POLITECNICO DI TORINO

Master's Degree in ICT for Smart Societies

Master's Degree Thesis

Container Migration for Mobile Services

Supervisors

Prof. Carla Fabiana CHIASSERINI

Prof. Paolo GIACCONE

Candidate

Yenchia YU

Academic Year 2021-2022

Abstract

Nowadays, the development of edge computing technologies enables the deployment of time-sensitive and large-data-volume services on infrastructure, which significantly promotes the development of the applications like connected vehicles and the intelligent industry. Together with the development of mobile communication technologies (e.g., 5G), edge computing architecture is gaining more and more traction, leading people to imagine the possibility of deploying time-critical mobile services on edge to reduce computational power requirements on mobile devices, especially in application scenarios like autonomous driving or unmanned aerial vehicles (UAVs).

However, in actual practice, the deployment of time-critical mobile services on the edge server still has many challenges, especially the communication latency. When a mobile client moves, the communication latency between service and client will increase as the communication distance increases, which is mainly related to the unpredictable network congestion. In this case, the communication latency will no longer be guaranteed when the mobile client reaches a specific distance with respect to the fixed edge service. The most feasible solution to this problem is to enable the mobility of mobile services among the edge network, making the mobile services can always be reachable in the edge server close to the mobile client. Since edge services are normally containerized, service migration in the edge network could be generalized into container migration. Even though compared to the VM migration, container migration is very lightweight and flexible, the migration costs and benefits still need careful balancing in time-critical services migration.

In the work of this thesis, the study on container migration technology will focus on the following two aspects: container migration realization and migration cost modeling. In the part of container migration realization, general requirements and constraints for container migration will be discussed. And in practice, Podman container will be used to containerize services since it has better integration with CRIU, a key tool to realize container migration on Linux machines. In the part of migration cost modeling, the container migration cost will be studied from the aspect of service downtime and migration resource usage (including CPU, memory, storage, and network usage). An overall cost model will be proposed in order to define different migration policies for different kinds of edge applications.

Since container migration is a new study topic in the edge computing area, the study in this thesis will focus more on the low-layer migration implementation and performance. Future work could concentrate on the study of high-layer orchestration or the discussion of the extreme conditions on container migration, completing the technical stack and pushing the adoption of this technique in real applications.

Table of Contents

Chapter 1

Introduction

Nowadays, the development of edge computing technologies enables the deployment of time-sensitive and large-data-volume services on infrastructure, which significantly promotes the development of the applications like connected vehicles and the intelligent city. More specifically, the main concept of the edge computing is to allow the computation to be performed at the edge of the network, which can be any computing or network resources physically close to the client. As Figure 1.1 shows, in the edge computing paradigm, the clients act as both the data producer and consumer. They can request services and content from the cloud. More importantly, they can perform computing tasks from the edge/cloud. However, in this very general paradigm, the clients are mostly considered to be fixed and do not move by time. According to the definition in [1], the edge should be able to perform computing offloading, data storage, caching, and processing, as well as distribute request and delivery services from cloud to user.

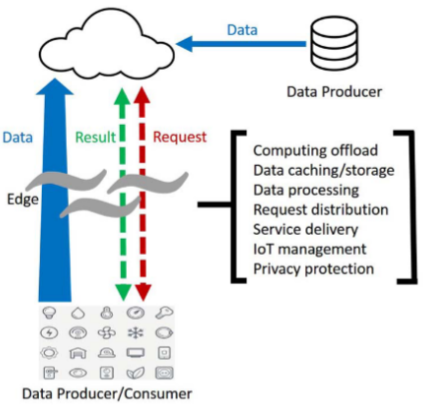


Figure 1.1: Edge computing paradigm [1]

With the development of mobile communication technologies (e.g., 5G), the study of edge computing architecture has gradually moved its target to support the deployment of mobile services. Generally speaking, a mobile service is the service that provides computational power and information from static servers to the mobile clients. Such design has the advantage of reducing the resource consumption on mobile devices but also has the weakness of high communication load and unstable communication performance. In recent study, people have started to consider the possibility of deploying time-critical mobile services on edge, especially for application scenarios like autonomous driving or unmanned aerial vehicles (UAVs).

Such kind of computational paradigm is normally considered as Mobile Edge Computing (MEC), which was first proposed by the European Telecommunication Standard Institute (ETSI) in 2014 [2]. The general architecture of MEC system is as Figure 1.2 shows. It contains two main key components: the mobile deceives and the MEC servers. The MEC servers are topically small-scale data centers deployed by the cloud and telecom operators in close proximity to end users and can be co-located with wireless APs [3]. Even though the general system seems quite simple, there still exist many challenges to be solved before implementing such kind of system.

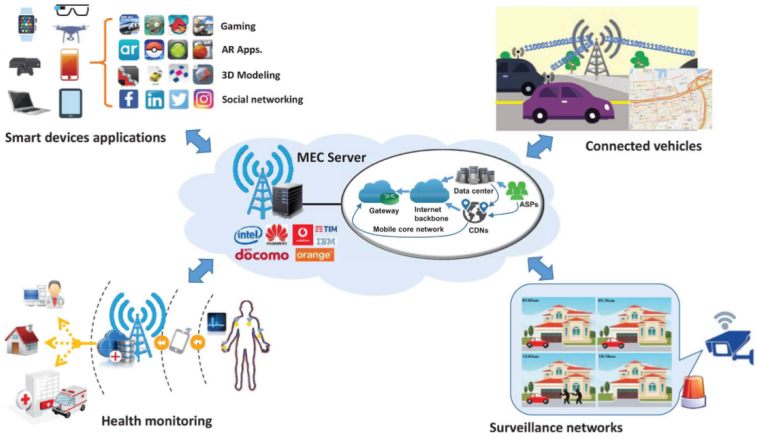


Figure 1.2: Architecture of the MEC system [3]

According to [3], the major challenges to implement the MEC system include four aspects. The first aspect is the deployment of the MEC system. The problem to be considered includes the site selection for MEC servers, MEC network architecture design, and server density planning. The second aspect is related to caching system management in MEC server. As mentioned in the previous part, the edge server is the mediator with limited resources between the client and the cloud. The way the mobile edge server caches the services and related data should be carefully designed in order to guarantee its availability and efficiency. The third aspect is mobility

management for MEC. Since the mobile device is moving, the corresponding service on the MEC server should be able to follow the trajectory of the user in order to guarantee and optimize the quality of service (QoS). And the last aspect of the challenge is related to energy saving.

1.1 Context and Problem Statement

In the current times, edge computing applications are normally deployed in a virtualized environment like containers. In this case, the study of the mobility of edge applications can be generalized into the mobility of containers. The purpose of this thesis is to validate the mobility of containers and research the container migration solution for mobile services working in edge networks like MEC systems. To be more specific, imagine that in a time-critical mobile application scenario like connected vehicles, one of its key QoS indexes is the communication latency between the mobile client and its corresponding services. In the current edge solution, the mobile service is deployed on a specific edge server. When the mobile client moves, the communication latency between service and client will increase as the communication distance increases, which is mainly related to unpredictable network congestion. In this case, the communication latency will no longer be guaranteed when the mobile client reaches a specific distance with respect to the fixed edge service. The most feasible solution to this problem is to enable the mobility of mobile services among the edge network, making the mobile services can always be reachable in the edge server close to the mobile client. Since edge services are normally containerized, service migration in the edge network could be generalized into container migration.

This thesis will discuss the performance and implementation of container migration based on the result of experiments and simulations. The study will cover the topic of container migration from realization to management stack. In the realization stack, the study will focus on file synchronization, service containerization, container checkpoint/restore, and container network migration. The major contribution in this stack is the adoption of overlay network architecture to solve container network migration problems. In the management stack, the contribution is the proposal of an optimization model to minimize the migration cost.

1.2 Thesis Structure

The work done in this thesis is presented according to the following organization.

Chapter 1 - Introduction

The first chapter introduces the high-level motivation of this thesis. It discusses the reason for the container migration study and lists this thesis's target.

Chapter 2 - State of the Art Review

The second chapter introduces the state of the art of container migration technologies. In this chapter, the difference between stateful and stateless migration is discussed. The techniques of stateful migration and their corresponding downtime model are dentally introduced. Meanwhile, a brief summary of recent studies on stateful container migration is also presented in this chapter, aiming to provide an overview of the study in this area.

Chapter 3 - Background Technologies

The third chapter introduces the background technologies used in this thesis. In this chapter, the general implementation and the main features of the tools in four categories are introduced: containerization, checkpoint/restore, file synchronization and software defined network.

Chapter 4 - Realization Stack: File synchronization

The fourth chapter demonstrates the result of the performance test on two selected file synchronization tools. In this chapter, the performance matrix to be compared includes the file deduplication rate, CPU usage, memory usage, storage usage, and network usage. According to the test result, the optimal file synchronization tool to be used in the container migration scenario is determined.

Chapter 5 - Realization Stack: Stateful Container Migration

The fifth chapter demonstrates the implementation detail and the result of stateful container migration. In this chapter, the implementation of the stateful application is firstly introduced. Then, the method and detail to containerize the stateful application are discussed. Then, the migrated state of the application with different migration methods is validated. In the end, the performance of different migration methods is compared

Chapter 6 - Realization Stack: Overlay Network

The sixth chapter demonstrates the implementation and test of an overlay network among containers during migration. In this chapter, the reason why container migration needs network management is firstly discussed. Then the container overlay network implementation detail is introduced. In the end, a common real edge service is used to test the container migration over the overlay network.

Chapter 7 - Management Stack: Migration Cost Model

The seventh chapter introduced a migration cost model for container migration management. In this chapter, a simple migration cost model based on the edge resources usage is created.

Chapter 8 - Management Stack: Migration Policies

The eighth chapter discusses the policies for container migration. In this chapter, the policies for the selection of migration techniques and migration strategies are introduced. Especially, the iterative pre-copy migration strategy is detailed introduced and validated.

Chapter 9 - Conclusion

The ninth chapter concludes the container migration topics presented in this thesis. In this chapter, the advantages and weaknesses of container migration are discussed. And future research directions in this area are proposed.

Chapter 2

State of the Art Review

This chapter provides a state-of-the-art review of the core concept of this thesis and the related works. Specifically, Section 2.1 reviews the main migration techniques, while Section 2.2 provides the corresponding migration downtime models. Section 2.3 summarizes the recent works related to service migration.

2.1 Category of Container Migration

The study of virtual environment migration techniques begins with virtual machines (VMs) migration. Migrating VMs between different physical hosts inside a data center could help achieve the goal of energy efficiency, load balancing, and high availability [4]. With the development of virtualization technologies, containers become more and more popular, especially in edge computing scenarios. Most of the key concepts of container migration techniques inherits from the VM migration. Since the containers are much "lightweight" than the VMs (see Section 3.1 for more details), the migration of containers is much faster and consumes much less resources than VM migration. However, migrating containers among resource limited edge servers still need carefully balancing the migration efficiency and the migration cost. So people proposed many different kinds of migration methods. In summary, the migration techniques can be catorgorized into two classes, which are stateless and stateful migration.

2.1.1 Stateless Container Migration

Stateless container migration means that the container losses its whole states during the migration procedure. Even though it sounds unreasonable, the stateless migration fit the nature of containers. In the original design, the containers are built to be stateless. It means that the containers should contain only the stateless

applications. By definition, a stateless application/process is a process that does not store any knowledge or references to the past transactions. Each transaction of the process is made as if it is from scratch by the first time. In this case, the process has no so called "state" to be transferred to the destination during the migration procedure. There only needs two simple steps to perform the stateless container migration: 1) Re-initiate the container from the same base image on the destination host, and 2) shut down the old container. However, in modern times, most of the applications are stateful. So even thought the stateless migration is very simple and efficient, it is less used in real practise.

2.1.2 Stateful Container Migration

Stateful container migration means that the container preserves all its state during the migration procedure and being able to continue working from its last state before migration after it is resorted on the destination host. On contrary to the stateless container migration, the container to be migrate should be stateful, which means that it should contains stateful applications. Otherwise, the stateful migration becomes meaningless. And by definition, a stateful application is the application that performs with the context of previous transitions and the current transaction may be affected by the previous ones. Since in stateful container migration, the majority of the states to be migrate is the memory states of the container/application. The realization of stateful migration normally needs the help of external tools to perform the checkpoint and restore operation. And according to different procedure to transfer the memory state of the application, the stateful container migration can again be classified into three basic classes, which are cold, pre-copy and post-copy migration.

Cold Migration

Cold migration is the technique to migrate a container with single checkpoint and synchronize procedure. As Figure 2.1 shows, there are three steps to perform the cold migration. First, checkpoint the container and shutdown the container. Second, synchronize the checkpoint image to the destination host. Third, restore the container from the checkpoint image. The reason why this migration method is considered to be "cold" is because that the original container is shut downed when the checkpoint procedure is finished. The container can not provide any service before it is restored. With respect to other "warm/live" migration procedure, such kind of migration is a "cold" procedure.

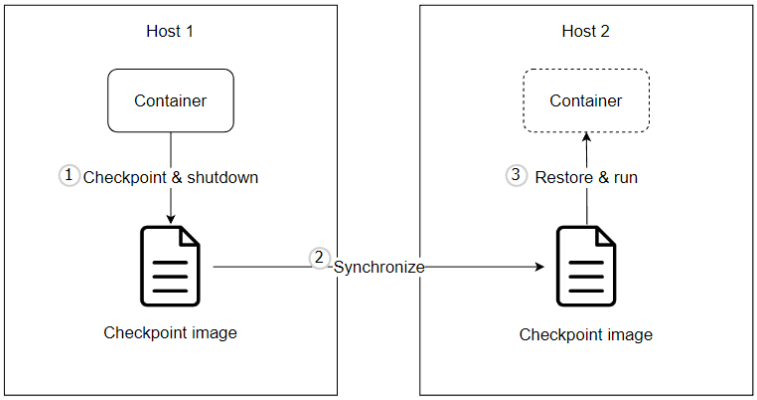


Figure 2.1: Cold migration procedure

Pre-copy Migration

Pre-copy migration is a live migration technique that migrates a container with twice checkpoint and synchronize procedure. As Figure 2.2 shows, there are five steps to perform the pre-copy migration. The first step is called pre-checkpoint. In this step, the external tool only checkpoints the container's memory content and leave the original container running. The second step is to synchronize the first checkpoint image to the destination. Then, in the third step, the external tool compares the current memory content with the previous checkpoint and finds out the dirty memory pages (the memory pages with changed content compared to the first checkpoint). After the dirty pages are find out, the external tool checkpoints the dirty pages and the remaining non-memory states and shutdown the container. The fourth step is to synchronize the second checkpoint image to the destination host. The last step is to restore the container on the destination host from the first and second checkpoint image. When the container is restored, the container continues working from the state before the second checkpoint. With pre-copy migration, the container can continue provide services until the second checkpoint. The duration when the application in the container is interrupted is much shorter than the cold migration. And step 1 to 4 can be iteratively repeated in order to reach an even smaller interruption time.

Post-copy Migration

Post-copy migration is a live migration technique that migrates a container with single checkpoint and synchronize procedure. As Figure 2.3 shows, there are eight steps to perform the post-copy migration. The first step is to suspend the container and perform the checkpoint. However, different from the cold and pre-copy migration, the post-copy migration technique only checkpoints a minimal subset

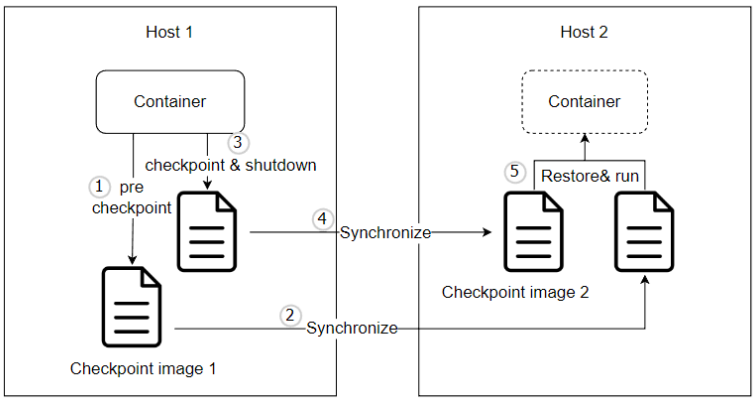


Figure 2.2: Pre-copy migration procedure

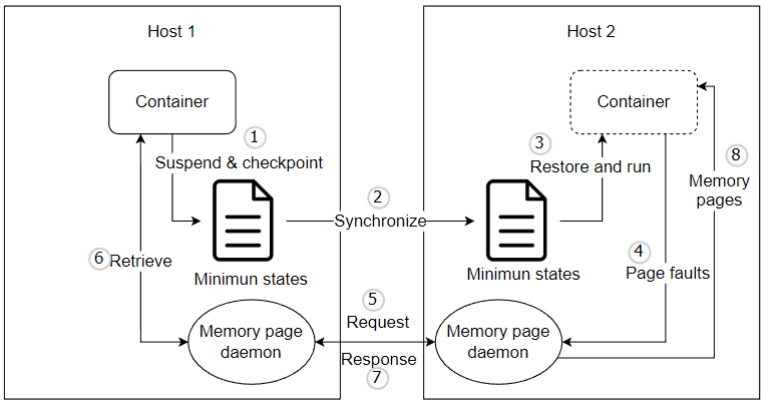


Figure 2.3: Post-copy migration procedure

of the execution state of the container (CPU state, registers and, optionally, non-pageable memory). In the second step, the minimum state image is synchronized to the destination host. In the third step, the container is restored form the minimum state checkpoint image. In this case, when the application inside the container accesses the non-transmitted memory pages, the process generates memory page faults. Instead of crashing the process, the memory page daemon running on the destination host catches the page faults (step 4) and sends requests of the memory pages to the memory page daemon on the original host (step 5). When the memory page daemon on the original host receives the memory page requests, it retrieves the target memory pages from the suspended container (step 6) and transmits them back to the memory page daemon on the destination host (step 7). When the destination memory page daemon has the required memory pages, it provides the memory pages to the container and consumes the memory page faults. Such

kind of migration technique can minimize the migration time. But the migration procedure is more complex. And if the restored container frequently accesses the non-synchronized memory pages, the performance of the restored container could decrease due the the frequent memory retrieve procedure.

2.2 Stateful Migration Time and Downtime

For the stateful container migration, the most important performance index are the migration time and migration downtime. The migration time is the total duration to finish the entire migration procedure. The migration downtime is defined as the duration of the interrupt of the containerized service due to container migration. For different stateful migration techniques, the migration downtime is different.

2.2.1 Cold Migration

According to [5], the detailed timeline of the cold migration procedure is as Figure 2.4 shows. The migration time is equal to the sum of checkpoint time T_c, transfer time T_t, and restore time T_r as Equation 2.1 shows. Due to the container is stopped when the migration procedure starts and is resumed when the migration procedure finished, the migration downtime of cold migration is equal to the migration time as Equation 2.2 shows.

$$T_{\text{migrate}} = T_c + T_t + T_r \tag{2.1}$$

$$T_{\text{down}} = T_{\text{migrate}} \tag{2.2}$$

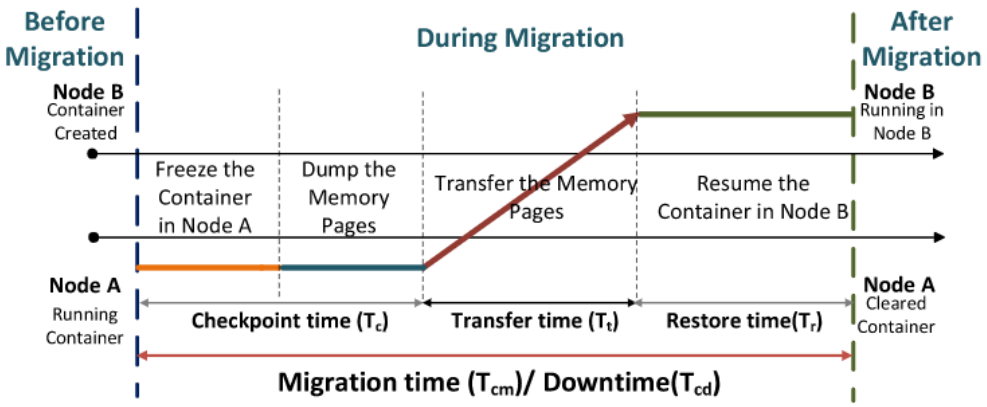


Figure 2.4: Cold migration timeline[5]

2.2.2 Pre-copy Migration

According to [6], the detailed timeline of the pre-copy migration procedure is as Figure 2.5 shows. This migration model was designed for VM migration. Since the migration of VM and container are consistent in the conceptual point of view, we can inherit the model to container migration. With the proposed migration procedure, the total migration time can be summarized as Equation 2.3 shows. In the equation,T_{pre} means the total duration of the pre-migration,initialization and reservation phase. And T_{mem} represents the iterative memory copy and stop-and-copy phases. T_{post} is the total duration of commitment, post-migration and activation phases. Comparing the the cold migration, the total migration time T_{migrate} of pre-copy migration is longer. On the other hand, the migration downtime of pre-copy migration is calculated as Equation 2.4 shows. T_d represents the duration to perform the stop and copy operation and transmit the dirty memory pages. $T_{\text{post}}^{'}$ means a part of post migration time in which the migrated container is not complete activated. As the result, the exact migration downtime of pre-copy migration is strongly dependent to the dirty page rate of the container to be migrate. Comparing to other migration method, the migration downtime of pre-copy migration can be the shortest when the dirty page rate of the container is relatively small. On the other hand, the pre-copy migration may consumes a large amout of network resources during the iterative memory copy phase.

$$T_{\text{migrate}} = T_{\text{pre}} + T_{\text{mem}} + T_{\text{post}} \tag{2.3}$$

$$T_{\text{down}} = T_d + T_{\text{post}}^{'} \tag{2.4}$$

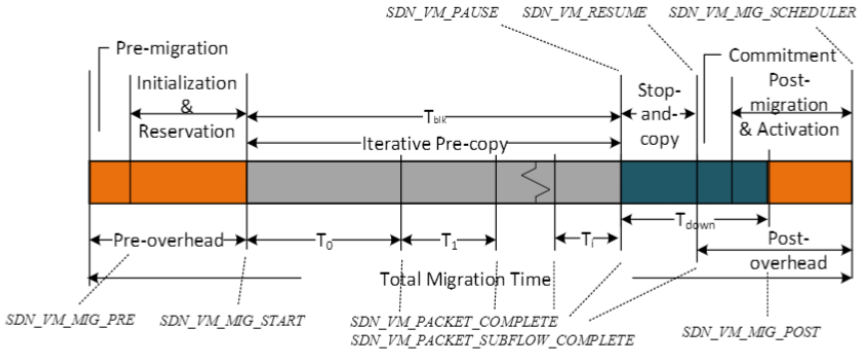


Figure 2.5: Pre-copy migration timeline[6]

2.2.3 Post-copy Migration

According to [7], the timeline of the post-copy migration procedure is as Figure 2.6 shows. This migration model was designed for VM migration. Since the migration of VM and container are consistent in the conceptual point of view, we can inherit the model to container migration.The total migration time is summarized as Equation 2.5 shows. T_{prepare} is the time between initiating the migration and transferring the state of the container to the target node. In this phase, the minimum states (processor states and non-pageable memory) to be transmitted is checkpointed. T_{transmit} is the duration to transmit the minimum checkpoint state. T_{resume} includes the time to restore the container from the minimum checkpoint image and the time to fetch the missing memory pages from the original host. In this case, the duration of T_{resume} is strongly correlated to the container's memory size and structure. Since post-copy does not require iterative copy of the memory, the total migration time is normally shorter than pre-copy migration. For the migration downtime of post-copy migration, it is equal to the transmission time of the minimum states. To be notice, the migration downtime of post-copy migration normally is longer that pre-copy migration. The reason is that for post-copy migration it needs to transfer all the non-pageable memory in the downtime phase while the pre-copy migration migration method only transfers the dirty memory pages.

$$T_{\text{migrate}} = T_{\text{prepare}} + T_{\text{transmit}} + T_{\text{resume}} \tag{2.5}$$

$$T_{\text{down}} = T_{\text{transmit}} \tag{2.6}$$

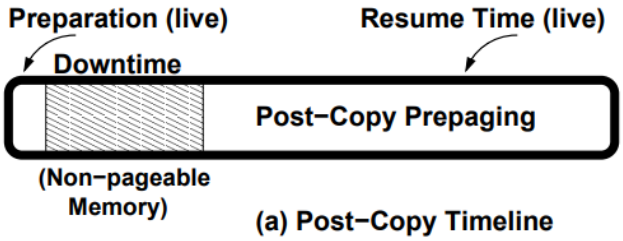


Figure 2.6: Post-copy migration timeline[7]

2.3 Related Works

The recent study of VM/container migration covers different phases from migration technique introduction to migration orchestration. which provide the a rich knowledge base for the study of this thesis.

Migration Techniques. [8] gives a detailed summary on the state of art on the migration techniques, which provides a quick insight to the related technologies. [9] focuses on the migration techniques used in network edge, which provides a more specific knowledge align to the target of this thesis.

Migration Network Managements. The study of network management during migration targets at guarantee the cross VM/container connection can be restored without re-connection after the migration. To reach this target, architectures based on different technologies are proposed: MPTCP [10], QUIC [11], PMIPv6 [12], LISP [13], and so on. However, the applicability and implementation difficulty of these technologies need to be carefully balanced. More detailed discussion will be done in the future section.

