UMS: Live Migration of Containerized Services across Autonomous Computing Systems

Thanawat Chanikaphon
HPCC Lab, School of Computing and Informatics
University of Louisiana at Lafayette, LA, USA
thanawat.chanikaphon1@louisiana.edu

Mohsen Amini Salehi
HPCC Lab, Computer Science and Engineering Department
University of North Texas
mohsen.aminisalehi@unt.edu

***Abstract*—Containerized services deployed within various computing systems, such as edge and cloud, desire live migration support to enable user mobility, elasticity, and load balancing. To enable such a ubiquitous and efficient service migration, a live migration solution needs to handle circumstances where users have various authority levels (full control, limited control, or no control) over the underlying computing systems. Supporting the live migration at these levels serves as the cornerstone of interoperability, and can unlock several use cases across various forms of distributed systems. As such, in this study, we develop a ubiquitous migration solution (called UMS) that, for a given containerized service, can automatically identify the feasible migration approach, and then seamlessly perform the migration across autonomous computing systems. UMS does not interfere with the way the orchestrator handles containers and can coordinate the migration without the orchestrator involvement. Moreover, UMS is orchestrator-agnostic, *i.e.,* it can be plugged into any underlying orchestrator platform. UMS is equipped with novel methods that can coordinate and perform the live migration at the orchestrator, container, and service levels. Experimental results show that for single-process containers, the service-level approach, and for multi-process containers with small ($<$ 128 MiB) memory footprint, the container-level migration approach lead to the lowest migration overhead and service downtime. To demonstrate the potential of UMS in realizing interoperability and multi-cloud scenarios, we examined it to perform live service migration across heterogeneous orchestrators, and between Microsoft Azure and Google Cloud.**

***Index Terms*—Containerized Services, Live Migration, Autonomous Computing Systems, Heterogeneous Orchestrators**

I. Introduction

Applications in smart IoT-based systems, such as those in assistive technologies and autonomous vehicles, 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 user (data) proximity, thereby, fulfilling the latency constraints. The inherent resource shortage and lack of elasticity on the edge, however, has given birth to a new distributed computing paradigm operating based on a continuum of tiers that can include the edge, fog, and cloud systems. To overcome the shortage of edge elasticity, the ability of live service relocation (*i.e., service migration*) across the edge-to-cloud continuum is crucial. In addition, enabling service migration can be instrumental in overcoming other longstanding challenges of modern distributed systems, such as user mobility, vendor lock-in, energy efficiency, load balancing, and realizing multi-cloud.

As an exemplar use case, consider a pair of smartglasses that is used along with the edge-cloud continuum to provide ambient perception for the blind and visually impaired people via real-time services for identification of obstacles and detecting of approaching objects. In a hypothetical scenario that is suggestive of the future we hope to create, a blind person enters a coffee shop where people are utilizing the resource-limited on-premise edge server to play an online game. To procure resources for the assistive services of the blind person, the gaming service has to be migrated to the cloud without any significant interruption for the gamers. Migration in the opposite direction can occur when the disabled person leaves the place. In analogy, this is much like a priority seat reserved for disabled people in the public transport systems. Another motivational use case for the live service migration is to avoid vendor lock-in via seamless migration of services across multi-clouds, *i.e.,* from one cloud provider to another.

Provided that modern software engineering methodologies, such as DevOps and CI/CD, predominantly exploit containers [1] and container orchestrators (*e.g.,* Kubernetes) for service deployments, the key to achieve service migration is to enable the *live migration of the containerized services* across computing systems. To migrate a containerized service, one may argue that we only need to checkpoint, transfer, and restore the service container. Indeed, at the high level, this is a valid argument and a container can be transparently checkpointed at the source and transferred to the destination. However, the problem is that, upon container restoration, the destination orchestrator does not recognize and adopt it to offer any management facilities (*e.g.,* scaling). The current remedy to this problem (*e.g.,* [4], [8]) is to make invasive changes to the platform of the underlying computing systems. Although the invasive approaches are generally efficient in the sense that they impose a low (lightweight) migration overhead, different computing systems are often controlled autonomously and system administrators do not have the authority to modify both source and destination systems. Moreover, the systems potentially employ distinct orchestrators (*e.g.,* Kubernetes and Mesos), whereas, the existing works only perform migration across homogeneous ones that curbs the usability of migration and has vendor lock-in implications. To our knowledge, there is no live service migration solution that can offer the best of both worlds: (i) operating ubiquitously across autonomous systems and heterogeneous orchestrators; and (ii) maintaining the migration efficiency.

