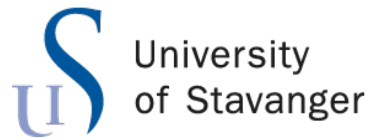
University of Stavanger

Faculty of Science and Technology

MASTER'S THESIS

Study program/Specialization: Computer Science	Spring semester, 20.21. Open ~~/ Restricted~~ access
Writer: Mohammed Ashraff Hathibelagal	……………………………………… (Writer's signature)
Faculty supervisor: Gianfranco Nencioni External supervisor(s): Rosario Garroppo	
Thesis title: Testbed for Analyzing the Migration of MEC-Assisted 5G-V2X Services	
Credits (ECTS): 30	
Key words: MEC, V2X, ETSI, Edge, Docker, 5G	Pages: …67………………… + enclosure: ………… Stavanger, June 10th/2021……………….. Date/year



Faculty of Science and Technology
Department of Electrical Engineering and Computer Science

Testbed for Analyzing the Migration of MEC-Assisted 5G-V2X Services

Master's Thesis in Computer Science

by

Mohammed Ashraff Hathibelagal

Internal Supervisor

Gianfranco Nencioni

External Supervisor

Rosario Garroppo

Abstract

With the advent of 5th generation mobile networks (5G), the automotive industry can manufacture vehicles that are capable of communicating not only with each other, but also with everything else around them using an ultra-reliable communication channel that offers high data rates and extremely low latencies. This allows the development of applications that can offer advanced features such as autonomous navigation, remote driving, non-line-of-sight awareness, and vehicle platooning. Such applications are expected to leverage the Multi-Access Edge Computing (MEC) paradigm and support User Equipment (UE) mobility.

In this thesis, a testbed was built to compare three different strategies for migrating three different MEC applications offering V2X services under two different network conditions. The applications were containerized using Docker and were capable of communicating with the ETSI MEC sandbox using the recommended open APIs it exposed. The three strategies were compared based on viability, service downtime observed, and amount of state preserved after the migration.

The results obtained from this testbed showed that that all the three strategies were viable. But there was also a very obvious trade-off to make in any migration scenario: either decrease service downtime or decrease the amount of state preserved after the migration. This meant that applications that needed a high level of user or application-specific state preservation tended to experience more service downtime.

Acknowledgements

I'm extremely grateful to my supervisors Dr. Gianfranco Nencioni and Dr. Rosario Garroppo. This thesis wouldn't have been possible without their guidance, feedback, and patience.

Contents

Abstract i

Acknowledgements ii

1 Introduction 1

2 Background 4
- 2.1 The MEC Host . . . 5
- 2.2 An MEC Application . . . 6
- 2.3 The MEC Platform . . . 6
- 2.4 System Level Management Entities . . . 7
  - 2.4.1 The MEC Orchestrator . . . 7
- 2.5 Host Level Management Entities . . . 7
- 2.6 MEC Services . . . 8
  - 2.6.1 Additional Support for V2X Services . . . 9
- 2.7 Mobility . . . 9
  - 2.7.1 UE Mobility in a 5G RAN . . . 10
  - 2.7.2 Application Mobility in an MEC System . . . 10

3 Related Works 11
- 3.1 Virtualization or Containerization . . . 11
- 3.2 Earlier Testbeds . . . 12
  - 3.2.1 Service Replicated on Multiple Edge Hosts . . . 13
  - 3.2.2 Support for Live Migration . . . 14
  - 3.2.3 A Two-Phase Approach . . . 15
  - 3.2.4 An Application-Aware Strategy . . . 16

4 A Baseline Testbed Implementation 18
- 4.1 Hardware and Software Specifications . . . 18
- 4.2 Phases . . . 19
  - 4.2.1 Service Pre-relocation . . . 19
  - 4.2.2 Service Relocation . . . 20
  - 4.2.3 Role of the V2X Application . . . 21

5 A More Stateful Testbed **22**
5.1 Approach . . . 23

6 Testbed With Support for Live Migration **25**
6.1 The CRIU Project . . . 26
6.2 Approach . . . 26

7 Experimental Setup **29**
7.1 The ETSI MEC Sandbox . . . 29
7.1.1 The APIs . . . 29
7.1.2 The Scenario . . . 31
7.2 The API Gateway . . . 32
7.3 Applications . . . 32
7.3.1 Basic Application . . . 33
7.3.2 User State-Preserving Application . . . 34
7.3.3 Application With a Stateful Workload . . . 34
7.4 Variable Network Conditions . . . 36

8 Results **37**
8.1 Container Sizes . . . 38
8.2 Duration of Pre-relocation Phase . . . 39
8.2.1 Time Spent on the Source MEC Host . . . 39
8.2.2 Time Needed to Transfer the Tarball . . . 41
8.2.3 Time Spent on the Target MEC Host . . . 41
8.3 Duration of Relocation Phase . . . 42
8.3.1 Time Spent on the Source MEC Host . . . 42
8.3.2 Time Needed to Transfer Data . . . 44
8.3.3 Time Spent on the Target MEC Host . . . 45
8.4 Analysis . . . 46
8.4.1 Service Downtime . . . 47
8.4.2 Amount of state preserved . . . 48
8.4.3 Viability . . . 48

9 Conclusion and Future Works **50**

List of Figures **51**

List of Tables **53**

A Code Listings **54**
A.1 Application A1 . . . 54
A.1.1 Source Code: app.js . . . 54
A.1.2 Source Code: package.json . . . 55
A.1.3 Dockerfile . . . 55

A.2 Application A2 55
A.2.1 Source Code: app.js 55
A.2.2 Source Code: package.json 57
A.2.3 Dockerfile 58
A.3 Application A3 58
A.3.1 Source Code: app.js 58
A.3.2 Source Code: package.json 59
A.3.3 Dockerfile 59
A.4 Gateway Configuration 60
A.4.1 Source Code: httpd.conf 60
A.4.2 Dockerfile 62
A.5 Congestion Implementation 62
A.5.1 Source Code: congestion.sh 62

Bibliography **64**

Chapter 1

Introduction

The experience of being on the road is going to change significantly as telecommunications service providers transition to 5th generation mobile networks (5G). In the coming years, we can expect more optimized road traffic, better in-vehicle infotainment services, and more road safety.

With 5G, the automotive industry and roadside-infrastructure manufacturers can make use of networks that offer not only ultra-low latencies and higher peak data speeds, but also assured qualities of service. This allows them to create and support novel applications for use cases such as advanced driver assistance, vehicle platooning, and eventually, even fully autonomous driving.

Most of these use cases fall under the purview of V2X, a hypernym that stands for Vehicle-to-everything, and currently covers concepts involving wireless communications from Vehicles to Vehicles (V2V), Vehicles to Pedestrians (V2P), Vehicles to Cloud (V2C), and Vehicles to Infrastructure (V2I) [1]. Because the use cases can generally tolerate only minuscule latencies and expect extremely reliable communication channels, it is only 5G that can currently enable their widespread adoption.

Furthermore, to support such low latencies and high reliability, the cloud paradigm, which involves communicating with distant servers, is far from adequate. The compute and data storage resources required must be available closer to the edge of the network, and thus, closer to the end user.

Therefore, the transition to 5G also invariably involves adoption of the Multi-access Edge Computing (MEC) paradigm.

Because the underlying network supports User Equipment (UE) mobility, the MEC system needs to support application mobility in order to ensure service continuity to the end user. In other words, the MEC system needs to support the relocation of application instances and user-specific states from one MEC host to another as the UE moves out of and into the areas covered by the respective MEC hosts. This is especially important for V2X services because the primary actors involved, cars, are expected to cover large distances over a short period of time.

This project focused on creating a testbed for analyzing the feasibility and performance of three-different migration strategies for MEC-assisted 5G-V2X services. Based on important metrics such as viability, service downtime, and amount of data transferred given different network conditions, the strategies were compared and analysed. Because real vehicles and a full-fledged MEC system were not available for this project, a specific configuration of the ETSI MEC sandbox was used to simulate significant portions of the environment required.

This thesis builds on earlier migration strategies and testbed implementations and contributes the following:

1. Realistic MEC applications that use ETSI MEC service APIs, and that need different amounts of state preserved to function correctly
2. A full-fledged API gateway that allows the MEC applications to connect to the ETSI MEC sandbox, and thus the MEC service APIs
3. Analysis of migration strategies based on the movement of the Vehicular User Equipment (VUE) through the ETSI MEC sandbox's simulated environment
4. Analysis of the amounts of data transferred between MEC hosts during migrations, and how the amounts vary in the presence of network congestion.

This thesis is structured in a way that reflects the sequence of steps I underwent to arrive at the final results. It starts off with a background chapter that introduces the reader to the key components and concepts involved in an MEC system and how they interact with each other. The next chapter mentions the latest trends in the migration of edge applications and describes in detail earlier testbeds that are similar to the testbed implemented for this thesis.

Now that the reader has enough context, the next three chapters describe the implementation details of the testbed. Each of these chapters represents a step in the evolution of the testbed and care is taken to ensure that the reader understands the rationale behind the evolution.

The next chapter describes in detail the experimental setup of the testbed. In addition to explaining how the testbed interacts with the ETSI MEC sandbox, it also talks about the implementation details of the three MEC applications used in the experiments.

The chapter on results then explains the results obtained from running the experiments and presents them as several graphs. It also includes an analysis section that discusses how the migration strategies perform given specific performance metrics. Then, in the last chapter, I give the conclusion of this thesis and briefly describe how this project can be improved upon in the future.

The appendix contains the complete source code of all the applications developed during this project, including all the configuration files required to install their dependencies and to containerize them.

Chapter 2

Background

In addition to network slicing and Radio Access Network (RAN) enhancements, MEC is widely considered to be one of the key enabling technologies for meeting several demanding performance metrics targeted by 5G networks. Indeed, in a 5G network, several components of an MEC system can be mapped onto application functions (AF) [2].

The European Telecommunications Standards Institute (ETSI) Industry Specification Group (ISG) [3] has a series of group specifications and group reports that describe the components of MEC systems, outline their behaviors, and mention the use case scenarios they can or must support. Together, these specifications and reports rigorously define the MEC framework and provide two reference architectures.

One of the reference architectures is referred to as a generic architecture, and is the architecture that this thesis follows. The other is a variant that's ideal for an MEC system deployed in a Network Functions Virtualization (NFV) environment.

MEC, as mentioned earlier, allows for the placement of compute, storage, and network resources at the edge of a network, allowing telecommunications service providers to offer applications that demand extremely low end-to-end latencies and high bandwidth efficiencies. As such, the MEC framework and the reference architectures specify how these hardware resources are to

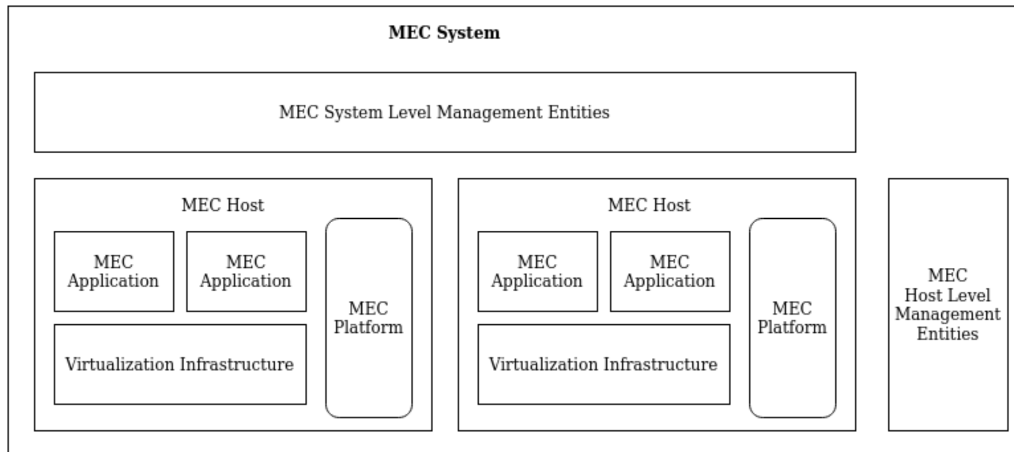


Figure 2.1: MEC System Overview

be used and the software entities that are to run on them [4]. Figure 2.1 gives a quick overview of the entities that are relevant to this thesis.

2.1 The MEC Host

MEC hosts are the most important entities in an MEC system. These are often powerful, datacenter-grade computers [5] placed at locations close to the telecommunications network operator's base stations, wireless network aggregation sites, and other wireless access points of presence.

There does not have to be a one-to-one mapping between a radio node and an MEC host. In fact, usually, an MEC host serves multiple radio nodes. The telecommunications network operator can decide the mapping based on technical parameters such as maximum expected workload, security requirements, Quality of Service (QoS) and Quality of Experience (QoE) expected by users of popular apps [6], and business parameters such as available site facilities [2].

The MEC host is where MEC applications run, so it is responsible for also hosting the virtualization infrastructure—which includes the data plane—and the MEC platform [4].

