Good Shepherds Care For Their Cattle: Seamless Pod Migration in Geo-Distributed Kubernetes

Paulo Souza Junior, Daniele Miorandi, Guillaume Pierre

Good Shepherds Care For Their Cattle: Seamless Pod Migration in Geo-Distributed Kubernetes

Paulo Souza Junior
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
paulo.souza@inria.fr

Daniele Miorandi
U-Hopper
Trento, Italy
daniele.miorandi@u-hopper.com

Guillaume Pierre
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
guillaume.pierre@irisa.fr

***Abstract*—Container technology has become a very popular choice for easing and managing the deployment of cloud applications and services. Container orchestration systems such as Kubernetes can automate to a large extent the deployment, scaling, and operations for containers across clusters of nodes, reducing human errors and saving cost and time. Designed with "traditional" cloud environments in mind (i.e., large datacenters with close-by machines connected by high-speed networks), systems like Kubernetes present some limitations in geo-distributed environments where computational workloads are moved to the edges of the network, close to where data is being generated/consumed. In geo-distributed environments, moving around containers, either to follow moving data sources/sinks or due to unpredictable changes in the network substrate, is a rather common operation. We present MyceDrive, a stateful resource migration solution natively integrated with the Kubernetes orchestrator. We show that geo-distributed Kubernetes pod migration is feasible while remaining fully transparent to the migrated application as well as its clients, while reducing downtimes up to 7x compared to state-of-the-art solutions.**

***Index Terms*—Stateful Migration, Kubernetes, Geo-Distributed, Containers**

I. Introduction

A leading design principle of modern cloud computing systems is that cloud resources should be treated as cattle rather than pets. In other words, cloud resources such as virtual machines and containers (the cattle), and the applications they contain should be designed so they may fail and be easily replaceable with other ones, without impacting the overall system (the herd). This design principle has proven to be extremely successful for guaranteeing the robustness and stability of many cloud platforms such as the popular Kubernetes container orchestration platform [1].

However, this principle is often interpreted in an excessive fashion such as "containers must remain totally stateless" or "any management action performed on a container turns it into a pet" [2]. On the contrary, an arguably rational management of "cattle" resources may be to care for them and maximize their usefulness, while keeping in mind that they should not create service disruption in case they die – for instance, because of a crash of the server which runs them.

One topic in which "rational" management of container resources may provide tangible benefits without violating the pet/ cattle principle is the choice of which server should be used to execute a given container. In geo-distributed environments such as fog computing platforms, every server may be installed in a different location, and each container may need to run in a specific server [3]. When runtime conditions such as user location change, it may become useful to migrate the concerned container to another location.

For example, Augmented Reality based on mobile devices enables the creation of games such as escape rooms and seeking treasures within an entire city [4]. These kinds of games require very tight interactions between the players' devices and the game application. As the players move within the city, it becomes necessary to migrate the application with its full state to servers ideally reachable from the user device through a single network hop. However, to maintain game continuity, it is also important to reduce the downtime during which the game is unresponsive while being migrated.

This paper presents MyceDrive, a pod migration technique integrated within the Kubernetes container orchestration system. Following the pet/cattle analogy, MyceDrive avoids killing a healthy running pod and waiting for Kubernetes to restart a new one when it is, instead, possible to migrate it to a different server. Migration is totally transparent to the application running inside the pod as well as the clients having open TCP connections to the migrated pod. MyceDrive relies on DMTCP [5] to checkpoint the container's memory state and open network connections at the migration time, and to resume this saved state in the new pod. MyceDrive integrates in regular Kubernetes deployments with no need for any specific CPU architecture, OS, kernel modules or hypervisor.

MyceDrive is designed to operate in geo-distributed environments where each server is placed in a different location,

This work is part of the FogGuru project which has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 765452. The information and views set out in this publication are those of the author(s) and do not necessarily reflect the official opinion of the European Union. Neither the European Union institutions and bodies nor any person acting on their behalf may be held responsible for the use which may be made of the information contained therein.

and network links between servers may be slow. Even in such difficult operating conditions, we show in our evaluations that it introduces low service downtimes during pod migration, up to 7x faster than state-of-the-art technologies.

This paper is organized as follows: Sections II and III present the technical background and related work. Section IV discusses design and implementation. Section V evaluates MyceDrive and, finally, Section VI concludes.

II. Background

a) Kubernetes: Kubernetes is a container orchestration platform that automates the deployment, scaling, and management of containerized applications in large-scale computing infrastructures such as a cluster and a datacenter [6]. It relies on container runtime systems such as Docker, and it is in charge of creating, deploying, and running containers within a group of server machines.

A Pod is the smallest scheduling unit in Kubernetes. It consists of one or more containers and possibly data volumes. Kubernetes guarantees that the containers which belong to a pod execute in the same machine and share the same set of resources such as a single private IP address within the cluster.

Following the cattle principle, a pod is expected to be mortal and may fail at any time. By default, when a pod stops, its data volumes are deleted, and its private IP address is recycled to be assigned to a new pod. Declaring the pod as a StatefulSet makes these resources persistent, so a newly created pod may later re-attach to them.

