Evaluating and Improving LXC Container Migration between Cloudlets Using Multipath TCP

By

Yuqing Qiu

A thesis submitted to the Faculty of Graduate and Postdoctoral Affairs in partial fulfillment of the requirements for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Carleton University

Ottawa, Ontario

Abstract

The advent of the Cloudlet concept—a “small data center” close to users at the edge is to improve the Quality of Experience (QoE) of end users by providing resources within a one-hop distance. Many researchers have proposed using virtual machines (VMs) as such service-provisioning servers. However, seeing the potentiality of containers, this thesis adopts Linux Containers (LXC) as Cloudlet platforms. To facilitate container migration between Cloudlets, Checkpoint and Restore in Userspace (CRIU) has been chosen as the migration tool. Since the migration process goes through the Wide Area Network (WAN), which may experience network failures, the Multipath TCP (MPTCP) protocol is adopted to address the challenge. The multiple subflows established within a MPTCP connection can improve the resilience of the migration process and reduce migration time. Experimental results show that LXC containers are suitable candidates for the problem and MPTCP protocol is effective in enhancing the migration process.

Acknowledgement

I would like to express my sincerest gratitude to my principal supervisor Dr. Chung-Horng Lung who has provided me with valuable guidance throughout the entire research experience. His professionalism, patience, understanding and encouragement have always been my beacons of light whenever I go through difficulties. My gratitude also goes to my co-supervisor Dr. Samuel A. Ajila whose informative comments and advice have assisted me to understand my research more clearly.

I would also like to thank Jerry Buburuz, who is the computer network administrator of my department. He solved many of the technical problems I encountered during the research process with great efficiency and kindness. I'm also grateful to Sanstream Technology for providing funding to my research.

Lastly, I want to take this opportunity to thank my husband Yifan, my twin sister Yuji, my beloved parents, and my dear friends Muci, Shayna, Man Si and my other friends. Their love, comfort and support are incredibly important to me. Knowing that they are by my side is a real blessing to me.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AWS	Amazon Web Service
BlkIO	Block I/O
CFS	Completely Fair Scheduler
CRIU	Checkpoint and Restore in Userspace
IaaS	Infrastructure as a Service
IETF	Internet Engineering Task Force
IoT	Internet of Things
IPC	Inter-process Communication
ISP	Internet Service Provider
KVM	Kernel-based Virtual Machine
LAN	Local Area Network
LXC	Linux Container
MPTCP	Multipath TCP
NAT	Network Address Translation
NIS	Network Information System
PaaS	Platform as a Service
PID	Process Identifier
QoE	Quality of Experience
RT	Real Time Scheduler
RTT	Round Trip Time

SaaS	Software as a Service
TC	Traffic Controller
TCP	Transmission Control Protocol
UTS	Unix Timesharing System
VEPA	Virtual Ethernet Port Aggregator
VLAN	Virtual Local Area Network
VM	Virtual Machine
WAN	Wide Area Network
XaaS	Anything as a Service

Chapter 1 Introduction

Cloud Computing has been widely employed to ensure convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort or service provider interaction [1]. Due to the deployment and facility costs, there are only a limited number of data centers around the globe that provide Cloud Computing services. This means that if users are located far away from those data centers, their requests and data have to travel a long way through the Internet, which makes their Quality of Experience (QoE) prone to network latency, especially when they are running latency-sensitive applications. This issue has been amplified with the increasing popularity of mobile devices and the potentialities of Internet of Things (IoT), all of which are considered to be at the edge of the network and therefore may suffer from service degradation when they need to access resources in data centers.

Under this circumstance, a new computing paradigm named Fog Computing was brought into light. Fog Computing was first used by Bonomi *et al*., [2] who suggested that since fog is closer to the ground than cloud, Fog Computing is closer to end users and therefore will be more efficient to support latency-sensitive applications. The "small data centers" that constitute the Fog Computing layer are called Cloudlets. Similar to the concept of Cloud Computing which implements virtual machines (VMs) for service provisioning, many researchers have proposed VM-based Cloudlets for Fog Computing. Although VMs can offer an ideal isolated environment to serve edge

users, they are generally heavyweight and may introduce a large amount of overheads to the Cloudlet.

Based on the above considerations, instead of using VM, this thesis explores the implementation of container—a lightweight virtualization tool for Linux kernel to function as the service provisioning platform in Cloudlets.

1.1 Motivation

The motivation to perform this research is to provide more efficient solutions to the virtual servers in Cloudlets. Compared with large data centers, there are three distinctive characteristics of Cloudlets listed below:

1) Fast service initiation: As the hypothesis of Fog Computing suggests, Cloudlets could be deployed in shopping centers, public transportation stations, restaurants, parks, convenience stores, schools, corporations and so on. Depending on the scale of Cloudlets and the density of user applications, a large one may need to serve hundreds of end users. In this case, once the users of mobile devices or IoT offload their applications or part of them to the Cloudlet, the virtual servers are expected to display a high level of agility in respect of virtual server creation and startup.
2) Limited capacity of Cloudlets: It is reported by James Hamilton [3], an Amazon Web Service (AWS) distinguished engineer, that a typical datacenter has at least 50,000 servers and sometimes more than 80,000 servers. This figure greatly outnumbers that of the proposed servers in a Cloudlet, not to mention the

computing capability of these servers. With limited capacity and yet a relatively large number of applications to serve, it's crucial to keep the virtual servers' footprints small so that resource utilization can be maximized.

3) User mobility: The initial purpose of creating the Fog Computing model is to better support users at the edge, in particular mobile devices and IoT applications. Apparently, these types of applications require mobility support. The virtual servers in the Cloudlets should be able to efficiently migrate to neighboring ones as the user changes their location. Ideally, the service should not be interrupted, or the interruption should be mitigated within an acceptable range.

To align with the aforementioned characteristics, between the candidates of VMs and containers, this research chooses containers for the following reasons:

1) Quick creation and startup time: Taking only a few minutes to create and a few milliseconds to boot up, containers are best suited to meet the fast service initiation requirement. VM, on the other hand, is no rival in terms of efficiency, since it usually needs no less than ten minutes to create and no less than one minute to start. In addition, once the image of a container is created and stored into cache, the following containers with the same image can be created even faster, taking only dozens of seconds.
2) Small footprint: The architecture of a container makes it occupy much less disk space and consume much less RAM than a VM does. The implementation of containers in Cloudlets certainly makes better use of the limited resources and

reduces overheads in the meantime.

3) Migration support: Although container technology is quite new and still needs many improvements, there are already open-source migration tools that can help containers to migrate, like Checkpoint and Restore in Userspace (CRIU) [4] and Flocker [5]. With the help of these tools, containers can migrate to neighboring Cloudlets and continue to serve user applications. Furthermore, to cope with the possibility of connection lost during container migration, the Multipath TCP (MPTCP) protocol [6] has been adopted in this thesis to increase fault tolerance.

1.2 Contributions

The main contributions of this thesis can be summarized as follows:

1) This thesis explores the potentiality of containers to be used in Cloudlets in the context of Fog Computing. Experimental results collected from Linux Containers (LXC) [7] with multiple, complex, embedded applications running (see Section 4.2) demonstrate that they can efficiently deliver services and perform computational-intensive tasks on behalf of the end users.
2) This thesis performs experiments and analyzes the results on a container migration technique based on CRIU and a Linux script written by Anderson [8]. The migration technique adopts a three-step approach to checkpoint, copy and then resume the container on the remote host (see Section 4.3).
3) This thesis emulates the cooperation between VM and LXC containers by running LXC containers inside VMs (see Section 3.1). The idea is to demonstrate that

LXC containers are not only replacements, but also additions. In the case where one must use a VM, there is still room for containers. Moreover, the fact that containers can be built on top of a VM accentuates their flexibility and lightness.

4) This thesis employs MPTCP to enhance the continuity of connection between two hosts and to accelerate the migration process. With the help of MPTCP, if the connection of one subflow breaks down, another subflow will take over and guarantee that the migration will not be interrupted (see Section 4.4).

1.3 Thesis Outline

The rest of the thesis is organized as follows:

Chapter 2 introduces the background information on Fog Computing, Cloudlet, containers and MPTCP. Related literature in the area of Fog Computing and container technology is reviewed in this chapter.

Chapter 3 describes the proposed approach to deploy containers in Cloudlets and explains the working mechanisms of container migration. In addition, MPTCP-specific operations during the migration process are also covered. Finally, this chapter is concluded by pointing out current limitations in the areas of LXC container migration.

Chapter 4 presents the experiment results and the performance analysis of LXC containers and their migration between hosts. Migration measurements and discussions under MPTCP are also provided.

Chapter 5 concludes this thesis and suggests possible directions for future work.

Chapter 2 Background and Related Work

This chapter introduces the four underlying concepts of this thesis: Fog Computing, container, CRIU and MPTCP. Section 2.1 describes the model of Fog Computing along with its characteristics and areas of implementation. In particular, Cloudlet, being the essential component in the Fog Computing model, is elaborated on in Section 2.2. Section 2.3 provides a detailed introduction of the container technology. Section 2.4 discusses CRIU, the software tool that lays the foundation for container migration. Section 2.5 introduces MPTCP, the protocol which facilitates the migration process. Section 2.6 presents a literature review in the relevant areas. Section 2.7 points out the comparisons between this thesis and related work.

2.1 Fog Computing

According to the brief explanation in the first chapter, the concept of Fog Computing was brought forth in an attempt to cater users' requirements at the edge of the network, including mobile devices such as cell phones, tablets, and IoT, like smart grids, smart homes, intelligent transportation and smart cities. The way Fog Computing can facilitate the tasks of end users is to exist as a highly virtualized platform between users and the Cloud Computing data centers while providing compute, storage and networking services in the meantime [2]. As Figure 2.1 depicts, the idealized blueprint is a three-hierarchy framework with data centers on the top performing the traditional tasks of resource allocation and service provisioning, Fog Computing servers in the middle interacting with both sides and heterogeneous end

users at the bottom doing offloading, downloading and various computations on-the-fly.

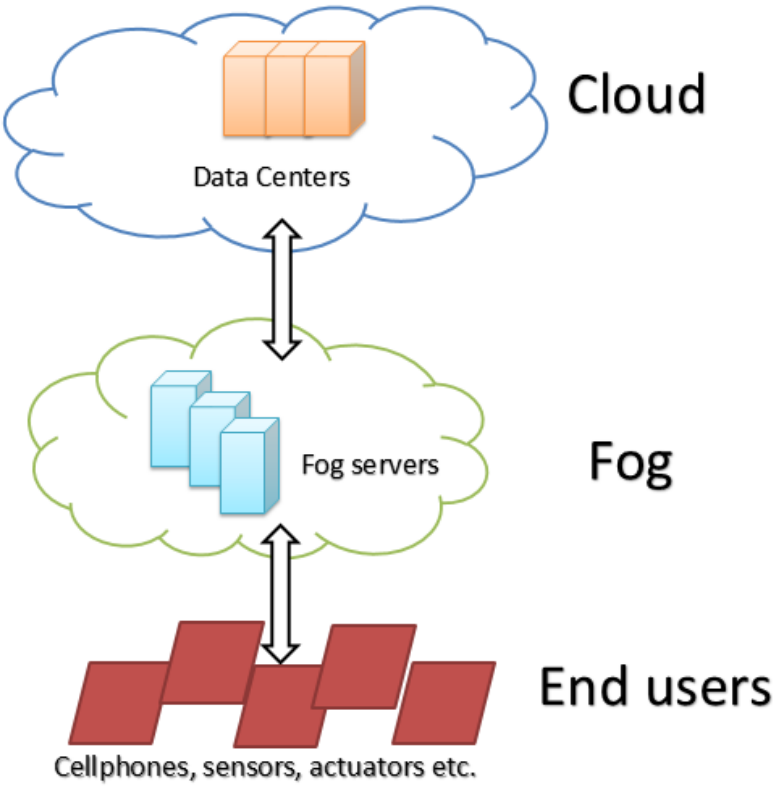

Figure 2.1 Three-hierarchy Framework

This unique location of Fog servers ensures that they function in a similar manner as data centers, but differentiating from them through some modifications tailored for end users. The modifications should reflect the distinctive features of Fog Computing:

1) Vicinity to end users: Distance reduction between Fog servers and end users will greatly improve the network performance of applications which are easily affected by network delays. The proposed idea is to place those servers one-hop away from the users so that the network latency is confined to Local Area Network (LAN), not Wide Area Network (WAN).

2) Large-scale deployment: Given the fact that fog servers are close to users at the

edge, the only way that they can be readily accessible is through facility ubiquity. Unlike the Cloud Computing data centers that have a limited number of centralized locations, Fog Computing servers offer the advantage of vicinity and accessibility by large-scale deployment in shopping centers, public transportation stations, restaurants, parks, convenience stores and schools, etc.

3) Heterogeneity: This applies to both the end users and the Fog servers. As there are many types of devices involved in the concept of IoT, the applications they offload to the servers could be in different flavors. Likewise, in order to cope with these, Fog servers are not limited to Cloudlets—the "small data centers". As authors in [9] point out, Fog servers could be resource-poor devices such as set-top-boxes, access points, routers, switches, base stations and end devices, or resource-rich machines such as Cloudlet and IOx. IOx is a product from Cisco that works by hosting applications in a Guest Operating System running in a hypervisor directly on the Connected Grid Router (CGR) [10].
4) Local services: Fog servers are location-aware, which indicates that the kind of services they provide depends on the characteristics of their coverage range. For example, if the servers are deployed in a shopping mall, specific store information could be cached in the servers and offered to customers who can retrieve it through Wi-Fi-connected mobile devices or directly from the electronic navigation screen in shopping malls.
5) Centralized and distributed management: The above feature implies that the management of Fog Computing servers takes two forms: centralized and

distributed. Local businesses can own and manage their Fog servers and local government can adopt a centralized approach to supervise the pricing model, security inspection and copyright control etc. of those individually-managed servers.

The above characteristics of Fog Computing demonstrate that it's more of a complementary component in the general computing framework than a merely extension of Cloud Computing to the edge [11]. Table 2.1 summarizes the differences between them.

Table 2.1 Comparisons of Fog Computing and Cloud Computing [11]

	Fog Computing	**Cloud Computing**
Target User	Mobile users and IoT	General Internet users
Service Type	Limited localized information services related to specific deployment locations	Global information collected from worldwide
Hardware	Limited storage, compute power and wireless interface	Ample and scalable storage space and compute power
Distance to Users	In the physical proximity and communicate through single-hop wireless connection	Faraway from users and communicate through IP networks
Working Environment	Outdoor (streets, parklands, etc.) or indoor (restaurants, shopping malls, etc.)	Warehouse-size building with air conditioning systems
Deployment	Centralized or distributed in regional areas by local business (local telecommunication vendor, shopping mail retailer, etc.)	Centralized and maintained by Amazon, Google, etc.

Among the various compositions of Fog servers, the containers employed by this thesis that act as virtual servers are applied to Cloudlets, which are generally regarded

as small data centers. The term Cloudlet is self-explanatory in that it's a combination of "Cloud" as in Cloud Computing and "let" as in the diminutive suffix denoting small. Though not as competitive as large data centers in terms of server capability, Cloudlets are still expected to be trustful and resource-rich from the perspective of end users. Borrowing the mechanisms from Cloud Computing, many researchers apply VMs to Cloudlets to fulfill this expectation.

2.2 VM-based Cloudlets

Authors in [12] describe the working mechanism of VMs in a Cloudlet (see Figure 2.2).

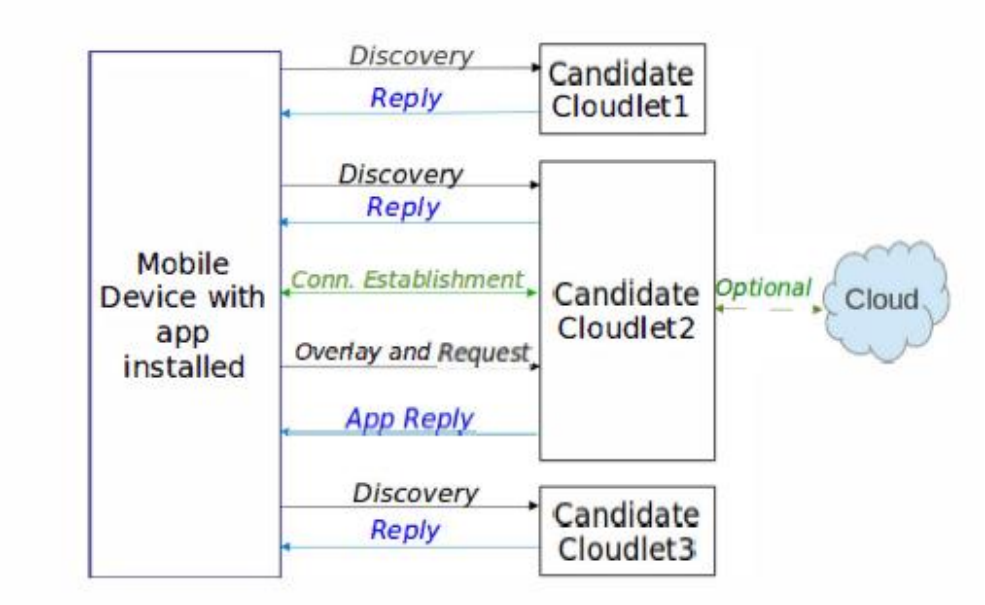

Figure 2.2 Cloudlet Architecture [12]

A VM of the mobile device runs on a nearby Cloudlet. When some computational-intensive applications need to be run by the mobile device, it initiates a connection to the Cloudlet, which performs the task on the VM and then returns the

results back to the mobile device. In some cases, the Cloudlet may need to communicate with the data centers to get some resources. Depending on the density of Cloudlet deployment in the area, there may be multiple Cloudlets (candidate Cloudlets) that can offer the service. The candidates need to meet two criteria: one-hop away from the user and the possession of base VM of the mobile operating system. The latter one is based on the approach from [13], which will be discussed in sub-section 2.2.1. The choice of the optimal Cloudlet among the candidates relies on four crucial factors:

1) Processing speed of the Cloudlet: Since it's not accurate to compare the speed merely from the processor speed (in GHz) and the number of CPU cores of the Cloudlet servers, a more comprehensive method is to calculate the speed parameter from the combination of a small, CPU-intensive benchmark test, the current available CPU usage and a given weight.
2) Available memory at the Cloudlet: This parameter takes two factors into account: the memory necessary for the creation of launch VM and the memory needed by the application.
3) WAN latency: Timestamps are collected to calculate the duration from mobile devices sending Cloudlet discovery message to receiving replies from the Cloudlets. The longer the duration, the lower the score of the Cloudlets. Then this score is multiplied with a weight, which is decided by the amount of data to be transferred.
4) Bandwidth of the fixed network: The network bandwidth available to the

candidate Cloudlets vary from time to time. Therefore, it's important to keep track of the current available bandwidth when serving applications like video streaming.

After the decision of the most suitable Cloudlet, the mobile devices will start offloading the applications to and then downloading from the Cloudlet once the service finishes.

2.2.1 Dynamic VM Synthesis

As indicated by section 2.1, the ubiquity of Cloudlets is one of the highlights of Fog Computing. However, without the premise of a simple management model of Cloudlets, it will be difficult to achieve the expected scope of coverage. In an effort to address this challenge, Satyanarayanan *et al*. in [13] proposed transient customization of Cloudlet infrastructure. Featuring pre-use customization and post-use cleanup, the fundamental purpose of this approach is to restore the software state of VM-based Cloudlets to the pristine state without manual intervention. The way in which the customization and cleanup are implemented is called dynamic VM synthesis.

The basic idea is to preload the base VM in Cloudlets. Once the mobile device establishes connection and delivers a small VM overlay, the infrastructure applies the overlay to the base to derive the launch VM, which starts to be executed. When the service is finished, the VM is abandoned, enabling the infrastructure to restore to the pristine state. Figure 2.3 shows the whole process.

