This is a pre-print version. The paper has been published in the proceedings of 2022 IEEE 33rd Annual International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)
original DOI: 10.1109/PIMRC54779.2022.9977561

Enabling Application Relocation in ETSI MEC: A Container-Migration Approach

Francesco Barbarulo, Carlo Puliafito, Antonio Virdis, Enzo Mingozzi
Department of Information Engineering, *University of Pisa*, Pisa, Italy
{f.barbarulo@studenti., carlo.puliafito@ing., antonio.virdis@, enzo.mingozzi@}unipi.it

***Abstract*—ETSI MEC is a standard for edge computing which allows the execution of services - called MEC applications - on hosts in user proximity. One of the emerging concepts within ETSI MEC is that of MEC application relocation, i.e., the migration of a MEC application between edge hosts. ETSI MEC identifies several approaches to relocate a MEC application along with its internal state. However, some of these approaches devote the transfer of the application state to the application itself, whereas others rely on costly virtual-machine migration procedures. To overcome the above limitations, in this work we extend ETSI MEC to support MEC application relocation by exploiting container-migration technologies. We evaluate the performance of our implementation over a small-scale edge testbed, showing the overall benefits of the proposed approach.**

***Index Terms*—ETSI MEC, container migration, relocation**

I. Introduction

Edge computing provides end users with services in their proximity, by leveraging a geo-distributed infrastructure of micro data centres that are co-located with access networks or deployed close to them. Having computing power in proximity of the final users enables low latency and high bandwidth, which are needed by many emerging applications, as for example the Internet of Things and augmented/virtual reality. Given the great interest raised by edge computing, the European Telecommunications Standards Institute (ETSI) is working on creating a standardised, open, and multi-vendor edge-computing environment, which is known as Multi-access Edge Computing (MEC) [1].

In ETSI MEC, edge services are called *MEC applications*[1], which typically run as virtual machines (VMs). More recently, ETSI MEC has been investigating the support for alternative virtualisation technologies, such as containers [2]. Containers are indeed more lightweight and faster to boot up compared to VMs. This is mainly due to the fact that containers do not need their own entire operating system. However, on the one hand, in ETSI MEC vision: "containers are intended to be ephemeral and stateless; state (i.e. data that needs to live beyond the life of a container instance) is stored outside of a container" [2]. On the other hand, ETSI MEC considers MEC applications to be either stateless or stateful, as in the latter case they can locally maintain some user-related information. For example, a MEC application for augmented reality can statelessly analyse video frames one-by-one, or it can process a frame based on a buffer of previous frames, i.e., the state. Stateful MEC applications can be *statefully* relocated, i.e., migrated among MEC hosts along with their state [3] for several reasons. First, relocation is commonly used to maintain proximity between a moving user and her MEC application instance [4]. Indeed, user mobility is for example a key aspect of 3GPP 5G systems, wherein MEC is seen as a natural solution to enable edge computing. Moreover, 3GPP and ETSI are actively working together to enable mutual mobility support between 5G core and ETSI MEC management [5]. Second, MEC application relocation can be used to meet application preferences, such as the co-location with specific context information or services exposed by MEC hosts. Finally, it can be used to optimise the usage of MEC resources [6], e.g., to balance the load during high-load periods or to reduce the number of active hosts during low-load periods, thus enabling more dynamic power-saving mechanisms.

When relocating a MEC application, one must ensure *service continuity*, which involves two aspects. Firs, network connectivity between the client application running on the user device and the MEC application instance must be preserved after the latter has been relocated. ETSI MEC identifies different ways to do this, which are all up to the MEC system. One example is DNS support, which requires the client-side application to locally update the IP address associated to the MEC application instance by querying a DNS server. However, this solution does not fit highly-dynamic scenarios like mobility ones [7]. Another example is the reconfiguration of the underlying traffic-routing transparently to the application [6], e.g., through SDN-based techniques, which yet introduces further complexity in the network. Alternative methods exist in the literature, e.g., [8], which leverages on transport protocols to maintain active connections transparently to the application. A second aspect involved in MEC application relocation is that the state of the MEC application instance, which is called *user context* in ETSI MEC terminology, must be transferred to the target MEC host. The standard identifies several ways to do this, all of which present some limitations. An approach considers user context transfer as an application-level operation and therefore devoted as a responsibility of the application developer [6]. As a result, if the application does not implement mechanisms to transfer user context, stateful relocation of the MEC application is not possible. Another approach is to statefully migrate the whole VM [9]. While this approach is transparent to the application, it generates a user context having a large footprint, which may impair Quality of

