A CONTAINER-BASED CODE OFFLOADING FRAMEWORK FOR MOBILE EDGE COMPUTING APPLICATIONS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF INFORMATICS OF
THE MIDDLE EAST TECHNICAL UNIVERSITY
BY

HAKAN MESUT DUR

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
IN
THE DEPARTMENT OF INFORMATION SYSTEMS

SEPTEMBER 2021

A CONTAINER-BASED CODE OFFLOADING FRAMEWORK FOR MOBILE EDGE COMPUTING APPLICATIONS

Submitted by Hakan Mesut Dur in partial fulfillment of the requirements for the degree of **Master of Science in Information Systems Department, Middle East Technical University** by,

Date: *10.09.2021*

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name : Hakan Mesut Dur

Signature : _____

ABSTRACT

A CONTAINER-BASED CODE OFFLOADING FRAMEWORK FOR MOBILE EDGE COMPUTING APPLICATIONS

Dur, Hakan Mesut

MSc., Department of Information Systems

Supervisor: Assoc. Prof. Dr. Altan Koçyiğit

September 2021, 40 pages

Recently, the use of mobile devices has increased tremendously. This leads to the growing complexity and diversification of mobile applications. However, mobile devices generally do not keep up with this growth and they usually suffer from low performance for complex applications. In order to improve the performance of such applications, devices can make use of nearby computation platforms such as powerful edge servers. This thesis proposes a container-based code offloading framework that provides distribution transparency and automatic migration for mobile applications. The framework supports the Python programming language and makes use of proxy objects created by the Pyro library for code offloading. Docker containers are used to run offloaded code and keep the application state. These containers are automatically migrated to the nearest edge servers in case of mobile user relocation. A sample application is developed to validate the framework.

Keywords: Code Offloading, Mobile Application, Migration, Docker

ÖZ

MOBİL UÇ HESAPLAMA UYGULAMALARI İÇİN KONTEYNER TABANLI BİR KOD TAŞIMA ÇERÇEVESİ

Dur, Hakan Mesut

Yüksek Lisans, Bilişim Sistemleri Bölümü

Tez Yöneticisi: Doç. Dr. Altan Koçyiğit

Eylül 2021, 40 sayfa

Son zamanlarda mobil cihazların kullanımı olağanüstü düzeyde artmıştır. Bu, mobil uygulamaların karmaşıklığının ve çeşitliliğinin artmasına yol açmaktadır. Ancak mobil cihazlar genellikle bu büyümeye ayak uyduramazlar ve karmaşık uygulamalar için düşük performans sıkıntısı yaşarlar. Bu tür uygulamaların performansını artırmak için cihazlar, güçlü uç sunucular gibi yakındaki hesaplama platformlarından yararlanabilirler. Bu tez, mobil uygulamalar için dağıtım şeffaflığı ve otomatik göç sağlayan konteyner tabanlı bir kod taşıma çerçevesi önermektedir. Çerçeve, Python programlama dilini destekler ve kod taşıma için Pyro kütüphanesi tarafından oluşturulan vekil nesnelerini kullanır. Docker konteynerler taşınan kodu çalıştırmak ve uygulamanın durumunu tutmak için kullanılır. Bu konteynerler, mobil kullanıcının yer değiştirmesi durumunda otomatik olarak en yakın uç sunuculara göç ettirilirler. Çerçeveyi doğrulamak için örnek bir uygulama geliştirilmiştir.

Anahtar Sözcükler: Kod Taşıma, Mobil Uygulama, Geçiş, Docker

To My Family

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Assoc. Prof. Dr. Altan Koçyiğit, for his valuable support and patience throughout my studies.

I would like to thank my friends. They have always support me with their valuable feedbacks. That gave me mental relief in my troubled times.

Finally, I would like to thank my mother, my father, and my sister for becoming irreplaceable parts of my life. They have always encouraged me and showed me understanding. In addition, I am especially grateful to my sister for helping me until the last moment in developing my thesis presentation.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

CLR	Common Language Runtime
CPU	Central Processing Unit
CRIU	Checkpoint/Restore in Userspace
iOS	iPhone Operating System
MAUI	Mobile Assistance Using Infrastructure
MCC	Mobile Cloud Computing
MEC	Mobile Edge Computing
PC	Personal Computer
URI	Uniform Resource Identifier
VM	Virtual Machine

CHAPTER 1

INTRODUCTION

1.1. Motivation

Most of the current mobile applications have evolved incredibly and had demanding requirements for mobile devices. Applications such as image processing, mobile gaming, connected vehicles, and augmented reality are processing intensive. These applications have low latency demands and high energy costs. With the increasing use of mobile devices and such applications, the need to maintain large amounts of data sets and timely processing of their data is also increasing.

It is difficult to satisfy these requirements on the end-user side due to battery limitations, low computing power, and low memory of mobile devices. In this case, it makes sense to use cloud systems with rich resources. Since cloud infrastructures have higher storage space and computing power than mobile devices, they work faster with bigger data sets. However, traditional cloud systems may not be useful for time-critical services due to their distance to the user. This physical distance between the places where data is generated and processed causes an intolerable delay.

On the other hand, it is possible to provide services that satisfy low latency requirements by bringing the source-rich servers close to the end-user. It is a promising idea for developers to use offloading for this. End-users can benefit from low latency, high bandwidth, and high computation power and enjoy a better user experience by offloading heavy computing tasks to a nearby edge server. This approach is often encountered in the literature as Mobile Edge Computing (MEC) [1].

Latency is low when a mobile device is stationary and connected to the same edge server. However, when the mobile user moves, the distance between the mobile device and the edge server changes. If the distance increases, the mobile user loses connection with the current edge server and switch to a closer edge server. When this happens, the service provided on the previous edge server needs to be migrated to an edge server closer to the mobile user. Thus, one of the critical points in MEC is maintaining service continuity at different locations. While it sounds suitable, moving edge server services with the user is difficult and can cause performance degradation.

1.2. Objectives and Scope

This thesis presents a framework that offloads computation intensive tasks to edge servers and performs service migration between edge servers. Our goals are 1) supporting computation offloading to edge servers from mobile users 2) creating a structure suitable for migration to adapt to the movement of the mobile user for the sake of continuity 3) developing a framework that allows offloading and service migration.

We recommend a structure based on container-based virtualization, remote procedure calls, and proxy usage to achieve our goals. Since Python is a general-purpose, high-level programming language and is highly preferred in scientific computing, our framework supports applications written in Python. Also, we use the Pyro [2] library in Python for remote method calls via proxies and the Docker technology [3] for containerization. Pyro is beneficial for remote method calls and is perfect to create a seamless structure by enabling objects on different machines to communicate with each other through the network connection. On the other hand, containers are a lightweight option among virtualization techniques. They provide ease of use, fast deployment and service isolation, and Docker is one of the most popular container platforms. In our approach, clients and servers communicate among themselves through the Pyro library, and the services on servers run in a container. Combining the container structure and the use of the Pyro library makes it easy to write applications with offloadable parts and creates a generic framework.

1.3. Thesis Structure

This thesis is structured as follows:

Chapter 2 gives a summary of the background and related technologies. Chapter 3 summarizes related work. Chapter 4 explains the model used in the framework presented in this thesis. Chapter 5 shows how to implement the framework in a prototype and its usage in sample applications. Finally, Chapter 6 concludes the work by discussing contributions and future works.

CHAPTER 2

BACKGROUND

This section briefly describes the main structures, technologies, and approaches used to implement our study.

2.1. Offloading

With the advancements in computing and communication technologies, the use of mobile devices has considerably increased. Consequently, embedded and mobile applications running on these devices have been gaining popularity day by day. However, such new generation applications run on systems with limited resources. These limitations negatively affect the performance of mobile devices such as smartphones in terms of both energy consumption and processing time [4]. Smartphones are powered by batteries, and their processing and storage capacities are low. Their sensors are capable of producing a lot of data in a short amount of time, but the storage and timely processing of such data are not usually easy on such platforms. Moreover, mobile devices use wireless network connections, which provide lower data rates compared to wired network connections. Hence, they also have limited communication resources. On the other hand, the complexities of applications such as artificial intelligence, image processing, and video processing has been increasing. Therefore, modern applications suffer from performance and quality of experience issues. Computation offloading is suggested as a solution to limited resource problems on mobile devices [5], [6]. Simply put, this means having the computational work done on a resource-rich and robust platform. Offloading computational tasks to a platform with powerful resources allows overcoming performance and quality of experience issues caused by limitations such as low computation power and insufficient storage space.

According to the granularity of the offloaded code units, offloading or application partitioning can be divided into two broad categories: fine-grained and coarse-grained [7]. In fine-grained application partitioning, classes, objects, or functions may be executed in a remote platform, and results are returned to mobile systems [8]. In coarse-grained offloading [9], the application is partitioned usually at the application or virtual machine (VM) level. There are some critical differences between fine-grained and coarse-grained partitioning. The communication cost is generally low when a coarse-grained partition is made, but it may take a long time to offload an entire application [7]. On the other hand, the delay may be lower in fine-grained offloading since only the parts that require heavy work and tire the mobile device will be offloaded.

2.2. Mobile Cloud Computing

When the platform for offloading is cloud, it is called Mobile Cloud Computing (MCC). The purpose of MCC is to provide a better user experience by giving the rich resources of the cloud to the mobile user [10]. In client-server structures, the client benefits from a powerful server. Using the cloud instead of this powerful server does not break the basic client-server layout. In MCC, a virtualized system such as a VM is generally used on the cloud side. Virtualization creates multiple independent virtual computers with an abstract layer using software on physical hardware [11].

VMs bring several advantages. First, since they are an isolated environment, they are safer against any virus or malfunction because a VM problem does not cause the entire cloud system to crash. Second, sharing physical resources among VMs instead of using multiple hardware reduces cost and simplifies maintenance. Third, VM transfer, which can be performed without any physical structure, accelerates the work.

MCC structure generally has two architectures [12]. In the first one, the cloud hardware remains stationary and serves the mobile device, while in the second architecture, other mobile devices form a group and act as a cloud serving a mobile device. Figure 1 shows the first architecture. In this way, servers, databases, and VMs in a stationary cloud center interact with Personal Computers (PC) and mobile devices such as smartphones and laptops. However, by going further and separating the remote resources by their characteristics, it can be divided into four as distant immobile clouds, nearby immobile machines, nearby mobile machines, and hybrid systems [13]. Distant immobile clouds include public and private clouds, and there are many servers. They are highly available and resource-rich. Amazon EC2 [14] and Microsoft Azure [15] are examples of distant immobile clouds. Nearby immobile machines can contain cloudlets [9] or private clouds of public places. These resources are closer to the mobile user. Nearby mobile machines include portable devices such as tablets, notebooks, smartphones. The aim here is to take advantage of their proximity to the mobile user. In the hybrid model, different options are utilized according to user needs by combining previous options such as distant stationary clouds and proximate immobile machines [16].