To enable such a ubiquitous and efficient service migration, in this paper, we develop Ubiquitous Migration Solution

(UMS) that provides migration for different levels of authority the users may have over the underlying computing systems: *full control:* allowing for changes at the platform level of the source and destination systems; *limited control:* allowing changes only to the service image in both systems; and *no control:* that disallows any changes to the underlying systems.

UMS acts as an umbrella solution encompassing the three following migration approaches that correspond to the aforementioned authority levels: (A) *orchestrator-level migration approach* that requires full control over both source and destination systems to be able to make changes in their orchestrator; (B) *service-level migration approach* that demands a limited control only to change the service container image; and (C) *container-level migration approach* that does not demand any control over the underlying systems or services.

Supporting multiple migration approaches raises a challenge within UMS to transparently detect the structure of the underlying container and engage the appropriate migration approach. In addition, to support heterogeneous orchestrators, UMS has to be able to coordinate the migration across source and destination systems, irrespective of their underlying platforms. Beyond these, UMS has to choose the appropriate container(s) for migration. To handle all these complications, we design UMS to be a multi-layered such that it can abstract the decision making aspect, from the migration coordination challenges, and from the core migration process.

In summary, this paper makes the following contributions:

- Developing UMS, a framework that enables seamless and lightweight live migration of containerized services across autonomous computing systems with potentially heterogeneous orchestrators[1].
- Developing live container migration approaches operating at the orchestrator, container, and service levels.
- Demonstrating the feasibility of live migration of containerized services across heterogeneous orchestrators (Kubernetes, Mesos, K3S, and Minishift) and between Microsoft Azure and Google Clouds. We also analyse the imposed overhead of different migration approaches.

The rest of this paper is organized as follows: Section II provides a background for the live container migration and its related studies. Section III presents the design and implementation of UMS. Section IV describes the evaluation and the result. Finally, Section V concludes our work.

II. Related Works

Even though the container design principle is often interpreted that containers is ephemeral and migrating persistent storage data [2] suffices container migration, many developers and researchers disagree, and several research works have been undertaken to enable the checkpoint/restore of containerized services. At the orchestrator level, there have been attempts to integrate the checkpoint/restore ability into Kubernetes. Even though the discussion for this began in 2015 on the Kubernetes GitHub repository , there was no tangible outcome until 2020, when Schrettenbrunner presented the proof-of-concept of Kubernetes pod migration in his Ph.D. dissertation [4]. The work requires modifying the source code of the Kubelet and the container runtime interface (CRI) to support the checkpoint/restore operation. Tran *et al.,* [8] extended the work and implemented the API server to enable the live migration across two Kubernetes clusters. Souza *et al.,* [6] presented MyceDrive, a solution to migrate containers within a Kubernetes cluster based on the service-level migration approach. To the best of our knowledge, there has been no prior attempt to carry out migration at the orchestrator level based on the container-level migration approach.

[1]UMS and the experimental data are all available at: https://github.com/hpcclab/NIMS

III. Live Migration of Services Across Autonomous Platforms

A. Architectural Overview of UMS

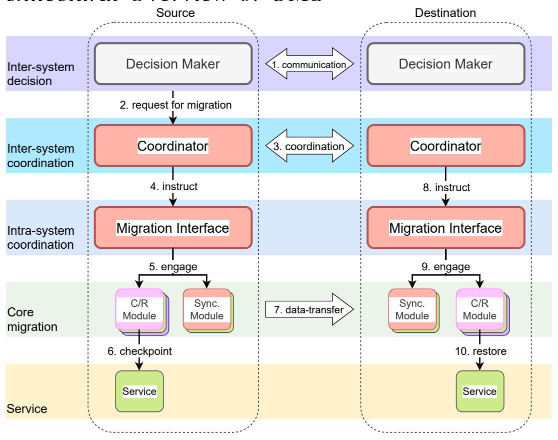

Fig. 1: Layered view of the UMS architecture. The live migration is performed within the top four layers of the architecture. Red components represent the contributions of this paper.

At the high level, the live container migration consists of five layers, shown in Figure 1, namely the *inter-system decision*, *inter-system coordination*, the *intra-system coordination*, the *core migration* and the *service*. UMS encompasses the first four layers and the last layer only includes the containerized service, which is incognizant of the migration process.

