Experimental Comparison of Migration Strategies for MEC-Assisted 5G-V2X Applications

Mohammed A. Hathibelagal
University of Stavanger
Stavanger, Norway
ma.hathibelagal@stud.uis.no

Rosario G. Garroppo
University of Pisa
Pisa, Italy
rosario.garroppo@unipi.it

Gianfranco Nencioni
University of Stavanger
Stavanger, Norway
gianfranco.nencioni@uis.no

***Abstract*—The introduction of 5G technology enables new V2X services requiring reliable and extremely low latency communications. To satisfy these requirements computing elements need to be located at the edge of the network, according to the Multi-access Edge Computing (MEC) paradigm. The user mobility and the MEC approach lead to the need to carefully analysing the procedures for the migration of applications necessary to maintain the service proximity, fundamental to guarantee low latency. The paper provides an experimental comparison of three different migration strategies. The comparison is performed considering three different containerized MEC applications that can be used for developing V2X services. The experimental study is carried out by means of a testbed where the user mobility is emulated by the ETSI MEC Sandbox. The three strategies are compared considering the viability, the observed service downtime, and the amount of state preserved after the migration. The obtained results point out some trade-offs to consider in any migration scenario.**

***Index Terms*—5G, Multi-access Edge Computing, V2X, Container, Migration**

I. Introduction

The experience of being on the road is going to change significantly as telecommunications service providers transition to the 5^{th} generation of 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 support ultra-low latency services other than applications requiring higher peak data. The 5G ultra-low latency feature enables the support of novel or enhanced 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]. The cloud paradigm is inadequate to satisfy the low latency and high reliability requirements of some V2X services. Indeed, having the applications in the cloud implies data exchange with distant servers. The required compute and data storage resources must be available closer to the edge of the network, and thus, closer to the end user. Therefore, other than 5G, these services require another enabler technology such as Multi-access Edge Computing (MEC).

The MEC standards by the European Telecommunications Standards Institute (ETSI) consider that both MEC application and MEC platform may be consumers of a set of MEC services, such as Radio Network Information Service (RNIS) [2] or Location Service (LS) [3]. MEC application can use MEC services to get contextual information, such as information on users in the cells or user location and velocity. The contextual information given by MEC services enhance V2X applications. As discussed in [1], on one hand the design of efficient MEC-integrated V2X applications require the usage of MEC services. On the other hand, the definition of new MEC services or changes to existing ones are necessary to support enhanced V2X applications.

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 (MEH) to another as the UE moves out of and into the areas covered by the respective MEHs. The relocation feature is especially important for V2X services because the primary actors involved (i.e. cars) are expected to cover large distances over a short period of time.

The strategies for migrating the applications should be analysed taking into account the two application classes: stateful or stateless. In the case of stateful, the application migration requires transferring and synchronizing the service state between the original and relocated application instance to provide service continuity. Furthermore, the synchronization of the application state depends on the implementation of the application itself. The application developer should consider the migration issues, designing the application in such a way that multiple instances of the application can run concurrently, and the state (context) of the application instance can be captured in the source instance and copied to another instance independently from the operation of the instance itself. In this manner, the relocated application instance in the target MEH can continue in a seamless manner from the state of the application instance in the source MEH at the time of UE disconnection from that. On the contrary, the support for the migration of a stateless application is relatively simple because

most likely no application context is necessary to transfer, as well as no synchronization should be guaranteed between the instance at the source MEH and at the target MEH.

A. Paper Contribution

The paper is focused on the strategies for application migration between MEHs. The paper has the following assumptions:

- The paper does not consider the Radio Access Network (RAN), and specifically the radio interface. The motivation is that the paper is focusing in the application migration between two different MEH and the traffic exchanged during the migration is not transmitted over the radio interface. User mobility and varying channel conditions add impairments on the traffic path between UE and MEH. For this reason, the wireless network connectivity does not impact on the performance of the migration between MEHs. The paper, instead, assumes that there is a transport network with an available path between the two MEHs.
- The paper does not consider the latency of the V2X applications (as for example in [4]) because this parameter is mainly related to the link between the MEH and the UE. This feature may be part of the decision algorithm deciding when to perform the Point-of-Access (PoA) handover or/and the MEH handover. Furthermore, to improve the performance of the path between the UE and the MEH some approaches, such as duplicate MEHs, can be considered [5]. Instead, the paper is focusing on the downtime as interruption of the MEC V2X application, so as part of the reliability.

The main contributions of the paper are the followings.

- Two extensions of the pre-relocation strategy presented by Campolo et al. [6] to support applications that require the preservation of only user state and of both user and application states, respectively. The considered applications differs from the amounts of state information necessary for resuming them correctly after the migration is completed. The extensions cope with the set of commands and tools to use for the implementation of the pre-relocation and the relocation procedures. Although earlier testbeds had tried live migrations, they did not have well-defined pre-relocation stages and how to implement them.
- The comparison of migration strategies is performed by using the emulation of user mobility and MEC Application Programming Interfaces (APIs) given by the ETSI MEC Sandbox [7]. Previous works did not consider this.
- The comparison is carried out considering three simple applications developed with the goal of using RNIS and LS MEC services and that have different state-preservation requirements. The ETSI MEC Sandbox is used to access to MEC service via MEC API during the emulation of user mobility. The applications are simple but can be seen as a microservice for more complex V2X Application. Contrary to the modern monolithic architecture where an application self-contains all the components, the microservice architecture is an approach to developing an application as a set of small independent services. Each of the services is running in its own independent process [8]. More complex V2X services can use the elementary application we used in our analysis to access the MEC services. The developed applications use both LS and RNIS, which, as previously mentioned, are important in V2X use cases [1].
- The experimental analysis has as final target to assess the viability of the migrated container at the target MEH for each migration strategy and application. Earlier analysis focused more on the size of the container and associated data being migrated, presenting results mainly in terms of service downtime and re-location latency.

The paper is organized as follows. Section II overviews the related works. Section III presents the considered migration strategies, while Section IV describes the testbed used for the experimental analysis. Further, this section presents the three applications considered in the study. Section V shows the experimental results, while the summary of the performance given by the compared strategies is presented in Section VI. Finally, Section VII concludes the paper.

II. Related Works

Service migration is a very active research topic. Migration strategies are closely coupled to the technologies used for resource isolation and hosting the applications. According to Wang et al's survey [9], the two most widely-used technologies are: Virtual Machine (VM) technology and container technology. 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 [10]. Different optimization problems have been studied for the optimal VM placement and migration frameworks, such as the recent [11] [12].

Compared to VMs, containers 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 the 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 [13]. Nevertheless, containers offer some important advantages, such as small image size, very small memory footprint, and fast instantiation times. These features make containers interesting for edge use cases. As reported in Randazzo et al. [14] Docker is one of the most widely deployed container platforms. It is also worth mentioning that there is a rise in a new breed of VMs based on unikernels [15], and another emerging technology called kata-containers [14]. However, these emerging lightweight VM approaches are still experimental and not widely used. Thus, this paper focuses primarily on container-based migration strategies. Fondo-Ferreiro et al. [16] proposes an architecture for automated orchestration for deploying applications at the best location and even that relocate them, when necessary, to satisfy the Quality of Service (QoS) requirements. They

implemented the proposed architecture in an experiment that demonstrates how Open Source MANO (OSM) can automate the relocation of a video processing application that helps drivers to remember the latest traffic sign viewed.