[1]MEC applications can also consume services produced by other entities, such as other MEC applications or the MEC platform itself.

Service during relocation. For this reason, the latter approach is proposed only for relocation between a MEC system and an external cloud and left for further study [3].

The core idea of our work is to take the best out of the above solutions, considering MEC application instances as stateful containers and leveraging container-migration techniques to transfer user context, as container state, to a target MEC host. By doing this, the footprint of user context is significantly reduced compared to that of VMs [10], leading to improved service continuity for the final user. Furthermore, user-context transfer becomes a responsibility of the Virtualisation Infrastructure Manager (VIM), which is an element in the ETSI MEC architecture that manages the lifecycle of VMs or containers. The migration process becomes thus transparent to the application and has no longer to be handled by developers. We thus propose MEC application relocation based on container migration to become a built-in mechanism for managing resources in ETSI MEC systems. This mechanism can be indeed considered as an enabler for relocation in different scenarios, e.g., user mobility or resource optimisation.

The contribution of this work is therefore the following:

- an extension to ETSI MEC and its Application Programming Interfaces (APIs) that let it trigger, support, and coordinate container-based MEC application relocation in two scenarios, i.e., user mobility and resource shortage;
- a Proof-of-Concept (PoC) implementation, which validates our proposal and shows the benefits of container-based MEC application relocation in a mobility scenario.

A few existing works leverage container migration in the context of ETSI MEC [11], [12]. However, to the best of our knowledge, no work provides details on which ETSI MEC functionalities are used to enable container migration, nor describes how to extend ETSI MEC and its APIs in this direction. This work also differs from our previous one [13], as it presents extensions to ETSI MEC that make container migration a core mechanism for managing MEC resources in different scenarios, as triggered either by user mobility or resource shortage.

The rest of the paper is organised as follows. Section II provides a background on ETSI MEC reference architecture, highlighting the core elements that we consider in this work. Then, Section III discusses the design of our solution. Next, Section IV describes the PoC implementation of our proposal and reports the performance evaluation that we carried out over a small-scale edge testbed. Finally, Section V concludes the paper and outlines the future work.

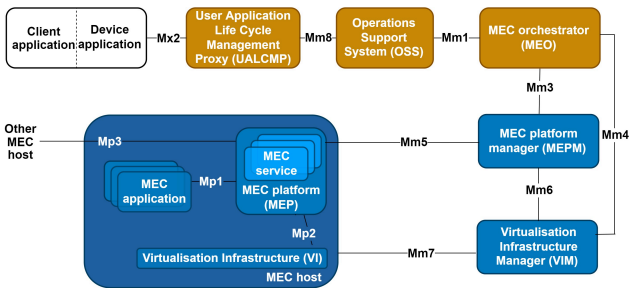

Fig. 1. Simplified ETSI MEC architecture (only the reference points considered in the paper are shown).

II. Reference Architecture of ETSI MEC

In this section we provide a brief description of the ETSI MEC reference architecture, introducing the functional elements and the reference points - i.e., the interfaces among elements - involved in this work. The reference architecture of ETSI MEC is defined in details in [9] and depicted, in a simplified version, in Fig. 1. Colours in figure represent different levels in which elements operate: white elements run on User Equipments (UEs); blue elements are deployed on MEC hosts, whereas orange elements operate at system level.

