Live Pod Migration with CRIU, User Space Checkpoint and Restore in Linux

Supervisor: **Berat C. Senel**

Student: **Ufuk Bombar**

Contents

1 Introduction

A modern Operating System manages the processes. These processes mainly consist of programs the users want to execute. In an environment with multiple users, Operating System should be able to ensure the processes are executed sufficiently, efficiently, and just. When these processes request access to the hardware resources of the system, the problem of managing processes and resources becomes even more difficult. There are many implementations of different purposed Operating Systems such as *Linux*, *macOS*, *Windows*, *OpenBSD*, etc.

In recent years a new computation paradigm has shifted the focus to the cloud. However, this meant the processes of different users had to run on the same machine. Application developers have begun to containerize their applications. Containerization enabled cloud providers to manage large numbers of processes with little overhead.

There is also a new paradigm on the horizon named Edge Computing. To reduce the latency, containerized applications should run physically closer to the users. Which meant the processes should run on resource-limited infrastructure. Containerized applications are seen as a valid solution for these resource-limited environments since they have a very limited overhead compared to other solutions such as virtual machines. Besides isolation and security aspects, and the friction of containerizing the legacy workloads, there exists another core research topic named container migration. Since in a dynamic edge environment, the scheduler may change the location of the deployment of the workload, the containerized application must migrate to a different host without losing its state to ensure minimal downtime of the application [19].

In this project, the previous work on live migration will be shown, the development plan will be given and the progress of the live pod migration system on the Kubernetes will be presented.

2 Specifications

Creating an application for a distributed system requires knowledge about a lot of intricate details. In this section, I will present the technologies that will be used to have a proper implementation of the live migration system.

Most of the codebase in distributed system technologies are implemented with the Go Programming language. This is because Go is designed to be a system programming language. It employs a set of features for making developing easier for the developer and the program efficient and fast. One of the features that make the deployment rather easy is the garbage collector. Go, developers do not have to track the memory allocations, and this results in a shorter time spent on the implementation stage of the projects. In addition to that, Go has rich support for lightweight threads. Go standard library comes with a variety of synchronization tools and channels for enabling easy communication of these threads. This is why, in most distributed system projects like Kubernetes, etc. Go is used. Thus the Go Programming language will be used to implement live migration.

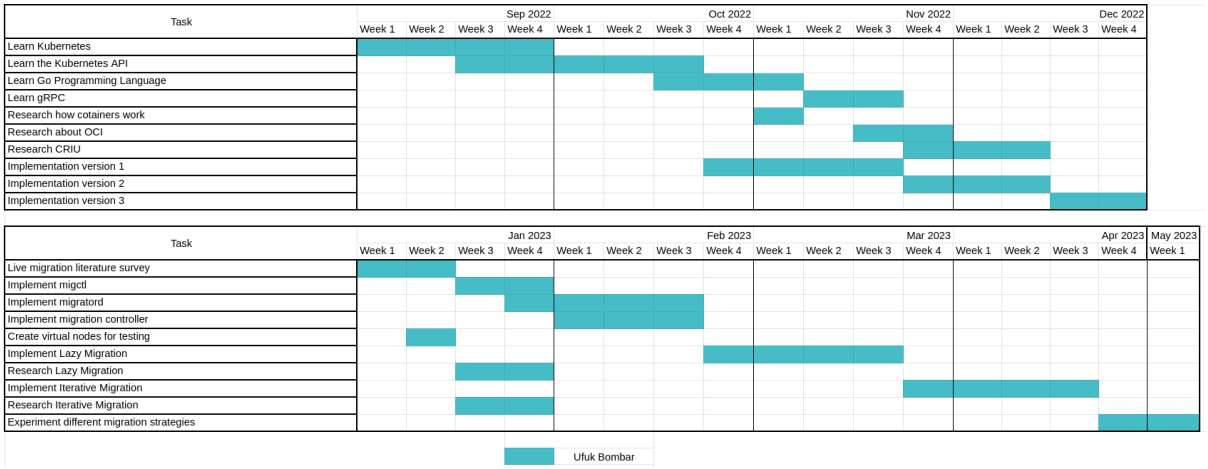

Figure 1: Grantchart for the project with only one student.

For the programming environment, Linux will be used. This is because, one of the essential programs that enable live migration, CRIU, is specifically implemented for Linux Operating System. It does not work for Windows and macOS machines, thus this renders Linux an adequate environment for development. For the second half of the project, more specifically Debian is selected as the Linux distribution because of the stability of the system itself.

As for the container runtime, Docker is seem fit for this project. Docker is a high-level container engine with rich image-building features. However, in the implementation phase, it is seen that Docker does not fully support checkpointing and restoring containers. Thus in the implementation, the system directly communicates with containerd which is much more lightweight and used as a backbone by Docker.

Lastly, Kubernetes will be used as the container orchestration tool. This is because Kubernetes is the most widely used for orchestrating workloads. The version of Kubernetes that will be used is 1.27 since in this specific version the pod checkpointing feature is already added to the code base with extensions.

For making the development process more robust, the project roadmap has been planned. First, the main tasks are discussed, then these tasks are divided into smaller tasks. The grant chart can be seen in Figure 1.

3 Bibliography

3.1 Cloud Computing

Before Cloud Computing chunky mainframes were used by companies in the 1950s. These room-sized machines could only be operated by one user at the same time. Also, these mainframes were incredibly expensive and not many companies could afford them thus there were out of reach for new companies or the ones with little capital. As a result in the

1970s IBM released a proto-operating system called VM [30]. This program permitted its admins to create virtual machines on a single physical hardware. The VM then evolved into a software category known as hypervisors. Nowadays, they are used to virtualize other operating systems.

