上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

学士学位论文

BACHELOR'S THESIS

论文题目：The Impact of Docker Containers on Service Latency

学生姓名：章佐铭

学生学号：5120309626

专　　业：计算机科学与技术

指导教师：李超

学院(系)：电子信息与电气工程学院

Docker 容器对服务延迟的性能影响研究

摘要

通常来说，绝大部分的Web服务都是部署在云服务商所提供的虚拟机上。然而，由于虚拟机会带来极大的性能下降，服务质量(QoS)将会因此受到影响。在众多的服务质量因素中，服务的延迟尤为重要。最近，一种名为Docker的容器，又被称为“轻量级虚拟机”，应运而生。随着它的流行，在云端部署服务又多了一种新的选择。与其他研究吞吐量的文章不同，本文重点在于研究在不同的Docker配置下对服务延迟产生的影响。我们发现当采用CPU配额参数时将会产生比较大的长尾延迟，同一CPU上两个及以上的容器之间将会随着CPU负载的不同产生或好或坏的影响。采用Linux网桥将会产生一个相对固定长度的延迟，而非之前在其他研究中提到的百分比下降。采用AUFS将会在诸如打开文件和扫描文件夹这样的操作中产生额外的延迟，但不会对读写已打开的文件产生性能影响。

关键词： Docker；容器；延迟；Linux 网桥；AUFS

THE IMPACT OF DOCKER CONTAINERS ON SERVICE LATENCY

ABSTRACT

Traditionally, many web services are held on virtual machines (VMs) provided by cloud computing suppliers. Since VMs bring about dramatic performance degradation compared to bare metal, the quality of service (QoS) is affected. Among all the QoS features, service latency is of crucial importance. With the prevalence of Docker, containers, also called “lightweight VM”, offer another choice to deploy web applications on the cloud. This paper takes the first to thoroughly analyze the impact of different Docker configurations on service latency. We conclude that the CPU quota configuration might lead to a long tail latency. Running more than one container on a single CPU can lead to pros and cons according to the CPU workload. Docker bridge could lead to a fixed amount of latency degradation instead of a percentage fallen. Using AUFS could bring about extra latency when opening a file or traversing the file system, and has no effect on writing data to a file.

Key words: Docker; container; latency; Linux bridge; AUFS

Contents

1 Introduction ---- 1
1.1 Motivation ---- 1
1.2 Related works ---- 4
1.3 Contributions ---- 7
1.4 Organization of This Paper ---- 8
2 Background ---- 9
2.1 Docker ---- 9
2.1.1 Container and virtual machine ---- 9
2.1.2 Resource Isolation Using Namespace ---- 11
2.1.3 Resource Limitation Using CGroups ---- 12
2.2 Related Linux Technologies ---- 14
2.2.1 Linux Bridge & Veth Pair ---- 14
2.2.2 Scheduler ---- 15
2.2.3 Tail Latency ---- 16
3 Latency Characterization ---- 18
3.1 Apache Thrift ---- 18
3.2 Experimental Methodology ---- 19
3.3 Containerizing and resource limitation ---- 19
3.4 CPU Configurations ---- 21
3.4.1 Baseline: Native Platform ---- 21
3.4.2 Case 1: Using CPU Shares ---- 22
3.4.3 Case 2: Using CPU Quota ---- 23
3.4.4 CPU Interference ---- 26
3.5 Network Isolation ---- 30
3.5.1 Case 1: Server Receives Data ---- 30
3.5.2 Case 2: Server Sends Data ---- 30
3.5.3 Analysis ---- 31
3.6 File Operations Using AUFS ---- 32
4 Conclusion and Discussions ---- 38
5 Future Works ---- 40

References ---- 44
Acknowledgements ---- 48

Chapter One Introduction

Began from an open-source advanced container engine of dotCloud, a Platform-as-a-Service (PaaS) supplier, Docker is becoming one of the most promising virtualization platform. It significantly shortens the process of packing, shipping and running applications ([27], Merkel D., 2014: 2.). By packing all the dependencies of the application into several image layers, you can carry the package around and run it with simple commands on almost every laptop, personal computer, and even cloud center as long as running a Linux operating system.

Unlike traditional virtual machines, which use hardware-level virtualization, Docker containers employ system-level virtualization and share the same kernel with the host machine ([31], Soltesz S., 2007: 275.). Many researches have proved that containers have a better performance in most cases than VMs. Due to these performance reasons, many companies are trying to move services from virtual machines to containers ([13], He S., 2012: 15.). However, containers do add additional layers compared to bare-metal hardware, which leads to certain degree of performance degradation.

Docker was born to replace virtual machines to some extent. Nowadays, the widely known Infrastructure-as-a-Service (IaaS) platforms like Amazon EC2 uses virtual machines to run applications like cache and database. Most of these applications not only focus on throughput, but also favor real-time low latency. However, related work of Docker focus mostly on containers' influence on throughput instead of the latency degradation. Since Docker provides many choices of resource isolation, in this paper, we will do research on how these configurations will affect the latency performance of real time applications.

1.1 Motivation

Many modern web services like Google and Facebook are interactive. Responses should be returned very soon otherwise users might complain. Also, these services are dynamic. Data centers process huge amount of data based on the user input and response in very limited time. For example, it requires thousands of Memcached machines to do a simple request through Facebook servers ([28], Nishtala R., 2013: 385.) and tens of thousands of index servers to do a Bing search ([17], Jalaparti V., 2013: 219.). In these cases, not only the throughput is of crucial importance to serve as many clients as possible concurrently, latency should also be taken into account to provide users with the best interactivity. Each additional time cost in one of the service backend layers would increase the overall latency. If all of these separated services require software virtualization layer, a millisecond's latency might be amplified to several thousand milliseconds, thus greatly influencing the overall performance of the service.

Tail latency is another issue to care about ([6], Dean J., 2013: 74.). In Map-Reduce task ([7], Dean J., 2008: 107.), a program is processed by hundreds of machines. The result of each map machine is passed to a central reduce machine, so the reduce one has to wait all map ones before moving on. In this case, a map work done over one minute encumbers the whole system even if others are finished in several seconds. Assume that a task has one hundred sub tasks and each sub task has a 99% probability to finish in one microsecond while 1% to finish over 1 second. Then the overall performance of this job is 63.3% probability to finish over one second, which is a rather bad performance. The one-in-one-thousand situation becomes a common case.

The CTO of GigaSpaces claimed a list of interesting phenomenon. He pointed out that latency is a serious matter that can lead to huge profit lost in many companies. Every 100ms of latency would cost Amazon 1% of lost in sales. Also, every extra of 0.5 seconds wasted on generating a search page can drop Google's network traffic 20%. Moreover, if a broker's electronic trading platform can not catch up others' and gets 5 milliseconds behind the competition, they would lose $4 million in revenues per milli-

second. Even these latencies seem relatively small, people hate waiting. They feel repulsed by these less interactive services, quickly click away and finally do other things like turning to the opponents' services. People are talking about how to scaling up the capacity of their services, but they sometimes neglect the importance of building low-latency ones. Service suppliers should try their best to decrease service latency, increase interactivity, and finally lower the customer defection rate ([4], Colgate M., 1996: 23.).

Despite the increasing need of virtualization technologies to decrease latency, Docker doesn't seem to be focusing on this part. In fact, although Docker provides us with a simple way to deploy applications, the technologies it employs are not so latency-friendly. Like what has been mentioned in IBM's technical report ([10], Felter W., 2015: 171.), Docker containers take the Linux bridge as the method of network isolation. However, it shows that Docker containers even perform worse in transmission throughput and also have a longer network latency compared to KVM ([19], Kivity A. 2007: 225.). Actually, all technologies used by Docker are not new ones. Most of them have already existed since the year of 2007, and the concept of container also occurs at that time ([31], Soltesz S., 2007: 275.). Docker container is just a combination of these simple technologies. With the concept of Docker images and the emergence of Docker Hub, Docker quickly wins the eyes of system deployers. However, since most of these technologies are provided by old versions of Linux Kernel and they focus on resource isolation instead of latency, it will take Docker a long time to find ways to replace those inefficient technologies and thus decreasing latency lost.

Unlike Google or Facebook, which has dedicated data centers for their services, most small companies cannot afford the cost of hardware and the following maintenance. They can only deploy services on cloud centers like Amazon EC2 ([3], Shankar S., 2009.) and Microsoft Azure ([5], Copeland M., 2015: 27.). As we have mentioned above, these cloud centers use virtual machines to provide hardware virtualization and have a significant performance cost compared to bare metal. The occurrence of Docker thus providing another choice for these customers. Many cloud center service suppliers

provide container services in recent years. To simplify the deployment of applications, these small companies are considering to use Docker cloud. Since the additional layer of virtual machine brings about significant performance lost ([16], Huber N., 2011:563.) and is part of the source reason of long latency, it is very important for them to know the trade off between the convenience and latency performance degradation of using Docker to deploy latency-sensitive applications.

Previous researches mainly focus on the throughput of CPU, memory and I/O. Some of them talks about memory footprint and the latency brought about by Docker network bridge and methods to shorten this latency. However, these methods are not suitable for public cloud. This paper is intended to solve the problem from the customers' perspective. Although customers can not the change the services provide by cloud service suppliers, they have the choice to choose their start up configurations and the policies to build their services. We focus on the effect of Docker containers on latency with respect to various configurations. We analyze the effect of Docker container configurations to web service situation. This analysis provides customers with the potential latency cost of Docker containers and helps them to build services with the awareness of these possible degradation.

1.2 Related works

The appearance of Docker is in the year of 2012. However, the history of Linux containers is more than just several years. In the year of 2001, as an initial implementation of "virtual private servers", Linux-VServer project came into existence. However, it has never been merged to the mainstream Linux operating systems. There are other Linux containers like OpenVZ, which is mainly used to host web applications, that also doesn't share a position in the mainstream Linux. Finally, in the year of 2007, as many features including namespaces and chroot are added to Linux kernel, Linux Container (LXC) was finally added to the mainstream Linux and becomes the most widely used containers since then.

Several institutions and researchers have published related performance evaluation work on Docker. Most of them focus on throughput, while a few are concerned with latency. Researchers from IBM ([10], Felter W., 2015: 171.) use KVM as a representative hypervisor and Docker as a representative container and compare the performance of bare metal, virtual machine and container. They use various workloads to stress CPU, memory, and I/O resources. They have found that containers overwhelm virtual machines in almost every case concerning throughput. After these workloads, the research also shows experiments on some real world applications including MySQL and Redis Cache. Both of these real world applications exhibit a better performance for Docker containers than virtual machines. The report also reveals that the startup time of KVM is 50x slower than that of Docker containers. Kavita ([2], Agarwal K., 2015: 8.) tries to increase the number of containers on a host machine with the same kind of workload. He finds that the overall density of containers on a machine is highly dependent on the most demanded resource. He also concludes that virtual machines have significantly higher overheads than containers concerning memory footprint. He uses Kernel Same Page Merging (KSM), a memory de-duplication technology, and finds a 60 times opportunities to lower the memory cost of a virtual machine compared to containers. Canonical does a similar work as Kavita comparing LXD and KVM. All these virtual machines are running Ubuntu 14.04 operating system. Experiment measurements reveals that on a host machine containers have 14.5x higher density than virtual machines. The density bound is mainly caused by memory limitation. Also, the work shows a 57% reduction in network latency than virtual machines. But it doesn't show that the LXD is using Linux bridge technology.

