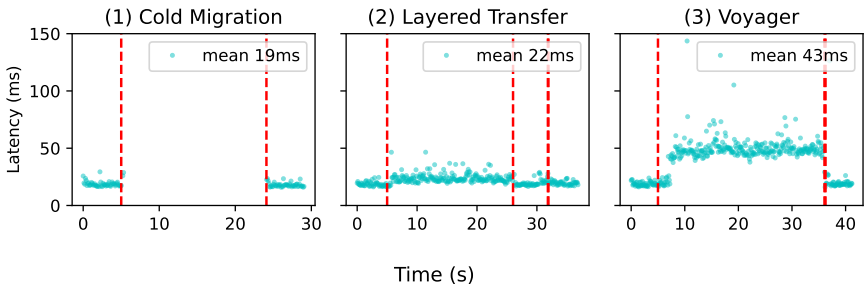

Figure 6.2: Read Latency Comparison

- Sequential Writing
- Random Writing
- New File Writing

6.3.1 *Read Latency*

In order to investigate the read latency during the migration, the experiments **Only Read** and **Read heavy** have been conducted.

Figure 6.2 visualizes one of the experiment runs with the input parameters of the **Only Read** experiment from Table 5.1. The red dotted lines signals the end of a stage. Therefore, the first line marks the point in time where the migration signal is given by the coordination client (Stage 0).

In the first plot (cold migration) the latency baseline is established at around 24 ms per request. In the first stage of the cold migration, all requests are routed to the container on the source machine. In the second stage of the cold migration, the container on the source machine is turned off, in order to freeze the state of the volume. Afterward, the files are transferred to the source machine. Naturally, the container is not available in this stage and therefore no requests can be processed. When all files are copied to the target machine, the container on the target machine can be booted up. After a small initialization time of the container, all requests are routed to the target machine, where the latency is back at around 20 ms in the third stage.

The second plot (2), visualizes the read latency for the Layered Transfer migration approach. A slight increase of latency can be observed between the first and the second segment. In the second segment, the migration is in Stage 1. This means that all requests are routed to the source machine, where all changes are written to the top layer. During this segment, the lower layer is copied to the target machine. The slight change in latency is attributed to the fact that the volume is now layered, and the machine has to copy files in the background.

In the third segment, the lower layer of the volume has been copied completely to the target machine. This layer together with the remote upper layer on the source machine are union mounted into one volume. Since there have been no writing request in this experiment, the remote upper layer of the source is empty, which means that not a single read request needs to be read from the source machine, which could affect the latency. All files are present at the target machine. After a small cleanup time, the migration is completed. When executing this experiment 5 times in a row and calculating the mean, the mean read latency for the cold migration is 20 ms. For the Layered Transfer a latency of 22 ms is calculated. This means in comparison to the baseline, there is a slight increase in the read latency, however it only differs 2 ms.

In the third plot (3), the result of the Voyager is displayed. Here, an interesting pattern can be observed. While the latency in the first and last Stage reads around 20 ms, the latency approximately doubles in the second Stage. In this segment the voyager migration is in Stage 2 (as described in Figure 3.2) which means, that all requests are routed to the target machine, while the initial data from the source is remotely mounted. this means that every read request gains one extra hop, since the data has to travel from the source through the target to the client. Executing this experiment five time and calculating the mean results in a latency of 48 ms. This is more than twice as long as the latency of the layered transfer approach at 22 ms. According to the data, the layered transfer approach improves the read latency in comparison to the Voyager approach significantly, by more than 50%.

Adjusting the size of the initial files to 0.5 MB and to 2 MB, does not change the observed patterns in the results of the only-read latency. It merely increases and decreases the latency for larger and smaller files, respectively.

6.3.2 *Random Write Latency*

This section analyzes the random-write latency during the three types of migrations. Therefore, the experiment only-random is conducted.

Figure 6.3 displays the latencies for the migrations. In the first subplot (1) the cold migration is visualized. The mean latency for this migration is 6 ms per request. In the second subplot (2) the Layered Transfer approach is displayed. In this plot can be observed that right after the migration signal is given at the first vertical dotted line, that the latency raises quickly. At this point, the first checkpoint is created, and nearly every random write request triggers a CoW operation. This raises the latency of the requests to approximately 284 ms on average. In the third subplot (3) a similar behavior can be observed for the

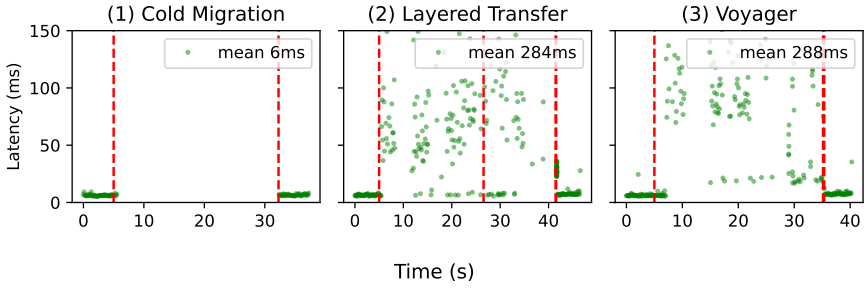

Figure 6.3: Random Write Latency Comparison

Voyager migration. The mean latency here is, similar to the latency during the Layered Transfer migration, at 288 ms. Even though in the read-only experiment in Figure 6.2 a difference in latency due to the longer period of remote reading could be observed, this effect can not be observed in this experiment.

When changing the initial file size to 0.5 MB, it can be observed, that the mean random write latency of the voyager migration drops to 169 ms on average for 5 experiment runs. The average latency for the Layered Transfer migration only drops to 200 ms, which results in a 31 ms difference between the two approaches. Therefore, in this experiment, the latency of the Layered Transfer migration is 18% worse than the latency of the voyager migration.

For initial files with a size of 2 MB, the average random write latency raises to 400 ms for the Layered Transfer migration. The voyager migration reaches on average 335 ms. Therefore, in this experiment, the latency of the Layered Transfer migration, is 19% worse than the latency during the voyager migration.

6.3.3 *New Write and Sequential Write Latency*

In order to evaluate the write latency for sequential writes and new writes, the experiments **Only Sequential** and **Only New** have been conducted. The results for all three different types of migration can be found in Figure 6.4 for only sequential write- and only new write requests respectively.

In both figures can be observed, that the different stages of the migration do not have any significant effect on the latency of both new writes and sequential writes. Therefore, both request types are analyzed together in this section. As baseline, a mean latency of 7 ms and 6 ms is established. Neither the Layered Transfer nor the Voyager method deviate significantly from that baseline. This is due to the working principle of the layered volume. Naturally, files that are newly written to the disk, are stored in the upper layer of the union mount file system. Since the file did not exist before, no CoW action has to be

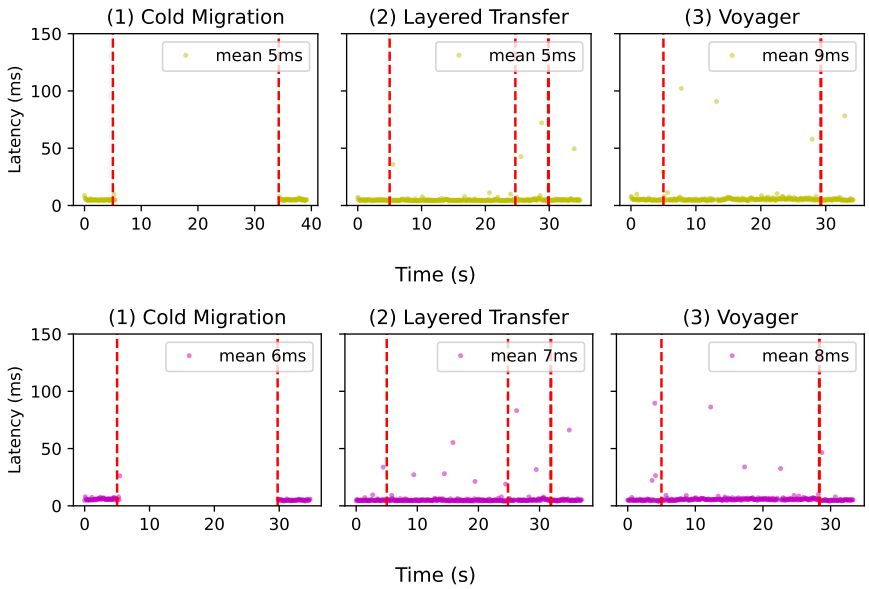

Figure 6.4: Sequential Write Latency Comparison

performed and the latency does not depend on the layered volume. This is very similar for the sequential write requests. Since sequential requests write to the same file, in the worst case there is one single CoW operation for the first request. All consecutive requests will not trigger CoW operations since the file is already located in the upper layer.