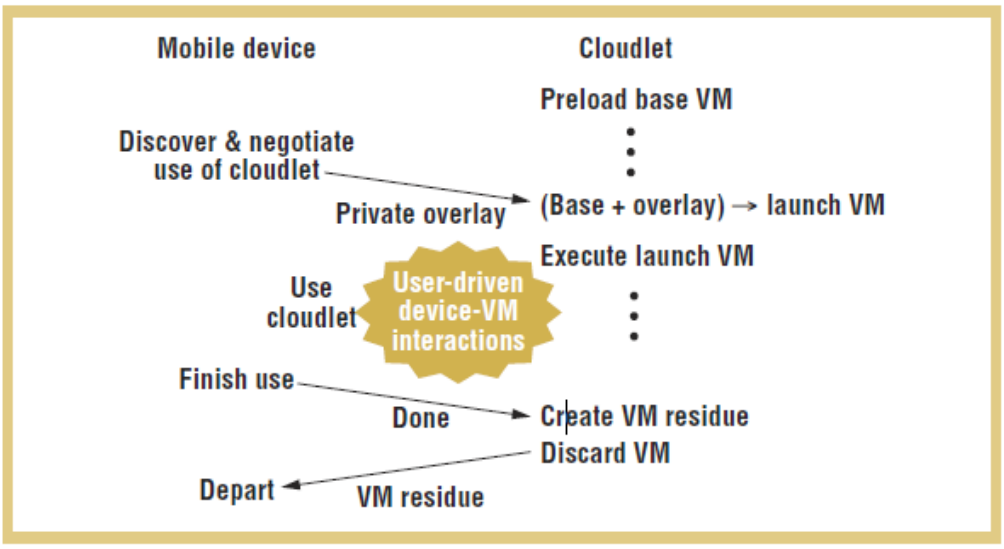

Figure 2.3 Dynamic VM Synthesis [13]

To explore the feasibility of this hypothesis, the authors of [13] have built a proof-of-concept prototype called Kimberley and have chosen six large applications as well as null to indicate the intrinsic overhead of the prototype. For a VM with 8 GB of virtual disk, the experiment results show that VM overlay size is reduced to 100 to 200 MB, which is an order of magnitude smaller than the full VM size. When these VM overlays are synthesized with the base VM, the total synthesis time over a 100Mbps connection ranges from 60 to 90 seconds, which are satisfactory for a proof-of-concept prototype, yet unacceptable for real-world scenarios.

Improvements of the above prototype are conducted by researchers in [14]. They extend the concept of VM overlay to include both disk and memory snapshots. With a series of optimizations, they have been able to reduce the synthesis time to no more than 10 seconds (see Figure 2.4).

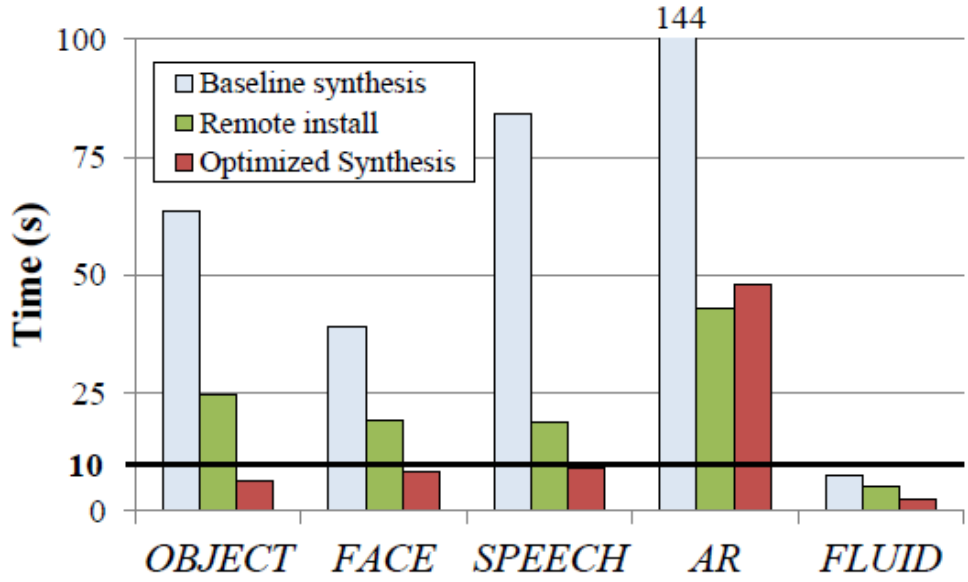

Figure 2.4 First Response Times [14]

The adoption of VMs in Cloudlets is a reasonable and suitable solution until the emergence of containers. The lightweight quality of containers has made researchers reconsider the virtual servers in Cloudlets. Although VMs are capable to satisfy the various requirements of Cloudlets, won't it be even better if we choose containers to reduce overheads? In addition, it will also be feasible and interesting to cooperate containers with VMs. The next section will introduce containers and explain the reasons why containers have the ability to challenge VMs in the virtual server area.

2.3 Introduction to Containers

A Linux container is an operating-system-level virtualization method for running multiple isolated Linux systems (containers) on a control host using a single Linux kernel [7]. This demonstrates that Linux containers reside directly on the host operating system without any abstraction layer of hardware or guest operating system.

Figure 2.5 and Figure 2.6 show the difference in architecture between VMs and containers.

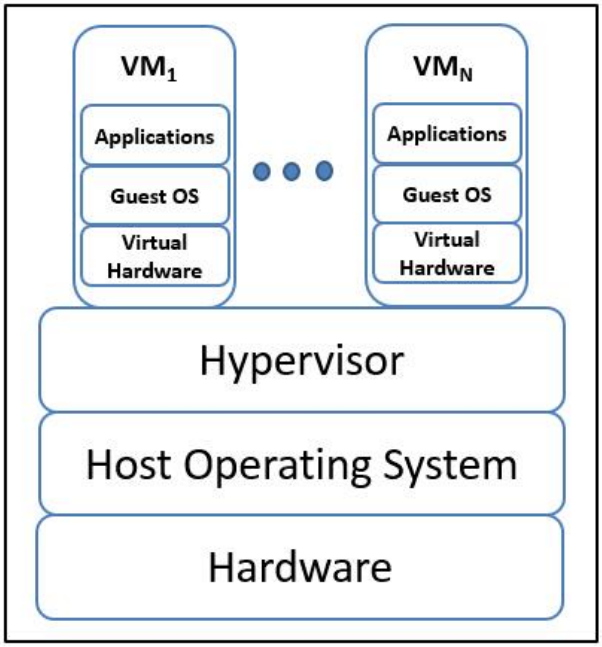

Figure 2.5 VM Architecture

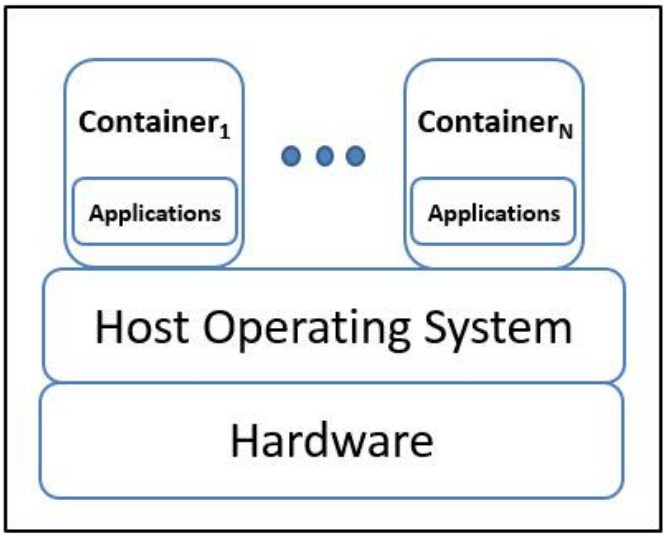

Figure 2.6 Container Architecture

Despite this difference, a container works pretty much in the same way as a VM does—allocating and isolating resources. Instead of using a hypervisor, the realization

of the container technology is attributed to two Linux kernel features: namespace and cgroup, also known as control group.

2.3.1 Namespaces and Cgroups

Namespaces provide the isolation of resources for each container. It enables each container to see the underlying system from its own point of view. As of now, there are six namespaces: inter-process communication (IPC), network, mount, process identifier (PID), user and Unix timesharing system (UTS). The functions of each namespace are summarized below:

1) IPC namespace isolates the IPC resources, such as shared memory, semaphore and message queue.
2) Network namespace isolates system resources related with networking, including network interfaces, network devices, IPv4 and IPv6 protocol stacks, routing tables, iptables rules, firewalls and port numbers (sockets), etc.
3) Mount namespace isolates the set of filesystem mount points, which means that each process can have their own root filesystem.
4) PID namespace isolates the PID number space. PIDs in a new PID namespace start at 1.
5) User namespace isolates security-related identifiers and attributes, in particular, use IDs and group IDs, the root directory, keys, and capabilities. A process's user and group IDs can be different inside and outside a user namespace. For example, UID from 0 to 1999 inside a container can be mapped to UID from 10000 to

11999 on the host.

6) UTS namespace isolates two system identifiers: the hostname and Network Information System (NIS) domain name.

Cgroup controls the allocation of resources to each container. It achieves resource metering and limiting through nine subsystems (a.k.a. resource controllers): block I/O (BlkIO), CPU, cpuacct, cpuset, device, freezer, memory, net_cls and net_prio. The function descriptions of each subsystem are summarized in Table 2.2.

Table 2.2 Cgroup Subsystems

Subsystem	Description
BlkIO	Keeps track of and sets limits to block I/O operations. It offers two policies for controlling access to I/O: proportional weight division and I/O throttling (Upper limit).
CPU	Allocates CPU resources based on two schedulers: Completely Fair Scheduler (CFS) and Real-Time Scheduler (RT).
cpuacct (CPU accounting)	Reports usage of CPU resources.
cpuset	Assigns CPU cores and memory nodes.
Devices	Allow or deny access to system devices.
Freezer	Suspends or resumes cgroup tasks.
Memory	Controls access to memory resources, and reports on memory usage
net_cls	Tags network packets with a class identifier (classid) that allows the Linux traffic controller (tc) to identify packets.
net_prio	Dynamically sets the priority of network traffic per each network interface. Network priority is used to differentiate packets that are sent, queued, or dropped.

2.3.2 Features of Containers

The aforementioned kernel capabilities save containers the overheads of installing and configuring virtual hardware and guest operating system. The fact that containers are installed directly on the host and share the host's system resources has brought forth many distinctive features, which will be discussed below:

1) Fast startup, termination, creation and deletion: The startup and termination time of Linux containers are calculated by the milliseconds. And it takes a little longer than ten minutes to create a new container if the required base image hasn't been cached yet. After the first time, the following containers with the same base image will be created in just a few minutes. The deletion time of a container is the time needed to remove the filesystem and related configuration files of the container. Usually this just takes a few seconds.
2) Small footprint: Thanks to the sharing of host operating system, a container only occupies the space needed for the base image and the applications inside. This lightweight feature of containers helps increase workload density and resource utilization.
3) Near native performance: Without the extra overheads induced by hypervisors, Linux containers offer near native performance. A graphical illustration of this quality will be presented in sub-section 2.3.3.
4) Limited to Linux operating system: Since containers depend on the support of Linux kernels, they cannot run under Windows. In an Ubuntu operating system, the containers created can be in different flavors of Linux, like Fedora and Debian.

5) Potential security risks: With containers directly installed on top of the host operating system, there are concerns that the container user may get hold of the host kernel either by accident or on purpose. Seeing this problem, container developers have been dedicating to reducing security risks.

The above features of containers make containers suitable for a wide range of use cases. Authors in [15] listed some of the areas that containers can contribute to (see Table 2.3).

Table 2.3 Use Cases for Linux Containers [15]

Use Case	What's New	Benefit
Development and testing	Replicate self-contained application images on any certified host platform and infrastructure.	Reduce dependency problems. Choose where to deploy.
Low-overhead Linux distributions	Encapsulate libraries and services in the container image.	Reduce OS overhead. Enable the use of small-footprint operating system.
Cloud-native applications	Abstract physical resources and create stateless web and application tiers.	Avoid being locked into a cloud application framework.
Scaling up and down	Spin up containers when workload is heavy. Retire them when they are no longer needed.	Use resources more efficiently.
PaaS	Transport application binaries and configurations in containers. Most PaaS frameworks are already container-based.	Allow PaaS frameworks to interoperate.
Intercloud portability	Natively support Docker containers in public and private clouds.	Exchange application components between clouds.
Microservices	Design applications as suites of services, each written in the best	Scale just the microservices that need

	language for the task.	more resources, not the entire application. Allow different teams to manage different microservices.
Compute-enabled storage	Bring computing resources to data instead of bringing data to computing resources.	Speed up computing operations by not having to move large volumes of big data. Use any language or runtime system.
Shared network functions and services	Virtualize network services in containers instead of on virtual machines or network devices.	Accelerate startup and shutdown, improving user experience.
Containerized control and management planes	Transport control-plane and management-plane software.	Reduce overhead and maintain image integrity.
Edge computing and Internet of Everything solutions	Package, distribute, and run applications and services at the edge to process data closer to the origin.	Conserve bandwidth from the edge to the core. Enable the right kind of services for the particular type of data and type of analysis.
Policy-aware networks	Grant or block access to containers based on policy. For example, apply Quality of Service (QoS) to containers.	Improve the application experience by giving priority to containers with critical services. Improve security.
Network application containers	Run third-party applications in containers on the network operating system.	Add new capabilities to the base network operating system.

2.3.3 Performance Comparison between Containers and VMs

The previous section elaborates on the advantages of containers by a qualitative approach. This section will present two research works on the performance benefits of containers based on a quantitative research.

Joy *et al*. in [16] perform a comparison between Amazon EC2 VM and Docker containers [17]. The comparison focuses on two aspects: application performance and

scalability.

1) Application performance: For the first test, the authors choose a front end application server hosting Joomla php and a backend server hosting PostgreSQL database for storing and retrieving data for the Joomla application. Jmeter utility is used for stress testing the application by simultaneously sending requests to the Joomla application till the server experiences problems in performance. Figure 2.7 shows the results. According to the figure, within the duration of 600 seconds, Docker is able to process three times as many the requests as those processed by VMs.

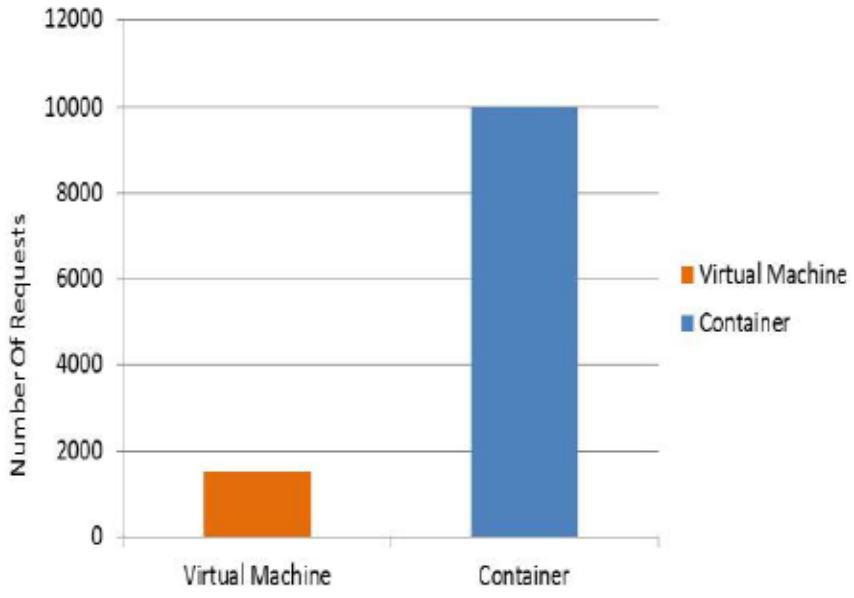

Figure 2.7 Processed Requests in 600 Seconds [16]

2) Scalability: The auto scaling feature of AWS EC2 is used for VM. The feature scales up VM instances when it reaches maximum CPU load. As for container scalability, Kubernetes clustering tool is adopted. Both VM and Docker run a load

balanced WordPress application. Jmeter triggers the scaling of VM by increasing the CPU utilization above 80%, while the same Jmeter sends concurrent request to front end WordPress to initiate container scaling. Results reveal that the calculated time for scaling up is three minutes for VMs and only eight seconds for Docker containers, which means containers outperform VMs by 22 times faster in scalability (see Figure 2.8).

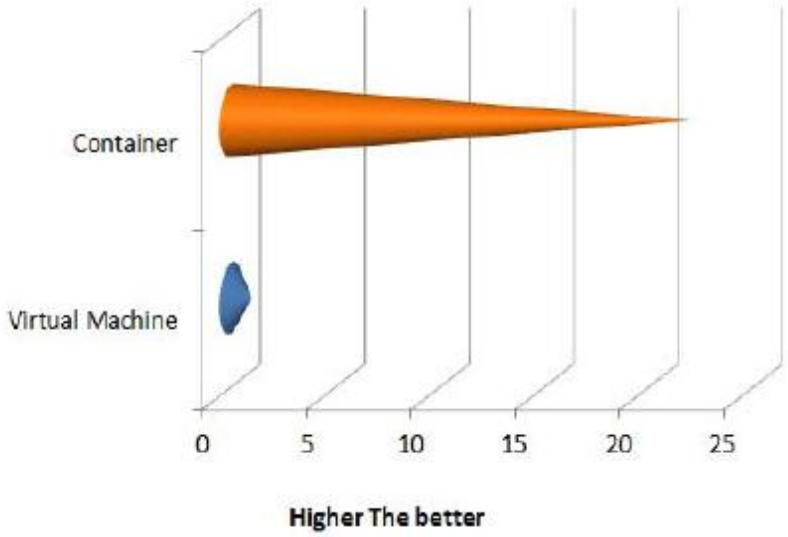

Figure 2.8 Scalability between VMs and Containers [16]

The x-axis in the Figure 2.8 represents the ratio of scalability. Based on the above experiment outcomes, the authors draw the conclusion that containers can be used for application deployments to reduce resource overhead. However, when running applications with business critical data, VMs will be a better fit.

In [18], the researchers choose kernel-based virtual machine (KVM) and Docker as representatives of the hypervisor-based and container-based virtualization technologies respectively. The server applications involved in [18] are Redis and

MySQL. Only the outcomes related with MySQL are presented.

The choice of MySQL reflects the consideration that it is a popular relational database widely used in the cloud, and it stresses memory, IPC, network and filesystem subsystems. The *sysbencholtp* benchmark is run against a single instance of MySQL 5.5.37. Five different configurations are measured: MySQL running normally on native Linux, MySQL under Docker using host networking and a volume (Docker net=host volume), using a volume but normal Docker networking (Docker NAT volume), storing the database within the container filesystem (Docker NAT AUFS) and MySQL running under KVM. Figure 2.9 shows the simulation results.

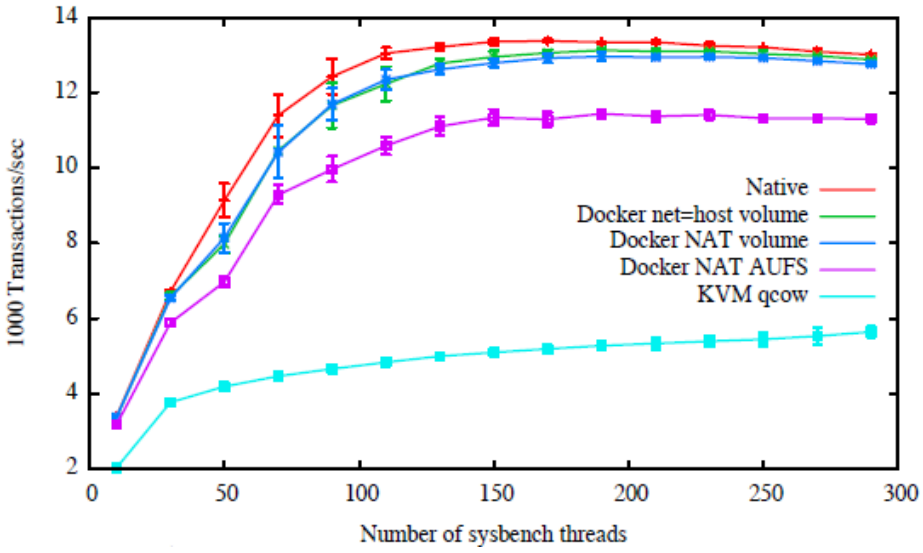

Figure 2.9 MySQL Throughput (transaction/s) vs. Concurrency [18]

The graph proves that Docker has similar performance to native, with the difference asymptotically approaching 2% at higher concurrency. KVM has much higher overhead, higher than 40% in all measured cases. In addition, the adoption of AUFS in Docker induces more overhead than that of host and NAT.

After a thorough observation of all the results, the authors conclude that Docker equals or exceeds KVM performance in every case they have tested. However, taken into account the case of AUFS, Docker is not without overheads. There are tradeoffs made between ease of management and performance.

