Migrating Pods in Kubernetes

Thesis · December 2020

DOI: 10.13140/RG.2.2.31821.97762

Jakob Schrettenbrunner

h_da

HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

fbi

FACHBEREICH INFORMATIK

Hochschule Darmstadt

- FACHBEREICH INFORMATIK -

Migrating Pods in Kubernetes

Abschlussarbeit zur Erlangung des akademischen Grades
Master of Science (M.Sc.)

vorgelegt von

Jakob Schrettenbrunner
742343

am 22.12.2020

Referent: Prof. Dr. Stefan T. Ruehl
Korreferent: Prof. Dr. Ronald Charles Moore

Eigenständigkeitserklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Darmstadt, den 22.12.2020

Jakob Schrettenbrunner

Abstract

Since its release in 2014, Kubernetes has become the leading container orchestration platform. Its ability to provide a consistent interface across different public cloud providers, private clouds, as well as bare metal deployments is one of the main reasons for its popularity.

When attempting to rely solely on Kubernetes for deployment, it may be necessary to deploy applications that have not been designed for cloud environments. Kubernetes considers resources, especially Pods, to be ephemeral and requires multiple instances of applications to be deployed if high-availability should be guaranteed. If a node has to be taken down for maintenance, Pods are simply killed and restarted somewhere else. When doing so with applications that do not support high-availability, this will lead to downtime. For virtual machines, this problem has been solved with live migration, which allows moving virtual instances to another host without turning them off. Popular container runtimes like Docker support checkpoint and restore mechanisms, which would allow implementing (live) migration for containers, and therefore Kubernetes, as well.

The increasing use of edge computing enables additional use cases for migration. For example, workloads could be migrated between different edge nodes to minimize latency for moving clients in 5G networks. Supporting Pod migration would strengthen Kubernetes' suitability for deploying on the edge.

This thesis analyzes how Pod migration can be integrated into Kubernetes. It argues that Pods should be cloned instead of attempting to reschedule them. By deleting the old Pod, the result is identical to an actual migration. The introduction of a MigratingPod resource is proposed to properly reflect the migration in the API, with a migration controller handling the migration itself. A prototype was implemented to evaluate the proposed design, which demonstrates that the approach is viable and integrates well with existing components. A benchmark shows that the chosen strategy to create full checkpoints is sufficient to migrate small workloads with reasonable downtimes. To achieve full network transparency and to be able to migrate larger Pods, faster migration strategies like pre- and post-copy need to be implemented by container runtimes and Kubernetes itself.

Zusammenfassung

Seit seiner Veröffentlichung im Jahr 2014 hat sich Kubernetes zur führenden Plattform für die Container-Orchestrierung entwickelt. Einer der Hauptgründe für seine Beliebtheit ist die Fähigkeit, eine konsistente Schnittstelle über verschiedene Public-Clouds, Private-Clouds, sowie physische Server hinweg bereitzustellen.

Soll bei der Bereitstellung ausschließlich auf Kubernetes gesetzt werden, kann es notwendig sein, auch Anwendungen bereitzustellen, die nicht für Cloud-Umgebungen konzipiert wurden. Kubernetes betrachtet Ressourcen, insbesondere Pods, als kurzlebig und erfordert die Bereitstellung mehrerer Anwendungsinstanzen, wenn Hochverfügbarkeit gewährleistet sein soll. Wenn ein Server für Wartungsarbeiten heruntergefahren werden muss, werden die von ihm ausgeführten Pods einfach entfernt und auf einem anderen Server neu gestartet. Bei älteren Anwendungen ohne Unterstützung für Hochverfügbarkeit kann dies zu Ausfallzeiten führen. Für virtuelle Maschinen wurde dieses Problem in Form von Live-Migration gelöst. Diese erlaubt es, virtuelle Instanzen auf einen anderen Host zu verschieben, ohne sie abzuschalten. Gängige Container-Runtimes wie Docker unterstützen Checkpoint- und Restore-Mechanismen, die es erlauben, (Live-)Migration auch für Container und damit auch für Kubernetes zu implementieren.

Die steigende Nutzung von Edge-Computing bietet zusätzliche Anwendungsfälle für Pod Migration. Es können beispielsweise Anwendungen zwischen verschiedenen Edge-Knoten migriert werden, um die Latenz für bewegliche Geräte in 5G-Netzwerken zu minimieren. Durch die Unterstützung von Pod-Migration würde Kubernetes seine Eignung für den Einsatz im Zusammenhang mit Edge-Computing verbessern.

Diese Arbeit analysiert, wie Pod-Migration in Kubernetes integriert werden kann. Es wird argumentiert, dass Pods geklont werden sollten, anstatt zu versuchen sie zu verschieben. Wird der alte Pod danach gelöscht, ist das Resultat identisch zu einer tatsächlichen Migration. Es wird vorgeschlagen, eine MigratingPod-Ressource einzuführen, um die Migration in der API korrekt abzubilden, während ein Migrationscontroller die Migration selbst abwickelt. Zur Evaluierung des vorgeschlagenen Designs wurde ein Prototyp implementiert, welcher die Funktionsfähigkeit dieses Ansatzes und dessen gute Integrationsfähigkeit mit bestehenden Komponenten bestätigt. Ein Benchmark zeigt, dass die gewählte Strategie, vollständige Checkpoints zu erstellen, ausreicht, um kleine Anwendungen mit akzeptablen Ausfallzeiten zu migrieren. Um vollständige Netzwerk-Transparenz zu realisieren und um größere Pods migrieren zu können, müssten schnellere Migrationsstrategien wie Pre- und Post-Copy von Container-Laufzeiten und Kubernetes selbst implementiert werden.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Motivation

Since the release of Docker in 2013 [Hyk13; Mes18], container technology has become increasingly popular. By sharing the kernel of the host system, containers are a more lightweight alternative to virtual machines [SLC17; CLL19], but provide similar levels of isolation between processes.

In 2014, Kubernetes was released [Bed18], which provides a way to deploy thousands of containers in a predictable and manageable way. Since then it has become the leading container orchestration platform, which is now supported by all major cloud providers [Ama; Mic; Goo]. Recently, it has gained additional attention with regards to multi-cloud deployments: Kubernetes can provide a consistent, standardized way of deploying applications across different providers. Some providers even offer managed Kubernetes for multi- and private-cloud [MSV20].

Like any computer system, the servers hosting the virtual machines and containers require periodic maintenance in the form of software updates and hardware replacements. Especially the latter frequently require the system to be turned off or restarted. With one system hosting many applications at once, doing so affects many users at the same time.

For virtual machines, this problem can be evaded by the use of live migration. The virtual machines running on the host that requires maintenance are moved to a different host before the maintenance is performed. With IP addresses getting migrated as well, downtime can be reduced to only a few hundred milliseconds [Cla+05]. Modern hypervisor software [VMw; KVM; Pro] and cloud operating systems like OpenStack [Ope20b] support live migration. Some cloud providers rely on it to reduce downtime for customers as well [Goo20; Bur18].

In a Kubernetes cluster Pods, an abstraction around one or more containers, are usually moved between hosts by terminating them and restarting them on another host. For modern applications that are built to be deployed in the cloud and include high-availability concepts or are entirely stateless this is a viable solution: multiple instances are running at the same time and can take over for an instance that is temporarily unavailable. When attempting to rely solely on Kubernetes for deployment, it can also be necessary to deploy older applications. Those applications might not support high-availability, and moving Pods can therefore cause downtime.

Another emerging use case for Pod migration is edge computing, which pushes computing resources towards the edge of the internet (e.g. mobile network towers) in order to reduce the physical distance and subsequently the latency for clients [Sat17]. With 5G networks, mobile devices can benefit from such low latencies as well. Migrating services between different edge nodes allows to keep latency minimal, even if those devices move physically [Add+20]. Due to the consistency it can provide, Kubernetes is also used for the deployment of workloads in edge clouds [Naq19]. Supporting Pod migration can help with solidifying this popularity.

Furthermore, migration could help with resource utilization. Free capacities of a cluster could be filled with long running tasks that are migrated to other nodes when resources are needed by other applications. Advanced scheduling could also move applications between nodes easily, in order to spread resource usage more evenly.

Some container runtimes already support migrating containers between hosts, e.g. Docker [Kaz15] or Podman [Pod]. Using those existing capabilities migration of Pods can be implemented in Kubernetes. Doing so could reduce downtimes of slow-starting stateful applications hosted in Kubernetes significantly when moving them between nodes.

1.2 Aim of this Work

Some container runtimes, e.g. Docker [Kaz15] or Podman [Pod], are already capable of creating and restoring checkpoints of containers. Using those features, container migration and, in the case of Kubernetes, subsequently Pod migration can be realized as well.

Since Kubernetes was specifically designed for cloud-native applications, it is more concerned with recovering from failure than to prevent it. Workloads and hardware is expected to fail and mechanisms are in place to restart or redeploy as necessary. Migrating workloads was not considered for its design. Most importantly, Pods are considered to be ephemeral and not as a resource that needs to be preserved.

The goal of this thesis is to analyze how Pod migration can be integrated into Kubernetes and its API while adhering to existing concepts and principles of the project.

Chapter 2 gives an introduction to Kubernetes, a brief overview about container runtimes and a more detailed explanation of container migration.

It is followed by a brief summary of related work in chapter 3 with regards to container migration in general and workload migration in Kubernetes.

The important design decisions necessary when implementing Pod migration are discussed in chapter 4.

The prototypical implementation that was created to evaluate the previous decisions is described in chapter 5, with the evaluation itself and its results following in chapter 6.

The final chapter 7 summarizes the results and lists a few open questions and topics for future work.

CHAPTER 2

Fundamentals

2.1 Kubernetes

“Kubernetes is an open source container orchestration engine for automating deployment, scaling, and management of containerized applications” [Kubl]. It was open-sourced by Google in 2014 and was inspired by their internal cluster-management system called *Borg* (see [Ver+15]). In 2018 it was handed over to the *Cloud Native Computing Foundation* (CNCF), a subsidiary of the *Linux Foundation*.

Kubernetes is very modular and consists of various components. In combination they facilitate declarative deployment of containerized workloads. Kubernetes takes care of scheduling, container management and basic monitoring and provides load balancing, service discovery, storage orchestration, secrets and configuration management, and more. [Kubz]

2.1.1 The Kubernetes API

The central component of a Kubernetes cluster is the *kube-apiserver*, which serves the Kubernetes API. It acts as the single source of truth for the configuration as well as the current status of the cluster. All other components primarily communicate by creating, modifying and deleting resource objects stored by the API server. The communication is, with a few exceptions, unidirectional: the components query the API, but the API server rarely queries the components. This way the cluster is very decoupled, which simplifies the deployment of additional components. The users of a cluster, both humans as well as other applications, also interact with the cluster using the same API. In fact, there is no difference between a user and a Kubernetes component with regards to the API. [Kubw; Kubk]

A specialty of the Kubernetes API is its declarative behavior. Most APIs behave in a imperative manner: calling an API endpoint leads to an immediate interaction with the resource that is represented by the API. For example, a cloud provider might offer an API that allows to manage virtual machines. When a virtual machine is created with the API, the API server forwards the request to a hypervisor that immediately creates the virtual machine. Once the API call is complete, the virtual machine is already created. When requesting a list of virtual machines, only the machines that exist on the hypervisor are shown.

In the declarative API of Kubernetes, the *desired* resources are described. A resource that is created in the API does not immediately lead to an action. Instead, other components in the cluster will eventually notice that a new desired resource was created and will then try to create the real resource as well. Once the resource is created, the representation in the API will be updated with its status. A list of resources in the Kubernetes API can include resources that already exist in the cluster but also ones whose creation is still pending.

2.1.2 API Objects

All resources in a Kubernetes cluster are represented as API *objects*. Objects are usually represented in YAML. An example object can be seen in figure 2.1. The first two fields are the *apiVersion* of the representation and the *kind* of resource that is represented by the object. In combination they specify the exact schema of the object. [Kubx]

```
apiVersion: example/v1
kind: Example
metadata:
    name: example
    namespace: default
    labels:
        mylabel: myvalue
    uid: 39b420ed-4235-49c5-be3d-982cc1337c0b
spec: {}
status: {}
```

Figure 2.1: An example of a Kubernetes object represented in YAML.

All objects have metadata attached to them, whose basic structure is the same for all kinds of objects. It contains a name that is used as an identifier. It has to be unique for all objects of the same kind within a namespace. Namespaces allow to divide a single Kubernetes cluster into multiple virtual clusters to isolate resources from each other. The API server assigns each object a UID to uniquely identify them across the whole cluster. [Kubp]

The metadata also contains two sets of key-value pairs. The most common are labels, which are used as identifiers to select subsets of objects. For example, if a resource belongs to a specific application, the name of that application could be attached as an *app* label on the object [Kubm]. The second set contains annotations. They are intended for specifying parameters for components of the cluster that interact with the object [Kuba]. The metadata can also contain a list of Finalizers. They prevent the object from being deleted from the API until no more Finalizers are present [Kubj; Kubg].

The desired state of the resource that the object represents is described in the *spec* field. Its structure depends on the kind of the object. The cluster components then work try adapt the state of the cluster towards the specification. The actual state of the resource is then reflected by the *status*, which is populated by the cluster components. [Kubx]

2.1.3 Pods

Even though Kubernetes is a container orchestration system, there is no container resource. Instead, containers are described as part of a *Pod*, which is the smallest unit of computation and the most common resource in a Kubernetes cluster. Figure 2.2 shows an example of a simple Pod object. [Kubt]

```
apiVersion: core/v1
kind: Pod
metadata:
  name: example
spec:
  containers:
  - name: redis
    image: redis:latest
    ports:
      - containerPort: 6379
        protocol: TCP
    volumeMounts:
      - mountPath: /data
        name: data
  volumes:
    - name: data
      emptyDir: {} # an ephemeral directory on the node the Pod runs on
```

Figure 2.2: An example of a Kubernetes Pod containing a single Redis[1]container. The status field has been left out for brevity.

Pods consist of one or more containers, the former being the most common. All containers within a Pod share a network namespace, meaning each Pod has a single IP that is shared by all of its containers. [Kubt]

Pods also share storage in the form of *volumes*. From the perspective of the Pod, a volume is simply a directory that can be mounted into the Pod's containers. The volume can be a local

[1] Redis is an open source in-memory data store with optional persistence, see https://redis.io).

directory on the node or remote storage, for example provided by a cloud provider [Kuby]. A volume can also claim a *PersistentVolume*, which serves as an abstraction of persistent storage resources and hides details like the actual storage backend from the consumer of the storage. To request a PersistentVolume, a *PersistentVolumeClaim* has to be created, which contains the desired properties of the volume and will be matched with a PersistentVolume that fulfills those criteria [Kubr].

In addition to the regular containers that a Pod consists of, it can also contain *initContainers*. These get executed in the order they are defined and before any of the regular containers are started. They are used to prepare the Pod for the application running in the containers, e.g. by creating configuration files. [Kubh]

2.1.4 Workloads and Controllers

Pods usually do not get created directly. Instead, workload resources can be created, which allow the abstraction of various use cases. Each kind of workload resource has a corresponding controller. Controllers provide the logic of workload resources. They monitor the API for the kind of objects they are responsible for and create Pods as specified in those objects. Depending on the use case the workload resources try to cover the controllers follow different strategies when doing so. [Kubc]

The simplest kind of workload is a *Job*. Jobs get executed once by creating one or more Pods, waiting for them to terminate, and reporting whether they have executed successfully [Kubi]. There are also workload resources that do not result in Pods getting created directly, but instead other workload resources are used to do so. One of these resources are *CronJobs*. They can be used to create *Jobs* in a regular interval [Kubd].

For deploying applications, there are multiple options available. For stateless applications, which Kubernetes is primarily intended for, a *Deployment* resource can be used. It starts a specified number of identical Pods and allows to scale up or down. Additionally, it helps with upgrades by allowing to do rolling upgrades[1] as well as rollbacks [Kubf].

It is also possible to deploy stateful applications with Kubernetes. In this case a *StatefulSet* can be used. Similar to Deployments, it also allows to deploy multiple instances, but ensures uniqueness and a consistent order. It supports upgrades as well, which also are executed in order and wait for each Pod to be fully running or fully deleted before continuing with the next [Kubv].

[1] A rolling upgrade only replaces one Pod at a time instead of all of them at once, allowing for faster recovery in case of errors.

The final kind of workload resource is the *DaemonSet*. It allows to run a Pod on all (or a subset of) nodes in the cluster. Its most common use is running networking plugins and monitoring agents [Kube].

2.1.5 Networking, Service Discovery and Load Balancing

Within a Kubernetes cluster each Pod has its own IP address. Containers within the Pods share a network namespace and do not have individual addresses. All Pods within a cluster can communicate with each other by default [Kubt]. That is only the specification, though, as Kubernetes does not provide that networking capability on its own. These capabilities have to be implemented through networking plugins instead. Networking plugins have to implement the *Container Network Interface* (CNI) in order to be used with Kubernetes. As mentioned in section 2.1.4, these networking plugins are often deployed using Kubernetes itself [Kubb].

There are multiple options regarding service discovery, all of which are controlled by creating *Service* objects. A *Service* uses labels to select the Pods that should be accessible through it. The service controller then creates *Endpoints* for each of them. It is also possible to manually create Endpoints if cluster-external services should be made available. Modern applications that are specifically designed to run on a Kubernetes cluster can query the API for these Endpoints and decide on their own which one they want to use. Additionally, Services are also exposed through environment variables and DNS entries, provided a DNS add-on is installed in the cluster[1]. For older applications, a cluster IP or a node port can be assigned to a service. Both provide basic load balancing and are available within the entire cluster. [Kubu]

2.1.6 Cluster Architecture

A Kubernetes cluster consists of a control plane, consisting of one or more servers, and multiple worker nodes (see figure 2.3). While it is possible to install a cluster on a single server, the control plane is usually separated from the nodes running the workloads. The control plane is scalable and can be used to manage hundreds or thousands of worker nodes as a single cluster.

[1] The de-facto standard add-on that serves this purpose is CoreDNS. https://coredns.io

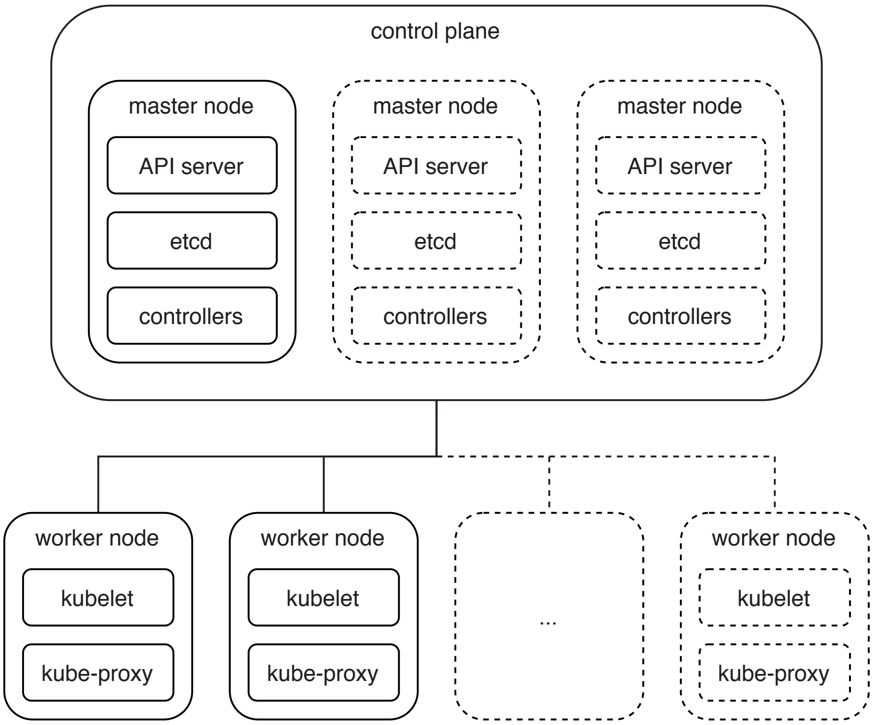


Figure 2.3: Architecture of a Kubernetes cluster.