As results for the mean calculation of 5 experiment repetitions for sequential writes, the cold migration results in 6 ms, the layered transfer results in 6 ms and the voyager approach results in 8 ms. The mean values for 5 experiment repetitions of the new-writes result in approximately the same values: 9 ms for the cold migration, 6 ms for the layered transfer and 6 ms for the voyager approach. The small differences of the latencies between approaches and between request types is very small (< 2 ms) and can therefore be neglected.

Adjusting the initial file size to 0.5 MB and to 2 MB does not change anything to the latency of the only-new and the only-sequential experiments.

6.4 CPU UTILIZATION

This section compares the CPU utilization of the Cold migration, the Layered Transfer migration and the Voyager migration approaches.

Figure 6.5 visualize the CPU utilization of the source and the target machine during the migration per used migration method. The figure

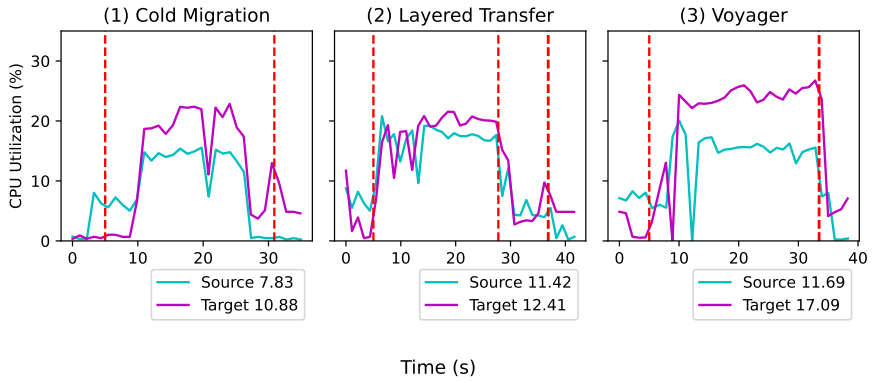

Figure 6.5: CPU Utilization Only Read

depicts the single run of the only-read experiment configuration. This experiment is chosen to be presented since it shows the differences in CPU load between the migration methods best.

In the first sub-figure (1) of Figure 6.5, the CPU utilization spikes for cold Migration spikes after the migration has started and drops after it is finished. Naturally, before the migration of the source machine is higher than on the target machine since the service is running there. After the migration, the service runs on the target machine, and the observed effect is reversed. In comparison to sub-figure (2), the Layered Transfer migration, has an overall higher CPU utilization on both the source and the target machine. At around second 27 on the x-axis, the CPU utilization is observed to drop. This is attributed to the fact that the largest amount of files has been transferred to the target machine. Only the remaining changes that have occurred during the migration have to be transferred here. When comparing this to the Voyager migration, in sub-figure (3), it can be observed that the mean CPU utilization is even higher. The observed drop from sub figure (2) is can not be found here, since the Voyager migration skips stage 1 where initial files are copied before the remote mount stage. Therefore, the voyager migration does two tasks in parallel. Copying the initial data from the source to the target machine and using remote reads in order to provide a working service. In the Layered Transfer approach, these tasks are divided in two stages, which leads to a lower CPU utilization during the migration.

Figure 6.6 visualizes the mean CPU utilization of running 5 experiments with the same profile on the y-axis, per profile on the x-axis. Sub-figure (1) and (2) display the source- and target machine CPU utilization, while sub-figure (3) displays the mean from both machines. In this figure, it can be observed that the Cold Migration approach has the lowest CPU utilization across all experiment profiles. This is attributed to the fact, that the CPU is not used for running the service during the migration. The CPU utilization for the Layered Transfer approach is slightly higher than the one of the cold migration,

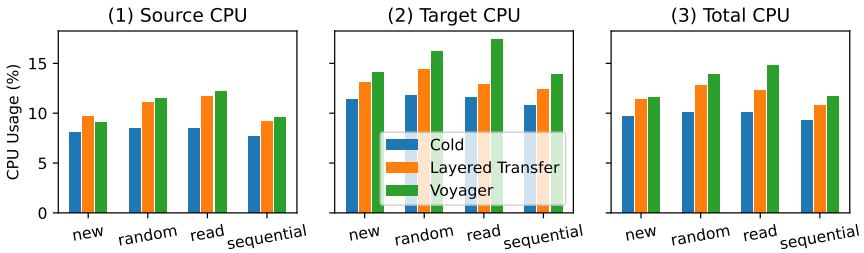

Figure 6.6: CPU Utilization per Experiment with 1000 files a 1 MB

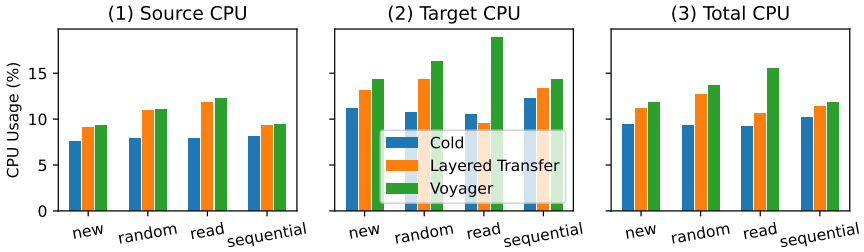

Figure 6.7: CPU Utilization per Experiment with 500 Files a 2 MB

since during the transfer, the CPU also has to be used for running the service. The Voyager migration approach on average has the highest CPU utilization across all experiments. Only for the only-read and the only-sequential approaches, it has a slightly lower utilization than the Layered Transfer method. However, the difference might be attributed to unknown processes running in the background on the source machine during the migration, since it is so minimal.

When comparing the different profiles, the only-read experiments have the highest CPU utilization, while the only-new and the only-sequential experiments have the lowest. This pattern corresponds to the results found when measuring the latencies during the migration in Section 5.3.

Furthermore, it can be observed that the target machine has in all cases a higher mean CPU load than the source machine. This can be attributed to the fact that the target machine copies the files from the source machine using a pull approach.

When changing the initial file size to 0.5 MB per file, no significant change in the measured data can be observed. When the initial file size, on the other hand, is changed to 2 MB per file, a difference on the target machine can be observed. In Figure 6.7 it can clearly be observed, that the CPU utilization of the only-read experiment using the Layered Transfer migration approach is approximately as half as high as the voyager CPU utilization. The voyager approach uses 18.8%, while the Layered Transfer approach only uses 9.1% of the CPU capacity.

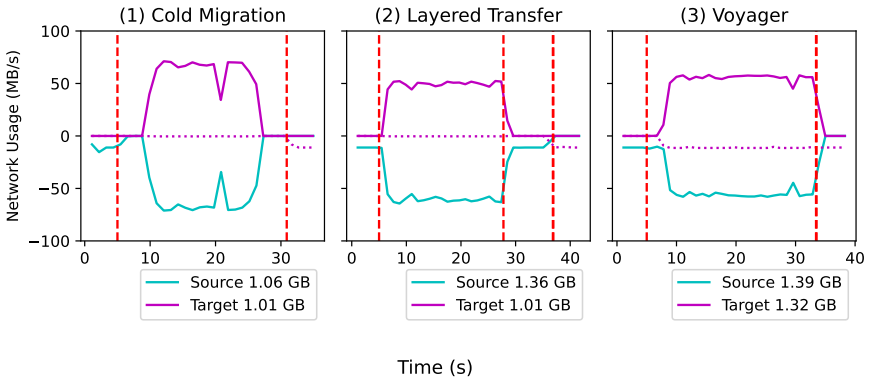

Figure 6.8: Network Comparison Only Read

6.5 MEMORY USAGE

In all conducted experiments, the memory usage is around 2.9 GB for both the source and the target machine. No significant changes in the memory usage is visible. This is the memory usage of all processes running on the machine accumulated.

6.6 NETWORK TRAFFIC

This section introduces the results of the network traffic measurements during the three different migration approaches.

Figure 6.8 visualizes the network usage per second for each of the three migration approaches with the only-read profile. In order to create the most insightful plot, where no lines are covered by other lines, incoming network traffic is visualized as positive, while outgoing traffic is visualized as negative. The network traffic between the source and the target machine is in the main focus here. Therefore, the plot displays the outgoing traffic of the source machine and the incoming traffic of the target machine. This traffic is visualized as a full line, while the outgoing traffic of the target is drawn as a dotted line. In the legend, the total number of transferred gigabytes can be found for the outgoing traffic of the source machine and the incoming traffic of the target machine. The visualized values include both the traffic used for the migration and the traffic used by the running service. The different migration stages are marked by the vertical red dotted lines.

