國 立 清 華 大 學

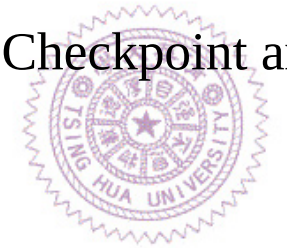
資訊工程研究所

碩士論文

使用中斷點和回復機制

實現容器叢集中的遷移和高可用性

Container Migration and High Availability in Docker Swarm using Checkpoint and Restoration



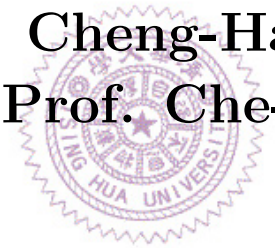
研究生：103062622 黃晟豪（Cheng-Hao Huang）

指導教授：李哲榮 教授（Prof. Che-Rung Lee）

中華民國一零五年七月

Container Migration and High Availability in Docker Swarm using Checkpoint and Restoration

Student: Cheng-Hao Huang
Advisor: Prof. Che-Rung Lee



Department of Computer Science
National Tsing Hua University
Hsinchu, Taiwan, 30013, R.O.C.

July 2016

中文摘要

容器技術自 2013 年 Docker 發表後在全世界迅速竄紅,Container 解決了維護人員在伺服器進行大量部屬時的痛點，使得環境部屬只需要建立完容器映像檔後就可以進行大量部屬，並對每一個容器環境進行隔離。

在這篇論文中，我們提出了在 Docker swarm 叢集中，將容器在多個節點中相互搬移。另外，可以針對特定的容器定期設定 checkpoint 儲存至雲端儲存空間，若叢集中的節點遇到不正常的離線時，可以及時回復最近的容器狀態到健康的節點上。

Abstract

More and more software developers and information-technology professionals embrace the container technology, because it packs the required libraries and settings of software into a single image so that the deployment can be easily done anywhere without the issues of compatibility. On top of single containers, orchestration tools are essential to automate the deployment and operation of complex systems that involve multiple containers on cluster of machines. In this thesis, we present the idea of using checkpoint-and-restore technique to enhance the functionality of Docker Swarm, a state-of-art orchestration tool for Docker containers. Two major functions are focused: high availability and migration. We used multi-version checkpoints to enhance the availability of containers and investigated the optimal storage and performance for checkpoint-and-restore. For migration, we leverage the shared storage and Docker Swarm's scheduler to make migration easier. We also studied the possibility of live migration for our implementation. Experiments show that pre-dump and track-memory will save about 10% $\sim$ 20% container checkpoint frozen time and at least 200% storage space.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Cloud computing, which utilizes a pool of computing resources to provision elastic, on-demand, and measured services via network, has reshaped the way that people use computers [4]. In the era of cloud computing, the centralized management of computational resources, such as data center, is not only more efficient, in terms of power and performance, to provide various services, but also supports higher security and availability to the provisioned services [12, 15]. One of the key technology to achieve those goals is virtualization [14, 23, 30, 26, 9]. The abilities of virtualized machines to perform emulation, isolation, consolidation, and migration simply make resource management and high availability easier.

However, the service model of cloud computing is not perfect for all kinds of applications. The recent advance of Internet Of Things (IOT) has brought new challenges that confront the centralized architecture of cloud computing. A conservative estimation has shown that by the end of the decade, 22 billion connected devices will constantly generate semi-structured or unstructured data in real-time. The interconnection of these devices will create immense data to strain the bandwidth and to lower the network availability. With such amount of network traffic, the centralized architecture will face huge amount of network failures or delays. Moreover, many IOT services possess time sensitive data. The network latency from the devices to cloud platforms can simply annihilate the possibility to fulfill the real-time requirement of IOT services.

One of the promising solutions is fog computing, proposed in [8, 28], which distributes massive number of decentralized edge nodes near IOT devices. An edge node

need not be as powerful as the cloud platform, but should be able to carry out a substantial amount of storage, communication, control, configuration, measurement and management of IOT devices. Because of the proximity to end points, edge nodes can reduce the latency between devices. In addition, the dense geographical distribution can help the quick recovery of failures with redundancy.

For an edge node in fog computing, virtual machines are no longer a good solution to deploy services, because of VMs' long booting time and large performance overhead. An emerging technology, called container, appears just-in-time to fill the vacancy. A container [27] can be viewed as an operating system level virtual machine, which runs as an isolated process in userspace and shares the same kernel with other containers. The kernel namespace [7] of host OS is used to isolate each container's environment, such as PID, IPC, network, and mount. The cgroups in host OS can further control the hardware usage of each container, such as CPU, memory, network, and disk I/O. Since containers need not to emulate the hardware, their performance is much better than that of Virtual Machine [29, 13, 18]. Figure 1.1 shows the architectural comparison between a virtual machine and a container.

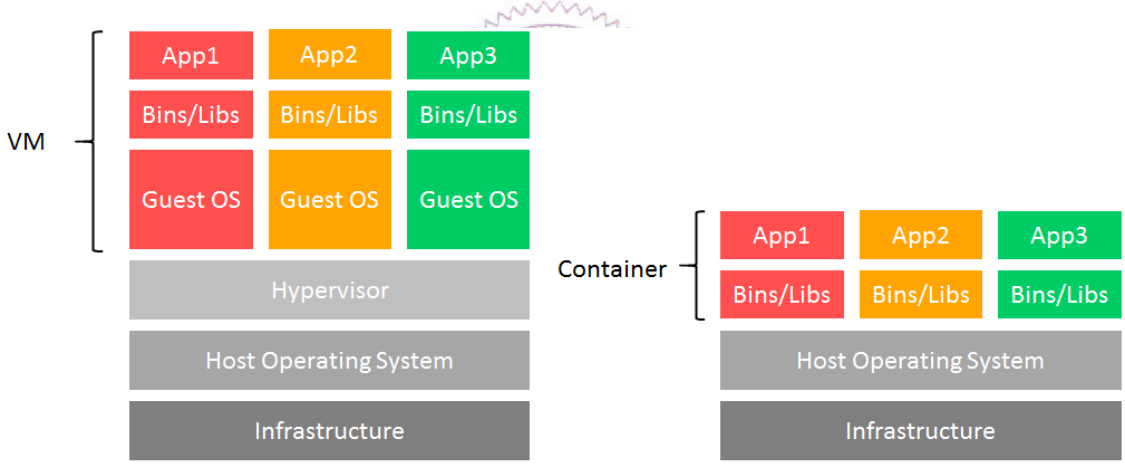


Figure 1.1: Virtual Machine and Container architecture

Among many emerged container systems, such as LXC [16], LXD, and Open VZ, Docker [22] is the most popular container engine. Docker uses Docker daemon to manage multiple containers in a single node. Through the Command Line Interface (CLI), Docker clients can manage containers with operations, such as pulling a repository from the registry, running a container, committing a container to a new image, uploading the image to the registry, and terminating a running container.

On top of single containers, orchestration tools are essential to automate the deployment and operation of complex systems that involve multiple containers on cluster of edge nodes. Several container orchestration tools have been or are being developed. Kubernetes [5] is a system developed by Google, which utilizes hierarchical architecture to manage thousands of containers for different systems on a large clusters. Apache Mesos [17] is another orchestration system, whose original design is to co-allocate multiple heterogeneous distribute systems, such as MPI and MapReduce, into a group of machines. Recently, it also includes containers into the system.

For fog computing, both Kubernets and Mesos may be too complex for a small cluster of edge nodes. Alternatively, Docker Swarm [2], which is a container orchestration system fully compatible with Docker commands, may be a better choice for managing containers on multiple edge nodes.

Docker Swarm is an on-going project with several under developing functions. Our survey found that Swarm lacks of migration mechanism, which is critical for fail over and dynamic provisioning. In addition, the current high availability support in Swarm is incomplete. Swarm can only restart a failed container, but not resume its work. Since for IOT fog computing, many real-time services may utilize in-memory methods to process data, restarting a failed container may loss many valuable data.