All the benefits offered by containers are appealing to Cloudlets in Fog Computing. However, one essential feature that the virtual servers in Cloudlets need to have is the ability to migrate. Section 1.1 has mentioned two migration tools: Flocker and CRIU. Flocker is specifically designed for Docker containers. Since this thesis adopts LXC containers, CRIU is chosen as the migration tool.

2.4 Introduction to CRIU

CRIU is the acronym of Checkpoint/Restore in Userspace. It is a software tool that can be applied to freeze a running application (or part of it) and checkpoint it to a hard drive as a collection of files. Then these files can be restored, enabling the application to run from the point it was frozen at [4]. An illustrative description of CRIU is shown in Figure 2.10.

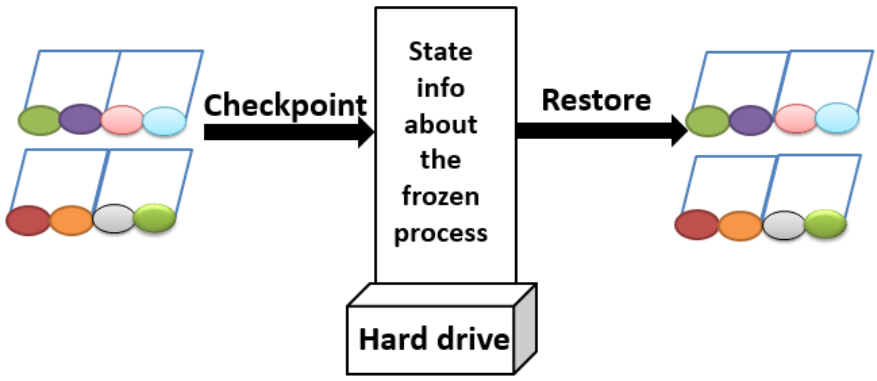

Figure 2.10 CRIU

The potential use cases of CRIU cover a wide range [4]:

1) Container live migration: This has been integrated with LXC and OpenVZ containers [19]. A running container can be checkpointed and restored on the remote host. When the container restarts, it will continue from where it gets stopped.
2) Slow-boot services speed up: The subsequent starts of a slow service can benefit greatly from the files being checkpointed.
3) Seamless kernel upgrade: CRIU offers the possibility of upgrading a kernel without stopping critical activity or reboot.
4) Network load balancing: This is a Transmission Control Protocol (TCP) connection repair project [20] that can cooperate with CRIU. After the TCP connection is dumped, CRIU will read the socket state and restore it back to let the protocol resurrect the data sequence.
5) HPC issues: CRIU has two roles: load balancing a computational task over a cluster and periodic state save to avoid re-computation in case of a cluster crash.
6) Desktop environment suspend/resume.
7) Process duplication: The function of CRIU in this case is similar to the system call fork().
8) Snapshots of applications.
9) Advanced debugging and testing: If there are some bugs reported on an application running on the production host, the developer could retrieve the application's states through CRIU and solve the issue on the remote host.

Currently CRIU is still under active development. This thesis focuses on its first use case. It should be noted that at CRIU's current stage, the container's migration down time is the duration from it being checkpointed to being restored on the remote host. In real life implementations, this time may be too long and may affect QoE. Based on this consideration, a new project called Process Hauler (P. Haul) [21] aims to achieve "live" migration by decreasing the duration when the container is stopped. This could be achieved using CRIU's pre-dump action. As P. Haul is not as mature as CRIU and the latter has already been integrated with LXC containers, this thesis chooses CRIU to perform the migration task.

2.5 Multipath TCP

Considering the fact that TCP establishes just one connection between the container and the remote host during migration, the success of migration is fully dependent on the stability of that connection. To address this challenge, this thesis leverages the protocol of MPTCP to increase the number of connections between the container and the remote host.

The prerequisite for appreciating the merits of MPTCP is to understand first how TCP works. Each TCP connection is identified by a 5-tuple, including source and destination IP address, source and destination port number and the underlying protocol identifiers. Any change in these five tuples will result in the re-establishment of TCP connection. In this context, MPTCP is developed to address this problem. Having borne in mind the predominant role of TCP, the design goals of MPTCP were

defined as follows [22]:

1) MPTCP should be capable of using multiple network paths for a single connection.
2) MPTCP must be able to use the available network paths at least as well as regular TCP, but without starving TCP.
3) MPTCP must be as usable as regular TCP for existing applications.
4) Enabling MPTCP must not prevent connectivity on a path where regular TCP works.

Figure 2.11 shows the place of MPTCP in the protocol stack.

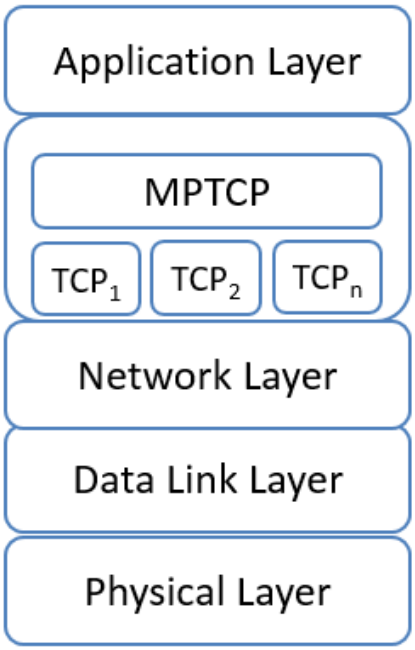

Figure 2.11 MPTCP in the Stack

Similar to TCP which establishes connection by a three-way handshake approach, MPTCP operates in the same way with the exception that MP_CAPABLE and a

random key are added to the SYN, SYN+ACK and ACK segments (see Figure 2.12).

With this MPTCP connection established, another subflow, i.e., one TCP connection joins in through a similar three-way handshake process but with MP_JOIN added to SYN, SYN+ACK, ACK segments. Now that the subflows have been established, hosts on both ends can exchange data over any of the active subflows.

To terminate MPTCP connection, similar to the operation of TCP, both graceful and abrupt release are supported. In the case of MPTCP, the FINISH (FIN) and RESET (RST) signals are used to gracefully and abruptly close the concerned subflow. To close the whole MPTCP connection which may have multiple subflows, DATA_FIN and MP_FASTCLOSE are adopted.

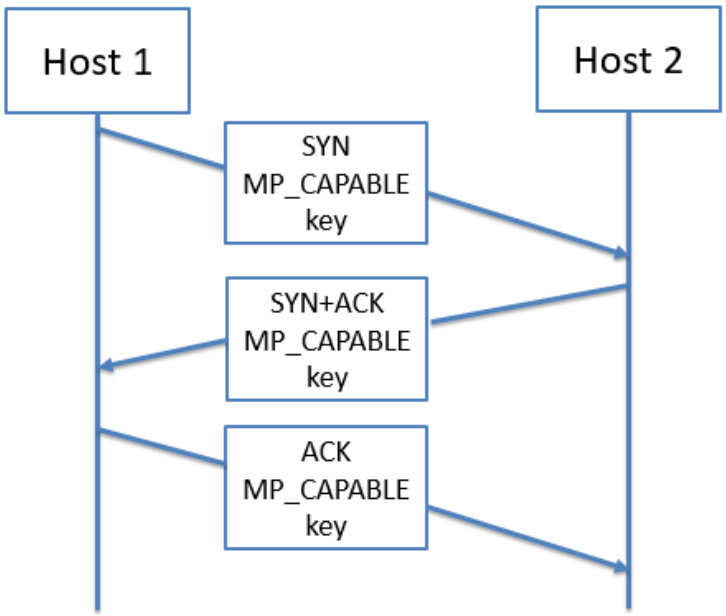

Figure 2.12 MPTCP Three-way Handshake

A summary of MPTCP signals are displayed in Table 2.4 [23].

Table 2.4 Summary of MPTCP Signals [23]

Signal	Name	Function
MP_CAPABLE	Multipath TCP Capable	Checks the capability of the end host on establishing a MPTCP connection
MP_JOIN	Join Connection	Adds additional subflow to existing MPTCP connection
REMOVE_ADDR	Remove Address	Removes failed subflow
MP_PRIO	Multipath Priority	Informs subflow priority
MP_FASTCLOSE	Fast Close	Closes MPTCP connection abruptly
ADD_ADDR22	ADD Address	Informs the availability of additional IP address to the paired host

2.6 Related Work

This section provides a literature review in the area of container technology, Fog Computing and MPTCP. Section 2.6.1 introduces a research that applies Docker to Edge Computing scenarios. Section 2.6.2 presents a research work on VM-based Cloudlets and discusses the general architectural component required for VM migration. Although the title points to VM, the authors clarify in a footnote that the model presented in the paper can also be generalized to container-based implementations. Section 2.6.3 shows the work done to perform VM migration between Cloudlets under MPTCP.

2.6.1 Evaluation of Docker as Edge Computing Platform

Ismail *et al*. in [24] explore the feasibility of using Docker as Edge Computing platforms. The term Edge Computing can be regarded as an alias of Fog Computing.

The authors listed five requirements that Docker can fulfill:

1) Deployment and Termination: By default, Docker allows remote deployment of container through an API, i.e., Shipyard WebUI or Docker Client. As for application setups, Dockerfile is used to integrate the necessary commands. Then Dockerfile is pushed to the local registry Docker Registry or Docker Hub for future use. For service termination, a newly-added feature by Docker will automatically clean up the container once it is no longer needed. However, this feature should be implemented with caution in case the data volume attached to the container will be lost.
2) Resource and Service Management; Service Registry, Resource Join and Leave the Edge: Docker Swarm supports service registry and discovery in order to manage services in containers. Figure 2.13 describes the setup of Docker Swarm cluster with Consul, one of the supported backend models providing cluster information.

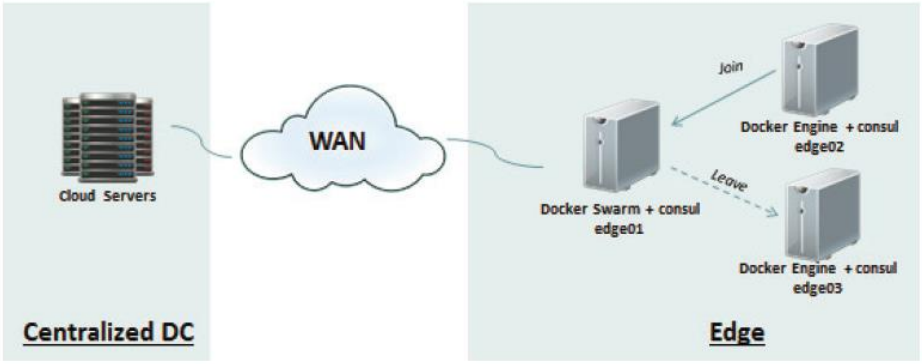

Figure 2.13 Edge Environment Setup of Docker Swarm and Consul as Service Discovery Backend [24]

3) Fault Tolerance: With the help of CRIU, Docker containers can be checkpointed

and restored, which makes service recovery possible. In addition, Flocker which is a Docker container data volume manager allows data volumes to follow the container. With the export/import functionality, Docker container can achieve an offline migration. However, as memory or CPU instruction cannot be preserved by this feature, the application within the container must be stateless and loosely coupled with any interacting components. One more mechanism which enhances fault tolerance is the distributed Docker-based edge zones across geographical sites. This reduces the impact brought by single point of failure.

4) Caching: The container's data volume could be cached to improve performance as well as to enhance capability of failure recovery. By creating a shared space for container's volumes, they could be dynamically collected at the edges.
5) Applications: The authors deploy Hadoop on their testbed and observe that installation is facilitated, setup time is shortened and configuration errors are reduced with the help of Docker and Docker Ferry.

In conclusion, the authors point out that Docker is a viable candidate for Edge Computing platforms by offering fast deployment, elasticity and good performance over VMs.

2.6.2 Towards VM Migration in Fog Computing

In [25], Bittencourt *et al*. elaborate on the roles of VMs in Fog Computing Cloudlets. In particular, the authors establish a Fog architecture to support both VM

and container migration and identify the key challenge as the understanding of how migration could be performed so that users do not notice performance degradation in their applications, regardless of the type of applications they are running. In order to make migration feasible, the authors are dedicated to define the architecture and discuss its components, interfaces and interactions, along with information needed to support the migration decision-making process.

The architecture adopts a four-tier structure. At the top of the architecture, Fog-enabled applications runs in a mobile device that has the ability to communicate with Cloudlets in the lower layer. At the mobile device layer, the Fog API enables data and computation offloading and migration control for the application developer. The Cloudlet layer controls user's data and initiates the migration. At the bottom layer, the Cloud layer, replication among data centers and load balancing within a data center are deemed as necessary. The service interaction between the Cloud layer and the Cloudlet layer can be achieved through exiting infrastructures, including Software as a Service (SaaS), Platform as a Service (PaaS), Infrastructure as a Service (IaaS) and Anything as a Service (XaaS), etc. Figure 2.14 summarizes the proposed architecture.

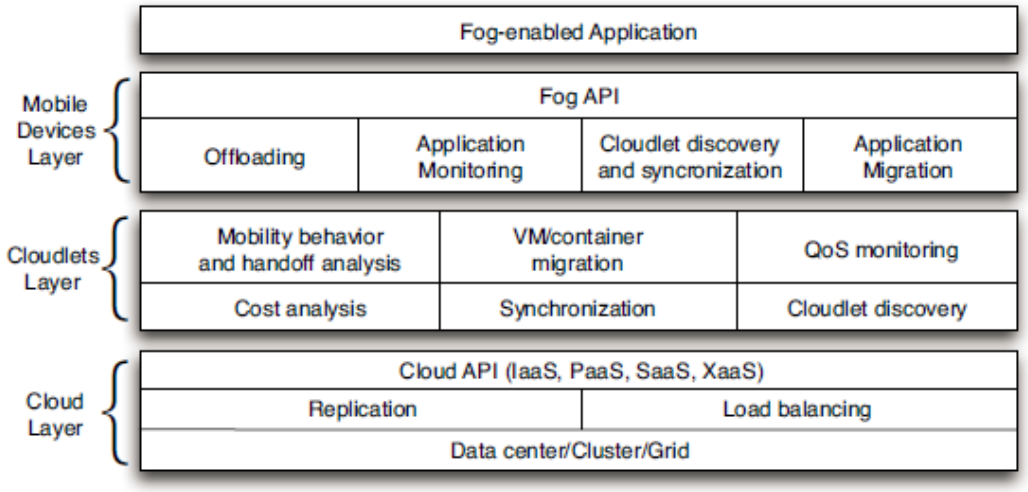

Figure 2.14 Fog Computing Layered Architecture [25]

After a detailed discussion of each layer's functionalities, the researchers make the conclusion that VM and container migration remains an important enabling capability in Fog Computing and security concern is a challenge since Cloudlets hosted by third party infrastructure providers needs to be trusted.

2.6.3 Seamless Live VM Migration for Cloudlet Users with Multipath TCP

The authors in [23] implement VM migration between Cloudlets. What's different from the approach proposed by [25] is the leverage of MPTCP during the migration process. As introduced in Section 2.5, MPTCP makes it possible to establish multiple subflows between two hosts. Here in paper [23], the two hosts are mobile device and the VM server instance inside the Cloudlet. As the user of mobile device moves from the range of Cloudlet1 to that of Cloudlet2, IP address of the user changes. Fortunately, under MPTCP, this change won't affect the connection with the VM

server instance. To emulate this scenario, the authors use bridges to represent interfaces (see Figure 2.15).

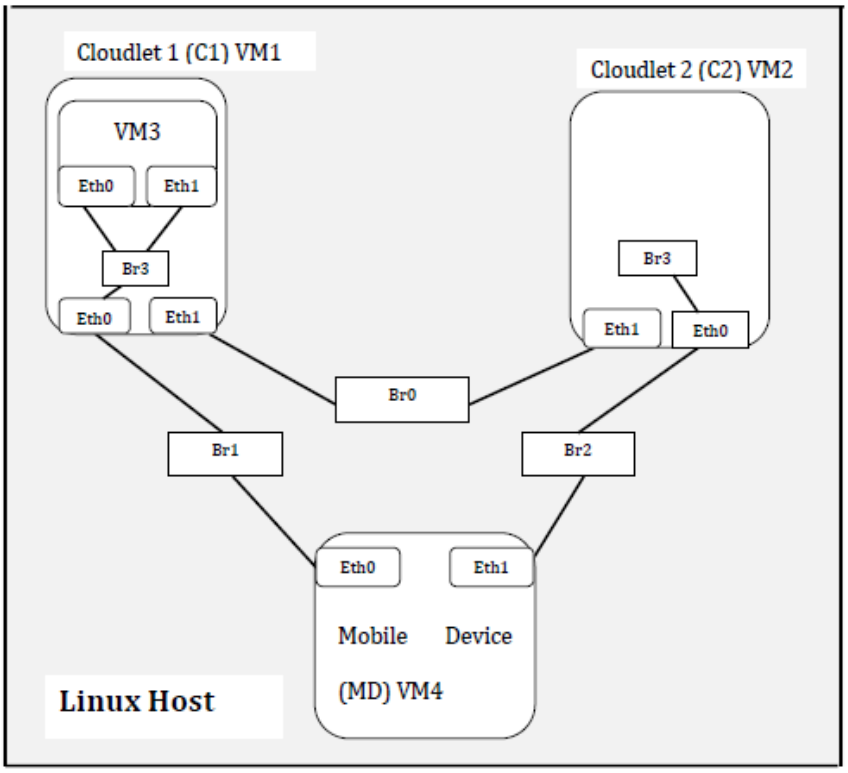

Figure 2.15 VMs and Bridged Networking inside a Linux Host [23]

In Figure 2.15, VM3 inside VM1 stands for the VM instance server used by the mobile device. VM4's Eth0 and Eth1 emulate a single Wi-Fi interface of a mobile device. VM4 accesses VM3 through one interface at the time (Eth0 or Eth1). The movement of the mobile device is emulated by configuring VM4's interfaces to UP or DOWN state. The following Figure 2.16 presents the throughput performance between VM3 and VM4 before and after VM3's migration from VM1 to VM2.

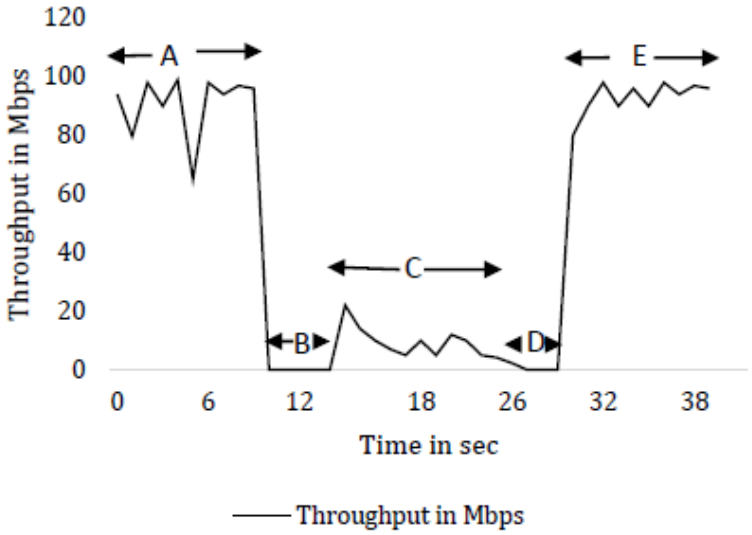

Figure 2.16 Throughput Performance [23]

The time slot A denotes the time when VM4 connects to VM3 within the scope of VM1. After some time, the Eth0 interface of VM4 is turned off, meaning that the mobile user starts to move out of the reach of VM1. There is a time of 5 seconds (time slot B) when VM4 is not connected to any of the nearby Cloudlets (VMs). The user (VM4) regains connection with VM3 through VM2 during time slot C, when the throughput decreases significantly as the traffic goes through WAN. At some point, because of the low throughput, VM3 will be inaccessible at the application level, as shown by time slot D. But this only lasts for 3 seconds, after which the mobile device is fully covered by VM2 and the migration of VM instance server (VM3) has finally completed.

2.7 Comparison with Related Work

This section points out the differences of this thesis compared to the related work presented in the previous section.

