POLITECNICO DI TORINO
Repository ISTITUZIONALE

Design and Implementation of Microservice Migration at the Edge

Original
Design and Implementation of Microservice Migration at the Edge / Yu, Y., Calagna, A., Giaccone, P., Chiasserini, C.F.. - ELETTRONICO. - (2024). (IEEE WCNC 2024 Dubai (United Arab Emirates) 21-24 April 2024) [10.1109/WCNC57260.2024.10570769].

Availability:
This version is available at: 11583/2984710 since: 2024-05-18T10:30:36Z

Publisher:
IEEE

Published
DOI:10.1109/WCNC57260.2024.10570769

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2024 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

27 November 2026

Design and Implementation of Microservice Migration at the Edge

Yenchia Yu, Antonio Calagna, Paolo Giaccone, Carla Fabiana Chiasserini
Politecnico di Torino, Torino, Italy

***Abstract*—Stateful migration has emerged as the dominant technology to support microservice mobility at the network edge while meeting the end users' QoE requirements. In this context, our work addresses the two pivotal challenges of implementing and orchestrating the migration process. We first introduce a novel orchestration framework that efficiently realizes stateful migration and effectively orchestrates the migration process by fulfilling both network and application KPI targets. Then, through experimental validation using realistic microservices, we show that our solution improves migration performance, yielding up to 80% decrease of the migration downtime with respect to the state of the art. Finally, we demonstrate that our framework can be exploited to successfully address critical scenarios featuring latency-sensitive microservices and strict user QoE requirements.**

***Index Terms*—Edge computing, Service migration, Orchestration, Mobile networks**

I. Introduction

Edge computing has been acknowledged as the state-of-the-art paradigm to bring applications closer to the mobile end users. To fully exploit the benefits of edge computing architectures, applications are increasingly designed in the form of connected microservices (MSs), taking advantage of the lightweight container virtualization technology. In this context, MS migration has gathered momentum as the key technology to ensure continuous proximity of latency-sensitive and bandwidth-consuming MSs with mobile end users.

In this paper, we focus on *stateful* migration, which is used whenever keeping track of an MS state is essential to guarantee service continuity. In fact, despite the current trend favouring the development of stateless MSs, stateful MSs are still extremely common due to the complexity in refactoring legacy monolithic applications [1]. Furthermore, according to service-oriented architecture patterns [2], some essential stateful utility services will always be required, even if stateless service implementation will become dominant.

Existing issues. Although migration represents a powerful tool to let MSs "follow" users as they move so that latency requirements can still be met, in practice, some service disruption during a migration process is unavoidable and must be accounted for. This is because (i) stateful container migration techniques require freezing the MS state, and (ii) the network connection between the server running the MS and the mobile end users has to be migrated, along with the containerized MS. Further, as discussed in detail in Sec. II, no existing migration framework enables an effective and efficient implementation of the stateful MS migration process at edge scale. We fill this gap by proposing a Migration Orchestration framework for microServices at the Edge (MOSE), designed to attain stateful migration of latency-critical edge MSs.

Technical challenges. Implementing an effective and efficient migration framework at the edge is challenging since:

(i) the duration of service disruption due to the migration process depends on a number of factors, such as coding optimization, communication protocols, and limited nodes' computational capabilities, thus its minimization is not trivial;

(ii) upon statefully migrating an MS, the established connection thereof with the mobile end user needs to be preserved;

(iii) the stateful migration process needs to be independent of the specific MS and the underlying edge technology;

(iv) to effectively actuate and configure migration, it has to be envisioned an orchestrator that can collect and aggregate all relevant metrics. To the best of our knowledge, our work is the first to jointly tackle all the above aspects.

Novelty. The main contributions of this work are as follows:

- We present MOSE, which implements stateful MS migration at the edge and orchestrates the migration process to ensure minimal impact on the user's QoE (Sec. IV). Specifically, MOSE (i) enables the implementation of stateful migration of a generic MS at edge scale, while (ii) preserving its connection with the mobile end user using our COAT solution [3]. Importantly, (iii) its agnostic design enables a seamless and efficient integration on top of already existing edge platforms, (iv) it features several optimizations of the migration workflow, thus minimizing the experienced service disruption duration, and (v) leveraging the PAM model [4], it effectively configures the migration process, so that the target migration KPIs and the vertical's objectives are met, even in the presence of strict reliability and robustness requirements.
- We validate MOSE using realistic MSs and, by experimentally comparing MOSE to our previous setup in [3], we show how it greatly reduces service disruption thanks to an effective migration orchestration (Sec. V).