Eder ([9], Eder J., 2015.) did a very simple work using kernel bypass ([23], Liu J., 2006: 29.). He concluded that applications running in a container does not have an obvious impact on its network latency performance. He uses *OpenOnload* together with *netperf* to realize the bypass. The results are shown and compared with their average, mean and 99th-percentile round trip latency. From that report, he concludes that there

are almost no performance degradation using Docker containers. However, this test is only suitable for private Docker cloud rather than public Docker cloud. This is because kernel bypass requires direct access to *Network Interface Card* (NIC), which has potential security problems in public cloud since one can modify the content of other containers as long as they want. However, in a private Docker cloud, kernel bypass can be a very good choice. Conventionally, once a packet is sent, it has to go through user space, kernel space and finally arrives at the NIC. With kernel bypass, the packet can be directly sent from user space to NIC, which saves some time.

On the other hand, IBM's report ([10], Felter W., 2015: 171.) shows that Docker container has a significant impact on overall network performance. The report uses *nuttcp* to measure throughput and also *netperf* to gauge latency. The report shows that there is over 80% degradation on network round trip latency and also consumes more CPU cycles transferring a single byte using Docker containers than natively. So why there exists such a big difference between Eder's work and IBM's report? The key point lies in the fact that one uses kernel bypass while the other doesn't. Using kernel bypass in a Docker container leads to shorter latency than Docker bridge or Docker host and even faster than bare metal without kernel bypass. From the public cloud's perspective, since it is not allowed for customers to use kernel bypass due to security reasons ([8], Dua R., 2014: 610.), IBM's work is more valuable in this case. However, IBM's report only uses a single group of comparison. It doesn't incorporate more comparison groups to further develop the relationship between round trip latency and other variables like the size of each packet transmitted.

There are also works showing that containers don't have a significant performance lost concerning network performance compared to bare metal. Xavier ([38], Xavier M. G., 2013: 233.) uses Xen as an representative of virtual machines and compares its performance to various kinds of containers including LXC, OpenVZ, and VServer. He presses these technologies with various kinds of well-known benchmarks and draws to the conclusion that containers outperform virtual machines in every high performance

cases. However, different from other researches, his work doesn't show a high performance lost in I/O cases. This is because these old-type containers don't employ technologies including AUFS or Linux bridge. In our work, we find that these two technologies are key reasons to the latency lost in Docker containers.

1.3 Contributions

This thesis makes the following contributions:

First and foremost, we investigate the latency slowdown caused by CPU configurations. We explore and compare two kinds of configurations, the first one is CPU shares and the second one is CPU quota. We find that CPU shares almost has no impact on the performance lost while it cannot limit the CPU usage of a container when only one container is running on a CPU. On the other hand, CPU quota can successfully limit the CPU usage of a container, but it has the potential to lead to a rather long tail latency. Besides, when running more than one container on a single CPU, containers may interfere with each other. However, the interference can be positive or negative based on the CPU use rate.

Secondly, we build a research platform to evaluate the network latency performance of Docker containers. The platform employs a client-server architecture. The server is hosted in a Docker container and we measure the round trip latency of a client request. We choose two situations, the first is server sending data and the second is server receiving data. We compare two configurations: using Docker host and Linux bridge. We draw the conclusion that containers do have some impact on the performance lost using Linux bridge compared to directly using the host machine's port. However, this performance is not as exaggerated as described in IBM's report that Linux bridge causes 80% performance lost compared to bare metal. In fact, this is more like a fix-length performance degradation. The smaller the transmitted message is, the more relatively significant the performance slowdown can be.

Thirdly, we analyze the latency impact of Docker using AUFS to do file operations.

We find that Docker containers have no impact on performance when writing an existing opened file. However, when it comes to operations relevant to the file system instead of a single file, situation changes. Operations like opening a file would lead to extra latency due to locating the file in multiple AUFS layers and the extra cost of creating the copy-on-write layer. When listing a directory which has many hidden files in low layers, the hidden files will also be scanned instead of just the superficial ones. The total scanning time is linear to the sum of the number of hidden and superficial files.

1.4 Organization of This Paper

The following sections are organized as follows: Section 2 introduces some background information about Docker related technologies. Section 3 carries out experiments and gives analysis about their affects to service latency. We talk about conclusions in Section 4 and discuss future works in Section 5.

Chapter Two Background

2.1 Docker

Docker, an open-source advanced container implemented by dotCloud, is making a huge impact on the field of cloud computing. Docker wraps the whole runtime environment into the unit of Docker containers to divide and allocate resources. It is a platform designed for developers and system administrators to build, ship and release distributed applications. It is also a cross-platform, portable and easy of use container solution. Docker is implemented in Go language and its source code is hosted on Github. Docker provides developers with a fast and automatic way to deploy applications. It incorporates many operating system-level Linux kernel technologies like namespaces and control groups to provide resource isolation, resource limitation and security.

2.1.1 Containers and virtual machines

Many people are familiar with virtual machines. In our everyday life, we might run Ubuntu Linux on Windows PC using VMware or play games in a Windows virtual machine using Parallel Desktop on Mac OS. These are all hardware-level virtual machines ([19], Kivity A. 2007: 225.). They use software to simulate the instructions that are used by the operating system running in the virtual machine as if they are just operating on the bare metal. Hardware-level virtual machines like VMWare and Parallel Desktop are more used on personal PCs, while Xen and KVM are more used on servers and public/private clouds.

On the other side is the container-based virtualization, which is also called operating-system-level virtualization ([31], Soltesz S., 2007: 275.). OS-level virtualization's coming into people's eyesight is because of the *chroot* mechanism introduced in Unix-like operating systems. Chroot was traditionally used to run multiple services in a multi-

user environment and leave each one not affecting each other's running applications on the same machine. Began from chroot, a lot of prototype containers were implemented like the famous LXC, FreeBSD jail and OpenVZ. Although Docker became famous in recent years, container is not a new concept since the appearance of container technologies mentioned above can be traced back to the year of 2007. It is the mature of cloud environment, the spring of web applications, and also the completeness of Docker environment that makes Docker so popular. Unlike hardware-level virtual machines, OS-level virtual machines use the host operating system, but each container only has access to its own files. A container contains all the files it needs to run a program, it has its own libraries, **/boot** directory, **/usr** directory, **/home** directory and so on. The whole running container can even only have a single file as long as the binary program you want to run has no dependency. Also, programs running inside a container cannot see processes outside the container, including those running directly on the native host and also applications running in other containers. This is implemented using the Linux namespace mechanism. Containers also use chroot mechanism to limit resources like CPU usage, memory usage and I/O usage.

IBM's report ([10], Felter W., 2015: 171.) shows that in almost all cases (except for network latency), Docker containers performs better than virtual machines and is very competitive to native. This is because virtual machines have to use software to simulate hardware. It can cost several instructions to simulate a single instruction using software, thus dramatically slowdown the performance. While on the other hand, container processes are just running using the host operating systems, except for the fact that they are isolated and resource limited. Containers also exhibits a much shorter start time than virtual machines, while most virtual machines use tens of seconds or even several minutes to start, it only takes a container several seconds or even less than a second. It also takes much less memory footprint to run processes than virtual machines because containers don't need those extra files to start a whole operating system.

Although containers do bring us about better overall performance than virtual machines, there also exist a lot of problems to be solved. The most important part is security. As I have mentioned above, containers use the host operating system, which means that any security threats on the host operating system can be made use of to attack the host and other containers running on the same machine. The host operating system can see everything running inside a container, and the content is not private for users. Once a container process successfully takes the administration of the host operating system, it can operate all other container processes at its will.

Another current limitation for containers is that most of them can only run on Linux. Since running a process in container requires the host operating system to be Linux, running Linux processes on Windows systems may not be practical. Also, since containers do not have the authority to modify its host operating system, it is not possible to load kernel modules dynamically, thus limiting its ability.

2.1.2 Resource Isolation Using Namespace

The mechanism of Linux Namespaces is a way of isolating resources ([27], Merkel D., 2014: 2.). System resources like PID, IPC, network, etc. are no longer global, but belongs to a particular namespace. The resource in each namespace are transparent to other namespaces. To create a new namespace, we only need to specific the corresponding flag when calling *clone* function. LXC and Docker libcontainer use this feature to realize resource isolation. Processes in different containers belong to different containers. They are transparent to each other and will not interfere with each other.

Traditionally, many resources are organized globally in Linux and many other Unix-like systems. Although it allows to allocate some authorities to certain IDs and the root UID 0 user is allowed to do almost everything, other user IDs will be limited. For example, user UID n is not able to kill the processes of UID m ($m \neq n$) if m is not a descendant of n, but they are allowed to see each other. Sometimes, we might want processes and users not be able to see each other, here Linux namespace comes.

Using KVM cannot always allocate resources properly. Each user needs a separate kernel and a full suit of user applications. Linux namespace allows to run a single kernel on a physical machine and all resources are abstracted by namespace. This allows us to put a set of processes into containers and each container is isolated from others. Some certain shares are also allowed among containers to lower the isolation of them.

Take the PID namespace as an example. Suppose globally there are 30 processes running on the machine, processes 11 to 20 belong to container A and processes 21 to 30 belong to container B. Here, container A sees processes 11 to 20 as processes 1 to 10, while container B sees processes 21 to 30 as processes 1 to 10. The root user can see all processes 1 to 30 as their global PIDs. A can not see processes in B and it is not able to see processes 1 to 10 owned by the root user, so cannot B. Thus the PID isolation is implemented.

There are two ways to create new namespaces. When using *fork* or *clone* system calls to create new processes, there are certain options to choose to share the same namespace with parent or create a new namespace. We can also use *unshare* system call to separate namespace from one's parent process.

2.1.3 Resource Limitation Using CGroups

CGroups is the abbreviation of Control Groups ([27], Merkel D., 2014: 2.). It is a mechanism provided by Linux kernel that can limit, record and isolate the resources used by process groups. It was introduced to Linux Kernel 2.6.24 in 2007. CGroups can let you define groups for processes in the system and allocate resources to them, including CPU time, system memory, network bandwidth and a combination of them. You can monitor these CGroups, refuse the access of resources and even dynamically allocate resources in running groups.

CGroups mainly provide the following functions:

Resource limitation: Limit the resource usage, like memory usage upper bound and cache limitation of file system.

Prioritization: Control the priorities of processes when dealing with resources.

Accounting: Monitoring the processes, mainly used to account.

Control: Suspend, resume or run processes.

There are many concepts in CGroups. *Task* refers to a process in the system. *Control group* is a group of processes divided by a certain standard. The basic unit of CGroups are implemented by control groups. A process can either be added to a control group, or can be moved from one control group to another. The processes in a control group can use resources allocated to this group. They are also under the limitation of the resources of this certain group. Control groups can be organized as a *hierarchical* format, i.e., a control group tree. The son node of a control group inherits and is limited by its parent node's attributes. There are also *subsystem*s in CGroups, and one subsystem is a resource controller. For example, the CPU subsystem is used to control CPU allocation.

