technische universität
dortmund

Master Thesis

State-Preserving Container Orchestration in Failover Scenarios

Henri Schmidt
February 2023

Supervisors:
Prof. Dr. Dr. Klaus-Tycho Förster
Dr. Zeineb Rejiba
Dr. Raphael Eidenbenz

Reviewer:
Prof. Dr. Dr. Klaus-Tycho Förster
Dr. Raphael Eidenbenz

Technical University Dortmund
Faculty for Computer Science
Chair for practical Computer Science (LS-4)
https://ls4-www.cs.tu-dortmund.de

In Cooperation with:
Hitachi Energy Switzerland

Abstract

Containers have been widely adopted for deployment of high availability applications and services. This adoption is in part due to the native support of fault tolerance mechanisms in container orchestration frameworks such as Kubernetes. While Kubernetes provides service replication as a fault tolerance mechanism for stateless applications, service replication does not satisfy requirements for stateful applications. Currently this shortcoming is addressed by data replication in databases. This requires a tight coupling and modification of the stateful application to support high availability. Thus, this thesis proposes a new Checkpoint/Restore (C/R) Kubernetes operator to achieve fault tolerance for stateful applications without any modification of the application. The operator takes a checkpoint in a configurable interval. In case of a fault a new application container is created automatically from the most recent checkpoint. We compare the proposed approach with a more conventional approach in which we pull and restore the application state from the application through an API. We measure the overhead of both methods, the service interruption and the recovery time in case of faults. We find the C/R Operator has similar performance in recovery time as the traditional approach, but does not need any application modification. The results signify C/R as a promising technology for a fault tolerance mechanism for stateful applications.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Contributions . . . 2
- 1.3 Organization . . . 2

2 Literature Review **3**
- 2.1 Failover Scenarios . . . 3
- 2.2 Fault Tolerance Mechanisms . . . 3
  - 2.2.1 Replication . . . 4
  - 2.2.2 Redundancy . . . 4
  - 2.2.3 Checkpoint Restore . . . 4
- 2.3 Container Orchestration . . . 6
- 2.4 Reflections . . . 6

3 Fundamentals **9**
- 3.1 Kubernetes . . . 9
  - 3.1.1 Architecture . . . 9
  - 3.1.2 Resources . . . 11
  - 3.1.3 Operator . . . 12
  - 3.1.4 Kubernetes Node Architecture . . . 12
- 3.2 Checkpoint/Restore . . . 13
- 3.3 OCI Image format . . . 14
- 3.4 Container runtime . . . 15
  - 3.4.1 seccomp . . . 15
  - 3.4.2 cgroups . . . 15
  - 3.4.3 capabilities . . . 16
  - 3.4.4 namespaces . . . 16

4 System Design **19**
- 4.1 C/R Operator . . . 19
  - 4.1.1 Design Principles . . . 19

4.1.2 Overview . . . 19
4.1.3 Components . . . 20
4.1.4 Limitations . . . 22
4.2 Pull State Operator . . . 22
4.2.1 Design principles . . . 22
4.2.2 Overview . . . 23
4.2.3 Controllers . . . 24
4.2.4 Configuration . . . 25

5 Implementation 27
5.1 Pull State Operator . . . 27
5.2 C/R Operator . . . 30
5.3 Overcoming Implementation Obstacles . . . 31
5.3.1 CPU Architecture . . . 31
5.3.2 Debugging the communication between runc and CRIU . . . 32

6 Evaluation 35
6.1 Experimental Setup . . . 35
6.1.1 Applications . . . 35
6.1.2 Cluster . . . 36
6.1.3 Experiments . . . 36
6.2 Overhead . . . 37
6.3 Checkpoint performance . . . 41
6.4 Failover . . . 41
6.5 Recovery time comparison . . . 43
6.6 State Quality . . . 44
6.7 Discussion . . . 44

7 Conclusion & Future Work 47
7.1 Conclusion . . . 47
7.2 Future Work . . . 47

A Appendix 49
A.1 Failover with Kubernetes Service . . . 49

Table of Figures 51

Bibliography 60

Affidavit 60

Chapter 1

Introduction

1.1 Motivation

Containers are a lightweight virtualization scheme, that has gained popularity in recent years. They isolate different applications and confine them to a separate process space, networks and disk through Linux kernel features such as *seccomp*, *cgroups*, network and mount namespaces. In 2015, the Open Container Initiative (OCI) [58] was founded to standardize two specifications for the image format and the runtime of containers. Multiple implementations for containers exist on Linux at the moment.

For applications with availablity and scalability requirements, running containers on one single machine is not sufficient. Orchestration frameworks help system administrators to deploy and manage multiple containers on multiple nodes in a cluster. A prominent orchestration framework mainly developed by Google is Kubernetes [46]. It provides the abilities to automatically deploy replicated services, manage networking, load-balancing and persistent storage.

Fault tolerance or high availability is one of the central features of orchestration frameworks. For stateless applications replication is a way to achieve high availability as no state needs to be synchronised. If one replica fails any other replica can continue serving the requests. On the other hand, replication for stateful applications is more challenging. State synchronisation between the replicas is the major reason for this. This problem can be solved by state machine algorithms such as Raft [57] or Paxos [47]. These algorithms can be implemented by the application itself or the state can be transferred to a data store implementing a state machine algorithm.

Another approach to state replication is redundancy. In this approach all inputs to the application are redundantly done on two different nodes. This necessitates deterministic applications.

A new fault tolerance mechanism is the employment of Checkpoint/Restore (C/R) to restore the running application on another node from a memory checkpoint made pre-

viously. The advantage of this procedure is a lowered application complexity, as state synchronisation algorithms are not required anymore. Furthermore, it requires less resources on the second standby node. On the other hand, checkpointing an application regularly introduces an overhead, because processes need to be frozen to save a consistent snapshot of the processor state and the memory.

Recently, there have been efforts to integrate minimal checkpoint functionality into Kubernetes [2], but no evaluation of C/R in combination with Kubernetes exists currently.

1.2 Contributions

Achieving fault tolerance for stateful applications is inherently more difficult than for stateless containers, as the state needs to be taken into account while designing fault tolerance mechanisms. This work evaluates an approach for fault tolerance in Kubernetes using C/R as the fault tolerance mechanism. This is achieved by first designing and implementing the architecture proposed in this work and subsequently performing experiments to evaluate the performance of the proposed approach.

First, failover scenarios are identified by exploring possible faults in applications. Next, fault tolerance mechanisms for stateful containers are outlined. Moreover, two of the outlined mechanisms are implemented in the Kubernetes orchestration framework. Both approaches achieve a recovery time of 3.5-6.5 seconds in the Experimental evaluation. While the C/R Operator has more overall overhead, it does not need any modification of the application. In failover scenarios some amount of state will be lost and it requires a uniform CPU architecture across source and target node. Although the C/R Operator needs some prerequisites to function, it is a promising technology for use in failover scenarios.

1.3 Organization

Chapter 2 presents related works and summarises the current state of scientific works. First, it shows failover scenarios, then it illustrates fault tolerance mechanisms, container orchestration and adds some reflections about the technologies. Chapter 3 details the fundamental technologies used in this work. It describes Kubernetes, C/R, the OCI image format and the internals of container runtimes. In Chapter 4 the design for the implemented system is described. After the design description, Chapter 5 presents the implementation. First, it details processes of the operators. Then it describes some obstacles and debugging methods used in the implementation process. Chapter 6 presents the experimental evaluation of the implemented method and discusses the results. It shows overhead and recovery time in case of failover. Lastly, Chapter 7 summarizes the findings and presents future work.

Chapter 2

Literature Review

First, the literature review shows failover scenarios in Section 2.1. Then different fault tolerance mechanisms are presented in Section 2.2. Lastly, Section 2.4 adds some thoughts about the presented mechanisms.

2.1 Failover Scenarios

There are several failover scenarios mentioned in the literature. More specifically for container systems, faults can happen at the node, container or application level [48]. Faults on the node level are either hardware failures [49] or failures of the virtual machine [6]. On the Container level failures are described as pod process failures by Vayghan et al. [67]. These types of failures happen when the orchestration framework detects that the process running inside the container has stopped. This is similar to the fail-stop model used by Zhou and Tamir [73, 74]. Application level failures can happen through bugs in the application [51]. Other failures include network failures at the node level or resource exhaustion at the application level.

2.2 Fault Tolerance Mechanisms

As stated in Section 1.1, there are fundamentally three approaches to make stateful applications fault tolerant. One commonly used replication scheme that provides scalability in addition to fault tolerance is replication with state synchronisation. Another approach is to let the application run redundantly on two nodes and switch over to the second node should the first one fail. Recently, C/R is discussed in the literature as an alternative to the two aproaches mentioned above.

2.2.1 Replication

For stateless applications replication is one easy solution to get better fault tolerance. The replicas can be completely independent of each other as no state has to be synchronised between them. Stateful applications on the other hand require consensus protocols to be replicated as they require a consistent state between all the replicas to function. Replication is always subject to the PACELC theorem [1], which is an extended version of the CAP theorem. The CAP theorem states that in a case of a partition, either availability or consistency needs to be sacrificed in order to be partition tolerant. PACELC is an extension of this theorem to include latency. Under normal operation, replication schemas need to choose between latency or consistency.

Netto et al. [53, 55, 54] show an algorithm based on the raft consesus protocol, that guarantees availability and partition tolerance while only guaranteeing eventual consistency. This type of state synchronisation is optimal for applications where absolute consistency is not required.

Gray et al. [17] implement a redundancy scheme, that is configurable for either high latency with consistency or only eventual consistency with lower latency. The authors also discuss the advantages and disadvantages regarding latency and consistency.

2.2.2 Redundancy

Redundancy relies on the fact that the application being redundantly run is deterministic. Both copies of the application get the same exact input and compute the exact same result. If a fault is detected in the main application, failover to the second application happens and resumes normal operation.

Vayghan et al. [67, 68] propose such a redundancy mechanism implemented as a Kubernetes controller and a forwarding process that replicates the network traffic to the standby node.

Johansson et al. [20] compare a redundant setup with hot standby to a setup with cold standby in the contexts of virtual distributed controller nodes. Hot and cold standby are concepts that describe how fast a standby system can take over in the case of failure. While hot standby systems are optimized for the fastest recovery possible, cold standby systems have more leeway in the recovery process, use less resources and are often more cost effective.

2.2.3 Checkpoint Restore

C/R is a technique primarily used for migration of containers from one machine to another [21, 18, 62, 16, 6, 52, 56, 70, 64, 50, 65, 66]. The cited works use different strategies and highlight different aspects of the process. Minimizing downtime and transferred data is a major objective of these works.

Work	Migration	Failover	Kubernetes	Customized C/R
Juniot et al [21]	✓	✗	✓	✗
Gundall et al [18]	✓	✗	✗	✗
Rattihalli et al [62]	✓	✗	✓	✗
Govindaraj et al [16]	✓	✗	✗	✗
Cao et al [6]	✓	✓	✗	✗
Nadgowda et al [52]	✓	✗	✗	✗
Oh et al [56]	✓	✗	✗	✗
Xu et al [70]	✓	✗	✗	✗
Schrettenbrunner [64]	✓	✗	✓	✗
Ma et al [50]	✓	✗	✗	✗
Yokata et al [65]	✓	✗	✗	✗
Terneborg et al [66]	✓	✓	✗	✗
Zhou et al [73]	✗	✓	✗	✓
Zhou et al [74]	✗	✓	✗	✓
Venkatesh et al [69]	✗	✗	✗	✓
Chen et al [7]	✗	✓	✗	✓
Kannan et al [22]	✗	✓	✗	✓
Müller et al [51]	✓	✓	✓	✗

Table 2.1: Comparison of the reviewed fault tolerance mechanisms

Another research direction is the optimization of the C/R process, especially the optimization of Checkpoint/Restore in Userspace (CRIU) [9], the prevalent tool used for C/R [73, 74, 69, 7, 22]. These works dive deep into the checkpointing process and optimize for checkpoint and recovery speed.

Rattihalli et al. [62] explores C/R for fast auto-scaling and adjusts the allocation of containers in a Kubernetes cluster using container migration. They improve CPU and memory allocation by migrating containers to underutilized nodes.

Schrettenbrunner [64] implements integrated Kubernetes components to enable migration of Kubernetes pods to other machines. The author argues that pods should be first copied to the new machine and deleted on the source in a subsequent step.

Finally, Müller et al. [51] propose an architecture for migration and failover with Kubernetes pods while maintaining and replaying network connections using an interceptor. Even though their work constitutes a good starting point for integration of C/R into Kubernetes, they omit any experimental evaluation of the proposed architecture.

2.3 Container Orchestration

The explored works focus mainly on two different container orchestration frameworks. By far, the most popular one is Kubernetes or derivates of Kubernetes in the cloud. One notable alternative is Docker Swarm [4].

Kubernetes is a modular and extensible framework that manages the deployment and administration of containers on multiple nodes. It is able to detect failures of components in the system and restart failed components on the same or other nodes. As shown in Table 2.1, many works concentrate on Kubernetes as their orchestration framework. The table also shows no works expect from Müller et al. [51] exists, that uses C/R for failover scenarios in Kubernetes.

Other works explore Docker Swarm as the Orchestration Framework of their choice [19, 71, 72]. Docker Swarm is simpler than Kubernetes in the sense that most of the components are fixed and hidden from the user. So the user is not responsible for managing those components. Furthermore Docker Swarm is not as extensible as Kubernetes with no way to extend the control components.

2.4 Reflections

The presented fault tolerance mechanisms have very different properties depending on the assumptions and implementation details. Using replication faulty replicas can be detected even when they do not adhere to the fault-stop model as there is a majority vote to determine a consensus in the cluster. Redundancy as well as C/R rely on other fault detection mechanisms like a heartbeat signal that is not able to detect semantic faultiness of a node.

One advantage of redundancy and C/R with respect to replication is the possibility of implementing both techniques in an application transparent manner. This means no modification to the source code of the application is necessary. In cases where the cost of modification is prohibitive such as certification cost or legacy application, where the source code is lost, this is an important factor.

Although redundancy works without modifying the application, this requires the application to be deterministic in its execution. Otherwise, only replaying all inputs from one copy to the other does not produce the same state changes on both copies. The same is true for replay of cached requests after a checkpoint is restored as suggested by Müller et al. [51].

If inconsistency is acceptable, both redundancy and C/R can be used even on non-deterministic applications. In the non-deterministic case, redundancy is almost guaranteed to yield different results, whereas C/R mitigates this as long as most of the execution of the application is deterministic.

Redundancy on the other hand has a better recovery time than C/R, as the only operation needing to happen on a fault is the switchover itself, while C/R needs to first restore the checkpoint, replay missing inputs and then do the switchover.

This thesis focuses only on C/R and its implementation using the Kubernetes orchestration framework. It also only focuses on experimental evaluation while responding to node faults.