In the first subplot (1), the traffic of the cold migration is depicted. The total amount of bytes transferred is approximately 10 GB. Clearly, the incoming target traffic is approximately the same as the outgoing source traffic. The small amount of extra data that is transmitted is attributed to the service, that answers to read requests. In comparison to the second subplot (2), where the Layered Transfer migration is plotted, a change in the outgoing traffic on the source machine can

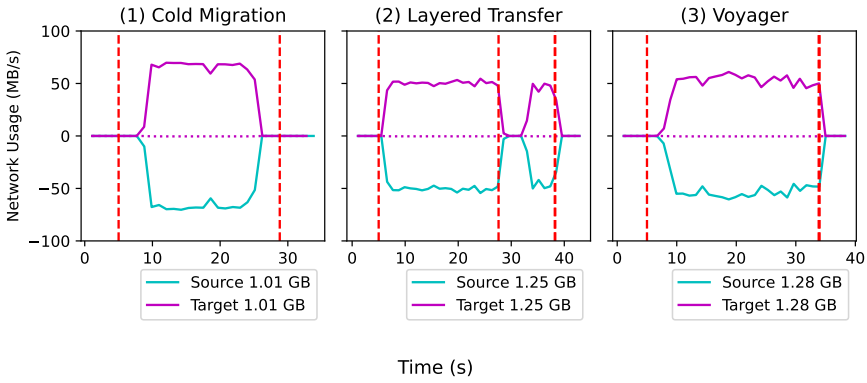

Figure 6.9: Network Comparison Only Random

be observed. While the cold migration stops the service during the migration, the Layered Transfer approach continues the service for the whole migration time. Since, only read requests are coming in, and the Layered Transfer approach runs the service on the source machine for most of the time, this results in an overall higher outgoing network traffic on the source machine. In the third subplot (3), where the Voyager approach is displayed, a clear difference can be observed. Like the Layered Transfer method, the Voyager method runs the service during the whole time of the migration, which leads to a higher outgoing network traffic than the cold migration has. However, since the Voyager migration switches the service to the target machine, before copying the initial files and remote reading them during the migration from the source machine, there is a higher network traffic on both the source and the target machine. All data requested by the service user at the target machine, has to be remotely red from the source machine. The average outgoing source traffic and incoming target traffic across 5 experiment runs are 13.2 GB and 10 GB for the Layered Transfer approach and 14 GB and 13.4 GB for the Voyager method respectively. This shows, that the extra amount of data transferred by the Voyager method is more than twice as high as the extra amount of data transferred by the Layered Transfer approach.

In Figure 6.9, the differences between the three migration methods in terms of network traffic for the only-random profile are shown. In the first subplot (1) the cold migration is displayed. Like in Figure 6.8 nearly no extra data on top of the initial data is transmitted. This is different to the Layered Transfer migration, in the second subplot (2). Since the client sends only random write requests during the migration, many files are being changed and for almost every change, a CoW has to be performed. Therefore, the amount of data to be transferred grows during the migration. This can be observed in the second peak in migration stage 2. Naturally, the total amount of data transmitted is higher than the amount of data transmitted during the cold

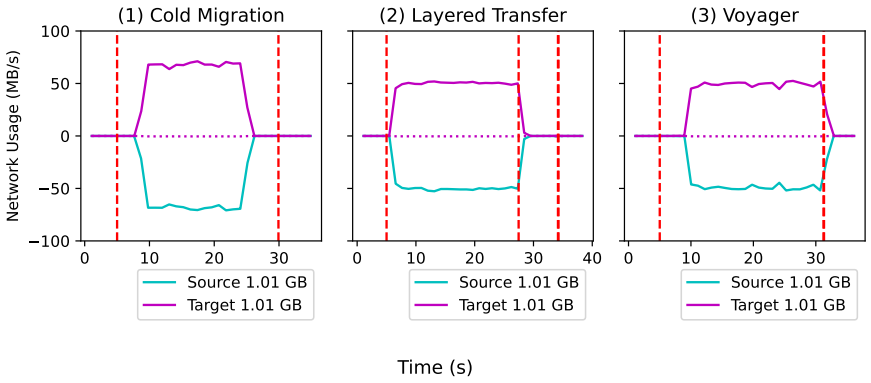

Figure 6.10: Network Traffic Comparison New Writes

migration, since data is changed during the migration. In the third subplot (3) the Voyager migration is shown. Like the Layered Transfer migration, it allows the service to change files during the migration, which also here results in a higher amount of data transmitted. This number is slightly higher than the amount of data transmitted in the Layered Transfer migration, however this arbitrarily swaps, for every experiment run and is therefore not attributed to the migration type. On average, for 5 experiment runs for the only-random profile, the values are approximately the same at 12.5 GB for the Layered Transfer approach and 12.6 GB for the voyager method.

In contrary to the read requests, send in Figure 6.8, the payload of the requests send in this experiment is very small. Therefore, the payload of the service requests do not change the outcome of the total amount of transferred bytes, as observed in Figure 6.8.

Figure 6.10 shows the network traffic during the migration of the three different approaches for the profile only-new. The graph for only-sequential looks exactly the same in terms of change rate and total sum of exchanged bits, and is therefore not listed here. This can be attributed to the fact, that none of the approaches triggers a CoW action nor uses the actual remote read from via the target on the source machine.

When chaining the file size of the initial files to 2 MB, the graphs change as displayed in Figure 6.11 for the only read experiment. Here, it can be observed, that due to the increased file size, the total data transferred increases in two ways. First, all responses to requests to the running service are 2 MB instead of 1 MB as before. In the Layered Transfer migration now the outgoing network traffic is 2.2 GB of data while the incoming data to the target remains around 1 GB. This is attributed to the fact, that the service is running on the source machine for most of the time. It has to respond with responses of 2 MB to the

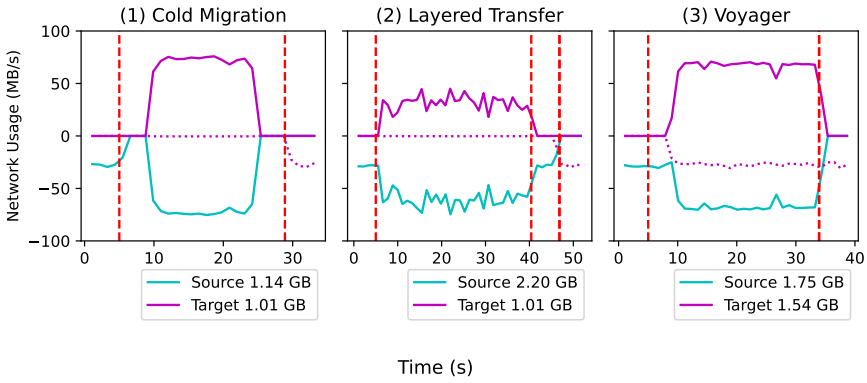

Figure 6.11: Network Traffic Comparison Reads 2MB with 500 Initial Files

service user during the migration. This data is therefore not registered as incoming on the target server. In the Voyager migration, on the other hand, this overhead of data can be observed on the target machine, which runs the service during the migration. Here, the outgoing traffic increases to 1.54 GB. In contrary to the Layered Transfer migration, the outgoing network traffic of the source increases as well to 1.75 GB. This is inevitable since all data requested by the service has to be remotely read on the source, which increases its outgoing traffic.
Note, that the total migration time is higher for the Layered Transfer migration, which gives the service more time to send reading requests in that experiment. While on average in this experiment 392 requests have been sent during the voyager migration, 522 requests have been sent during the Layered Transfer migration. The difference of number of requests amounts to 130 requests, which results in additional 260 MB which would have been sent during the voyager migration if it had taken as long as the Layered Transfer migration.
For the random-only profile with a file size of 2 MB, the transmitted data between source and target machine increases to 1.44 GB for the Layered Transfer method and to 1.4 GB for the Voyager approach. This means 40 MB more traffic for the Layered Transfer migration than for the Voyager migration. This is attributed to the fact, that the Layered Transfer method executes CoW operations on the source machine, which results in additional data. This additional data is then copied to the target machine. In contrast to that, the voyager approach performs the CoW operations on the target machine, which means they do not have to be copied during the migration.

When changing the file size of initial files to 2 MB or 0.5 MB, no significant difference can be observed in terms of total transferred data and change rates for the profiles random-write, new-write and sequential write.