In this thesis, we propose a method that utilizes the checkpoint and restore mechanism to enhance the functionalities of current Docker Swarm implementation, migration and high availability more specifically. Checkpoint and restoration [6] can freeze a running process state and save the processing information to checkpoint images. Later, the checkpoint images can be restored to continue the process state. Since a container is a special process in the host system, the checkpoint and restore is a right tool to use. In the design and implementation, we leveraged as many as Docker Swarm's APIs to make future integration easier. We used Docker Swarm's scheduler to help the migration decision. For high availability, we designed a multiple versions checkpoint mechanism to further enhance the availability. Moreover, we investigated the methods to optimize the space and performance of checkpoint and restore.

We have designed 6 experiments, including the container migration performance and the influence of the parameters on container checkpoint time and container

checkpoint image size. The experimental results presents that the container checkpoint time will be affected by many ways, such as memory size and many container checkpoint at the same time, etc. In addition, whenever we used pre-dump and track-memory technology, it will get better performance and use smaller storage in the most cases.

The rest of this thesis is organized as follows. Chapter 2 introduces the background of our thesis. Chapter 3 presents implementation of system architecture, Chapter 4 demonstrates the experimental results. Chapter 5 presents related works. Finally Chapter 6 concludes the thesis.

Chapter 2

Background

This chapter introduces the used packages, including Docker, Docker Swarm, and CRIU (Checkpoint-and-Restore In Userspace).

2.1 Docker

Docker is an open-source container engine, which provides an additional layer of abstraction and automation of operating-system-level virtualization in Linux. Moreover, Docker has extra image management and layered file system to optimize the performance of container and to reduce disk space. Docker images and Docker containers share the same image layer file if they have the same data. To initialize a container, Docker creates a writable layer that allow the container application to change file system.

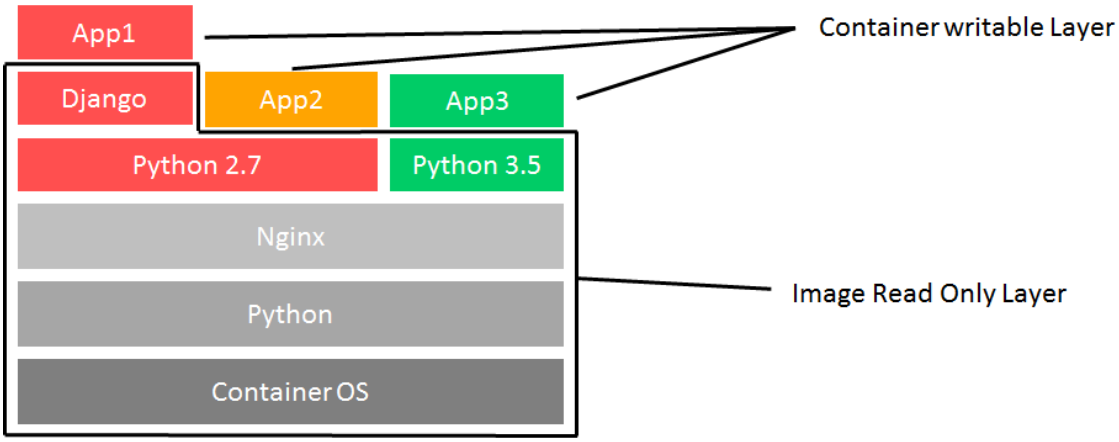


Figure 2.1: Docker layered file system

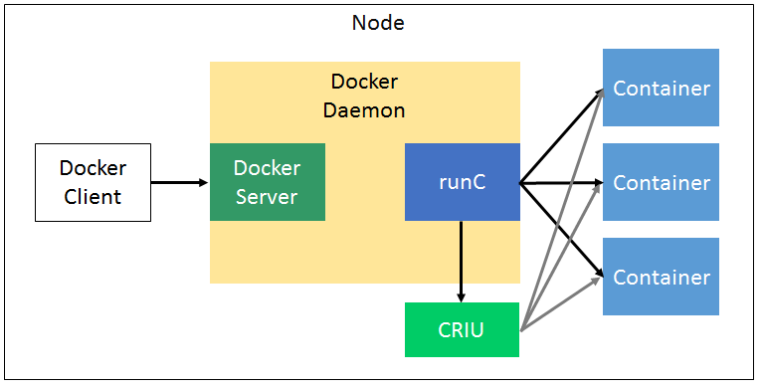


Figure 2.2: Single node Docker

Docker is a typical client-server architecture, which consists of two parts: Docker client and Docker daemon. Docker client provides the CLI (Command Line Interface) to send/receive requests to/from Docker Daemon. Also, Docker supports RESTful API [10] to communicate with Docker daemon. Docker daemon runs as system service, which serves three importance functions:

- Receive and handle requests from Docker clients.
- Manage containers.
- Manage container images.

When running, a Docker daemon will receive requests from Docker clients or remote RESTful API. After receiving requests, daemon will pass the requests to a router, which finds the corresponding functions to handle the requests. By default, Docker daemon listens to UNIX socket requests. To receive remote RESTful APIs or Docker Swarm requests, the daemon listens to the TCP socket.

2.1.1 runC

After version 0.9, Docker used *libcontainer* as its default execution environment, and *runC* [3] is a repackaging of *libcontainer*, which provides CLI to support the containers management, such as the control of namespace, cgroups, apparmor, network, capabilities, firewall, etc.

2.2 Docker Swarm

Docker Swarm is an orchestration tool for Docker, which gathers several Docker engines together into one virtual Docker engine. Docker Swarm provides standard Docker API, so it can be connected by Dokku, Docker Machine, Docker Compose, Jenkins, DockerUI, Drone, etc. Of course, it also supports Docker client as well.

Docker Swarm has two components: Swarm manager and Swarm node. A Swarm manager is the manager which handles the requests from Docker client and RESTful API and manages multiple Docker nodes'. A Swarm node is an agent which sends heartbeats to the discovery service to ensure the Docker daemons are alive in the cluster.

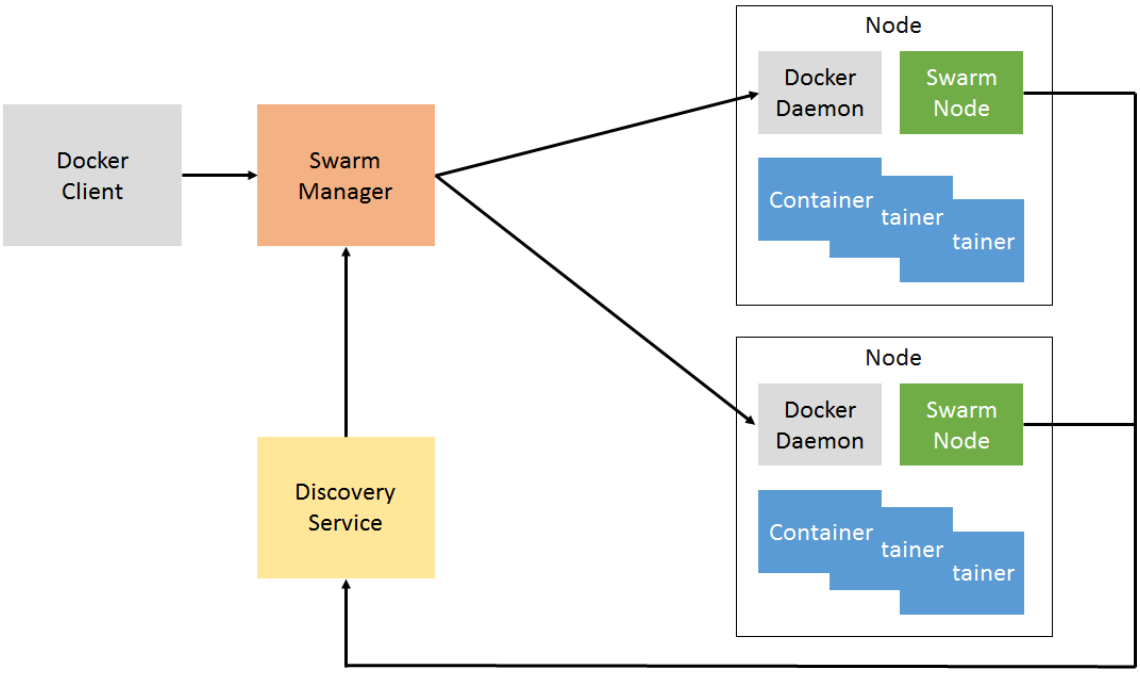


Figure 2.3: Docker Swarm architecture

2.2.1 Discovery services

Docker Swarm provides multiple Discovery Services in the back-ends, which are used to discover the nodes in the cluster. There are:

- Using a distributed key/value store, like Consul, Etcd and Zookeeper.
- A static file or list of nodes.