UEs are end devices, such as smartphones, IoT devices, or vehicles. The *client application* is the application logic running on the UE, which communicates with a MEC application running on a MEC host in its proximity. The definition of the communication interface between client and MEC application is not covered by ETSI MEC. Since the client application is MEC-unaware, it needs to be complemented by a *device application* that interacts with the MEC system for control operations, such as MEC-application instantiation request.

On MEC hosts, MEC applications run on a *Virtualisation Infrastructure* (VI) and can be packaged as VMs or containers. MEC applications can interact with the *MEC platform* (MEP) to consume and/or provide *MEC services*. The standard defines a number of MEC services, such as the RNIS, which provides radio network information to MEC applications. Yet, new services can be designed by third parties and onboarded in an ETSI MEC system. In the reference architecture, two elements are involved in the host-level management. The *MEC Platform Manager* (MEPM) manages the life cycle of MEC applications, including informing the system-level management of application-related events, and handles aspects such as traffic rules and authorisations to MEC services. The VIM instead manages compute and network resources of the VI and reports performance and faults information about the virtualised resources to the MEPM.

At system level, the *User Application Life Cycle Management Proxy* (UALCMP) is the entry point to the MEC system for any device application. Requests coming from a device application first need to be granted by the *Operations Support System* (OSS) and are then forwarded to the *MEC orchestrator* (MEO). The MEO maintains a view of the whole system and makes core decisions such as selecting where to instantiate a MEC application, based on aspects like latency, available resources, and available MEC services.

III. Extending ETSI MEC

MEC application relocation is under investigation by ETSI MEC and is specified in [6]. As we anticipated in Section I, the standard defines several approaches to statefully relocate a MEC application. However, those approaches either require direct intervention from the application itself or - in case

of VMs - can produce a big-sized user context, which may cause MEC application relocation to take a long time, thus impairing user experience. To overcome the above limitations, in this work we assume MEC application instances as stateful containers and consider user context as container state. As a result, user-context transfer is no longer devoted to the application but becomes a functionality of the VIM, which can leverage available container-migration technologies.

Container migration is being used as an effective relocation technique in the context of microservices. The de-facto standard for container migration is Checkpoint/Restore In Userspace (*CRIU*[2]), and migration support is already included as part of the Docker runtime and as an experimental feature of Kubernetes. However, relocating MEC applications using container migration requires also extensions to ETSI MEC to let it trigger, support, and coordinate the proposed approach. To this purpose, in this section we describe an extension of the APIs over reference points Mm1, Mm3 and Mp1 and the modifications in the information flow to include interaction with the VIM for what concerns user-context transfer.

In the following, we will use the terms *source* and *target* when referring to the MEC hosts handling the MEC application instances execution respectively *before* and *after* the migration occurs. The proposed procedure consists in the following steps: (i) create a container at the target MEC host; (ii) checkpoint (i.e., save) user context as container state and transfer it to the target MEC host; and (iii) restore container execution at the target MEC host based on the transferred state.

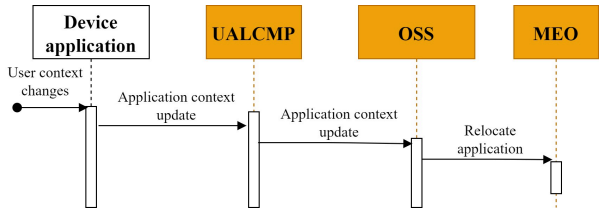

Fig. 2. MEC application relocation triggered by end-device mobility.

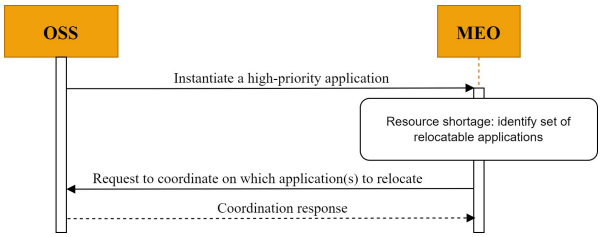

Fig. 3. MEC application relocation triggered by resource shortage.