To make a set of Pods available from the outside world, a Service acts as a single public network interface that creates internal networking routes to the concerned pods [7].

A Deployment describes the desired state of an application. The Deployment Controller is in charge of monitoring the actual application state and of resolving any discrepancy between the desired and the observed state, for example, by adding or removing application pods [8]. Deployments may be updated by their administrators at any time, which typically triggers the Deployment Controller to create or delete pods.

b) DMTCP: DMTCP (Distributed MultiThreaded CheckPointing) is a user-level checkpointing package for distributed applications [5]. It can checkpoint a group of processes and later recreate them. DMTCP works in user space and requires no modification to the processes' binary nor the Linux kernel.

Snapshotting a group of processes requires one to start a DMTCP Coordinator on the host machine, and then launch the concerned processes by replacing their startup command with as "`dmtcp_launch [command]`". The `dmtcp_launch` prefix registers the process with the coordinator and wraps some of the application's library and system calls to track the creation of new threads or processes, OS resources such as locks and open files, and network sockets.

When the coordinator receives a snapshot request, it creates a consistent dump of the concerned processes' memory and OS resources into a single gzipped file that can later be used to restart the running application in the same or another machine.

III. Related Work

a) Kubernetes-based migration: In Kubernetes, the simplest way to migrate a running pod is to stop it, then redeploy a new one in a different server. The disk state of the deleted pod may be preserved by declaring it as a StatefulSet and by re-attaching the preserved data volume in the newly-created pod [6]. The StatefulSet requires the declaration of at least one PersistentVolume that is used to keep the disk state. The new pod may be accessed by its users using the same IP address as the old one by exposing the pod using a Kubernetes Service. From Kubernetes' point of view, this fully respects the "cattle" principle as the migration procedure treats pods as disposable units which may be deleted and replaced at any time.

However, from the application's point of view, this migration procedure presents two major weaknesses. First, it requires the application to be designed in such a way that it immediately dumps its entire runtime state to disk upon receiving the SIGTERM signal. Any unsaved state (e.g., a variable maintained in memory) cannot be recovered after migration. This is a significant issue considering that most workloads in Kubernetes exploit standard third-party software such as Redis, Postgres and, ElasticSearch [9] which may or may not have this capability.

Second, stopping a running pod at a random time implies breaking the open TCP connections between the pod and its end-users at the time of the migration. This means that pod migration is visible from the external world, and possibly creates inconsistencies between the clients and the server pod because the clients have no way of determining whether their latest request could be executed before the pod failure [10].

In geo-distributed environments, simply re-attaching a disk volume after pod migration would imply long-distance remote access to the migrated pod's data, which may negate the benefits of the migration in the first place. Incremental volume checkpoint techniques may be used to improve the speed of geo-distributed data volume migration [11]. In this paper, we do not address this topic and instead focus on the complementary problem of migrating other pod resources such as memory state and networking connections.

b) LXD container migration: LXC is the native container runtime system in Linux environments. Its improved version called LXD supports container migration either by using built-in libraries [12], [13] or by relying on CRIU (Checkpoint/Restore In Userspace) [14], [15].

c) CRIU-based migration: CRIU is a Linux kernel module to snapshot and later restart the contents of a container's memory pages, open files, etc [16]. The snapshot does not contain the entire container image but only the modifications within. It is, therefore, necessary to have the same image in the destination node to restart from the snapshot. The container stays unavailable while being snapshot and during state transfer until a new container is created.

Multiple migration systems exploit CRIU. For instance, it is used to migrate MPI applications to improve workload resilience to anticipated hardware failures [17]. Docker-based edge computing environments may also be used for checkpointing, suspending, and potentially migrating long-running blocking FaaS functions [18]. Finally, H-Container migrates containerized applications across computing nodes of different ISA architectures by adapting LLVM using CRIU [19].

The work in [20] proposes Redudancy Migration to reduce the migration downtime by buffering incoming network packets during migration and replaying them on the migrated container. This allows one to avoid the necessary time for client machines to detect packet loss and retransmit. This work shows that container migration does not necessarily imply breaking open TCP connections at the migration time. However, it relies on the active participation of client nodes to update their routing rules during migration. Conversely, we aim at making migration fully transparent for the client nodes.

Although CRIU has been used in multiple container migration systems, it features two significant limitations which impact all migration techniques based on it. First, CRIU is a Linux kernel module that requires modifying the OS of the cluster's server machines. It supports only a small number of kernel versions, particularly on ARM processors, which may create conflicts with other platform requirements. Second, CRIU is currently not able to checkpoint and recreate open network connections transparently to the clients. This implies that open network connections to client machines are necessarily broken upon container migration. Finally, no CRIU-based container migration is currently available for Kubernetes as an integrated tool for pod migration.

d) DMTCP-based migration: To our best knowledge, we are the first to exploit DMTCP for migrating containers, as DMTCP's original motivation is to provide fault-tolerance properties to running processes. Compared to CRIU, DMTCP has two main advantages. First, it runs entirely in userspace and is therefore agnostic to the Linux kernel version. Second, it can checkpoint and recreate network socket state, which give us the opportunity to maintain open connections with the clients machine during migration.