Actual Application Migration. In the study related to actual application migration, [14] provides an example of the migration of an MQTT based AR application which was directly containerized by container runtime runC [15]. However, in this example, it only preserves the MQTT session states during the migration and adopts connection handoff method to manage the network problem. The study [5] provides an example to migrate mobile core network components containerized by Docker container. And it mentioned that the TCP connection problem during migration could be solve with TCP re-establishment feature from Linux version 3.5.

Migration Orchestration. In the study related to migration orchestration, the orchestration model mostly focus on minimizing the migration downtime (e.g. [6], and [16]), and resource usage cost during the migration (e.g., [17], and [18]).

Chapter 3

Background Technologies

3.1 Containerization: Podman

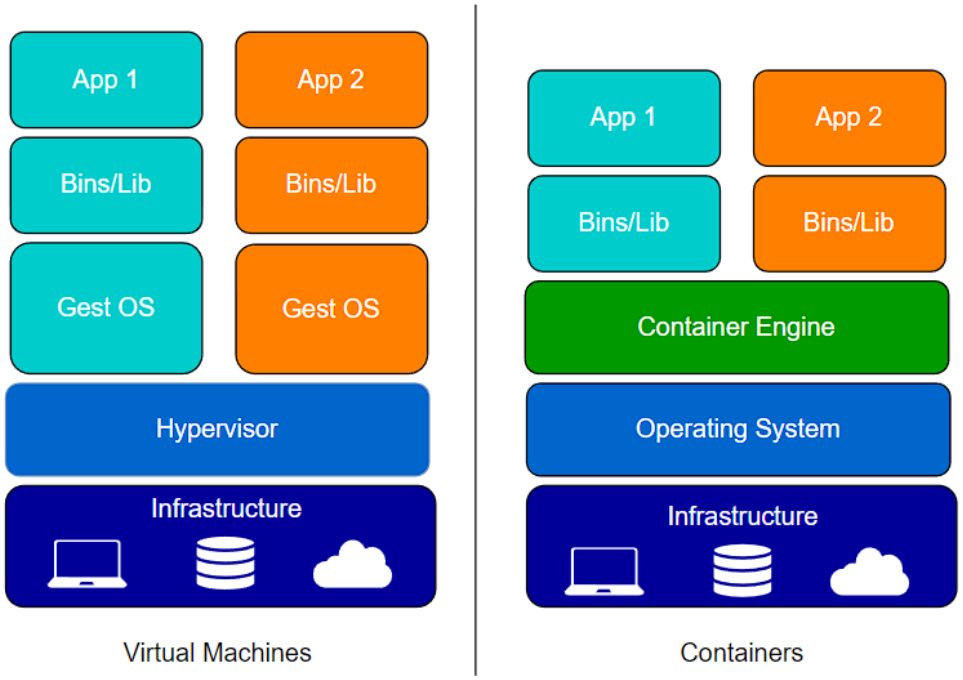


Figure 3.1: Virtual machines vs containers

In edge/cloud computing architecture, there are two commonly used tools to create isolated environments for different applications: virtual machines (VM) and containers. The chief difference between these two techniques is the level of isolation. As Figure 3.1 shows, a VM is an emulation of a physical computer. For each VM,

the emulation starts from the operating system level, which makes the environment inside the VM completely independent from the host operating system and the hypervisor. The VM hypervisor is the tool to create and manage VMs. With this architecture, VMs provides a securer and completely isolated environment. However, such kind of architecture has the shortcomings like long starting time and low efficiency. In container architecture, the containers create isolated environments from the user space. In this case, all containers running on the same host machine share the same operating system kernel. The container engine is the software that accepts user requests, pulls images, and runs the container from the end user's perspective. Even though in this architecture, the isolation level of the container is relatively lower, the containers show the advantage in image size, start time, and efficiency. Compared to VM, the property of the container is more suitable for the mobile service migration scenario.

There are many different implementations of container engines, including Docker [19], LXD [20], Podman [21], etc. Considering the support on the function of container migration, Podman is selected to be the container engine in the following study. Podman is a daemonless, open-source, Linux native container engine. It targets to make it easy to run, build, share and deploy applications using Open Containers Initiative (OCI) Containers and Container Images. It relies on an OCI-compliant Container Runtime (runc, crun, runv, etc) to interface with the host operating system and creates the running containers.

A Podman container is the runtime initiation of a container image. It is a standard Linux process isolated by namespace. Containers under the control of Podman can either be run by root or by a non-privileged user. To be noticed, the containers should be run by the root user in order to perform the migration function. The container image is a file comprised of system libraries, system tools, container metadata, application executable, and other necessary information to run a container. Podman uses the container images in the format defined by OCI, which defines the layers and metadata. When an application needs to be containerized, the executable of the application can be built into a image layer from scratch or on top of a proper parent image. This process could be done automatically by the container engine with a command script such as the Dockerfile.

3.2 Container Checkpoint/Restore: CRIU

Checkpoint/Restore In Userspace (CRIU) [22] is a Linux software that can freeze a running container (or an individual application), checkpoint its state to disk, and restore it to a local or remote machine in the checkpointed state with the checkpoint file. CRIU is one of the most commonly used tools to perform container migration. Most of the popular container engines (e.g., Podman, Docker, and LXC)

have different levels of integration with CRIU to support the containers' mobility.

There are two main steps to checkpoint a process (or a container, which is a tree of processes) with CRIU. First, CRIU uses Linux system call `ptrace()` to pause and seize a running process in order to collect the necessary information for checkpointing. The way CRIU obtains the detailed information is to read and dump the process data from Linux userspace interface `/proc/<PID>/*`. Secondly, after CRIU obtains all the necessary information, CRIU collects the states (majorly the memory content) of the process using a parasite code. To perform this step, CRIU first pauses the target process. When the process is paused, a parasite code is injected into the process. After the injection, the process is released and continues running together with the parasite code. At this moment, the parasite code can see and access everything the process can see and access. Meanwhile, the parasite code acts as a daemon waiting for commands from the main CRIU process. When the preparation of the checkpoint is ready, the main CRIU process commands the parasite code to dump the states of the target process to disk. After the state dumping is finished, the parasite code is removed from the process. At this moment, the checkpoint procedure is finished. Users can command CRIU to kill or leave the target process running after checkpointing with a simple option. From the perspective of the target process, it is unaware of all the operations done by CRIU. No additional operation is needed on the target process side to support the checkpoint procedure. In this view, CRIU shows its advantage of being able to checkpoint any Linux process.

CRIU can restore the checkpointed process (or container) to the original or a new machine. In the latter condition, the checkpoint image should be copied to the destination host in advance. To restore the checkpointed process with CRIU, four main steps are required. First, CRIU reads the checkpoint image and finds the information about the shared resources (e.g., files and shared memories) with the process to be restored. CRIU will first restore the shared resources before restoring the checkpointed process. Secondly, CRIU re-creates the checkpointed process (and its children processes) by calling fork() system call on the new host. Thirdly, CRIU restores all resources (e.g., open files, namespaces, and sockets) to the newly created process except for the resources of memory mapping exact location, timers, credentials, and threads. The reason for delaying the restoring of mentioned resources is that they should be restored when other resources are completely restored. In the last step, CRIU restores the remaining resources and releases the control of the target process. The restored process will continue running from exactly the same state as the original process when it is checkpointed. To be noticed, the restoring process requires the PID of the original process (and its children processes) is not occupied on the host machine. Otherwise, the restoring fails at the very beginning of the procedure.

3.3 File Synchronization

3.3.1 Rsync

Rsync [23] is a famous file synchronization tool on Unix-like operating systems. The basic function of this tool is to efficiently copy the target files to a local or remote directory. It is famous for its delta-transfer algorithm during file synchronization. As Figure 3.2 shows, when rsync performs file synchronization, it splits the file into blocks. Then, the destination rsync daemon sends the checksum of the blocks to the source rsync daemon (where the modified new files locate). Then, rsync applies a rolling block on the new files and performs the checksum on the rolling blocks. When the checksum of a rolling block matches one of the checksum values received from the destination, the block can be considered unchanged. After the rolling block goes through the entire new file, all the changes on the new file can be found out. In the data transfer phase, rsync only sends the new changes to the destination and insets the changes to the correct position of the files on the remote. The delta-transfer algorithm significantly reduces the amount of data to be sent over the network. Moreover, during the data transfer phase, rsync provides options for applying different compression algorithms. With the property of high efficiency, rsync is commonly adopted in the use-cases like daily backup and data recovery.

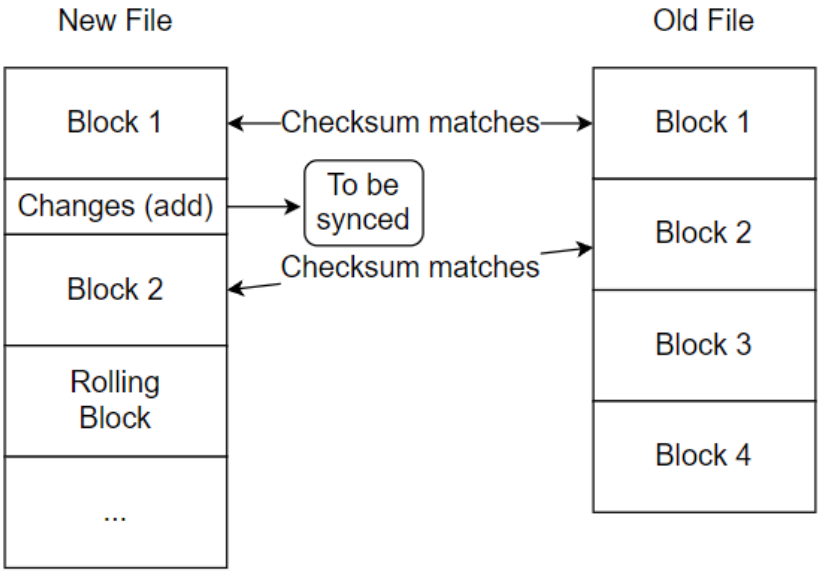


Figure 3.2: Delta-transfer algorithm

3.3.2 Borg

BorgBackup (Borg) is a tool duplicated to provide an efficient and secure way to backup data to a local or remote respository. It is famous for its deduplicating backup feature. The structure of the backup repository is as Figure 3.3 shows. On the highest level is the manifest of the repository. It references all archives in the repository, acting as the register map. The archives are the immutable file representing the backups. However, the archives itself does not storing any data of the back files. The real backup files are splited and stored in chunk objects. The archives only stores the IDs of chunks which construct a specific backup. Every time user creates a new backup, borg splits the backup files into chunks with a specific algorithm and checks if any of the chunks match the ones already exist in the repository (even if the chunk is from complete different file). Borg only stores the chunks which do not have any match in the old repository. In this case, the backup files in the same repository are deduplicated, which significantly reduced the requirement on storage for multiple backups. When the user wants to restore a specific version of backup, borg collect the file chunks with corresponding IDs according to the archive and reconstruct them to normal files. With the nice deduplication feature and additional optional security features, borg are commonly used in daily backup and secure storage scenarios.

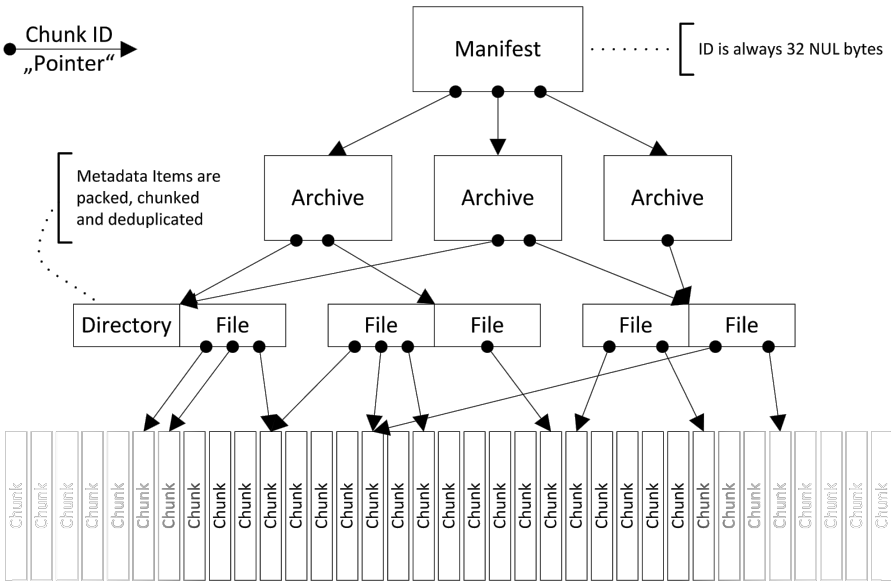


Figure 3.3: Borg object graph [24]

3.4 SDN: Open vSwitch

Software-defined networking (SDN) is an approach of network management that targets at making the network more flexible and easier to manage. Compared to the traditional network that uses dedicated hardware devices to control the network traffic, SDN proposes to create and control the network via software. The SDN architecture separates the network into the control and the data plane. In the control plane, there may exist one or more SDN controllers. The SDN controller is a centralized network intelligence that can orchestrate and manage the network. They decide how network traffic is handled according to the user's setting/programming. The data plane is responsible for forwarding the traffic based on the control plane's decision.

Open vSwitch (OvS) [25] is a powerful tool for constructing an SDN data plane. It is a multilayer software switch that supports standard management interfaces and opens forwarding functions to programmatic extension and control. There are two interesting features of OvS for container migration scenarios, listed as follows:

- Geneve, GRE, VXLAN, STT, and LISP tunneling
- QoS (Quality of Service) configuration, plus policing

Open vSwitch is composed of a set of components with specific functions as Figure 3.4 show. The components targeted at implementing the function of OvS are located in the kernel and userspace layer, with the functions as follows:

- ovs-vswitchd: is a daemon that implements the switch.
- ovsdb-server: a lightweight database server that ovs-vswitchd queries to obtain its configuration.
- ovsdb: a network-accessible database system.
- fast datapath module (optional): a high-performance forwarding module using a Linux kernel.

To provide a flexible management of the switch, OvS also provide a set of components target at provide management function, listed as following:

- ovs-vsctl: a tool for querying and updating the configuration of ovs-vswitchd.
- ovs-appctl: a tool that sends commands to running Open vSwitch daemons.
- ovs-ofctl: a tool for querying and controlling OpenFlow switches and controllers.
- ovs-dbtool: a tool to manage ovsdb.
- ovs-dpctl: a tool for configuring the switch kernel module.

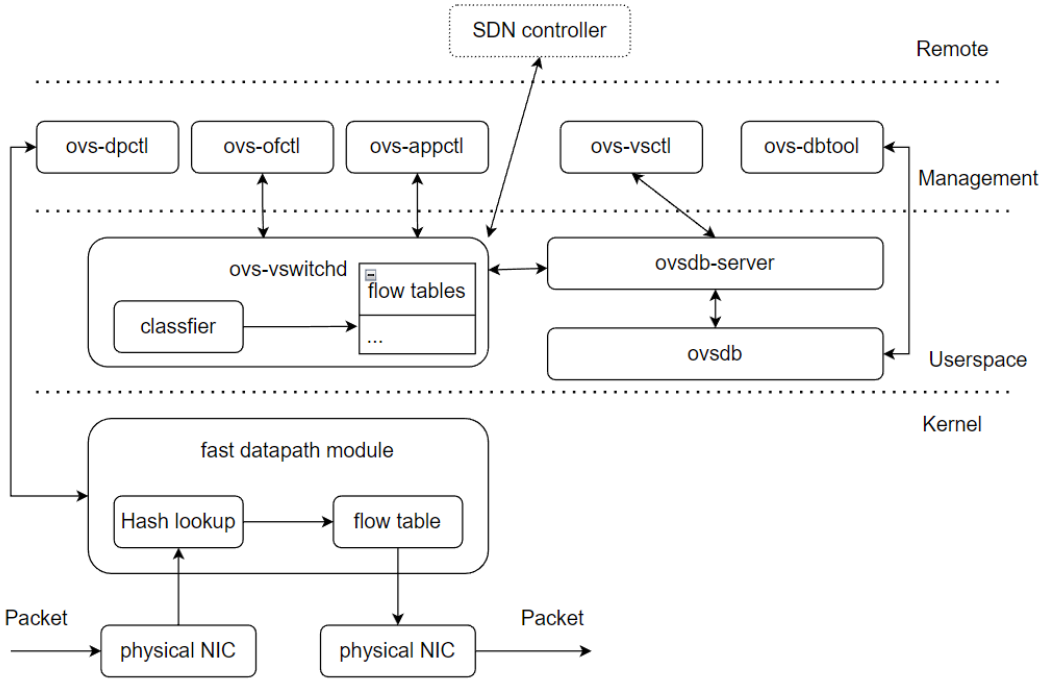


Figure 3.4: Open vSwitch system architecture

Chapter 4

Realization Stack: Migration File Synchronization

In the realization of container migration, one of the most critical and basic stacks is file synchronization. There are two types of files need to be synchronized in the migration procedure: container image and checkpoint image. The container image is the base image to create/restore the container on the host machine. The synchronization of this kind of image is not time critical since it can be pre-downloaded to a specific machine and used whenever needed. Online container image repositories such as Docker Hub can easily distribute the container image to different host machines. So, the container image is not the target of the study in this thesis. On the other hand, the synchronization of the checkpoint image can significantly affect the overall performance of container migration. In different migration methods, the more efficient the checkpoint image synchronization is, the lower migration downtime will be, and lower dirty memory pages will be generated. Considering that the resources on the edge server (e.g., CPU, memory, network, disk) are limited, the study of this chapter will focus on finding the optimal setting for checkpoint image synchronization. Famous file synchronization tools on Linux operating system "rsync" and "borg" are selected as the main tools to be studied.

4.1 Analysis Methodology

The study of the file synchronization tools will be based on experiments. The performance matrix to be analysis includes deduplication efficiency, CPU usage, memory usage, network usage and storage. In the test of deduplication performance, dedicated test scenarios will be designed and applied to each tool according to their property in order to find the maximum capability. According to [26], the system related performance matrix can be obtained with different tools on Linux system.

In our study, the following tools will be used:

- **CPU:** time -p <pid>
- **Memory**: cat /proc/<PID>/statm
- **Network**: nethogs
- **Storage**: du -sh <file>

With these tools, we can obtain the information related to the interested performance matrix, listed at following:

- **User mode CPU seconds:** Total number of CPU-seconds directly used by the process (in user mode), in seconds.
- **Kernel model CPU seconds:** Total number of CPU-seconds used by the system on behalf of the process (in kernel mode), in seconds.
- **Elapsed real time:** Elapsed real (wall clock) time used by the process, in [hours:]minutes: seconds.
- **Percentage of the CPU:** Percentage of the CPU that this job got, equal to (CPU user time+ CPU kernel times)/ CPU elapsed time, in percentage.
- **Maximum resident set size:**The peak resident set size (Peak RSS or Max RSS) refers to the peak amount of memory a process has had up to that point, in bytes.
- **File size:** The total size of the synchronized file,in bytes.
- **Transmitted data size:** Transmitted data size, in bytes.

To ensure the common baseline for the test and comparison of the target tools, the experiments in this chapter are done on a bare machine with the following settings:

- **OS:** Ubuntu 20.04
- **CPU:** Intel® Core™ i7-7700HQ CP @2.8GHz
- **Memory:** 16GiB DDR4 System Memory
- **Network:** Intel® Dual Band Wireless-AC 8265 (WiFi 5)
- **Storage:** Samsung MZVLW512HMJP (512GB SSD)

4.2 Deduplication performances

4.2.1 Rsync Deduplication

In order to simplify the setting of the test environment and avoid intra-machine synchronization optimization, the deduplication performance test of rsync is done with two containers running on the same host as Figure 4.1 shows. In container 1, a file of random bytes will be generated/modified by a python script, and Rsync will be used to sync the file to container 2. The synchronization log printed by rsync (includes the sent/received bytes and transmission speed) will be collected into a log file for further analysis.

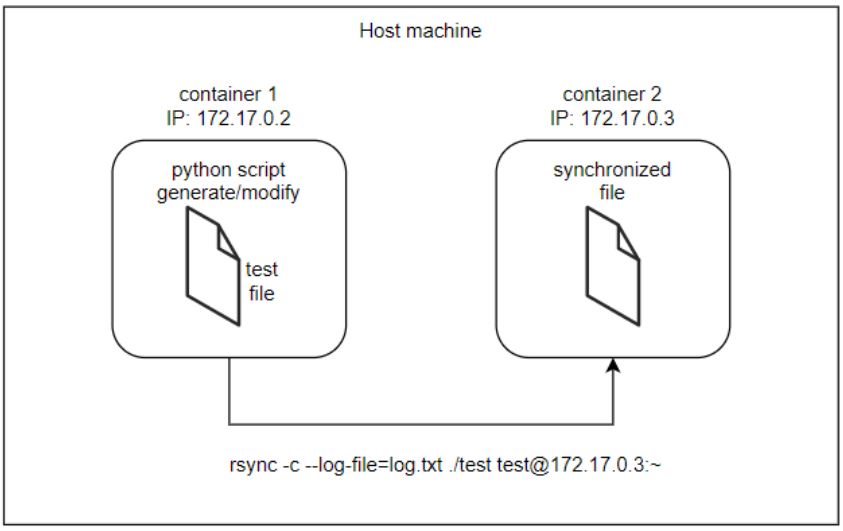


Figure 4.1: Rsync deduplication test topology

Basic deduplication behavior test

As mentioned in Section 3.3.1, rsync synchronizes the files using the delta-transfer algorithm, which transfers the file in blocks of a specific size. However, how will rsync behavior when the change of the file is less than one block size is not clear in the documentation. In order to understand the exact behavior of rsync, a test is done with the procedure as Listing 4.1 shows. The synchronization block size of rsync is set to 1024 bytes. In each iteration, 128 random bytes are added to the end of the file and synchronized to remote with rsync. The sent and received data amount can be obtained from the log of rsync on the master host (container 1 in Figure 4.1). To be noticed, the data compression option during transmission is turned off.

Listing 4.1: Rsync test algorithm (fixed block size and data increase rate)

```
SET delta-transfer block size to 1024 bytes
FOR i IN RANGE [1,N]:
    Open the existing file
    Add 128 random bytes to the end of the file
    Save and close the file
    Use rsync to synchronize to remote
    Sleep 1 second
```

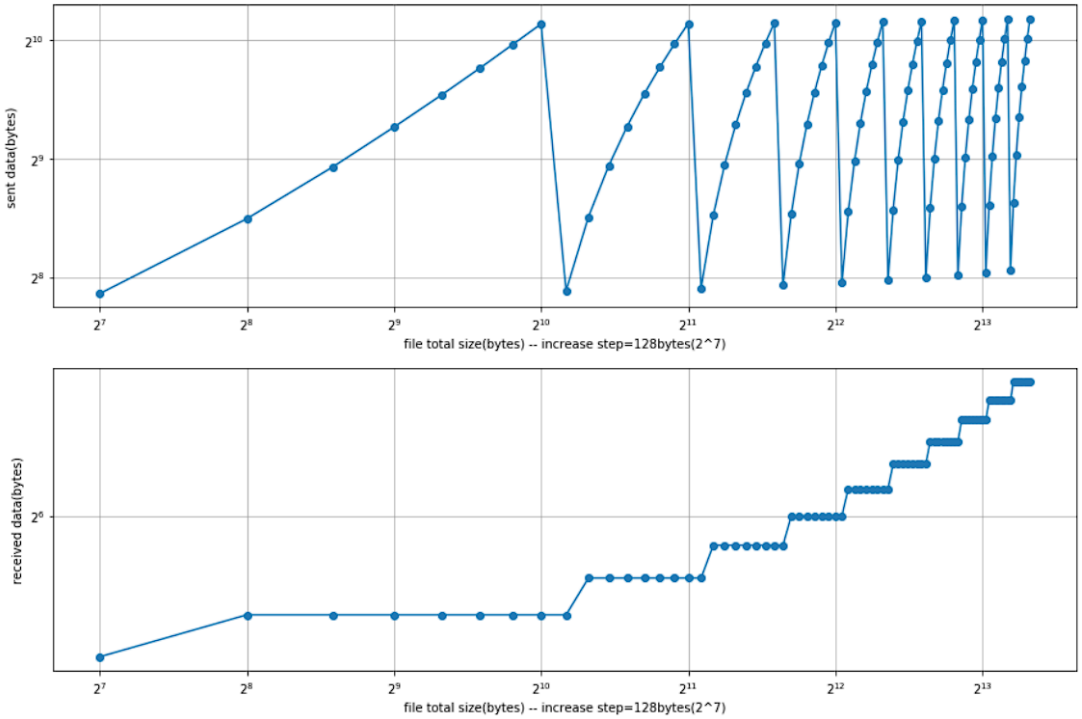


Figure 4.2: Rsync basic deduplication behavior test result

The test result is as Figure 4.2 shows. The x-axis is the total size of the file to be synchronized. The raised trend of the sent data amount is because when the size of the changed file is less than the block size of the delta-transfer algorithm, rsync will synchronize all the data in that changed block. When the total changed size of the file is larger than a block size (1024 bytes in this test), the sent data amount in that iteration decreases, and the received data amount increases a step. This phenomenon is because this additional block is full and unchanged in the future synchronization, so rsync only checks the block's checksum and no longer re-transmits it.

Deduplication performance with different block size

With the basic deduplication behavior test of rsync, it is clear that rsync can not deduplicate the old data when the size of old data is smaller than one synchronization block. So the setting of block size is crucial to the synchronization performance. In order to understand the effect of block size, a test as Listing 4.2 is performed. In the test, the data increase rate is fixed to 5120 bytes per iteration. The size of the synchronization block varies from 128 bytes to 6400 bytes, with the step of 128 bytes in each iteration. The sent and received data amounts are obtained from the log of rsync on the master host (container 1 in Figure 4.1). To be noticed, the data compression option during transmission is also turned off.

