Google Summer of Code

PROPOSAL

Develop a Kubernetes operator that automates the management of container checkpoints

Mentors: Adrian Reber, Radostin Stoyanov, Prajwal S N

By
Parthiba Hazra

CRIU

kubernetes

Table of Contents

1. Abstract

Container checkpointing has emerged as a critical feature in Kubernetes for preserving the state of containers and facilitating tasks like live migration and fault recovery. However, the current implementation lacks essential management functionalities, such as limiting the number of checkpoints and automating their cleanup, which can lead to resource exhaustion and operational challenges. To address these issues, we propose enhancing our existing minimal Kubernetes operator dedicated to managing container checkpoints.
The enhanced Kubernetes operator will include advanced features such as garbage collection policies, including global policies, per-namespace, per-pod, and per-container policies. This will empower users to define how the operator should handle different containers based on their specific requirements. Additionally, the proposal suggests considering optional functionalities such as automatic checkpoint creation, replacement of old checkpoints, storage limit management, metrics collection, alerting, and event logging. These features would provide a comprehensive solution for container checkpoint management in Kubernetes, subject to further discussion and potential implementation if time permits.

2. Objectives

2.1. Garbage Collection Mechanism

The current functionality of the Kubernetes operator for managing container checkpoints focuses on controlling the number of checkpoint archives stored on the local disk per namespace/pod/container combination. This is achieved by limiting the maximum number of checkpoints per combination, with a default value of 10, and specifying the directory where checkpoint archives are stored.

To extend this functionality, our objective is to implement some more flexible policies which will empower users to provide different policies for different containers, pods and namespaces. Additionally, new aspects of garbage collection, such as orphaned checkpoint cleanup and time threshold cleanup policy, will be introduced.

Key aspects of the garbage collection mechanism include:

- *Dynamic Checkpoint Threshold Policy*: The operator will introduce a dynamic checkpoint threshold policy that provides flexibility to users to set different maximum numbers of checkpoints for individual containers, pods, and namespaces. Users can define specific threshold values for each resource, enabling fine-grained control over checkpoint management based on workload requirements and resource constraints.

- *Orphaned Checkpoint Cleanup Policy*: An orphaned checkpoint cleanup policy will be incorporated, allowing users to specify whether checkpoints associated with deleted pods or namespaces should be automatically deleted. When enabled, this policy ensures that orphaned checkpoints are promptly removed.
- *Time Threshold Policy*: The operator will support a time threshold policy that enables automatic deletion of checkpoints older than a user-defined time threshold. Users can specify the maximum retention period for checkpoints, ensuring that outdated or obsolete data is periodically purged to free up storage space and optimize resource utilization.

2.2. Automated Checkpoint Management(optional)

Automated checkpoint management will streamline administrative tasks and enhance operational efficiency within the Kubernetes operator. While the primary focus remains on the garbage collection mechanism, optional features will include:

- *Lifecycle Management*: The operator will automate the lifecycle management of checkpoints, including the creation, replacement, and deletion of checkpoints based on predefined policies and system constraints. This automation reduces manual intervention and minimizes the risk of human error, leading to more reliable and consistent checkpoint management.
- *Storage Quota Enforcement*: To prevent unchecked growth of checkpoint data and mitigate storage-related issues, the operator will enforce storage quotas or limits on checkpoint archives. By monitoring storage utilization and enforcing predefined quotas, the operator ensures that checkpoint data remains within acceptable bounds and avoids exceeding available storage capacity.
- *Metrics Collection and Alerting*: The operator will integrate with Kubernetes monitoring and alerting systems to collect relevant metrics related to checkpoint utilization, storage capacity, and garbage collection efficiency. By monitoring key performance indicators (KPIs) and triggering alerts based on predefined thresholds, the operator enables proactive management of checkpoint resources and facilitates timely intervention in case of anomalies or capacity constraints.
- *Event Logging and Auditing*: Comprehensive event logging and auditing capabilities will be incorporated into the operator to provide visibility into checkpoint management activities and ensure compliance with organizational policies and regulatory requirements. By maintaining detailed logs of checkpoint operations, including creation, modification, and deletion events, the operator enables traceability and accountability for checkpoint-related actions.