IV. System Design

Migrating a Kubernetes pod from one server to another is conceptually very simple. In principle, one simply needs to stop and checkpoint the memory state and system-level resources of the "source" pod, transfer the checkpoint data to the destination server, then restart a new pod from this checkpoint. We do not address the migration of the pod's disk volumes as this was the topic of a separate paper [11].

As illustrated in Figure 1, MyceDrive's architecture is composed of two elements. First, an Execution Agent (EA) is integrated into every application container. It controls the container's lifecycle, such as triggering a checkpoint and restarting from a saved checkpoint. Second, a Migration Coordinator (MC) is deployed out of the application pod (for example, in the server running the Kubernetes Control Plane). It is in charge of interacting with the Kubernetes API to start or finish pods and coordinating the migration by sending requests to the EA. To issue checkpoint and restart commands, the

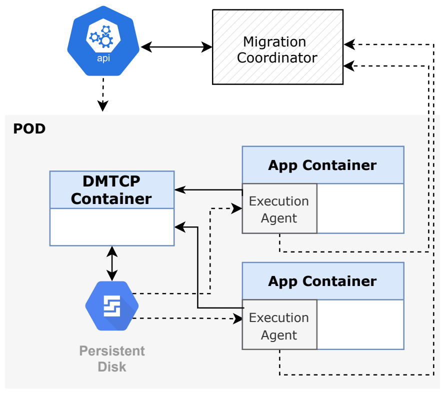


Fig. 1: MyceDrive System architecture.

```
1  containers:
2    - name: nginxcontainer
3      image: enrichednginx
4      ports:
5      - containerPort: 80
6      volumeMounts: # Shared volume with DMTCP binaries
7      - name: dmtcp-shared
8        mountPath: /dmtcp
9      env: # Environment variables with EA's configuration
10     - name: MIGR_COOR
11       value: coord.api  # hostname to reach the coordinator API
12     - name: START_UP
13       value: "/usr/sbin/nginx -g 'daemon off;'"
14       # Application startup command to be wrapped by DMTCP
15     lifecycle:
16       # End script to make sure the checkpoints are created
17       preStop:
18         exec:
19           command: [ "./end_container" ]
20   - name: dmtcpcontainer
21   # Container running dmtcp_coordinator
22     image: dmtcp:dev
23     env:
24     - name: DMTCP_CHECKPOINT_DIR
25       value: /dmtcp/checkpoints
26     volumeMounts: # Shared volume with the DMTCP binaries
27     - name: dmtcp-shared
28       mountPath: /share
```

Listing 1: Pod specification file to enable MyceDrive.

EA interfaces with a lightweight DMTCP container which runs the `dmtcp_coordinator` in charge of managing every application process and creating the checkpoint.

A. Execution Agent

To support migration, pods must include an EA in each container and an additional DMTCP container per pod. This way, every process gets started using the `dmtcp_launch` command. This can be done with a few simple modifications in the pod specification file, as illustrated in Listing 1.

When a container starts, it executes an "entry point" script in charge of spawning one or more processes within the container. This entry point is included in the container image together with all the files needed to execute it. To allow DMTCP to checkpoint these processes, we need to start them using the DMTCP wrappers. Lines 2-5 of the pod specification request the creation of a container running an enriched version of the standard Nginx image. This can be done by exploiting the layered structure of Docker container images [21]. Instead of modifying the container image, we create an additional layer containing the `dmtcp_launch` entry point script that overrides the original one.

To execute DMTCP in the application containers, we need to make DMTCP's binaries and libraries available in these containers. This applies to every application container within the pod, as well as the additional DMTCP container. Unfortunately, this represents a total size of about 200 MB. We prefer not adding these files in the image layer itself as this would unnecessarily increase the image size and potentially slow down the pod deployment process. Instead, we group the necessary files in a read-only data volume that is pre-staged in the worker node and mounted by every concerned process. Lines 6-8 in the pod spec request this volume mount in the `dmtcpcontainer` container.

The new container entry point is a generic script that must be able to wrap any created process. It, therefore, needs configuration containing the list of processes it should start and the address of the Migration Controller where these processes should connect. Lines 10-14 of the pod spec provide these configurations. The EA uses line 13 to start the container application process. It then remains inactive until requested to checkpoint the pod.

The application containers need to issue the checkpointing command when requested to do so. To maintain the consistency of the pod state before and after migration, the checkpointing operation and the container shutdown must be issued atomically. This is done by triggering the `end_container` command as the last operation to be executed before the container stops. This command creates the checkpoint and informs the MC that it can be transferred to the destination node. Lines 15-19 specify this.

The final part (lines 20-28) requests the creation of the container running the DMTCP coordinator. Similar to the application container, this container mounts the read-only volume with access to the DMTCP binaries and libraries.

When the pod starts, it creates the nginx and the dmtcp containers with their respective volume mounts and entry points. The entry point of the nginx container starts nginx using the `dmtcp_launch` wrappers which register details about the process with the MC such as the process name, status, and meta-data. Any further process forked by the application process automatically inherits the same wrappers.

