POLITECNICO DI TORINO
Repository ISTITUZIONALE

TCP Connection Management for Stateful Container Migration at the Network Edge

Original
TCP Connection Management for Stateful Container Migration at the Network Edge / Yu, YEN-CHIA; Calagna, Antonio; Giaccone, Paolo; Chiasserini, Carla Fabiana. - ELETTRONICO. - (2023). (Intervento presentato al convegno IEEE MedComNet 2023 tenutosi a Ponza (Italy) nel 13-15 June 2023) [10.1109/MedComNet58619.2023.10168849].

Availability:
This version is available at: 11583/2978170 since: 2023-04-26T14:14:55Z

Publisher:
IEEE

Published
DOI:10.1109/MedComNet58619.2023.10168849

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

©2023 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)

25 April 2024

TCP Connection Management for Stateful Container Migration at the Network Edge

Yenchia Yu
Politecnico di Torino
Torino, Italy

Antonio Calagna
Politecnico di Torino
Torino, Italy

Paolo Giaccone
Politecnico di Torino
Torino, Italy

Carla Fabiana Chiasserini
Politecnico di Torino
Torino, Italy

***Abstract*—Container migration has emerged as the most effective way to ensure proximity of time-critical microservices at the network edge with mobile end devices. However, ensuring service continuity while migrating microservices that rely on an established TCP connection is still a significant technical challenge. In this paper, we investigate such pivotal issue and propose COAT, a novel, yet simple, network architecture that leverages overlay network technology to achieve seamless TCP connection migration. Through experimental validation using sample microservices, we show that, compared to the traditional approach that does not support connection migration, our solution enables the successful migration of microservices relying on an established TCP connection, at the cost of a 14% maximum increase of the migration duration. Importantly, our solution to the problem of connection migration does not require the use of a dedicated protocol, or any modification to the application source code or the kernel.**

***Index Terms*—Migration, Mobile services, Overlay Networks, Edge computing**

I. Introduction

In recent years, edge computing has been acknowledged as the state-of-the-art paradigm to overcome the bandwidth and latency challenges in cloud computing architectures. The main idea of edge computing is to bring applications, computational capabilities, and storage facilities closer to the end users, thus significantly reducing processing and communication delays. Moreover, to fully exploit the benefits of cloud and edge computing architectures (namely, scalability, availability, and resiliency), applications are often designed in the form of microservices chains, taking advantage of the lightweight container virtualization technology [1].

Concurrently, due to the rapid development of mobile communication networks (e.g., 5G/6G), the main consumers of the edge services have evolved from static to mobile devices, such as connected cars and Unmanned Aerial Vehicles (UAVs) – scenarios that require the support of high-demanding, latency- and bandwidth-critical applications. In this context, container migration techniques have gathered attention as an effective solution to address mobility challenges by ensuring continuous proximity of edge microservices with mobile end devices.

Two fundamental migration strategies can be identified, i.e., stateless and stateful migration, with the second being used whenever keeping track of the microservice internal state is essential to guaranteeing service continuity. Importantly, despite the current trend favouring the development of stateless microservices, stateful microservices are still extremely common due to the complexity in refactoring legacy monolithic applications [2].

C. F. Chiasserini and P. Giaccone are also with CNIT, Parma, Italy.

In this work, we therefore focus on stateful container migration, and, specifically, on the problem of connection migration. Indeed, upon statefully migrating a microservice between edge servers, it is critical to provide seamless migration of its network connection with the mobile end users, in order to minimize service disruption. We underline that, despite many recent studies have experimentally demonstrated the potential and effectiveness of stateful container migration techniques, just few of them have actually investigated the akin connection migration issue. Moreover, the existing solutions are mostly application-specific and based on either kernel or protocol customization, which we argue to lack generality and to be complex to implement and unfeasible to integrate with container virtualization technology.

To fulfill this gap and allow for a performant and efficient migration of stateful microservices, we propose COAT, a novel, yet simple, network architecture that, independently from the specific microservice, permits to preserve the established connection thereof with the mobile end users, during stateful migration. Specifically, due to its wide popularity and practical relevance, we focus on the TCP transport layer protocol [3], which by itself does not support the mobility of the connection endpoints. The benefits of COAT can thus be summarised as follows:

- It seamlessly migrates a generic microservice container with established TCP connections;
- It keeps track of the TCP connection states upon migration, thus avoiding reconnection procedures;
- It preserves all the data queued inside the TCP socket, thus preventing data losses;
- It performs the microservice stateful migration procedure in an agnostic way with respect to either the server or the client side of the connection.