3. Implementation

3.1. Garbage Collection Policies

Adding new garbage collection policy involves three major steps:

- Update the `CRD` definition to store the new policies(modify the existing CRD).
- Refactor the `runGarbageCollector()` function to run the different types of garbage collection policies, probably we should run different *go routines* for the different types of policy.
- Integrate the new policy logic into the garbage collection mechanism of the Kubernetes operator.

3.1.1 Dynamic Checkpoint Threshold Policy

To implement the Dynamic Checkpoint Threshold Policy, we need to extend the existing `CheckpointRestoreOperatorSpec{}` struct to include fields for `GlobalPolicies` `ContainerPolicies`, `PodPolicies`, and `NamespacePolicies`. Each of these policies will allow users to define maximum checkpoint thresholds for specific containers, pods, and namespaces respectively. We also need to modify the garbage collection logic so that it first applies the container policy, then the pod policy, followed by the namespace policy, and finally the global policy.

```
type CheckpointRestoreOperatorSpec struct {
    CheckpointDirectory          string
`json:"checkpointDirectory,omitempty"`
    GlobalPolicies               GlobalPolicySpec
`json:"globalPolicy,omitempty"`
    ContainerPolicies            []ContainerPolicySpec
`json:"containerPolicies,omitempty"`
    PodPolicies                  []PodPolicySpec
`json:"podPolicies,omitempty"`
    NamespacePolicies            []NamespacePolicySpec
`json:"namespacePolicies,omitempty"`
}
```

```
func handleWriteFinished(ctx context.Context, event fsnotify.Event, spec
CheckpointRestoreOperatorSpec) {
    // Extract namespace, pod, and container information from the
checkpoint archive.
    // Use this information to determine which policy to apply.
    // Then, select the appropriate checkpoints to delete based on the
policy.

    var details checkpointDetails
    if err := getCheckpointArchiveInformation(log.FromContext(ctx),
event.Name, &details); err != nil {
        log.Error(err,
"runGarbageCollector():getCheckpointArchiveInformation()")
        return
    }

    // Apply container policy if present
     if len(spec.ContainerPolicies) > 0 {
        for _, containerPolicy := range spec.ContainerPolicies {
            if containerPolicy.Namespace == details.namespace &&
containerPolicy.Pod == details.pod && containerPolicy.Container ==
details.container {
                archivesToDelete :=
containerPolicy.getArchivesToDelete(details)
                deleteArchives(ctx, archivesToDelete,
containerPolicy.MaxCheckpoints)
                return
            }
        }
    }

    // Apply pod policy if present
     if len(spec.PodPolicies) > 0 {
        for _, podPolicy := range spec.PodPolicies {
            if podPolicy.Namespace == details.namespace && podPolicy.Pod ==
details.pod {
                archivesToDelete := podPolicy.getArchivesToDelete(details)
                deleteArchives(ctx, archivesToDelete,
podPolicy.MaxCheckpoints)
                return
            }
        }
    }

    // Apply namespace policy if present
    if len(spec.NamespacePolicies) > 0 {
        for _, namespacePolicy := range spec.NamespacePolicies {
            if namespacePolicy.Namespace == details.namespace {
                archivesToDelete :=
namespacePolicy.getArchivesToDelete(details)
                deleteArchives(ctx, archivesToDelete,
namespacePolicy.MaxCheckpoints)
                return
            }
        }
    }

    // Apply global policy if present
    // Here I keep the global policy unchanged but we can add new
    // policies like MaxCheckpointsPerPod, MaxCheckpointsPerNamespace.
    // I need to discuss this with mentors.
    if spec.GlobalPolicies.MaxCheckpointsPerContainer != nil {
        archivesToDelete :=
spec.GlobalPolicies.getArchivesToDelete(details)
        deleteArchives(ctx, archivesToDelete,
*spec.MaxCheckpointsPerContainer)
        return
    }
}

// Define getArchivesToDelete as a method of each policy struct
func (cp ContainerPolicySpec) getArchivesToDelete(details
checkpointDetails) map[int64]string {
      // Logic to fetch checkpoints eligible for deletion based on the
policy
}

func (pp PodPolicySpec) getArchivesToDelete(details checkpointDetails)
map[int64]string {
      // Logic to fetch checkpoints eligible for deletion based on the
policy
}

func deleteArchives(ctx context.Context, archivesToDelete map[int64]string,
maxCheckpoints int) {
    // Logic to delete older checkpoints until maxCheckpoints is satisfied
}
```

- We iterate through each policy type (container, pod, namespace, and global) and apply the corresponding policy if it matches the details extracted from the checkpoint archive.