6.7 CONSUMED DISK SPACE

In this section, the consumed disk space during the migration is evaluated. Due to the use of CoW operations during the migration, files are necessarily duplicated. This means that the layers persisted on the disk can be larger than the size of the mounted volume. This difference is regarded as access disk space consumption.

Naturally, the access space consumption for the cold migration is always 0 since no layered volumes are used, and therefore CoW is not performed. Additionally, in cases where only reading requests are sent to the server, also no CoW operations are performed. This is also the case for requests, that trigger a new file to be written. Therefore, solely the experiments only-sequential and only-random are investigated in this section.

On average of 5 executed Layered Transfer migrations with the same parameters, the only-sequential experiment results consumes 0.1% of disk space additional to the size of the volume on both the source and the target machine. Since only sequential writes are being performed, most of the writing operations will not trigger a CoW operation. This results in such a low additional disk space consumption. For the voyager migration, the average additional disk space consumption of 5 migrations amounts to 0.

The only-random experiment on the other hand results in more variable values, since chances are high that all requests will trigger a CoW operation. For the Layered Transfer migration, on the source machine, the average additional disk space consumption across 5 experiment runs amounts to 23.9%, while on the target machine it amounts to only 3%. The voyager migration shows opposite behavior. Here, the average disk space consumption on the source machine naturally amount to 0%, while it amounts to 21.9% on the target machine. This is a logical behavior, since the voyager migration does not make any use of a layered volume on the source machine. All, changes happening during the migration, are written to the target machine. The Layered Transfer migration, on the other hand, makes use of a layered volume both on the source and the target machine, which results in potential additional disk space consumption on both machines. Here, most of the changes happening during the migration are written to the source machine and only a few to the target machine. In total, the additional disk space consumption is slightly higher for the Layered Transfer migration. Figure 6.12 provides a visual comparison between the Layered Transfer and the voyager implementation in terms of disk space consumption.

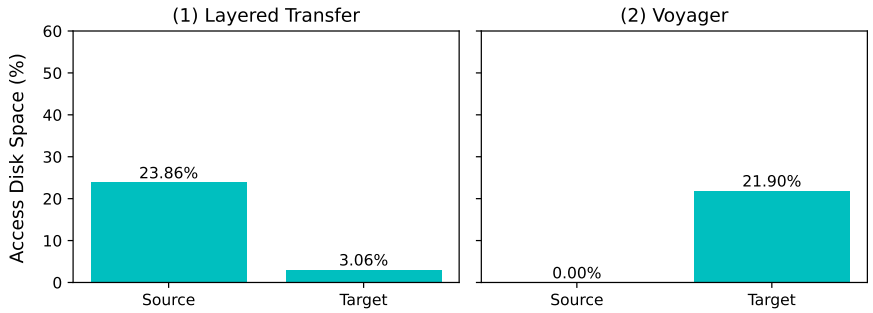

Figure 6.12: Additional Disk Space Consumption 1MB 1000 Files Only Random Write

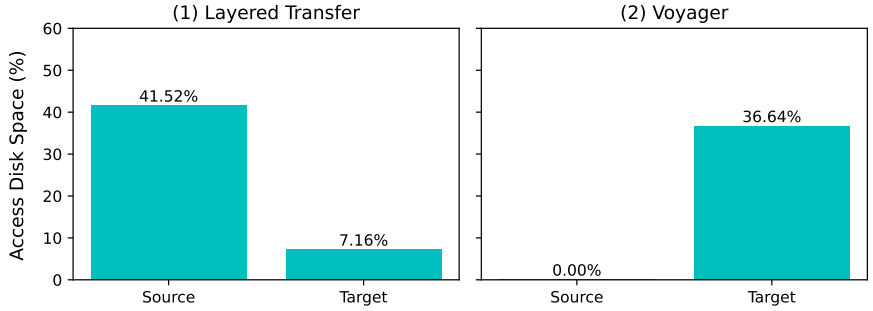

Figure 6.13: Additional Disk Space Consumption 2MB 500 Files Only Random Write

Since the additional disk space consumption is attributed to CoW operations, the difference between the size of the layers and the size of the volume grows with larger files. For example Figure 6.13 shows, when increasing the initial file size from 1 MB to 2 MB, the average access disk space consumption raises up to 7.2% on the target and to 41.5% on the source machine for the layered transfer. For the voyager migration, the access disk space consumption increases to 36.6% on the target machine.

When decreasing the file size to 500KB, the access disk space consumption drops to 13.6% on the source and to 1.3% on the target for the Layered Transfer migration, as observed in Figure 6.14. For the Voyager migration, it drops to 13.5%. In this case, the additional disk space consumption is approximately the same for the Layered Transfer migration and the voyager migration, while for the other cases, the Layered Transfer migration consumed more disk space on average.

6.8 DOWNTIME

In this section, the downtime of the service during the migration is compared. In the experiment setup, the reverse proxy will respond with a 502 HTTP error, when the service it redirects the traffic to, is

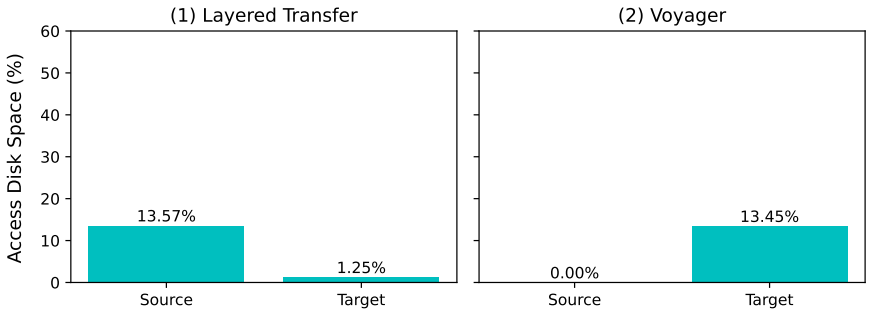

Figure 6.14: Additional Disk Space Consumption 500KB 2000 Files Only Random Write

not available. The downtime is measured by counting the number of requests that result in a 502 HTTP error and dividing this number by the number of total requests sent. Naturally, since the Layered Transfer and the Voyager migration are designed to be live migrations, the service is fully available during the migration. In the conducted experiments, the calculated downtime for both approaches is 0%. For all the conducted experiments, no single request resulted in a 502 response for the Layered Transfer approach and the voyager approach. The cold migration, on the other hand, is in all conducted experiments unavailable for 100%.

6.9 REAL WORLD APPLICATION PROFILES

The presented experiment configurations only-read, only-sequential, only-random and only-new, which have been introduced in the former sections of this chapter, have been chosen to create a clear overview on which type of request influences which KPI in which magnitude. However, it is not very likely that an application exclusively handles requests of one kind. Therefore, the experiments read-heavy and write-heavy, as introduced in Table 5.1, are conducted. The read-heavy experiment typifies an application that reads from the file system most of the time, with some write operations in between. This could for example be a database with customer records. The write-heavy experiment, typifies an application that mainly performs writer operations. An example for this is a logging database or a sensor database, which perform write operations most of the time with few reads in between.

Conducting the experiments with the read-heavy profile results in approximately the same values as the only-read experiments. No significant deviations are found. The outcomes, resulting from executing the write-heavy experiments, result in approximately the same values as the only-sequential experiments. Like for read-heavy, no significant

deviations are observed.

7 DISCUSSION

This section of the thesis discusses the results, that have been introduced in Chapter 6.

As introduced in Section 6.2, the migration time does not change significantly for any profile and any migration method, up to a file size of files to be migrated of 1 MB. With larger files, as tested for 2 MB, the migration time raises significantly when using the Layered Transfer approach, for the profiles only-read and only-random. For the only-random profile, the growing migration time can be attributed to the fact that the service runs on the source machine during the most time of the migration. This leads to more CoW operations on the source side, which produces more data that has to be copied to the target machine eventually. In general, more time is required, for copying more data. In addition to that, the migration time of the only-read profile also increases with growing file sizes. However, it is not completely clear, where this increase of migration time can be attributed to, since the only-read profile does not increase the amount of data to be transferred in any way. It can be argued, that the voyager migration is better suited for scenarios where the migration time has to be kept as low as possible. Especially, when the file size of the files to be migrated is larger than 1 MB.

When investigating the read latency, it is evident from the measurements in Section 5.3, that the Layered Transfer migration approach has an advantage in read latency in comparison to the voyager migration. Due to the fact that the service needs to read most of the data remotely from the source container, reading leads to a higher latency during the voyager migration, than during the Layered Transfer migration. Therefore, it can be argued, that the Layered Transfer migration approach, is preferred for migrations of services with a read heavy profile over the voyager migration. For write heavy applications, no significant difference between the methods could be observed.