The rest of the paper is organized as follows. Section II provides an overview of container and connection migration, thus introducing the main tools we used to design our solution. Section III describes the COAT network architecture and our enhanced version of the stateful container migration process. Section IV details the realistic microservices and the

testbed setup we use in our validation experiments, which are presented in Section V. Finally, Section VI discusses some relevant related work while highlighting the novelty of our study, while Section VII draws our conclusions.

II. Technological Background and Solution Strategy

This section presents an overview of container migration and the primary enabling tools to implement it. Further, it introduces the TCP connection migration challenges and the technologies we leverage to tackle these issues, namely, `TCP_REPAIR` mode and overlay networks.

A. Container migration

Container migration enables container relocation across hosts while meeting critical time constraints. Two fundamental container migration techniques have emerged: stateless and stateful migration. In this work, we focus on the latter, which is used whenever keeping track of the service state is fundamental to ensure service continuity. Hence, stateful container migration enables moving not only the container template image from source to destination host, but also the service internal state. In other words, the migrated container can seamlessly restore its previous working state, thus guaranteeing minimal impact on the Quality of Experience (QoE) of the final users. The fundamental off-the-shelf tools required to implement stateful container migration are CRIU and Podman, as detailed below.

CRIU [4]. Checkpoint/Restore In Userspace (CRIU) is widely considered the key tool for stateful migration from a process layer perspective. It implements two major procedures: (i) the *checkpoint procedure*, which freezes a running process, collects its internal state, and encapsulates it into an image, and (ii) the *restore procedure*, which creates a new process and restores its state by leveraging a previously acquired checkpoint image. Such checkpoint image mainly includes: (i) the CPU-context state, e.g., the processes tree structure and the associated registers, (ii) the network sockets, (iii) the memory content, and (iv) the open file descriptors. Importantly, CRIU features the `tcp-established` option, which instructs CRIU to collect, along with the internal state of the container, the information related to the currently active TCP connection, thus allowing for a successful restoration of the TCP connection state during migration.

Podman [5]. It is an open-source tool designed to develop, manage, and run containers and pods according to the Open Container Initiative (OCI) standards. Among the many off-the-shelf container engines, e.g., Docker and LXC, Podman is the one featuring the strongest integration with CRIU, by directly leveraging its APIs and, thus, effectively supporting container migration at the microservice layer. As a container engine, Podman enables the creation of isolated container environments by leveraging kernel namespaces and exposes the option to customize a container network namespace, thus providing high flexibility on the container network configuration.

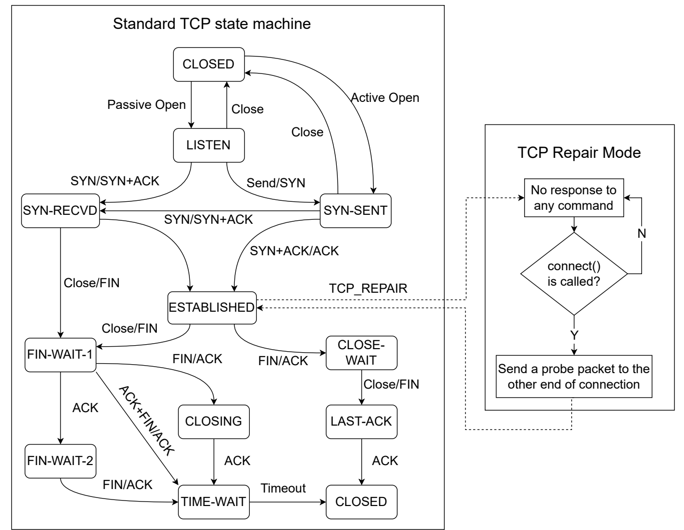


Fig. 1: TCP state machine diagram [7] with Repair Mode

Leveraging both CRIU and Podman, multiple stateful migration strategies can be defined. We focus on the simplest one, i.e., Cold Migration, consisting of the following steps: (1) creation of a snapshot of the container (named “checkpoint”) at the source host, (2) transfer of the checkpoint image from source to destination host, (3) restoration of the container at the destination host.

B. Connection migration

Connection migration is one of the critical issues concerning the migration of microservices with an always-established connection. We focus on TCP as connection-oriented transport protocol, as it is often used for legacy and modern edge applications [3], and we discuss the features of TCP that we can leverage to support connection migration. Notably, once a TCP connection is established, the protocol does not provide a way to modify or redirect such connection, unless through a complete re-connection procedure. To overcome this issue, and, hence, enhance the migration of TCP connections, a special option for the TCP socket has been introduced from Linux kernel version 3.5 onwards, namely, `TCP_REPAIR` [6].