Chapter 3

Fundamentals

The fundamentals chapter details fundamental technologies used in this work. Section 3.1 describes the Kubernetes architecture, resources, operator fundamentals and node architecture. A description of the C/R technology is given in Section 3.2. Section 3.3 illustrates the standard image format used by Container Runtime Engine (CRE)s. Section 3.4 presents the kernel features used by container runtimes to implement containers.

3.1 Kubernetes

Kubernetes is a Container Orchestration Framework widely used in the industry [8]. It has a modular architecture and enables users to manage containers across multiple nodes. It handles scheduling, networking, access control and provides capabilities for data storage.

3.1.1 Architecture

The basic Kubernetes architecture is displayed in Figure 3.1. In its base configuration, Kubernetes differentiates between the control plane and compute machines (also called worker nodes). The control plane runs all the components needed for managing the cluster and the worker nodes run the actual applications hosted on the cluster. For High Availability, multiple nodes can run the control plane. Control plane nodes are by default configured to only run control plane components, but can also be configured to run application workloads.

Every node has a Kubelet [41] process running on it that communicates over a REST-API with the API server. Kubelet is responsible for interacting with the underlying container runtime through the Container Runtime Interface (CRI). It is able to start and stop containers, get logs, run commands in containers and generally get information of the running containers on the node. One new capability of Kubelet is the ability to make checkpoints of running containers. This feature was merged very recently and is still in the experimental stage. [3]

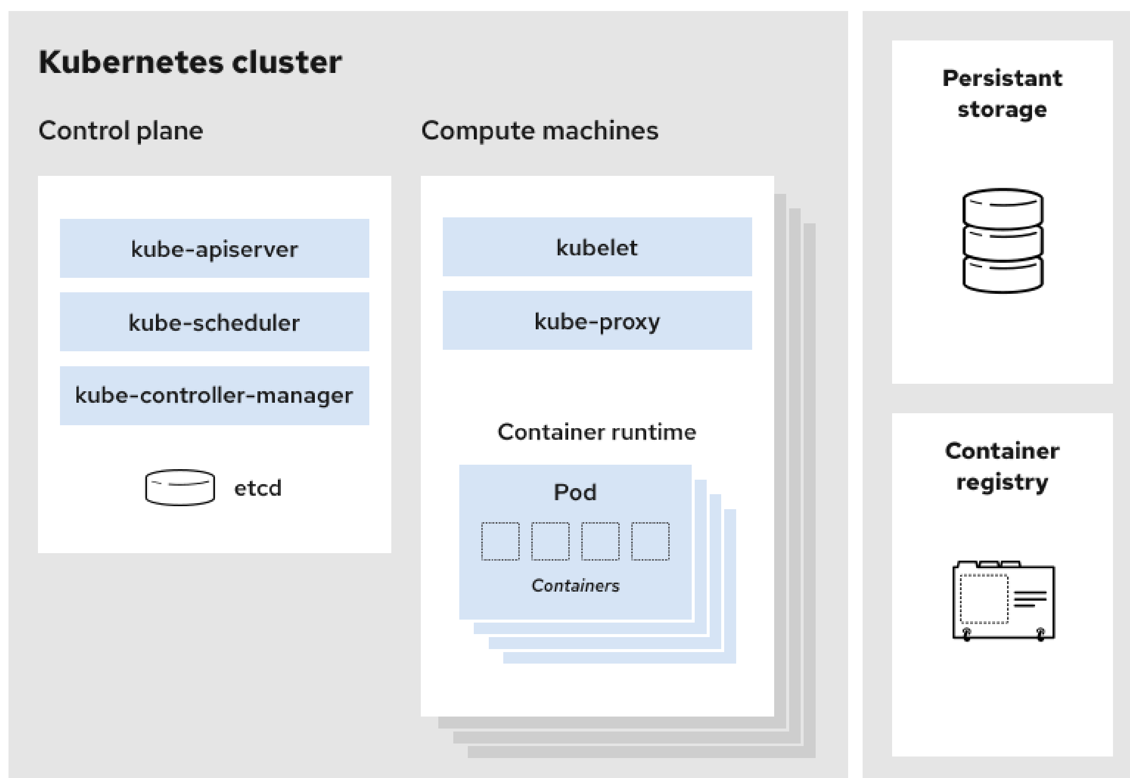

Figure 3.1: Kubernetes Architecture [63]

The kube-controller-manager [38] is a component that contains all the standard controllers for the default resources available in Kubernetes. Controllers are responsible for implementing the behaviour specified by the resources.

All the configuration information as well as the resources of the cluster are stored in etcd [15]. Etcd is a highly available distributed key-value store. In the context of Kubernetes, the configuration information is mostly all the resources that were either created by the user such as deployments or services or other resources created by components of the cluster itself. It also usually stores configuration information for flannel. Flannel is a technology that provides overlay networking for the cluster. It implements routing for cluster internal IP addresses.

The kube-proxy [40] component is responsible for implementing part of the Service (see Section 3.1.2) behaviour. It works on top of the overlay network created by flannel and the Container Networking Interface (CNI). For service instances defined by the Service resource it handles implementing round-robin routing between all the Pods. There are multiple routing backends kube-proxy can use. The simplest is iptables and works by inserting iptables rules with the forwarding information into the host system.

Pods are the actual running instances of the application. They consist of multiple containers that share the same network namespace.

Supporting systems for a Kubernetes cluster include persistent storage and a container registry. As the name suggests, the persistent storage provides storage for containers and can be easily connected to the cluster. The container registry stores the necessary container images needed for the cluster.

3.1.2 Resources

Kubernetes works by managing resources in a cluster. These resources are watched by controllers.

Pod Pods [43] are an abstraction over containers and can contain multiple containers. Containers in the same Pod share the network stack with each other so that other containers are accessible on localhost.

In Kubernetes, Pods are the smallest schedulable unit. Pods are scheduled through the kube-scheduler. The scheduler examines the available nodes and their properties to find a suitable node for the given Pod. As soon as a suitable node is chosen the kube-scheduler creates a binding resource. The kubelet process on the node listens to these binding resources and instantiates the containers for the Pod.

ReplicaSet A ReplicaSet [44] is a resource that deploys multiple Pods. In the event that Pods are stopped or deleted, the ReplicaSet will recreate them. It always tries to have the specified number of Pods running.

DaemonSet A DaemonSet [36] is similar to a ReplicaSet in the sense that multiple Replicas are managed. The big difference is that the DaemonSet starts a replica on every node. This is useful for daemons or other helper applications like a logging agent that need to run on every node in the cluster.

Deployment The Deployment [37] is an extension of the ReplicaSet. It manages creating, deleting and updating ReplicaSets for new configurations. Deployments will create a new ReplicaSet for new configurations and slowly scale down the old ReplicaSet while scaling up a ReplicaSet with the new configuration.

Service Services [45] in Kubernetes are used to route or load-balance requests to the correct Pods. As Pods in the cluster are assigned dynamic IPs, networking between different Pods is difficult. This problem is solved by Services by providing a single IP for all replicas of an application. It dynamically updates the list of Pods so applications trying to access an application only need to communicate with the Service. Services also provide capabilities to expose applications outside the cluster.

3.1.3 Operator

Operators [42] are a concept in Kubernetes to automate the management of application Pods. An Operator can contain multiple Controllers and Custom Resource Definitions (CRDs). Controllers implement functionality and resources store information about the cluster.

A CRD defines fields and their types for the custom resource. After the CRD is installed, resources of that type can be created in Kubernetes. Resources themselves only hold data and do not define any functionality. They often contain a specification of the state and the current status of the system. Section 3.1.2 explores some of the standard resources in Kubernetes and the functionality the standard controllers implement based on these resources.

Controllers add functionality to resources through the reconcile loop. Their goal is to bring the system into the state specified by the resource. For this purpose, the Kubernetes Application Programming Interface (API) provides the capability to watch resources. This is typically used by controllers to watch the specific resources belonging to their operator. Standard resources can also be watched by controllers to add additional functionality to existing resources. For example, annotations can be used to store operator-specific information in Deployments. Configuration values can be set through this mechanism. Every change in the resource triggers a reconcile cycle in the controller. The controller then uses this reconcile cycle to change the real state of the system to match the specified state in the resource. It also updates the status of the resources to match the state of the system. The controller can end the reconcile cycle in one of three ways. First, the reconcile was a success and this resource does not need to be reconciled again. Second, there was an error in the reconciliation process and it will be scheduled again. Third, no error occurred but the reconciliation needs to happen at a later point.

3.1.4 Kubernetes Node Architecture

As explained before Kubelet runs on each node. Kubelet is responsible for synchronising the state of the node with the state of the API server. So if a new Pod or Binding resource is created in the API server, the kubelet reacts to this event and creates the corresponding containers on the node. It can also load static Pods from other sources like files or HTTP Servers. Static Pods are then reflected as mirror Pods in the API server. Kubelet can communicate with several different CREs through the CRI.

Figure 3.2 shows the node architecture. CRI-O, cri-dockerd and containerd all implement the CRI and can communicate with Kubelet through this interface. In previous versions, dockerd was directly supported as a backend for Kubelet, but this support has been removed and now needs the cri-dockerd adapter. Dockerd in turn uses containerd

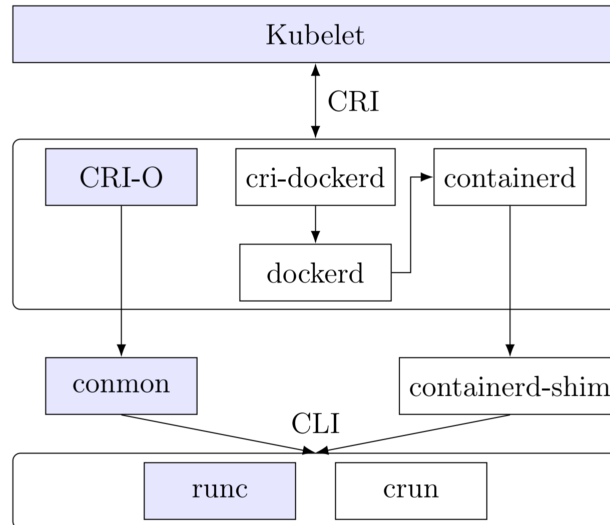

Figure 3.2: Kubernetes Node Architecture

as a backend for running containers. The CRI abstraction allows the Kubelet to support multiple CREs.

In detail the CRI uses gRPC with protobuf messages, usually over UNIX Sockets, for communication. There is also a command line tool called `crictl` that can be used to send CRI messages to container engines. Rather than documenting the specification for the CRI in written text, the specification is done through code and comments only.

In the lower layers of the stack, the container runtime engines use runc or crun to run containers on the host. In addition, conmon and containerd-shim are used by CRI-O and containerd, respectively. Both of these programs keep namespaces alive and monitor the process running in a container.

The components marked in blue are used in this thesis to run the experiments in Chapter 6. Containerd does not support checkpointing containers over CRI yet. In containerd it is only possible to do it through their own command line client.

3.2 Checkpoint/Restore

Checkpointing is a technique to save the complete runtime state of a process. CRIU [9] is a tool that implements this in userspace. The CRIU project implemented several linux kernel patches [14] to be able to implement checkpointing from userspace.

The basic procedure to checkpoint a process as executed by CRIU is to stop the process using either *ptrace* system calls or *cgroups* freezer [10]. In practice not only one process needs to be checkpointed, but the whole process tree starting with the process. This is important for processes that start child processes while running. Freezing the process is needed for the memory to be kept consistent while checkpointing.

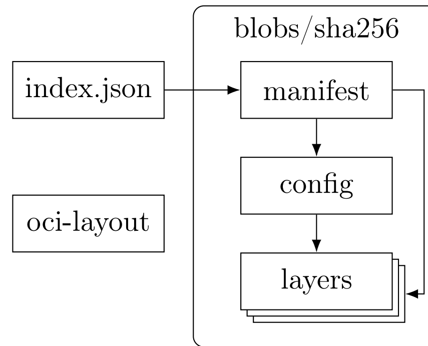

Figure 3.3: Layout of a OCI-Image

After freezing the process tree, CRIU writes all information it can gather about the processes to image files. This is mostly information found in `/proc`, such as file descriptor information, pipe parameters, memory mapped files and processor state information such as registers.

To save the memory, all memory pages of the process need to be exported. This is done by injecting a parasitic code snippet [12] via ptrace system calls into the process and jumping to it. The parasitic code snippet exports all used memory pages by using the vmsplice [11] system call to copy memory pages to pipes opened by CRIU. CRIU reads from these pipes and writes the memory content into image files.

One problem while checkpointing is the handling of open TCP socket connections [13]. With kernel version 3.5, it is now possible to repair established TCP connections. For the use-case in this thesis, we use CRIU's `tcp-close` option to close open TCP connections on restore. Clients then need to reconnect to the server as soon as the restore is finished.

When checkpointing containers, CRIU also deals with *cgroups* and *namespaces*. Some namespaces are created by the CRE and some namespaces are restored by CRIU.

Usually, CRIU shuts down the process after successful checkpointing to avoid inconsistent state for example with open socket connections. For fault tolerance this is not desired and can be disabled with the caveat of inconsistencies in socket data.

3.3 OCI Image format

Starting a container is usually done through images in the OCI image format. This image format specifies how the state of the container filesystem should be set up during startup. It also allows to add annotations that designate useful metadata. Fundamentally, the OCI image format consists of only two files and a folder full of blobs as shown in Figure 3.3. The two root files are the `oci-layout` file, that designates the specific version of the layout used and the `index.json`, which includes a pointer to all manifest files in the blobs. The simplest OCI image only includes one manifest file, but more can be added for different architectures. Usually the blobs directory includes a sha256 directory in which all files are

named after their sha256sum. References to files work through these sha256sums, called digests. The manifest file in the OCI image includes a reference to an image config file, which includes specific configuration options for this image and references the other blobs in the directory and designates a file type for them. In the image config file, we see a declaration of layers for the finished file system. These layers correspond to the blobs present in the image, but often are referenced through a different digest. The reason for this is that blobs are often compressed, but the layers in the image config include the digest of the uncompressed tar archive.

The job of CRI-O is to create the correct file system for the container to start in. To do this overlay filesystems are used, to stack the different layers contained in the OCI image on top of each other. All layers are mounted as read only, except the topmost layer which then represents the actual filesystem of the container.

At this stage, runc started through conmon can create a running container. To do this runc needs to allocate new namespaces for the container, set the correct cgroups, capabilities and possibly activate seccomp.

3.4 Container runtime

Container runtimes need special kernel features to set up a running container. The following sections detail the kernel features used by containers.

3.4.1 seccomp

seccomp [26] or secure computing mode is a linux kernel feature that restricts the access to system calls for a given process. There are two versions of seccomp available. The first version of seccomp restricts access to all system calls expect for the ones that allow to exit the process. The second version also called seccomp-bpf is an extension of the previous version and allows a fine grained access control to system calls.