- If a policy matches, we call the `getArchivesToDelete()` method to fetch the checkpoints eligible for deletion based on the policy.
- Then, we call the `deleteArchives()` function to delete older checkpoints until the maximum checkpoint threshold specified by the policy is satisfied. We do this under each policy loop, not at the end.

3.1.2 Orphaned Checkpoint Cleanup Policy

To implement the orphaned checkpoint cleanup policy, we'll extend the `CRD` definition of the `CheckpointRestoreOperator` to include a new boolean option `deleteOrphanedCheckpoint` in the `PodPolicySpec{}`(maybe also with the NamespacePolicySpec). This option will allow users to specify whether to delete orphaned checkpoints associated with deleted pods. The default value of `DeleteOrphanedCheckpoint` should be `false`.

We can Implement a pod watcher to handle deletion events:

```
// Watch Pods
podWatcher, err := clientset.CoreV1().Pods("").Watch(context.Background(),
metav1.ListOptions{})
if err != nil {
    panic(err.Error())
}
defer podWatcher.Stop()

for {
    select {
    case event := <-podWatcher.ResultChan():
        if event.Type == watch.Deleted {
            pod, ok := event.Object.(*corev1.Pod)
            if !ok {
                fmt.Println("Error: could not convert object to Pod")
                continue
            }
            fmt.Printf("Pod deleted: %s\n", pod.Name)
            // Print other details of the deleted pod
            fmt.Printf("Deleted Pod Details: %+v\n", pod)

            // Check if there's a matching policy for the deleted pod
            if matchingPolicy := getMatchingPodPolicy(pod,
spec.PodPolicies); matchingPolicy != nil {
```

```
                // Check DeleteOrphanedChekpoint set to true or not
                if matchingPolicy.DeleteOrphanedCheckpoint {
                    deleteOrphanedCheckpoints(pod)
                }
            }
        }
    }
}

func deleteOrphanedCheckpoints(pod *corev1.Pod) {
    // Logic to delete orphaned checkpoints associated with the deleted pod
}
```

3.1.3 Time-Based Checkpoint Retention Policy

To implement the time-based checkpoint policy, we'll extend the `CRD` definition of the `CheckpointRestoreOperator` to include a new field `timeThreshold` in the `ContainerPolicySpec{}`, `PodPolicySpec{}`, `NamespacePolicySpec{}`, and the `GlobalPolicySpec{}`. This field will allow users to specify a time threshold after which checkpoints associated with the respective resource will be deleted.

We have to implement a periodic check to delete checkpoints exceeding the time threshold:

```
func runTimeBasedCheckpointPolicy(spec CheckpointRestoreOperatorSpec) {
    // Run the policy check periodically
    ticker := time.NewTicker(24 * time.Hour) // Check once every 24 hours

    for range ticker.C {
        // Check and delete checkpoints exceeding time threshold
        checkAndDeleteCheckpoints(spec)
    }
}
func checkAndDeleteCheckpoints(spec CheckpointRestoreOperatorSpec) {
    // Logic to iterate through checkpoints and delete those exceeding the
time threshold
    // based on the policies specified in the spec
}
```

The frequency at which the policy checks for eligible checkpoints depends on factors such as the expected volume of checkpoints, the granularity of time thresholds specified, and the desired responsiveness of the cleanup process.

3.1.4 Potential structure of CheckpointRestoreOperator:

```
apiVersion: criu.org/v1
kind: CheckpointRestoreOperator
metadata:
  name: checkpointrestoreoperator-sample
spec:
  globalPolicy:
    maxCheckpointsPerContainer: 5
    maxCheckpointsPerPod: 9
    timeThreshold: 40d
    checkpointDirectory: /var/lib/kubelet/checkpoints
  namespacePolicies:
    - namespace: default
      maxCheckpoints: 30
    // ....
  podPolicies:
    - namespace: default
      pod: my-pod-1
      maxCheckpoints: 8
      deleteOrphanedCheckpoint: true
      timeThreshold: 30d
    // ....
  containerPolicies:
    - namespace: default
      pod: my-pod-1
      container: my-container-1
      maxCheckpoints: 6
   // ....
```

```
NOTE: In cases where conflicting policies are applicable to multiple
resource types (such as containers, pods, namespaces), a priority order
should be established to determine precedence. This priority order
should be based on the specificity of the policy settings.
```

3.2. Automated checkpoint management features(optional)

3.2.1 Automatic Checkpoint Creation

The implementation steps may involved:

- *Define CRD Fields*: Add fields in the Custom Resource Definition (CRD) to specify the automatic checkpoint creation threshold. This could include options such as time-based thresholds or event-based triggers.
- *Watch for Events*: Implement watchers to monitor relevant events such as pod creation, updates, or specific triggers based on use cases.
- *Trigger Checkpoint Creation*: Based on the defined threshold criteria, trigger the creation of checkpoints for pods or containers.