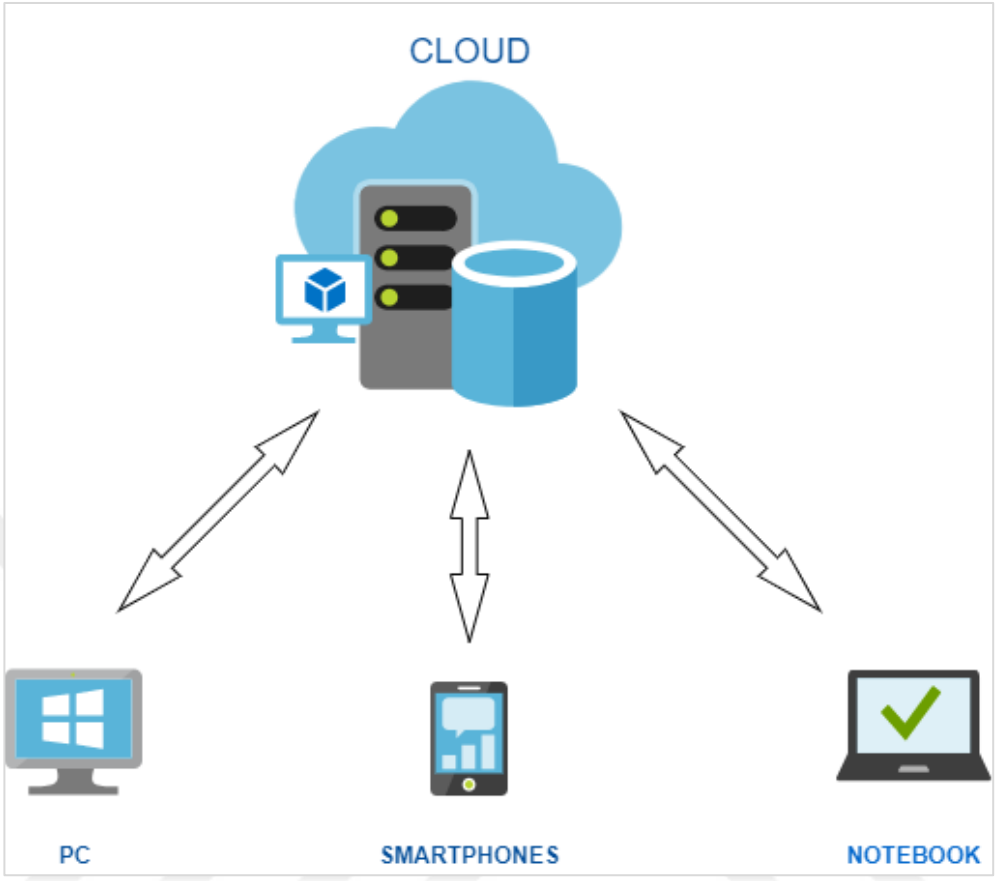

Figure 1: General architecture between cloud and mobile users

2.3. Mobile Edge Computing

Although MCC is handy, there are areas where it falls short, especially when it comes to low latency. The cloud's distance to the user and the sharing of its bandwidth by many users cause the connection between the cloud and the user to be slow. A problem that may occur in the cloud or a network crash also negatively affects service availability for a long time. At this point, MEC emerges [17]. Figure 2 shows general MEC architecture. Until the advent of MEC, devices placed in the network edge were used as only access points. MEC is a structure that can be an alternative to MCC, aiming to present cloud resources at the edge of the network, closer to the user. As rich sources will be closer to the user, the distance traveled by transmitted data will be shortened, and a lower latency and location awareness can be achieved. In addition, the traffic on the core network is reduced. MEC is used in many areas that require low latency, such as augmented reality, multiplayer gaming, video analytics, connected vehicles.

Although MEC provides service proximity to the user, this proximity will disappear when the user is highly mobile, which may cause performance degradation or disconnection. Thus, services need to be migrated between edge servers. Service migration provides continuity even though the user is moving.

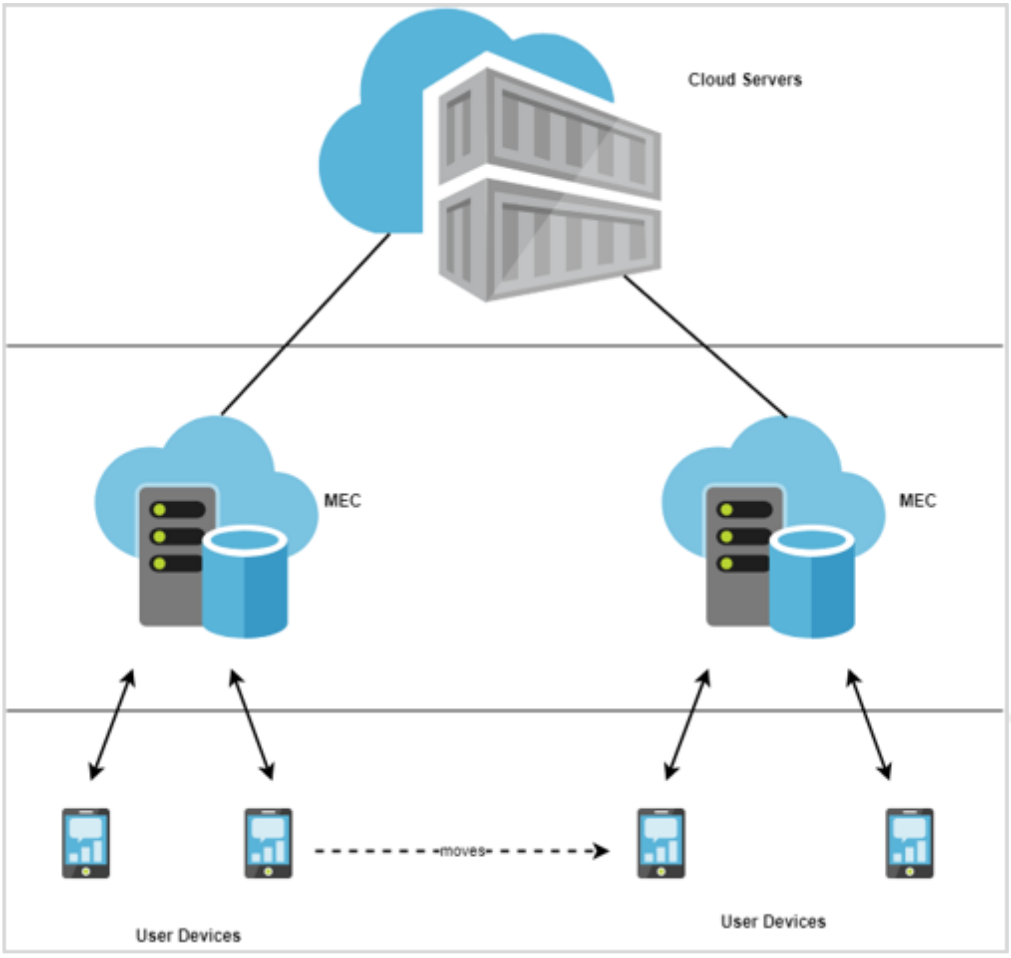

Figure 2: General MEC Architecture

2.4. Virtualization and Containerization

When a software program is offloaded from one computer to another, for example, from a physical machine to a cloud system, some structure is needed to run it properly and reliably. Usually, a certain operating system(s), other dependent software, and some libraries are necessary to support a software application. On the other hand, in order to efficiently use available resources, several applications should be run on the shared hardware platforms. However, if such applications require different versions of operating systems, relevant software, and libraries, they cannot be run on the same system. Moreover, due to security and reliability concerns, users (or the cloud provider itself) may not want to run applications in a single execution environment shared by many applications. Thus, keeping the necessary libraries, configuration files, and all

other dependencies for each application in the same execution environment while providing isolation across different applications is needed. Virtualization offers a handy solution to this problem: VM. A VM contains both the operating system and the application(s). A computer running five VMs includes a hypervisor and five different operating systems running on it. A hypervisor is a code that enables multiple guest operating systems to be created on a physical server.

On the other hand, container-based virtualization is seen as a lightweight option over the VM. Although VMs and containers have similar benefits, they are different in resource management and architecture, as seen in Figure 3. Containers virtualize the operating system, while VMs virtualize the hardware with the hypervisor. Instead of dealing with hardware, containerization bundles everything together and reduces it to a single application. The container contains the code, its dependencies, and the operating system it needs. Containers are more advantageous for most of the situations than VMs because they do not run separate operating systems and have less overhead [18], [19].

Docker [3] is a popular container engine that uses Linux kernel features to build containers on an operating system. Unlike a VM, Docker allows the system to be used in isolation by sharing the Linux kernel instead of creating a whole virtual operating system.

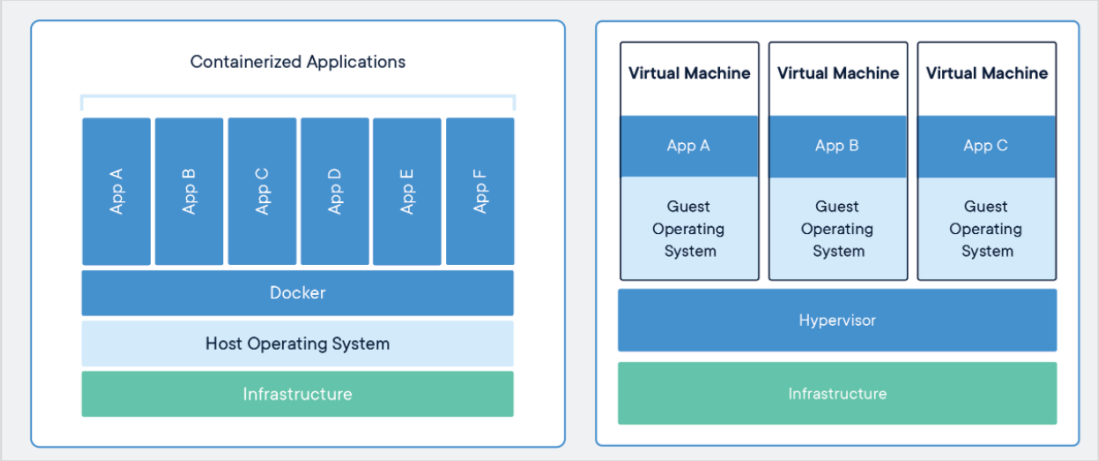

Figure 3: VM and container architectures [20]

Docker components and their relations are depicted in Figure 4. An image is the packaged version of the application to be run on the Docker container with the instructions. Docker container is the area where packaged images can be run. An image is created from a configuration file called Dockerfile. Docker uses a client-server architecture, and several containers can reside on the same system. Docker daemon manages images, containers, and volumes and can communicate with other daemons. A Docker client communicates with the Docker daemon via commands. The Docker registry is the repository where images are stored.

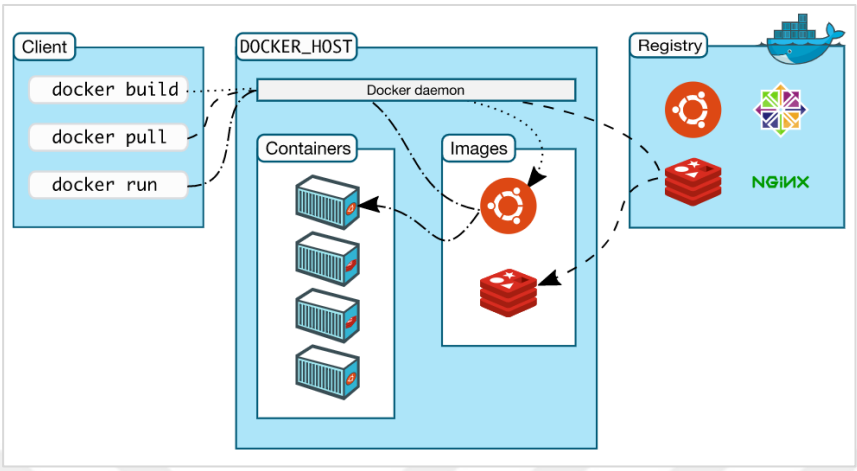

Figure 4: Docker components [21]

2.5. Stateless – Stateful Applications

State refers to any mutable conditions of a system. A stateless application does not keep any information about previous operations. For example, if a calculator always displays zero each time it is turned on and not the most recent process, it is a stateless application. A stateful application remembers something about its state every time it runs. Therefore, statefulness needs persistent storage.

