

國立清華大學

資訊工程研究所

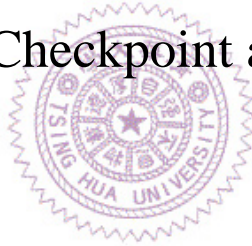
碩士論文

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



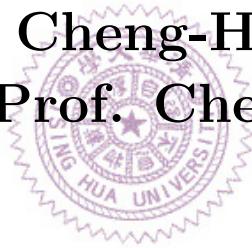
研究生：103062622 黃晟豪 (Cheng-Hao Huang)

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

Container becomes a popular technology since Docker has published in 2013. Container technology has solved programmer and system maintainer when they have to deploy their applications in the servers. They just need to build the container images, and the container images can run on everywhere they want. Container technology also isolates each container environments just likes a single host machine.

In this paper, we not only migrate the Docker containers between two physical machines, but also migrate the Docker containers in the Docker Swarm cluster. Furthermore, we enhance the high availability in the Docker Swarm cluster, the container can be set that checkpoint the container images to remote storage service; whenever the Docker Nodes are fail, the containers which run in the failed Docker Nodes will restore the newest checkpoint to the alive Docker Nodes as well.

# Contents

|  |          |
|--|----------|
| Chinese Abstract   | i        |
| Abstract   | ii       |
| Contents   | iv       |
| List of Figures  | vi       |
| List of Tables   | vii      |
| List of Algorithms   | viii     |
| <b>1 Introduction</b>  | <b>1</b> |
| <b>2 Background</b>  | <b>3</b> |
| 2.1 Docker . . . . .   | 3        |
| 2.1.1 Docker Client . . . . .                                | 4        |
| 2.1.2 Docker Daemon . . . . .                                | 4        |
| 2.1.3 runC . . . . .   | 5        |
| 2.2 CRIU . . . . .   | 5        |
| 2.3 Docker Swarm . . . . .                                   | 5        |
| 2.3.1 Discovery services . . . . .                           | 6        |
| 2.3.2 Scheduler . . . . .                                    | 6        |
| 2.3.3 High availability of Swarm Manager . . . . .           | 7        |
| 2.3.4 High availability of Docker Swarm containers . . . . . | 7        |
| <b>3 Design and Implementation</b>                           | <b>8</b> |
| 3.1 Docker . . . . .   | 8        |
| 3.1.1 Docker Client . . . . .                                | 8        |



|          |   |           |
|----------|---|-----------|
| 3.1.2    | Docker Daemon . . . . .   | 9         |
| 3.2      | Docker Swarm Configuration . . . . .                                | 9         |
| 3.3      | Docker Containers Migration in Docker Swarm . . . . .               | 9         |
| 3.4      | Docker Swarm Checkpoint and Restoration                             |           |
|          | Rescheduling Policy . . . . .                                       | 11        |
| 3.4.1    | Docker Swarm Container Checkpoint Ticker . . . . .                  | 11        |
| 3.4.2    | Docker Swarm Restore Rescheduling Policy . . . . .                  | 12        |
| 3.4.3    | High Availability of Swarm Manager in Docker Swarm Check-           |           |
|          | point and Restore Rescheduling Policy . . . . .                     | 13        |
| <b>4</b> | <b>Experiments</b>  | <b>16</b> |
| 4.1      | Environment . . . . .   | 16        |
| 4.2      | Container Migration Time . . . . .                                  | 17        |
| 4.3      | The Influence of Container Checkpoint Time on Container Process     |           |
|          | Time . . . . .  | 18        |
| 4.4      | The Influence of Container Memory Size on Container Checkpoint Time | 20        |
| 4.5      | The Influence of Container Memory Usage on Container Checkpoint     |           |
|          | Image Size . . . . .  | 21        |
| 4.6      | The Influence of Many Containers Checkpoint in The Same Time on     |           |
|          | Container Checkpoint Time . . . . .                                 | 24        |
| 4.7      | The Influence of Container Checkpoint Versions on Container Check-  |           |
|          | point Time . . . . .  | 24        |
| <b>5</b> | <b>Related Work</b>   | <b>27</b> |
| <b>6</b> | <b>Conclusion</b>   | <b>28</b> |

# List of Figures

|     |   |    |
|-----|---|----|
| 1.1 | Virtual Machine and Container architecture . . . . .                    | 2  |
| 2.1 | Docker layered file system . . . . .                                    | 3  |
| 2.2 | Single node Docker . . . . .  | 4  |
| 2.3 | Docker Swarm architecture . . . . .                                     | 6  |
| 3.1 | Docker Swarm with remote storage server . . . . .                       | 10 |
| 3.2 | Containers checkpoint versions in remote storage server . . . . .       | 12 |
| 4.1 | Docker Swarm migration with remote storage server . . . . .             | 18 |
| 4.2 | Container checkpoint time of container process time . . . . .           | 19 |
| 4.3 | Container Checkpoint Time Influence of Container Process Time . . . . . | 20 |
| 4.4 | 1MB container process's checkpoint time . . . . .                       | 21 |
| 4.5 | 100MB container process's checkpoint time . . . . .                     | 22 |
| 4.6 | 1GB container process's checkpoint time . . . . .                       | 22 |
| 4.7 | Allocated memory without change process's checkpoint time . . . . .     | 23 |
| 4.8 | Many containers checkpoint in the same time . . . . .                   | 25 |
| 4.9 | Checkpoint versions of container checkpoint time . . . . .              | 26 |

# List of Tables

4.1 Experiment environment . . . . . 16

4.2 Virtual Machine experiment environment . . . . . 17

4.3 Process allocated memory’s checkpoint image size . . . . . 24

4.4 Redis and Redis benchmark’s checkpoint image size . . . . . 24





# List of Algorithms

|   |  |    |
|---|--|----|
| 1 | Checkpoint Ticker Algorithm . . . . .    | 14 |
| 2 | Restore Rescheduling Algorithm . . . . . | 15 |



# Chapter 1

## Introduction

Nowadays, cloud computing [5] has been used widely. People would like to use a lot of normal computers and gather them into a cluster. An increasing number of companies build their data centres or use cloud platforms to construct their business application. However, the more computers we have, the larger of a power consumption problem we have to solve. At the same time, to make sure every computers' process in the cluster are alive, high availability [12, 16] becomes a more important role in cloud computing.