In the 1990s telecom companies started offering internet services. With many users wanting to be online a new industry has been born. Many companies were competing to have online services. However, since the hardware was still expensive relative to today, some companies started offering their hardware bundled with hypervisors. This paradigm is named IaaS [31].

In the middle and late 2000s, Cloud Computing started to mature. When the tech industry was growing rapidly companies such as *Amazon*, *Google*, *Microsoft*, and others started offering cloud computing platforms. This helped businesses cut upfront costs and enabled them to grow faster [29].

3.2 Linux

Linux was begun as a hobby project by Linus Torwards in 1991 [36]. It was inspired by the UNIX project which was developed by Ken Thompson and Dennis Ritchie and funded by ATT Bell Labs [26]. Like Unix, Linux also followed the Unix philosophy in addition to Open Source Software movement. This led to the wide adoption of the Operating System.

3.2.1 CGroups and Linux Namespaces

Linux uses specific mechanisms to allow multiple tenants to use the same Operating System. These two mechanisms are named Cgroups and Linux Namespaces.

CGroups are first introduced by Google engineers to the Linux kernel in 2006 [23]. The goal was to construct a layer for the Linux kernel to do resource limiting, prioritization, and accounting. First, version 1 of Cgroups is introduced. After their introduction they are considered too complex and a simpler version which is called Cgroupsv2 is suggested, most of the technology such as Kubernetes and Openshift still depends on Cgroups version 1. But with patches, they are migrating to Cgroupsv2.

Cgroups are used to imitate a standalone system. The processes in that Cgroup has restricted memory and limited CPUShares. Behind the hood, the Linux kernel takes care of the complexities. In addition, cloud providers use the accounting aspect of CGroups to monitor a group of processes' resource usage. This is beneficial to the cloud providers because they can keep track of the usage and bill their clients accordingly [15].

Linux namespaces are also used along with Cgroups. It is a feature of the Linux kernel for isolating processes where namespaces ensure the processes belonging to a Linux namespace have their virtual filesystem as well as a network stack. They are prohibited from accessing outside this filesystem. Additionally, their network access is also isolated from each other, thus they are treated as a standalone machines. They can not communicate with other processes from other namespaces [18]. Together, Cgroups and Linux Namespaces create the foundation for containerized applications.

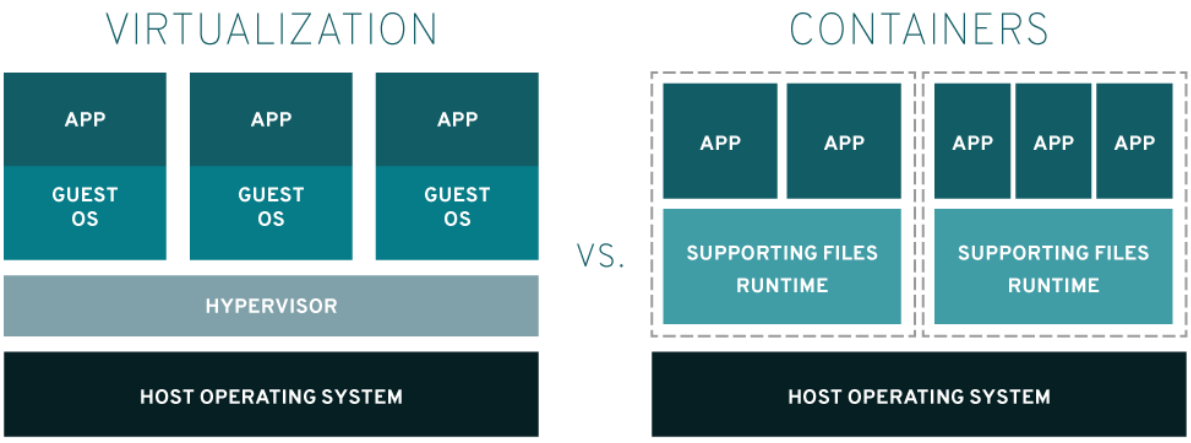

Figure 2: Virtualization vs Containers[34].

3.2.2 CRIU

Linux kernel enables processes to be stopped and retained. With Cgroups this functionality is extended so that processes belonging to a Cgroup can also be stopped, retained, and snapshotted. CRIU stands for Checkpoint Restore in User Space because CRIU is a project that aims to perform these actions without executing any code in kernel space. CRIU project currently integrated into main container runtimes such as *Docker*, *OpenVZ*, *LXC/LXD*, and *Podman*[35].

CRIU is not only intended for live migration. Likewise, there are other interesting areas of use cases. Here some of them are presented.

- Forensic Process Analysis: CRIU can be used for creating a checkpoint of a production process. This checkpoint then can be examined for any bugs or errors. This is in fact, implemented into Kubernetes version 1.24 [25].
- Creating Fault-Tolerant Systems: In a critical system, the vital processes can periodically be saved. So that in case of a critical error, the previous is instantaneously restored to provide maximum up-time.
- Dryrun: When a new application with a lot of preprocessing is starting an old checkpoint it can be used to speed up the initiation.

3.3 Container Tools and Runtimes

Containers are processes that are isolated from each other and have resources such as their filesystem, CPU, and memory [34]. They do not have a large overhead such as Virtual Machines. The difference between containers and VMs can be seen in figure 2.

There are different container runtimes and tools for various Operating Systems. For instance, Windows containers can only work on Windows machines [32]. Currently, most

popular containers run on Linux via the methods discussed such as Cgroups and Linux Namespaces. The programs managing the containers are named container runtimes. There are various runtimes available to choose from. The most popular ones are *Docker*, *Podman*, and *LXC*. These runtimes use slightly different mechanisms to ensure security, efficiency, ease of use, and isolation.