Potential Functions:

- *`watchPodEvents()`*: Function to watch for pod creation, updates, or deletions.
- *`triggerCheckpointCreation()`*: Function to initiate the creation of checkpoints based on predefined thresholds.

3.2.2 Storage Limit or Quota Management

The implementation steps may involved:

- *CRD Definition*: Add fields in the CRD to specify storage limits for checkpoints per container.
- *Checkpoint Creation Watcher*: Implement a watcher to monitor checkpoint creation events, either through Kubernetes Informers or by watching HTTP POST requests to the checkpoint creation endpoint(`https://localhost:10250/checkpoint/namespace/podId/container`)
- *Storage Limit Check*: Upon checkpoint creation event, verify if the storage limit for the container has been exceeded by comparing the size of existing checkpoints against the defined limit.

It's not a common practice to directly interfere with the Kubernetes API server's operations from an operator. Instead, the operator can handle the checkpoint creation event and we can proceed to delete the newly created checkpoint and through an error. This looks not a great way to handle the case, we can discuss it and reach out to a better solution.

3.2.3 Metrics Collection and Alerting

The implementation steps involved:

- Identify key metrics that indicate the health and performance of the checkpoint operator. This could include:
    - *`Disk space usage` in the directory where checkpoints are stored.*
    - *`Number of checkpoints` present.*
    - *`Checkpoint creation rate`.*
    - *`Frequency of garbage collection` runs.*
    - *`Errors encountered` during checkpoint management operations.*
- Integrate *`prometheus`* metrics library into operator codebase. Define the code to expose the metrics identified in the previous step and ensure that these metrics are accessible through HTTP endpoints or compatible interfaces for scraping by Prometheus.
- Configure alerting rules in the monitoring system (like Prometheus Alertmanager) based on the metrics we will define.

This feature will need the frequency of checkpoint creation and errors during checkpoint management, so we have to record the runs of *`runGarbageCollector()`* and errors encountered each time in the program.

3.2.4 Event Logging

The implementation steps involved:

- Event Logging Structure: Define a structured format including
    - *Type of operation (e.g., creation, modification, deletion)*
    - *Timestamp*
    - *Resource identifier (e.g., namespace, pod, container)*
    - *Additional context or metadata*

```go
type LogEntry struct {
    Operation string
    Timestamp time.Time
    Resource  string
    Metadata  map[string]string
}
```

- Integrate Logging Library: we should choose a logging library supporting structured logging (e.g., Logrus, Zap).
- Emit log messages at relevant code points.
- Log Rotation and Retention: Implement policies to manage log files' size and lifespan. (Example - using Logrus with lumberjack for log rotation)

3.3. References

- https://kubernetes.io/docs/concepts/extend-kubernetes/operator/
- https://kubernetes.io/docs/tasks/extend-kubernetes/custom-resources/custom-resource-definitions/
- https://kubernetes.io/docs/concepts/architecture/garbage-collection/
- https://github.com/kubernetes/client-go
- https://github.com/checkpoint-restore/checkpoint-restore-operator
- https://kubernetes.io/docs/reference/node/kubelet-checkpoint-api/
- https://kubernetes.io/blog/2022/12/05/forensic-container-checkpointing-alpha/
- https://kubernetes.io/blog/2023/03/10/forensic-container-analysis/
- https://github.com/kubernetes/kubernetes/pull/115888
- https://github.com/kubernetes/enhancements/issues/2008

4. Timeline

4.1. Before May 1

- Research and understand essential features for a Kubernetes operator specifically in the context of garbage collection mechanisms, including common functionalities and best practices.
- Engage with mentors to discuss feature priorities, implementation strategies, and any unresolved technical challenges.

4.2. May 1 - May 26 (community bonding period)

- Familiarize myself with the existing codebase and work style of the team.
- Set up the development environment and ensure all necessary tools and dependencies are installed.
- Begin exploring code samples, documentation, and relevant resources to gain a deeper understanding of the project scope and objectives.