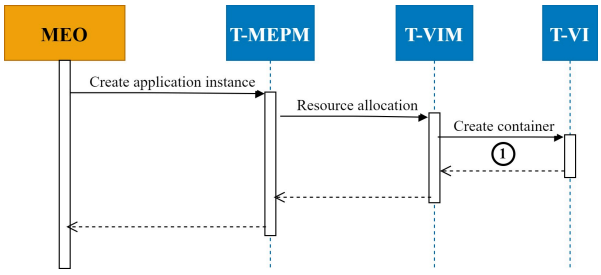

Fig. 4. Create container at the target MEC host.

[2]criu.org/Main_Page

A. Initiating relocation and creating the container

As ETSI MEC states in [6], MEC application relocation can be initiated by different functional elements and for different purposes, here we focus on two of them: one handling user mobility and one resource shortage. In the first scenario, when the user moves, it may be indeed necessary to relocate the MEC application instance to a MEC host that is closer to the new location of the user. In the second scenario, instead, relocation is initiated due to resource shortage. Specifically, when a MEC host does not have enough resources for the instantiation of a high-priority MEC application, one or more low-priority ones can be relocated to free up resources.

For what concerns the first scenario, ETSI MEC specification [6] reports a possible example wherein relocation is initiated based on notifications provided by the Radio Network Information Service (RNIS), a MEC service which - among other things - informs the MEC system when the UE changes its serving cell. Alternatively, we propose MEC application relocation to be initiated by the device application. As this is already in charge of requesting instantiation or termination of a MEC application, we believe that it could also be responsible of asking for relocation. This assumption is in line with [9], which considers this possibility, even though only for relocation to external clouds, which is left for further study.

The procedure then works as follows: in compliance with [14], the instantiation of a new MEC application creates an *application context*. This is a data structure maintained by the MEC system, which contains information such as the requirements on the geographic area of interest for the user, the MEC application identifier, the MEC application address and the *callback reference*, which is the address exposed by the device application to receive notifications from the MEC system. As shown in Fig. 2, our relocation procedure is initiated when the user context changes, i.e., any part of the above information changes. This can be due to user mobility, to a vertical handover (i.e., one where the access technology changes, such as from 5G to Wi-Fi or vice versa) or to an horizontal handover in case of Wi-Fi-based access. As a result, the device application issues an application context update [14] to the UALCMP over reference point Mx2. The UALCMP forwards this update to the OSS. This grants the request and then contacts the MEO over Mm1. Using the API over Mm1 [15], the OSS can ask for starting or stopping a MEC application instance by issuing a POST request to `/app_instances/{appInstanceId}/operate`. The request body is `OperateAppRequest`, whose attribute `changeStateTo` can be changed to either `started` or `stopped` for the purpose. We extended the API to include

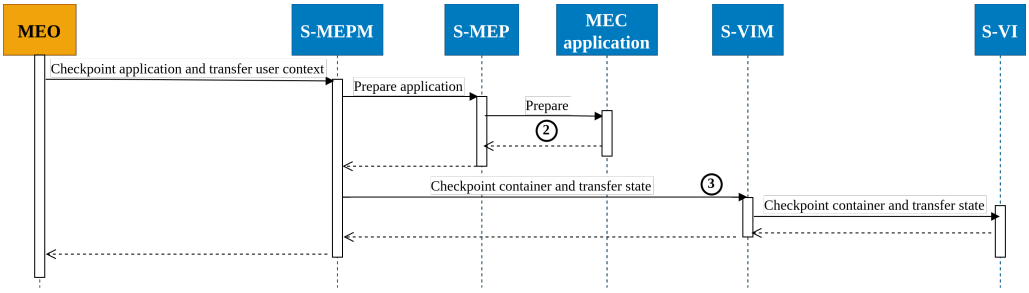

Fig. 5. Checkpoint and transfer user context.