- Docker Hub as a hosted discovery service.

In addition, it supports any modules that satisfy Discovery API interface.

2.2.2 Scheduler

Docker Swarm scheduler decides which nodes to use when creating and running a container. The scheduling has two steps: First, the scheduler follows user's filters to decide which nodes can be used. Second, the scheduler undergoes multiple strategies to select the best node in the cluster.

2.2.3 High availability of Swarm Manager

Swarm Manager responds to the cluster and manages the resources of multiple Docker Daemons which are served by Swarm Nodes is Docker node or Swarm node? you used both and very confusing. If the Swarm manager dies, a new one will be created to handle with the interruption of service.

The high availability feature allows Docker Swarm has multiple Swarm Manager instances. System manager can create a primary manager and multiple replica instances. Whenever requests are sent to replica instances, requests will be automatically proxied to the primary manager. In addition, if the primary manager fails, the other replica instances will lead a new primary manager.

2.2.4 High availability of Docker Swarm containers

Docker Swarm has a rescheduling policy, which can be set when a container is started. Whenever Swarm nodes fail, Swarm manager will restart all of the containers in that Swarm node to other alive Swarm Nodes.

2.3 CRIU

CRIU [1] (Checkpoint/Restore in Userspace) is a package for checkpointing and restoring of a process. The checkpointing creates a complete snapshot of a process's state, including memory contents, file descriptors, and open TCP connections. CRIU can be used for suspending and resuming processes, or migrating them from one

machine to another. more introduction about CRIU. Like multiple version. What's the differences between predump and dump. How it takes the snapshot, etc. Put some figures....

2.3.1 Checkpoint

The checkpoint procedure relies on /proc file system which CRIU takes the information of process. It needs the pipes parameters, descriptors information, and memory maps. There are 3 steps:

Step 1. Walk through the /proc/$pid/task directory recursively and collect the process tree.

Step 2. Collect tasks' resources and dump to the image files. If the request is predump, CRIU will just collect process tree, and memory pages. Otherwise, it will collect process tree, memory pages, namespaces, cgroups, share memory, and networks, etc. While checkpoint request parameter has track-memory, CRIU will compare checkpoint images and memory pages to reduce storage space.

Step 3. After dumped, CRIU detaches from tasks and they continue to execute.

2.3.2 Restore

When CRIU restores the process, it will morph itself into the tasks it restore. There are 4 steps:

Step 1. CRIU reads image files and resolves which processes share with which resources. Later shared resources are restored by someone process and all the others either inherit one on the 2nd stage (like session) or obtain in some other way.

Step 2. CRIU calls fork() procedure many times by process tree to create the processes which are needed to be restored.

Step 3. Restores all resources such as memory mappings, timers, namespaces, cgroups, threads, etc.

Step 4. Morph CRIU itself into the tasks and continue the process.

Chapter 3

Design and Implementation

3.1 Docker

Native Docker has two parts, Docker Client and Docker Daemon. Docker Daemon has several components include server, engine, registry, graph, driver and runC. To support Checkpoint and Restoration request, some of these steps should be implemented.

3.1.1 Docker Client

There are 3 Docker commands implementation in Docker Client, including Checkpoint, Restore, and Migrate. In Checkpoint command should have these configurations:

- Image directory - Dump checkpoint image directory.
- Work directory - Dump checkpoint image log directory.
- Leave running - After dumping the checkpoint images, consideration is required whether the container keep running.
- Pre-dump - Pre-dump checkpoint memory image to minimize frozen time.
- Pre image directory - Define which version image to compare.
- Track memory - Track memory to pre image directory image to minimize disk space.

In Restore command, it should have these configurations:

- Image directory - Checkpoint image directory to restore from.
- Work directory - Directory for restore log.
- Force - Force restoring the container from image directory whether container is running.

In Migrate command, it focuses on Docker Swarm Scheduler filter configurations. Meanwhile, in Run command, it is possible for the container to set filter configurations with environment variables or labels. Therefore, the implementation of environment variables and labels in Migrate command is required.

3.1.2 Docker Daemon

In native Docker Daemon, it does not support Checkpoint and Restore commands. Fortunately, it is already implemented in runC, so we have to add a proxy between Docker Daemon and runC, which can handle Docker Client's Checkpoint and Restore requests.

3.2 Docker Swarm Configuration

As Figure 3.1, it shows a remote storage server for saving Docker containers dump checkpoint images and it should have fault tolerant to avoid service shutdown.

3.3 Docker Containers Migration in Docker Swarm

Docker Swarm creates containers through Swarm scheduler to dispatch Docker nodes. If a specific node needs to be assigned to create a container, Swarm schedule parameter should be set. In migrating a container in Docker Swarm, the Swarm schedule parameter should be assigned in order to avoid the container to restore to the same node.

Step 1. Check Docker Swarm cluster has at least two Swarm nodes.

Step 2. Parse Docker Client requests to analyse label and environment variables, and transform label and environment variables to Docker Swarm filters.

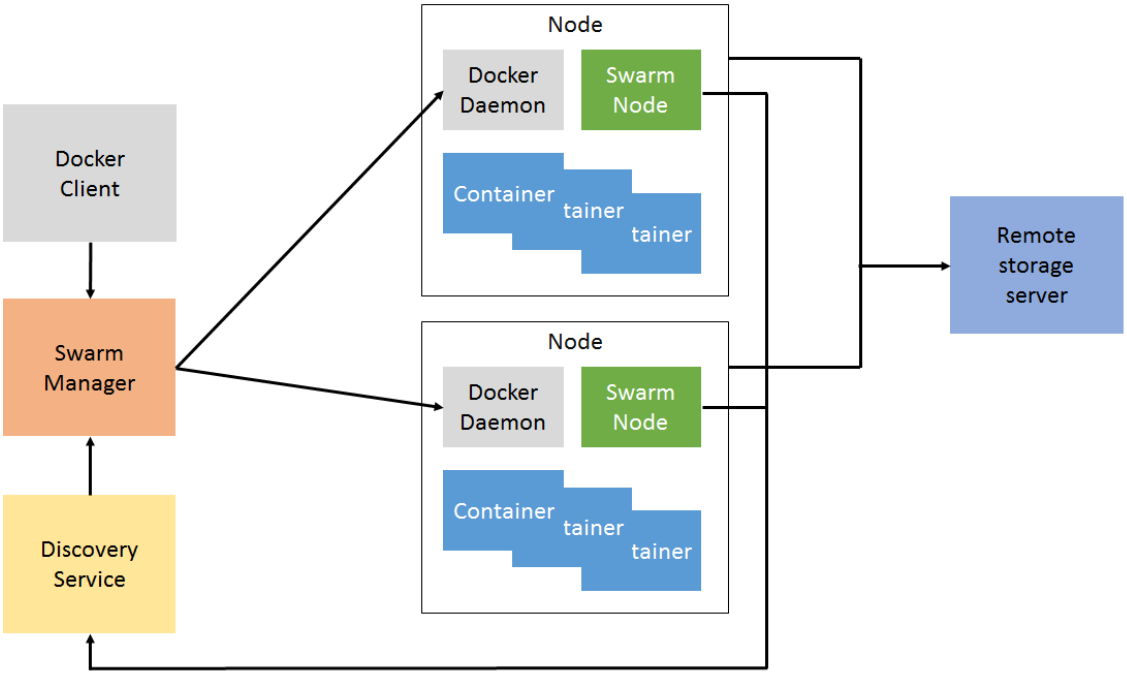


Figure 3.1: Docker Swarm with remote storage server

Step 3. The Swarm schedule parameter should be added to make sure the container will not be migrated to the same node.

Step 4. Create empty container on the Docker Swarm scheduler which node its pointing to, this could lead to downloading the container image if it doesn't exist in the be chosen node.

Step 5. Pre-dump the container's checkpoint images which we want to migrate to decrease container's frozen time.

Step 6. Dump the container's checkpoint images by tracking memory from the pre-dumped checkpoint images.

Step 7. Restore the container's checkpoint images to the empty container that Docker Swarm scheduler points to.

Step 8. Delete the checkpoint images.