Before introducing and validating MOSE and showing its performance, we discuss some relevant related work while highlighting the novelty of our study in Sec. II and we provide some preliminaries in Sec. III.

This work was supported by Leonardo SpA, the EU under the Italian NRRP of NextGenerationEU, partnership on "Telecommunications of the Future" (PE00000001 - program "RESTART"), and the EC through Grant No. 101095890 (PREDICT-6G project) and Grant No. 101095363 (ADROIT6G project).

II. Related Work

Our work relates to two main research areas: stateful migration of containers and migration of service-user connections.

Relevant examples of works in the first area include [5], which gives an overview of current container migration techniques along with their fundamental metrics, and [6], which presents a promising application of stateful migration by using CRIU. However, in spite of the large body of work in the field, few studies have addressed the implementation of a migration framework at the network edge. Among these, [7] proposes a solution for mobile service continuity in edge-enabled WiFi networks and addresses service disruption minimization using a real-time kernel. [8], instead, focuses on the components of the mobile core network and demonstrates that container PreCopy outperforms other migration strategies and virtualization technologies. A proof-of-concept orchestration architecture is introduced in [9], for improving fault-tolerance by leveraging container migration. Further, [10] formulates an optimization problem that aims at meeting QoS requirements, while relocating edge applications.

As for connection migration, existing solutions are mostly application specific [11], and based on kernel [12] or protocol [13], [14] customization. It follows that integrating them with the container virtualization technology is impractical.

Novelty. MOSE enables an effective orchestration of the MS migration at the network edge while preserving the connection between the MS and the mobile end users. It achieves this goal by leveraging and effectively combining (i) the PAM model we introduced in [4] to configure the migration process, and (ii) our overlay network solution [3] to enable connection migration. MOSE builds upon the aforementioned solutions creating a novel, full-fledged algorithmic framework and technological solution that, unlike the existing alternatives, *minimizes service disruption in a way that is application independent and requires neither dedicated protocol nor modifications to the kernel or the application source code*.

III. Preliminaries

Stateful container migration enables the relocation across edge hosts of containerized MSs whose internal state must be migrated to ensure service continuity. The essential off-the-shelf tools to implement it are CRIU and Podman. *CRIU* (https://criu.org) is the key tool from the process viewpoint, which implements (i) the checkpoint procedure that freezes a running process and encapsulates it into an image, and (ii) the restore procedure that creates a new process and restores its state by using the checkpoint image. *Podman* (https://podman.io) is instead an open-source tool to develop, manage, and run containers and pods. Among all off-the-shelf container engines (e.g., Docker), it features the strongest integration with CRIU.

Leveraging both CRIU and Podman, different stateful migration strategies can be defined. The simplest one is *Cold Migration*, consisting of the following steps: (1) creation of a checkpoint image at the source host, (2) transfer of such image from source to destination host, (3) restoration of the container at the destination host. Throughout these steps, the MS needs to be stopped at the source host while it is yet to be restored at destination host, thus causing service disruption – commonly referred to as "downtime" (T^{down}). To minimize the service downtime, it has been envisioned the *Iterative PreCopy* strategy, which draws on the MS *dirty-page rate* concept, i.e., the number of memory pages a MS modifies per time unit. This strategy consists of: (i) the iterative transfer of dirty pages to the destination host while the MS is still running at the source, (ii) a *Stop&Copy* stage, during which the MS is stopped at the source host and the remaining dirty pages are transferred to the destination host, where the MS will be eventually restored. Hence, by minimizing the amount of data to be transferred over the network, this approach allows trading a longer *total migration duration* (T^{mig}) for a shorter downtime, at the cost of an increased traffic burstiness.