B. Migration Controller

Migrating a pod requires one to coordinate multiple actions to be performed in the source and destination nodes. We also need to preserve the information about the migration, pod status, and meta-data despite the fact that the pod is about to be deleted. This is the role of the Migration Controller (MC).

The MC is a REST API which needs to run in one node of the Kubernetes cluster. A single MC deployment can manage an entire Kubernetes cluster regardless of the number of pods. It provides the following methods:

register registers a new application container. The MC distinguishes normal containers being started within a pod from migrated containers by comparing the container's labels and whether the container is marked for migration with a list of already-registered containers.

remove removes a registered container, receiving as argument the container name and labels. It returns whether this container is used in migration or not, informing if it should create a checkpoint or proceed with termination.

migrate initiates a pod migration. It receives as parameters the source node, the destination node, a Kubernetes label used to constrain the choice of node where the new pod must be started, and the labels of the application deployment and names of its containers.

copy notifies the MC that a checkpoint is ready to be moved. The MC then uses the Kubernetes copy routine to move the checkpoint between containers, and returns a confirmation that the checkpoint was correctly transmitted.

Pod migration may be requested by the users or administrators by calling the `migrate` method of the MC.

C. Keeping network connections open

Making pod migration transparent to the client processes requires maintaining open network connections across the migration operation. This essentially requires three properties: (i) preserving the TCP socket state such as port numbers, TCP sequence numbers and buffered incoming/outgoing packets; (ii) routing packets from/to the client machines without changing the pod's public IP address; and (iii) reducing the migration downtime as much as possible to avoid connection timeouts.

Preserving socket state: DMTCP was designed to checkpoint not only individual processes but also entire distributed applications such as MPI, which maintain long-lived network connections between the processes. Contrary to CRIU, DMTCP checkpoints socket state in the same way it checkpoints file descriptors, pipes, signal handlers and semaphores.

Maintaining network routes: Kubernetes dynamically assigns a unique private IP address to every running pod. Since the source pod is still running when creating the destination pod, it is impossible to give the new pod the same private IP address as the old one. On the other hand, Kubernetes Services enable users to provide a stable public IP address that acts as a load-balancer for a group of pods. Services are not implemented using a proxy process but as a set of network routing rules injected in the kernel of all worker nodes in the cluster. When a Service detects a change in the set of pods, it triggers a corresponding reconfiguration of these internal routes. As illustrated in Figure 2, we place the pods to be migrated behind a Service that ensures that client machines can keep communicating with the new pod using the same public IP address as the old pod.

Reducing the downtime: A Kubernetes service must first *detect* the creation of the new pod before a notification can be issued to request the creation of new internal network routes. It may therefore take up to 10 to 30 seconds before new networking routes are created. This creates long downtimes as perceived by the clients, and it possibly breaks networking connections due to TCP timeouts.

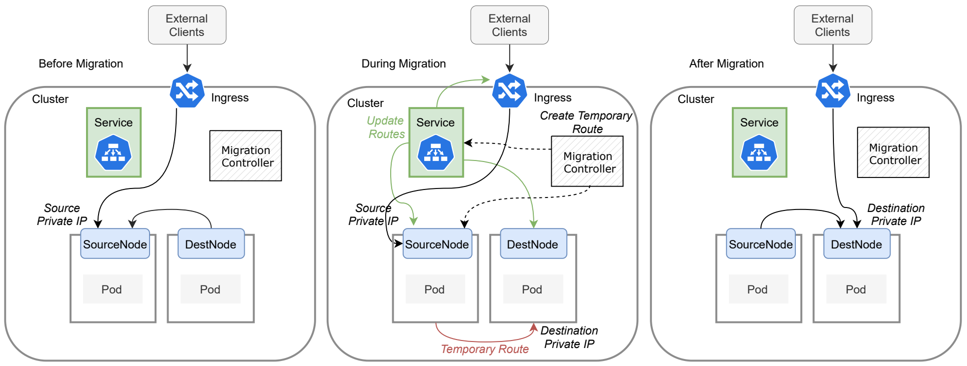


Fig. 2: Internal networking routes before, during and after migration.

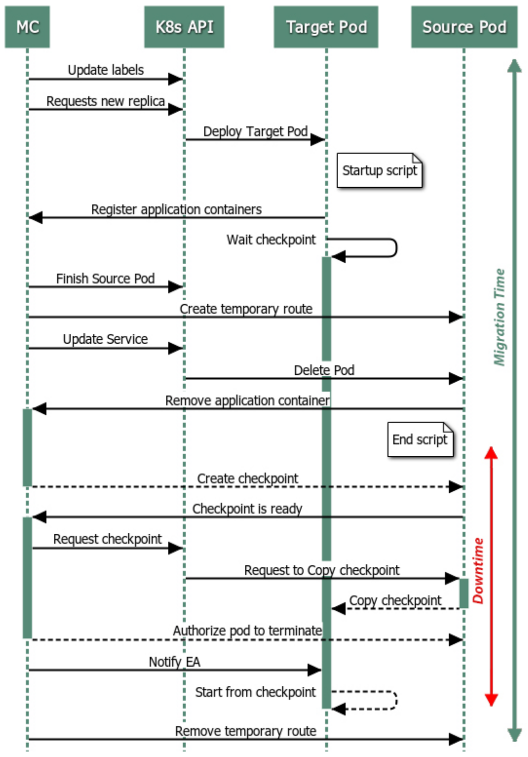


Fig. 3: Pod migration process.

