Migrating Kubernetes Pods Among Geographically Distributed Edge Clusters of a Smart City

Leonardo Poggiani, Carlo Puliafito, Antonio Virdis, Enzo Mingozzi,
Department of Information Engineering, University of Pisa, Pisa, Italy

I. Introduction

Edge computing complements the cloud by providing cloud-like services in close proximity to the final users. An edge computing system is composed of several **edge clusters** (a.k.a micro data centers) that are widely distributed over a geographical area and located in proximity to access networks or within the core network of the telco operator. Each edge cluster comprises one or more edge servers exposing services to the final users. Due to its proximity to users and clients, edge computing is a key enabler of Smart City applications (e.g., smart vehicles and traffic management, augmented/virtual reality, emergency applications) having stringent requirements in terms of latency, throughput, and privacy.

In an edge-computing system, an edge service may need to **dynamically migrate** between geographically distributed clusters. Migration lets the service retain its internal runtime state (e.g., session state of a client, progress of a machine-learning inference) and continue its execution on the destination cluster. Service migration proves useful in several scenarios. For instance, it may be leveraged when one wants to perform cluster maintenance or when there is the need to minimize energy consumption in the edge system. More importantly, service migration may help to preserve proximity between the service and its **mobile client**. Let us consider the following use case, which is depicted in Figure 1. A drone for last-mile delivery traverses a Smart City, moving toward the recipient of the parcel. Along the path, the drone periodically streams video frames of the surrounding environment to a flight-assistance edge service (blue line in figure). This is in charge of detecting objects in video frames and alerting the drone whether it is approaching an obstacle, hence allowing the drone to avoid it. To this purpose, the edge service has to maintain a state related to a window of last processed frames. As shown in figure, when the drone changes point of access to the network, it may get topologically far away from its edge service (dashed blue line). As a result, the service migrates to a different edge cluster. After migration, communication between the two endpoints benefits again from proximity between the two (green line). Migrating an edge service between clusters requires to establish a relationship between the clusters as well as a workflow to collect service state, transfer it and use it to restore service execution at destination.

This extended abstract provides a general overview of a workflow that we presented in [1] for migrating **Kubernetes Pods** between geo-distributed Kubernetes clusters. Kubernetes is the *de-facto* standard for automating service orchestration. In Kubernetes, a Pod represents a service instance. Each Pod may be a single container, namely an executable software unit packaging application code and its dependencies. Nonetheless, a number of design patterns allow a Pod to be the composition of multiple, tightly coupled containers. In this case, the main container runs the application logic while the other container(s) provides additional functionalities (e.g., logging, proxying) without the need to modify the main application code.

Our migration workflow presents the following key characteristics, which distinguish our solution from related works in the field:

- It migrates both single-container and **multi-container** Pods;
- It transfers the checkpoint (i.e., service state) directly between edge clusters, hence avoiding to rely on an intermediary node. Any triangulation would indeed have non-negligible impact especially in edge computing, where edge servers are geo-distributed;
- It relies on standard Kubernetes API and can be therefore used with any vanilla Kubernetes installation.

II. The proposed migration workflow

Figure 2 depicts our proposed workflow to migrate a Kubernetes Pod from a source cluster to a destination one. The figure highlights the components involved (in blue) and the sequence of steps covered (in green). In step ①, a *LiveMigration* Kubernetes object of our design is created on the source cluster, which indicates that Pod migration is needed. A *Source Migration Operator* (SMO) of our design detects this event and starts the migration workflow. Firstly (see step ②), the SMO establishes a peering relationship between the involved

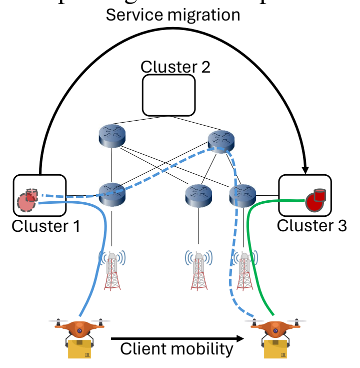

Fig. 1. A service-migration use case to support client mobility.