Step 9. If the container has been migrated which has set the Checkpoint and Restoration Rescheduling Policy, it will restart the Checkpoint and Restoration Rescheduling Policy 3.4.

3.4 Docker Swarm Checkpoint and Restoration Rescheduling Policy

In Docker Swarm, it has Rescheduling Policy. As the reschedule policy is set when we start a container, whenever Swarm nodes fail, Swarm Manager will restart the containers to another alive Swarm Nodes.

To improve this policy, a Checkpoint Ticker should be needed to keep every containers' checkpoints up to date. Whenever Swarm Nodes fail, Swarm Manager will restore the containers which their checkpoints have been dumped by Swarm Manager to the remote storage server. Otherwise, the Checkpoint Ticker provides the version of checkpoint image by tracking memory. It only dumps the different memory pages' checkpoint to new version checkpoint images.

By experiment results, when many containers checkpoint in the same time, all of the containers checkpoint time will increase, especially the remote storage is hard disk. To avoid many containers checkpoint in the same time, the checkpoint requests from different containers will be collected in a checkpoint queue. It only allows one container checkpoint at a time.

In addition, Docker Swarm Checkpoint and Restoration Rescheduling Policy also supports high availability whenever Docker Swarm primary manager fails, the others Swarm Manager replica instances will lead to a new primary manager. After replica has chosen a new primary manager, it will restart the container's Checkpoint Tickers. By experiment, it can save at least 2 times storage space when VG_i bigger than 5.

3.4.1 Docker Swarm Container Checkpoint Ticker

Algorithm 1 shows the Container Checkpoint Ticker's algorithm. In line 1 and 2, checkpoint-time period T_i and version-group VG_i are set when the container C_i is created through Docker Swarm Manager, and every T_i is independent. Each T_i in the Docker Swarm cluster has its starting time and time ticker, time ticker will dump checkpoint image repetitively at regular intervals. If version-group VG_i has not been set, it will be set to 5. Checkpoint version $V_i = 0$ and pre-dump checkpoint version $pre\text{-}dumpV_i = 0$ should be defined for each container C_i.

In line 4 to 18, for each container is running and C_i checkpoint ticker period T_i

pounds, C_i, Swarm Manger will do line 5 to 17. In line 6 to 11, if the container C_i does not has any pre-dump image, newest version-group VG_i checkpoint is full or previous pre-dump average images size over 500 MB, Swarm Manger will send the request to Swarm Node N_i to create a new directory and pre-dump the checkpoint image as pre-dump version $pre\text{-}dumpV_i$. After pre-dumping the container checkpoint image or C_i has previous version checkpoint images, Swarm Manager will send requests to dump checkpoint image to Swarm Node N_i to dump checkpoint version V_i. Whenever dumping checkpoint image, CRIU will track the memory difference to the previous checkpoint images or the pre-dumping checkpoint images to reduce image disk space usage. As it is shown as Figure 3.2, every container has each pre-dump checkpoint directories and each pre-dump checkpoint directories have version-group versions checkpoint images. In line 12 to 14, Swarm Manager sends delete checkpoint request to N_i to delete oldest Pre-dump version directory whenever container has more than three Pre-dump versions directory.

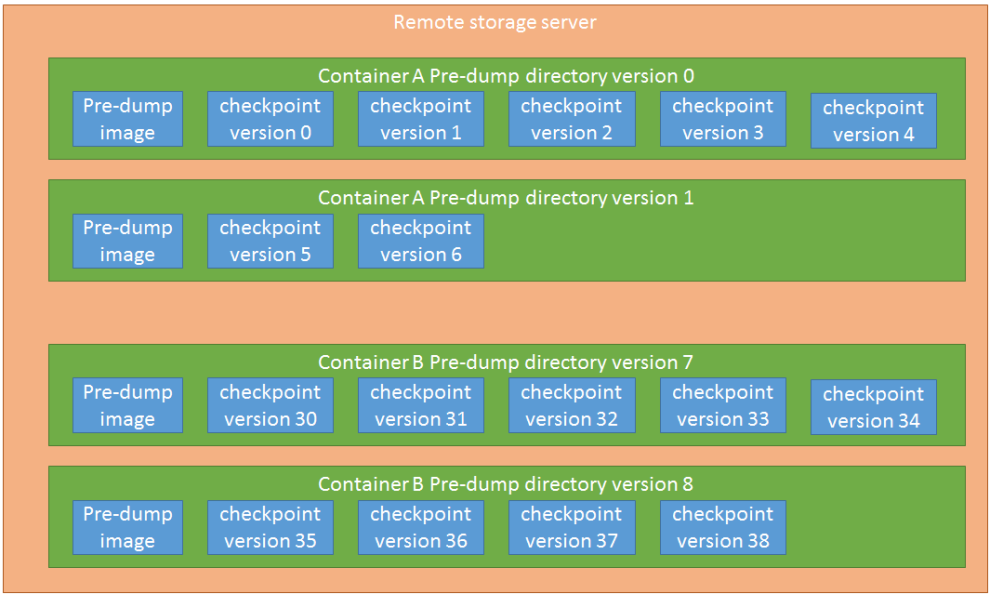


Figure 3.2: Containers checkpoint versions in remote storage server

3.4.2 Docker Swarm Restore Rescheduling Policy

Algorithm 2 gives a specific explanation of Restore Rescheduling Policy. In line 1, R_i label is set when the container C_i is created by Docker Swarm Manager. In

line 3 to 7, whenever Swarm Nodes N_i fail, Swarm Manager will execute procedure RESTORE CONTAINER that restores every container C_i which has R_i label to another Swarm Nodes N_i.

In line 9 to 21, each container C_i executes procedure RESTORE CONTAINER; Swarm Manager will choose a Swarm Node N_r and create an empty container C_r in N_r. In this step, Swarm Manager will check whether N_r has the container's image. If N_r doesn't have the container's image, N_r will download the container's image from Docker remote registry and create the empty container C_r. As result of creating container C_r, Swarm Manager restores the fail container C_i checkpoint image to C_r. To avoid a fail restoration, Swarm Manager will retry to restore the second last version's checkpoint, and it will retry version-group VG_i (default is 5) for several instances. The container C_i checkpoint image will be removed after restoring the container C_i.

Whenever Swarm Manager's attempts to retry are all failed for VG_i times, it will create and restart a new container as normal Docker Swarm Rescheduling Policy.

3.4.3 High Availability of Swarm Manager in Docker Swarm Checkpoint and Restore Rescheduling Policy

Whenever Docker Swarm primary manager fails, the others Swarm Manager replica instances will lead a new primary manager. After replica leading a new primary manager, it searches every Docker Node's containers which has Checkpoint and Restore Rescheduling Policy's label. If the containers have Checkpoint and Restore Rescheduling Policy's labels, Docker Swarm new primary manager will restart container checkpoint tickers.

Algorithm 1 Checkpoint Ticker Algorithm

Input: The set of Swarm Nodes: $\{N_1, N_2, ..., N_m\}$; The set of containers: $\{C_1, C_2, ..., C_n\}$; Version-group: VG_i; Checkpoint-ticker period: T_i;

Output: pre-dump images $pre\text{-}dumpV_i$; dump images V_i ;

Set T_i and VG_i labels when create the container C_i;
initial $V_i = 0$, $pre\text{-}dumpV_i = 0$;

for each C_i in N_i **do**
    while C_i running & T_i pounding **do**
        if version % $VG_i == 0$ **then**
            if $pre\text{-}dumpV_i - 1$ average images size< 500 MB **then**
                add pre-dump checkpoint image $pre\text{-}dumpV_i$ request to queue;
            end if
        end if
        add dump checkpoint image V_i request to queue;
        if VG_i directory > 3 **then**
            delete oldest pre-dump checkpoint $pre\text{-}dumpV_i$ directory;
        end if
        $V_i = V_i + 1$;
        if V_i % $VG_i == 0$ **then**
            $pre\text{-}dumpV_i = pre\text{-}dumpV_i + 1$;
        end if
    end while
end for

Algorithm 2 Restore Rescheduling Algorithm

Input: The set of Swarm Nodes: $\{N_1, N_2, ..., N_m\}$; The set of containers: $\{C_1, C_2, ..., C_n\}$; Version-group: VG_i; Checkpoint-ticker period: T_i; Restore rescheduling policy label:R_i;

Output: Restored containers:C_{ri};

Set R_i labels when creating the container;

if N_i fail **then**
    for all C_i in fail N_i has R_i **do**
        RESTORE CONTAINER;
    end for
end if

procedure RESTORE CONTAINER
    N_r ⟵Create a empty container C_r
    for V_i downto V_i - VG_i **do**
        C_{ri} ⟵ Restore checkpoint V_i;
        if Restore C_{ri} success **then**
            break
        end if
    end for
    Delete the container checkpoint image;
    if Restore container C_i fail **then**
        C_r ⟵ Restart the container; C_i;
    end if
end procedure

Chapter 4

Experiments

4.1 Environment

We developed our experiments environment cluster were listed as table 4.1, and created 4 virtual machines in the computer that each of their resource were listed as table 4.2. We will mount NFS to each virtual machines and test 2 types each of NFS, hard disk, and memory(tmpfs). In addition, we will test the difference between **Direct** method that is defined as new checkpoint image and **Track-memory** method that is defined as pre-dump checkpoint image which is followed by checkpoint images.