3.4.2 cgroups

cgroups [24, 25] are a linux kernel technology to limit or reserve physical resources used by processes. They can be used to manage resources such as CPU, memory, Disk I/O or networking for a given process tree. Accounting is also provided for cgroups so that all the processes runnning under the same cgroup can be measured together. Furthermore, cgroups enable the freezing of all processes inside of the group. On most systems Systemd is managing cgroups and CREs should request their cgroups through Systemd.

3.4.3 capabilities

Similar to seccomp, capabilities are a mechanism to enable processes to access specific subsystems of the kernel [23]. In contrast to seccomp, capabilities are usually used to grant processes running as a non-root user additional rights. In the context of containers this is useful to grant special privileges to a process running in a container as every process in the container runs as non-root.

3.4.4 namespaces

The namespace feature in the linux kernel [30] provides isolation of various aspects of the system to containers. Namespace types include: net, mnt, pid, user, ipc, uts and time

net namespace Network namespaces [31] isolate the network stack for processes. This means all the network interfaces, routing tables and firewall rules are isolated.

mnt namespace The mount namespace [29] isolates the different mounts between processes. Containers use this kind of namespace to isolate the filesystem of the container. Bind mounts can be used to connect the filesystem of the host to the filesystem of the container.

pid namespace PID namespaces [32] isolate the visibiliy of processes and can be nested. PID numbers inside the namespace start again from 1 and only the processes in the current and child PID namespaces are visible. For containers this leads to the container processes only being able to interact with processes inside the container.

user namespace A user namespace [33] isolates the UID and GID spaces and other security related features such as keys and capabilities. This gives processes the ability to, e.g. run as the root user in their namespace but still be restricted in the parent namespace. Namespaces get associated with the user namespace that the creating process is in upon creation of the namespace.

ipc namespace Inter process communication namespaces [28] isolate access to SYS V inter process communication resources and posix message queues. Sysv ipc resources include message queues, shared memory and semaphores.

uts namespace UNIX time sharing namespaces [34] enable the isolation of hostname and NIS domain names. In the container context this is often used, to give the container a unique hostname.

time namespace Time namespaces [27] isolate time and date settings for the current system and where introduced in Kernel version 5.6. As time namespaces are so new, no container engine has added support for this feature yet.

Chapter 4

System Design

This chapter describes the system design of the Kubernetes operator for fault tolerance with C/R in Section 4.1. The alternative Pull State approach is also described in Section 4.2. The Pull State Operator is used as a comparison to the C/R Operator.

4.1 C/R Operator

The C/R Operator implements C/R operations for use in failover scenarios. It is using experimental features from Kubernetes to implement checkpointing. Restore is handled directly by the CRE.

This section contains the design principles (Section 4.1.1), an overview (Section 4.1.2) and a description of the components (Section 4.1.3) of the C/R Operator.

4.1.1 Design Principles

The operator was designed with the following principles in mind. First, full Kubernetes integration using the latest features. Second, application transparency meaning the application itself needs to implement as little as possible for the fault tolerance to work.

4.1.2 Overview

Figure 4.1 shows the basic architecture of the operator. The operator is built around the new Kubernetes alpha feature released in v1.25 that enables checkpointing through Kubelet. As shown in the figure, the Controller Manager requests a checkpoint from the Kubelet. This request travels through some layers until it reaches CRIU and creates a checkpoint. Another communication is coordinated between the Agents, for transferring the checkpoints to the correct nodes for restore.

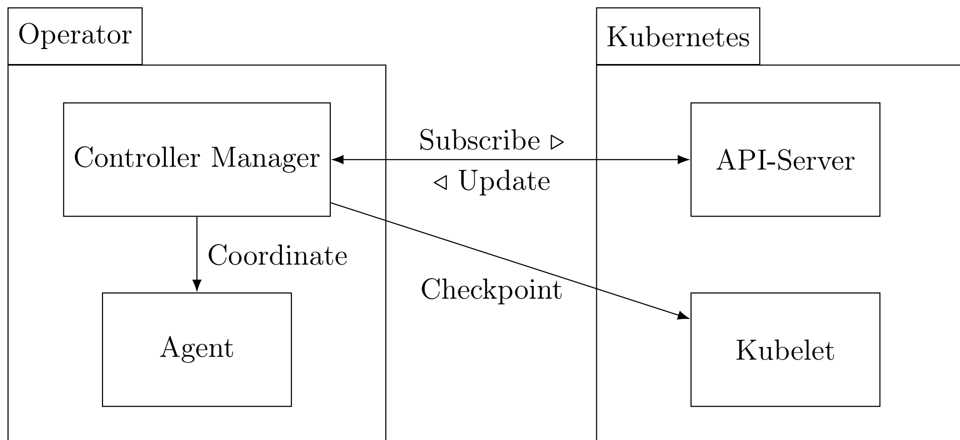

Figure 4.1: C/R Architecture overview

4.1.3 Components

There are only two main components in the operator: the Controller Manager and the Agent. The Controller Manager contains all the controllers needed for the operator and is responsible for coordinating the Agents running on each node. The Agents are checkpointing containers, creating checkpoint images and transferring them between nodes.

Controller Manager

The Controller Manager consists of four controllers and two handlers. Each controller watches changes to one resource type in the cluster. To watch a resource in the cluster, the controller subscribes to new messages concerning the resource and in turn receives updates from the API server for the resources. The two handlers are responsible for checking the liveness of Pods and for creation of checkpoints.

Node Controller This controller watches node resources and is notified if the state of the nodes changes. It keeps an internal list of nodes and also adds a label to the node to identify them by their internal cluster IP. Although this information is available in other parts of the node data structure, only labels can be used to select a specific node for scheduling. These labels are later used by the Liveness Handler to select the appropriate node for scheduling Pods.

Agent Controller This controller watches DaemonSets. Filtering of the correct DaemonSet is done manually by the controller As the Agents are deployed as a DaemonSet, the controller listenes to changes in this DaemonSet and records changes in the available Agents.

C/R Controller This controller watches Deployments and filters them based on a well known annotation. This annotation called `cr_mode` is identifying the Deployments that should be monitored. There are also additional configuration information provided by the annotations of the Deployment. The interval for checkpointing is supplied by the `cr_interval` annotation. Every annotation is prefixed by `hitachienergy.com/`. The C/R Controller keeps an internal list of monitored Deployments and notifies the Checkpoint Handler and the Liveness Handler of new Deployments.

Pod Controller Finally, there is a Pod controller that listens for changes in Pods. This controller responsible for keeping track of all the Pods belonging to the Deployments configured for checkpointing. The Pod Controller will also already determine the target node for recovery using the node information provided by the Node Controller.

Checkpoint Handler The Checkpoint Handler starts a thread for every monitored Deployment. It uses the information provided by the Pod Controller, Node Controller and C/R Controller to checkpoint the Pods of the Deployment. The C/R Controller provides the Checkpoint Handler with the configured interval. The Pod Controller supplies the information about Pods belonging to the Deployment. The Node Controller keeps a list of the Nodes. From the Node list the Checkpoint Handler gets the information about how to reach Kubelet. The *ContainerCheckpoint* feature flag [3] enables an HTTP API on Kubelet to create checkpoints using CRIU.

After the checkpoint is created the Checkpoint handler will instruct the Agent running on the node of the checkpointed Pod to transform the checkpoint into an OCI image. Subsequently the Agent gets instructed to transfer the OCI image to the predetermined target node for recovery.

Liveness Handler The Liveness Handler also starts a thread for every monitored Deployment. It uses the information gathered from the C/R controller to execute liveness probes. The liveness of Pods is checked in the Liveness Handler as a means of detecting Pod failures faster than Kubernetes can. In the standard Kubernetes architecture liveness probes are executed by Kubelet on the corresponding node. If a node goes down there is a fairly generous timeout of 40 seconds (configured by `node-monitor-grace-period` [39]) until the node is considered NotReady. This grace period cannot be reduced arbitrarily as this can lead to an unstable system with continuous and quick state changes of the nodes. After that the standard Pod toleration timeout of 5 minutes will start [35]. After both timeouts are exceeded the Pods are restarted on other Nodes. Even if liveness probe intervals are configured shorter, the restart for Pods only happens after Pods are evicted from the unreachable node. To mitigate this long detection of the Pod failure a manual liveness probe is fired by the Liveness Handler. Liveness probes are HTTP requests to the

configured endpoint. The liveness probe is considered failed if either the request fails or if the status code is outside the range 200-299. If three consecutive liveness probes fail, the Liveness Handler will initiate recovery. Recovery is done by deleting the old Pod and creating a new Pod on the target node with the checkpoint image as the container image. The CRE of the target node will detect that the provided image is a checkpoint image and restore the container using CRIU.

Agent

As described earlier the Agents are running as a DaemonSet, which starts one instance of the Agent on every node (see Section 3.1.2). A checkpoint generated by Kubelet can not be restored as is and needs to be converted into an image readable by the underlying CRE. The Agent is responsible for creating these checkpoint images in OCI format from the checkpoint archives generated by Kubelet.

The Agent also features a file transfer capability which is used to transfer the built image to other Agents. Transferred images are inserted into the corresponding image store so that the CRE can create containers from these images.

4.1.4 Limitations

Our C/R Operator assumes that Pods have exactly one container, i.e., it cannot checkpoint Pods with more than one container. Since it is considered best practice to have one container per Pod and Pods with more than one container are rare in practice, this limitation is acceptable. Similarly, the Kubelet API does not support checkpointing Pods with multiple containers. Although it is possible to checkpoint a single container from a Pod with multiple containers.

4.2 Pull State Operator

The Pull State Operator was developed as a parallel fault tolerance approach to the C/R Operator. This allows us to compare the performance of both approaches and use the Pull State Operator as a baseline.

This section describes the design principles (Section 4.2.1), an overview (Section 4.2.2) and a description of the controllers (Section 4.2.3) of the Pull State Operator. It also presents the configuration of the Pull State Operator in Section 4.2.4.

4.2.1 Design principles

We evaluated two approaches for the Pull State Operator considering it's usage as a baseline. First, we evaluated a push based design in which the application can write the application state to the operator periodically or after every state change. In case of a

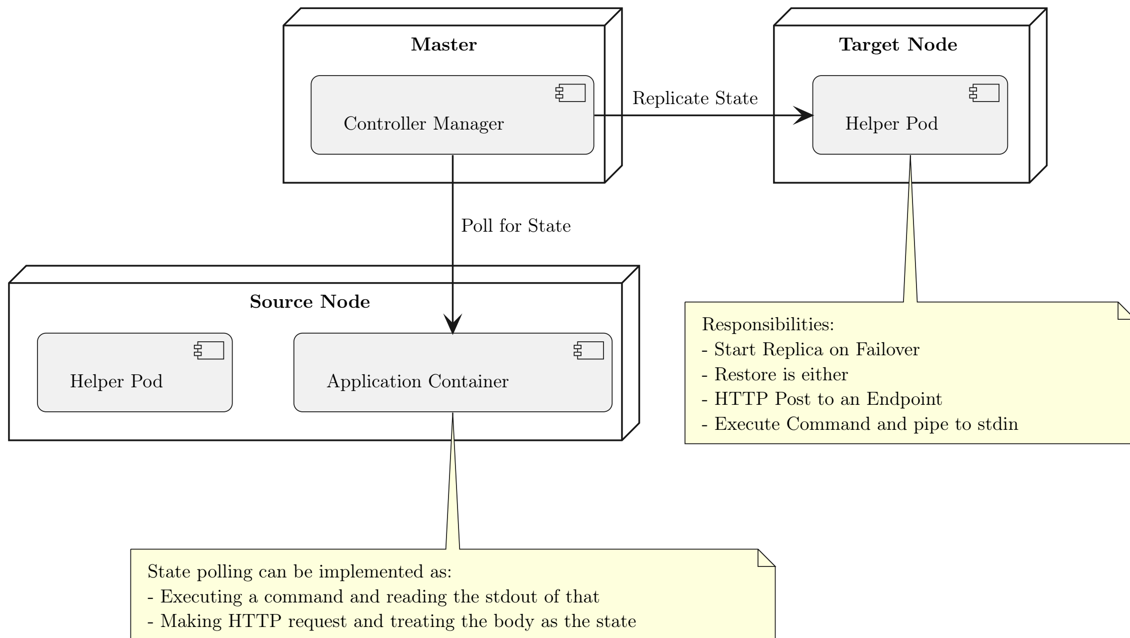

Figure 4.2: Architecture Overview: Pull State Operator

fault, the application would be restarted by Kubernetes and request the last state from the operator and continue. Such a push based approach is similar to a replicated database because the application has full control over the data flow. However it requires the application to be familiar with all the details on how to connect to the operator and how to store data with the operator.

Second a pull based design, in which the application provides an API for the operator to get and restore the state. The operator periodically pulls the state from the application and transfers it to the target node. In case of a fault, the application is started on the target node and the operator calls the restore API to restore the state. This approach is similar to the builtin probes that Kubernetes uses to determine readiness or liveness of the application.

We decided to use the pull based design for two main reasons:

- The design is closer to the C/R approach for stateful failover and is therefore easier to compare.
- Kubernetes already has the concept of probes so adding another probe is more aligned with existing functionality.

4.2.2 Overview

Figure 4.2 shows the architecture overview of the Pull State Operator. The operator consists of two components: the *Controller Manager* and the *Helper Pods*.

The Controller Manager is responsible for getting the state from monitored applications, selecting target nodes and coordinating the Helper Pods. The Helper Pods are responsible for holding the state on the target nodes and restoring the state in case of failover. The Helper Pods are deployed as a DaemonSet. Therefore, there is always one Helper Pod running per node.

In normal operation the Controller Manager requests the state from monitored Pods in a configurable interval. This state is distributed to the corresponding Helper Pods. Should a fault occur, Kubernetes restarts the Pod. The Controller Manager observes new Pod arrivals and sends a signal to the helper Pod to restore the state whenever a new Pod has arrived. This only happens if the Pod is restarted because of a Fault. New Pods for a Deployment can also arrive if a Deployment is scaled up. For example if the user requests a higher number of replicas. Such Pods are ignored by the Pull State Operator because the new Pod is not replacing a faulty Pod.

4.2.3 Controllers

The Controller Manager contains and manages multiple controllers and three handlers. Every controller in the Controller Manager is responsible for watching one resource type and reconciling it.

Helper Controller The Helper Controller listens for changes in DaemonSet resources and keeps the internal list of the Helper Pods up to date.

Node Controller The Node Controller adds a label to the node to identify them by their internal cluster IP. Although this information is available in other parts of the node data structure, only labels can be used to select a specific node for scheduling. These labels are later used by the Restore Handler (See Section 5.1) to select the appropriate node for scheduling Pods.