3.3.1 Docker

Docker is one of the most popular containerization tools where it uses the high-level *Docker Engine* to manage containers. With Docker Hub, the images of different applications can be stored, altered, and published from registries. Docker Engine uses *containerd* for managing lower-level details [14]. Yet, containerd also accesses the Linux's universal container component *runc* [17]. Runc establishes the Open Container Initiative, which standardizes how higher-level runtimes will interact with containers [33].

3.3.2 Podman

Podman is a daemon-less open-source container manipulation tool [8]. Apart from Docker, it does not employ a daemon by default which makes podman much more lightweight to install on a system. Podman is designed to support Linux systems only where it can use various container runtimes such as *runc*, *crun*, and *runv* [7]. Thus it is not possible to create and run Windows containers. Additionally, podman completely supports an OCI-compliant runtime interface [27]. If requested, the daemon of podman can be enabled and accessed via RES API. It also supports checkpointing when installed with CRIU. Additionally, as the name suggests podman does not only use to manipulate containers but pods. This is an advantage since in container orchestration tools such as Kubernetes, pods are designated as the smallest unit of work. With all of these features combined podman is seen as an alternative to Docker.

3.3.3 Containerd

Containerd is a low-level container daemon that glues different Linux system calls that are made possible by various kernel features to have an abstraction of a container [13]. It is also used as a backbone in many container tools and runtimes such as *Docker*. Containerd also uses *runc* to manage containers. The relation of containerd with Docker is given in Figure 3. As seen containerd only facilitates a container and the rest of the operations such as managing the volumes, networks, images, and other higher-level elements are done by Docker [13].

3.4 Kubernetes

Kubernetes also known as k8s, is an open-source container orchestration tool. It provides automated deployment, scalability, and management. It can work with multiple different container runtimes where it can manage multiple nodes inside a cluster. There are other

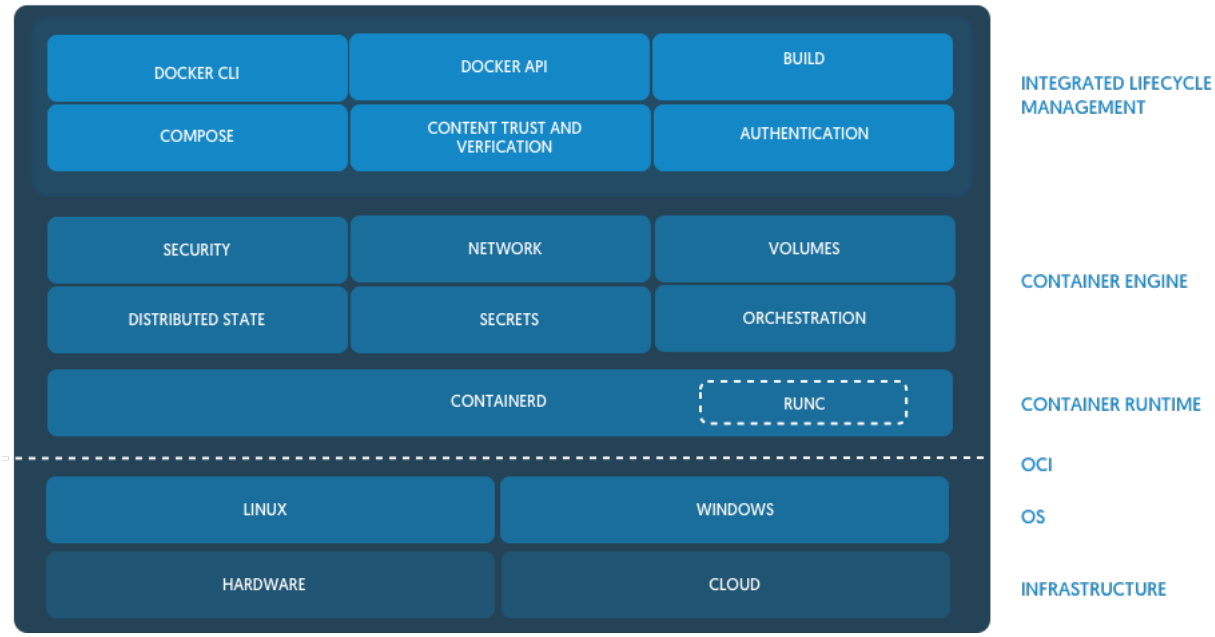

Figure 3: Relation of containerd and Docker [13].

alternatives to Kubernetes such as *Docker Swarm* and *RedHat OpenShift* however, k8s is more popular as a container orchestration tool [5]. Also *Openshift* uses Kubernetes as a backbone and it can be considered an extension of Kubernetes [9].

Kubernetes employs a declarative API instead of an imperative API. In the imperative API, the requests are sent to the API and they directly cause changes in the system. This is not suitable for highly distributed applications because it means there should be a high amount of communication between the API server and the nodes. To solve this problem Kubernetes uses the declarative API where an object and its state are declared [2]. To ensure the object's actual state is as desired Kubernetes uses the controller mechanism. Simply, a loop checks for changes in the state of the object. If the object's state is not desirable, then the controller does certain modifications to the cluster or does external changes [4].

One of the fundamental objects in Kubernetes is called Pod. They represent a unit of work with a set of containers with shared storage and network resources [6]. There are also higher-level objects such as Deployments. When a Deployment is declared it ensures its Pods are running without downtime. If an error occurs or a node goes down then the Deployment controller deploys the Pods into another node where the location is selected by various criteria such as the available resources in the candidate node.