The *Inter-system decision layer* is responsible for determining the essentials of the live migration process: *What* containerized service(s) is/are the appropriate one(s) for the migration? *Where* should they be migrated to? and *When* is the appropriate time for the migration? After a decision is made, a request is sent to the *Coordinator* that is accountable for arranging the migration process between the two computing systems. This component determines *"how"* to perform the live migration via transparently identifying the feasible migration approach for the service in question. Then, it instructs the *Migration Interface* to engage (call) the modules required to carry out the determined container migration approach. In fact, Migration Interface abstracts the migration coordination from the supported migration approaches. The *Core migration* layer comprises the modules needed to perform the migration procedure, including the *Checkpoint/Restore (C/R) module* and the *Synchronization module* that transfers the checkpoint files to the destination system.

Although we have implemented the entirety of UMS, **this work concentrates on the live migration mechanism of it** (*i.e.,* Inter-and Intra-system coordination layers, and the Core migration layer) to enable seamless and lightweight live service migration across autonomous systems with potentially heterogeneous platforms. At this point, we have placeholders for the *Decision Maker*, such that the migrating container and the destination system are provided as inputs. In the future, we will extend UMS to automatically make such decisions.

B. Mechanics of the Live Migration Coordination

Our designed live migration mechanism constitutes three phases that are controlled by the source Coordinator. Figure 2 elaborates on the sequence of actions in each phase.

Pre-migration Phase: The migration process begins with the source Coordinator receiving the migration request consisting of two main pieces of information: the containerized service to be migrated; and the destination system to be migrated to. In Step 1, the source Coordinator determines the migration approach in consultation with the orchestrator. In response from the orchestrator, the Coordinator receives the *Specification* of the container in question, including its structure.

In Step 2, a request is sent to verify that the destination Coordinator is available. This step is designed to perform authentication and authorization across systems in the future. Upon confirming the availability, in Step 3, the destination Coordinator is sent the Specification to create a container identical to the one at the source system. Even though, in theory, the destination container creation overhead can be waived via overlapping it with the source container checkpointing step, creating it from early on in the migration process has two benefits: (A) it guarantees the availability and allocation of resources at the destination system for the migration; thus, the migration can be performed safely; and (B) creating the destination container provides the endpoint for the peer-to-peer data-transfer between the source and destination containers in the next phase. As such, in Steps 4—6, the destination Coordinator creates the new container via its orchestrator and then informs the source Coordinator in Step 7.

Migration Phase: To avoid inconsistency in the state of the migrating container that can be caused by the arriving messages from the orchestrator, at the beginning of Step 8, the source Coordinator blocks the container from receiving any control messages (*e.g.,* deleting). To handle the messages received from the user or other services (*i.e.,* data plane), a temporary delegate container, called *Frontman*, is deployed to inform the requester(s) about the temporary service unavailability and asks them to retry. Next, the source Coordinator instructs the migrating container to be checkpointed into the storage. It is noteworthy that the reason we use stop-and-copy approach for the migration is that the current container runtimes (*e.g.,* Docker) and orchestrators do not support per-copy and post-copy [4] approaches. Although the source container can be terminated after the checkpointing step, to be able to cope with the failures that can occur during the migration, we maintain the source container in an inactive state that is not running and has no dirty pages in the memory until the source Coordinator confirms the safe restoration of the service at the destination.

In Step 9, the Synchronization module begins to transfer the checkpoint files to the destination container in a peer-to-peer manner. The checkpoint files comprise service memory pages and necessary metadata. The data in persistent storage are transferred concurrently as needed. Finally, upon successful checkpointing, in Step 11, the source Coordinator informs its peer to restore the container from the destination storage.

Post-migration Phase: After confirming that the service at the destination system started successfully, the source Coordinator informs the migration requester of the new endpoint. Then, in Step 15, the source Coordinator requests the source orchestrator to delete the migrated container. To handle the requests (data plane) received after the migration completion (Step 14), the Frontman container starts redirecting the requests to the new service location. After the DNS is updated with the new endpoint information, the Frontman container is disposed.