2.2 An MEC Application

An MEC application is a software-only entity that runs on virtualization infrastructure that is available on the MEC host. It generally interacts with MEC services, either consuming the data they provide or providing user data to them. MEC applications can also provide services themselves, and such services can be registered with the MEC platform.

MEC applications may or not be affected by UE mobility. Those that are affected are referred to as UE mobility-sensitive applications, and are the focus of this thesis.

2.3 The MEC Platform

An MEC application is expected to interact with the MEC system via the MEC platform that's present on the MEC host. Therefore, the MEC platform offers two important types of APIs: application support APIs and application service management APIs [7]. Using these APIs, an MEC application can perform tasks such as discovering available MEC services and registering any service it offers itself.

The MEC platform also provides reference points that are necessary for access to persistent storage, routing IP packets to MEC applications, and implementing DNS rules [4].

In a 5G deployment, the MEC platform is a host level entity that can play the role of an AF to interact with a 5G Network Function (NF). Indeed, in such a deployment, after an MEC application is instantiated on an MEC host, it sees traffic only after the MEC platform acts as an AF and requests the Policy Control Function (PCF) to steer traffic to it [2].

2.4 System Level Management Entities

An MEC system has several management entities. These entities can function either at the system level or at the MEC host level. The most important system level management entity is the MEC orchestrator, which, in a 5G deployment, can be mapped onto an AF that can interact with the 5G Network Exposure Function (NEF).

2.4.1 The MEC Orchestrator

Not all MEC applications can run on all MEC hosts. This is because MEC applications often have very specific latency and compute requirements, and an MEC host might not be able to satisfy all of them. Therefore, there is a need for an entity that can decide on which host an MEC application is to be instantiated. According to ETSI GS MEC 003, the MEC orchestrator serves as that entity [4]. As such, the orchestrator is responsible only for decision making and generating triggers, and relies on host level management entities to affect the actual instantiation, management and termination of MEC applications.

Additionally, the orchestrator can receive requests for application instantiation and termination from the Operations Support System (OSS), which is another system level management entity.

2.5 Host Level Management Entities

Host level management entities are responsible for carrying out MEC-specific tasks at the MEC host level. These are the entities that handle the actual lifecycles of MEC applications on individual MEC hosts. The following host level management entities are relevant to this thesis:

MEC Platform Manager - The platform manager is an entity responsible for deciding which services an MEC application is allowed to use, for maintaining the DNS configuration, and for reporting various events an

MEC application raises to the MEC orchestrator. Additionally, it is capable of sending fault and performance measurement reports to the orchestrator.

Virtualization Infrastructure Manager (VIM) - The VIM is another low-level management entity responsible for receiving triggers from the MEC orchestrator and allocating or releasing resources available on the MEC host. Consequently, this is the entity that actually instantiates the containers or VMs running the MEC applications.

2.6 MEC Services

An MEC service is a service registered in the MEC platform's service registry, and usually serves MEC applications or the MEC platform itself. MEC services can offer REpresentational State Transfer (REST)-based APIs, and can support popular formats for data exchange such as JavaScript Object Notation (JSON) and Protocol Buffers (Protobuf).

MEC services are usually not accessed directly by MEC applications. Instead, they're expected to be accessed via an API gateway, which is a feature that accompanies the MEC platform or built into it. This facilitates load balancing and mobility of the services.

An MEC system may offer several basic services as outlined in ETSI GS MEC 002 [8]. Two such services are relevant to this thesis:

Location Service - This service offers location-specific data. For example, it allows an MEC application to determine the location of a UE. The location can be the GPS coordinates of the UE, the cell ID, or both. This service also gives the MEC application the list of radio nodes that are currently being served by the MEC host.

Radio Network Information Service (RNIS) - This service offers detailed information about the state of the radio network. For example, an MEC application can use this service to get information about Radio Access Bearers (RABs), 5G UE measurement reports, and UE timing advance [9].

The information offered by both the aforementioned services can be consumed using both the request-response and publish-subscribe models. With the request-response model, the MEC application makes requests whenever necessary and immediately gets a response. With the publish-subscribe model, the MEC application is automatically notified in real time by the MEC service whenever important events occur, so long as the MEC application is subscribed to them.

2.6.1 Additional Support for V2X Services

According to ETSI GR 022 [10], V2X services need additional features and functionality from an MEC system.

A V2X service generally falls into one of the following use case groups: safety, convenience, vulnerable road user, and advanced driving assistance. Each of these use case groups have varying requirements. To cater to such requirements, the MEC system should:

1. allow V2X services to get information about the reliability of a communication channel
2. support multi-network and multi-vendor scenarios

To better support V2X applications, ETSI suggests an additional service called the V2X Information Service (VIS) [11]. By subscribing to VIS, a V2X application can get access to additional events such as those related to changes in network congestion and channel reliability.

2.7 Mobility

As mentioned earlier, VUEs can be expected to move from one location to another. Colloquially, this can be defined as the VUE moving from one serving cell to a neighboring one in the 5G RAN. In this section I briefly describe UE and application mobility.

2.7.1 UE Mobility in a 5G RAN

In the context of the 5G RAN, mobility involves the UE moving away from one 5G base station (gNodeB) Distributed Unit (gNB-DU), towards another gNB-DU. In order to ensure that the gNB-DU is aware of the signal strength the UE is experiencing, it sends measurement reports to the gNB-DU. The gNB-DU in turn sends these reports to its corresponding Central Unit (gNB-CU) for decision making.

Based on the information it gets, in order to minimize service degradation, the gNB-CU can initiate a mobility procedure that leads to the handover of the UE from the current gNB-DU to a different one [12].

It is worth mentioning that while the above procedure is sufficient in the case of intra-gNB mobility, in the case of inter-gNB mobility where the gNB-CU itself needs to be changed, an alternative procedure involving the Access and Mobility Management Function (AMF) is followed in order to complete the handover.

2.7.2 Application Mobility in an MEC System

Application mobility in an MEC system is the primary focus of this thesis. From the perspective of an MEC application, most of the details related to UE mobility are abstracted away. Nevertheless, application mobility is still closely connected to UE mobility, and is usually triggered by it. As mentioned earlier, as the UE moves from one serving cell to another in a 5G network, the MEC host that is serving it will eventually change too. Thus, to maintain service continuity and offer optimal performance, there is now a need to migrate the applications the UE was running on the previous MEC host, referred to as the source MEC host, to the latest chosen MEC host, referred to as the target MEC host [13].

In the case of common applications, which could already be present on the target MEC host, there may still be a need for migrating the user-specific state to avoid service disruptions.

Chapter 3

Related Works

Finding the optimal strategy for migrating applications from one MEC host to another is a very active research topic, and extensive literature has been published about it.

Migration strategies are closely coupled to the technologies used for resource isolation and hosting the applications. According to Wang et al's survey [14], the two most widely-used such technologies are: Virtual Machine (VM) technology and container technology.

3.1 Virtualization or Containerization

Applications packaged into full-fledged virtual machines offer the highest degree of resource isolation and control. They run their own copies of the operating systems they need and access emulated hardware [15]. They are thus completely separated from the other VMs that could be running on the same physical hardware. VMs have been traditionally used for migrating services from one edge host to another. However, they have a large overhead and do not allow for efficient utilization of resources [16, 17].

Containers, compared to VMs, are far lighter because they rely on the operating system running on the edge host for a lot of functionality. Tools like Docker make use of separate namespaces and control groups to ensure isolation of processes. However, because all the containers running on the

same edge host use the same kernel, the degree of isolation is not as high as that available with VMs [18]. Nevertheless, containers offer the following advantages that make them ideal for edge use cases:

1. small image size
2. very small memory footprint
3. and fast instantiation times

Consequently, they have mostly replaced VMs as the technology for packaging and hosting applications running on the edge. Randazzo et al mention that Docker is one of the most widely deployed container platforms [19].

It is also worth mentioning that there is a rise in a new breed of VMs based on unikernels, which run minimal operating systems known as library operating systems. IncludeOS and HermiTux are examples of such operating systems [20, 21]. These VMs are only slightly larger than containers, quick to boot, and are compatible with common hypervisors, thus offering all the advantages of traditional VMs.

On a similar note, there is another emerging technology called kata-containers. Kata-containers are VM-based containers that are expected to be as fast and flexible as containers, but as secure and isolated as traditional VMs. This security-oriented technology is based on Intel Clear containers and is the successor of a now obsolete hypervisor-based runtime called hyper.sh runV [19].

Because trends in the literature suggest traditional virtualization approaches are turning obsolete and emerging lightweight VM approaches are still experimental and not widely used, in this thesis, I focus primarily on container-based migration strategies.

3.2 Earlier Testbeds

From the literature, it is very clear that creating a testbed is the ideal approach to follow for measuring the performance of a migration strategy.

Most researchers keep their testbeds simple by having exactly two computers in their setup: one serving as the source MEC host, and the other the target MEC host. Most used physical workstations as the MEC hosts, but some experimented with VMs too.

Most approaches did not involve transferring low-level details such as the contents of the source container's memory pages and CPU registers. Some did, and chose to call their migrations "live migrations".

Table 3.1 gives an overview of the testbeds encountered during the literature review phase of this thesis.

	Technology	Live?	Strategy
Farris et al (2017)	Docker	No	Replicate services and transfer only application state
Addad et al (2018)	LXC and CRIU	Yes	Live and iterative migration
Campolo et al (2019)	Docker	No	Pre-relocate filesystem and transfer only state later
Bellavista et al (2019)	Docker Compose	No	Proactive application-aware handoff extension to the basic Docker migration protocol

Table 3.1: Overview of earlier testbeds

3.2.1 Service Replicated on Multiple Edge Hosts

In March 2017, Farris et al [22] described an early proactive strategy for service migration, which involved maintaining replicas of the service on multiple MEC hosts to minimize downtime. Their testbed consisted of two physical workstations acting as the MEC hosts. Both workstations had the Docker engine installed on them.

The metrics they considered were total migration time and initialization

time. However, because there was no actual migration of the container filesystem, the downtimes they observed were a function of the Docker volume (DV) size they used, and were on average less than 2 seconds. Indeed, in their tests, they used the same Docker container image for all scenarios and changed only the Docker volume size.

Although this approach results in very short service interruptions, it does not use the resources available on the MEC host efficiently. This is because the service needs to be instantiated on several MEC hosts even though the UE will connect to only one of them at any given instant. Farris et al have suggested optimizations such as replicating the service only on MEC hosts that lie in the direction of movement of the UE.

3.2.2 Support for Live Migration

Most testbeds make a distinction between transferring the actual filesystem of the container and the user state. Furthermore, they either assume the service itself is stateless, or do not place any emphasis on the setup of its initial default state. This often results in slow boot up times for the service. For services that have complex initial states, it might be better to not build them from scratch during container boot up. Examples of such services could be Massively Multiple Online Role-Playing Games (MMORPGs), where start up times can be long, and the game-world state is just as important as the user state.

A live migration-based approach can overcome the above problems. In this type of migration, the service does not start with a fresh new state on the target MEC host. Instead, it simply resumes from the state it was in on the source MEC host. This can be accomplished by various methods.

Earlier methods involved maintaining a log of all events generated by the container on the source MEC host and replaying the log on the container of the target MEC host. They were simply a container-oriented adaptation of the system trace and replay approach commonly used in live migration of VMs [23]. This was quite error prone, especially when there are lots of asynchronous events present. Therefore, more recent methods favor directly

copying both the filesystem contents and memory pages of the container on the source MEC host to the target MEC host [24].

For instance, in 2018, Addad et al [17] created an experimental testbed to evaluate the performance of a live migration strategy in the context of 5G. They were trying to minimize both downtime and total migration time.

Their testbed consisted of virtualized nodes running Ubuntu 16.04, and used Linux Containers (LXC) and the Checkpoint/Restore In Userspace (CRIU) project for creating application containers and managing their memory pages. They ran their experiments with two containers: a blank Linux container and a larger Linux container that had a video streaming server installed in it.

They compared both stateful and stateless migration scenarios. Additionally, their strategies could be classified as those that were ideal for scenarios where the VUE path was pre-determined, and those ideal for unknown-path scenarios.

They managed to achieve downtimes of approximately 1050 ms for the blank container and 1300 ms for the video streaming container.

Transferring both the filesystem contents and memory pages can be time consuming, especially if the MEC host has a large RAM. Therefore, these researchers followed an iterative approach to transferring the memory. This resulted in larger bandwidth consumption, but kept the service downtime low.

3.2.3 A Two-Phase Approach

Campolo et al in 2019 created a relatively simple testbed in order to evaluate a custom migration strategy based on Docker containers [25]. They too focused primarily on service downtime, but included other metrics. Additionally, they tested the performance of their strategy under multiple network conditions and their strategy was specifically designed for V2X services.

Their testbed included two physical workstations that had the Docker Engine installed on them. During their tests, they used two containers, the main difference between them being their filesystem sizes. They also used multiple Docker volumes, the main difference between them being their file sizes.

Their migration strategy had two phases and relied extensively on the use of Docker volumes. In the first phase, called the service pre-relocation phase, they migrated only the filesystem of the source container. In the second phase, called the service relocation phase, they migrated only the additional state of the source container, which was now expected to be in a small Docker volume. Because there was no downtime during the service pre-relocation phase, transferring the Docker volume and booting up the container accounted for most of the downtime they observed.