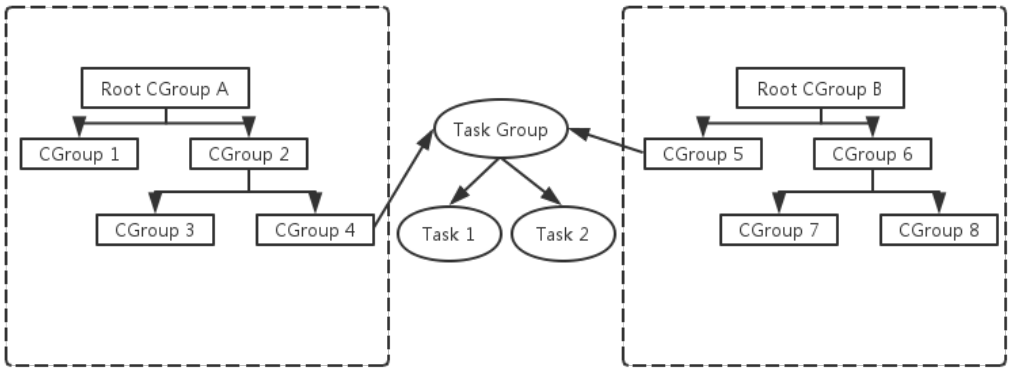


Figure 2.1 Control groups

When creating new control groups, there exists a lot of limitations. Each time a new hierarchy is created, all tasks in this subsystem are in the default control group of that hierarchy, i.e., root group. A subsystem can be at most attached to one hierarchy and a hierarchy can be attached many subsystems. A task can be the members of many control groups, but none of these groups can be in the same hierarchy. A newly forked task will automatically become its parent's control group member. It can then be moved

to different control groups on demand. As is shown in Figure 2.1.

2.2 Related Linux Technologies

2.2.1 Linux Bridge & Veth Pair

Bridge mode is the default network setting of Docker. When using this mode, each container will be allocated a network namespace and separate IP. When we start Docker daemon, it will create a virtual network bridge called *docker0* on the host machine. All Docker containers created on this machine will be connected to the virtual network bridge. *Virtual network bridge* works like a physical switch, thus all containers on the host machine are connected in a two-layer network through a switch. In this network, each container should have an IP address. Docker will choose a private IP different from the host IP and sub net defined in RFC1918 and allocate it to docker0, and each container will choose an unused IP from this sub net. For example, Docker may choose the 172.17.0.0/16 subnet and it will allocate 172.17.42.1/16 to docker0 bridge. You can use *ifconfig* to monitor docker0 since its working as a virtual network interface card. Assume that the IP address of host machine is 10.10.101.105/24, the topology of a single machine environment is shown in Figure 2.2.

To create the above network settings, Docker will first create a pair of virtual network called *veth pair*. Veth always occurs in pairs. They form into a data tunnel and the information comes into one end will go out to the other end. So veth endpoints are usually used to connect two network devices. Docker names one of the veth pair as *eth0* and put it inside the newly created container. The other end is put on the host machine, named like *veth23f6*. This network device will also be added to docker0 network bridge. It can simply check by using the **brctl show** command. Docker will then choose an IP from the subnet and allocate it to the newly created container.

In bridge mode, all containers connecting to the same network bridge can communicate with each other. Containers can also communicate with the outside world.

This is implemented by modifying the *Iptable*. Iptable will help do the transmission that all packets sent to eth0 of Docker container will be sent out to docker0 first and then the outside world. All packets sent to docker0 will first decide which container it belongs to and finally sent to the corresponding eth0 of the Docker container.

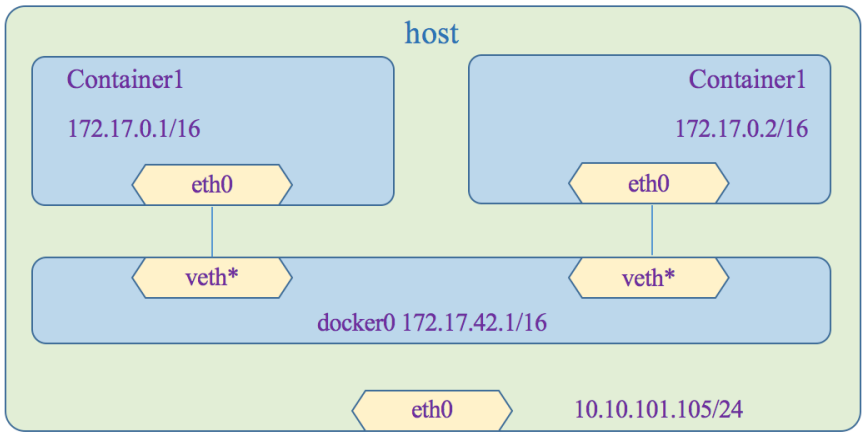


Figure 2.2 Linux bridge

2.2.2 Scheduler

The *scheduler* class was introduced in Linux version 2.2. It has implemented real-time, non-preemptive and non-real-time task scheduling policies. In Linux version 2.4, a relatively simple scheduler was implemented and it's running in O(N) time complexity. Earlier Linux 2.6 scheduler is called O(1) scheduler. It aims to solve the problem of O(N) scheduler's having to iterate all the task queue to decide the next task and it is more efficient. O(1) scheduler is easy to expand and more effective. However, the implementation of O(1) scheduler is very heavy and need huge amounts of code. It is hard to understand and thus difficult to manage.

In Linux Kernel version 2.6.21, the scheduler implemented by Kolivas, called *Complete Fair Scheduler* (CFS) was incorporated ([1], Aas J., 20015: 1). Its main idea is to provide the fairness in terms of providing CPU time to different tasks. When CPU time allocated to a certain task loses balance, it should be allocated enough time to be scheduled on the CPU.

To realize fairness, CFS maintain a time quantity in a place called *virtual runtime*.

The less virtual runtime is, the less time a task has been allowed to run on CPU, which means that it needs more time to be scheduler to CPU. CFS also includes the sleep fairness concept to make sure that those not running tasks (e.g. waiting I/O) will be allocated a certain amount of CPU time when they finally need.

However, different from former Linux schedulers, CFS is not maintaining tasks in a running queue. Instead, it maintains a *red-black tree* (RB-tree) based on time priority. RB-tree has many interesting and useful attributes. First, it is self-balance, no route on the tree is longer than twice of other trees'. Secondly, it runs at a O(log n) speed (n is the number of tree nodes), which means that you can insert and delete tasks fast.

Tasks are saved in the RB-tree according to their time. The more time a task needs to be scheduled on a CPU, the more left side it is on the RB-tree. For the sake of fairness, the scheduler will choose the left corner task on the RB-tree to be scheduled next. Tasks will be then added its real run time. At the same time, since its virtual runtime has been increased, it is moved several steps right on the RB-tree. Thus, all tasks on the RB-tree is chasing each other and they form a dynamic balance on CPU scheduling time.

Aside from CFS, which mainly aims at interactivity for desktop users, there are real time requests for server uses. Linux has implemented two of them. The first one is *SCHED_FIFO*. It implements a first-in-first-out algorithm. Once a task starts to be executed, it will continue to go on until it gives up CPU at its will, blocked, or preempted by higher priority real-time tasks. When two tasks are of the same priority, they are scheduled according to the first-in-first-out principle. The other algorithm is SCHED_RR, and it has the concept of time slices. Process with the same priority once uses up its time slice, it will give way to the next task in the queue, and it is then assigned to the tail of the waiting queue.

2.2.3 Tail Latency

Nowadays, not every service is so simple like a client and server mode. If a certain client sent a request, the server side might just be an interface. The real work is handled

by the huge data center in the background. For example, in a Google search operation, the request words might be processed by hundreds of machines doing Map-Reduce work. Each machine would have to give their results to a central reduce machine. Thus, the reduce machine has to wait all the works to be done before it can finally move to the next step. In this case, once a simple map work is done over one second and all other works are done within one microsecond, the overall time consumption of this work would be limited to the slowest one.

The concept of tail latency was first proposed by Google ([7], Dean J., 2008: 107.). It has brought great attention since then. To observe the tail performance of Docker in this paper, we use another measurement: 99% performance, or the *99^{th}-percentile*. This is used to show the tail latency performance of our work. Since Docker containers are often used in the cloud environment, we add this to solid our research.

Chapter Three Latency Characterization

In this section, we first describe our experimental methodology and then evaluate various Docker configurations including CPU, network and file system.

3.1 Apache Thrift

Thrift was first started from the famous company Facebook ([30], Slee M., 2007: 8.). In the year of 2007, it was submitted to Apache and became an open source project. At that time, Thrift was used to solve the problem in Facebook that various systems needed to transmit huge amount of data while the language environment are different. Thrift can support many kinds of languages like C++, C#, Cocoa, Erlang, Haskell, Java, Perl, PHP, Python, Ruby. Thrift can work as a high performance communication middleware among different languages. It supports the serialization and many kinds of RPC services. Thrift is suitable for building large scale data communication and storage tools. In large systems, the inner data communication has obvious advantage over JSON and XML.

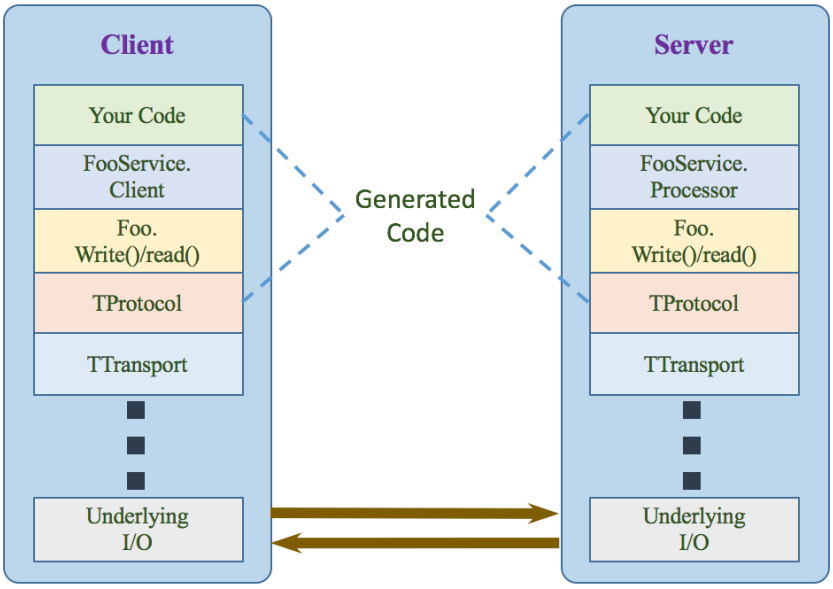


Figure 3.1 Mechanism of Apache Thrift

Thrift incorporates a client and server architecture. It has its inner Transport Protocol *TProtocal* and transport standard *TTransports*. The mechanism of Thrift is described as shown in Figure 3.1.

Thrift enables you to choose the transport protocol between client and server. The transport protocols are generally divided into text and binary. The latter one is with better performance. Compared to XML and JSON, its packet size is very small. It also has affinity to high concurrency, large data and multi language environment.