Listing 4.2: Rsync block size test algorithm (fixed data increase rate)

```
FOR i IN RANGE[1,N]:
    Open the existing file
    Add 5120 random bytes to the end of the file
    Save and close the file
    SET delta-transfer block size to (128*i) bytes
    Use rsync to synchronize to remote
    Sleep 1 second
```

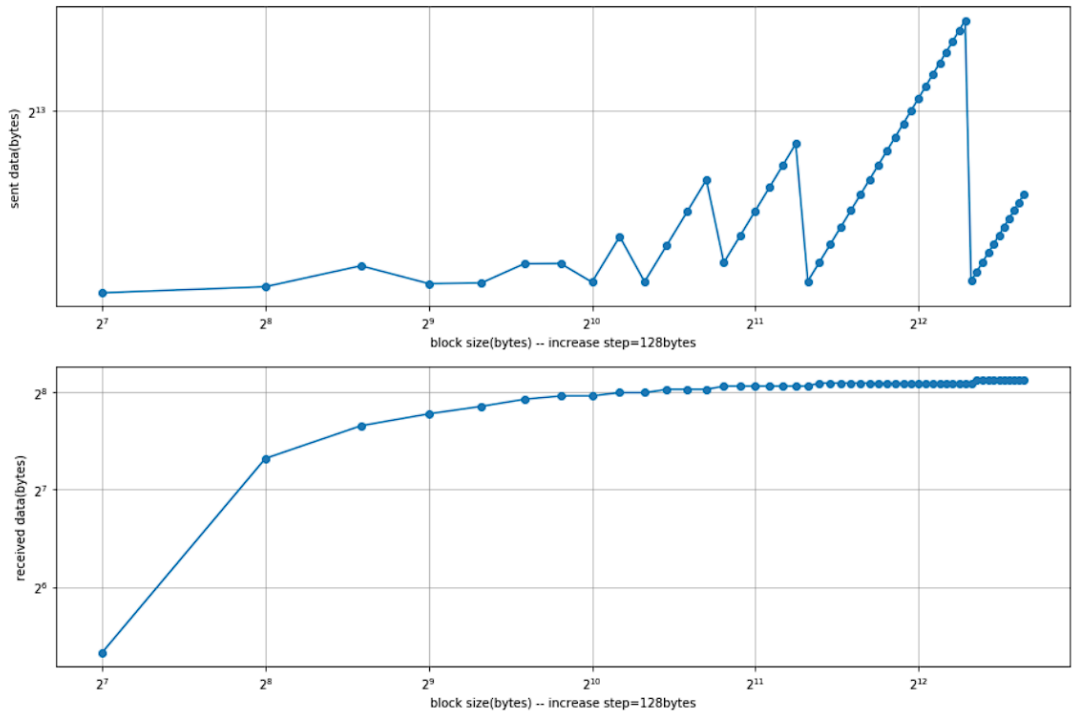


Figure 4.3: Rsync deduplication behavior under different block size

The test result is as Figure 4.3 shows. When the block size of the delta-transfer algorithm is less than 1024 bytes, the sent data size in each synchronization is more or less close to the increased data size (5120 bytes). It gives efficient synchronization in the aspect of the sent data amount. However, the received data size increases rapidly in this phase because the small block size leads to more checksum data being sent from the remote to perform the deduplication check as the total size of the file increases by iteration. When the block size is larger than 1024 bytes, the synchronized data in the final block is difficult to be completely filled. In this case, the data in the last block is synchronized again in the next iteration, leading to the increase in the sent data. However, due to the large block size, even though the total file accumulates in each iteration, the received data size increases slowly in this phase. However, comparing the scale of sent and received data size, the smaller block size provides a better overall synchronization performance.

Deduplication performance with slight change at the beginning of file

In the previous test, a specific amount of random bytes were added to the end of the file and synchronized to remote in each iteration. In this kind of test scenario, the deduplication performance of rsync is very well. It is interesting to ask whether the deduplication algorithm works when the random bytes are added to the beginning of the file. To answer this question, a test is done as Listing 4.3 shows. In the test, the block size is set to 128 bytes. The file to be synchronized is initiated with 1024 random bytes and synchronized to the remote. In each iteration, a random byte is added to the beginning of the file.

Listing 4.3: Rsync test algorithm (extreme slight change)

```
SET delta-transfer block size to 128 bytes
Initiate the file with 1024 random bytes
Synchronize the file to remote with rsync
FOR i IN RANGE[1,N]:
    Open the existing file
    Add 1 random byte to the beginning of the file
    Save and close the file
    Synchronize the file to remote with rsync
    Sleep 1 second
```

The test result is as Figure 4.4 shows. When only 1 byte is added to the beginning of the file, the sent data amount observed by rsync is only 136 bytes, which is much less than the original file size. This phenomenon proves that the deduplication algorithm works well even if an extremely slight change happens at the beginning of the file. However, compared to the change of the file, the overhead to perform the synchronization is much larger. It leads to the conclusion that the synchronization cost is considerable when there is only a slight change in the file.

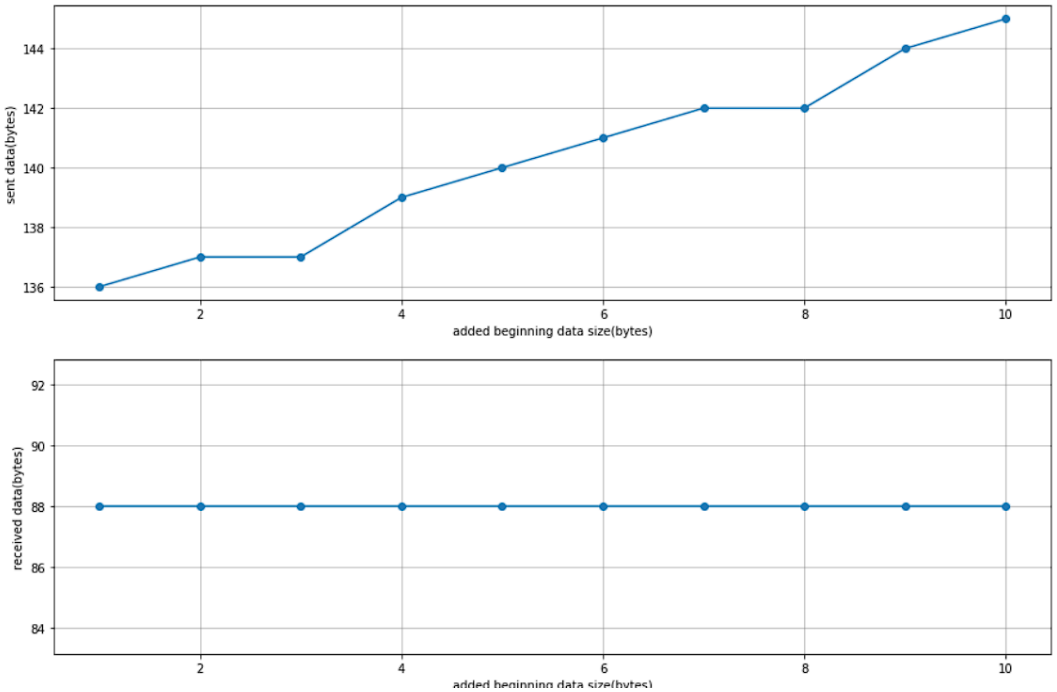


Figure 4.4: Rsync deduplication behavior with small random bytes added to the beginning of file

Deduplication performance with random modification in file

In the container migration scenario, it is more common to face the condition that there has a "replacement" of content in parts of the file to be synchronized (especially the memory content of the checkpoint image of the container). In order to analyze the deduplication performance of rsync under this condition, a simulation test is done. In the test, a file of 1024 random bytes is generated and synchronized to the remote. Then, traverse the local file. Each byte has a probability of 0.5% to be replaced with another random byte. After the modification, the file is synchronized to remote with different block size settings. The result is as Table 4.1 shows. When the size of the synchronization block is smaller, it is more possible for rsync to find complete unchanged blocks and deduplicate them in the second synchronization. When the block size is larger than 256 bytes, the content in each block has some slight changes, so the deduplication algorithm does not work. However, when the block size is smaller, more checksum data need to be sent to the origin, which leads to a larger amount of received data in the second synchronization. In order to improve the synchronization efficiency of rsync in this kind of scenario, the block size setting needs to consider the distribution of the "replaced" content in the file. Only a "matched" block size can make the deduplication algorithm works well.

Table 4.1: Random modification synchronization test result

Block size [bytes]	32	64	128	256	512
1st sync sent [bytes]	1122	1122	1122	1122	1122
1st sync recv [bytes]	35	35	35	35	35
2nd sync sent (max) [bytes]	400	530	880	1122	1122
2nd sync recv (max) [bytes]	280	135	85	60	50

4.2.2 Borg Deduplication

As mentioned in Section 3.3.2, borg deduplicates the backup file by comparing the file chunks. So, to test and obtain the optimal deduplication performance of borg, the key is to configure the chunking algorithm properly. According to the borg documentation, each file chunk's size is not a constant value. Borg chunker cuts the target files into chunks according to the Buzhash algorithm, which is a rolling hash algorithm. The chunker produces a chunk in the size of $2^{\text{HASH\_MASK\_BITS}}$ bytes every time the last HASH_MASK_BITS bits of the hash are zero. In order to adjust the chunking behavior, borg provides an option "–chunker-params=CHUNK_MIN_EXP, CHUNK_MAX_EXP, HASH_MASK_BITS, HASH_WINDOW_SIZE" to set the chunking parameters, with the meaning listed as following:

- CHUNK_MIN_EXP: minimum chunk size, default equal to 19 (2^{19} bytes).
- CHUNK_MAX_EXP: maximum chunk size, default equal to 23 (2^{23} bytes).
- HASH_MASK_BITS: target chunk size, default equal to 21 (2^{21} bytes).
- HASH_WINDOW_SIZE: default 4095 bytes.

According to the borg documentation, when the chunker parameter setting is configured to "–chunker-params=10,23,16,4095", the chunker creates a large number of file chunks and costs more resources. However, it results in a fine-grained deduplication performance. So in the following test, this setting is adopted as the basic test configuration.

The test topology of borg is as Figure 4.5. The tested file will be backed up to a local repository by borg. Since the remote repository of borg is implemented with SSHFS, which is different from the file synchronization concept we study, the remote repository of borg will not be detailly studied in this thesis (more details will be discussed in Section 4.3.4). In each test, a file containing 1MB or random bytes will be generated and backed up to the archive. Considering the realization of borg's deduplication function, the following five test scenarios are designed to validate the deduplication performance:

- Test 1: Add 1024 random bytes to the end of the file.
- Test 2: Add 1024 random bytes to the beginning of the file.
- Test 3: Replace the first 1024 bytes of the file with different random bytes.
- Test 4: Randomly replace bytes in the file with a probability of 0.5
- Test 5: Replace the first 1024 bytes of the file with different random bytes and modify the name of the file.

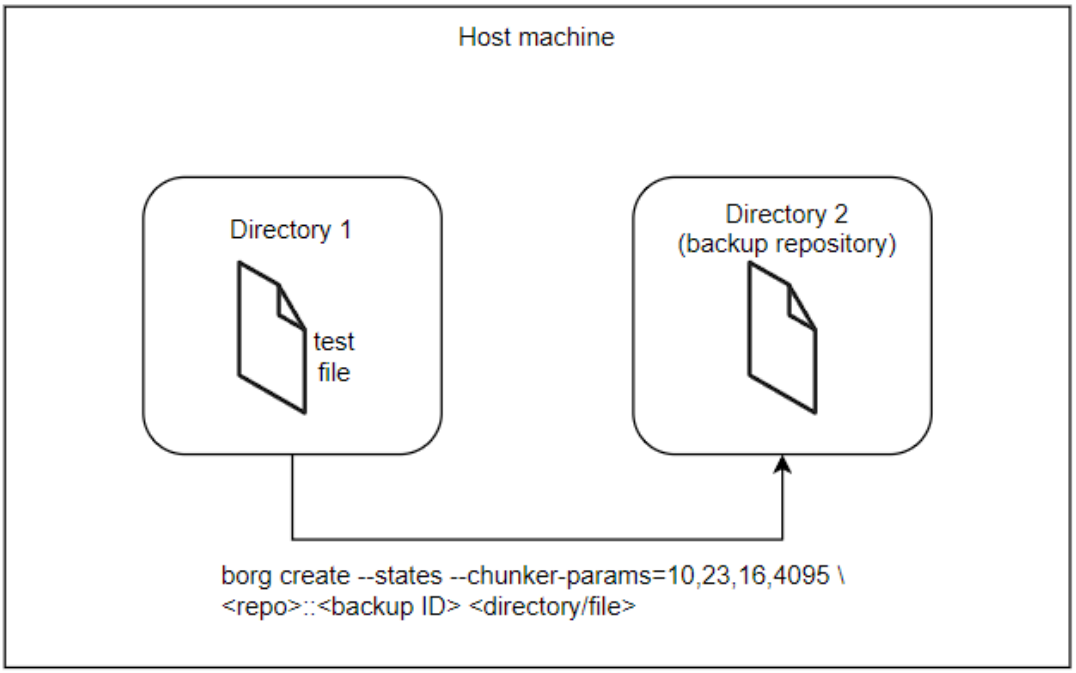


Figure 4.5: Borg deduplication test topology

The test result is as Table 4.2 shows. Comparing the file and backup archive size in tests 1-3, we can conclude that the deduplication algorithm of borg works very well when only a small continuous portion of the file is changed. The result in test 4 shows that when the change of the file is distributed to the entire file, even if only a small amount of data is changed, the deduplication algorithm of borg is completely failed. It shows that the deduplication algorithm of borg is not very robust to every scenario. However, the result of test 5 shows the strength of borg that other synchronization tools do not have: perform deduplication among different files. Comparing the number of archive file chunks in each test, it is easy to find out that the deduplication performance of borg strongly correlates to the

	Test 1	Test 2	Test 3	Test 4	Test 5
Original file size [KB]	1024	1024	1024	1024	1024
Modified file size [KB]	1025	1025	1024	1024	1024
1st archive size [MB]	1.05	1.05	1.05	1.05	1.05
2nd archive size [MB]	1.05	1.05	1.05	1.05	1.05
Total archive size [MB]	1.06	1.15	1.13	2.11	1.15
File duplication rate	99.90%	99.90%	99.90%	99.95%	99.90%
Archive deduplication rate	99.05%	90.48%	92.38%	N/A	90.47%
# of chunks in 1st archive	17	13	14	14	23
# of chunks in 2nd archive	17	13	14	22	23
# of resued chunks	14	10	11	0	20

Table 4.2: Borg deduplication test result

number of reused chunks. Due to the special algorithm borg adopted to generate the file chunk, even though the original file size in each test is the same, the number of generated archive chunks is different. In conclusion, the dynamic of archive chunks generation of borg does bring more security to the file storage but also brings more challenges in the performance of file deduplication.

4.3 System related performance

In order to test the system resource usage performance of the file synchronization tools under different conditions, the following performance matrix will be collected from the following test scenarios:

- original: Sync/backup the original file to the remote machine/repository, original file size is 100 MB.
- add begin: When remote machine/repository already has the original file, add 1024 bytes to the beginning of the local file and then sync/backup to remote machine/repository
- add begin 1: When remote machine/repository already has the original file, add only 1 byte to the beginning of the local file and then sync/backup to remote machine/repository (extreme condition)
- add end: When remote/repository already has the original file, add 1024 bytes to the end of the local file and sync/backup to remote/repository
- modify begin: When remote/repository already has the original file, modify

1024 bytes at the beginning of the local file and sync/backup to remote/repository

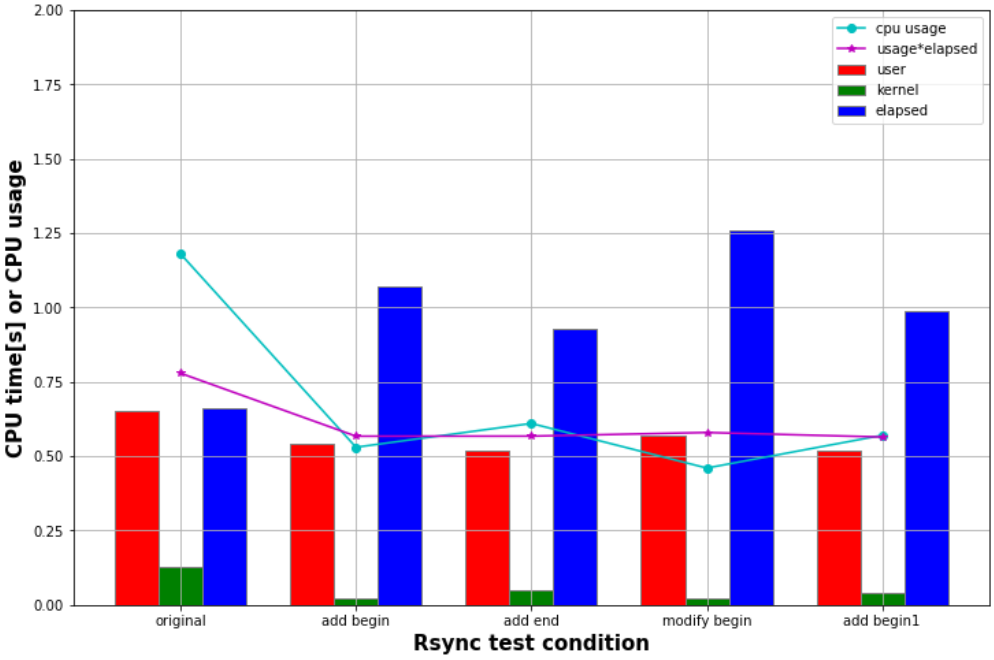
For the file synchronization tools rsync and borg, the block/trunk size setting significantly affect the synchronization performance. In order to make them work under a similar baseline, the following settings are adapted to the corresponding tools:

- Borg chunker size: –chunker-params=10,23,16,4095
- Rsync block size: -B 1024

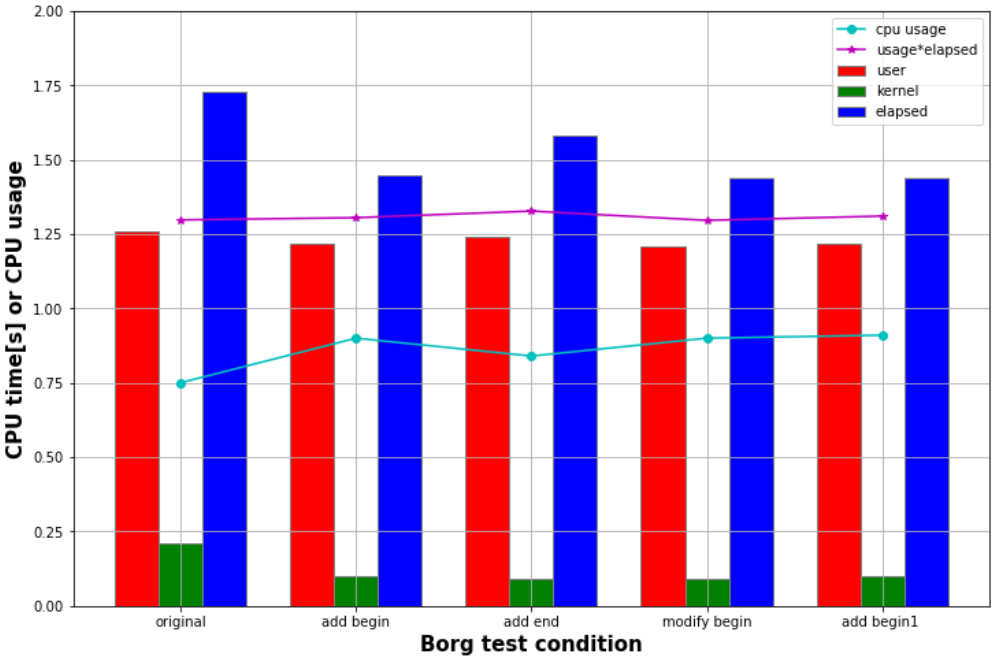
4.3.1 CPU Cost Measurement

The result of CPU cost measurement is as Figure 4.6 shows. In general points of view, borg consumes more CPU resources than rsync. In Fig.4.6a, it is interesting to point out that the CPU usage of rsync is very high in synchronizing the original file. However, the elapsed time in this procedure is much shorter than in other scenarios. In order to make this behavior easier to understand, we also plotted the value of the CPU usage multiplies by the elapsed time. This value equals the user mode CPU time plus the kernel mode CPU time. The value in the "origin" scenario returns to a similar level as other test cases. Among all the test scenarios, the "modify begin" scenario has the highest elapsed time, meaning it takes the longest time to complete the synchronization task. The reason is that the modification at the beginning of the file will take rsync more time to perform the rolling checksum (for the delta-transfer algorithm) and to send out the modified file blocks.

For the measurement result of borg (as Figure 4.6b show), the value of CPU usage times elapsed time for all cases is basically the same. However, the elapsed time in the "original" scenario is longer than other cases. The main reason is related to the creation of the initial archive and managing all the archive chunks (no exit chunks in this case).



(a) Rsync



(b) Borg

Figure 4.6: File synchronization tools CPU usage usage measurement

4.3.2 Memory Cost Measurement

The result of memory cost measurement is as Figure 4.7 shows. In conclusion, borg uses more memory (around 10 times) than rsync during the backup/sync procedure. And during the synchronization/backup procedure, the way to change the file has no significant effect on the memory usage of the tools.

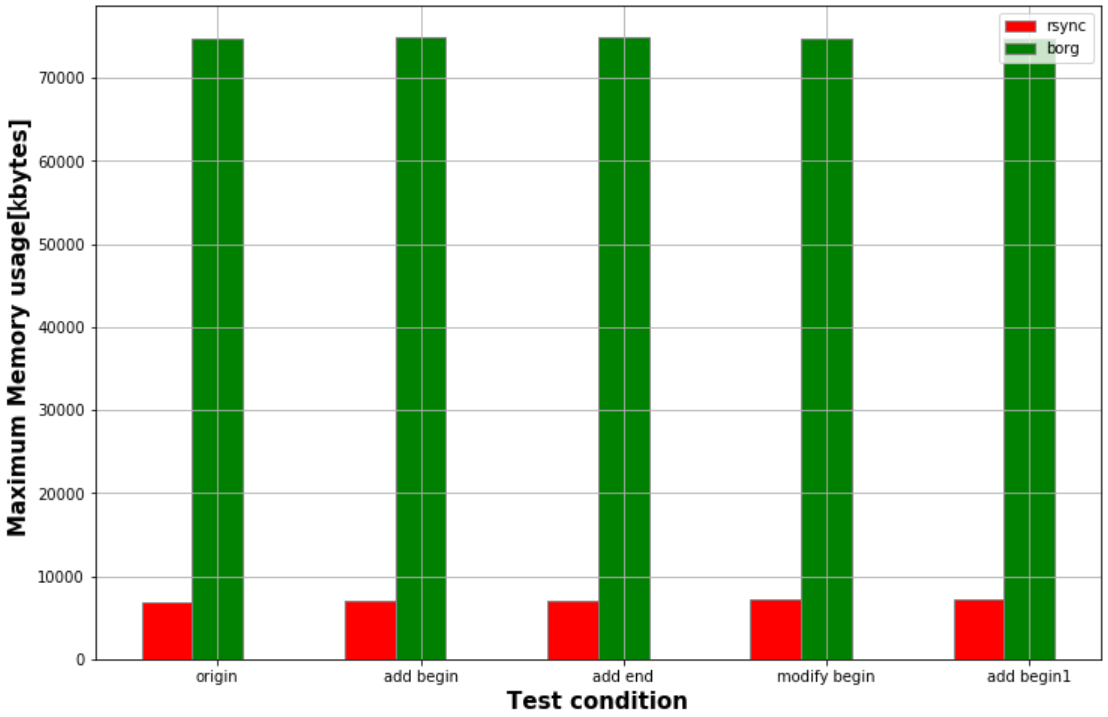


Figure 4.7: Memory usage test result (borg and rsync)

4.3.3 Storage Cost Measurement

The result of storage cost measurement is as Figure 4.8 shows. In the "original" scenario, rsync synchronize the file to the remote. In this case, the synchronized file's size is exactly the same size as the original local file. In the case of the borg, it splits the files into chunks and stores them in the local repository. There are some overheads in the backup due to the chunking, so archive size is a little larger than the actual size of the file. In the rest test scenarios, the synchronized files by rsync are all in the same size of the original file. However, for borg, the size of archives in these cases is much smaller than the original file. The reason is that the archive of the original file is already in the repository. Borg only backup the modified chunks of the file. Since the modification is relatively small, the size of

new stored chunks is also very small.