When the `TCP_REPAIR` option is used, the TCP socket is switched into a special mode where any native TCP action performed on the socket has no effect (as depicted in Figure 1). In this condition, the state of the TCP connection can be successfully “checkpointed” by CRIU and restored on a new host machine, with a probe packet being eventually sent to notify the other connection end point that the communication can be resumed. However, the `TCP_REPAIR` option is not widely used due to the following required conditions to achieve a successful connection restoration: (i) address consistency: the microservice container, when migrating from source to destination host, has to be assigned the same IP address; (ii) network reachability: when moved to the destination host, the microservice container must be able to directly reach the other end involved in the communication. In other words, the

TCP_REPAIR option only provides the possibility to freeze and collect the state of the TCP socket, thus not tackling scenarios in which the IP address may change after migration. Moreover, to successfully resume the communication flow, the probe packet has to be correctly received at the destination, which is not trivial in the case of migration between distinct private networks.

We address the above requirements for TCP repair mode by defining a proper logical *overlay network* in which traffic flows can be dynamically managed. To do so, we leverage Open vSwitch (OvS) [8], a production-quality, multilayer virtual switch that provides two functions that are crucial for our purposes: (i) overlay network creation, and (ii) network flow management. Indeed, it creates overlay networks based on Virtual Extensible LAN (VXLAN) – a technique that encapsulates OSI layer 2 Ethernet frames within layer 4 UDP datagrams. Once the overlay network is established, users can easily define or change the behavior of the virtual switches, e.g., forwarding rules, through the OpenFlow protocol.

III. COAT Network Architecture

We now present our solution, named Container OverlAy TCP (COAT) architecture, which effectively supports TCP connection migration and addresses the akin networking challenges by leveraging the previously introduced tools. In addition, we integrate the COAT architecture in the stateful container migration procedure, yielding an enhanced procedure referred to as COAT migration, which enables the migration of microservices that rely on an established TCP connection.

The proposed COAT architecture is depicted in Figure 2, which includes three fundamental blocks, namely, the source host, the destination host, and the mobile end device. Source and destination hosts resemble the edge nodes that run a microservice before and after the migration process, respectively. The mobile end device, instead, is the node hosting the containerized client application that generates requests to be served by the microservice. The connectivity between the microservice and the client container is enabled by an overlay network implemented using interconnected virtual switches and customized network namespaces.

To effectively implement such architecture, we leverage the features provided by OvS to firstly create a virtual switch for each physical host and configure each of them to ensure their interconnection, thus defining the “backbone” of the overlay network. Secondly, we create two custom network namespaces, one for the microservice at the source host and one for the client container at the mobile end device. Both are then connected with the virtual switches, to complete the overlay network. Thirdly, we use Podman to run both the microservice and the client, and bind them to their dedicated network namespaces, hence connecting them to the overlay network. Once this third step is completed, the microservice and the client can communicate using the TCP protocol on top of the newly defined overlay network.

We underline that, when the microservice migration is performed, the TCP connection between the microservice and the client is preserved by (i) leveraging the TCP_REPAIR option to collect the connection state, and (ii) imposing an exact recreation of the microservice namespace at the destination host, especially in terms of its IP address configuration. Thus, COAT effectively solves the network address consistency problem since, thanks to the overlay network, the same IP address can be easily replicated at the destination host. Furthermore, since overlay networks enable the creation of a distributed network among multiple machines and to dynamically manage the traffic flows, direct reachability between microservice and client is always guaranteed, even after the migration process has been completed. Nevertheless, to effectively integrate the COAT architecture with the traditional stateful migration process, additional operations are required, which involve the creation and replication of customized network namespaces and the management of the flow control rules.

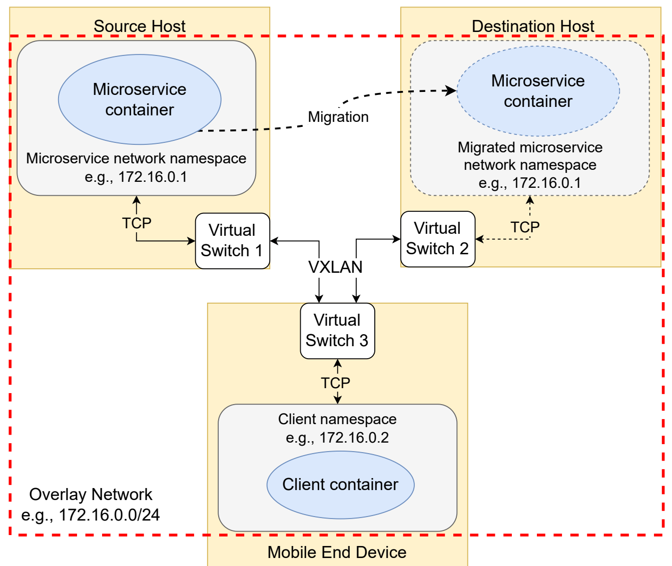


Fig. 2: COAT network architecture

To address such critical issues, we introduce COAT migration, which is an enhanced version of the stateful container migration process consisting of the steps illustrated in Figure 3 and described below.