Container-based applications tend to be stateless because, simply put, a container appears, does its job, and disappears. However, a volume is used to make a Docker container stateful [22]. Any database, library, or other configuration files can be kept in the volume, which does not affect the container's size because the volume content is separate from the container. The files in the volume can be read, modified, deleted and new ones added by the container.

2.6. Pyro Library

Pyro [2] is a Python library that facilitates the development of applications that can communicate with each other over the network. With Pyro, method calls through a network can be made transparently, and the Pyro takes care of finding the correct object. It can be used for different Python versions.

Pyro, which is useful for distributed systems, has some important concepts. A proxy is an object that replaces the real object. It is used in an application as if it is a real object. Figure 5 shows a sample class diagram for the proxy pattern. In general, two classes are implemented, such as Proxy and RealSubject, which conform to an interface. The method invoked from the proxy is delegated to reach the object of the real class.

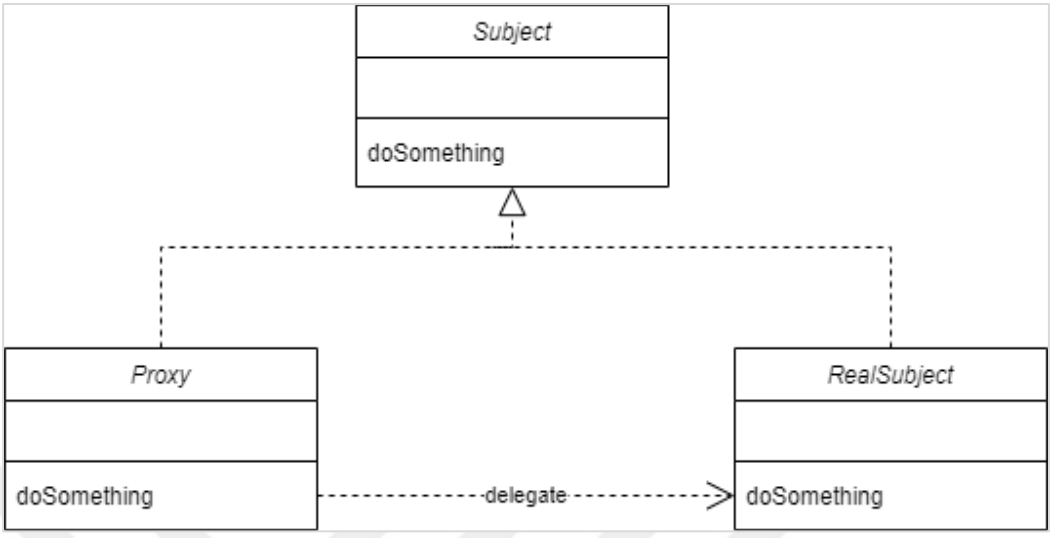

Figure 5: A class diagram of proxy pattern

On the other hand, Pyro library doesn't need different implementations. A proxy is created by Pyro. Methods of the real Pyro object are used in the background on the machine contains it, and the results are returned to the proxy. Thus, the code used the proxy does not know which object it is actually working with. The relation between objects is shown in Figure 6. The Pyro proxy is created with the help of a Uniform Resource Identifier (URI). The URI contains the object name, server name, and port number, and through it, the proxy can reach the correct Pyro object at the correct machine. A Pyro object is a remotely accessible object registered with the Pyro. It is no different from other objects, except that Pyro knows that methods of this object can be called remotely. Pyro Daemon listens for remote method calls and returns results from correct objects to proxies. Pyro objects must be registered with at least one daemon to be accessed remotely.

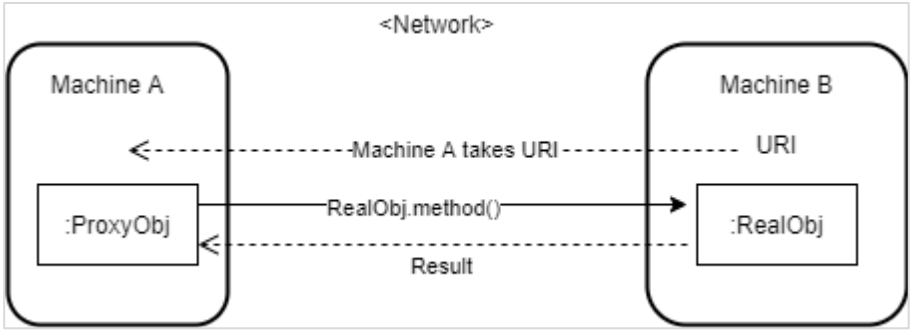

Figure 6: Relation between Pyro proxy and actual objects

CHAPTER 3

RELATED WORK

Studies related to the work presented in this thesis are given in this section. First, it is necessary to mention offloading. The first step of MCC and MEC is to offload from mobile devices to a resourceful cloud structure.

As a system, Mobile Assistance Using Infrastructure (MAUI) [23] focuses on energy conservation by using method-level offloading on the cloud. It is a structure built on the reflection feature of the Microsoft .Net Common Language Runtime (CLR). The MAUI system uses a profiler to build up a profile of application methods on the mobile device. According to energy-saving criteria, it solves a code offload problem in the solver with Central Processing Unit (CPU) and network cost data. This analysis is done for the methods that the programmer has predetermined and marked with annotations. If there is an offloading decision, data of the relevant method is sent to the server, and a response at the end of remote execution is received.

Like MAUI, a system proposed by Kosta et al. [24], ThinkAir, is based on marking methods suitable for offloading with @remote annotation on the client. A controller decides whether these methods will be offloaded based on their historical data about energy consumption, latency and environmental conditions such as network connectivity. For this, hardware, software, and network profilers are used, and an energy consumption problem is solved with their data. On the cloud server, the client handler maintains the connection between the client and the cloud, ensures that the offloaded code is executed, and the results are returned. Here, VMs are used, new ones are created according to demand, and the unused ones are destroyed to have dynamic control.

On the other hand, Wu et al. [25] focus on the cloud side of the business rather than the code offloading decision and code partitioning with their Rattrap platform. It is mentioned that using VMs in the cloud causes excessive resource consumption and the disadvantageous situation of long start-up times. That is why researchers are developing a lightweight Android Container. Sharing common resources and code cache mechanism were used to increase efficiency. As a result, it is aimed to reduce both the start-up time and the memory and disk usage.

Xu et al. [26] offer an offloading method for internet of connected vehicles that will reduce offloading delay and manage resources at the edge. Internet of connected vehicles is a concept that emerged after the internet of things, so in the beginning, remote clouds were generally used for offloading here too. Instead of remote clouds,

authors suggest roadside units and macro base stations for edge computing. In this way, the distance and offloading delay decreases. But this time, node selection for the offloading destination creates a problem. That's why authors recommend adaptive computation offloading method. With the multi-objective evolutions algorithm based on decomposition, possible destinations are determined and the final destination is decided with normalization techniques.

Kaya et al. [27] offer a framework that includes a call graph model to determine the parts to be offloaded in a software. It is possible to offload different parts of an application to remote servers. But while doing this, some combinations of offloading may negatively affect application performance. A call graph is created in runtime by monitoring the metrics of the application components, and profitable offloading decisions are made according to best partitioning in this graph. A factory class decides to create a local object, or a proxy based on the offloading decision. Thus, in this framework, distribution transparency is provided by inversion of control.

Lin et al. [28] suggest a framework that disables both MCC and MEC. They recommend code offloading to nearby mobile devices as there is no guarantee that offloading to remote clouds will always be profitable in terms of time and energy. The proposed framework called Circa runs on the iPhone operating system (iOS) platform and takes advantage of the surrounding iOS devices. Nearby devices are listed with the help of the iBeacon technology that allows local devices to discover each other. Then the offloading task is distributed to these devices. Experiments with different task allocation algorithms show that power consumption and task completion time have decreased.

Although there are good results from offloading in MCC, the necessity of using MEC, which brings resources closer to the user, has emerged for applications requiring low latency. Recent studies are more focused on this. At this point, containers are preferred on edge servers because they are more lightweight than VMs, and their deployment is faster.

Tang et al. [29] present a study that focuses on applications with real-time requirements in autonomous vehicles. Edge computing brings the computations and storage closer to the user, as opposed to cloud computing. For this reason, authors used edge computing to perform fast calculations in autonomous vehicles. Network edge has a Docker container-based framework. Here, there is a message processing layer that receives the messages and data from the autonomous vehicle. There are three separate managers that are responsible for images, containers and resources. The multiple-dimensional knapsack problem is used to offload to the right edge server. Also, the container manager uses a pre-run strategy. According to this strategy, some containers can be kept running with little resource expenditure before an offload request arrives. Thus, there is slightly less latency for time-critical applications.

The movement of mobile clients may cause interruptions in services. To prevent this, Ma et al. [30] state that services should be sent to the nearby servers on edge. A

container consists of layers. Actually, the main difference between a container and an image is the writable container layer over read-only layers. The data added, changed, or saved to the container is kept here. The base image layers are sent before the migration starts, and in the case of migration, it is considered sufficient to send only this thin writable layer. However, in write-heavy cases, it is recommended to use data volume.

As suggested by the study above, data volume is used by Campolo et al. [31] in their research. The aim of the study is the horizontal migration of tasks offloaded from the vehicle at the edge. The authors assumed that the route of the vehicle is known. In this way, the application is sent to the target host beforehand. In the study, service pre-relocation time and service downtime are examined. Freezing the container with the docker export command, sending the exported file, and importing it on the new host causes pre-relocation time. On the other hand, transferring the data volume and running the container on the new host with this volume causes downtime.

In [32], proactive service migration with container technology is proposed for stateless microservices. Data volume should be used because microservices are stateless. Data volume synchronization is done periodically from the source edge server to possible target servers. When migration is required, first data volume synchronization is performed, then the container is stopped, and data volume synchronization is performed again. After this point, the container is started on the target server, and user traffic is directed here from the old server.

Zhang et al. [33] prioritized checkpointing in their work. Using Checkpoint/Restore in Userspace (CRIU) serves to freeze a container in its last state. But since such migrations send the entire file system of the container, the amount of migrated data increases. Therefore, there is an extra load on the network. Also, the checkpointing process creates a delay. On the other hand, it would be pointless to use checkpoints for stateless containers. Adaptive compression algorithm is used for sending the file turned into a checkpoint. A fixed algorithm is not used because it is desired to use a compression algorithm that adapts to changing network conditions. The results show that the adaptive algorithm is practical.

Nadgowda et al. [34] present Voyager, which is a service for complete container state migration. Unlike classical checkpointing, the file system is sent between the host and target servers while the container on the host is still running. After the container starts running on the target host, it does remote reads and local writes. In other words, all data does not arrive at the target host, and it is retrieved in the background with lazy replication.

CHAPTER 4

PROPOSED MODEL

This section describes the model we recommend. The main components of the system and how they are used are explained.

4.1.Approach

In this study, we focused on developing a framework that provides distribution transparency so that developers who want to use it in their applications can benefit from this framework for offloading application pieces to nearby servers. The main idea is to use edge servers to run computation and storage intensive application parts of mobile applications on nearby resource-rich edge servers. At this point, there is a client-server relationship between the mobile application and the edge server. The mobile application acts as a client and offloads its processing-intensive parts to the edge server. At the same time, we have provided a structure suitable for service migration between edge servers to mitigate to potential problems caused by the mobility of the application user.