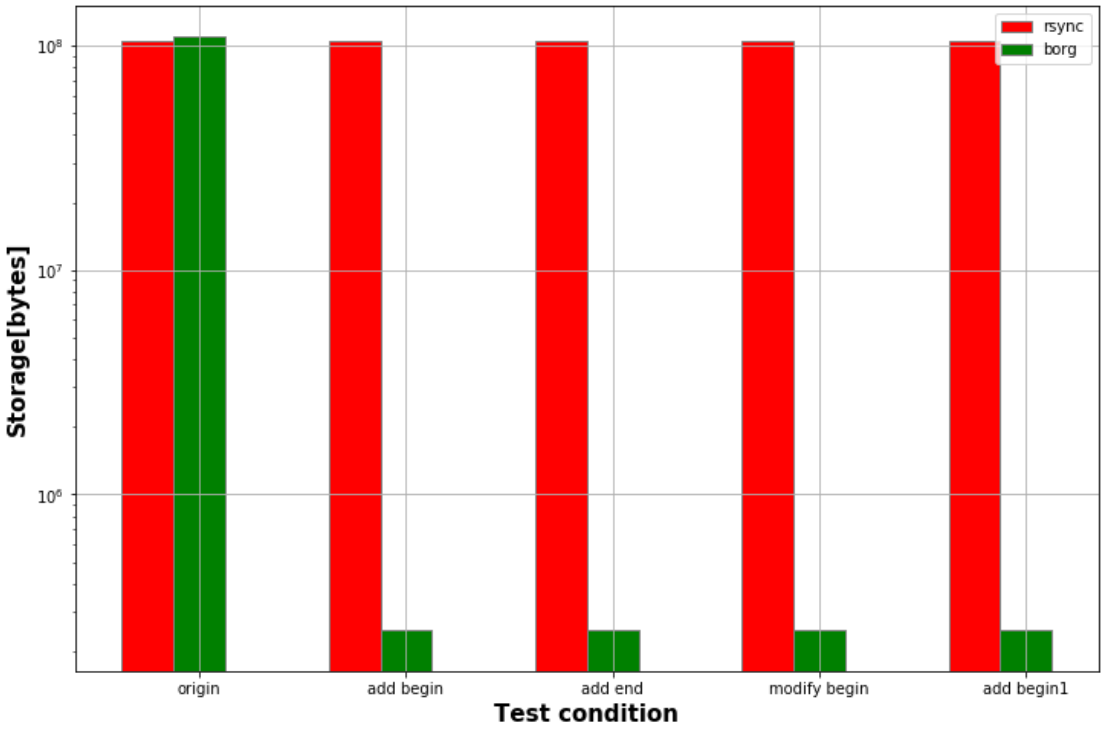


Figure 4.8: Storage usage test result (borg and rsync)

4.3.4 Network Cost Measurement

The result of network cost measurement of rsync is as Figure 4.8 shows. In the "original" scenario, no file exists on the remote destination. When rsync synchronizes the local file to the remote, it behaves as directly sending a file to the remote without applying the delta-transfer algorithm. So, in this case, the received data measured by rsync is close to zero (10^{-5} MB level). However, the received data measured by the externel tool "nethogs" is much larger. The reason for this phenomenon is that rsync only measures the user data it actually transmits. Instead, nethogs as an external tool, measures all the data rsync communicates with the remote host, including other necessary network overheads (e.g., packet header, ssh establishment packets). In other test scenarios, we can see that the communication overhead is relatively small compared to the actual payload. Furthermore, the smaller transmitted data amount in these cases again proves that the delta-transfer algorithm does help rsync reduce a huge amount of network traffic, significantly increasing the synchronization efficiency.

On the other hand, the network cost of borg is not discussed in this thesis. As Figure 4.5 shows, the backup repository of borg is located on the local host in this test. Borg does provide the option to make borg synchronize the backup to the remote. However, borg implements this function with an SSHFS, which is a network file system based on SSH File Transfer Protocol (SFTP). In this kind of implementation, the file is directly stored in the remote file system without a local replica. Since the behavior is quite different from the "synchronization" concept we discuss here, this thesis does not study the network property of borg.

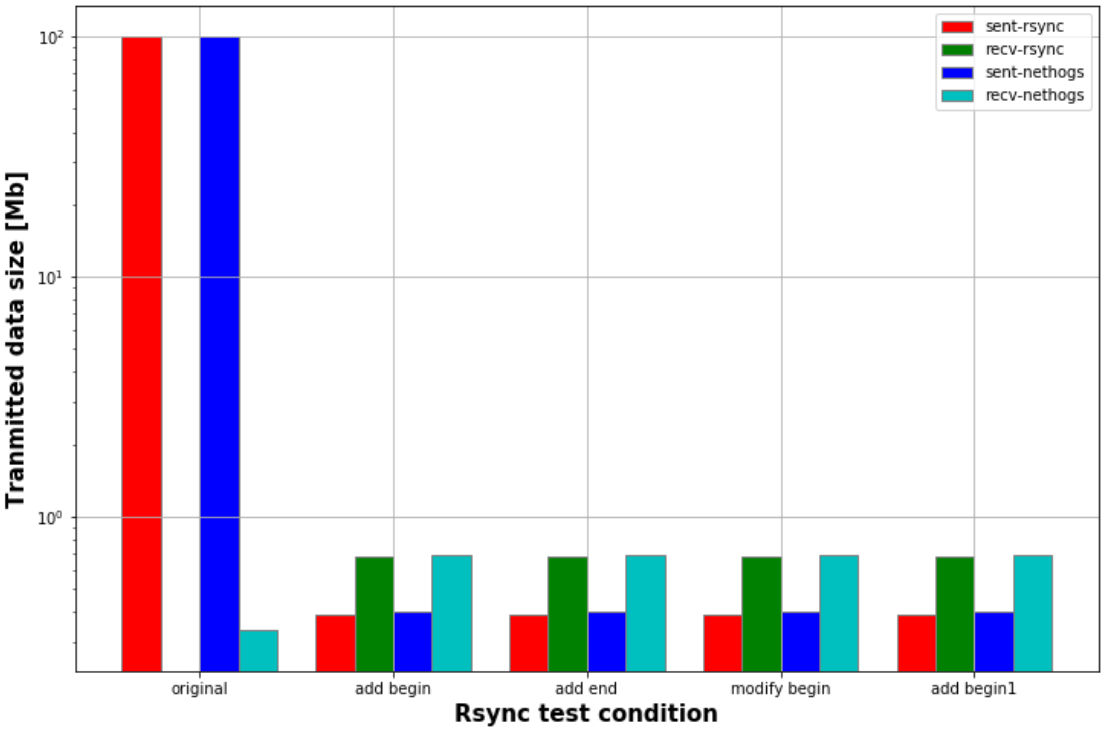


Figure 4.9: Rsync network usage test result

4.4 Synchronization Tool For Container Migration

With the study of rsync and borg in this chapter, the deduplication behavior and file synchronization performance of these two tools are clear to us. Considering the usage of them in the container migration scenario, rsync is more suitable for this use case than borg. The reasons for this conclusion are listed in the following:

- **More robust to random changes:** In the container migration scenario,

the major component in the checkpoint image is the application's memory. In the general case, the changes in the memory page are random. According to the previous test, the deduplication algorithm of rsync can perform better than borg when proper block size is configured.

- **Less consumption on system resources:** In edge computing scenario, the system resources are valuable and limited. According to the previous test, rsync consumes fewer resources than borg in the sense of CPU and Memory.
- **Simpler architecture:** From the usage point of view, rsync is simpler than borg. With rsync, synchronization can be easily performed by specifying the location of the synchronization source and destination. But borg needs to create a specific repository for the backup.

Chapter 5

Realization Stack: Stateful Container Migration

The stateful container migration targets at moving a container together with its internal working states (including container's states and application states) from one host machine to another. The most noteworthy advantage of this technique is that the migrated container can restore to the working state before migration on the new host and avoid the long warmup time. In order to perform stateful container migration, there are three main basic steps:

1. Freeze a running container and checkpoint its states to disk.
2. Move the checkpoint image to the destination host.
3. Restore the container from the checkpoint image.

To freeze and checkpoint a container, one of the most powerful tools is CRIU (Checkpoint/Restore In Userspace). As mentioned in Section 3.2, it supports most of the migration methods. Among the most popular container engines, Podman container has the best integration with CRIU. So the container migration experiments in this thesis will be done with Podman and CRIU. However, the Podman container has not yet integrated the post-copy migration method in CRIU so the study will focus on cold and pre-copy container migration.

5.1 Stateful Local Application

A stateful application is the one that performs with the context of previous transactions, and the current transaction may be affected by what happened during previous transactions. The simplest stateful application is a simple counter, as Listing 5.1 shows.

Listing 5.1: Simplest stateful application: counter

```
#include <stdio.h>
#include <unistd.h>
int main() {
  long long n = 0;
  while (1) {
    printf("%lld\n", ++n);
    sleep (1);
  }
  return 0;
}
```

However, the simple counter can not represent the common property of stateful applications. In order to better study the migration behavior, a stateful application simulator is designed as Figure 5.1 shows. The application's states can be categorized into memory state and non-memory state. The non-memory state includes the CPU states, the network states, and so on. In most cases, the non-memory state is stable and relatively small in size. The memory state is the memory the application allocated for the specific working scenario. Typically, the memory state is dynamic and relatively large in size.

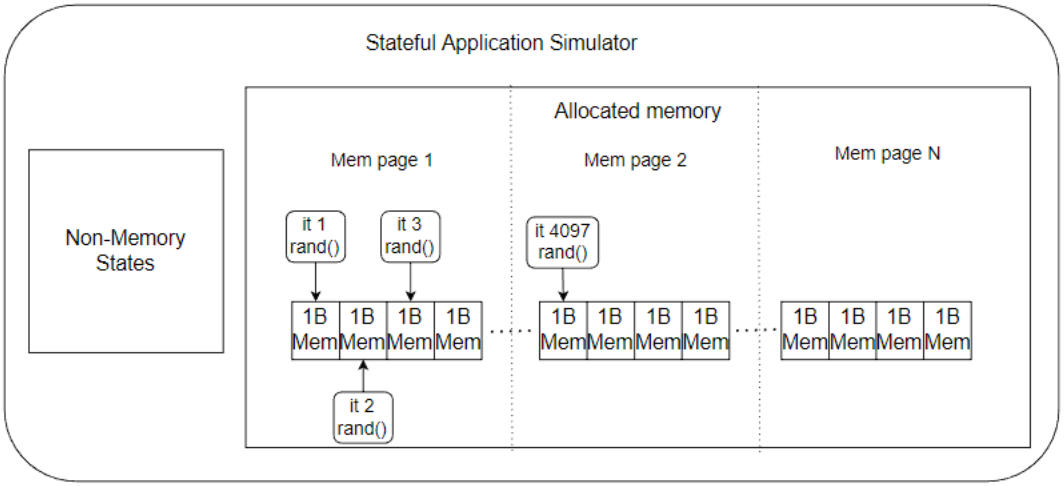


Figure 5.1: Basic stateful application simulator

In order to quantify the dynamic of an application's memory content, which affects the migration performance most, the concept of dirty page rate is proposed. The dirty page rate is defined as the number of "dirty" memory pages an application can generate per second. In Linux operating system, the size of a memory page by default is 4096 bytes. A memory page is considered to be "dirty" even if only one byte of the memory page is modified. To simulate the application's memory

behavior, the simulator modifies the memory content byte by byte in each iteration. In this thesis, the stateful application simulator is implemented in the C language. In order to understand the behavior of an C based application, tool uftrace [27] can be used to trace the application. The trace result is as Fiugre 5.2 shows. The generation of a random byte to modify the memory content takes around 0.12 us. So, in theory, the dirty page rage of the simulator is around 2034 pages per second (when the application allocates more than 2034 pages).

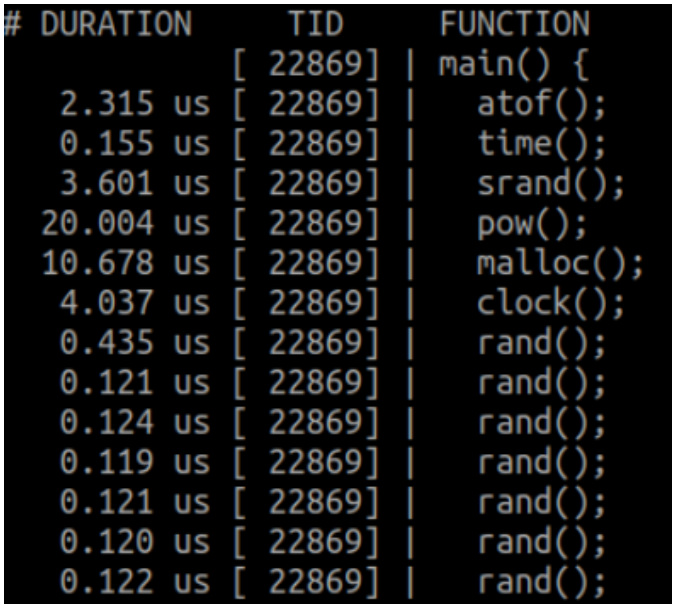


Figure 5.2: Trace result of stateful application simulator

5.2 Application Dirty Page Rate Control

In Section 5.1, the stateful application simulator is implemented with the maximum dirty page rate it can achieve. However, in the actual case, the change of the application's memory content will not always stay at the highest rate but statistically converge to a mean speed. In order to better simulate the state of an application, the previous stateful application simulator needs some modification to simulate the behavior of specific dirty memory pages rate.

$$dpr = \frac{\text{changedBytes}}{\text{pageSize} \cdot \text{elapsedTime}} \tag{5.1}$$

$$dpr_{\text{NEW}} = \frac{\text{dirtyPages}}{\text{timeInterval}} \tag{5.2}$$

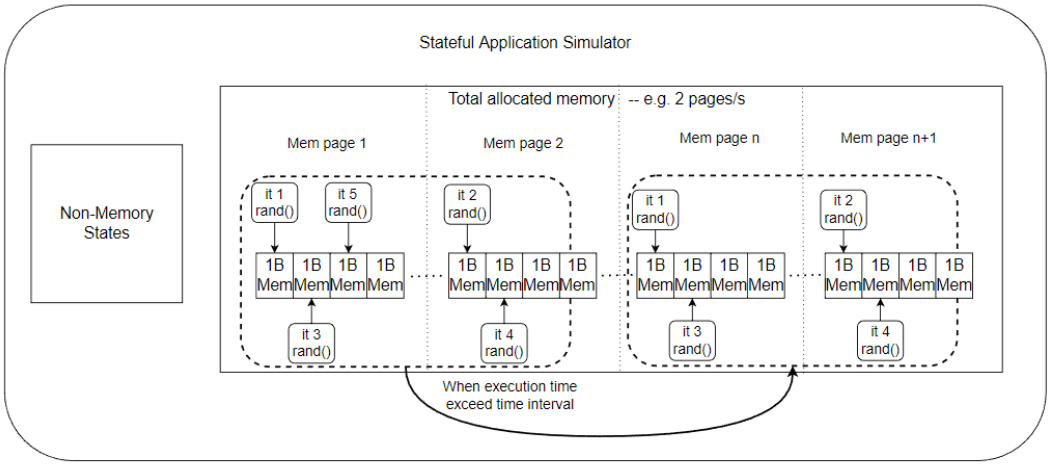
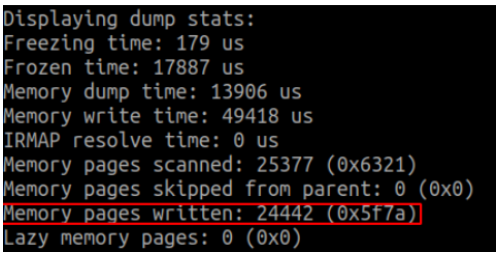


Figure 5.3: Stateful application simulator with controllable dirty page rate

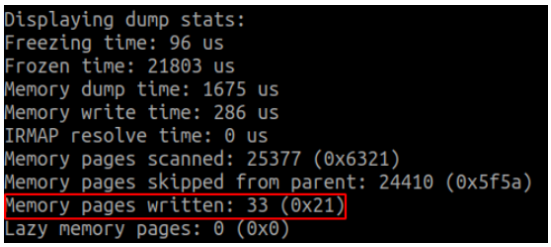
In order to reach this target, the key is related to the definition of the dirty page rate. In most cases, the dirty page rate is defined as the fraction of the number of changed bytes to the memory page size and elapsed time as Equation 5.1 show. With this definition, it is tough to precisely control the dirty page rate since it is difficult to precisely control the number of changed bytes in the memory content within a specific elapsed time. To solve this problem, the dirty page rate could be re-defined as Equation 5.2 shows. The dirty page rate is defined as the fraction of the number of dirty pages and the time interval to compare the memory content. In the container migration scenario, a memory page is considered "dirty" even if only one byte in the memory page is changed. Meanwhile, for a single memory page, even if its content has changed multiple times between two checkpoint procedures, this memory page will still be considered as a single dirty page. With these two special properties, the stateful application simulator with controllable dirty page rate can be implemented as Figure 5.3 shows. Between the comparing time interval, the application simulator iterates page by page among a specific set of memory pages. And in each iteration, only one byte in the memory page is modified. After changing one byte in a memory page, the application checks whether the comparing time interval is exceeded or not. Since the execution time to change one byte in a memory page is very short, the checking of the time interval is very precise and sensitive. When the comparing time interval exceeds, the iteration moves to the next set of memory pages. When the external tools (e.g., CRIU) measures the number of dirty page of the application with a period as the comparing time interval, the calculated dirty page rate will be stable and precise as we desire. In theory, if we assume it takes 0.12us to change one byte in the memory content, the maximum controllable dirty page rate could reach $8.3 * 10^6$ pages per second (only

if the application allocates more than $8.3 * 10^6$ memory pages).

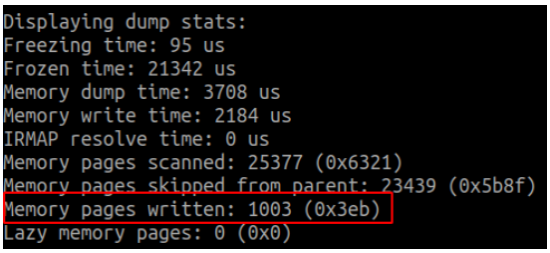
In order to validate the controllable dirty page rate simulator design, a test is done with the new simulator and obtains the result as Figure 5.3 shows. In this test, the new implementation of the simulator is containerized. It allocates 100MB of memory at the initialization phase. As Figure 5.4b shows, in the pre-dump procedure, Podman dumped 24442 memory pages. When the target dirty page rate is set to 30 pages/s, 33 dirty memory pages are obtained after 1 second as Figure 5.4b shows. And 1003 and 5004 dirty pages are obtained when the target dirty page rates are set to 1000 pages/s and 5000 pages/s. The additional dirty pages might be generated by the non-controllable memory content of the application simulator (e.g., memory used by the timer). When the target dirty page rate is large enough, the effect of the non-controllable memory becomes negligible.



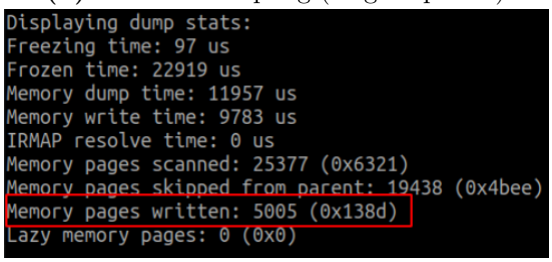
(a) Podman pre-dump log



(b) Podman dump log (target dpr=30)



(c) Podman dump log (target dpr=1000)



(d) Podman dump log (target dpr=5000)

Figure 5.4: Controllable stateful application simulator container checkpoint test

5.3 Applications Containerization

The containerization of the application means running the application inside a container. And the container is created and managed by the container engine. Since performing container migration requires the container engine to have a good integration of CRIU, Podman is selected as the container engine. The container engine creates a container from the container image. Due to the migration of the Podman container does not yet support the migration of the container's file system, the application's executable should be pre-built into the container image. Moreover,

the container image should be synchronized to the target host machine before the container migration. In this section, the stateful application simulator introduced in Section 5.1 is selected as the application to be containerized. There are two ways to build the application executable into a container image: from scratch and from a base image. The instructions for Podman to build a container image could be written in "Container file" or "Dockerfile" format.

Table 5.1: Application container image size

	scratch	**ubuntu**	**alpine**	**busybox**
Base image size	0	75.2MB	5.87MB	1.46MB
App library	shared	shared	static	static
App size	20KB	20KB	872KB	872KB
Container image size	4.07MB	75.2MB	6.77MB	2.37MB

As Table 5.1 shows, when the container image is built from scratch or from base image ubuntu, the applications can be compiled with dynamic linking to the shared library. In this condition, the size of the stateful application simulator is very small. However, when building the container image from scratch, the shared libraries and other tools to support the running of the application are copied into the container image, which makes the final container image size much larger. The container image built from the ubuntu base image has a very similar size to the base image. However, the base image itself is very large. So ubuntu base image is not the optimal method to create the container image. Because of the minimization property of base image alpine and busybox, the application should be statically compiled with the used libraries. In this case, the size of the application's executable becomes much larger. However, since the base image of the busybox is very small, the final container image is small enough and suitable for the migration scenario.

5.4 Process migration with CRIU

The study of this thesis aims at statefully migrating a containerized application. As introduced in Section 3.2, the nature of container migration with CRIU is the migration of a tree of processes. So, the first experiment is to validate the process migration with CRIU is stateful or not. The simplest stateful application, a counter as Listing 5.1 shows, is selected as the process to be migrated. The migration topology is as Figure 5.5 shows.

The migration procedure and result are as Figure 5.6 shows. The first step is to run a process of a counter on VM 1 and obtain its PID (Step 2 in Figure 5.6a).

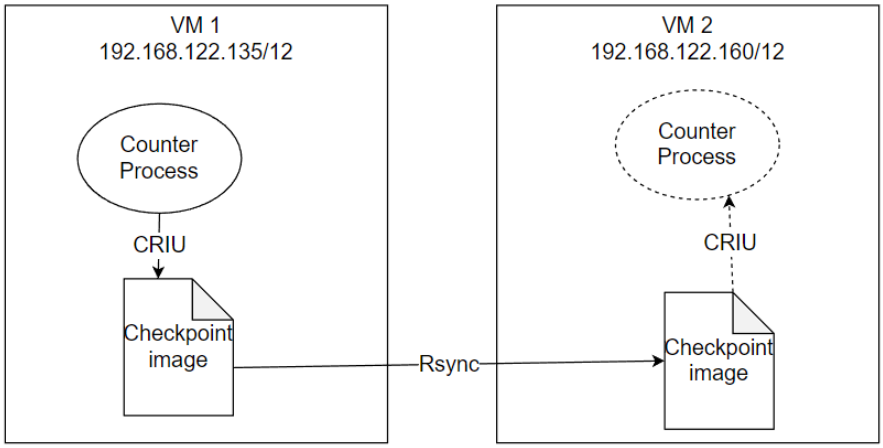
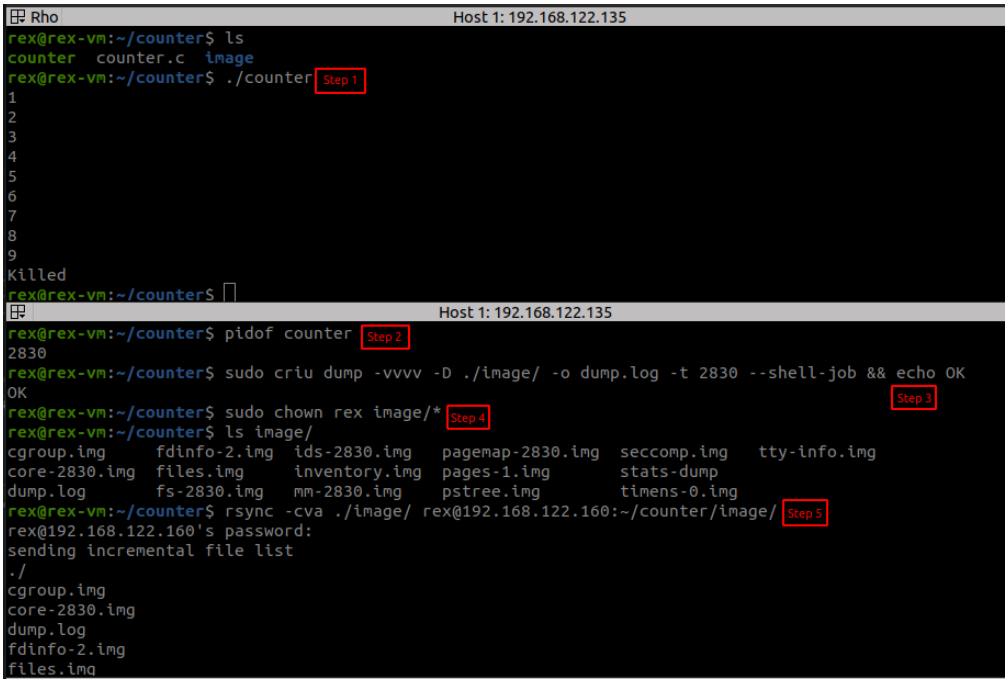


Figure 5.5: Process migration test topology

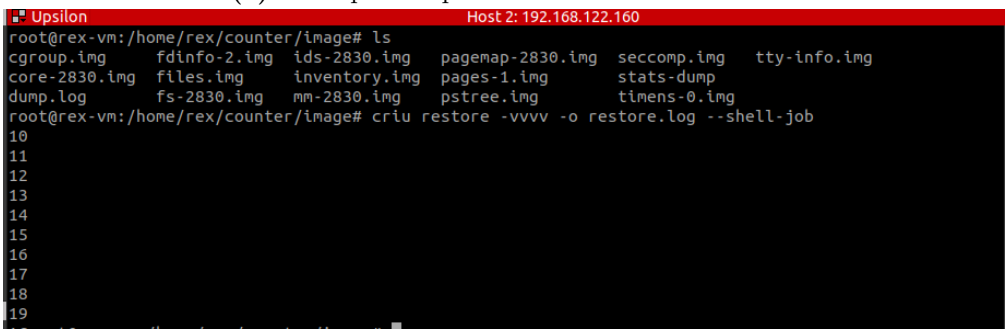
Then, the process can be checkpointed with CRIU (Step 3 in Figure 5.6a). Since the process can only be checkpointed under the root permission, the owner of the checkpoint images should be changed to a normal user (Step 4 in Figure 5.6a) in order to allow rsync to synchronize them to the remote (Step 5 in Figure 5.6a). Once the checkpoint image is synchronized to the remote VM, CRIU can restore the process from the checkpoint image. As Figure 5.6b shows, the restored counter starts counting from where it was checkpointed. In conclusion, the process migration with CRIU is stateful.