- Step 1: Checkpoint the running container at the source host using Podman with the tcp-established option. Both the microservice state and the established TCP connection state are now dumped into the checkpoint image and stop running.
- Step 2.1: Clear the network namespace, thus preventing network configuration conflicts in the following steps.
- Step 2.2: Transfer the checkpoint image from source to destination host.
- Step 2.3: Re-create and configure the network namespace at the destination host to match the original one, which is required for the later container restore procedure to be successful.
- Step 3: Update the network flow of the TCP connection.

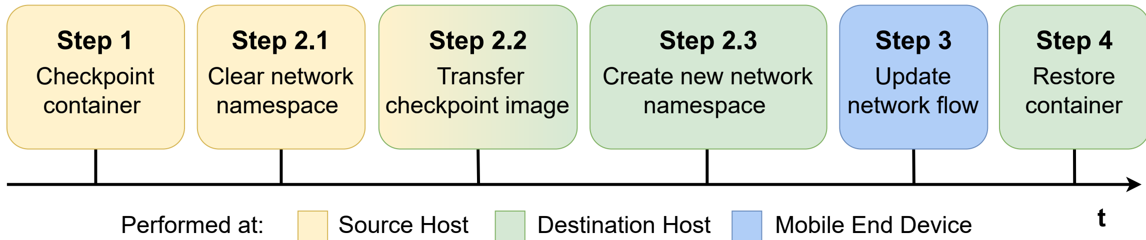


Fig. 3: Enhanced stateful container migration procedure integrating the COAT network architecture

Firstly, update the flow control rule in OvS. During the network namespace recreation, a new virtual network interface is generated, along with a new MAC address. The ARP table at the client host is then cleared in order to ensure a successful ARP discovery process once the TCP connection is restored.

- Step 4: Restore the container from the checkpoint image. Now, the microservice, and its established TCP connection, can resume from its previous working state.

We notice that Steps 2.1, 2.2, and 2.3 in Figure 3 may be executed in parallel, in order to speed up the migration procedure. However, as in this work we aim at validating the novel architecture and at assessing the impact of each step on the total duration of the migration process, in our performance evaluation all steps will be executed sequentially.

To summarize, COAT makes it possible to define an enhanced stateful container migration procedure to effectively support microservices that rely on an already established TCP connection. In particular, the proposed network architecture (i) allows for the migration of the TCP connection state, thus avoiding any reconnection procedure, (ii) preserves all the data queued inside the TCP socket, hence avoiding packet loss, and, (iii) does not require any modification at either the server or the client application to support a stateful migration.

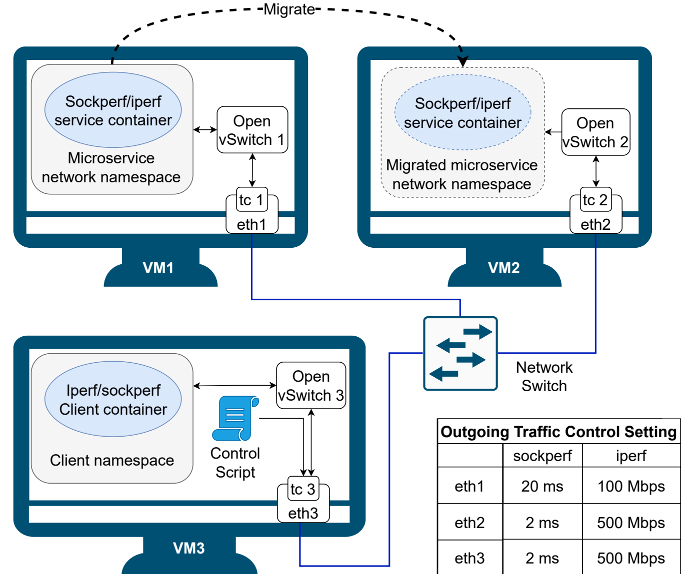
IV. Our Testbed

In this section, we briefly describe the testbed we developed to validate the connection migration in COAT architecture and to assess the performance of the COAT migration procedure.