The Control Plane

The control plane is the management layer of Kubernetes. It primarily contains the kube-apiserver (see section 2.1.1), which serves the Kubernetes API. The API server stores the API objects in *etcd*[1], a distributed key-value store. The aforementioned controllers are also part of the control plane. They are executed by the *kube-controller-manager*, which bundles the different controllers into a single unit for easier deployment.

Finally, the control plane contains the *kube-scheduler*. It takes care of scheduling the Pods to the compute nodes. It does so in a similar fashion as controllers: It monitors the API for unscheduled Pods, runs a scheduling algorithm, and updates the Pods with information about the node it was scheduled to. [Kubk]

Nodes

Nodes are the servers within a cluster that perform the actual work. In order to become a Kubernetes node, a server runs an agent called *kubelet*. Kubelet watches the API for Pods that have been assigned to the node it is running on and creates the containers that have been specified. It also takes care of monitoring the health of the containers and the applications within them and updates the Pods' status in the API. [Kubo]

[1] https://etcd.io

Nodes also run the *kube-proxy*, which takes handles the cluster IPs and node ports (see section 2.1.5) by forwarding the requests to the appropriate Pods or external endpoints. [Kubu]

In order to manage containers, kubelet talks to a container runtime (see section 2.2) through the *Container Runtime Interface* (CRI)[1]. The CRI defines actions to create, modify, monitor and delete containers, which the runtimes have to implement if they want to be compatible with Kubernetes. This allows any container runtime to be used with Kubernetes, without requiring explicit support by the Kubernetes project. [YuJ16]

2.2 Container Runtimes

Containers allow running processes in a controlled and isolated environment. The Linux operating system provides several mechanisms to provide such isolation. As configuring these mechanisms correctly is complex, container runtimes were created to simplify the process. The most famous among them is Docker, which was released in 2013. It is also the one initially used by Kubernetes.

The first container runtime for Linux was not Docker, but the *LXC* project[2], which started in 2008. The project is still active and the first versions of Docker were even using LXC internally. Docker then transitioned to its own *libcontainer*[3] library. With the launch of the *Open Container Initiative* (OCI)[4] in 2015, Docker extracted first *runc* and later *containerd* as independent projects. [Hyk15; Cro15]

Both runc and containerd are considered to be container runtimes, but containerd depends on runc internally and uses it to create containers, while itself focusing on managing those containers. Container runtimes can roughly be divided into higher- and lower-level ones.

2.2.1 Lower-Level Runtimes

In an attempt to make container runtimes more modular, the OCI introduced a *runtime specification* [Ope20a]. The specification includes a JSON based configuration schema, a lifecycle, and a set of actions that can be performed on a container. It does not, however, specify a full API to interact with runtimes following the specification.

[1] https://github.com/kubernetes/cri-api

[2] https://linuxcontainers.org/lxc/

[3] Which is heavily inspired by a tool from Google called *let me contain that for you (lmctfy)*.

[4] https://opencontainers.org

Any runtime that directly interacts with the operating system to create containers can be considered a lower-level runtime. The runtime specification of the OCI is supposed to be implemented by such lower-level runtimes. Higher-level runtimes then interact with lower-level runtimes using this specification. The best example for such a lower-level runtime in terms of the OCI is runc, which serves as the reference implementation of the runtime specification. It still uses Docker's libcontainer library internally and provides a command line interface as the only way of interaction [Hyk15]. A second example of a lower-level runtime that implements the OCI runtime specification is *Kata Containers*[1]. It provides lightweight virtual machines that behave like containers instead of relying on the process isolation capabilities of the Linux kernel.

Other examples for lower-level runtimes are the aforementioned *LXC* and Amazon's *firecracker* [2], a hypervisor for lightweight virtual machines which can also be used as a lower-level runtime for containerd, and as the hypervisor for Kata Containers.

2.2.2 Higher-Level Runtimes

Higher-level runtimes can be defined as runtimes that do not interact with the operating system directly, but use another lower-level runtime (both in a relative sense and following the definition in the previous section) to do so. The purpose of higher-level runtimes is to manage the containers and the images they can be created from, and to provide additional features like advanced networking. In the context of Kubernetes, higher-level runtimes are also the ones which implement the CRI to allow *kubelet* to interact with them.

The OCI runtime specification can help when interacting with the lower-level runtimes. Since it does not include a full API, support for the different runtimes still has to be implemented individually. The specification causes those runtimes to behave pretty similarly and allows for a consistent data model. [Ope20a]

As mentioned in the beginning of this section, *containerd* is a higher-level runtime that was extracted from Docker. Containerd implements the CRI, allowing it to be used directly with Kubernetes. Docker remains a container runtime which now uses the extracted containerd internally, and extends it with additional features [Cro15]. It is also the only runtime that has direct support built into kubelet and does not need to implement the CRI by itself [YuJ16]. At the time of writing the removal of said integrated support is in progress, though.

Other well-known runtimes that include an implementation of the CRI are *Podman*[3] and *cri-o*[4], both using runc as their lower-level runtime as well.

[1] https://katacontainers.io
[2] https://firecracker-microvm.github.io/
[3] https://podman.io
[4] https://cri-o.io

An example for a runtime that does not implement the CRI is *LXD*[1]. It is a higher-level runtime for *LXC* and attempts to provide a similar experience as virtual machines, but using LXC containers to do so.

2.3 Checkpoint-Restore

In order to migrate a Kubernetes Pod, the containers it consists of need to be migrated. Since containers are essentially just regular processes running in isolation, the problem can be generalized to migrating processes. Before explaining the functionality, the history of *Checkpoint-Restore* (C/R) will be laid out briefly.

The idea of creating a checkpoint of a process from which to resume later was already described in a patent by IBM, applied for in 1971 [And+73]. Later there were multiple implementations of C/R for distributed (operating) systems. By transferring the checkpoint to another system, processes were migrated to optimize resource utilization [RR81; PM83; TLC85; LLM88; AF89; Ste96; BL98] [2].

For the Linux operating system, there also exist several implementations of C/R that focus on transparency, operating in user-space only, or both [BW01; LH10; ZN01; Due05; MKK08]. The migration technique is considered transparent when it is not required to modify the program that shall be checkpointed in any way. In order to do so, the program that performs the checkpoint needs to access the memory of the target process. Accessing the memory of other processes requires privileged access in the Linux operating system, and therefore requires explicit support by the kernel. Because the task of retrieving the target process' memory was performed by the kernel in such solutions (e.g. [LH10; ZN01; Due05; MKK08]), they were not considered to operate in user-space only. Without transparency, it is much easier to solely operate in user-space. Solutions like [BW01] require the target program to include a library, which provides an interface for the checkpointing tool to access the memory. This way, no kernel operations are required.

Achieving both transparency and user-space operation greatly simplifies using C/R, as neither the operating system kernel nor the target program need to be modified. As transparency and operating only in user-space are contradicting, it is hard to find a solution that fulfills both goals. The first solution to do so is *Checkpoint/Restore In Userspace* (CRIU), which was released in 2012. While one could argue that it is not completely in user-space since it did require a few additions to the Linux kernel, most of those changes are, with a few excep-

[1] https://linuxcontainers.org/lxd

[2] Not all of these explicitly call their mechanism checkpoint/restore (or checkpoint/restart), but the concept is the same.

tions, generic, and can be used for other tasks as well. [Jon11; Jon12; Mic12] The most relevant tasks, first and foremost storing user memory, are executed in user-space. Furthermore, all required changes were accepted into the kernel, so CRIU can be used without any additional modification to the kernel. [RV14]

There are other user-space solutions for C/R like *Distributed MultiThreaded CheckPointing* [Ary+]. As the runtimes mentioned in section 2.2 only include support for CRIU, it is the only relevant solution for this thesis. After a short summary of how CRIU works, runtime support will be discussed further.

2.3.1 Checkpoint-Restore in Userspace

The general strategy is the same for all C/R solutions: freeze the process (or process tree), persist the state to disk, optionally transfer it somewhere, restore the state, and resume the process(es). CRIU is no different, it uses some special techniques to perform certain steps within user-space, though. The following section will describe how a regular C/R process works, which creates a full checkpoint from which to restore later. There are other methods that will be discussed in section 2.3.2. For simplicity it is assumed that only a single process is checkpointed, the technique is similar for process trees, though.

The first step, freezing the process, is necessary to prevent mutation of the process state as well as new child processes from being created. CRIU does so using the the *ptrace* system call[1], which allows to control the execution of a process [CRI17; Reb19, 2:45].

Next, the state of the process needs to be persisted to disk. The state consists of the process table entry, system memory pages, and open files (which on Linux include network sockets as well). To be able to access all of the resources the process is using without requiring kernel privileges, CRIU injects a piece of parasite code into the process it is checkpointing. The code starts a small daemon that CRIU then connects to. After all resources have been written to disk, the parasite code is removed again to maintain transparency in case the process continues to run. This depends on the flags provided by the user, which can be set to kill the process immediately after the checkpoint is complete, to avoid any further actions by the process. [Reb19, 3:04]

In order to restore a process, CRIU morphs itself into the process contained in the checkpoint. To do so, it first copies the memory from the checkpoint to the system memory. It then forks child processes (if a process tree is restored), recreates file descriptors, re-establishes

[1] Issuing a regular process signal like STOP would affect the process table entry, and the operation would no longer be transparent [CRI17].

network connections and reorganizes memory pages correctly. To complete the restoration, it then jumps into the checkpointed process at the exact instruction it was stopped at to continue its execution. [Reb19, 7:26]

2.3.2 Migration Strategies

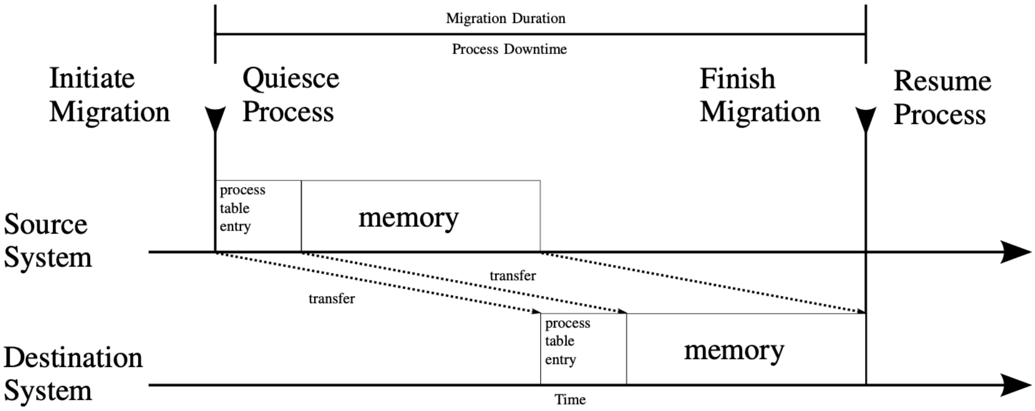
There are different strategies when it comes to migrating processes [Mil+00]. Four general strategies have emerged: creating and transmitting a full checkpoint , pre-copy, post-copy and a hybrid of the previous two [Pul+19]. The challenge is to balance time (downtime as well as total duration of the migration), network usage, and likelihood of failure.

If the downtime during a migration is very short (a few hundred milliseconds) and occurs with full networking transparency (and therefore no termination of active network connections), the migration is considered to be *live* [Cla+05]. With increasing amounts of memory to migrate, only the last three strategies are viable to achieve live migration.

CRIU, next to the previously described full checkpointing, also supports the pre- and post-copy strategies. A hybrid approach can be achieved by using both at the same time [Pul+19]. In the following, the first three alternatives (excluding the hybrid strategy) are briefly described and compared with each other with regards to the properties mentioned at the beginning of this section.

The easiest approach is creating a checkpoint, transferring it to another system, and restoring the process. Assuming the C/R process itself works, this process is the one with the least network usage and the lowest risk of failure: sharing files over a network is easy and reliable, and all data is only transmitted once. In exchange, this approach leads to a downtime that is equal to the duration of the complete migration process (see figure 2.4). If the checkpoint is not transferred while it is being created (which could be done using network storage) the process will take even longer, as the time it takes to transmit the checkpoint will be included as well. [RV14; Pul+19]

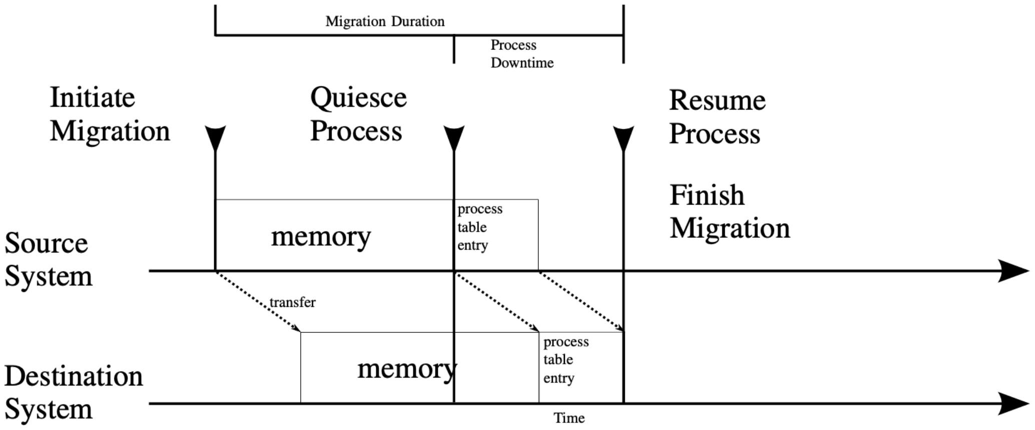
If memory data is transmitted while the checkpoint is still being created, the data that is already present on the target system can be used to start restoring the memory. This strategy is called *pre-copy* migration. Instead of freezing the process instantly, memory is copied to the target system in iterations, copying only memory that has changed since the last transfer. The process keeps running during this step. At some point the process gets paused, the remaining memory as well as other, smaller parts of the checkpoint are transferred, and the process is resumed on the target system. The smaller the amount of remaining modified memory, the faster the migration can be performed. In order to be effective, the point in time where the process is frozen has to be chosen carefully. This technique is also visualized



Source: [Reb16b]

Figure 2.4: Migration with a full checkpoint

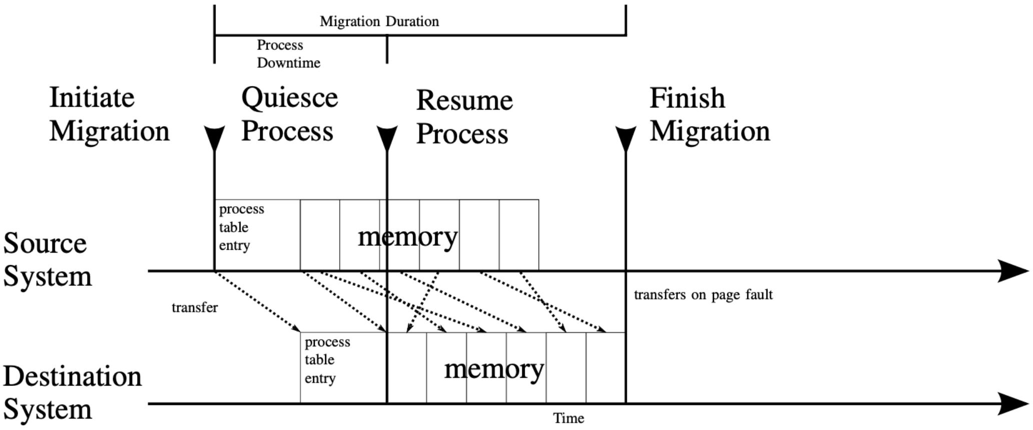
in figure 2.5. This strategy *can* be a lot faster in terms of downtime than performing a full checkpoint. Very memory-intensive processes can lead to a rather long total migration time and cause a high network load, as memory has to be copied multiple times. At the same time this strategy is still quite safe as the migration can be cancelled and the process resumed on the source node at any time. [RV14; Pul+19]



Source: [Reb16b]

Figure 2.5: Migration with pre-copy strategy

The pre-copy strategy can also be inverted to become *post-copy* migration. Here, the process is frozen immediately. The process table entry and other information is copied first, in order to resume the process on the target system as fast as possible. Memory is then forwarded from the source system when the process tries to access it, while also getting fully copied in parallel. The downtime is similar to pre-copy migration, but the total time required decreases significantly. On the other hand, the risk of failure greatly increases, as failing to forward parts of the memory in time can cause serious performance issues or crashes. In contrast to *pre-copy* migration, it is also much harder to recover from such a failure, since the process is already running on the target node. Additionally, the performance is heavily degraded during the migration, especially immediately after the process has been resumed because page faults take much longer to be fulfilled. [RV14; Pul+19]



Source: [Reb16b]

Figure 2.6: Migration with post-copy strategy

2.3.3 Checkpoint-Restore with containers

As explained at the beginning of this section, containers are merely isolated process (trees), and migration of the same is therefore equivalent to migration of processes. In fact, the isolation even simplifies migration, or restoration, to be more precise. Since the migration C/R procedure attempts to be as transparent as possible, properties like *Process Identifier*s (PIDs) are migrated as well. Because those have to be unique within a Linux namespace, the isolation based on such namespaces leveraged by containers can help avoiding collisions, which would cause the restoration of a checkpoint to fail. This is especially useful when migrating a container between systems, as collisions are more likely in that case. When staying on the same system a collision only occurs if another process that was started between creation and restoration of the checkpoint got the PID of the process to restore assigned. [Reb19, 9:57; Reb16a]

CRIU was initially developed for OpenVZ, which therefore also was the first container runtime to support C/R using CRIU [Kol12]. LXC/LXD and Docker implemented support in 2015, and containerd as well as runc therefore support it as well [Can15; Kaz15]. Other runtimes that are based on runc, like Podman and cri-o also support it [Pod], or are working on doing so at the time of writing [Reb20]. The implementations differ in some aspects, though. Docker and containerd create a checkpoint from the internals of a container. The configuration of the container itself is not included. If a checkpoint is to be restored, a matching container has to be created first [Doc18]. Podman on the other hand stores a full checkpoint that also includes the configuration of the container, and can be restored with a single command without requiring manual container creation [Pod].

CHAPTER 3

Related Work

The support for C/R by container runtimes described in section 2.3.2 lead to multiple evaluations of its performance in different scenarios and several implementations of container migration.

Puliafito et al. [Pul+19] evaluated the different migration strategies supported by CRIU and runc with regards to downtime and total migration duration. They limited bandwidth and increased network latency to imitate an edge cloud scenario. Their results are as expected, with full checkpointing having the longest, and post-copy and hybrid migration showing the shortest downtime.

Torre et al. [Tor+19] assessed the performance of Docker's support for checkpoint restore as a way of container migration. Since Docker only supports full checkpointing, pre- and post-migration are not considered. They compare migration downtime in multiple different scenarios with varying amounts of artificial network and cpu load.

Voyager [Nad+17] is a migration service for runc containers. It utilizes the aforementioned CRIU to migrate the container's state and uses union mounts to provide lazy migration of the container's file system in the background. This way, containers can be restored on the target host quickly, even if they hold a lot of data.

A proof of concept of container migration using CRIU was implemented for a bachelor's thesis [Win17]. Even though it was handed in in April 2017, it does not use the support for CRIU that was implemented for Docker in 2016[2], but presents a custom implementation instead.

Addad et al. [Add+20] compared different strategies for transferring the file system of containers for live migration in an edge cloud scenario, while using LXC and CRIU to perform pre-copy migration of the containers.

OpenVZ not only supports C/R itself, as mentioned in section 2.3, but also transparent migration of containers, including networking [MKK08].

[2] The relevant Pull Request can be found on GitHub at https://github.com/moby/moby/pull/22049

Zap [Osm+03] implements the same concept as Pod migration in Kubernetes. It provides isolated environments that contain multiple processes, which it also calls pods. These pods can also be migrated between different hosts transparently. They use a kernel module that performs C/R on the pods, which are then transferred between the hosts. The file system of the pods is mounted remotely, so it does not need to be included in the transfer.