This paper focuses on the experimental comparison of different migration strategies, considering three different typologies of applications.

Focusing on the migration strategies, Farris et al. [17] describe an early proactive strategy for service migration, which involved maintaining replicas of the service on multiple MEHs to minimize downtime. They performed the experimental analysis of the proposed approach using a testbed composed of two MEHs each one implemented by a PC with the Docker engine. As metrics, they consider the total migration time and the initialization time. However, because there is no actual migration of the container filesystem, the downtimes they observe (on average lower than 2 s) are a function of the used Docker Volume (DV). Indeed, in their tests, they used the same Docker container image for all scenarios and changed only the DV size. Although this approach shows very short service interruptions, it does not efficiently use the MEH resources. In fact, the service needs to be instantiated on several MEHs even though the UE is connected to only one of them at any given instant. Farris et al. have suggested optimizations such as replicating the service only on MEHs that lie in the direction of movement of the UE.

Most studies 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 an 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 MEH. Instead, it simply resumes from its state on the source MEH. This can be accomplished by various methods. Earlier methods involved maintaining a log of all events generated by the container on the source MEH and replaying the log on the container of the target MEH. They are simply a container-oriented adaptation of the system trace and replay approach commonly used in a live migration of VMs [18]. This approach is quite error-prone, especially when there are lots of asynchronous events. Therefore, more recent methods favour directly copying both the filesystem contents and memory pages of the container on the source MEH to the target MEH [19].

For instance, Addad et al. [20] propose a live migration strategy in the context of 5G, aimed to minimize both downtime and total migration time. Their experimental analysis has been carried out with a testbed consisting of virtualized nodes running Ubuntu 16.04. They used Linux Containers (LXC) and the Checkpoint/Restore In Userspace (CRIU) project for creating application containers and managing their memory pages. Their experiments ran with two containers: a blank Linux container and a larger Linux container with a video streaming server installed in it. They compared both stateful and stateless migration scenarios. Their results show 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 MEH has a large RAM. Therefore, Addad et al. followed an iterative approach to transferring the memory. This approach results in larger network resources consumption but keeps the service downtime low. More recently the authors propose two different algorithms to optimize network resources allocation as well as to adjust the network usage to minimize slice migration overhead [21]. Stojanov et al. [22] analyze the performance of a new CRIU feature, the so-called image cache/proxy. Their results show that the total-copy using rsync generates a lower migration time than the image-cache/proxy technique. We use this result for a detailed definition of a migration strategy based on the CRIU tool with the total-copy using rsync.

Campolo et al. [6] propose a custom migration strategy based on Docker containers. The performance has been experimentally evaluated by means of a simple testbed composed of two PCs with the Docker Engine. The analysis considers two containers with different filesystem sizes and multiple DVs with different file sizes. Their strategy is specifically designed for V2X services [5]. Their migration strategy has two phases and relies extensively on the use of DVs. In the first phase, called the service pre-relocation phase, they migrate only the filesystem of the source container. In the second phase, called the service relocation phase, they migrate only the additional state of the source container, which is now expected to be in a small DV. Because there is no downtime during the service pre-relocation phase, transferring the DV and booting up the container account for most of the observed downtime. This strategy allowed them to reduce service downtimes to approximately 2 seconds, so long as the DV is 10 KB or smaller. Furthermore, the downtime is largely independent of 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. Indeed, the application acts differently depending on the currently active phase. 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 store all of its state in a DV.

The previously cited works focused only on the size of the container's filesystem, the user state, and the container memory pages. They are application-agnostic strategies, i.e. they work the same regardless of the service application being migrated. Bellavista et al. [23] suggest a proactive handoff strategy very similar to [6], with the extension that their strategy is 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. The experimental analysis has been carried out on a heterogeneous testbed composed of two PCs and a Raspberry Pi3, all running Linux. Instead of Docker alone, they use Docker Compose to simplify the code necessary to instantiate multiple containers and to set up their DVs in an error-free manner. The considered service is a Java web application, using the MongoDB database. Thus, their application has 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 considered service has a modular architecture and can be easily split into distinct layers. For example, the distinction between the service layer and the data layer 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.

III. Migration Strategies

The MEC application migrates in two phases: i) the pre-relocation and ii) the relocation. The MEC orchestrator is responsible for triggering each phase at an appropriate time. Figure 1 gives an overview of the sequence of events that occur in a successful migration. In this work, the compared migration strategies differ by the actions taken in the event indicated as `A` and `B` in the figure.

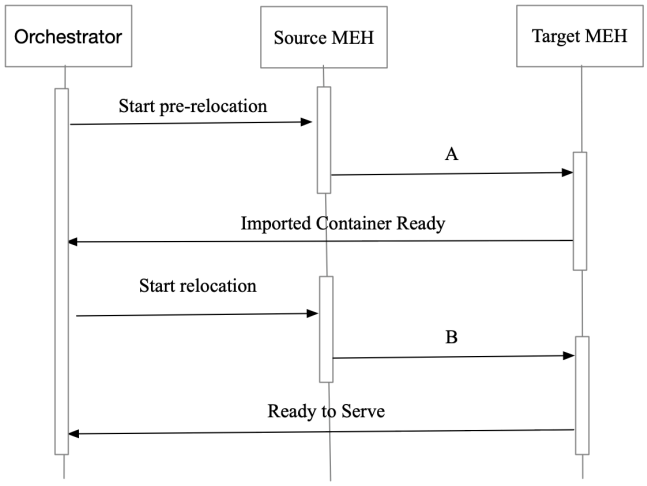

Fig. 1. General scheme of the migration procedure

The service pre-relocation could be triggered by i) MEC applications (client or MEC side), ii) Source/target MEHs using the associated Radio Network Information Service (RNIS) or data planes, and iii) the MEC Orchestrator (MEO) [24]. In the presented experimental analysis, the assumption is the MEO generates the triggers. Like in [6], the MEO is implemented as a set of shell commands and scripts. This approach allows for creating a manual MEO and to time all the phases, i.e. running the appropriate Docker Engine commands and timing them.

A. Simple Filesystem Preservation (SFP)

A first study on the performance obtained for this kind of migration strategy is presented in [6].

The strategy is characterized by the following procedure `A`. The layered filesystem of the Docker container on the source MEH is flattened and exported into a tarball using the Docker `export` command. This tarball is then securely copied to the target MEH using the `scp` tool and used to create a new container image using the Docker `import` command.

Throughout the service pre-relocation phase, the source MEH continues to serve the UE without any interruptions. On the contrary, the container on the source MEH is shut down during the relocation phase. As soon as this phase begins, the UE starts experiencing service downtime and the rest of the steps have to be completed as quickly as possible.

The procedure `B` consists in shutting down the container and copying its DV to the target MEH.