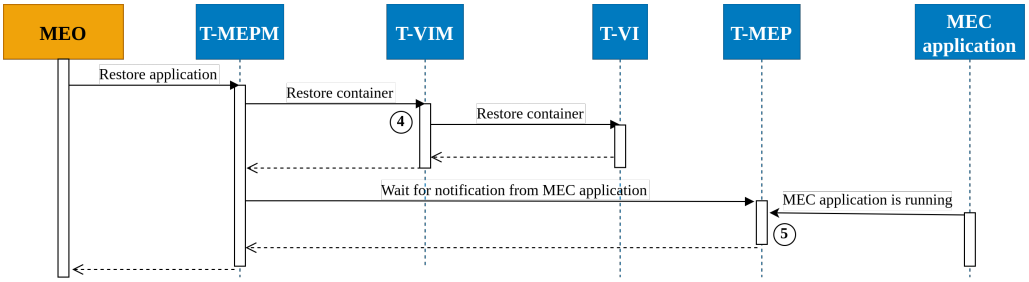

Fig. 6. Restore user context.

also the value `relocated`, hence allowing the OSS to request MEC application relocation. As a result, the MEO handles the request to decide whether relocation is actually needed and eventually selects a target MEC host. Different policies can be used by the MEO for the purpose; however, these are out of the scope of this work.

The second scenario is that of relocation due to resource shortage [15]: whenever a request to instantiate a MEC application could not be executed due to lack of resources in the virtualised environment, a lower-priority MEC application instance can be terminated to free resources. Termination is indeed the only option when user-context transfer is either left to the application, which might not implement it, or performed as (a costly) VM migration. Container migration can be instead a key enabler of relocation in this specific scenario. Fig. 3 illustrates the procedure to initiate relocation in this context. When the OSS requests to instantiate a high-priority MEC application, the MEO selects a MEC host and verifies if resources are sufficient. If they are not, the MEO coordinates with the OSS to decide which (if any) low-priority MEC application instance(s) to terminate or relocate. Coordination takes place through a dedicated interface over Mm1. We extended the API to allow either termination or relocation in case of resource shortage. Specifically, we modified all the exchanged data structures and their attributes by replacing the term `Termination` with a more generic `Management` and by including `RELOCATION` as one of the `ManagementOptions`.

As shown in Fig. 4, the relocation procedure starts with the MEO contacting the target MEPM over reference point Mm3 and requesting the creation of a new MEC application instance, in line with the specifications in [15]. The target MEPM then asks the target VIM to allocate resources (compute, storage, and network) for the new container, and the VIM instructs the target VI for the purpose.

B. Managing user context as container state

Once the container is created on the target MEC host, checkpoint and transfer of the user context can start, as shown in Fig. 5. To let the MEO start this procedure, we extended the API over Mm3 similarly to the extension made to Mm1 and described in Section III-A. Specifically, we included the `checkpointed` value for the `changeStateTo` attribute of `OperateAppRequest` to start the operation of checkpointing and transfer. Besides, we included the `targetHost` attribute to indicate where to relocate the MEC application.

When the source MEPM receives the request from the MEO, it informs the MEC application instance to prepare for checkpoint. Even though this preparation phase is not mandatory, it can be used: (i) to let the MEC application instance finish memory-intensive computation before checkpointing, thus reducing memory footprint; (ii) to let MEC and client applications coordinate, as we describe in our PoC implementation (see Section IV). The preparation phase requires an extension to Mp1 API [16] and works as follows. When it starts running on a MEC host, a MEC application instance can subscribe to a notification to prepare for user-context transfer. This communication consists in creating a `AppMigrationNotificationSubscription` over Mp1 by performing a POST request to `/applications/{appInstanceId}/subscriptions`. Then, when user

context transfer is to be performed, the source MEP checks if a subscription exists for the MEC application instance. If it does, it sends an `AppMigrationNotification` to the MEC application instance. The MEC application instance in turn can perform its preparation procedure, if it designed to do so. Next, it notifies the source MEP that preparation has ended and checkpoint can start, by sending an `AppMigrationConfirmation` as a POST request to `/applications/{appInstanceId}/confirm_migration`. Finally, the source MEP forwards this confirmation to the source MEPM. If instead the application is relocation-agnostic, the source MEP immediately returns the control to the source MEPM, thus keeping the overall relocation procedure transparent to the MEC application. The source MEPM can now request the source VIM to checkpoint the container state as a collection of files on disk and transfer this state to the target MEC host. To this aim, it provides the identifier of the container and the address of the target MEC host. The source VIM interfaces with the source VI to perform the actions. After checkpoint starts, the container that encapsulates the MEC application becomes not available to serve client requests.

