Transparent Fault Tolerance for Stateful Applications in Kubernetes with Checkpoint/Restore

Henri Schmidt
TU Dortmund
Dortmund, Germany
henri.schmidt@tu-dortmund.de

Zeineb Rejiba
Hitachi Energy Research
Dättwil, Switzerland
0000-0001-8807-3105

Raphael Eidenbenz
Hitachi Energy Research
Dättwil, Switzerland
0009-0009-6204-5988

Klaus-Tycho Förster
TU Dortmund
Dortmund, Germany
0000-0003-4635-4480

***Abstract*—This paper presents a solution providing fault tolerance for stateful containerized applications that is transparent, i.e., the application does not require to structure or manage its state in any particular fashion. In the case of faults, such as node crashes or node isolation, the application resumes execution on another node. The solution relies on a Kubernetes operator and a tool to periodically checkpoint containers and restore from the latest checkpoints in case of a node failure.**

Experimental evaluations reveal the trade-offs between overhead due to checkpointing, i.e., CPU load, memory, network bandwidth, reduced availability, and the performance during recovery, i.e., outage time, state quality. Compared to a non-transparent solution, the transparent solution yields similar downtimes and state quality at an increased overhead.

***Index Terms*—fault tolerance, container orchestration, reliability, checkpointing**

I. Introduction

Containers have become the de-facto standard for packaging, deploying, and managing software. Their benefits are that they provide isolation, come with dependencies included, they are easily distributed via registries. Moreover, lightweight virtualization allows containers to run on a wide variety of host platforms at a performance that is close to native. Moreover, container orchestration frameworks—first and foremost, Kubernetes—provide the necessary means to run dynamic container based workloads across a cluster and thus enable a serverless platform-as-a-service type of experience. Thus, application developers are freed from thinking in confinements of device boxes. Instead, they can see a cluster as an elastic sea of compute power at their disposal.

By contrast to traditional deployments of software, containers are considered *ephemeral* entities[1], i.e., a container might be stopped and restarted at any point in time, potentially in another node. When such a restart or a "migration" to another node occurs, a new container instance is spawned, i.e., a new instance starts with fresh memory and any previous state in memory is lost. For this reason, only stateless containers are typically deployed into container orchestration frameworks without any additional state preserving measures. Containerized applications that rely on persistent state must implement additional measures such as shared volumes, external databases, etc. for storing their state and recovering state in a case of failure. Beyond recovery from faults, also planned migrations, e.g., if an application is moved from one server rack to another, require special measures for transferring state.

As a result, if an application developer wants to create a stateful application that is containerized and managed by a container orchestration framework, they spend significant effort on devising an infrastructure that can persist the state, and on adding the functionality to store, retrieve and restore from a state snapshot. Such additional effort can become a major factor to decide against containerization of existing software. Merely determining what makes up the state of a legacy application is already a challenging endeavour that can take months. Thus, the lack of proper support for stateful containers prevents organizations from modernizing their software portfolio and reaping the benefits that containerization brings.

What current container orchestration frameworks are missing is support for *transparent fault tolerance*, i.e., a feature that allows containers to fail and resume from the same state without requiring the application to provide special measures.

In this paper, we present and evaluate a solution that achieves transparent fault tolerance. Concretely, we are the first to design, implement and evaluate a Kubernetes operator that harnesses the recent integration of the Checkpoint/Restore in Userspace (CRIU) tool into the Kubernetes ecosystem. Our operator strategically checkpoints stateful containers, transfers the checkpoint to a target node and restores the container from that checkpoint in case of a failure. Thereby a checkpoint includes the container's entire state. Once the container is restored from that checkpoint it resumes operation from that exact state. The application developer does not need to provide any mechanisms to persist or transfer state.

The main contributions of our work are:

- The design of a Kubernetes operator that provides transparent fault tolerance for containerized applications.
- An experimental evaluation of the solution's performance in terms of the downtime and state quality experienced by the applications in the presence of faults.
- An experimental evaluation of the overhead in terms of impact on CPU load, network, memory and interference to application containers when there are no faults.
- A comparison of our operator with a non-transparent solution in terms of performance and overhead.

[1]https://docs.docker.com/develop/develop-images/dockerfile_best-practices/#create-ephemeral-containers

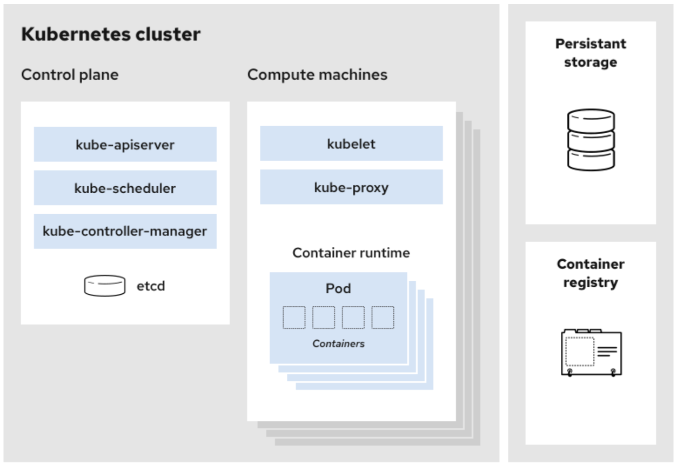

Fig. 1: Kubernetes Architecture [2].

II. Background

This section provides an overview of the tools and concepts that our proposed Checkpoint/Restore (C/R) solution builds upon. In particular, it introduces Kubernetes, CRIU as well as the details of the recent checkpointing feature in Kubernetes.

A. Kubernetes

Kubernetes is an open source container orchestration framework that is widely used in industry [1]. 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. Additional features include automated rollouts and rollbacks, self-healing, automated scaling as well as extensibility.

In the following, we provide an overview of the Kubernetes architecture, the operator concept, and fault tolerance features.

1) Kubernetes architecture: Fig. 1 depicts a high level overview of the Kubernetes architecture. As can be seen, a cluster consists of compute machines, called *nodes*, and the *control plane*. The control plane contains the components needed for managing the cluster. The nodes run the actual user applications. The control plane is often run on its own physical device, but it can also be collocated with a compute node. Moreover, it can also be distributed across multiple devices for redundancy. *Pods* are the components that contain application containers. Pods constitute the scheduling unit in Kubernetes.

The Kubernetes components are described in the following:

- **API Server**. Exposes the Kubernetes API via HTTP.
- **Controller manager**. Contains all the standard controllers for the default resources available in Kubernetes. *Resources* in the Kubernetes terminology refer to specific API objects that can be managed by Kubernetes. These include pods, services, deployments, etc. Each resource is continuously watched by its corresponding controller to ensure that its current state matches its desired state.
- **etcd**. A persistent distributed key-value store, which stores resources and cluster configuration information.
- **Scheduler**. Component responsible for determining the most suitable node for a given pod.
- **kubelet**. Agent running on each node in a Kubernetes cluster. It communicates over a REST API with the API server and is responsible for interacting with the underlying container runtime.
- **Container runtime**. Component responsible for managing the containers' lifecycle. Examples runtimes are CRI-O[2] and Containerd[3]. Lower level container management details are instead handled by tools such as runc[4].
- **kube-proxy**. Enables communication towards pods. It also implements part of the *Service* concept, where a Service refers to a Kubernetes resource used as a unified way for exposing application pods, regardless of the pods' underlying IP addresses.