A. Microservices

We perform two independent sets of experiments using *sockperf* and *iperf3* as examples of stateful microservices to migrate. They indeed resemble real-world microservices with established TCP connection and their features, briefly described below, allow us to effectively assess the network performance.

Sockperf is a network benchmarking utility over socket API. It is a powerful tool to perform network latency measurements, which can provide a full log of each packet's transmission timestamps in sub-nanosecond resolution. In our testbed, we configure sockperf to measure the network latency using the TCP protocol, in order to assess the impact of COAT migration on the communication latency. Specifically, latency is measured at the sockperf client side through the so-called *ping-pong test*, which calculates the time difference between the timestamp in which a probe packet is sent to the sockperf server and the one in which the corresponding response from the sockperf server is received.



Outgoing Traffic Control Setting		
	sockperf	iperf
eth1	20 ms	100 Mbps
eth2	2 ms	500 Mbps
eth3	2 ms	500 Mbps

Fig. 4: Testbed setup for COAT migration process

Iperf3 is a popular, lightweight tool for active measurements of the achievable bandwidth on IP networks. We use iperf3 to determine the impact of COAT migration on the communication throughput. To this end, we configure iperf3 to measure the throughput at the client side using the TCP protocol in reversed mode, i.e., the iperf3 server sending data to the iperf3 client, and we set the measurement interval to 100 ms.

B. Testbed and experimental setup

In our experiments, we leverage a cloud computing architecture featuring Intel Xeon Skylake CPU to instantiate three identical virtual machines (VMs). As shown in Figure 4, VM1 and VM2 represent two edge servers acting, respectively, as source and destination of the microservice migration process. Further, VM3 hosts the client container, thus acting as an end device that interacts with the edge servers. The migration procedure is controlled by a script running on VM3, which passes the control commands to VM1 and VM2 using the Secure Shell Protocol (SSH). Specifically, consistently with Figure 3, the commands corresponding to Step 1, Step 2.1 and

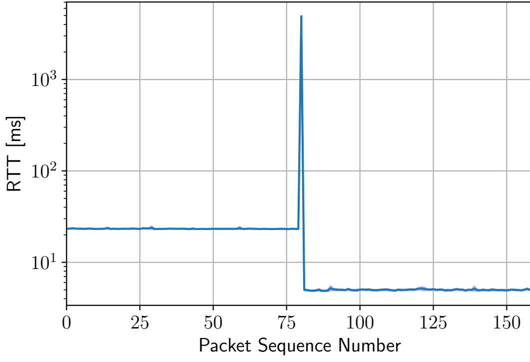


Fig. 5: Sockperf migration experiment: RTT measurement as a function of the packet sequence number

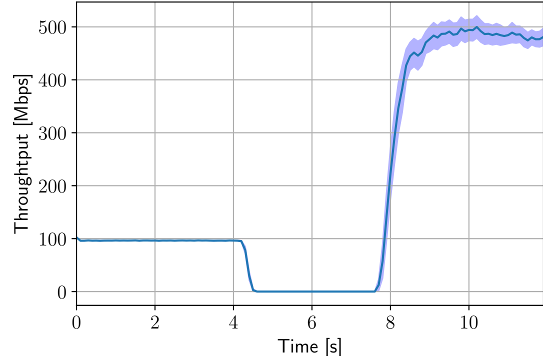


Fig. 6: Iperf3 migration experiment: throughput temporal evolution

Step 2.2 are executed at the source host (VM1), those related to Step 2.3 and Step 4 run at the destination host (VM2), while Step 3 runs locally at the end device (VM3). The three VMs can communicate with each other using their default network interfaces, i.e., eth1, eth2 and eth3 (resp.), which are provisioned by the underlying virtualization technology. To emulate realistic values of network latency and throughput, specific queuing disciplines are applied to the default network interface of each VM using `tc` – a tool used to configure the Linux kernel traffic scheduler. Through these disciplines, the outgoing traffic of the network interface on each VM can be manipulated as needed. In our testbed, we assume that VM2 has a closer physical distance to VM3 than VM1. Therefore, in the latency experiment, we apply a 20 ms delay to eth1 and a 2 ms delay to eth2 and eth3. Similarly, in the throughput experiment, we limit the bandwidth of eth1 to 100 Mbps, and that of eth2 and eth3 to 500 Mbps.

The results shown in the following have been obtained by averaging over 50 runs, and computing the 90% confidence interval.