A project that comes very close to Pod migration in Kubernetes is *KubeVirt*. It allows to run virtual machines as parts of Pods, which can be migrated between hosts. This way it is possible to migrate applications between Pods within a cluster, but the Pods themselves do not get migrated [Kubac]. The way migration is triggered and reflected in their API violates Kubernetes' API conventions though, and is therefore not a suitable approach.

There is one prototypical implementation of Pod migration for Kubernetes available on GitHub [Zhi19]. It is only a very rough prototype which extends the Kubernetes command line client with an additional command, and uses a slightly modified kubelet and an additional agent to perform the migration. It does not support triggering migrations using the Kubernetes API.

Cloudify, an orchestration platform and provider, claims to support live migration for Pods in Kubernetes clusters, but does not provide additional information apart from a demo video and the announcement blog post [Clo20].

CHAPTER 4

Conceptual Design

This chapter covers several design decisions that have to be made when implementing Pod migration in Kubernetes.

First, a set of goals is laid out to serve as a guide for the various decisions. Section section 4.2 covers how a migration should be defined and represented in the Kubernetes API. It is followed by an explanation of the components of a Pod that have to be migrated. The necessary changes to the CRI to support C/R are explained in section 4.4. Two aspects of the migration strategy that can help reducing the downtime are discussed afterwards. The last two sections cover networking and failure handling.

4.1 Goals

The primary goal is moving Pods between nodes within a Kubernetes cluster without terminating and restarting the Pods. In addition, there are several secondary goals that should be fulfilled in order to create a usable and useful addition to Kubernetes.

1. The API integration must be consistent with existing concepts of Kubernetes. The API is the most depended upon component and serves as the user interface of the cluster. Introducing new concepts requires consumers to adapt, which greatly reduces the usability.
2. Migrations should be as quick as possible, or more importantly, the downtime should be as short as possible. One of the main advantages of migration is avoiding long startup times, which would be cancelled out by slow migrations.
3. The implementation should be easy to use. This refers to manual or external invocation, e.g. a custom scheduler that optimizes resource usage by migrating workloads, as well as automatic migrations, for instance an eviction of a node or when a Pod gets deleted.

4. Migration needs to integrate well with Kubernetes' networking. There is no use in migrating applications if they cannot communicate properly afterwards. Ideally, the migration is fully transparent from a networking perspective.

4.2 Extending the Kubernetes API

Everything within a Kubernetes cluster is controlled through the Kubernetes API. Therefore Pod migration needs to be integrated into the API as well. The API is the most important component of a Kubernetes cluster, and therefore its extension is the most important aspect when implementing new features. As with any externally exposed API, changes are hard to revise later. The API must be capable of two things to support Pod migration:

- It must be possible to declare that a Pod should be migrated. Components of the cluster need to be able to determine that a migration should occur, or that a Pod is part of a migration. In addition, it should be possible to specify certain parameters regarding the migration process.
- It must be simple to trigger a migration. This relates to both integration with existing components and resources as well as having good usability for Kubernetes users.

Even though both capabilities overlap, there is a difference. The first focuses more on the internals of Kubernetes and the implementation complexity that goes along with it. The second is more concerned with usability. Since usability is hard to categorize on its own, implementation effort when using the API and similarity to existing API constructs will be evaluated instead.

4.2.1 Kubernetes API Conventions and Rules

In order to maintain a consistent interface, the Kubernetes project provides a set of API rules and conventions. When changing existing resources, specific requirements regarding compatibility must be fulfilled. In short, no new required fields can be added, and any change must not change the existing semantics of the API [Kubab]. When adding new resources to the API, the object schema must match the basic structure of all objects in the cluster as described in section 2.1.2. They are also restricted to the same create, retrieve, update and delete operations as all other resources. Any additional actions can be implemented as ephemeral resources that usually implement only a subset of those operations. These resources do not get persisted themselves, but modify other resources instead [Kubaa].

As explained in section 2.1.1, the API of Kubernetes is declarative. This declarative behavior complicates modeling one-time actions like restarting a Pod [1]. Nevertheless, one-time actions are still possible with the best example being Jobs. They are executed only once and get deleted by a garbage collector after they have finished execution [Kubi].

4.2.2 Cloning instead of Migrating

When thinking about migrating a Pod in the API for the first time, the logical approach that comes to mind is to simply move a Pod from one node to another. That would require to reschedule Pods, which does not fit their acyclic lifecycle. Each Pod starts in the *Pending* phase. Once it is scheduled to a node and all containers were created it progresses to the *Running* phase. If the containers of a Pod terminate, it either moves to *Succeeded* or *Failed*, depending on the exit codes of the containers. It can also be in the *Unknown* phase in cases of uncertainty, e.g. due to communication issues with the node. A Pod never moves back to the Pending phase, which would be required to reschedule it. Instead, when Pods are supposed to be moved or the node they run on becomes unavailable, they get deleted and the controllers that created them are responsible for creating a new Pod if necessary [Kubs]. While this keeps the lifecycle simple, it makes integrating migration difficult. Modifying the Pod lifecycle to support returning to the Pending phase and getting scheduled to another node would be difficult, as the impact on other components is hard to predict.

There is an action that fits the lifecycle much better than migrating Pods: cloning them. By cloning Pods there is no need to move Pods between nodes. Instead, a second Pod is created, which follows the regular Pod lifecycle. When it would usually create its containers, it copies them from the Pod that should be migrated instead. As the reason for Pod migration is preserving the state of the Pod, and not the Pod object, cloning a Pod and then deleting the old one is essentially the same as performing a migration.

Even though easier to integrate, cloning still requires changes to the API. Most importantly, there needs to be a way to specify that one Pod (from here on referred to as *target Pod*) should be a clone of another Pod (now called *source Pod*). Since the topic of this thesis is Pod migration, the process will continue to be referred to as migration, even though it is actually implemented by cloning. The exceptions are names that should reflect the actual behavior, and whenever cloning is referred to explicitly.

[1] Which is not supported by Kubernetes and just serves as an example.

4.2.3 Extending Pod Spec

Focusing on implementation complexity, the simplest approach to integrate Pod migration into the API is to extend the Pod specification. Adding a reference to the source Pod is sufficient to indicate that a migration should occur. When a Pod's spec contains such a reference, its containers would get cloned from the referenced Pod. Compared to the approaches that will follow, this one is very easy to implement in kubelet, as the Pod specification contains all relevant information directly, and kubelet does not have to request or monitor additional resources. Since such a reference is an addition to the Pod specification and an optional property, the mentioned compatibility rules would be fulfilled.

The migration procedure (see figure 4.1) would appear as follows, when observing only the API: At first (1), only the source Pod exists. The migration starts once a target Pod that holds a reference to the source Pod is created (2) and scheduled to a node. After the migration is completed, the source Pod gets deleted (3) and only the target Pod remains (4).

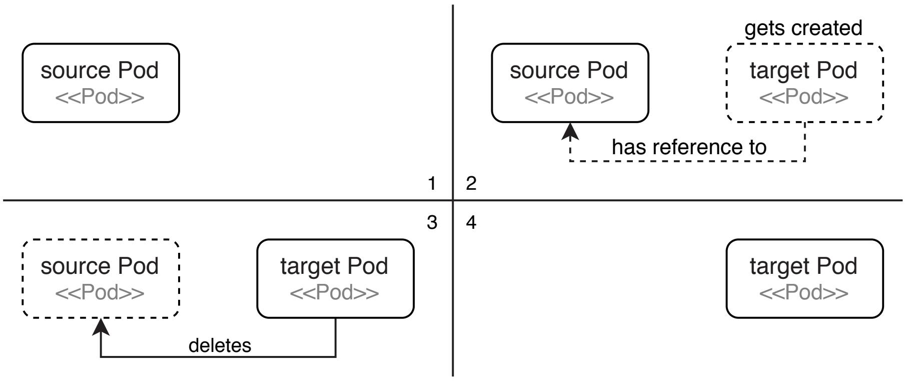


Figure 4.1: Migration process in the API when using a new field in the Pod spec.

There are several disadvantages to this approach, though. First, creating the target Pod will directly affect the source Pod, as all its containers get terminated during the migration (if they remained running, it would be a clone operation, not a migration). In the current API, objects do not affect each other unless they are both managed by the same controller. While other Kubernetes components should be able to handle this behavior, as Pods can also fail for external reasons, it would be an unpredictable side-effect for any other user than the one triggering the migration.

It is also hard to constrain which Pods can be migrated. Any Pod can be selected as the source Pod of a migration when creating a target Pod. This poses the risk of migrating the wrong Pod by accident. Doing so would not be successful in most cases, as the target Pod needs to be as similar as possible to the source Pod in order for the checkpoint restoration to succeed. A similar cause of failure is the specification of the source and target Pods being to different, e.g. by having different volume mounts configured. Both of these risks of failure can be lessened using validation, but it is hard to fully prevent accidental migrations of the wrong Pod. It is also difficult to find an elegant solution for further actions like deleting the source Pod or reconfiguring networking, as such tasks would usually be performed by controllers, which do not exist for Pods.

Furthermore, it is hard to specify parameters of the migration process. One example of such a parameter, if cloning should be supported as well, would be whether the source Pod should continue to run after a checkpoint was created. While such parameters could be defined on the target Pod, as this is the one initiating the migration, they could affect both Pods and it can therefore be argued that they should be present on the source Pod (or even both) instead.

There are also issues when it comes to triggering the migration. With this approach the only trigger for a migration is creating a target Pod manually. But the usual mechanisms in Kubernetes that cause a Pod to be moved do so by deleting it. In order to migrate Pods instead of deleting them, all of those mechanisms would need to be adapted to create a target Pod and wait for the migration to finish. This would lead to a lot of implementation effort and most likely to duplicated implementations as well. The same applies to workload controllers that should leverage migration, which would require it to be implemented for each kind of workload individually.

4.2.4 ClonePod Task

Instead of modifying the Pod specification, a migration or (to be more precise) a clone task can be defined as a resource in the API. Such a resource (e.g. called *ClonePod*) would then, in conjunction with a controller, take care of creating the target Pod, tracking the status, and deleting the source Pod. Similar to *Jobs* it could get garbage collected after the migration is finished.

Figure 4.2 shows how a migration using a ClonePod would be represented in the API. With the source Pod already existing, a newly created ClonePod assumes control (1) over that source Pod. The source Pod is then owned by the ClonePod, which is represented in the metadata of the Pod. The ClonePod's controller then creates (2) a target Pod, which is also owned by the ClonePod. To figure out whether a Pod should clone another can be determined by looking at its owner: if it is a ClonePod object, a migration or clone operation should occur. After the migration is complete, the ClonePod's controller deletes (3) the source Pod. Finally, the ClonePod releases the control (4) over the target Pod and gets garbage collected eventually.

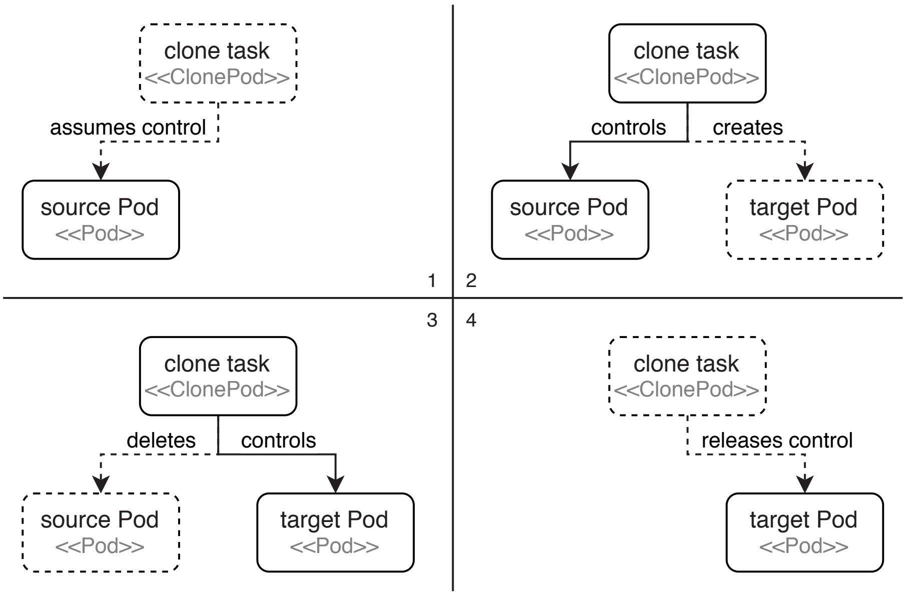


Figure 4.2: Migration process in the API when using ClonePod task.

This solves several problems of the previous approach. Pods no longer affect each other directly. Instead, a controller takes control over them for the duration of the migration. This also allows to specify parameters independently of the Pods and makes it easier to track the status of the migration. The process becomes safer and easier to use as well: the target Pod no longer needs to be created by the user or component requesting a migration; the clone controller can do it instead. This leads to less errors due to incompatible source and target Pods, as the controller can create an exact duplicate automatically. While it is still not possible to react to Pod deletions, adapting existing mechanisms that cause them is easier, as they only have to create a ClonePod task instead of a complete Pod, which reduces implementation effort. As mentioned in the beginning, the controller can also take care of further actions like deleting the source Pod or reconfiguring the network. It does so in a cleaner way than the previous approach, as there is clear ownership for the duration of the migration.

All these benefits come at the expense of higher implementation complexity. If only an object is used to trigger a migration, any component that needs to be aware of it has to constantly observe which object owns a Pod to notice when a migration occurs. While the Pod is not part of an ongoing migration, it is unclear whether it will ever get migrated or was migrated in the past.

With a ClonePod task it is also still possible to clone any Pod, and therefore potentially the wrong one. Since the ClonePod controller takes care of creating the new Pod and ensuring compatibility, accidental migrations are less likely to fail than with the previous approach, though.

4.2.5 MigratingPod

If the implementation should be more in line with current Kubernetes concepts, adding a new *MigratingPod* resource is the best approach. A MigratingPod is used like any other workload controller. If a Pod should be moved between nodes by migration instead of deleting and recreating it entirely, a MigratingPod has to be created instead of a regular Pod. The MigratingPod's controller then creates a regular Pod according to the template contained in the MigratingPod's specification.

A migration, visualized in figure 4.3, would then start with both the source Pod and the MigratingPod already existing (1), and the MigratingPod owning the source Pod. When a migration is triggered, e.g. by a modification of the MigratingPod's specification, the MigratingPod's controller creates (2) a target Pod. Similar to the ClonePod, a Pod can be identified as a target Pod of a migration by checking whether it is owned by a MigratingPod. After the migration is complete, the source Pod is deleted (3). Both the target Pod and the MigratingPod remain (4), with the MigratingPod continuing to own the target Pod created by its controller.

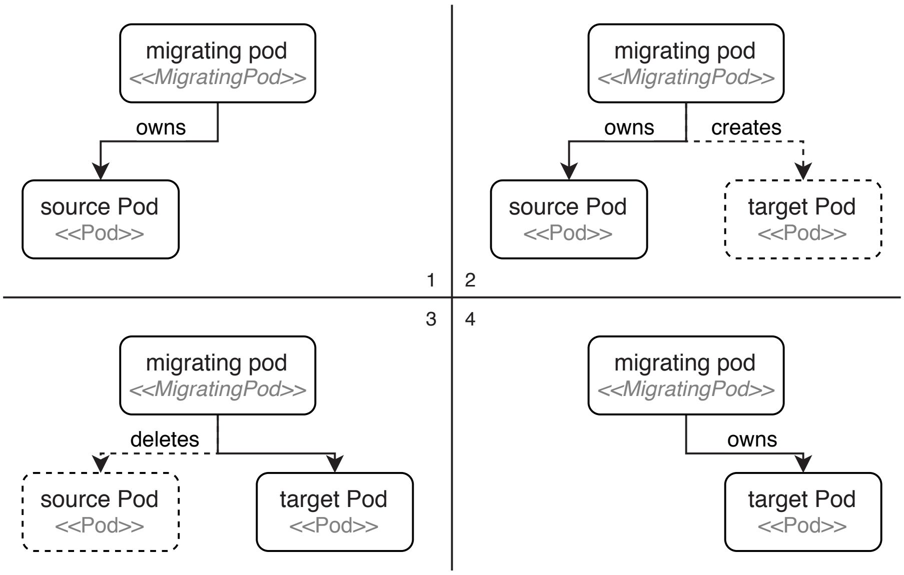


Figure 4.3: Migration process in the API when using a MigratingPod.

The main difference between a MigratingPod and the previously described ClonePod is the ownership of the Pods. The ClonePod is only temporarily related to the Pods for the duration of the migration. After the migration completes, the ClonePod is deleted. The MigratingPod instead owns the Pods as long as they exist. In order to delete the Pod, the MigratingPod has to be deleted.

This tight control of the MigratingPod controller over its Pods prevents accidental migrations of the wrong Pod. Migrations can get triggered easily by modifying the specification of the MigratingPod. With the help of Finalizer it is even possible to implement migration on deletion of Pods. As explained in section 4.2.1, Pods are usually moved between nodes by deleting them and relying on their controller to create a new one that gets scheduled somewhere else. By adding a Finalizer, the deletion of Pods owned by MigratingPods can be delayed until a migration is performed, which allows MigratingPods to integrate with existing mechanisms that move Pods within a cluster (e.g when draining a node for maintenance).

Integrating MigratingPods into existing controllers is simple as well. For example, Jobs could be extended with a field that indicates whether its Pods should migrate on deletion. The controller can then create a MigratingPod instead of a regular one to realize this behavior[1].

One aspect can be seen as both an advantage and a disadvantage: the MigratingPod is specific to migration. While it would be possible to implement a controller that is flexible enough to support both, this complicates both usage and implementation. But apart from relying on the same technique (C/R and transferring the checkpoint), the remaining aspects of cloning and migration are quite different. Therefore, while it is more work to implement and maintain an additional controller for cloning, the separation makes sense. Another disadvantage of the MigratingPod is a loss of flexibility, as Pods have to be created with later migration in mind.

4.2.6 Summary

Since the first of the two required capabilities laid out at the beginning of this section is a fundamental technical requirement for implementing Pod migration, all of the proposed solutions fulfill it. Therefore the second one will be decisive. The only advantages of extending the Pod spec are reduced implementation complexity within Kubelet, and very high flexibility. After further consideration it becomes clear that the former gets cancelled by increased complexity when integrating migration with other components, and the latter yields a higher risk of errors.

Using a ClonePod resource significantly improves the usability as users no longer have to create the target Pod themselves, which also ensures the target Pod is compatible to the source Pod as it is created automatically. This also leads to less implementation effort for triggering a migration, both for users as well as within other components. Since the ClonePod is accompanied by a controller, additional tasks (e.g network configuration) and the deletion of the source Pod can be performed in a clean way.

[1] Of course it is not a drop-in replacement and heavily depends on how MigratingPods are implemented. But there should not be any logic required to trigger the migration, only to handle it properly if one occurs.

Of all approaches, using a MigratingPod to manage Pods that should be migratable is the most in line with existing Kubernetes concepts. Pods are rarely created directly, but usually by workload controllers instead. Using a controller that owns the Pods that should migrate also provides a clear hierarchy and avoids seemingly independent objects effecting each other, which can be hard to predict. In addition, they are even easier to integrate with other components. Jobs could for example simply exchange regular Pods with MigratingPods to enable migration. Finally, it is the only option that allows to implement migration on deletion. All of these advantages only come at the cost of flexibility.

	extend Pod spec	ClonePod	MigratingPod
kubelet impl. complexity	◔	◕	◕
flexibility	◕	◑	◔
integration complexity	●	◑	◔
risk of (user) errors	◕	◑	◔
user implementation effort	◕	◑	◔
consistency with Kubernetes	✗	✗	✓
data correctness	✗	✓	✓
handle additional tasks	✗	✓	✓
migrate on delete	✗	✗	✓

Table 4.1: Summary of the comparison of implementation alternatives

Because of the limited amount of use cases for Pod migration in Kubernetes, flexibility is not very important. Since the MigratingPod approach is the best option in almost other facets, it is the best fit for Kubernetes from the proposed alternatives (see table 4.1 for another brief overview).

4.3 Pod Components to Migrate

A Kubernetes Pod consists of more than just containers. As explained in section 2.1.3, they also contain init containers, mounted network volumes, configuration values and metadata. Not all of these Pod components have to be migrated, though.