C. Establishing Service Migration Approaches Operating at Different Levels

Once the coordination mechanism is performed, as shown in Figure 1, the Migration Interface is instructed to conduct the core migration via engaging the appropriate approach. For that purpose, Coordinator detects the architecture of the containerized service that itself is dictated by the level of changes we can force to the underlying systems. Depending on how widely the service is designed to be migrated and the level of authority we have to configure the source and destination systems, as shown in Figure 3, the following live migration approaches are needed: orchestrator level, service level, and container level.

a) Orchestrator-level migration approach: To enable this approach, we need to configure the orchestrator of the source and destination systems to be fully compatible. More specifically, the orchestrator should be configured to call the same Checkpoint/Restore and Synchronization modules used by the underlying container runtime. Although this approach is invasive, it enables containers to be efficiently migrated without requiring any modifications. Tran *et al.,* [8] developed a live migration solution across two Kubernetes clusters using this approach. They achieved checkpoint/restore via modifying the Kubernetes source code and utilizing Network File System (NFS), a shared storage solution, to transfer the checkpoint files. However, it is not viable across *autonomous* systems, such as those in the edge-cloud and multi-cloud scenarios.

To enable migration of containerized services without any shared storage, we developed a new synchronization module for [8] to receive the destination address within the migration request, and transfer the checkpoint files to that address. To mitigate the migration overhead, we furnished the synchronization module to overlap the container checkpointing and file transfer steps (see Steps 8 and 9 in Figure 2), *i.e.,* the file transfer step starts without waiting for the checkpointing step

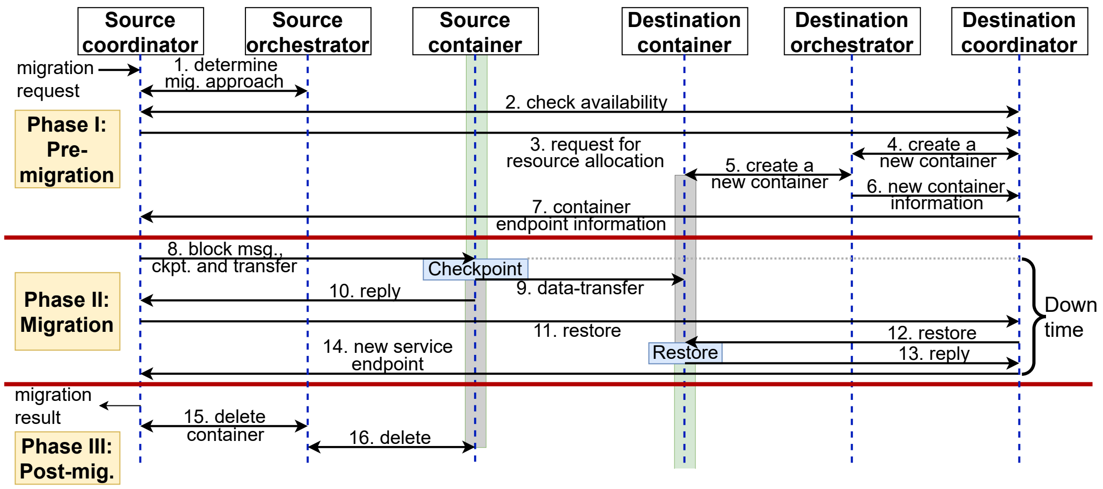

Fig. 2: Three phases of live container migration. The bidirectional arrows represent request and reply between entities. The green vertical boxes represent the containerized service in the active state, and the gray ones show them in the inactive state.

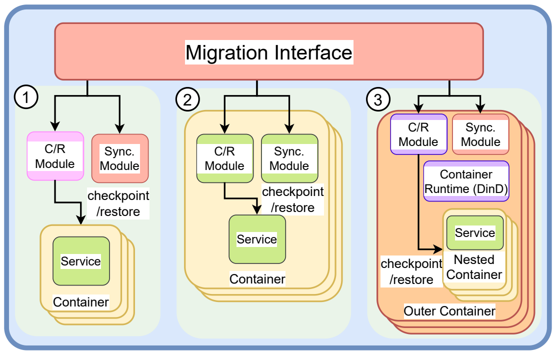

Fig. 3: Three migration approaches: ① orchestrator level, ② service level, and ③ container level.

to finish. This is accomplished via monitoring `write` events in the file system and copying the changed to the checkpoint file.

b) Service-level migration approach: The orchestrator-level approach demands full control over the underlying orchestrators, however, often our authority is limited and we cannot make changes beyond service images and their deployments. To enable migration under these circumstances, we require a non-invasive migration approach that can function at the service level without any cooperation from the orchestrator.