This strategy allowed them to reduce service downtimes to approximately 2 seconds, so long as the Docker Volume was 10 KB or smaller. Furthermore, the downtime was largely independent of the the actual size of the container.

This strategy is feasible only if the application running inside the container is custom-built to support it. This is because the application needs to be aware of the migration, and act differently depending on the currently active phase. More precisely, when there is no migration happening, the application must store most of its state on the container's filesystem. But during the pre-relocation phase, it must stop writing to the container's local filesystem and instead store all of its state in a Docker volume.

In this thesis, Campolo et al's approach was used as a baseline reference implementation, so it is described in further detail in a later chapter.

3.2.4 An Application-Aware Strategy

All of the testbeds considered up to this point were application-agnostic. That is, they work the same regardless of the service application being migrated. These testbeds focused only the size of the container's filesystem, the user state, and the container memory pages.

Bellavista et al [26] suggest a proactive handoff strategy very similar to that of Campolo et al, with an extension being that their strategy was application aware. Consequently, instead of transferring the service as one monolithic container, their approach suggests splitting it up into multiple containers and transferring each of them individually.

Their testbed had three Linux computers, and was heterogeneous. It was composed of two workstations and a Raspberry Pi3. Docker alone was not ideal for their use case, so they used Docker Compose. This helped them simplify the code needed to instantiate multiple containers and setup their Docker volumes in an error-free manner.

The service they migrated was a Java web application that used MongoDB as its database. Thus, their application had two distinct layers: a service layer and a data layer, each of which could be migrated separately.

One of the obvious disadvantages of this approach is that it is feasible only if the service involved has a modular architecture and is easy to split into distinct layers. The distinction between the service layer and data layer, for example, is not always clear. Furthermore, in some cases, such as when closed-source applications are considered, such a split might not be possible at all.

Chapter 4

A Baseline Testbed Implementation

The approach suggested by Campolo et al was chosen as a baseline implementation for this project, primarily because it was aimed specifically at V2X services. Furthermore, it was very accessible because it relied only on open-source software and commodity hardware, making it easier for me to compare my results with theirs.

4.1 Hardware and Software Specifications

The hardware and software used in the testbed are expected to have a large impact on the service downtimes and most other metrics observed. Therefore, their specifications are important if this testbed is to be replicated and obtain similar results.

Two Intel Next Unit of Computing (NUC) small form factor workstations were used as the two MEC hosts. Each had 16 GB of RAM, an Intel Core i7 processor, and a solid state drive. The workstations could communicate with each other over Ethernet.

Both the workstations ran Ubuntu 20.04 LTS as the operating system and had Docker Engine 20.10.4 installed on them. Furthermore, to support

live migration, CRIU 3.15 was manually compiled from its source code and installed on both the workstations.

4.2 Phases

This implementation migrated the MEC application in two phases, and the MEC orchestrator was responsible for triggering each phase at an appropriate time. The sequence diagram in Figure 4.1 gives an overview of the sequence of events that occur in a successful migration.

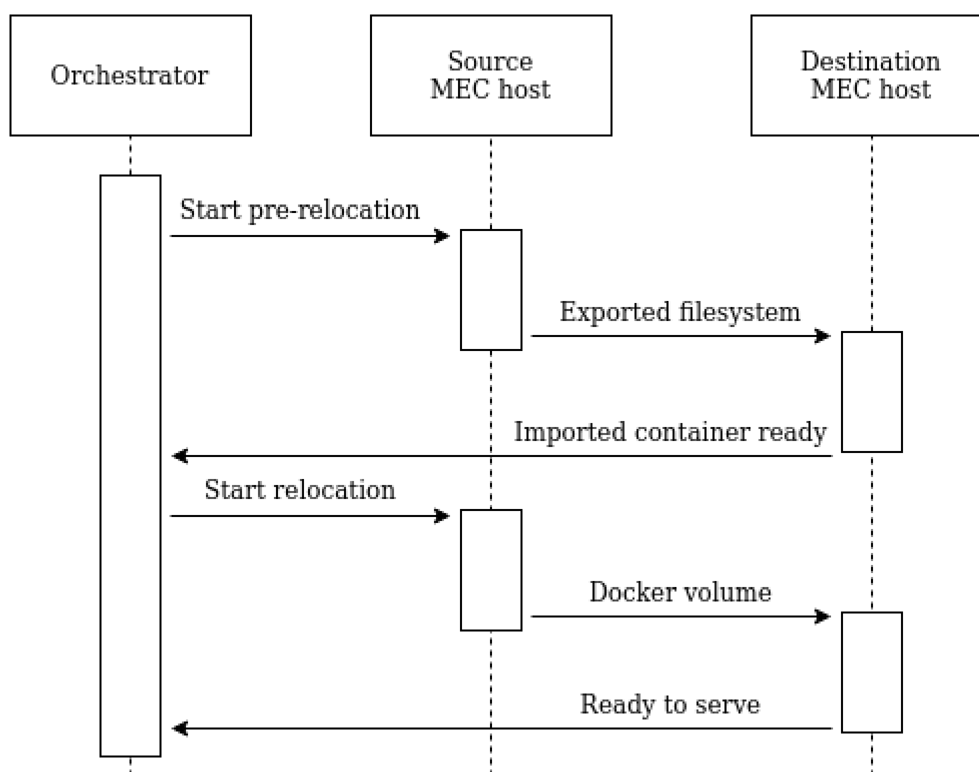


Figure 4.1: Overview of baseline migration strategy

4.2.1 Service Pre-relocation

When a migration is necessary, the service pre-relocation phase is triggered. In a real world MEC system, according to ETSI GR MEC 018 [27], this trigger could from any of the following entities:

1. MEC applications

2. UE clients of MEC applications
3. Source/target MEC platforms using the associated RNIS
4. Source/target MEC platforms using the associated data planes
5. the MEC orchestrator

For this thesis, the orchestrator was chosen as the entity generating the triggers. Like Campolo et al, as a part of the baseline testbed implementation, I used shell commands and scripts to create a manual orchestrator and time all the phases. This meant running the appropriate Docker Engine commands and timing them.

Once the trigger to start the pre-relocation is received, the layered filesystem of the Docker container on the source MEC host is flattened and exported into a tarball using the Docker `export` command. This tarball is then securely copied to the target MEC host using the `scp` tool and used to create a new container image using the Docker `import` command.

4.2.2 Service Relocation

Throughout the service pre-relocation phase, the source MEC host continues to serve the UE without any interruptions. In the service relocation phase, however, the container on the source MEC host is shut down. From this point on, the UE starts experiencing service downtime and the rest of the steps have to be completed as quickly as possible.

After the container is shut down, the DV it was using is copied to the target MEC host. The contents of the DV were assumed to be merely Floating Car Data (FCD) packets in Campolo et al's testbed, so the remote copy mechanism used was a combination of the Linux `dd` and `nc` commands. The same commands were used in my implementation too. Because there is no encryption overhead involved and no time is spent on the SSH handshake, this copy operation can be very quick.

Once the DV is available on the target MEC host, the container image

created in the previous phase is instantiated and booted up so the UE can connect to it, at which point the service downtime ends. The DV is mounted as the container is booted up.

4.2.3 Role of the V2X Application

In order to ensure the integrity of the UE's data available on the target MEC host after the migration, the V2X service running on the container is expected to be aware of the need for a migration and implement the following simple logic:

1. Is a migration in progress?
2. If yes then,
    (a) Stop writing to the container's filesystem
    (b) Start writing to Docker volume mounted
3. If no then,
    (a) Write to the container's filesystem normally

This algorithm ensures that there are no changes in the filesystem after the tarball is created during the pre-relocation phase. It also ensures that all the new information the UE generates during the service pre-relocation phase is available on the DV.

The above logic was implemented in the applications by exposing an endpoint the orchestrator could use to specify the current phase of migration. The applications are described in more detail in a later chapter.

Chapter 5

A More Stateful Testbed

One of the biggest pitfalls of the baseline testbed implementation is that it copies only the container's filesystem and UE-related state from the source MEC host to the target MEC host. It does not preserve any application configuration or application-related state that is not present on the filesystem.

Furthermore, the `Dockerfile` used to build the container at the source MEC host is not available at the target MEC host. Because the container at the target MEC host is built solely using the imported filesystem, it will not be aware of common and crucial initialization instructions such as `CMD` or `ENV`.

Several Dockerized Linux applications use environment variables in order to store configuration settings [28]. These could include critically important details, such as the value of the `PATH` variable, which specifies the locations where the Linux OS looks for executable files, or the `PWD` variable, which specifies the current working directory. Without access to these details, the MEC application, when it starts on the target MEC host, is unlikely to behave the same way it did on the source MEC host.

To overcome these issues, the baseline testbed implementation was extended to make it more stateful.

5.1 Approach

Fortunately, by making only minor changes to the baseline testbed implementation, all the environment variables, all the Docker instructions, and several application settings can be preserved. Figure 5.1 gives an overview of this approach.

The Docker `save` and `load` commands are better at preserving state. Therefore, the Docker `export` command at the source MEC host is replaced with the Docker `save` command. Similarly, at the target MEC host, the Docker `import` command is replaced with the Docker `load` command. This is an approach Bellavista et al mention in their article [26].

Unlike the Docker `export` command, however, the Docker `save` command works only with container images. This means that a running or pre-instantiated container cannot be saved directly. To overcome this limitation, the Docker `commit` command is run before the `save` command. This generates a new container image identical to the currently running container.

The tarball generated by the Docker `save` command is, as one might expect, slightly larger. This is because it contains not only additional state details, but also details about all the necessary parent layers, such as their tag names and versions. In order to reduce the tarball size, it can be compressed using the `gzip` tool. There is no need for a corresponding explicit decompression step at the target MEC host because the Docker `load` command can handle compressed archives. However, because the compression and decompression operations themselves were found to be time-consuming, they were not included in this implementation.

It is worth mentioning that the Docker `commit` command, by default, temporarily freezes the container while it creates a container image from it. This is necessary to stop changes in the container state during the commit operation, which could potentially lead to data corruption. But a consequence of this is that there are potentially two service downtimes during the migration: one during the commit operation and the other during the actual handover.

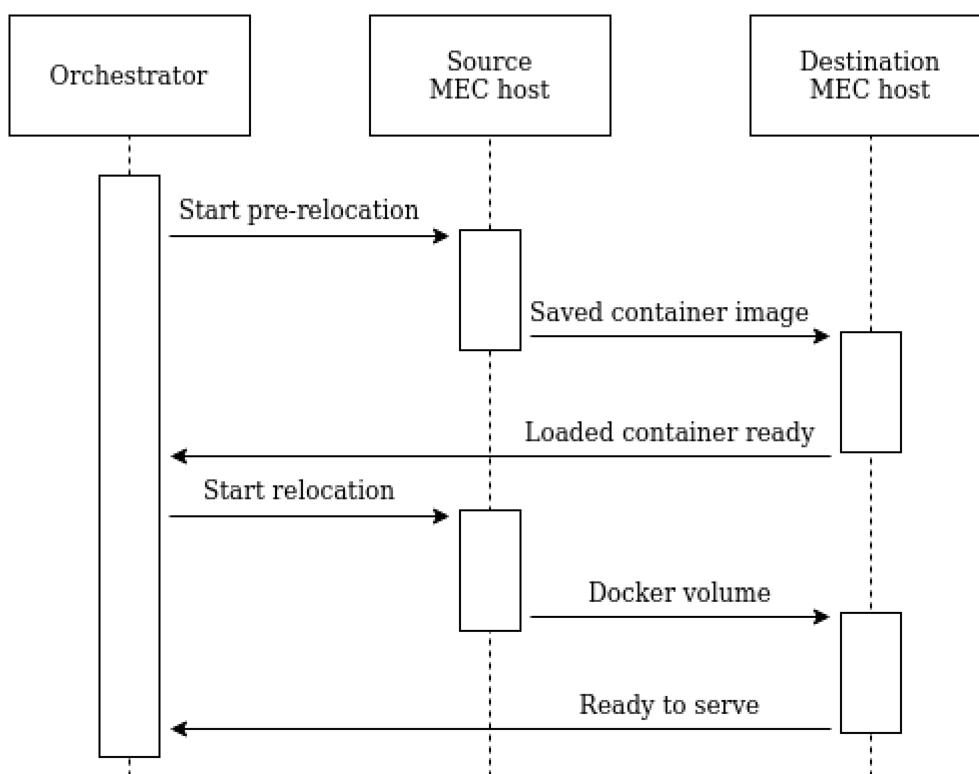


Figure 5.1: Overview of more stateful migration strategy

Chapter 6

Testbed With Support for Live Migration

Although the previous approach is capable of preserving much of the container state, it is still the responsibility of the MEC V2X application(or its developers) to maintain the list of all environment variables and settings it needs. This is necessary because items in the list are to be passed individually to the Docker `commit` command as input parameters. As a result, the previous approach too is limited only to open source applications or closed source ones that are willing to share the list.