Section 6.4 introduces the CPU utilization during the different types of migration. From the collected data, it is evident that the Layered Transfer migration has a lower CPU utilization across all profiles across the source and the target machine combined. However, the largest difference can be observed on the target machine. This can be attributed to the following phenomenon. Both, the voyager migration and the Layered Transfer migration perform two main tasks: copying

the initial files from the source to the target machine, and remotely reading files on the target machine from the source machine. During the voyager migrating, both of these tasks are performed at the same time. The Layered Transfer migration, on the other hand, first performs the largest part of the copying task, and executing the remote reading task afterward. This means the Layered Transfer migration performs the tasks in a more sequential way than the voyager migration, which performs both tasks in parallel. This leads to a lower CPU utilization for the Layered Transfer migration. This effect might even be increased when migrating to machines with a weaker CPU configuration, then tested in the experiments. In summary, the Layered Transfer migration is preferred over the voyager migration for CPU critical scenarios.

As introduced in Section 6.6, the Layered Transfer migration transmits less data in total than the voyager migration for the only-read profile. Since the voyager migration performs more remote reads, the additional data, requested by the service, is both added to the source outgoing traffic and the target incoming traffic. For a file size of 2 MB (Figure 6.11) can be observed that even the migration time is lower for the voyager, and therefore there are less reading requests, the total amount of send data between the source and the target machine, is the same for the Layered Transfer - and the voyager migration. For the profiles only-sequential, only-random and only-new, no significant difference could be observed. Since less data per request is transmitted using the Layered Transfer migration compared to the voyager migration, the Layered Transfer migration is regarded as more data saving, and therefore recommended for migrations in bandwidth critical scenarios.

In Section 6.7, the additional consumed disk space is introduced. Since the CoW operation duplicates data on the disk, only the profile only-random makes a difference in the measured data. The other profiles do not trigger the CoW operation at all, or in a neglectable amount. In general, the total amount of consumed disk space seems to grow with the size of the migrated files. This is reasonable, since only whole files are copied by the CoW operation. This means, CoW on larger files consumes more additional disk space. This holds for both, the voyager and the Layered Transfer migration. While the Layered Transfer migration consumes most additional disk space on the source machine and some on the target machine, the voyager migration consumes all additional disk space on the target machine. Therefore, the total amount of additional consumed disk space is consumed by the Layered Transfer migration, with a small difference to the voyager migration. However, the recommendation for disk space critical migrations depends on the disk space of the target and source machine. Scenarios where the disk space is critical on the source machine are advised to use the voyager migration, while scenarios with critical disk space on the target ma-

chine are advised to use the Layered Transfer migration. Scenarios in which disk space is sparse in general are advised to use the voyager or the cold approach, since the total amount of additional disk space is slightly lower.

In Section 6.9, the results for the real-world application profiles are introduced. It can be observed, that the read-heavy experiments yield approximately the same results as the only-read experiments, while the write-heavy experiments yield approximately the same results as the only-sequential experiments. Therefore, the occasional writes, which are executed during the migration in the read-heavy experiments, are not found to have an influence on the general outcome of the KPI measurements. The same holds for the write-heavy experiments in respect to the occasional reads.

The observed decrease of migration performance for the KPIs latency, network traffic and disk space consumption, is attributed to the fact that CoW operations need to be performed during the migration. This holds for both the Voyager and the Layered Transfer migration. Therefore, it might not be beneficial to migrate a volume with very large files (>100 MB) using one of these approaches. In a scenario where a single byte in a 100 MB file is changed, the CoW operation will produce 100 MB of access data. In such a scenario, it is more advantageous to employ a different migration strategy, or to replace the CoW mechanism by a different approach.

8 CONCLUSION

In Table 8.1 it is shown how the Layered Transfer migration approach improves the voyager approach in respect to the investigated KPIs. This table is considered the summarized answer to research questions **RQ1** and **RQ3**. In general, one can conclude that the Layered Transfer migration uses less CPU resources than the voyager approach. Additionally, less data in total is consumed by the Layered Transfer approach. Furthermore, for specific cases like read heavy applications, the service has a lower latency during the Layered Transfer migration. The KPIs additional disk space consumption, in general, could not be improved on. However, the Layered Transfer migration uses less disk space on the target machine than the Voyager approach. The migration time on the other hand could not be improved by the Layered Transfer migration. In all conduced experiments, no downtime could be measured. Therefore, the voyager approach and the Layered Transfer migration are considered equally available.

Summarizing the findings from the research, it is concluded that the cold migration performs slightly better in comparison to the Layered Transfer migration. However, it does introduce downtime to the application, which is not the case for the Layered Transfer migration. This downtime decreases the quality of service of the application, especially when migrating multiple times a day. This answers the second research question, **RQ2**.

This research contributes to the research field of live migration by introducing a novel approach of stateful live migration of container-

KPI		Description
Migration Time	-	slightly slower
Latency	+/-	decreased for read, increased/identical for random write
CPU Utilization	+	decreased
Memory Usage	=	equal
Network Traffic	+	decreased for read
Additional Disk Space	-/+	decreased in total, increased for on target
Downtime	=	no measured downtime

Table 8.1: Improvements per KPI

ized applications. It introduces and analyses the different strengths and weaknesses in comparison to the state of the art. The evaluation of the process serves as exploration of the strength and weaknesses. In addition to that, it provides recommendation for practitioners on which live migration approach to use in which scenario. Furthermore, the research provides an implementation for the Voyager migration and the Layered Transfer migration approaches, that can be used and adapted by researchers and practitioners.

8.1 TECHNICAL LIMITATIONS

Regarding the design and the implementation of the Layered Transfer migration approach, the following technical limitations emerge. As the Layered Transfer migration is constructed, it does not allow for a live migration of a volume, that is accessed by a container that keeps files constantly open. In order to make sure that all writes are consistently transmitted to the target machine, the migration process needs to make sure that the files to be copied are not open and will therefore not be changed in the future. This limitation can mitigated by communicating to the service to release the file and make sure it is not altered after its layer is copied. However, the application needs to support such functionality.
In addition to that, the Layered Transfer migration process is constructed to migrate stateful services that provide control over their writes. In the conducted experiments, writes are triggered by an external HTTP request, which is redirected to the target machine using a reverse proxy. This makes the writes controllable from the outside. However, some applications perform writes independent of outside triggers, like indexing or removing temporary files. If these writes happen during the migration, there is a chance that they are not transmitted to the target machine and the migration is inconsistent. However, this is mitigated by the fact that the container is started with the exact same persistent data on the target machine. This will lead to triggering the internal writing eventually on the target machine. Ideally, such internally triggered are controllable from the outside of the application. Then they can be disabled during migrations, which mitigates this limitation. Since the current implementation is based on CoW operations, it is not suitable to migrate a volume with very large files. The larger the files get, the more data is duplicated during a CoW operation. In order to mitigate this, the CoW mechanism has to be replaced. Furthermore, replacing the CoW mechanism improves the KPI measurement results.

8.2 LIMITATIONS OF THE EVALUATION

A limitation of the evaluation of the implementation is the humongous amount and range of input parameters. The parameters initial file size, number of initial files and the behavior profile of the application have been taken into account. However, the scope of this research is only large enough to explore a small range of the named parameters. In addition to that, there are more parameters, that can potentially influence the outcome of the results. Some of these parameters are CPU power, network configuration, bandwidth, number of created checkpoints during the migration and the efficiency of the implementation of the migration process itself. Naturally, this does not invalidate but complement the presented results.

8.3 OUTLOOK

Future work could be focused on mitigating the overhead, that emerges by the CoW operations. This could be done by chunking the files and only copying the edited chunk to the upper layer. A different approach is to use a union file system, that supports move-on-write operations, that move the file to the upper layer instead of copying it. This would completely remove the necessity of additional disk space. In addition to that, conducing a more in depth exploration and analysis of the implemented migration approach, with a broader range of parameters, can be conducted. In the current research, the live migration has been implemented and evaluated, using HTTP requests as write triggers. However, services like SQL databases require different access protocols. Future research could focus on implementing the Layered Transfer migration for stateful TCP connections. This would connect this topic to the research on migrating containers including their active TCP connections, conducted by Yu et al. [20]in 2023. A different kind of related research could design and investigate a mechanism to "leave a layer behind" on machines. In scenarios, where a container is temporarily migrated to a different machine, it might be beneficial to leave the lower layers behind on the original machine and reuse them when the container is migrated back. Therefore, only the upper layer from the new machine has to be copied, since the lower layers will already be present on the original machine. Of course, this works best for read heavy applications, that perform few CoW operations, which basically make the existence of files in the lower layers obsolete.