5.5 Container Migration Procedure

There are two methods to migrate a container with CRIU. The first way is to checkpoint and restore the container from external. In this method, CRIU considers the container as a tree of processes and directly checkpoints this process tree. In this case, the migration procedure will be the same as the process of migration introduced in Section 5.4. However, since the containers are running inside a complex system (including the container runtime, container engine, and so on), checkpointing the container from external might lead to some errors or unexpected behavior. One of the most common errors is that the restored container is out of the control of the container engine. So, in real practice, people prefer the "internal" method, which checkpoints and restores the container from the container engine. With this method, the container engine calls CRIU to perform the container checkpoint and restore procedure. So the container engine is aware of the migration of containers and adopts necessary management operations. Compared to the external method, the internal method is simpler and safer. However, the internal



(a) Checkpoint a process with CRIU on VM 1



(b) Restore process with CRIU on VM 2

Figure 5.6: Process migration with CRIU

method needs the support from the container engine. Among the popular container engines, we found that Podman has the best integration whit CRIU. So Podman is selected as the container engine in this study. But even in this case, Podman only supports cold and pre-copy migration.

Due to the limited support for the migration method of the Podman container engine, the following study will only focus on cold and pre-copy migration. The experiment topology is as Figure 5.7 shows. There are 2 VMs running on the host machine with the following setting:

- OS: Ubuntu 20.04

- Linux kernel: 5.15.0-46-generic
- CPU: Intel® Core™ i7-7700HQ CP @2.8GHz
- Memory: 16GiB DDR4 System Memory
- Network: Intel® Dual Band Wireless-AC 8265 (WiFi 5)
- Storage: TOSHIBA (1TB HDD)

The configuration of each VM is as follows:

- OS: Ubuntu 20.04
- Linux Kernel: 5.8.18-050818-generic
- CPU: 2 virtual CPU cores @2.0GHz
- Memory: 4GiB DDR4 System Memory
- Network: Bridged network
- Storage: VirtIO Disk (50GB HDD)

Podman in version 4.2.0-dev and CRIU in version 3.17 are installed in each VM. The experiment aims to validate the stateful migration of containerized applications and compare the migration performance of cold and pre-copy migration methods.

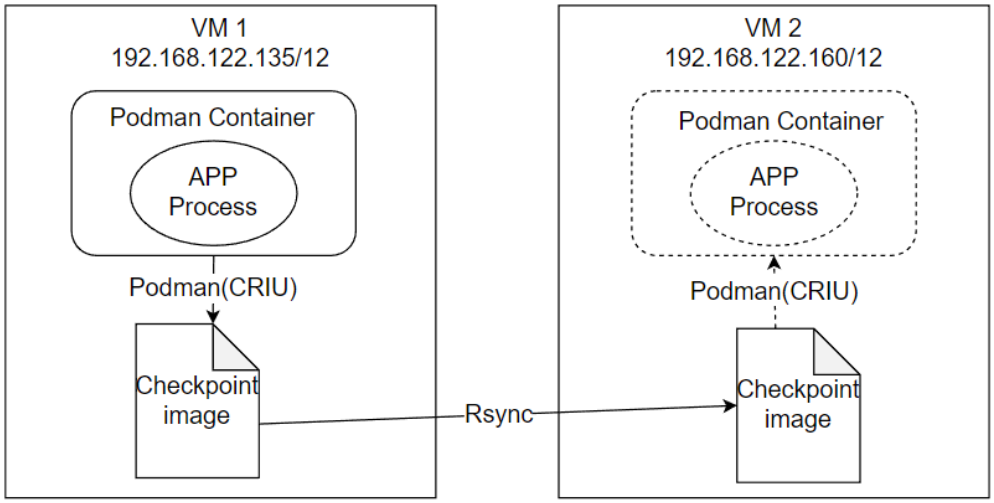


Figure 5.7: Podman container migration topology

5.5.1 Cold Migration

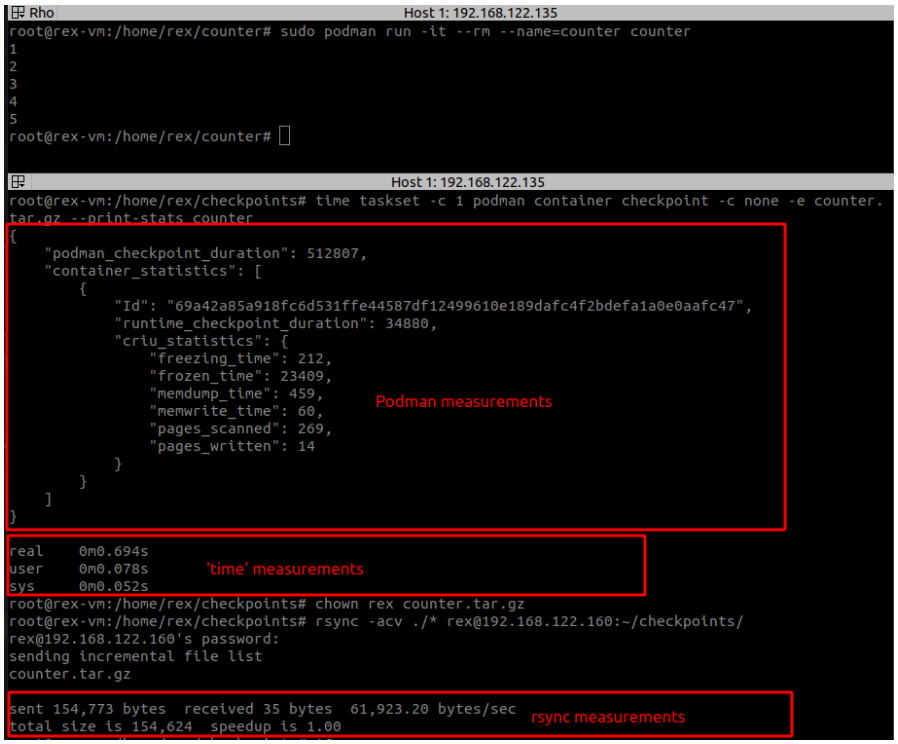
In this experiment, a simple counter as introduced in Section 5.1, is containerized and migrated using the cold migration method. The migration procedure is as Listing 5.2 shows. First, run a container with the counter on VM 1. Then checkpoint the container using Podman's container checkpoint operation. In the middle loop, the Ubuntu command `taskset` is used to limit the checkpoint process to use only one CPU core. And in the outer loop, the Ubuntu tool `time` is used to measure the CPU usage of the checkpoint procedure. Since the Podman container checkpoint function is working under the root user, the ownership of the generated checkpoint image should be changed to the normal user to simply the synchronization process. Then, the checkpoint image is synchronized to VM 2 using rsync. On VM 2, we can restore the container using Podman's container restore command. After the restoration, we can enter the container to see the working status of the counter.

Listing 5.2: Podman container cold migration procedure

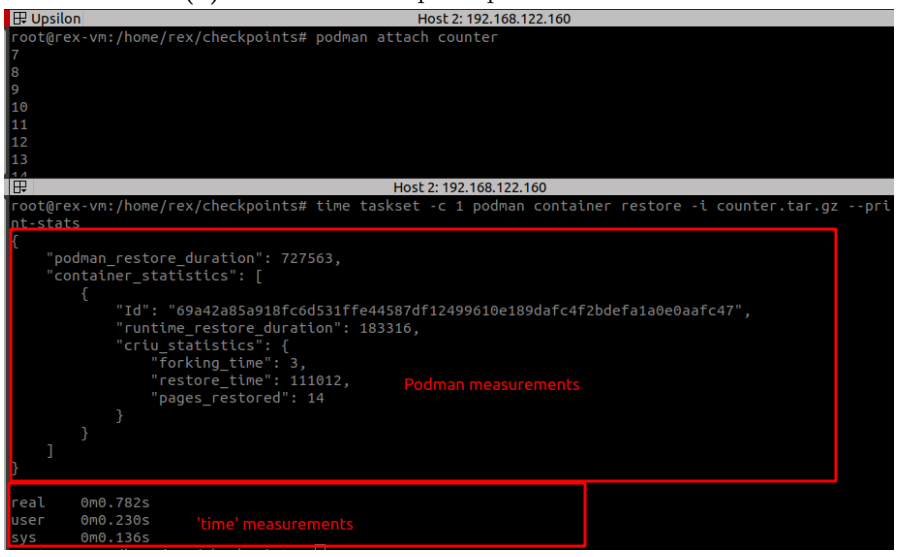
```
# On VM 1 — Checkpoint
> sudo podman run \
        --runtime /usr/bin/runc
        -it --rm --name=counter counter
> time taskset -c 1 podman container checkpoint \
        --runtime /usr/bin/runc
        -c none   # No compression
        -e counter.tar.gz \ # Export checkpoint location
        --print-stats \ # Print statistics measured by Podman
        counter # Target container to be checkpoint
> chown <user_name> counter.tar.gz
> rsync -acv ./* <user@dst_ip>:~/checkpoints/

# On VM2 — Restore
> time taskset -c 1 podman container restore \
        --runtime /usr/bin/runc
        -i counter.tar.gz \
        --print-stats
> podman attach counter # Enter the restored container
```

The actual migration test procedure and the result are shown in Figure 5.8. The container on VM 1 is checkpointed when the counter counts to number 5. The checkpoint duration measured by Podman is around 0.5 seconds. The external tool `time` measures the checkpoint procedure lasts for 0.694 seconds, which is longer than Podman's measurement. The reason for this difference is that the measurement of `time` starts from the checkpoint command executed, which includes some warmup time for the actual checkpointing procedure. After the container restoration on VM2 is finished, we immediately enter the container and find out the counting is from number 7, meaning the container migration is stateful. (The

missing of number 6 is due to the manual re-entering of the container is too slow.)



(a) Container checkpoint procedure on VM 1



(b) container restore procedure on VM 2

Figure 5.8: Cold migration with Podman container

5.5.2 Pre-Copy Migration

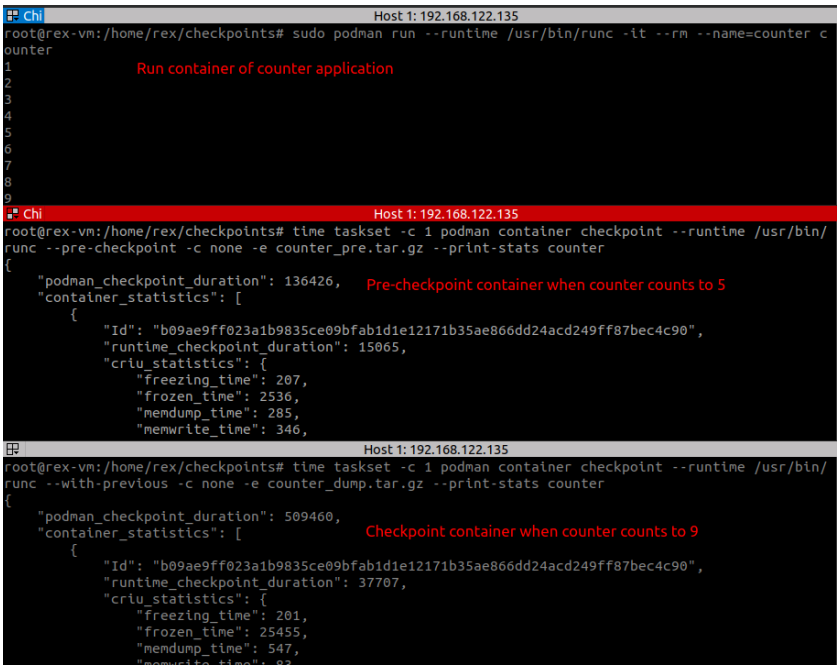
In this experiment, a simple counter as Section 5.1 introduced is containerized and migrated using the pre-copy migration method. The migration procedure is as Listing 5.3 shows.

Listing 5.3: Podman container cold migration procedure

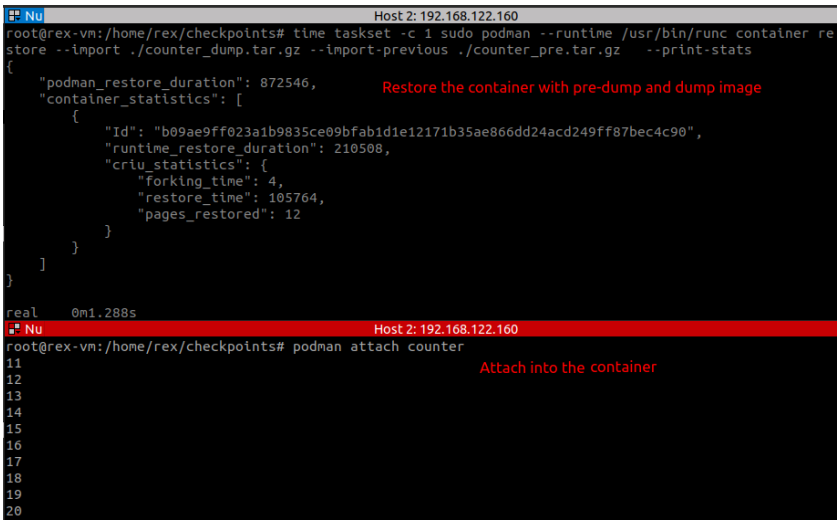
```
## On VM 1 — Checkpoint
> sudo podman run \
        --runtime /usr/bin/runc
        -it --rm --name=counter counter
# Pre-checkpoint container
> time taskset -c 1 podman container checkpoint \
        --runtime /usr/bin/runc \
        --pre-checkpoint \
        -c none \
        -e counter_pre.tar.gz \
        --print-stats \
        counter
> chown <user_name> counter_pre.tar.gz
> rsync -acv ./* <user@dst_ip>:~/checkpoints/

# Checkpoint container based on pre-checkpoint
> time taskset -c 1 podman container checkpoint \
        --runtime /usr/bin/runc \
        --with-previous \
        -c none \
        -e counter_dump.tar.gz \
        --print-stats \
        counter
> chown <user_name> counter_dump.tar.gz
> rsync -acv ./* <user@dst_ip>:~/checkpoints/

## On VM2 — Restore
> sudo podman container restore \
        --runtime /usr/bin/runc  \
        --import ./counter_dump.tar.gz \
        --import-previous ./counter_pre.tar.gz   \
        --print-stats
> podman attach counter # Enter the restored container
```

In pre-copy migration, the first checkpoint needs to use the option `-pre-checkpoint` in the Podman command. With this option, Podman only checkpoints the memory used by the container and leaves the container running after the checkpoint procedure is finished. After the first checkpoint image is synchronized, we need to checkpoint the container again with `-with-previous` option. In this case, Podman looks up the previous checkpoint and compares it to find the changed memory pages (dirty memory pages). In this checkpoint, Podman only checkpoint the dirty

memory and the rest stats of the container. In the restoration procedure, we need to identify the location of the first and second checkpoint images. In this case, Podman will combine the two checkpoints and restore the container to the last checkpoint state.



(a) Container checkpoint procedure on VM 1



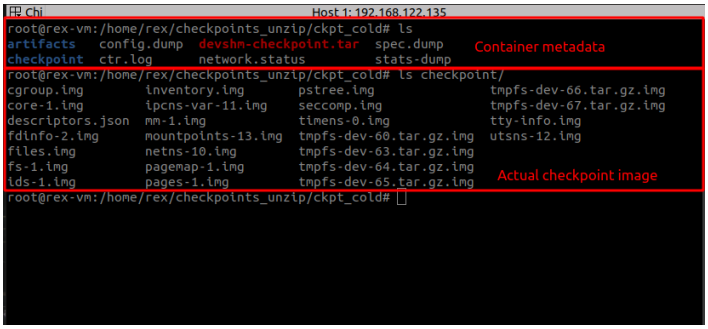
(b) container restore procedure on VM 2

Figure 5.9: Pro-copy migration with Podman container

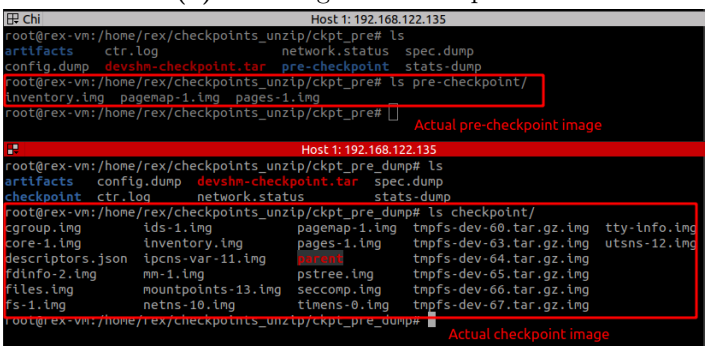
The test result is as Figure 5.9 shows. The first checkpoint procedure is done when the counter inside the container counts to number 5. And the second checkpoint procedure is done when the counter counts to number 9. After the container is restored on VM2, we enter the container immediately and see the container is counting form number 11, which also means the pre-copy migration is statful, and the container is restored to the last checkpoint state. (The missing of number 10 is due to the manual re-entering of the container is too slow.)

5.5.3 Comparison of cold and pre-copy migration

In this section, the performance of cold and pre-copy migration with the Podman container engine is discussed. As Section 2.2.2 introduced, the major difference between these two techniques is how they migrate the memory state. The implementation of Podman follows the same concept. As Figure 5.8a shows, the checkpoint image of cold migration includes all the container states. For the pre-copy migration, as Figure 5.10b shows, the first checkpoint image only contains the memory pages used by the container. At the second checkpoint, Podman dumps the dirty pages, and the rest of the container states.



(a) Cold migration checkpoint



(b) Pre-copy migration checkpoint

Figure 5.10: Checkpoint image content comparison

For a stateful application, most of its stat is the memory content. So the pre-copy migration seems to be able to remove one of the major components in the migration downtime: the checkpoint image transmission time. However, the efficiency of these two techniques needs a deeper study. So, a simple migration performance test is done to study the size of the checkpoint image and the CPU usage during checkpointing. The stateful application simulator with controllable dirty page rate mentioned in Section 5.2 is used in this test. The target dirty page rate of the application simulator is set to 100 pages/s. For the pre-copy migration, the second checkpoint is done 1 second after the first checkpoint is finished. So, in theory, there should be around 0.49 MB of dirty memory to be transmitted at the second checkpoint. The performance test result is as Figure 5.11 to Figure 5.13 show.

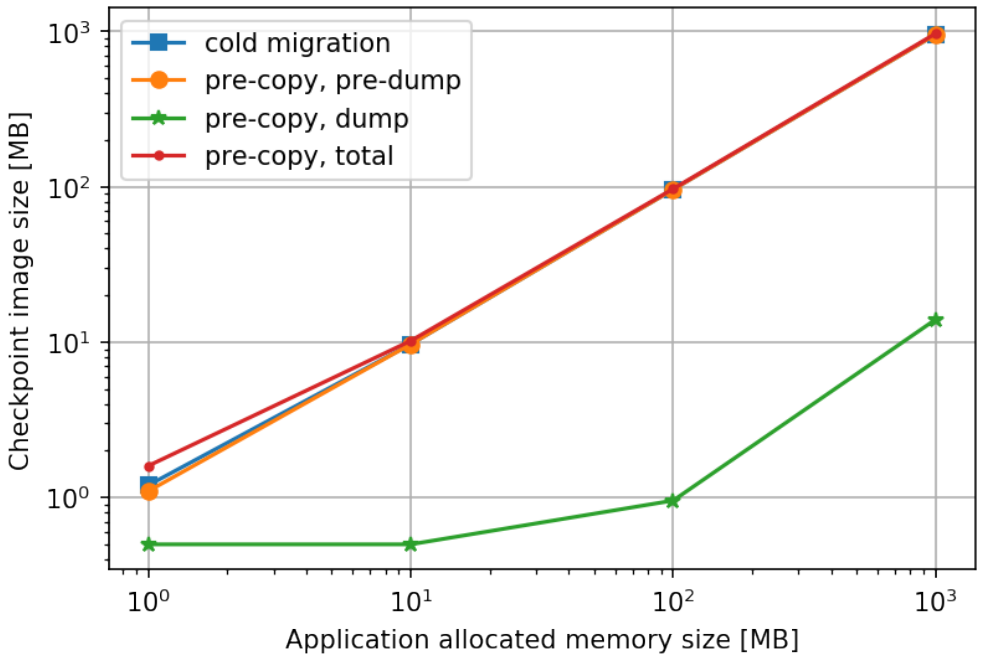


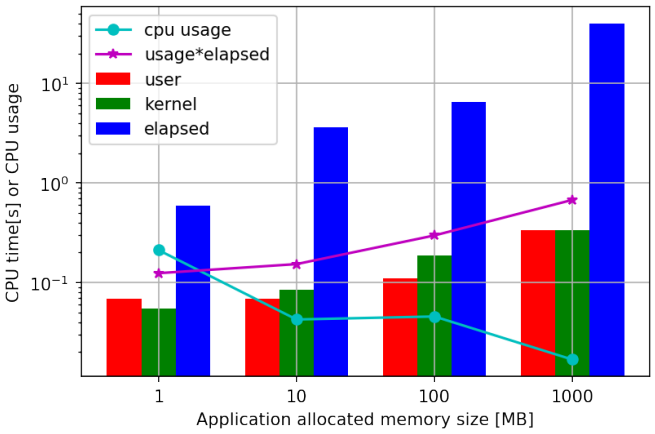
Figure 5.11: Checkpoint image size comparison

Checkpoint image size comparison The checkpoint image size test result is as Figure 5.11 shows. When the application's memory usage is small, the second checkpoint image size in pre-copy migration is close to the theoretical value (0.4 MB). However, the summation of the size of the first (pre-dump) and second (dump) checkpoint images in the pre-copy migration method are higher than the checkpoint image of cold migration. Since the small checkpoint image can be quickly transmitted in a normal speed network, the pre-copy migration could be

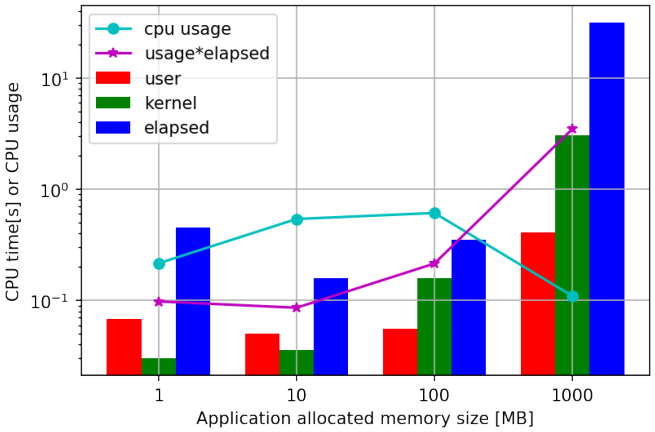
considered to have a lower overall efficiency in this case. When the application's memory usage is large, we can notice that the second checkpoint image size in pre-copy migration is increasing. The main reason for this phenomenon is that when the application's memory usage is large, the first checkpoint image size will also be large. It takes a long time to write the first checkpoint to the disk and transmits it to the remote. The frozen application is released at some point before the first checkpoint image is completely stored and transmitted, which leads to more dirty pages to be transmitted at the second checkpoint. However, even in this case, the size of the second checkpoint image is much smaller than the cold migration checkpoint image. At this moment, the overhead of the pre-copy migration becomes negligible. The efficiency of checkpoint image synchronization becomes very high in pre-copy migration when the target application allocates a huge amount of memory.

CPU usage comparison in container checkpoint phase The CPU usage during the checkpoint procedure is as Figure 5.12 shows. For both cold migration and pre-copy migration, the overall CPU usage is very low and decreases as the application's memory usage increases. The main reason is that the major operation in the checkpoint procedure is the copy operation which requires little CPU resource. Meanwhile, it is because the elapsed time of the procedure is very large and increases significantly. In fact, the CPU time for the entire checkpoint procedure increases when the application's memory usage increases, as the line "usage*elapsed" in the figure shows. From the total CPU time usage (pre-dump + dump for pre-copy migration) point of view, the pre-copy method uses more or less similar CPU time to cold migration. However, when the application's memory usage is large, the elapsed time and CPU time (kernel+user) in the second checkpoint are much less than the one in cold migration. Such kind of behavior can significantly reduce the migration downtime due to the checkpoint procedure.

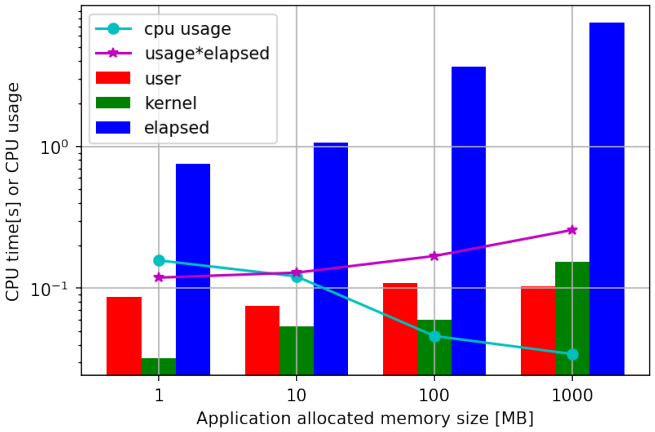
CPU usage comparison in container restore phase The CPU usage during the checkpoint procedure is as Figure 5.13 shows. The CPU usage in the container restore phase behaves similarly to the checkpoint phase. To be noticed, the overall CPU time and the elapsed time of cold and pre-copy migration are very similar. It means that merging two checkpoint images of the pre-copy migration requires little CPU resources and does not affect much on the overall performance.