For a migration strategy to support all applications, even those that were not built to run in an MEC scenario, it should not depend on any inputs from the applications. So, the extended testbed from the previous chapter was further extended to support live migration.

In this extension, the contents of all the memory pages, CPU registers, and other resources used by the container on the source MEC host are additionally copied to the target MEC host during the migration. Such a migration is referred to as a live migration in the literature [29]. It is important to note that the migration can now be fully transparent to the MEC application. That is, after a successful live migration, the MEC application would generally not even notice that it was migrated.

6.1 The CRIU Project

CRIU is a tool that can be used to create a detailed copy of a process that is running inside a container. It can record important details such as the contents of the relevant memory pages, contents of CPU registers, the sockets currently being used, files currently open for I/O operations, and mountpoint-related information [30]. It does so using `ptrace`, a system call meant for creating a process trace.

CRIU needs to be controlled by the Docker Engine. Because this is currently an experimental feature, it is available only after Docker Engine is manually configured to enable it. In this extension of the testbed, CRIU was installed and enabled on both the source and target MEC hosts.

6.2 Approach

The approach followed is very similar to the approaches discussed earlier, but there are a few fundamental differences. The first difference is that during both the pre-relocation and relocation phases, the Docker `checkpoint` command is run on the source MEC host to freeze the V2X application's container and save its state. By default, this operation immediately stops the container. The `leave-running` flag is set during the pre-relocation phase to keep the container alive afterwards. Doing so is not necessary during the relocation phase, however, because the container is not expected to be alive on the source MEC host any more.

The output of the checkpoint operation is a directory containing several CRIU image files. These files are copied to the target MEC host so that they can be used to restore the checkpoint-ed V2X application there. But doing so is possible only if a valid container is already present and active on the host.

Therefore, during the service pre-relocation phase, the container also needs to be built on the target MEC host. For common applications, the easiest way to achieve this is to use Docker Hub or any other container registry

available in the MEC system. For custom applications, however, the Docker `commit`, `save`, and `load` commands should be used to set up the container, as described in the previous approach. Because a custom V2X application was used in this project, the latter approach was followed.

The application is checkpoint-ed twice in order to leverage the `rsync` tool and minimize the service downtime. This way, the entirety of the memory pages are copied during the pre-location phase, and only the changed bits are copied during the actual relocation. This is important because the memory pages can often be as large as the container's filesystem.

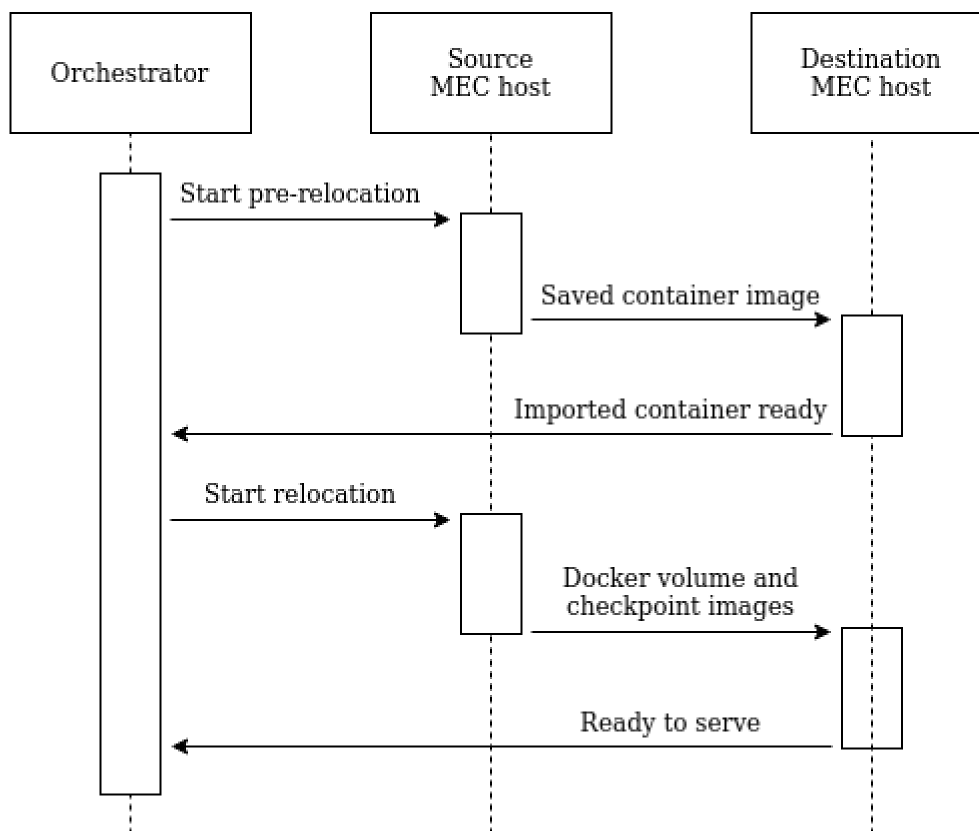


Figure 6.1: Overview of live migration strategy

	Strategy 1	Strategy 2	Strategy 3
Preserves filesystem contents	Yes	Yes	Yes
Preserves container configuration	No	Yes	Yes
Preserves memory pages, CPU register contents	No	No	Yes
Application needs to be aware of migration?	Yes	Yes	No
Is Live?	No	No	Yes
Tools used	Docker Engine	Docker Engine	Docker Engine + CRIU

Table 6.1: Overview of the three migration strategies

Chapter 7

Experimental Setup

A common set of experiments were run on the baseline implementation of the testbed and both the extended implementations in order to gather metrics, which could then be used for comparison and analysis. Figure 7.1 gives an overview of the components and layout of the experimental setup at the start of the migration.

7.1 The ETSI MEC Sandbox

All the experiments run during this project needed an MEC platform offering necessary MEC services. Because setting up a real MEC platform was not feasible, the ETSI MEC Sandbox environment, which is based on the AdvantEDGE mobile edge emulation platform [31], was chosen for emulating one.

7.1.1 The APIs

As Figure 7.2 shows, the ETSI MEC Sandbox is an interactive environment targeted at developers. It offers several commonly used MEC service APIs with OpenAPI-compliant descriptions. These REST-based APIs accept inputs and generate responses in the form of both JSON and YAML documents. In this project, only the JSON format was used.

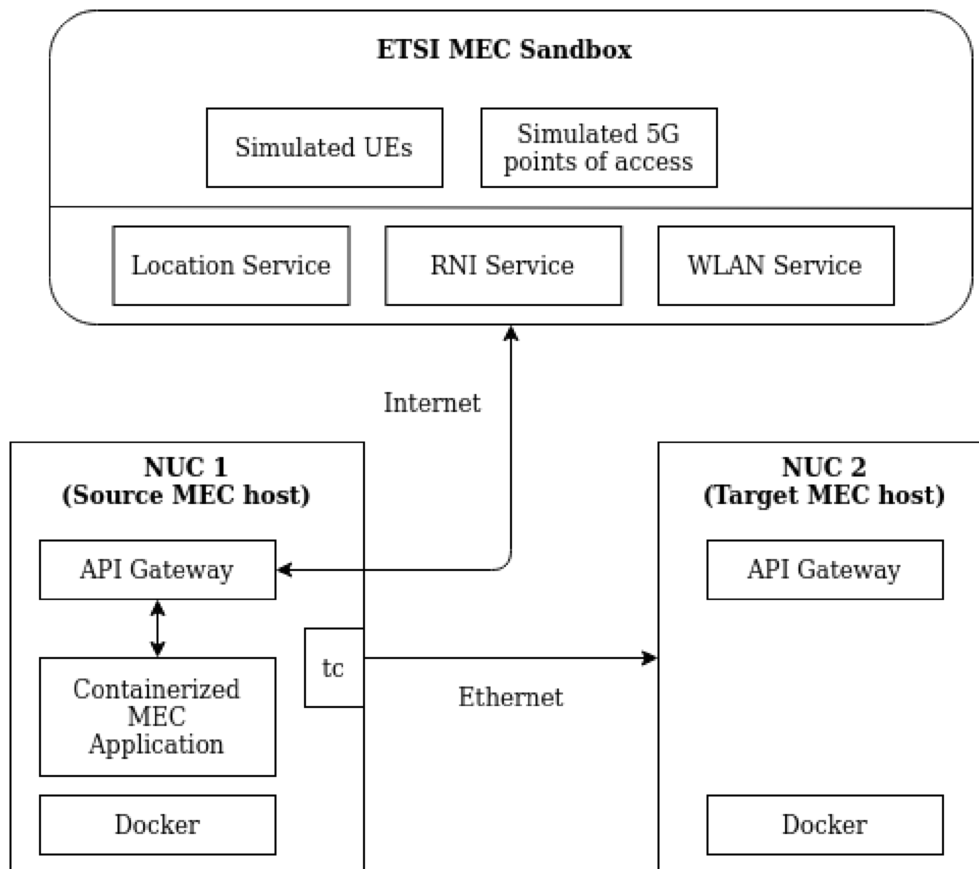


Figure 7.1: Overview of the experimental setup at the start of the migration

As mentioned earlier, the Location Service and Radio Network Information Service APIs are among the most important APIs any MEC application would need. This sandbox, at the time of this thesis, supported both APIs but implemented only a limited subset of the endpoints mentioned by the ETSI ISG[9, 32]. For example, the Location Service API implementation did not support most of the distance and area related subscription endpoints. Similarly, the RNIS API did not support endpoints that could fetch S1-U bearer information or layer 2 measurements information.

Consequently, the V2X MEC applications built were designed to work around the limitations.

7.1.2 The Scenario

The ETSI MEC sandbox, at the time of this thesis, offered one scenario with three different network configurations. The scenario was set to emulate the urban environment in the city of Monaco, with a configurable number of stationary, fast, and slow moving UEs.

The three different network configurations differed primarily in the network technologies they supported. The simplest, named `4g-macro` supported only 4G points of access. The next, named `4g-wifi-macro`, supported both 4G and WiFi points of access. In this project, however, only the third configuration, called `4g-5g-wifi-macro` was used. This was the most flexible configuration, and supported all the network technologies that are likely to be available in a real-world implementation of the scenario.

In the `4g-5g-wifi-macro`, nineteen 5G small cell points of access, eleven WiFi points of access, and ten 4G macro cell points of access were available. The fast moving UEs, which represented cars, had a velocity of $20ms^{-1}$ and the slow moving ones, which represented cyclists or pedestrians, had a velocity of $9ms^{-1}$.

It is important to note that the sandbox does not specify the location of the MEC hosts. It gives only the locations of the radio points of access and the zones they belong to. Wang et al mention that it is very common to place MEC hosts in close proximity to the mobile base stations [33].

Experiments were first run assuming that an MEC host was present near each of the nineteen 5G points of access in the sandbox environment. For the sake of simplicity, it was also assumed that each MEC host served exactly one 5G small cell base station. Then, experiments were run assuming that MEC host were associated only with zones instead of the individual points of access they contained.

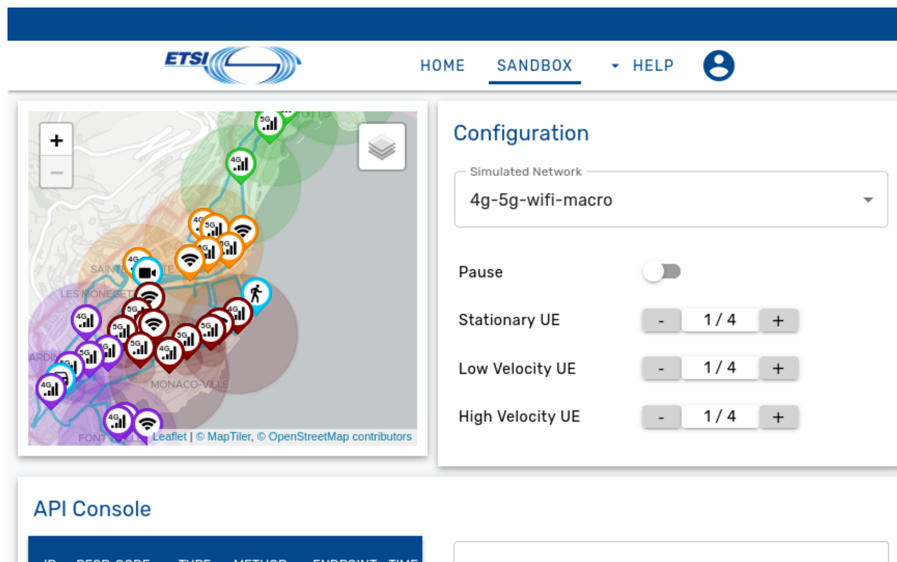


Figure 7.2: Screenshot of the sandbox UI

7.2 The API Gateway

As mentioned earlier, MEC applications are usually expected to connect to the MEC services they need through an API gateway. Therefore, an API gateway was created using Apache HTTP Server 2.4. Using its `mod_proxy` module, a reverse proxy was setup so that any requests to it were routed to the ETSI MEC sandbox. The custom configuration file is included in the appendix of this thesis.

The gateway was containerized and was set up on both the source and target MEC hosts. It had the same Uniform Resource Locator (URL) on both and the MEC applications could interact with it by using a Docker bridge network.