The authors in [24] choose Docker as Edge Computing platforms, while this thesis chooses LXC containers. Docker is a container tool built on top of LXC. One of the core ideas of Docker is to run a single application per container. However, LXC containers can support multiple processes on a single container. These two technologies have different strengths for different applications. For the purpose of this thesis, LXC has proven to be more suitable since it is close to operating-system-level virtualization and offers better migration support. Although the authors mention CRIU in [24], in practice the cooperation between CRIU and Docker containers is merely a proof-of-concept at this stage. And the export/import function of Docker cannot preserve application state information. Conversely, CRIU works better on LXC and can save the states of applications running inside. Even though Docker provides many tools to facilitate cluster management, which is a feature that LXC has yet to develop, LXC is still the appropriate choice since it aligns with the focus of this thesis—to gain insights into the feasibility of using container and performing container migration in Cloudlets.

As for [25], the authors conduct an empirical investigation in the Cloudlet layer, while this thesis is about practical implementations. This thesis shares with [25] the consensus that VM and container migration should be one of the essential focuses of Cloudlets. Additionally, this thesis also looks into the viability of LXC containers with regards to service provisioning. With all the enthusiasm around VM and container migration aside, the fundamental prerequisite is that they should be able to provide service in a satisfying manner. This thesis evaluates this aspect based on the

performance of CPU, memory, network and disk I/O. The experiment setup and results can be found in Chapter 4.

For the research work [23], there are two major differences as opposed to this thesis. First, this thesis selects containers instead of VMs for Cloudlet platforms. Experiment results shown in the Section 2.3.3 have proven that the intrinsic lightness of containers will make better use of the resources in Cloudlets. Second, the use of MPTCP in [23] targets the connection between the mobile device and the VM server instance, whereas MPTCP in this thesis improves fault tolerance of the connection between a container and the remote host and reduces migration time. The behaviors of the end users are beyond the scope of this thesis.

Chapter 3 Methodology and Approach

This chapter describes in detail how a LXC container-based Cloudlet works and how migration is operated and improved with the help of MPTCP. Section 3.1 introduces the framework of the Cloudlet model, including the configurations of containers. Section 3.2 explains how migration of LXC containers is achieved through the collective work of CRIU and *rsync* command. Based on the migration, section 3.3 expands on the topic by discussing how MPTCP makes the migration process more fault-tolerant and quicker. Finally, section 3.4 points out the current limitations in the area of LXC container migration.

3.1 LXC Container-based Cloudlet Framework

As indicated in the previous chapter, this thesis adopts LXC containers as Cloudlet platforms. Figure 3.1 illustrates the proposed framework.

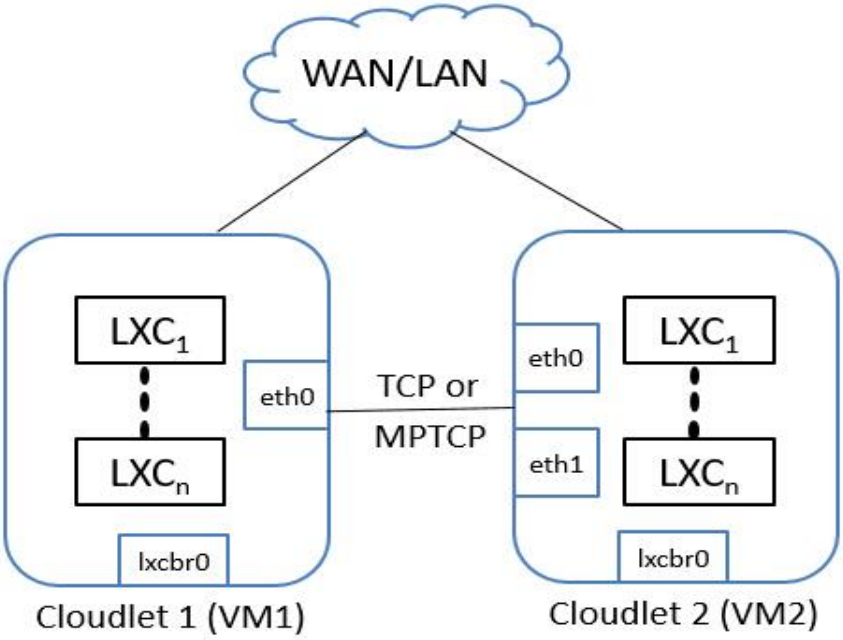

Figure 3.1 LXC Container-based Cloudlet Framework

In the above figure, both Cloudlet1 and Cloudlet2 are represented by VMs, with multiple LXC containers running inside. The considerations behind this design will be further elaborated in sub-section 3.1.1. The lxcbr0 interface (10.0.3.0/24) is LXC's default network bridge connecting all containers on the 10.0.3.0/24 LAN with their host. In Cloudlet2, the eth0 and eth1 interfaces are established for the purpose of container migration from Cloudlet1 to Cloudlet2. The network protocol during the migration process could be TCP or MPTCP depending on the kernel configurations of VMs.

3.1.1 Considerations behind Building LXC on Top of VMs

This thesis chooses to evaluate the effectiveness of LXC containers on top of a VM host, which is not an approach commonly adopted by container researchers. There are concerns that the overhead of VMs will pose a negative impact on the performance of containers. However, for the benefit of the research conducted in this thesis, the following considerations indicate that building LXC containers on top of VMs is a well-justified choice.

1) To show how a LXC container cooperates within a VM instead of a physical computer: Multiple experiments conducted on physical computers have proved that the lightness of containers offers them a significant advantage over VMs. However, given the fact that VMs are more financially feasible than physical hosts, it will be useful to test how a container performs on a VM host. Building another VM on top of a VM host, i.e., nested virtualization does not enjoy a wide range of

applications because of the high requirements of the physical and virtual hosts and the accumulated overheads incurred. On the contrary, building a container on top of a VM is common, since containers are lightweight. Therefore, the framework adopted in this thesis represents Cloudlets with VMs and service delivery platforms with LXC containers.

2) To emulate the limited capacities of Cloudlet servers: In addition to using VMs to function as virtual hosts, the second purpose of choosing VMs is to emulate the situation of a physical host with limited capacities. As pointed out in the previous chapter, unlike data centers which are facilities with powerful computing capacities, the composition of Cloudlet servers is diverse, ranging from small, local businesses to large, regional enterprises. The VMs, with a comparatively small number of CPUs, limited RAM and disk space could represent the small physical servers provided by local businesses. The evaluation of the performance of such VMs may shed light on the Cloudlet servers in real-world scenarios.

3) To mitigate the impact of possible security flaws: In the case of LXC containers, this thesis employs privileged containers which are defined as any container where the container uid 0 is mapped to the host's uid 0 [26]. The implication of such mapping is that there is a possibility that the container user may get full root privileges on the host with some manipulation. Since CRIU currently only supports privileged containers, in order to mitigate the potential risks, this thesis uses VMs to act as an extra layer of protection for the physical host. The stakes of messing around with root privileges on a VM host are much lower than that of a

physical computer. Consequently, even if some false operations have exposed the host to the container, catastrophic outcomes could be prevented by just shutting down or deleting the VM host. The Cloudlet service providers could continue to serve end users by ginning up a new VM.

3.1.2 LXC Container Network Configurations

The network configurations of LXC container are the practical implementations of the network namespace introduced in sub-section 2.3.1. Five types of network configurations [27] will be presented in this section. These configurations determine how a container interacts with the outside world through networking. When a container initiates the service delivery process, it selects one of the five types of container network configurations based on the specific requirements of the service:

1) Empty: Empty network type creates only the loopback interface, which means the container communicates only with itself and is not exposed on any network.
2) veth: This stands for virtual Ethernet and is the default network type for LXC containers with Ubuntu as a template. The veth works in the form of a pipe creating a peer network device with one side assigned to the container and the other side attached to a bridge specified with the *lxc.network.link* in the LXC container configuration file. By default, *lxcbr0*, as illustrated in Figure 3.1, is specified with *lxc.network.link*. Figure 3.2 shows how the veth type works.

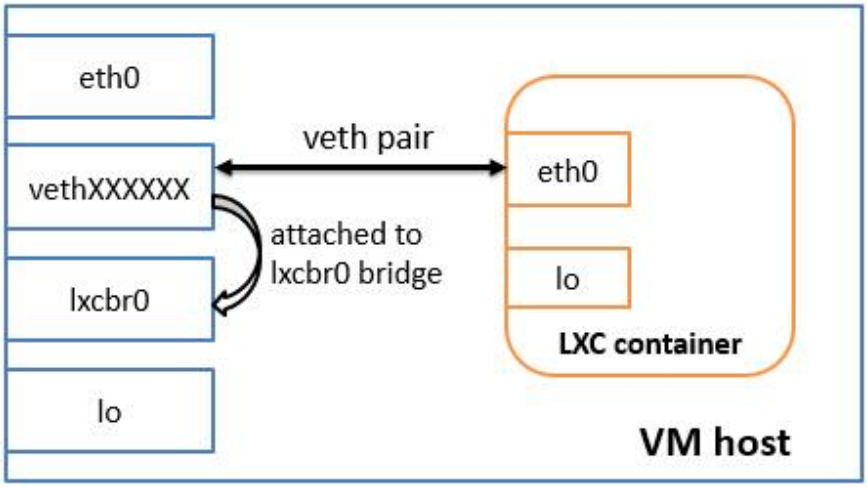

Figure 3.2 veth Network Type

The vethXXXXXX interface is the host's side of peer network device; its counterpart is the eth0 inside the container, which cannot be seen on the host because the container and the host belong to two network namespaces.

3) vlan: The virtual LAN (vlan) interface is linked with the interface (typically a physical interface) specified with *lxc.network.link* and assigned to the container. The vlan identifier is specified with the option *lxc.network.vlan.id*. This network type creates a regular vlan interface in the network namespace of a container.

4) macvlan: The macvlan network type takes a network interface specified with *lxc.network.link* and creates multiple virtual network interfaces with different MAC addresses assigned to them. There are three modes in which macvlan will use to communicate between macvlan on the same upper device: private, virtual Ethernet port aggregator (VEPA) and bridge. Private mode means any communication between devices on the same upper device is disallowed. VEPA mode assumes that the adjacent bridge returns all frames where both source and

destination are local to the macvlan port. Bridge mode directly delivers frames from one interface to another internally.

5) phys: Physical mode assigns an already existing interface specified with *lxc.network.link* to the container. After the container is booted up, the interface disappears on the host and reappears inside the container with the same IP address. If the IP address is routable, the container can establish communications with the outside world. This type comes in handy when a container runs applications like web server that needs to be accessed by edge users.

Since veth is the default network type for Ubuntu LXC containers and it's the most suitable type for the experiments conducted in this thesis, three containers adopt this type. Only one Tomcat web server container chooses the phys type because it allows the container to be accessible from other VMs on the same physical host.

3.2 LXC Container Migration

Mobility is one of the most prominent characteristics of Cloudlet service recipients. As a consequence, it is of great importance that LXC containers support such mobility by providing migration services between Cloudlets. This section proposes a possible solution of managing IP addresses of neighboring Cloudlets and explains the working mechanisms of LXC container migration.

Before any forms of migration takes place, the container should be aware of the IP address of the Cloudlet to which it will migrate. Since end users' movement

determines if migration is going to be conducted, the server container will not be informed until it detects that the network connection between the end user and the container has been severed. There lies three types of possibilities when such disconnection occurs:

1) The end user has manually cut off the connection regardless of its geological position, which could indicate that the end user finds the service unnecessary. In this case, a timeout session should be introduced so that containers with incomplete services can be deleted to release valuable resources for other ones.
2) The end user still remains within the range of Cloudlet, but some network or device error has caused the disconnection. In this case, the timeout session will be triggered while the container waits for the reconnection. If the recovery time is longer than the maximum timeout session, the end user has to establish new connections with the container and service delivery has to start from the beginning. If the recovery time is shorter than the maximum timeout session, the connection will be rebuilt and service resumed.
3) The end user has moved to the range of another Cloudlet and still needs the service. This is the case where migration should take place to transfer the existing container to the desired Cloudlet and knowledge of neighboring Cloudlet's IP addresses becomes vital. To cope with this problem, this thesis puts forward a solution based on a centralized administration of Cloudlets.

3.2.1 IP Addresses of Neighboring Cloudlets

Before the proposed solution is introduced, the definition of neighboring Cloudlets should be clarified. In this thesis, if the Round Trip Time (RTT) between two Cloudlets is less than 150ms, they are considered as neighboring Cloudlets. This figure of 150ms is presented in [13], where the authors list the mappings of response times with subjective impression (see Table 3.1).

Table 3.1 Mappings of Response Times with Subjective Impression [13]

Response Time	**Subjective Impression**
< 150ms	Crisp
150ms – 1s	Noticeable to annoying
1s – 2s	Annoying
2s – 5s	Unacceptable
> 5s	Unusable

Then in the paper of VM migration between Cloudlets [23], the authors take this idea and apply it to the theory of VM migration between Cloudlets. The authors state that VM migration is considered only if the RTT between Cloudlets is less than 150ms. Consequently, if the RTT between two Cloudlets exceeds 150ms, migration will be unnecessary and a new VM needs to be created to serve the end user.

For Cloudlets within the 150ms range, this thesis proposes a central administration technique, in which all neighboring Cloudlets get registered and all end users connected to these Cloudlets get assigned a unique identifier. After the registration, the Cloudlet manager will be notified of each Cloudlet's IP address/addresses and it will advertise this information to all the Cloudlets within its

management range.

Based on the proposed model, every time an end user establishes connections with a Cloudlet, the identifier of that user will be passed to the Cloudlet manager in a "Cloudlet—identifier" fashion so that the manager will keep track of which Cloudlets are serving which users. This design is both convenient for neighboring Cloudlets to know the trace of the end user and essential for Cloudlet owners to keep a record of any users using their services.

If an end user with identifier U1 connects with Cloudlet1 first and then moves to a new location to connect with Cloudlet2, the manager will notice that U1 has switched from Cloudlet1 to Cloudlet2 and will inform Cloudlet1 of this change. Since Cloudlet1 has already known Cloudlet2's IP address/addresses in advance, it will initiate the migration of the container which is serving the end user with identifier U1.

It should be noted that the establishment of the central Cloudlet manager is not just for migration purpose. As public service providers, Cloudlets need to be monitored and regulated by a certain kind of organization, whether it be governmental or private. The central manager tracks not only the information of IP addresses and user identifiers, but also the Cloudlet's owner, location, scale, privacy and security laws, etc. Figure 3.3 illustrates the proposed solution.

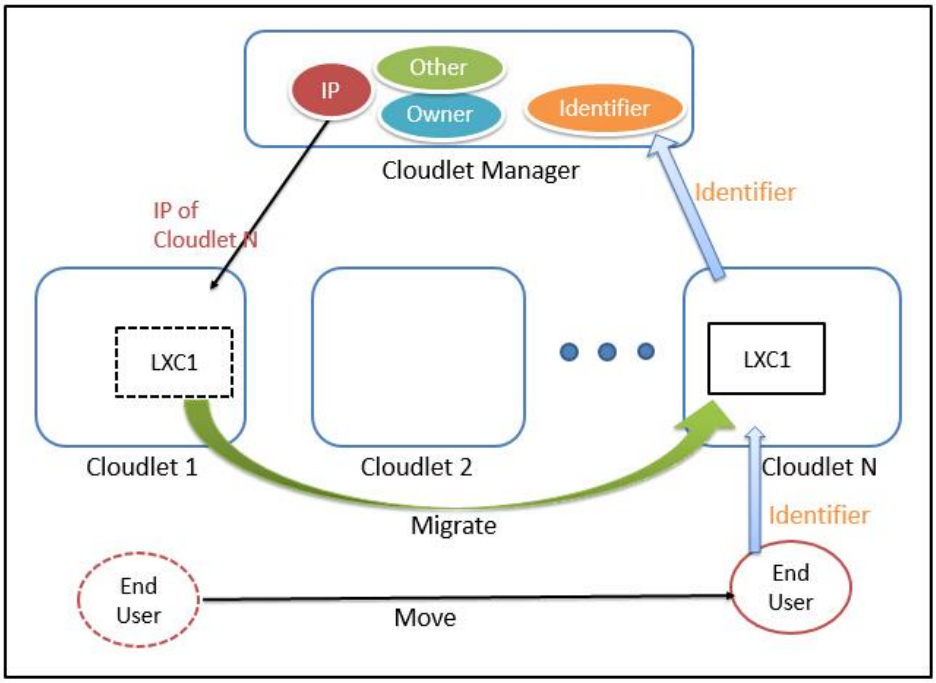

Figure 3.3 Central Administration of Neighboring Cloudlets

3.2.2 Migration Mechanism

After the IP addresses are retrieved from the manager, the next step is to conduct the migration. To migrate a LXC container, this thesis adopts a three-step approach based on a Linux script written by Anderson [8]. The three steps are explained as follows:

Step 1: Checkpointing the running container on the original host with the *lxc-checkpoint -s* command (see Figure 3.4). The CRIU utility introduced in section 2.4 offers the checkpointing functionality. With this command, CRIU will first stop the container and then dump all its related information to the directory specified by the –D option, denoting checkpoint directory. In most cases, the directory is chosen as */tmp/checkpoint*. The information being checkpointed is stored into multiple files with filename extension .img and a log file named *dump.log*. The log file contains

important messages of what is really happening during the checkpoint process and therefore serves as an irreplaceable reference for debugging.

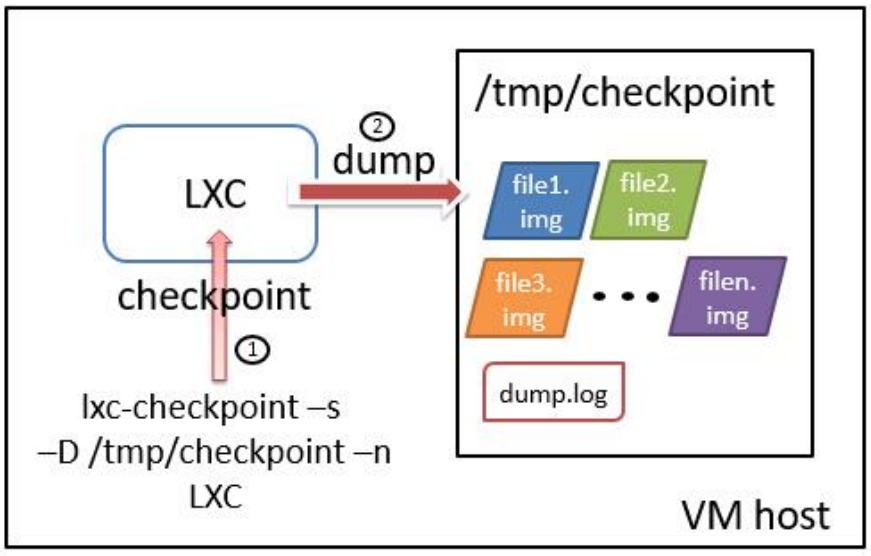

Figure 3.4 Step 1: Checkpointing Operation

Step 2: Copying two directories related with the container over to the remote host with the *rsync* command (see Figure 3.5) which is a fast and extraordinarily versatile file copying tool [28]. With this tool, */tmp/checkpoint* and */var/lib/lxc/container_name* are copied over to the next Cloudlet. As mentioned in step 1, the first directory contains all the state information related to the container at the time of its being checkpointed. The second directory is the default directory of privileged containers. It's the whole filesystem of a container being migrated. With these two directories, the container, along with its running applications will keep intact on the remote host. The files in the */tmp/checkpoint* directory are also prerequisite for the restore process in step 3.

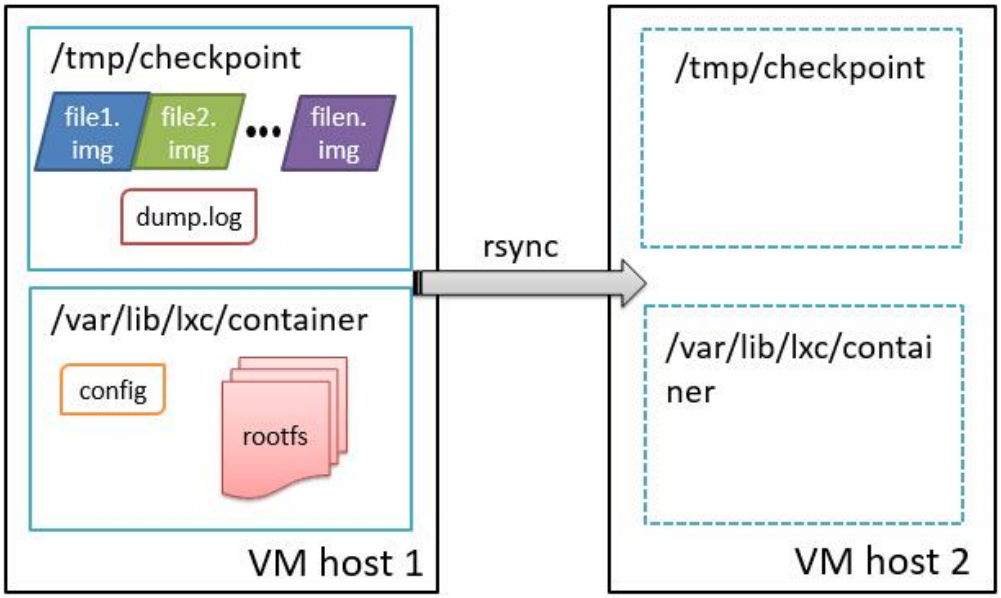

Figure 3.5 Step 2: Copying Operation

Step 3: Restoring the container on the remote host with the *lxc-checkpoint –r* command (see Figure 3.6). After the *rsync* command has finished copying the two directories, first, the container will be restarted with the restore functionality provided by CRIU in the *lxc-checkpoint –r* command. Then, this command retracts the information from the */tmp/checkpoint* directory to resume the container from the state where it gets checkpointed. During this process, a couple of new files and a log file named *restore.log* are added to the directory. At this point, the migration process is finished and the container can continue to serve the edge user on the new Cloudlet.