Fig. 6 shows the procedure that allows restoring the MEC application instance at the target MEC host. This procedure begins after the MEO is informed that the state of the MEC application instance has been successfully transferred to the target MEC host. The MEO then contacts the target MEPM by using our extended APIs. Specifically, the MEO changes the value of `changeStateTo` to `restored`. The MEPM then asks the target VIM to restore the container through the target VI, using the container that was previously created (see end of Fig. 4). However, this operation differs from a simple container start, as now the container is launched along with the state that has been transferred from source (see end of Fig. 5). Next, the target MEPM is notified that the container is restored. Therefore, the MEC application startup phase starts. This phase is specified in [16] and is used when a MEC application instance is started for the first time. We consider it also as a step at the end of MEC application relocation, to ensure that the MEC application instance is running properly after instantiation/relocation. This is done by making the target MEP wait for a notification from the MEC application instance. This indeed issues a POST request to `/applications/{appInstanceId}/confirm_ready` over Mp1, specifying an `AppReadyConfirmation` as request body. This confirms that relocation was successful.

IV. Implementation and Performance Evaluation

In this section, we describe our PoC implementation and then report a performance evaluation of our solution, carried out over a small-scale edge testbed. The logic of the MEC-system modules is written in Python (v3.6+) and implements the modified procedures described in the previous sections. Docker is used as the VI, CRIU for container checkpoint and restore, and rsync to transfer the user context to the target MEC host. Interactions between the VIM and VI are implemented through Ansible, an ease-of-use open-source automation tool that allows system configuration and orchestration of advanced tasks. Ansible modules to control the docker runtime are taken from the *community.docker* collection. The modules for container checkpoint and restore, instead, are not available and we developed them for the purpose.

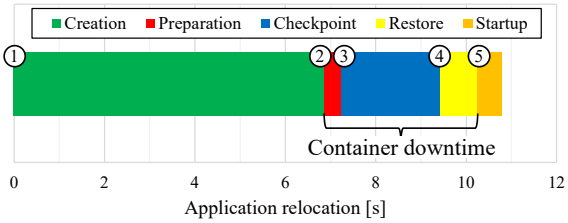

Fig. 7. Breakdown of MEC application relocation.

To maintain network connectivity between the client application and the MEC application instance after migration, we rely on a custom extension of the QUIC transport-layer protocol [17]. This extension leverages custom QUIC frames to let the server inform the client of an imminent relocation and of the new server IP address after relocation. Further details can be found in [8]. We integrated this extension within ETSI MEC by letting the QUIC-based procedures be triggered in the preparation phase, which is described in Section III-B.

The testbed used for the evaluation includes: two MEC hosts, one node for the system-level management and one for the end device. The MEC hosts and the management node are mini PCs equipped with an Intel Celeron Quad-Core at 1.99GHz, 8GB RAM, 64GB SSD and Ubuntu 18.04.4 (kernel 5.6.0). The mini PCs communicate over a switched Ethernet network having a maximum throughput of $100\,\mathrm{Mbps}$. To replicate realistic network conditions in our testbed [18], we leverage `tc-netem` to emulate a RTT of $120\,\mathrm{ms}$ between the two MEC hosts. Moreover, both the MEC hosts are configured as Wi-Fi access points providing connectivity to the end device. The latter is a Raspberry Pi 4, with a Quad core Cortex-A72 (ARM v8) at 1.5GHz, 4GB RAM, and 64GB SD, running Raspberry Pi OS 10. The wireless connection between the end device and the MEC hosts has an average RTT of $9.557 \pm 3.304\,\mathrm{ms}$ and a packet loss of 15%. For what concerns the application, we consider a simple client-server RESTful interaction over QUIC, wherein the client periodically issues POST requests to the server and the latter responds with an *echo* message. When encapsulated within a container as a MEC application, our server has an average memory footprint of $60\,\mathrm{MB}$. We performed ten independent replicas of each test case for statistical soundness.