2) Operators in Kubernetes: In addition to the standard controllers provided by the Controller manager, it is possible to add custom controllers to Kubernetes via the *Operator*[5] concept. Operators are based on one or multiple controllers as well as optional *Custom Resource Definitions* (CRDs). Operators are typically used to automate complex application tasks, such as self-healing, scaling and updates. A CRD defines a specification for custom resources, including the expected fields and resource metadata. After the CRD is installed, resources of that type (i.e. specific instances based on the CRD) can be created and managed in the Kubernetes cluster. *Controllers* use the *watch* capability offered by the Kubernetes API to continuously monitor associated resources. Every change in the resource triggers a *reconcile loop* which ensures that the current state of the resource matches its desired state. Since CRDs are optional in an operator, it is possible to create only the custom controller and configure it to watch and add additional functionality to standard Kubernetes resources (e.g., pods).

3) Fault Tolerance in Kubernetes: Kubernetes provides two mechanisms for fault tolerance, self-healing and replication. *Self-healing* refers to the feature that a failing container is restarted, first on the same node, eventually on another node. The Kubernetes Service concept provides *replication*, i.e., multiple instances of the same container are run in parallel. Requests to the service are dispatched to one of the instances. If one fails, requests are handled by the remaining instances. Both mechanisms are intended for stateless containers, as state is not synchronized nor preserved.

B. CRIU

CRIU[6] is a tool that can freeze the memory state of a running Linux application and save it as a collection of files on disk (*checkpointing*). These files can then be used to *restore* the application at the same state it was frozen at. During checkpointing CRIU operates on the process tree and uses

[2]https://cri-o.io/
[3]https://containerd.io/
[4]https://github.com/opencontainers/runc
[5]https://kubernetes.io/docs/concepts/extend-kubernetes/operator
[6]https://criu.org

the *ptrace()* system call or *cgroups freezer*[7] to pause the main process as well as its child processes. This step is needed to ensure a consistent state during checkpointing. After freezing the process tree, CRIU writes all process information to image files. This is mostly information retrieved from `/proc`, such as file descriptor information, pipe parameters and memory-mapped files. To save the memory, all memory pages of the process need to be exported. This is done using a technique called *parasite code injection*[8]. This parasite code replaces a part of the original process code and collects the memory content from within the address space of the process and writes it to image files. Once this is done, the parasite code is removed, and the original code is placed in the process again. At this point, the original process can be killed, or it can remain running, depending on the use case.

For the restore step, CRIU starts by reading checkpoint files from disk and then recreating the process tree as it was before checkpointing. Additionally, CRIU maps all the memory pages back to their original locations. As a last step, it loads all the security settings. At this point, the process resumes execution from the point at which it was checkpointed. Note that CRIU can also handle C/R for TCP sockets using the `TCP_REPAIR` socket option, which is available since kernel version 3.5.[9]

C. Integration of Minimal Checkpointing in Kubernetes

With Kubernetes v1.25, minimal checkpointing support was added as an alpha feature. A main use case for the feature was *forensic analysis* [3], where the main idea is to checkpoint a suspicious container and inspect the checkpointed version without impacting the original one.

To enable minimal checkpointing the *ContainerCheckpoint* feature gate must be enabled in the Kubernetes cluster, CRI-O must be selected as container runtime, and CRIU support must be enabled in CRI-O. Note that Containerd does not yet support CRIU. Checkpointing is then available via the kubelet API. To perform a checkpoint, an HTTP POST request has to be sent to the following endpoint `https://<node_ip>:10250/checkpoint/<namespace>/<pod>/<container>` where `<node_ip>` is the IP address of the node where the pod is running and 10250 is the kubelet port number. It is also necessary to specify the kubelet's self-signed certificates for authentication. Once this request is submitted to the kubelet, it will be forwarded to CRI-O, then runc and finally CRIU. Upon successful checkpoint creation, a tar archive will be available at `var/lib/kubelet/checkpoints/checkpoint-<pod>-<namespace>-<container>-<timestamp>.tar`.

Unlike checkpointing, the restore process cannot be performed via the kubelet API. Instead, the process consists of the following two steps. First, the checkpoint tar archive must be converted into an image format that can be pushed to a registry. The Open Container Initiative (OCI) format can be used for this purpose, since it is a standardized image format that includes the file system layers and metadata. Additionally, the following annotation needs to be added to the image: `io.kubernetes.cri-o.annotations.checkpoint.name=<container>` to indicate that it is a checkpoint. Next, the created image must be used in the pod specification. When the kubelet instructs CRI-O to create a container using this image, CRI-O will detect that the image refers to a checkpoint rather than a standard container image. As a result, it will restore the container based on this checkpoint.

Finally, since C/R feature in Kubernetes is still experimental, it comes with some noteworthy limitations. The security risks that come with C/R are not yet fully analyzed. The checkpoint image format used for restore purposes is not yet standardized. It is currently only supported by CRI-O.

III. Related Work

Transparent fault tolerance was first investigated in the 1990s and 2000s for applications on an operating system [4], and subsequently studied for web services [5], [6], virtual machines [7]–[9], and high-performance computing [10]. With the emergence of containers and orchestration frameworks such as Kubernetes, fault tolerance research received increased attention. Thereby, the only work with explicit focus on transparency is Borges et al. [11]. It proposes a transparent state machine replication solution that harnesses Kubernetes. Netto et al. [12]–[14] adds explicit state synchronization to Kubernetes' replication using the Raft consensus protocol [15]. Vayghan et al. [16], [17] propose a mechanism that uses a Kubernetes controller and a forwarding process to duplicate the network traffic to a standby node. Thus, by providing the same inputs to each instance, the states shall be implicitly synchronized. The downside of such synchronization is that establishing the same level of redundancy after a failover requires some additional form of synchronization, since a newly started standby must be brought to the same state as the currently running instance. Moreover, the container logic must be deterministic. Johansson et al. [18] compare setups with hot standby to a setup with cold standby in the contexts of virtual distributed controller nodes in Kubernetes. By contrast, our proposed solution should be classified neither as a hot nor a cold, but as a warm standby solution. Whereas the standby container is not running until a failover occurs, the state is continuously transferred to the target node. Moreover, the container image is pulled to the target node a priori.

Note that except Vayghan and Borges' work all mentioned solutions are non-transparent. Harnessing C/R technology can avoid this problem. Most work on C/R addresses migration of containers from one machine to another [19]–[30]. Thereby the authors explore different strategies and aspects of migration, with the objective to minimize downtime and transferred data volume. In particular, Oh et al. [25] discuss some limitations of container migration using persistent volumes, such as the need to modify the underlying application to support saving/restoring state to the volume. Another line of C/R research focuses on optimizing CRIU on the level of the Linux kernel [31]–[35].

[7]https://criu.org/freezing_the_tree
[8]https://criu.org/Parasite_code
[9]https://criu.org/tcp_connection

Finally, there is some limited work on using C/R in Kubernetes. Rattihalli et al. [21] harnesses C/R for fast auto-scaling in Kubernetes and for optimizing the allocation in cluster by migrating containers to underutilized nodes. Similarly, Schrettenbrunner [27] implements integrated Kubernetes components to enable migration of pods to other machines. Müller et al. [36] propose an architecture for migration and failover with Kubernetes pods while maintaining and replaying network connections using an interceptor. Their work constitutes a good starting point for integration of C/R into Kubernetes, but it lacks an experimental evaluation of the proposed architecture.