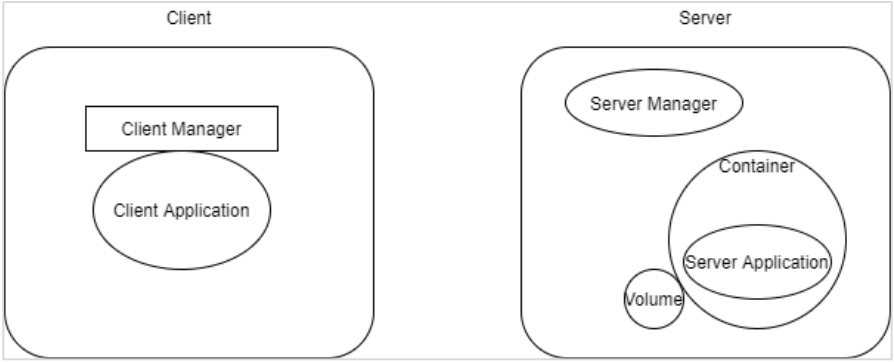

Figure 7: Client and server in the framework

Figure 7 shows the framework parts. The developer is responsible for developing the client application and the server application. The framework contains a server manager that is running on the server as an application. A client manager is a library imported into the client application. The client and server managers provide communication between client and server at first. Then the client application and server application can communicate freely. In addition, the client manager realizes the situation where the user gets closer to a new server and triggers the service migration. The new and the old server's server managers perform the migration. Using Docker containers on

the server speeds up offloading and migration. Also, using volume feature for containers lets us keep the necessary files needed for the containerized application together. The framework supports the Python programming language.

A snapshot of a sample system can be seen in Figure 8. When mobile users are connected to the edge server, a container for that client's application is started on the server by the server manager. This container runs the server application developed. The client manager receives the URI of the server application from the server manager and enables the client application and the server application to communicate. The server manager on a server can communicate with multiple clients.

Figure 8: A snapshot of a sample system

Figure 9 shows when one of the mobile users moves and switches to another edge server while there is an operation in progress on the server. The client manager connects to the new server manager and asks to get the volume from the old server and the result of the unfinished operation. This volume will be used to attach to the new container. When the result of the incomplete operation returns to the new server and then to the client, the communication with the old server is terminated. If there is no an incomplete operation on server, just volume transfer is performed, and new container is run on the new server.

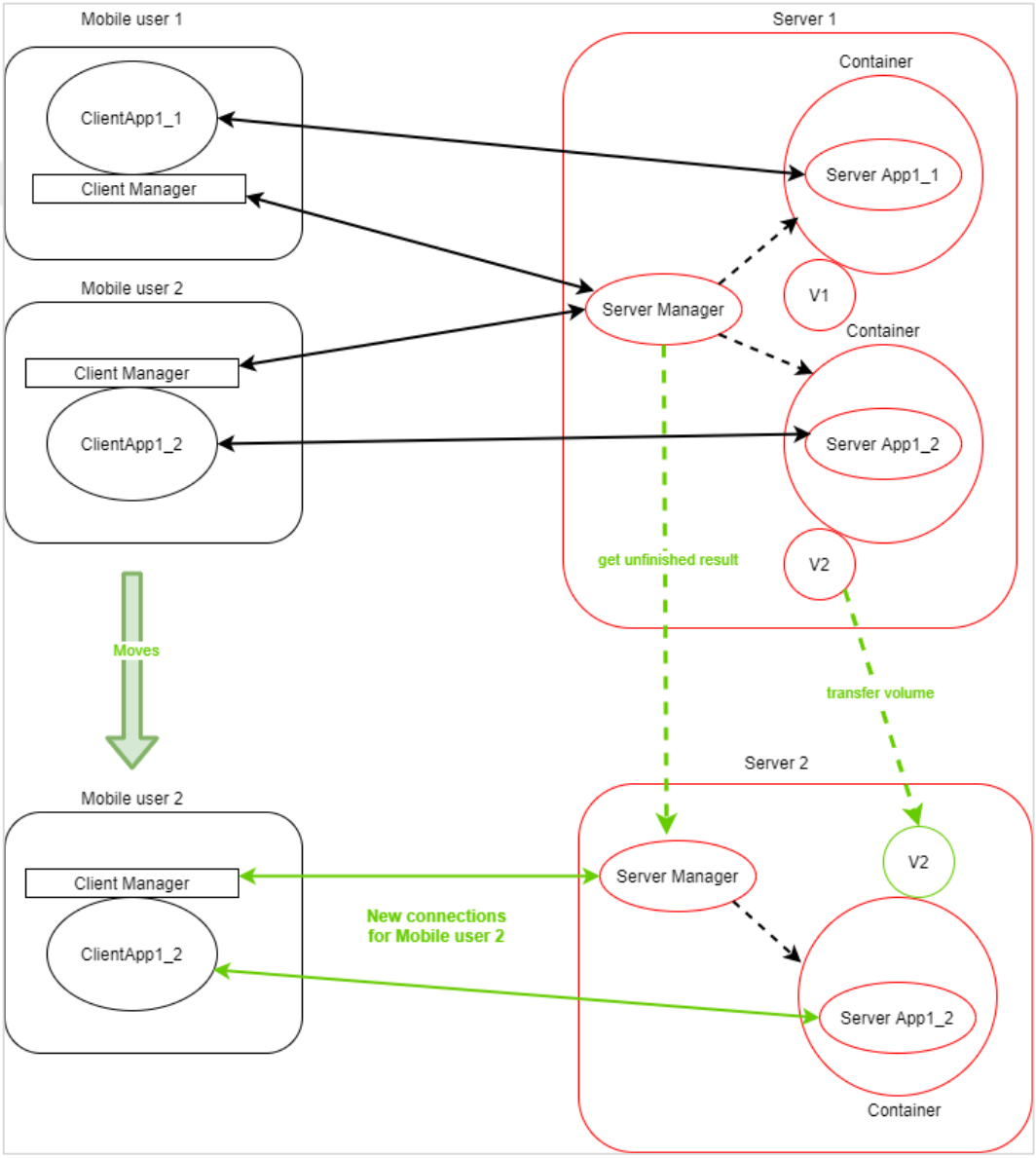

Figure 9: The view of migration when there is a processing on the server

4.2.Client-Side

The user and the application on the device that is used constitute the client-side. The client-side includes two main parts, client manager and client application. The client manager is imported into the client application as a library. Also, we assume that the client-side has limited resources and can properly communicate with the server-side.

4.2.1. Client Manager

The client manager is a library that is designed to serve the developer's client application. The developer can easily use the functions that will establish a connection with the server by importing the client manager into their program.

The main task of the client manager is to establish the communication between the client application and the server application. For this purpose, it uses a proxy of the server manager to use the exposed methods of the server manager. After running a container, it takes the URI of the server application to create a proxy object and give it to the client application. The client manager creates a unique client id and uses it when communicating with the server manager. In this way, it is known which client manager and which server manager are communicating.

In addition, the client manager follows the user location in a separate thread and initiates service migration between servers in case of server change.

4.2.2. Client Application

The developer imports the client manager while developing the client application. Through the methods of the client manager, the client application creates and uses the proxy object of the server application. The developer gives their application a specific name. This name is given to the server manager by the client manager at the beginning to use the correct base volume at the server. In this way, a copy of the base volume containing the necessary files for this application can be used when a new container is started on the server. Information about volumes is described in more detail in Section 4.3.3.

4.3.Server-Side

Server-side refers to the parts that the client uses on the server. It includes four main parts, server application, container, volume, and server manager. These are located at the edge server. Server manager is a generic application, and we assume it works all the time on edge servers. The server application is the code written by the developer, and it contains offloaded parts of the client application. The container exposes the server application, and a volume attached to it contains server application itself and its dependencies.

4.3.1. Server Application

The developer designs the server-side of their application according to their own requirements. At this point, there is no need to add an extra library like Pyro. The server application contains one class. A container will expose the methods of this class, and the client application uses these methods. The files that the server application needs will be accessed from the volume. The server application's file itself is located in the volume too. Since the container always looks for the same name, the name of the server application must always be the same for all applications: "serverToDo.py". Also, the name of the class must always be the same: "ToDoClass". On the other hand, the server application writes the result of an operation to a file in the volume, which can be used when the migration occurs.

4.3.2. Container

A Docker container is run on the edge server for each client connecting to the server. The container is started with a volume attached to it. Container's job is to expose the server application's class that is available in the volume via the Pyro library and write the server application's URI to a file in the volume. Therefore, the client manager can create an object of the server application's class, and the client application can use its methods. Containers work with different volumes because the needs for each application are different. Information about volumes is described in more detail in Section 4.3.3.

4.3.3. Volume

When the container uses the server application, there may be some files it will want to access. These files are mounted to the container via a volume. The server application developed is also located in this volume. Volume usage is required for persistent storage [21]. Therefore, in case of migration, the volume is transferred to the new server, and the container on the new server accesses the changed files.

There are different base volumes for various applications on the server because each application's required configuration file, database, etc., may differ. The developer must put this base volume on the server with the same name as mentioned in the client application. It will be baseVolume_"name". In our framework, host volumes are used. A host volume is a directory on the host machine, and it allows both the host and the container to reach it. Thus, the server manager is able to reach the files. Figure 10 shows the host volume usage.

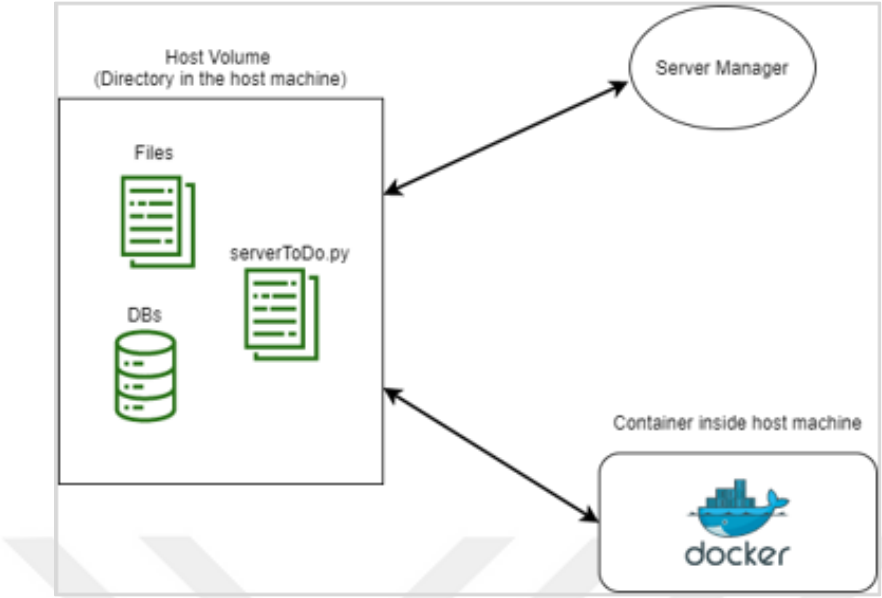

Figure 10: Host Volume

4.3.4. Server Manager

A server manager runs on an edge server. The server manager exposes its methods via the Pyro library. When a client manager creates the server manager's proxy object, it can access these methods remotely.

The server manager's main task is to run containers. It finds the correct base volume with the provided client application name and attaches a copy of it to the container. Then, it returns the server application's URI to the client by reading it from the container's volume. It uses the client id to track which client is working with which container and volume. The second task of the server manager is to manage the service migration in case of a location change. For this, it performs the volume transfer between the old and new servers and delivers the result of the incomplete operation to the client through the new server. Then, it deletes the old container and its volume.

