Live Containerized Service Migration across Edge to Cloud Continuum

Thanawat Chanikaphon, Mohsen Amini Salehi
{*thanawat.chanikaphon*1,*amini*}@*louisiana.edu*
High Performance Cloud Computing (HPCC) Lab
School of Computing and Informatics, University of Louisiana at Lafayette

I. Introduction

Applications in smart IoT-based systems, such as those in assistive technologies [8] and autonomous vehicles [10], often have low-latency constraints to serve their goals. That is why edge computing has emerged to bypass the network bottleneck and bring the computing to the data proximity, thereby, fulfilling these latency constraints. The inherent resource shortage and lack of elasticity on the edge, however, has given birth to distributed systems with a continuum of computing tiers that can include the edge, fog, and cloud systems.

Services on the edge and fog systems desire mobility owing to the user or data mobility, and the necessity of relocating to the cloud upon the edge/fog oversubscription. More specifically, live migration of containerized microservices is required for service mobility, elasticity, and load balancing purposes [9]. Consider the example of a blind person who uses smart glasses and needs real-time process of observed objects enters a coffee shop where few people are watching low-latency video streaming using the available edge. Upon arrival of the blind person, to make room for the blind application, the video streaming service has to be migrated to cloud without any interruption in the watching.

Modern services on edge, fog, and cloud systems predominantly exploit containers and orchestrators. Although container runtimes (e.g., Docker) and orchestrators recently provided native live migration support [5], [11], they do not allow migration across autonomous computing systems with heterogeneous orchestrators. Our hypothesis is that non-native and non-invasive support for the live container migration is the need of hour and can unlock several new use cases. That is, the native support implies making changes in the computing infrastructure platform whereas non-native support does not and is doable across homogeneous and heterogeneous orchestrators. The non-native support is described in more detail in Section II.

In this study, we develop a non-native and non-invasive live container migration method via leveraging the nested container runtime system (Docker-in-Docker [6]). We design the architecture and develop the solution to enable container migration across heterogeneous orchestrators, e.g., between Kubernetes [4] and Mesos [7].

II. Non-Native Support of the Live Container Migration

A container runtime can enable the non-native support of the live migration by relying on an intermediate layer to enable the feature. That is, it does not require the runtime itself to have the ability of live migration, thus, providing a higher degree of generality than the native support. The intermediate layer for the container-based service can be subdivided into 2 categories, which are the *nested container runtime* and the *init process*.

1) Nested Container Runtime: In this approach, a containerized service consists of 2 layers: the outer container (or the host-level container) and the nested container. The outer container run the migration-enabled nested container runtime as the main process managing the nested container. The actual service is deployed as a nested container. With this approach, the service can be moved between the nested container runtime without directly interfering with the outer container runtime. The outer runtime can control the scheduling, isolation, and resource usage of the nested container through its parent (the outer container).

2) Init Process: Similar to the nested container runtime approach, a containerized service consists of 2 processes: the init process and the service's process. The init process, running as the main process of the container, spawns the service's process as its child on startup. The service's process can be checkpointed and restored as a child of another container init process without migrating the whole container. However, this approach requires developers to build container images having the migration-enabled init process in the first place.

III. Architectural Overview of the Live Container Migration System

The architecture of our designs consists of 4 main components, illustrated in Figure 1. The *orchestrator* and the *coordinator* operate at cluster-level management. The *runtime module* and *synchronization module* operate at the container-level management.

1) Coordinator: The coordinator is the main component that performs migration between clusters. The migration process is initiated by sending a request to the coordinator of the source cluster. Communication between clusters, monitoring of container status during the migration, and recovery in case of failure are done through coordination between the source cluster coordinator and the destination cluster coordinator.

2) Orchestrator: An orchestrator is a native component that already exists in the cluster. To enable live migration support, the orchestrator must be aware of the status of the container when being migrated. This mechanism can be seamlessly added using the *operator pattern* [2]. The operator will watch the occurring events during the migration and respond as required. Moreover, to relieve the burden of developers, the orchestrator requires a feature to automatically modify the container structure, such as adding a nested container runtime to the container. This feature can be implemented using an *interceptor pattern* [13]. By the interceptor detecting and modifying requests to the orchestrator, the developers can deploy services as usual.

3) Synchronization Module: During the migration, the applications are checkpointed to the checkpointed files. It is necessary to have a volume to store these files in each service. This module is responsible for synchronizing the checkpointed files between 2 volumes (source and destination). The connection between the two services is peer-to-peer *i.e.,* not through the coordinators, in order to minimize the hop of data transfer.

4) Runtime Module: The runtime module consists of the actual application and other necessary components that enable live migration features. The application is either a process or a nested container depending on the type of non-native support used.

In the case of using the nested container runtime, the orchestrator cannot directly manipulate the nested container such as auto-restart the failed container (self-healing). This problem can be solved by connecting the nested container to the orchestrator with the *ambassador pattern* [3]. Figure 1 elaborates on the internal components of the runtime module using the nested container runtime. The nested container has its own ambassador, which is the sibling container of the outer container. When an ambassador container is created, it also creates a nested container. If the nested container exits, the ambassador exits with the same status code, allowing the orchestrator to recognize the state of the nested container and decide to operate self-healing. Another component required in the nested container runtime approach is *Engine*. Operations related to nested containers such as create, checkpoint, restore, etc. must be requested through the Engine, which handles depending on that service migration status.