(a) Cold migration checkpoint CPU usage

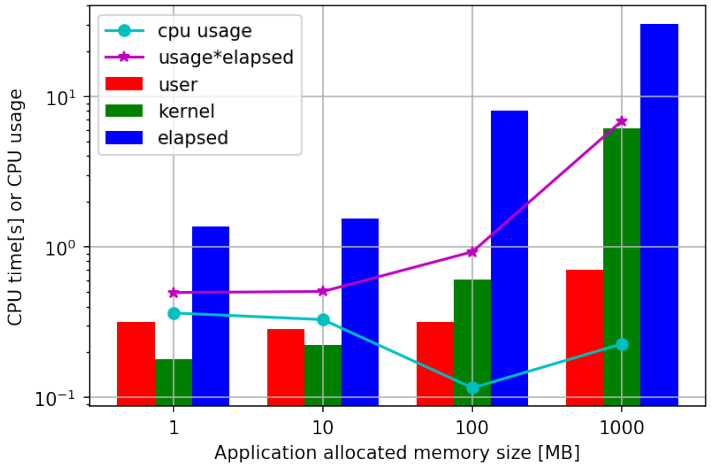


(b) Pre-copy pre-dump CPU usage

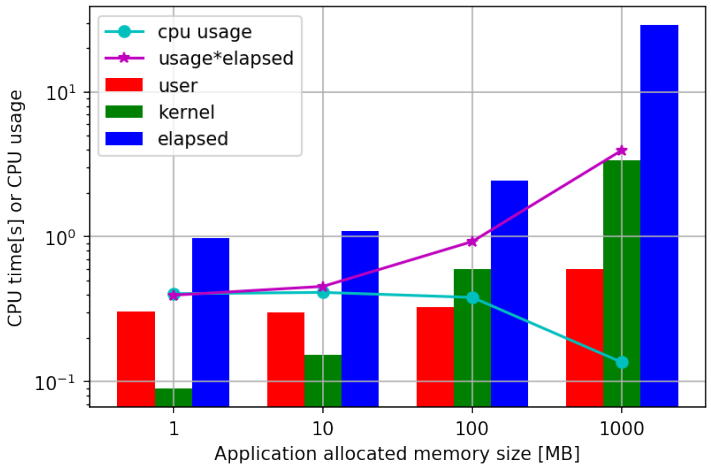


(c) Pre-copy dump CPU usage

Figure 5.12: CPU usage comparison in checkpoint procedure



(a) Cold migration



(b) Pre-copy migration

Figure 5.13: container restore CPU usage comparison

Chapter 6

Realization Stack: Overlay Network

In recent works related to container migration (e.g., [14]), the research majorly only focuses on reducing migration downtime. The container's networking problem during the migration usually is not discussed. In some cases, the networking problem is simply considered to be managed by the orchestrator but without introducing any implementation details. However, in the scenario of containerized mobile services migration, the networking problem is not avoidable since mobile services usually have the capability to communicate with external mobile clients. The network management problem becomes even more crucial when containerized services have "always connected" connections. In such kinds of migration scenarios, the requirement of network management during migration could be summarized into the following two aspects:

- After service container migration, the external clients should be able to restore the connection to the migrated service without any additional operation.
- After service container migration, the connection should restore as fast as possible.

6.1 Stateful edge application

As introduced in Section 5.1, for a local application, the major and dynamic part of its states are related to the application's memory content, which contributes to the majority part of the checkpoint image. The other states (e.g., process credentials, pipes information, and socket state) are relatively less in size and typically static. However, in edge applications, the socket state becomes one of the most important states and contributes a significant part to the checkpoint image.

Since the TCP protocol is commonly adopted as the communication protocol in edge applications today, the study of container's network migration in this thesis will focus on migrating TCP connections.

To perform the study, the MQTT broker is selected as the edge service to be migrated. MQTT is one of the most well-known and popular application layer publish-subscribe protocols based on TCP. It is commonly adopted as the standard communication protocol in IoT and edge applications. The protocol defines two types of network entities: client and broker. According to the MQTT clients' behavior, the clients can be divided into publishers and subscribers. As Figure 6.1 shows, clients will register their interested topic and other information in the broker's registration table. The communication messages are identified by topics. When a publisher sends the messages to the broker, the broker will forward the messages to subscribers interested in the corresponding topic. When the QoS level of the communication is set to 1 or 2, acknowledgment from the subscriber for each message is needed in order to guarantee the reliable communication. The messages that are sent but not yet acked are considered to be "in-flight" and stored in the "in-flight" message queue. When the "in-flight" message queue is full, the new coming messages will not be sent to the subscriber immediately but queued inside the message queue for different topics. In the study of this thesis, the registration table, message queue, and in-flight message queue of the MQTT broker are the memory states that need to be preserved during the service migration. The connections between the MQTT broker and corresponding clients are the network states to be migrated.

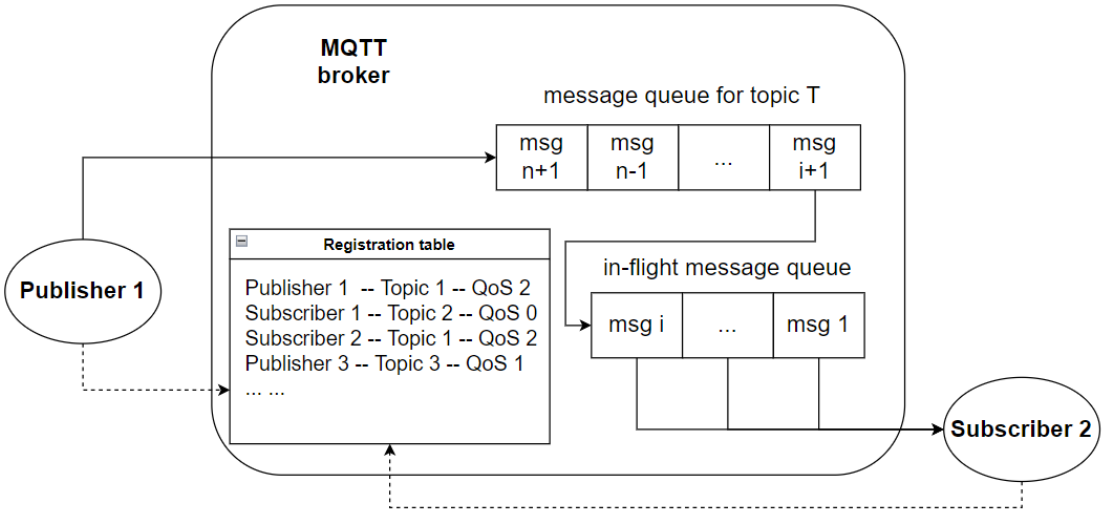


Figure 6.1: MQTT broker states

6.2 Linux TCP connection repair

In order to realize the migration of a network connection, the first step is to explore the mobility of the underlying protocol of the connection. In the study of this thesis, the underlying protocol is TCP. From Linux kernel version 3.5, the TCP socket includes a special option "TCP_REPAIR". This option is designed for CRIU to perform active TCP connection migration in 2012 [28]. When this option is used, the TCP socket will be switched into a special mode where any native TCP actions performed on the socket do not result in anything. In this condition, CRIU could dump the state of the TCP connection on the original host and restore the connection on the new host.

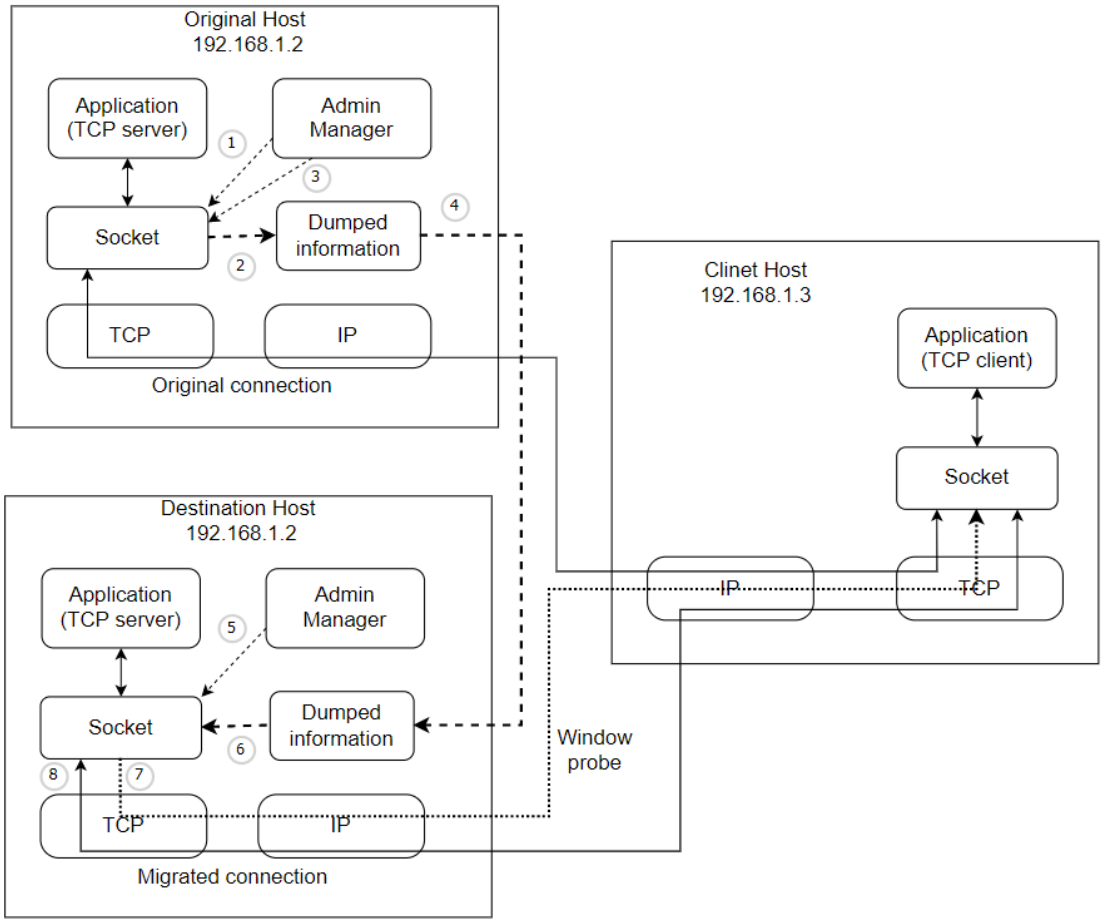


Figure 6.2: TCP repair procedure

The migration procedure is shown as Figure 6.2. Assume the initial TCP connection is in the "established" state, and the migration is performed on the

TCP server side. The first step to perform the migration is to switch the socket on the TCP server into repair mode by an admin manager (step 1). The TCP client will stay in the "established" state and has no idea about the changes in the TCP server. Then, the admin manager starts to collect the information about the connection state (step 2). The collected information includes the maximum segment size, window size, packet sequence number, timestamps, and the content in the send and receive queue. After collecting all the required information, the admin manager closes the socket on the original host (step 3) and moves the collected information to the destination host (step 4). To be noticed, when the socket is closed during the repair mode, no "FIN" and "ACK" packets will be sent. So in the example case, the remote will not be aware that the original TCP server is closed.

Due to the property of TCP protocol, the connection can only be migrated to the host in the same subnet with the same IP address as the original host. To restore the connection, the admin manager will create a new socket and put it immediately into repair mode (step 5). Then, the collected information from the original host will be restored to the newly created socket (step 6). Once the new socket has been restored to an approximated state as the one on the old host, the admin manager switches the socket to the "established" state and sends a window probe to the remote client (step 7) in order to restart the traffic. After the remote client receives the probe, the socket and the connection resume to normal working states (step 8).

Since the Linux TCP repair option is specially designed for CRIU, CRIU has a very nice integration with the network migration-related API in the Linux socket. CRIU can act as the admin manager to realize the operation on the socket status. It is often used to migrate processes with TCP connections among Linux machines. However, this network migration method requires the migration destination host to be in the same sub-net to have the same IP address as the original host. There are more solutions to realize this requirement on bare machines. But in a containerized environment, additional topological design is needed in order to fulfill the IP address requirement.

6.3 Linux Namespace

As mentioned in Section 6.2, the migration of a TCP connection requires the source and destination container have the same IP address. To fulfill this requirement, manipulating the container's network namespace could be the solution. A namespace is a feature in the Linux kernel that partitions and isolates the resources for different processes on the host machine. It is the fundamental aspect of containers (e.g., LXC, Docker, and Podman). Every time a container is created, the container engine creates a set of namespaces to provide the isolation layer for it. On the

Linux system, there are eight types of namespaces:

- Cgroup: Isolates Cgroup root directory.
- IPC: Isolates system V IPC, POSIX message queues.
- Network: Isolates network devices, stacks, ports, etc.
- Mount: Isolates mount points.
- PID: Isolates process IDs.
- Time: Isolates boot and monotonic clocks.
- User: Isolates user adn group IDs.
- UTS: Isolates Hostname and NIS domain name.

For the Podman container, there are two ways to manipulate the container's namespaces. The first way is to create a container and then manipulate its network namespace, as the command in Listing 6.1 shows. With the command *lsns*, all the namespaces in the host system will be listed. The networking setting of the container can be easily manipulated when the correct network namespace ID is found.

Listing 6.1: Manipulate container's default namespace

```
# Run a container only with lo interface
> sudo podman run -d --net=none alpine sh \
  -c 'while sleep 3600; do :; done'

# Find the network namespace of the container
> lsns
3022332257 net 2  9854 root sh -c while sleep 3600; do :; done

# Create virtual network interface (ifA) for the container
> ip link set netns 9854 dev ifA

# Bring up the network namespace
> nsenter -t 9854 -n ip link set ifA up

# Assign a specific IP address for the created namespace
> nsenter -t 9854 -n IP addr add 172.16.0.10/12 dev ifA
```

However, identifying the container's namespaces is not always easy, especially when plenty of containers are running on the same host machine. So, the second way is to create a customized network namespace and then bind the container to that network namespace. The procedure is shown in Listing 6.2.

Listing 6.2: Bind a container to a customized network namespace

```
# Create and bring up a customized network namespace "nsA" on host
    machine
> ip netns add nsA
> ip link set netns nsA nsA_if
> ip -n nsA addr add 172.16.0.10/12 dev nsA_if
> ip -n nsA link set nsA_if up

# Bind a container to the customized network namespace
> sudo podman run -d --network ns:/run/netns/test alpine sh \
  -c 'while sleep 3600; do :; done
```

6.4 Podman container overlay network

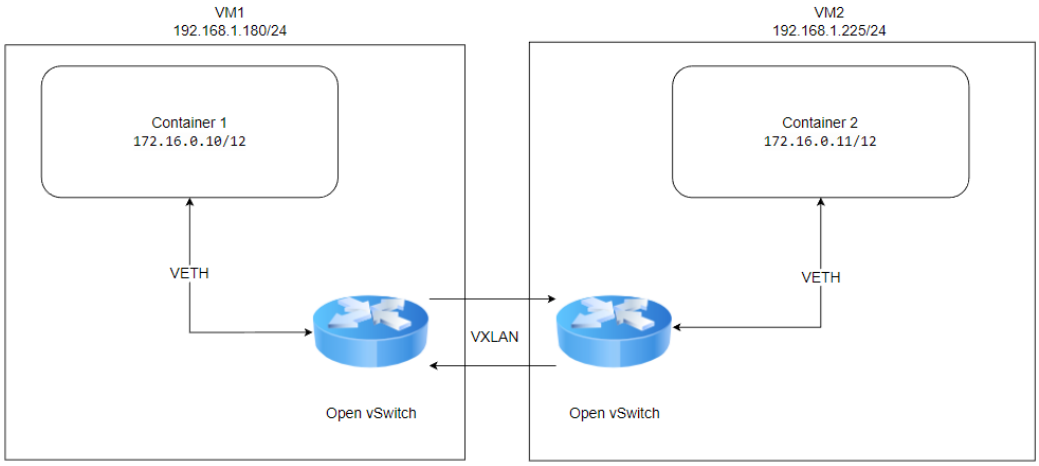


Figure 6.3: Podman container overlay network topology

With the manipulation of a container's network namespace, it is possible to guarantee that the container has the same IP address after migration. However, the networking problem of container migration is not entirely solved yet. For security and management issues, the network of different sets of mobile applications should be isolated from each other and the host. In container architecture, the most common solution is using the bridge network. For most container implementations, the bridge network is the default network mode. Containers connected to the same network bridge are assigned IP addresses in the same subnet and are able to communicate with each other directly. However, the network bridge is reachable only for containers on the same machine. Due to the NAT conversion on the network bridge, it is impossible to reach the container from other machines unless

the containers are exposed to the host through the port mapping techniques. However, in this case, from the mobile client's point of view, the destination of the connection is not the container but the host server. If the container migrates, additional operations on the mobile client are needed to recover the connection, which violates the first requirements mentioned at the beginning of this chapter. To the problem, the study of this thesis proposed the container overlay network solution.

An overlay network is a logical network created on top of an existing physical network. It enables the possibility of creating a distributed network among multiple host machines. Some containerization tool, such as Docker, has provided overlay network option for the user during the creation of the container. However, the containerization tool used in this study, Podman, does not provide any option to directly create an overlay network with containers. So, an overlay network structure for the Podman container is designed from scratch with the combination of Open vSwitch (OvS) as Figure 6.3 shows. There is an OvS installed on each host machine. The OvSes are pre-configured to connect to each other. The containers running on each host will be assigned static IP addresses belonging to the same subnetwork by manipulating their network namespace as mentioned in Section 6.3. Then, the corresponding virtual network interface of each container will be "plugged" into the OvS with VETH. At this moment, the overlay network is created, and containers on different host machines can directly reach each other, as Figure 6.4 shows.

6.5 Podman container migration with overlay network

With the techniques mentioned in Section 6.2 to 6.4, the network connections of a container are possible to be migrated. In the study of this thesis, the migration test is done with an MQTT broker. The network topology for the migration is designed as Figure 6.5 shows. There are two VMs act as the edge servers in the edge network. On each VM, OvS is installed and configured to connect to each other. At the beginning, the containerized MQTT broker and publisher are running on VM1 and containerized MQTT subscriber is running on VM2. All the containers are connected to corresponding OvS and work under the same overlay network segment. In order to make the states of the MQTT broker easier to observe, the tool "tc" is used to limit the up-link speed of VM1's network interface, which leads the messages to be queued inside the MQTT broker. The target of this experiment is to migrate the container with the MQTT broker from VM1 to VM2. If the connection between MQTT clients and the broker can recover after the migration, the queued messages can be delivered to the subscriber. In this case, the container's network connection migration can be considered successful.

```
root@rex-vm:/home/rex# podman exec -it 0c /bin/sh
/ # ifconfig
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

podA-netns Link encap:Ethernet  HWaddr A2:F1:A6:C3:89:55
          inet addr:172.16.0.10  Bcast:0.0.0.0  Mask:255.240.0.0
          inet6 addr: fe80::a0f1:a6ff:fec3:8955/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:22 errors:0 dropped:0 overruns:0 frame:0
          TX packets:12 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1732 (1.6 KiB)  TX bytes:936 (936.0 B)

/ # ping 172.16.0.11
PING 172.16.0.11 (172.16.0.11): 56 data bytes
64 bytes from 172.16.0.11: seq=0 ttl=42 time=9.303 ms
```

(a) Ping result of container 1 on VM 1

```
root@rex-vm:/home/rex# podman exec -it 15 /bin/sh
/ # ifconfig
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

podB-netns Link encap:Ethernet  HWaddr 0A:0F:B2:95:FC:E8
          inet addr:172.16.0.11  Bcast:0.0.0.0  Mask:255.240.0.0
          inet6 addr: fe80::80f:b2ff:fe95:fce8/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:11 errors:0 dropped:0 overruns:0 frame:0
          TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:866 (866.0 B)  TX bytes:796 (796.0 B)

/ # ping 172.16.0.10
PING 172.16.0.10 (172.16.0.10): 56 data bytes
64 bytes from 172.16.0.10: seq=0 ttl=42 time=3.168 ms
```

(b) Ping result of container 2 on VM 2

Figure 6.4: Podman container overlay network test result

In order to perform the migration of the MQTT broker, the procedure is as Figure 6.6 shows. The first step is to create an MQTT application in the container overlay network. On VM1, use "create and bind" method to create a namespace

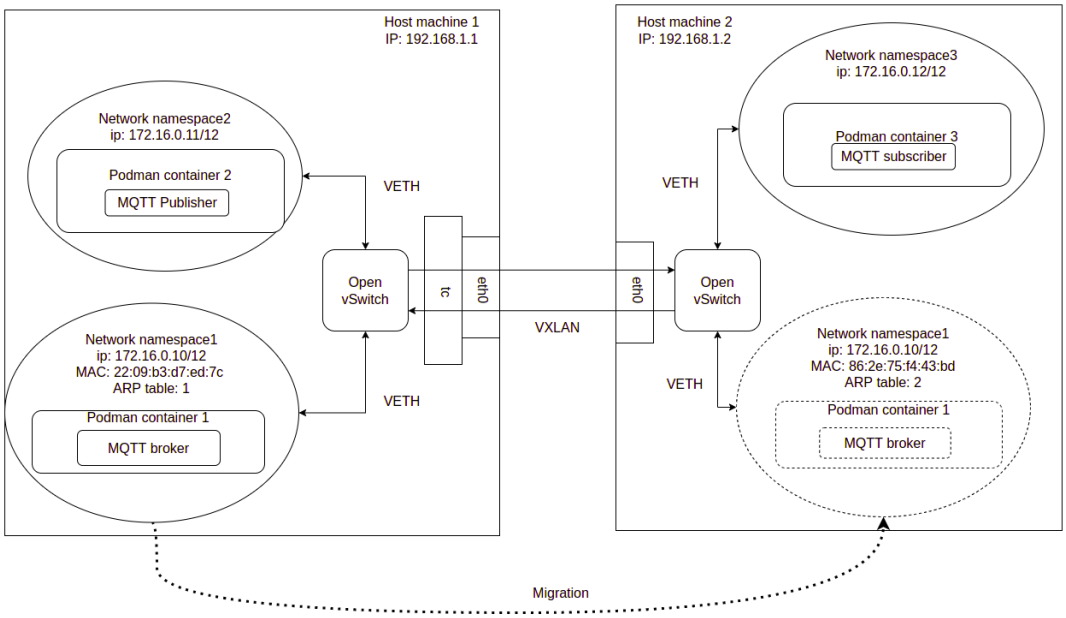


Figure 6.5: MQTT overlay network topology

named "mqtt" with overlay network configuration and bind the container of the MQTT broker to this namespace (step 1). To be noticed, it can not use the "direct manipulation" method to manipulate the container's auto-generated namespace. The main reason is that when the checkpointed container is being restored on the remote host, Podman can not automatically generate the modified namespace. Second, create the container for the MQTT publisher on VM1 and use the "direct manipulation" method to connect the container to the overlay network. Since the container of the MQTT publisher is not going to be migrated, the "direct manipulation" method is easier. Third, create the container for MQTT subscriber on VM2. Then, start MQTT broker, subscriber, and publisher.

After the creation of the experiment environment, the following procedure shows the way to migrate the container over an overlay network. The first is to slow down the speed of VM1's network interface to make the messages queue in the MQTT broker. When the message queue is almost full, checkpoint the container using Podman's command (step 7). Then, release the network speed limit on VM1 and copy the checkpoint image to VM2. Then create a namespace named "mqtt" on VM2 (step 9) and delete the one on VM1 (step 8). From the container's point of view, the "mqtt" namespace on VM2 is the same as the one on VM1, so the container can successfully restore on VM2. From the application's point of view, the containerized service can be reached from the same IP address before and after migration. However, from the network point of view, the container has a

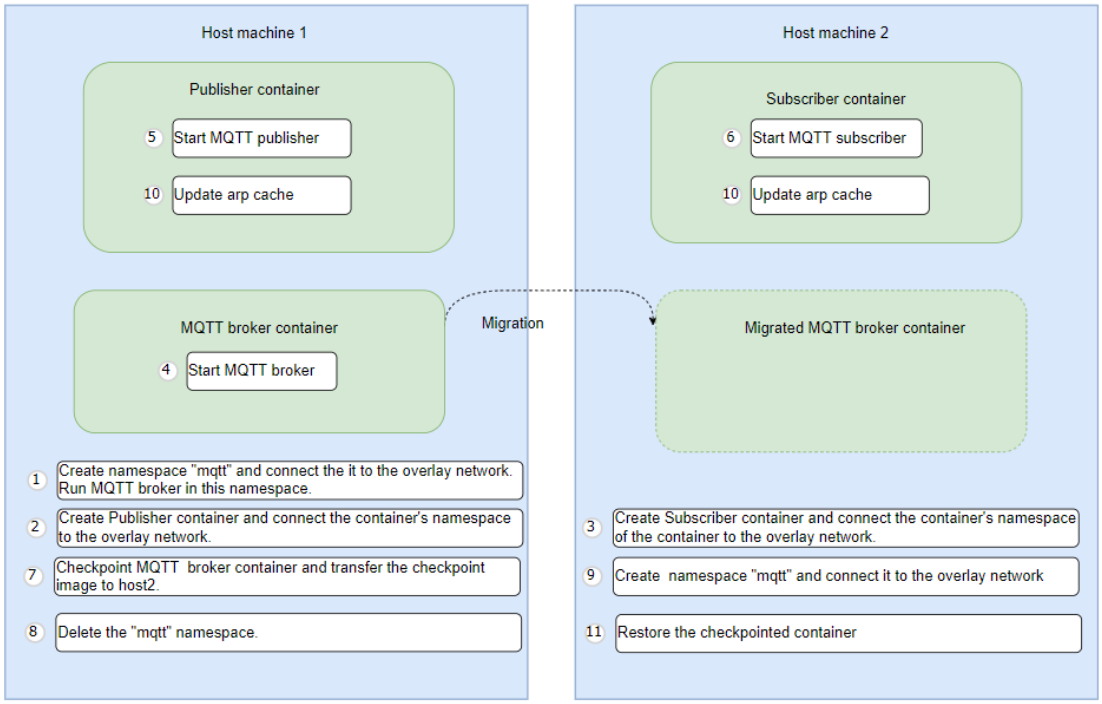


Figure 6.6: MQTT broker migration procedure

completely new MAC address and ARP table. If the container is restored to VM2 directly, it will take some time (around 10 seconds) for clients to notice the change of the broker's MAC address and update their own ARP table. Before the ARP table is updated, the communication between clients and broker will not restore. So after the new "mqtt" network namespace is created on VM2, we should first update the ARP table on the client's container (step 10) and then restore the checkpointed container to VM2 (step 11). After restoring, the communication will recover immediately, and the queued message inside the broker will be delivered to the subscriber.