Virtualization is a popular technology that is used wildly in cloud computing, including virtual operating systems [24, 30], computer hardware platforms, storage devices [27] and computer network resources [9]. Virtual Machine [15] is a technology to emulate a particular computer like a real computer. It can partition the physical machine's resources, such as CPU, memory, storage, and network. A hypervisor [32, 14] uses execution to manage and share the host physical machine hardware, it allows many different Virtual Machines to be isolated from each other.

Container [28] is an operating system level virtualization which runs as an isolated process in userspace on the host operating system and shares the same operation system kernel with other containers. It provides kernel namespace [8] such as PID, IPC, network, mount, and user namespace to isolate each container environment from the host operation system. In order to control hardware resource such as CPU, memory, network, and disk I/O, container uses cgroups to limit each of the containers resources. Container doesn't have hypervisor to isolate from the host operating system, therefore, it can offer better performance than Virtual Machine

[29, 13, 19]. There are already many software to control container, such as LXC [17], LXD, Open VZ, etc. For now, Docker [23] is the most popular container engine. The Virtual Machine and Container architecture is shown as Figure 1.1.

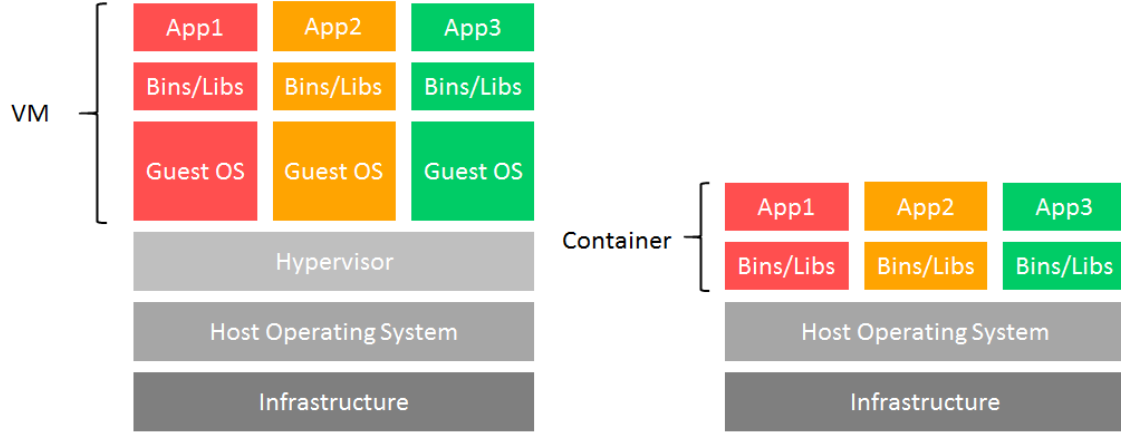


Figure 1.1: Virtual Machine and Container architecture

Checkpoint and restoration [7] can freeze a running process state and save process information to checkpoint images. It can be used to checkpoint image files to restore the process state. These features can be used to checkpoint and restore containers because each container is a process in the host operating system.

In this paper, we choose container but not Virtual Machine to do migration between two host machines, because container need less disk storage and less network transport resources. We not only use checkpoint and restoration to migrate the containers between two physical computers but also migrate the containers in the Docker Swarm cluster. Moreover, we improve high availability and rescheduling feature in Docker Swarm using checkpoint and restoration than keep versions of container checkpoint images in the remote storage server that whenever the computer fails, Docker Swarm manager will restore the containers from the checkpoint images. In some cases, we use pre-dump and track-memory to save about 10% 20% container checkpoint frozen time and at least 200% storage space.

# Chapter 2

## Background

### 2.1 Docker

Docker is an open-source project container engine. It 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 reduce disk space. Docker images and Docker containers share the same image layer file if they have the same data. When Docker creates container, Docker will create a writable layer that allow the container application to change file system. Docker has two parts; Docker Client and Docker Daemon.