COAT [3] is a network solution that, independently from the specific MS, preserves the connection between the MS and the mobile users during stateful migration. Using Open vSwitch (OvS) (www.openvswitch.org), it defines a logical *overlay network* in which the behavior of the virtual switches (e.g., the forwarding rules) can be easily defined or changed through the OpenFlow protocol. To integrate COAT with the traditional migration process, we enhanced the Stop&Copy stage, through three additional steps: *(i)* network namespace clearing, preventing network configuration conflicts in the following steps; *(ii)* re-creation of the network namespace at the destination to match the original one, so that the later container restore procedure can successfully take place; *(iii)* update of the connection network flow by redirecting it toward the new network namespace.

The PAM model [4] provides an analytical expression of the migration duration and downtime, (i) independently from the specific MS, (ii) encompassing both the traditional and the COAT migration process, and (iii) accounting for the processing time overhead due to the migration tools. We use PAM to configure the migration process, so as to fulfill the migration KPI targets and the vertical's objective.

IV. Orchestration of MS Migration

A. Overview of the MOSE solution

We start by introducing a simple, yet relevant, MS migration scenario, as depicted in Fig. 1. The example scenario includes an Unmanned Aerial Vehicle (UAV) as mobile device, connecting to different 5G base stations (gNBs) as it flies over the area of interest. Due to the UAV's limited computational resources, some critical functions, such as flight control with collision avoidance, must be hosted at an edge server in the form of MSs. To minimize the service latency, such MSs have to be deployed on and, hence, migrated to, the edge server that is co-located with the gNB currently serving the UAV, as the latter changes its point of access. This fact, along with the need to maintain the internal state of the considered MS, makes stateful container migration the key technology to ensure the continuous fulfillment of the service requirements.

MOSE effectively configures and implements a stateful migration process while addressing all the relevant technical

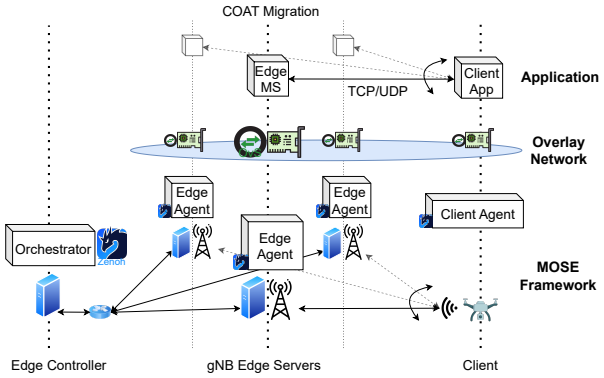


Fig. 1: Reference system architecture integrating MOSE.

challenges: (i) it minimizes service disruption, (ii) it preserves connectivity at the application level, (iii) it migrates an MS independently from the specific MS and the underlying edge technology, and (iv) it attains the migration KPI targets and the vertical's objective. MOSE does so by leveraging three main components: a *migration orchestrator*, which properly configures the migration process; a *migration agent* residing at each edge host and mobile device and implementing the MS migration; a *migration protocol* enabling the interaction between migration orchestrator and agents, and between agents.

Fig. 1 depicts the above scenario, mapping the MOSE components onto network entities: the MOSE orchestrator is deployed on an edge controller that is responsible for monitoring and managing the edge hosts, while the MOSE agent is hosted at every edge host (co-located with a gNB) as well as at the mobile device, e.g., a UAV consuming the service hosted at one of the edge servers. The migration protocol is based on Zenoh (https://zenoh.io), a highly scalable pub/sub/query protocol that ensures extremely low latency and high throughput, greatly outperforming the most popular communication protocols like MQTT, Kafka, and DDS [15]. Given its simple mechanisms for Remote Procedure Calls (RPC) and low latency optimizations, Zenoh enables an efficient message exchange between the migration entities. Moreover, its discovery mechanism regulates the automatic interconnection between agents and orchestrator, thus making MOSE highly flexible and scalable. As discussed in Sec. III, our COAT migration process relies on an overlay network to preserve the network connection established between the MS and the mobile end user upon migration. We thus design the MOSE agent both to implement the COAT migration process *and* to configure the required overlay network using OvS.