3.2 Experimental Methodology

Since we focus on the latency of real-time services, we incorporate a client-server model which tests the round trip latency for several operations. We conduct our experiment on two HP MicroServer nodes (Intel Xeon E3-1220L processor, 2.3GHz), each with 4GB installed RAM. We employ Apache Thrift to let client side use RPC calls to call the server side and server then return the result. Python is the experiment language. For each call, we measure the latency based on its start time and end time.

3.3 Containerizing and resource limitation

There are a lot of container resource limitation parameters:

cpu-period: This means the period for the kernel to schedule the Docker process. It is used together with the **cpu-quota** parameter. The unit of these two parameters are both microseconds. When these parameters are set, it means that the processes can not use longer than **cpu-quota** time during each **cpu-period** time duration. Once the process reaches its time slice, it will be cut off and not able to use the remaining time slice.

cpu-quota: As is mentioned above, **cpu-quota** together with **cpu-period** has a limitation that their minimum value should be no less than 1000 us. When **cpu-quota** is larger than **cpu-period**, it means that the container can use more than one CPU core resources.

cpu-shares: When this parameter is set, assume that two containers have different shares and running on the same core. Container A has a share of 1024 and container B has a share of 512. If both containers are CPU intensive, which means that they take almost all the time to do CPU calculation. The CPU time used by container A and container B should be at a ratio of 1024: 512, which is 2: 1.

net: There are four choices for this parameter. **bridge** uses the network namespace mechanism, which means the container uses Linux bridge to communicate to the outside world. It has its own IP address different from the host machine. Socket used by the container are mapped to a socket on the host machine. Which is just like NAT mechanism. **host** means the container directly use the host network port and there is no network isolation. **container** let several containers running on a single machine share a same network namespace, which means they have the same IP address and contend for ports. **none** doesn't mean the container has no network communication with the outside world. It just leaves the user to control the network settings.

p a: b: This is used together with **net=bridge** (the default network setting). Which means that the host's port **a** is mapped to the inside container port **b**, and it's acting like the NAT. All requests come to host's port **a** will be sent to container's port **b**, and all messages sent from container's port **b** will be transferred to host's port **a**.

v a: b: With this option, we can map the host's files or directories to container's file system. So we can bypass the AUFS mechanism and directly access the host files. For example, if we use **-v /home/username:/home**, when we enter the Docker container's **/home** path, we can see all the files in the host's **/home/username** directory. This just acts like the traditional Linux **mount** command, and we can have access to the files and directories in **/home/username** directory. If we visit files in a container that are not mounted from host, we might open a new file which is copied from the original file and all the operations are done in this new file.

In all cases, server is running in a Docker container. To narrow down the experiment interference, we first let **CPU #3** (totally 4 CPUs, 0 - 3) excluded from the CPU

auto scheduling mechanism, which means that only our container can run on this CPU and all other applications have no access to it. This is implemented using the CPU affinity mechanism ([24], Love R., 2003: 8.) and we add **isolcpus=3** Linux kernel boot option when starting the server host machine. We also disable all interrupts to happen on **CPU #3**, thus making sure no additional context switch ([21], Li C., 2007: 2) would happen. Each time we run the server container, we have to use the **cpuset-cpus="3"** to force our container run on the specific CPU. **cpuset-cpus** argument is very similar to **taskset -c** command since they can both assign a task on a dedicated CPU core. The difference is that **cpuset-cpus** can only be applied to the whole container while **taskset -c** can be applied to any process. You can even run **taskset -c** in a container to let a process in container run on a certain CPU core.

3.4 CPU Configurations

In this experiment, we let the client run natively on a machine and the server run in a Docker container on the other machine. Server container uses option **net=host** to expose all host's ports to the container. The client directly calls the server, without extra information like parameters sent or return values received. In each experiment, client continuously sends 1,000,000 requests to the server and then notes down the round trip latency.

3.4.1 Baseline: Native Platform

In our baseline case, we run the server process natively. To make use of CPU affinity, we use **taskset -c 3** to let our process run on the target **CPU #3**. The experiment is repeated for 10 times. Each time 1,000,000 requests are transmitted between client and server. The CDF ([15], Hopper T., 2014.) result is shown as the red line in Figure 3.2. The mean, median and 99th-percentile position of the measurements are listed in Table 3.1. Most of the latencies are between 200 and 300 microseconds, and the average and

median measurements are about 240 microseconds. However, there are still 1% latencies beyond 278 us and these long latencies would be very common in the real production world. This phenomenon might be caused by the interference of background processes, non-FIFO scheduling, multicore scheduling ([22], Li J., 2014: 1), and interference from other virtual machines or containers in the cloud environment ([39], Xu Y. 2013: 329).

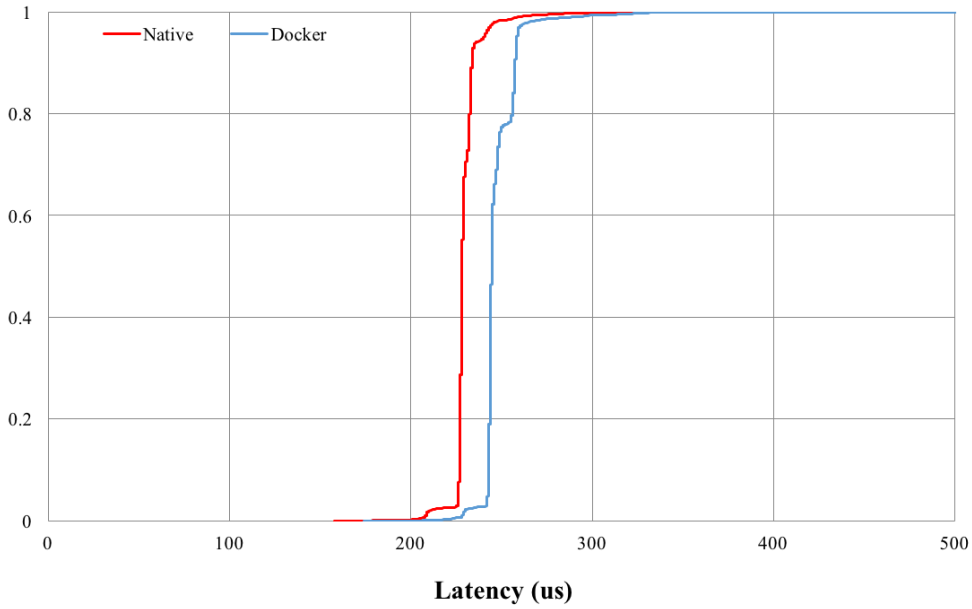


Figure 3.2 The CDF of latency using bare metal and Docker container

Table 3.1 Latency measurements of bare metal and Docker container

	mean(us)	median(us)	p99(us)
Bare metal	240.7	241.0	278.0
Docker container	255.2	255.0	295.0

3.4.2 Case 1: Using CPU Shares

We run the server process with the **cpuset-cpus="3"** setting to realize CPU affinity and the **cpu-shares=1024** as a default setting in CFS scheduler. We run the test for 10 times. Each time 1,000,000 requests are transmitted between client and server. The CDF result

is shown as the blue line in Figure 3.2, and the mean, median and 99th-percentile position of the measurements are also listed in Table 3.1.

From the above two test cases, we observe that when using Docker, the CPU latency almost shares the same CDF curve as using bare metal, except a little bias showing an additional fixed amount cost for CPU. Comparing both from the 99th-percentile column in Table 3.1 and the CDF curves in Figure 3.2, Docker container does not have a significant impact on the tail latency performance when using CPU shares. Just like mentioned in the report of IBM, Docker containers do have impact on CPU performance. However, the degradation is very low, 4% in IBM's report about throughput and about 6% about the mean, median, and 99th-percentile performance in our research. We conclude that when running CPU-intensive applications in a Docker container, the performance effect would be very small. Unlike VMs, which use hardware-level virtualization technology, Docker container's instructions do not need to be emulated by VMM. However, Docker containers share the same Linux kernel and use the same instructions as the host machine. An x86 instruction needs to be translated to several instructions to run on an ARM CPU using VMs. With the help of equation 3-1 ([26], Menascé D A., 2005: 407), we can do a rough calculation of the virtualization slowdown, where f_p stands for the fraction of privileged instructions executed by a VM and N_e stands for the average number of instructions required by the VMM to emulate a privileged instruction. The reason why Docker containers bring about a slightly slowdown is because when performing CPU isolation and limitation, the kernel needs to first check the namespace of the running process, thus the additional instructions would cause the extra latency.

$$S_v = f_p \times N_e + (1 - f_p) \tag{3-1}$$

3.4.3 Case 2: Using CPU Quota

Apart from **cpu-shares**, there exist other parameters which limit the resource usage of CPU. **cpu-period** means the period for the processes in the Docker container to be scheduled on the CPU. It is often used together with **cpu-quota** parameter. The unit of these two parameters are both microseconds. When these parameters are set, it means that processes in the container can use no more than **cpu-quota** time during each **cpu-period** time duration. To test whether these two parameters would have the same side effect as **cpu-shares** when only one container is assigned to a CPU core, it can take all the cycles of that CPU, we carry out the following experiment:

We first fix **cpu-period=10000** and vary the value of **cpu-quota** to see the relationship between these two parameters. We choose values 1,000, 1,500, 2,000, 2,500, 3,000, 4,000, 5,000, 7,000, 10,000 for **cpu-quota** and observe the results. These parameters are chosen because the minimum value of both these parameters are 1,000, and the magnitude gap is also set small. For each test, it is performed for 1,000,000 requests. We measure the mean, median, and 99th-percentile of the latencies. We also take the number of requests whose latency is greater than 1,000 us, the minimum time slice, into consideration. These results are shown in Table 3.2.

Table 3.2 Measurements of latency using 'cpu-period' and 'cpu-quota'

quota(us)	mean(us)	median(us)	p99(us)	# > 1,000us	# × quota
1000	1035.4	307.0	7556.0	104996	104996000
1500	579.5	258.0	5816.0	68221	102331500
2000	489.0	281.0	4765.0	50187	100374000
2500	361.7	260.0	3232.0	37334	93335000
3000	291.9	259.0	1741.0	14273	42819000
4000	258.0	253.0	331.0	39	156000
5000	263.2	255.0	341.0	0	0
7000	256.7	250.0	341.0	0	0
10000	262.6	255.0	333.0	0	0

From Table 3.2, we observe that latency increases incredibly when CPU quota only

counts for a small ratio of the total CPU period. From 1,000 to 4,000, all mean, median and 99th-percentile are decreasing and so is the number of test cases whose latency is greater than 1,000 us. Figure 3.3 also shows the relationship between **cpu-quota** and the number of requests whose latency is greater than 1,000 us. From this picture, we can see that the number first drops fast and then slowly as quota increases. When quota reaches over 3,000 us, latency suddenly drops fast and finally goes to about zero at 4,000.