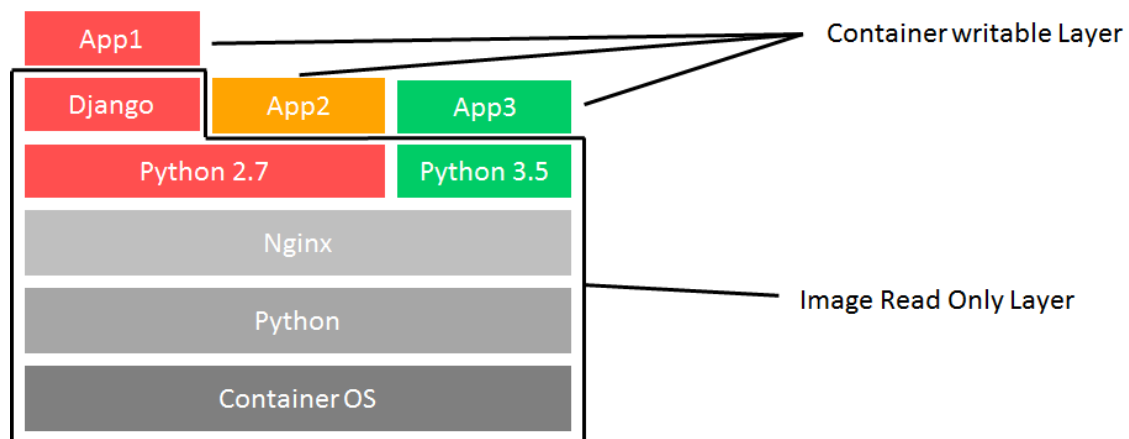


Figure 2.1: Docker layered file system

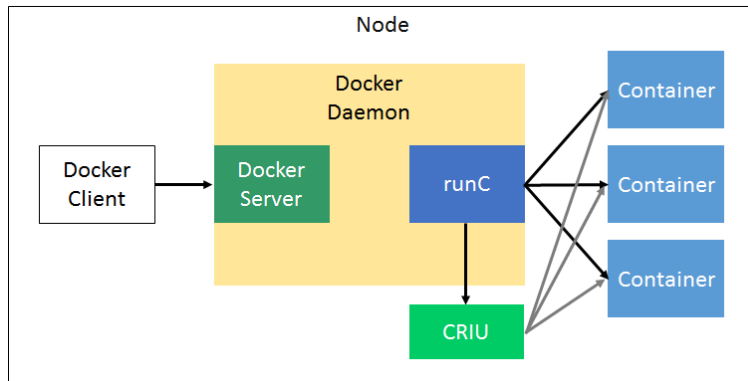


Figure 2.2: Single node Docker

### 2.1.1 Docker Client

Docker is typical client/server architecture application. Docker Client is a CLI (Client Line Interface) in Docker. Docker uses Docker Client to send and receive requests to Docker Daemon. Also, Docker supports remote RESTful API [10] to send and receive HTTP requests to Docker Daemon. In additional, it has been implemented by more than 10 programming languages.

### 2.1.2 Docker Daemon

Docker Daemon is a daemon that runs as system service. It has three of the most importance features:

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

When Docker Daemon is running, it will run a server that receives requests from Docker Clients or remote RESTful API. After receiving requests, server will pass the requests by router to find the handler to handle the requests. By default, Docker Daemon listens to UNIX socket requests, it serves root permission, or docker group membership. Whenever user wants Docker Daemon to listen to remote RESTful API or Docker Swarm requests, it has to enable the TCP socket.

### 2.1.3 runC

When Docker came into being, Docker used LXC as its container execution environment. After Docker version 0.9, Docker dropped LXC but used libcontainer as its default execution environment. runC [4] is basically a repackaging of libcontainer, which is a CLI tool for running containers according to the OCI(Open Container Initiative) [3] specification. It doesn't need any dependency from the operating system, it can control the Linux Kernel, including namespace, cgroups, apparmor, network, capabilities, firewall, etc. runC provides a standard interface to support the containers management that Docker can use it to control the containers.

## 2.2 CRIU

CRIU [1] (Checkpoint/Restore in Userspace) stands for Checkpoint and Restore in User Space, creates a complete snapshot of the state of a process, including things like memory contents, file descriptors, and even open TCP connections. It can be used for suspending and resuming processes, or migrating them from one machine to another.



## 2.3 Docker Swarm

There are many Docker cluster software, like Docker Swarm[2], Google Kubernetes [6], Apache Mesos [18], etc. We choose Docker Swarm because it supports native Docker API, and Docker usage. We don't need to learn the other Docker cluster software.

Docker Swarm is a native clustering for Docker. It gathers several Docker engines together into one virtual Docker engine. Docker Swarm serves 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.

In Docker Swarm, it has two components which are Swarm Manager and Swarm Node. Swarm Manager is the manager which handles Docker Client and RESTful API requests and manages multiple Docker Nodes' resources. Docker Node is an agent which sends heartbeat to Discovery Service to ensure Docker Daemon is alive in the cluster.

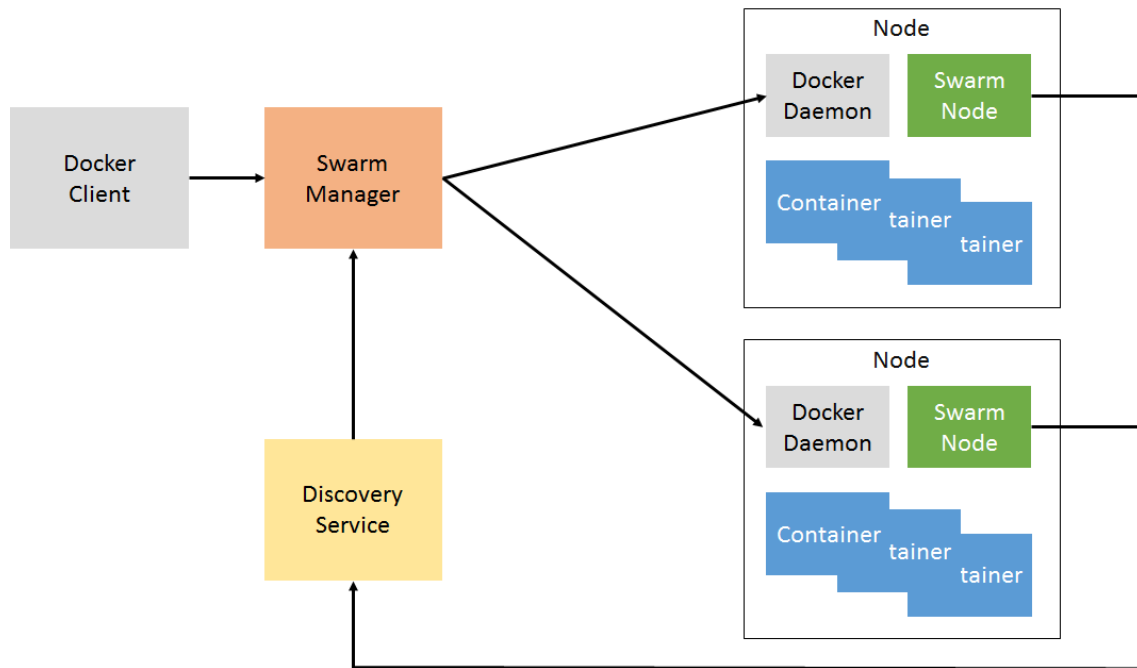


Figure 2.3: Docker Swarm architecture

### 2.3.1 Discovery services

Docker Swarm provides multiple Discovery Services backends. They are used to discover the nodes in the cluster. There are:

- Using a distributed key/value store, like Consul, Etc and Zookeeper.
- A static file or list of nodes.
- Docker Hub as a hosted discovery service.

Otherwise, it also supports any modules that satisfy Discovery API interface.

### 2.3.2 Scheduler

Docker Swarm scheduler decides which nodes to use when creating and running a container. It has two steps: First, the scheduler follows user's filters to decide which nodes conform to it. Secondly, it undergoes multiple strategies to select the best node in the cluster.