In case the resources in a node are not enough, then the Deployment might want to delete the Pods and deploy them to another node. Since there are no mechanisms for migrating them, Pods are regarded as ephemeral. But this means the state of the Pod will be lost if the scheduler decides to change the assigned node of a pod [1]. This problem can be solved by the live migration functionality which is the topic of this project.

3.4.1 KubeVirt

As discussed before, VMs are also used in Cloud Computing. Because there is a layer of OS between the hypervisor implementing live migration is relatively easier than the containers [28]. Because Kubernetes can function with various runtimes, there are extensions such as KubeVirt where instead of containers, lightweight virtual machines are used [11].

KubeVirt is intended for Kubernetes workloads that cannot be easily containerized [10]. However, KubeVirt uses a different architecture where agents are deployed instead of VMs. The containerized agents use QEMU to create virtual machines in the node. Then these agents control and manipulate the VMs ensuring they are running smoothly.

3.5 Live Migration

Live migration refers to a process or a group of processes to be transferred to a different machine while preserving the CPU, memory, network, and storage states [21]. There are migration implementations that can handle different work units such as VMs, containers, and pods. As discussed previously, solutions for VM already exist and are used in the industry. A solution to live pod migration on the other hand not yet fully implemented. Pods are composed of containers thus implementing a solution to live container migration results in a practical working live pod migration system with little effort.

Containers are also suitable for migrating across machines since they are isolated processes. This means that only the parts of containers require migration nevertheless, they have intricate details that need addressing. These parts include the memory, storage, network, and in some advanced cases devices like GPUs or TPUs. There exist some old attempts at solving the process migration problem:

- *Sprite Operating System*: By default, the application interface of the kernel is implemented to support process migration [22].
- *CRAK: Linux Checkpoint and Restart As a Kernel Module*: A Linux kernel module that is implementing basic process migration in kernel-space [16].

There is also a more modern implementation:

- *CRIU: Checkpoint Restore in Userspace*: Used by many container runtimes to implement container checkpointing. It supports page servers for different types of migrations [12] [24].

The most basic form of live migration is called *Cold (Offline) Migration*. It is the process of dumping the memory and storage info of a process, transferring it, and then restoring it [20]. However, this results in huge losses of time when the container is frozen. There are also other methods proposed in the literature.

- *Lazy Migration*: This method is proposed for handling processes with huge memory usage. If that kind of process is dumped then the resulting file would be huge and can cause latency in transferring state. Which can lengthen the offline time

of the process. To overcome this, the process is migrated with empty memory to the destination. Then two-page server daemons serve the required pages to the destination process. When all of the pages are sent the old process is killed thus the migration is completed.

- *Iterative Migration*: This method systematically sends large files before the transfer while the process is still active. Then, when the process is migrated, the delta information (the difference between the current and previous) is sent. This reduces the downtime of the migration and stress on the network. This type of migration heavily utilizes file differentiation tools.

3.5.1 Kuberentes and Live Migration

There has already been an effort toward enabling the checkpointing and restoring of stateful containers in Kubernetes. This work is not only useful for migrating containers but, also for rebooting the node without losing the state of the containers, quick startup, and forensic container inspection. The implementation adds an API endpoint to the kubelet that makes it possible to create a checkpoint of a container while it continues to run in the background [3]. However, there is no fully supported version of the live migration mechanism in upstream Kubernetes at the time of this work.

4 Analysis

In the analysis section, the functional and non-functional requirements for the system will be discussed.

4.1 Functional Requirements

In the intermediary report, it has been decided that the live pod migration system is composed of 3 main applications. Namely, *migctl*, *migration controller*, and *migratord*. However, *migctl* which is intended to be the initiator of the migrations and a command line tool is scrapped. Thus there are only 2 applications that are required to perform live pod migration. The first application's name is *migratord*. This application will be a daemon process that will be running on all of the nodes. It will be responsible for the live migration by communication with the node's container runtime and other migrators. It will also be the one to send the checkpoint files. The second application is named the *migration controller*. It is responsible for handling live migration jobs and requests. The functional requirements for these components are given in this section. Additionally, the detailed design of the system is discussed in the design section.

4.1.1 Migrator Controller

- The migration controller should be able to connect to the Kubernetes API server.

- The migration controller should be able to validate requests and spawn live migration jobs.
- The migration controller should be able to cancel a migration if requested.
- The migration controller should be able to push the migration job's state to the next one.

4.1.2 Migratord

- The migratord should access the node's container runtime.
- The migratord should be able to create a checkpoint of a container.
- The migratord should be able to restore a container from the dump file.
- The migratord should be able to send the dump file to the destination node.
- The migratord should be able to validate the migration with its peer migratord.
- The migratord should be able to clear any remaining files after the migration job.

4.2 Non-Functional Requirements

The non-functional requirements which describe the limitations of live pod migration are listed for each component located in the following sections. Time limitations described in the following sections are selected from popular implementations such as *kubectl*, and *docker*.

4.2.1 Migration Controller

- The migration controller should be able to run in a containerized environment.
- The migration controller should be able to catch old events in case of a restart.

4.2.2 Migratord

- The migratord should update the state transfers in a maximum of 45 seconds.
- The migratord should be able to send and receive a container.
- The migratord should be able to run on a Linux machine.
- The migratord should run on every node in the Kubernetes cluster.

5 Design

In this section, the design of the system will be presented. First, a high-level overview is described. The different parts of the system will be addressed and the FSM diagram will be shown. Then in the low-level design section, detailed attributes will be presented about each program.