As explained in section 2.1.3, Pods provide an isolated environment that is partially shared between its containers. This environment is called the *sandbox*. How this sandbox is implemented is up to the container runtime's CRI implementation. In the case of containerd, the sandbox is an additional, very small container that acts as an anchor for shared Linux namespaces [YuJ16]. Since this sandbox is just the environment for the containers to run, and its state is not supposed to affect the containers within it, it will not be migrated.

In addition to regular containers, a Pod can also contain init containers. These init containers are executed one after another when a Pod is first started and before the regular containers are created. While those init containers are executed, the Pod is not yet considered to be running. Therefore, if a migration is to occur, the Pod should just get terminated and started regularly on the target node. Init containers therefore do not need to be migrated. When using a controller to perform the migration, the init containers could just get stripped from the target Pod. Otherwise init containers need to be skipped when a Pod contains a reference to a sourcePod.

Storage is either mounted from a remote storage provider or from the local node. There is no need to migrate remote storage, but the *AccessMode*, if set to *ReadWriteOnce*, of a PersistentVolume can prevent a volume from being mounted by multiple Pods at the same time, which would prevent the target Pod from mounting it while the source Pod still exists. Therefore it either needs to be a requirement that the AccessModes of used PersistentVolumes are set to *ReadOnlyMany* or *ReadWriteMany*, which are not supported by all volume types, or it must be possible to transfer a PersistentVolume between two Pods that are part of a migration. While it would be possible to migrate local storage as well, it would increase the amount of data that has to be transferred between nodes, and would lead to longer downtimes, which is contradictory to the goals.

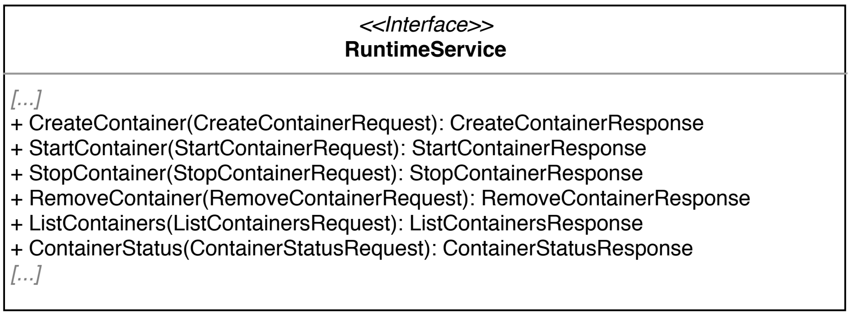
Configuration and secrets also do not need to be migrated since they live in the API. They can just be attached to the new Pod as well.

4.4 Extending the Container Runtime Interface

Initially, kubelet only supported Docker as a container runtime. In order to support other runtimes as well, the CRI was defined. It is an interface that can be implemented by any container runtime in order for it to be usable with Kubernetes. It consists of two interface definitions: the *RuntimeService*, which is used for managing Pod sandboxes and containers, and the *ImageService*, which is used to pull images. A subset of the container management methods of the RuntimeService can be seen in figure 4.4.

Since C/R of containers is performed by the runtime, and the CRI is the interface between kubelet and the runtime, the CRI needs to support C/R as well. Following the existing scheme, two new methods can be added to the RuntimeService: one that creates a checkpoint, and one that allows to restore one (see figure 4.5).

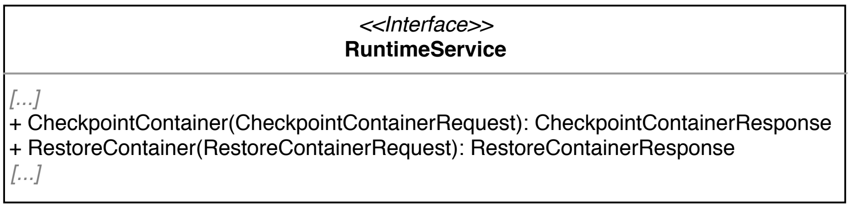
When it comes to the parameters of those methods, a few more aspects have to be considered. First, the parameters depend on the behavior of those methods. There exist at least



<<Interface>>
RuntimeService

[...]
+ CreateContainer(CreateContainerRequest): CreateContainerResponse
+ StartContainer(StartContainerRequest): StartContainerResponse
+ StopContainer(StopContainerRequest): StopContainerResponse
+ RemoveContainer(RemoveContainerRequest): RemoveContainerResponse
+ ListContainers(ListContainersRequest): ListContainersResponse
+ ContainerStatus(ContainerStatusRequest): ContainerStatusResponse
[...]

Figure 4.4: The container related methods of the CRI RuntimeService.



<<Interface>>
RuntimeService

[...]
+ CheckpointContainer(CheckpointContainerRequest): CheckpointContainerResponse
+ RestoreContainer(RestoreContainerRequest): RestoreContainerResponse
[...]

Figure 4.5: New methods to be added to the RuntimeService for checkpoint/restore.

two different behaviors when it comes to command line interfaces. Docker's checkpoint and restore[1] commands only handle the processes inside of a container. In order to restore the processes, a container needs to be created first and then a checkpoint can be restored inside it [Doc18]. In contrast, Podman does not require a container to be created before restoring. The checkpoints it creates include the specification of the container, and an identical one will be created when restoring.

While the latter approach might be more user friendly when manually using the runtime on the command line, it is not a good fit for Kubernetes. By restoring the container specification from a checkpoint, the container matches the specification of the source Pod, but not necessarily the one of the target Pod. This would then be considered a deviation from the desired state that kubelet will attempt to correct, which, in the worst case, can lead to the container getting recreated. Including the container specification in the checkpoint would ensure that the containers are identical, which makes the migration safer as restoration will fail if the containers are too different. But this should be ensured by the Kubernetes API, not by the runtime.

The behavior and parameters of the methods will therefore follow Docker's approach and require an existing container to restore a process. As shown in figure 4.6, some parameters are encapsulated into additional structs. This is intended to make it easier to add more parameters in the future, e.g. to support different migration strategies. This will also be further explained in section 5.2.

[1] There is no specific restore command in dockers CLI. Instead, a checkpoint to restore can be passed to the start command.

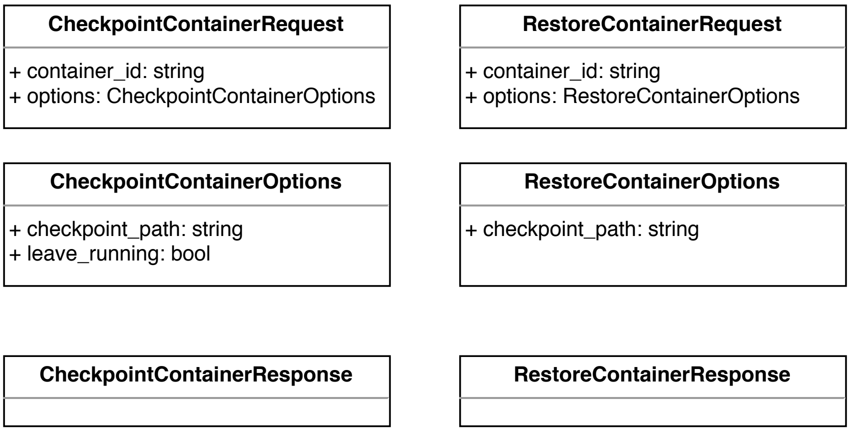

CheckpointContainerRequest
+ container_id: string
+ options: CheckpointContainerOptions
RestoreContainerRequest
+ container_id: string
+ options: RestoreContainerOptions
CheckpointContainerOptions
+ checkpoint_path: string
+ leave_running: bool
RestoreContainerOptions
+ checkpoint_path: string
CheckpointContainerResponse
RestoreContainerResponse

Figure 4.6: New parameter structs to be added for C/R.

4.5 Migration Strategy

The strategy for migrating the containers from one node to another is constrained by the capabilities of CRIU. As explained in section 2.3.2, there are tree different options, each with its own strengths and drawbacks with regards to time, success rate and network usage. The higher-level runtimes mentioned in this thesis (Docker, containerd, runc and cri-o), if at all, only support the first (and simplest) strategy: creating full checkpoints. Therefore, the focus will be on that strategy for the remainder of this work.

The migration process requires communication and synchronization between the two nodes involved. Usually all communication is performed through the API, and nodes are not required to directly talk to each other. On the other hand, the networking specifications of Kubernetes require Pods on different nodes to be able to communicate. In addition, the pre- and post-copy strategies would also require direct communication between nodes. To reduce complexity, and to simplify adding support for the more advanced strategies later, the migration process will require direct communication.

In order to perform the migration two problems have to be solved: The checkpoint has to be transmitted from one node to the other, and the process needs to be synchronized between the nodes.

4.5.1 Transmitting the Checkpoint

Multiple options for transmitting the checkpoint are available. The runtimes that support C/R require a path to store the checkpoint. Any file based transfer method can therefore be used. The simplest approach would be storing the checkpoint locally, transmitting it to the target node (e.g. using *Secure Shell* (SSH) or a http endpoint in kubelet), and restoring from there. This approach requires the checkpoint to be written to and read from disk twice, in addition to a single network transfer. In theory the checkpoint could be held in memory instead of disk persistence, which would eliminate disk speed as a bottleneck, but that would require enough free memory to do so.

By mounting a folder from the target node to the source node (or vice versa), the checkpoint could be written to the target node (or read from the source node) directly. This would eliminate one of the read and write operations, and the checkpoint is still transferred over the network only once. In addition, this approach is also transparent to the container runtime, and requires no manual transfer of the checkpoint.

One of the nodes has to act as a file server for the duration of the migration, though. Since constantly mounting shared folders between all possible pairs of nodes in a cluster would lead to a quadratically growing number of connections, the mounts would have to be created on demand. Whenever a migration is to occur one of the nodes has to be configured as a file server, and the other needs to mount the shared folder.

A dedicated file server can be a better solution instead. Both nodes can mount the same volume from the file server to exchange the checkpoint. The checkpoint will be transmitted over the network twice, but none of the nodes has to act as a file server temporarily, which is more complicated than merely mounting an already configured volume. In addition it is even possible, at least in small clusters, to constantly mount the shared folder of the file server, as all nodes can use the same one. With larger clusters the storage management functionality of Kubernetes, namely PersistentVolume, could be leveraged to mount storage on demand [Kubr].

4.5.2 Choreographing the Migration

As explained in section 2.3.2, the downtime of the full checkpoint strategy begins with the start of the checkpoint creation, and ends once the checkpoint is fully restored. In order to keep the downtime as short as possible, the time between the checkpoint and restore steps has to be minimized. The previously discussed transmission strategy has the largest impact on that. There are further steps that take up time and can be optimized, though.

The migration sequence is visualized in figure 4.7 and begins with the target node finding (1) a new Pod that is part of a migration. Since the CRI extension discussed in section 4.4 requires a container to be created before it can be restored, this step (2) is performed first. The container creation also causes the necessary images to be retrieved, which can take a long time depending on image size and the transfer speed of the image registry. Once the containers are created (but not started yet) the kubelet on the target node requests (3) a migration of the source Pod from the source node. The source node creates (4) checkpoints of the source Pod's containers and responds (5) with the locations of the checkpoints as soon as the creation is complete. The process then continues on the target node with the restoration (6) of the created checkpoints.

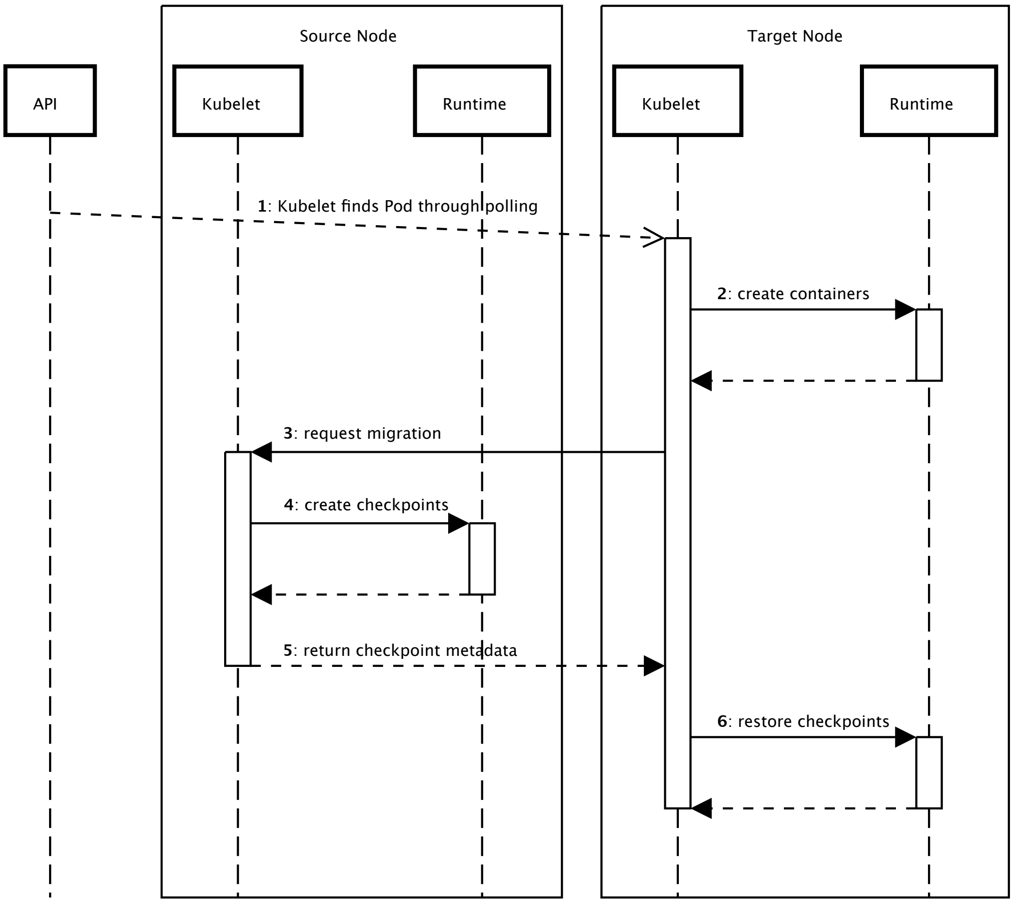


Figure 4.7: Sequence diagram of the migration strategy. The diagram is simplified and leaves out a few steps for brevity.

Due to internals of kubelet, which will be further explained in chapter 5, migration is executed on the Pod level. The steps 2, 4 and 6 in the visualization handle all containers of the Pod in parallel. Without this limitation the migration could probably be optimized further as smaller containers (in terms of memory usage) could be migrated more quickly and free up resources (especially network bandwidth) for the remaining containers.

When restoring the containers, the best approach would be to resume the processes in all containers at the same time. As mentioned before, containers using less memory might be restored faster than ones with higher consumption. If the network connections within the Pod can be restored, resuming all containers synchronously could prevent timeouts and other network errors. Since it is currently not possible to restore containers in a paused state, this will not be considered further, though.

4.6 Networking

Ideally the networking integration of Pod migration would be fully transparent to other Pods and cluster external actors. The easiest way of achieving this would be to migrate the IP address with the Pod. Kubernetes generally relies on CNI plugins when it comes to networking [Kubb]. As this includes *IP address management* (IPAM), it strongly depends on the networking plugin (or, more precisely, the IPAM plugin) in use, whether the IP address of a Pod can be migrated [Kubn].

Instead of migrating the IP, Services can be used to mask the changing IP address. Services are available in multiple variants, some of which allow to hide the IPs of the Pods belonging to the Service. The three options are *ClusterIP*, which exposes a single virtual IP in the cluster, *NodePort*, which exposes a service on a configured port on each node in the cluster, and *LoadBalancer*, which uses an external load balancer (e.g. from a cloud platform) to achieve essentially the same as *ClusterIP* with the addition of being available outside of the cluster as well. The purpose of Services, next to service discovery, is increasing availability. A service can be mapped to one or more Pods. Each Pod contains an array of conditions, which includes a *Ready* condition that indicates whether a Pod is able to handle requests. Services respect this condition and only forward requests to Pods that are ready. If the Ready condition is properly updated during the migration, Services should be able to abstract the migration of the Pod from a network perspective.

This abstraction alone will not provide full transparency though. To have that property, active network connections would need to survive the migration. Keeping TCP connections established without migrating the IP is very hard, though. As TCP connections are identified by the IP and Port combinations of both participants, an IP address cannot change without breaking the connection [Pos81]. A solution to that could be a proxy that forwards existing connections to the new IP address using Network Address Translation until they get disconnected. As the kube-proxy is running on all nodes and takes care of ClusterIP and NodePort services already it could be extended to do so. As handing over TCP connections between processes is not trivial, this solution would go beyond the scope of this thesis. The existing features of Services will be used instead, waiving the possibility to achieve network transparency.

4.7 Handling Failure

Many things can go wrong during a migration. On the source node the creation of the checkpoint, or, when using more advanced techniques, the preparation of the migration can fail. The same can happen on the target node when restoring a container. If network connectivity is lost, the transmission of the checkpoint can fail, which also happens if the shared storage used for transferring a checkpoint becomes unavailable. Finally, the application within the Pod could crash after the migration.

The most straightforward approach to handle failure during a migration, be it a process or a virtual machine, is to abort the migration and resume execution on the source node. This works for full checkpoints as well as pre-copy mechanisms, but will not for post-copy. With a full checkpoint it is even possible to retry restoring on another node. Integrating such an abort mechanism in Kubernetes incurs additional complexity. When the migration is triggered by deletion, it will not be possible to cancel the deletion of the source Pod, as that would break the Pod lifecycle. If the migration was triggered by another method that does not directly delete the Pod (e.g. modifying a MigratingPod), aborting the migration would only require to update the Pod status properly.

For some failures it will not be possible to recover though. If the source node becomes unavailable before the checkpoint is fully created, a migration will not be possible. The same applies to a pre-copy migration. For post-copy, both nodes have to be available until the migration is complete.

If the application crashes after the migration (meaning all containers were restored successfully), and a full checkpoint was used to migrate, it is possible to restore the container again, as the checkpoint still exists. It is not a practical approach though, as it can not be verified when exactly the application crashed. Restoring the same checkpoint multiple times could lead to application logic getting executed multiple times as well, causing unpredictable behavior.

CHAPTER 5

Implementation

In order to evaluate the design decisions that were discussed in the previous chapter, a prototype was implemented, which is described in detail in this chapter. Not everything is implemented as designed, though; mostly to reduce implementation effort.

Most importantly, the prototype does not solely use a MigratingPod to represent migrations in the API, but uses a hybrid approach of MigratingPod and an extension of the Pod spec. The latter is much easier to implement and allowed to develop the prototype incrementally, focusing on the actual migration first, followed by improvements to usability and integration.

Another simplification for the prototype was done with regards to the checkpoint transmission. Instead of the more flexible approach of dynamically mounting PersistentVolumes, a pre-configured volume will be used. Furthermore, there is no failure handling implemented as it is difficult to do when using deletion to trigger the migration, which evolved to be the primary method used in the prototype. The prototype also does not handle init containers as described in section 4.3.

5.1 Components

The prototype consists of three components: Kubernetes, containerd and a migration operator. Of the different Kubernetes components, which are all contained in a single repository[2], *kubelet*, *kubectl*, *kube-apiserver* and *kube-controller-manager* were modified. The modified versions are based on version *1.19.0-beta.2*[3].

[2] https://github.com/kubernetes/kubernetes

[3] The exact commit is `27687161d9ac0566c64a6db6f1fa2d393b6fd2f8`, which is 10 days older than version *1.19.0-beta.2*.

Containerd is used as the runtime for the prototype. Modifications ware made based on commit `35e623e` of the containerd CRI plugin. Runc version *1.0-rc92* is used as the lower level runtime.

Runc relies on CRIU to perform C/R. During development of the prototype, CRIU version *3.14* was used. Runc instructs CRIU to evaluate an additional configuration file[1], which allows to set additional CRIU parameters when using it with runc. This configuration file was used to enable the following two configuration options of CRIU:

- `tcp-established`
  CRIU supports migration of established TCP connections. This option explicitly enables this feature. Omitting it would cause the creation of the checkpoint to fail when TCP connections are established.
- `tcp-close`
  Restoring established TCP connections only works when the IP address that is used in that TCP connection is available (see section 4.6). Setting this option causes criu to close open TCP connections contained in the checkpoint when restoring the container.

