UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS REITOR JOÃO DAVID FERREIRA LIMA
PROGRAMA DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

Gian Augusto Ortiz

IMPLEMENTAÇÃO DE UM SERVIÇO DE CHECKPOINT PARA ARQUITETURA DE CHECKPOINT/RESTORE DE STATEFUL CONTAINERS NO KUBERNETES

Florianópolis, Santa Catarina – Brasil
2023

Gian Augusto Ortiz

IMPLEMENTAÇÃO DE UM SERVIÇO DE CHECKPOINT PARA ARQUITETURA DE CHECKPOINT/RESTORE DE STATEFUL CONTAINERS NO KUBERNETES

Trabalho de Conclusão de Curso submetido ao Programa de Graduação em Ciências da Computação da Universidade Federal de Santa Catarina para a obtenção do Grau de Bacharel em Ciências da Computação.
Orientador: Odorico Machado Mendizabal, Dr.

Florianópolis, Santa Catarina – Brasil
2023

Notas legais:

Não há garantia para qualquer parte do software documentado. Os autores tomaram cuidado na preparação desta tese, mas não fazem nenhuma garantia expressa ou implícita de qualquer tipo e não assumem qualquer responsabilidade por erros ou omissões. Não se assume qualquer responsabilidade por danos incidentais ou consequentes em conexão ou decorrentes do uso das informações ou programas aqui contidos.

Catalogação na fonte pela Biblioteca Universitária da Universidade Federal de Santa Catarina.
Arquivo compilado às 23:37h do dia Monday 11th December, 2023.

Gian Augusto Ortiz
Implementação de um Serviço de Checkpoint para arquitetura de Checkpoint/Restore de Stateful Containers no Kubernetes / Gian Augusto Ortiz; Orientador, Odorico Machado Mendizabal, Dr. – Florianópolis, Santa Catarina – Brasil, 13 de novembro de 2023.
74 p.

Trabalho de Conclusão de Curso – Universidade Federal de Santa Catarina, INE – Departamento de Informática e Estatística, CTC – Centro Tecnológico, Programa de Graduação em Ciências da Computação.

Inclui referências

1. Checkpoint/Restore, 2. Conteiner Stateful, 3. Kubernetes, 4. Event Sourcing, 5. Tolerância a falhas, I. Odorico Machado Mendizabal, Dr. II. Programa de Graduação em Ciências da Computação III. Implementação de um Serviço de Checkpoint para arquitetura de Checkpoint/Restore de Stateful Containers no Kubernetes

CDU 02:141:005.7

Gian Augusto Ortiz

IMPLEMENTAÇÃO DE UM SERVIÇO DE CHECKPOINT PARA ARQUITETURA DE CHECKPOINT/RESTORE DE STATEFUL CONTAINERS NO KUBERNETES

Este(a) Trabalho de Conclusão de Curso foi julgado adequado(a) para obtenção do Título de Bacharel em Ciências da Computação, e foi aprovado em sua forma final pelo Programa de Graduação em Ciências da Computação do INE – Departamento de Informática e Estatística, CTC – Centro Tecnológico da Universidade Federal de Santa Catarina.

Florianópolis, Santa Catarina – Brasil, 13 de novembro de 2023.

Lúcia Helena Martins Pacheco, Dr.
Coordenador do Programa de Graduação
em Ciências da Computação

Banca Examinadora:

Odorico Machado Mendizabal, Dr.
Orientador
Universidade Federal de Santa
Catarina – UFSC

Márcio Bastos Castro, Dr.
Universidade Federal de Santa Catarina –
UFSC

Frank Augusto Siqueira, Dr.
Universidade Federal de Santa Catarina –
UFSC

Este trabalho é dedicado a todos que encontraram uma pedra,
em seu caminho e passaram por ela.

AGRADECIMENTOS

Os agradecimento principais deste trabalho de conclusão de curso são a minha família, principalmente aos meus pais, Laércio Aparecido Ortiz e Rosana Aparecida Gonçalves Ortiz, que dentro do possível proveram tudo que podiam para que eu trilhasse minha jornada até aqui, dentre muitos problemas que ocorreram pelo caminho e seu amor incondicional. Agradecimento aos meus irmão, João Victor Ortiz e Luan Henrique Ortiz, que foram meus primeiros amigos e que carrego fortes lembranças de momentos bons nesta vida. Também quero agradecer a minha namorada, Beatriz Alves dos Santos, que não me deixou pensar em nenhum momento que eu era incapaz de terminar a graduação durante o tempo que esteve comigo, neste ponto também quero destacar meu pai que certa vez me pediu para que eu não desistisse e eu segui seu pedido. Todos da minha família estiveram do meu lado este ano me apoiando de alguma forma e me alegrando sempre que puderam.