IV. Model and Design Goals

In containerized systems, faults can happen at the node, container, application or network level [37]. Thereby the different levels have typical fault causes: hardware or VM failures lead to node failures [23], [38], software bugs lead to application failures [36], etc. Our work considers faults on any level as long as they affect the application container's execution or communication to the Control plane. Moreover, we assume a *fail-stop* model [31], [32]. Byzantine faults are ignored. The two prime fault scenarios are *node crashes* and *node isolation* due to network failures. Both scenarios require a failover of the application to another node. Applications are assumed *stateful* and *critical*, i.e., downtime shall be minimized. Finally, the effort for the application developers to enable failovers for their application shall be negligible, or even zero (full *transparency*).

V. Proposed Solution: C/R Operator

The proposed C/R Operator implements C/R operations for use in failover scenarios. For the checkpoint step, it uses the experimental checkpoint feature in Kubernetes. As for the restore, it is handled directly by the CRE. In this section, we provide a detailed description of the components of the C/R Operator. Then, we discuss practical aspects of the implementation, including challenges and known limitations.

A. Design

Fig. 2 depicts the architecture of the C/R operator. At a high level, the operator is comprised of two main components:

- **Controller Manager**. Contains all the controllers needed to perform the C/R operator functionalities. In addition, it is responsible for coordinating Agents running on each node. It can be further be broken down into four controllers and two handlers. Each controller is responsible for watching changes to one resource type in the cluster. As for the handlers, they are responsible for checking the liveness of Pods and for instructing agents to perform checkpoint-related operations.
- **Agents**. Components based on the Kubernetes Daemonset concept, i.e., one agent runs on each node. They are responsible for checkpointing containers, creating checkpoint images and transferring them between nodes.

In the following, we provide more details about the design of each of these components.

1) Node Controller: This controller watches node resources and is notified upon node state change. It maintains an internal node list and adds a label to each node to identify it by its internal cluster IP. These labels are later used by the Liveness Handler to select the appropriate recovery node. It is worth noting that the node IP address is already available in other parts of the node data structure. However, to ensure compatibility with the scheduler requirements, it is specified as a label, too.

2) Agent Controller: This controller is responsible for watching the DaemonSets representing Agents. More specifically, it listens to changes made to this DaemonSet and records those changes in the available Agents.

3) Config Controller: This controller watches Deployments and filters them based on a custom annotation that we call *cr_mode*. This annotation identifies Deployments that should be checkpointed. In addition, such deployments should have the *cr_interval* annotation to indicate the checkpointing interval. The Config Controller maintains a list of all Deployments having those annotations and it notifies the Checkpoint Handler and the Liveness Handler when such Deployments are created.

4) C/R Controller: This controller watches Pod resources and keeps track of all the Pods belonging to the Deployments configured for checkpointing. Additionally, it determines the recovery node using the node information provided by the Node Controller.

5) Checkpoint Handler: The Checkpoint Handler uses information provided by the Config Controller, the Node Controller and the C/R Controller to checkpoint Pods of monitored Deployments. More specifically, the Config Controller provides the Checkpoint Handler with the configured checkpoint interval, whereas the C/R controller provides it with the information of the Pods belonging to the Deployment. The Node Controller provides the Checkpoint Handler with the node list, including the information needed to reach the Kubelet. This information is then used to request a checkpoint via Kubelet API. Once the checkpoint is created, the Checkpoint Handler instructs the Agent running on the node of the checkpointed pod to convert the checkpoint into an OCI image. To achieve this, the Agent uses the *Buildah* image build tool[10]. As a last step, the Checkpoint Handler instructs the Agent to transfer the OCI image to the predetermined recovery node.

6) Liveness Handler: The Liveness Handler uses the information retrieved from the Config Controller to execute liveness probes. In the standard Kubernetes architecture, when a Pod is configured with liveness probes, it is the Kubelet's responsibility to execute those probes. If a node goes down, a timeout of 40 seconds is observed, after which the node will be considered *NotReady*. This timeout can be configured using the `--node-monitor-grace-period` option of the Kubernetes Controller Manager[11]. Reducing this grace period arbitrarily can lead to an unstable system with quick and

[10]https://buildah.io/

[11]https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/

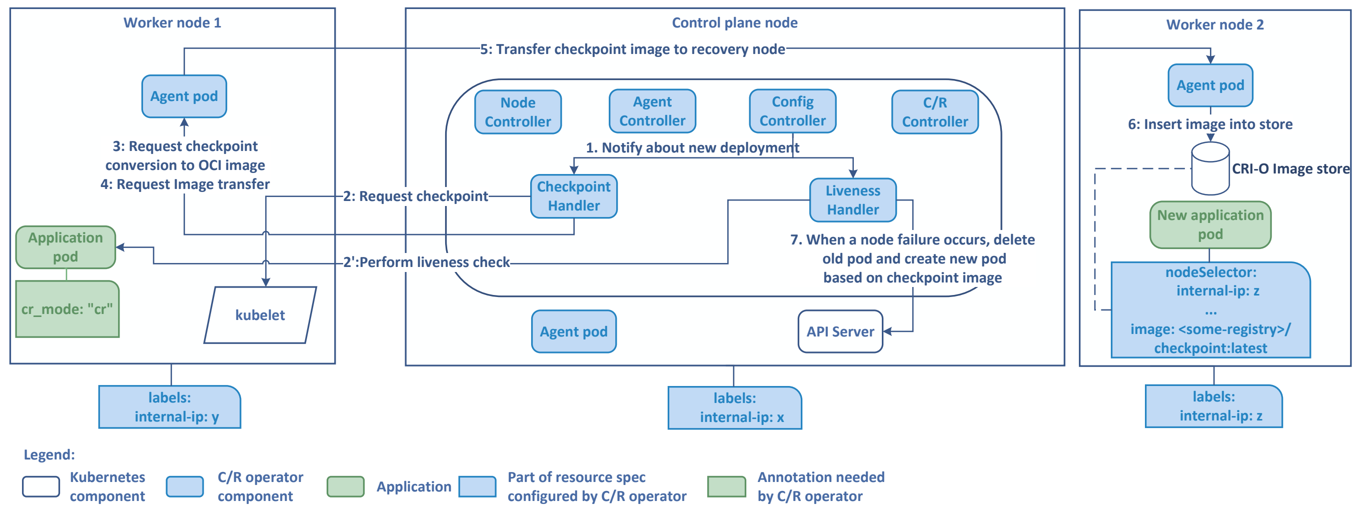
Worker node 1
Agent pod
3: Request checkpoint conversion to OCI image
4: Request Image transfer
Application pod
cr_mode: "cr"
kubelet
labels: internal-ip: y
Control plane node
5: Transfer checkpoint image to recovery node
Node Controller
Agent Controller
Config Controller
C/R Controller
1. Notify about new deployment
2: Request checkpoint
Checkpoint Handler
Liveness Handler
2':Perform liveness check
7. When a node failure occurs, delete old pod and create new pod based on checkpoint image
Agent pod
API Server
labels: internal-ip: x
Worker node 2
Agent pod
6: Insert image into store
CRI-O Image store
New application pod
nodeSelector: internal-ip: z ... image: <some-registry>/ checkpoint:latest
labels: internal-ip: z
Legend:
Kubernetes component
C/R operator component
Application
Part of resource spec configured by C/R operator
Annotation needed by C/R operator