5.2 Extending the CRI

The CRI is defined in the *Protocol Buffer* (protobuf)[2] language and implemented as a gRPC[3] client and server. As explained in section 4.4, the CRI consists of two interfaces: the `RuntimeService` and the `ImageService`. Both are implemented as `services` in protobuf, as well as Go interfaces (where the latter is called `ImageManagementService`).

The extension of the CRI consists of three steps. First, the new methods that were defined in section 4.4 need to be added to the protobuf services. Next, the client implementation within kubelet has to be adapted, and finally the new methods need to be implemented for the container runtime in use: containerd.

5.2.1 The Protocol Buffer Specification

Two new methods were added to the RuntimeService service in the protobuf definition. A service in protobuf is simply a collection of *Remote Procedure Call*s (RPCs), which consist of a name, a request and a response. The requests and the responses are `messages`, which can be

[1] Located at `/etc/criu/runc.conf`

[2] Protobuf is a language- and platform-neutral data-serialization format from Google. https://developers.google.com/protocol-buffers

[3] gRPC is a remote procedure call framework. https://grpc.io

compared to structs in programming languages like C or Go. The RPC and message definitions can be seen in figure 5.1. The aforementioned gRPC client and server are automatically generated from the protobuf description.

```
service RuntimeService {
    // [...]
    rpc CheckpointContainer (CheckpointContainerRequest)
        returns (CheckpointContainerResponse) {}
    rpc RestoreContainer (RestoreContainerRequest)
        returns (RestoreContainerResponse) {}
    // [...]
}

message CheckpointContainerRequest {
    string container_id = 1;
    CheckpointContainerOptions options = 2;
}

message CheckpointContainerResponse {}

message CheckpointContainerOptions {
    string checkpoint_path = 1;
    bool leave_running = 2;
}

message RestoreContainerRequest {
    string container_id = 1;
    RestoreContainerOptions options = 2;
}

message RestoreContainerResponse {}

message RestoreContainerOptions {
    string checkpoint_path = 1;
}
```

Figure 5.1: The RPCs and messages that were added for C/R.

The Go interfaces were extended by following the existing implementation. The two new methods were added with the fields of the request and response messages as their call and return parameters. When the gRPC implementation is generated, the protobuf messages are turned into structs, allowing the `CheckpointContainerOptions` and `RestoreContainerOptions` to be used as parameters of the methods. The Go interfaces are also the reason why the options messages and structs exist: changing the parameters within the options does not change the method signatures, which makes it easier to adapt the implementations by different container runtimes to changes.

5.2.2 Kubelet's Remote Runtime

Kubelet heavily uses interfaces to keep itself modular and facilitate testing. This also applies to the interaction with container runtimes. One of them is the CRI's `RuntimeService`, which is implemented by the `RemoteRuntimeService`, a thin wrapper around the generated gRPC client that extends it with logging. The new methods were implemented by mimicking the existing implementation, which is very similar for all methods.

There is another implementation of the interface, the `instrumentedRuntimeService`, which also needs to be updated with the new methods. It wraps a `RuntimeService` with metrics collection and the new methods can follow the existing implementation as well.

5.2.3 The containerd CRI Plugin

Generally speaking, the CRI is implemented by providing a gRPC server that implements the protobuf definition of the CRI. As explained before, the CRI already provides an automatically generated gRPC server that fulfills this requirement. The runtime only needs to implement a Go interface to provide the runtime specific logic. With the extension of the CRI, that Go interface was also extended with the two new methods, which now have to be implemented by containerd's CRI plugin as well.

Creating the checkpoint only requires calling the internal method of containerd with the checkpoint path and the identifier of the container. If the parameter `leave_running` is set to false, the container is killed after the creation is complete.

Starting and restoring a container is a very similar action in containerd. Both are handled by the same internal function. A container can be restored by providing an additional parameter to that function. The existing `StartContainer()` method can therefore be reused and its logic was extracted into a new function that gets called by both the start and restore methods, with the latter additionally passing the path to the checkpoint.

5.3 Extending the API

As mentioned in the beginning of this chapter, a hybrid approach of MigratingPod and a reference in the Pod spec was implemented for the prototype. The MigratingPod resource is implemented as part of an operator that is described in section 5.4.3. This section will only cover the extension of the Pod resource with a reference to a source Pod to migrate from.

5.3.1 Internals of the kube-apiserver

In order to support versioning, the kube-apiserver contains multiple Go types for each resource. One is used as the internal representation and the others are the versioned variants. The kube-apiserver always converts the versioned representations into the internal representation when receiving an object, processes it internally, and then converts it back into a versioned one when responding. When an object is persisted in etcd, its latest versioned representation is used as well. Using this internal representation, any versioned representation can be converted into any other version by first converting to the internal and then to the desired one. [Kubab]

Most of the conversion logic is generated automatically. In cases where automatic conversion is not possible, manual conversion functions are used. Therefore modifying a resource can sometimes require to adapt such manual conversion functions as well. Since validation is also performed against the internal representation, only one validation function exists for each resource.

5.3.2 Extending the Pod spec

Section 4.2.1 laid out several rules and conventions with regards to changing the existing API. Even though these rules are not as important when only implementing a prototype, all necessary changes fulfill them. As only completely new and optional fields were added, they they do not require special care.

There are also rules regarding API versioning. Since the goal is implementing a prototype, these were ignored in favor of easier implementation. The extension of the Pod spec was applied to the current stable version `v1` and the internal representation of course.

The versioned variants and the internal representation are defined as Go structs. The fields of those structs have several tags[1] that indicate how they should be serialized in protobuf and JSON. The latter also defines how the field is called in the YAML representation.

The reference to the source Pod was implemented as a new `clonePod` field in the Pod spec (see figure 5.2). The reference only consists of the name of the target Pod. Since namespaces provide quite strict separation, it does not make sense to clone a Pod from another namespace, and as only Pods can be cloned, only the name of the Pod is required. Comments are used to provide arguments for code generation. In this case the `+optional` parameter is provided to indicate that the field can be omitted, which is relevant for automatically generated validation.

[1] Tags allow to add attributes to fields of a struct that can be retrieved using reflection.

```
type PodSpec struct {
    // [...]
    // ClonePod is a reference to a pod that should be cloned by this pod.
    // This parameter is only relevant when a pod is first created.
    // +optional
    ClonePod string json:"clonePod,omitempty" protobuf:"[...]"
}
```

Figure 5.2: The `clonePod` field that was added to the PodSpec go types. In the struct definition the name of the field has to start with an uppercase letter due to the way name exporting is handled in Go. The value of the `protobuf` tag is omitted for brevity.

The automatic conversion functions are able to handle the addition without manual modification. No validation, apart from the generated one, was implemented for the prototype.

5.4 The MigratingPod

Like all workloads in Kubernetes, the MigratingPod consists of a resource and a controller. The MigratingPod resource allows to specify the desired state through the API, and the migration controller works towards achieving that state.

The controller has to take care of creating a Pod when a MigratingPod is first created and of migrating the Pod whenever it gets deleted. In order to react to deletion it uses Finalizers to prevent the Pod from being deleted right away. For the prototype, the controller was implemented following the operator pattern, which will be explained briefly before describing how the migration operator works.

5.4.1 The Migration Finalizer

As explained in section 2.1.2, any object within the Kubernetes API can have a set of Finalizers as part of its metadata to delay their deletion. Instead of deleting them, the `deletionTimestamp` that is also part of the metadata will be set. The API server will then wait until all Finalizers are removed before actually deleting the object from the API. This behavior provides an asynchronous and decoupled way for controllers (or any other actor) to react to the deletion of API objects. To implement a Finalizer, a controller can add an entry to the list of Finalizers. Once the controller notices that the `deletionTimestamp` is set, it can perform the required actions and remove the Finalizer once it is done. [Kubj; Kubg]

In the API, Finalizers are implemented as strings consisting of a namespace and a name, separated by a slash. To migrate Pods on deletion, `podmig.schrej.net/Migrate`[1] will be used as the finalizer. As Finalizers are currently not respected by kubelet, modifications were necessary to postpone deletion, which will be described in section 5.2.2.

As explained in section 2.1.5, Services are used to hide the IP addresses of the Pods managed by a MigratingPod. A endpoint controller creates the Endpoints of a Service based on the Service's label selector. It only creates Endpoints for Pods whose Ready condition is fulfilled and which do not have a deletion timestamp set. When triggering the migration of a Pod by deletion, the endpoint controller usually deletes its Endpoint and cause it to become unreachable. The downtime of the migration would start immediately, voiding the optimizations discussed in section 4.5.2. The endpoint controller was therefore modified to keep the Endpoint if the Pod has a migration finalizer. Since the Ready condition of the source Pod is disabled once the container migration starts, the Endpoint still gets deleted once the source Pod is frozen und therefore unable to handle requests.

5.4.2 The Operator and Controller Patterns

The operator pattern is a way of extending Kubernetes. It combines *CustomResourceDefinitions*, which allow to define custom resources, with the controller pattern that is used throughout Kubernetes in various forms. The most common use of the operator pattern is managing applications. With an operator, applications can be deployed by creating objects in the API that describe the application configuration. The controller then deploys that application by creating other resources such as Deployments or StatefulSets. An operator can also be used to automate maintenance tasks like creating backups. The pattern is not limited to deploying applications though, it can also be used to manage external resources. [Kubq]

The controller pattern is inspired by control loops in robotics and automation. Such control loops observe certain parameters and attempt to bring them closer towards desired values. Controllers in Kubernetes do the same for API objects. The objects define the desired state in their spec field, and the controller tries to adapt the actual state of (parts of) the cluster to match it. Common examples for controllers within Kubernetes are the various workloads (Deployments, StatefulSets, Jobs, etc.), but even kubelet can be considered to be a controller that manages external resources in form of containers. [Kubc]

[1] schrej.net is the personal domain of the author and serves as a unique namespace.

5.4.3 The Migration Operator

The migration operator was implemented using *kubebuilder*[1], a *Software Development Kit* (SDK) for building operators. The SDK takes care of monitoring the custom objects for changes and calls the responsible controller to handle them. It also contains tools to automatically generate the CustomResourceDefinitions from type definitions. As everything else is handled by the SDK, only the type definitions and the implementation of the controller will be explained in detail.

The MigratingPod Custom Resources

The type definitions (see figure 5.3) include the `MigratingPod` and the `MigratingPodList`. The latter is required as any answer of the kube-apiserver has to be an object. It consists of the regular type metadata, special metadata for lists, and a list of `MigratingPods`. The `MigratingPod` itself also has type metadata and, since it is a single object, object metadata. It further has a `Spec` field, which is used to specify the desired state, and a `Status` field to indicate the actual state of the object. The spec only has a single `Template` field that is a `PodTemplateSpec`. The same is also used by other workload resources to describe the Pods that should be created by the controller. The status contains a `State` (a rough indication of the overall status of the MigratingPod) and `ActivePod`, which contains the name of the Pod that currently executes the containers.

The Migration Controller

The part of the controller that has to be implemented when using kubebuilder is called a `Reconciler` and consists of two methods. The important one is `Reconcile(...)`, which is part of the main controller loop. The second one, `SetupWithManager(...)`, takes care of creating the controller and registering it in the controller manager, indicates which resources should be watched, and contains some additional configuration for the built-in functionality provided by the SDK. For the MigratingPod controller, the `MigratingPod` type is set as the resource to monitor. It is also necessary to indicate which kinds of resources get created by the controller so they can be watched as well. As migration controller only creates Pods, only Pods are watched. In order to make it easier to query the Pods that belong to a specific controller, the Pods are indexed by the name of MigratingPod that they belong to.

The reconcile function is called whenever something that the controller is responsible for changes. Ideally, control loops, which the reconcile function is a part of, are idempotent

[1] https://kubebuilder.io

```go
// MigratingPodSpec defines the desired state of MigratingPod
type MigratingPodSpec struct {
    // Template describes the pods that will be created.
    // +kubebuilder:validation:Required
    Template corev1.PodTemplateSpec json:"template"
}

// MigratingPodStatus defines the observed state of MigratingPod
type MigratingPodStatus struct {
    // State indicates the state of the MigratingPod
    State string json:"state"

    // ActivePod is the name
    ActivePod string json:"activePod"
}

// MigratingPod is the Schema for the migratingpods API
// +kubebuilder:object:root=true
type MigratingPod struct {
    metav1.TypeMeta   json:",inline"
    metav1.ObjectMeta json:"metadata,omitempty"

    Spec   MigratingPodSpec   json:"spec,omitempty"
    Status MigratingPodStatus json:"status,omitempty"
}

// MigratingPodList contains a list of MigratingPod
// +kubebuilder:object:root=true
type MigratingPodList struct {
    metav1.TypeMeta json:",inline"
    metav1.ListMeta json:"metadata,omitempty"
    Items           []MigratingPod json:"items"
}
```

Figure 5.3: Type definitions for the MigratingPod custom resource.

and only rely on state that they can observe. This allows them to function even if the state in the API is inaccurate. It also makes the controller stateless, which helps with scaling and high-availability.

The reconcile function of the MigratingPod controller (see figure 5.5 for a visualization) starts by fetching the MigratingPod and the Pods owned by it. If the MigratingPod is deleted and there still exist Pods with a owner reference to that Migrating Pod, their migration finalizers are removed so they can get garbage collected. Next, the source and target Pods are identified based on their creation timestamp; the newer one is selected as the source Pod. If there is only one Pod, it is considered the source Pod. If there are more than two Pods, the function returns an error as that should not happen.

Most controllers within Kubernetes update the status first, before modifying any resources. Following this approach, the migration controller does so as well. As described before, the MigratingPod's status contains a `State` and an `ActivePod`. The State is comparable to a Pod's Phase, but it contains cycles (see figure 5.4) and the Phase of Pods does not. A MigratingPod starts in the *Creating* state. Once the Pod it controls reaches the Running phase, the MigratingPod also becomes *Running*. If the Pod gets deleted it transitions to *MigrationPending* until the target Pod is created and it shifts to *Migrating*. Once the target Pod is running, it goes back to *Running*. It can also reach the *Invalid* state, which is currently only reached when there are more than two Pods owned by the MigratingPod. The source Pod is considered to be the `ActivePod` unless the target Pod is running.

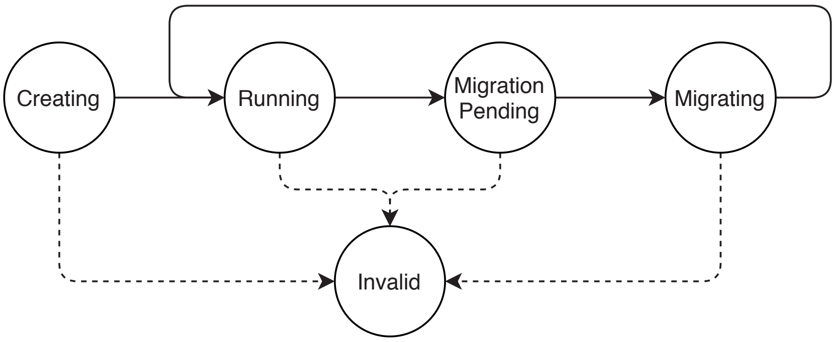


Figure 5.4: Visualization of MigratingPod State transitions

After the status has been updated, the actual behavior of the MigratingPod is applied. As controllers are invoked very frequently, the behavior is not implemented as a sequence. Instead, the controller only executes one step based on the current state of the MigratingPod. If the step changes the Pods belonging to the MigratingPod, this will cause the reconcile function to be invoked again, and the next step can be executed.

If no source Pod exists and there are no other Pods owned by the controller, an initial Pod gets created. Its name consists of the name of the MigratingPod followed by a dash and a zero. A reference to the MigratingPod is added to the list of owners in the metadata. One of the owners can also be marked as the controller of the Pod, which is done as well.

If the source Pod exists but has a deletion timestamp set and is not considered active anymore (based on the previously determined status), the migration finalizer of the Pod gets removed so the deletion can complete. If it has a deletion timestamp set and no target Pod exists, a target Pod gets created, which triggers the migration. The name of the target Pod is derived from the source Pod by increasing the number at the end by one. This prevents creating multiple target Pods by leveraging the optimistic locking capabilities of the API: if a Pod with the same name already exists, the API server will simply reject it. There is no need to explicitly delete the source Pod, as its deletion started the whole process in the first place.

The status is not updated a second time after Pods were modified. The change will cause the reconcile function to be run again, which will then update the status accordingly.

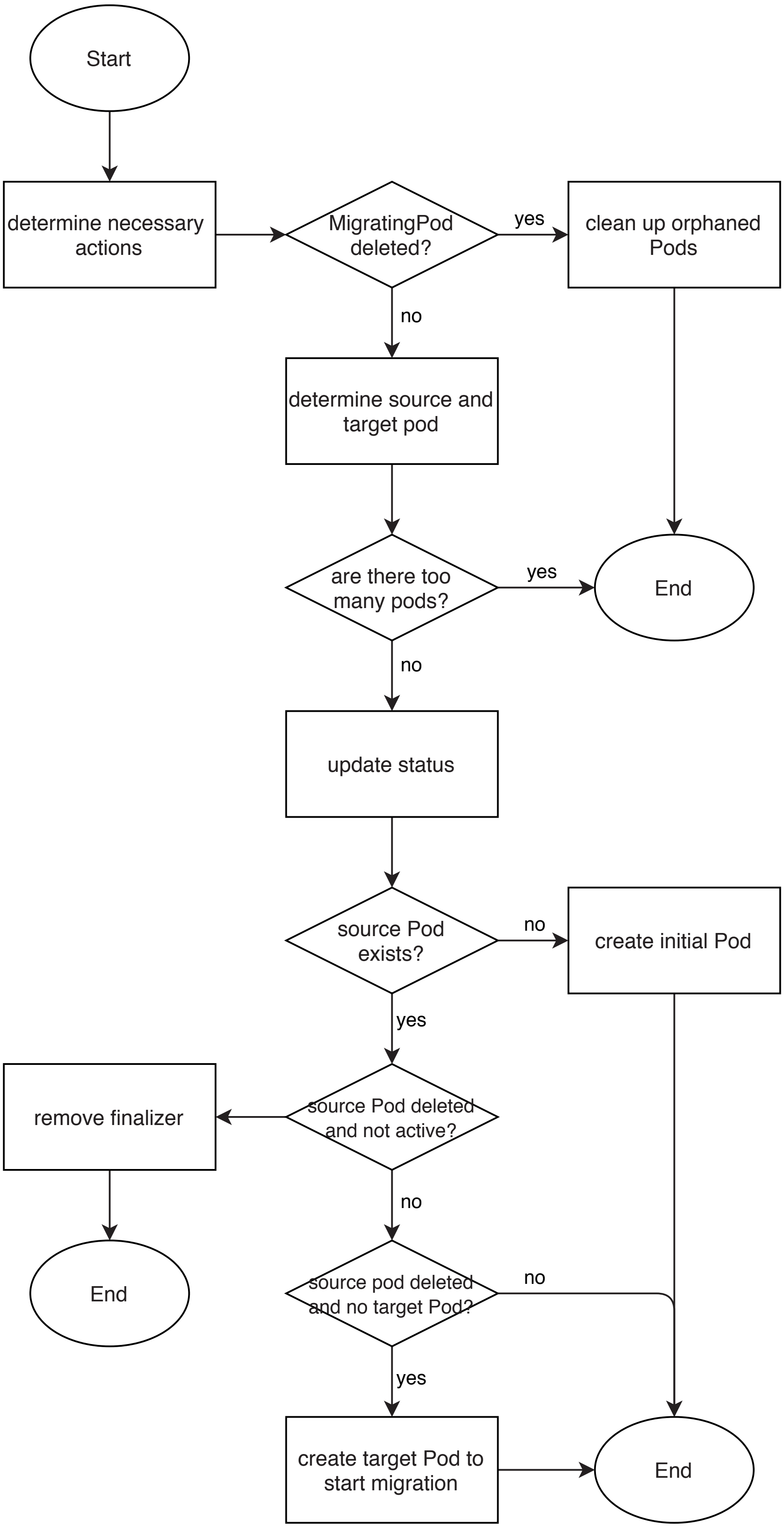


Figure 5.5: Flowchart of the migration controller's reconcile workflow

5.5 Integrating Migration into Kubelet

The largest part of the implementation is the integration into kubelet. It needs to be able to handle several things for migration to work. The first step is recognizing that a Pod is supposed to clone another Pod when it is scheduled to the node. It also needs to be able to create and restore container checkpoints. Finally, there must to be a way for the kubelet instance running on the target node to request the checkpoints from the source node's kubelet instance.