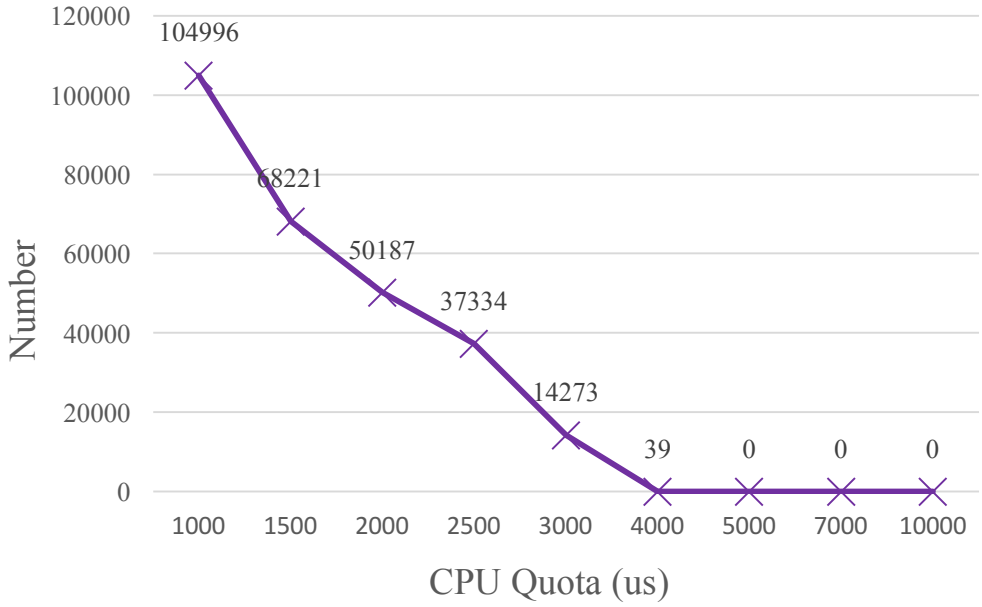


Figure 3.3 Number of requests' latency beyond 1 ms

Unlike using CPU share, once only a single process is using the CPU, it can take all the CPU resources, using CPU period together with CPU quota options has a force cut off when the CPU usage is over the limited number. Since the client is calling the server continuously, once a request has finished, another request immediately follows. If the server process uses up its quota during one period, it is sure to give up the CPU and wait until the next period comes. This can cause a very long tail latency in real time services, which is shown as a sudden rise in the 99th-percentile. Once the service is CPU-intensive or being visited quickly, it will add unwilling latency to the service, thus

reducing the overall performance.

To prove the above theory, we first compute the last column in Table 3.2. Assume we need in total time t_{cpu} to do all the computation, which means the total time the process is running on CPU. CPU quota is t_q, and CPU period is t_p. Total number of requests blocked by the CPU options n is computed as follows: $n = t_{cpu}/t_q$. Thus, total time t_{total} needed to compute all the requests is: $t_{total} = n \times t_p$. So once t_{cpu} is determined, we can see that $n \times t_q = t_{cpu}$ is also determined. In our experiment, we assume the CPU time cost for each request is $t_{request}$, and the total number of requests is r. So we see that $t_{cpu} = t_{request} \times r$ is determined, and $n \times t_q$ must be also determined, which is shown as the last column in our experiment. We can observe that for the case 1000, 1500, 2000, 2500, the products are around 100,000,000, which satisfies our formula. However, when t_q comes to over 3000, the product falls incredibly. This phenomenon occurs because at this time, t_q is greater than the overall CPU used. We use *htop* command to measure CPU usage and the CPU use rate of that CPU is around 30%. This is the reason why it suddenly falls at the 3000 point, which has $3000 \div 10000 = 30\%$, and then quickly goes to 0. We also see from Figure 3.3 that when it comes to over 4000, the mean, median and 99th-percentile are almost not affected, which means that when the CPU usage of the application is less than the ratio of quota to period, it has low impact on the latency performance.

3.4.4 CPU Interference

There is no meaning running a single process on a CPU core while at the same time limiting its available CPU resources. In the real world Docker cloud, if the CPU resources allocated to a container is very limited and less than one CPU core, sharing a single core among many containers can not be avoided. Once a latency sensitive container is allocated such configurations of resources, there is an urgent need to know the interference between the newly added container and the existing latency sensitive one.

In this experiment, server side is held in a container on **CPU #3** of the server host

machine. Server uses **net=host** to eliminate the interference of Linux bridge. Client side is also held on a dedicated CPU on the client machine, while not inside a container. The client continuously pings the server using Apache Thrift, while at the same time no additional data is transmitted between the two processes except for the necessary Apache Thrift overhead. Although the server container is authorized to use all the memory and network resources, since we are focusing on the CPU interference of Docker containers, CPU quota is limited. We fix CPU period 10000 us not changed among all the experiments. On the other hand, CPU quota is varying from 5000 us to 9000 us for the server container. At the same time, another container is running on the same CPU as the server container. This container is continuously running a matrix multiplication process. The matrix multiplication involves two 512 x 512 matrixes. In each iteration, we log down the execution time. The sum of the CPU quotas of the matrix container together with the latency container is equals to the CPU period container. For example, if the CPU quota of latency container is 9000 us, then that of matrix container is $10000 - 9000 = 1000\ us$. The reason why we choose the number over 5000 while less than 9000 is because this period makes sure that the latency container has over 30% of CPU, which is the maximum CPU consumption mentioned in the previous sections, while at the same time the two containers can interfere with each other. We log down the execution time of 100 iterations and the average measurement is shown in Figure 3.4. The green bar shows the result when the latency container is working together with the matrix multiplication container, while the gray bar represents that the latency container is not working. The mean, median and 99th-percentile measurements of the latency container are shown in Figure 3.5.

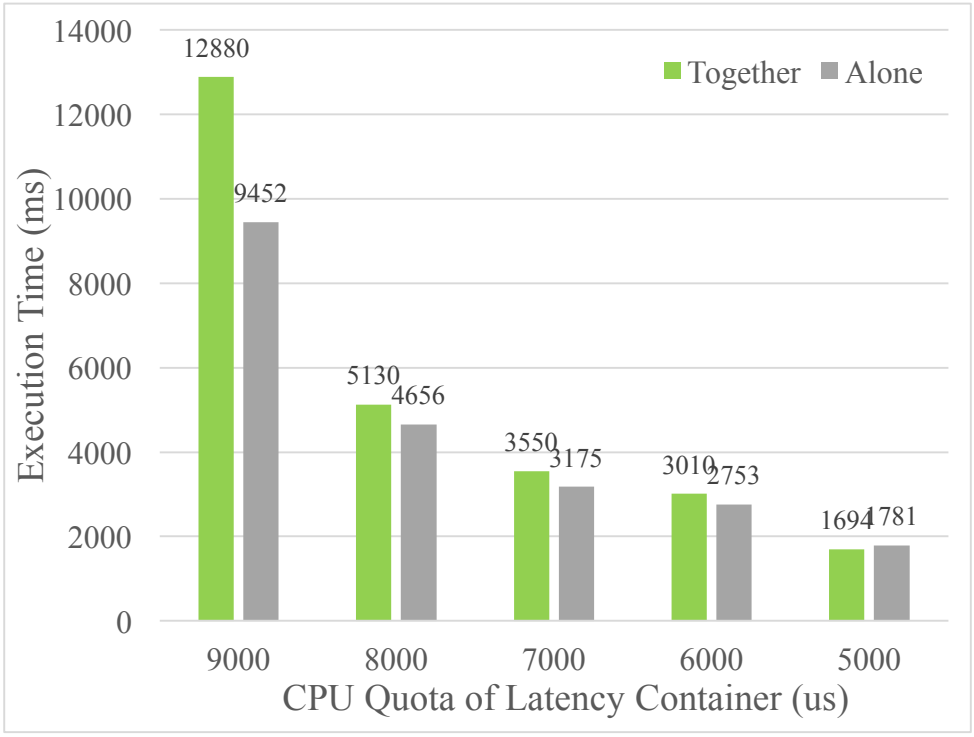


Figure 3.4 Execution time of matrix multiplication

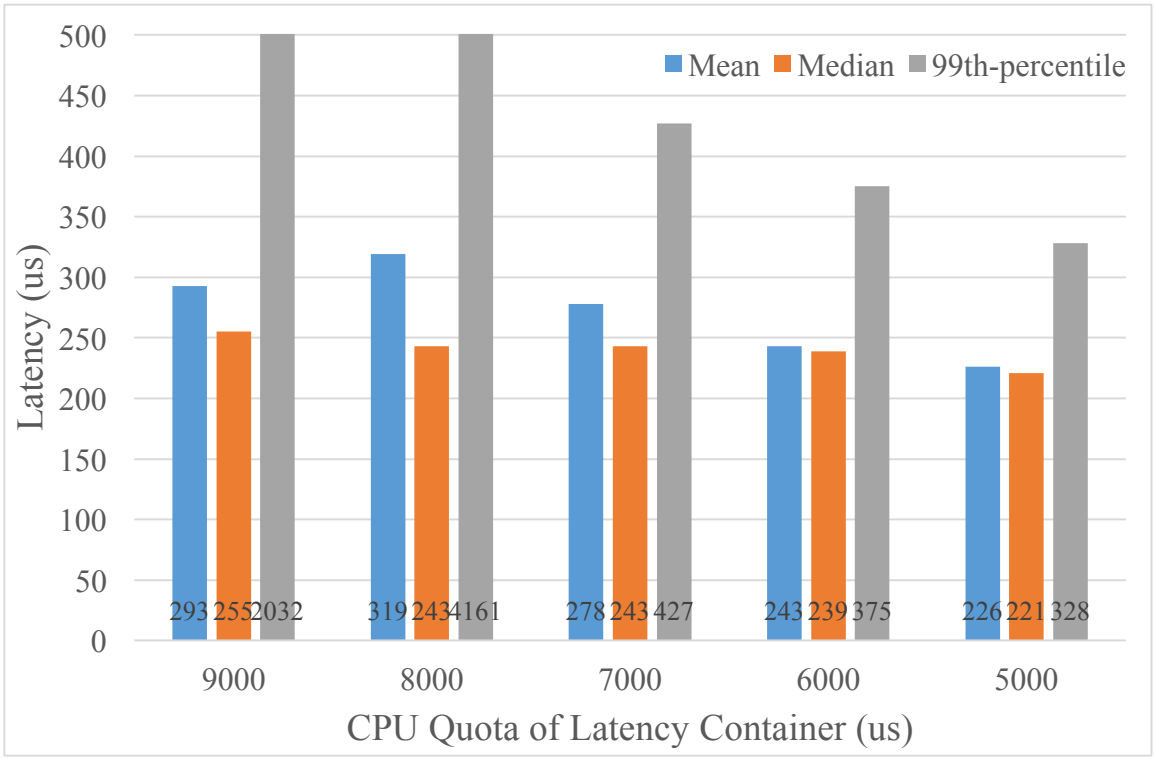


Figure 3.5 Mean, median, and 99th-percentile of the requests when two containers are running together

We can observe from Figure 3.4 that when the CPU quota is from 9000 to 6000, execution time of the matrix multiplication when running together with the latency container is much higher than running the matrix multiplication container alone. This is because when two containers are running together, the matrix multiplication container doesn't only face the need to do extra context switch caused by CPU limitation, but it is also unwillingly preempted by the latency container. These extra latencies lead to an increment in the overall execution time.