Instead of waiting until the Service discovers the new pod, we thus explicitly notify the Service via the K8s API by updating its labels, forcing an immediate update of the entire Service. However, there remains a period of time when both pods co-exist, and incoming network traffic may be routed to one or another. Therefore, the EA in the source node also injects a temporary route so any traffic received by the source pod gets re-routed to the new pod (see Figure 2). This temporary route is removed after the source pod has been deleted and internal routes have been re-established.

D. Migration coordination

Figure 3 illustrates the different actions that are required to migrate a pod after the `migrate` method of the MC is called.

The first step of the migration procedure is dedicated to preparing the migration as well as ensuring that Kubernetes creates the destination pod in the chosen destination node: we update the labels attached to the Kubernetes Deployment to specify that only two nodes (the source and destination nodes) are acceptable to run pods of this application, and by adding an anti-affinity rule which states that the pods should be placed in different machines.

We then update the Deployment a second time to request the creation of a new pod replica for this application. Because of the placement constraints, Kubernetes can only choose to deploy this new pod in the destination node. The new pod starts its DMTCP container as well as its application container(s). The entry point of the application container(s) registers the containers with the MC. The `register` method identifies that this is a migrated container because it already has a registered container under the same ID. It then blocks the call until a snapshot has been created and copied to the destination node, which effectively delays the launch of the application container.

Third, the MC triggers the network route updates by requesting the source node to inject a temporary route to the destination node and by updating the Service, so it immediately notices the new pod. Up until this point, the source pod keeps running normally.

Fourth, the MC requests the K8s API to terminate the source pod. This triggers the end script to be executed. The script calls the `remove` method in the MC, and detects if it should snapshot the pod before terminating or if this is a normal pod termination that does not require a snapshot. Once the snapshot has been created and compacted, the EA notifies the MC, which then initiates the copy of this checkpoint from the source to the destination node via the K8s API.

Finally, the MC's `register` method returns the checkpoint name and authorizes the target pod's entry point to restart the application containers from the checkpoint.

V. Evaluation

A. Experimental setup

We evaluate this work using a fog computing testbed composed of five Raspberry Pi (RPI) single-board computers model 3 B+ with quad-core 1.2 GHz CPU, 1 GB of RAM and a 32 GB micro-SD storage card. This type of machine

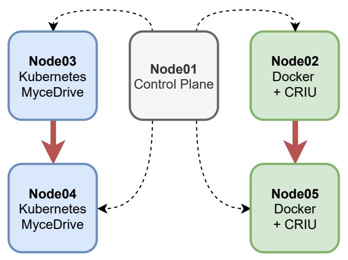


Fig. 4: Testbed organization.

TABLE I: Functional comparison.

	K8s	CRIU	Myce-Drive
Stateful migration	✘	✔	✔
Supports any kernel version	✔	✘	✔
Maintains open network connections	✘	✘	✔
Transparent migration for the application	✘	✔	✔
Transparent migration for the clients	✘	✘	✔
Uses unmodified application images	✔	✔	✘

is frequently used to prototype fog computing platforms [22], [23], [24] The RPIs use the HypriotOS Linux distribution, Kubernetes 1.16 and Docker 19.03.5.

To compare MyceDrive-driven and CRIU-driven migration, we organize the infrastructure with four workers nodes and a control plane node. As illustrated in Figure 4, two workers run Kubernetes while two others have only Docker and CRIU. The control plane node is used to coordinate migration either between the two Kubernetes nodes using MyceDrive, or between the two Docker nodes using CRIU. The two sets of worker nodes use slightly different Linux kernel configurations because CRIU and Kubernetes require the usage of mutually-incompatible kernel modules.

Although MyceDrive can handle multiple pod migrations simultaneously, for the sake of evaluation, we perform a single pod unit migration at a time.

B. Comparison and metrics

We compare three different migration techniques using three real-world applications.

a) Migration techniques: We contrast MyceDrive with two other techniques. First, K8s migration consists of simply requesting the Kubernetes API to terminate the running pod and to create a new one in the selected node.

The second migration technique relies on CRIU to migrate Docker containers. We use the same hardware, system-level, and application-level configurations as for the experiments based on Kubernetes. The main exception is that CRIU requires a different set of Linux modules that had to be built separately. We also do not include the additional image layer required by MyceDrive, considering that CRIU migration does not require this layer. Finally, since DMTCP compresses its checkpoints, we compress CRIU checkpoints as well.