Fig. 2: C/R Operator component interactions.

frequent node state changes. Once the node monitor grace period is over, the standard Pod toleration timeout[12] of 5 minutes will start. After both timeouts have elapsed, the Pods are restarted on other Nodes. Even if shorter liveness probe intervals are configured, the Pods are only restarted after their eviction from the unreachable node. To mitigate the slow detection of the Pod failure, our C/R operator implements its own Liveness Handler by performing 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 the recovery process. This process consists in the deletion of the old Pod and the creation of a new Pod. In this step, deleting the old pod and relying on the ReplicaSet to create a replacement pod is not enough, since the ReplicaSet Controller will perform multiple attempts at recreating the pod on the same node before moving it to a different one. To prevent this, the C/R operator creates the pod itself on the target node using the original ReplicaSet information. The created pod also uses the checkpoint image as the container image. The container runtime of the target node will detect that the provided image is a checkpoint image and it will delegate the actual restore to runc and CRIU.

B. Implementation Details

Since writing a Kubernetes operator from scratch is a complex process that requires a deep knowledge of Kubernetes internals, multiple open source libraries and tools have been created to facilitate this task. In this work, we used the Operator Software Development Kit [13]. The Operator SDK can bootstrap operators with all the necessary components and Kubernetes resources. As the C/R Operator targets an existing Kubernetes resource (i.e. Deployments having the `cr-mode` annotation), we omitted the creation of a CRD when we bootstrapped the operator. After obtaining the components generated from the Operator SDK, we modified them to include our custom logic, namely the functionality of the Node Controller, C/R Controller, Config Controller and Agent Controller. We also implemented the Checkpoint Handler and the Liveness Handler from scratch.[14]

In the following, we outline some generic known limitations of our system, as well as practical considerations that need to be observed for a successful restore process. Due to space limitations, we highlight the most important ones here and list additional ones in the README of our GitHub repository.

1) Known 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. At the the Kubelet API level, checkpointing Pods with multiple containers is not supported yet. However, it is possible to checkpoint a single container from a Pod with multiple containers.

2) Special considerations for the Restore Step:

- During restore, CRI-O needs to first setup all the container mounts correctly. As a result, it is necessary that the filesystem on the source and target nodes are the same.
- To ensure a successful process restore, it is necessary that everything is exactly the same as it was during checkpointing. In particular, the exact same container image needs to be available at the target node, as CRI-O matches the image name and tag to restore containers. At the time of conducting this work, we noticed that due to a bug, CRI-O lacks a check of the image version.

[12]https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#defaulttolerationsecond
[13]https://sdk.operatorframework.io/

[14]The C/R Operator source code is available at https://github.com/hitachienergy/checkpoint-restore-operator.

- Incompatibilities between the source and target system in terms of CPU architecture often lead to illegal instructions being executed upon restoring a process. Most applications that support different CPU flags scan these during startup and do not expect the hardware to change while the process is running. So, the exact CPU flags need to be recorded during checkpointing. The restore can only happen on systems with the same architecture and a superset of the flags.

VI. Baseline: Pull State Operator

For the sake of the evaluation (VII), we implemented an alternative operator called *Pull State (PS) Operator*. PS Operator enables fault tolerance similarly to C/R Operator. However, in contrast to the C/R Operator, the PS Operator does so in a *non-transparent* manner. The design principle of PS Operator is as follows. The application provides an API that can be queried by the operator to retrieve and restore the state. The operator periodically pulls the state from the application and transfers it to the target node. When a fault occurs, 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 built-in readiness or liveness probes used by Kubernetes.

The *architecture* of the PS Operator is largely similar to the C/R Operator's architecture, with a few differences regarding naming and functionality. Like the C/R Operator it is comprised of two main components:

- **Controller Manager**. Responsible for retrieving the state from monitored applications via the APIs they provide. In addition, it selects recovery nodes and acts as a coordinator between the Helper Pods.
- **Helper Pods**. Components based on a DaemonSet with similar functionality as C/R Operator's Agents. They store the application state on the target nodes and restore it when a failover occurs.

The Controller Manager contains the following controllers and handlers.

- **Helper Controller**. Listens for changes in the Helper DaemonSet resources and ensures that the list of the Helper Pods is up to date.
- **Node Controller**. Similar to the Node Controller in C/R Operator.
- **Pod Controller**. Watches for pod changes and maintains the list of pods including pod states for every monitored Deployment. Additionally, it notifies the Restore Handler whenever a pod is created or deleted.
- **Pull State Controller**. Watches Deployments for custom annotations that contain parameters related to state pulling, such as `ps_port`, the API's port for pulling the state, and `ps_interval`, the time interval between the pulls.
- **Liveness Handler**. Similar to the Liveness Handler in the C/R Operator: when three consecutive liveness checks fail, it initiates the recovery by deleting the old Pod and creating a new pod on the recovery node.
- **State Handler**. Monitors deployments for changes in pod states and distributes up-to-date pod state information to all Helper Pods.
- **Restore Handler**. Collects deleted pod and new pod events and determines whether there are pods that must be restored. If there are, it instructs the Helper Pod running on the recovery node to restore the application container's state using the respective application API.

Like with C/R Operator, the PS Operator uses the Operator SDK to generate the boilerplate code for the controllers.

VII. Evaluation

By means of experiments in a Kubernetes test bed with two stateful test applications, we assess the performance of the proposed C/R Operator. We reveal the tradeoffs between the failover quality and the entailed overhead, and we compare it to the non-transparent PS Operator's performance.

A. Test Applications

We use two containerized stateful applications for the sake of our evaluation, a simple counter application, called $Count$, and the popular in-memory key-value database $Redis$[15] as a real-world application example. $Count$ serves as a worst case example for C/R, since the ratio of checkpoint size and actual state is maximal.

1) Count Application: $Count$ has a single counter c as internal state. Starting from $c = 0$, the application logic is to increase c periodically, i.e., every 100 ms by default. $Count$'s state c can be queried from outside via HTTP GET request and set via HTTP POST request at `/state`. $Count$'s health can be queried at `/health`. We implemented $Count$ as a Nodejs[16] application using the expressjs[17] framework.

2) Redis: To simulate realistic stateful applications, we enhanced the official Redis container image available on Docker Hub[18] with a health check interface and the option to fill the database with a given amount of random data upon start. For the sake of the evaluation, we use four configurations of our $Redis$ application, one without any data, and the others with 1 MB, 10 MB and 100 MB of random data. We refer to those variants as $Redis$, $Redis_{1m}$, $Redis_{10m}$, and $Redis_{100m}$.

B. Experimental Setup

We performed the experiments on a two-node Kubernetes cluster using the two mentioned applications, $Count$ and $Redis$. More details on the setup are given in the following.

1) Kubernetes Cluster: The test bed cluster consists of two physical machines which are connected via a 100 Mbit/s Fast Ethernet switch. The first machine, which we denote as *Source* node, has an Intel Xeon E5-2660 v2 CPU with 8 cores and advanced vector extension (AVX) version 1. The second, called *Target* node, has an Intel Xeon E5-2660 v3 CPU with 10 cores and AVX2. Both machines have 64 GB of RAM.