The `httpd:2.4` image available on Docker Hub was used as the base image for the gateway.

7.3 Applications

Most of the research encountered during the literature review phase dealt with the migration of blank container images or containers having generic applications that were not aware of the MEC context. Consequently, their

results often did not account for application startup time or the actual feasibility of the migration in the real world.

An important contribution of this thesis is that I report the migration of actual MEC V2X applications that are capable of interacting with MEC services. The utility of these applications when used independently can be considered limited, but it is very likely that they could serve as useful modules inside larger, real-world V2X applications.

These applications relied on the API gateway for interacting with the MEC services. This was necessary in order to ensure that the URL they used to access a service did not change as they were migrated from one MEC host to another.

It's also worth mentioning that the URL used to access the sandbox itself changed whenever it was restarted. Thanks to the API gateway, the applications did not have to be rebuilt or re-configured every time the sandbox was restarted. Indeed, only the gateway's configuration had to be changed.

Three containers, each containing a different V2X application, were used in my experiments.

7.3.1 Basic Application

The simplest of the applications merely functioned as an RNI service consumer. More precisely, it followed the request-response model to query RAB information every 200 milliseconds using the RNIS API. Migrating this application served as an initial test for the viability of the testbed implementation.

This application is considered simple because its functionality does not depend on any state information, be it user state or application state. Consequently, it does not have to depend on the contents of a Docker volume or any other form of persistent storage.

The basic application was built using NodeJS 15.14.0 and `got` 11.8.2, which is an HTTP request library.

7.3.2 User State-Preserving Application

As its name suggests, the user state-preserving application needs the user-specific state preserved to function correctly. To satisfy a realistic use case, this application was designed to perform a single, atomic change to the user state. This change involved incrementing a counter that was a part of the user state.

During this experiment, the value of the counter is of critical importance. For a migration to be considered successful, the counter should not start from zero at the target MEC host. Instead it should resume from the value it was at at the source MEC host. This emphasizes the point that although the application itself is stateless, it is critical that the user-specific state be preserved.

Implementation-wise, the counter counted zones. Its value was incremented whenever the UE's current zone changed. This was done using information offered by the Location Service API.

This application too was built using NodeJS and `got`. It was designed to be aware of the migration and used the DV to store the counter value and run-time logs once the pre-relocation phase started. It was also capable of retrieving the counter value from the DV, if available during startup.

It's also worth emphasizing that, much like the basic application, this application too did not need to preserve any application instance-specific state to function correctly. It needed only the user-specific state.

7.3.3 Application With a Stateful Workload

Key-value stores are indispensable in most web applications today. They are often used as a cache to store the results of expensive or time-consuming computations [34]. As such, they are large hash tables, with unique keys pointing to important values.

	Application 1	Application 2	Application 3
State preservation requirements	No state necessary	User-specific state necessary	User-specific and application-specific state necessary
What it does	Query RAB information every 100ms using the RNIS API and log it	Keep a count of the number of zones a VUE has travelled through using the Location Service API	Use Memcached to store the list of all useable access points in the city
Uses Docker Volume?	No	Yes	No
Is aware of migration?	No	Yes	No

Table 7.1: Overview of the three MEC applications migrated during the experiments

Memcached has been used as a high-speed, in-memory, key-value datastore in edge computing scenarios [35]. Therefore, it was chosen as a candidate for testing the live migration strategy. Accordingly, Memcached 1.6.9 was used as the application with an in-memory stateful workload. Because Memcached offers an easy to use command-line interface over Telnet [36], the Linux `telnet` utility was used to retrieve access point-related information on it. This information was obtained from the Location Service API using cURL, which is a Linux command-line utility.

All the three applications were run inside their own separate containers. The basic and user state-preserving applications' container images were built with the `Node 16-alpine3.11` image as the base. The stateful application's container image was built with the `memcached 1.6.9` image as the base. All of the base images were pulled from Docker Hub. The complete source code for the applications can be found in the appendix.

7.4 Variable Network Conditions

Containers running the three applications described above were migrated from the source MEC host to the target MEC host under two different network conditions: normal and congested.

Under normal conditions, the network had a bandwidth of approximately 1 Gbps. To simulate network congestion, traffic shaping was done using the Linux `tc` utility. To configure `tc` appropriately, a classful queueing discipline of type Hierarchical Token Bucket (HTB) was set up. A class was then added with its `rate` set to 100 Mbps. Thus, whenever necessary, the bandwidth could be easily reduced to 100 Mbps.

The complete code for the Bash script simulating network congestion can be found in the appendix.

Chapter 8

Results

For easy readability, let us refer to the basic application container as `A1`, the user state-preserving application container as `A2`, and the stateful application container as `A3`. Similarly, let us call the baseline migration strategy `M1`, the more stateful migration strategy `M2`, and the fully stateful, live migration strategy `M3`. Lastly, let us refer to the normal network condition as `N1`, and the congested network condition as `N2`.

Each final result obtained is based on a combination of the application, the migration strategy, and the network condition. In other words, each final result is a function of the three. For example, `(A1,M1,N1)` is the result obtained for the basic application migrated using the baseline migration strategy under normal network conditions and `(A3,M3,N2)` is the result obtained for the stateful application, migrated using the live migration strategy under the congested network condition.

I also mention several intermediate results that are independent of the network condition. These are identified using only two parameters. For example, `(A1,M3)` is the result obtained for some specific stage in the migration of the basic application using the live migration strategy.

8.1 Container Sizes

There was a significant difference between the sizes of the container images and the tarballs generated from the container instances. This was expected because the Docker `export`, `save`, and `checkpoint` commands function differently, and preserve varying amounts of run-time application state, as discussed earlier.

As can be seen in figure 8.1, `A1` had the smallest container image while `A3` the largest. These sizes were obtained using the Docker `inspect` command.

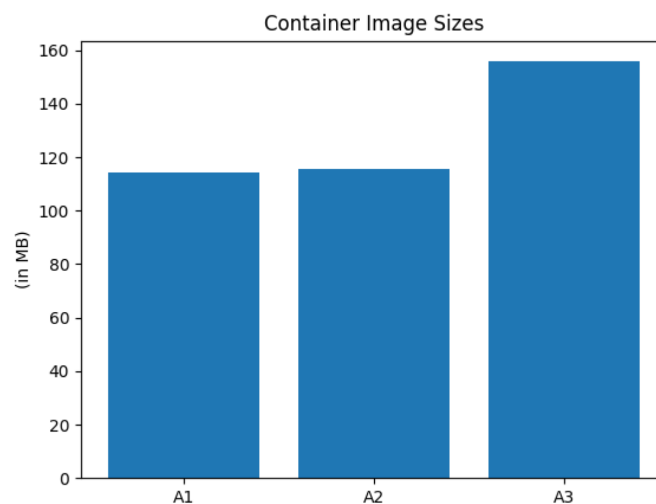


Figure 8.1: Container image sizes

Figure 8.2 shows how the tarball sizes vary by migration strategy. For migration strategies `M1` and `M2`, there were only minor variations in the tarball sizes for applications `A1` and `A2`. This is because there was either no state preserved or very little state preserved, respectively. In the case of the `M3` migration strategy, however, there was a noticeable increase in tarball size for all applications.

Similarly, strategy `M1` generated the smallest tarballs for all applications. And strategy `M3` generated the largest tarballs for all applications.

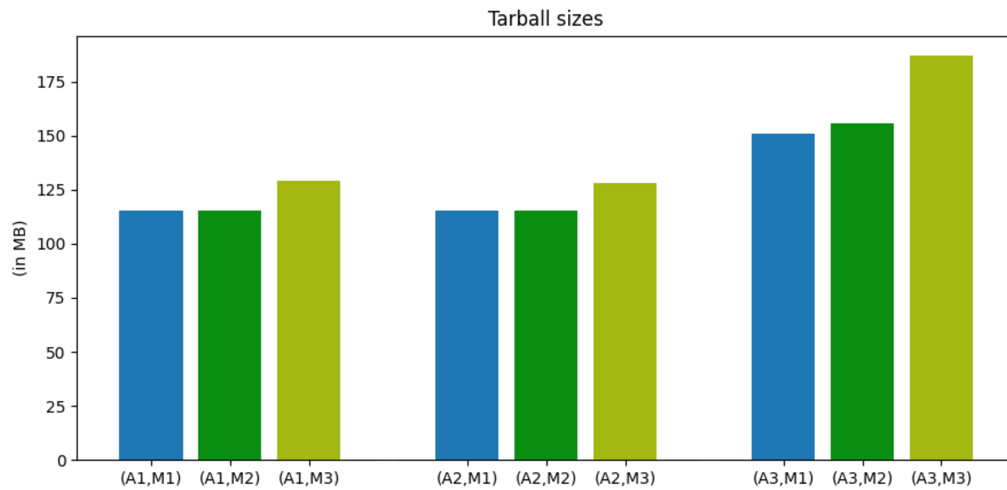


Figure 8.2: Tarball sizes

8.2 Duration of Pre-relocation Phase

The sizes mentioned above are expected to not have any significant impact on the service downtime. This is because they are generated and transferred during the service pre-relocation phase. Of course, this holds true only if the service pre-relocation starts at the right time. Therefore, it is important for the orchestrator to know the overall duration of the pre-relocation phase. This duration is the sum of the durations mentioned in this section.

The Linux `time` utility was used to time all the operations discussed below. Because the exact durations varied by a few milliseconds every time the commands were run, the average of five runs was calculated and used as the final result.

8.2.1 Time Spent on the Source MEC Host

In the case of strategy `M1`, this is only the time taken by the Docker `export` command. For migration strategy `M2`, this is the sum of the times taken by the Docker `commit` and `save` commands. And for strategy `M3`, this is the sum of the times taken by the Docker `commit`, `save`, `checkpoint` commands. Figure 8.3 shows the times measured.

In all scenarios, strategy `M1` is the fastest. This is expected because only one Docker command needs to be run and only the filesystem contents are preserved. And with three different Docker commands and full state preservation, strategy `M3` is the slowest.

In strategies `M2` and `M3`, the Docker `commit` operation, as mentioned earlier, temporarily froze all the containers. As shown in Figure 8.4, although the freeze durations were very short and the MEC applications being tested were not affected at all, they were recorded for future optimization purposes. This is because they could potentially degrade the end user experience for some MEC applications.

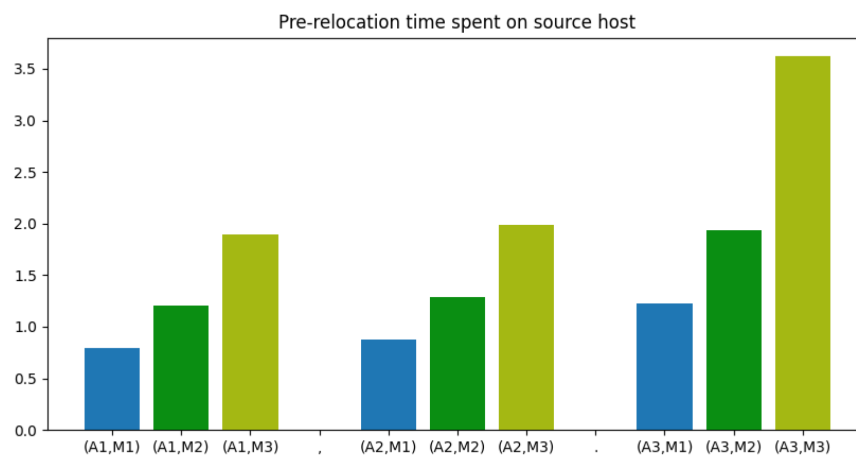


Figure 8.3: Pre-relocation time spent on source host (in seconds)

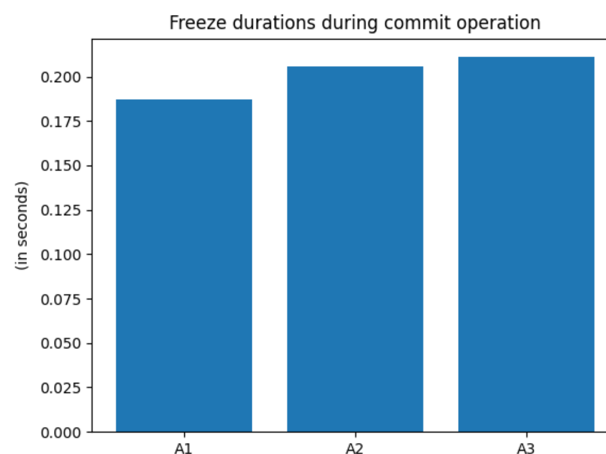


Figure 8.4: Freeze durations during commit operation

8.2.2 Time Needed to Transfer the Tarball

This is the time taken by the `scp` command to transfer the tarball generated on the source MEC host to the target MEC host. Because this is a function of the network condition, two different results were obtained, as can be see in figure 8.5.

Again, because strategies `M1` and `M2` generate smaller tarballs, they takes the least amount of time to complete. Strategy `M3`, on the other hand, takes slightly longer, especially when the network is experiencing congestion.