Note that these three container migration techniques are not functionally equivalent. As illustrated in Table I, K8s migration is stateless as it stops the application in the source pod before restarting it in the destination. The only way to preserve the state is to request the application to save its state to the disk when being stopped. Migration is therefore not transparent for the application. This technique does not allow one to maintain open network connections during migration, and therefore, it is not transparent for the clients. On the other hand, CRIU-based migration checkpoints the application's memory state. CRIU imposes strong constraints on the choice of Linux kernel and modules. It also does not maintain open network connections. Its migration is therefore transparent for the application but not for the clients. Finally, MyceDrive is the only migration scheme that combines all these good properties at once. On the other hand, it requires creating modified container images with the additional DMTCP layer.

b) Available bandwidth: In a geo-distributed environment, the servers are located close to the end-users but necessarily far from one another. We evaluate migration with limited available bandwidth between the nodes using typical values found in real-world edge computing environments [25], reshaped using the Linux tc (traffic control) command.

c) Migrated applications: We base our evaluation on three representative applications of fog workloads [26]. The first one is based on MongoDB whose storage engine uses both in-memory and disk storage. We exercise it using a single client that acts as a sensor producing temperature readings in Celsius with a timestamp. The randomly generated data are produced every 1 ms, and then inserted into the database.

The second application is a Redis in-memory data store that we exercise using the same workload as for MongoDB.

Lastly, we use the Mosquitto MQTT message broker with a producer and a consumer that simulate a sensor producing data and an application processing them. We use the same data generated for the other applications.

All three applications maintain and frequently update a majority of their state in main memory, which constitutes the most difficult scenario for stateful container migration. They also maintain long-lived open network connections to their clients, which implies that breaking network connection is treated as a server failure and has a strong negative impact on client-perceived QoS.

d) Evaluation metrics: Migration is typically evaluated using two main metrics. *Migration time* is defined as the entire duration from the moment migration is initiated by the administrator until the last operation is complete. Migration time includes the duration of operations such as pulling container images, exposing services, etc. This metric is important for the administrators as it defines the duration of a complex reconfiguration which may require additional resources compared to simply running containers.

The second metric is *downtime*, which measures the time during which the container does not serve any client workload. We measure downtime from the clients' point of view by checking if the application is reachable and whether it answers client requests. Downtime is important for the application and its clients as it defines the time during which the application is not performing its normal operations.

TABLE II: Checkpoint sizes in original and compressed size.

Migration technique	CRIU	MyceDrive	K8s
MongoDB	41 MB (6.9 MB)	3.1 MB (2.1 MB)	–
Mosquitto	38.2 MB (5.9 MB)	3.2 MB (3 MB)	–
Redis	18.5 MB (5.1 MB)	1.8 MB (1.7 MB)	–

C. Migration performance

Figure 5a shows the migration time for every application and tool over different bandwidth conditions. CRIU has the longest migration time when running MongoDB and Mosquitto applications, followed by MyceDrive and then K8s. In all these cases, migration time decreases when more bandwidth is available between the nodes. For Redis, the migration times follow the same pattern using CRIU-based migration, but they remain mostly constant for the other two migration techniques, with large standard deviations in the case of MyceDrive. This indicates that migration time is dominated by factors other than the memory snapshot transfer. We believe this is due to deploying the target pod before transferring the snapshot.

Figure 5b presents the migration downtimes as perceived by the clients. In CRIU-based migration, the downtimes are mostly equal to the migration times because all migration operations occur while the migrated container is stopped. On the other hand, we observe that downtimes are significantly smaller in the case of MyceDrive. This is due to the fact that MyceDrive delays the termination of the source pod as long as possible until the snapshot has been created. Also, the strategies for keeping network connections open discussed in Section IV-C reduce the downtime as perceived by the clients.

In the case of Redis, MyceDrive observes a very low downtime compared to the migration time. This is consistent with the observation that migration time is dominated by the image download and starting operations rather than snapshot transfer. There as well the available network bandwidth does not have a significant influence on downtimes.

Finally, K8s migration produces the shortest downtimes. However, K8s migration is stateless, so no memory snapshot is copied during migration. K8s migration requires developers to redesign their applications to be able to lose their memory state with no ill effect, which can often be a difficult operation [27].

The two stateful migration techniques experience significant downtime reduction when more network bandwidth is available. Greater bandwidth ensures faster interactions between the servers and reduces the time needed to copy the snapshots from the source to the destination server. Table II shows the respective uncompressed and compressed checkpoint sizes: DMTCP generates smaller snapshots, even though CRIU snapshots seem to obtain better compression ratios.

MyceDrive's downtimes vary largely from one application to another. The main factor determining downtime is the compressed snapshot's size, combined with the available network bandwidth to transfer the snapshot. The largest DMTCP snapshot belongs to Mosquitto, which also happens to have the largest downtimes combined with the greatest variability due to different available bandwidths.

D. Resource usage

Pod migration is a complex operation that requires the system to convey additional tasks while continuing to process its normal workload. We now evaluate the additional resource usage caused by migration, compared to K8S as the baseline of resource usage for managing pods and containers. We measure CPU and memory usage using the `dstat` tool. This section reports only the resource usage measured during migration with 3000 kbps available bandwidth, as there was no meaningful difference with the other evaluated bandwidths.