CPU	Intel Xeon(R) CPU-E5-2630 v3 @ 2.40GHz x 32
Memory	64 GB
Disk	1.0 TB
OS	Ubuntu 15.10 64 bit
Kernel version	4.4.4-040404-generic
Docker version	1.10.3
Docker Swarm version	1.2
Virtual Box version	5.10.14

Table 4.1: Experiment environment

CPU	Intel Xeon(R) CPU-E5-2630 v3 @ 2.40GHz x 4
Memory	8 GB
Disk	40 GB
OS	Ubuntu 15.10 64 bit
Kernel version	4.4.4-040404-generic
Docker version	1.10.3
Docker Swarm version	1.2

Table 4.2: Virtual Machine experiment environment

4.2 Container Migration Time

As shown in Figure 4.1, native is a based line of the other kinds of experiment. We choose Redis[25] as our migration container, because it uses memory to save its data. Therefore, we can check the correctness of Redis data to make sure container migration is success.

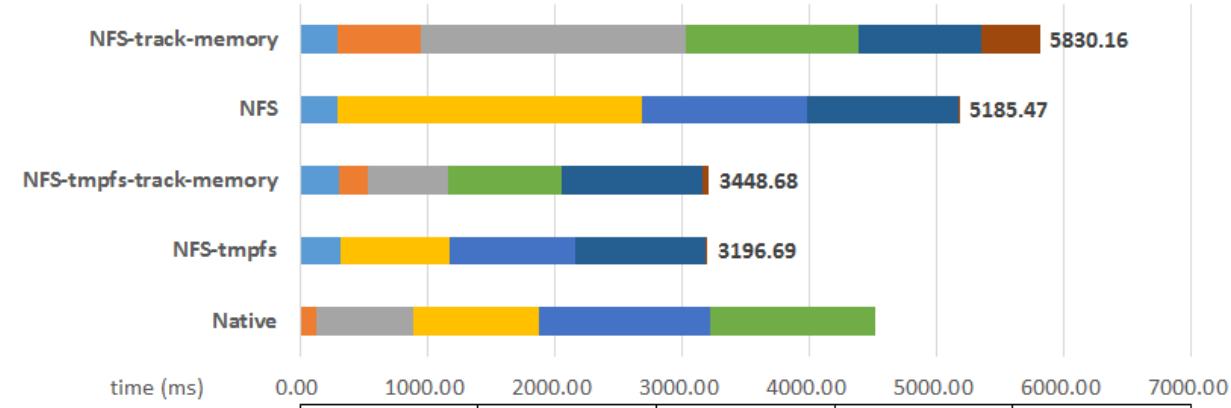
In the beginning, creating the container on another Swarm Nodes is fast for all environments because we assume that they have downloaded their container images from the remote repository, thus, they do not need to transport container images from the other Swarm Nodes.

Next, in **Track-memory** method, the container has to pre-dump the checkpoint image at the version-group created. After pre-dumping the checkpoint image, dumping checkpoint will track memory different with the pre-dump checkpoint image. On the other hand, **Track-memory** method, checkpoint ticker dumps the checkpoint image directly. The result of total checkpoint time, **Track-memory** method is longer than the other one, because it has to add pre-dumping checkpoint time and dumping checkpoint time. Although **Track-memory** method is longer, but it provides the less frozen time to the container that improves CPU's utilization.

Third, Restoring the container to the container which has already been created in the Swarm Node. Except NFS with memory, they all have nearly performance.

Final, it has big disparity of the delete checkpoint image step at **Track-memory** method, because it has more image files and directories than the version without pre-dumping version.

After all, the checkpoint and restoration of container in memory is faster than hard disk in NFS. As result of Figure 4.1, NFS with memory has the best performance in this experiment.



	Native	NFS-tmpfs	NFS-tmpfs-track-memory	NFS	NFS-track-memory
Create container		319.20	310.00	300.80	295.55
Pre-dump	133.88		225.08		657.86
Checkpoint-track-memory	754.36		631.95		2080.38
Checkpoint	990.69	858.41		2385.28	
Restore-track-memory	1346.70	992.76		1294.80	
Restore	1294.80		891.65		1358.67
Remove container		1020.29	1099.38	1198.54	959.44
Delete checkpoint		5.44	59.43	5.43	467.28

Create container
Pre-dump
Checkpoint-track-memory
Checkpoint
Restore-track-memory
Restore
Remove container
Delete checkpoint

Figure 4.1: Docker Swarm migration with remote storage server

4.3 The Influence of Container Checkpoint Time on Container Process Time

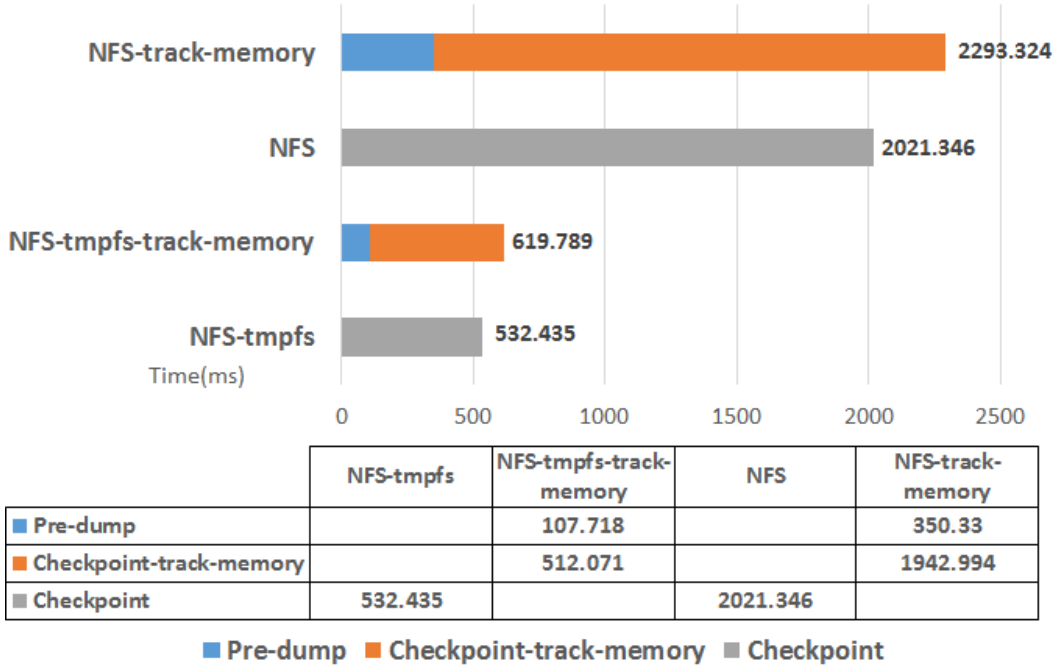
In this experiment, sysbench[19] is used to test performance of process's CPU execution time. Our parameter of sysbench is:

sysbench –test=cpu –cpu-max-prime=200000 run

It needs to run around 11 minutes in native container without dumping any checkpoint to find the 200000th prime number.