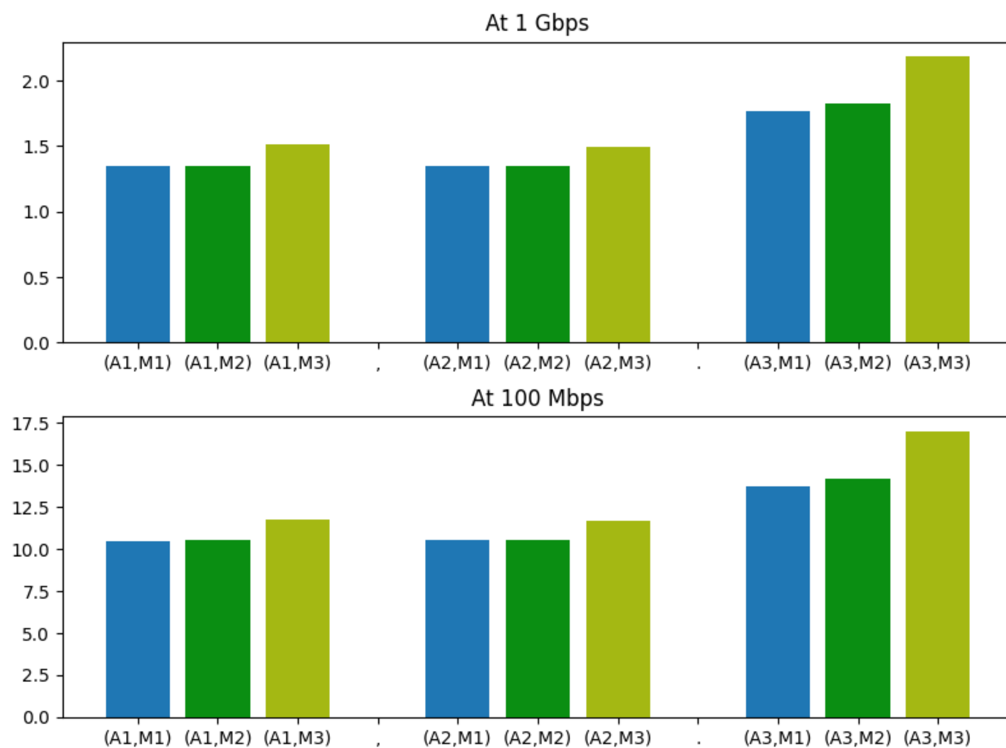


Figure 8.5: Pre-relocation time spent copying the tarball (in seconds)

8.2.3 Time Spent on the Target MEC Host

In the case of strategy `M1`, this is the time taken by the Docker `import` command. For strategies `M2` and `M3`, this is the time taken by the Docker `load` command. As can be seen in figure 8.6, the times are identical for both strategies `M2` and `M3` because the checkpoint data is not used yet, and thus does not contribute to the duration.

Strategy `M1` is again the fastest because it does it not preserve any Docker layers information. Strategies `M2` and `M3` were much slower because they had preserved information about six Docker layers, each of which had to be restored individually.

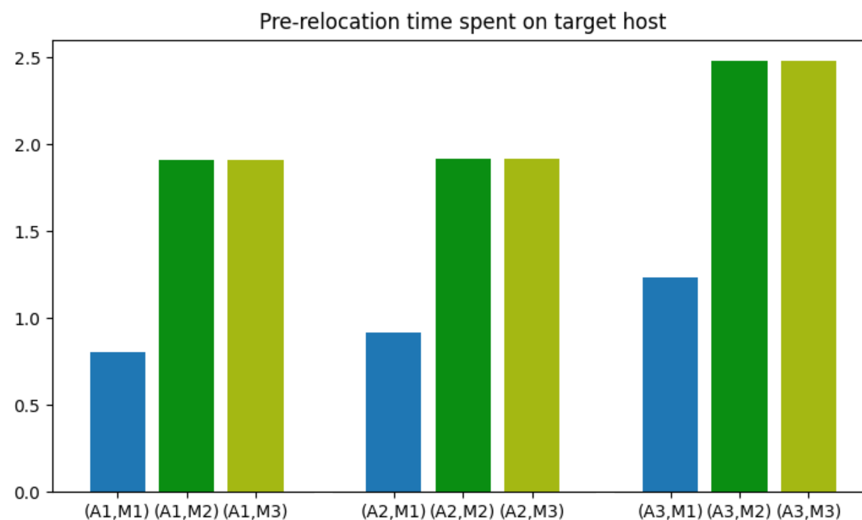


Figure 8.6: Pre-relocation time spent on the target host (in seconds)

8.3 Duration of Relocation Phase

Throughout the pre-relocation phase, in all three implementations of the testbed, the UE had largely uninterrupted access to the MEC application. In fact, it was unaware of the phase all together. The relocation phase, however, does introduce service downtimes, which were recorded and measured.

8.3.1 Time Spent on the Source MEC Host

In all the three strategies, the following operations needed to be run on the source MEC host:

- Stop the container using the Docker `stop` command
- Create a tarball of the DV used by the container using the Linux `tar` utility.

In strategy `M3`, however, the Docker `checkpoint` operation had to be run additionally to create a new checkpoint. This had to be done before stopping the container.

For applications `A1` and `A3`, the DVs were always empty. In case of application `A1`, the DV was empty because it was completely stateless. It needed to store neither user nor application instance state. In the case of application `A3`, on the other hand, the DV was empty because all the state was being stored in the memory of the container.

The DV of application `A2` was populated with data, and the amount of data it contained was directly proportional to the duration of the pre-relocation phase. This meant that network congestion increased the size of the DV. This effect, although mentioned, was not fully accounted for in Campolo et al's testbed [25]. Therefore the final results generated by my implementations of the testbed and its extensions can be expected to be more realistic.

Figure 8.7 shows the size of the data volume for application `A2` in the absence and presence of network congestion. As can be seen, the size increased significantly in the presence of network congestion.

The `tar` command is very quick, and the difference between the times to archive the two data volumes is, on average, less than 0.001 seconds. Therefore, it was assumed to be a constant while calculating the total time spent on the source host.

Figure 8.8 shows the total time spent at the source MEC host during the relocation phase. As can be seen, strategies `M1` and `M2` were the fastest. Strategy `M3` however was consistently slower because of the additional checkpoint operation, and all applications took markedly longer times. It is also worth noting how fast the strategies `M1` and `M2` are for application `A3`, which doesn't store any state in the file system. This only implies that they ignored most of the state information `A3` needs in order to run correctly on the target host.

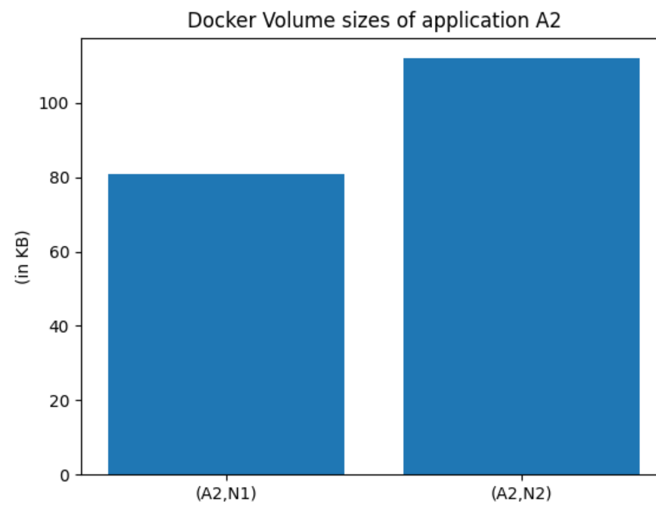


Figure 8.7: Size of Docker Volume for application A2 given two network conditions

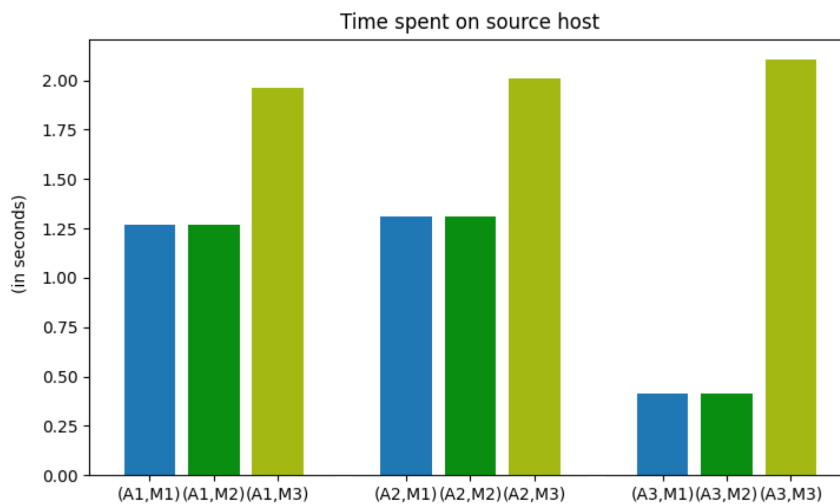


Figure 8.8: Time spent on source host during relocation

8.3.2 Time Needed to Transfer Data

The `nc` tool was used to copy the DV's tarball from the source MEC host to the target MEC host. This was necessary only in the case of application `A2` because it was the only application that used the DV to store additional state while the pre-relocation was in progress.

In strategy `M3`, the `rsync` utility was additionally used to transfer the

checkpoint files generated by the the CRIU tool. This transfer was configured to use SSH in order to maintain data confidentiality. All the three applications generated at least 13 MB of checkpoint data, with `A3` generating, on average, 16 MB. With the `rsync` command, it was noted that, on average, only 2.9 MB of updated data had to be transferred, resulting in an average speedup of 4.52x, as could be seen from the command's logs. In other words, compared to `scp`, the `rsync` command was about 4.52 times faster.

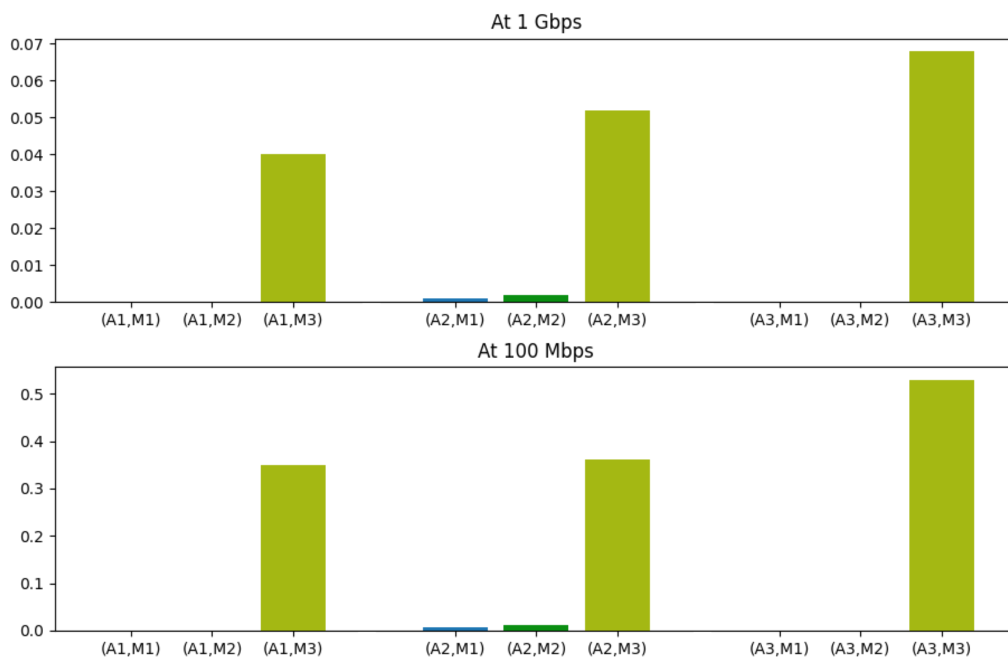


Figure 8.9: Time to copy the additional user and application state to target host (in seconds)

8.3.3 Time Spent on the Target MEC Host

In strategies `M1` and `M2`, the following operations needed to be run on the target MEC host:

1. Create a new DV by running the Docker `volume create` command and add to it the contents of the tarball received from the source MEC host
2. Boot up the container by running the Docker `run` command, mounting the volume just created

In strategy `M3`, however, a slightly different set of commands had to be run:

1. Create a new DV by running the Docker `volume create` command and add to it the contents of the tarball received from the source MEC host
2. Use the Docker `create` command to create a new instance of the container without booting it up
3. Use the Docker `start` command to start the container from the appropriate checkpoint

All the three strategies showed promising results. All of them managed to complete all their operations in less than 1 second, with `M1` always being the fastest, and `M3` always being the slowest. Figure 8.10 gives an overview of the total amount of time spent performing operations on the target MEC host.