Figure 6a shows the average CPU usage of each node during migration, in every possible scenario with the three migration techniques and the three evaluated applications. Node1 has the highest CPU utilization in all scenarios, as it runs the control plane responsible for managing the migration. CRIU performs container migration from Node2 to Node5. We can see an increase of about 2.5% CPU usage in those nodes whenever the migration is triggered. On the other hand, K8s and MyceDrive migrate pods from Node3 to Node4. We can see an increase of CPU usage in the order of 4% to 5% during migration, with no significant difference between K8s and MyceDrive. We conclude that this is the normal CPU cost of starting and stopping Kubernetes pods, and that MyceDrive does not generate any significant increase in CPU usage compared to K8s. Finally, we note that the CPU usage remains identical regardless of the application.

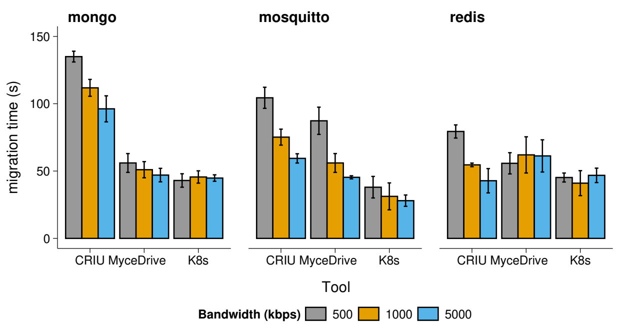
Figure 6b shows the additional memory usage compared to running regular containers or pods without migration. We can see that Node1 bears the greatest cost, with an additional 94 MB when migrating the Mongo application using MyceDrive. Other applications observe similar numbers. This is the memory footprint of the MC to manage migration within a Kubernetes cluster. On the other hand, CRIU-based migration does not require a complex migration controller and can be scripted instead, resulting in a lower memory footprint in Node1. Finally, K8s migration incurs the lowest memory consumption as no additional software needs to be deployed in the control plane node to organize migration. MyceDrive also generates a slightly greater memory usage in the worker nodes involved in the migration, which corresponds to the memory footprint of the EA attached to every pod.

We conclude that MyceDrive incurs almost no additional CPU costs compared to CRIU and K8s migration, and that the memory footprint of the MC and the EAs remain reasonable.

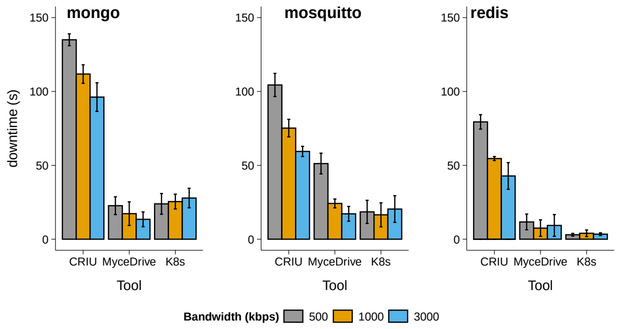
VI. Conclusion

Migration is an essential functionality in large-scale virtualized platforms. It is difficult in particular in geo-distributed environments because limited network capacity between the nodes slow down the migration operations and potentially impose unacceptably long downtimes.

We proposed MyceDrive, which implements stateful and fully transparent container migration in geo-distributed environments. MyceDrive is integrated with Kubernetes and can

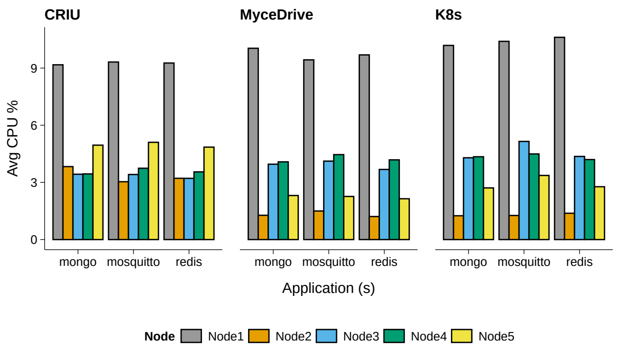


(a) Total migration duration.

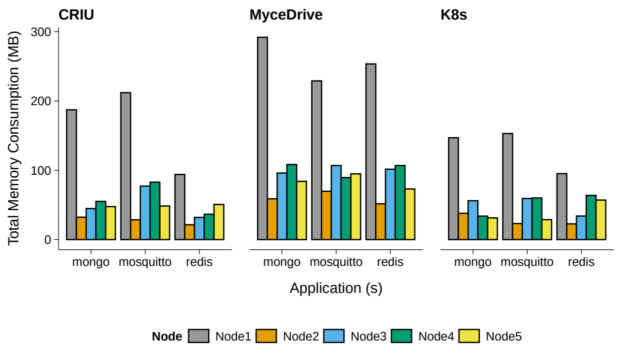


(b) Migration downtime.

Fig. 5: Migration times.



(a) Average CPU usage during migration.



(b) Total memory usage during migration.