Agradecimentos também devem ir aos meus amigos de diversos grupos, em principal os BMs, os quais a UFSC me deu e que tornaram meus dias melhores com discussões banais e às vezes filosóficas e profundas, ao grupo de amigos Capivari Lighters, que embora não nos vemos com tanta frequência tenho certeza que foram essenciais para eu me tornar quem eu sou hoje e quem eu serei amanhã. Aos outros amigos que não se encontram na união destes dois também gostaria de agradecer pelos momentos que passamos durante o ano de 2023 e me livraram de muita dor de cabeça através de risadas e bons momentos, como as periódicas idas à Lanchonete Quebra Gelo nas quintas feiras.

Agradeço também aos meus professores durante toda a minha jornada de conhecimento e formação. Estes me ensinaram muitas coisas que ao longo do tempo posso ter julgado inútil, mas que no final me serviram para muitas coisas dentro e fora das instituições de ensino. Mas, o maior aprendizado que levo deste processo foi o aprendizado em como aprender, sinto que fui fornecido da melhor formação possível para que eu possa aprender o que for necessário para minha vida.

“Sorte é o que acontece quando a preparação encontra oportunidade.”
Seneca

“Nenhum homem jamais foi sábio por acaso.”
Seneca

RESUMO

Em sistemas distribuidos, percebemos a alta difusão da utilização de virtualização leve, através de contêineres, para criação de implantação de novas aplicações, principalmente entre microsserviços e computação distribuída em serviços em nuvem. Entretanto, este método possui alguns desafios para prover alta disponibilidade dos serviços. Orquestradores de contêneires resolvem muitos dos desafios, como a escalabilidade, a distribuições e administração de contêneires em *clusters*. Porém, alguns desafios persistem, como o tema deste trabalho que é a alta disponibilidade para serviços em contêineres do tipo *Stateful*. Estes tipos de serviços necessitam que o estado seja persistido em memória e, em uma falha, seja possível reiniciar o serviço para o mesmo estado anterior à falha. Neste trabalho, utilizamos técnicas de *Checkpoint/Restore* para realizar o salvamento do estado de um serviço e posteriormente realizar a restauração do serviço, em caso de uma falha. Juntamente com estas técnicas, também aplicamos as técnicas do padrão de projeto de *Event Sourcing* para prover diferentes tipos de recuperação no trabalho. Ainda, focamos em prover uma solução que funcione de maneira transparente no orquestrador de contêneires Kubernetes, que é atualmente o mais utilizado. Por fim, conseguimos prover um serviço de *Checkpoint/Restore* para Kubernetes em contêineres do tipo *Stateful* através de técnicas do padrão de projeto *Event Sourcing* utilizando uma arquitetura agnóstica a tecnologias. Também conseguimos abordar uma forma de realizar o mesmo salvamento do estado e posterior recuperação utilizando CRIU e buildah para construção de imagens de recuperação do estado seguindo o padrão do *Open Container Initiative*.

Palavras-chaves: Checkpoint/Restore. Conteiner Stateful. Kubernetes. Event Sourcing. Tolerância a falhas.

ABSTRACT

The world of distributed systems is seeing the increase in the usage of lightweight virtualization, also known as containers, to create applications, like microservices distributed in cloud systems. But, the usage of containers for distributed system has many challenges on the high availability of the service. Container orchestrators, like Kubernetes, solve this challenges by providing tools to manage containers, scale them and distribute them. Although, the high availability of Stateful containers has other challenges than Stateless containers, like the fault tolerance on achieving the same state that the previous failed application, this is the focus of this work. To achieve the same state as the previous failed container, we use Checkpoint/Restore techniques and the Event Sourcing pattern to checkpoint the state of the application and when it fails we restore it to the previous state. The usage of the Event Sourcing pattern is to investigate a new way to achieve the Checkpoint/Restore, also we show how to achieve the same result using traditional Checkpoint/Restore on Linux with CRIU. This work creates a framework for Kubernetes for fault tolerance with Checkpoint/Restore using Event Sourcing.