A

APPENDIX

In order to provide all resources, used for the research, this appendix chapter is added.

A.1 EXPERIMENT MEASUREMENTS

This section complements the thesis document by adding the full dataset of the measured KPIs in table form. For every experiment type, the table displays the mean of 5 identical experiment runs. The following sections are ordered by their application profile configuration.

A.1.1 *Only Read*

Migration Method	cold	layered transfer	voyager
number_of_initial_files	1000.00	1000.00	1000.00
initial_file_size	1000000.00	1000000.00	1000000.00
migration_time	25.80	29.88	29.08
mean_cpu_source	8.52	11.68	12.18
mean_cpu_target	11.57	12.87	17.38
mean_cpu	10.05	12.27	14.78
netin_target	1005964084.80	1007078077.80	1322804003.80
netout_source	1058930921.80	1343226252.40	1391262913.20
mean_read	20.82	20.02	39.41
mean_sequential_write	NaN	NaN	NaN
mean_random_write	NaN	NaN	NaN
mean_new_write	NaN	NaN	NaN
mean_mem_source	2835616301.20	2835616301.20	2835658335.40
mean_mem_target	2808330343.90	2808254229.20	2808254229.20
missed_requests	251.40	0.00	0.00
additional_disk_space_source	1.00	1.00	1.00
additional_disk_space_target	0.00	1.00	1.00
total_requests	355.00	397.60	389.00

Table A.1: Only Read 1000 Files 1 MB

Migration Method	cold	layered transfer	voyager
number_of_initial_files	500.00	500.00	500.00
initial_file_size	2000000.00	2000000.00	2000000.00
migration_time	25.63	42.45	29.42
mean_cpu_source	7.86	11.81	12.27
mean_cpu_target	10.47	9.51	18.87
mean_cpu	9.17	10.66	15.57
netin_target	1005788609.80	1007214607.80	1547024644.20
netout_source	1142978759.60	2205732690.80	1759492294.40
mean_read	134.21	149.71	132.61
mean_sequential_write	NaN	NaN	NaN
mean_random_write	NaN	NaN	NaN
mean_new_write	NaN	NaN	NaN
mean_mem_source	2846027627.00	2846027627.00	2846027627.00
mean_mem_target	2805570705.26	2805561240.94	2805595626.40
missed_requests	249.40	0.00	0.00
additional_disk_space_source	1.00	1.00	1.00
additional_disk_space_target	0.00	1.00	1.00
total_requests	353.20	522.60	392.60

Table A.2: Only Read 500 Files 2 MB

Migration Method	cold	layered transfer	voyager
number_of_initial_files	2000.00	2000.00	2000.00
initial_file_size	500000.00	500000.00	500000.00
migration_time	27.25	31.21	34.07
mean_cpu_source	8.30	10.95	12.35
mean_cpu_target	11.67	13.24	16.54
mean_cpu	9.99	12.09	14.45
netin_target	1006963093.00	1008044645.80	1258692991.80
netout_source	1034375891.60	1182412803.60	1302928765.20
mean_read	14.87	14.75	33.59
mean_sequential_write	NaN	NaN	NaN
mean_random_write	NaN	NaN	NaN
mean_new_write	NaN	NaN	NaN
mean_mem_source	2846027627.00	2846027627.00	2846027627.00
mean_mem_target	2805635264.00	2805655855.00	2805717628.00
missed_requests	263.80	0.00	0.00
additional_disk_space_source	1.00	1.00	1.00
additional_disk_space_target	0.00	1.00	1.00
total_requests	369.60	410.60	439.20

Table A.3: Only Read 2000 Files 0.5 MB

A.1.2 *Only Sequential*

Migration Method	cold	layered transfer	voyager
number_of_initial_files	1000.00	1000.00	1000.00
initial_file_size	1000000.00	1000000.00	1000000.00
migration_time	28.49	29.86	26.05
mean_cpu_source	7.70	9.14	9.54
mean_cpu_target	10.78	12.39	13.85
mean_cpu	9.24	10.77	11.69
netin_target	1005879605.60	1007939465.60	1007579997.20
netout_source	1005878318.00	1008066076.60	1007419744.00
mean_read	NaN	NaN	NaN
mean_sequential_write	5.80	6.78	6.28
mean_random_write	NaN	NaN	NaN
mean_new_write	NaN	NaN	NaN
mean_mem_source	2844359280.56	2844161837.00	2844161837.00
mean_mem_target	2805335150.40	2805370544.69	2805498849.00
missed_requests	278.20	0.00	0.00
additional_disk_space_source	1.00	1.00	1.00
additional_disk_space_target	0.00	1.00	1.00
total_requests	382.20	397.40	358.80

Table A.4: Only Sequential 1000 Files 1 MB

Migration Method	cold	layered transfer	voyager
number_of_initial_files	500.00	500.00	500.00
initial_file_size	2000000.00	2000000.00	2000000.00
migration_time	23.72	26.74	23.36
mean_cpu_source	8.13	9.31	9.38
mean_cpu_target	12.27	13.39	14.32
mean_cpu	10.20	11.35	11.85
netin_target	1005441445.60	1008388068.20	1006760297.60
netout_source	1005436387.20	1008495358.40	1006607569.00
mean_read	NaN	NaN	NaN
mean_sequential_write	11.53	6.09	7.07
mean_random_write	NaN	NaN	NaN
mean_new_write	NaN	NaN	NaN
mean_mem_source	2841475559.51	2841545213.20	2841545213.20
mean_mem_target	2806052418.74	2805954481.47	2805726378.80
missed_requests	231.80	0.00	0.00
additional_disk_space_source	1.00	1.00	1.00
additional_disk_space_target	0.00	1.00	1.00
total_requests	334.80	366.20	332.00

Table A.5: Only Sequential 500 Files 2 MB

Migration Method	cold	layered transfer	voyager
number_of_initial_files	2000.00	2000.00	2000.00
initial_file_size	500000.00	500000.00	500000.00
migration_time	34.49	32.96	28.93
mean_cpu_source	6.84	8.93	9.43
mean_cpu_target	10.19	12.75	14.03
mean_cpu	8.51	10.83	11.73
netin_target	1006891307.20	1008470858.40	1009119824.20
netout_source	1006893452.00	1008613049.00	1008949628.60
mean_read	NaN	NaN	NaN
mean_sequential_write	12.02	9.00	6.30
mean_random_write	NaN	NaN	NaN
mean_new_write	NaN	NaN	NaN
mean_mem_source	2834687875.29	2833191747.80	2832828399.00
mean_mem_target	2806468170.00	2806587022.59	2806575758.00
missed_requests	327.60	0.00	0.00
additional_disk_space_source	1.00	1.00	1.00
additional_disk_space_target	0.00	1.00	1.00
total_requests	441.80	428.00	388.00

Table A.6: Only Sequential 2000 Files 0.5 MB

A.1.3 *Only Random*

Migration Method	cold	layered transfer	voyager
number_of_initial_files	1000.00	1000.00	1000.00
initial_file_size	1000000.00	1000000.00	1000000.00
migration_time	24.34	33.83	29.32
mean_cpu_source	8.46	11.11	11.52
mean_cpu_target	11.74	14.41	16.17
mean_cpu	10.10	12.76	13.85
netin_target	1005882577.20	1247753492.20	1273696514.00
netout_source	1005877020.00	1247832359.80	1273507148.00
mean_read	NaN	NaN	NaN
mean_sequential_write	NaN	NaN	NaN
mean_random_write	6.71	220.89	183.98
mean_new_write	NaN	NaN	NaN
mean_mem_source	2845050333.00	2845050333.00	2845050333.00
mean_mem_target	2806755684.80	2806755684.80	2806755684.80
missed_requests	237.40	0.00	0.00
additional_disk_space_source	1.00	1.23	1.00
additional_disk_space_target	0.00	1.04	1.22
total_requests	340.80	436.60	391.80