4.3 May 27 - July 12 (phase I)

- **May 27 - June 3:** Extend CRD definition and start the implementation to support dynamic checkpoint threshold policies.

- **June 4 - June 17:** Finish the Implementation of dynamic checkpoint threshold policy handling in the operator, including container, pod, and namespace policies.
- **June 18 - June 24:** Add test cases and validate dynamic checkpoint threshold policy functionality. Refactor codebase as needed.
- **June 25 - July 1:** Begin implementing orphaned checkpoint cleanup policy.
- **July 2 - July 7:** Complete implementation of orphaned checkpoint cleanup policy along with necessary test cases.
- **July 8 - July 12(midterm Evaluation):** Engage in discussions with mentors to review the project status and outline plans for Phase II.

4.4 July 13 - Aug 26 (phase II)

- **July 13 - July 19:** Start Implementing time-based checkpoint retention policy.
- **July 20 - July 26:** Complete adding new functionalities along with test cases.
- **July 27 - Aug 2**: Reserved for any unresolved challenges during implementation of garbage collection mechanisms.

  If everything is on track, the implementation of garbage collection policies should be finished here and I will engage in discussions with mentors to collectively decide which tasks to prioritize next, whether it's working on optional features or focusing on documentation..
- **Aug 3 - Aug 16:** Based on the discussion we can start work on the optional features or documentation of newly added garbage collection policies.
- **Aug 17 - Aug 26:** Finalize documentation, including setup instructions, usage guidelines, also add all information about the checkpoint restore operator to the CRIU website along with example use cases.
- **Aug 26 - Sep 2(final evaluation):** Use this time to address any unforeseen delays or emergencies that may have occurred during the project timeline.

4.5 After August 26 (Post-Completion Activities)

- Conduct a thorough evaluation of the project outcome and quality, including functionality, performance, and adherence to project goals and requirements.
- Gather feedback from mentors, users, and the community to assess the impact and usability of the checkpoint management operator.
- Identify areas for further enhancement or optimization based on feedback received and lessons learned during the GSoC program.
- Prioritize and plan future development efforts to address any identified issues, implement new features, or improve existing functionalities.

5. About Me

5.1 Who am I

Name: Parthiba Hazra

Email: parthibahazra@gmail.com

Github: https://github.com/Parthiba-Hazra

matrix/gitter: @parthiba:matrix.org

Linkedin: https://www.linkedin.com/in/parthiba-hazra-a6132023a/

Twitter: https://twitter.com/Parthib38453353

University: Maulana Abul Kalam Azad University of Technology, West Bengal

Location: Kolkata, India

Time zone: IST (India Standard Time) UTC/GMT +5:30 hours

As an undergraduate CS student from India, I've been actively engaged in Golang and Kubernetes for two years, contributing to open-source projects for more than a year. I had the privilege of participating as an LFX mentee under CNCF(konveyor) during the LFX mentorship program 2023 term-03. Currently, I'm learning about distributed systems and Rust, while exploring Kubernetes in depth. My passion lies in DevOps practices and pushing the boundaries of cloud-native technologies.

5.2 Open-Source Contribution History

Criu -

- **Add support for list command to get the list of container checkpoints #115**
- **Extend inspect output format with json #113**

Konveyor -

- **LFX: Extend use-case of detecting deprecated Kubernetes API usage #441**
- **Add an option to let the user provide the entire key PEM data as the answer #1078**
- **Added jaeger to images so it runs out of the box #297**
- **Do not generate violations when there are no incidents #281**

Buildpacks -

- Add feature to pull-policy for pulling images according to user specified pulling interval (e.g., hourly, daily, weekly, interval=5d4h30m) #2075
- feat: added a mac-address flag to build option, so that user can manually set the MAC-address. #2028
- Add an option to retag rather than replacing the target image while rebasing #2023
- Add support to override UID for container #2017

Keda -

- Update prometheus_metrics_test.go #4437

5.3 Commitments and Obligation

- I will be able to work full-time (a minimum of 40 hours a week).
- I have end-of-semester exams in the middle of June for one week, during which I plan to dedicate a minimum of 25 hours. I will make up for any backlog in the subsequent week.
- I have no further obligations so I'll be able to work for a minimum of 40 hours a week till the end of the program.