5.1 High-Level Design

The system proposed is designed for Kubernetes clusters with multiple nodes where it is required to transfer a container without losing the state with minimal downtime. It is also important to design the system with the Kubernetes components so that the components can be reused and the design philosophy of Kubernetes will be respected. Additionally, edge cases also need to be handled. Since this is a proof of concept, some of the advanced cases such as ones with complex network configurations and ones with mounted volumes are disregarded. The design work focuses on simple pods. The containers inside a pod will be accounted for individually. The main architecture for the system can be seen in Figure 4.

In the previous iterations, the migration is initiated by the command line tool *migctl*. However, this approach is not required since the system is made compatible with the Kubernetes architecture it is possible to reuse already existing components. This meant instead of creating and maintaining a separate command line tool, it is possible to extend Kubernetes with custom resource definitions and controllers and use *kubectl* to create, delete and alter custom resource definitions. To achieve this, two custom resource objects are proposed. They are named *LPMJobRequest* which contains the migration details and the *LPMJob* which represents the actual live pod migration.

When an LPMJobRequest object is created it should contain minimal information about the pod name and the destination node name. Note that both of these custom resources will live in a namespace and the pod should live inside the same namespace. All the other information such as the source-node-name will be filled automatically by the *migration controller*. When the validation of an *LPMJobRequest* is successful, a *LPMJob* object is spawned. When an *LPMJob* is created, for each container living inside the pod the corresponding state will be registered thus, each of the containers inside the pod will have its state information and it will be migrated individually.

The process *migratord* will be deployed to all of the nodes and will run as a daemon process. It will also listen for the changes on *LPMJob* objects. It will have connectivity with the container runtime installed on the node and since the container runtime interface is standardized it will also support multiple runtimes which have checkpointing capabilities. Then, for each container, the *migratord* on the nodes will perform tasks to finish and proceed to the next state. A container may have only one state. The pod's state is determined by the earliest container state. Additionally, when designing the states it is also important to design them flexibly. This is especially important if we want to extend and apply different methods of migration into our system. Below the states and

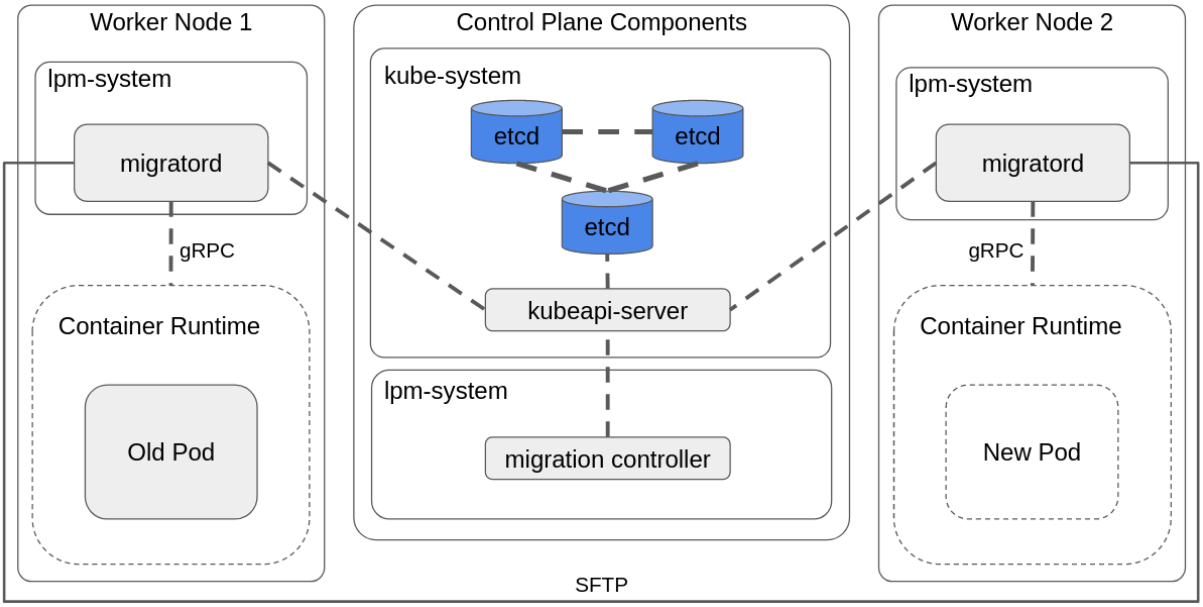

Figure 4: High-level architecture of the proposed migration system coupled with Kubernetes.

the descriptions will be revealed. The state diagram can be found in Figure 5. Although, these actions are primarily intended for cold migration.

- In the *Processing* state, the destination node's migratord will prepare the image of the container. At the same time, the source node's migratord will prepare the container for migration. In more advanced techniques the storage and memory can be pre-dumped.
- In the *Checkpointing* state, the source node's migratord will stop and dump the process. The server will wait for further instructions.
- In the *Transfering* state, the migrators send the dumped checkpoint file. The receiver receives the file and confirms its integrity.
- In the *Restoring* state, the destination node's migratord will restore the process from the previously received dump file.
- In the *Cleaning* state, all of the temporary files are cleansed from the system. The live migration is completed. The state of the job object is switched into *Done*.

5.2 Low-Level Design

In the low-level section, the components of software architecture and design will be introduced. For each component, the interfaces and their role in the program will be mentioned.