Given the assumption that the DV contents are merely Floating Car Data (FCD) packets, the remote copy mechanism is a combination of the Linux `dd` and `nc` commands. 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 MEH, the container image created in the previous phase is instantiated and booted up so the UE can connect to it. The DV is mounted as the container is booted up. At this point, the service downtime ends. To ensure the integrity of the UE's data available on the target MEH after the migration, the service running on the container is expected to be aware of the migration. Furthermore, the service implements the Algorithm 1, which ensures that there is no change in the filesystem after the tarball is created during the pre-relocation phase.

Algorithm 1 Algorithm implemented by the migration-aware service

```
1: if Migration in progress then
2:     Stop writing to the container's filesystem
3:     Start writing to mounted DV
4: else
5:     Write to the container's filesystem normally
6: end if
```

Algorithm 1 also ensures that all the new information the UE generates during the service pre-relocation phase is available on the DV. Obviously, in this phase, application data could be lost due to the stop of writing to the container's filesystem.

The algorithm is implemented in the applications by exposing an endpoint the orchestrator could use to specify the current phase of migration.

B. Filesystem and Container Configuration Preservation (FCCP)

The SFP presents a big pitfall: 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 MEH is not available at the target MEH. Because the container at the target MEH 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 to store configuration settings [25]. 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 is unlikely to behave the same way it did on the source MEH when it starts on the target MEH. To overcome these issues, the baseline SFP needs to be extended to add some state information. In particular, by making only minor changes to the SFP implementation, all the environment variables, all the Docker instructions, and several application settings can be preserved. The changes lead to FCCP presented and studied in [23]. The Docker `save` and `load` commands are suitable for preserving the basic state information. Therefore, the differences between SFP and FCCP are only in the procedure `A`. In the procedure `A` of FCCP, the Docker `export` command at the source MEH, used in SFP, is replaced with the Docker `save` command. Similarly, at the target MEH, the Docker `import` command is replaced with the Docker `load` command. Unlike the Docker `export` command, 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.

As one might expect, the tarball generated by the Docker `save` command is slightly larger. Indeed, the output of the `save` command contains not only additional state information but also details about all the necessary parent layers, such as their tag names and versions. 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 MEH because the Docker `load` command can handle compressed archives. However, the compression and decompression operations themselves are time-consuming. The tradeoff between the latency added by the compression/decompression operations and the tarball size suggests sending the tarball without compression.

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

C. Full State Preservation (FullSP)

Although the FCCP is capable of preserving much of the container state, it is still the responsibility of the MEC 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, also FCCP can be applied 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 a MEC scenario, it should not depend on any inputs from the applications.

In the case of FullSP, the contents of all the memory pages, CPU registers, and other resources used by the container on the source MEH are additionally copied to the target MEH during the migration. Such a migration is referred to as a live migration in [22]. It is important to note that the migration can now be fully transparent to the MEC application. In other words, after a successful live migration, the MEC application would generally not even notice that it was migrated.

To support this kind of migration, tools such as CRIU are necessary. CRIU can create a detailed copy of a process that is running inside a container. In particular, CRIU 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 [26]. To record these data, CRIU uses `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. To use this tool, CRIU needs to be installed and enabled on both the source and target MEH.

A fundamental difference between the FullSP and the other two migration strategies is that during both the pre-relocation and relocation phases, the Docker `checkpoint` command is run on the source MEH to freeze the 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. However, this setting is not necessary during the relocation phase because the container is not expected to be alive on the source MEH anymore.

The output of the checkpoint operation is a directory containing several CRIU image files. These files are copied to the target MEH so that they can be used to restore the checkpoint-ed application. But this action 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 MEH. For common applications, the easiest way to achieve this is to use Docker Hub or any other container registry available in the MEC system. However, for custom applications the Docker `commit`, `save`, and `load` commands should be used to set up the container, as described in FCCP. The latter approach has been considered in the study. In this manner, the results can be related to a general custom application.

The application is checkpointed twice in order to leverage the `rsync` tool and minimize the service downtime. This

way, in the procedure A the entirety of the memory pages are copied to the target MEH, and in the procedure B of the actual relocation only the changed bits are copied. This is important because the memory pages can often be as large as the container's filesystem. Other than the checkpoints operations, during A the container image is saved to the target MEH, while in B the DV is transferred from the source MEH to the target MEH.

Table I summarizes the main features of the described migration strategies.

	SFP	FCCP	FullSP
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 I
OVERVIEW OF THE THREE MIGRATION STRATEGIES

IV. EXPERIMENTAL SCENARIO

Figure 2 gives an overview of the components and layout of the experimental scenario.

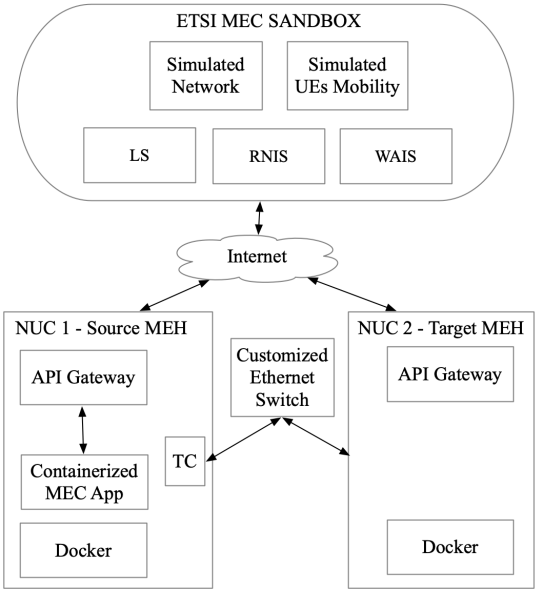

Fig. 2. Overview of the experimental scenario

Two Next Unit of Computing (NUC) small form factor workstations were used as Source and Target MEH. Each has 16 GB of RAM, an Intel Core i7 processor, and a solid-state drive. The workstations can communicate with each other using a customized 1Gb Ethernet link. In the figure, TC represents the Traffic Control utility of Linux used to limit the data rate (e.g., to 100 Mbps) and to add packets loss in some experiments. Furthermore, each NUC has another network interface for its connection to Internet. Both the workstations run Ubuntu 20.04 LTS as the operating system and have Docker Engine 20.10.4 installed on them. Furthermore, to support live migration, CRIU 3.15 is manually compiled from its source code and installed on both the MEHs.

As shown in the figure, the MEC platform is represented by the ETSI MEC Sandbox environment [7].

As Figure 3 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.

At the time of this work, other than WLAN Access Information Service (WAIS) the ETSI MEC Sandbox supported both Location Service (LS) and Radio Network Information Service (RNIS) APIs but implemented only a limited subset of the endpoints mentioned by the ETSI ISG [2], [3]. For example, the LS 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. Furthermore, the ETSI MEC sandbox 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 available network configurations are different primarily in the network technologies they supported, see [7].

