國 立 清 華 大 學

資訊工程研究所

碩士論文

容器在Swarm叢集中的遷移

Checkpoint and restore containers in Docker Swarm

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

Checkpoint and restore containers in Docker Swarm

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

Thread-Level Speculation (TLS) is one of the parallel frameworks. TLS can avoid the analysis problem of compiler-directed code parallelization and this is helpful for programmers to generate parallel programs. However, the performance is the most important issue for parallel programs. Therefore, we analyse the performance of hardware Thread-Level Speculation (TLS) in the IBM Blue Gene/Q computer.

This paper presents a performance model for hardware Thread-Level Speculation (TLS) in the IBM Blue Gene/Q computer. The model shows good performance prediction, as verified by the experiments. The model helps to understand potential gains from using special purpose TLS hardware to accelerate the performance of codes that, in a strict sense, require serial processing to avoid memory conflicts. Based on analysis and measurements of the TLS behavior and its overhead, a strategy is proposed to help utilize this hardware feature. Furthermore, we compare the performance of hardware Thread-Level Speculation and OpenMP. Based on the performance analysis, we give a direction for deciding between this two parallel frameworks. And the results can not only help users to utilize the TLS but also suggest potential improvement for the future TLS architectural designs.

Acknowledgements

Contents

Chinese Abstract i

Abstract ii

Acknowledgements iii

Contents iv

List of Figures vi

List of Tables vii

List of Algorithms viii

1 Introduction 1

2 Background 2

2.1 Docker 2

2.1.1 Docker client 2

2.1.2 Docker daemon 3

2.1.3 runc 3

2.2 Docker Swarm 3

2.2.1 Discovery services 4

2.2.2 Scheduler 4

2.2.3 High availability 5

2.3 CRIU 6

3 Design and Implementation 7

3.1 Docker 7

3.1.1 Docker client 7
3.1.2 Docker daemon 7
3.2 Docker Swarm configuration 7
3.3 Docker containers migration in Docker Swarm 8
3.4 Docker Swarm rescheduling policy 8

4 Conclusion **9**

List of Figures

2.1 Single node Docker 2
2.2 Docker Swarm architecture 4

List of Tables

List of Algorithms

Chapter 1

Introduction

This is Chapter 1, introduction.

Chapter 2

Background

2.1 Docker

Docker [2] is a open-source project container engine. It provides an additional layer of abstraction and automation of operating-system-level virtualization in Linux. Docker has two parts, including Docker client and Docker daemon.

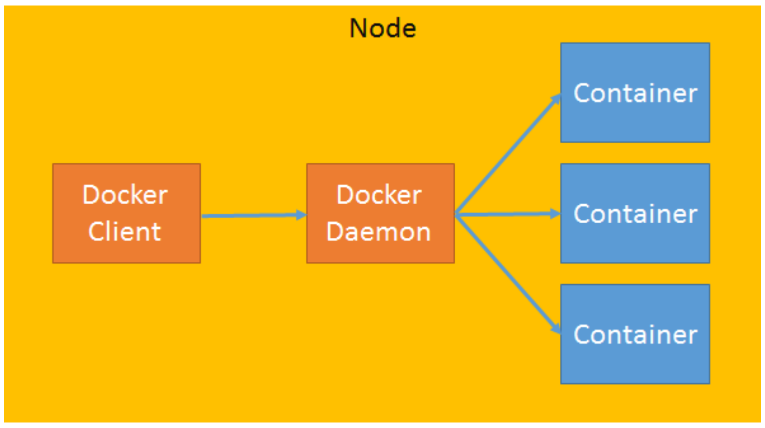

Figure 2.1: Single node Docker

2.1.1 Docker client

Docker is typical Client/Server architecture application. Docker client uses Docker command to send and receive requests to Docker Daemon. Also, Docker supports remote RESTful API 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 two the most importance features:

- Receive and handle Docker client's requests.
- Manage containers.

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.

2.1.3 runc

runc is a CLI tool for running containers according to the OCI specification. It doesn't need any dependency from the system, it can control Linux Kernel include namespace, cgroups, apparmor, netlink, capabilities, firewall, etc. runc provides a standard interface to support the containers management. Docker can use runc to control the containers.

2.2 Docker Swarm

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. It also support Docker client of course. In Docker Swarm, it has two components which are Swarm master and Swarm node. Swarm master 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.