In the first test, we measured the duration of MEC application relocation, which is shown in Fig. 7. The tags reported in figure match with the numbers reported in Figs. 4, 5, and 6 and indicate the beginning of a new phase of MEC application relocation. We highlight that the results reported in Fig. 7 do not include the overheads due to Ansible. As reported in Fig.

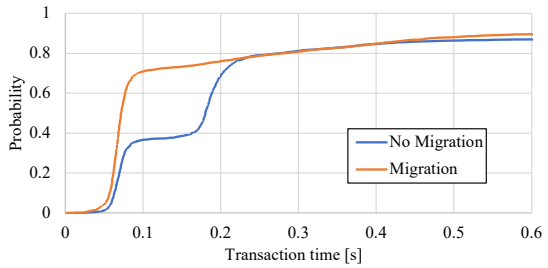

Fig. 8. MEC application relocation for a 10 minutes handover frequency.

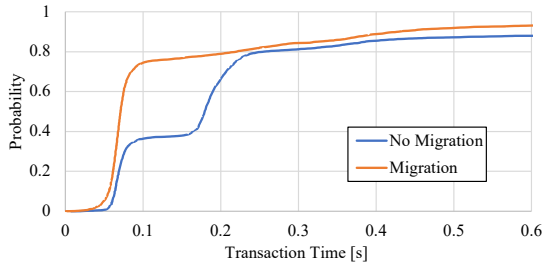

Fig. 9. MEC application relocation for a 30 minutes handover frequency.

7, the overall relocation lasts for slightly less than 11 s. Yet, *container downtime* (i.e., the time interval during which the container is not running on any MEC host) lasts about 3 s.

The second test aims at evaluating the benefits of the relocation procedure in a mobility scenario. In each run of the test, which lasts two hours, the end device performs periodic handovers between the Wi-Fi networks exposed by the two MEC hosts, and the MEC application instance is relocated accordingly. We consider two handover frequencies - and consequently two relocation frequencies - respectively 10 and 30 minutes. We compare the performance of our approach against a baseline wherein Wi-Fi handover is performed but relocation is not. Fig. 8 and Fig. 9 show the ECDF of the transaction time for the 10- and 30-minutes frequencies, respectively. We define transaction time as the time between the beginning of a request from the client and the reception of the corresponding response sent by the server. As shown, about 70% (for the 10-minutes frequency) and nearly 80% (for the 30-minutes frequency) of the transactions take less than 0.1 s when relocation is performed, demonstrating the benefits of maintaining proximity to the MEC application instance. When instead no relocation is performed, the ECDF shows a bi-modal behaviour, as the end device moves closer and farther from the MEC host where the MEC application instance runs. To conclude, in all scenarios ∼10% of requests experience transaction times above 0.6 s, which is coherent with the Wi-Fi packet loss of 15% experienced in the testbed environment.

V. Conclusions and Future Work

In this work, we presented an extensions to ETSI MEC to support stateful MEC application relocation by leveraging container-migration technologies. Our approach lets the relocation procedure be fully transparent to the application and to the developer. We demonstrated through experimentation over a small-scale edge testbed that the proposed approach can improve performance in a mobility scenario, compared to the case where no relocation is performed. As future work, we plan to evaluate our solution under multi-device and multi-application scenarios.

Acknowledgements

This work was partially supported by the Italian Ministry of Education and Research (MIUR) in the framework of the CrossLab project (Departments of Excellence) and by the European Commission through the H2020 project Hexa-X (Grant Agreement no. 101015956).

References

[1] ETSI, "Multi-access Edge Computing (MEC)," Mar. 2019, etsi.org/technologies/multi-access-edge-computing.