The study is carried out using the `4g-5g-wifi-macro` configuration, which is the most flexible and supports all the network technologies that are likely to be available in actual network deployments. The scenario is composed of nineteen 5G small cell PoAs, eleven WiFi PoAs, and ten 4G macro cell PoAs. As concerning the user mobility, the ETSI MEC Sandbox allows the configurations of Stationary UEs, which represent smart city IoT connected devices such as street cameras, smart sensors, etc., the low Velocity UEs, which change location relatively slowly, and the fast moving UEs, which represent motorized vehicles moving along the main city roads. The experimental analysis considers only fast moving and low velocity UEs.

It is important to note that the sandbox does not specify the location of the MEHs. It gives only the locations of the radio PoA and the zones they belong to. Experiments are firstly run assuming that a MEH is present near each of the nineteen 5G PoAs. For the sake of simplicity, it is also assumed that each MEH serves exactly one 5G small cell base station. Then, experiments are run assuming that a MEH is associated only with zones instead of the individual PoA they contain.

The MEC applications are usually expected to connect to the MEC services they need through an API gateway, see figure 2. Therefore, an API gateway is created using Apache HTTP

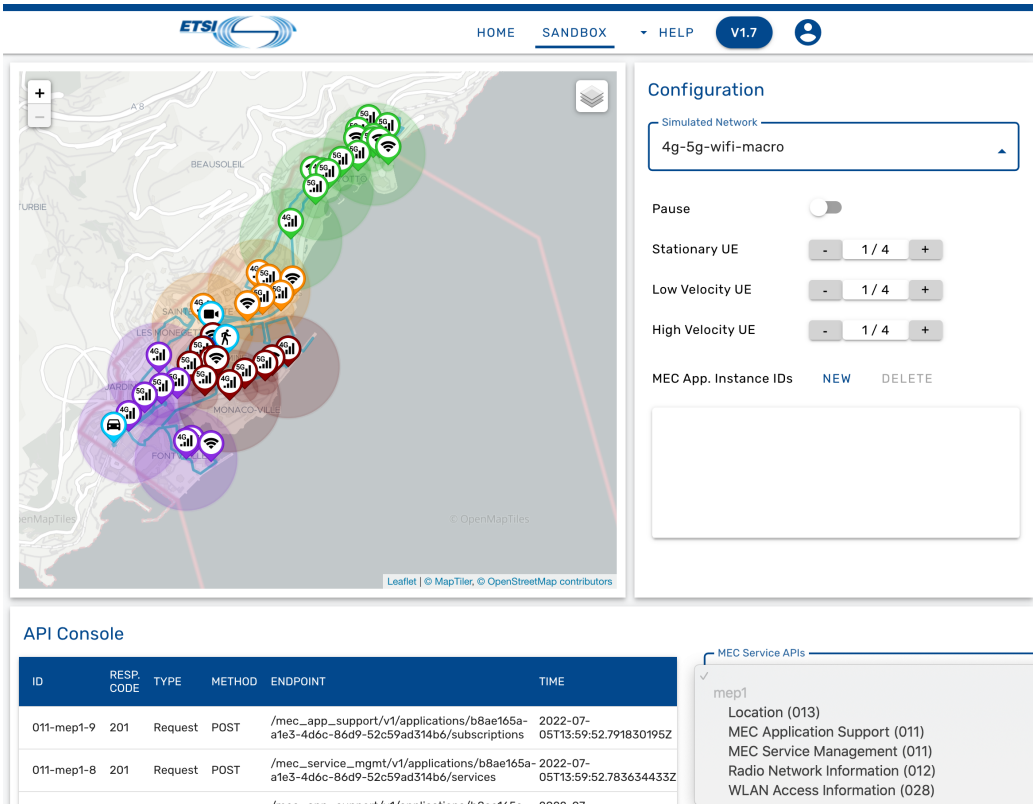

Fig. 3. Screenshot of the ETSI MEC Sandbox user interface

Server 2.4. Using its `mod_proxy` module, a reverse proxy is set up so that any request is routed to the ETSI MEC Sandbox.

The gateway is containerized and is set up on both the source MEH and the target MEH. It has the same Uniform Resource Locator (URL) on both and the MEC applications can interact with it using a Docker bridge network.

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

A. *Applications*

The experimental study considered three containers, each containing a different simple application that uses MEC services. The considered applications are example of microservices for V2X applications that consume MEC services. Considering the necessary state information to preserve during the migration, these applications can be classified as follows.

- **Basic Application (BA)**. The simplest application merely operates as an RNIS consumer. As shown in [1], RNI API and its required enhancement can be used for supporting V2X applications for road safety, such as intersection movement assist and queue warning, or for advanced driving assistance, such as ”Real Time Situational Awareness and High Definition Local Maps” and ”high definition sensor sharing (or see-through)”. The BA is a simple periodic request (with period set to 200 ms) of RNI MEC service. As shown in [27], the developed application requests E-RAB info, which gives data such as the E-UTRAN Cell Global Identifier, the information of users per cell, maximum and guaranteed E-RAB bit rate in uplink and downlink. Migrating this application serves as an initial test for the viability of the testbed implementation. The functionality of this application does not depend on any state information, be it user state or application state. Consequently, the application does not depend on the contents of a DV or any other form of persistent storage.
  The basic application is built using NodeJS 15.14.0 and `got` 11.8.2, which is an HTTP request library.
- **User State-Preserving Application (USPA)**.
  The USPA is a simple periodic request (with period set to 200 ms) of LS. As shown in [27], this application is a simple counter that increments its value whenever the UE’s current zone changes. In general, the LS MEC service can be used for the ”vulnerable road user” V2X use case group. Indeed, the MEC system is evolving for enabling the support of accurate positioning in a short or real-time, exploiting the new capability provided, for example, by 5G system. In general, LS offers information on user position and velocity that can be used to predict possible dangerous events.
  As its name suggests, USPA needs the user-specific state preserved to function correctly. To satisfy a realistic use case, USPA is designed to perform a single, atomic change to the user state. This change involves the in-

crement of a counter that is a part of the user state. During the migration, the value of the counter is of critical importance. Indeed, for a migration to be considered successful, the counter should not start from zero at the target MEH. Instead, it should resume from the value assumed at the source MEH. This emphasizes the point that although the application itself is stateless, the user-specific state must be preserved.
This application too is built using NodeJS and `got`. It is designed to be aware of the migration and uses the DV to store the counter value and run-time logs once the pre-relocation phase started. It is also capable of retrieving the counter value from the DV, if available during startup. It is also worth emphasizing that, much like BA, USPA too does not need to preserve any application instance-specific state to function correctly. USPA needs only the user-specific state.

- **Application With a Stateful Workload (ASW)**. This application interacts with the LS MEC service to store the list of visited zones during the UE movement and the number of PoAs of each visited zone. These information are stored using Memcached. Memcached is a high-speed, in-memory, key-value datastore in edge computing scenarios [28]. 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 [29]. As such, they are large hash tables, with unique keys pointing to important values. Memcached represents an interesting benchmark for testing the live migration strategy of applications with in-memory stateful workload. The selection of Memcached 1.6.9 offers an easy to use command-line interface over Telnet [30], the Linux `telnet` utility is used to retrieve access point-related information on it. This information is obtained from the LS API using the Linux command-line utility `cURL`.