In summary, from a high-level perspective, the application running on board of the mobile device (client application) interacts with an edge MS through a generic network connection. Leveraging MOSE, such MS, together with the connection established with the client application, can be migrated across different edge hosts in a transparent way with respect to both the client application and the MS itself. Further, leveraging the migration orchestrator, the migration procedure can be configured to attain (i) the migration KPI targets, thus guaranteeing minimum impact on the mobile device's QoE,

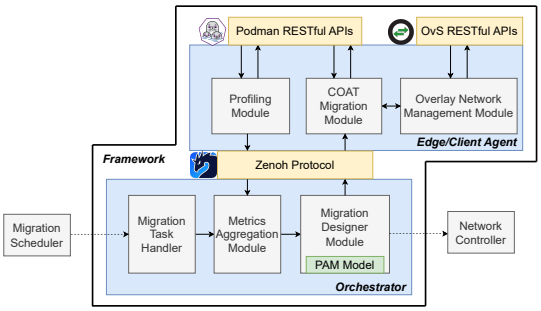


Fig. 2: MOSE agent and orchestrator: modules and libraries.

and (ii) the vertical's objective (i.e., to either minimize the experienced service disruption or the resource consumption in terms of required network bandwidth and CPU usage). Below, we describe the fundamental characteristics of the MOSE components. Their structure and the libraries they exploit are also illustrated in Fig. 2.

B. The MOSE components

The MOSE agent runs at the mobile devices and at each edge host; in the case of MS migration, we refer to source (destination) agent as the instance running at the source (destination) host. It consists of the following three main modules. The *COAT Migration* module is responsible for implementing the COAT migration process according to the configuration provided by the MOSE orchestrator. The *Overlay Network Management* module is in charge of managing and configuring the overlay network, thus enabling our COAT migration strategy. Finally, the *Profiling* module allows the agent to characterize all the relevant aspects regarding migration, i.e., to estimate the available bandwidth on the links connecting edge hosts, measure the MS state size and dirty-page rate, and estimate the PAM model parameters. The resulting metrics, collected periodically, are sent to, and processed by, the MOSE orchestrator, running on an edge controller. The functionality of these modules strongly rely on two fundamental sets of RESTful APIs, namely, Podman and OvS, both introduced in Sec. III. While the former is used by the Profiling and COAT Migration modules to interact with the Podman container engine, the latter is leveraged by the Overlay Network Management module to configure the overlay network. Finally, the message exchange between agent and orchestrator is regulated through the Zenoh protocol.

The MOSE orchestrator includes the following modules.

The *Migration Task Handler* is responsible for processing the migration task whenever this is triggered by a migration scheduler. The design and implementation of the migration scheduler are orthogonal to our solution and out of the scope of this work. The handler assigns the migration task to MOSE by specifying: (i) the ID of the container to be migrated, (ii) the IDs of the source and destination agents, and (iii) the target KPIs, along with the objective to be fulfilled, i.e., minimization of resource usage or minimization of migration downtime. Depending on this objective, the appropriate migration technique is selected and configured using our algorithm in the migration

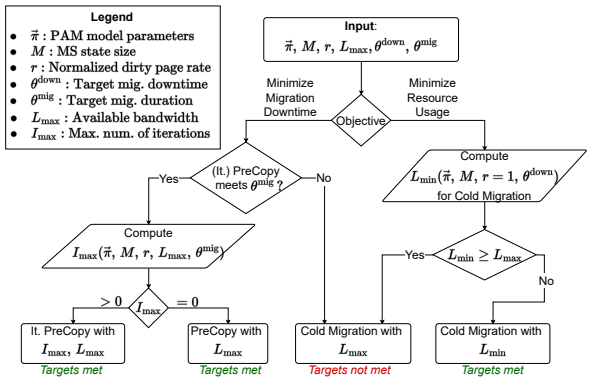


Fig. 3: Configuration algorithm executed by the MOSE orchestrator.

designer module described below. The *Metrics Aggregation* module is instructed by the migration task handler about the set of information that has to be collected to fulfill the given task. The *Migration Designer* uses the aforementioned set of information and the PAM model to predict an upper bound on the migration KPIs and properly set the migration parameters with the aim to fulfill the required level of QoE. This module executes the orchestration algorithm, summarized in Fig.3, defining a configuration that specifies (i) which migration strategy to adopt (i.e., Cold Migration, PreCopy, or Iterative PreCopy), (ii) the network bandwidth that the network controller must allocate for the migration process, and, in case of Iterative PreCopy, (iii) the number of iterations to execute. To apply such configuration, the Designer module conveys it to the COAT migration module of both source and destination agents, using Zenoh protocol.