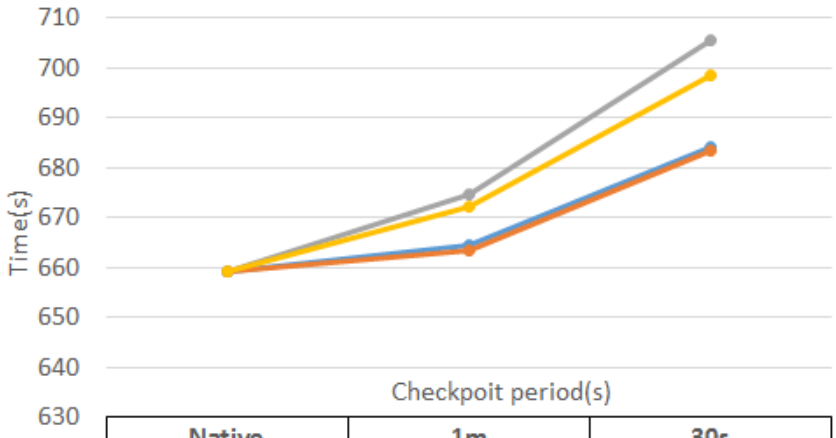
In Figure 4.2, this figure lists every checkpoint time in the experiment environments, also, if remote storage uses the memory to save the checkpoint images, it will get better performance. In the checkpoint restore rescheduling policy (section 3.4), we set a parameter of checkpoint-ticker period T_i which will checkpoint repeatedly for every checkpoint-ticker over.

As Figure 4.3, the result of the checkpoint-ticker period T_i is in direct ratio to the container process execution time. As these experiment results, the checkpoint-ticker period T_i has a big influence of container execution time, it is an important parameter in checkpoint restore rescheduling policy that if T_i is too small, the container will get a bad performance. But if T_i is too big, whenever Swarm Node fails, the restore container needs to execute the process again. The worst, it might lose some important data. However, no matter the total checkpoint time in **Track-memory** method, the process's execution time are nearly **Direct** method. This situation shows that the pre-dump checkpoint time has no influence on the process's execution time.



	NFS-tmpfs	NFS-tmpfs-track-memory	NFS	NFS-track-memory
Pre-dump		107.718		350.33
Checkpoint-track-memory		512.071		1942.994
Checkpoint	532.435		2021.346	

Figure 4.2: Container checkpoint time of container process time



	Native	1m	30s
NFS-tmpfs	659.266	664.4987	684.2486
NFS-tmpfs-track-memory	659.266	663.5849	683.29059
NFS	659.266	674.74035	705.44272
NFS-track-memory	659.266	672.20544	698.54033

Figure 4.3: ontainer Checkpoint Time Influence of Container Process Time

4.4 The Influence of Container Memory Size on Container Checkpoint Time

In this experiment, there are 3 different memory size processes measured. These processes allocate 1 MB, 100 MB, and 1 GB and change the memory with random values quickly in a while loop.

As shown in Figure 4.4, the pre-dump checkpoint time is at most 1/5 of the checkpoint time and track-memory checkpoint time is about the same as checkpoint time.

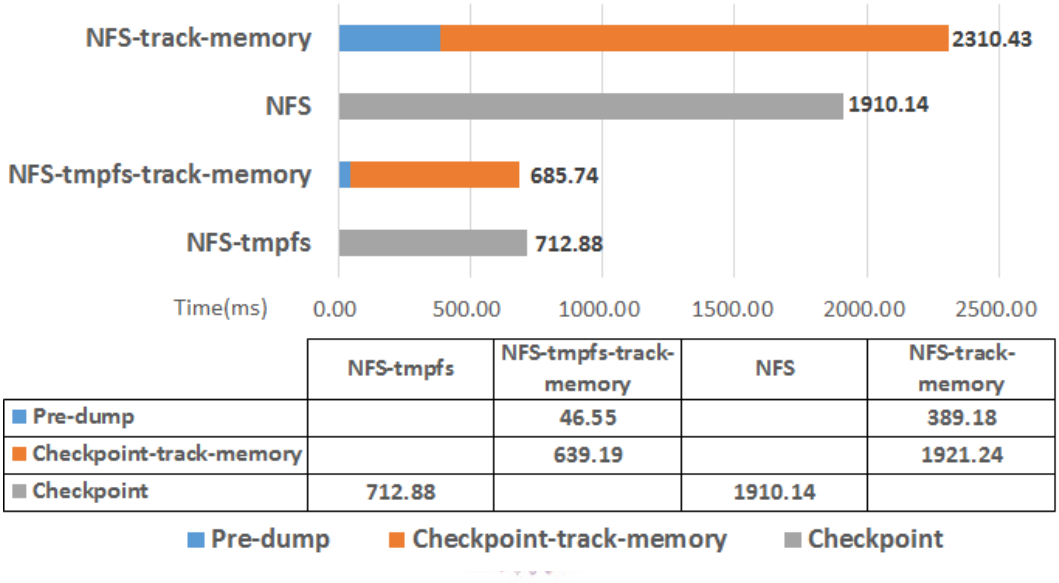
However, in Figure 4.5, the pre-dump checkpoint time is almost a halt of the checkpoint time in the 100 MB process. In this scenario, the track-memory checkpoint time is still nearly as checkpoint time, but the total checkpoint time is about 1.5 times longer then checkpoint time without track-memory, it means when every version-group is created, the first total checkpoint time is 1.5 times longer then **Direct** method's checkpoint time .

In the 1 GB process, Figure 4.6 shows that the pre-dump checkpoint time is almost

as same as the checkpoint time. It situation shows that the total checkpoint time is about 2 times longer then **Direct** method when every version-group is created.

In contrast, If the process has allocated the memory but didn't use it, the result will show as Figure 4.7. This figure shows whatever how many memories are allocated in the process, the pre-dump checkpoint time and checkpoint time are all smaller than the process which is allocated memories with changing it.

Follow these experiments, the memory's change has a big influence about container checkpoint time. It is not a effectiveness way when the process need to change memory a lot in the checkpoint-ticker period T_i time.



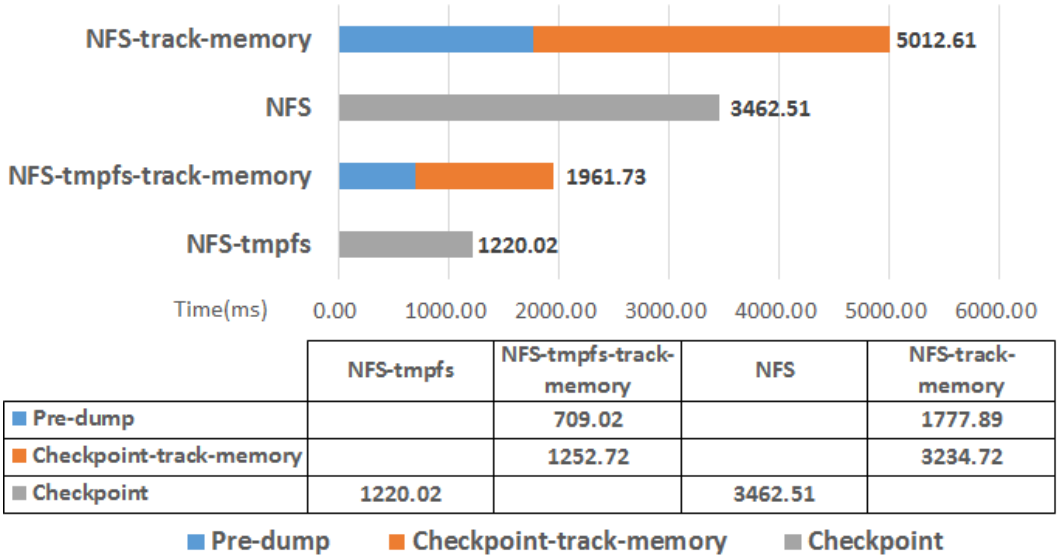
	NFS-tmpfs	NFS-tmpfs-track-memory	NFS	NFS-track-memory
Pre-dump		46.55		389.18
Checkpoint-track-memory		639.19		1921.24
Checkpoint	712.88		1910.14	

Figure 4.4: 1MB container process's checkpoint time

4.5 The Influence of Container Memory Usage on Container Checkpoint Image Size

In the section 4.4, the container checkpoint time is direct ratio to process memory usage. In this section, the checkpoint images size will be estimated.