All three applications run inside their separate containers. The container images of BA and USPA are built with the `Node 16-alpine3.11` image as the base. The ASW container image is built with the `memcached 1.6.9` image as the base. All of the base images are pulled from Docker Hub. Table II summarizes the main features of the three applications considered in the experimental analysis.

B. Variable Network Conditions

Containers running the three applications are migrated from the source MEH to the target MEH under two different network conditions: normal and congested. Under normal conditions, the network has a data rate of approximately 1 Gbps. The simulation of the network congestion is obtained using the Linux `tc` utility. To configure `tc` appropriately, a classful queueing discipline of type Hierarchical Token Bucket (HTB) is set up. A class with its `rate` set to 100 Mbps has been added to reduce the data rate to 100 Mbps when the congestion scenario is simulated. For a data rate set to 100 Mbps, different tests have been carried out setting diverse packet loss (PL) probability, i.e. 0%, 0.1%, 1%, and 5%.

	BA	USPA	ASW
State preservation requirements	No state necessary	User-specific state necessary	User-specific and application-specific state necessary
Offered service	Query and log RAB information every 200 ms using the RNIS API	Keep a count of the number of zones visited by the UE, using the LS API	Use Memcached to store the list of all useable PoAs in the visited zone during the UE mobility
Docker Volume usage	No	Yes	No
Migration awareness	No	Yes	No

TABLE II
OVERVIEW OF THE THREE MEC APPLICATIONS CONSIDERED IN THE EXPERIMENTAL ANALYSIS

V. PERFORMANCE ANALYSIS

The performance analysis considers the combination of the application features, the migration strategy, and the network condition. Most of the presented results are obtained using the reference configuration represented by no packet loss and data rate set to 100 Mbps. In all tests considering a 1Gbps link, the observed latency due to the transfer of data necessary for the application migration is one-tenth of that observed in the reference scenario.

All the code used for the experimental performance analysis is available in [27].

A. Container Sizes

For each container, the value of its size is obtained using the Docker `inspect` command. The obtained results indicate that `BA` and `USPA` give similar values, i.e. 111.82 MB against 112.14. On the contrary, the ASW container has a size of 152.17 MB.

The analysis of the tarballs shows a significant difference between the size of the container images and the tarballs generated from the container instances. This difference is expected because the Docker `export`, `save`, and `checkpoint` commands operate differently and preserve varying amounts of run-time application state, as discussed earlier.

Figure 4 shows the tarball sizes for the different combinations `Application-Strategy`. For migration strategies `SFP` and `FCCP`, there are only minor variations in the tarball sizes for the applications `BA` and `USPA`. In both cases, `FCCP` leads to a less than 0.1% increment of the tarball size with respect to the `SFP` strategy.

The figure shows that for the applications `BA` and `USPA`, the strategies `SFP` and `FCCP` generate quite a similar tarball size because there is either no state preserved or very little state preserved, respectively. The increment of size is limited to about 3%. The `FullSP` migration strategy shows a noticeable

increase in tarball size for all applications. As an example, in the case of BA the tarball size increment given by FullSP over FCCP is equal to more than 10%. In general, SFP gives the smallest tarballs for all applications, while FullSP the largest ones. The maximum is observed with the application ASW. The observed increment with respect to FCCP is more than 19%.

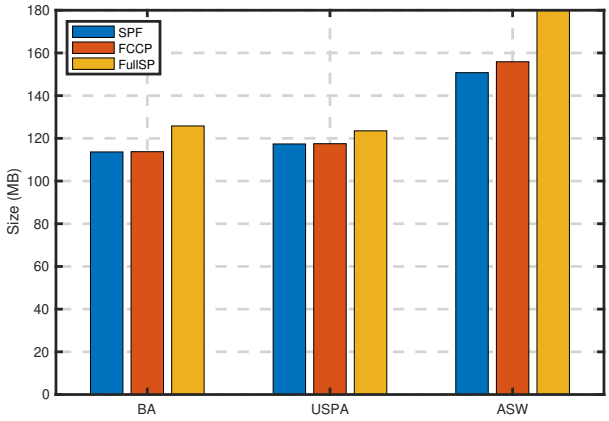

Fig. 4. Tarball sizes

B. Duration of Pre-relocation Phase

The tarball sizes are expected to have no significant impact on the service downtime. This result is because the tarballs are generated and transferred during the service pre-relocation phase. Of course, this holds only if the service pre-relocation starts at the right time. Therefore, the orchestrator needs to know the overall duration of the pre-relocation phase. This duration is the sum of the following components:

- Time to prepare the pre-relocation information at the Source MEH;
- Time to transfer the Tarball;
- Time spent on the Target MEH.

The Linux time utility was used to time all the operations discussed below. Because the exact duration varied by a few milliseconds every time the commands were run, the shown results are obtained by averaging the values observed in five different runs.

1) Time to prepare the pre-relocation information at the Source MEH: The estimation of this parameter depends on the migration strategy. In particular, in the case of SPF, this parameter is only due to the time taken by the Docker command export. Instead, for FCCP, it can be estimated by the sum of the times taken by the Docker commands commit and save. The sum of the time taken by the Docker commands commit, save, and checkpoint represents the estimation of this parameter in the case of FullSP. Figure 5 shows the results. In all scenarios, the strategy SPF is the fastest. This result is because only one Docker command needs to be run, and only the filesystem contents are preserved. On the contrary, with three different Docker commands and full state preservation, FullSP is the slowest.

In strategies FCCP and FullSP, the Docker operation commit, as mentioned earlier, temporarily freezes all the containers. The freeze periods observed during the commit command are relatively short and almost similar for all the applications. Indeed, the minimum duration is 178 ms for the BA, while 197 ms is the maximum observed for ASW.

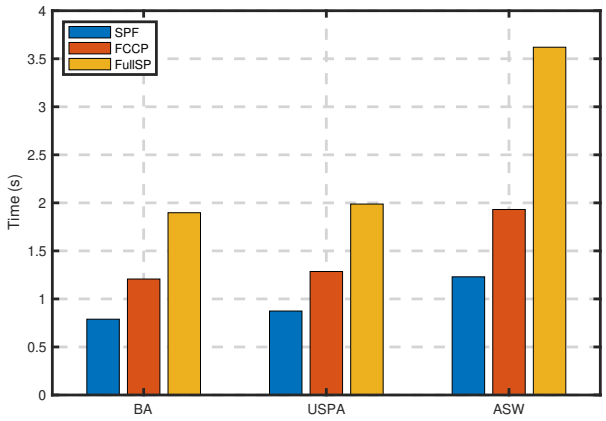

Fig. 5. Time to prepare the pre-relocation information at the Source MEH

2) Time to Transfer the Tarball: This parameter is equal to the time taken by the command scp to transfer the tarball from the source MEH to the target MEH. The observed values are approximately equal to the Tarball size divided by the link data rate, i.e. 1 Gbps or 100 Mbps.

3) Time spent on the Target MEH: In the case of SPF, this parameter corresponds to the time taken by the Docker command import. For the strategies FCCP and FullSP, the value of this parameter is related to the time taken by the Docker command load. As shown in figure 6, the results are identical for both strategies FCCP and FullSP because the checkpoint data is not used yet. Thus, it does not contribute to the time necessary to complete the pre-relocation operations at the target MEH.