As we can notice from Figure 3.4, running a single container not always defeat running two containers together concerning the execution time. In the last column of Figure 3.4, to our surprise, the green bar is slightly shorter than the gray bar! There is no reason running two applications can have less context switch than only one application and two applications are sure to consume more CPU resources than a single one. So what factor leads to this strange phenomenon? Since matrix multiplication only involves memory access and CPU execution, there must be something changed in either of these two stages. However, the memory access position is not changing during all the process, CPU change should be the key reason. We finally located that with the increment of CPU workload, the CPU use rate is also increasing. Besides, the value of CPU frequency does not always keep the same. When workload is lower, CPU frequency will be lower to save energy consumption. We use the command **cat /proc/cpuinfo | grep "MHz"** to see the CPU frequency. From my observation, when the CPU workload is low, CPU is working at the frequency of 1,600 MHz on my computer, which stands for the situation where only the matrix multiplication container is running. However, when we run the two containers together, the increment of CPU workload brings about CPU frequency and it rises to an average of 1,955 MHz. Although the total required CPU cycles don't change, the increment of CPU frequency lead to de decrement of total execution time.

On the other hand, the latency container goes through the same situation. When the overall CPU is low, the increment in context switch lead to the performance degradation.

When CPU quota comes to 6000 and 5000, the performance of the latency container even becomes better than the original latency container without any interference as is shown in Table 3.1.

3.5 Network Isolation

In this experiment, server sends or receives various length data to or from the client. We choose message sizes 1KB, 10KB, 30KB, 50KB, 70KB and 100KB. All message sizes are chosen from *SPECWeb2009* ([33], 2009.) as the standard web message sizes. We test 1 million requests for each experiment.

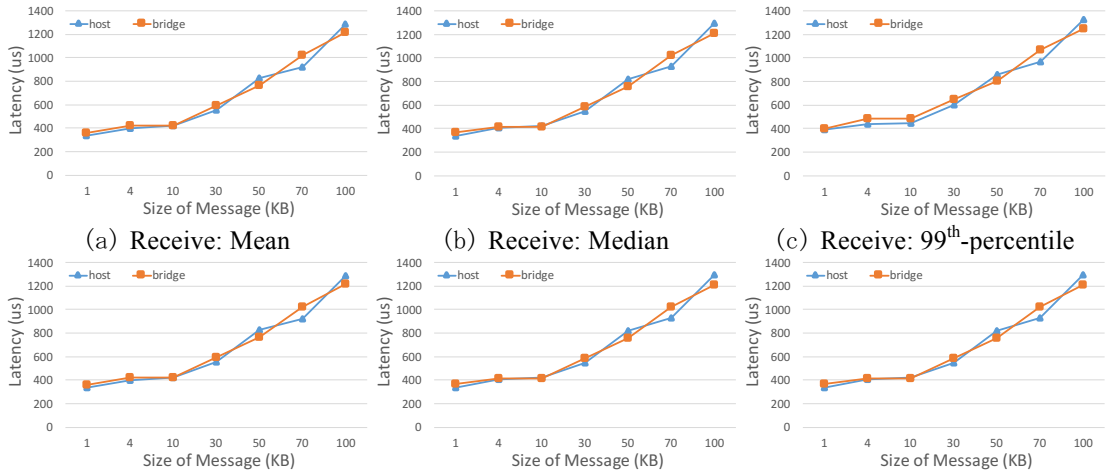
Sever is hosted in a Docker container on one machine and client is running natively on another machine. Both server and client are assigned to a dedicated CPU to reduce performance interference. We compare two Docker configurations, the first one is to use **net=host**, which means the container directly uses the host network port and the network isolation mechanism is not working. The second one is to use **p portA: portB,** which means container uses Linux bridge to communicate to the outside world. The **portA** used by the container is mapped to **portB** of the host machine, and it's working similar to the *NAT* mechanism ([36], Tsirtsis G., 2000).

3.5.1 Case 1: Server Receives Data

In this case, client sends data to server using RPC calls and pass data through string parameters. We compare the mean, median and 99th-percentile result between using **net=host** and **p portA: portB**. Results are shown in Figure 3.6(a-c).

3.5.2 Case 2: Server Sends Data

This time, client calls server with a single parameter indicating size and server returns the corresponding size string. Results similar to Section 3.5.1 are shown in Figure 3.6(d-f).



(a) Receive: Mean (b) Receive: Median (c) Receive: 99th-percentile

(d) Send: Mean (e) Send: Median (f) Send: 99th-percentile

Figure 3.6 Mean, median, and 99th-percentile latency when server receives and sends data.

3.5.3 Analysis

As is shown in both figures, most of latencies using **p portA: portB** are no more than 110% of latencies using **net=host** in each size group. In fact, when file size becomes much larger, the results are very close to each other, and the slowdown percentage is nearly zero. Some of the comparison groups even witness a phenomenon where Docker bridge outperforms Docker host. This is because many other factors could slightly change the latency, including CPU scheduling, preempting, network contention, etc. They are outside the scope of this study.

However, according to IBM, there is a 80% slowdown using Docker bridge ([10], Felter W., 2015: 171.). So why our experiment draws a totally different result from IBM's research? There are several reasons. First, from our observation of the experiment, the degradation is 'one-time'. It only adds a fixed latency to each trip. In our experiment, when the size of message is small, the impact of this fixed latency becomes large — nearly 10%. However, when message size is much larger, the impact of this latency is negligible, and the random vibration takes the dominance of the difference.

In IBM's report, it uses Docker on both server and client side, which means a single

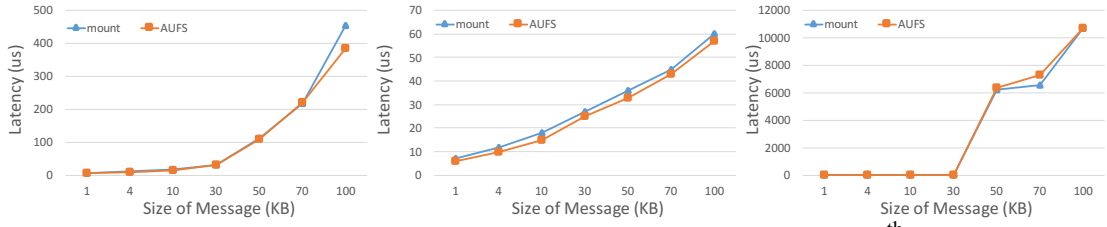
round trip message goes through bridges four times, each side two. While in our experiment, it only happens at server side and totally two times. IBM's report uses 100B and 200B message size, while messages used in our experiment are much larger. Also, we use python and Thrift, which add much extra cost to the CPU and network, so the baseline might be much larger than in IBM's report. The size 0 situation is over 200 us as shown in the CPU experiment section. While all latencies in IBM's report are less than 100us. So we can not simply describe the impact of Docker bridge on the network performance as a percentage slowdown. It is a fixed-length degradation.

To understand the fixed-length degradation, we dive into the source code of Linux bridge. Taking the receive stage as an example, when a message has arrived at the NIC, it is uploaded to the upper layer in the network stack. It first checks whether this message is used by more than one application or it is to be multicast. If not, it does no process including large amount of data copy. If so, the message is copied several times and then sent to each receiver, thus costing huge amount of extra latency. However, apparently in our experiment and IBM's experiment, the message is only used by a single application, without multicasting. Also, the message transmission is done by transferring a pointer to the message. So these extra functions will only cause a fixed extra time. This is why it affect so much in IBM's report while in ours slightly.

3.6 File Operations Using AUFS

Many applications like SQL server and Hadoop involve disk I/O operations. Disk operation latency of Docker container is thus another issue we should care about. In this experiment, only one application is hosted in a Docker container. The application continuously writes content of different sizes to a file. The length written each time are 1KB, 4KB, 10KB, 30KB, 50KB, 70kB, and 100KB. We compare two situations, the first one is to directly write messages inside the container, while the second is to write messages to a file outside and mounted to the container using **v** option. To make sure that the messages are written to the disk immediately instead of staying in the cache

area, each time we write a single message, we will use the **flush()** function to force the system to flush messages onto the disk. In each experiment group, the application continuously writes 10,000 times and we repeat this operation for 10 times. We only discuss about the write case instead of the read one to avoid the pre-read situation where blocks of files are read in advance. We note down the mean, median and 99th-percentile measurements. The results are shown in Figure 3.7.



(a) Mean (b) Median (c) 99th-percentile

Figure 3.7 Mean, median, and 99th-percentile file write latency using mount and AUFS.

From Figure 3.7, we can observe that there is almost no impact on performance using AUFS when writing files. This result is the same as the one drawn from IBM's disk I/O report. This phenomenon is explained as follows. When a file is open, due to the copy-on-write feature of AUFS, a new file is created to hide the original file and a pointer to this file is returned to the application. The following operations are just like the normal operations on a disk file, and there is no fix-length latency degradation as in the network case.

However, there are cases reported that operating on files involves a significant performance degradation. Considering the feature of AUFS, we guess that the latency lies in the operation of opening a file. To confirm our guess, we carry out another group of experiment where the application continuously open and close a file 1,000,000 times. We note down the time required to open the file and the comparison is shown in Figure 3.8.

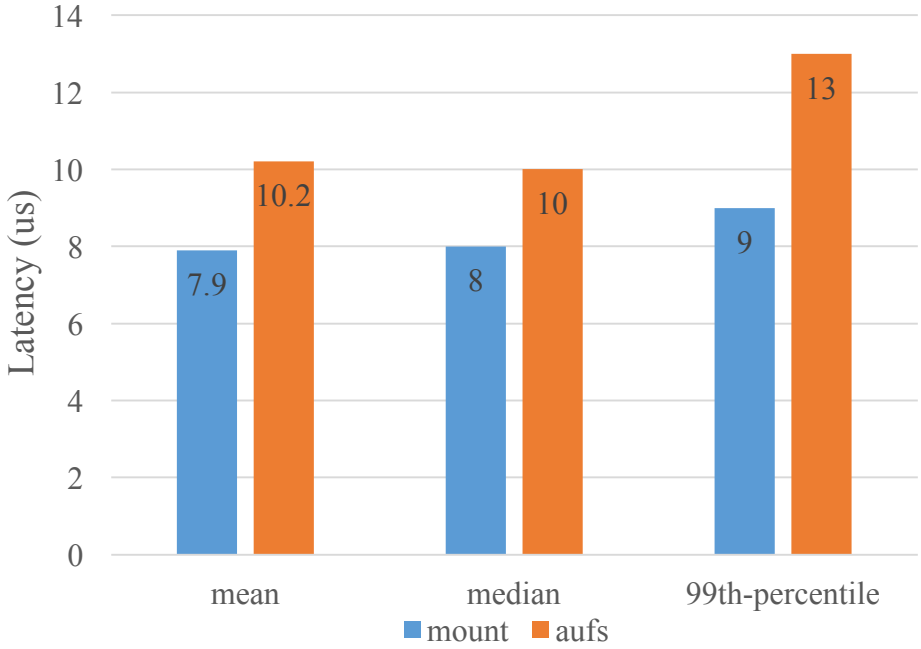


Figure 3.8 Latency Comparison between mount and AUFS when opening a file.

From Figure 3.8, we can find that there is a huge difference between using mount and AUFS concerning the open time of a file. This is because when using AUFS, since a Docker container is implemented combining several layers together, it must perform several additional functions to decide which layer the file is in. Sometimes it even has to copy another version of an existing file to perform the copy-on-write operation. These costs are huge compared to the time to open a file on the bare metal. Also, when first time writing to an existing file in the container, the larger the original file is, the longer the operation latency will be, which is caused by the cost of copy-on-write feature.