[2] ETSI, "Multi-access Edge Computing (MEC); Study on MEC support for alternative virtualization technologies," Nov. 2019, etsi.org/deliver/etsi_gr/MEC/001_099/027/02.01.01_60/gr_MEC027v020101p.pdf.

[3] ——, "Multi-access Edge Computing (MEC); Phase 2: Use Cases and Requirements," Jan. 2022, etsi.org/deliver/etsi_gs/MEC/001_099/002/02.02.01_60/gs_MEC002v020201p.pdf.

[4] C. Puliafito, E. Mingozzi, and G. Anastasi, "Fog computing for the internet of mobile things: Issues and challenges," in *IEEE SMARTCOMP*, 2017, pp. 1–6.

[5] ETSI, "Harmonizing standards for edge computing—A synergized architecture leveraging ETSI ISG MEC and 3GPP specifications," 2020, etsi.org/images/files/ETSIWhitePapers/ETSI_wp36_Harmonizing-standards-for-edge-computing.pdf.

[6] ——, "Multi-access Edge Computing (MEC); Application mobility service API," Feb. 2022, etsi.org/deliver/etsi_gs/MEC/001_099/021/02.02.01_60/gs_MEC021v020201p.pdf.

[7] ——, "Enhanced DNS support towards distributed MEC environment," Sep. 2020, etsi.org/images/files/ETSIWhitePapers/etsi-wp39-Enhanced-DNS-Support-towards-Distributed-MEC-Environment.pdf.

[8] C. Puliafito, L. Conforti, A. Virdis, and E. Mingozzi, "Server-side QUIC connection migration to support microservice deployment at the edge," *Elsevier Pervasive and Mobile Computing*, vol. 83, Jul. 2022.

[9] ETSI, "Multi-access Edge Computing (MEC); Framework and reference architecture," Mar. 2022, etsi.org/deliver/etsi_gs/MEC/001_099/003/03.01.01_60/gs_MEC003v030101p.pdf.

[10] S. Ramanathan, K. Kondepu, M. Razo, M. Tacca, L. Valcarenghi, and A. Fumagalli, "Live migration of virtual machine and container based mobile core network components: A comprehensive study," *IEEE Access*, vol. 9, pp. 105 082–105 100, 2021.

[11] M. V. Ngo, T. Luo, H. T. Hoang, and T. Q. Ouek, "Coordinated container migration and base station handover in mobile edge computing," in *GLOBECOM 2020*, 2020, pp. 1–6.

[12] C. Campolo, A. Iera, A. Molinaro, and G. Ruggeri, "MEC support for 5g-v2x use cases through docker containers," in *IEEE WCNC*, 2019.

[13] F. Barbarulo, C. Puliafito, A. Virdis, and E. Mingozzi, "Extending ETSI MEC towards stateful application relocation based on container migration," in *IEEE WoWMoM*, Jun. 2022, pp. 367–376.

[14] ETSI, "Multi-access Edge Computing (MEC); Device application interface," Apr. 2020, etsi.org/deliver/etsi_gs/MEC/001_099/016/02.02.01_60/gs_MEC016v020201p.pdf.

[15] ——, "Multi-access Edge Computing (MEC); MEC Management; Part 2: Application lifecycle, rules and requirements management," Feb. 2022, etsi.org/deliver/etsi_gs/MEC/001_099/01002/02.02.01_60/gs_MEC01002v020201p.pdf.

[16] ——, "Multi-access Edge Computing (MEC); Edge platform application enablement," Dec. 2020, etsi.org/deliver/etsi_gs/MEC/001_099/011/02.02.01_60/gs_MEC011v020201p.pdf.

[17] J. Iyengar and M. Thomson, "QUIC: A UDP-based multiplexed and secure transport," May 2021, datatracker.ietf.org/doc/html/rfc9000.

[18] C. Puliafito, C. Vallati, E. Mingozzi, G. Merlino, F. Longo, and A. Puliafito, "Container migration in the fog: A performance evaluation," *Sensors*, vol. 19, no. 7, 2019.