SPF is again the fastest because it does not preserve any Docker layers information. Whereas FCCP and FullSP are much slower because they preserve information on about six Docker layers, each of which has to be restored individually.

C. Duration of Relocation Phase

Throughout the pre-relocation phase, in all three implementations of the testbed, the UE has largely uninterrupted access to the MEC application. In fact, the application is unaware of the phase. On the contrary, the relocation phase does introduce service downtimes, which must be measured and analysed. Also, the relocation phase can be analysed considering three different contributions, as the pre-relocation.

1) Time to prepare the relocation information at the Source MEH: In all three strategies, the following operations need to be run on the source MEH:

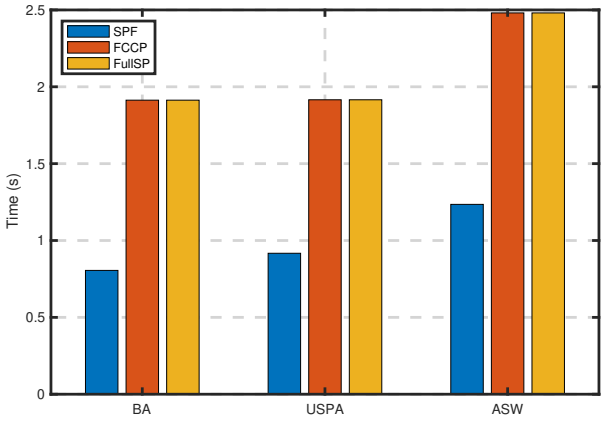

Fig. 6. Time spent on the target MEH during Pre-Relocation

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

In the strategy `FullSP`, the Docker operation `checkpoint` has to be additionally run to generate a new checkpoint. This operation has to be done before stopping the container.

For applications `BA` and `ASW`, the DVs are always empty. In the case of `BA`, the DV is empty because this application is completely stateless. It needs to store neither user nor application instance state. In the case of `ASW`, the DV is empty because all the state is being stored in the memory of the container.

The DV of `USPA` contains data. The amount of data is directly proportional to the duration of the pre-relocation phase. A side effect of this behaviour is that the network congestion increasing the time necessary to transfer the tarball from the source to the target MEH increases the size of the DV. Although mentioned, this relation has not been fully accounted for in the testbed presented in Campolo et al. [6]. Therefore the final results shown in this paper can be expected to be more realistic. Indeed, the experimental measurements carried out with the `USPA` indicate a DV size of about 80 KB with the 1 Gbps link, while size of about 110 KB is observed in the scenario with a 100 Mbps link.

The command `tar` is very quick, and the difference between the times to archive the two data volumes is, on average, less than 1 ms. Therefore, in the estimation of the total time spent on the source MEH, this contribution is assumed to be a constant.

Figure 7 shows the total time spent at the source MEH during the relocation phase. The strategies `SPF` and `FCCP` are the fastest. However, `FullSP` is consistently slower because of the additional checkpoint operation. Thus, `FullSP` implies that all applications take a markedly longer time. It is also worth noting how the strategies `SPF` and `FCCP` are fast for the application `ASW`, which does not store any state in the file system. This result is because these strategies ignore most of the state information that `ASW` needs in order to run correctly on the target MEH.

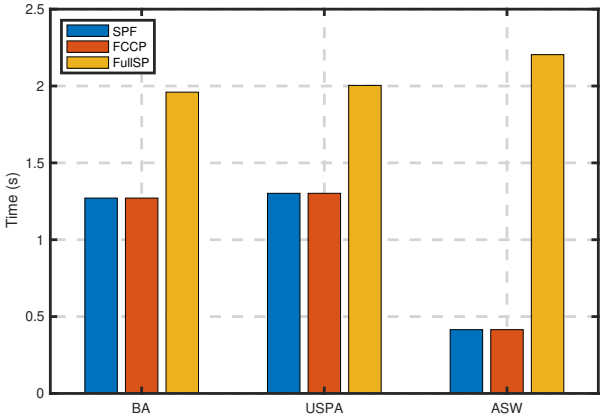

Fig. 7. Time to prepare the relocation information at the Source MEH

2) Time to Transfer Data: The tool `nc` is used to copy the DV's tarball from the source MEH to the target MEH. This action is necessary only for `USPA` because this application is the only one that uses the DV to store additional state while the pre-relocation is in progress.

In the case of `FullSP`, the utility `rsync` is additionally used to transfer the checkpoint files generated by the CRIU tool. This transfer is configured to use SSH to maintain data confidentiality. All applications generated at least 13 MB of checkpoint data, with `ASW` generating, on average, 16 MB. With the command `rsync`, it was noted that, on average, only 2.9 MB of updated data have to be transferred, resulting in an average speedup of 4.52x. In other words, compared to `scp`, the command `rsync` is about 4.52 times faster. Figure 8 shows the measured times for transferring additional user state and application state during the relocation phase in the scenario with a 100 Mbps link. The scenario with a 1 Gbps link shows results with times that are about one-tenth of those shown in the figure. For sake of simplicity, these results are not reported in the paper.

A second set of simulation is focused on the evaluation of the impact of the PL probability on this performance parameter. The chosen PL values consider the fact that usually this link uses a transport network infrastructure not involving wireless communications. Thus, the considered values are 0.1%, 1% and 5%. PL increases the time necessary for transferring the tarball in the pre-relocation phase, i.e. the amount of data shown in Figure 4.

In the case of `USPA` and `ASW`, the increment of the transferring time increases the amount of data acquired during the pre-relocation. These data will be transferred as additional user or/and application state during the relocation phase. The experimental results show that the time for transferring the tarball has a negligible increment (less than 4%) when the PL probability is 0.1% and 1%. On the contrary, with PL set to 5%, the results show an increment of the transferring time

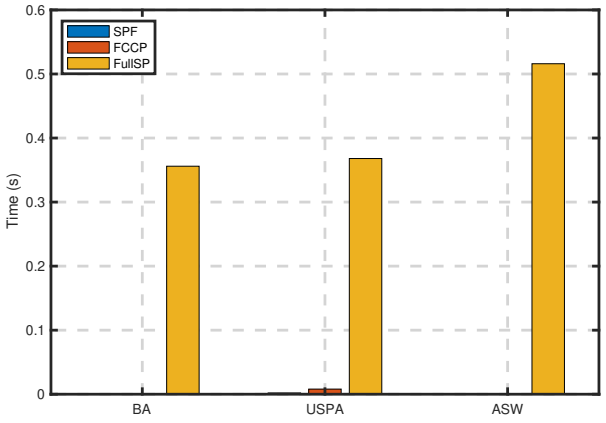

Fig. 8. Time to copy the additional user and application state to target MEH - Reference configuration: data rate set to 100 Mbps and no PL.

equal to 71.72% and 75.64% in the case of the smallest and the largest tarball size respectively. Table III highlights the impact of the PL on the time to copy the additional user and application state to the target MEH.