Building an image over another is a very convenient feature of Docker. You can simply add some files or run **apt get** command to install applications to the new image. All modifications are done in a new copy-on-write layer based on the original image layers. If a new file shares the same name as an existing file, the original file will be hidden and only the newly added file can be seen by user.

Just like open operations which take extra time due to locating the file in multiple

layers, programs like **ls** which involve traversing the directory have to go through unneeded hidden files. To prove this, we build images from the official Ubuntu image. Each time a new image is built, we add Linux kernel source code directory (53.9MB) to the **/home** directory of the previous image. We build 32 such images, each has a new Linux kernel source code directory located in **/home** hiding the original directory. In each directory, we run the time **ls -R /home >> /dev/null** command to note down the time consumption. Each experiment is done 100 times and we choose the mean measurement of the results. The outputs are shown as the green line in Figure 3.9. It is easy to observe from the figure that with the increment of the number of layers, latency is increasing too. Also, the line is almost linear except for some vibration. This might because the **ls -R** command will go through each layer of the **/home** directory. The reason why intercept is not zero is because the additional cost of output messages to **/dev/null** and there exists branch cut in each scan of the hidden layer to avoid too much additional time cost.

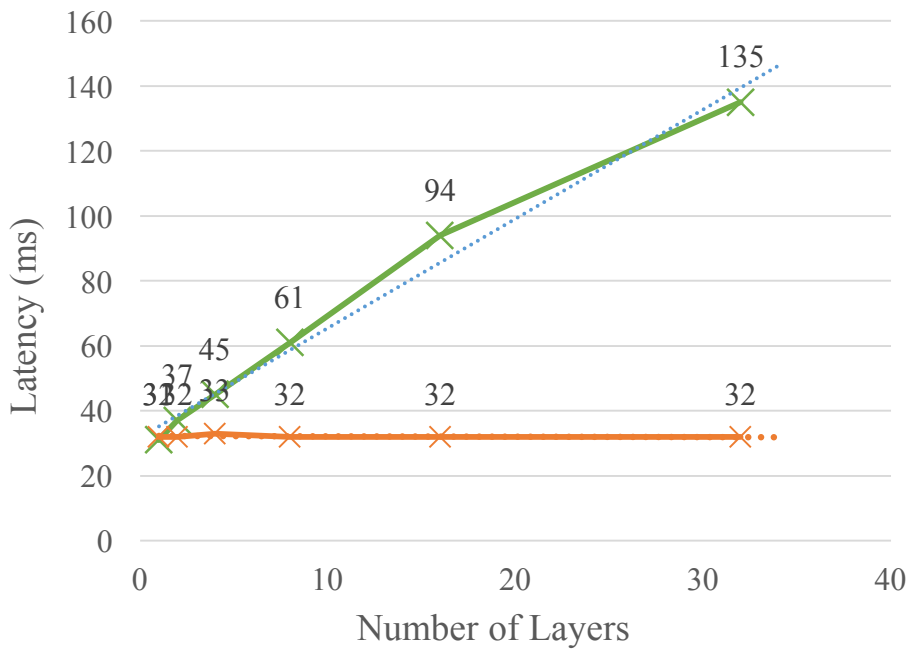


Figure 3.9 ls -R latency measurement.

Docker provides a method to flatten the images. Although this method is originally meant to erase the hidden files in the low layers and decrease the number of layers (the total number of layers is limited to 42 originally and it becomes to 127 recently), it does some help in this situation. The recommended method to build a new image is to write a *Dockerfile*. However, Docker provides us with other methods to build a new image. These are two commands **save** and **export**. All these commands can be acted on a running container instead of images. For example, if we execute the **save** command on a running container, all files in the lower layers and the current running layer will be saved to a **.tar** local file. If we use **import** to save the original file as a new image, we would find that the layers' tree structure of the image is still saved using the **tree** option. On the other hand, if we use the **export** command, and again import the newly created **.tar** file to a new image, we would find that there is only one layer left. This means that all the layers are flattened to a single layer, and if there are hidden files, the hidden files are thrown from the image. Only the ones you can see in a container are left. Also, the original image size of the one containing 32 layers of Linux source files is 1.911 GB. The one with only one layer of Linux source files is 241.8 MB. If we flatten an image with 32 layers of Linux source files, we would find the new one is 241.6 MB. The two values are very close but slightly different. So where does this 0.2 MB go? This is easy to explain. If we compress both the original Ubuntu image and also the image with one layer of Linux source files, we find that both these two new images are shrinking with a size of 0.2 MB. Actually, the original Ubuntu images is composed of many layers, and there also exists opportunities to compress this image.

In the following experiment, we flatten all the created images from one additional layer of Linux source files and 32 layers of Linux source files. We also execute the **ls -R** command in the Linux source file tree to note down the execution time for 100 times each image. We choose the average measurement as the final result. These results are shown as the orange line in Figure 3.9. From Figure 3.9, we can find that the orange line is almost horizontal. Also, the execution time is just very close to the one as the

original image with only one additional layer of Linux source files. So we can conclude that there is no extra time wasted on scanning the hidden files when the original image is flattened.

Since this method is so fascinating, is there any additional cost? Just like building a new image will involve huge amount of time, actually, there is. We log down the additional time to export the running image and create a new one, which is shown in Figure 3.10. To our surprise, the additional cost is not very high — less than 10 seconds in every case! Although there exists a linear increment as the number of layer increases, the execution time of the 32-layer situation is rather close to that of the 1-layer situation. This additional time consumption is very cheap in deploying a new service. Another drawback is that the new image is not based on old images and we can not make use of the layer structure to save disk storage. However, disk storage is almost the cheapest one nowadays and programmers usually don't mind much about the extra cost. We can even believe that the saved execution time and the increased service quality will overwhelm the extra cost of the additional start time.

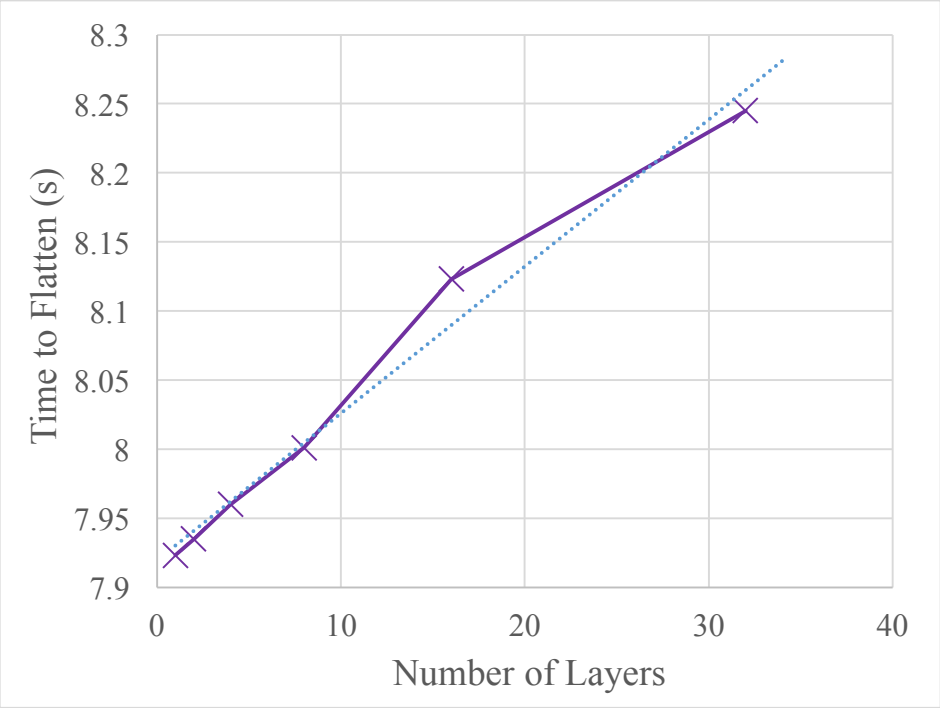


Figure 3.10 Time to flatten images

Chapter Four Conclusion and Discussions

Docker is not so real-time-latency-friendly. Although **cpu-shares** is a good choice as it doesn't cause a huge tail latency, in public cloud, many applications share a single CPU, using **cpu-shares** can lead to a single application consumes more CPU than it is allowed to use when other applications are not running on this CPU. This is good for consumers, but not fair since a consumer enjoys more resources than he has paid for. Also, when service providers continuously allocate and de-allocate containers on a CPU, applications running on that core will go through performance vibrations. While using **cpu-quota**, it is suitable for those CPU intensive works like many scientific workflow and HPC applications ([38], Xavier M. G., 2013: 233), but not those real time interactive services due to the possible long tail latency.

Docker assumes that users traditionally run applications on a desktop version of Linux, and thus enable the CFS while disable the real time one. There are two ways to let the servers have this additional real-time feature. The first one is to add options for Docker to support real-time scheduling. The other is more difficult but brings a lot good — implement another scheduling algorithm that takes both throughput and real time performance into consideration, and we won't need to reboot the machine to change the setting of CPU scheduler.

When at least two containers are running on a single CPU using CPU Quota, there is sure to be certain level of interference, either for a latency-sensitive container or a CPU intensive one. When workload on this CPU is at a low level, the interference is shown as an increment in context switch and slowdown the performance. However, when workload is high, the auto-rise in CPU frequency may speed up all of the running containers.

Many applications involve small size of messages, like Redis Cache ([40], Zawodny J., 2009.) and NoSQL databases ([34], Strauch C., 2011.). When using Linux

bridge to transmit message, it takes much more additional time due to short message size and high throughput, thus reducing the overall performance. But for applications like web gallery, since a photo is usually more than 1MB, the fix-length latency slow-down is overwhelmed by the system vibration, thus being less important in these cases. When using Linux bridge together with small size messages, it's not so latency-friendly since the extra time consumption is relatively large. We hope that a new method of transmitting information between Docker containers on different machines can be implemented to reduce this overall latency.

Some applications include many file operations. If the application simply opens several big files one time and continuously write and read them, then we can neglect the disk I/O overhead since operating an open file using AUFS is just like natively. However, if the application involves huge number of file open operations, it can cause a huge extra latency (over 20%). So applications running in Docker should not incorporate too many open operations and sometimes the design of the application has to be changed to suit Docker, thus bringing the programmers a lot of trouble.

Another thing we should bare in mind when using Docker is to reduce overall layers. Operations like traversing the file system involves going through hidden files in the low layers of AUFS, and we don't need these useless additional cost. One way to avoid this is to merge the different layers together and delete unnecessary files, either at container start time or image build time. However, additional time would be needed to merge the layers at these two stages. But these time are usually tolerable.