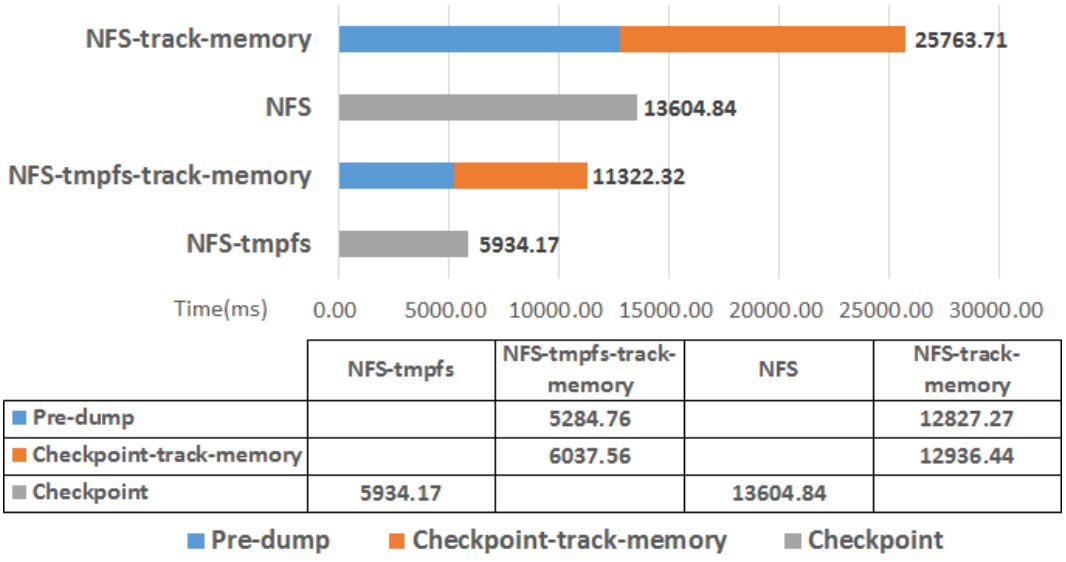
In Table 4.3, it shows that the experiments in the section 4.4. If the process uses more memory, the checkpoint image size will be larger. However, although process allocates memory a lot, the checkpoint size still small unless the memory be used.



	NFS-tmpfs	NFS-tmpfs-track-memory	NFS	NFS-track-memory
Pre-dump		709.02		1777.89
Checkpoint-track-memory		1252.72		3234.72
Checkpoint	1220.02		3462.51	

Pre-dump Checkpoint-track-memory Checkpoint

Figure 4.5: 100MB container process's checkpoint time



	NFS-tmpfs	NFS-tmpfs-track-memory	NFS	NFS-track-memory
Pre-dump		5284.76		12827.27
Checkpoint-track-memory		6037.56		12936.44
Checkpoint	5934.17		13604.84	

Pre-dump Checkpoint-track-memory Checkpoint

Figure 4.6: 1GB container process's checkpoint time

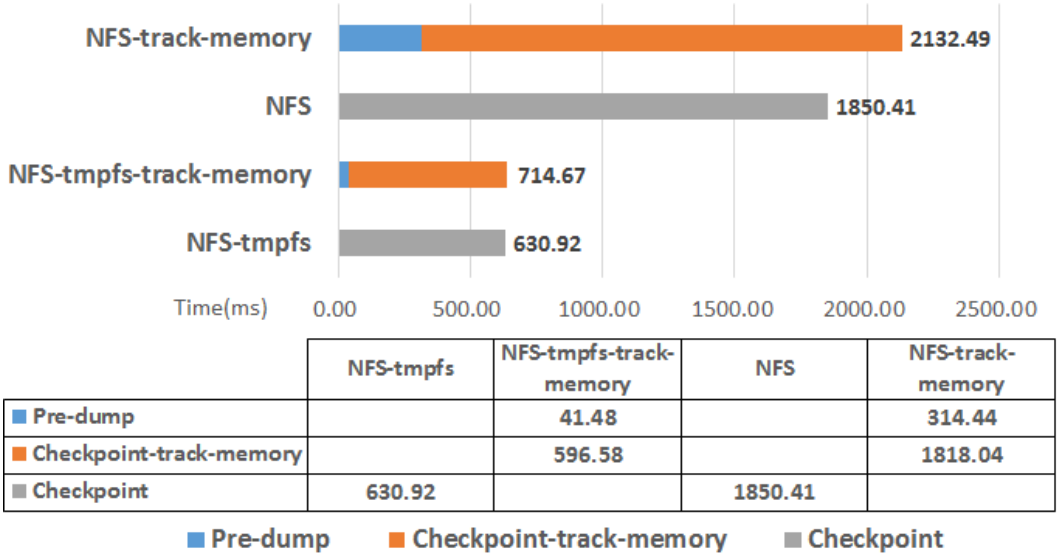


Figure 4.7: Allocated memory without change process's checkpoint time

In Table 4.3, it shows that track-memory version of checkpoint is useless, because it takes more checkpoint time and more storage spaces. To change this view, we test Redis benchmark for daily application. The redis-benchmark command is:

redis-benchmark -t set -l

This command will send 100000 requests with 50 parallel clients. The results shown as table 4.4, the pre-dump checkpoint image size is similar as checkpoint image size, but the track-memory checkpoint image size is smaller than the others. This experiment result shows that the Table 4.3 is the worst case in checkpoint image size. In this case, if track-memory checkpoint is used, it will save more than 2 times storage space.

As these results, we observe the container checkpoint image size is related with the container checkpoint time. When the container checkpoint time increasing, the container checkpoint image size will increase, too. When the container checkpoint image size over 500 MB, it means that the container memory has changed a lot, pre-dump checkpoint is an inefficient action. To avoid unnecessary container pre-dump, whenever container checkpoint group's average image size over 500 MB, next checkpoint group will not pre-dump the container checkpoint.

	1 MB	100 MB	1 GB	Allocate memory without changing
Pre-dump	1 MB	100 MB	1 GB	102 KB
Checkpoint-track-memory	1 MB	100 MB	1 GB	90.9 KB
Checkpoint	1 MB	100 MB	1 GB	175.2 KB

Table 4.3: Process allocated memory's checkpoint image size

	Redis	Redis-benchmark
Pre-dump	9 MB	14 MB
Checkpoint-track-memory	2.3 MB	2.3 MB
Checkpoint	9.6 MB	14 MB

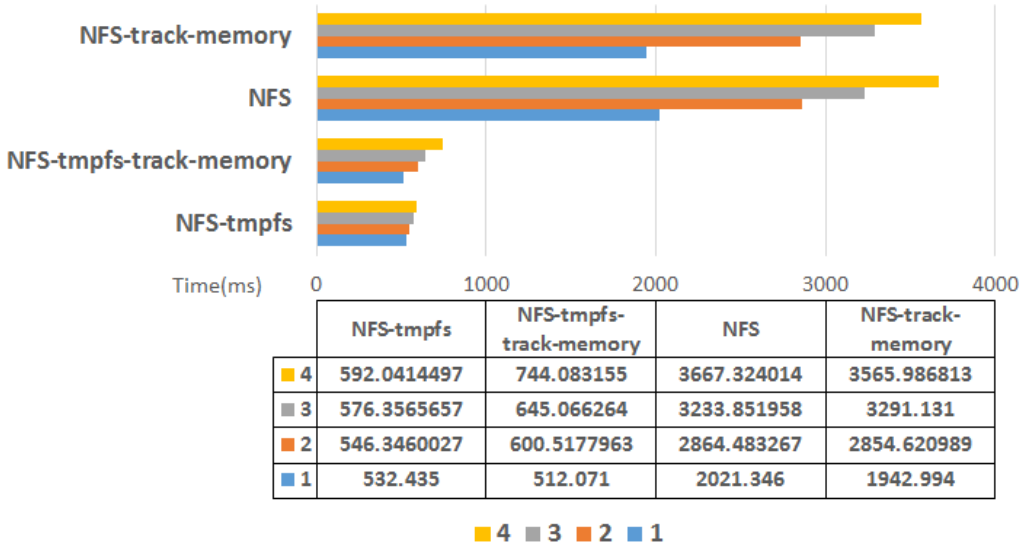
Table 4.4: Redis and Redis benchmark's checkpoint image size

4.6 The Influence of Many Containers Checkpoint at The Same Time on Container Checkpoint Time

Figure 4.8 demonstrates the influence of checkpoint time when many containers checkpoint in the same. As more containers checkpoint in the same time, the more checkpoint time for each container has to take. In storing in disk case, it almost needs twice times when 4 containers checkpointing in the same time. That's the reason that we implement checkpoint queue to solve this problem.