[15]https://redis.com/
[16]https://nodejs.org
[17]https://expressjs.com
[18]https://hub.docker.com/_/redis/

Each machine hosts a Kubernetes compute node. The Target node additionally hosts all Kubernetes control plane components as well as the operators. The test application runs on the Source node initially. The Target node acts as standby node, i.e., execution should continue on the Target node in case of a failure on the Source node.

Note that the roles of the two nodes must be chosen as described due to the different AVX versions. According to the restrictions mentioned in Section V-B2, a checkpoint produced on a device with AVX2 fails to restore on a device with AVX. In the concrete case of $Count$, the contained OpenSSL library uses AVX2 instructions to speed up hash calculations if AVX2 is available. Restoring the application on a device without AVX2 leads to illegal instructions.

2) Container Orchestration Stack: The used Kubernetes version was 1.25.4. It was installed through Kubeadm with the *ContainerCheckpoint* feature flag enabled. As container runtime engine, we used a modified version of CRI-O 1.25.4. Our modification[19] fixes an issue with the C/R functionality, and it disables the cleanup of the CRIU logs. The latter modification allowed us to harness the logs for performance analysis and debugging. At the lowest level, Runc[20] is used to run the containers. Finally, container networking is set up using Flannel[21] and the Flannel CNI plugin[22].

C. Fault Injection

We implemented two options for simulating node failures: rebooting the host device, and manipulating IP tables so as to inhibit the relevant Ethernet communication between the failing node and the control plane, i.e., between the Source node and the Target node in our case.

We manually confirmed that the fault tolerance mechanism works in both cases. For the automated execution of performance evaluation experiments, we decided to use the option of IP tables manipulation, since it enabled us to repeat the experiments more quickly and more reliably.

D. Metrics

We are interested in the performance during the *failover*, i.e., when there are faults and the system recovers from the faults, as well as in the *overhead* when there are no faults.

1) Failover Metrics: The following metrics are used for assessing the failover quality.

- *Recovery time*, i.e., the time between the occurrence of a fault that renders the application unavailable and the time at which the application is reachable again with recovered state. Moreover, we are interested in the *breakdown* of the recovery time, i.e., which operations of the detection and restoration process take how long.
- *State discrepancy*, i.e., the difference of the recovered application state and the application state at fault time. The lower the discrepancy, the higher is the *state quality*.

Recovery time is measured by the Python script that automates the experiment execution. The script measures the time difference between the time when it injects the fault, i.e., the Source node is disconnected, and the time when the new pod instance has restored its state and serves HTTP requests again.

State discrepancy is defined for the $Count$ application as $\Delta_c = c_{pre} - c_{post}$, where c_{pre} is the last value of the counter variable c before the failover, and c_{post} is the first value of c after the failover. The state discrepancy Δ_c of $Count$ is also measured by the experiment automation script. The script queries the application state, $\tilde{c}_{pre}$, via an HTTP request right before it disconnects the node. As soon as the pod has recovered the script retrieves the state, $\tilde{c}_{post}$, via HTTP request and approximates Δ_c by $\tilde{c}_{pre} - \tilde{c}_{post}$. Note that state discrepancy is not evaluated for the $Redis$ applications.

2) Overhead Metrics: The following metrics are used for assessing the overhead.

- General device resource consumption metrics, i.e., *CPU load*, *memory*, and *disk usage*.
- *Network usage*, i.e., the number of bytes sent and received per time unit on each node.
- *Frozen time*, i.e., the time span in which an application container's execution is ceased during checkpointing.
- *Application response times* as perceived by a user of the application.

The CPU load, memory, disk, and network usage were measured using Prometheus[23] and its Node Exporter[24] plugin. Thereby, we set the data granularity to 1 s, which is the lowest value recommended by Prometheus.

Frozen time and application response times are closely related and allow us to measure the impact of the checkpointing on availability. Thereby, frozen time is calculated from CRIU logs, and the response times are gathered via a measuring app developed specifically for this purpose (next section).

E. Response Time Measuring App

To reflect how a user perceives the service that is deployed with the fault tolerance mechanisms, we wrote a measuring app in Python that continuously issues a request to the test application, $Count$ or $Redis$. The measuring app thereby issues a request and waits for the response. When the response is received, it records the response time along with the time the request was issued. Then it issues the next request. The *response time* is measured by the difference between the time a request is issued and the time the response is received. Requests are issued with a timeout of 1 s. Hence, if the service is unavailable for more than a second, we expect to see multiple consecutive entries of 1 s.

F. Experiment Scenarios

All experiments follow one of two basic scenarios: application execution with failover, or without failover. The latter type is used when measuring overhead.

[19]https://github.com/hitachienergy/checkpoint-restore-operator/blob/main/0001-customization.patch

[20]https://github.com/opencontainers/runc

[21]https://github.com/flannel-io/flannel

[22]https://github.com/flannel-io/cni-plugin

[23]https://prometheus.io

[24]https://github.com/prometheus/node_exporter

TABLE I: Failover Experiments Overview

Metrics	Operators	Apps	p	N	Fig
Response times	C/R	Count	10s	1	3
Failover breakdown	C/R	Count, Redis	10s	100	4
Recovery times	C/R, PS	Count, Redis	10s	100	–
State quality	C/R, PS	Count	1..10s	100	5

TABLE II: Overhead Experiments Overview

Metrics	Operators	Apps	p	D	Fig
Response times	C/R	Count	10s	5 min	6
Frozen Time	C/R	Count, Redis	10s	6 h	7
CPU, mem, disk	C/R, PS	Count	1..10s	30 min	8,9
Network	C/R, PS	Count	1..10s	30 min	10

A *failover experiment* provokes one failover of the application from the Source node to the Target node. Concretely, in a failover experiment, the monitored application is started on the Source node and the operator is allowed to register and checkpoint the new pod. A failure of the Source node is then simulated by adding IP tables rules that disconnect it from the Kubernetes cluster. After disconnection, the operator detects the fault and restores the Pod on the Target node from the latest checkpoint.

An *overhead experiment* runs the system for a sufficiently long duration while the application on the Source node. No faults are simulated. No failovers are expected.

Tables I and II provide an overview of conducted experiments. Thereby, if Redis is listed as a tested app, we mean all four variants of Redis. p indicates the used checkpointing interval. N is the number of experiment repetitions. D indicates the duration of one overhead experiment.

G. Experiment Results

We start the presentation of results with Fig. 3, which illustrates how the system with the C/R Operator behaves from an application user's perspective. The plot shows the application response times of the *Count* application as measured by the measuring app. We can see the delayed response due to the checkpointing about every 10 s, where a single request takes between 500 ms to 800 ms to be answered. At 23 s, we observe that the injected fault causes 5 subsequent requests to time out after 1 s before the recovery on the Target node has completed, and response times are back to single digit milliseconds.

In the remainder, we first investigate the speed (VII-G1) and state quality (VII-G2) of the failovers, before exploring the checkpointing overhead (VII-G3).

1) Failover Breakdown and Recovery Times: Fig. 4 contains the breakdown of the restore operation into individual steps. The figure shows the average durations for each phase per test application. The steps are

1) *Pod scheduling and creation*. Kubernetes decides on which node the Pod should be scheduled and creates the Pod in CRI-O.