	FCCP-0%	FCCP-5%	FullSP-0%	FullSP-5%
BA (s)	0	0	0.356	1.310
USPA (s)	0.008	0.038	0.368	1.651
ASW (s)	0	0	0.516	2.112

TABLE III
IMPACT OF PL ON THE TIME TO COPY THE ADDITIONAL USER AND APPLICATION STATE TO THE TARGET MEH - DATA RATE EQUAL TO 100 MBPS

The results show that the time for copying the additional data remain below 0.04 s in the case of `FCCP` also for PL equal to 5%. This value is negligible respect to the time required for the other operations to complete in the relocation phase that are in the order of 1 s. On the contrary, the results obtained with `FullSP` highlight that with a PL probability of 5% the time for transferring the additional data is in the order of seconds, such as the time required for the other operations of the relocation phase. In other words, the packet loss probability of 5% duplicates the service downtime.

3) Time Spent on the Target MEC Host: When the strategies `SPF` and `FCCP` are used, the following operations need to be run on the target MEH:

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

A slightly different set of commands has to be run in the case of `FullSP`:

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

All compared strategies show to complete all their operations in less than 1 second, with `SPF` always being the fastest, and `FullSP` always being the slowest. Figure 9 gives an overview of the total amount of time spent performing operations on the target MEH.

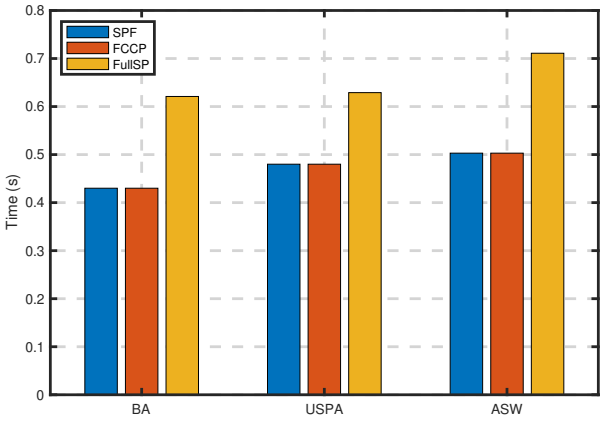

Fig. 9. Time spent on target MEH during relocation

VI. COMPARISON SUMMARY

The presented experimental analysis allows the comparison of the considered migration strategies using the following parameters:

- Service Downtime;
- Amount of preserved state;
- Viability.

A. Service Downtime

Service downtime is one of the most important performance parameters. All the migration strategies have been designed to minimize this parameter. The downtime starts as soon as the Docker command `stop` is run on the source MEH and ends only when the MEC application has started successfully on the target MEH.

Figure 10 shows the observed values of the service downtime in the case of the reference configuration (i.e. data rate equal to 100 Mbps and no packet loss). The figure points out that operations on the source MEH are responsible for a significant portion of the downtime. The copy operations contribute very little to the total service downtime, primarily because of the small amounts of data transferred during the relocation phase. As mentioned earlier, the bulk of the data is transferred during the pre-relocation phase. Lastly, the time spent performing operations on the target MEH is relatively short, with the strategy `FullSP` taking up the most time.

The link capacity, i.e. 100 Mbps or 1 Gbps in our testbed, has only a minor impact on service downtimes for strategies

SPF and FCCP. On the contrary, FullSP shows a non-negligible contribution to the downtime for all applications.

These results indicate SPF as the fastest, regardless of the application being migrated. Therefore, if minimizing service downtime is the only priority, SPF would be the best choice for application migration. It is worth noting that in the case of PL probability set to 5%, the values shown in Table III lead to almost double the service downtime due to the high copy operation in the relocation stage. For example, in the scenario "ASW, FullSP", the 5% of PL leads to a service downtime of 5.027 s against the 3.431 s observed when no loss is added to the 100 Mbps link between the two MEHs.

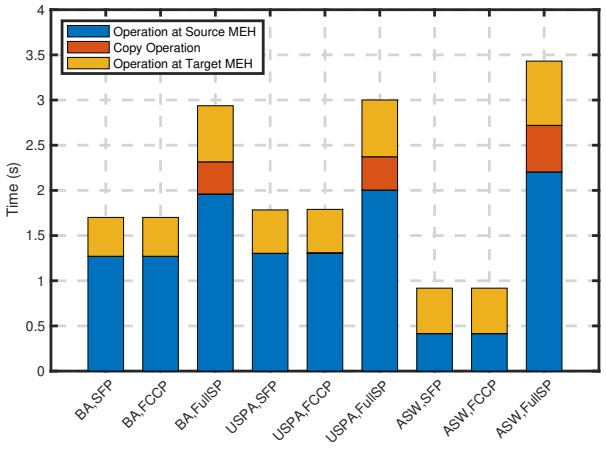

Fig. 10. Observed service downtimes - Scenario with data rate between MEHs equal to 100 Mbps and no PL

B. Amount of state preserved

As discussed earlier, there is no configuration information available on the target MEH when SPF is used. As a result, the Docker command run does not know how to start the MEC application. In the experimental tests, this problem has been overcome by manually passing the instruction CMD to the Docker command run.

In the cases of FCCP and FullSP, there is enough configuration information available to start all the MEC applications. But only FullSP can preserve the complete state of the application ASW. Therefore, when state preservation is the key criteria, FullSP represents the most suitable strategy.

C. Viability

For the sake of completeness, two different viability metrics are considered. These are the most important metrics because if a migration strategy is not viable, the other metrics either cannot be calculated or do not matter.

The first viability metric evaluates whether an application can start and run correctly on the target MEH after the migration. As expected, SPF is the least viable because even the application BA, which needs no state at all, cannot start without manual intervention. On the other hand, FullSP is the most viable because it can correctly run all applications. Table IV gives an overview of this viability metric.

Application Type	SFP	FCCP	FullSP
BA	Yes (with some manual inputs)	Yes	Yes
USPA	No	Yes	Yes
ASW	No	No	Yes

TABLE IV
VIABILITY OF MIGRATION STRATEGIES FOR THE V2X APPLICATIONS

The second viability metric evaluates whether the migration strategy can be completed within the time period when the application can remain in the target MEH without performing another migration. As observed during the experiments carried out using the city scenario simulated by the ETSI MEC Sandbox, this metric depends only on where the MEHs are located and does not help in comparing the migration strategies.

Indeed, in the scenario with a MEH associated with each 5G small cell PoA, the results show that all the migration strategies are largely viable in the case of the 1 Gbps link. When the MEHs are associated with zones, all the migration strategies are largely viable even in the case of a 100 Mbps link.

However, there are some paths a client could take in the city that have areas where none of the migration strategies is viable, regardless of link capacity and MEH placement. In these areas, the client switches between zones (or between PoAs) too quickly for a successful migration. Table V gives an overview of the amounts of time available for migration. The table shows that in the worst conditions, the available time to complete the application migration is about 1 s in the case the MEH is associated with each PoA or 1.8 s when associated with Zones. These values are lower than the time necessary to complete the migration, as observed during the experiments.