### 2.3.3 High availability of Swarm Manager

In Docker Swarm, Swarm Manager responds to the cluster and manages the resources of multiple Docker Nodes at scale. If Swarm Manager dies, we have to create a new one and deal with the interruption of service.

The High availability feature allows Docker Swarm has multiple Swarm Manager instances. We can create a primary manager and multiple replica instances. Whenever we send requests to replica instances, it 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.

### 2.3.4 High availability of Docker Swarm containers

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





# 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.

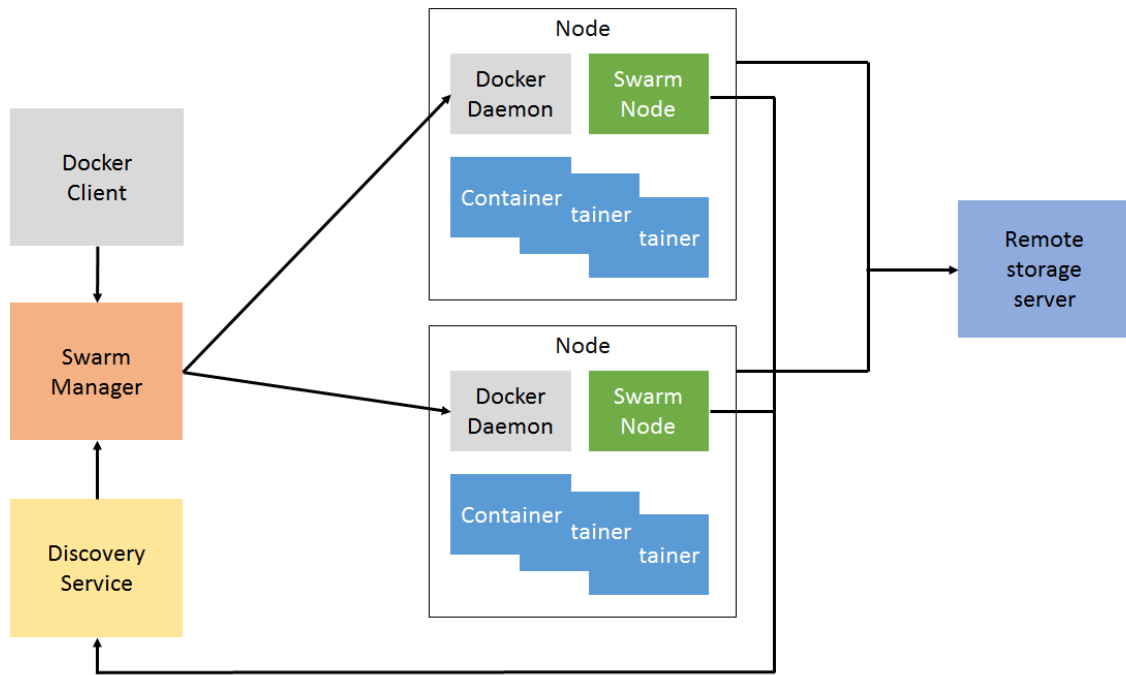


Figure 3.1: Docker Swarm with remote storage server

- 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  $pre\_dumpV_i = 0$  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  $pre-dumpV_i$ . 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.

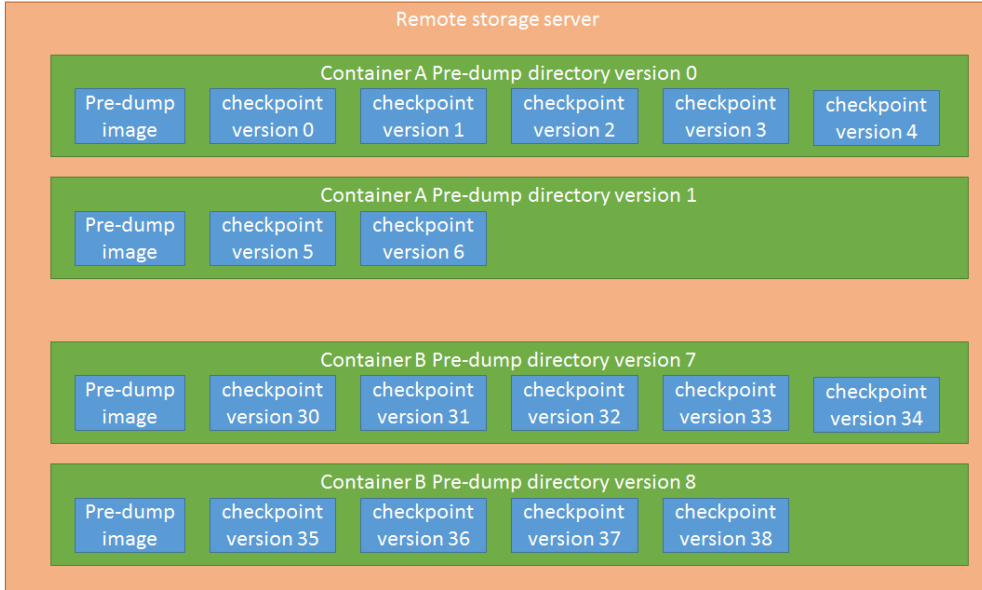


Figure 3.2: Containers checkpoint versions in remote storage server

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

**Output:** pre-dump images  $pre-dumpV_i$ ; dump images  $V_i$  ;

```
1: Set  $T_i$  and  $VG_i$  labels when create the container  $C_i$ ;  
2: initial  $V_i = 0$ ,  $pre-dumpV_i = 0$ ;  
3:  
4: for each  $C_i$  in  $N_i$  do  
5:   while  $C_i$  running &  $T_i$  pounding do  
6:     if version %  $VG_i == 0$  then  
7:       if  $pre-dumpV_i - 1$  average images size < 500 MB then  
8:         add pre-dump checkpoint image  $pre-dumpV_i$  request to queue;  
9:       end if  
10:    end if  
11:    add dump checkpoint image  $V_i$  request to queue;  
12:    if  $VG_i$  directory > 3 then  
13:      delete oldest pre-dump checkpoint  $pre-dumpV_i$  directory;  
14:    end if  
15:     $V_i = V_i + 1$ ;  
16:    if  $V_i \% VG_i == 0$  then  
17:       $pre-dumpV_i = pre-dumpV_i + 1$ ;  
18:    end if  
19:  end while  
20: end for
```