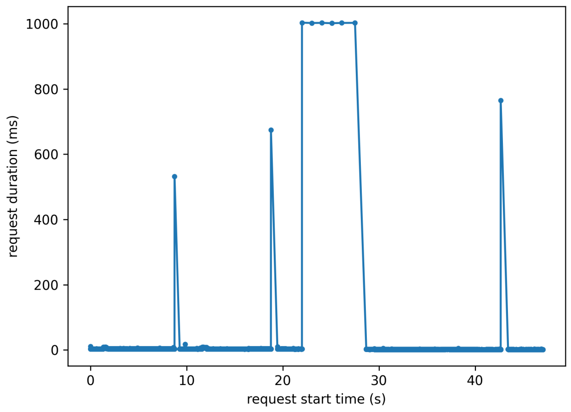

Fig. 3: Response times w/ fault at 23 s.

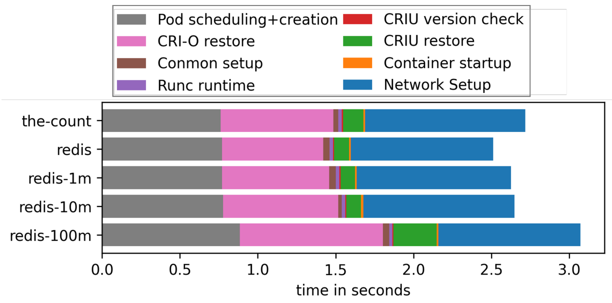

Fig. 4: Breakdown of restore w/ C/R Operator.

2) *CRI-O restore*. CRI-O creates all resources for the restore and prepare everything Conmon needs for the restore.
3) *Conmon setup*. The container runtime monitor Conmon[25] is set up and calls Runc.
4) *Runc runtime*. Runc sets up namespaces, mounts, and other resources.
5) *CRIU version check*. Runc calls CRIU to verify compatibility.
6) *CRIU restore*. The actual restore process.
7) *Container startup*. CRI-O container setup.
8) *Network setup*. Network communication is set up.

The data shows that the three phases, pod scheduling and creation, CRI-O restore, and network setup, dominate the restore phase, each taking up a bit less than 1 s. We also observe that the two phases growing most with the checkpoint size are the CRI-O restore and the CRIU restore phase, which is probably due to the increased amount of data that is processed in those phases.

The comparison of recovery times, which include fault detection and restore, shows that both C/R and PS Operator achieve recovery times between 4.5 s to 7.5 s. The C/R Operator exhibits a median of 5.87 s and PS a median of 5.94 s.

Since both operators issue liveness probes every 1 s and initiate a failover after three consecutive failed probes, the fault detection takes about 3 s. The result suggests that the failover

[25]https://github.com/containers/conmon

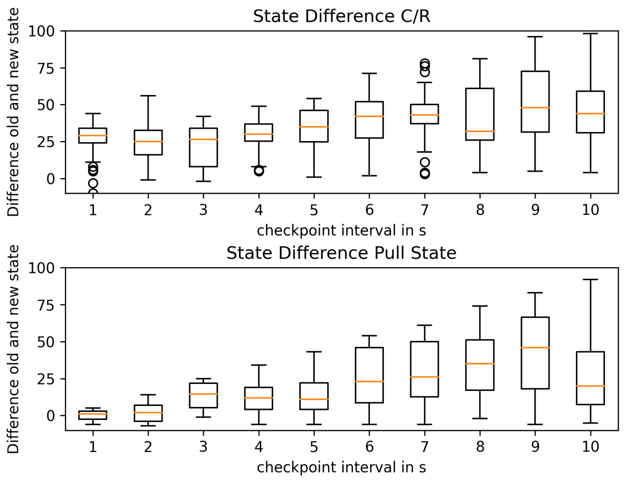

Fig. 5: State quality comparison of PS and C/R Operator.

takes about 3 s as well, which is in line with our findings regarding the failover breakdown.

It is more of a surprise that also with the PS Operator restore takes about 3 s, even though the state to be restored by $Count$ is much smaller. However, while less time is spent on restoring state, more time is spent on starting the container instance and initializing the application. In fact, *Quick startup* of containers that otherwise take long to initialize is one of the known use cases of CRIU [39]. Accordingly, depending on application startup, recovery times for the C/R Operator can be significantly faster than for the PS Operator.

2) State Quality: The worst case regarding state quality is if the fault happens just before a checkpointing, and thus the latest available checkpoint is p old. In the case of $Count$, which increments its counter every 100 ms, the checkpoint would lack $10p$ increments. Accordingly, we expect the state discrepancy to be between 0 and $10p$.

Fig. 5 shows the distributions of experienced state discrepancies before and after a failover of $Count$ depending on the checkpointing interval p. With PS, the expected upper bound of $10p$ is visible for all p. With C/R, the upper bound is only visible for $p \leq 5$ s. For smaller values of p, the slower process of checkpointing and checkpoint transfer causes the states to deviate by more than 25 consistently. Another noteworthy phenomenon is that we encounter some negative values. Since the state is measured by an HTTP request, such cases may happen if the reconnection to the network concludes only after the restored $Count$ has already incremented c a few times, and the measured $\tilde{c}_{post}$ is larger than $\tilde{c}_{pre}$.

3) Checkpointing Overhead: Fig. 6 shows an example of the response times with C/R as experienced by the measuring app. The visible peaks of about 900 ms every $p = 10$ s are caused by the process freezing during checkpointing. The distributions of frozen times as logged by CRIU are shown in Fig. 7 along with the average amount of memory written during one checkpointing operation for the different container types. Note that the frozen times are well below the observed

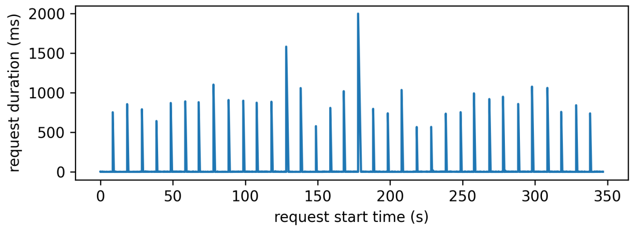

Fig. 6: Application response times with C/R, no faults.

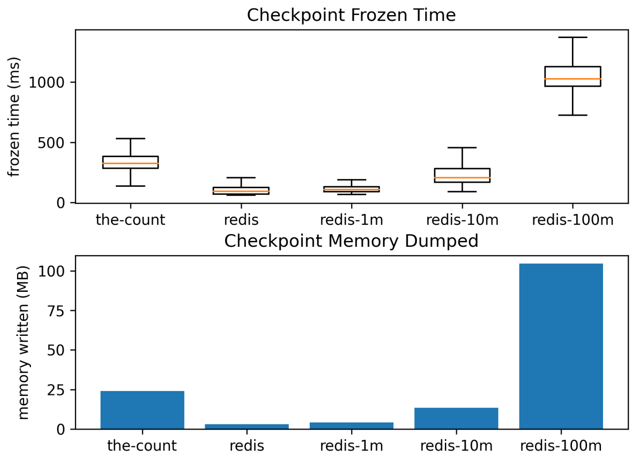

Fig. 7: Distribution of frozen times and average amount of memory written per checkpoint.

response times. The reason is that the frozen time collected by CRIU is the time tasks were in frozen state, which is lower than the freezing time, i.e., the 'time it took CRIU to freeze the process tree'[26]. Additionally, the processing of the request and networking adds to the response time. When comparing the different applications, there is a clear linear dependency of frozen time and checkpoint size on the amount of data added to Redis. Moreover, we can observe that despite the minimal state, our $Count$ implementation requires 25 MB while the highly optimized $Redis$ container uses only 1 MB. The reason for the large footprint of $Count$ is Nodejs.