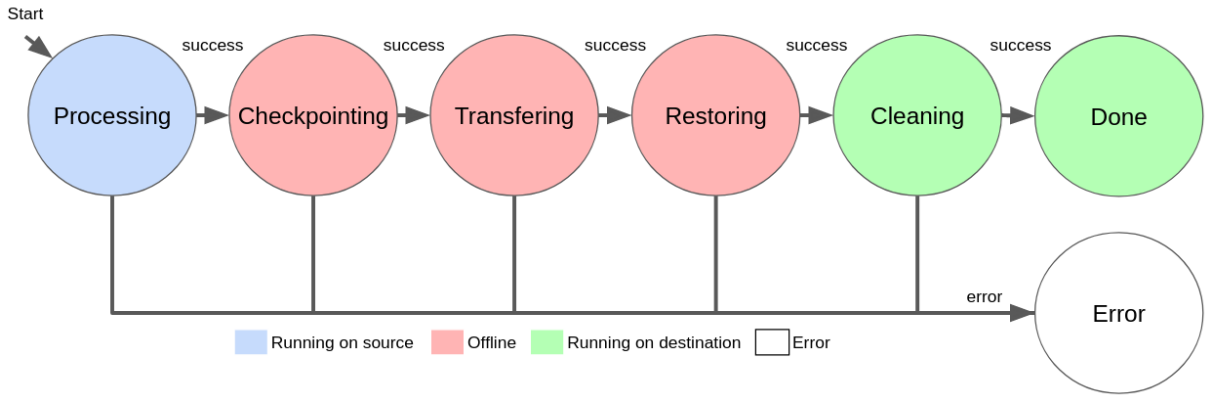

Figure 5: State diagram of a container when performing live migration.

5.2.1 Migration Controller

The *migrator controller* will implement a simple custom controller logic. Kubernetes go libraries already provide informers and listers which handles the complex synchronization logic. Thus Migration controller only needs to implement the handlers which will be invoked when an update of a custom object is received.

The *migrator controller* will listen for changes on two custom Kubernetes objects *LPMJobRequest* and *LPMJob*.

- *LPMJobRequest*: When a new object is created with the information of pod-name and destination-node-name, the handler for migration controller will fill in all the other necessary information required to perform live pod migration. When the handler receives an object with all of the values filled, it validates and creates the *LPMJob* object.

- *LPMJob*: When a valid *LPMJobRequest* object is created, an *LPMJob* object is created. This object will hold the state information as well as the container id for each container in the given pod. The states of the containers are updated and the pod's final state is determined by the earliest container state.

5.2.2 Migratord

Migratord also borrows a similar architecture from the Migration Controller. In addition, there are interfaces for communicating with the container runtime, the Kubernetes API server, and remote file transfer. It listens for changes in the *LPMJob* object. When a change event occurs the object is sent to the corresponding state handler. F or each handler *migratord* has to do a certain unit of work. These works alter the state of the containers and other Kubernetes API objects. Additionally, they might send checkpoint files to the destination node's *migratord*. The tasks performed by these state handlers are given below:

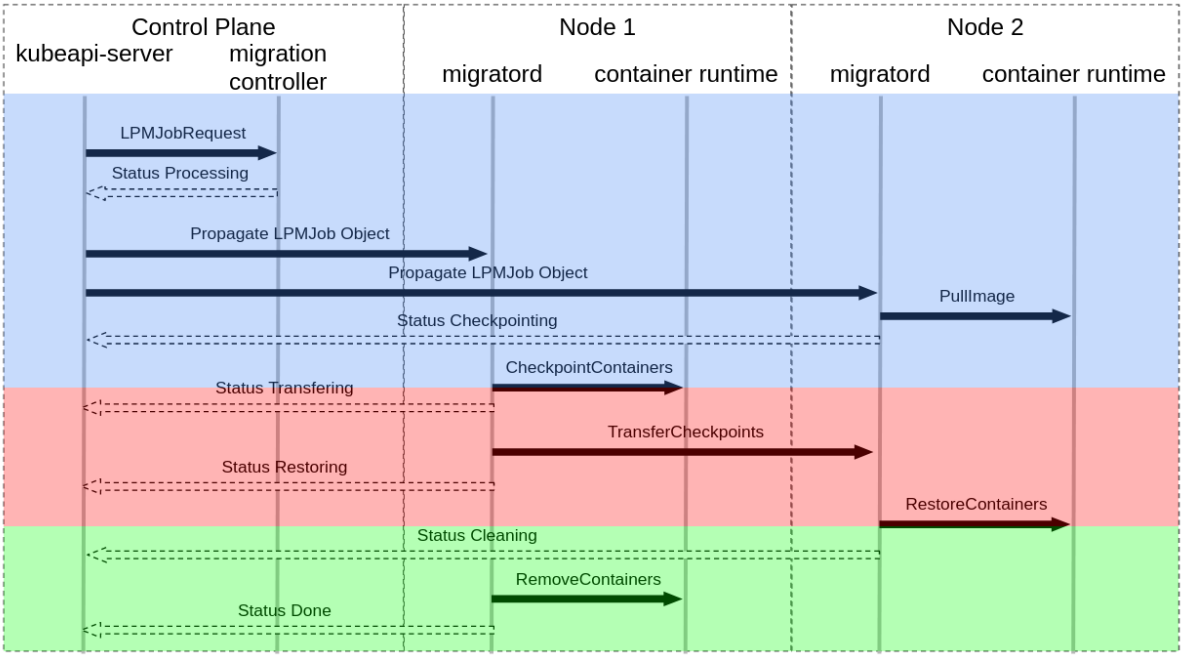

Figure 6: Interaction diagram for the proposed system.

- *Pull Image*: To restore a container it is required to have the container image. Before the migration, the destination node's migratord commands its container runtime to pull the image from the registry.
- *Checkpoint Container*: Migratord commands its container runtime to stop the container and prepare a checkpoint file.
- *Transfer Checkpoint*: The migratord of the source node directly initiates an SFTP connection with the destination node. The checkpoint file is sent over the network.
- *Restore Container*: After receiving the checkpoint file, the destination node's migratord commands its container runtime to create a container and restore it with the checkpoint file.
- *Remove Containers*: When the container started to run on the destination node, both migratords delete the temporary files including the checkpoint file.