Fig. 6: Resource usage during migration using 3000 kbps bandwidth.

be used to migrate entire Kubernetes pods rather than single containers. We showed that, although the total pod migration times are similar to those achieved by CRIU-based migration, it exhibits downtimes up to 7x shorter than those from CRIU.

This work demonstrates that the pet/cattle principle does not necessarily impose terminating a perfectly running pod and restarting a new one from scratch elsewhere when workload relocation becomes necessary. Instead, it is perfectly possible to migrate pods from one node to another without requiring the application to shut down. Good shepherds care for their cattle, and they migrate their herds from one pasture to another when the seasons change.

REFERENCES

[1] Atomist blog, "Kubernetes clusters: Pets or cattle?" Aug. 2019, https://blog.atomist.com/kubernetes-clusters-pets-or-cattle/.
[2] R. Bias, "The history of pets vs cattle and how to use the analogy properly," Cloudscaling, Sep. 2016, https://bit.ly/3BaROTm.
[3] A. Fahs, G. Pierre, and E. Elmroth, "Voilà: Tail-Latency-Aware Fog Application Replicas Autoscaler," in *Proc. IEEE MASCOTS*, Nov. 2020.
[4] B. Sumner, "City-wide augmented reality gaming," https://gtc.inf.ethz.ch/research/city-wide-ar-gaming.html, 2016.
[5] J. Ansel *et al.*, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *Proc. IEEE PDP*, 2009.
[6] K. Hightower, B. Burns, and J. Beda, *Kubernetes: Up and Running Dive into the Future of Infrastructure*, 1st ed. O'Reilly Media, Inc., 2017.
[7] A. Fahs and G. Pierre, "Proximity-aware traffic routing in distributed fog computing platforms," in *Proc. IEEE/ACM CCGrid*, May 2019.
[8] V. Chemitiganti, "Kubernetes concepts and architecture," Platform9 blog, May 2019, https://bit.ly/3JX3w9D.
[9] Datadog, "11 facts about real-world container use," Nov. 2020, https://www.datadoghq.com/container-report/.
[10] M. van Steen and A. S. Tanenbaum, *Distributed Systems*, 3rd ed., 2017, ch. 8.3, https://www.distributed-systems.net/index.php/books/ds3/.
[11] P. Souza Junior, D. Miorandi, and G. Pierre, "Stateful container migration in geo-distributed environments," in *Proc. IEEE CloudCom*, Dec. 2020.
[12] V. Gite, "How to move/migrate LXD VM to another host on Linux," Apr. 2021, https://bit.ly/3aXIbwB.
[13] R. de Jesus Martins *et al.*, "Virtual network functions migration cost: from identification to prediction," *Computer Networks*, vol. 181, 2020.
[14] Y. Qiu *et al.*, "LXC container migration in cloudlets under multipath TCP," in *Proc. COMPSAC*, 2017.
[15] S. Pickartz *et al.*, "Migrating LinuX containers using CRIU," in *Proc. HiPC*, 2016.
[16] OpenVZ team, "CRIU," https://criu.org/.
[17] M. Sindi and J. R. Williams, "Using container migration for HPC workloads resilience," in *Proc. HPEC*, 2019.
[18] P. Karhula, J. Janak, and H. Schulzrinne, "Checkpointing and migration of IoT edge functions," in *Proc. EdgeSys*, 2019.
[19] A. Barbalace *et al.*, "Edge computing: The case for heterogeneous-ISA container migration," in *Proc. ACM VEE*, 2020.
[20] K. Govindaraj and A. Artemenko, "Container live migration for latency critical industrial applications on edge computing," in *Proc. ETFA*, 2018.
[21] G. Rotsaert, "Docker layers explained," Mar. 2019, https://dzone.com/articles/docker-layers-explained.
[22] P. Bellavista *et al.*, "Feasibility of fog computing deployment based on Docker containerization over RaspberryPi," in *Proc. ACM ICDCN*, 2017.
[23] M. Tamiru, G. Pierre, J. Tordsson, and E. Elmroth, "mck8s: an orchestration platform for geo-distributed multi-cluster environments," in *Proc. IEEE ICCCN*, Jul. 2021.
[24] A. van Kempen, T. Crivat, B. Trubert, D. Roy, and G. Pierre, "MEC-ConPaaS: An experimental single-board based mobile edge cloud," in *Proc. IEEE Mobile Cloud*, Apr. 2017.
[25] S. Noghabi *et al.*, "The emerging landscape of edge computing," *GetMobile: Mobile Computing and Communications*, vol. 23, 2020.
[26] A. Ahmed, H. Arkian, D. Battulga, A. J. Fahs, M. Farhadi, D. Giouroukis, A. Gougeon, F. O. Gutierrez, G. Pierre, P. R. Souza Jr,

M. Ayalew Tamiru, and L. Wu, “Fog computing applications: Taxonomy and requirements,” 2019, http://arxiv.org/abs/1907.11621.

[27] J. Adersberger *et al.*, “Patterns and pains of migrating legacy applications to Kubernetes,” Open Source Summit, 2019, https://bit.ly/3G99B0I.