To make the service-level migration happen, the Checkpoint/Restore and Synchronization modules must be embedded within the container. As a result, only the *service* memory footprint should be checkpointed and restored within another container at the destination without the need to migrate the entire container. However, we note that, this approach desires developers to build a migratable container image in both source and destination systems. Moreover, this approach entails developer involvement in the details of the live migration. In this study, we adopted an existing service-level migration solution, known as FastFreeze container [9], and extended it to work with the orchestrator.

c) Container-level migration approach: The service-level migration approach is not applicable in circumstances where the user does not have the authority to change the service image. As such, we develop the container-level migration as a non-invasive approach that can perform the live migration in a self-sufficient manner.

For that purpose, we leverage the ability of container runtimes to perform container checkpoint/restore independent from the orchestrator. However, this ability alone cannot resolve service migration problem, because upon migration, the destination orchestrator does not recognize and evades from managing the migrated service container. To overcome this problem, our solution is to nest the migratable container within an outer one. On one end, the outer container maintains the binding with the destination orchestrator, and on the other end, it hosts the migrated service as its nested container.

As shown in part ③ of Figure 3, the outer container encompasses a container runtime (*e.g.,* Docker engine), a Checkpoint/Restore module (*e.g.,* CRIU), and a Synchronization module. This arrangement at the source enables the outer container to migrate its nested one as a nested container of a peer outer container in the destination without any orchestrators' involvement. It is noteworthy that the nested container is just a regular container without any specific adjustments that is managed (*e.g.,* in terms of resource usage tracking) by the outer container. To implement the idea of container nesting, we adopt Docker-in-Docker, which is a Docker engine, and deploy it inside the outer container. To synchronize the checkpoint files without any shared storage across systems, we employ the same method explained in the orchestrator-level approach.

D. UMS Implementation

We develop the Coordinator and the Migration Interface as web services. Except in the container-level approach, we pack the Migration Interface into the Docker-in-Docker container. Recall that, in the orchestrator-level approach, the service is a regular container, and in service-level approaches, the service is containerized in FastFreeze-enabled container. We use Rsync over SSH, packed into the Migration Interface, for the data-transfer in the container-level and orchestrator-level approaches. FastFreeze has a built-in data transfer tool called *CRIU Image Streamer* that can stream the checkpoint files to the destination without buffering them in the local storage. The destination container is configured with MinIO, an s3 compatible object storage, to receive the checkpoint files.

IV. Evaluation

Our evaluations encompass three different aspects of the system: (A) We examine the factors contribute to the latency overhead of each migration approach while varying the sizes of container memory footprints; (B) While the other evaluations utilize a benchmarking application with configurable (static) memory footprint, in this part, we inspect the migration performance for a real-world application with a dynamic memory footprint; and (C) We study the feasibility of the live migration across heterogeneous orchestrators and multi-clouds.

We created two VMs to simulate cloud-based computing systems, each one in a different physical machine connected by a 1 Gbps link. Each VM includes 8 vCPUs, 16 GiB memory, 50 GiB storage, and a Kubernetes orchestrator. Lastly, UMS is deployed for each orchestrator on each VM.

In most of the experiments, we deployed a popular benchmarking application called `memhog` [7]. The reason we use `memhog` is that it can be configured with a static memory footprint, which is a decisive factor in the migration performance. We instruct `memhog` to allocate a certain amount of memory (in the range of ≈0—1,024 MiB), write random data to the allocated memory, and print a counter number at every second. For FastFreeze, we assume that the FastFreeze container image is available at both source and destination systems.

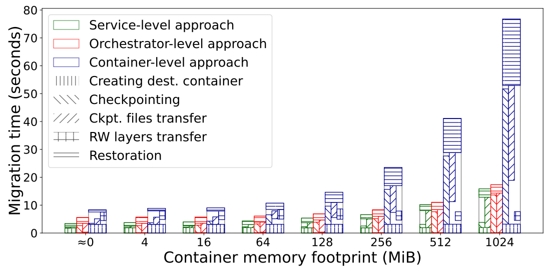

Fig. 4: Detailed live migration time for different approaches across computing systems. The 90% confidence interval is negligible.