The algorithm input is given by: (i) the metrics and parameters retrieved through the profiling module from the source and destination agents, and (ii) the migration task as defined by the migration task handler, including the target KPIs and the driving vertical's objective. When the goal is to minimize resource usage (right branch), the Cold Migration strategy is adopted, as it is the one that minimizes the amount of required bandwidth and CPU consumption. The algorithm then uses the PAM model to compute the minimum required bandwidth to meet the target downtime and checks whether it exceeds or not the estimated available bandwidth. When instead the goal is to minimize the migration downtime (left branch), the Iterative PreCopy strategy is enacted, as it minimizes service disruption. The algorithm preliminary checks whether this strategy meets the target migration duration and, if so, it computes the maximum number of iterations to meet such target. Importantly, in the PAM model we consider the worst-case situation, i.e., the MS dirty-page rate is set to its maximum value, so that we obtain an upper bound to the migration and downtime duration.

The MOSE protocol is implemented via pub/sub/query mechanisms provided by Zenoh. It regulates: (i) the signaling of migration tasks, (ii) the actuation of the steps of the migration process, and (iii) the transfer of checkpoint images.

As an example, Fig. 4 illustrates the communication flow between the orchestrator and the agents when an Iterative PreCopy migration task is performed. The migration task is

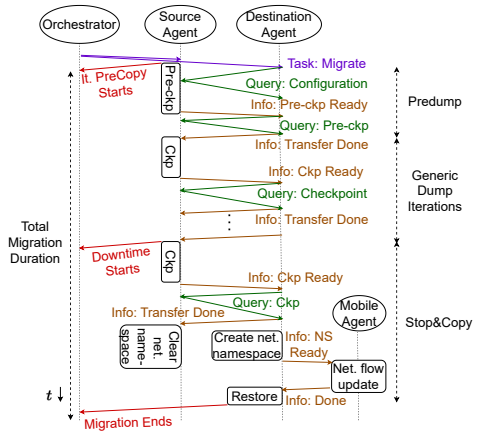


Fig. 4: MOSE protocol under Iterative PreCopy and COAT Stop&Copy.

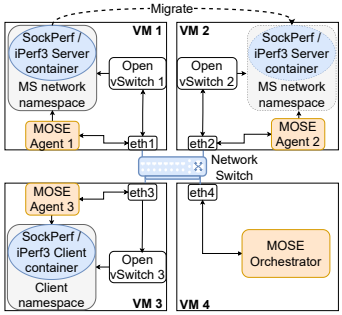


Fig. 5: Testbed setup for the MOSE framework.

published on a Zenoh topic and is accessed by both source and destination agents. First, the source agent informs the orchestrator that migration has begun. Then, while the source agent starts the pre-checkpoint procedure, the destination agent queries the required information to restore locally the MS, e.g., the MS's network namespace configuration setting. After the pre-checkpoint, the source agent asks the destination agent to gather the related pre-checkpoint image. Similarly, the source agent runs the checkpoint stage and notifies the destination host, which queries the corresponding image. After the checkpoint image is transferred, the MS's network namespace at the source host is cleared, while the namespace is recreated and configured at the destination host. Upon being informed of the migration of the MS's network namespace, the mobile agent updates the client's network flow towards the agent, thus preserving the MS established connection with the client application. Finally, the destination agent restores the MS and notifies the orchestrator accordingly.

We remark that our communication solution enables an automated migration with no need for additional remote control protocols such as SSH, thus minimizing the migration latency.