4.7 The Influence of Container Checkpoint Versions on Container Checkpoint Time

As Figure 4.9, **Direct** method's checkpoint time average have 2 horizontal lines. Two **Track-memory** method's trendline are slow increasing when the container checkpoint versions growing up. Each of them has one point of intersections with **Direct** method's checkpoint time line, the values are and 5 and 16. Follow these results, checkpoint-version default value sets to 5 is a better choice, and it should



	NFS-tmpfs	NFS-tmpfs-track-memory	NFS	NFS-track-memory
4	592.0414497	744.083155	3667.324014	3565.986813
3	576.3565657	645.066264	3233.851958	3291.131
2	546.3460027	600.5177963	2864.483267	2854.620989
1	532.435	512.071	2021.346	1942.994

Figure 4.8: Many containers checkpoint in the same time

not exceed than 16.

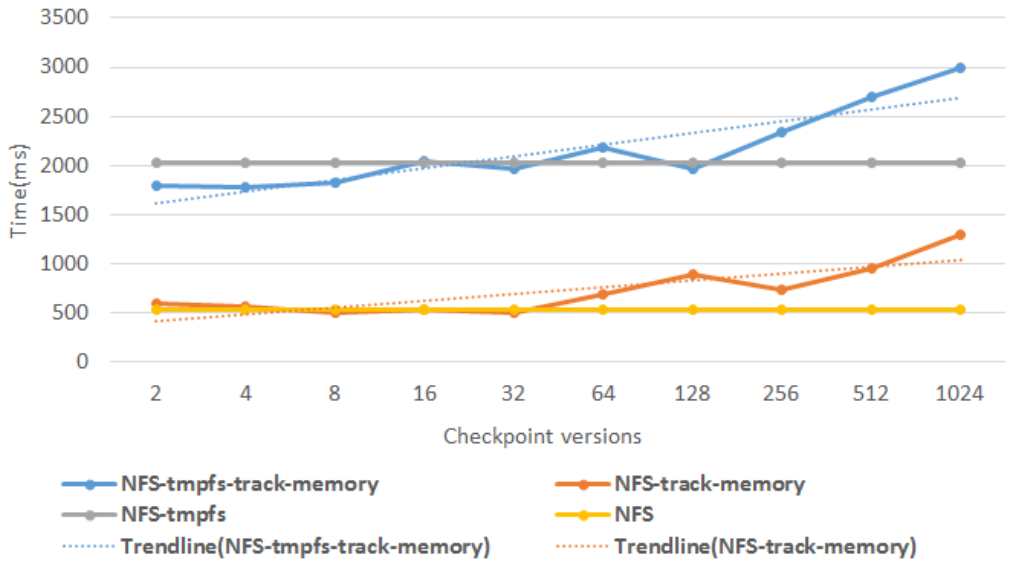


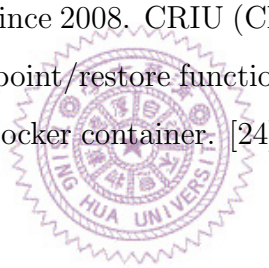
Figure 4.9: Checkpoint versions of container checkpoint time

Chapter 5

Related Work

The migration of Virtual Machine [11, 21] has been researched for a long time. However, it always has to take a long time because Virtual Machine's resources such as memory, and image spaces are large.

Container migration is a growing researches. Most of researches used checkpoint and restoration to accomplish container migration. Checkpoint-restoration of Linux applications [20] was a project since 2008. CRIU (Checkpoint/Restore in Userspace) is a project to implement checkpoint/restore functionality for Linux in userspace. In [31], it used CRIU to migrate Docker container. [24] used checkpoint to live migrate OpenVZ container.



Chapter 6

Conclusion

CRIU provides the container for dumping the checkpoint image and restoring it in the single host machine. In this thesis, we have purposed it to expand to Docker Swarm in the cluster. To support high availability in the Docker Swarm cluster, we also use checkpoint ticker and restoration to recover last container work state in the other Swarm Node. In addition, we implement pre-dump checkpoint to reduce container frozen time and storage spaces; in many cases, it saves about $10\% \sim 20\%$ frozen time and at least 200% storage usage.

In the future, we plan to implement live migration the Docker container to support network applications' migration. For now, we have done Docker network's restore, however, it will lost some packets when container migration and disconnect about 1 second, this problem will be solved In addition, Virtual Machine has some researches in power consumption algorithm, we want to extend power consumption algorithm in Docker Swarm to enhance power consumption saving in the Docker Swarm cluster.

Bibliography

[1] Criu. `https://criu.org/Main_Page`.

[2] Docker swarm. `https://docs.docker.com/swarm/overview/`.

[3] runc. `https://runc.io/`.

[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[5] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[6] Sukadev Bhattiprolu, Eric W. Biederman, Serge Hallyn, and Daniel Lezcano. Virtual servers and checkpoint/restart in mainstream linux. *SIGOPS Oper. Syst. Rev.*, 42(5):104–113, July 2008.

[7] Eric W Biederman and Linux Networx. Multiple instances of the global linux namespaces. In *Proceedings of the Linux Symposium*, volume 1, pages 101–112. Citeseer, 2006.

[8] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[9] NM Mosharaf Kabir Chowdhury and Raouf Boutaba. A survey of network virtualization. *Computer Networks*, 54(5):862–876, 2010.

[10] Jason H Christensen. Using restful web-services and cloud computing to create next generation mobile applications. In *Proceedings of the 24th ACM SIGPLAN*

conference companion on Object oriented programming systems languages and applications, pages 627–634. ACM, 2009.

[11] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[12] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 161–174. San Francisco, 2008.

[13] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. Virtualization vs containerization to support paas. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 610–614. IEEE, 2014.

[14] Robert P Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, 1974.

[15] Jim Gray and Daniel P. Siewiorek. High-availability computer systems. *Computer*, 24(9):39–48, 1991.

[16] Matt Helsley. Lxc: Linux container tools. *IBM devloperWorks Technical Library*, 2009.

[17] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[18] Ann Mary Joy. Performance comparison between linux containers and virtual machines. In *Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in*, pages 342–346. IEEE, 2015.

[19] Alexey Kopytov. Sysbench: a system performance benchmark. *URL: http://sysbench. sourceforge. net*, 2004.

[20] Oren Laadan and Serge E Hallyn. Linux-cr: Transparent application checkpoint-restart in linux. In *Linux Symposium*, pages 159–172. Citeseer, 2010.

[21] Haikun Liu, Hai Jin, Cheng-Zhong Xu, and Xiaofei Liao. Performance and energy modeling for live migration of virtual machines. *Cluster computing*, 16(2):249–264, 2013.

[22] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[23] Karissa Miller and Mahmoud Pegah. Virtualization: virtually at the desktop. In *Proceedings of the 35th annual ACM SIGUCCS fall conference*, pages 255–260. ACM, 2007.

[24] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. Containers checkpointing and live migration. In *Proceedings of the Linux Symposium*, volume 2, pages 85–90, 2008.

[25] Matti Paksula. Persisting objects in redis key-value database. *University of Helsinki, Department of Computer Science*, 2010.

[26] Aameek Singh, Madhukar Korupolu, and Dushmanta Mohapatra. Server-storage virtualization: integration and load balancing in data centers. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 53. IEEE Press, 2008.

[27] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 275–287. ACM, 2007.

[28] Salvatore J Stolfo, Malek Ben Salem, and Angelos D Keromytis. Fog computing: Mitigating insider data theft attacks in the cloud. In *Security and Privacy Workshops (SPW), 2012 IEEE Symposium on*, pages 125–128. IEEE, 2012.

[29] Miguel G Xavier, Marcelo V Neves, Fabio D Rossi, Tiago C Ferreto, Timoteo Lange, and Cesar AF De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240. IEEE, 2013.

[30] Li Yan. Development and application of desktop virtualization technology. In *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on*, pages 326–329. IEEE, 2011.

[31] Chen Yang. Checkpoint and restoration of micro-service in docker containers. 2015.