6.6 Overlay network monitoring

Networking monitoring is an important method to validate the correctness of the network setting and to find out the bottleneck/bugs in the network design. However, when the containers are working with an overlay network, the network topology becomes complex. It is difficult to monitor applications' communication and trace a specific packet's flow. According to the proposed topology of the Podman overlay network, the corresponding monitoring method is also studied as Figure 6.7 shows.

There are three critical monitoring points in the container overlay topology:

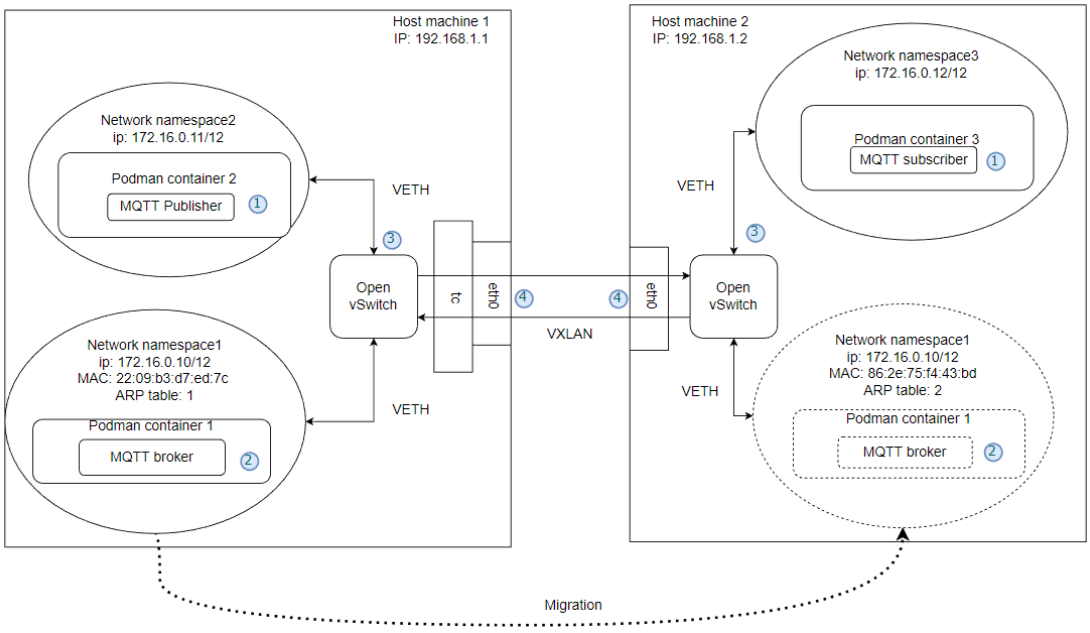


Figure 6.7: Podman container overlay network monitoring points

container network interface (points 1 and 2), OvS port (point 3), and host network interface (point 4). To capture the network traffic in the monitoring points, "tcpdump" is the simplest tool to use. The motoring methods are shown in Listing 6.3. Capturing the traffic in monitor point 1 and 2 could help to understand if the applications are communicating with the outside world correctly or not. However, in our experiment scenario, due to the containers having different configurations and behaviors (migrate or not), the monitoring methods are different. For the containers of MQTT clients (point 1), they are fixed on a specific host machine. So we can directly enter the container and use "tcpdump" to capture the traffic. For the container of the MQTT broker, which migrates from one host to another, we can run "tcpdump" directly inside the customized network interface of the container. Since the customized network interface will be created before creating or restoring the container, the complete behavior of the container migration will be recorded. Capturing the traffic in monitor point 3 could help to understand the working state of vETH connection between the container and OvS. Since the vETH is created by connecting container's network interface and a virtual network interface dedicated for OvS (in this case named pod-ovs), the traffic could be captured by dumping the dedicated virtual network interface. Capturing the traffic on monitor point 4 could help to understand the communication between OvSes on different host machine. Since OvS by default uses port 4789 on the host's network interface, the traffic could be captured by dumping the traffic flow port 4789 on the host's network

interface.

Listing 6.3: Podman container overlay network monitoring method

```
# At monitor point 1: Capture traffic inside fixed container
> sudo podman exec -it <container name> /bin/bash
(container)> tcpdump -w <output_file_name>.pcap

# At monitor point 2: Capture traffic from customized container network namespace
>  ip netns exec MQTT tcpdump -w <output_file_name>.pcap

# At monitor point 3: Capture traffic from dedicated virtual network interface for Open vSwitch
> tcpdump -i pod-ovs -w <output_file_name>.pcap

# At monitor point 4: Capture traffic from port 4789 on host network interface
> tcpdump -i eth0 port 4789 -w <output_file_name>.pcap
```

The monitoring results are shown in Figure 6.8 6.11. As Figure 6.8 shows, from the MQTT client's (publisher) point of view, the MQTT broker's IP address is 172.16.0.10, which is the overlay network IP address. MQTT client creates an IPv4 TCP connection with the MQTT broker on top of the overlay network. In this case, the communication is completely independent to the host machine's IP address. In Figure 6.9, it is worth noting that the captured result is completely the same as Figure 6.8. This monitoring point targets at checking the vETH configuration between the container and OvS is correct or not. In Figure 6.10, the monitoring result shows that when the overlay network packets pass the OvS, they are encapsulated into IPv4 UDP packets in the format of vxlan and route to the OvS on the corresponding machine. The monitored result in Figure 6.11 shows that after the MQTT broker migrated to the new host, the TCP connections with MQTT clients can be directly restored without any handshaking.

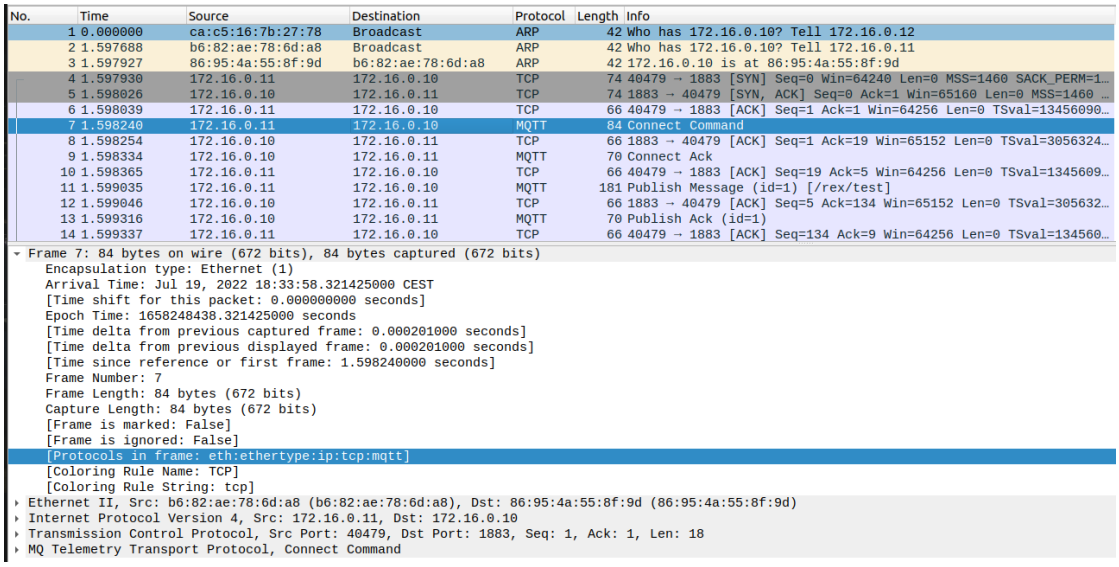


Figure 6.8: Overlay network monitoring inside container of MQTT publisher

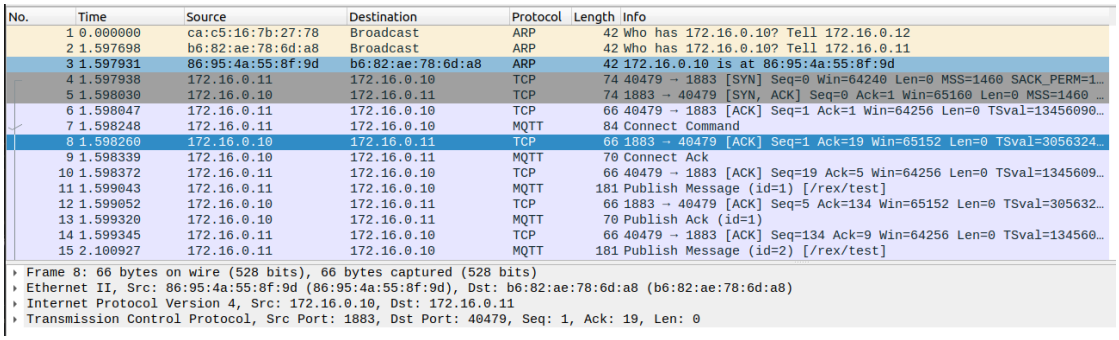


Figure 6.9: Overlay network monitoring on vETH connection

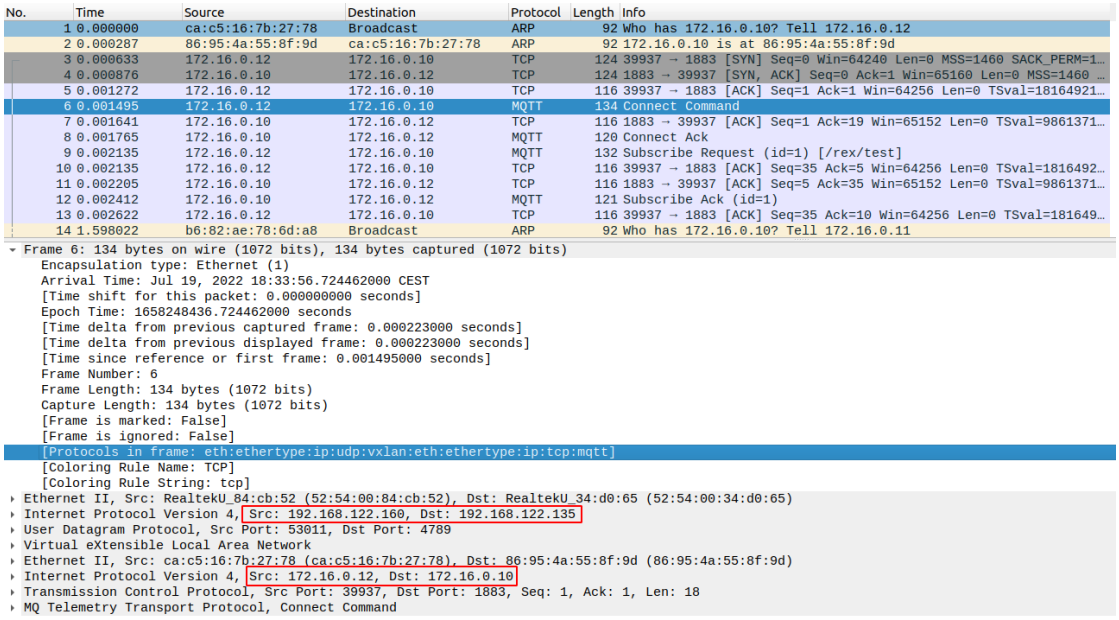


Figure 6.10: Overlay network monitoring between Open vSwitch

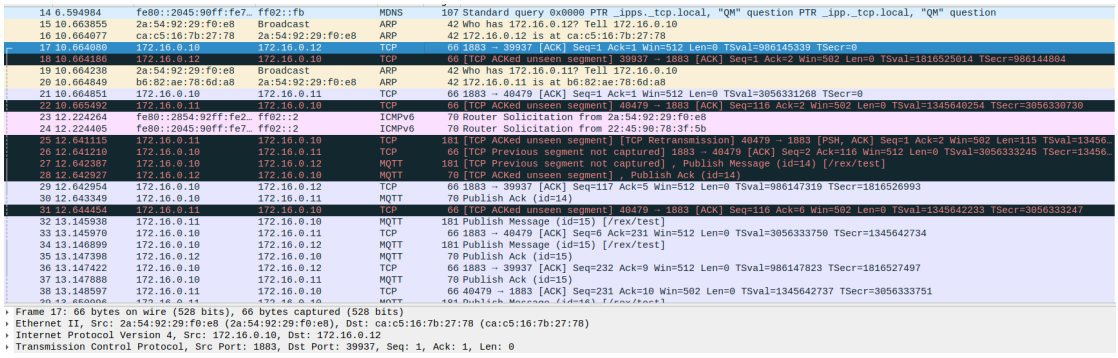


Figure 6.11: Overlay network monitoring in migrated container

Chapter 7

Management Stack: Migration Cost Model

The recent study on the container/VM migration management are mostly focused on minimizing the migration downtime ([6], and [16]). However, due to the limited resources on the edge servers, the consideration of resource usage during and after the migration is also very important. So in this chapter, a migration management model based on edge resources (CPU, memory, store and network) usage is proposed.

7.1 Cost Model Definition

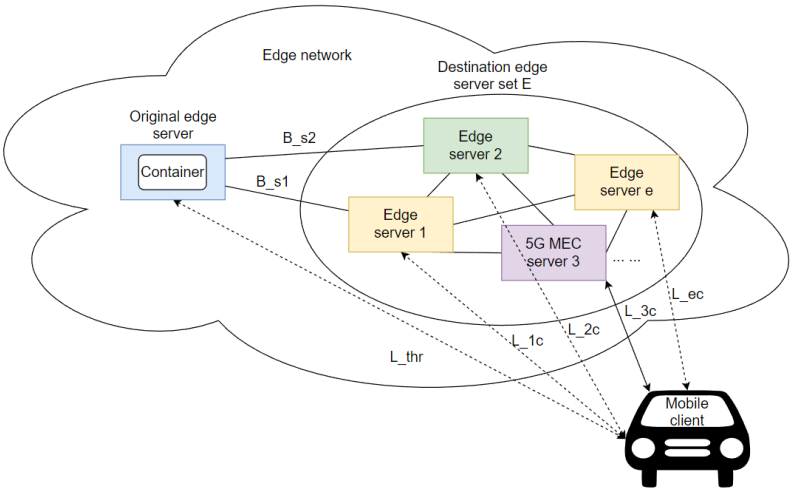


Figure 7.1: Optimization scenario

The optimization model targets at providing the lowest migration cost solution in the scenario, as Figure 7.1 shows. In the edge network, there is a set of destination edge servers E, which are geographically close to the mobile client. The communication latency between each destination edge servers L_{ec} are pre-measured. The migration process is triggered when the communication latency of the original connection L_O is larger than a specific threshold L_{thr}. Assume that the container migration uses pre-copy migration, with a single iteration of pre-copy. The following optimization mode targets at minimizing the migration cost from the resource usage point of view.

Decision variables:

- x_e: 0 or 1 (node e is selected at the migration destination or not).

Mobile service related parameters:

- R_{dp}: container dirty page rate, page/s.
- M_c: Container memory usage, bytes.
- Load_c: CPU load occupied by the container, (0,1].
- S_{nms}: The container's state size except for memory and network queue.
- R_c: Compression ratio of the dumped container state in file synchronization, (0,1].
- N_{active}: 0 or 1, exist active network connection between service and client during the migration.
- S_n: Queued message size during TCP connection migration, bytes. Worst case 10 Mibs.
- T_{maxDown}: Maximum downtime the service can tolerate.

Host related parameters:

- S_{page}: Memory page size, default 4096 bytes.
- Load_s: Current CPU load of source host, (0,1].
- Load_e: Current CPU load on edge server e, (0,1].
- NC_e: Total number of CPU cores on host e.
- M_e: Available memory on edge server e.

- B_{se}: Bandwidth between the source server and edge server e.
- s_{ps}: source server processing speed in IPS per CPU core.
- s_{pe}: destination server processing speed in IPS per CPU core.
- $C_{\text{CPU}e}$: Unit CPU usage cost on edge server e, euro/core.
- $O_{\text{CPU}e}$: CPU cost offset, guarantee the minimum CPU cost when load balancing is considered in the objective function, (0,1].
- $C_{\text{MEM}e}$: Unit memory cost on edge server e, euro/byte.
- $C_{\text{HDD}e}$: Unit storage cost on edge server e, euro/byte.
- $C_{\text{NET}e}$: Unit network cost from source to edge server e, euro/byte.

To be noted, the cost here does not consider the live cycle of the hardware devices.

Secondary parameters:

- T_{preSync}: The required time to synchronous the pre-dumped image to the destination.
- T_{dump}: The required time to completely dump a container based on the pre-dump image.
- T_{dumpSync}:The required time to synchronous the dumped image to the destination.
- T_{restore}: The required time for the destination server to restore the container.
- T_{down}: Overall migration down time.
- D_m: Dirty memory generated between pre-dump and dump.

Objective functions:

$$\begin{aligned} \min \quad \sum_{e} x_e \cdot \Big[& C_{\text{CPU}e} \cdot \frac{s_{\text{ps}}}{s_{\text{pe}}} \cdot \text{Load}_c \cdot (\text{Load}_e + O_{\text{CPU}e}) + \\ & + (C_{\text{HDD}e} + R_c \cdot C_{\text{NET}e}) \cdot \\ & \cdot (M_c + T_{\text{preSync}} \cdot R_{\text{dp}} \cdot S_{\text{page}} + N_{\text{active}} \cdot S_n + S_{\text{mns}}) + \\ & + C_{\text{MEM}e} \cdot M_c \Big] \end{aligned} \tag{7.1}$$

In the container migration scenario, all the variables change over time. In order to guarantee that the optimal destination provided by the model is still the optimal

one when the actual migration is performed, the solving to the optimization model should be as fast as possible. In order to reach this requirement, the objective function of the optimization model is provided as Equation.7.1 shows. In this objective function, only one decision variable is involved, which is the selection of the migration destination. In the object function, the migration cost in the aspect of CPU, memory, storage, and network usage are all considered. With such a simple mode, a simple iteration and comparison algorithm can be used to find the solution in a short time.

Constrains:

$$T_{\text{down}} \leq T_{\text{maxDown}} \tag{7.2}$$

$$\sum x_e \cdot \left(\text{Load}_e + \frac{s_{\text{ps}}}{s_{\text{pe}}} \cdot \text{Load}_c \right) \leq \sum x_e \cdot \text{NC}_e \tag{7.3}$$

$$M_c \leq \sum x_e \cdot M_e \tag{7.4}$$

$$\sum x_e = 1 \tag{7.5}$$

$$\frac{M_c}{T_{\text{preSync}}} > R_{\text{dp}} \cdot S_{\text{page}} \tag{7.6}$$

$$T_{\text{preSync}} = \frac{M_c}{R_c \cdot (1 - \text{Load}_s) \cdot s_{\text{ps}}} + \sum_e x_e \cdot \frac{M_c \cdot R_c}{B_{se}} \tag{7.7}$$

$$D_m = T_{\text{preSync}} \cdot R_{\text{dp}} \cdot S_{\text{page}} \tag{7.8}$$

$$T_{\text{dump}} = \frac{S_{\text{nms}} + N_{\text{active}} \cdot S_n + D_m}{(1 - \text{Load}_s) \cdot s_{\text{ps}}} \tag{7.9}$$

$$\begin{aligned} T_{\text{dumpSync}} = & \frac{S_{\text{nms}} + N_{\text{active}} \cdot S_n + D_m}{R_c \cdot (1 - \text{Load}_s) \cdot s_{\text{ps}}} \\ & + \sum_e x_e \cdot \frac{(S_{\text{nms}} + N_{\text{active}} \cdot S_n + D_m) \cdot R_c}{B_{se}} \end{aligned} \tag{7.10}$$

$$T_{\text{restore}} = \frac{S_{\text{nms}} + N_{\text{active}} \cdot S_n + D_m + M_c}{\sum_e x_e \cdot (1 - \text{Load}_e) \cdot s_{\text{pe}} \cdot R_c} \tag{7.11}$$

$$T_{\text{down}} = T_{\text{dump}} + T_{\text{dumpSync}} + T_{\text{restore}} \tag{7.12}$$

$$x_e \in \{0,1\} \ \forall \ e \in E \tag{7.13}$$

$$T_{\text{preSync}}, T_{\text{dump}}, T_{\text{dumpSync}}, T_{\text{restore}}, T_{\text{down}} \in R^{+} \tag{7.14}$$

The constraints bound the selection of the destination host from the following main aspects. Constrain.7.2 limits that the migration downtime should not exceed the maximum downtime the application can tolerate. Constrain.7.3 limits that the CPU load on the migration destination should not exceed its own capacity after the container migration. Constrain.7.4 limits that the migration destination should have enough available memory for the migrated container to use. Constrain.7.5 limits that there could only have one migration destination. Constrain.7.6 limits that the speed of the pre-copy procedure should be faster than the generation of dirty memory pages in order to guarantee the pre-copy migration is efficient.

7.2 Migration Cost Estimation

Since the migration cost model introduced in Sec.7.1 is defined with real-world resources cost, the general cost of the migration can be easily estimated. Assume that the hardware used on the edge servers have the same setting, listed as Table.7.1 shows.

	Specification	**Unit cost**
CPU	Intel i7 12th gen, 3.6 GHz,12 cores	470 €
Memory	16 GB DDR4	70 €
Storage	512 GB SSD	50 €
Network	2.5 Gbps	30 €/month

Table 7.1: Assumed edge server setting

In this case, the edge server-related parameter can be estimated, as Table.7.2 shows. Assume that the CPU load on each edge server is 0.3 and the available memory size is 13 GB.

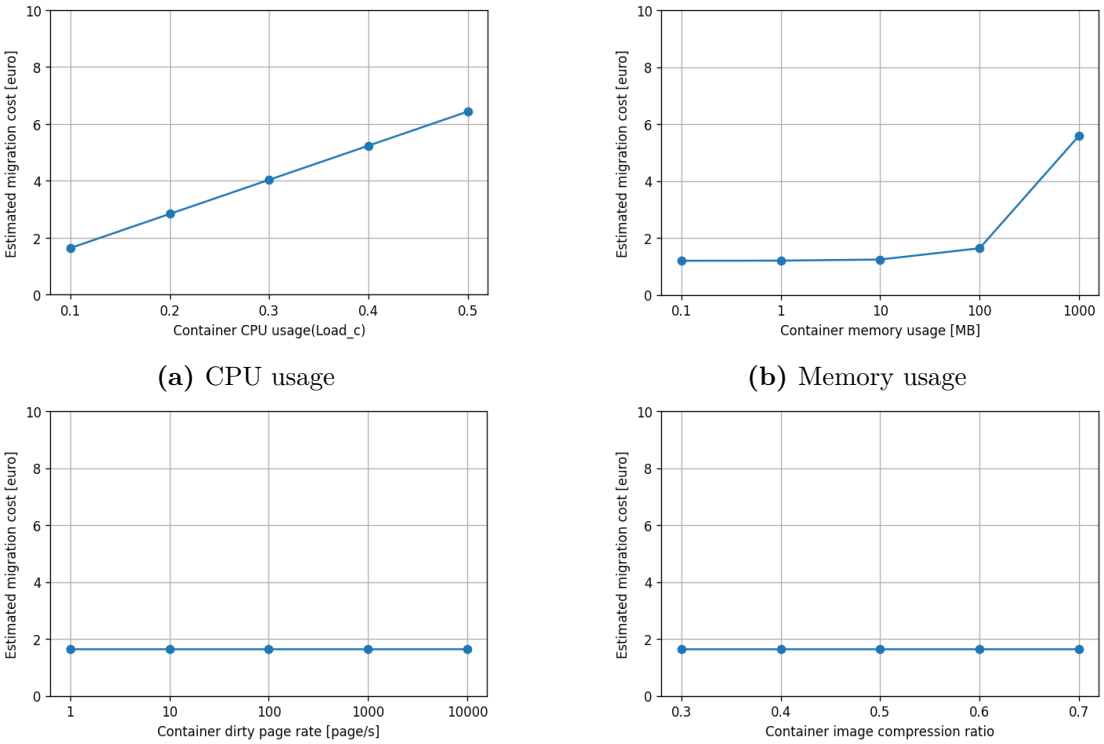
$C_{\text{CPU}e}$	40 €/core	Load_e and Load_s	0.2
$C_{\text{MEM}e}$	$4.3 \cdot 10^{-9}$ €/byte	S_{page}	4096 bytes
$C_{\text{HDD}e}$	$9.39 \cdot 10^{-11}$ €/byte	B_{se}	$3.2 \cdot 10^{8}$ byte/s
$C_{\text{NET}e}$	$2.16 \cdot 10^{-12}$ €/byte	NC_e	12
S_{page}	4096 bytes	M_e	$1.3 \cdot 10^{10}$ bytes
s_{ps} and s_{pe}	$1.13 \cdot 10^{11}$	$O_{\text{CPU}e}$	0.1

Table 7.2: Edge server related parameter estimation

Assume that the container to be migrated has the properties as Table.7.3. Since we assume the container to be migrated does not has an active network connection during the migration, S_n does not work in this case. Since we only want to estimate the migration cost, T_{maxDown} is not considered now. As a result, the estimated migration cost of this container is 1.64 euros. To be noticed, the resources cost here do not consider their live cycle. In order to obtain the actual cost, the cost obtained from the model should be scaled down according to the hardware's life cycle.