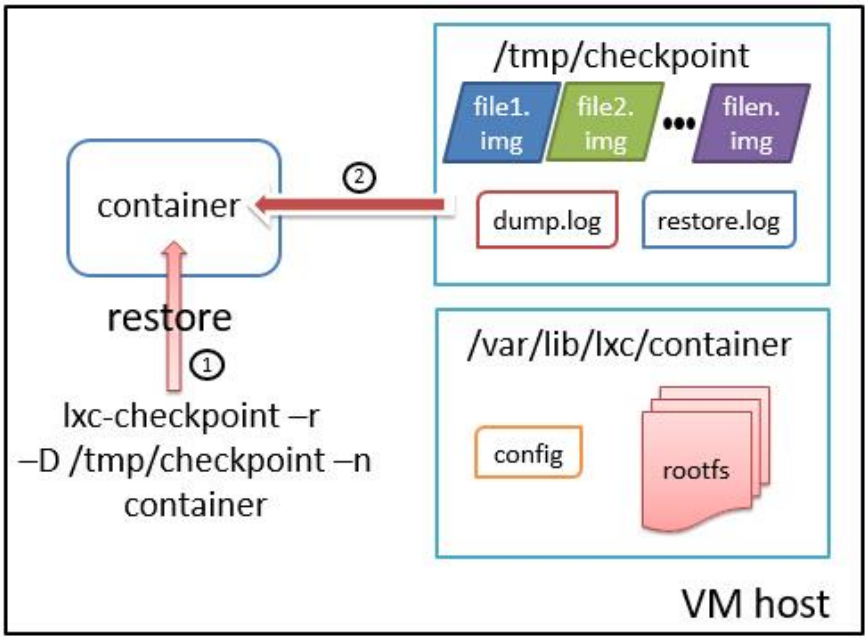

Figure 3.6 Step 3: Restoring Operation

3.3 Improving Resilience and Reducing Migration Time Using MPTCP

The success of the copying process in step 2 is dependent upon the stability of the network connection between two Cloudlets. As the traffic goes through the WAN during the copying operation, multiple factors could affect the connection. If it becomes highly congested, or even worse, gets broken, the service downtime of the end user will increase. In order to provide a better guarantee to this process and to make it less prone to failures, this thesis replaces the traditional TCP with MPTCP as the underlying protocol over WAN. The reasons are twofold: to improve resilience and to reduce migration time.

1) To improve resilience: MPTCP supports multipath paths, i.e., multiple TCP subflows between the source and destination. The increase of the number of

subflows between two Cloudlets reduces the possibility of connection lost due to network failure. Unlike the case with TCP where the transfer has to start all over again once the only connection gets broken, with MPTCP, the transfer could seamlessly switch to other working subflows, ensuring the continuity of the copying process.

2) To reduce migration time: Transmitting over multiple paths between Cloudlets also results in a faster transfer. Based on the MPTCP protocol, during data transmission, the majority of the traffic goes through whichever path with the shortest RTT. For Wi-Fi, the strongest signal strength or the highest bandwidth is preferred. The experiment results which will be shown in Chapter 4 prove that with MPTCP, the migration time is smaller than that of TCP.

3.3.1 Migration Model with MPTCP: Adding a New Subflow

To emulate multiple subflows supported by MPTCP, this thesis adopts a one-to-two mapping in which one interface on the source's side establishes connections simultaneously with two interfaces on the destination's side. The migration model is illustrated in Figure 3.7.

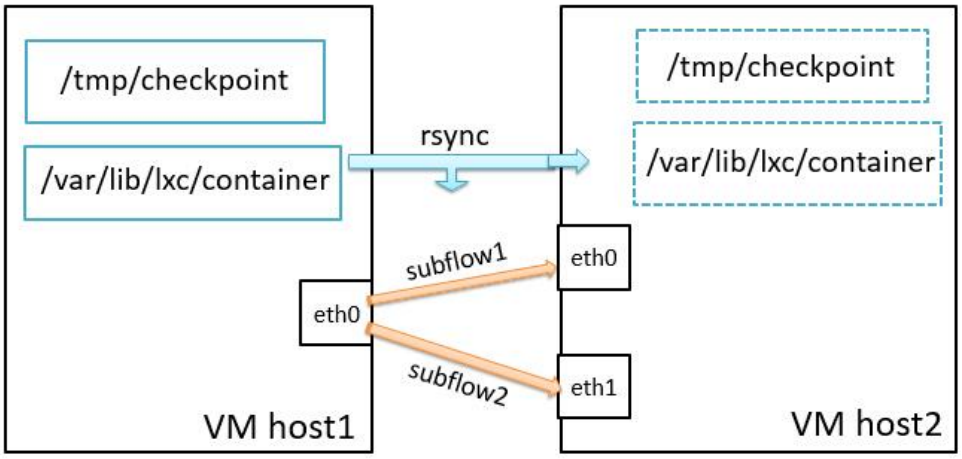

Figure 3.7 Migration Model with MPTCP: Two Subflows

In the above model, subflow1 is established first during the connection setup stage between VM host1 and VM host2. Since host1 receives host2's IP addresses from the Cloudlet manager, it knows that host2 has another IP address. Therefore, it attempts to initiate another subflow—subflow2 to join the connection. It should be noted that MPTCP designers are well aware of the potential risks of malicious subflows joining established connections. This means in the three-way handshake session, host1 needs to abide by the crypto algorithm negotiated between the two ends during the connection setup stage of subflow1. Internet Engineering Task Force (IETF) provides the description on the working mechanisms of adding a new subflow and removing an existing IP address [29].

As the session starts, the first SYN packet with MP_JOIN option carries a token, a random number and an address ID. The purpose of the token is to identify the MPTCP connection and to function as a cryptographic version of the receiver's key. The random number is used to prevent replay attacks on the authentication method.

Finally, the address ID shows the source IP address of the packet and also prevents setting up duplicate subflows on the same path.

When the other end receives the SYN packet with MP_JOIN option, it should respond with a SYN/ACK packet with MP_JOIN option that contains a random number and an algorithm-specific authentication code. If the token included in the first SYN packet is unknown to the receiver, it will send back a rest (RST) signal.

Now that SYN and SYN/ACK packets have been exchanged, the third ACK packet containing the sender's authentication information is sent by the subflow's initiator. This packet is important as it is the only time that the sender's packet includes algorithm-specific code. To make sure that it has successfully reached the destination, the recipient should send back a regular TCP ACK packet in response. Before this ACK packet is received, the subflow is set as PRE_ESTABLISHED. After it is received by the sender, the state of the subflow will be switched to ESTABLISHED. At this point, subflow2 has been added to the connection and is ready to transmit data between the two ends. Whether it's going to be put into use immediately or wait as a backup is decided by a flag bit set in the first SYN packet. The whole process of adding a new subflow is shown in the diagram of Figure 3.8.

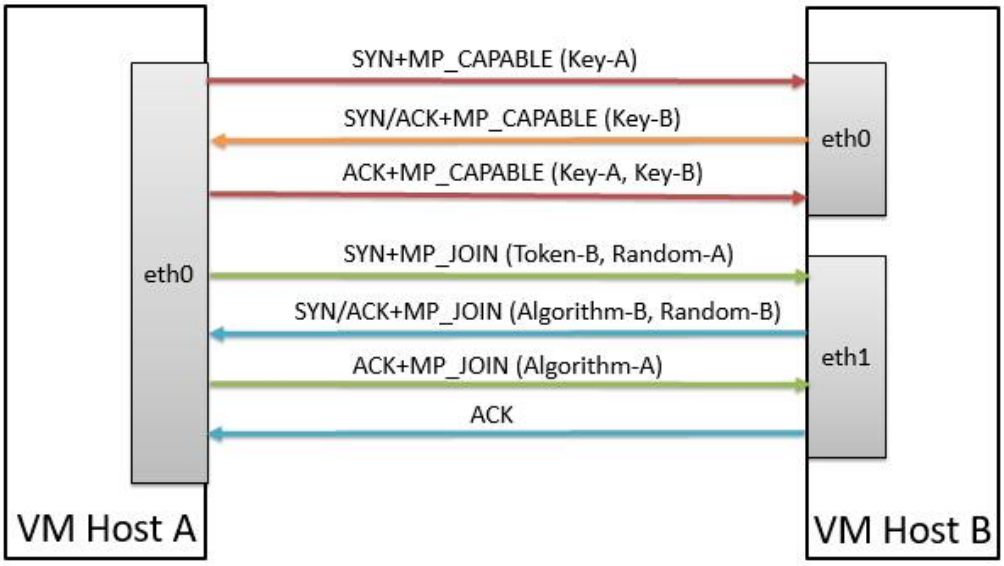

Figure 3.8 Adding a New Subflow

3.3.2 Migration Model with MPTCP: Removing an IP Address

To emulate an error causing the breakdown of a subflow, the experiment conducted in this thesis manually disconnects one of the interfaces on VM host2, which is analogous to removing an IP address. The migration model with severed subflow is illustrated in Figure 3.9.

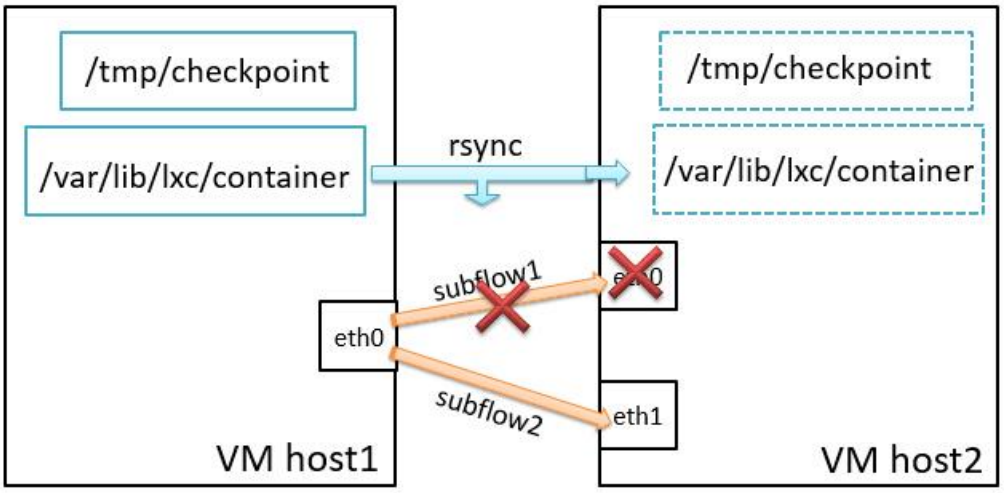

Figure 3.9 Removing an IP Address

In this model, during the lifetime of an MPTCP connection, eth0 on VM host2 becomes invalid. Since the interface is on host2, it should announce this so that its peer –host1 will remove the subflow related with this interface.

To remove the subflow, host1 will send the REMOVE_ADDR option to host2. Upon receiving this option, instead of shutting down the path immediately, host2 should first initiate a TCP keepalive on the path and wait for any responses indicating the path should not be removed. In addition, typical TCP validity tests such as ensuring sequence and ACK numbers are correct must also be undertaken on the subflow. If no responses regarding TCP keepalive are received, both hosts should then proceed to send RST signals on the affected subflow.

In the case of MPTCP, the RST signal works only on a subflow level. As multiple subflows may exist in a MPTCP connection, data transfer between two ends will not be interrupted and the data lost due to the failure of that interface can be retransmitted on other subflows. This demonstrates a higher degree of resilience than regular TCP where the removal of an IP address on the path will ultimately kill the connection.

3.4 Limitations of LXC Container Migration

Although the migration technique described in section 3.2 can fulfill the requirements of running an existing container on the remote host with preserved state information, it is not live migration because the container gets entirely suspended during the whole process using the current CRIU command. A bare Ubuntu container, i.e., no extra software packages installed, takes up around 350 MB, not to mention

other containers with large applications. As a consequence, there is a high requirement on the underlying network in order to minimize the transmission time to an acceptable level for end users. The implementation of MPTCP protocol is one solution, but it comes with tradeoffs. Moreover, it still remains a challenge to popularize MPTCP over the WAN.

The above problems have been more or less effectively solved in the case of VM migration. Having accumulated decades of experience, the area of VM migration is far more mature than that of containers.

In the following part, section 3.4.1 introduces the most popular live migration technique adopted by famous VM vendors. Section 3.4.2 summarizes the current progress of LXC container migration.

3.4.1 VM Live Migration Mechanism: Pre-copy

A VM is significantly larger than a container. In most cases, just the RAM size of a VM is equal to or larger than a container. Take the t2.nano instance provided by AWS for example, it has a RAM size of 512 MB, which already exceeds the size of a whole-packaged bare Ubuntu container.

For this reason, the storage and memory migration of a VM are considered separately. Over the years, researchers have come up with effective ways to migrate a running VM's memory, CPU and hardware device state. One of the most famous solutions is called pre-copy live migration which is implemented in most hypervisors such as VMWare, XEN and KVM [30].

Pre-copy live migration involves three phases: push phase, stop-and-copy phase and pull phase. In the push phase, all memory pages of the VM are copied to the remote host while it's still running on the source host. Then as the VM may generate dirty pages during its execution, these pages are sent to the remote host in an iterative fashion. In the stop-and-copy phase, the VM is temporarily paused and all remaining pages will be copied over to the remote host. In the pull phase, VM is resumed on the remote host. If it needs to access a page that's still on the previous host, this page will be pulled.

The most obvious advantage of the pre-copy live migration is that VM downtime is minimized to the smallest. During the iterations of dirty pages copying, the VM continues to run and will not be affected at all.

Besides this, there are other techniques to live migrate memory or storage part of a running VM and they have been incorporated into a few and simple commands.

3.4.2 Current LXC Container Migration Situation

As for LXC container migration, CRIU does a one-time dump of the container's states on the local host, which means the container is halted during this period. Since there is no support in CRIU to then transfer the container to the remote host, other copying commands like *rsync* has to cooperate with CRIU to complete the migration. This implies that the service downtime equals to the sum of the dump time, the transfer time and the restore time on the remote host. Although dump and restore operations take just a few seconds, the transfer time remains the bottleneck. As

containers grow bigger, and the applications they are running become latency-sensitive, this stop-and-copy technique is far from enough to equip containers with more complicated services.

Seeing this problem, container developers are currently working on a new project called P. Haul [21], which is mentioned in section 2.4. Based on the pre-dump action of CRIU, P. Haul works in a way similar to the pre-copy migration by conducting a series of incremental dumps while keeping the container running. This project has been integrated with LXD container [31], which is a container "hypervisor" and a new user experience for LXC. At the time of writing this thesis, both P. Haul and LXD container are under active development. Only with certain configurations will a live migration of LXD containers succeed. Both projects are still not ready for complicated, real-world scenarios.

Eventually more support will come, but for now, this thesis evaluates LXC container migration based on current available resource–CRIU and *rsync* command. The experiment results are presented in the next chapter.

Chapter 4 Experimentation and Analysis

This chapter presents the implementation of four LXC containers with different types of applications: a Tomcat web server, a PostgreSQL database, a C code emulating a computational-intensive task and an empty one with no extra software installed. The experiment setup and configurations are shown in Section 4.1. The performance of LXC containers compared with VMs is evaluated from various aspects, the results and discussions of which are presented in Section 4.2. Section 4.3 reveals the statistics relevant to container migration under regular TCP. Section 4.4 provides experimental results on how MPTCP improves the migration process.

4.1 Experiment Setup and Configurations

The experiments are conducted with two sets of setups. The first set is performed on a desktop with two VMs for LXC container evaluation and migration under TCP. The second set is performed on a laptop with two VMs for LXC container migration under MPTCP. The reason for these setups is because the desktop used for the first set is located in the university lab. Due to the constraints of network configuration, tests regarding MPTCP are not successful. The laptop used for the MPTCP test is located in a residential area with no extra network configuration. Experiments regarding MPTCP work out well. The following Table 4.1 and Table 4.2 show the information about the physical and virtual hosts for the first and second sets, respectively. In the rest of this chapter, the VMs for the first set will be referred to as VM1, VM2 and those used for the second set as VM3 and VM4.

Table 4.1 Host Information of First Set

	Physical Host (Desktop)	**Virtual Host (VM1)**	**Virtual Host (VM2)**
Processor	Intel Core i7-4770 (3.4 GHz, 4 cores, 8 threads)	2 vCPUs (2 threads)	2 vCPUs (2 threads)
RAM	16 GB	4 GB	4 GB
Disk	941 GB	20 GB	20 GB
Operating System	Windows 7 (64-bit)	Ubuntu 14.04 LTS (64-bit)	Ubuntu 14.04 LTS (64-bit)
Linux Kernel		3.19-generic	3.19-generic

Table 4.2 Host Information of Second Set

	Physical Host (Laptop)	**Virtual Host (VM3)**	**Virtual Host (VM4)**
Processor	AMD A4-5000 (1.50 GHz, 4 cores, 4 threads)	1 vCPU (1 thread)	1 vCPU (1 thread)
RAM	4 GB	1 GB	1 GB
Disk	500 GB	20 GB	20 GB
Operating System	Ubuntu 14.04 LTS (64-bit)	Ubuntu 14.04 LTS (64-bit)	Ubuntu 14.04 LTS (64-bit)
Linux Kernel	3.18-mptcp	3.19-generic/3.18-mptcp	3.19-generic/3.18-mptcp

The VM hypervisor on the desktop and the laptop is Virtualbox 5.0.16. LXC 1.1.5 and CRIU 1.7.2 are installed on all the four VMs. The latest release of MPTCP—version 0.90 [32] is installed on the laptop, VM3 and VM4.

The network configurations of both sets are illustrated in Figure 4.1, Figure 4.2 and Figure 4.3, respectively. In Figure 4.1, VM2 is configured with two interfaces, eth0 and eth1. The Network Address Translation (NAT) adaptor is for VM2 to access

the Internet and the host-only adaptor is for VM2 to be accessible by VM1 for migration. During the migration process, VM1 is the source and VM2 is the destination. In Figure 4.2 and Figure 4.3, in addition to wlan0, an external wireless adapter DWA-160B2 is used to generate another wireless interface ra0 on the host. Two Wi-Fi connections (wifi1 and wifi2) provided by two Internet Service Providers (ISPs), Rogers and Bell Canada, can be interchangeably connected to wlan0 and ra0. In Figure 4.2, on VM4, eth0 is bridged with wlan0, and eth1 with ra0, enabling VM4 to have two physical interfaces. From the eth0 interface of VM3, two subflows can be established with VM4. In Figure 4.3, either wlan0 or ra0 is bridged with eth0 on VM4, which is running under a TCP-kernel. The purpose of such design is to add comparisons to the case in Figure 4.2.