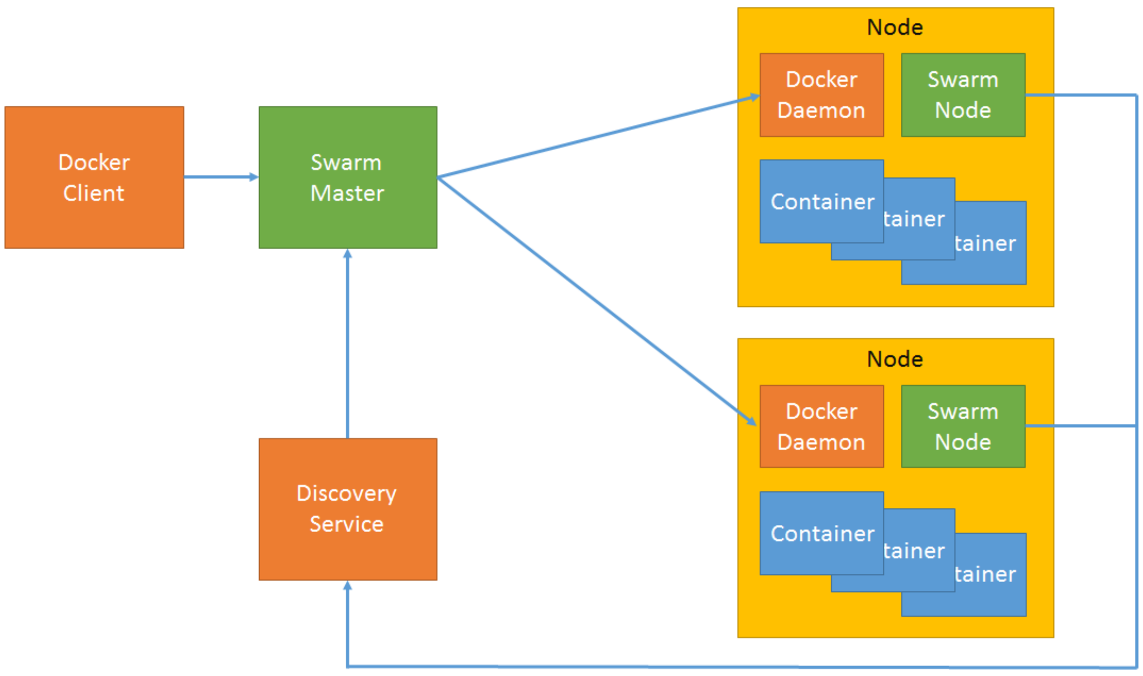

Figure 2.2: Docker Swarm architecture

2.2.1 Discovery services

Docker Swarm provides multiple Discovery Services backends. They are used to discover 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

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

2.2.2 Scheduler

Docker Swarm scheduler decides which nodes to use when creating and running a container. It has two steps. First, It follows user's filters to decide which nodes are conform. Second, It passes through strategies to select the best node in the cluster.

Filter

Filters are divided into two categories, node filters and container configuration filters. Node filters operate on characteristics of the Docker host or on the configuration of the Docker Daemon. Container configuration filters operate on characteristics of containers, or on the availability of images on a host. The node filters are:

- Constraint
- Container slots
- Health filter

The container configuration filters are:

- Affinity
- Dependency
- Port filter

Strategies

The Docker Swarm scheduler features multiple strategies for ranking nodes. Swarm currently supports these values:

- Spread
- Binpack
- Random

Spread and Binpack strategies compute rank according to a nodes available CPU, its RAM, and the number of containers it has. It selects a node at random. Under the Spread strategy, Swarm optimizes for the node with the least number of containers. The Binpack strategy causes Swarm to optimize for the node which is most packed. The Random strategy uses no computation, chooses nodes at random regardless of their available CPU or RAM.

2.2.3 High availability

In Docker Swarm, Swarm manager responses the cluster and manages the resources of multiple Docker nodes at scale. If Swarm master 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 others replica instances will lead a new primary manager.

2.3 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 live migrating them from one machine to another.

2 and cite test [5, 4, 3].

Chapter 3

Design and Implementation

3.1 Docker

In native Docker, It has two part, Docker client and Docker daemon. Docker daemon has many components include server, engine, registry, graph, driver and runC. To support checkpoint and restore request, some of these steps should be implemented.

3.1.1 Docker client

3.1.2 Docker daemon

In native Docker daemon, it doesn't support checkpoint and restore command. Fortunately, it is already implemented in runC, so we have to add some code to proxy Docker client's requests through Docker daemon to runC.

3.2 Docker Swarm configuration

We prepare a remote storage server for saving Docker containers checkpoint images. It should have fault tolerant to avoid service shout down. For these reasons, We choose glusterFS to be our experiments remote storage server. After setting up glusterFS, we mount it to every Docker nodes in the same folder. To avoid mount it as absolute path in every Docker nodes, we mount it to Docker root which we can change configuration in Docker daemon.

3.3 Docker containers migration in Docker Swarm

Docker Swarm creates containers through Swarm scheduler to dispatch Docker nodes. If we want to specific assign which node we want to create containers, we have to set filters like constraint, affinity or dependency. To migrate containers in Docker Swarm, we must avoid the containers which we want to migrate that migrate to an other nodes, instead of 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.

Step 3 Add constraint filter to make sure the container which we want to migrate does not migrate to the same node.

Step 4 Pre-dump the container which we want to migrate to decrease container frozen time.

Step 5 Checkpoint the container by tracking memory.

Step 6 Create empty container on the Docker Swarm scheduler chooses node.

Step 7 Restore the container to the Docker Swarm scheduler chooses node.

Step 8 Delete the checkpoint images.

3.4 Docker Swarm rescheduling policy

Chapter 4

Conclusion

This is conclusion.

Bibliography

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

[2] Docker. `https://www.docker.com/`.

[3] L Grinberg, D Pekurovsky, SJ Sherwin, and GE Karniadakis. Parallel performance of the coarse space linear vertex solver and low e nergy basis preconditioner for spectral *hp* elements. *Parallel Computing*, 35(5):284–304, 2009.

[4] Lance Hammond, Mark Willey, and Kunle Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VIII, pages 58–69, New York, NY, USA, 1998. ACM.

[5] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. *SIGARCH Comput. Archit. News*, 23(2):414–425, May 1995.