Kubelet's implementation is quite complex, especially since it contains a large amount of interfaces to facilitate testing. To make it easier to follow the changes made for this prototypical implementation, some of those interfaces are omitted. The same applies to parameters of functions and methods mentioned in this section.

Before explaining what changes were necessary to support Pod migration, the Pod synchronization loop is described. It is the main component of Kubelet that interacts with the container runtime and the only one that modifies the containers running on the node, and therefore has to take care of creating and restoring checkpoints as well. Next, the procedure on the target node is described. While this does resemble the logical order of C/R, the procedure actually starts with the target Pod getting scheduled to the target node. It is followed by the creation of the checkpoint, which is triggered by the target node due to the optimization laid out in section 4.5.2. Finally, the update of the source Pod's status is explained.

5.5.1 The Pod Synchronization Loop

Kubelet also follows the controller pattern that was described in section 5.4.2 to adapt the state of the node towards the defined state in the API. In contrast to controllers, it does not observe only the API for changes, but also the state of the node it is running on.

There are multiple different reasons why kubelet needs to (re-)synchronize a Pod with the desired state in the API: new Pods get scheduled to the node, they get updated or deleted, or the state of the node changes (e.g. a container crashes). It is possible that some of these events occur at the same time. A likely example would be a container crashing while the Pod is modified in the API. In order to process the tasks sequentially, each Pod has its own task queue and a worker. The worker is running a loop that pulls the tasks from the queue and calls the main `syncPod()` method of kubelet.

The `syncPod()` method can be compared to the `Reconcile()` method of the migration operator: it contains the entire synchronization logic and gets executed whenever the state of a Pod changes. It does not directly start with a status update, but checks if the synchronizing

Pod should be killed first, as killing the Pod makes the remaining steps obsolete and allows to skip them. The Pod's status is updated afterwards. As with the migration controller, this is the only time the status is updated. Any modifications performed in the remainder of the method will cause the loop to execute again, which also leads to a status update. By updating the status first, it is also ensured that external changes on the node are reflected in the status before they are corrected.

If a Pod is not supposed to be running, e.g. when it has a deletion timestamp set or the status update determined it is in a failure state, it will be killed after the status update. A few additional checks and actions like mounting volumes and fetching secrets are then performed, before the `SyncPod()` method of the container runtime manager[1] gets called. The container runtime manager is an abstraction of all interactions with container runtimes, and uses the `RemoteRuntimeService` and `RemoteImageService` (see section 5.2.2) to talk to the container runtime in use.

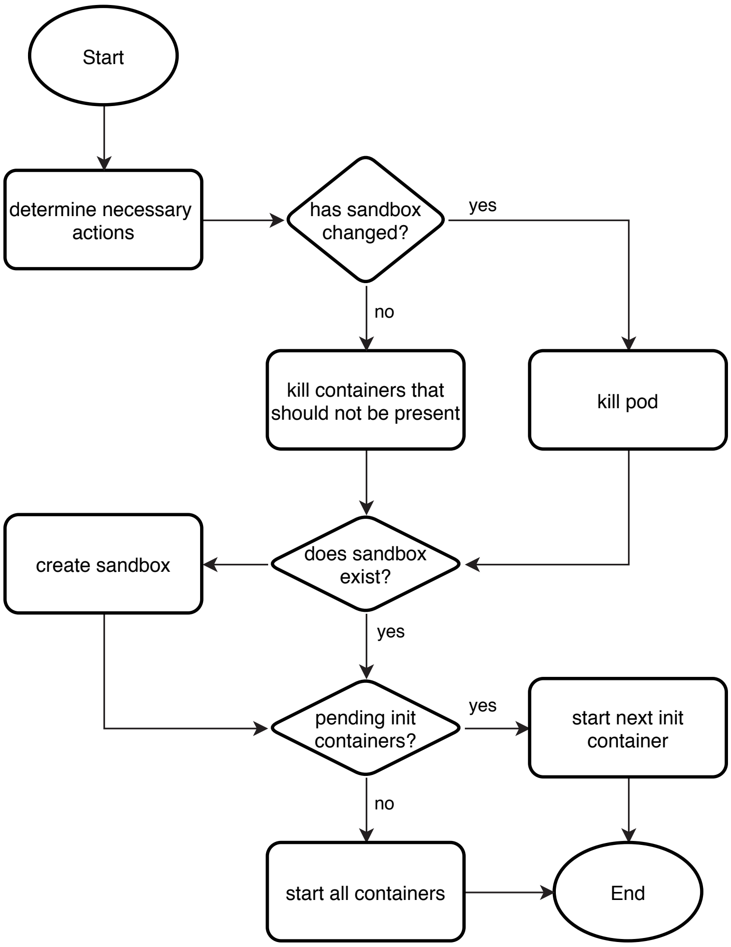


Figure 5.6: Flowchart of the `SyncPod()` method of the runtime manager.

[1] Actually called `kubeGenericRuntimeManager` and located in the `pkg/kubelet/kuberuntime` package.

The `SyncPod()` method (visualized in figure 5.6) first determines what actions are necessary. Some changes to a Pod can require the Pod sandbox (the isolated environment shared by all containers of a Pods) to be recreated. If that is the case, the Pod gets killed first, which removes all containers and the existing sandbox. If not, any containers that should not be present in the pod will get terminated. Next, the sandbox will be created if it does not exist already, which is the case for new Pods or when it was removed in the previous step. If the Pod contains init containers and some of them have not been executed yet, the next init container is started and the synchronization process terminates. This also relies on the fact that the synchronization loop will be triggered when a change to the Pod's containers is noticed. Once no init containers are left, all containers that are part of the Pod are started. If containers were running previously but are terminated now (usually meaning the process crashed) they are restarted based on the restart policy defined in the Pod specification.

5.5.2 Restoring on the Target Node

The migration process within kubelet begins with the target Pod getting scheduled to the target node. Once detected, it enters the Pod synchronization loop as usual and proceeds through it until just before the Pod's containers get started.

Previously a single method was called for each container of the Pod that takes care of both creating and starting the container. As explained in section 4.4, restoring a container requires to create it first. In order to share the creation logic between starting and restoring, the logic to create the container was extracted from the container runtime manager's `startContainer(...)` method into a new `createContainer()` method. The `startContainer(...)` method now calls `createContainer()` at the beginning.

If no containers exist yet, which indicates that the Pod is newly created, and the `ClonePod` field (see section 5.3.2) contains a reference to another Pod, the containers will be migrated from the referenced Pod instead of being started regularly.

As discussed in section 4.5.2, time can be saved by creating the containers first. Therefore, the previously extracted `createContainer()` method is called for all containers of the Pod. After the creation is complete, the containers on the source node need to be checkpointed. To do so, the source node is contacted using a new HTTP API endpoint served by kubelet, which allows to request the migration of a Pod. The kubelet instance on the source node creates the checkpoints (see section 5.5.3) and responds with the paths of the created checkpoints. A method to issue this request was implemented as part of a migration manager, which is explained in further detail in the following section as well.

Once the creation of the checkpoints is complete, a new `restoreContainer()` method is called for each container and its corresponding checkpoint. The restore method is identical to the `startContainer()` method (apart from the `createContainer()` call in the beginning), but calls the new `RestoreContainer()` method of the `RemoteRuntimeService` instead of `StartContainer()`. After the restoration is completed for all containers, the checkpoints are deleted.

5.5.3 Checkpointing on the Source Node

On the source node the migration process starts once it is requested by the target node. A new endpoint was added to the existing HTTP API of kubelet for that purpose. The new endpoint expects the UID of the Pod and the names of the containers that should be prepared for migration as an argument.

In addition, a very simple migration manager, contained in new `migration` package, was implemented to track ongoing migrations and facilitate asynchronous communication. It consists of a list of ongoing migrations and holds references to a Kubernetes API client, the pod manager (which contains a list of all Pods that are scheduled to the node) and a `prepareMigrationFunc` function. The latter is implemented in the `kubelet` package to avoid import cycles[1], which are forbidden in Go.

Figure 5.7 shows the process of preparing the migration on the source node. The handler of the API endpoint (implemented as a method of the migration manager) starts by fetching the Pod that the migration is requested for from the pod manager. It then creates a migration object to track the migration and calls the `prepareMigrationFunc`. Since any modification of a Pod needs to be performed by the previously described Pod synchronization loop, the function pushes a new synchronization request into the Pod's queue. Those requests contain a `SyncPodType`, indicating the reason for the sync. A new `SyncPodMigrate` type was added and is used to indicate that the migration of the Pod should be prepared.

The synchronization loop was extended to handle such a `SyncPodMigrate` update type, which it does fairly early as the rest of the logic can be skipped if a migration was prepared. It calls a new `PrepareMigratePod()` method that was added to the runtime manager to handle the preparation. Since full checkpointing is the chosen strategy, a checkpoint gets created for

[1] This is a common pattern in kubelet: As the `kubelet` package is the main package of kubelet, it has to import the migration package. But in order to trigger the migration, the `migration` package has to access other parts of the `kubelet` package, which would require it to import the `kubelet` package. As that causes an import cycle, the function is implemented in the `kubelet` package instead and passed to the migration manager when it is instantiated, which can call it as needed.

all containers for which a checkpoint was requested using the new `CheckpointContainer()` method of the `RuntimeService`. This also causes the containers to terminate and therefore marks the beginning of the downtime.

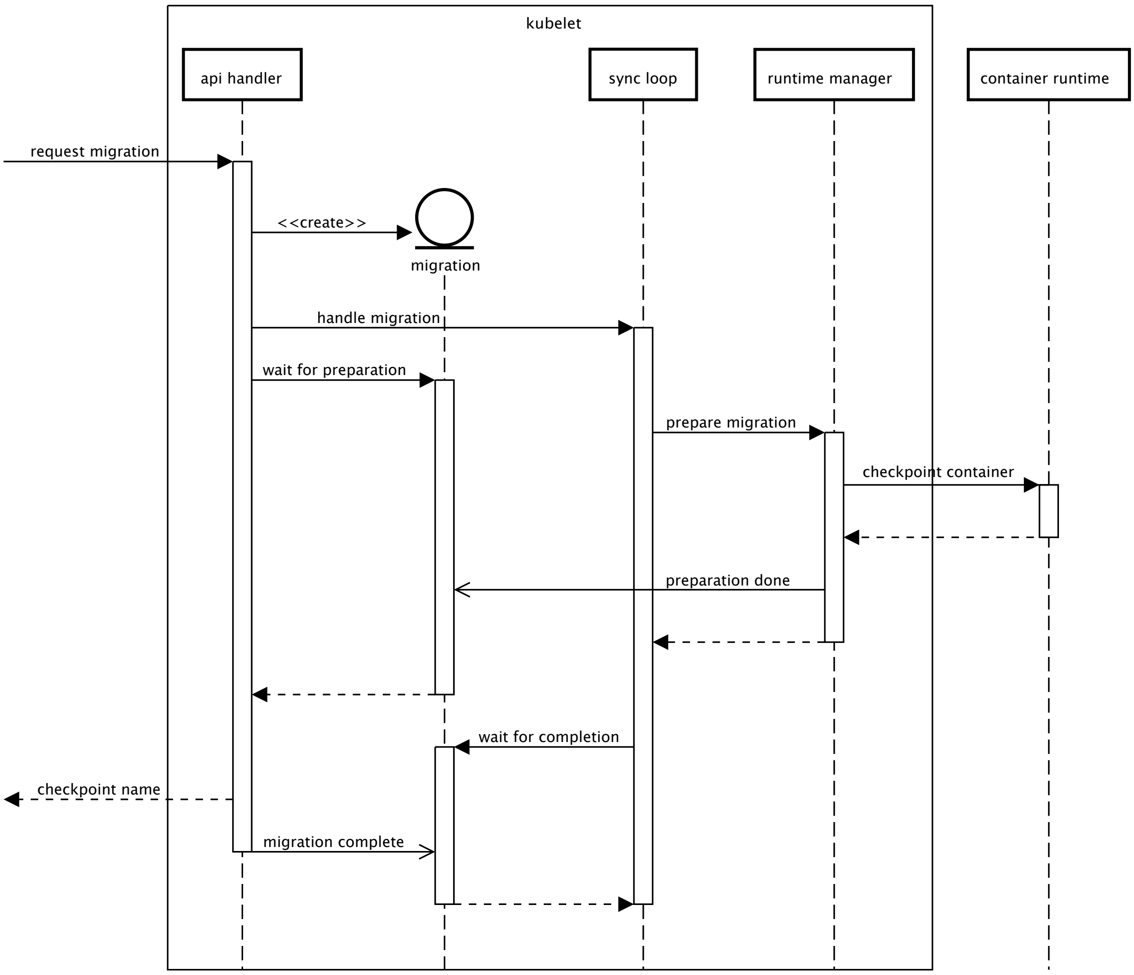


Figure 5.7: Sequence diagram of the migration preparation process on the source node.

The API endpoint is supposed to respond once the preparation of the migration is complete. Since the `prepareMigrationFunc` only adds a synchronization request to the queue, it returns immediately and does not wait until the preparation is complete. To be able to wait for the preparation to complete, the migration object contains a *channel*. Channels are a very simple data structure in Go that allows to exchange messages between asynchronous functions. The API handler waits until it receives a message on that channel, which message is sent by the runtime manager after the checkpoints are created. The handler then responds to the API request, which causes the migration process to continue on the target node as explained in the previous section.

In order to prevent anything else from interacting with the Pod, the synchronization loop is blocked until the preparation is completed. To do so, migration object contains a second channel, on which the synchronization loop waits for a message after the runtime manager is finished with creating the containers. A message is sent on the channel by the API handler after it responded to the request, causing the synchronization loop to unblock. The synchronization loop then performs a final step: it updates the status of the source Pod.

5.5.4 Updating the status of the source Pod

There are three reasons why updating the status of the source Pod is important. First, the Pod status is supposed to reflect the actual state of a Pod as accurately as possible, and that also applies to Pods whose containers were migrated away.

Secondly, Kubelet also incorporates the Pod status into the decision whether a Pod should be running. If it determines that the Pod is supposed to be running, it will try to restart containers, which is something that must not happen during a migration, as it can lead to the same container running on two nodes at the same time, leading to unpredictable effects.

The third reason is related to Services. As mentioned in section 5.4.1, the Endpoints of a Service are affected by the Ready condition that is part of the Pod status. In order to switch from the source to the target Pod, the source Pod's Ready condition has to be disabled and the one of the target Pod needs to be enabled.

The target Pod's Ready condition will be updated by the regular process that also determines readiness when a Pod is started without migration. In order to disable the Ready condition of the source Pod, the `TerminatePod()` method of kubelet's status manager is called. It sets the status of all the Pod's containers to *Terminated*, which disables the Ready condition of the Pod. In addition it also causes the Pod's phase to transition to *Terminated*, which leads to kubernetes trying to remove the Pod's containers, instead of restarting them.

5.5.5 Respecting the Pod Finalizer

Usually, finalizers are respected solely by the API server. Adding a finalizer to a Pod does not prevent kubelet from deleting it immediately. In order to migrate a Pod on deletion, a checkpoint needs to be created before the Pod is fully deleted. Kubelet therefore needed to be modified to delay the deletion for a Pod if the migration finalizer described in section 5.4.1 is present.

The simplest way to delay the deletion is to not consider the Pod deleted as long as the specific finalizer is set. There are several locations in kubelet that decide whether a Pod is considered deleted. Two of them lead to the immediate deletion of the Pod:

- The `checkAndUpdatePod(...)` function in the `pkg/kubelet/config` package decides whether a Pod should be deleted after it was updated in the API, which causes its Pod worker to be stopped and the Pod to be terminated.
- In the pod synchronization loop described in figure 5.6 Pods are killed if they have a deletion timestamp set.

All locations that rely solely on the deletion timestamp and consider a Pod to be deleted when the timestamp is set were extended to also check whether the finalizers list of the Pod includes the `podmig.schrej.net/migrate` finalizer and do not delete the pod if it does. Since kubelet continues to monitor its assigned Pods, it will notice when the finalizer is removed and proceed to delete the Pod as it would normally do.

CHAPTER 6

Evaluation

In this chapter the prototype described in the previous chapter is evaluated. The evaluation is twofold. It begins with an evaluation of the functionality of the prototype with regards to the goals laid out in section 4.1. The second part examines the downtime of Pods during a migration and compares it to deleting and creating a new Pod, which is the current practice in Kubernetes.

6.1 The Test Cluster

The Kubernetes cluster used for evaluation consists of a single master and two worker nodes. The two virtual servers used for the nodes are *c4.16xlarge.2 Elastic Cloud Servers*[2] running on the *Open Telekom Cloud*[3]. They have 64 virtual[4] *Intel Xeon Gold 6266C* CPU cores clocked at 3.00 GHz and 128 GB of memory. Their network connection has a bandwidth of up to 40 Gbit/s. Both worker nodes have a shared folder mounted using *Network File System* (NFS) version 4 for transferring the snapshot, which is provided by a third server that matches the specifications of the two nodes. The shared folder is provided from an in-memory disk to ensure storage bandwidth is not a limiting factor. The master node is using a smaller *s2.medium.2*[5] server, with one CPU and two GB of memory.

A minimal setup of Kubernetes is deployed[6]. The master node is running the kube-apiserver, etcd and the kube-controller-manager. Both nodes are running kubelet, kube-

[2] https://docs.otc.t-systems.com/en-us/usermanual/ecs/en-us_topic_0091224748.html

[3] https://open-telekom-cloud.com

[4] Servers in the c4 category have dedicated CPU cores that are not over-provisioned.

[5] https://docs.otc.t-systems.com/en-us/usermanual/ecs/en-us_topic_0035470101.html

[6] Following, for the most part, the *Kubernetes The Hard Way* Guide by Kelsey Hightower, see https://github.com/kelseyhightower/kubernetes-the-hard-way.

proxy and containerd. The authorization capabilities of kubelet were disabled by setting `authorization.mode` to `AlwaysAllow` in its configuration. The networking plugin in use is Calico[1] with default configuration. All servers are running *Ubuntu Linux 20.04*.

6.2 Functionality

The primary goal defined in section 4.1 is the ability to migrate Pods. Technically the prototype is not capable of that. As explained in section 4.2.2, Pods are not migrated, but cloned, as cloning fits the API concepts of Kubernetes much better. The intention of Pod migration is to keep the *state* of the Pod, and not necessarily the *object*, though. By deleting the old Pod after the clone is created, the behavior of migration can be implemented successfully. The primary goal is therefore considered to be fulfilled by the prototype.

The next sections will cover the secondary goals of section 4.1, with the exception of short downtimes, which will be benchmarked in section 6.3.

6.2.1 API Integration

The first and most important secondary goal is about API integration. Migration should be in line with existing concepts of the API and be compatible with existing mechanisms of Kubernetes..

The prototype is able to migrate Pods when they get deleted. As explained in section 4.2.5, this way the migration integrates with the existing eviction mechanism without any further modifications. The eviction mechanism is used when resource limits of a node are exceeded and a Pod needs to be moved, and when a node is drained, meaning that all Pods are removed and it is excluded from future scheduling. The latter is commonly used to prepare a node for maintenance, which is one of the use cases for Pod migration mentioned in the introduction. Testing this behavior by executing `kubectl drain <node name>` shows that the mechanism is working as intended: the Pod gets migrated to another node.

The prototype also does not add any new API behavior. When observing the the MigratingPod through the API, it appears like any other workload controller, with the exception that a Pod belonging to a MigratingPod require more time to be deleted. Therefore, any clients that observe Pods through the API should not require changes. The only exception

[1] https://www.projectcalico.org/

is that tracking exact instances of an application becomes more difficult as the Pod that contains the instance can change. This can easily be circumvented by adding an annotation or a label that identifies a specific instance.