V. Experimental analysis

We now use our testbed under the settings introduced in Section IV to validate the COAT architecture and to evaluate the migration performance according to our enhanced stateful container migration procedure.

We start by looking at the TCP connection round trip time (RTT) (Figure 5) and the experienced throughput (Figure 6), before, during, and after a microservice migration. In particular, Figure 5 shows the RTT measured by sockperf as a function of the packet sequence number. To highlight the impact of the migration process on the experienced RTT, a measurement window of 160 samples has been extracted around such event, with packet with sequence number 80 being the one transferred during the migration. Consistently with the scenario we are tackling and the settings presented in Section IV, the experienced RTT decreases when the sockperf microservice is moved from the source to the destination host. Also, one can observe a peak in the RTT values corresponding to the time interval during which the migration procedure takes place. Interestingly, two considerations can be drawn: (i) despite the migration process, the packet transmission is successful (hence also enabling a correct RTT measurement), and (ii) the value of RTT measured for the packet transmitted in correspondence of the migration reflects the total duration of the COAT migration process. In summary, **from the experiment we can conclude that (i) the TCP connection migration is successful, and (ii) no packet loss is experienced at transport layer during the migration process, which validates the proposed COAT architecture.**

Figure 6 presents the temporal evolution of the throughput measured by the iperf3 client. In this scenario, the migration of the iperf3 server container is performed in the time window between 4 and 8 seconds. After the iperf3 server container is migrated, the measured throughput increases significantly, which is again consistent with our settings. Indeed, the end device can experience a larger link bandwidth with the destination host than with the source host (see Section IV). However, during the migration of the iperf3 server container, the iperf3 client experiences zero throughput, as evident from the plot in Figure 6. It follows that: (i) even though the client is unaware of the microservice migration process, it still experiences a service disruption, and (ii) the duration of such disruption notably corresponds to the total duration of the COAT migration process. In summary, in spite of the fact that the COAT migration can successfully migrate an active TCP connection, a service disruption during migration is unavoidable. Our next objective is therefore to thoroughly characterize the service disruption time, along with its components, and to identify which step of the migration process contributes the most to service disruption.

To this end, we perform an experimental analysis to break the migration duration measured by sockperf and iperf3 into different components. First, we notice that, in our experiments, the main migration control script runs on the end device, hence most of the commands to implement the COAT migration procedure need to be passed to specific hosts via an SSH tunnel. Using SSH tunnels to remotely control other hosts inevitably introduces an additional delay in the procedure.

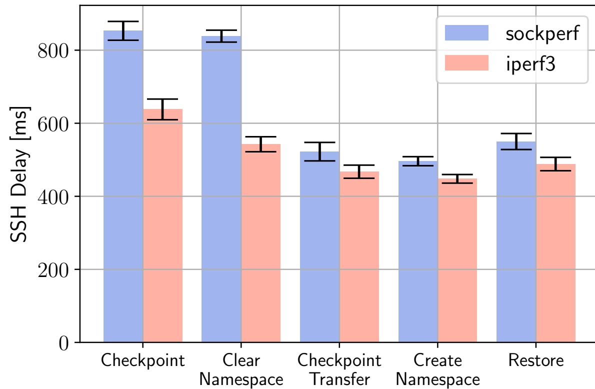


Fig. 7: Additional SSH delay introduced to remotely execute the commands related to the COAT migration procedure

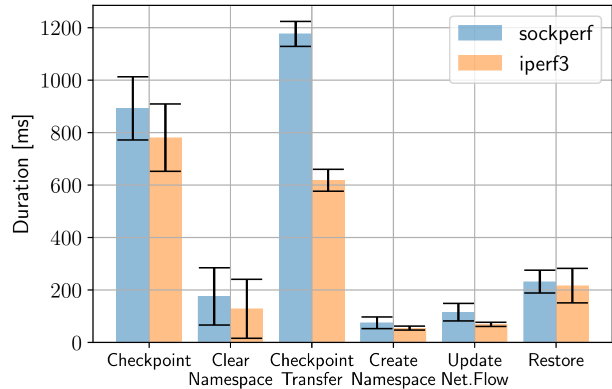


Fig. 8: Duration of each step of the COAT migration process, for the sockperf and iperf3 experiments

TABLE I: COAT migration performance using sockperf and iperf3

	Sockperf	Iperf3
COAT migration duration	2.67 s	1.87 s
SSH delay	2.41 s	1.95 s
Total migration duration (COAT migration duration + SSH delay)	5.08 s	3.82 s
Service disruption duration	5.11 s	3.71 s
SSH time overhead	47.16%	52.56%
COAT time overhead	13.67%	13.48%

Thus, the total migration duration mainly consists of two contributions: (i) the SSH delay and (ii) the actual duration of the migration steps.