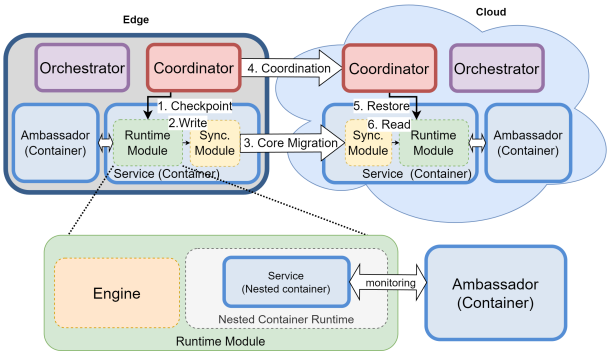

Fig. 1: Bird-eye view of the architecture.

IV. Evaluation

In each experiment, we set up two VMs in two different physical machines. A selected orchestrator is installed per VM. We developed the test application, which consumes an amount of memory as configured and counts a number every second. We chose an implementation for each approach as follows: (A) the work of Vu *et al.,* [12] as the native approach, (B) Docker-in-Docker as the non-native approach using the nested container runtime, and (C) FastFreeze [1] as the non-native approach using the init process.

We evaluated the performance in live migration between these approaches in two scenarios. In the first scenario, we performed the live migration between homogeneous Kubernetes orchestrators and varied the size of the application memory footprint. The metric of the evaluation is the service downtime. It can be observed by letting the application print a number every second and calculate the excess of 1 second between two printed numbers. Figure 2 demonstrates how the downtime increases when the size of the application memory footprint grows.

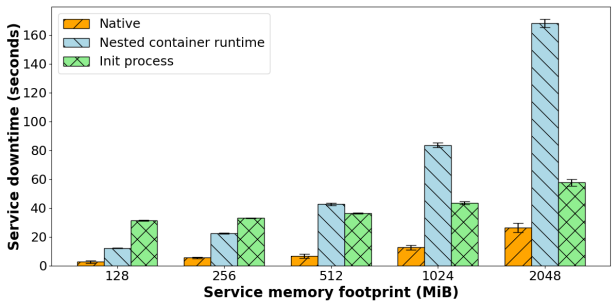

Fig. 2: Service downtime in migration across homogeneous orchestrators.

In the second scenario, we evaluated the possibility to migrate across heterogeneous orchestrators using each approach. We set up 5 container orchestrators for migrating a 128 MiB memory footprint application from them to Kubernetes. The result is shown in Table I. *K8s* is the migration from the same version of Kubernetes. *Mesos* is the migration from Apache Mesos. *K3s* is the migration from K3s, the lightweight version of Kubernetes, for simulating the edge-to-cloud migration. *Mini.* is the migration from Minishift, a single-node version of OpenShift, for simulating the migration between different distributions of Kubernetes. *Mini.** is the migration from Minishift running on a different operating system for simulating the migration between different versions of the OS kernel.

Approach	Required changing	Migration time (seconds)					Downtime (seconds)				
		K8s	Mesos	K3s	Mini.	Mini.*	K8s	Mesos	K3s	Mini.	Mini.*
Native	Orches-trator	3.14	Inf.	Inf.	Inf.	Inf.	3.06	Inf.	Inf.	Inf.	Inf.
Nested container runtime	Nothing	17.33	17.70	17.23	17.49	Inf.	12.21	12.08	12.13	12.75	Inf.
Init process	Appli-cation	36.75	35.65	36.91	36.87	37.57	31.29	31.32	31.35	31.30	32.59

TABLE I: Migration time and service downtime of a 128 MiB memory footprint application during the live migration from various orchestrators to Kubernetes.

REFERENCES

[1] Linux Plumbers Conference 2020. Linux Plumbers Conference 2020 (24-28 August 2020): FastFreeze: Unprivileged checkpoint/restore for containerized applications · Indico. Online; Accessed on 7 May 2022.

[2] The Kubernetes Authors. Operator pattern — Kubernetes. Online; Accessed on 7 May 2022.

[3] The Kubernetes Authors. The Distributed System ToolKit: Patterns for Composite Containers, July 2020. [Online; accessed 8. Aug. 2022].

[4] The Kubernetes Authors. Production-Grade Container Orchestration, July 2022. [Online; accessed 5. Jul. 2022].

[5] CRIU. Docker - CRIU. Online; Accessed on 7 May 2022.

[6] Docker. Docker - Official Image | Docker Hub, June 2022. [Online; accessed 30. Jun. 2022].

[7] The Apache Software Foundation. Apache Mesos, May 2022. [Online; accessed 5. Jul. 2022].

[8] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 68–81, 2014.

[9] Chunlin Li, Jingpan Bai, Yuan Ge, and Youlong Luo. Heterogeneity-aware elastic provisioning in cloud-assisted edge computing systems. *Future Generation Computer Systems*, 112:1106–1121, 2020.

[10] Bigi Varghese Philip, Tansu Alpcan, Jiong Jin, and Marimuthu Palaniswami. Distributed real-time iot for autonomous vehicles. *IEEE Transactions on Industrial Informatics*, 15(2):1131–1140, 2018.

[11] Adrian Reber. Minimal checkpointing support by adrianreber · Pull Request #104907 · kubernetes/kubernetes. Online; Accessed on 8 Aug 2022.

[12] SSU-DCN. SSU-DCN/podmigration-operator. Online; Accessed on 7 May 2022.

[13] Wikipedia contributors. Interceptor pattern — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Interceptor_pattern&oldid=937343798, 2020. [Online; accessed 4-July-2022].