6.2.2 Usability

The usability goal focused primarily on the invocation mechanism. The API integration described in the previous section also benefits the usability. If migration seamlessly integrates with existing concepts, using it becomes much easier.

Being able to set off a migration by deleting an object itself has both positive and negative aspects. It allows to easily trigger migrations without having to learn and implement new API endpoints. Any existing API client that can delete a Pod can also be used to migrate one. On the other hand, there is currently no safer way of migrating a Pod. As laid out in section 4.7, triggering the migration by deletion makes it a lot harder to handle failure during the migration. When using the prototype, the Pods just remain in an error state indefinitely and have to be cleaned up manually.

The likelihood of failure itself is another aspect of usability. While the approach to use a MigratingPod can prevent errors caused by the desired state, there are still other reasons for failure that can be caused by the user. As the writable file-system layers of containers are not migrated, restoring a container will fail if the process within it tries to access files that do not exist. As only files within PersistentVolumes that are mounted from remote storage will be migrated, folders that should be part of a volume need to be carefully selected beforehand. Errors regarding missing files will only appear after restoring the Pod on the target node. This makes them hard to predict and can lead to a bad user experience.

6.2.3 Networking

The last secondary goal is to achieve a good integration with Kubernetes' networking. As explained in section 4.6, Service can be used to mask the changing IP address. The benchmark described in section 6.3.1 shows that this approach is working well. But, as mentioned, open TCP connections will be terminated during the migration when the Pod's IP address is not available on the target host. Therefore, the prototype fails to deliver full networking transparency.

Depending on the higher-level protocol and the implementation in use, this lack of transparency can have different effects. Regular HTTP connections (e.g. when running an API service or a simple web server) will fail, but can easily be retried. As HTTP connections are relatively short-lived, the number of affected connections is also lower compared to other applications. If connections are kept open for long periods (e.g. when hosting database services or game servers), all connections that are active at the time of migration will be terminated and have to be re-established after the migration is complete. As the connections are not closed cleanly, many applications might wait for a specified time-out period before attempting to reconnect, which could lead to additional perceived downtime for clients.

6.3 Downtime

Migrating an application between two servers always leads to some amount of downtime for that application, during which it is not accessible. From a networking perspective this usually results in delayed responses or completely unanswered requests. One of the goals of Pod migration is to reduce such downtime for stateful applications that take a long time to start, which leads to long downtimes when moving them between nodes by deleting the current Pod and creating a new one.

The C/R process used to migrate the Pods also takes time. Using Pod migration only provides a benefit when the process is faster than simply restarting the application. In order to get a baseline for such a decision, both approaches are compared in this section.

6.3.1 Method

The time it takes to migrate a Pod depends on the amount of memory the containers within it are using. In order to compare different memory sizes, a small Go program[1] was written that first allocates a specified amount of memory, and then starts a web server that provides a single endpoint returning the amount of times it has been called.

A Pod that includes the test program was then deployed on the test cluster with different memory ballast sizes using two different workload controllers: a Deployment that represents the default behavior of Kubernetes, and a MigratingPod. The whole process is automated with a second program[2] that takes care of creating the Deployments and MigratingPods, deleting a Pod to move it somewhere else, tracks the downtime and cleans up after each test. It executes each test configuration multiple times.

[1] The test program is located at `code/podmigration-operator/evaluation/testapp` on the attached storage medium.

[2] The evaluation program is located at `code/podmigration-operator/evaluation/eval.go` on the attached storage medium.

6.3.2 Results

The tests were executed ten times each with the allocated memory ranging from zero to 100 gigabytes in increasing intervals. The results (see table 6.1 and figure 6.1) show that even though migration is almost as fast as moving Pods by deleting and recreating them for up to 200 MB of memory, migration takes increasingly longer than recreating the Pod when more memory is allocated.

memory in GB	0	0,1	0,2	0,5	1	2,5	5	10	15	20	25	50	100
without migration	1,7	1,6	1,7	1,6	1,7	1,7	1,5	1,7	1,7	1,7	1,9	2,6	4,5
with migration	2,1	2,0	2,0	4,3	4,0	7,8	14,0	26,7	38,4	51,7	63,5	128,5	255,0

Table 6.1: Mean downtime in seconds of Pod moves between nodes with 10 executions each.

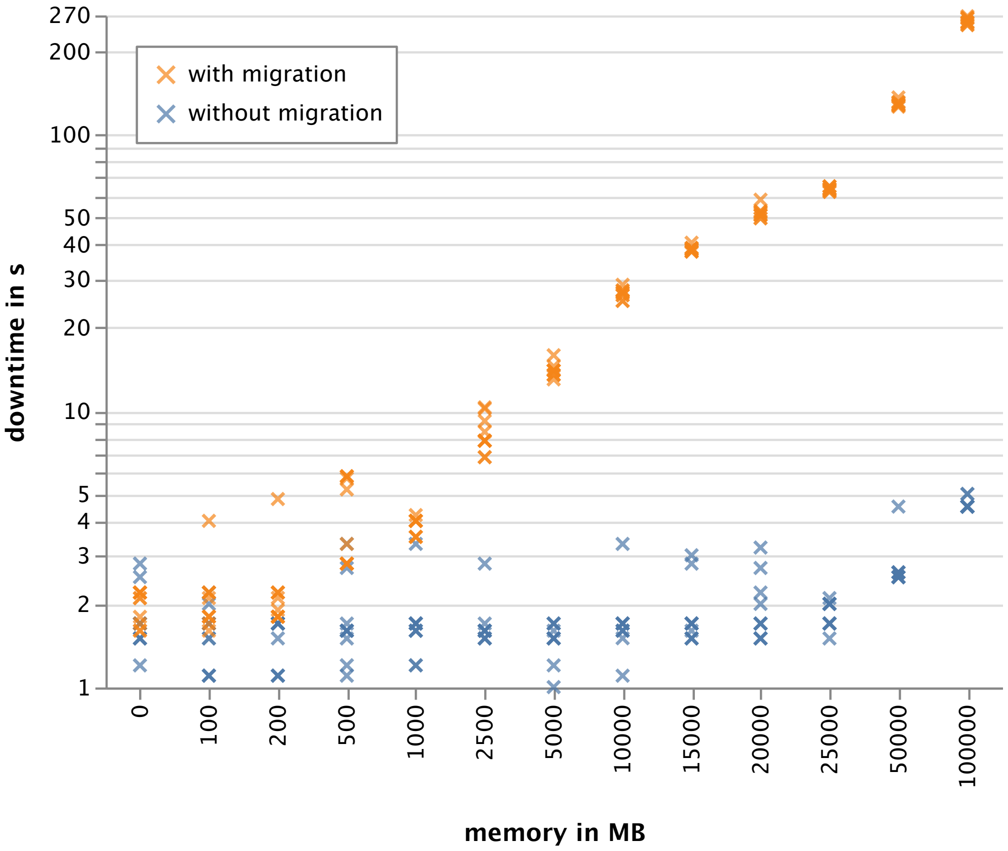


Figure 6.1: Downtimes for different amounts of allocated memory with and without migration for 10 executions each.

The occasionally high deviation of downtime when moving pods without migration that can be seen in figure 6.1 is most likely caused by the loose coupling of the involved components: if the Deployment controller, the Endpoint controller or kubelet do not notice the state change right away, the creation of a replacement Pod is delayed, which results in a longer downtime. The same applies to moving Pods with migration, which is additionally affected by variances in networking speed. As the benchmark was performed in a cloud environment, the network is a shared medium whose performance can vary.

The mean downtime caused by migration correlates with the amount of allocated memory. This correlation is caused by the bandwidth limits of the remote storage that is used to transfer the checkpoint. The NFS volume provides write speeds of approximately 0.5 GB/s and read speeds of about 1.1 GB/s[1]. Since each checkpoint is written and read exactly once, the average value of both (0.8 GB/s) can be used to predict the transfer duration. Without any artificial memory allocation, the downtime still takes about two seconds. This is therefore assumed to be the overhead caused by the migration process itself. downtime can be predicted with a simple calculation:

$$downtime = (memory\_size/bandwidth) * 2 + overhead$$

For example, with 10 GB of allocated memory, 0.8 GB/s bandwidth and 2 s overhead the equation predicts 27 s of downtime, which is reasonably close to the 26.7 s measured during the benchmark.

Whether migration with such long downtimes is useful depends on the use case. The test application allocates memory as fast as possible, leading to only a small increase in startup time, even when allocating large amounts of memory. Filling the same amount of memory with useful data, e.g. by reading from disk or querying an API, would take significantly longer. If the goal is to move Pods with slow starting applications between nodes, using migration makes sense only when the startup time of the migration is longer than the downtime during the migration.

To reduce the downtime, the overhead needs to be reduced or the bandwidth has to be increased. Closer inspection of the migration process in containerd shows that about 1.2 s elapse between the start of the checkpoint creation and the end of the restoration. The remaining 0.8 s appear to be caused by Kubernetes itself, most likely by the Endpoint controller and kube-proxy. While there is room for improvement with regards to the bandwidth, it is ultimately limited by the physical networking bandwidth between the nodes.

[1] This was tested using `dd` to read and write a 50GB file to the mounted folders on the nodes. The file system cache was cleared before reading the file.
Writing: `dd if=/dev/zero of=/var/lib/kubelet/migration/bench bs=1M count=50000`
Reading: `dd if=/var/lib/kubelet/migration/bench of=/dev/null bs=1M count=50000`

CHAPTER 7

Conclusion

7.1 Summary

Since moving Pods between nodes would require severe changes to their lifecycle, it is proposed that they are cloned instead. By deleting the old Pod after the cloning is complete, the end result is identical to migration. From three alternatives, adding a MigratingPod resource and a migration controller to declare and manage migrations was identified as the most suitable solution. The CRI, which is used to interact with container runtimes, needs to be extended with two additional methods to allow Kubernetes to use the checkpoint and restore operations present in common container runtimes. The downtime caused by the migration can be reduced by creating the containers of the new Pod before beginning to create checkpoints of the old Pod's containers, and services allow to mask the changing IP address when a Pod is migrated.

A prototypical implementation of Pod migration was created using containerd as the container runtime. It demonstrates that the proposed design is viable and able to integrate with existing components. A benchmark was performed, which shows that the chosen strategy to create full checkpoints is sufficient to migrate small workloads with reasonable downtimes. For workloads that consume more memory, the approach is only applicable if the applications take a long time to start.

7.2 Future Work

In order to integrate Pod migration into the Kubernetes project, additional work is required. Since Kubernetes is a large project with many stakeholders, significant changes have to be submitted as an *enhancement proposal*. The proposal is then discussed by the community and, if it meets certain requirements, eventually accepted. The design proposed in this thesis can serve as a foundation for such a proposal, and the prototype can be used to evaluate the benefits of Pod migration for Kubernetes.

The prototype also does not fulfill all criteria of live migration. The following sections will discuss the required improvements to full fill those criteria, as well as several other aspects that can be improved.

7.2.1 Networking

Major improvements can be made with regards to networking. In order to support live migration, the migration needs to be completely transparent from a networking perspective. As explained in section 4.6, this can only be achieved by migrating the IP address of the Pod to the new node. Whether that is supported depends on the networking plugin in use.

Calico, which was also used during development of the prototype, like most network plugins, uses one subnet per node by default. Doing so keeps routing tables small, which improves networking performance. Calico only uses those subnets as pools when automatically assigning addresses, though. By explicitly assigning an address to a Pod, any address within the cluster's network can be used on any node. However, this alone does not allow to migrate the address yet, as it is still bound to the old Pod during the migration and therefore cannot be assigned to the new one.

In order to support live migration, a standardized way of specifying a Pod's IP address must be provided and it must be possible to transfer an IP address from one Pod to another during a migration. If the target Pod uses the same IP address as the source Pod, TCP connections can then be restored to achieve network transparency.

7.2.2 Faster Migration Strategies

The second necessary improvement to support live migration is reducing the downtime. It was shown in section 6.3 that the physical limitations of the chosen migration strategy prevent it from being fast enough for live migration.

CRIU and runc support two faster migration strategies, pre- and post-copy, as explained in section 2.3.2. Both do not transmit a full checkpoint, but work incrementally or lazily instead. While this does not reduce the time it takes to transfer the checkpoints, it reduces the downtime during the procedure significantly.

To implement those faster strategies, the CRI needs to be extended with additional parameters, which has been considered in the changes proposed by this thesis. Support for those faster strategies also needs to be implemented in a container runtime. Common runtimes, including containerd, currently do not support those advanced strategies.

In order to perform pre- and post-copy efficiently, CRIU requires a *page-server* which has to be running on one of the nodes in order to transmit the incremental checkpoints or fetch memory on demand [CRI19]. It needs to be decided how the page-server is managed, especially whether this is a task for kubelet or the container runtime.

7.2.3 Local Volumes and Container File System

The prototypical implementation does not migrate the containers' file systems, nor local volumes of the Pod (see section 4.3). As stated in section 6.2.2, this can cause the restoration of containers to fail if the process within it tries to access files that are not migrated.

Especially temporary files, caches and logs are often not placed in volumes, as using the container file system is usually faster and easier. By migrating local volumes and the container file system as well, less care would be necessary when deciding which folders to mount from remote storage, which makes migration easier to use and less error-prone.

Depending on the amount of data that changed in the file system and the size of the local mounts, this could have a significant impact on downtime, since all files need to be present on the target system in order to restore the checkpoint. The files should therefore be transferred in an iterative or lazy manner, similar to the checkpoints. Voyager [Nad+17], which was discussed in chapter 3, proposes a lazy migration technique that could be used for this purpose.

7.2.4 PersistentVolumes for Checkpoint Transfer

If the strategy of transferring a full checkpoint is kept, the storage used to transfer the checkpoint could be allocated in a more flexible way. In order to reduce the amount of read and write operations, the prototype stores the checkpoint on a remote volume that is mounted by both nodes involved in the migration. The storage is required to be mounted at a specific location before a migration is performed.

In large clusters with many nodes this could lead to issues with too many nodes mounting and using the same storage. Instead, the storage should be mounted on demand when a migration should be performed. Kubernetes provides storage management for Pods in the form of PersistentVolumes, which could be used to do so. The migration controller could claim a PersistentVolume, which gets mounted by the involved nodes for the duration of the migration and get released again afterwards. While this would be an ideal approach on cloud platforms that allow to dynamically allocate storage, it could lead to issues if storage is provided manually. When no PersistentVolumes are available or can be created on demand, it would no longer be possible to migrate any Pods.

7.2.5 Safer Migration Triggers and Failure Handling

The only way of causing Pods to migrate that is implemented in the prototype is deletion. As explained in section 4.7, this makes it hard to implement failure handling, as the deletion of a Pod cannot be reverted in case the migration fails.

A safer way to trigger migrations should be implemented in addition to facilitate failure mitigation. The Pods owned by a MigratingPod have predictable names which include an incrementing counter. Adding a new field to the MigratingPod resource that specifies the name of a Pod that is supposed to be migrated would be a simple solution. To trigger a migration, the field just has to be set to the name of the active Pod. Similar to the Eviction resource for Pods, a *Migration* resource could be implemented for MigratingPods, which automatically sets the field to the active Pod in the MigratingPod's status.

In order to allow migrating a Pod to a specific set of nodes, the *node selector* of Pods could be used. When the migration controller notices that this field has been changed it can create a target Pod as usual. As the target Pod contain the node selector, the scheduler will assign it to a matching node automatically, and the Pod will get migrated as intended.

Acronyms

C/R	*Checkpoint-Restore* ix, 12–14, 16, 18–20, 27, 29, 31, 37, 38, 47, 57, 76
CNCF	*Cloud Native Computing Foundation* 4
CNI	*Container Network Interface* 8, 34
CRI	*Container Runtime Interface* ix, 10–12, 20, 28–30, 32, 37, 39, 60, 62, 76
CRIU	*Checkpoint/Restore In Userspace* 12–14, 16, 18, 31, 37, 61, 62
IPAM	*IP address management* 34
NFS	*Network File System* 54, 59
OCI	*Open Container Initiative* 10, 11, 66
PID	*Process Identifier* 16
protobuf	*Protocol Buffer* 37–40
RPC	*Remote Procedure Call* ix, 37, 38
SDK	*Software Development Kit* 43
SSH	*Secure Shell* 31

Glossary

containerd containerd is a higher-level container runtime that was extracted from Docker. 10, 11, 16, 28, 31, 36, 37, 39, 55

CronJob A CronJob creates Jobs periodically. 7

DaemonSet A DaemonSet deploys identical Pods on all (or a subset of) Nodes of a cluster. 8

Deployment Deployments help with deploying and upgrading horizontally scalable applications. 7, 42, 57, 58

Endpoint The Endpoints of a Service are either Pods or external resources. 8, 42, 52, 58, 59

Finalizer Finalizers can be added to Kubernetes objects to delay ther deletion until they are removed again. 6, 27, 41, 42

Job A Job creates one or more identical Pods that run a finite task and waits for them to finish. 7, 22, 42, 65

kube-apiserver
The kube-apiserver serves the Kubernetes API, performs validation and persists the data to etcd. vii, 4, 9, 36, 40, 43, 54

kube-proxy
The kube-proxy runs on each node and provides parts of the functionality of Services. 54, 59

kubelet
Kubelet is Kubernetes' node agent. 54, 55, 58

PersistentVolume
PersistentVolumes represent persistant storage volumes that can be remotely mounted by Pods. 7, 29, 32, 36, 56, 62, 63

Pod
A Pod is the smallest unit of computing in Kubernetes, consisting of one or more containers. 2, 3, 6–10, 12, 19–31, 34–36, 39–45, 47–50, 52–63, 65, 66, 76

runc
runc is the de-facto standard low-level container runtime of the OCI. 10, 11, 16, 18, 31, 37, 61

Service
Services bundle a set of Pods for service discovery and load balancing. 8, 34, 42, 52, 56, 65

StatefulSet
StatefulSets help with deployment of stateful applications. They are similar to Deployments but ensure unique names and ordering. 7, 42

Bibliography

[Add+20] Rami Addad et al. “Fast Service Migration in 5G Trends and Scenarios”. In: *IEEE Network* 34.2 (2020), pp. 92–98 (cit. on pp. 2, 18).

[AF89] Yeshayahu Artsy and Raphael Finkel. “Designing a Process Migration Facility: The Charlotte Experience”. In: *Computer* 22.9 (1989), pp. 47–56 (cit. on p. 12).

[Ama] Amazon Web Services, Inc. *Amazon EKS | Managed Kubernetes Service*. Amazon Web Services. URL: `https://aws.amazon.com/eks/` (visited on 7 Dec. 2020) (cit. on p. 1).

[And+73] D. Anderson et al. “Central Processing Unit with Hardware Controlled Checkpoint and Retry Facilities”. U.S. pat. 3736566A. International Business Machines Corp. 29 May 1973 (cit. on p. 12).

[Ary+] Kapil Arya et al. *DMTCP : Distributed MultiThreaded Checkpointing*. URL: `http://dmtcp.sourceforge.net/` (visited on 21 Sept. 2020) (cit. on p. 13).

[Bed18] Joe Beda. *4 Years of K8s*. Kubernetes Blog. 6 June 2018. URL: `https://kubernetes.io/blog/2018/06/06/4-years-of-k8s/` (visited on 18 Dec. 2020) (cit. on p. 1).

[BL98] Amnon Barak and Oren La’adan. “The MOSIX Multicomputer Operating System for High Performance Cluster Computing”. In: *Future Generation Computer Systems* 13.4 (1998), pp. 361–372 (cit. on p. 12).

[Bur18] Catherine Burke. *Improving Azure Virtual Machine resiliency with predictive ML and live migration*. Microsoft Azure Blog. 27 Nov. 2018. URL: `https://azure.microsoft.com/de-de/blog/improving-azure-virtual-machine-resiliency-with-predictive-ml-and-live-migration/` (visited on 12 Nov. 2020) (cit. on p. 1).

[BW01] M. Bozyigit and M. Wasiq. “User-Level Process Checkpoint and Restore for Migration”. In: *ACM SIGOPS Operating Systems Review* 35.2 (2001), pp. 86–96 (cit. on p. 12).