Table A.7: Only Random 1000 Files 1 MB

Migration Method	cold	layered transfer	voyager
number_of_initial_files	500.00	500.00	500.00
initial_file_size	2000000.00	2000000.00	2000000.00
migration_time	26.50	37.58	27.95
mean_cpu_source	7.86	11.00	11.10
mean_cpu_target	10.77	14.35	16.26
mean_cpu	9.32	12.68	13.69
netin_target	1005441240.60	1455736038.80	1397200600.00
netout_source	1005435812.60	1455790061.80	1397038180.80
mean_read	NaN	NaN	NaN
mean_sequential_write	NaN	NaN	NaN
mean_random_write	6.55	408.52	335.59
mean_new_write	NaN	NaN	NaN
mean_mem_source	2840849286.02	2842108996.20	2842108996.20
mean_mem_target	2806121968.65	2806184072.68	2806035759.00
missed_requests	258.40	0.00	0.00
additional_disk_space_source	1.00	1.42	1.00
additional_disk_space_target	0.00	1.07	1.37
total_requests	362.00	474.20	378.40

Table A.8: Only Random 500 Files 2 MB

Migration Method	cold	layered transfer	voyager
number_of_initial_files	2000.00	2000.00	2000.00
initial_file_size	500000.00	500000.00	500000.00
migration_time	26.20	34.72	33.88
mean_cpu_source	8.61	11.15	12.04
mean_cpu_target	12.50	14.78	17.07
mean_cpu	10.56	12.96	14.55
netin_target	1006883080.00	1146861621.20	1241801740.00
netout_source	1006878003.20	1146968151.80	1241590527.60
mean_read	NaN	NaN	NaN
mean_sequential_write	NaN	NaN	NaN
mean_random_write	8.46	200.89	169.60
mean_new_write	NaN	NaN	NaN
mean_mem_source	2843412356.49	2843182442.60	2843182442.60
mean_mem_target	2806047084.00	2806047084.00	2806047084.00
missed_requests	253.40	0.00	0.00
additional_disk_space_source	1.00	1.14	1.00
additional_disk_space_target	0.00	1.01	1.13
total_requests	359.20	445.60	437.00

Table A.9: Only Random 2000 Files 0.5 MB

A.1.4 *Only New*

Migration Method	cold	layered transfer	voyager
number_of_initial_files	1000.00	1000.00	1000.00
initial_file_size	1000000.00	1000000.00	1000000.00
migration_time	26.78	29.74	27.20
mean_cpu_source	8.11	9.64	9.12
mean_cpu_target	11.35	13.14	14.05
mean_cpu	9.73	11.39	11.58
netin_target	1005952321.60	1007370067.60	1007380980.00
netout_source	1005951633.20	1007481990.00	1007226846.60
mean_read	NaN	NaN	NaN
mean_sequential_write	NaN	NaN	NaN
mean_random_write	NaN	NaN	NaN
mean_new_write	5.91	5.71	6.69
mean_mem_source	2840461220.65	2840812769.20	2840812769.20
mean_mem_target	2805540031.00	2805550048.24	2805581213.00
missed_requests	259.40	0.00	0.00
additional_disk_space_source	1.00	1.00	1.00
additional_disk_space_target	0.00	1.00	1.00
total_requests	365.20	395.80	370.60

Table A.10: Only New 1000 Files 1 MB

Migration Method	cold	layered transfer	voyager
number_of_initial_files	500.00	500.00	500.00
initial_file_size	2000000.00	2000000.00	2000000.00
migration_time	26.97	28.42	23.86
mean_cpu_source	7.58	9.12	9.32
mean_cpu_target	11.21	13.16	14.36
mean_cpu	9.39	11.14	11.84
netin_target	1005514236.00	1006786658.40	1006703578.40
netout_source	1005513044.60	1006897735.80	1006556183.00
mean_read	NaN	NaN	NaN
mean_sequential_write	NaN	NaN	NaN
mean_random_write	NaN	NaN	NaN
mean_new_write	22.60	6.80	6.50
mean_mem_source	2836775196.52	2841413552.94	2841505806.60
mean_mem_target	2806790934.00	2806790934.00	2806790934.00
missed_requests	259.00	0.00	0.00
additional_disk_space_source	1.00	1.00	1.00
additional_disk_space_target	0.00	1.00	1.00
total_requests	366.60	382.80	337.00

Table A.11: Only New 500 Files 2 MB

Migration Method	cold	layered transfer	voyager
number_of_initial_files	2000.00	2000.00	2000.00
initial_file_size	500000.00	500000.00	500000.00
migration_time	27.93	32.47	28.06
mean_cpu_source	8.05	9.15	9.40
mean_cpu_target	12.19	13.38	14.66
mean_cpu	10.12	11.26	12.03
netin_target	1006967069.00	1008433264.80	1008572925.60
netout_source	1006962408.40	1008560329.80	1008399966.60
mean_read	NaN	NaN	NaN
mean_sequential_write	NaN	NaN	NaN
mean_random_write	NaN	NaN	NaN
mean_new_write	5.68	7.68	8.21
mean_mem_source	2834684780.65	2833080395.52	2837141822.60
mean_mem_target	2806790934.00	2806790934.00	2806790934.00
missed_requests	273.20	0.00	0.00
additional_disk_space_source	1.00	1.00	1.00
additional_disk_space_target	0.00	1.00	1.00
total_requests	377.00	423.00	379.20

Table A.12: Only New 2000 Files 0.5 MB

A.1.5 *Read Heavy*

Migration Method	cold	layered transfer	voyager
number_of_initial_files	1000.00	1000.00	1000.00
initial_file_size	1000000.00	1000000.00	1000000.00
migration_time	22.58	28.05	29.37
mean_cpu_source	8.96	11.25	11.32
mean_cpu_target	11.93	12.78	16.16
mean_cpu	10.44	12.02	13.74
netin_target	1005972869.40	1020228321.20	1265468316.60
netout_source	1048807305.80	1269055683.80	1332599601.60
mean_read	18.78	19.39	41.20
mean_sequential_write	5.34	9.70	16.80
mean_random_write	6.54	255.46	254.98
mean_new_write	5.06	6.80	8.42
mean_mem_source	2839058656.18	2840896395.20	2840896395.20
mean_mem_target	2803926244.39	2804157940.00	2804157940.00
missed_requests	220.00	0.00	0.00
additional_disk_space_source	1.00	1.01	1.00
additional_disk_space_target	0.00	1.00	1.02
total_requests	323.20	379.20	392.20

Table A.13: Read Heavy 1000 Files 1 MB

Migration Method	cold	layered transfer	voyager
number_of_initial_files	500.00	500.00	500.00
initial_file_size	2000000.00	2000000.00	2000000.00
migration_time	27.61	39.54	27.19
mean_cpu_source	6.52	11.05	11.50
mean_cpu_target	9.16	10.37	18.63
mean_cpu	7.85	10.71	15.06
netin_target	1005749534.80	1042218305.60	1446657983.20
netout_source	1122223585.20	1912318106.40	1594921505.00
mean_read	125.88	145.14	134.79
mean_sequential_write	5.05	11.64	11.88
mean_random_write	5.63	428.96	370.51
mean_new_write	5.33	8.09	5.62
mean_mem_source	2837006917.53	2836544987.80	2840944178.80
mean_mem_target	2804339372.53	2804263902.20	2804263902.20
missed_requests	266.00	0.00	0.00
additional_disk_space_source	1.00	1.03	1.00
additional_disk_space_target	0.00	1.00	1.02
total_requests	373.20	493.40	370.40

Table A.14: Read Heavy 500 Files 2 MB

Migration Method	cold	layered transfer	voyager
number_of_initial_files	2000.00	2000.00	2000.00
initial_file_size	500000.00	500000.00	500000.00
migration_time	28.86	29.99	33.14
mean_cpu_source	7.82	10.79	11.85
mean_cpu_target	11.12	13.08	15.91
mean_cpu	9.47	11.93	13.88
netin_target	1006943538.00	1015069253.20	1210221081.20
netout_source	1029353233.00	1148415362.80	1244689435.20
mean_read	14.15	14.55	34.83
mean_sequential_write	5.56	13.29	12.97
mean_random_write	7.81	152.52	508.82
mean_new_write	7.59	6.58	6.31
mean_mem_source	2833522937.30	2837570382.12	2840587113.40
mean_mem_target	2804816345.00	2804816345.00	2804816345.00
missed_requests	277.40	0.00	0.00
additional_disk_space_source	1.00	1.01	1.00
additional_disk_space_target	0.00	1.00	1.01
total_requests	385.80	398.60	430.00