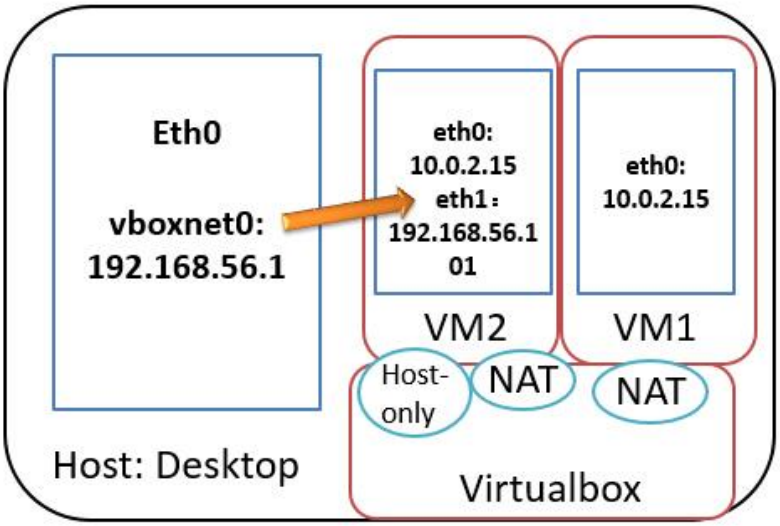

Figure 4.1 Network Configurations of First Set

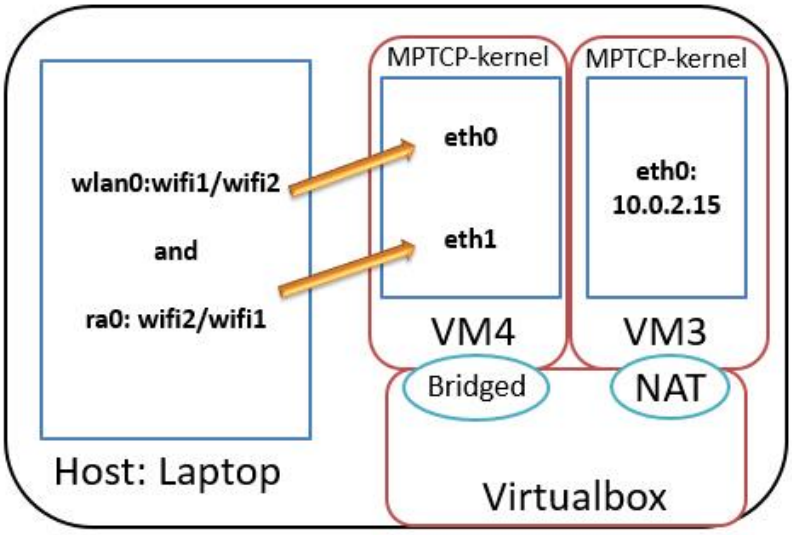

Figure 4.2 Network Configurations of Second Set: MPTCP Kernel

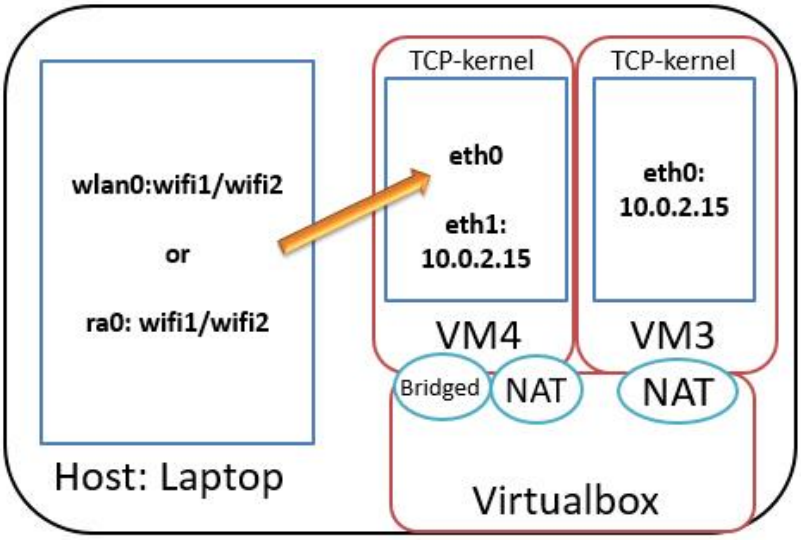

Figure 4.3 Network Configurations of Second Set: TCP Kernel

4.1.1 Reasons for Choosing the Four Containers

In the experiments performed from section 4.2.1 to 4.2.4, four LXC containers with Ubuntu 14.04 LTS as operating systems are chosen for evaluation and migration test under TCP. Table 4.3 shows their information.

Table 4.3 Four LXC Containers

Container	Application Type
Tomcat	Web Server
C code	Floating-point calculation
PostgreSQL	Database
Null	

The reasons for choosing these containers are two-fold. From the perspective of end users, the above applications have significant practical values. The container with Tomcat web server can provide a variety of services, including web page browsing, video streaming and downloading/uploading. Take video streaming for example. By pre-caching the videos from the content provider, the Tomcat container enables edge users to directly access the video within a one-hop distance. This not only improves user experience, but also releases WAN bandwidth.

Needless to say that PostgreSQL has a wide application as a database server, which offers services related to data analysis, storage, data manipulation, archiving, etc. One of the many use cases of database servers is in shopping malls, where customers could retrieve location-specific information from the database container.

The container with C code runs a floating-point two-dimensional array division for one million times. This code emulates a CPU-intensive task in real life. These tasks could take the form of face recognition, video and graphic manipulation, and so on. In fact, the ability to process highly-computational applications is one of the expectations of Cloudlet servers.

Lastly, the null container is included to demonstrate the intrinsic overhead of a LXC container with Ubuntu 14.04 LTS as template.

From the perspective of performance, Tomcat web server and PostgreSQL database stress memory and block I/O and display a high disk usage. The C code consumes a lot of CPU power when it's running. By comparing their performance results with VMs, this thesis offers a comprehensive view on the benefits of containers.

4.2 Performance of LXC Containers

The following subsections present the experimental results of LXC container performance with the help of benchmark tools. During the benchmark tests from sub-section 4.2.1 to sub-section 4.2.4, each container is also running their own application. For comparison, the performance statistics of VMs running the same applications as containers are also collected. The figures presented below are the average values of ten tests for each experiment. Standard deviation and confidence intervals (95%) are also calculated and presented in the following tables. The t-distribution is chosen to calculate the confidence intervals because the sample size is small [33].

It should be noted that only one benchmark tool is adopted for each test. Since there are intrinsic limitations of benchmark tools, the following results should be viewed critically. However, from the results, we can still deduce valuable implications and discover insightful patterns.

4.2.1 CPU Performance

The benchmark tool of CPU performance is *sysbench* 0.4.12, which is a modular, cross-platform and multi-threaded benchmark tool for evaluating OS parameters [34]. The test is to measure the time elapsed to pick out all the prime numbers of 20,000 random numbers. The less time the system takes, the better its CPU performance is. Figure 4.4 shows the results from four containers and VMs and Table 4.4 presents the standard deviation and confidence intervals (95%).

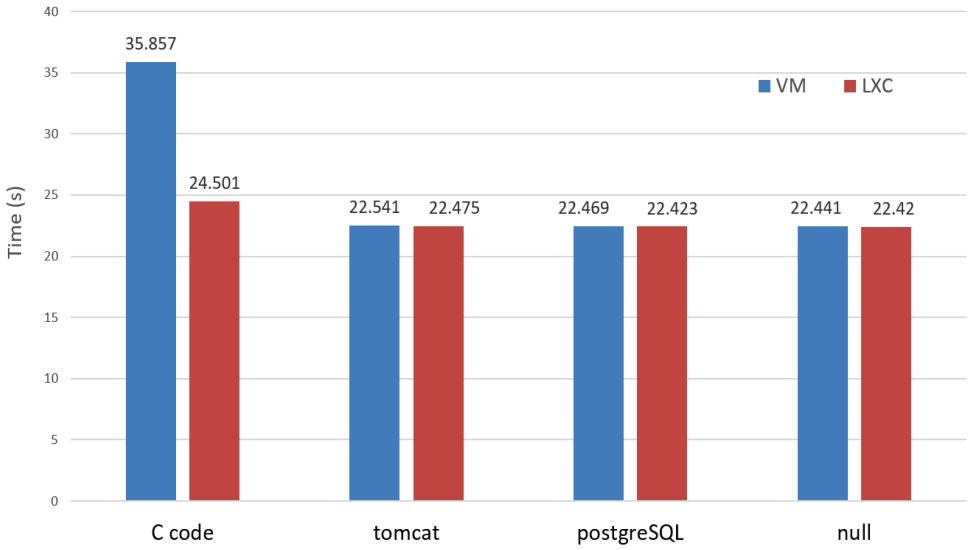

Figure 4.4 CPU Performance

Table 4.4 Results of CPU Performance (second)

		Mean	Standard Deviation	Confidence Interval (95%)
C code	VM	35.857	1.3329	35.857 ± 0.9535
	LXC	24.501	0.9070	24.501 ± 0.6488
Tomcat	VM	22.541	0.1088	22.541 ±0.0779
	LXC	22.475	0.0685	22.475 ± 0.0490
PostgreSQL	VM	22.469	0.1380	22.469 ± 0.0987
	LXC	22.423	0.0243	22.423 ± 0.0174
Null	VM	22.441	0.0501	22.441 ± 0.0358
	LXC	22.420	0.0311	22.420 ± 0.0222

Figure 4.4 indicates that for Tomcat, PostgreSQL and null pairs, CPU performance is almost equal. This is because the Tomcat and PostgreSQL applications are in the idle state, whereas the C code is running a CPU-intensive program. The more CPU the applications consume, the greater the differences between the performance of containers and VMs. The null pair displays the smallest difference with only 0.02s, while the C code VM takes 11.356 more seconds to finish the test than the container does. In Table 4.4, the standard deviation and confidence intervals of four LXC containers are smaller than those of VMs. In this case we can conclude that container performs better than VM in terms of CPU performance and its advantage becomes more obvious as the applications get more CPU-intensive.

4.2.2 Memory Performance

The benchmark tool of memory performance is *mbw*, short for memory bandwidth. It determines available memory bandwidth by copying large arrays of data into memory. One of the advantages of this tool is that it is neither tuned to extremes, nor aware of hardware architecture, which models the behavior of real applications [35]. To ensure accuracy, the swap feature has been turned off and caches have been cleared before each round of experiment.

By default, *mbw* performs three tests: MEMCPY, DUMB and MCBLOCK. They evaluate the available memory bandwidth by copying random bytes of memory, assigning elements in the second array to the first one, and copying block size of memory, respectively. The memory size is chosen as 2GB. Since each container and

VM has a total of 4GB RAM, after the applications and the systems consume a certain amount of memory, 2GB is the maximum available memory to be allocated to the test. The memory block size for MCBLOCK test is 262,144 bytes, a random number chosen by the system. The results are shown in Figure 4.5, Figure 4.6 and Figure 4.7. The results' standard deviation and confidence intervals (95%) are shown in Table 4.5, Table 4.6 and Table 4.7.

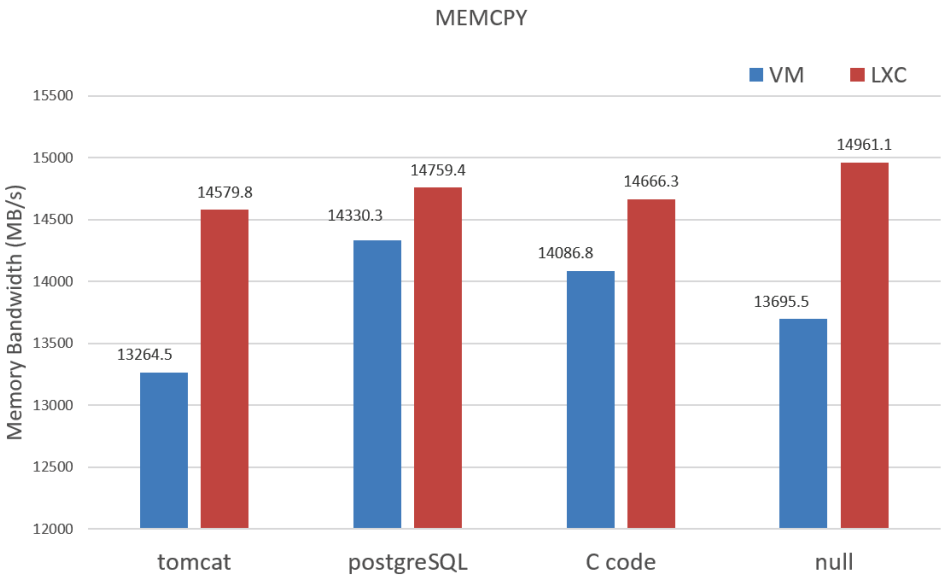

Figure 4.5 MEMCPY Test

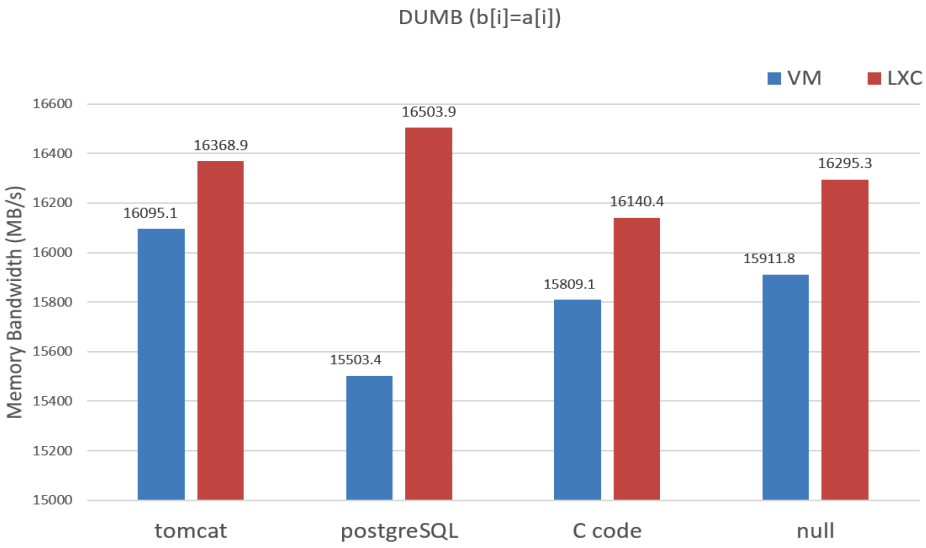

Figure 4.6 DUMB (b[i]=a[i]) Test

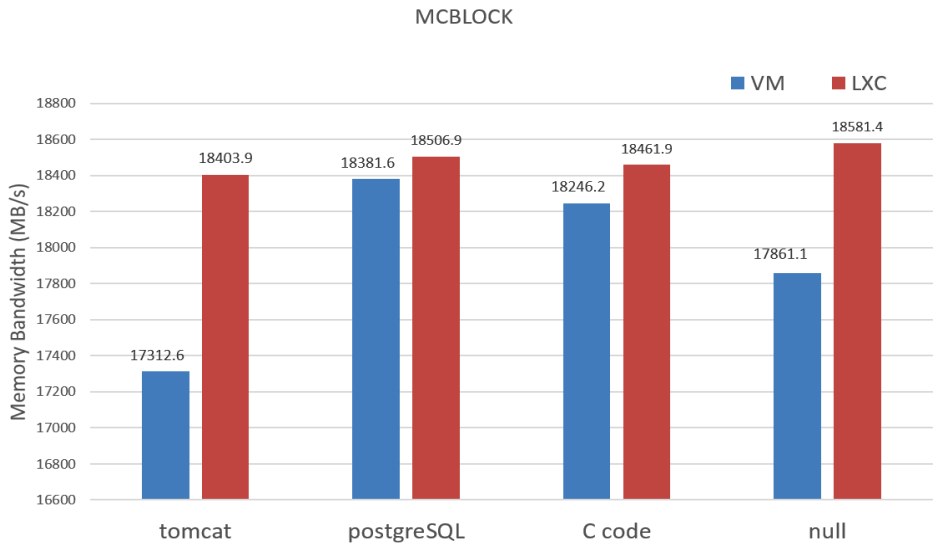

Figure 4.7 MCBLOCK Test

Table 4.5 Results of MEMCPY Test (MB/s)

MEMCPY		**Mean**	**Standard Deviation**	**Confidence Interval (95%)**
Tomcat	VM	13264.5	1194.27	13264.5 ± 854.33
	LXC	14579.8	524.46	14579.8 ± 375.18
PostgreSQL	VM	14330.3	841.68	14330.3 ± 646.97
	LXC	14759.4	792.28	14759.4 ± 566.76
C code	VM	14086.8	1506.87	14086.8 ± 1158.28
	LXC	14666.3	509.50	14666.3 ± 364.48

Null	VM	13695.5	665.39	13695.5 ± 475.99
	LXC	14961.1	1235.12	14961.1 ± 883.55

Table 4.6 Results of DUMB Test (MB/s)

DUMB (b[i]=a[i])		**Mean**	**Standard Deviation**	**Confidence Interval (95%)**
Tomcat	VM	16368.9	576.80	16368.9 ± 412.62
	LXC	16095.1	477.85	16095.1 ± 341.83
PostgreSQL	VM	15503.4	1539.20	15503.4 ± 1183.13
	LXC	16503.9	990.71	16503.9 ± 708.71
C code	VM	15809.1	704.90	15809.1 ± 504.25
	LXC	16140.4	1000.82	16140.4 ± 769.30
Null	VM	15911.8	1027.66	15911.8 ± 735.14
	LXC	16295.3	733.25	16295.3 ± 524.53

Table 4.7 Results of MCBLOCK Test (MB/s)

MCBLOCK		**Mean**	**Standard Deviation**	**Confidence Interval (95%)**
Tomcat	VM	17312.6	882.64	17312.6 ± 631.40
	LXC	18403.9	699.43	18403.9 ± 500.34
PostgreSQL	VM	18481.6	574.43	18481.6 ± 441.54
	LXC	18506.9	1055.46	18506.9 ± 755.03
C code	VM	18246.2	1008.57	18246.2 ± 721.49
	LXC	18461.9	920.66	18461.9 ± 707.68
Null	VM	17861.1	1015.53	17861.1 ± 726.47
	LXC	18581.4	784.96	18581.4 ± 561.52

The above Figure 4.5 to Figure 4.7 show that the differences between containers and VMs vary according to application types. In all of the three tests, the memory bandwidth of LXC containers is higher than that of VMs, which indicates a better performance on the container's side. The bandwidth of MCBLOCK test is noticeably higher than those of the other two because it copies memory by blocks. In Table 4.5, only the null container containers display a larger standard deviation and confidence

intervals than its VM. In Table 4.6, it's the C code container which exceeds its VM, and in Table 4.7, the standard deviation of the PostgreSQL container is almost twice as much as that of the VM. In general, containers outperform VMs, but in some cases, as discussed above, VMs have a more stable performance in the memory tests.

4.2.3 Disk I/O Performance

The benchmark tool to measure disk I/O performance is *bonnie++*, which is aimed at characterizing hard drive and filesystem by performing a number of simple tests [36]. This thesis conducts the tests of sequential output (by block), sequential input (by block) and random seeks on the four containers and VMs. The display of a higher speed of these tests indicates a better disk I/O performance of the system. To prevent the occurrence of caching, the recommended test file size should at least double the RAM size of the system. Since the RAM size of each container and VM is 4GB, the test file size of the experiments in this thesis is chosen as 10GB. Results are shown in Figure 4.8 and Figure 4.9. Statistics of standard deviation and confidence intervals are shown in Table 4.8, Table 4.9 and Table 4.10.