Pod Controller The Pod Controller listens for changes in Pods. This controller manages a Pod list for every monitored Deployment. Changes in the Pod metadata are captured and the internal list is kept up to date with the state of the Pods in the cluster. The Pod Controller also generates deleted Pod every time a Pod is deleted and a new Pod event every time a Pod is created. These events are passed to the Restore Handler.

Pull State Controller This controller listens for changes in Deployments. With operator specific annotations Deployments can configure parameters for the state pulling.

Liveness Handler The Liveness Handler is similar to the Liveness Handler in the C/R Operator. To mitigate the long detection of Pod failures in Kubernetes as described in

option	purpose
ps_mode	If this option is set to http-sf, it identifies the Deployment as monitored by the Pull State Operator.
ps_interval	The time interval in seconds for pulling the state.
ps_port	The port inside the container.
ps_path	The path of the state request. A GET request is used to retrieve the state. A POST request to this endpoint is used to restore the state.

Table 4.1: Pull State Operator Configuration options

Section 4.1.3 a manual liveness probe is fired by the Liveness Handler. Liveness probes are HTTP requests to the configured endpoint. The liveness probe is considered failed if either the request fails or if the status code is outside the range 200-299. If three consecutive liveness probes fail, the Liveness Handler will initiate recovery. Recovery is done by deleting the old Pod and creating a new Pod on the target node.

State Handler The State Handler uses the information gathered by the Pull State Controller and Pod Controller to request the state in the configured interval for the Deployments. For every Deployment a thread is started. This thread gets the state from all Pods of the Deployment and distributes it to all Helper Pods.

Restore Handler The Restore Handler runs in a separate thread and reacts to deleted Pod and new Pod events. The events are combined to detect Pods that need to be restored. If a Pod that needs to be restored is detected, the Restore Handler instructs the Helper Pod that is running on the node of the Pod to restore the state over the configured endpoint.

4.2.4 Configuration

Deployments can specify the options specified in Table 4.1 as annotations to influence the behaviour of the operator. As with the C/R Operator these configuration options are prefixed by `hitachienergy.com/`. The annotations are read by the Pull State Controller and are made available to the other parts of the operator as indicated in the Sections before.

Chapter 5

Implementation

This Chapter is divided in two Sections. Section 5.1 shows the implementation details of the Pull State Operator. The next Section (Section 5.2) describes the implementation details of the C/R Operator. Section 5.3 discusses some obstacles in the implementation process and also describes an approach to debugging the communication between runc and CRIU.

5.1 Pull State Operator

We used the Operator Software Development Kit (SDK) [59] for the implementation of the Pull State Operator. The Operator SDK is a SDK that helps bootstrapping operators with all the necessary components and Kubernetes resources. First, the Operator SDK creates a ServiceAccountRole, ServiceAccountRoleBinding, ConfigMap and Deployment for the operator. All these resources are needed for the Controller Manager to be deployed and connected to the Kubernetes API-Server.

While many operators deploy custom resources with CRDs, the Pull State Operator does not require custom resources but only adds metadata to Deployments. This metadata is used as the source of all necessary information. The following paragraphs detail the core processes of the Operator.

Deployment Registration Process The deployment registration process is initiated when a new Deployment is detected. The Pull State Controller is registered as a listener for Deployment changes at the API server. Therefore, the API server notifies the Pull State Controller of new Deployments. If a Deployment has the `ps_mode` annotation with correct values, it will be registered with the operator. The operator keeps a list of registered Deployments and their respective configuration.

Listing 5.1 shows an excerpt of a Deployment. The displayed Deployment is configured with an interval of 10 seconds, the endpoint to retrieve and restore state is `/state` and