4.4.Migration

As mentioned in Section 4.3.1, the developer develops the server application so that it writes the results of operations to a file in the volume. Each time a new process is started, the content of the file is cleared. In this way, if the content of this file is empty when there is a location change, it is understood that a process is in progress. When the client manager detects the location change while there is a process in progress in the server, it sends the old server manager's Pyro URI and its client id to the new server manager. Thus, the new server manager can create a proxy of the old server manager and reach the correct volume. Then it requests the result of the unfinished transaction. We assume that servers can communicate with each other. In the framework we

designed, the server information to which the client is connected is taken from a location module by the client manager.

The activity diagram of migration, while a code offloading operation is in progress, is shown in Figure 11. First, a volume transfer occurs at the beginning of the transfer. When the process is completed, the result is sent to the new server, and volume synchronization is made because there may have been changes in the volume files during the process. Then, the manager on the old server deletes the container and volume so that the resources are not restricted. After that, the connection is completely transferred to the new server.

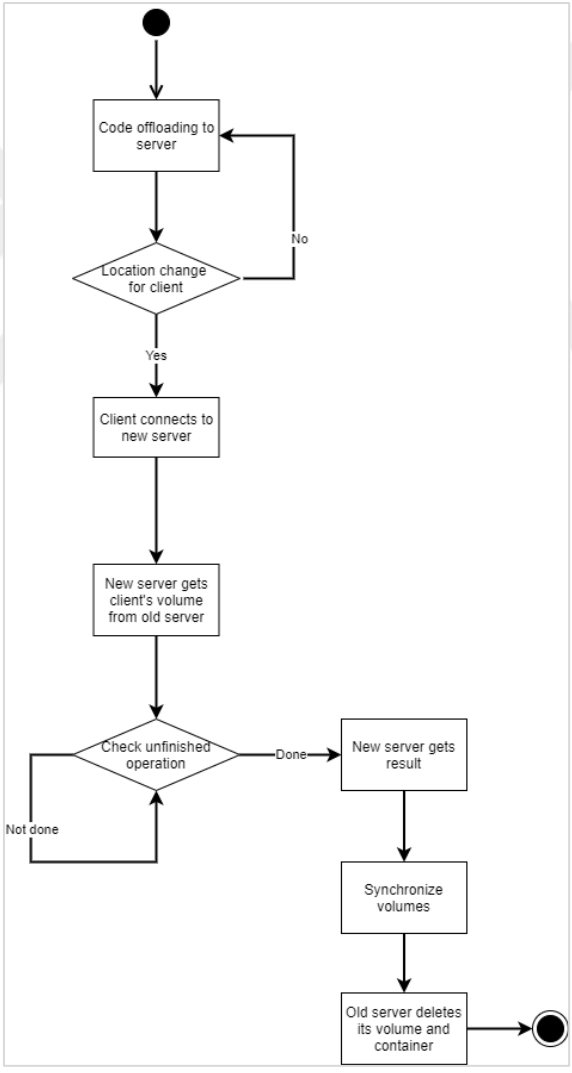

Figure 11: Activity diagram of a migration when there is a processing on the server

The activity diagram of migration when there is no offloading operation is shown in Figure 12. This time, there is only volume transfer but no synchronization.

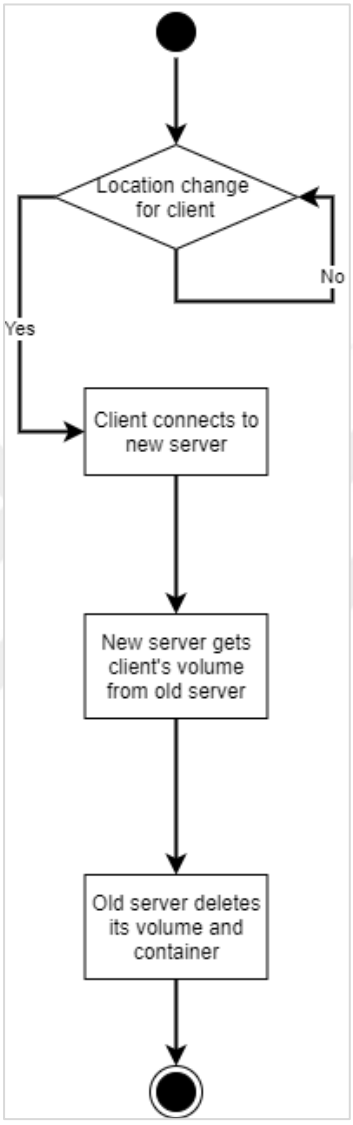

Figure 12: Activity diagram of a migration when there is no offloading

CHAPTER 5

PROTOTYPE IMPLEMENTATION AND SAMPLE APPLICATIONS

In this section, we describe the prototype implementation and how it works in a sample application. Section 5.1 describes how the server manager and client manager implemented and communicate. In Section 5.2, how the framework works is explained with example applications.

5.1. Prototype Implementation

In this section, the prototype implementation of the components that constitute the framework is detailed.

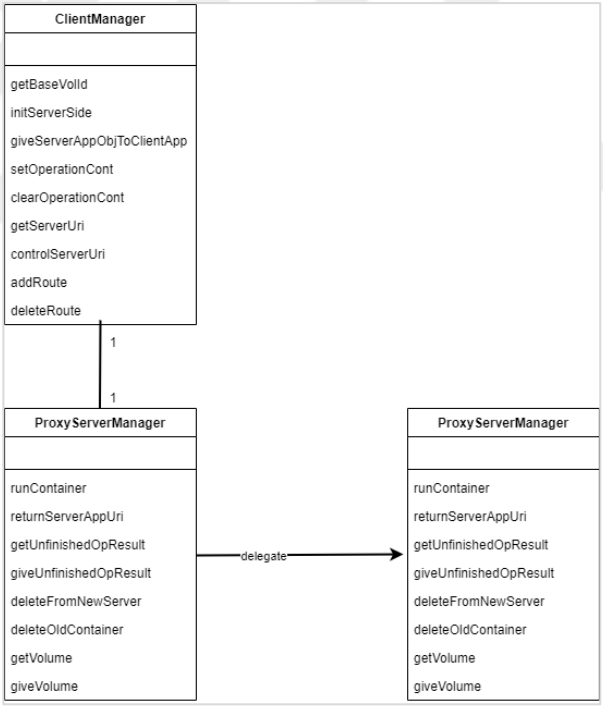

Figure 13: Class diagram for managers

Figure 13 shows the relationship between the client manager and the server manager. The client manager is a library imported into the client application and uses a proxy of the server manager. This proxy is created by the Pyro library via URI and uses the method calls as if it is the real one. The proxy directs its methods to the server manager on the remote server over the network. In the prototype implementation, it is assumed that there is no communication error between client and edge server or between edge servers.

5.1.1. Server Manager

The server manager is designed as a service running on the server machine. Server manager actually contains a single class named ServerManager, and its methods are exposed with Pyro by registering the class with a daemon. These methods are used by clients and server managers on other servers. To publish the Pyro object, the server manager creates a Pyro daemon and registers its class with the daemon.

The class in server manager has important functions to be used by other server managers and client managers. These are runContainer, returnServerAppUri, getUnfinishedOpResult, giveUnfinishedOpResult, deleteFromNewServer, deleteOldContainer, giveVolume and getVolume. The runContainer method uses the client id and base volume name received from the client application. The correct base volume is selected for the volume connected to the container with the base volume name, and a copy of the base volume is created. The client id is also used in its naming so that which client uses which volume is followed. Server manager needs a port that no one else uses to communicate with the container to be created. This method also finds an unused port for container communication. It writes the port to a file in the volume, and the container reads it from that file. As a final step, a container is started. A Docker container is built from an image. The correct image name to use the start of the container is obtained from the volume. At this point, we assume that the necessary image is available on the server. Our containerized code is generic for server applications that have previously mentioned file names and used to expose their classes, hence its methods. Exposing the methods creates a URI. In order to create a proxy object that will use these methods, this URI must be sent to the client. The returnServerAppUri method returns the URI required to connect to the server application used in the created container.

The getUnfinishedOpResult method is used when the client connects to a new server when there is a method call in processing at the server. The client manager gives the new server manager the URI of the old server manager. With this information, the new server manager connects the old server manager with the Pyro proxy and calls its giveUnfinishedOpResult method. The old server manager selects the correct volume with the client id information received from the new server manager and sends the volume using Rsync, waits for the unfinished operation to finish, sends the result to the new server, and performs volume synchronization. The reason for synchronization is the possibility of changes in the files in the volume during the operation. Receiving the result, the client calls the deleteFromNewServer method of the new server manager, and it calls the old server manager's deleteOldContainer method. This method finds the container to be stopped with the client id, deletes it and its volume. Thus, all files belonging to an old client are eliminated, and the resource is not limited to the server.

If there is no offloading operation when the client connects to a new server, getVolume method of the new server is used. It is similar to getUnfinishedOpResult method. After creating old server manager's proxy, it calls giveVolume method. This method only transfers the volume to the new server. Again, old container and its volume is deleted in the same way.

5.1.2. Client Manager

When a stationary user is in question, there is no migration. Figure 14 shows the sequence diagram of the client application with the client manager and the server manager to create the proxy of the server application.

The client manager is not a service like a server manager; it is a library imported to the client application. The client manager first learns the name for base volume from the client application, then establishes Pyro connection with server manager by creating its proxy. Apart from these, the client manager has two main methods that call the previously mentioned server manager methods used by the client application. First, initServerSide allows running a container on the server by calling proxy's runContainer method. The second one is the giveServerAppObjToClientApp method. It enables the client manager to create a proxy object of the server application with the URI information it receives from the server manager's returnServerAppUri method. Thus, the client application can call the methods of the server application. In addition, the client manager constantly checks whether the server to which the client is connected has changed in a separate thread with controlServerUri method. In case of change, it calls the server manager's necessary methods for migration: getUnfinishedOpResult or getVolume, and deleteFromNewServer. Also, setOperationCont and clearOperationCont methods are used by the client to set events based on whether there is an operation on the server. addRoute and deleteRoute methods are used to direct traffic to servers because Docker network range is the same for all servers.