The SSH delay is presented in Figure 7, for each step of the migration procedure and for both the cases where sockperf and iperf3 are migrated. The duration of such delay varies from 500 ms to 800 ms, depending upon the specific network latency setting, the amount of transferred data (e.g., the command itself), and the exchanged certificates to ensure a secure connection with the remote host. Importantly, such SSH delay can be easily avoided by applying better remote control mechanisms – a relevant, interesting aspect that is, however, out of the scope of this work.

Next, the actual duration of each migration step is investigated in Figure 8. By comparing such duration for the two experiments with sockperf and iperf3 (resp.), one can observe that the values measured for sockperf are higher. This is due to the fact that sockperf is characterized by a larger microservice state size, mainly consisting of the memory allocation, which we measured to be, on average, 10 MB for sockperf and 1.5 MB for iperf3. Moreover, we remark that the duration of the checkpoint transfer step is affected by the network setting we described in Section IV. Interestingly, the fundamental migration steps, namely, checkpoint, transfer, and restore, dominate the COAT migration duration, and the duration of these steps is consistent with the results of our previous study on stateful container migration modeling [9]. Thus, we can conclude that the **COAT migration does not introduce any time overhead in the three most-impactful migration steps (checkpoint, transfer, and restore) with respect to the original process**. Indeed, the COAT architecture only introduces three additional steps in the COAT migration process, namely, clear namespace, create namespace, and OvS flow update (see Figure 3), whose delay contribution is almost one order of magnitude smaller than that of the checkpoint, transfer, and restore steps. Remarkably, such additional steps are independent of the microservice state size; hence, their delay overhead would represent an even smaller percentage of the total overhead in case microservices with larger state size were considered.

In summary, we measured three duration metrics: (i) the COAT migration duration, comprising all COAT migration steps, (ii) the total migration duration, computed summing the above COAT migration duration and the SSH delay, (iii) the service disruption duration experienced by sockperf and iperf3 upon migration (also accounting for the SSH delay).

Notice that the SSH delay for the checkpoint step (i.e., Step 1 in Figure 3) is omitted from the calculation mentioned in (ii), since it happens before the migration process starts. As reported in Table I, by comparing the total migration duration and the service disruption duration, for both the sockperf and iperf3 experiments, their difference is negligible, which means that the measurements provided by such experiments are consistent with our breakdown analysis.

In conclusion, compared to the traditional stateful migration process, we can safely conclude that **the additional steps introduced by our COAT solution to enable seamless connection migration determine an increase on the migration duration up to roughly 14%.** We argue that such overhead is reasonably small compared to the great advantage COAT provides in easily, yet effectively, supporting migration for microservices with established TCP connection. On the other hand, in the two scenarios we considered, the overhead due to SSH remote control represents approximately 50% of the measured migration duration. Hence, it is critical that, in place of SSH tunnels, more efficient remote control mechanisms are used to minimize the duration of a migration procedure.

VI. Related Work

Stateful container migration has recently attracted a great deal of interest. An extensive survey on service migration in Multi-access Edge Computing (MEC) environments can be found in [10], while [11] presents an overview of current container migration techniques along with their fundamental metrics. However, far too little attention has been paid to the connection migration problem. Indeed, many studies, e.g., [12], [13], suggest to perform re-connection after a container is migrated by customizing the application source code and making the client aware of the migration process. To the best of our knowledge, only few studies discuss solutions to enable connection mobility in a completely transparent manner for the client. Such solutions are mostly based on dedicated protocols, network proxy, overlay network tunneling, and software-defined networking (SDN).

The works [14], [15] propose Multi-Path TCP (MPTCP) protocol as an effective solution to implement seamless connection migration, since it permits to define multiple subflows for the same connection. Similarly, [16], [17] thoroughly investigate the QUIC protocol and propose an extension thereof to effectively support server-side connection migration, thus advocating its validity for container migration. Other approaches, e.g., the ones proposed in [18], [19], leverage the cloud platform's network proxy to hold and redirect active connections with external clients while performing intra-cloud or inter-cloud service migration. Likewise, [20], [21] design dedicated network proxies to redirect the network flows for general connection migration purposes. Furthermore, [22] investigates the Locator/Identifier Separation Protocol (LISP), i.e., an overlay routing level on top of legacy IP, and suggests how to enhance it to effectively support VMs mobility management. In addition, [23] presents an SDN-based seamless UAV controller migration testbed, which addresses the connection migration issue by manipulating the MAC addresses and leveraging SDN flow duplication functionality.