the port is 8080. The liveness probe is configured for endpoint `/health` on the same port. This liveness probe configuration is used by Kubernetes and our Liveness Handler.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: the-count-deployment
  annotations:
    hitachienergy.com/ps_mode: 'http-sf'
    hitachienergy.com/ps_interval: '10'
    hitachienergy.com/ps_path: '/state'
    hitachienergy.com/ps_port: '8080'
spec:
...
  template:
  ...
    spec:
      containers:
        - name: the-count
          image: the-count:latest
          livenessProbe:
            httpGet:
              path: /health
              port: 8080
            initialDelaySeconds: 3
            periodSeconds: 1
```

Listing 5.1: Example Deployment

For every new Deployment the State Handler is notified and starts a new goroutine. The state handler then starts requesting the state periodically in the configured interval. The state of the Pod is replicated and sent to every Helper Pod over gRPC[1].

If `http-sf` is configured as the `ps_mode` the Liveness Handler is also notified of the new Deployment and also starts a new goroutine for the Deployment. The Liveness Handler uses the standard liveness probe configuration of the Pod and executes the same liveness requests as Kubelet. This is done to detect node failures in addition to Pod failures.

The standard mechanism of detecting node failures in Kubernetes is deliberately slow to mitigate continuous and quick node readiness state changes in poor network connectivity scenarios. Standard liveness probes rely on Kubelet to report Pod failures to the API-Server and are therefore not detected, if the node fails. The Pull State Operator also supports using the standard Kubernetes node failure detection with `ps_mode` set to `http`. However this is not further discussed, as it is slow.

In `http-sf` mode the Liveness Handler sends liveness probes in addittion to Kubelet. Assuming the workload and Controller Manager are running on different nodes, the liveness

[1] https://grpc.io/

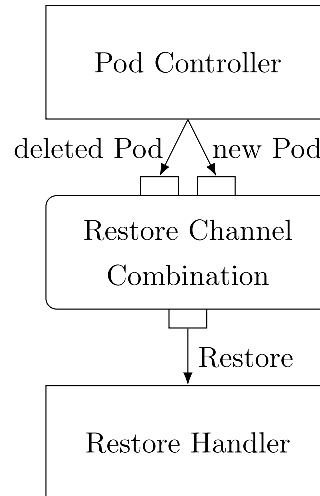

Figure 5.1: Restore Channel Combination

probes will fail if a node failure occurs. Doing this enables the Controller Manager to detect failures faster than the standard Kubernetes fault detection mechanisms.

Restore Process The restore process is triggered by new Pod and deleted Pod events. Figure 5.1 shows the flow of events from the Pod Controller to the Restore Handler. Both event types have their own channel per Deployment and are generated by the Pod Controller. The channels are created lazily on first use. A restore channel combines both the new Pod channel and the deleted Pod channel into one restore channel as shown in the Figure. The combination takes the first new Pod event after a deleted Pod event and combines them into a restore event.

The Restore Handler runs in a goroutine and continuously listens for restore events from the restore channel. If a restore appears the restore event is sent to the correct Helper Pod, which in turn executes the actual restore request.

In http-sf mode the Liveness Handler will delete the old Pod and create a new Pod as soon as the fault is detected. The new Pod is currently scheduled randomly on the next available node. At this point in time Kubernetes has not detected the failing node yet and if the scheduling is left to Kubernetes, there is a chance that the new Pod will be scheduled on the failing node. This necessitates the use of a node selector to guarantee the new Pod is not scheduled on the failing node. The node selector is set to match the `internal-ip` label added to Nodes by the Node Controller to schedule the new Pod on the correct node.

Creating new Pods The ability to create a new Pod for an existing ReplicaSet is not provided by Kubernetes, but the operator needs this ability to function correctly. It is not enough to delete Pods, because the ReplicaSet will try to create the Pod on the same node a few times, before trying to schedule the Pod on another node. The operator needs to

guarantee that new Pods are scheduled on functioning nodes. To achieve this the operator first deletes the Pod and then creates a new Pod with the correct labels for the ReplicaSet. This prevents the ReplicaSet from creating a new Pod itself.

5.2 C/R Operator

The C/R Operator is similar in design to the Pull State Operator. There are two major differences between the Pull State Operator and the C/R Operator. First, the way state needs to be handled and second how the restore works.

Checkpointing process The checkpointing process is performed by the Checkpoint Handler as described in Section 4.1.3. From the point of view of the operator checkpointing is done by first requesting the checkpoint from Kubelet and then transforming the checkpoint into an OCI image. As of Kubernetes version 1.25, the checkpointing feature is in alpha stage and can be enabled by the `ContainerCheckpoint` feature gate of Kubelet. A simple HTTP request is required to create a checkpoint. Kubelet then requests checkpointing of the container from the CRE through the CRI. CRI-O then receives this checkpoint request and gathers metadata and passes the checkpoint request to runc. Runc in turn calls CRIU to perform the checkpoint as described in Section 5.3.2.

CRIU now needs to first stop the process using ptrace system calls, dump all the information contained in the various namespaces and finally dump all the memory pages from the process. The CRIU image files and the metadata added by CRI-O are combined into an tar archive and are written to disk.

This tar archive is then processed by the Agent into an OCI image. This transformation adds some metadata to the checkpoint and enables it to be imported into the CRE. The OCI image format for checkpoints is not yet standardized and is therefore specific to the used CRE. For CRI-O it is only necessary to add the standard manifest, index, config and a layer containing the compressed archive. There also needs to be a special annotation called `io.kubernetes.cri-o.annotations.checkpoint.name` in the manifest to mark the image as a checkpoint image.

Restore process Restoring a checkpoint is done by starting a Pod with the OCI image generated from the checkpoint. When comparing the Pod creation process between a normal Pod and a checkpointed Pod is at first largely the same. At the time the container with the checkpoint image is created CRI-O will identify the checkpoint based on the added annotation. For restoring to work CRI-O needs to first setup all the container mounts correctly. If there is any difference in the mounts, restoring will fail. This can be the case when e.g. the filesystem on source and target system is different. After setting up the mounts CRI-O delegates the actual restore to runc. Runc takes the provided mounts,

sets up all the namespaces and starts CRIU for restore. CRIU then starts transforming itself into the checkpointed process, restoring all namespaces, memory and processor state.

To restore the process successfully everything needs to be exactly the same as while checkpointing. The exact same image needs to be available for restore to work. A Bug in CRI-O is currently a missing check of the image version. CRI-O matches the image name and the tag to restore containers. This is a problem if two different versions of the same image exist on the source and target node. Another problem are different bundle paths of Pods with and without infrastructure containers. Infrastructure containers are used by CRI-O to bind namespaces that need a running process for them to be retained. If such namespaces are not needed for a Pod the infrastructure container is not created. The information about the existence of the infrastructure container is not yet recorded in the checkpoint metadata. As the support for dropping infrastructure containers is not stable yet, it can be disabled in the configuration. If the configuration of source and target node is different restoration will fail.

Another problem is the lacking support for cgroup namespaces in CRIU in conjunction with cgroups version 2. Cgroups are bound to the Pod instead of the container, so CRIU needs to map the existing cgroups to the new cgroups of the new Pod. For this task runc provides a mapping from the old to the new Pod cgroups to CRIU. With cgroups version 2 a new cgroups namespace feature is introduced to isolate cgroups. This feature removes the need to provide the mapping as the restored cgroups are isolated by the namespace. CRIU does not support these namespaces yet and restores the cgroups without a namespace. This means with cgroups version 2 the Pod has two sets of cgroups attached to it after restore. Kubelet is running a garbage collection process periodically to remove orphaned cgroups. As the restored cgroups have no attached Pod the Kubelet removes them as part of this garbage collection, killing every process running inside the cgroup. To mitigate this issue cgroup support in CRIU needs to be disabled. This has the disadvantage of dropping any cgroups created inside the container. As creating cgroups is rare for application containers this feature can safely be disabled.

5.3 Overcoming Implementation Obstacles

This Section first details an obstacle with the implementation concerning clusters with heterogeneous CPU architectures in Section 5.3.1. Then it describes the debugging process for intercepting the messages between runc and CRIU in Section 5.3.2.

5.3.1 CPU Architecture

An important factor while restoring processes from one machine on another machine is the CPU architecture. Incompatibilities between the source and target system often lead to illegal instructions being executed upon restoring a process. Most applications that

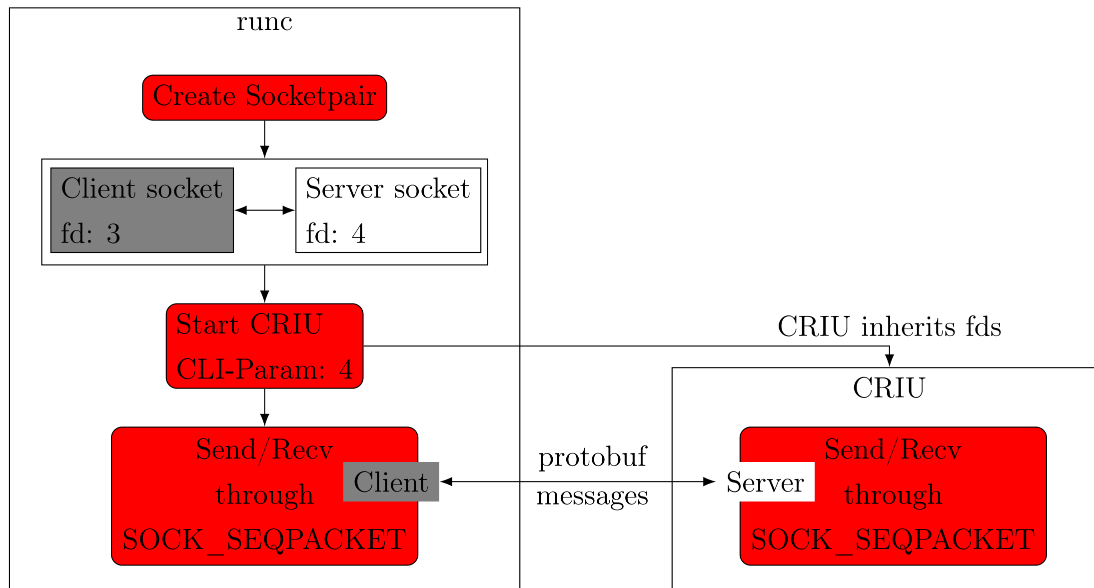

Figure 5.2: Communication between runc <-> CRIU

support different CPU flags scan these at the start of the program and do not expect the hardware to change while the process is running. So while checkpointing the exact flags of the CPU need to be recorded. The restore can only happen on systems with the same architecture and a subset of the flags.

One possible solution is to implement a signal handler for SIGILL and perform a rescan of CPU flags and to recover to a save point in program execution. This can only be implemented in application code. Therefore, it would not satisfy the application transparency of the C/R approach.

The easiest and most efficient solution is to require the target machine to have a superset of the CPU flags of the source machine.

5.3.2 Debugging the communication between runc and CRIU

One major aspect that is lacking in the current implementation of the C/R features in Kubernetes is error reporting. Being able to look at the communication between runc and CRIU enables more debugging options as one can better analyze the cause of CRIU failures with the provided options.

Communication between runc and CRIU works through raw RPC messages passed through a `UNIX_SEQPACKET` socket. To start the communication runc creates a socketpair and opens all the necessary files, that CRIU needs. Thereafter, runc forks and execs itself and gives CRIU the file descriptor number of one end of the socket pair, through which runc and CRIU communicate. This works because open file descriptors are inherited to child processes unless the file is opened with the `FD_CLOEXEC` flag or later marked as non-inheritable.

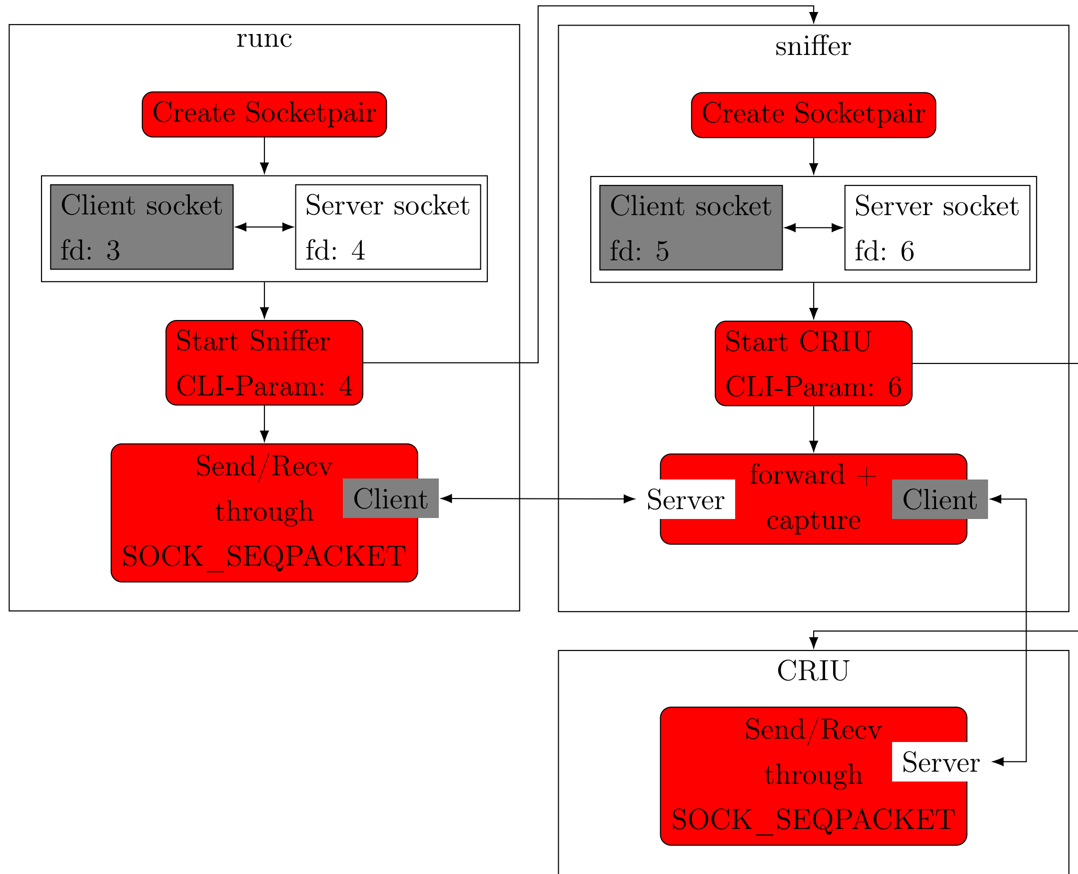

Figure 5.3: Communication between runc <-> CRIU

After establishing this communication link protobuf encoded messages are passed between runc and CRIU. Inside of these messages references to open file descriptors are made that designate for example the directory containing a checkpoint. Sniffing on this process is achieved by replacing the CRIU executable with a sniffer program that opens a socket pair itself. The sniffer then receives messages from the runc socket, captures them into a file and forwards them through its own socket pair to CRIU. Other open file descriptors from runc must be replicated at the exact same file descriptor number to enable CRIU to access the correct files. Conceptually, this is relatively simple as the linux kernel increments the file descriptors for every open call that is made with file descriptors 0, 1 and 2 being reserved for `stdin`, `stdout` and `stderr` respectively. To not block the lower file descriptor numbers the sniffer first opens `/dev/null` 100 times and then proceeds to open the socket pair, the capture files and everything else it needs. After this step is complete the 100 dummy file descriptors are closed and can be reused for replicating open files in runc.

Chapter 6

Evaluation

This chapter describes the experimental setup that was used to evaluate the proposed Kubernetes operator, as well as an analysis of the obtained performance results. Section 6.1 describes the experimental setup including the used applications, the cluster hardware and the procedure of the failover experiments. Section 6.2 presents the performed experiments measuring overhead in regards to CPU usage, memory usage, disk usage, network traffic and service disruption. The checkpoint performance for the different applications is shown in Section 6.3. Next, Section 6.4 presents the results of the failover experiments. Section 6.5 compares the recovery times between the Pull State and the C/R Operator and Section 6.6 discusses the state quality of both methods. At last, Section 6.7 adds a discussion of the presented results.