6 Progress and Report

The first step is to create the environment where the live migration will occur. To achieve that, I decided to use minikube. Minikube is an easy-to-use tool for creating a local Kubernetes cluster. It also allows adding nodes to the cluster. Since Kubernetes 1.27 is equipped with a pod checkpointing feature I decided to use minikube nodes for the

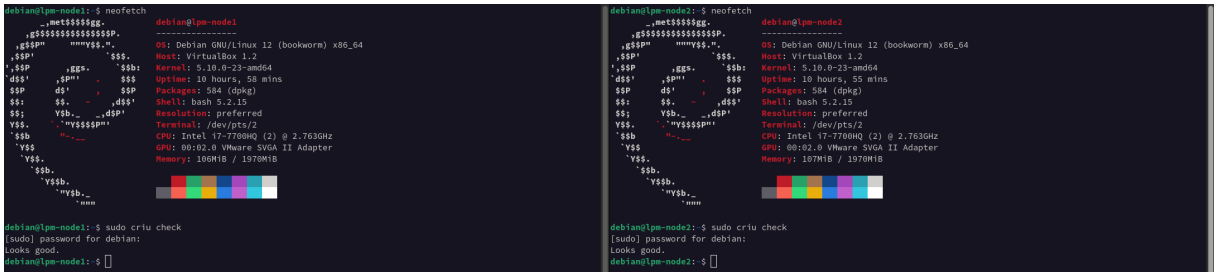

Figure 7: Two virtual nodes running in the same host with CRIU installed.

nodes I will perform live migration. However, the nodes run on a Docker container thus some of the features are unstable. Thus, I discarded the Docker backbone and tried to use KVM as the minikube backbone. Similar problems occurred again thus I decided to abandon minikube altogether. Then I created two virtual machines and named them vnode1 and vnode2. Then installed CRIU and Docker onto them. I also enabled the Docker experimental mode for checkpointing and restoring Docker containers. As a test, I connected them through an SSH connection and commanded *neofetch*. The resulting output can be seen in Figure 7.

The exact implementation as discussed consisted of two programs, *migration controller* and *migratord*.

In the first iteration, I implemented a Kubernetes controller and migratord applications which has a controller loop. This version of migratord has embedded in a Kubernetes control loop. In this way, I tried to implement the migration functionality directly with Pod. However, after many errors and exceptions, my supervisor and I decided to have a simpler product that demonstrates the core principles of pod migration.

In the second iteration, I implemented the migratord as a standalone application. However, the design did not allow any support for multiple migration jobs with multiple roles. This meant a process cannot handle more than one migration job both incoming and outgoing. Additionally, there are no mechanisms for synchronizing the state of the jobs. Thus, I decided to do a third iteration.

In the third iteration, I implemented a proper synchronization mechanism that uses queues for processing jobs. I also implemented image copying functionality. Also, I implemented the command line tool migctl. I used this tool to initiate migrations and see the states.

In the fourth and final iteration, I scrapped most of the code since Kubernetes provides most of the boilerplate code. I have implemented the migration controller using Kubernetes' sample custom (which can be found here: https://github.com/kubernetes/sample-controller) controller repository. I created two custom resource definitions. Then using the sample controller repository, I implemented *migratord*. Then pushed it into my Dockerhub. Then created a daemon set object with the migratords. I also allowed it to be communicated with the node's containerd container runtime. Then I used my checkpoint transfer code to implement the checkpoint transfer feature. Then I implemented the handlers for different states. In the end, I have the proposed system.

I have published the project as a GitHub repository which can be found here: https://github.com/ubombar/live-pod-migration. I also prepared a demo video for the presentation. It can be reached by the link on the presentation.

6.1 Requested Changes

There was incredibly beneficial feedback in this project, which they all addressed in the second iteration. They are given below as a bullet point list:

- *Use already existing tools*: As presented in the design section, Kubernetes is a very large and complex application. Since it is very popular and widely used, there are many tools made for specific tasks as well as ease the development experience. One feedback I received is instead of creating a standalone command line tool, just use the kubectl to create, delete, and update objects. On top of that, I utilized the synchronization mechanism of Kubernetes. This decision incredibly speed up the development process and allowed me to spend more time on research and development of more core aspects.

- *Keeping the design simple*: In the first two iterations, I slowly realized the complexity of the system and proposed ad hoc solutions. this made the system even more complex and clunky. With the last iteration, I simplified the design so that most of the complexity is erased. The parts with complex logic are also abstracted.

- *Design with scalability in mind*: Since this is a project related to the computer science branch distributed systems, it is important to reduce single points of failures. My old iterations allowed the system to be collapsed since it employed a primitive synchronization mechanism and centralized design. The inclusion of custom controllers, allowed the logic to be scaled. With this design, more than one custom controller can be deployed to ensure reliability.

- *Validate your sources*: In addition to the technical requests and feedback I am given for the report. To have a simple design, it is required to have a piece of valid knowledge about the field. Since the technology of containers has many details, it is important to validate sources. In some cases, I have encountered oversimplified explanations from some of the blogs. Thus in my next iteration, I have validated a piece of information from other sources.

- *Prevent doing over-extensive research and go off-topic*: Another aspect of searching is to have a bibliography that is just enough. For the first version, I thought I did not have a bibliography section. There were also a lot of subjects since the topic of containers involves deep knowledge of operating systems, networks, etc. I tried to do as much research as possible. However, I am advised not to do over-extensive research since it meant the impossibility of completing it. Thus I am well advised to do abstraction during this research.