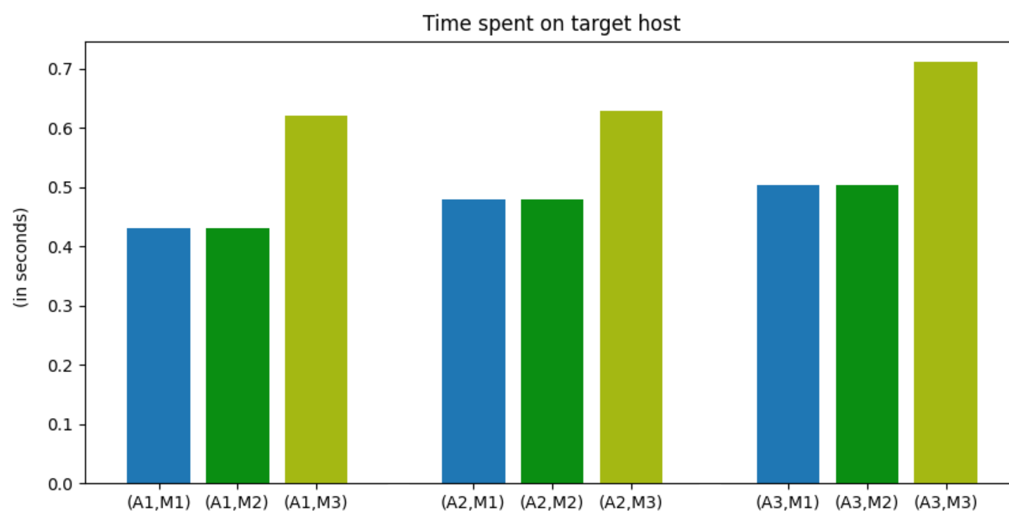


Figure 8.10: Time spent on target host during relocation

8.4 Analysis

Using the results obtained, the three migration strategies could be compared and analysed. The following metrics were considered:

8.4.1 Service Downtime

This is one of the most important metrics of the testbed, and all the migration strategies were designed to minimize it. It starts as soon as the Docker `stop` command is run on the source MEC host and ends only when the MEC application has started successfully on the target MEC host.

As can be seen in Figure 8.11, operations on the source MEC host were responsible for a significant portion of the downtime. The copy operations contributed very little to the total service downtime, primarily because of the small amounts of data transferred during the relocation phase. The bulk of the data, as mentioned earlier, was transferred during the pre-relocation phase. Lastly, the time spent performing operations on the target MEC host was relatively short, with strategy `M3` taking up the most time.

Network congestion had only a minor impact on service downtimes for strategies `M1` and `M2`. Strategy `M3`, however, did see a large increase in the downtime for all the three applications.
Given these results, it's obvious that strategy `M1` is the fastest, regardless of the application being migrated. Therefore, if minimizing service downtime is the only priority, `M1` would be the best choice for application migration.

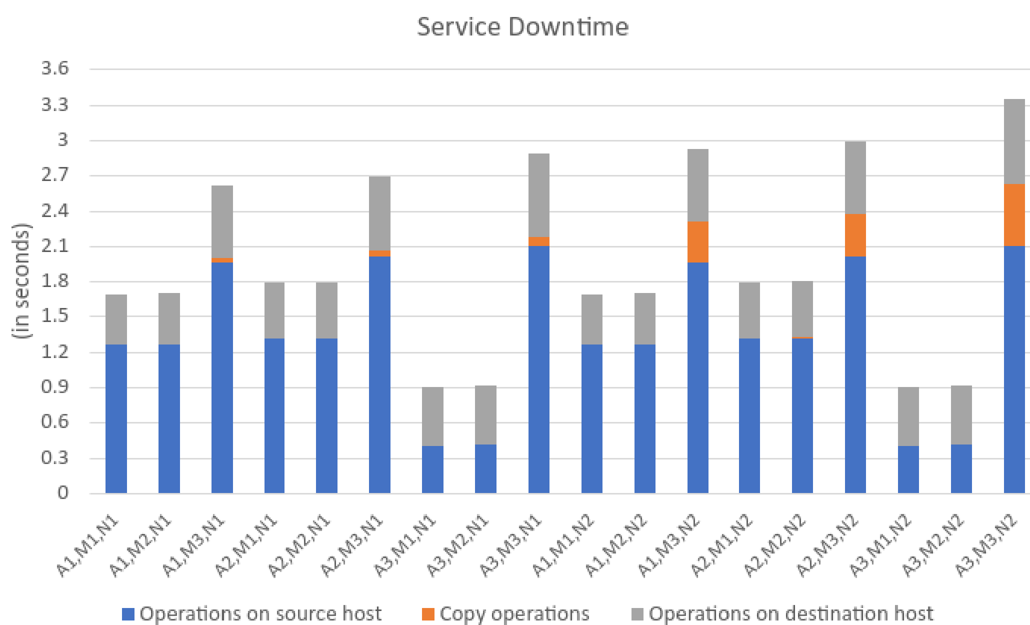


Figure 8.11: Overview of service downtimes

	Strategy M1	Strategy M2	Strategy M3
Application A1	Yes (with some manual inputs)	Yes	Yes
Application A2	No	Yes	Yes
Application A3	No	No	Yes

Table 8.1: Viability of migration strategies for the V2X applications

8.4.2 Amount of state preserved

As discussed earlier, with strategy `M1`, there was no configuration information available on the target host. As a result, with this strategy, the Docker `run` command didn't know how to start the MEC application. This problem was overcome by manually passing the `CMD` instruction to the Docker `run` command.

For strategies `M2` and `M3`, there was enough configuration information available to start all the MEC applications. But only strategy `M3` was able to preserve the complete state of application `A3`.

Therefore, if state preservation was the only criteria, strategy `M3` would be the ideal choice.

8.4.3 Viability

For the sake of completeness, two different viability metrics were considered. These were the most important metrics because if a migration strategy is not viable, the other metrics either can't be calculated or don't matter.

The first viability metric questioned whether an application was able to start and run correctly on the target MEC host after it was migrated using a given migration strategy. As expected, strategy `M1` was the least viable because with it even application `A1`, which needed no state at all, couldn't start without manual intervention. On the other hand, strategy `M3` was the most viable because it could correctly run all the three applications. Table 8.1 gives an overview of this viability metric.

	If edge host is associated with each POA	If edge host is associated with each zone
Average time available for migration	12.301s	53.151s
Shortest available time observed	1.037s	1.854s

Table 8.2: Overview of amounts of time available for migration

The second viability metric questioned whether a migration strategy was possible at all in the time frames available in the city scenario the ETSI MEC sandbox simulated. This metric, it was observed, depended only on how the MEC hosts were placed and didn't help in comparing the migration strategies. The rest of this section explains why.

If there was an MEC host associated with each 5G small cell point of access, it was observed that all the migration strategies would be largely viable in the absence of network congestion. On the other hand, if the MEC hosts were associated with zones in the city, instead of the individual points of access, all the migration strategies would be largely viable even in the presence of network congestion.

There were, however, some paths a VUE could take in the city that had areas where none of the migration strategies would be viable, regardless of network congestion and MEC host placement. This was because, in those areas, the VUE would switch between zones and points of access too quickly for a successful migration. Table 8.2 gives an overview of the amounts of time available for a migration.

Chapter 9

Conclusion and Future Works

In this thesis, three migration strategies were used to migrate three different MEC applications from a source MEC host to a target MEC host. All of them used the two-phase migration strategy described by Campolo et al as a foundation. The primary difference between the three strategies was the amount of user-specific and application-specific state and configuration data they preserved.

It was found that each migration strategy was viable so long as it was used to migrate only those applications whose user-specific and application-specific state and configuration data preservation requirements it could meet. Strategy `M1` was the fastest strategy, and it can be considered ideal for migration of applications that are specifically designed for it. Strategy `M3` was found to be the slowest, but it can be used to migrate any application that is supported by the CRIU tool.

Strategy `M2` preserves enough state and is generic enough to potentially support a large number of open source applications. It is also nearly as fast as strategy `M1`. Therefore, in scenarios where minimizing service downtime and preserving small amounts of application configuration data and user-specific state are both critically important, it would be the ideal strategy.

The results of this testbed will apply to real-world MEC applications that are based on common frameworks and components such as NodeJS

and Memcached. However, it could be extended in the future by integrating it with an MEC system that has a full-fledged orchestrator and other management-level entities that make the SmartRelocation feature possible. Doing so would result in a more automated workflow and more accurate observations of service downtimes and amounts of data transferred during the experiments. Furthermore, the MEC applications developed during this thesis could be upgraded to use the publish-subscribe model instead of the request-response model while accessing the MEC services. This would lead to better optimized bandwidth and CPU resource consumption.

List of Figures

2.1 MEC System Overview 5

4.1 Overview of baseline migration strategy 19

5.1 Overview of more stateful migration strategy 24

6.1 Overview of live migration strategy 27

7.1 Overview of the experimental setup at the start of the migration 30
7.2 Screenshot of the sandbox UI 32

8.1 Container image sizes 38
8.2 Tarball sizes 39
8.3 Pre-relocation time spent on source host (in seconds) 40
8.4 Freeze durations during commit operation 40
8.5 Pre-relocation time spent copying the tarball (in seconds) . . 41
8.6 Pre-relocation time spent on the target host (in seconds) . . 42
8.7 Size of Docker Volume for application A2 given two network conditions 44
8.8 Time spent on source host during relocation 44
8.9 Time to copy the additional user and application state to target host (in seconds) 45
8.10 Time spent on target host during relocation 46
8.11 Overview of service downtimes 47

List of Tables

3.1 Overview of earlier testbeds 13

6.1 Overview of the three migration strategies 28

7.1 Overview of the three MEC applications migrated during the experiments 35

8.1 Viability of migration strategies for the V2X applications . . 48

8.2 Overview of amounts of time available for migration 49

Appendix A

Code Listings

A.1 Application A1

A.1.1 Source Code: app.js

```
var got = require('got');

// The URL of the RNIS API through the API gateway
var url = 'http://gateway-app/rni/v2/queries/rab_info';

function getRABInfo() {
    // Make a HTTP GET request to the URL
    got(url).then((response) => {
        // Parse the JSON document returned
        var data = JSON.parse(response.body);
        try {
            // Print the E-RAB info obtained
          console.log(data['cellUserInfo'][0]['ueInfo'][0]['erabInfo']);
        } catch(e) {
            console.log('No RAB info available');
        }
    });
}

// Repeat every 200 ms
setInterval(getRABInfo, 200);
```

A.1.2 Source Code: package.json

```
{
  "name": "basic-v2x",
  "version": "1.0.0",
  "description": "Basic v2x application",
  "main": "app.js",
  "keywords": [
    "v2x"
  ],
  "author": "Hathibelagal",
  "license": "ISC",
  "dependencies": {
    "got": "^11.8.2"
  }
}
```

A.1.3 Dockerfile

```
FROM node:16-alpine3.11

COPY . .

RUN npm install

CMD node app.js
```

A.2 Application A2

A.2.1 Source Code: app.js

```
var got = require('got');
var fs = require('fs');
var express = require('express');
var app = express()

// The URL of the Location Service API through the API
   gateway
var url = 'http://gateway-app/location/v2/queries/users';
```

```
8
9 var oldZone = -1;
10 var zoneCounter = 0;
11 var inMigration = 0;
12 function getUserInfo() {
13     // Make a HTTP GET request to the URL
14     got(url).then((response) => {
15         var data = JSON.parse(response.body);
16         console.log(inMigration);
17         if(!inMigration) {
18             console.log(zoneCounter);
19         console.log(JSON.stringify(data));
20         } else {
21             var runtimeLogs = "";
22             try {
23                 runtimeLogs = fs.readFileSync('/data/
   runtimedata.dat', 'utf-8');
24             } catch(noFile) {}
25                 runtimeLogs += "\n" + JSON.stringify(data);
26                 fs.writeFileSync('/data/runtimedata.dat',
   runtimeLogs);
27         }
28         try {
29             var zone = data['userList']['user'][0]['zoneId'
   ];
30             if(zone != oldZone) {
31                 oldZone = zone;
32                 zoneCounter += 1;
33
34                 // persist state in volume
35                 var state = {
36                     'counter': zoneCounter,
37                     'zone': oldZone
38                 };
39                 fs.writeFileSync('/data/state.json', JSON.
   stringify(state));
40             }
41         } catch(e) {
42             console.log('No zone info available');
43         }
44     });
45 }
46
47 // user state init
```

```
48 try {
49     var data = JSON.parse(fs.readFileSync('/data/state.json'
    , 'utf-8'));
50     zoneCounter = data['counter'];
51     oldZone = data['zone'];
52     console.log("Using saved state");
53 } catch(e) {
54     console.log("No state present");
55 }
56
57 // Repeat every 200 ms
58 setInterval(getUserInfo, 200);
59
60 // An endpoint that this application exposes so that
61 // it can be told when the migration phase changes
62 app.get('/changePhase', (req, res) => {
63     if(req.query.inMigration) {
64         inMigration = 1;
65     } else {
66         inMigration = 0;
67     }
68     fs.writeFileSync('/data/runtimedata.dat', "");
69     res.status(200).send("OK");
70 });
71
72 // Start a web server so the endpoint is accessible
73 app.listen(3000, () => {
74     console.log("Started app");
75 });
```

A.2.2 Source Code: package.json

```
1 {
2   "name": "some-state-v2x",
3   "version": "1.0.0",
4   "description": "V2X application needing user state",
5   "main": "app.js",
6   "keywords": [
7     "v2x"
8   ],
9   "author": "Hathibelagal",
10  "license": "ISC",
```

```
  "dependencies": {
    "got": "^11.8.2",
    "express": "^4.17.1"
  }
}
```

A.2.3 Dockerfile

```
FROM node:16-alpine3.11

COPY . .

RUN npm install

EXPOSE 3000

CMD node app.js
```

A.3 Application A3

Note that the container for application `A3` contains only the Memcached server. The NodeJS code to add data to it needs to be run independently or in its own separate container.