6.1 Experimental Setup

This section describes the setup for the performed experiments. In Section 6.1.1 the used applications for the experiments are presented. Section 6.1.2 describes the hardware setup of the cluster. Section 6.1.3 gives an overview over the gathered metrics and performed experiments.

6.1.1 Applications

There are mainly two applications used for the experiments, the count and Redis[1].

The count is a Nodejs[2] application using the expressjs[3] framework. The count has internal state in form of a counter which is counted up every $100ms$. This state can be requested using a GET request with path `/state` and modified using a POST request with

[1] https://redis.com/
[2] https://nodejs.org/en/
[3] https://expressjs.com/de/

Name	CPU	AVX	RAM
Source node	Intel Xeon E5-2660 v2	AVX	64GB
Target node	Intel Xeon E5-2660 v3	AVX2	64GB

Table 6.1: Experimental machines

the same path. The count also includes a `/health` endpoint which is used to check the liveness of the application.

Redis is a lightweight key-value store. In our experiments we used Redis without any data, Redis with $1MB$, $10MB$ and $100MB$ of random data inserted. The Redis variants are named *redis*, *redis-1m*, *redis-10m* and *redis-100m* respectively.

6.1.2 Cluster

The experiments were performed on a two node cluster with two different physical machines. Table 6.1 shows the CPU and RAM of both nodes. Both machines are connected over a 100 Mbit/s Fast Ethernet switch.

All of the Kubernetes control plane components as well as the operator are running on the target node. The Kubernetes version is 1.25.4, which was installed through Kubeadm with the *ContainerCheckpoint* feature flag enabled. A modified version of CRI-O is used as the container runtime engine for the nodes. The modified version of CRI-O includes a small bugfix and disables the cleanup of the CRIU logs. At the lowest level, runc is used to run the containers. Container networking is setup using flannel and the flannel CNI plugin.

6.1.3 Experiments

We are interested in multiple metrics of the implemented Pull State and C/R Operator. First, we are interested in the overhead both operators incur on the system. The overhead is measured in terms of overall CPU, memory, network and disk usage. Furthermore, the service disruption is measured through requests round trip times. Next, we are looking into checkpoint performance in terms of process freezing time and dumped memory.

In case of failover, we are interested in the actual service disruption as measured from outside the cluster, recovery time and the state quality. Service disruption is measured in terms of request round trip times. Recovery time is the time between fault injection and the return to normal operation. With state quality we describe how much state is lost due to the fault. As both operators can not guarantee perfect state recovery, we are interested in the recency of the recovered state.

Failover experiments are always conducted in the following way. The monitored application is started on the source node and the operator is allowed to register and checkpoint

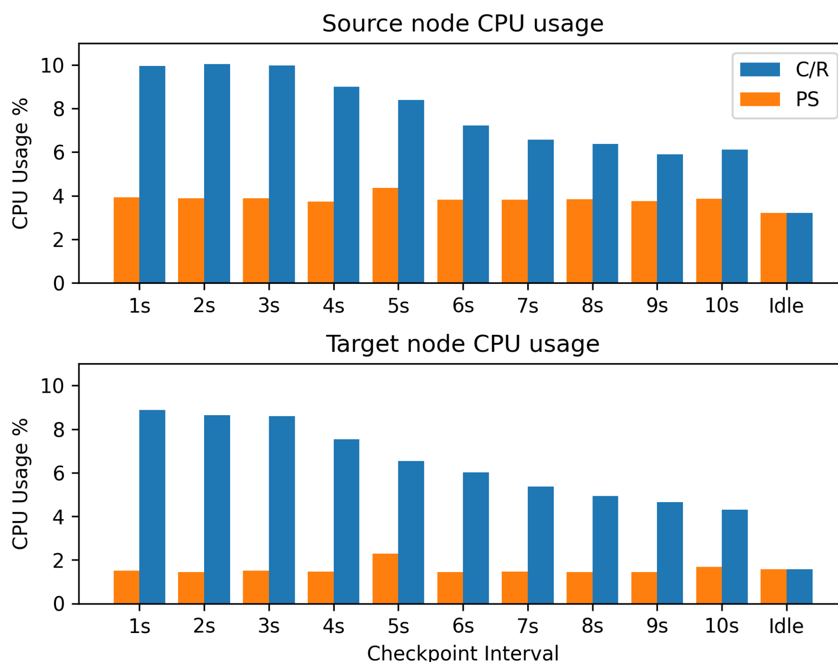

Figure 6.1: Overhead CPU statistics

the new Pod. The source node is disconnected from the cluster through an iptables rule. After disconnection, the proposed operator detects the fault and restores the Pod on the target node. Checkpoints of the count application on the target node cannot be restored on the source node, as the target node supports AVX2, but the source node only supports AVX (see Section 5.3.1). If AVX2 support is available openssl uses these instructions to speed up hash calculations. Restoring the application leads to illegal instructions being executed.

6.2 Overhead

Overhead is the overall impact on the system by using both operators. This includes the impact on the major resources of the system such as CPU, memory, network and disk. Overhead is also a measure of the impact on the monitored process. Such as measuring the impact on the service provided by the application.

We use Prometheus [61] and node exporter [60] to measure the overall impact on the system. These programs help monitoring the overall system load. Prometheus is set to gather performance data every second as recommended in their documentation as the lowest possible value [5]. Each experiment is run over 30 minutes and gathers data in that amount of time.

In Figure 6.1 the average CPU load is shown for different checkpoint intervals with the C/R Operator and the Pull State Operator. Measurements are performed using the

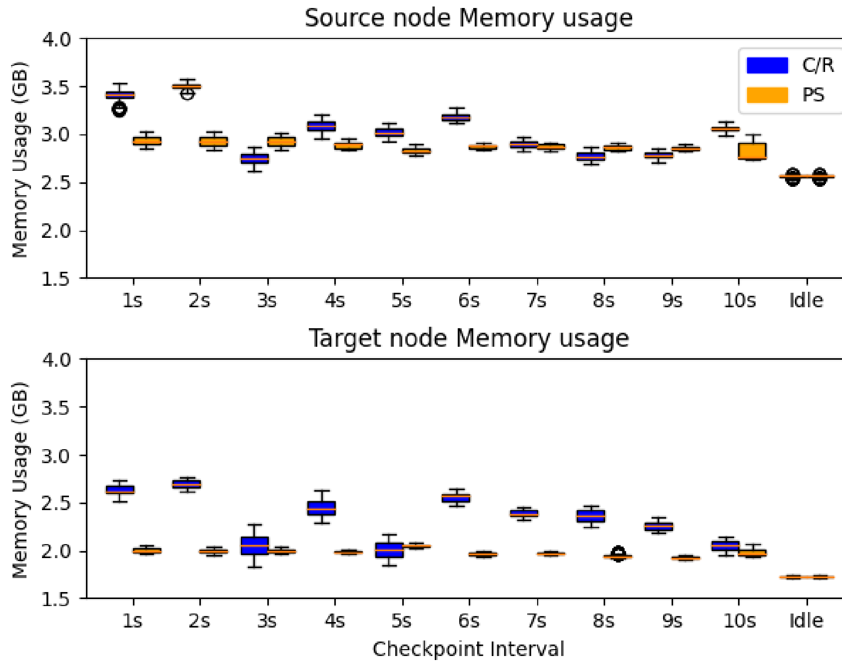

Figure 6.2: Overhead Memory statistics

example count application running on the source node. The operator is running on the target node along with all the control plane services for Kubernetes. The graphic shows the average CPU usage over every core while varying the checkpointing interval from one checkpoint every second to one checkpoint every ten seconds. As is apparent from the data an increase in checkpoint interval or decrease of checkpoint frequency leads to lower CPU consumption on source and target node. It is also clear that the Pull State Operator is more resource efficient than the C/R Operator. This result is expected as the Pull State Operator only needs to request a single value from the count. The checkpointing process for the C/R Operator is vastly more complex as the whole memory and all process resources need to be written to disk and converted to an OCI image.

Figure 6.2 shows the same measurements for memory consumption. In this case, we do not observe any significant correlation between memory consumption and checkpoint interval. From an analytical standpoint this is expected, as the checkpointing with the C/R Operator only temporarily consumes more memory to process the data. The Pull State Operator also holds one instance of the state in memory.

Statistics about the network consumption are also important, as the checkpoints need to be continuously transferred to the target node. Figure 6.3 shows the transmitted (TX) and received (RX) data for the source and target node. The y-axis of the plot is in a logarithmic scale. From the graphic it is clear that RX from the target and TX from the source are nearly identical in case of the C/R Operator. On the other hand, the data from the Pull State Operator shows inflated values for the transmitted bytes. These high values

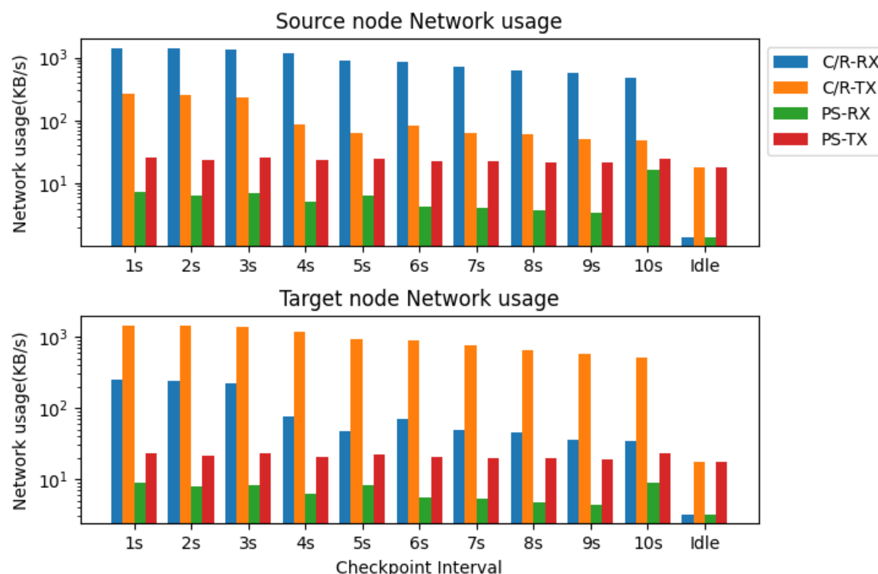

Figure 6.3: Overhead Network statistics

for the transmitted data are easily explained by the data gathering mechanism. The Pull State Operator uses so little data transmission that the data is dominated by the gathering of statistics.

The most severe increase in traffic is by 1410 KB/s comparing the transmission and reception from the source and target node with 1s checkpoint interval. This corresponds to a low amount of traffic compared to the overall link capacitiy of 100 Mbit/s. TX from source and RX from target being the same is to be expected, as we are transferring the checkpoints from one node to the other and no other significant network traffic is happening on the machines.

Figure 6.4 shows the overall disk writes and reads on both nodes. The y-axis is in logarithmic scale. The disk reads do not seem correlated at all with the experiments. Disk writes seem to be correlated similarly to the rest of the statistics, with lower checkpoint interval leading too higher disk writes.

Another aspect of overhead for the system is the service interruption incured on the application. This is measured using a client which makes requests as fast as possible and records the answer time. The measurements are performed over five minutes and result in the graph shown in Figure 6.5. Figure 6.5-a shows measurements of roundtrip time for requests against the count application. Figure 6.5-b shows the duration distribution of the delayed requests. The experiments were performed using a 10s interval time for checkpointing. The figure shows the expected behaviour. Every 10s there is a requests that gets delayed by the checkpointing time. The duration of the checkpointing is roughly one second.

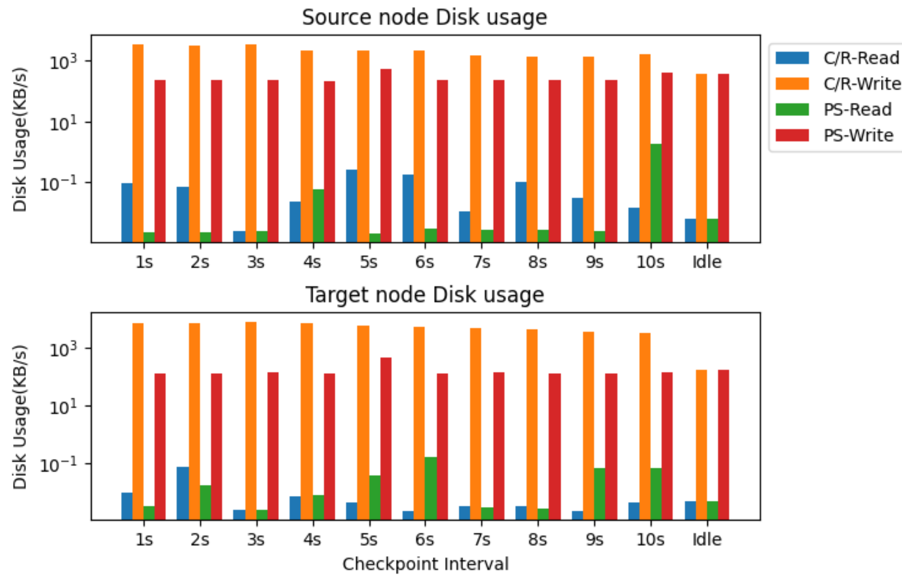

Figure 6.4: Overhead Disk statistics

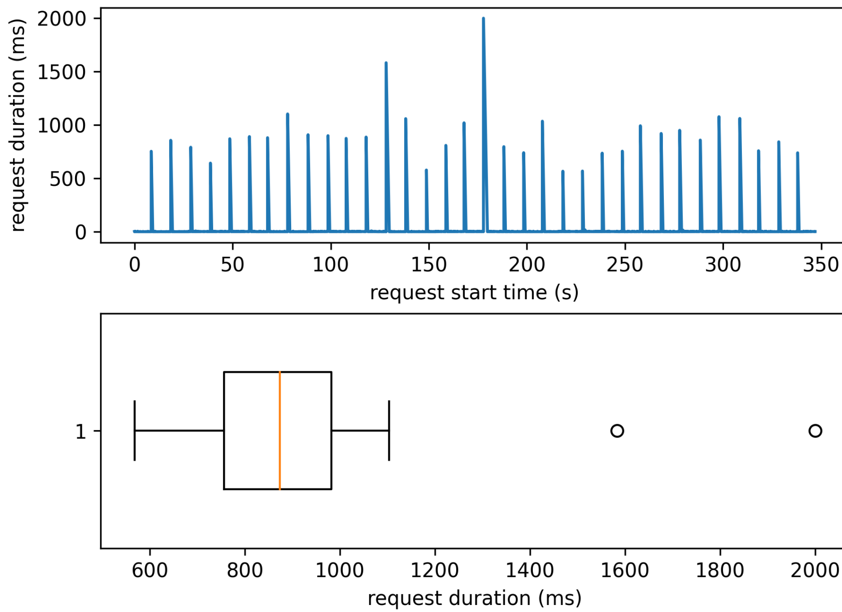

Figure 6.5: Request round trip times

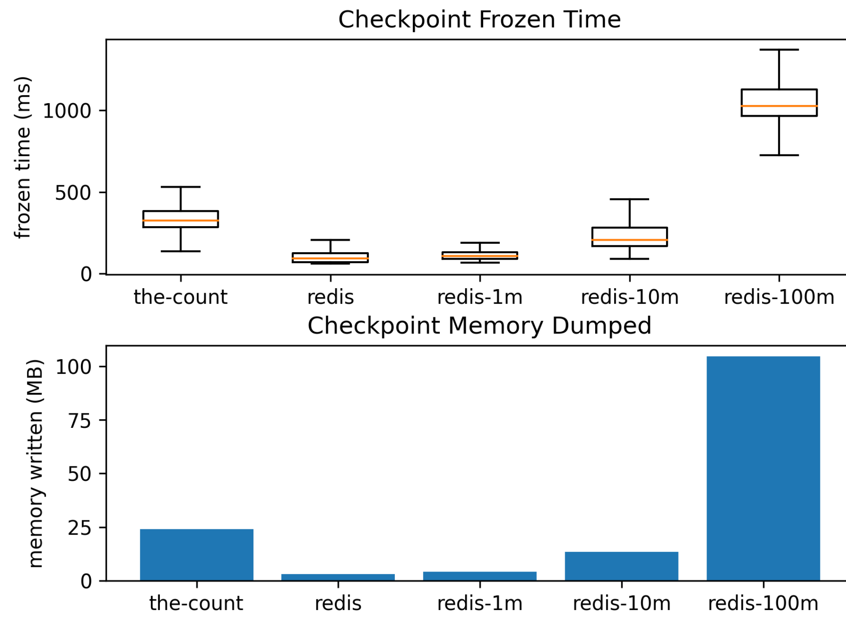

Figure 6.6: Checkpoint freezing time & memory dumped

6.3 Checkpoint performance

Checkpoint performance is the time it takes for the checkpoint to be created. Additionally CRIU reports the amount of memory written to disk while checkpointing.

Figure 6.6-a shows how long processes are frozen. The applications shown are the count application and our variants of redis. Figure 6.6-b shows the amount of memory dumped for the processes. The two figures show a correlation between frozen time during checkpointing and memory written to disk. The data is gathered over roughly 2000 data points for every application. As we can see checkpointing the count requires dumping roughly $25MB$ of memory, while the empty redis variant only requires approximately $1MB$. This is due to the high amount of memory used by Nodejs.

6.4 Failover

The durations corresponding to the different steps of the failover process are measured for the count example application, redis and the redis variants with 1MB, 10MB and 100MB of random data inserted.

Figure 6.7 shows a timeline of the failover process for the C/R Operator. Every section in the timeline designates a phase of the failover process. First, Pod scheduling and creation is executed by Kubernetes. In this phase, Kubernetes decides on which node the Pod should be scheduled and creates the Pod in CRI-O. The next phase shows how much time CRI-O

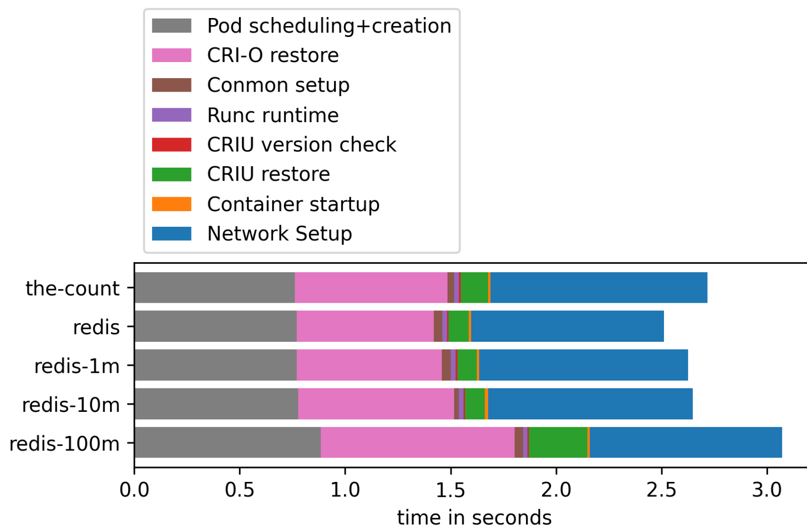

Figure 6.7: Failover timeline

needs to create all the resources for the restore and prepare everything conmon needs for the restore. Conmon setup is the time conmon is running until it calls runc in the next phase. runc first sets up the namespaces, mounts and any other resources the container needs and subsequently calls CRIU for a version check. After the confirmed version check, the real restore process with CRIU happens. This is also the time at which CRI-O reports the container as started. The last second of most of the restore operation is spent on network setup until the container is reachable again from the outside. In Figure 6.7, we omit a roughly 3 second long fault detection time for clarity. The fault detection time is similar for all applications and is comprised of 3 failed liveness probes executed every second.

We can see that the two phases growing the most with higher amount of memory dumped are the CRI-O restore and the CRIU restore time. This is probably due to the larger amount of data that needs to be copied for the restore.

From the outside, the failover appears like an outage of approximately 6 seconds. In Figure 6.8 a failover event is shown as seen from an external application that sends requests to the count application during a failover event. The graphs show the durations observed by such requests. The graphs shows the request durations to the the-count example application during a failover event. Smaller spikes indicate checkpoint events similar to Figure 6.5. In the large spike we see 6 failed requests that experienced the timeout of 1 second while requesting the application. After this time the application is available again

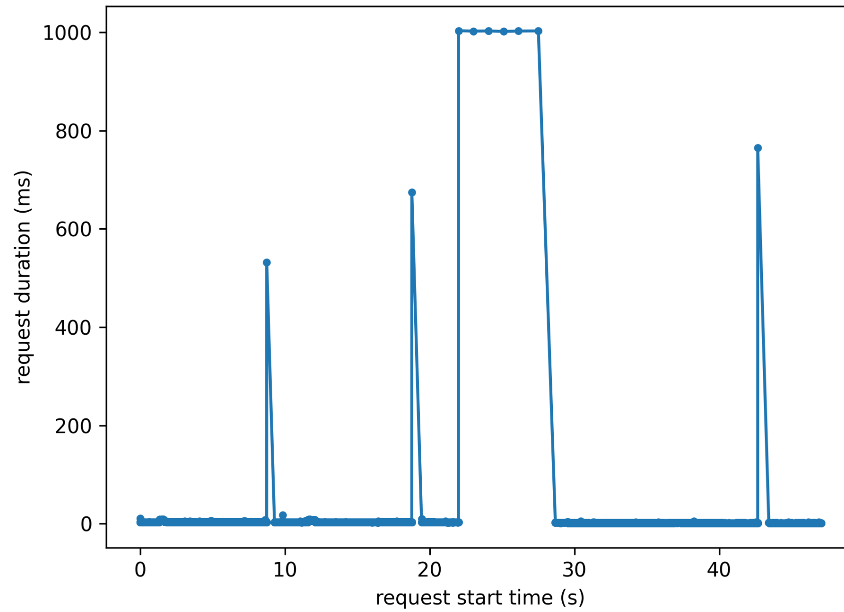

Figure 6.8: Requests to application while failover

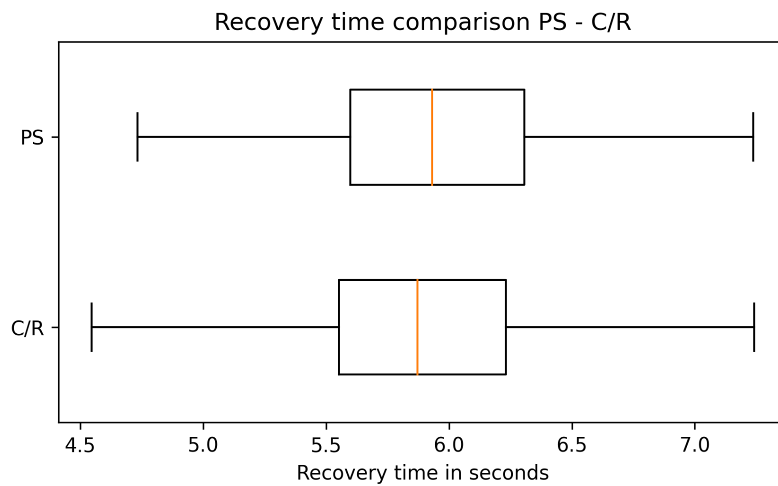

Figure 6.9: Comparisons of recovery times

and can be used normally. The experiments were performed over 100 runs and the graph shows the average times over all 100 runs.

6.5 Recovery time comparison

Figure 6.9 shows a comparison of the C/R Operator and the Pull State Operator in terms of recovery times. Recovery time is here measured as the time elapsed from fault injection

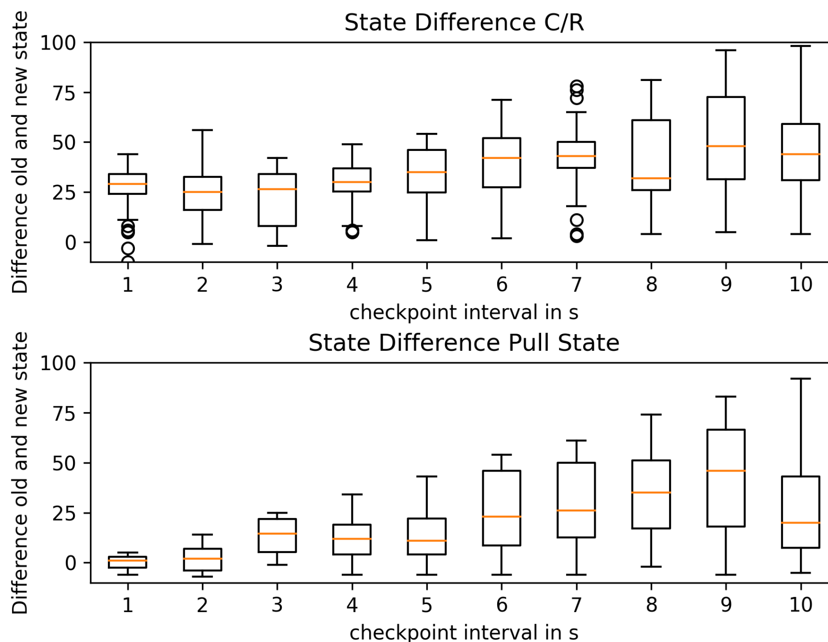

Figure 6.10: State quality of Pull State vs C/R Operator

to the time at which the application is reachable with recovered state. As can be seen in the figure the recovery times of both operators are fairly similar.

6.6 State Quality

Another metric to be considered to benchmark C/R for fault tolerance is the state quality. This refers to the amount of state lost during failover. Figure 6.10 shows the difference in state before and after failover using different checkpointing intervals I. The experiemnts are performed using the-count example application, which increments its count every $100ms$. The state lost during failover is therefore expected to be between 0 and $10 * I$. Values below zero are encountered, should the reconnection to network mentioned earlier take so long that the new count is already higher than the old count. The data for $1s$ and $2s$ interval checkpoints shows irregularities, because the checkpointing happens so frequently, that instabilities start to occur in the process. Otherwise, the trend in the data is as expected.