V. Experimental Results

A. Testbed setup

The testbed we developed comprises four identical virtual machines (VMs) and is hosted on a server featuring AMD Ryzen 7 5700G CPU. As depicted in Fig. 5, VM1 and VM2 represent two edge servers, acting, respectively, as the source

TABLE I: Comparison between MOSE and COAT [3]: MS migration KPIs

	Scenario / Duration	COAT [3]	MOSE: Downtime Minimization		MOSE: Resource Usage Minimization	
		90% C.I. [ms]	90% C.I. [ms]	Improvement [%]	90% C.I. [ms]	Improvement [%]
SockPerf MS	**Downtime,** T^{down}	$3,100.11 \pm 28.02$	509.05 ± 9.50	83.58	764.16 ± 47.30	75.35
	Total, T^{mig}	$3,100.11 \pm 28.02$	$1,622.07 \pm 64.66$	47.68	764.16 ± 47.30	75.35
iPerf3 MS	**Downtime,** T^{down}	$1,648.85 \pm 30.48$	517.42 ± 9.07	68.65	651.36 ± 16.31	60.52
	Total, T^{mig}	$1,648.85 \pm 30.48$	$2,174.13 \pm 31.80$	-31.84	651.36 ± 16.31	60.52

and the destination host. VM3 hosts the client container, thus acting as the end device that interacts with the edge servers. VM4 acts as the edge controller, which is responsible for monitoring and managing the edge hosts. To build our framework, we deploy three independent instances of the MOSE agent, respectively, on VM1, VM2, and VM3, and one MOSE orchestrator on VM4.

We conducted two independent sets of experiments employing SockPerf (https://github.com/Mellanox/sockperf) and iPerf3 (https://iperf.fr) as examples of stateful MSs to migrate. SockPerf is a network latency benchmarking tool to measure the communication latency in ping-pong connectivity tests on the client side, while iPerf3 is a popular tool for active measurements of the achievable bandwidth on IP networks. Both require a TCP connection, and for both we deploy the server at the source host (VM1) and the client on the mobile device (VM3). Leveraging the profiling module (see Sec. IV-B), we characterize such MSs under default settings in terms of state size (resulted in 10 MB and 3 MB for SockPerf and iPerf3, resp.), and dirty-page rate (resulted for both MSs between 3 and 5 pages/s, which, given a default page-size of 4,096 B, is a relatively low value compared to their state size).

According to the mobility scenario shown in Fig. 1, we first consider a mobile device (VM3) moving away from the gNB co-located with the source edge host (VM1) and approaching the one co-located with the destination host (VM2); the increased latency triggers the migration process. Further, we consider a network slice dedicated to the migration process so as to ensure a high-data rate network connection between the source and destination hosts.

B. MOSE migration performance

Leveraging our testbed, we experimentally characterize the migration process, for both SockPerf and iPerf3, in (i) the COAT testbed [3], which relies on scripts and SSH tunnels to control the cold migration procedure, and when (ii) the MOSE framework is used with resource usage minimization and (iii) downtime minimization set as a goal. Specifically, we consider the 90% confidence interval over 100 repetitions, and provide the average time improvement under MOSE with respect to COAT. Further, for the MOSE experiments, we set the target downtime and migration duration to 1 s and 5 s (resp.). The results are shown in Tab. I, which reports the values of the fundamental migration KPIs, i.e., the downtime T^{down} and the total migration duration T^{mig}.

When the objective is to minimize the downtime, the MOSE orchestrator selects the Iterative PreCopy as a migration strategy, which, as mentioned, allows trading a shorter downtime for a longer migration duration. Given the low dirty-page rate of the two MSs, such strategy allows for an average downtime approximately equal to 500 ms (with about 9 ms margin of error for the 90% confidence interval) and a value of total migration duration that is still quite small. Instead, when the objective is to minimize the resource usage, the Cold Migration strategy is selected, which trades a smaller network bandwidth for a longer downtime. Importantly, in this case, downtime and total migration duration turn out to be almost the same: the average downtime is about 760 ms and 650 ms, with 47 ms and 16 ms margins of error for the 90% confidence interval, for, respectively, SockPerf and iPerf3 (the latter is slightly lower thanks to its smaller state size).