Keywords: Checkpoint/Restore. Stateful Container. Kubernetes. Event Sourcing. Fault tolerance.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE CÓDIGOS

LISTA DE ABREVIATURAS E SIGLAS

OCI	Open Container Initiative
CRI	Container Runtime Interface
API	Application Programming Interface
GPU	Unidade de Processamento Gráfico
HTTP	Hypertext Transfer Protocol
CRIU	Checkpoint/Restore in Userspace
DNS	Domain Namespace
CPU	Central Processing Unit
TCP	Transmission Control Protocol

CONTEÚDO

1 INTRODUÇÃO

Com o crescimento de técnicas de utilização de microsserviços e computação em nuvem, se criou uma alta necessidade para a alta disponibilidade dos serviços (VAYGHAN et al., 2021). A virtualização leve através da utilização de contêineres, desenvolvidos em cima das funcionalidades do Linux, como *cgroups* e *namespaces*, que possibilitam a limitação e priorização de recursos sem máquinas virtuais e a isolação completa de processos, redes e arquivos, respectivamente, possibilitou uma melhoria no processo de disponibilização destes microsserviços na nuvem (LAADAN; HALLYN, 2010). Desta forma, para facilitar o processo de gerenciamento e comunicação entre diversos contêineres de microsserviços na nuvem, aplicações com este intuito foram criadas (VAYGHAN et al., 2021), chamadas de orquestradores de contêineres, entre elas a mais popular é o Kubernetes(KUBERNETES, 2023i).

Em geral, os orquestradores de contêineres proveem um grande ferramental para a replicação, recuperação e manutenção de alta disponibilidade para os contêineres, suprindo a maioria das necessidades que aplicações sem estado de memória interna, do tipo *Stateless*, possuem (VAYGHAN et al., 2021). Entretanto, quando precisamos de aplicações que possuam estado em memória, aplicações *Stateful*, e, que este estado seja importante para as futuras execuções da aplicação, falta suporte nos orquestradores de contêineres, principalmente em adoção das práticas de *Checkpoint/Restore* (MÜLLER; MEINHARDT; MENDIZABAL, 2022) para o melhoria do tempo da recuperação de contêineres com falhas. Este tipo de técnica permite salvar o estado de memória em uma nova imagem para recuperá-la mais tarde. Embora, orquestradores de contêineres possuam suporte para aplicações com estado em armazenamento de dados persistente, como o StatefulSet no Kubernetes, não existe suporte nativo para o *Checkpoint* de estado em memória da aplicação(TRAN; VU; KIM, 2022).

Desta forma, este trabalho desenvolve um serviço transparente para realizar o *Checkpoint/Restore* de aplicações *Stateful* executando em contêineres no Kubernetes. Este serviço possibilita a tolerância a falhas das aplicações com estado, onde, quando uma falha ocorrer a aplicação será reiniciada ao último ponto de execução antes da falha.

1.1 OBJETIVOS

1.1.1 Objetivo Geral

Desenvolver um serviço transparente para realização de *Checkpoint/Restore* de aplicações de contêineres *Stateful* que execute de maneira nativa no orquestrador de contêineres Kubernetes.

1.1.2 Objetivos Específicos

- Desenvolver um serviço transparente para *Checkpoint/Restore* de aplicações *Stateful* executando no orquestrador de contêineres Kubernetes com foco em memória volátil das aplicações.
- Coletar métricas da performance da solução implementada para garantir eficiência no *Checkpoint/Restore*, comparando com execução sem a aplicação.
- Identificar e avaliar classes de problema em que a ferramenta não tem adequação para utilização.