Fig. 8 and 9 compare the average CPU load and disk usage for both the C/R and the PS Operator depending on the checkpointing interval. Note that Fig. 9 uses a logarithmic scale. The data shows that PS causes an almost constant CPU and disk usage. By contrast, C/R's CPU and disk usage is at a significantly higher level and grows with shorter checkpointing interval. The higher usage is caused by the vastly more complex checkpointing. Moreover, the usage is similarly high on the Target node due to the checkpoint reception and storage, whereas the overhead of storing $Count$'s c counter is negligible in the case of PS.

The memory usage measurements revealed that while there is a slight but inconsistent trend of higher usage with shorter checkpoint interval, there are no statistically relevant differences between C/R and PS. All measured values were between

[26]https://criu.org/Statistics

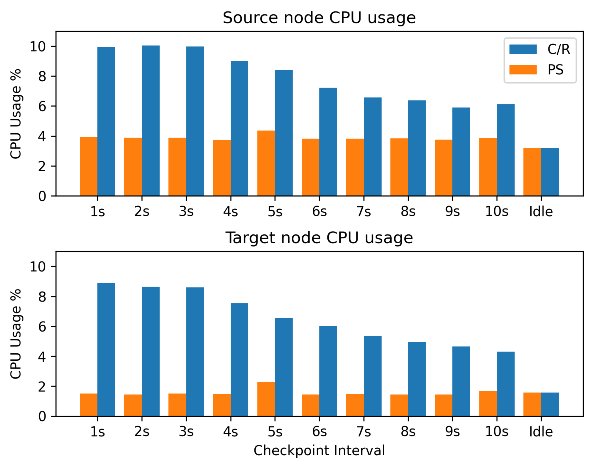

Fig. 8: Average CPU usage, no faults.

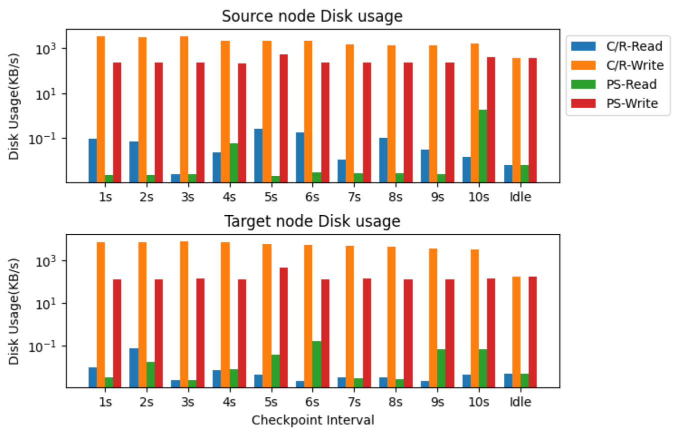

Fig. 9: Average disk usage, no faults.

2.5 GB and 3.6 GB. Memory usage without applications and operators running was at 2.5 GB.

Fig. 10 shows the transmitted (TX) and received (RX) network data for the Source and Target node depending on the checkpointing interval. Note the logarithmic scale. The results for C/R suggest a net data flow from Source to Target node due to the checkpoint transfers in the order of 1 MB/s, with a maximum of 1.4 MB/s for $p = 1$ s. PS causes significantly lower network traffic, in the order of 10 kB/s to 25 kB/s. In fact, the network usage with PS is so low that the numbers are dominated by the measurement data gathering mechanism with the effect that transmission is higher than reception on both nodes. Note that the nodes send the measurement data to Prometheus, which is on a third device.

VIII. Conclusion and Future Work

The presented C/R Operator provides transparent fault tolerance for stateful applications in Kubernetes. It yields recovery from node failures in about 6 s. Thereby, state changes of 1 s or more are typically lost due to the time needed for checkpointing and checkpoint transfer.

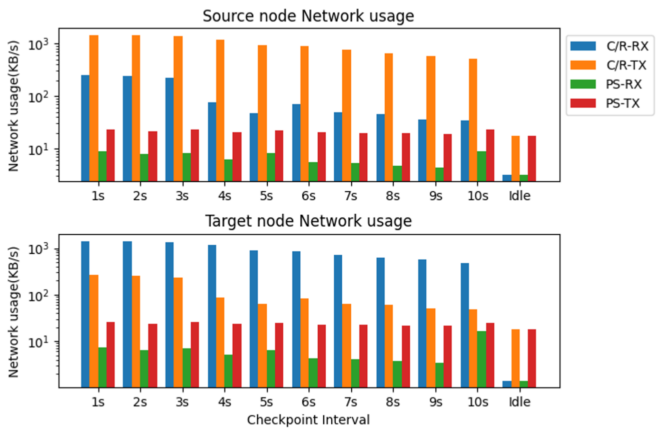

Fig. 10: Average network usage for checkpoint transfers.

Our evaluation shows that compared to a non-transparent Pull State solution, C/R's transparency comes at the expense of runtime overhead. For checkpoint intervals of 10 s or shorter, the CPU load overhead with our test apps is at 2-6%, disk and network usage overhead is in the order of 1 MB/s. Moreover, C/R causes intermittent unavailability of more than 600 ms due to application freezing. Memory usage does not increase significantly. While the focus of our work was on transparency and minimal downtime during failovers, the overhead can be reduced if availability and state quality requirements are less stringent. If losing minutes or even hours of state is tolerable during a failover, the frequency of checkpointing, and thus also the overhead, can be drastically reduced.

The question of whether the runtime overhead of a transparent solution such as the C/R Operator is worth the saved development effort needed for a more efficient custom solution for managing, storing and recovering state depends on the particular application's availability requirements, complexity, and expected lifetime. C/R is a promising solution in cases where the development of other state preserving mechanisms is costly. Such cases include the containerization of monolithic or otherwise complex legacy software, where an application's state is often greatly intertwined with the logic and dispersed across components. Moreover, no developers who are knowledgeable of the code might be available. Another use case is if state quality or availability requirements are less pronounced. Then the runtime overhead can be kept small and thus C/R is an easy way to make a stateful application fault-tolerant.

One caveat when considering C/R is the restrictions on cluster heterogeneity. Concretely, restore is only possible on nodes where the CPU architecture implements a superset of the source node instruction set.

A promising direction for future work is to make C/R more efficient by supporting *incremental checkpointing*, i.e., calculating only the difference of state when checkpointing repeatedly instead of a full checkpoint. CRIU's incremental memory dumps[27] might be harnessed for this purpose.

[27]https://criu.org/Incremental_dumps

References

[1] Cloud Native Computing Foundation. Cncf annual survey 2022. [Online]. Available: https://www.cncf.io/reports/cncf-annual-survey-2022/

[2] RedHat, Inc. Introduction to kubernetes architecture. [Online]. Available: https://www.redhat.com/en/topics/containers/kubernetes-architecture

[3] A. Reber. Forensic container checkpointing in kubernetes. [Online]. Available: https://kubernetes.io/blog/2022/12/05/forensic-container-checkpointing-alpha/

[4] T. Bressoud, "Tft: a software system for application-transparent fault tolerance," in *28th International Symposium on Fault-Tolerant Computing (FTCS)*, 1998, pp. 128–137.