7 Conclusion

In conclusion, live pod migration relies on many complex and intricate mechanisms to function properly. Sending containers and pods across devices will allow us to advance and solve other difficult problems. It is also very important in the edge computing world. Thus live pod migration acts as a milestone in computer science.

References

[1] Kubernetes Authors. *Deployments*. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.

[2] Kubernetes Authors. *Imperative Commands*. URL: https://kubernetes.io/docs/tasks/manage-kubernetes-objects/imperative-command/.

[3] Kubernetes Authors. *Kubelet Checkpointing API*. URL: https://kubernetes.io/docs/reference/node/kubelet-checkpoint-api/.

[4] Kubernetes Authors. *Kubernetes Controllers*. URL: https://kubernetes.io/docs/concepts/architecture/controller/.

[5] Kubernetes Authors. *Kubernetes Overview*. URL: https://kubernetes.io/docs/concepts/overview/.

[6] Kubernetes Authors. *Pods*. URL: https://kubernetes.io/docs/concepts/workloads/pods/.

[7] Podman Authors. *Podman Docuemntation*. URL: https://docs.podman.io/en/latest/index.html.

[8] ReadHat Authors. *What is Podman?* URL: https://www.redhat.com/en/topics/containers/what-is-podman.

[9] RedHat Authors. *Openshift: Kubernetes Overview*. URL: https://docs.openshift.com/container-platform/4.11/getting_started/kubernetes-overview.html.

[10] KubeVirt Community. *Architecture — KubeVirt*. URL: https://kubevirt.io/user-guide/architecture/.

[11] KubeVirt Community. *Live Migration — KubeVirt*. URL: https://kubevirt.io/user-guide/operations/live_migration/.

[12] *CRIU*. URL: https://criu.org/.

[13] Michael Crosby. *What is containerd runtime?* URL: https://www.docker.com/blog/what-is-containerd-runtime/.

[14] Johan Fischer. *Containerd vs Docker*. URL: https://earthly.dev/blog/containerd-vs-docker/.

[15] Kernel Documentation Group. *Cgroup's Freezer*. URL: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/freezer-subsystem.html.

[16] Paul H Hargrove and Jason C Duell. "Berkeley lab checkpoint/restart (blcr) for linux clusters". In: *Journal of Physics: Conference Series*. Vol. 46. 1. IOP Publishing. 2006, p. 067.

[17] Solomon Hykes. *Spinning Out Dockerâs Plumbing: Part 1: Introducing runC*. URL: https://www.docker.com/blog/runc/.

[18] Scott van Kalken. *What Are Namespaces and cgroups, and How Do They Work?* URL: https://www.nginx.com/blog/what-are-namespaces-cgroups-how-do-they-work/.

[19] Lele Ma et al. "Efficient Live Migration of Edge Services Leveraging Container Layered Storage". In: *IEEE Transactions on Mobile Computing* 18 (2019), pp. 2020–2033.

[20] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. "Containers checkpointing and live migration". In: *Proceedings of the Linux Symposium*. Vol. 2. 2008, pp. 85–90.

[21] Pokhrel Niroj. "Live container migration: Opportunities and challenges". In: *Aalto University* (2017).

[22] John K. Ousterhout et al. "The Sprite network operating system". In: *Computer* 21.2 (1988), pp. 23–36.

[23] Steve Ovens. *A Linux sysadmin's introduction to cgroups*. URL: https://www.redhat.com/sysadmin/cgroups-part-one.

[24] Adrian Reber. *Container Live Migration Using Runc and Criu*. URL: https://www.redhat.com/en/blog/container-live-migration-using-runc-and-criu.

[25] Adrian Reber. *Kubernetes and Checkpoint/Restore*. URL: https://stackconf.eu/talks/kubernetes-and-checkpoint-restore/.

[26] Dennis Ritchie. *The Evolution of the Unix Time-sharing System**. URL: https://www.bell-labs.com/usr/dmr/www/hist.html.

[27] Soma Sharma. *Difference between Docker, Kubernetes, and Podman*. URL: https://medium.com/javarevisited/difference-between-docker-kubernetes-and-podman-8b03a4cf03bc.

[28] Abraham Silberschatz, Peter B Galvin, and Greg Gagne. *Operating System Concepts, 10e Abridged Print Companion*. John Wiley & Sons, 2018.

[29] AWs Cloud Team. *About AWS*. URL: https://aws.amazon.com/about-aws/. (accessed: 01.18.2023).

[30] IBM Cloud Team. *Cloud Computing History*. URL: https://www.ibm.com/cloud/blog/cloud-computing-history. (accessed: 01.14.2023).

[31] IBM Cloud Team. *What is IaaS (Infrastructure-as-a-Service)?* URL: https://www.ibm.com/topics/iaas. (accessed: 01.18.2023).

[32] Microsoft Cloud Team. *Windows and Containers*. URL: https://learn.microsoft.com/en-us/virtualization/windowscontainers/about/.

[33] OCI Team. *About OCI*. URL: https://opencontainers.org/about/overview/.

[34] Readhat Team. *What is a linux container?* URL: https://www.redhat.com/en/topics/containers/whats-a-linux-container.

[35] Redhat Team. *CRIU - Checkpoint/Restore in user space.* URL: `https://access.redhat.com/articles/2455211`.

[36] Jayant Verma. *History Of Linux â How Did Linux Start And Who Created Linux?* URL: `https://www.linuxfordevices.com/tutorials/linux/history-of-linux`. (accessed: 01.18.2023).