---

---

**Algorithm 2** Restore Rescheduling Algorithm

---

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

**Output:** Restored containers:  $C_{ri}$ ;

```
1: Set  $R_i$  labels when creating the container;
2:
3: if  $N_i$  fail then
4:   for all  $C_i$  in fail  $N_i$  has  $R_i$  do
5:     RESTORE CONTAINER;
6:   end for
7: end if
8:
9: procedure RESTORE CONTAINER
10:   $N_r \leftarrow$  Create a empty container  $C_r$ 
11:  for  $V_i$  downto  $V_i - VG_i$  do
12:     $C_{ri} \leftarrow$  Restore checkpoint  $V_i$ ;
13:    if Restore  $C_{ri}$  success then
14:      break
15:    end if
16:  end for
17:  Delete the container checkpoint image;
18:  if Restore container  $C_i$  fail then
19:     $C_r \leftarrow$  Restart the container;  $C_i$ ;
20:  end if
21: 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[26] 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.

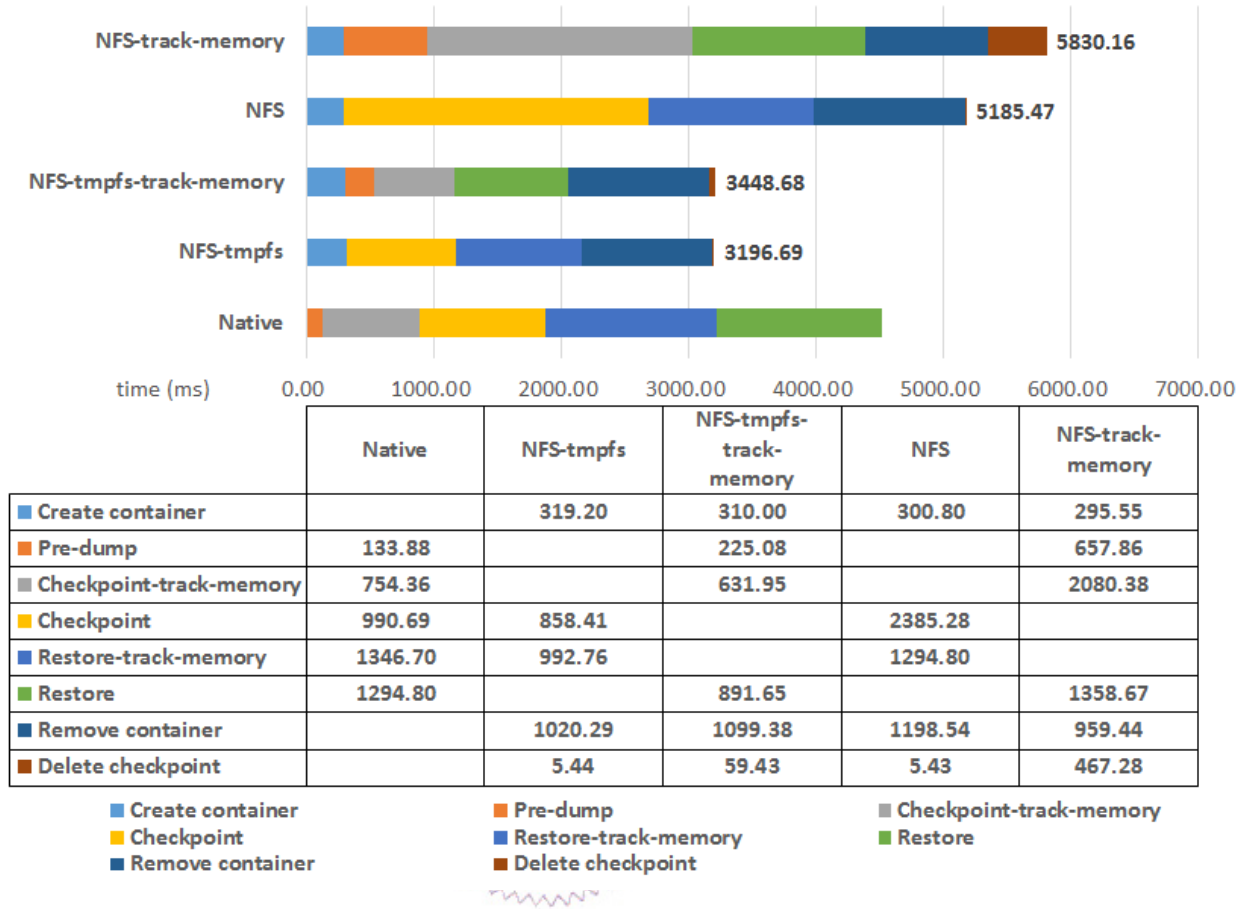


Figure 4.1: Docker Swarm migration with remote storage server

### 4.3 The Influence of Container Checkpoint Time on Container Process Time

In this experiment, sysbench[20] 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.

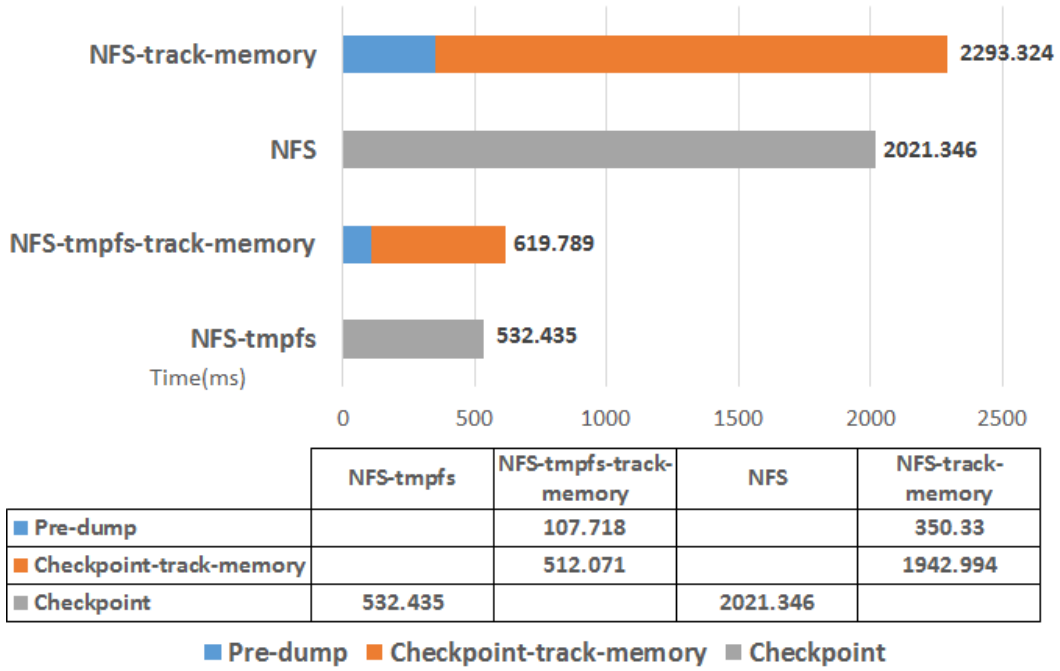


Figure 4.2: Container checkpoint time of container process time