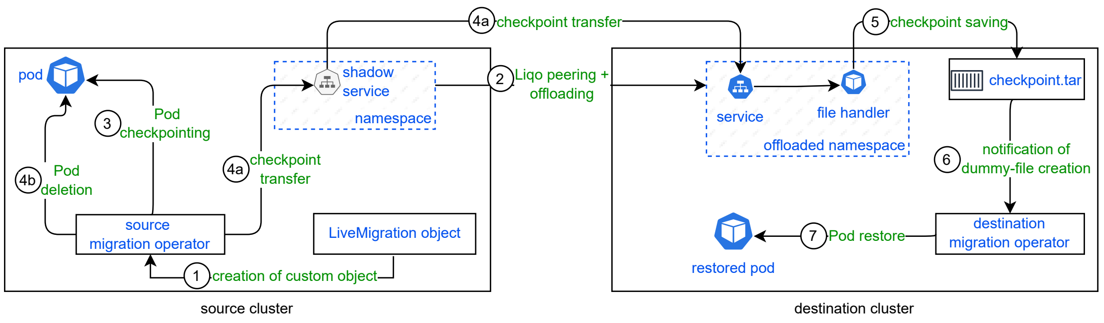

Fig. 2. The proposed migration workflow.

clusters, which is needed later to directly transfer service checkpoint between them. To create such a bond between clusters, the SMO leverages **Liqo**[1]. This is a tool, developed at the Polytechnic University of Turin, for establishing a lightweight federation among Kubernetes clusters. Through Liqo, a destination (or provider) cluster can expose all or part of its resources to a source (or consumer) cluster. By means of a virtualization layer, the source cluster views such resources as internal and can therefore use the standard Kubernetes API to offload Kubernetes objects of any kind to those resources. In particular, in our work we exploit Liqo to offload a Kubernetes *NodePort service* to the destination cluster. Once the NodePort service is offloaded, Liqo creates a *shadow copy* of said service on the source cluster, which allows any component on that cluster to send packets to the offloaded service. Thanks to Liqo and the offloaded NodePort service, the SMO may send the service checkpoint directly to the destination cluster, once the checkpoint is collected.

When the NodePort service is offloaded, a custom controller (not shown in figure) in the destination cluster detects this event. In response to this, the controller creates two Pods at destination. One is a file-handler Pod, which is in charge of receiving the checkpoint, once this is transferred. The other Pod is the *Destination Migration Operator* (DMO), which instead has the role of restoring Pod execution on the destination cluster.

In step ③, the SMO starts the checkpointing operation. If the Pod includes multiple containers, one checkpointing is performed for each container. Container checkpoints are started in parallel to one another. This guarantees consistency among containers, by minimizing the chances that a container fails when trying to interact with another one that has been already checkpointed and therefore stopped. To request a checkpoint, the SMO sends a `POST` request to the `/checkpoint/{namespace}/{pod}/{container}` path exposed by the *Kubelet*. In Kubernetes, the Kubelet is the management component deployed on each worker node of a cluster. Once the Kubelet receives a checkpoint request, it interacts with the underlying container runtime to perform the task. The container runtime is enabled with support to Checkpoint/Restore in Userspace (*CRIU*), namely the actual tool conducting checkpoint and restore of containers. The produced container checkpoint is a `.tar` archive including the container image and the container runtime state. The checkpoint(s) is saved on a persistent volume mounted by the SMO.

In step (4a), the SMO transfers the checkpoint(s) in parallel to the destination cluster. For the purpose, it leverages the NodePort service that was offloaded through Liqo in step ②. Specifically, for each checkpoint, the operator interacts with the shadow copy of the NodePort service by issuing a `POST` request to `http://{serviceIP}:8080/upload`. Hence, checkpoint data flows through the shadow copy and reaches the NodePort service at destination. When all checkpoints are transferred, the SMO creates a *dummy file* and sends it to indicate that checkpoint(s) transfer is completed. In step (4b), the SMO requests Pod deletion at source. This step may be performed in parallel to step (4a) or be postponed. However, it must always take place before restoring the Pod at destination.

The file-handler Pod receives the `.tar` checkpoint archive(s) and the dummy file. In step ⑤, the file-handler Pod writes the received content to a persistent volume that is mounted by both the file-handler Pod and the DMO. Our DMO waits for the creation of the dummy file, which informs that checkpoint transfer is over, and in step ⑥ it is notified of this event. Therefore, in step ⑦ the DMO begins the restore operation. For each of the Pod containers, the DMO builds an executable container image from the `.tar` archive containing the checkpoint. Then, it requests the Kubelet to restore the Pod using those images. However, note that the Kubelet exposes no restore-specific API, unlike the checkpoint case. Instead, the DMO first sends a request to the Kubelet to create a container from each generated image. When all containers are created, the operator asks the Kubelet to start a Pod composed of those containers. Our proposed workflow finishes when the restored Pod successfully runs at destination.

We implemented a Proof-of-Concept (PoC) of our solution, which is available on GitHub at `https://github.com/Unipisa/Pod-migration`.

References

[1] L. Poggiani, C. Puliafito, A. Virdis, and E. Mingozzi, "Live migration of multi-container kubernetes pods in multi-cluster serverless edge systems," in *IEEE 1st International Workshop on Serverless at the Edge (SEATED 2024) co-located with the IEEE 33rd International Symposium on High-Performance Parallel and Distributed Computing (HPDC 2024)*, 2024.

[1]https://liqo.io/