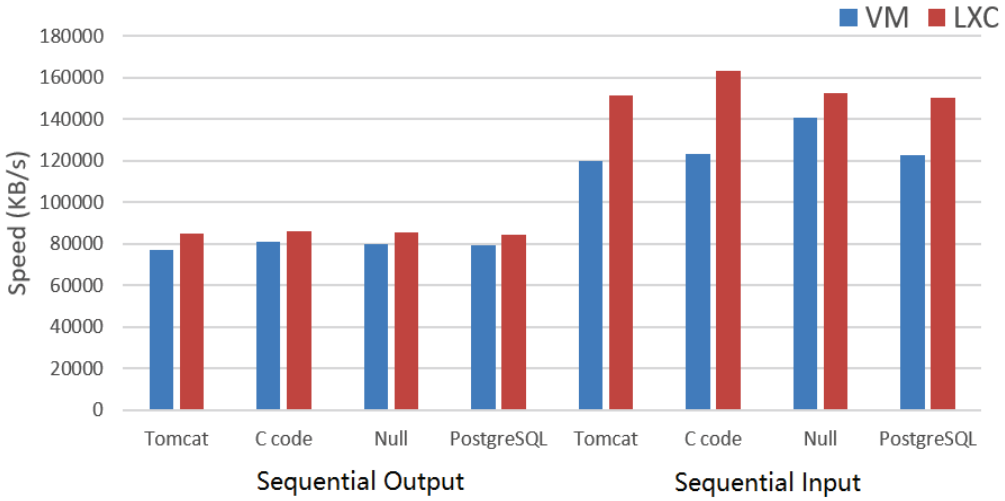

Figure 4.8 Sequential Output and Sequential Input

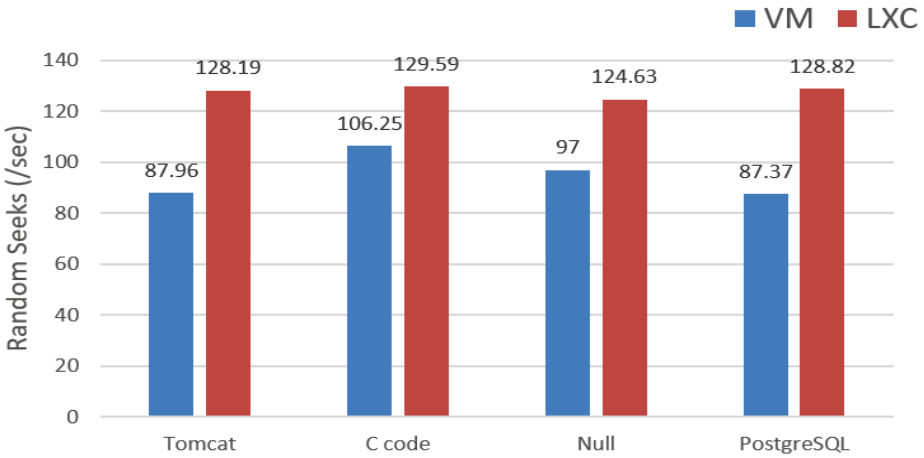

Figure 4.9 Random Seeks

Table 4.8 Results of Sequential Output Test (KB/s)

Sequential Output		**Mean**	**Standard Deviation**	**Confidence Interval (95%)**
Tomcat	VM	77204.5	3676.57	77204.5 ± 2630.06
	LXC	84750.1	2506.22	84750.1 ± 1792.84
C code	VM	81064.2	3463.85	81064.2 ± 2477.89
	LXC	86161.1	3170.22	86161.1 ± 2267.84
Null	VM	79752.1	2895.78	79752.1 ± 2071.52
	LXC	85370.6	2892.36	85370.6 ± 2069.07
PostgreSQL	VM	79191.0	3953.57	79191.0 ± 2828.21
	LXC	84028.4	2971.09	84028.4 ± 2125.39

Table 4.9 Results of Sequential Input Test (KB/s)

Sequential Input		Mean	Standard Deviation	Confidence Interval (95%)
Tomcat	VM	119561.9	15752.15	119561.9 ± 11268.41
	LXC	151035.7	17897.34	151035.7 ± 12802.98
C code	VM	123216.8	22522.71	123216.8 ± 16111.77
	LXC	163036.6	23696.84	163036.6 ± 16951.70
Null	VM	140601.1	20811.61	140601.1 ± 14887.73
	LXC	152448.2	22241.29	152448.2 ± 15910.46
PostgreSQL	VM	122695.6	23327.24	122695.6 ± 16687.3
	LXC	150155.8	26728.74	150155.8 ± 19120.59

Table 4.10 Results of Random Seeks Test (/s)

Random Seeks		Mean	Standard Deviation	Confidence Interval (95%)
Tomcat	VM	87.96	17.26	87.96 ± 12.34
	LXC	128.19	17.16	128.19 ± 12.27
C code	VM	106.25	10.21	106.25 ± 7.31
	LXC	129.59	17.45	129.59 ± 12.48
Null	VM	97.00	9.81	97.00 ± 7.02
	LXC	124.63	5.53	124.63 ± 3.96
PostgreSQL	VM	87.37	5.59	87.37 ± 4.00
	LXC	128.82	5.83	128.82 ± 4.17

From the above Figure 4.8 and Figure 4.9, we can observe an obvious advantage in the disk performance of containers as opposed to VMs. Their differences, though not substantial in terms of sequential output, have widened in the other two tests. In the case of sequential output (see Table 4.8), all LXC containers have a smaller standard deviation and confidence intervals than their VM counterparts. However, in sequential input (see Table 4.9), the trend is reversed, with all VMs outperforming containers. As for random seeks (see Table 4.10), Tomcat and null containers show an advantage over VMs, while the other two containers—C code and PostgreSQL—are

competed by VMs. The above comparisons indicate that in the three disk I/O tests, while LXC containers display a higher speed on average, they are completely outperformed by VMs in sequential input test in terms of standard deviation and confidence intervals and equaled with VMs in the test of random seeks.

4.2.4 Network Throughput

The benchmark tool used to measure network throughput is *netperf*, which provides tests for unidirectional throughput and end-to-end latency [37]. During the course of the throughput test, the four containers and VMs are running the *netperf* client and a VM on the same physical host is running the *netperf* server. Such an arrangement is to measure TCP throughput in megabits per second (Mbps) by sending and receiving TCP packets within a fixed duration (10 seconds, by default).

As the client and server are running on the same physical host, in order to better emulate a real-world scenario, the parameters of RTT and bandwidth are shaped to be close to the potential use cases of Cloudlets. As a result, RTT is set as 20ms, 60ms and 120ms with the network emulator (*netem*) function of the *tc* command in Linux. The theory of 150ms mentioned in section 3.2.1 is applied when choosing RTT parameters. Since Cloudlets outside the 150ms range are not candidates for any network communications, their parameters are not taken into account when designing experiments to evaluate LXC containers. As for the bandwidth, which is set as 600Mbps, 300Mbps and 100Mbps, the *wondershaper* command is used. Due to the fact that the maximum bandwidth between the VMs running *netperf* client and server

is between 600 and 650Mbps, the highest bandwidth chosen for the experiment is 600Mbps. Besides, as gigabit wireless connection has become available on the market, but not yet prevalent, a bandwidth range between 100Mbps and 600Mbps is a reasonable assumption. The experiment outcomes are shown in Figure 4.10 and Figure 4.11 and the statistics of standard deviation and confidence intervals are presented in Table 4.11 to Table 4.16.

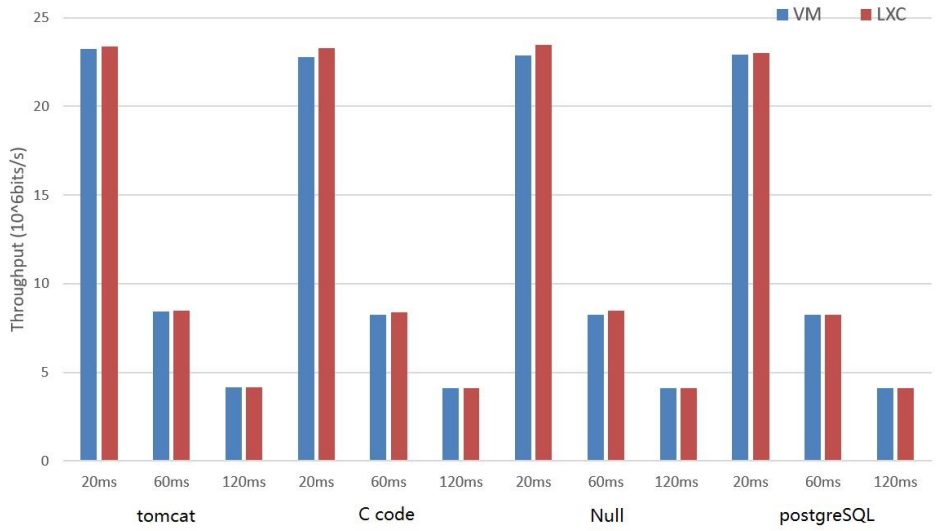

Figure 4.10 Network Throughput with Changed RTT

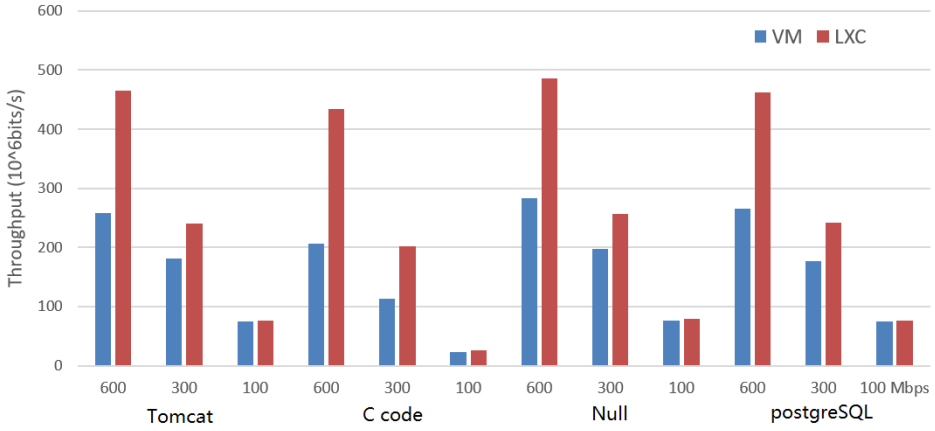

Figure 4.11 Network Throughput with Changed Bandwidth

Table 4.11 Results of Network Performance at RTT=20ms (Mb/s)

RTT=20ms		Mean	Standard Deviation	Confidence Interval (95%)
Tomcat	VM	23.24	0.011	23.24 ± 0.008
	LXC	23.39	0.012	23.39 ± 0.008
C code	VM	22.75	0.064	22.75 ± 0.046
	LXC	23.30	0.016	23.30 ± 0.011
Null	VM	22.87	0.019	22.87 ± 0.013
	LXC	23.48	0.016	23.48 ± 0.011
PostgreSQL	VM	22.89	0.019	22.89 ± 0.014
	LXC	22.98	0.015	22.98 ± 0.011

Table 4.12 Results of Network Performance at RTT=60ms (Mb/s)

RTT=60ms		Mean	Standard Deviation	Confidence Interval (95%)
Tomcat	VM	8.44	0.027	8.44 ± 0.019
	LXC	8.47	0.020	8.47 ± 0.014
C code	VM	8.23	0.019	8.23 ± 0.014
	LXC	8.38	0.018	8.38 ± 0.013
Null	VM	8.25	0.022	8.25 ± 0.016
	LXC	8.48	0.016	8.48 ± 0.011
PostgreSQL	VM	8.23	0.023	8.23 ± 0.016
	LXC	8.26	0.017	8.26 ± 0.012

Table 4.13 Results of Network Performance at RTT=120ms (Mb/s)

RTT=120ms		Mean	Standard Deviation	Confidence Interval (95%)
Tomcat	VM	4.15	0.007	4.15 ± 0.005
	LXC	4.16	0.005	4.16 ± 0.004
C code	VM	4.11	0.005	4.11 ± 0.004
	LXC	4.11	0.005	4.11 ± 0.004
Null	VM	4.11	0.010	4.11 ± 0.007
	LXC	4.12	0.006	4.12 ± 0.004
PostgreSQL	VM	4.11	0.012	4.11 ± 0.008
	LXC	4.12	0.008	4.12 ± 0.006

Table 4.14 Results of Network Performance at BW=600Mbps (Mb/s)

BW=600Mbps		Mean	Standard Deviation	Confidence Interval (95%)
Tomcat	VM	258.88	9.061	258.88 ± 6.482
	LXC	465.68	6.882	465.68 ± 4.923
C code	VM	206.52	7.516	206.52 ± 5.377
	LXC	434.73	8.622	434.73 ± 6.168
Null	VM	283.29	4.600	283.29 ± 3.290
	LXC	485.22	4.395	485.22 ± 3.144
PostgreSQL	VM	264.97	6.971	264.97 ± 4.987
	LXC	462.23	4.076	462.23 ± 2.916

Table 4.15 Results of Network Performance at BW=300Mbps (Mb/s)

BW=300Mbps		Mean	Standard Deviation	Confidence Interval (95%)
Tomcat	VM	181.20	6.176	181.20 ± 4.418
	LXC	240.43	4.341	240.43 ± 3.105
C code	VM	112.56	5.565	112.56 ± 3.981
	LXC	202.10	4.978	202.10 ± 3.561
Null	VM	197.88	6.330	197.88 ± 4.528
	LXC	257.28	5.143	257.28 ± 3.679
PostgreSQL	VM	177.40	5.316	177.40 ± 3.803
	LXC	242.61	6.038	242.61 ± 4.319

Table 4.16 Results of Network Performance at BW=100Mbps (Mb/s)

BW=100Mbps		Mean	Standard Deviation	Confidence Interval (95%)
Tomcat	VM	74.07	1.704	74.07 ± 1.291
	LXC	76.71	1.053	76.71 ± 0.753
C code	VM	23.32	1.150	23.32 ± 0.823
	LXC	25.50	1.125	25.50 ± 0.805
Null	VM	76.27	0.972	76.27 ± 0.695
	LXC	78.88	1.052	78.88 ± 0.753
PostgreSQL	VM	75.09	0.891	75.09 ± 0.638
	LXC	76.55	0.829	76.55 ± 0.593

In Figure 4.10, with changed RTT, LXC containers display nearly the same

performance as VMs, whereas in Figure 4.11, with changed bandwidth, clear differences could be observed. However, in the latter case, containers and VMs tend to be equal in throughput as bandwidth is reduced to 100Mbps. Also, in Figure 4.10, the throughput hovers between 4~25Mbps, while in Figure 4.11, the highest could reach nearly 500Mbps, and the lowest is around 20Mbps. This implies that prolonged RTT constrains network throughput more observably than reduced bandwidth does. In Table 4.11 to Table 4.13, the results of LXC containers are either the same as or slightly smaller than those of VMs. Both containers and VMs' standard deviation and confidence intervals are quite small. This is because as RTT is prolonged, the sender has enough time to transmit the packets, which causes very little variance in the throughput values in each test. However in Table 4.14 to Table 4.16, the RTT is kept as default (around 0.9ms), so throughput varies more considerably. Also, in Table 4.14 to Table 4.16,C code, PostgreSQL and null containers are outperformed by their VM counterparts respectively in terms of standard deviation and confidence intervals.

4.2.5 Boot Time

The boot time of a container is part of the service initiation duration. In this test, a bare Ubuntu LXC container and a basic Ubuntu VM, i.e., no extra software packages installed are selected to be started up for ten times. The following Figure 4.12 presents their average results.

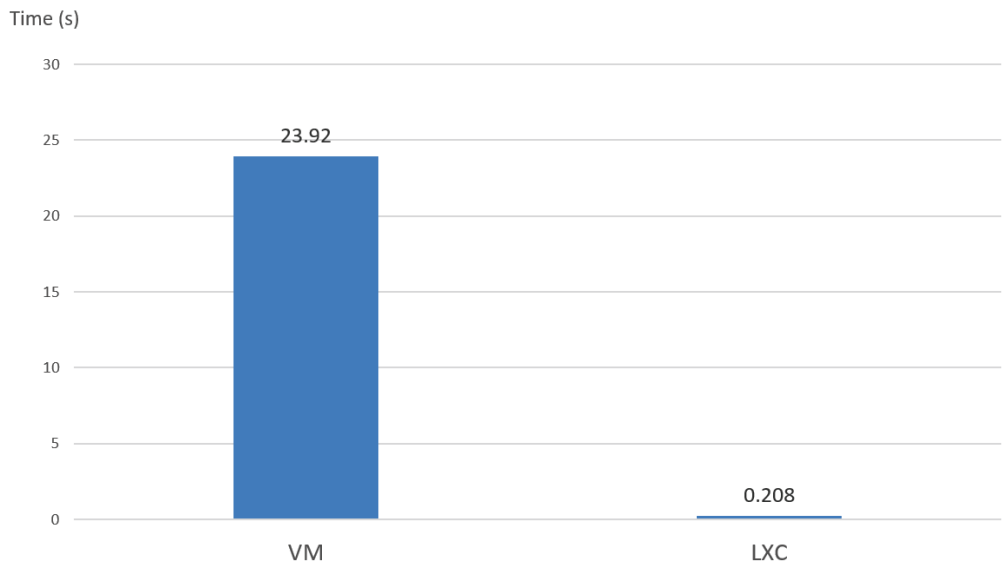

Figure 4.12 Boot Time

The above diagram indicates that the boot time of a LXC container is two orders of magnitude faster than that of a VM.

4.3 Performance Measurements of LXC Container Migration

This section measures the migration time of four containers from VM1 to VM2 using the *date* command in Linux. Similar to the parameters chosen in the network throughput test, the RTT is set as 20ms, 60ms and 120ms, the bandwidth is set as 600Mbps, 300Mbps and 100Mbps and the results are the average of ten consecutive times.

Before the display of results obtained by changing network parameters, migration time under default settings, along with the disk, memory and block I/O usage of four containers is measured and listed in the following Table 4.17. The purpose of doing so is to grasp an understanding of the four container's sizes in three aspects and to

identify the correlation between them and migration time. By using the *ping* command from VM1 to VM2, the default RTT is found to be 0.9ms. Maximum available bandwidth is obtained by monitoring the outcomes of *nload* command and it fluctuates around 600 to 650Mbps.

Table 4.17 Migration Time under Default Settings and Disk, RAM and Block I/O Usage of Containers

Container	Migration Time (s)	Disk Usage (MB)	RAM Usage (MB)	Block I/O Usage (MB)
Tomcat	61.8	577	129.54	50.74
PostgreSQL	46.2	476	59.51	37.64
C Code	36.6	356	35.82	20.68
Null	31.4	354	27.86	20.62

The above results show that in all the three aspects of usage, the Tomcat container consumes the most and the null container consumes the least. Naturally, it takes the longest time to migrate the Tomcat container (61.8 seconds) and the shortest (31.4 seconds) to migrate null. Moreover, migration time mainly depends directly on the size of disk usage since *rsync* command copies files from disk to disk.

The following two tables (Table 4.18 and Table 4.19) record the migration time with changed RTT and bandwidth. In order to show which parameter poses a greater impact on the migration time, an increase percentage is added to both tables. The percentage gain (last column) is calculated by using the differences between the figures on the fourth (RTT=120ms/BW=100Mbps) and the second (RTT=20ms/BW=600Mbps) columns and dividing the figures on the second column.

For example, the percentage gain of Tomcat container (450%) is calculated by using 523s (RTT=120ms) to subtract 95s (RTT=20ms) and then dividing 95s (RTT=20ms).

Table 4.18 Migration Time with Different RTTs

Container	RTT=20ms	RTT=60ms	RTT=120ms	% Gain
Tomcat	95s	267s	523s	450 %
PostgreSQL	66s	159s	307s	370 %
C Code	59s	137s	267s	350 %
Null	53s	136s	266s	400 %

Table 4.19 Migration Time with Different Bandwidth

Container	BW=600Mbps	BW=300Mbps	BW=100Mbps	% Gain
Tomcat	71s	95s	164s	130 %
PostgreSQL	50s	73s	101s	100 %
C Code	47s	61s	79s	70 %
Null	41s	56s	74s	80 %

From both tables, we can clearly identify the pattern where migration time increases as RTT is increasing, or as bandwidth is decreasing. The percentage gain column demonstrates by how much percent migration time has increased from RTT=20ms to RTT=120ms (see Table 4.18) and from BW=600Mbps to BW=100Mbps (see Table 4.19). Clearly, the change in RTT affects migration time more significantly than that of bandwidth. What this key finding implies is that where tradeoffs between RTT and bandwidth have to be made, it will be more helpful to decrease RTT than to increase bandwidth in order to ensure a satisfactory level of service quality between neighboring Cloudlets.

4.4 Performance Measurements of LXC Container Migration under MPTCP

As mentioned in section 4.1, experiments in this section are conducted on the second set. The two wireless adaptors (wlan0 and ra0) and two Wi-Fi connections (wifi1 and wifi2) have constituted a total of two MPTCP combinations and four TCP pairs (shown in Figure 4.13). Their qualities, including bit rate (bandwidth), link quality (how good the received signal is) and signal level (how strong the received signal is), are acquired from *iwconfig* command and listed in Table 4.20.