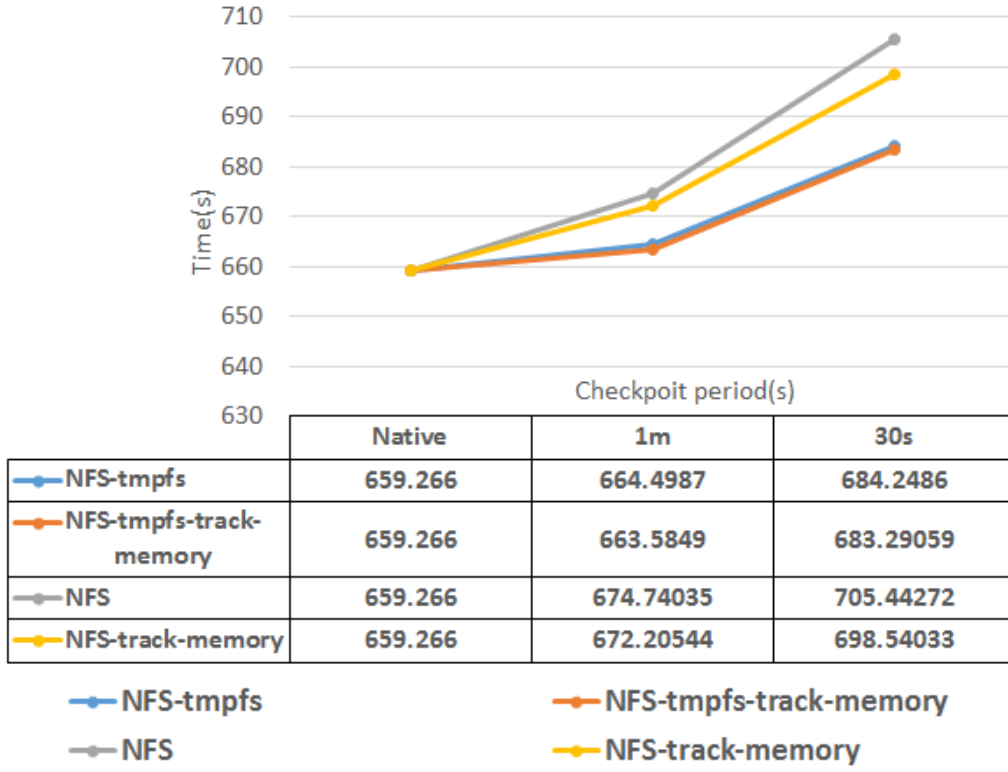


Figure 4.3: ontainer Checkpoint Time Influence of Container Process Time

## 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.

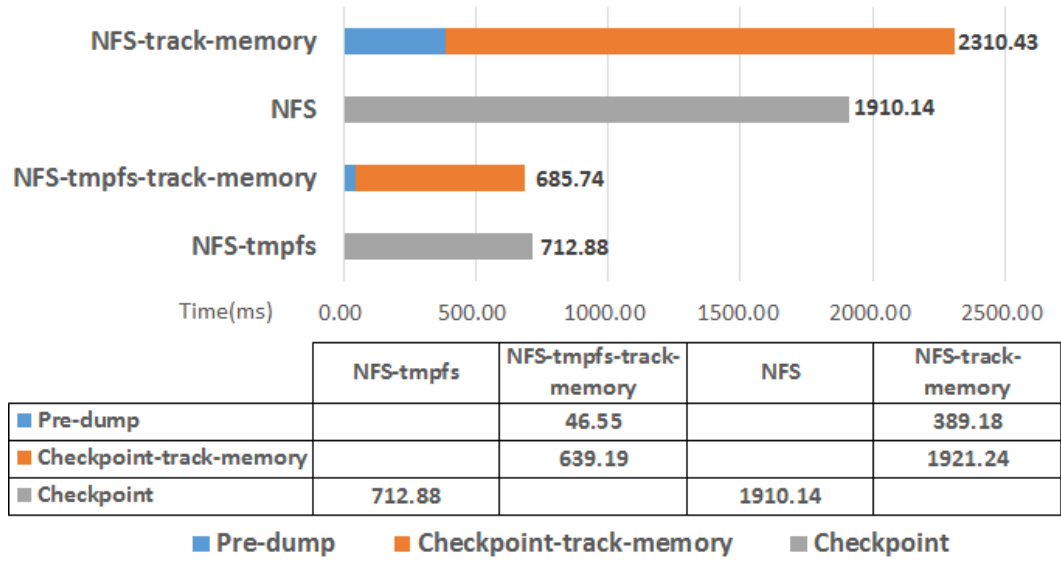


Figure 4.4: 1MB container process's checkpoint time

## 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.

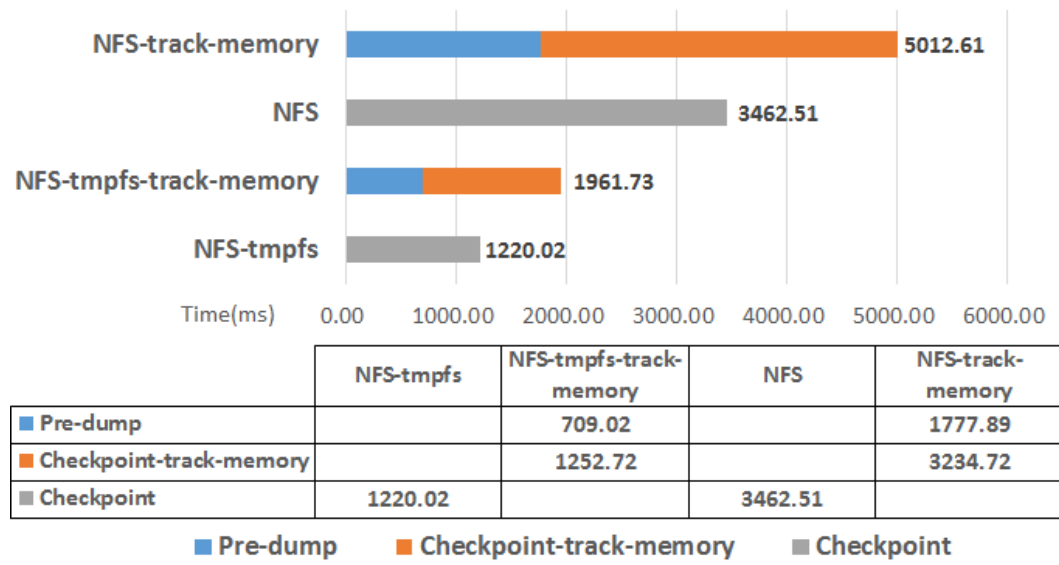


Figure 4.5: 100MB container process's checkpoint time

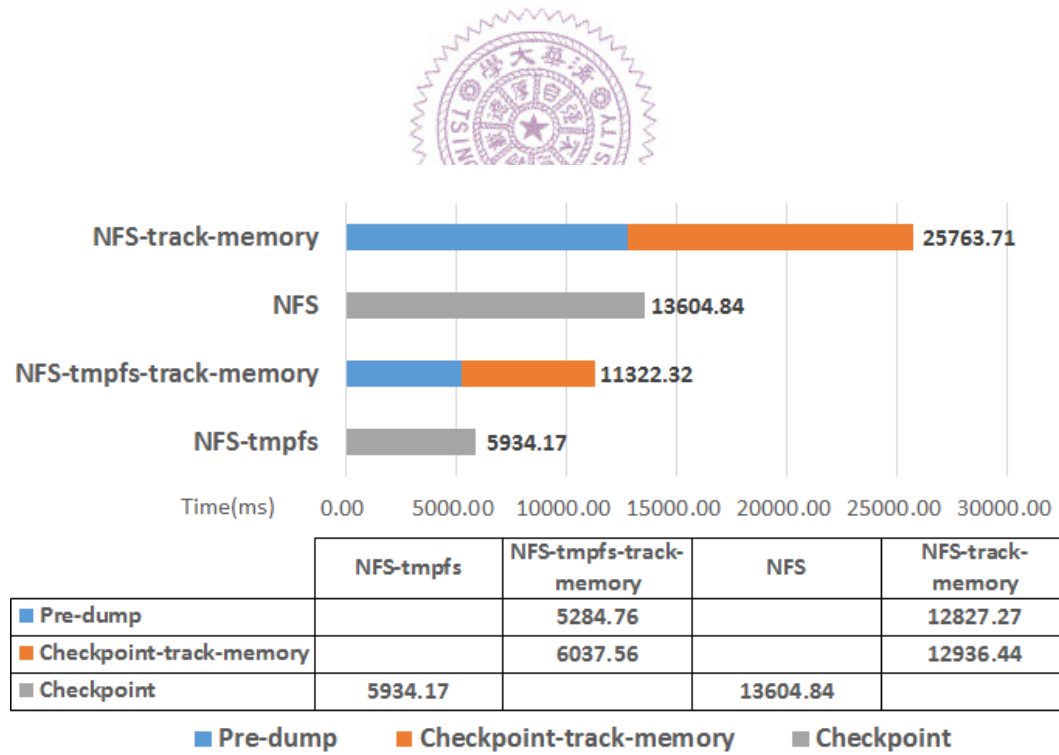


Figure 4.6: 1GB container process's checkpoint time

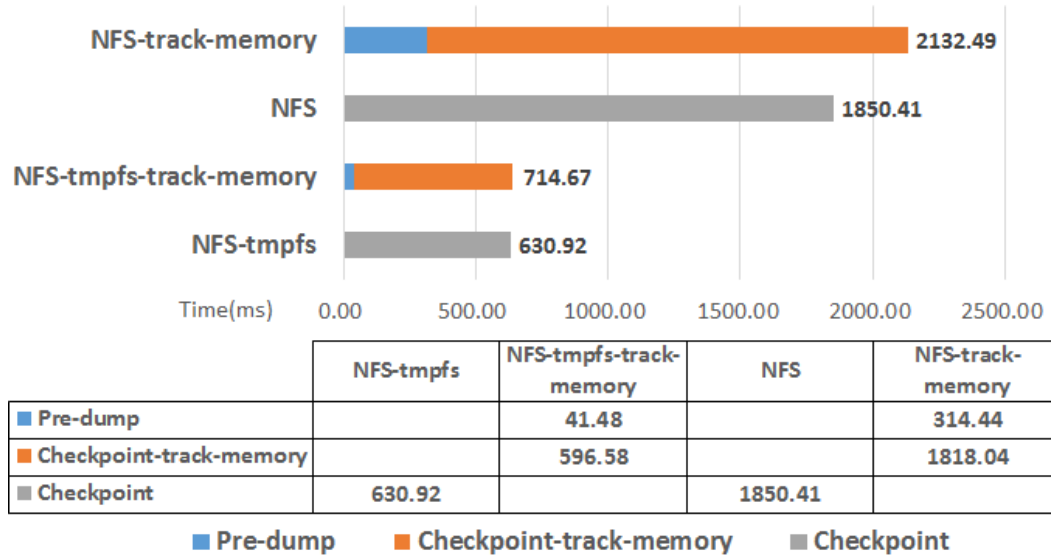


Figure 4.7: Allocated memory without change process’s checkpoint time

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 inefficiently 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 in 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

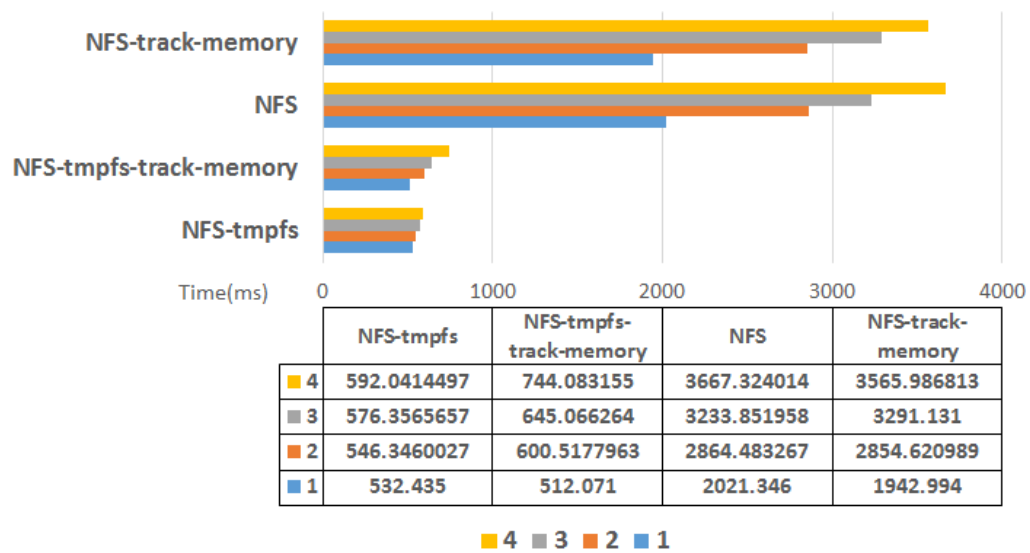


Figure 4.8: Many containers checkpoint in the same time

not exceed than 16.



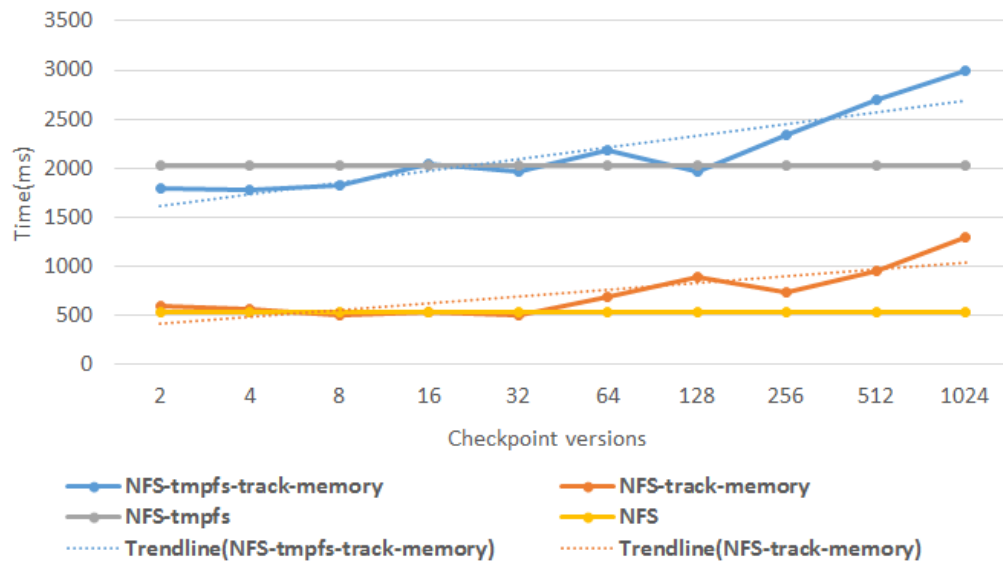


Figure 4.9: Checkpoint versions of container checkpoint time

# Chapter 5

## Related Work

The migration of Virtual Machine [11, 22] 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 [21] 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. [25] 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% ~ 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](https://criu.org/Main_Page).