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

TABLE V
OVERVIEW OF AMOUNTS OF TIME AVAILABLE FOR MIGRATION

VII. CONCLUSIONS

Using the two-phase migration approach described in Campolo et al. [6], the paper presents the experimental comparison of three different migration strategies. The primary difference between the three strategies is the amount of preserved user-specific and application-specific state and configuration data. The comparison considers three applications consuming MEC services. These applications have different features in terms of state preservation requirements, DV usage and migration awareness. The experimental results show that the viability of

the migration strategy depends on the features of the application. In particular, the application requirements in terms of preservation of user-specific and application-specific state and configuration data represent an important feature impacting the viability. `SFP` is the fastest strategy, and it can be considered ideal for the migration of applications without requirements for state and data preservation. However, also in the simple `BA` some manual inputs are necessary for having the application running in the target MEH. On the opposite side, `FullSP` is found to be the slowest, but it can be used to migrate any application that is supported by the CRIU tool. In the middle, `FCCP` 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 `SFP`. Therefore, in scenarios where minimizing service downtime and preserving small amounts of application configuration data and user-specific state are both critically important, `FCCP` represents the most suitable strategy.

The experimental results provide insights into the migration of real-world MEC applications that are based on common frameworks and components such as NodeJS and Memcached. However, this study can be extended in the future by integrating into the MEC system a full-fledged orchestrator and other management-level entities that make the smart relocation feature possible. Furthermore, the considered MEC applications could be upgraded to use the publish-subscribe model instead of the request-response one, while accessing the MEC services. This upgrade would lead to reducing data rate and CPU resource consumption.

ACKNOWLEDGMENT

This work was supported by the Norwegian Research Council through the 5G-MODaNeI project (no. 308909), and by the Italian Ministry of Education and Research (MIUR) in the framework of the CrossLab project (Departments of Excellence).

REFERENCES

[1] ETSI ISG, "Multi-access edge computing(mec); study on mec support for v2x use cases," *ETSI GR MEC 022 V2.1.1*, 2018.

[2] ——, "Multi-access edge computing (mec); radio network information api," *ETSI GS MEC 012 V2.2.1*, 2022.

[3] ——, "Multi-access edge computing (mec); location api," *ETSI GS MEC 013 V2.2.1*, 2022.

[4] C. Quadri, V. Mancuso, M. A. Marsan, and G. P. Rossi, "Edge-based platoon control," *Computer Communications*, vol. 181, pp. 17–31, 2022.

[5] S. Dabbene, C. Lehmann, C. Campolo, A. Molinaro, and F. H. P. Fitzek, "A MEC-assisted Vehicle Platooning Control through Docker Containers," in *2020 IEEE 3rd Connected and Automated Vehicles Symposium (CAVS)*, 2020, pp. 1–6.

[6] C. Campolo, A. Iera, A. Molinaro, and G. Ruggeri, "MEC support for 5G-V2X use cases through Docker containers," in *2019 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2019, pp. 1–6.

[7] ETSI MEC, "MEC Sandbox," 2022, accessed March 16, 2022. [Online]. Available: https://try-mec.etsi.org/

[8] D. Namiot and M. Sneps-Sneppe, "On micro-services architecture," *International Journal of Open Information Technologies*, vol. 9, pp. 24–27, 2014.

[9] S. Wang, J. Xu, N. Zhang, and Y. Liu, "A survey on service migration in mobile edge computing," *IEEE Access*, vol. 6, pp. 23 511–23 528, 2018.

[10] K. Gillani and J.-H. Lee, "Comparison of linux virtual machines and containers for a service migration in 5g multi-access edge computing," *ICT Express*, vol. 6, no. 1, pp. 1–2, 2020.

[11] S. Shahryari, F. Tashtarian, and S.-A. Hosseini-Seno, "Copam: Cost-aware vm placement and migration for mobile services in multi-cloudlet environment: An sdn-based approach," *Computer Communications*, vol. 191, pp. 257–273, 2022.

[12] Y. Liao, L. Shou, Q. Yu, Q. Ai, and Q. Liu, "Joint offloading decision and resource allocation for mobile edge computing enabled networks," *Computer Communications*, vol. 154, pp. 361–369, 2020.

[13] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[14] A. Randazzo and I. 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)*. IEEE, 2019, pp. 209–214.

[15] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran, "A binary-compatible unikernel," in *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2019, pp. 59–73.

[16] P. Fondo-Ferreiro, A. Estévez-Caldas, R. Pérez-Vaz, F. Gil-Castiñeira, F. J. González-Castaño, S. Rodríguez-García, X. R. Sousa-Vázquez, D. López, and C. Guerrero, "Seamless multi-access edge computing application handover experiments," in *2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR)*, 2021, pp. 1–6.

[17] I. Farris, T. Taleb, H. Flinck, and A. Iera, "Providing ultra-short latency to user-centric 5G applications at the mobile network edge," *Transactions on Emerging Telecommunications Technologies*, vol. 29, no. 4, p. e3169, 2018.

[18] H. Liu, H. Jin, X. Liao, L. Hu, and C. 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*, 2009, pp. 101–110.

[19] S. Pickartz, N. Eiling, S. Lankes, L. Razik, and A. Monti, "Migrating Linux containers using CRIU," in *International Conference on High Performance Computing*. Springer, 2016, pp. 674–684.

[20] R. A. Addad, D. L. C. Dutra, M. Bagaa, T. Taleb, and H. Flinck, "Towards a fast service migration in 5g," in *2018 IEEE Conference on Standards for Communications and Networking (CSCN)*. IEEE, 2018, pp. 1–6.

[21] R. A. Addad, D. L. C. Dutra, T. Taleb, and H. Flinck, "AI-Based Network-Aware Service Function Chain Migration in 5G and Beyond Networks," *IEEE Transactions on Network and Service Management*, vol. 19, no. 1, pp. 472–484, 2022.

[22] R. Stoyanov and M. J. Kollingbaum, "Efficient live migration of linux containers," in *International Conference on High Performance Computing*. Springer, 2018, pp. 184–193.

[23] P. Bellavista, A. Corradi, L. Foschini, and D. Scotece, "Differentiated service/data migration for edge services leveraging container characteristics," *IEEE Access*, vol. 7, pp. 139 746–139 758, 2019.

[24] ETSI ISG, "Mobile edge computing (mec);end to end mobility aspects," *ETSI GR MEC 018 V1.1.1*, 2017.

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

[26] Y. Takagawa and K. Matsubara, "Yet another container migration on freebsd," in *AsiaBSDCon 2019 Proceedings*, 2019, pp. 97–102.

[27] A. Hathibelagal, "MEC testbed," https://github.com/hathibelagal-dev/MECTestbed, 2021.

[28] S.-J. Cha, S. H. Jeon, Y. J. Jeong, J. M. Kim, S. Jung, S. Pack *et al.*, "Boosting edge computing performance through heterogeneous manycore systems," in *2018 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 2018, pp. 922–924.

[29] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele, "Many-core key-value store," in *2011 International Green Computing Conference and Workshops*. IEEE, 2011, pp. 1–8.

[30] A. Soliman, *Getting Started with Memcached*. Packt Publishing Ltd, 2013.