We finally recall that the main objective of our work is to enhance the stateful migration process to effectively support microservices with an always-established connection. We achieve this goal through an architectural solution based on overlay networks that, unlike previous solutions, is application-independent, requires no dedicated protocol and no modifications to the kernel or application source code.

VII. Conclusions

Container migration has become one of the fundamental technologies to support service mobility at the network edge. Nevertheless, multiple technical challenges still need to be addressed, especially those related to connection migration and service continuity. To fill this gap, we introduced COAT, a novel and effective, yet simple, network architecture that leverages overlay network technology to achieve seamless TCP connection migration. To effectively integrate our proposed architecture with the traditional stateful migration process, we envisioned an enhanced version of such procedure. We validated our solutions using popular, real-world microservices. In particular, our experimental analysis demonstrated COAT migration process successfully enables microservice stateful migration while effectively preserving the state of the TCP connection, at the cost of a 14% maximum increase of the migration duration. We finally remark that COAT architecture effectively addresses the problem of connection migration without requiring a dedicated protocol or any modification to the application source code or the kernel.

References

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

[2] A. Furda, C. Fidge, O. Zimmermann, W. Kelly, and A. Barros, "Migrating enterprise legacy source code to microservices: On multitenancy, statefulness, and data consistency," *IEEE Software*, 2018.

[3] D. Lee, B. E. Carpenter, and N. Brownlee, "Observations of UDP to TCP ratio and port numbers," in *ICIMP*, 2010.

[4] CRIU, "Checkpoint/restore," https://criu.org/.

[5] The Containers Organization, "Podman," https://podman.io/.

[6] J. Corbet, "TCP connection repair," https://lwn.net/Articles/495304/.

[7] L. L. Peterson and B. S. Davie, *Computer Networks, Fifth Edition: A Systems Approach*. Morgan Kaufmann Publishers Inc., 2011.

[8] Linux Foundation, "Open vSwitch," https://www.openvswitch.org/.

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

[10] 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.

[11] M. Terneborg, J. K. Rönnberg, and O. Schelén, "Application agnostic container migration and failover," in *IEEE LCN*, 2021, pp. 565–572.

[12] W. Bao, D. Yuan, Z. Yang, S. Wang, W. Li, B. B. Zhou, and A. Y. Zomaya, "Follow me fog: Toward seamless handover timing schemes in a fog computing environment," *IEEE Communications Magazine*, 2017.

[13] 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.

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

[15] F. Le and E. M. Nahum, "Experiences implementing live VM migration over the WAN with multi-path TCP," in *IEEE INFOCOM*, 2019.

[16] L. Conforti, A. Virdis, C. Puliafito, and E. Mingozzi, "Extending the QUIC protocol to support live container migration at the edge," in *IEEE WoWMoM*, 2021, pp. 61–70.

[17] C. Puliafito, L. Conforti, A. Virdis, and E. Mingozzi, "Server-side QUIC connection migration to support microservice deployment at the edge," *Pervasive Mobile Computing*, vol. 83, no. C, 2022.

[18] P. S. Junior, D. Miorandi, and G. Pierre, "Good shepherds care for their cattle: Seamless pod migration in geo-distributed kubernetes," in *IEEE ICFEC*, 2022, pp. 26–33.

[19] T. Benjaponpitak, M. Karakate, and K. Sripanidkulchai, "Enabling live migration of containerized applications across clouds," in *IEEE INFOCOM*, 2020, pp. 2529–2538.

[20] S. Kassahun, A. Demessie, and D. Ilie, "A PMIPv6 approach to maintain network connectivity during VM live migration over the internet," in *IEEE CloudNet*, 2014, pp. 64–69.

[21] M. Bernaschi, F. Casadei, and P. Tassotti, "SockMi: a solution for migrating TCP/IP connections," in *EUROMICRO PDP*, 2007.

[22] P. Raad, S. Secci, D. C. Phung, A. Cianfrani, P. Gallard, and G. Pujolle, "Achieving sub-second downtimes in large-scale virtual machine migrations with LISP," *IEEE Transactions on Network and Service Management*, vol. 11, no. 2, pp. 133–143, 2014.

[23] N. An, S. Yoon, T. Ha, Y. Kim, and H. Lim, "Seamless virtualized controller migration for drone applications," *IEEE Internet Computing*, vol. 23, no. 2, pp. 51–58, 2019.