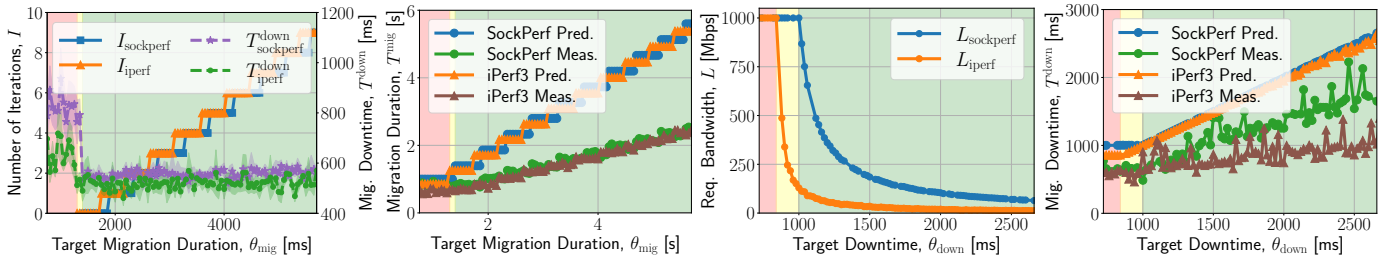
Notably, compared to COAT, MOSE greatly reduces the downtime, up to 80% and 70% for SockPerf and iPerf3 (resp.). *Such significant reduction allows passing from second to sub-second operations, thus making stateful container migration suitable for time-critical MSs.* This improvement is due to two key features of MOSE: (i) its efficient approach to signaling, which completely avoids the time overhead due to SSH; (ii) the direct interaction of the agents with the Podman and OvS APIs, resulting in a more efficient command execution.

C. MOSE orchestration strategy

We now demonstrate the effectiveness of our MOSE orchestration algorithm under the two vertical's objectives introduced above. To do so, we vary the target migration duration (Fig. 6a-6b) and downtime (Fig. 6c-6d), and show the configuration output by the orchestrator and the values obtained for the fundamental migration KPIs. Importantly, depending on the target values, we identify three main regions where the given target can be met (i) for neither SockPerf nor iPerf3 (red), (ii) for iPerf3 only (yellow), and (iii) for both MSs (green).

Migration downtime minimization. To attain this goal, MOSE configures the migration process so as to leverage the maximum available bandwidth for the link connecting source and destination hosts, and it computes the number of PreCopy iterations that allows meeting the target migration duration. We validate our orchestration algorithm by varying the target migration duration from 0.8 s to 6 s, and observe, for both SockPerf and iPerf3, the resulting migration configuration, along with the KPIs values that are actually experienced.

Fig. 6a presents the number of PreCopy iterations, I, and the corresponding downtime T^{down} as functions of the target migration duration θ_{mig}. When θ_{mig} cannot be met (red region), the MOSE orchestrator configures the migration process according to the cold migration strategy, thus resulting in a relatively high downtime. Instead, when the migration duration target can be met (green region), the MOSE orchestrator selects the Iterative PreCopy migration strategy and increases the number of iterations with the target migration duration. Due to the iPerf3 smaller state size, the Iterative PreCopy



(a) (b) (c) (d)

Fig. 6: MOSE performance for downtime (a-b) and resource usage (c-d) minimization. (a) No. of iterations for varying target migration duration and measured downtime; (b) Predicted and measured migration duration for varying target values; (c) Required network bandwidth for varying target downtime; (d) Predicted and measured downtime for varying target values.

strategy is feasible for smaller values of θ_{mig} relatively to SockPerf, resulting in a performance in the yellow region. Further, Fig. 6b depicts the total migration duration for varying values of θ_{mig}. Specifically, it shows the upper-bound that is predicted using the PAM model and the actual migration duration, both of which exhibit a behavior that is consistent with the configured number of iterations. Notably, the measured migration duration is always shorter than the prediction thereof, thus validating the ability of PAM of providing an upper-bound on such KPI. However, due to the accumulation of the error, the gap between the predicted and the migration duration becomes more evident as I increases.