A.3.1 Source Code: app.js

```
var memcached = require('memcached-promise');
var client = new memcached('127.0.0.1:3000');
var got = require('got');

async function addToMemcache() {
    var url = 'http://localhost:8080/' +
                'location/v2/queries/zones';
    var response = await got(url);
    var data = JSON.parse(response.body);
    var zones = data["zoneList"]["zone"];
    for(var i=0;i<zones.length;i++) {
        var zoneId = zones[i]["zoneId"];
```

```
13          var nPOA = zones[i]["numberOfAccessPoints"];
14          try {
15              await client.add(zoneId, nPOA, 100);
16              console.log(`Added ${zoneId}`);
17          } catch(e) {
18              console.log(`Not added ${zoneId}. May be present
    already`);
19          }
20      }
21  }
22
23  async function start() {
24      await addToMemcache();
25      // Test if the data is available on Memcache now
26      var data = await client.get('zone01');
27      console.log(data);
28  }
29
30  start();
```

A.3.2 Source Code: package.json

```
1  {
2    "name": "live",
3    "version": "1.0.0",
4    "description": "Stores POA data on Memcached",
5    "main": "app.js",
6    "author": "",
7    "license": "ISC",
8    "dependencies": {
9      "got": "^11.8.2",
10     "memcached": "^2.2.2",
11     "memcached-promise": "^1.0.1"
12   }
13 }
```

A.3.3 Dockerfile

```
1  FROM memcached:1.6.9
2
```

```
USER root
RUN apt-get update; apt-get install nodejs -y
USER memcache
```

A.4 Gateway Configuration

The following is the configuration of the reverse proxy for connecting to the ETSI MEC sandbox.

A.4.1 Source Code: httpd.conf

```
ServerRoot "/usr/local/apache2"
Listen 80

# Default modules
LoadModule mpm_event_module modules/mod_mpm_event.so
LoadModule authn_file_module modules/mod_authn_file.so
LoadModule authn_core_module modules/mod_authn_core.so
LoadModule authz_host_module modules/mod_authz_host.so
LoadModule authz_groupfile_module modules/
    mod_authz_groupfile.so
LoadModule authz_user_module modules/mod_authz_user.so
LoadModule authz_core_module modules/mod_authz_core.so
LoadModule access_compat_module modules/mod_access_compat.so
LoadModule auth_basic_module modules/mod_auth_basic.so
LoadModule reqtimeout_module modules/mod_reqtimeout.so
LoadModule filter_module modules/mod_filter.so
LoadModule mime_module modules/mod_mime.so
LoadModule log_config_module modules/mod_log_config.so
LoadModule env_module modules/mod_env.so
LoadModule headers_module modules/mod_headers.so
LoadModule setenvif_module modules/mod_setenvif.so
LoadModule version_module modules/mod_version.so
LoadModule unixd_module modules/mod_unixd.so
LoadModule status_module modules/mod_status.so
LoadModule autoindex_module modules/mod_autoindex.so
LoadModule dir_module modules/mod_dir.so
LoadModule alias_module modules/mod_alias.so

# Modules necessary for the gateway
```

```
29 LoadModule rewrite_module modules/mod_rewrite.so
30 LoadModule proxy_module modules/mod_proxy.so
31 LoadModule proxy_http_module modules/mod_proxy_http.so
32 LoadModule ssl_module modules/mod_ssl.so
33 LoadModule proxy_connect_module modules/mod_proxy_connect.so
34
35 <IfModule unixd_module>
36 User daemon
37 Group daemon
38 </IfModule>
39 ServerAdmin localhost
40 <Directory />
41     AllowOverride none
42     Require all denied
43 </Directory>
44 DocumentRoot "/usr/local/apache2/htdocs"
45 <Directory "/usr/local/apache2/htdocs">
46     Options Indexes FollowSymLinks
47     AllowOverride None
48     Require all granted
49 </Directory>
50 <IfModule dir_module>
51     DirectoryIndex index.html
52 </IfModule>
53 <Files ".ht*">
54     Require all denied
55 </Files>
56 ErrorLog /proc/self/fd/2
57 LogLevel warn
58 <IfModule log_config_module>
59     LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\"
    \"%{User-Agent}i\"" combined
60     LogFormat "%h %l %u %t \"%r\" %>s %b" common
61     <IfModule logio_module>
62       LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\"
    \"%{User-Agent}i\" %I %O" combinedio
63     </IfModule>
64     CustomLog /proc/self/fd/1 common
65 </IfModule>
66 <IfModule alias_module>
67     ScriptAlias /cgi-bin/ "/usr/local/apache2/cgi-bin/"
68 </IfModule>
69 <Directory "/usr/local/apache2/cgi-bin">
70     AllowOverride None
```

```
71     Options None
72     Require all granted
73 </Directory>
74 <IfModule headers_module>
75     RequestHeader unset Proxy early
76 </IfModule>
77 <IfModule mime_module>
78     TypesConfig conf/mime.types
79     AddType application/x-compress .Z
80     AddType application/x-gzip .gz .tgz
81 </IfModule>
82 <IfModule proxy_html_module>
83 Include conf/extra/proxy-html.conf
84 </IfModule>
85 <IfModule ssl_module>
86 SSLRandomSeed startup builtin
87 SSLRandomSeed connect builtin
88 </IfModule>
89 
90 SSLProxyEngine On
91 <VirtualHost *:*>
92 ProxyPass / https://try-mec.etsi.org/sbxfr7e7k2/
93 ProxyPassReverse / https://try-mec.etsi.org/sbxfr7e7k2/
94 </VirtualHost>
```

A.4.2 Dockerfile

```
1 FROM httpd:2.4
2 COPY ./httpd.conf /usr/local/apache2/conf/httpd.conf
```

A.5 Congestion Implementation

The following is the script that runs `tc` to simulate network congestion.

A.5.1 Source Code: congestion.sh

```
1 echo "Limiting"
```

```
2  # Create a classful queueing discipline of type Hierarchical
       Token Bucket
3  tc qdisc add dev eno1 root handle 1: htb default 99
4  echo $?
5  # Create a class with rate of 100 mbit
6  tc class add dev eno1 parent 1: classid 1:99 htb rate 100
     mbit
7  echo $?
8  # Wait until user hits enter
9  read a
10 # And cleanup
11 echo "Deleting"
12 tc qdisc del dev eno1 root
13 tc qdisc show
```

Bibliography

[1] Jian Wang, Yameng Shao, Yuming Ge, and Rundong Yu. A survey of vehicle to everything (v2x) testing. *Sensors*, 19(2):334, 2019.

[2] Sami Kekki, Walter Featherstone, Yonggang Fang, Pekka Kuure, Alice Li, Anurag Ranjan, Debashish Purkayastha, Feng Jiangping, Danny Frydman, Gianluca Verin, et al. Mec in 5g networks. *ETSI white paper*, 28:1–28, 2018.

[3] Fabio Giust, Xavier Costa-Perez, and Alex Reznik. Multi-access edge computing: An overview of etsi mec isg. *IEEE 5G Tech Focus*, 1(4):4, 2017.

[4] ETSI ISG. Multi-access edge computing (mec); framework and reference architecture. *ETSI Standards Search*, 2020.

[5] Dongkee Lee and Woohyun Nam. Case study of scaled-up skt* 5g mec reference architecture. *Intel Whitepapers*, 2021.

[6] Tero Lähderanta, Teemu Leppänen, Leena Ruha, Lauri Lovén, Erkki Harjula, Mika Ylianttila, Jukka Riekki, and Mikko J Sillanpää. Edge computing server placement with capacitated location allocation. *Journal of Parallel and Distributed Computing*, 153:130–149, 2021.

[7] ETSI ISG. Multi-access edge computing(mec);edge platform application enablement. *ETSI Standards Search*, 2019.

[8] ETSI ISG. Multi-access edge computing (mec); phase 2: Use cases and requirements. *ETSI Standards Search*, 2018.

[9] ETSI ISG. Multi-access edge computing (mec); radio network information api. *ETSI Standards Search*, 2019.

[10] ETSI ISG. Etsi gr mec 022 v2.1.1 (2018-09)multi-access edge computing(mec); study on mec support for v2x use cases. *ETSI Standards Search*, 2018.

[11] ETSI ISG. Multi-access edge computing (mec);v2x information service api. *ETSI Standards Search*, 2020.

[12] ETSI 3rd Generation Partnership Project. 5g; ng ran architecture description. *ETSI Technical Specifications*, 2018.

[13] ETSI ISG. Multi-access edge computing (mec); application mobility service api. *ETSI Standards Search*, 2020.

[14] Shangguang Wang, Jinliang Xu, Ning Zhang, and Yujiong Liu. A survey on service migration in mobile edge computing. *IEEE Access*, 6:23511–23528, 2018.

[15] Khasa Gillani and Jong-Hyouk Lee. Comparison of linux virtual machines and containers for a service migration in 5g multi-access edge computing. *ICT Express*, 6(1):1–2, 2020.

[16] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 171–172. IEEE, 2015.

[17] Rami Akrem Addad, Diego Leonel Cadette Dutra, Miloud Bagaa, Tarik Taleb, and Hannu Flinck. Towards a fast service migration in 5g. In *2018 IEEE Conference on Standards for Communications and Networking (CSCN)*, pages 1–6. IEEE, 2018.

[18] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[19] Alessandro Randazzo and Ilenia Tinnirello. Kata containers: An emerging architecture for enabling mec services in fast and secure way. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 209–214. IEEE, 2019.

[20] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. Includeos: A minimal, resource efficient unikernel

for cloud services. In *2015 IEEE 7th international conference on cloud computing technology and science (cloudcom)*, pages 250–257. IEEE, 2015.

[21] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A binary-compatible unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 59–73, 2019.

[22] Ivan Farris, Tarik Taleb, Hannu Flinck, and Antonio Iera. Providing ultra-short latency to user-centric 5g applications at the mobile network edge. *Transactions on Emerging Telecommunications Technologies*, 29 (4):e3169, 2018.

[23] Haikun Liu, Hai Jin, Xiaofei Liao, Liting Hu, and Chen Yu. Live migration of virtual machine based on full system trace and replay. In *Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 101–110, 2009.

[24] Simon Pickartz, Niklas Eiling, Stefan Lankes, Lukas Razik, and Antonello Monti. Migrating linux containers using criu. In *International Conference on High Performance Computing*, pages 674–684. Springer, 2016.

[25] Claudia Campolo, Antonio Iera, Antonella Molinaro, and Giuseppe Ruggeri. Mec support for 5g-v2x use cases through docker containers. In *2019 IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1–6. IEEE, 2019.

[26] Paolo Bellavista, Antonio Corradi, Luca Foschini, and Domenico Scotece. Differentiated service/data migration for edge services leveraging container characteristics. *IEEE Access*, 7:139746–139758, 2019.

[27] ETSI ISG. Mobile edge computing (mec);end to end mobility aspects. *ETSI Standards Search*, 2017.

[28] Brian Ward. *How Linux works: What every superuser should know.* no starch press, 2021.

[29] Radostin Stoyanov and Martin J Kollingbaum. Efficient live migration of linux containers. In *International Conference on High Performance Computing*, pages 184–193. Springer, 2018.

[30] Yuhei Takagawa and Katsuya Matsubara. Yet another container migration on freebsd. In *AsiaBSDCon 2019 Proceedings*, pages 97–102, 2019.

[31] James R Blakley, Roger Iyengar, and Michel Roy. Simulating edge computing environments to optimize application experience. 2020.

[32] ETSI ISG. Multi-access edge computing (mec); location api. *ETSI Standards Search*, 2019.

[33] Wei Wang, Yongli Zhao, Massimo Tornatore, Abhishek Gupta, Jie Zhang, and Biswanath Mukherjee. Virtual machine placement and workload assignment for mobile edge computing. In *2017 IEEE 6th International Conference on Cloud Networking (CloudNet)*, pages 1–6. IEEE, 2017.

[34] Mateusz Berezecki, Eitan Frachtenberg, Mike Paleczny, and Kenneth Steele. Many-core key-value store. In *2011 International Green Computing Conference and Workshops*, pages 1–8. IEEE, 2011.

[35] Seung-Jun Cha, Seung Hyub Jeon, Yeon Jeong Jeong, Jin Mee Kim, Sungin Jung, Sangheon Pack, et al. Boosting edge computing performance through heterogeneous manycore systems. In *2018 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 922–924. IEEE, 2018.

[36] Ahmed Soliman. *Getting Started with Memcached*. Packt Publishing Ltd, 2013.