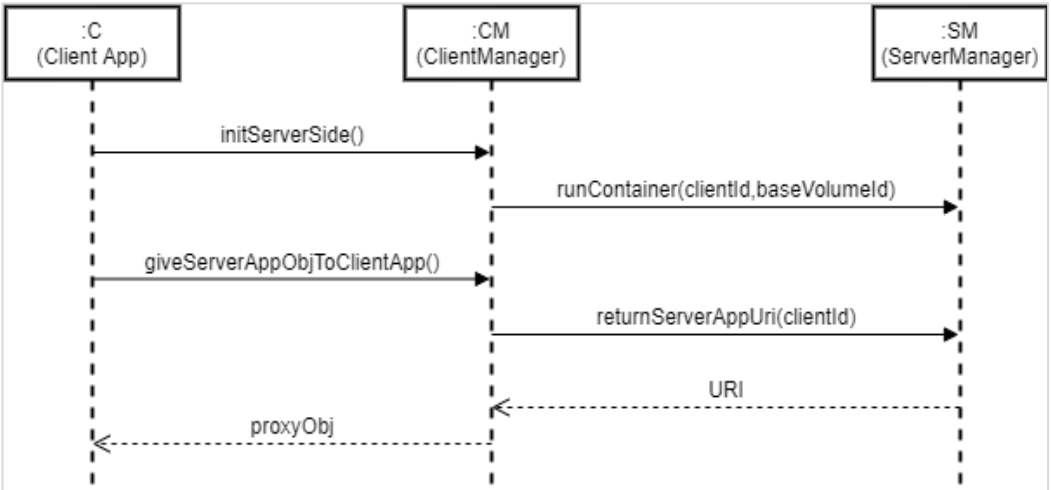

Figure 14: Sequence diagram of a client getting a proxy

5.1.3. Migration

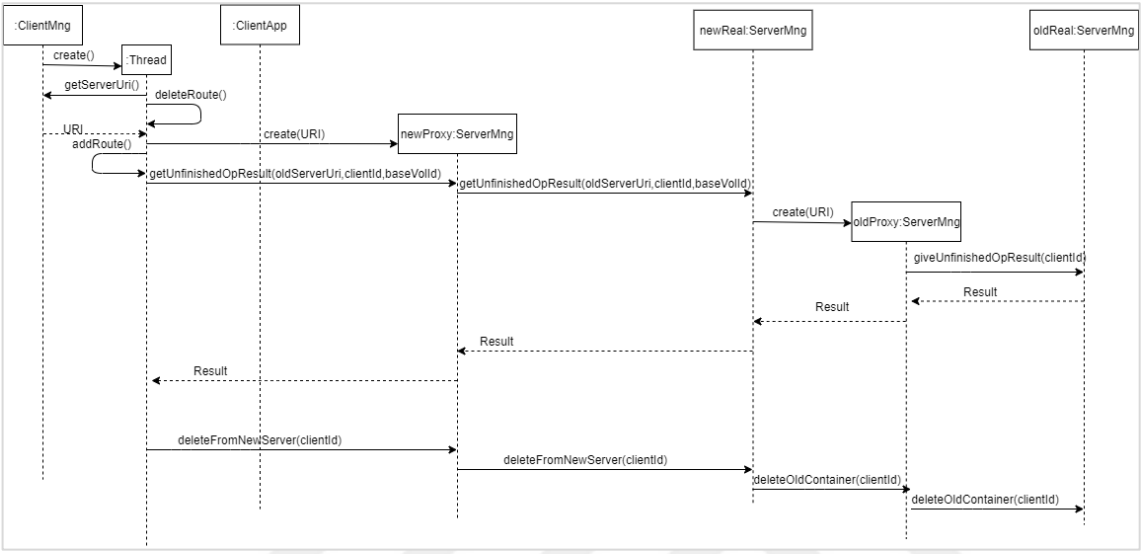

Figure 15: Sequence diagram of an application when migration happens while there is an offloading process

Figure 15 shows the sequence diagram when migration is needed while there is an operation going on at the server. The client manager checks a file where the server manager's URI is written in a separate thread. When the URI changes, the client manager gets new URI with getServerUri method. Then it creates new server manager's proxy and starts using it. Old URI is kept for use in migration. The client manager calls the getUnfinishedOpResult method of the new server manager. Thus, the new server manager asks the old server manager to transfer the volume first. Rsync is used for the volume transfer. It is a Linux-based tool that is used for transferring and synchronizing files and directories between computers. Then, the new server manager requests the result of the unfinished operation from the old server manager by giveUnfinishedOpResult. In the server application, the results are written to a file in the volume continuously. The file is cleared every time an operation starts. An empty file indicates a process in progress. When the result is written to the file, it means that the process is completed, and the result is sent to the new server manager. This way is preferred so that the result of the unfinished operation is not lost when the client's connection with the old server manager is lost. After that, the volume is synchronized with Rsync because there may have been changes in the volume files during the process. The second Rsync command only synchronizes the changed files. After the result is returned to the client, since the client no longer has anything to do with the old server, it asks the new server manager to delete the container and volume with the deleteFromNewServer method.

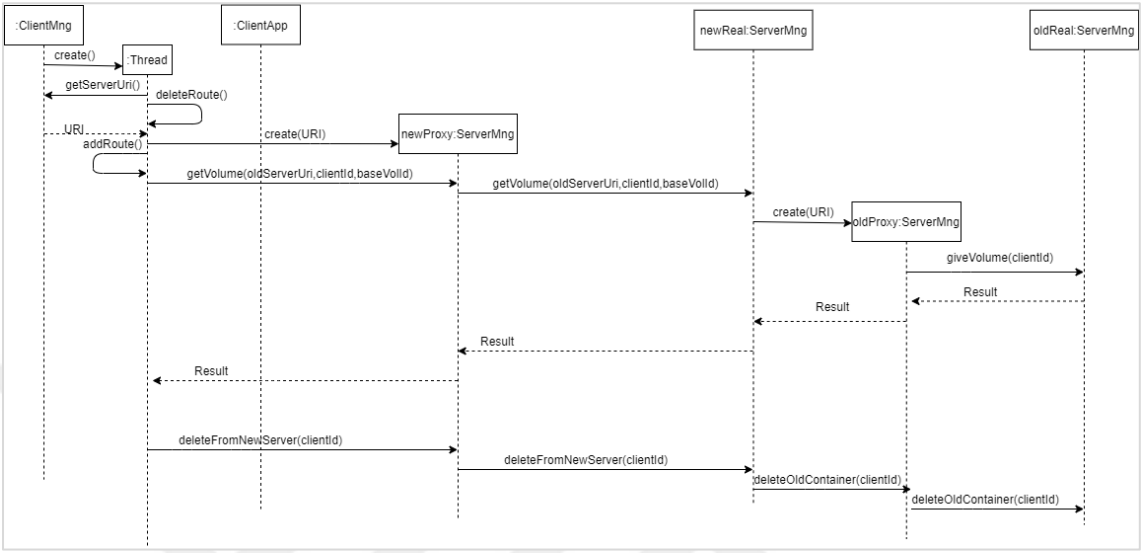

Figure 16: Sequence diagram of an application when migration happens while there is no offloading

Figure 16 shows the sequence diagram when there is no offloading. When the URI of the server manager is changed, the client manager gets URI with getServerUri method and creates a proxy. This time getVolume method is called. It calls the giveVolume method of the old manager. This method performs only volume transfer, no synchronization. After the volume is transferred deleteFromNewServer method initiates deletion of the old container and its volume.

5.2. Sample Applications

In this section we present two sample applications[1] that implements our offloading framework. The environment includes a host machine as a client and two VMs on it as edge servers.

[1] https://github.com/hknmstdr/Framework.git

The first application is the List application. The List application simply maintains a to-do list. On the client-side, the user is expected to add an item to the list. The list is actually located on the server. The input is sent to the server and added to the list. Then all the items in the list are requested by the client application. There are several pre-written items on the list. In order to carry out experiment, some delay has been set between item readings so that the reading process takes time, similar to a heavy computing job, and the migration is more easily observed while the operation continues.

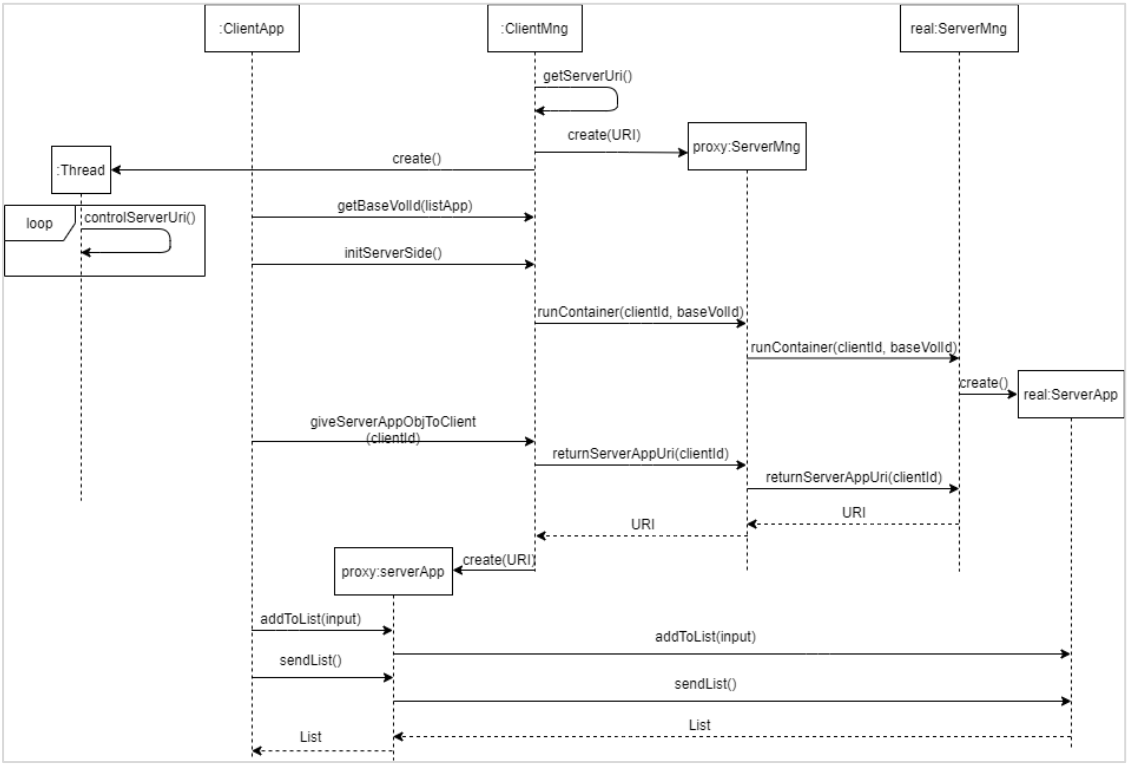

Figure 17: Sequence diagram of List application without migration

Figure 17 shows the sequence diagram of this application when there is no migration. The client manager learns the required name for the base volume to be used for that client on the server with the getBaseVolId method. For this application, this is "listApp". Meanwhile, when the server manager starts running, it writes its URI to a shared file between the host machine and VMs. The client manager gets this URI with getServerUri method to create a proxy of the server manager with the Pyro library. Then, it creates a unique client id. The client application calls the initServerSide and giveServerAppObjToClientApp methods of client manager to start a container on the server. It makes a proxy object of the class in the server application ready to be used in the client application.

After adding a new item to the list with the addToList method, all items on the list are read and returned from the server with the sendList method. These methods are the methods of the server application, and the client application calls these methods using its proxy.