Table A.15: Read Heavy 2000 Files 0.5 MB

A.1.6 *Write Heavy*

Migration Method	cold	layered transfer	voyager
number_of_initial_files	1000.00	1000.00	1000.00
initial_file_size	1000000.00	1000000.00	1000000.00
migration_time	25.28	29.82	26.21
mean_cpu_source	8.33	9.92	10.12
mean_cpu_target	11.27	12.54	14.38
mean_cpu	9.80	11.23	12.25
netin_target	1005902488.40	1032434699.20	1064000269.60
netout_source	1011897761.40	1062595415.40	1068983526.00
mean_read	18.78	19.94	37.61
mean_sequential_write	5.72	7.54	7.48
mean_random_write	6.87	184.68	243.92
mean_new_write	12.27	7.76	6.02
mean_mem_source	2840569384.48	2841991825.60	2841991825.60
mean_mem_target	2805997588.46	2806055268.50	2805728396.20
missed_requests	243.20	0.00	0.00
additional_disk_space_source	1.00	1.03	1.00
additional_disk_space_target	0.00	1.00	1.02
total_requests	350.40	397.00	360.80

Table A.16: Write Heavy 1000 Files 1 MB

Migration Method	cold	layered transfer	voyager
number_of_initial_files	500.00	500.00	500.00
initial_file_size	2000000.00	2000000.00	2000000.00
migration_time	24.33	31.36	24.98
mean_cpu_source	8.20	9.45	10.01
mean_cpu_target	11.53	11.94	14.60
mean_cpu	9.87	10.70	12.30
netin_target	1005485071.80	1064148135.00	1092640727.60
netout_source	1018045953.20	1152858576.40	1106973398.00
mean_read	106.09	123.94	115.49
mean_sequential_write	6.12	10.41	6.97
mean_random_write	5.88	222.99	295.65
mean_new_write	5.57	8.36	7.26
mean_mem_source	2837739021.81	2841162537.99	2841591384.80
mean_mem_target	2804536009.00	2804483465.15	2804302480.80
missed_requests	237.40	0.00	0.00
additional_disk_space_source	1.00	1.06	1.00
additional_disk_space_target	0.00	1.01	1.04
total_requests	340.60	412.20	348.60

Table A.17: Write Heavy 500 Files 2 MB

Migration Method	cold	layered transfer	voyager
number_of_initial_files	500.00	500.00	500.00
initial_file_size	500000.00	500000.00	500000.00
migration_time	12.66	13.18	9.66
mean_cpu_source	4.82	5.83	6.33
mean_cpu_target	7.05	8.08	9.47
mean_cpu	5.93	6.96	7.90
netin_target	251778528.60	256737670.60	259813905.20
netout_source	254165627.60	265188748.80	262943513.00
mean_read	21.96	13.65	18.93
mean_sequential_write	5.71	8.17	6.32
mean_random_write	6.54	184.71	54.02
mean_new_write	6.49	6.05	6.54
mean_mem_source	2840409533.96	2838443479.20	2840271384.53
mean_mem_target	2803368368.00	2803368368.00	2803368368.00
missed_requests	120.40	0.00	0.00
additional_disk_space_source	1.00	1.02	1.00
additional_disk_space_target	0.00	1.00	1.01
total_requests	224.40	230.40	195.20

Table A.18: Write Heavy 2000 Files 0.5 MB

BIBLIOGRAPHY

[1] *AUFS*. URL: https://aufs.sourceforge.net/.

[2] Thad Benjaponpitak, Meatasit Karakate, and Kunwadee Sripanidkulchai. "Enabling Live Migration of Containerized Applications Across Clouds." In: *Proceedings - IEEE INFOCOM* 2020-July (July 2020), pp. 2529–2538. ISSN: 0743166X. DOI: 10.1109/INFOCOM41043.2020.9155403.

[3] Marco Iorio, Fulvio Risso, Alex Palesandro, Leonardo Camiciotti, and Antonio Manzalini. "Computing Without Borders: The Way Towards Liquid Computing." In: *IEEE Transactions on Cloud Computing* 11.3 (2022), pp. 2820 –2838. ISSN: 2168-7161. DOI: 10.1109/TCC.2022.3229163. URL: https://doi.org/10.1109/TCC.2022.3229163..

[4] Paulo Souza Junior, Daniele Miorandi, and Guillaume Pierre. "Stateful Container Migration in Geo-Distributed Environments." In: *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom* 2020-December (Dec. 2020), pp. 49–56. ISSN: 23302186. DOI: 10.1109/CLOUDCOM49646.2020.00005.

[5] Paulo Souza Junior, Daniele Miorandi, and Guillaume Pierre. "Good Shepherds Care for Their Cattle: Seamless Pod Migration in Geo-Distributed Kubernetes." In: *Proceedings - 6th IEEE International Conference on Fog and Edge Computing, ICFEC 2022* (2022), pp. 26–33. DOI: 10.1109/ICFEC54809.2022.00011.

[6] Kiranpreet Kaur, Fabrice Guillemin, and Francoise Sailhan. "Container placement and migration strategies for cloud, fog, and edge data centers: A survey." In: *International Journal of Network Management* 32.6 (Nov. 2022). ISSN: 10991190. DOI: 10.1002/NEM.2212.

[7] Kiranpreet Kaur, Fabrice Guillemin, and Francoise Sailhan. "Live migration of containerized microservices between remote Kubernetes Clusters." In: (2023), pp. 114–119. DOI: 10.1109/INFOCOMWKSHPS57453.2023.10225858{\"{i}}. URL: https://hal.science/hal-03466765v2.

[8] Tobias Kurze, Markus Klems, David Bermbach, Alexander Lenk, Stefan Tai, and Marcel Kunze. "Cloud Federation." In: *The Second International Conference on Cloud Computing*. 2011. ISBN: 9781612081533.

[9] Demetrio Laganà, Carlo Mastroianni, Michela Meo, and Daniela Renga. "Reducing the Operational Cost of Cloud Data Centers through Renewable Energy." In: (2018). DOI: 10.3390/a11100145. URL: www.mdpi.com/journal/algorithms.

[10] Craig A Lee, Robert B Bohn, and Martial Michel. "The NIST Cloud Federation Reference Architecture." In: (2020). DOI: 10.6028/NIST.SP.500-332. URL: https://doi.org/10.6028/NIST.SP.500-332.

[11] *Liqo*. URL: https://liqo.io/.

[12] Lele Ma, Shanhe Yi, Nancy Carter, and Qun Li. "Efficient Live Migration of Edge Services Leveraging Container Layered Storage." In: *IEEE Transactions on Mobile Computing* 18.9 (Sept. 2019), pp. 2020–2033. ISSN: 15580660. DOI: 10.1109/TMC.2018.2871842.

[13] *Matplotlib — Visualization with Python*. URL: https://matplotlib.org/.

[14] Shripad Nadgowda, Sahil Suneja, Nilton Bila, and Canturk Isci. "Voyager: Complete Container State Migration." In: (2017). DOI: 10.1109/ICDCS.2017.91.

[15] Moustafa Najm and Venkatesh Tamarapalli. "Towards cost-aware VM migration to maximize the profit in federated clouds." In: *Future Generation Computer Systems* 134 (2022), pp. 53–65. DOI: 10.1016/j.future.2022.03.020. URL: https://doi.org/10.1016/j.future.2022.03.020.

[16] *Overlay Filesystem — The Linux Kernel documentation*. URL: https://docs.kernel.org/filesystems/overlayfs.html.

[17] *P.Haul - CRIU*. URL: https://criu.org/P.Haul#Destination.

[18] *Paramiko documentation*. URL: https://www.paramiko.org/.

[19] *Project Jupyter Documentation — Jupyter Documentation 4.1.1 alpha documentation*. URL: https://docs.jupyter.org/en/latest/.

[20] Yenchia Yu, Antonio Calagna, Paolo Giaccone, and Carla Fabiana Chiasserini. "TCP Connection Management for Stateful Container Migration at the Network Edge." In: *Mediterranean Communication and Computer Networking Conference* (2023), pp. 151–157. DOI: 10.1109/MEDCOMNET58619.2023.10168849.

[21] Fei Zhang, Guangming Liu, Xiaoming Fu, and Ramin Yahyapour. "A Survey on Virtual Machine Migration: Challenges, Techniques, and Open Issues." In: *IEEE Communications Surveys and Tutorials* 20.2 (Apr. 2018), pp. 1206–1243. ISSN: 1553877X. DOI: 10.1109/COMST.2018.2794881.

[22] *lsof Documentation*. URL: https://manpages.ubuntu.com/manpages/bionic/en/man8/lsof.8.html.

[23] *mount(8) — mount — Debian testing — Debian Manpages*. URL: https://manpages.debian.org/testing/mount/mount.8.en.html.

[24] *pandas - Python Data Analysis Library*. URL: https://pandas.pydata.org/.

DECLARATION

Put your declaration here.

Groningen, August 2024

Patrick Lindner