MPTCP Combination 1:

Subflow 1: wlan0: wifi1 ------ eth0 (VM4) ------ eth0 (VM3)

Subflow 2: ra0: wifi2 ------ eth1 (VM4) ------ eth0 (VM3)

MPTCP Combination 2:

Subflow 1: wlan0: wifi2 ------ eth0 (VM4) ------ eth0 (VM3)

Subflow 2: ra0: wifi1 ------ eth1 (VM4) ------ eth0 (VM3)

TCP Pair 1:

wlan0: wifi1 ------ eth0 (VM4) ------ eth0 (VM3)

TCP Pair 2:

wlan0: wifi2 ------ eth0 (VM4) ------ eth0 (VM3)

TCP Pair 3:

ra0: wifi1 ------ eth0 (VM4) ------ eth0 (VM3)

TCP Pair 4:

ra0: wifi2 ------ eth0 (VM4) ------ eth0 (VM3)

Figure 4.13 MPTCP Combinations and TCP Pairs

Table 4.20 Link Qualities

Rank of Link Strength	Link	Bit Rate (Mb/s)	Link Quality	Signal Level (-dBm)

1	ra0—wifi1	130	94/100	-52
2	ra0—wifi2	130	84/100	-59
3	wlan0—wifi1	52	55/70	-55
4	wlan0—wifi2	39	52/70	-58

Since wireless connections are not stable, the above figures are the averages of ten observations obtained between 10:00 and 10:10 am on a Monday morning in a residential apartment building. In general, the table presents these four connections in a descending order, with ra0—wifi1 being the strongest and wlan0—wifi2 being the weakest.

Experimental outcomes regarding subflow usage, migration time and CPU usage are presented in subsequent sub-sections.

4.4.1 Subflow Usage

There are two subflows for the connection of each MPTCP combination (as shown in Figure 4.13). In order to find out how traffic is distributed between them, the *ifstat* command in Linux is used to monitor the number of bytes transmitted over each interface, namely eth0 and eth1 on VM4. During the connection, eth0 is enabled all the time for two MPTCP combinations and eth1 has two states: disabled and then resumed.

4.4.1.1 Consistent During Connection

In this part, the two interfaces are consistent during the entire MPTCP connection and the ratio of subflow usage is shown in Table 4.21.

Table 4.21 Subflow Usage

Subflow	**Subflow Usage Ratio (subflow1:subflow2)**
MPTCP Combination 1: subflow1: wlan0—wifi1 subflow2: ra0—wifi2	1: 4.6
MPTCP Combination 2: subflow1: wlan0—wifi2 subflow2: ra0—wifi1	1 : 8.3

The above table shows for both MPTCP combinations, subflow2 (ra0-wifi2/wifi1) display a higher usage than subflow1 (wlan0-wifi1/wifi2). Based on the link strength from Table 4.20, it can be deduced that the stronger the connection of the subflow, the more traffic it will undertake. Evidently, this is also related to the subflow policy on two hosts. In all the MPTCP-related experiments conducted in this research, both subflows can be utilized. Since VM3 and VM4 used for the test are on the same physical host (see Table 4.2), traffic between them go through the same path, which means the RTT values of the four links listed in Table 4.20 are very close with negligible differences. In the order of Table 4.20, their RTT values are 1.037ms, 1.031ms, 1.042ms and 1.038ms, respectively. In this case, traffic will be distributed according to a combined evaluation of the subflow's strength.

4.4.1.2 One Eth Interface Disabled and Resumed During Connection

As pointed out previously, in both MPTCP combinations, subflow2 transfers the bulk of data. Since subflow2 is initiated from ra0 which is bridged to eth1 on VM4, how will subflow1 be affected if eth1 is abruptly disabled and then resumed during

connection? Figure 4.14 and Figure 4.15 answer this question by recording the interface usage on VM4.

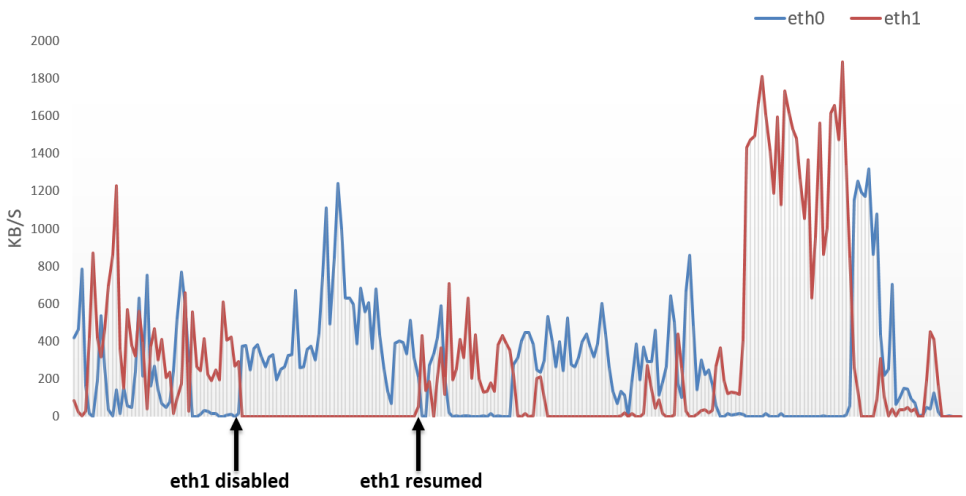

Figure 4.14 Subflow Usage of MPTCP Combination 1

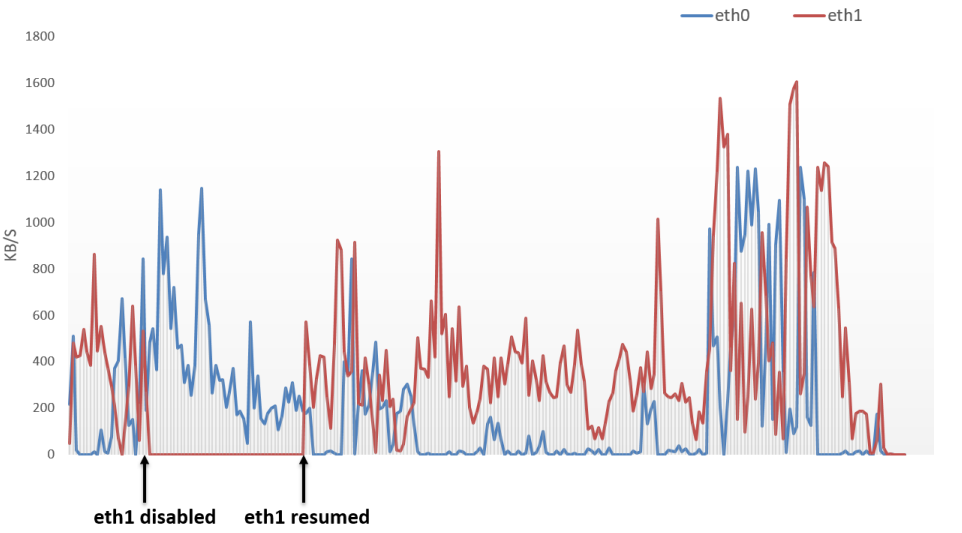

Figure 4.15 Subflow Usage of MPTCP Combination 2

The X axes of Figure 4.14 and Figure 4.15 represent time in 4 seconds intervals until around 250 seconds. No time gaps are observed after eth1 is disabled. From the two graphs we can clearly see how eth0 seamlessly takes over the traffic when eth1

gets disconnected. This proves the fact that MPTCP can enhance connection resilience.

4.4.2 Migration Time

Statistics from the previous sub-section imply that MPTCP transfers the most data on the more "capable" subflow of the two. In this light, migration between two hosts can become faster if the added subflow is stronger than the previous one. Same as the migration tests performed in section 4.3, the *date* command is also used here and the results are the averages of ten tests. What is different is that only one bare LXC container with Ubuntu operating system is chosen and the network parameters of RTT (0.9ms) and bandwidth (as indicated in Table 4.20) are kept as default. Table 4.22 reveals the results.

Table 4.22 Migration Time under MPTCP vs. TCP

MPTCP Combinations and TCP Pairs	Migration Time (s)
MPTCP Combination 1: subflow1: wlan0—wifi1 subflow2: ra0—wifi2	242
MPTCP Combination 2: subflow1: wlan0—wifi2 subflow2: ra0—wifi1	241
TCP Pair 1: wlan0-wifi1	285
TCP Pair 2: wlan0-wifi2	294
TCP Pair 3: ra0-wifi1	244
TCP Pair 4: ra0-wifi2	257

During the course of the migration, the two subflows are consistent during the established MPTCP connection. It can be observed from Table 4.22 that migration time is nearly the same under two MPTCP combinations. In the same table, TCP Pair 3 only takes 3 seconds longer to finish the migration than MPTCP Combination 2 does. TCP Pair 3 has only one link ra0-wifi1, whereas MPTCP Combination 2 also has an additional link of wlan0-wifi2, which has the weakest strength according to Table 4.20. This indicates that even though two subflows are transmitting data in MPTCP combination 2, it does not necessarily offer a significant advantage if the extra subflow is much weaker than the TCP connection.

On the other hand, if the extra subflow is decidedly stronger, as is reflected in MPTCP Combination 2 and TCP Pair 2 (as shown in Table 4.22), the migration time will enjoy a great reduction from 294 to 241 seconds.

It should be noted that migration time will be greatly reduced once P. Haul and LXD become more mature. The time shown in Table 4.22 should be viewed comparatively under the two scenarios (TCP and MPTCP). Different sets of host configurations will bring different results. Nevertheless, the core idea is that for a particular TCP connection, if the newly-joined subflow is stronger in strength, the majority of traffic will prefer the stronger one, making the migration process shorter. This proves the fact that MPTCP can reduce migration time.

4.4.3 CPU Usage

Even if the addition of a new subflow does not accelerate the migration process, it

still enhances it by providing an alternative path should the “capable” subflow fail, as demonstrated in sub-section 4.4.1.2. Nevertheless, the use of MPTCP puts a heavier burden on the system with regards to CPU utilization because of the extra CPU resources consumed by the additional subflow.

Monitored by the *sar* command in Linux, the CPU usage of user and system in MPTCP combinations and TCP pairs are listed in Table 4.23.

Table 4.23 CPU Usage under MPTCP vs. TCP

MPTCP Combinations and TCP Pairs	**CPU Usage (%user)**	**CPU Usage (%system)**
MPTCP Combination 1: subflow1: wlan0—wifi1 subflow2: ra0—wifi2	6.7	12.5
MPTCP Combination 2: subflow1: wlan0—wifi2 subflow2: ra0—wifi1	6.75	12.63
TCP Pair 1: wlan0-wifi1	5.53	7.25
TCP Pair 2: wlan0-wifi2	5.35	7.01
TCP Pair 3: ra0-wifi1	5.81	7.62
TCP Pair 4: ra0-wifi2	5.80	7.64

We can see from Table 4.23 that both MPTCP combinations display a higher CPU usage than that of TCP pairs. The differences in CPU utilization by user are not as considerable as they are by system. For MPTCP combinations, the figures in the third column almost double those in the second column.

Therefore, for hosts with demanding applications, undesired side effects may be incurred if MPTCP is applied during connections. The above table only shows the case with an almost empty host and two subflows. A more substantial increase in CPU utilization under MPTCP is expected in real-world situations. Therefore, as appealing as the advantages of MPTCP may seem, it requires a comprehensive evaluation on the hosts' capabilities in deciding whether choosing MPTCP is really beneficial for Cloudlet servers.

Chapter 5 Conclusions and Future Work

In this chapter, section 5.1 provides a summary on the approach adopted by this thesis and the major findings. Section 5.2 points out some areas for future work.

5.1 Conclusions

In an attempt to proposing a better solution to Cloudlet servers, this thesis conducts a research on LXC containers and their collaboration with MPTCP protocol. The major concern is to make the servers lighter and more capable and to make the migration process faster and more fault-tolerant.

VMs are widely acknowledged candidates as Cloudlet servers for many years. Early in 2009, authors in [13] have come up with a fairly advanced approach to prepare VMs for the service delivery process. However, ever since the advantages of the container technology have begun to unfold, researchers started to take a new perspective on Cloudlet servers and proposed the idea of using containers to replace VMs [24]. This thesis follows this trend by adopting LXC containers. But instead of simply taking VMs off the table, this thesis builds LXC containers on top of VMs, as opposed to on top of physical hosts presented in other papers. The idea is to benefit from VMs' economic feasibility and secure isolation while exploring the new possibilities brought by containers.

In order to evaluate how they perform on VMs, four LXC containers with different types of applications were chosen and tested from five aspects: CPU performance, memory performance, disk I/O performance, network throughput and

boot time. Experimental results showed that on average, containers either equaled or outperformed their VM counterparts in all of the five aspects with varying degrees of difference. However, in terms of standard deviation and confidence intervals, containers' advantages become less obvious. On some occasions, VMs show a more stable performance, especially in the disk I/O sequential input test. These findings demonstrate that in general, the intrinsic lightness of containers can reduce overheads and improve performance considerably, but in some cases, the stability of containers' performance is not as good as that of VMs .

As service delivery platforms, the ability to operate on one host is not enough. Therefore, this thesis also evaluates the current migration mechanism of LXC containers. The CRIU utility can help a container to preserve its states during its suspension and the *rsync* command implements the migration process by copying container-related files to the remote host. Experimental results under default settings showed that migration time is closely related with the disk size of the container. With changed network parameters of RTT and bandwidth, the increase in RTT prolongs the migration process more significantly than the decrease in bandwidth.

Even though a container is lightweight, a bare Ubuntu container still takes up around 350MB of disk space. To transmit this amount of data over the WAN, a stable, and preferably faster connection is vital to the migration process. In an effort to address this challenge, this thesis proposed using MPTCP protocol to replace the regular TCP protocol. MPTCP protocol can establish multiple paths between the source and destination during a MPTCP connection. If one subflow fails, the network

traffic can be seamlessly switched to other subflows without interruption. If all the subflows function well, depending on the configured subflow policy, the traffic could be sent over the strongest subflow, which means migration time can be greatly reduced. Experimental results showed that when the RTT of subflows are the same, a combined evaluation of bit rate, link quality and signal strength have been considered when choosing paths. Also, migration time is diminished if the additional subflow is noticeably stronger than the previous one. However, the benefits of MPTCP come at a price of high CPU utilization, especially the one consumed by system. Consequently, the choice of MPTCP should be made with discretion.

5.2 Future Work

The LXC container technology and its migration technique are still under active development. Due to the limitations of current CRIU, the migration mechanism presented in this thesis is not live migration. Based on this consideration, the future work of this thesis could focus on implementing the LXD container with P. Haul project mentioned in section 3.4.2. Furthermore, future research could also explore the possibilities of choosing other container tools such as Docker and OpenVZ Virtuozzo container as Cloudlet platforms.

In addition, the offloading process between the end user and the Cloudlet servers can be another area of interest. As the premise of service delivery, how offloading and downloading are achieved will involve a lot of research on the programming interface of applications and the initiation mechanism of Cloudlet servers.

References

[1] X. Wang and Y. Wang, “Energy-efficient multi-task scheduling based on MapReduce for cloud computing,” *Proc. of the 7th International Conference on Computational Intelligence and Security (ICCIS)*, Hainan, Dec2011, pp. 57-62.

[2] F. Bonomi, R. Milito, J. Zhu and S. Addepalli, “Fog computing and its role in the Internet of Things,” *Proc. Of the 1st ACM MCC workshop on Mobile Cloud Computing*, 2012, pp. 13-16.

[3] A rare peak into the massive scale of AWS, http://www.enterprisetech.com/2014/11/14/rare-peek-massive-scale-aws/. Last accessed on Jul 25, 2016.

[4] CRIU, https://criu.org/Main_Page. Last accessed on Jul 20, 2016.

[5] Flocker, https://clusterhq.com/flocker/introduction/. Last accessed on Jul 20, 2016.

[6] MPTCP, http://www.multipath-tcp.org/. Last accessed on Jul 20, 2016.

[7] LXC, https://en.wikipedia.org/wiki/LXC. Last accessed on Jul 20, 2016.

[8] Live migration of Linux containers, https://tycho.ws/blog/2014/09/container-migration.html. Last accessed on Jul 20, 2016.

[9] S. Yi, C. Li and Q. Li, “A survey of fog computing: concepts, applications and issues,” *Proc. of 2015 Workshop on Mobile Big Data*, New York, 2015, pp. 37-42.

[10] Cisco DevNet: IOx, https://developer.cisco.com/site/iox/technical-overview/. Last accessed on Jul 20, 2016.

[11] T. H. Luan, L. Gao, Z. Li, Y. Xiang, G. We and L. Sun, “Fog Computing: focusing on the mobile users at the edge,” Technical Report, Department of Computer Science, Cornell University, submitted for review, arXiv:1502.01815v3. Last accessed on July 28, 2016.

[12] S. Kommineni, A. De, S. Alladi and S. Chilukuri, “The cloudlet with a silver lining,” *Proc. of the 6th International Conference on Communication Systems and Networks (COMSNETS)*, Bangalore, Jan 2014, pp. 6-10.

[13] M. Satyanarayanan, P. Bahl, R. Caceres and N. Davies, “The case for VM-based cloudlets in mobile computing,” *Proc. of IEEE Pervasive Computing*, 2009, pp. 14-23.

[14] K. Ha, P. Pillai, W. Richter, Y. Abe and M. Satyanarayanan, “Just-in-time provisioning for cyber foraging,” *Proc. of the 11th Annual International Conference on Mobile Systems, Applications and Services*, New York, 2013, pp. 153-166.

[15] Linux containers: Why they’re in your future and what has to happen first, http://www.cisco.com/c/dam/en/us/solutions/collateral/data-center-virtualization/openstack-at-cisco/linux-containers-white-paper-cisco-red-hat.pdf. Last accessed on Jul 20, 2016.

[16] R. Morabito, J. Kjallman and M. Komu, “Hypervisors vs lightweight virtualization: a performance comparison,” *Proc. of 2015 IEEE International Conference on Cloud Engineering (IC2E)*, Tempe, Mar 2015, pp. 386-393.

[17] Docker, https://www.docker.com/. Last accessed on Jul 20, 2016.

[18] W. Felter, A. Ferreira, R. Rajamony and J. Rubio, “An updated performance comparison of VM and Linux containers,” *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Philadelphia, Mar 2015, pp. 29-31.

[19] OpenVZ, https://openvz.org/CRIU. Last accessed on Jul 20, 2016.

[20] TCP connection, https://criu.org/TCP_connection. Last accessed on Jul 20, 2016.

[21] P. Haul, https://criu.org/P.Haul. Last accessed on Jul 20, 2016.

[22] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure and M. Handley, “How hard can it be? Designing and implementing a deployable multipath TCP,” *Proc. of the 9th Symposium on Networked Systems Design and Implementation (NSDI)*, Berkeley, 2012, pp. 29-29.

[23] F. Teka, C.-H. Lung and S. Ajila, "Seamless live virtual machine migration with Cloudlets and multipath TCP," *Proc. of 39th Annual Computer Software and Applications Conference (COMPSAC)*, Taichung, Jul 2015, pp. 607-616.

[24] B. Ismail, E. Goortani, M. Karim, W. Tat, S. Setapa, J. Luke and O. Hoe, "Evaluation of Docker as edge computing platform", *Proc. of IEEE Conference on Open Systems (ICOS)*, Melaka, Aug 2015, pp. 130-135.

[25] L. Bittencourt, M. Lopes, I. Petri and O. Rana, "Towards VM migration in fog computing," *Proc. of the 10th International Conference on P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC)*, Krakow, Nov 2015, pp. 1-8.

[26] Linux Containers, https://linuxcontainers.org/lxc/security/. Last accessed on Jul 20, 2016.

[27] LXC.conf, http://manpages.ubuntu.com/manpages/precise/man5/lxc.conf.5.html. Last accessed on Jul 20, 2016.

[28] RSYNC, http://manpages.ubuntu.com/manpages/trusty/man1/rsync.1.html. Last accessed on Jul 20, 2016.

[29] TCP extensions for multipath operation with multiple addresses, https://tools.ietf.org/html/rfc6824. Last accessed on Jul 20, 2016.

[30] A. Shribman and B. Hudzia, "Pre-copy and post-copy VM live migration for memory intensive applications," *Proc. of European Conference on Parallel Processing*, 2012, pp. 539-547.

[31] LXD, https://linuxcontainers.org/lxd/introduction/. Last accessed on Jul 20, 2016.

[32] C. Paasch, S. Barre, et al., Multipath TCP in the Linux Kernel, http://www.multipath-tcp.org. Last accessed on Jul 20, 2016.

[33] L. Leemis and S. Park, "Output Analysis," in *Discrete-event simulation: a first course*, 2004, pp. 350-354.

[34] Sysbench, https://github.com/akopytov/sysbench. Last accessed on Jul 20, 2016.

[35] Mbw, https://github.com/raas/mbw. Last accessed on Jul 20, 2016.

[36] Bonnie++, http://www.coker.com.au/bonnie++/. Last accessed on Jul 20, 2016.

[37] Netperf, http://www.netperf.org/netperf/. Last accessed on Jul 20, 2016.