Chapter Five Future Works

Many future works can be done to reduce the latency and address the performance bottleneck. The first way is that Docker contributors take action. Just as we mentioned above, to reduce the tail latency caused by Docker, contributors can add the support for real-time scheduling algorithms to provide customers with more choice. Also, the current CPU quota granularity is not so real-time friendly. Since in our test most of the latencies are below 1000 us, and the smallest CPU quota is 1000 us, we would usually meet the situation where a service has used up its quota and waiting for the remaining period, thus Linux CGroups can employ lower CPU quota granularity. Docker can also take the priority to monitor all the containers on a host machine. Once it finds a container's CPU consumption behavior looks like a real-time one, it can give this container lower CPU quota granularity to satisfy the real-time demand. It can also increase the priority of the real-time container. Once a new request from users has come, the service can immediate preempt the CPU and quickly return the result, while at the same time, the upper bound of the CPU cost of this container is limited. A more interesting method is to let Docker daemon dynamically allocate resources when the containers are running. It can move those real-time containers to certain CPUs and left other CPUs used for those CPU intensive ones. Docker can then do optimizations on separate CPUs to let containers perform that. Since monitoring containers will cost host machine additional resources, cloud service suppliers can let the customers state their inclination before deploying services, and suppliers can then move the containers to the most suitable host machine. However, customers don't always choose the best configurations and thus exposing the potential problem that customers' services perform terribly and various type of containers interfere the performance of each other.

In the case of network latency, Docker is performing very bad and even falls behind

KVM ([10], Felter W., 2015: 171.). This long latency is caused by the overhead of Linux bridge. So the latency problem can be solved by simply discard Linux bridge and write Docker's own network stack. However, there is almost no opportunities for Docker network to perform exactly the same as using Docker host. Additional operations will always exist to provide network isolation and security between containers.

For the additional cost of file operations, there are several situations for Docker to improve. The first is to add the support for flatting at the container start time. However, this will add additional startup cost and containers may not so outperform virtual machines in start time. In another method, we don't flatten the container at start time. Once the application in the container has scanned some directories, the scanned directories are flattened. Although the first execution can be time consuming, the following operations on the same directory is much faster. The above two methods are just like the difference between C++ and C#. The former involves compiling code in advance and the second employs the method of just-in-time compilation.

Unfortunately, all above latency related questions of Docker are not solved currently. Cloud service customers can not wait until the new features of Docker come true. Then it is the customers' responsibility to take these extra latencies into consideration. Customers can buy a cloud machine, and configure the containers themselves. As we have mentioned above, sending a large trunk of data once at a time will reduce the effect of the extra fix-length latency. Thus many kinds of works are suitable for this case. Image bed, which is a storage system dedicated for photos, store and transmit images. Most of the images are very large (usually more than 1 MB) and we can fully make use of the size to narrow down the performance effect of network latency. Hadoop map-reduce applications are also suitable for this work since the files to be mapped and reduced are transmitted in the unit of trunk and these trunks are usually no less than 64 MB. If the application in a Docker container is some kind of database, we recommend that applications which interact with these databases transmit large amount of data once at a time. This can be done by predicting the users' requests and preload data from the

database. It can also be done by merging several requests together and query the database once to reach all these data. Fetching a lot of small size data separately can lead to a significant performance degradation as stated in the MySQL performance test of IBM report.

Another suggestion for service suppliers is to add new files instead of modify existing files. If we have to modify a 1 GB file, we have to copy the file and then operate on the newly created file due to the AUFS's copy-on-write feature. This can lead to two problems. The first problem is that the copy process will bring about huge slow start latency, while the other lies in the wasted disk storage. For those applications which can not avoid open many existing files in the low layers, a solution is to open these needed files when application start and avoid to close them during execution. This can lead to a slow application start time but increase the following operations' performance. However, due to the limited memory resource and valuable file descriptors, the limitation of this method is also obvious: you can not open two much files at a time. The best method to solve the traverse latency problem is to flatten the images in advance. However, this cannot solve the file open latency problem, since even if the image is flattened, you are still operating on a lower layer and can never avoid the copy-on-write process. But it does reduce some latency since there is no need for file system to scan the unneeded hidden layers.

A very serious problem of Docker containers is their security issues. People are working hard to address these problems. Docker contributors are trying to increase the security level of current Docker containers while maintain their current light-weight feature. However, this work is not that easy since Docker employs many Linux kernel technologies. An increment in the level of security and isolation can include the modification of kernel, which is hard to be done and difficult to be applied to all operating systems running a container. Another method comes from the opposite side. We can make the existing virtual machines more light-weight while still remain the isolation

and security capability. There are methods that run containers in virtual machines, however, this is totally unnecessary and ridiculous since applications have to go through the cost of two additional layers. The security and isolation level is no better than using only virtual machine while overhead incredibly increases. So we should think of other methods to combine virtual machines and containers together. VSphere Integrated Containers (VIC) is a new container technology provided by VMWare. Although it employs hardware-level virtualization, which can lead to worse performance than light-weight containers, it provides users with better isolation and security. It is also far different from traditional virtual machines since it provides the concept of images and layers. Although it doesn't have the performance advantage over containers, it takes the easy-to-deploy feature of Docker into consideration and provides better deployment experience for programmers.

REFERENCE

[1] Aas J. Understanding the Linux 2.6. 8.1 CPU scheduler[J]. Retrieved Oct, 2005, 16: 1-38.

[2] Agarwal K, Jain B, Porter D E. Containing the hype[C]//Proceedings of the 6th Asia-Pacific Workshop on Systems. ACM, 2015: 8.

[3] Shankar S. Amazon elastic compute cloud[J]. 2009.

[4] Colgate M, Stewart K, Kinsella R. Customer defection: a study of the student market in Ireland[J]. International Journal of Bank Marketing, 1996, 14(3): 23-29.

[5] Copeland M, Soh J, Puca A, et al. Overview of Microsoft Azure Services[M]//Microsoft Azure. Apress, 2015: 27-69.

[6] Dean J, Barroso L A. The tail at scale[J]. Communications of the ACM, 2013, 56(2): 74-80.

[7] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.

[8] Dua R, Raja A R, Kakadia D. Virtualization vs containerization to support paas[C]//Cloud Engineering (IC2E), 2014 IEEE International Conference on. IEEE, 2014: 610-614.

[9] Eder J. Accelerating red hat enterprise linux 7-based linux containers with solarflare openonload[R]. Red Hat Enterprise, 2015.

[10] Felter W, Ferreira A, Rajamony R, et al. An updated performance comparison of virtual machines and linux containers[C]//Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On. IEEE, 2015: 171-172.

[11] Gibson G A. Redundant disk arrays: Reliable, parallel secondary storage[M]. Cambridge, MA: MIT press, 1992.

[12] Hasenstein M. The logical volume manager (LVM)[J]. White paper, 2001.

[13] He S, Guo L, Guo Y, et al. Elastic application container: A lightweight approach

for cloud resource provisioning[C]. Advanced information networking and applications (aina), 2012 ieee 26th international conference on. IEEE, 2012: 15-22.

[14] Helsley M. LXC: Linux container tools[J]. IBM devloperWorks Technical Library, 2009.

[15] Hopper T. Cumulative Distribution Function[J]. Month, 2014.

[16] Huber N, von Quast M, Hauck M, et al. Evaluating and Modeling Virtualization Performance Overhead for Cloud Environments[C]//CLOSER. 2011: 563-573.

[17] Jalaparti V, Bodik P, Kandula S, et al. Speeding up distributed request-response workflows[J]. ACM SIGCOMM Computer Communication Review, 2013, 43(4): 219-230.

[18] James T Y. Performance evaluation of Linux Bridge[C]//Telecommunications System Management Conference. 2004.

[19] Kivity A, Kamay Y, Laor D, et al. kvm: the Linux virtual machine monitor[C]//Proceedings of the Linux symposium. 2007, 1: 225-230.

[20] Kratzke N. About Microservices, Containers and their Underestimated Impact on Network Performance[J]. CLOUD COMPUTING 2015, 2015: 180.

[21] Li C, Ding C, Shen K. Quantifying the cost of context switch[C]//Proceedings of the 2007 workshop on Experimental computer science. ACM, 2007: 2.

[22] Li J, Sharma N K, Ports D R K, et al. Tales of the tail: Hardware, os, and application-level sources of tail latency[C]//Proceedings of the ACM Symposium on Cloud Computing. ACM, 2014: 1-14.

[23] Liu J, Huang W, Abali B, et al. High Performance VMM-Bypass I/O in Virtual Machines[C]//USENIX Annual Technical Conference, General Track. 2006: 29-42.

[24] Love R. Kernel korner: CPU affinity[J]. Linux Journal, 2003, 2003(111): 8.

[25] Menage P, Seth R, Jackson P, et al. Linux control groups[J]. 2007.

[26] Menascé D A. Virtualization: Concepts, applications, and performance modeling[C]//Int. CMG Conference. 2005: 407-414.

[27] Merkel D. Docker: lightweight linux containers for consistent development and deployment[J]. Linux Journal, 2014, 2014(239): 2.

[28] Nishtala R, Fugal H, Grimm S, et al. Scaling memcache at facebook[C]//Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13). 2013: 385-398.

[29] Pendry J S, Sequent U, McKusick M K. Union mounts in 4.4 BSD-lite[J]. AUUGN, 1997: 1.

[30] Slee M, Agarwal A, Kwiatkowski M. Thrift: Scalable cross-language services implementation[J]. Facebook White Paper, 2007, 5(8).

[31] Soltesz S, Pötzl H, Fiuczynski M E, et al. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors[C]//ACM SIGOPS Operating Systems Review. ACM, 2007, 41(3): 275-287.

[32] Špaček F, Sohlich R, Dulík T. Docker as platform for assignments evaluation[J]. Procedia Engineering, 2015, 100: 1665-1671.

[33] SPECweb2009 E. commerce workload, 2009[J].

[34] Strauch C, Sites U L S, Kriha W. NoSQL databases[J]. Lecture Notes, Stuttgart Media University, 2011.

[35] Tseng H M, Lee H L, Hu J W, et al. Network virtualization with cloud virtual switch[C]//Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on. IEEE, 2011: 998-1003.

[36] Tsirtsis G. Network address translation-protocol translation (NAT-PT)[J]. Network, 2000.

[37] Wright C, Cowan C, Smalley S, et al. Linux security modules: General security support for the linux kernel[C]//null. IEEE, 2003: 213.

[38] Xavier M G, Neves M V, Rossi F D, et al. Performance evaluation of container-based virtualization for high performance computing environments[C]//Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on. IEEE, 2013: 233-240.

[39] Xu Y, Musgrave Z, Noble B, et al. Bobtail: Avoiding long tails in the cloud[C]//Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13). 2013: 329-341.

[40] Zawodny J. Redis: Lightweight key/value store that goes the extra mile[J]. Linux Magazine, 2009, 79.

Acknowledgements

As time goes by, it's drawing to the end of my four-year college study. Looking back to the past few years, I can still fetch up the days and nights when we students work hard to finish projects, when we stay up late to make up for the coming examination and when teachers kindly discuss with us in the class. It is SJTU that brings me to the world of Computer Science. She makes me grasp the beauty of the binary world and finally fall in love with it.

I would like to extend my sincere expression to my advisor Dr. Chao Li, who leads me to the world of scientific research. Without his patient instructions and practical advices, it would be hard for me to finish the graduation project. It is him that arouses my interest in research and makes me keep eager eyes on the unknown Computer Science world.

Thanks to my family and all the friends who accompany me during the college years. My achievement today is inseparable from your help and encouragement.