6.7 Discussion

In our work we design and implement a new approach for fault tolerance of stateful applications using C/R technology. This approach is integrated into the Kubernetes container orchestration framework in form of an operator and is using the latest features regarding

C/R. The implemented operators are evaluated based on their overhead and performance in a network failure scenario.

The analysis of the overhead of the presented C/R Operator shows no significant increase in regards to memory and disk usage. The CPU overhead shows an increase of CPU usage of 2-6%, which for mast clusters is negligible. The network overhead depends on the checkpoint size and interval as the checkpoint needs to be transferred to the target node. Depending on the cluster network setup and the amount of transferred data this can have a significant impact on the cluster. Another significant impact on the system is the service availability. Freezing the process for one second every 10 seconds means reduces the available processing time to 90%. Lower checkpointing intervals reduce he performance of the monitored application further.

The failover measurements indicate 4.5-7.5 seconds of recovery time after failover. Depending on application startup, recovery times for the C/R Operator can be vastly superior to recovery times of the Pull State Operator. The C/R Operator can skip application initialization on restore which can significantly speed up recovery time. A consideration for these measurements is that new Pod IPs were requested manually from Kubernetes to measure recovery time. If applications use Kubernetes Services to route requests to Pods, there will be intermittent failures. This is illustrated in Appendix A.1.

State quality for the Pull State Operator and the C/R Operator are similar and depend on the state saving interval. Both approaches are only viable for applications that do not need to recover exact state or will reach the same state after recovery. Similar to the redundancy approach this is only guaranteed for applications that are deterministic. Deterministic applications do not use any source of random data i.e. the current time, random numbers or timing differences in threads of the application to update their state. Another challenge in the context of state recovery is explicit or implicit state stored outside of the container, e.g. session tokens for authentication with external services.

A further consideration for using the C/R Operator is the requirement of homogeneous CPU architecture. Restore is only possible on nodes that implement a superset of the source node instruction set. In heterogeneous clusters this limits the available nodes for restore considerably.

An advantage of the C/R Operator when compared to the Pull State Operator is complete application transparency. No modification of the monitored application is required for the C/R Operator to work. The Pull State Operator needs an interface to pull the state and an interface for restoring the state. This is an advantage in case modification of the application is cost prohibitive or calculation of the internal state is computationally expensive.

Overall the C/R Operator can be considered for use in situations that satisfy the following conditions:

- not sensitive to intermittent service interruption
- do not require exact state recovery on failover
- executed in a homogeneous cluster
- cost prohibitive to modify

Another considerations is longer checkpointing times with higher memory consumption of the monitored application. On the other hand long startup times of applications are mitigated by the C/R Operator.

Chapter 7

Conclusion & Future Work

7.1 Conclusion

The thesis shows a promising new technology for fault tolerance with stateful applications in the Kubernetes container orchestration framework. Two approaches for fault tolerance were implemented and evaluated based on multiple metrics. The first approach is the Pull State Operator which pulls state in an configurable interval from the monitored application. The second approach is the C/R Operator which uses Checkpoint/Restore technology to implement fault tolerance. Both approaches have similar recovery times of 4.5-7.5 seconds from the fault till the application is available again. The C/R approach results in a higher performance overhead in CPU, network traffic and service disruption as the Pull State approach. The advantages of the C/R approach lie in the application transparency, meaning the application has no need to be adapted and ease of use as long as all prerequisites for usage of the technology are fulfilled.

7.2 Future Work

Future work includes multiple avenues for improvement of the technology.

First the standardization of the checkpoint image format for use with all CREs is an important step for maturity in the technology. Currently no standard image format exists for checkpoints. Every CRE implements their own format with slightly different metadata. The OCI community is in the process of specifying a standard for an image format for checkpoints. With this standard checkpoint images become interchangeable between CREs.

One improvement not explored in this work is the use of incremental checkpoints and tracking of changed memory pages. This approach could be used to improve checkpoint performance and lower network overhead. Incremental checkpointing allows CRIU to first take one full checkpoint and enable memory tracking. Every subsequent checkpoint saves

all memory pages that have changed since the last checkpoint instead of every memory page. For processes with substantial memory usage incremental checkpointing allows faster checkpointing because not all memory needs to be dumped. Incremental checkpointing also reduces the amount of transferred data between the nodes because each incremental checkpoint is smaller.

Another improvement would be faster fault detection in Kubernetes. The manual way to mark Pods as failed by the operators is not ideal and should be more closely integrated with Kubernetes. Currently the operators first recover the Pod and then transfer ownership of the Pod to the corresponding ReplicaSet. ReplicaSets are not meant to take ownership of Pods created by other controllers. Sometimes this leads to the ReplicaSet deleting the transferred Pod and replacing it with a new Pod. Also the integration with Services is not ideal at the moment. Using a Service to access the Pods leads to intermittent failures because Kubernetes needs to mark the failed Pod as faulty before the Service stops routing requests to the failed Pod. Improvements in this regard are left to future work.

Appendix A

Appendix

A.1 Failover with Kubernetes Service

Figure A.1 shows how the requests look in case of a failover with a Kubernetes service between the Pods and the requester. We can see intermittent failures until Kubernetes registeres the node as *NotReady* and drains all Pods from the failed node. The intermittent failures are caused by the load balancing performed by the Service. This problem not only exists with our approach, but is a general problem with replicated applications in Kubernetes.

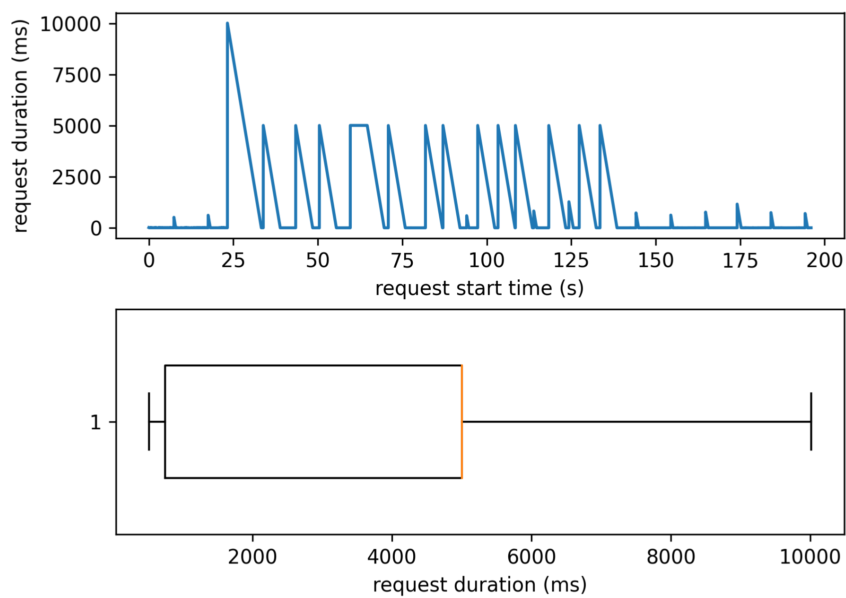

Figure A.1: Failover with a Kubernetes Service

List of Figures

3.1 Kubernetes Architecture [63] 10
3.2 Kubernetes Node Architecture 13
3.3 Layout of a OCI-Image 14

4.1 C/R Architecture overview 20
4.2 Architecture Overview: Pull State Operator 23

5.1 Restore Channel Combination 29
5.2 Communication between runc <-> CRIU 32
5.3 Communication between runc <-> CRIU 33

6.1 Overhead CPU statistics 37
6.2 Overhead Memory statistics 38
6.3 Overhead Network statistics 39
6.4 Overhead Disk statistics 40
6.5 Request round trip times 40
6.6 Checkpoint freezing time & memory dumped 41
6.7 Failover timeline 42
6.8 Requests to application while failover 43
6.9 Comparisons of recovery times 43
6.10 State quality of Pull State vs C/R Operator 44

A.1 Failover with a Kubernetes Service 49

Bibliography

[1] Daniel J Abadi. "Consistency Tradeoffs in Modern Distributed Database System Design". en. In: (), p. 6.

[2] Adrian Reber. *Minimal checkpointing support*. Sept. 10, 2021. URL: https://github.com/kubernetes/kubernetes/pull/104907.

[3] Adrian Reber. *Minimal checkpointing support*. Feb. 13, 2023. URL: https://github.com/kubernetes/kubernetes/pull/104907.

[4] Arif Ahmed et al. "Docker Container Deployment in Distributed Fog Infrastructures with Checkpoint/Restart". In: *2020 8th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)*. 2020 8th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud). Oxford, GB: IEEE, Aug. 2020, pp. 55–62. ISBN: 978-1-72811-035-6. DOI: 10.1109/MobileCloud48802.2020.00016. URL: https://ieeexplore.ieee.org/document/9126743/ (visited on 08/15/2022).

[5] Aliaksandr Valialkin. *What is the minimum scrape$_i$ntervalinPrometheus?*. Jan. 27, 2023. URL: https://stackoverflow.com/a/72704737/5625089.

[6] Jiajun Cao et al. "Checkpointing as a Service in Heterogeneous Cloud Environments". In: *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid). Shenzhen, China: IEEE, May 2015, pp. 61–70. ISBN: 978-1-4799-8006-2. DOI: 10.1109/CCGrid.2015.160. URL: http://ieeexplore.ieee.org/document/7152472/ (visited on 08/08/2022).

[7] Xiao Chen, Jian-Hui Jiang, and Qu Jiang. "A Method of Self-Adaptive Pre-Copy Container Checkpoint". In: *2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing (PRDC)*. 2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing (PRDC). Zhangjiajie, China: IEEE, Nov. 2015, pp. 290–300. ISBN: 978-1-4673-9376-8. DOI: 10.1109/PRDC.2015.11. URL: http://ieeexplore.ieee.org/document/7371873/ (visited on 08/15/2022).

[8] Cloud Native Computing Foundation. *CNCF Annual Survey 2022*. Feb. 13, 2023. URL: https://www.cncf.io/reports/cncf-annual-survey-2022/.

[9] CRIU authors. *CRIU*. Feb. 5, 2023. URL: https://criu.org/Main_Page.

[10] CRIU authors. *Freezing the tree*. Feb. 5, 2023. URL: https://criu.org/Freezing_the_tree.

[11] CRIU authors. *Memory dumping and restoring*. Feb. 5, 2023. URL: https://criu.org/Memory_dumping_and_restoring.

[12] CRIU authors. *Parasite Code*. Feb. 5, 2023. URL: https://criu.org/Parasite_code.

[13] CRIU authors. *TCP connection*. Feb. 5, 2023. URL: https://criu.org/TCP_connection.

[14] CRIU authors. *Upstream kernel commits*. Feb. 5, 2023. URL: https://criu.org/Upstream_kernel_commits.

[15] etcd authors. *etcd*. Feb. 5, 2023. URL: https://etcd.io/.

[16] Keerthana Govindaraj and Alexander Artemenko. "Container Live Migration for Latency Critical Industrial Applications on Edge Computing". In: *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA). Turin: IEEE, Sept. 2018, pp. 83–90. ISBN: 978-1-5386-7108-5. DOI: 10.1109/ETFA.2018.8502659. URL: https://ieeexplore.ieee.org/document/8502659/ (visited on 08/08/2022).

[17] Nicholas Gray et al. "A priori state synchronization for fast failover of stateful firewall VNFs". In: *2017 International Conference on Networked Systems (NetSys)*. 2017 International Conference on Networked Systems (NetSys). Gottingen: IEEE, Mar. 2017, pp. 1–6. ISBN: 978-1-5090-4394-1. DOI: 10.1109/NetSys.2017.7903964. URL: http://ieeexplore.ieee.org/document/7903964/ (visited on 08/10/2022).

[18] Michael Gundall et al. *Downtime Optimized Live Migration of Industrial Real-Time Control Services*. Mar. 24, 2022. arXiv: 2203.12935[cs]. URL: http://arxiv.org/abs/2203.12935 (visited on 08/12/2022).

[19] Cheng-Hao Huang and Che-Rung Lee. "Enhancing the Availability of Docker Swarm Using Checkpoint-and-Restore". In: *2017 14th International Symposium on Pervasive Systems, Algorithms and Networks & 2017 11th International Conference on Frontier of Computer Science and Technology & 2017 Third International Symposium of Creative Computing (ISPAN-FCST-ISCC)*. 2017 14th International Symposium on Pervasive Systems, Algorithms and Networks (ISPAN), 2017 11th International Conference on Frontiers of Computer Science and Technology (FCST), & 2017 Third International Symposium of Creative Computing (ISCC). Exeter: IEEE, June 2017, pp. 357–362. ISBN: 978-1-5386-0840-1. DOI: 10.1109/ISPAN-FCST-ISCC.2017.69. URL: http://ieeexplore.ieee.org/document/8121796/ (visited on 08/15/2022).

[20] Bjarne Johansson and Mats Ra. "Kubernetes Orchestration of High Availability Distributed Control Systems". In: (), p. 8.

[21] Paulo Souza Junior, Daniele Miorandi, and Guillaume Pierre. "Good Shepherds Care For Their Cattle: Seamless Pod Migration in Geo-Distributed Kubernetes". In: *2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC)*. 2022 IEEE 6th International Conference on Fog and Edge Computing (ICFEC). Messina, Italy: IEEE, May 2022, pp. 26–33. ISBN: 978-1-66549-524-0. DOI: 10.1109/ICFEC54809.2022.00011. URL: https://ieeexplore.ieee.org/document/9799256/ (visited on 08/15/2022).

[22] Sudarsun Kannan et al. "Optimizing Checkpoints Using NVM as Virtual Memory". In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 2013 IEEE International Symposium on Parallel & Distributed Processing (IPDPS). Cambridge, MA, USA: IEEE, May 2013, pp. 29–40. ISBN: 978-1-4673-6066-1 978-0-7695-4971-2. DOI: 10.1109/IPDPS.2013.69. URL: http://ieeexplore.ieee.org/document/6569798/ (visited on 08/15/2022).