[5] V. Dialani, S. Miles, L. Moreau, D. De Roure, and M. Luck, "Transparent fault tolerance for web services based architectures," in *Euro-Par 2002 Parallel Processing: 8th International Euro-Par Conference Paderborn, Germany*. Springer, 2002, pp. 889–898.

[6] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, "Eternal—a component-based framework for transparent fault-tolerant corba," *Software: Practice and Experience*, vol. 32, no. 8, pp. 771–788, 2002.

[7] H. Jo, H. Kim, J.-W. Jang, J. Lee, and S. Maeng, "Transparent fault tolerance of device drivers for virtual machines," *IEEE Transactions on Computers*, vol. 59, no. 11, pp. 1466–1479, 2010.

[8] D. J. Scales, M. Nelson, and G. Venkitachalam, "The design of a practical system for fault-tolerant virtual machines," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 4, pp. 30–39, 2010.

[9] F. Machida, M. Kawato, and Y. Maeno, "Redundant virtual machine placement for fault-tolerant consolidated server clusters," in *2010 IEEE Network Operations and Management Symposium-NOMS 2010*. IEEE, 2010, pp. 32–39.

[10] R. Stewart, P. Maier, and P. Trinder, "Transparent fault tolerance for scalable functional computation," *Journal of functional programming*, vol. 26, p. e5, 2016.

[11] F. Borges, L. Pacheco, E. Alchieri, M. F. Caetano, and P. Solis, "Transparent state machine replication for kubernetes," in *Advanced Information Networking and Applications: Proceedings of the 33rd International Conference on Advanced Information Networking and Applications (AINA-2019) 33*. Springer, 2020, pp. 859–871.

[12] H. Netto, C. Pereira Oliveira, L. d. O. Rech, and E. Alchieri, "Incorporating the raft consensus protocol in containers managed by kubernetes: an evaluation," *International Journal of Parallel, Emergent and Distributed Systems*, vol. 35, no. 4, pp. 433–453, 2020.

[13] H. V. Netto, A. F. Luiz, M. Correia, L. de Oliveira Rech, and C. P. Oliveira, "Koordinator: A service approach for replicating docker containers in kubernetes," in *2018 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2018, pp. 00 058–00 063.

[14] H. V. Netto, L. C. Lung, M. Correia, A. F. Luiz, and L. M. Sá de Souza, "State machine replication in containers managed by kubernetes," *Journal of Systems Architecture*, vol. 73, pp. 53–59, 2017.

[15] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm (extended version)," in *Proceeding of USENIX annual technical conference, USENIX ATC*, 2014, pp. 19–20.

[16] L. A. Vayghan, M. A. Saied, M. Toeroe, and F. Khendek, "Kubernetes as an availability manager for microservice applications," *arXiv:1901.04946*, 2019.

[17] ——, "A Kubernetes controller for managing the availability of elastic microservice based stateful applications," *Journal of Systems and Software*, vol. 175, p. 110924, 2021.

[18] B. Johansson, M. Rågberger, T. Nolte, and A. V. Papadopoulos, "Kubernetes orchestration of high availability distributed control systems," in *2022 IEEE International Conference on Industrial Technology (ICIT)*. IEEE, 2022, pp. 1–8.

[19] P. S. Junior, D. Miorandi, and G. 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)*. IEEE, 2022, pp. 26–33.

[20] M. Gundall, J. Stegmann, M. Reichardt, and H. D. Schotten, "Downtime optimized live migration of industrial real-time control services." [Online]. Available: http://arxiv.org/abs/2203.12935

[21] G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari, "Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 33–40.

[22] K. Govindaraj and A. 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)*. IEEE, 2018, pp. 83–90.

[23] J. Cao, M. Simonin, G. Cooperman, and C. Morin, "Checkpointing as a service in heterogeneous cloud environments," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 2015, pp. 61–70.

[24] S. Nadgowda, S. Suneja, N. Bila, and C. Isci, "Voyager: Complete container state migration," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 2137–2142.

[25] S. Oh and J. Kim, "Stateful container migration employing checkpoint-based restoration for orchestrated container clusters," in *2018 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 2018, pp. 25–30.

[26] B. Xu, S. Wu, J. Xiao, H. Jin, Y. Zhang, G. Shi, T. Lin, J. Rao, L. Yi, and J. Jiang, "Sledge: Towards efficient live migration of docker containers," in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*. IEEE, 2020, pp. 321–328.

[27] J. Schrettenbrunner, "Migrating pods in kubernetes," Ph.D. dissertation, Hochschule Darmstadt, 12 2020.

[28] L. Ma, S. Yi, and Q. Li, "Efficient service handoff across edge servers via docker container migration," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*. ACM, 2017, pp. 1–13.

[29] R. Stoyanov and M. J. Kollingbaum, "Efficient live migration of linux containers," in *High Performance Computing*, R. Yokota, M. Weiland, J. Shalf, and S. Alam, Eds. Springer International Publishing, 2018, vol. 11203, pp. 184–193, series Title: Lecture Notes in Computer Science.

[30] M. Terneborg, J. K. Ronnberg, and O. Schelen, "Application agnostic container migration and failover," in *2021 IEEE 46th Conference on Local Computer Networks (LCN)*. IEEE, 2021, pp. 565–572.

[31] D. Zhou and Y. Tamir, "Fault-tolerant containers using NiLiCon," in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2020, pp. 1082–1091.

[32] ——, "HyCoR: Fault-tolerant replicated containers based on checkpoint and replay," *arXiv:2101.09584*, 2021. [Online]. Available: http://arxiv.org/abs/2101.09584

[33] R. S. Venkatesh, T. Smejkal, D. S. Milojicic, and A. Gavrilovska, "Fast in-memory CRIU for docker containers," in *Proceedings of the International Symposium on Memory Systems*. ACM, 2019, pp. 53–65.

[34] X. Chen, J.-H. Jiang, and Q. Jiang, "A method of self-adaptive pre-copy container checkpoint," in *2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, 2015, pp. 290–300.

[35] S. Kannan, A. Gavrilovska, K. Schwan, and D. Milojicic, "Optimizing checkpoints using NVM as virtual memory," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE, 2013, pp. 29–40.

[36] R. H. Müller, C. Meinhardt, and O. M. Mendizabal, "An architecture proposal for checkpoint/restore on stateful containers," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*. ACM, 2022, pp. 267–270.

[37] W. Li, A. Kanso, and A. Gherbi, "Leveraging linux containers to achieve high availability for cloud services," in *2015 IEEE International Conference on Cloud Engineering*. IEEE, 2015, pp. 76–83.

[38] T. Louati, H. Abbes, and C. Cérin, "LXCloudFT: Towards high availability, fault tolerant cloud system based linux containers," *Journal of Parallel and Distributed Computing*, vol. 122, pp. 51–69, 2018.

[39] A. Reber, "Kubernetes and checkpoint/restore," 2023, Free and Open source Software Developers' European Meeting (FOSDEM) 2023. [Online]. Available: https://fosdem.org/2023/schedule/event/container_kubernetes_criu/