Resource usage minimization. To achieve this goal, MOSE selects Cold Migration and computes the minimum bandwidth, L, required on the link connecting the source and destination hosts that fulfills the target downtime. To validate our orchestration algorithm, we now vary the target downtime from 0.7 s to 2.5 s, and, again, we record the migration configuration and the KPIs values for both SockPerf and iPerf3. Fig. 6c shows L as a function of the target downtime θ_{down}. When θ_{down} cannot be met (red region), MOSE configures the migration to use the maximum available network bandwidth, i.e., 1 Gbps. Otherwise (green region), it computes through the PAM model the minimum amount of network bandwidth between source and destination hosts that meets the corresponding target. In fact, as θ_{down} increases, the value of L decreases. Since the iPerf3 container has a smaller state size than the SockPerf container, the reduction of the required bandwidth L is feasible for smaller values of θ_{down} (yellow region). Further, Fig. 6d presents both the upper-bound on the downtime (computed through the PAM model) and the value that is actually experienced, as θ_{down} varies. Both values exhibit a negative correlation with the value of L. Again, the measured value of downtime is always smaller than the predicted one, which underlines how the PAM model provides an accurate upper-bound on such KPI.

In summary, our results highlight that: (i) MOSE greatly improves the migration performance relatively to the state of the art; (ii) it can reduce the service downtime from seconds to sub-seconds, thus making migration feasible for time-critical applications; (iii) it can effectively configure the migration process to meet the KPI target values and accounting for different vertical's objectives.

VI. Conclusions

We addressed the challenges posed by stateful MS migration at the network edge by envisioning MOSE, a novel framework that orchestrates the migration process to ensure minimal impact on the user's QoE. Leveraging our testbed and realistic MSs, we showed that MOSE greatly outperforms the state of the art, with a reduction of the service downtime up to 80%. Further, we validated MOSE for varying migration KPI targets and accounting for different vertical's objectives, thus proving that MOSE can effectively address scenarios where reliability and robustness are of primary importance.

References

[1] A. Furda *et al.*, "Migrating enterprise legacy source code to microservices: On multitenancy, statefulness, and data consistency," *IEEE Software*, 2018.

[2] T. Erl, *Service-Oriented Architecture: Analysis and Design for Services and Microservices*, 2nd ed. USA: Prentice Hall Press, 2016.

[3] Y. Yu, A. Calagna, P. Giaccone, and C. F. Chiasserini, "TCP Connection Management for Stateful Container Migration at the Network Edge," in *IEEE MedComNet*, 2023.

[4] A. Calagna, Y. Yu, P. Giaccone, and C. F. Chiasserini, "Processing-aware Migration Model for Stateful Edge Microservices," in *IEEE ICC*, 2023.

[5] M. Terneborg, J. K. Rönnberg, and O. Schelén, "Application agnostic container migration and failover," in *IEEE LCN*, 2021.

[6] C. Puliafito, A. Virdis, and E. Mingozzi, "The impact of container migration on fog services as perceived by mobile things," in *IEEE SMARTCOMP*, 2020.

[7] O. I. Abdullaziz *et al.*, "Enabling mobile service continuity across orchestrated edge networks," *IEEE TNSE*, 2019.

[8] S. Ramanathan *et al.*, "Live migration of virtual machine and container based mobile core network components: A comprehensive study," *IEEE Access*, 2021.

[9] R. Müller, C. Meinhardt, and O. Mendizabal, "An architecture proposal for checkpoint/restore on stateful containers," in *ACM SAC*, 2022.

[10] G. Panek *et al.*, "5G-Edge Relocator: a Framework for Application Relocation in Edge-enabled 5G System," in *IEEE ICC*, 2023.

[11] N. An *et al.*, "Seamless virtualized controller migration for drone applications," *IEEE Internet Computing*, 2019.

[12] Y. Qiu, C. Lung, S. Ajila, and P. Srivastava, "LXC container migration in cloudlets under multipath TCP," in *IEEE COMPSAC*, 2017.

[13] P. Raad *et al.*, "Achieving sub-second downtimes in large-scale virtual machine migrations with LISP," *IEEE TNSM*, 2014.

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

[15] W. Liang, Y. Yuan, and H. Lin, "A performance study on the throughput and latency of Zenoh, MQTT, Kafka, and DDS," *arXiv*, 2023.