$\mathbf{Load}_c$	0.1	$R_{\mathbf{dp}}$	10 pages/s
R_c	0.7	$N_{\mathbf{active}}$	0
M_c	10^8 bytes	$S_{\mathbf{nms}}$	10^7bytes

Table 7.3: Assumed properties for the container to be migrated



(a) CPU usage

(b) Memory usage

(c) Memory dirty page rate

(d) Checkpoint image compression ratio

Figure 7.2: Estimated migration cost with variations on different container properties

What's more, this cost model can help estimate the migration cost change when

the container properties have changed. As Figures 7.2 shows, when variate different parameters in Table 7.3, the change trend on total migration cost can be obtained. When the CPU load of the container on the host and memory usage of the container increase, the total migration cost increases significantly. When the container's dirty memory page rate and the checkpoint image compression ratio increase, the total migration cost does not change much.

Chapter 8

Management Stack: Migration policies

In order to reach maximum efficiency during container migration, it is important to choose the proper techniques in each migration step according to the container's properties. In this chapter, the policies for the selection of migration techniques and migration strategies are discussed. Since the study of this thesis focuses on the migration of Podman containers, the policies and strategies might not be suitable for all container implementations.

8.1 Policies related to containers properties

There are three main properties of the container that affects the migration result: file system state, memory usage and network connection.

File system state. The file system state directly decides whether a Podman container can be successfully migrated or not. As Table 8.1 shows, a container's file system could be modified or not with respect to the base image. When the container's file system is complete the same as the based image, the container can be directly migrated. But when the container's file system is modified (e.g., add, delete or modify), the container can not be directly migrated. Otherwise, the migration reports an error and is interrupted in the restore phase. In order to solve this problem, the container with a modified file system can be rebuilt into a container image and synchronized to the destination host. In this case, the container's file system is exactly the same as the new image, and the restoring process can be successfully done.

File system state	Migratability
Unchanged	Migratable
Modified	Not direct migratable
Modified but rebuilt in image	Migratable

Table 8.1: Container mobility according to file system state

Memory usage. In container migration, the container's memory state is one of the key attributes affecting migration performance. For different memory characteristic, different methods should be adopted as Table 8.2 shows. For different memory characteristics of the container, a corresponding example and its migration performance are also introduced. To be noticed, T_{ckp} means the checkpoint time in cold migration or (last) second checkpoint time in (iterative) pre-copy migration. T_{sync} means the estimated required time to synchronize the checkpoint image or (last) second checkpoint image in (iterative) pre-copy migration. T_{res} means the restore time of the container. T_{dowm} means the migration downtime, which is equal to the sum of T_{ckp}, T_{sync} and T_{res}. The result is obtained from tests done on VMs as mentioned in Section.5.5, with a limitation of network bandwidth in 10 MB/s.

When the container uses only a small amount of memory (e.g., 1 MB), the container's checkpoint image is relatively small, which can be quickly transmitted to the destination. In this case, the cold migration method is better since it is easier to implement and more efficient (though it scarifies some migration downtime).

However, when the target container uses a median amount of memory (e.g., 10 MB), the cold migration might not always be the best choice. When the dirty memory page rate of the container is not too high (e.g., 100 page/s), the pre-copy migration method is the better choice. With the pre-copy migration method, the size of the second checkpoint image (in the stop-and-copy phase) can be significantly reduced, which requires less synchronization time and leads to a shorter migration downtime. However, when the dirty memory page rate of the container is high, the size of the second checkpoint image with the pre-copy method will be close to the overall memory usage of the container. In this case, the pre-copy method can not reduce migration downtime and introduces a lot of migration overhead. So when the dirty page rate of the container is high, it is always better to use the cold migration method.

When the container's memory usage is large, the performance in the reduction of the second checkpoint image's size of the per-copy migration method becomes worse. The reason for this phenomenon is that it requires more time to synchronize the first checkpoint image, and the containerized application generates more dirty memory pages during this time. The better solution to migrate such kind of container is to use iterative pre-copy migration method (will be introduced in Section 8.2).

Memory usage	**DPR**	**Migration method**	**Example**	$T_{\mathbf{ckp}}$ [s]	$T_{\mathbf{sync}}$[s]	$T_{\mathbf{res}}$[s]	$T_{\mathbf{down}}$[s]
small	low	cold	1 MB 10 page/s	0.5	0.1	1.3	1.9
small	middle	cold	1 MB 50 page/s	0.5	0.1	1.3	1.9
small	high	cold	1 MB 100 page/s	0.5	0.1	1.3	1.9
median	low	pre-copy	10 MB 10 page/s	1.1	0.01	1.1	2.21
median	middle	pre-copy	10 MB 100 page/s	1.1	0.09	1.1	2.29
median	high	cold	10 MB 1000 page/s	3.6	1	1.5	6.1
large	low	iterative pre-copy	100 MB 10 page/s	3.6	0.05	2.4	6.05
large	middle	iterative pre-copy	100 MB 100 page/s	3.6	0.45	2.4	6.45
large	high	cold	100 MB 10000 page/s	6.5	10	8	24.5

Table 8.2: Migration method selection according to container memory usage

Network connections	**Network management**
none	none
short period TCP connection	network proxy
long period TCP connection	overlay network

Table 8.3: Migration method according to container's network connection

Network connection. In order to preserve the network connection of the applications inside the container, different kinds of network management is required, as Table 8.3 shows (this study only discusses TCP connections). When the applications are without any network connection, the container can be migrated directly without any management. When the application inside the container has short-period TCP connections (the duration of connection is short), the better strategy is to migrate the container when no connection is alive and manage the IP address problem after migration with simple techniques like network proxy. For the containers with a long-period TCP connection (the duration of connection is long, no matter the amount of data transmitted), the overlay network topology is

the most operable way to realize stateful network migration.

8.2 Migration strategy

In this section, the overall migration strategy with the pre-copy migration method is discussed. The reason why the strategy focuses on pre-copy migration is that it has the potential to achieve minimum migration downtime. According to [6], the iterative pre-copy procedure could help to reduce the number of dirty memory pages to be transmitted in the stop-and-copy phase. Together with the migration cost model, the migration strategy is summarized as Figure 8.1 shows.

On the source server where containerized mobile service is running, the communication latency with the client is continuously monitored. When the communication latency with the client reaches a threshold, the migration procedure is triggered (step 1). Then the source server obtains the communication latency between each available edge server and the client and their resource usage (step 2). The source server could apply the migration cost model with the latency and resource usage information to determine the best edge server to migrate the container (step 3).

Before the migration starts, the source server should send a migration request to the destination edge server (step 4), which should contain the information of the container to be migrated. With the information, the destination server can prepare the environment for the migration: pull the container image from a remote repository (step 5) and pre-allocate the required resources (step 6). When the preparation procedure is finished, the destination server sends a response to the source server (step 7), telling it whether the migration request is accepted or not. If the migration request is accepted, the source server starts the iterative pre-copy procedure (steps 8 and 9). When the number of dirty memory pages in the pre-checkpoint image reduces to a specific threshold, the source server performs the stop-and-copy procedure (step 10) and synchronizes the final checkpoint image to the destination host (step 11). When the synchronization is finished, the source server notifies the destination server to restore the container from the checkpoint images (steps 12 and 13). The migration downtime is calculated as the duration of steps 10 to 13.8-0

In theory, the iterative pre-copy procedure could significantly reduce the number of dirty memory pages in the last checkpoint image (stop-and-copy phase). However, standard container engines (such as Podman, and Docker) do not initially provide such an iterative pre-copy function. In order to implement the iterative pre-copy migration with the Podman container, one of the solutions is to use the delta-transfer property of rsync (introduced in Section 3.3.1) as Listing 8.1 shows. In each iteration, Podman pre-checkpoints the target container to the file with the same name and synchronizes to the destination with rsync. In the first iteration, due to

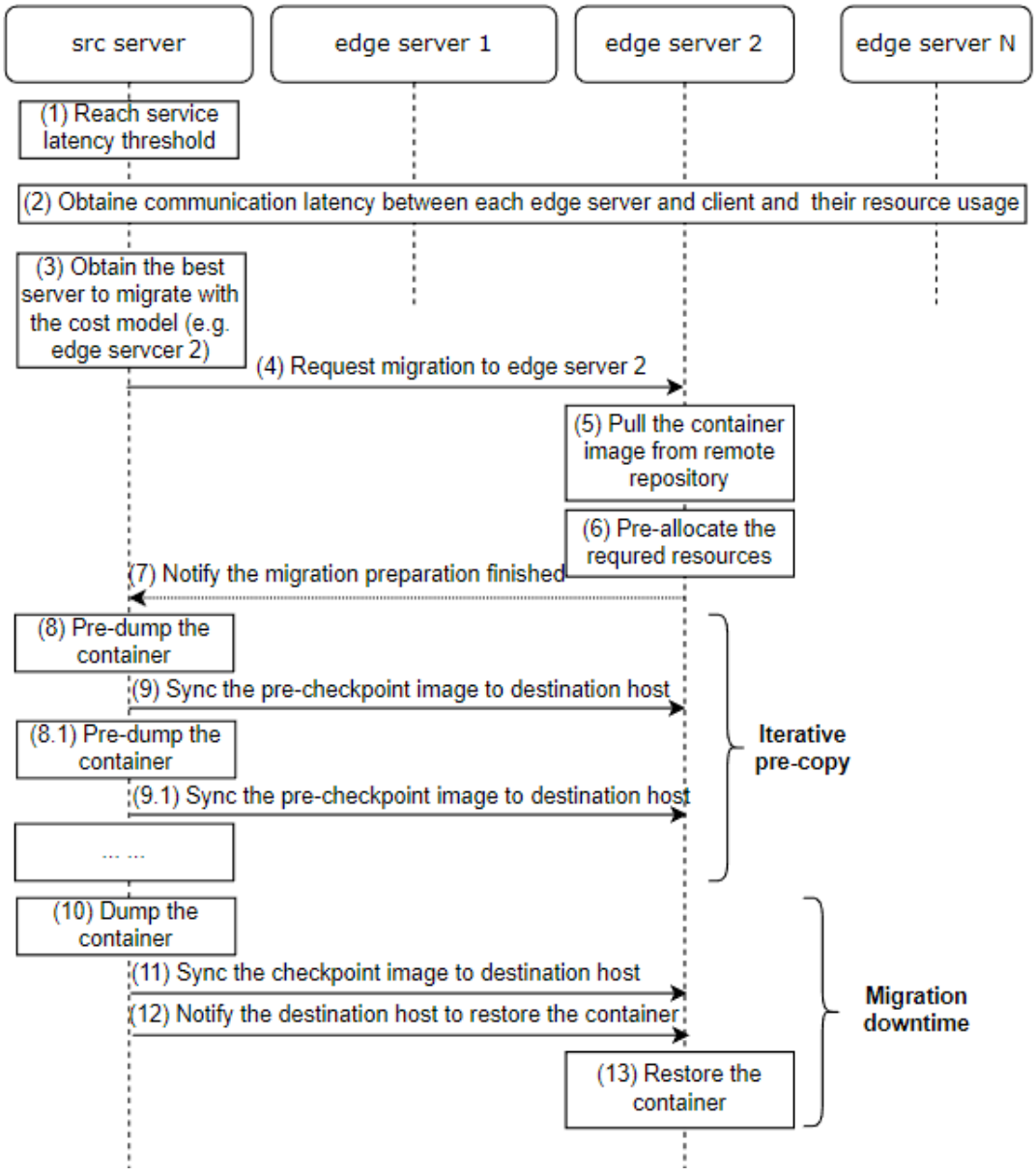


Figure 8.1: Pre-copy migration strategy

the destination having no checkpoint image yet, the amount of data sent by rsync will be the same as the original checkpoint image size. When the checkpoint image is large, it might require a long time. But in the following iteration, the transmitted data size will be the dirty memory generated during the previous synchronization. When the dirty page rate is not very large, the transmitted data size should reduce by iteration. When the transmitted data size is less than a specific threshold, the

iterative pre-copy procedure could be terminated, and perform the stop-and-copy procedure.

Listing 8.1: Podman iterative pre-checkpoint algorithm

```
FOR i in MaxIterations:
    Pre-checkpoint target container and store to file PRE_DUMP
    Synchronize PRE_DUMP to the destination host with rsync
    IF transmitted data size by rsync is smaller than a threshold
        BREAK
Dump the target container and store to file DUMP (stop-and-copy)
Synchronize DUMP to destination host
```

In order to validate the iterative pre-copy algorithm, a test is done with the VMs mentioned in Section 5.5 with the following additional settings:

- Source host network uplink speed: 10 MB/s
- Rsync delta-transfer block size: 1024 bytes
- Container memory allocation: 100 MB

The test result is as Figure 8.2 shows. In all test cases, the transmitted size of pre-checkpoint images has a significant reduction when the iteration number is equal to or greater than 2. However, when the container's dirty page rate increases, the reduction in transmitted pre-checkpoint image size becomes less.

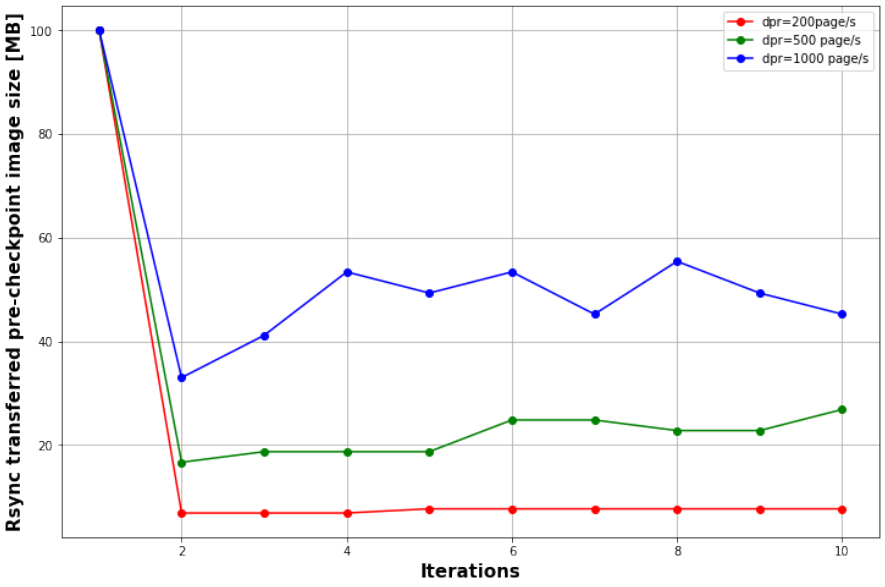


Figure 8.2: Iterative pre-checkpoint image size

Chapter 9

Conclusions

Nowadays, with the development of mobile communication technologies, more and more mobile application concepts are introduced to the view of the public, including but not limited to connected vehicles, UAVs, and portable health monitoring systems. In order to avoid the high communication cost with cloud solutions and the high deployment cost with local solutions, the corresponding mobile services are normally deployed in edge networks. However, such kinds of application scenarios require the system to have a very high quality of service. Even though with the edge solution, the communication quality requirements might not always be fulfilled, especially the requirement of communication latency. In this case, enabling service mobility in edge computing architecture becomes critical. Since mobile services/applications are normally containerized in edge architecture, the study of service mobility could be generalized into the study of container mobility.

In the work of this thesis, the complete technical stack of container migration has been studied. In the realization stack, the background technique to be studied is file synchronization. The study targets at selecting the most efficient tool for container checkpoint image synchronization. Two different tools are detailed tested and analyzed on the performance of file deduplication rate, CPU usage, memory usage, storage usage, and network usage. In conclusion, "rsync" is selected to be the optimal tool for container migration synchronization.

In the realization stack, the study focuses on the design of the stateful application simulator, the containerization of customized applications, and the validation of basic container migration. To be mentioned, a stateful application simulator with a controllable dirty memory page rate is designed and validated during the study. It provides a more accurate simulation of the application's memory behavior during migration. In order to create a minimum container image of the stateful application simulator, different base images and compiling methods are studied. Even though compiling the C-language-based application with the static shared libraries linking will increase the size of the application executable, the statically linked executable

with the busybox base image can create the minimum container image. With the container image, stateful containers are created. The basic migration tests validate the statefulness of different migration methods. They prove that the migration methods based on the tool "CRIU" are stateful.

In the advanced realization stack, the network management during container migration is studied. In recent studies, not many works addressed the operable detail of network management during container migration. So an overlay network solution is proposed and tested in the study of this thesis. This solution solves the IP address problem during container migration and guarantees the recovery of the active network connection. This study also discusses the methods to monitor and trace the network traffic on the overlay network.

In the management stack, a migration cost model and some migration policies are proposed. The cost model targets at selecting an optimal migration destination among the edge network by considering the resource usage during the migration. To be noticed, the model could estimate the actual migration cost in euro by linking the cost of the resources to the real world price, helping the operator better design the management strategy. The migration policies discuss the different selections of migration techniques when migrating containers with different properties. The study of the iterative pre-copy migration strategy provides a possible implementation with the current Podman container and rsync. A simple test is also done to validate and show the strength of this strategy.

9.1 Future works

In the study of this thesis, the complete technical stack of container migration is addressed. However, the study focuses more on validating the migration theory but not the actual deployment. It is reasonable to believe that this work could be extended for future research.

One limitation in this study is that the container migration procedures are done with scripts, which is efficient and easy to modify. But in actual deployment, a more complete system is requires in order to guarantee the reliability and flexibility of the migration process. Therefore, future studies are suggested to investigate the design of intelligent migration agents which are capable of communicating with the orchestrator and automatically adjusting the migration settings.

In addition, the study in this thesis focuses more on the low-layer migration implementation and performance analysis. Only a small part of container migration management is discussed. Future work could concentrate on the study of high-layer orchestration or the discussion of the extreme conditions on container migration, pushing the adoption of this technique in real applications.

Bibliography

[1] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. «Edge Computing: Vision and Challenges». In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646. DOI: 10.1109/JIOT.2016.2579198 (cit. on p. 1).

[2] *Mobile-edge computing—Introductory technical white pape.* https://portal.etsi.org/portals/0/tbpages/mec/docs/mobile-edge_computing_-_introductory_technical_white_paper_v1%2018-09-14.pdf. Accessed: 2022-09-05 (cit. on p. 2).

[3] Yuyi Mao, Changsheng You, Jun Zhang, Kaibin Huang, and Khaled B. Letaief. «A Survey on Mobile Edge Computing: The Communication Perspective». In: *IEEE Communications Surveys Tutorials* 19.4 (2017), pp. 2322–2358. DOI: 10.1109/COMST.2017.2745201 (cit. on p. 2).

[4] P. Getzi Jeba Leelipushpam and J. Sharmila. «Live VM migration techniques in cloud environment — A survey». In: *2013 IEEE Conference on Information Communication Technologies*. 2013, pp. 408–413. DOI: 10.1109/CICT.2013.6558130 (cit. on p. 6).

[5] Shunmugapriya Ramanathan, Koteswararao Kondepu, Miguel Razo, Marco Tacca, Luca Valcarenghi, and Andrea Fumagalli. «Live Migration of Virtual Machine and Container Based Mobile Core Network Components: A Comprehensive Study». In: *IEEE Access* 9 (2021), pp. 105082–105100. DOI: 10.1109/ACCESS.2021.3099370 (cit. on pp. 10, 13).

[6] Tianzhang He, Adel Toosi, and Rajkumar Buyya. «SLA-Aware Multiple Migration Planning and Scheduling in SDN-NFV-enabled Clouds». In: (Jan. 2021) (cit. on pp. 11, 13, 69, 79).

[7] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. «Post-Copy Live Migration of Virtual Machines». In: *SIGOPS Oper. Syst. Rev.* 43.3 (July 2009), pp. 14–26. ISSN: 0163-5980. DOI: 10.1145/1618525.1618528. URL: https://doi.org/10.1145/1618525.1618528 (cit. on p. 12).

[8] Tianzhang He and Rajkumar Buyya. «A Taxonomy of Live Migration Management in Cloud Computing». In: (Dec. 2021) (cit. on p. 13).

[9] Carlo Puliafito, Enzo Mingozzi, Carlo Vallati, Francesco Longo, and Giovanni Merlino. «Virtualization and Migration at the Network Edge: An Overview». In: *2018 IEEE International Conference on Smart Computing (SMARTCOMP)*. 2018, pp. 368–374. DOI: 10.1109/SMARTCOMP.2018.00031 (cit. on p. 13).

[10] Yuqing Qiu, Chung Horng Lung, Samuel Ajila, and Pradeep Srivastava. «LXC Container Migration in Cloudlets under Multipath TCP». In: vol. 2. IEEE Computer Society, Sept. 2017, pp. 31–36. ISBN: 9781538603673. DOI: 10.1109/COMPSAC.2017.163 (cit. on p. 13).

[11] Luca Conforti, Antonio Virdis, Carlo Puliafito, and Enzo Mingozzi. «Extending the QUIC Protocol to Support Live Container Migration at the Edge». In: *2021 IEEE 22nd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*. 2021, pp. 61–70. DOI: 10.1109/WoWMoM51794.2021.00019 (cit. on p. 13).

[12] Solomon Kassahun, Atinkut Demessie, and Dragos Ilie. «A PMIPv6 approach to maintain network connectivity during VM live migration over the internet». In: *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*. 2014, pp. 64–69. DOI: 10.1109/CloudNet.2014.6968970 (cit. on p. 13).

[13] Patrick Raad, Stefano Secci, Dung Chi Phung, Antonio Cianfrani, Pascal Gallard, and Guy Pujolle. «Achieving Sub-Second Downtimes in Large-Scale Virtual Machine Migrations with LISP». In: *IEEE Transactions on Network and Service Management* 11.2 (2014), pp. 133–143. DOI: 10.1109/TNSM.2014.012114.130517 (cit. on p. 13).

[14] Carlo Puliafito, Antonio Virdis, and Enzo Mingozzi. «The Impact of Container Migration on Fog Services as Perceived by Mobile Things». In: *2020 IEEE International Conference on Smart Computing (SMARTCOMP)*. 2020, pp. 9–16. DOI: 10.1109/SMARTCOMP50058.2020.00022 (cit. on pp. 13, 55).

[15] *Container runtime: runC*. https://github.com/opencontainers/runc. Accessed: 2022-10-03 (cit. on p. 13).

[16] Gang Sun, Dan Liao, Vishal Anand, Dongcheng Zhao, and Hongfang Yu. «A new technique for efficient live migration of multiple virtual machines». In: *Future Generation Computer Systems* 55 (Feb. 2016), pp. 74–86. ISSN: 0167739X. DOI: 10.1016/j.future.2015.09.005 (cit. on pp. 13, 69).

[17] Sumit Maheshwari, Shalini Choudhury, Ivan Seskar, and Dipankar Raychaudhuri. «Traffic-Aware Dynamic Container Migration for Real-Time Support in Mobile Edge Clouds». In: *2018 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS)*. 2018, pp. 1–6. DOI: 10.1109/ANTS.2018.8710163 (cit. on p. 13).

[18] Zitong Ma, Sujie Shao, Shaoyong Guo, Zhili Wang, Feng Qi, and Ao Xiong. «Container Migration Mechanism for Load Balancing in Edge Network Under Power Internet of Things». In: *IEEE Access* 8 (2020), pp. 118405–118416. DOI: 10.1109/ACCESS.2020.3004615 (cit. on p. 13).

[19] *Docker container*. https://www.docker.com/. Accessed: 2022-10-03 (cit. on p. 15).

[20] *LXD container*. https://linuxcontainers.org/. Accessed: 2022-10-03 (cit. on p. 15).

[21] *Podman container*. https://podman.io/. Accessed: 2022-10-03 (cit. on p. 15).

[22] *Checkpoint/Restore In Userspace*. https://criu.org/Main_Page. Accessed: 2022-10-03 (cit. on p. 15).

[23] *Rsync documentation*. https://linux.die.net/man/1/rsync. Accessed: 2022-10-03 (cit. on p. 17).

[24] *Borg object graph*. https://borgbackup.readthedocs.io/en/stable/internals/data-structures.html. Accessed: 2022-09-05 (cit. on p. 18).

[25] *Open vSwitch*. https://www.openvswitch.org/. Accessed: 2022-10-03 (cit. on p. 19).

[26] Brendan Gregg. *Linux Systems Performance*. https://www.brendangregg.com/Slides/LISA2019_Linux_Systems_Performance.pdf. Accessed: 2022-10-03 (cit. on p. 21).

[27] *Uftrace tool*. https://github.com/namhyung/uftrace. Accessed: 2022-10-03 (cit. on p. 39).

[28] Jonathan Corbet. «TCP connection repair». In: (2012). URL: https://lwn.net/Articles/495304/ (cit. on p. 57).