A. Analyzing the Overhead of Live Container Migration

In this experiment, our goal is to measure the overhead of live migration across two homogeneous Kubernetes-based systems upon varying the memory footprint of a single-process container, as shown in Figure 4. The overhead measurement metric is the migration turnaround time from the migration request at the source until the container runs at the destination. We conducted the experiment 30 times and reported the breakdown time of each contributing step to the overall overhead.

Unsurprisingly, the chart shows that the service-level approach imposes the lowest overhead as the overhead of checkpoint, transfer, and restore operations for the entirety of the container is more than doing so only for the service process. We observe that the orchestrator-level approach outperforms the container-level approach; however, recall that this is an invasive approach that implies changes in the underlying orchestrator, whereas the other two approaches do not. The overhead difference between the orchestrator-level approach and the container-level approach is ≈37% for small services, *i.e.,* containers with less than 128 MiB memory footprint. This is an important finding knowing that, in practice, the size of a majority of containerized services is less than 128 MiB. In addition, all migration times are considerably high; however, this is common for using cold migration technique [5].

Takeaway: *For single-process containers, the service-level migration outperforms other approaches. Moreover, the migration time of container-level approach is tolerable for small-size (<128 MiB) containers.*

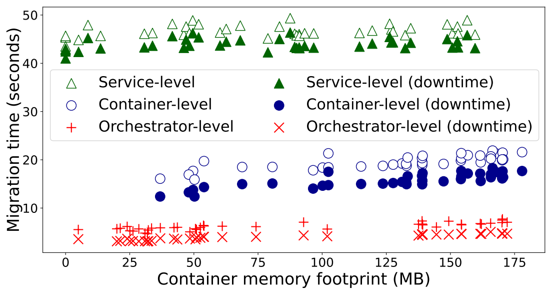

Fig. 5: Migration overhead time and service downtime of `YOLOv3-tiny` for orchestrator-level, container-level, and service-level approaches across Kubernetes orchestrators.

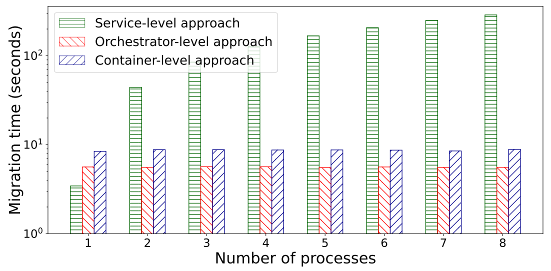

Fig. 6: Live container migration time comprises multiple concurrent processes. The 90% confidence interval is ≈±1.0, ±0.1, and ±0.1.

B. Impact of Dynamic Memory Footprint on the Migration

In this experiment, we aim to study the migration performance of a service with a dynamic memory footprint. We configured `YOLOv3-tiny` [3], a popular object detection application, within a container and fed it with an input image (160 KB) from their repository. The reason we chose `YOLOv3-tiny` is its predictable (linearly increased) memory footprint behavior upon progress in processing the input image.

To measure the downtime, we conducted the experiment 30 times for each migration approach. At each iteration, we randomly chose a time during the inference process and performed the migration. The service downtime was measured by adding a helper process to output a number every second. The downtime is the time interval between the first number printed after the migration and the last number printed before the migration, minus the one-second interval we had by default.

Figure 5 demonstrates that the service downtime depends on the migration approach. We notice that the downtime using the service-level approach is higher than `memhog` counterparts in Figure 4. Our hypothesis is that the reason for the higher downtime is one more process restoration that has to be performed for the helper process. To verify this, we conducted an experiment by configuring `memhog` to spawn 1—8 child processes with negligible (≈0 MiB) memory footprint, and performed the migration similar to Section IV-A.

Figure 6 shows that the service-level approach incurs a significantly higher migration overhead when there is more than one process. Our analysis shows that this overhead is due to FastFreeze internal mechanics. Specifically, it desires to spawn its child process with a predefined PID, which requires access to the kernel file. Without privileges, FastFreeze workaround is to keep spawning processes until it reaches the PID it desires, and this process imposes a constant overhead time at the restoration step. This derives the conclusion that the service complexity (*i.e.,* the number of processes running within a container) and privileges are decisive on the downtime of the service-level approach. For such services, even the container-level approach offers a significantly lower migration time.