- [2] Docker swarm. <https://docs.docker.com/swarm/overview/>.
- [3] Oci. <https://www.opencontainers.org/>.
- [4] runc. <https://runc.io/>.
- [5] 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.
- [6] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.
- [7] 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.
- [8] 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.
- [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] Ada Gavrilovska, Sanjay Kumar, Himanshu Raj, Karsten Schwan, Vishakha Gupta, Ripal Nathuji, Radhika Niranjana, Adit Ranadive, and Purav Saraiya. High-performance hypervisor architectures: Virtualization in hpc systems. In *Workshop on System-level Virtualization for HPC (HPCVirt)*. Citeseer, 2007.
- [15] Robert P Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, 1974.
- [16] Jim Gray and Daniel P. Siewiorek. High-availability computer systems. *Computer*, 24(9):39–48, 1991.
- [17] Matt Helsley. Lxc: Linux container tools. *IBM developerWorks Technical Library*, 2009.
- [18] 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.
- [19] 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.

- [20] Alexey Kopytov. Sysbench: a system performance benchmark. *URL: <http://sysbench.sourceforge.net>*, 2004.
- [21] Oren Laadan and Serge E Hallyn. Linux-cr: Transparent application checkpoint-restart in linux. In *Linux Symposium*, pages 159–172. Citeseer, 2010.
- [22] 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.
- [23] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.
- [24] 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.
- [25] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. Containers checkpointing and live migration. In *Proceedings of the Linux Symposium*, volume 2, pages 85–90, 2008.
- [26] Matti Paksula. Persisting objects in redis key-value database. *University of Helsinki, Department of Computer Science*, 2010.
- [27] 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.
- [28] 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.
- [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.
- [32] Andrew J Younge, Robert Henschel, James T Brown, Gregor Von Laszewski, Judy Qiu, and Geoffrey C Fox. Analysis of virtualization technologies for high performance computing environments. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 9–16. IEEE, 2011.