[Can15] Canonical Ltd. *Live Migration in LXD*. Ubuntu Blog. 5 June 2015. URL: `https://ubuntu.com/blog/live-migration-in-lxd` (visited on 12 Oct. 2020) (cit. on p. 16).

[Cla+05] Christopher Clark et al. "Live Migration of Virtual Machines". In: *Proceedings of the 2nd Conference on Symposium on Networked Systems Design &amp; Implementation - Volume 2*. NSDI'05. USA: USENIX Association, 2005, pp. 273–286 (cit. on pp. 1, 14).

[CLL19] MinSu Chae, HwaMin Lee and Kiyeol Lee. "A Performance Comparison of Linux Containers and Virtual Machines Using Docker and KVM". In: *Cluster Computing* 22.1 (2019), pp. 1765–1775 (cit. on p. 1).

[Clo20] Cloudify Platform Ltd. *Migrating Pods Between Nodes In The Same Kubernetes Cluster*. The Cloudify Blog. 5 July 2020. URL: `https://cloudify.co/blog/migrating-pods-containerized-applications-nodes-kubernetes-cluster-using-cloudify/` (visited on 18 Nov. 2020) (cit. on p. 19).

[CRI17] CRIU Authors. *Freezing the Tree - CRIU*. 15 Mar. 2017. URL: `https://www.criu.org/Freezing_the_tree` (visited on 16 Dec. 2020) (cit. on p. 13).

[CRI19] CRIU Authors. *Page Server*. 13 Jan. 2019. URL: `https://criu.org/Page_server` (visited on 11 Dec. 2020) (cit. on p. 62).

[Cro15] Michael Crosby. *Containerd: A Daemon to Control runC*. Docker Blog. 17 Dec. 2015. URL: `https://www.docker.com/blog/containerd-daemon-to-control-runc/` (visited on 15 Sept. 2020) (cit. on pp. 10 sq.).

[Doc18] Docker Authors. *Docker Checkpoint & Restore*. GitHub. 2018. URL: `https://github.com/docker/cli/blob/c12a4d3b34b3fa326bfbe8cdffce27a23504544e/experimental/checkpoint-restore.md` (visited on 9 Oct. 2020) (cit. on pp. 17, 30).

[Due05] Jason Duell. *The Design and Implementation of Berkeley Lab's Linuxcheckpoint/Restart*. LBNL–54941. Berkley, CA (US): Lawrence Berkeley National Laboratory, 2005 (cit. on p. 12).

[Goo] Google Inc. *Google Kubernetes Engine (GKE)*. Google Cloud. URL: `https://cloud.google.com/kubernetes-engine` (visited on 7 Dec. 2020) (cit. on p. 1).

[Goo20] Google Inc. *Live Migration | Compute Engine Documentation*. Google Cloud. 2 Oct. 2020. URL: `https://cloud.google.com/compute/docs/instances/live-migration` (visited on 12 Nov. 2020) (cit. on p. 1).

[Hyk13] Solomon Hykes. “The Future of Linux Containers”. PyCon. 2013. URL: https://www.youtube.com/watch?v=wW9CAH9nSLs (visited on 18 Dec. 2020) (cit. on p. 1).

[Hyk15] Solomon Hykes. *Introducing runC: A Lightweight Universal Container Runtime*. Docker Blog. 22 June 2015. URL: https://www.docker.com/blog/runc/ (visited on 15 Sept. 2020) (cit. on pp. 10 sq.).

[Jon11] Jonathan Corbet. *Checkpoint/Restart (Mostly) in User Space*. LWN.net. 19 Nov. 2011. URL: https://lwn.net/Articles/452184/ (visited on 12 Oct. 2020) (cit. on p. 13).

[Jon12] Jonathan Corbet. *Preparing for User-Space Checkpoint/Restore*. LWN.net. 31 Jan. 2012. URL: https://lwn.net/Articles/478111/ (visited on 12 Oct. 2020) (cit. on p. 13).

[Kaz15] Saied Kazemi. *How Did the Quake Demo from DockerCon Work?* Kubernetes Blog. 2015. URL: https://kubernetes.io/blog/2015/07/how-did-quake-demo-from-dockercon-work/ (visited on 1 June 2020) (cit. on pp. 2, 16).

[Kol12] Kir Kolyshkin. *[CRIU] CRtools 0.1 Released!* OpenVZ – LiveJournal. 24 July 2012. URL: https://openvz.livejournal.com/42414.html (visited on 12 Oct. 2020) (cit. on p. 16).

[Kubaa] The Kubernetes Authors. *Kubernetes/Community - API Conventions*. GitHub. URL: https://github.com/kubernetes/community/blob/04531ba1ae97b54e06feb055ba1678be54e235cc/contributors/devel/sig-architecture/api-conventions.md (visited on 30 Sept. 2020) (cit. on p. 21).

[Kubab] The Kubernetes Authors. *Kubernetes/Community - Changing the API*. GitHub. URL: https://github.com/kubernetes/community/blob/04531ba1ae97b54e06feb055ba1678be54e235cc/contributors/devel/sig-architecture/api_changes.md (visited on 30 Sept. 2020) (cit. on pp. 21, 40).

[Kubac] KubeVirt Authors. *Kubevirt/Kubevirt*. URL: https://github.com/kubevirt/kubevirt (visited on 13 Dec. 2020) (cit. on p. 19).

[KVM] KVM Project. *Migration*. KVM. URL: https://www.linux-kvm.org/page/Migration (visited on 12 Nov. 2020) (cit. on p. 1).

[LH10] Oren Laadan and Serge E Hallyn. “Linux-CR: Transparent Application Checkpoint-Restart in Linux”. In: *Proceedings of the Linux Symposium*. Ottawa Linux Symposium. Ottawa, 2010 (cit. on p. 12).

[LLM88] M. J. Litzkow, M. Livny and M. W. Mutka. “Condor-a Hunter of Idle Workstations”. In: *Proceedings of the 8th International Conference on Distributed Computing Systems*. San Jose, CA, USA, 1988, pp. 104–111 (cit. on p. 12).

[Mes18] David Messina. *5 Years Later, Docker Has Come a Long Way*. Docker Blog. 20 Mar. 2018. URL: `https://www.docker.com/blog/5-years-later-docker-come-long-way/` (visited on 18 Dec. 2020) (cit. on p. 1).

[Mic] Microsoft Corporation. *Azure Kubernetes Service (AKS)*. Microsoft Azure. URL: `https://azure.microsoft.com/en-us/services/kubernetes-service/` (visited on 7 Dec. 2020) (cit. on p. 1).

[Mic12] Michael Kerrisk. *LCE: Checkpoint/Restore in User Space: Are We There Yet?* LWN.net. 20 Nov. 2012. URL: `https://lwn.net/Articles/525675/` (visited on 12 Oct. 2020) (cit. on p. 13).

[Mil+00] Dejan S. Milojičić et al. "Process Migration". In: *ACM Computing Surveys* 32.3 (2000), pp. 241–299 (cit. on p. 14).

[MKK08] Andrey Mirkin, Alexey Kuznetsov and Kir Kolyshkin. "Containers Checkpointing and Live Migration". In: *Proceedings of the Linux Symposium*. Linux Symposium. Vol. 2. Ottawa, 2008, p. 8 (cit. on pp. 12, 18).

[MSV20] Janakiram MSV. "AWS Responds To Anthos And Azure Arc With Amazon EKS Anywhere". In: *Forbes. Cloud* (12 June 2020) (cit. on p. 1).

[Nad+17] Shripad Nadgowda et al. "Voyager: Complete Container State Migration". In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 2017, pp. 2137–2142 (cit. on pp. 18, 62).

[Naq19] Amar Naqvi. *Why Kubernetes Has Emerged as a Key Ingredient in Edge Computing*. The New Stack. 15 Nov. 2019. URL: `https://thenewstack.io/why-kubernetes-has-emerged-as-a-key-ingredient-in-edge-computing/` (visited on 16 Dec. 2020) (cit. on p. 2).

[Ope20a] Open Container Initiative. *Opencontainers/Runtime-Spec*. Open Container Initiative, 2020. URL: `https://github.com/opencontainers/runtime-spec` (visited on 28 Sept. 2020) (cit. on pp. 10 sq.).

[Ope20b] The OpenStack Project. *OpenStack Docs: Configure Live Migrations*. 31 Aug. 2020. URL: `https://docs.openstack.org/nova/victoria/admin/configuring-migrations.html` (visited on 14 Dec. 2020) (cit. on p. 1).

[Osm+03] Steven Osman et al. "The Design and Implementation of Zap: A System for Migrating Computing Environments". In: *ACM SIGOPS Operating Systems Review* 36 (SI 2003), pp. 361–376 (cit. on p. 19).

[PM83] Michael L. Powell and Barton P. Miller. "Process Migration in DEMOS/MP". In: *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*. SOSP '83. New York, NY, USA: Association for Computing Machinery, 1983, pp. 110–119 (cit. on p. 12).

[Pod] Podman. *Podman Checkpointing*. URL: `https://podman.io/getting-started/checkpoint.html` (visited on 9 Oct. 2020) (cit. on pp. 2, 16 sq.).

[Pos81] Jon Postel. *Transmission Control Protocol*. STD 7. RFC 793. RFC Editor, Sept. 1981 (cit. on p. 34).

[Pro] Proxmox Server Solutions GmbH. *Features - Proxmox VE*. URL: `https://www.proxmox.com/en/proxmox-ve/features` (visited on 12 Nov. 2020) (cit. on p. 1).

[Pul+19] Carlo Puliafito et al. "Container Migration in the Fog: A Performance Evaluation". In: *Sensors* 19.7 (2019), p. 1488 (cit. on pp. 14 sqq., 18).

[Reb16a] Adrian Reber. *From Checkpoint/Restore to Container Migration*. Red Hat Blog. 26 Sept. 2016. URL: `https://www.redhat.com/en/blog/checkpointrestore-container-migration` (visited on 17 Sept. 2020) (cit. on p. 16).

[Reb16b] Adrian Reber. "Process Migration in a Parallel Environment". Doctoral dissertation. Stuttgart: Universität Stuttgart, 2016 (cit. on pp. 15 sq.).

[Reb19] Adrian Reber. "Container Live Migration". 2019. URL: `https://media.ccc.de/v/ASG2019-120-container-live-migration` (visited on 22 Sept. 2020) (cit. on pp. 13 sq., 16).

[Reb20] Adrian Reber. *Provide Support for Checkpoint and Restore · Pull Request #4199 · Cri-o/Cri-o*. GitHub. 15 Sept. 2020. URL: `https://github.com/cri-o/cri-o/pull/4199` (visited on 18 Dec. 2020) (cit. on p. 16).

[RR81] Richard F. Rashid and George G. Robertson. "Accent: A Communication Oriented Network Operating System Kernel". In: *ACM SIGOPS Operating Systems Review* 15.5 (1981), pp. 64–75 (cit. on p. 12).

[RV14] Adrian Reber and Peter Väterlein. "Checkpoint/Restore in User-Space with Open MPI". In: *1. Baden-Württemberg Center of Applied Research Symposium on Information and Communication Systems - SInCom2014*. Furtwangen: Hochschule Furtwangen, 2014, pp. 50–54 (cit. on pp. 13–16).

[Sat17] M. Satyanarayanan. "The Emergence of Edge Computing". In: *Computer* 50.1 (2017), pp. 30–39 (cit. on p. 2).

[SLC17] S. Shirinbab, L. Lundberg and E. Casalicchio. "Performance Evaluation of Container and Virtual Machine Running Cassandra Workload". In: *3rd International Conference of Cloud Computing Technologies and Applications (CloudTech)*. 2017, pp. 1–8 (cit. on p. 1).

[Ste96] G. Stellner. "CoCheck: Checkpointing and Process Migration for MPI". In: *Proceedings of International Conference on Parallel Processing*. Honolulu, HI, USA: IEEE, 1996, pp. 526–531 (cit. on p. 12).

[TLC85] Marvin M. Theimer, Keith A. Lantz and David R. Cheriton. “Preemptable Remote Execution Facilities for the V-System”. In: *ACM SIGOPS Operating Systems Review* 19.5 (1985), pp. 2–12 (cit. on p. 12).

[Tor+19] Roberto Torre et al. “Towards a Better Understanding of Live Migration Performance with Docker Containers”. In: *European Wireless 2019; 25th European Wireless Conference*. 2019, pp. 1–6 (cit. on p. 18).

[Ver+15] Abhishek Verma et al. “Large-Scale Cluster Management at Google with Borg”. In: *Proceedings of the Tenth European Conference on Computer Systems - EuroSys ’15*. Bordeaux, France: ACM Press, 2015, pp. 1–17 (cit. on p. 4).

[VMw] VMware Inc. *What Is vMotion? | Live Migration of Virtual Machines*. VMware. URL: `https://www.vmware.com/de/products/vsphere/vmotion.html` (visited on 12 Nov. 2020) (cit. on p. 1).

[Win17] Sascha Winkelhofer. “Konzeption und Umsetzung von Live-Migration für Docker-Container”. Bachelor’s Thesis. Universität Ulm, 2017 (cit. on p. 18).

[YuJ16] Yu-Ju Hong. *Introducing Container Runtime Interface (CRI) in Kubernetes*. Kubernetes Blog. 19 Dec. 2016. URL: `https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/` (visited on 18 Nov. 2020) (cit. on pp. 10 sq., 28).

[Zhi19] Zhiyue Qiu. *Kubernetes-Pod-Migration*. May 2019. URL: `https://github.com/qzysw123456/kubernetes-pod-migration` (visited on 18 Nov. 2020) (cit. on p. 19).

[ZN01] Hua Zhong and Jason Nieh. *CRAK: Linux Checkpoint/Restart As a Kernel Module*. Technical Report CUCS-014-01. Columbia University, 2001 (cit. on p. 12).

Bibliography: Kubernetes Documentation

[Kuba] Kubernetes Authors. *Annotations*. Kubernetes. URL: `https://kubernetes.io/docs/concepts/overview/working-with-objects/annotations/` (visited on 13 Dec. 2020) (cit. on p. 6).

[Kubb] Kubernetes Authors. *Cluster Networking*. Kubernetes. URL: `https://kubernetes.io/docs/concepts/cluster-administration/networking/` (visited on 13 Dec. 2020) (cit. on pp. 8, 34).

[Kubc] Kubernetes Authors. *Controllers*. Kubernetes. URL: `https://kubernetes.io/docs/concepts/architecture/controller/` (visited on 13 Dec. 2020) (cit. on pp. 7, 42).

[Kubd] Kubernetes Authors. *CronJob*. Kubernetes. URL: `https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/` (visited on 13 Dec. 2020) (cit. on p. 7).

[Kube] Kubernetes Authors. *DaemonSet*. Kubernetes. URL: `https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/` (visited on 13 Dec. 2020) (cit. on p. 8).

[Kubf] Kubernetes Authors. *Deployments*. Kubernetes. URL: `https://kubernetes.io/docs/concepts/workloads/controllers/deployment/` (visited on 13 Dec. 2020) (cit. on p. 7).

[Kubg] Kubernetes Authors. *Extend the Kubernetes API with CustomResourceDefinitions*. Kubernetes. URL: `https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions/` (visited on 22 Sept. 2020) (cit. on pp. 6, 41).

[Kubh] Kubernetes Authors. *Init Containers*. Kubernetes. URL: `https://kubernetes.io/docs/concepts/workloads/pods/init-containers/` (visited on 13 Dec. 2020) (cit. on p. 7).

[Kubi] Kubernetes Authors. *Jobs*. Kubernetes. URL: `https://kubernetes.io/docs/concepts/workloads/controllers/job/` (visited on 13 Dec. 2020) (cit. on pp. 7, 22).

[Kubj] Kubernetes Authors. *Kubernetes API Concepts*. Kubernetes. URL: `https://kubernetes.io/docs/reference/using-api/api-concepts/` (visited on 14 Dec. 2020) (cit. on pp. 6, 41).

[Kubk] Kubernetes Authors. *Kubernetes Components*. Kubernetes. URL: `https://kubernetes.io/docs/concepts/overview/components/` (visited on 13 Dec. 2020) (cit. on pp. 4, 9).

[Kubl] Kubernetes Authors. *Kubernetes Documentation*. Kubernetes. URL: `https://kubernetes.io/docs/home/` (visited on 22 Sept. 2020) (cit. on p. 4).

[Kubm] Kubernetes Authors. *Labels and Selectors*. Kubernetes. URL: `https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/` (visited on 13 Dec. 2020) (cit. on p. 6).

[Kubn] Kubernetes Authors. *Network Plugins*. Kubernetes. URL: `https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/` (visited on 13 Dec. 2020) (cit. on p. 34).

[Kubo] Kubernetes Authors. *Nodes*. Kubernetes. URL: `https://kubernetes.io/docs/concepts/architecture/nodes/` (visited on 13 Dec. 2020) (cit. on p. 9).

[Kubp] Kubernetes Authors. *Object Names and IDs*. Kubernetes. URL: `https://kubernetes.io/docs/concepts/overview/working-with-objects/names/` (visited on 13 Dec. 2020) (cit. on p. 5).

[Kubq] Kubernetes Authors. *Operator Pattern*. Kubernetes. URL: `https://kubernetes.io/docs/concepts/extend-kubernetes/operator/` (visited on 14 Dec. 2020) (cit. on p. 42).

[Kubr] Kubernetes Authors. *Persistent Volumes*. Kubernetes. URL: `https://kubernetes.io/docs/concepts/storage/persistent-volumes/` (visited on 13 Dec. 2020) (cit. on pp. 7, 32).

[Kubs] Kubernetes Authors. *Pod Lifecycle*. Kubernetes. URL: `https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/` (visited on 13 Dec. 2020) (cit. on p. 22).

[Kubt] Kubernetes Authors. *Pods*. Kubernetes. URL: `https://kubernetes.io/docs/concepts/workloads/pods/` (visited on 13 Dec. 2020) (cit. on pp. 6, 8).

[Kubu] Kubernetes Authors. *Service*. Kubernetes. URL: `https://kubernetes.io/docs/concepts/services-networking/service/` (visited on 13 Dec. 2020) (cit. on pp. 8, 10).

[Kubv] Kubernetes Authors. *StatefulSets*. Kubernetes. URL: `https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/` (visited on 13 Dec. 2020) (cit. on p. 7).

[Kubw] Kubernetes Authors. *The Kubernetes API*. Kubernetes. URL: `https://kubernetes.io/docs/concepts/overview/kubernetes-api/` (visited on 13 Dec. 2020) (cit. on p. 4).

[Kubx] Kubernetes Authors. *Understanding Kubernetes Objects*. Kubernetes. URL: `https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/` (visited on 13 Dec. 2020) (cit. on pp. 5 sq.).

[Kuby] Kubernetes Authors. *Volumes*. Kubernetes. URL: `https://kubernetes.io/docs/concepts/storage/volumes/` (visited on 13 Dec. 2020) (cit. on p. 7).

[Kubz] Kubernetes Authors. *What Is Kubernetes?* Kubernetes. URL: `https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/` (visited on 13 Dec. 2020) (cit. on p. 4).

Contents of the Attached Disk

- The file `thesis_schrettenbrunner.pdf` is a digital version of this thesis. It uses a different font than the printed variant for better readability on screens.
- The directory `code` contains three subdirectories with the source code of the prototype.
  - `containerd-cri` is a fork of the containerd CRI plugin which includes the necessary changes to support C/R.
  - `kubernetes` is a fork of the monolithic kubernetes repository, which contains all of its components. The fork contains the necessary changes to support Pod migration, as well as the extended CRI.
  - `podmigration-operator` contains the migration operator, which includes the custom MigratingPod resource and the migration controller. The subdirectory `evaluation` contains the programs written for evaluation and the evaluation results.
- The directory `sources` contains copies of all online-only sources (except videos) that were referenced throughout this work.

The source code of the prototype can also be found on the author's GitHub account:

- `https://github.com/schrej/containerd-cri/tree/feature/checkpoint-restore`
- `https://github.com/schrej/kubernetes/tree/feature/pod-migration`
- `https://github.com/schrej/podmigration-operator`