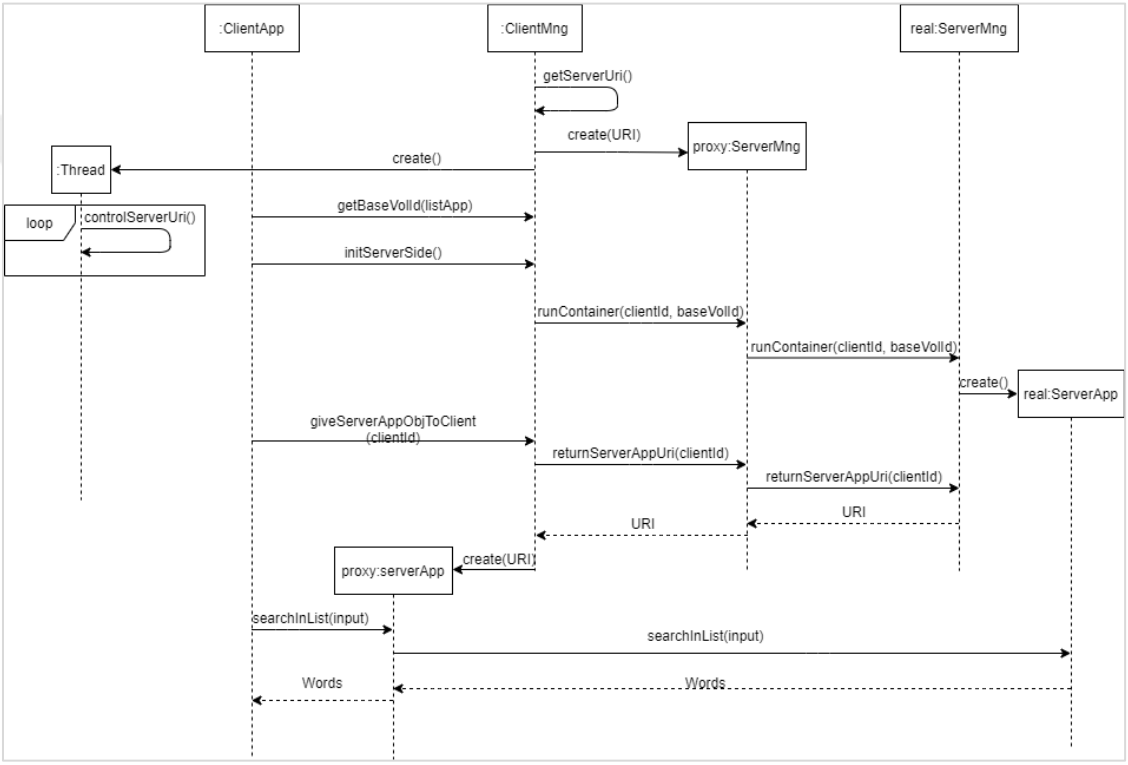

Figure 18: Sequence diagram of Search application without migration

We have developed a search application as the second example. This time, there is a dictionary database on the server, and the client application gets four letters from the user. These letters are sent to the server application as input. The server application searches for words containing the combination of these letters in the database and returns the result to the client application. Figure 18 shows the sequence diagram of this application when there is no migration. The same explanations as Figure 17 apply. The only difference is that the client application sends letters to the server using proxy's searchLetters method and receives the result.

```
# set base image (host OS)
FROM python:3.8

# set the working directory in the container
WORKDIR /code

# copy the dependencies file to the working directory
COPY requirements.txt .

# install dependencies
RUN pip install -r requirements.txt

# copy the content of the local src directory to the working directory
COPY src/ .

# command to run on container start
CMD [ "python", "./containerApp.py" ]
```

Figure 19: Dockerfile commands

We assume that the image used to create the container is already on the server. Dockerfile commands used to create this image are shown in Figure 19. FROM command is used for a base image. Since we used Python in our container, our base image is Python. Then we need to set the working directory in the container with WORKDIR. If there are dependencies, we need to inform and install them in the container. Our requirements file consists of only Pyro library. After that, the container application's directory that we used on our host machine is copied to the working directory in the container. Lastly, CMD command runs the application in the working directory when a container is started from the image.

5.3. Experiments

In the setup we designed to use these applications, two VMs running a Docker engine (version 20.10.7) with Ubuntu 20.04 LTS installed on a Windows 10 machine were used. The host machine is equipped with Intel core i5 up to 2.80 GHz and 16 GB RAM. VMs represent servers and runs the server managers, while the host machine represents the client and runs the client applications. Also, the Python version used in applications is 3.8.

Docker network range is 172.17.0.0/16 for our setup, and it is the same for both VMs. We need to direct traffic to this subnet on VM's IP address for our client applications. Therefore, we need to add a static route to our host's routing table in the client application's beginning. In this way, client application requests that want to reach address 172.17.x.x are directed to the VM.

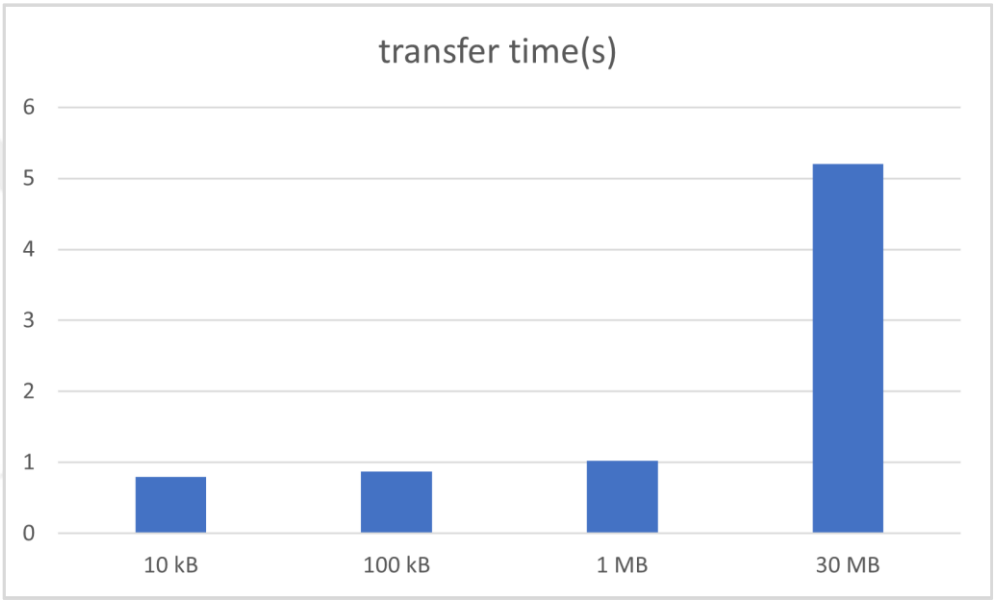

Figure 20 : Transfer times between VMs

Figure 20 shows the average transfer times for 10 kB, 100 kB, 1 MB, and 30 MB volumes when migration occurs between two VMs. These transfers occur while the processing-intensive operation continues on the old server. In other words, if the unfinished operation lasts longer than this time, the time it will take to run the container on the new server will only be the time spent on volume synchronization. This time depends on how many changes have been made, but in cases where there are no changes, synchronization control is less than a second.

Figure 21 and Figure 22 show the average time it takes for the client to get the same result from the server when there is and is not migration in applications. Times can change according to processing time, but the important criteria is their difference. In the case of migration, it takes an average of 1.2 seconds longer.

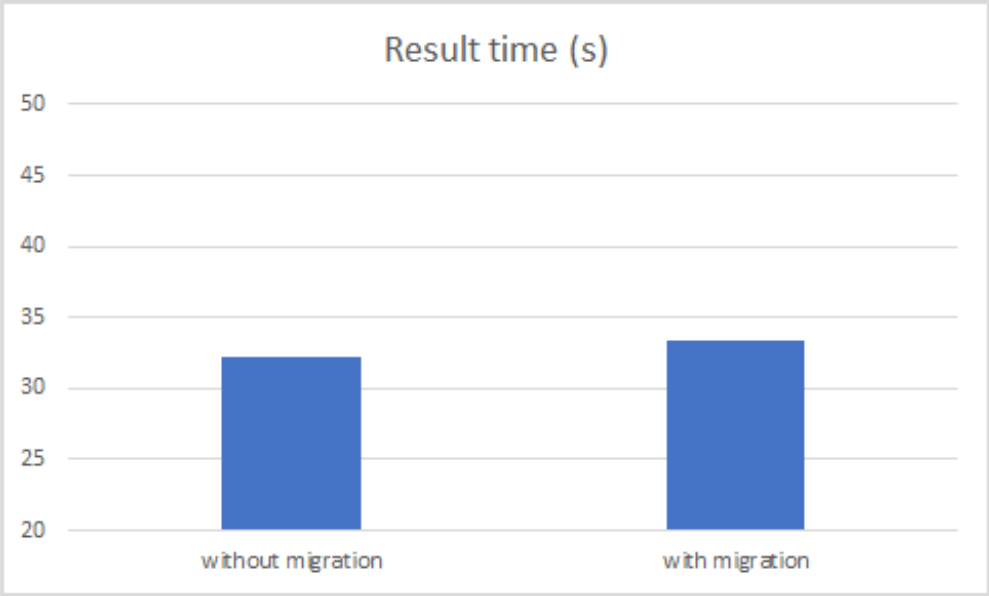

Figure 22 : Result return times in Search application

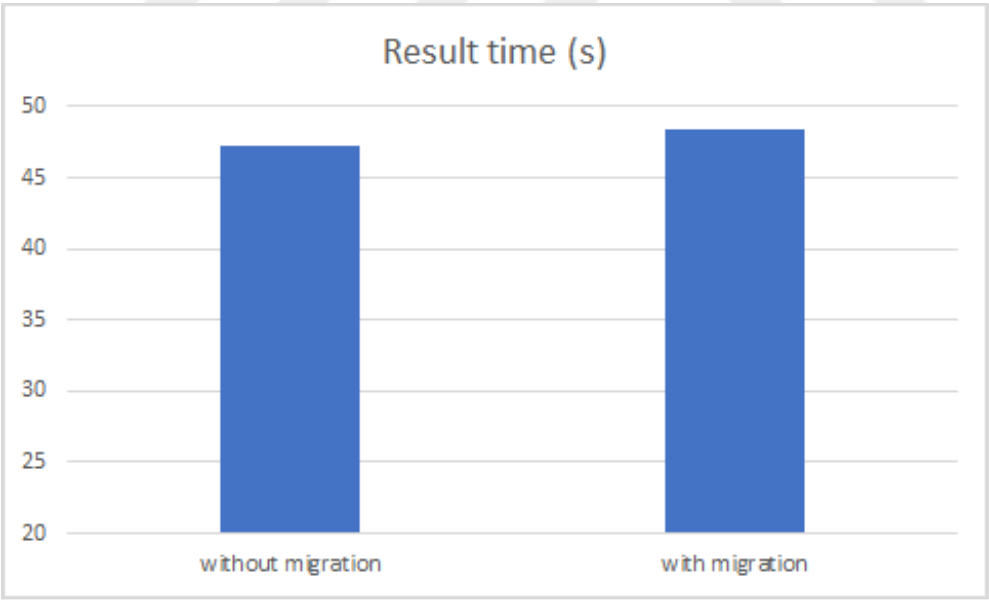

Figure 21 : Result return times in List application

Service handover times are shown in Figure 22 and Figure 23. In non-offloading situation, this time is the period between the start of the migration and the moment when the service on the new server is ready. On the other hand, in the case of offloading, this time is the period between the moment when the client receives the result of the unfinished operation and the moment the service on the new server is ready. Since the volume sizes are different (bigger for Search application) handover times are different for no offloading. However,