[23] Kernel development Community. *Capabilites*. Feb. 5, 2023. URL: https://man7.org/linux/man-pages/man7/capabilities.7.html.

[24] Kernel development Community. *Control Groups*. Feb. 5, 2023. URL: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cgroups.html.

[25] Kernel development Community. *Control Groups v2*. Feb. 5, 2023. URL: https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v2.html.

[26] Kernel development Community. *seccomp*. Feb. 5, 2023. URL: https://man7.org/linux/man-pages/man2/seccomp.2.html.

[27] Kernel development community. Feb. 13, 2023. URL: https://man7.org/linux/man-pages/man7/time_namespaces.7.html.

[28] Kernel development community. *ipc_namespaces - overview of Linux IPC namespaces*. Feb. 13, 2023. URL: https://man7.org/linux/man-pages/man7/ipc_namespaces.7.html.

[29] Kernel development community. *mount_namespaces - overview of Linux mount namespaces*. Feb. 13, 2023. URL: https://man7.org/linux/man-pages/man7/mount_namespaces.7.html.

[30] Kernel development community. *namespaces - overview of Linux namespaces*. Feb. 13, 2023. URL: https://man7.org/linux/man-pages/man7/namespaces.7.html.

[31] Kernel development community. *network_namespaces - overview of Linux network namespaces*. Feb. 13, 2023. URL: https://man7.org/linux/man-pages/man7/network_namespaces.7.html.

[32] Kernel development community. *pid_namespaces - overview of Linux PID namespaces.* Feb. 13, 2023. URL: https://man7.org/linux/man-pages/man7/pid_namespaces.7.html.

[33] Kernel development community. *user_namespaces - overview of Linux user namespaces.* Feb. 13, 2023. URL: https://man7.org/linux/man-pages/man7/user_namespaces.7.html.

[34] Kernel development community. *uts_namespaces - overview of Linux UTS namespaces.* Feb. 13, 2023. URL: https://man7.org/linux/man-pages/man7/uts_namespaces.7.html.

[35] Kubernetes authors. *Admission controllers.* Feb. 2, 2023. URL: https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#defaulttolerationseconds.

[36] Kubernetes authors. *DaemonSet.* Feb. 5, 2023. URL: https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/.

[37] Kubernetes authors. *Deployment.* Feb. 5, 2023. URL: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/.

[38] Kubernetes authors. *kube-controller-manager.* Feb. 5, 2023. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/.

[39] Kubernetes authors. *kube-controller-manager.* Feb. 2, 2023. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/.

[40] Kubernetes authors. *kube-proxy.* Feb. 5, 2023. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/.

[41] Kubernetes authors. *kubelet.* Feb. 5, 2023. URL: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/.

[42] Kubernetes authors. *Operator pattern.* Feb. 5, 2023. URL: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/.

[43] Kubernetes authors. *Pods.* Feb. 5, 2023. URL: https://kubernetes.io/docs/concepts/workloads/pods/.

[44] Kubernetes authors. *ReplicaSet.* Feb. 5, 2023. URL: https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/.

[45] Kubernetes authors. *Service.* Feb. 5, 2023. URL: https://kubernetes.io/docs/concepts/services-networking/service/.

[46] Kubernetes developers. *Kubernetes.* Feb. 13, 2023. URL: https://kubernetes.io/.

[47] Leslie Lamport. "The part-time parliament". In: *Concurrency: the Works of Leslie Lamport.* 2019, pp. 277–317.

[48] Wubin Li, Ali Kanso, and Abdelouahed Gherbi. "Leveraging Linux Containers to Achieve High Availability for Cloud Services". In: *2015 IEEE International Conference on Cloud Engineering.* 2015 IEEE International Conference on Cloud Engineering (IC2E). Tempe, AZ, USA: IEEE, Mar. 2015, pp. 76–83. ISBN: 978-1-4799-8218-9. DOI: 10.1109/IC2E.2015.17. URL: http://ieeexplore.ieee.org/document/7092902/ (visited on 08/15/2022).

[49] Thouraya Louati, Heithem Abbes, and Christophe Cérin. "LXCloudFT: Towards high availability, fault tolerant Cloud system based Linux Containers". In: *Journal of Parallel and Distributed Computing* 122 (Dec. 2018), pp. 51–69. ISSN: 07437315. DOI: 10.1016/j.jpdc.2018.07.015. URL: https://linkinghub.elsevier.com/retrieve/pii/S0743731518305136 (visited on 08/16/2022).

[50] Lele Ma, Shanhe Yi, and Qun Li. "Efficient service handoff across edge servers via docker container migration". In: *Proceedings of the Second ACM/IEEE Symposium on Edge Computing.* SEC '17: IEEE/ACM Symposium on Edge Computing. San Jose California: ACM, Oct. 12, 2017, pp. 1–13. ISBN: 978-1-4503-5087-7. DOI: 10.1145/3132211.3134460. URL: https://dl.acm.org/doi/10.1145/3132211.3134460 (visited on 08/05/2022).

[51] Rodrigo H. Müller, Cristina Meinhardt, and Odorico M. Mendizabal. "An architecture proposal for checkpoint/restore on stateful containers". In: *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing.* SAC '22: The 37th ACM/SIGAPP Symposium on Applied Computing. Virtual Event: ACM, Apr. 25, 2022, pp. 267–270. ISBN: 978-1-4503-8713-2. DOI: 10.1145/3477314.3507221. URL: https://dl.acm.org/doi/10.1145/3477314.3507221 (visited on 08/05/2022).

[52] Shripad Nadgowda et al. "Voyager: Complete Container State Migration". In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS).* 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS). Atlanta, GA, USA: IEEE, June 2017, pp. 2137–2142. ISBN: 978-1-5386-1792-2. DOI: 10.1109/ICDCS.2017.91. URL: http://ieeexplore.ieee.org/document/7980161/ (visited on 08/05/2022).

[53] Hylson Netto et al. "Incorporating the Raft consensus protocol in containers managed by Kubernetes: an evaluation". In: *International Journal of Parallel, Emergent and Distributed Systems* 35.4 (July 3, 2020), pp. 433–453. ISSN: 1744-5760, 1744-5779. DOI: 10.1080/17445760.2019.1608989. URL: https://www.tandfonline.com/doi/full/10.1080/17445760.2019.1608989 (visited on 08/08/2022).

[54] Hylson V. Netto et al. "State machine replication in containers managed by Kubernetes". In: *Journal of Systems Architecture* 73 (Feb. 2017), pp. 53–59. ISSN: 13837621.

DOI: 10.1016/j.sysarc.2016.12.007. URL: https://linkinghub.elsevier.com/retrieve/pii/S1383762116302752 (visited on 08/05/2022).

[55] Hylson Vescovi Netto et al. "Koordinator: A Service Approach for Replicating Docker Containers in Kubernetes". In: *2018 IEEE Symposium on Computers and Communications (ISCC)*. 2018 IEEE Symposium on Computers and Communications (ISCC). Natal: IEEE, June 2018, pp. 00058–00063. ISBN: 978-1-5386-6950-1. DOI: 10.1109/ISCC.2018.8538452. URL: https://ieeexplore.ieee.org/document/8538452/ (visited on 08/05/2022).

[56] SeungYong Oh and JongWon Kim. "Stateful Container Migration employing Checkpoint-based Restoration for Orchestrated Container Clusters". In: *2018 International Conference on Information and Communication Technology Convergence (ICTC)*. 2018 International Conference on Information and Communication Technology Convergence (ICTC). Jeju: IEEE, Oct. 2018, pp. 25–30. ISBN: 978-1-5386-5041-7. DOI: 10.1109/ICTC.2018.8539562. URL: https://ieeexplore.ieee.org/document/8539562/ (visited on 08/05/2022).

[57] Diego Ongaro and John Ousterhout. "In search of an understandable consensus algorithm (extended version)". In: *Proceeding of USENIX annual technical conference, USENIX ATC*. 2014, pp. 19–20.

[58] Open Container Initiative. *About the Open Container Initiative*. URL: https://opencontainers.org/about/overview/ (visited on 08/17/2022).

[59] Operator Framework. *Operator SDK*. Jan. 27, 2023. URL: https://sdk.operatorframework.io/.

[60] Prometheus Authors. *Node Exporter - Exporter for machine metrics*. Jan. 27, 2023. URL: https://github.com/prometheus/node_exporter.

[61] Prometheus Authors. *Prometheus - Monitoring system & time series database*. Jan. 27, 2023. URL: https://prometheus.io/.

[62] Gourav Rattihalli et al. "Exploring Potential for Non-Disruptive Vertical Auto Scaling and Resource Estimation in Kubernetes". In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 2019 IEEE 12th International Conference on Cloud Computing (CLOUD). Milan, Italy: IEEE, July 2019, pp. 33–40. ISBN: 978-1-72812-705-7. DOI: 10.1109/CLOUD.2019.00018. URL: https://ieeexplore.ieee.org/document/8814504/ (visited on 08/12/2022).

[63] RedHat, Inc. *Introduction to Kubernetes architecture*. Oct. 5, 2022. URL: https://www.redhat.com/en/topics/containers/kubernetes-architecture.

[64] Jakob Schrettenbrunner. "Migrating Pods in Kubernetes". PhD thesis. Dec. 2020. DOI: 10.13140/RG.2.2.31821.97762.

[65] Radostin Stoyanov and Martin J. Kollingbaum. “Efficient Live Migration of Linux Containers”. In: *High Performance Computing*. Ed. by Rio Yokota et al. Vol. 11203. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 184–193. ISBN: 978-3-030-02464-2 978-3-030-02465-9. DOI: 10.1007/978-3-030-02465-9_13. URL: http://link.springer.com/10.1007/978-3-030-02465-9_13 (visited on 08/05/2022).

[66] Martin Terneborg, Johan Karlsson Ronnberg, and Olov Schelen. “Application Agnostic Container Migration and Failover”. In: *2021 IEEE 46th Conference on Local Computer Networks (LCN)*. 2021 IEEE 46th Conference on Local Computer Networks (LCN). Edmonton, AB, Canada: IEEE, Oct. 4, 2021, pp. 565–572. ISBN: 978-1-66541-886-7. DOI: 10.1109/LCN52139.2021.9525029. URL: https://ieeexplore.ieee.org/document/9525029/ (visited on 08/05/2022).

[67] Leila Abdollahi Vayghan et al. “A Kubernetes controller for managing the availability of elastic microservice based stateful applications”. In: *Journal of Systems and Software* 175 (May 2021), p. 110924. ISSN: 01641212. DOI: 10.1016/j.jss.2021.110924. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121221000212 (visited on 08/05/2022).

[68] Leila Abdollahi Vayghan et al. “Kubernetes as an Availability Manager for Microservice Applications”. In: (), p. 10.

[69] Ranjan Sarpangala Venkatesh et al. “Fast in-memory CRIU for docker containers”. In: *Proceedings of the International Symposium on Memory Systems*. MEMSYS ’19: The International Symposium on Memory Systems. Washington District of Columbia USA: ACM, Sept. 30, 2019, pp. 53–65. ISBN: 978-1-4503-7206-0. DOI: 10.1145/3357526.3357542. URL: https://dl.acm.org/doi/10.1145/3357526.3357542 (visited on 08/08/2022).

[70] Bo Xu et al. “Sledge: Towards Efficient Live Migration of Docker Containers”. In: *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. 2020 IEEE 13th International Conference on Cloud Computing (CLOUD). Beijing, China: IEEE, Oct. 2020, pp. 321–328. ISBN: 978-1-72818-780-8. DOI: 10.1109/CLOUD49709.2020.00052. URL: https://ieeexplore.ieee.org/document/9284305/ (visited on 08/05/2022).

[71] Hanlin Zhang et al. “Multi-level Container Checkpoint Performance Optimization Strategy in SDDC”. In: *Proceedings of the 2019 4th International Conference on Big Data and Computing - ICBDC 2019*. the 2019 4th International Conference. Guangzhou, China: ACM Press, 2019, pp. 253–259. ISBN: 978-1-4503-6278-8. DOI: 10.1145/3335484.3335487. URL: http://dl.acm.org/citation.cfm?doid=3335484.3335487 (visited on 08/15/2022).

[72] Shuo Zhang et al. "A High-Performance Adaptive Strategy of Container Checkpoint Based on Pre-replication". In: *Security, Privacy, and Anonymity in Computation, Communication, and Storage*. Ed. by Guojun Wang, Jinjun Chen, and Laurence T. Yang. Vol. 11342. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2018, pp. 240–250. ISBN: 978-3-030-05344-4 978-3-030-05345-1. DOI: 10.1007/978-3-030-05345-1_20. URL: http://link.springer.com/10.1007/978-3-030-05345-1_20 (visited on 08/12/2022).

[73] Diyu Zhou and Yuval Tamir. "Fault-Tolerant Containers Using NiLiCon". In: *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS). New Orleans, LA, USA: IEEE, May 2020, pp. 1082–1091. ISBN: 978-1-72816-876-0. DOI: 10.1109/IPDPS47924.2020.00114. URL: https://ieeexplore.ieee.org/document/9139863/ (visited on 08/05/2022).

[74] Diyu Zhou and Yuval Tamir. *HyCoR: Fault-Tolerant Replicated Containers Based on Checkpoint and Replay*. Jan. 23, 2021. arXiv: 2101.09584[cs]. URL: http://arxiv.org/abs/2101.09584 (visited on 08/16/2022).

Eidesstattliche Versicherung

(Affidavit)

Schmidt, Henri

Name, Vorname
(surname, first name)

225340

Matrikelnummer
(student ID number)

☐ Bachelorarbeit
(Bachelor's thesis)

■ Masterarbeit
(Master's thesis)

Titel
(Title)

State-Preserving Container Orchestration in Failover Scenarios

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem oben genannten Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present thesis with the above-mentioned title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution before.

17.02.2023, Minden

Ort, Datum
(place, date)

Unterschrift
(signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to EUR 50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the Chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, Section 63 (5) North Rhine-Westphalia Higher Education Act (*Hochschulgesetz, HG*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund University will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:*

17.02.2023, Minden

Ort, Datum
(place, date)

Unterschrift
(signature)

***Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.**