Takeaway: *In practice, container downtime of the service-level approach predominantly depends on the privileges and number of processes running in the container, rather than the container memory footprint at the migration time.*

Approach	**Required changing**	K8s	Mesos	K3s	Minishift
Orchestrator-level	Orchestrator	6.94	infeasible	infeasible	infeasible
Service-level	Service	5.94	5.74	5.97	5.96
Container-level	Nothing	14.71	15.03	14.63	14.85

TABLE I: The migration time for a service with 128 MiB memory footprint. The 90% confidence interval is ≈±0.15.

C. Live Migration across Heterogeneous Orchestrators

One interesting use case for the live migration of containerized services is to enable heterogeneous orchestrators and multi-cloud scenarios. This enables enterprises to seamlessly migrate their deployed services to other cloud providers, hence, unlocking the longstanding vendor lock-in problem of the cloud environments, in addition to enabling room for more cost efficiency and service reliability. The service-level and container-level approaches, particularly, fit the public cloud use cases where neither live service migration is supported, nor users are allowed to modify the underlying cloud platforms. As such, the orchestrator-level approach that implies orchestrator-level changes and requires compatibility between systems is evidently not applicable. To implement this use case and evaluate it, we developed UMS on Mesos, K3s, and Minishift, and had them migrate the containerized single-process service to Kubernetes. For the sake of better comparison, we also include the case of Section IV-A[2].

[2]migrating AKS to GKE demo video is at: https://youtu.be/SIfIpPWZuls

Table I shows the mean migration time for each case after 30 live migration attempts. Even though the orchestrator-level approach requires changing in the underlying orchestrator, it cannot migrate (shown as *infeasible*) across autonomous systems, *e.g.,* multi-cloud, with heterogeneous orchestrators. Nevertheless, the container-level and the service-level approaches can cover all the cases. The former comes with the benefit of *nothing* to change in the underlying platforms, whereas the latter entails intervention in the service deployment.

Takeaway: *Live and seamless migration of containerized services is a viable solution to realize the notion of multi-cloud and resolve the problem of vendor lock-in.*

V. Conclusion

In this research, we developed UMS to support seamless and lightweight live migration of containerized services across autonomous systems with potentially heterogeneous orchestrators. UMS is equipped with a spectrum of migration approaches (namely, orchestrator-level, service-level, and container-level) that differ in their imposed overhead, and how liberally they can migrate containers across systems. The results demonstrate that UMS can perform low-overhead service migration across any two computing systems. We concluded that, although the service-level approach is the most lightweight, its performance is downgraded for multi-process containers with insufficient privileges. We also demonstrated a use case for UMS to perform container migration across heterogeneous orchestrators to realize the idea of multi-clouds.

Acknowledgement

This material is supported by National Science Foundation (NSF) under awards# CNS-2007209 and CNS-2047144.

References

[1] Chavit Denninnart, Thanawat Chanikaphon, and Mohsen Amini Salehi. Efficiency in the serverless cloud paradigm: A survey on the reusing and approximation aspects. *Software: Practice and Experience*.

[2] Pawissanutt Lertpongrujikorn and Mohsen Amini Salehi. Object as a service (oaas): Enabling object abstraction in serverless clouds. In *Proceedings of the 16th IEEE Cloud Conference*, Jul. 2023.

[3] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[4] Jakob Schrettenbrunner. *Migrating Pods in Kubernetes*. PhD thesis, 12 2020.

[5] Gursharan Singh, Parminder Singh, Mustapha Hedabou, Mehedi Masud, and Sultan S Alshamrani. A predictive checkpoint technique for iterative phase of container migration. *Sustainability*, 14(11):6538, 2022.

[6] Paulo Souza Junior, Daniele Miorandi, and Guillaume Pierre. Good shepherds care for their cattle: Seamless pod migration in geo-distributed kubernetes. In *Proceedings of the 6th IEEE International Conference on Fog and Edge Computing (ICFEC)*, pages 26–33. IEEE, 2022.

[7] Radostin Stoyanov and Martin J Kollingbaum. Efficient Live Migration of Linux Containers. In *Proceedings of the International Conference on High Performance Computing*, pages 184–193, 2018.

[8] Minh-Ngoc Tran, Xuan Tuong Vu, and Younghan Kim. Proactive Stateful Fault-Tolerant System for Kubernetes Containerized Services. *IEEE Access*, 10:102181–102194, 2022.

[9] Nicolas Viennot. FastFreeze: Unprivileged checkpoint/restore for containerized applications. https://lpc.events/event/7/contributions/642/. Online; Accessed on 7 May 2022.