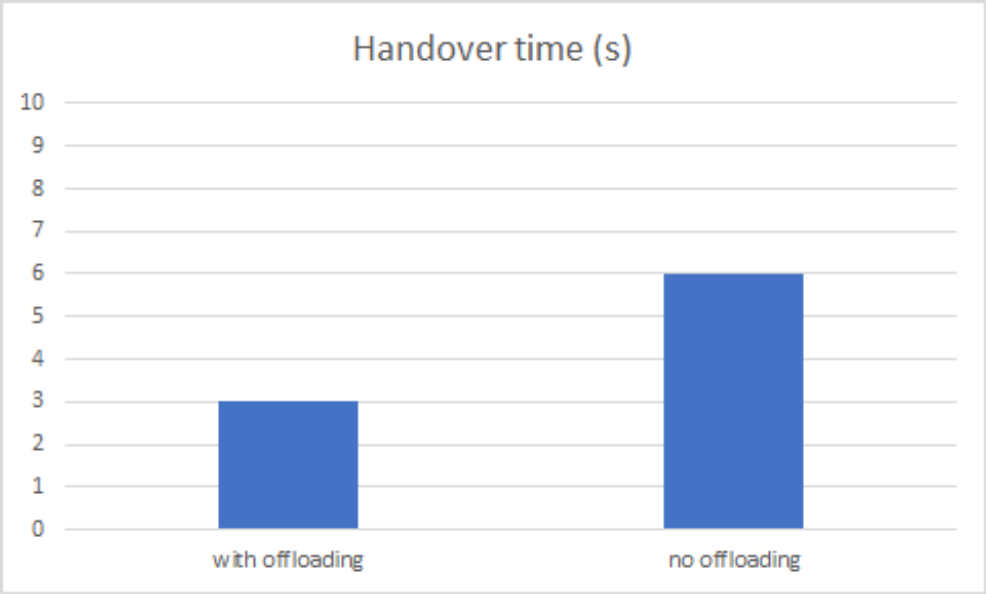

Figure 24 : Service handover times in List application

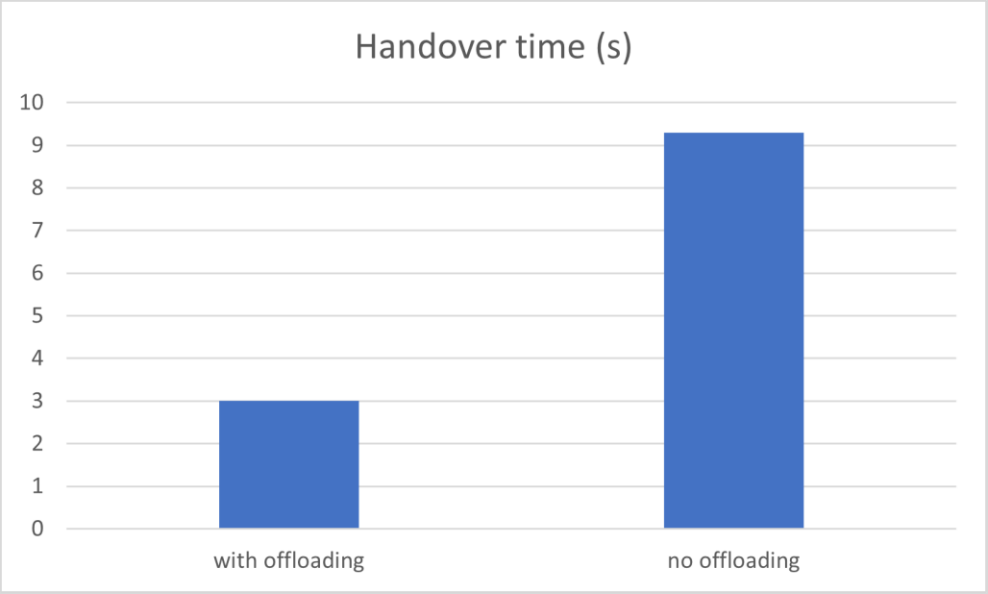

Figure 23 : Service handover times in Search application

it takes an average of 3 seconds with offloading situations. We assume that volume transfer is completed before unfinished operation ends on server.

CHAPTER 6

CONCLUSION

The goal of this thesis is to present a framework that enables code offloading in mobile applications and migrates services between servers at the network edge. Although mobile devices are advanced in terms of resources, they fall short of what applications can do. Processing-intensive sections can be run on servers at the network edge to overcome the resource limitations of mobile devices. Therefore, we have taken an approach where the developer will develop an application in two parts as client and server. The developer runs the computation-intensive parts on the server and uses the results on the client, that is, on the mobile device, through proxy objects. In addition, we ensure service continuity without losing information by migration between servers in cases where the mobile device is relocated.

In our study, we used proxy objects for code offloading. We have shown how useful the Python Pyro library is in creating applications with objects that can talk over the network with minimal effort. We used Docker containers for fast deployment and lightweight virtualization on edge servers. We used volume to make the container stateful, and we transferred this volume between servers in cases where migration is required. The framework we offer allows developers to easily create applications that meet distribution transparency and automatic migration goals.

We tested our framework with simple remote method calls. In more complex client applications, the expected result from the edge server may be used in another part of the application. In this case, the client application is halted only at the point where the result is needed. This behavior allows the client application to continue until the result is needed while the operation continues on the edge server.

In addition, we assumed that there were no connection problems in the prototype implementation of the framework we developed. As future work, precautions should be taken for connection problems between the client and the server. In case of a short-term disconnection, the client can be quickly reconnected by keeping the container on the server running. However, keeping the container running during long-term disconnections will cause server resources to be limited. Thus, after a certain amount time, this container and its volume can be deleted, and a completely new connection is started. If there is a connection problem between edge servers during migration, the volume transfer may be interrupted. In this case, it is necessary to synchronize the volumes after restoring the connection.

In order to use this framework, edge servers are needed at points close to the mobile user. That is why, for now, it can be offered to mobile users in areas conducive to technology development, such as technopoles, universities, and smart spaces. An even more advanced

scenario would be to use it in smart cities. Considering that the accessibility of edge servers covers an entire city, more complex users, such as connected vehicles, can also be evaluated.

In the framework we created, we chose to make a remote procedure call using the Pyro library, but on the other hand, using RESTful services on edge servers may be an alternative approach, or other remote procedure call models may be utilized. We assumed that the necessary container images are present on the servers. A connection with a repository such as Docker Hub can be made more attractive for more users. In our study, we used a shared document to follow the servers in the network. At this point, the objects in the network can be tracked using the Pyro name server. In addition, a framework can be developed in terms of security issues such as authentication for sharing information on servers.

REFERENCES

[1] N. Abbas, Y. Zhang, A. Taherkordi and T. Skeie, "Mobile Edge Computing: A Survey," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 450-465, 2017.

[2] I. d. Jong, "Pyro - Python Remote Objects - 4.80," [Online]. Available: https://pyro4.readthedocs.io/en/stable/. [Accessed 14 08 2021].

[3] Docker, Docker, [Online]. Available: https://www.docker.com. [Accessed 15 08 2021].

[4] M. Kaya and A. Koçyiğit, "Mobil Uygulamalarda Vekil Tabanlı Kod Taşıma Yönteminin Farklı Seviyelerdeki Bulut Bilişim Altyapılarının Kullanılması Durumundaki Başarımının Karşılaştırılması," in *UYMS*, 2014.

[5] F. Khodadadi, A. V. Dastjerdi and R. Buyya, "Internet of Things: an overview," in *Internet of Things*, 2016, pp. 3-27.

[6] H. Flores, P. Hui, S. Tarkoma, Y. Li, S. Srirama and R. Buyya, "Mobile Code Offloading: From Concept to Practice and Beyond," *IEEE Communications Magazine*, vol. 53, no. 3, pp. 80-88, 2015.

[7] W. Huaijun, T. Ling, L. Junhuai and G. Zhe, "Research and Implementation of Mobile Cloud Computing Offloading System Based on Docker Container," in *2017 10th International Symposium on Computational Intelligence and Design (ISCID)*, Hangzhou, 2017.

[8] K. Sinha and M. Kulkarni, "Techniques for Fine-Grained, Multi-site Computation Offloading," in *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, Newport Beach, 2011.

[9] M. Satyanarayanan, P. Bahl, R. Caceres and N. Davies, "The Case for VM-Based Cloudlets in Mobile Computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14-23, 2009.

[10] D. Huang and H. Wu, Mobile cloud computing: foundations and service models, Morgan Kaufmann Publishers, 2017.

[11] "Virtualization," IBM Cloud Education , 19 June 2019. [Online]. Available: https://www.ibm.com/cloud/learn/virtualization-a-complete-guide. [Accessed 15 08 2021].

[12] A. u. R. Khan, M. Othman, F. Xia and A. N. Khan, "Context-Aware Mobile Cloud Computing and Its Challenges," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 42-49, 2015.

[13] S. Abolfazli, Z. Sanaei, E. Ahmed, A. Gani and R. Buyya, "Cloud-Based Augmentation for Mobile Devices: Motivation, Taxonomies, and Open Challenges," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 1, pp. 337-368, 2013.

[14] Amazon Web Services, "Amazon EC2," Amazon Web Services, 2021. [Online]. Available: https://aws.amazon.com/tr/ec2/?ec2-whats-new.sort-by=item.additionalFields.postDateTime&ec2-whats-new.sort-order=desc. [Accessed 15 08 2021].

[15] Microsoft, "Azure," Microsoft, 2021. [Online]. Available: https://azure.microsoft.com/en-us/?form=MY01SV&OCID=MY01SV. [Accessed 15 08 2021].

[16] M. R. Rahimi, N. Venkatasubramanian, S. Mehrotra and A. V. Vasilakos, "MAPCloud: Mobile Applications on an Elastic and Scalable 2-Tier Cloud Architecture," in *2012 IEEE Fifth International Conference on Utility and Cloud Computing*, Chicago, 2012.

[17] M. T. Beck, M. Werner, S. Feld and T. Schimper, "Mobile Edge Computing: A Taxonomy," in *AFIN 2014 : The Sixth International Conference on Advances in Future Internet*, Lisbon, 2014.

[18] W. Felter, A. Ferreira, R. Rajamony and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Philadelphia, 2015.

[19] J. Bhimani, Z. Yang, M. Leeser and N. Mi, "Accelerating big data applications using lightweight virtualization framework on enterprise cloud," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Waltham, 2017.

[20] Docker, "Use containers to Build, Share and Run your applications," Docker, [Online]. Available: Docker.com/resources/what-container. [Accessed 15 08 2021].

[21] Docker Inc., "Docker overview," 2021. [Online]. Available: docs.docker.com/get-started/overview/. [Accessed 15 08 2021].

[22] Docker, "Use Volumes," Docker, 2021. [Online]. Available: docs.docker.com/storage/volumes. [Accessed 15 08 2021].

[23] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra and P. Bahl, "MAUI: making smartphones last longer with code offload," in *MobiSys '10: Proceedings of the 8th international conference on Mobile systems, applications, and services*, San Francisco, 2010.

[24] S. Kosta, A. Aucinas, P. Hui, R. Mortier and X. Zhang, "ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *2012 Proceedings IEEE INFOCOM*, Orlando, 2012.

[25] S. Wu, C. Niu, J. Rao, H. Jin and X. Dai, "Container-Based Cloud Platform for Mobile Computation Offloading," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Orlando, 2017.

[26] X. Xu, X. Zhang, X. Liu, J. Jiang, L. Qi and M. Z. A. Bhuiyan, "Adaptive Computation Offloading With Edge for 5G-Envisioned Internet of Connected Vehicles," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 8, pp. 5213-5222, 2021.

[27] M. Kaya, A. Koçyiğit and P. E. Eren, "An adaptive mobile cloud computing framework using a call graph based model," *Journal of Network and Computer Applications*, vol. 65, pp. 12-35, 2016.

[28] X. Lin, J. Jiang, C. H. Y. Li, B. Li and B. Li , "Circa: collaborative code offloading among multiple mobile devices," *Wireless Networks*, vol. 26, p. 823–841, 2020.

[29] J. Tang, R. Yu, S. Liu and J.-L. Gaudiot, "A Container Based Edge Offloading Framework for Autonomous Driving," *IEEE Access*, vol. 8, pp. 33713 - 33726, 2020.

[30] L. Ma, S. Yi and Q. Li, "Efficient service handoff across edge servers via docker container migration," in *SEC '17: Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, San jose, 2017.

[31] C. Campolo, A. Iera, A. Molinaro and G. Ruggeri, "MEC Support for 5G-V2X Use Cases through Docker Containers," in *2019 IEEE Wireless Communications and Networking Conference (WCNC)*, Marrakesh, 2019.

[32] I. Farris, T. Taleb, H. Flinck and A. Iera, "Providing ultra-short latency to user-centric 5G applications at the mobile network edge," *Transactions on Emerging Telecommunications Technologies*, vol. 29, no. 4, 2017.

[33] X. Zhang, W. Wu, C. Zhang and W. Song, "Dynamic Adaptive Network Edge Service Migration Method Based on a Docker Container," in *2019 IEEE 5th International Conference on Computer and Communications (ICCC)*, Chengdu, 2019.

[34] S. Nadgowda, S. Suneja, N. Bila and C. Isci, "Voyager: Complete Container State Migration," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, Atlanta, 2017.