1.2 ORGANIZAÇÃO DO TRABALHO

O restante desse trabalho está organizado da seguinte maneira: o Capítulo 2 apresenta alguns conceitos teóricos e técnicos de algumas ferramentas e paradigmas diretamente relacionados ao desenvolvimento dessa monografia. Na sequência, o Capítulo 3 apresenta trabalhos relacionados com o nosso tema e o que foi desenvolvido. O Capítulo 4 aborda a definição do nosso serviço, sua arquitetura e quais tecnologias serão utilizadas para prover o *Checkpoint/Restore*. No Capítulo 5 apresentamos a implementação do serviço, como ele foi feito, o que enfretamos de dificuldade na implementação, quais desafios foram resolvidos, quais desafios não foram resolvidos, abordamos duas perspectivas diferentes, a da implementação com técnicas de *Event Sourcing* e outra com a implementação com a utilização de CRIU. No Capítulo 6 apresentamos as análises experimentais que realizamos sobre o serviço, coletando métricas e entendendo os resultados obtidos. Por fim, no Capítulo 7 são apresentadas as considerações finais reforçando os resultados obtidos, apresentando partes que não conseguimos alcançar do trabalho e cobrindo possíveis trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Esta seção apresenta a fundamentação teórica por trás do desenvolvimento do trabalho, dando um grau de entendimento sobre temas que abordamos. São abordados assuntos sobre contêineres, orquestração de contêineres, *checkpoint/restore* e serviços *stateful* e *stateless*.

2.1 CONTÊINERES

Contêineres são um conjunto de funcionalidades de Kernel do Linux, como *namespaces* e *cgroups* (LAADAN; HALLYN, 2010) para virtualização leve do sistema (CHEN, 2015/10) (KUBERNETES, 2023i). Diferentemente de máquinas virtuais que precisam de muitos mecanismos de virtualização, através principalmente do *namespaces* e *cgroups* conseguimos limitar e priorizar recursos, isolar processos, redes e arquivos entre diferentes processos que estarão executando em contêineres separados (KUBERNETES, 2023i). Embora contêineres tenham sido criados inicialmente para o Linux atualmente já existem solução de conteinerização para outros sistemas (LAADAN; HALLYN, 2010).

Para melhorar o fluxo de criação de contêineres algumas ferramentas foram criadas, as chamadas *container runtimes* (*runtime* de contêineres), através destas é possível realizar todo o processo de criação de uma imagem de contêiner, definição da execução da imagem (programa a ser executado), inicialização, finalização e remoção do contêiner. Estas utilizam o processo de isolação do sistema que seria feito manualmente com as chamadas de sistema para *cgroups* e *namespaces*.

Entre as *runtimes* de contêineres temos algumas mais utilizadas, o Linux Containers, a primeira a ser criada e com funcionalidades rudimentares quando comparadas às outras, o cri-o que é uma *runtime* leve otimizada para Kubernetes, o containerd que provê simplicidade, dando ao usuário apenas o que é necessário para utilização da *runtime* e o docker que é o mais utilizado e construído em cima do containerd, provendo mais funcionalidades do que o primeiro. O orquestrador de contêineres Kubernetes utiliza o containerd como *runtime* padrão e disponibiliza a CRI, interface de *runtime* de contêineres, para definir uma interface que *runtimes* devem seguir para serem utilizadas no Kubernetes, cri-o, containerd e docker seguem a implementação da CRI.

2.1.1 containerd

O containerd é uma *runtime* de contêineres implementada com o foco em ser leve e simples de utilizar, portanto não possui recursos muito sofisticados, mas apenas a base para se trabalhar com contêineres no ambiente Linux. Ela é construída como um

frontend para o runc, que é de fato a *runtime* de contêineres utilizada como *backend*. Desta forma, o containerd provê uma API para acesso à *runtime* de contêineres runc (CONTAINERD, 2023).

O containerd conta com uma interface de linha de comando que permite trabalhar de maneira ágil com os contêineres. Além disso, provê uma API, que pode ser utilizada com linguagens de programação, como Go, para estender as funcionalidades a partir de chamadas, permitindo o desenvolvedor criar aplicações em cima das chamadas à API.

2.1.2 cri-o

cri-o é uma *runtime* de contêineres projetada para ser leve e otimizada para o uso específico no orquestrador de contêineres Kubernetes, não é possível utilizar cri-o fora do Kubernetes(CRI-O, 2023). Ela gera imagens de contêiner compatíveis com *Open Container Initiative* (OCI). Como o containerd ela funciona como um *frontend* para outras *runtimes* de contêineres que servem como *backend*. Mas, ela pode funcionar tanto com contêineres do runc quanto com contêineres do Katana. Atualmente o Kubernetes fornece em sua versão alpha 1.25 a funcionalidade de *Container Checkpoint* para esta *runtime* (KUBERNETES, 2023d), embora, não haja suporte suficiente para realizar todo o processo de *Checkpoint/Restore* através das funcionalidades providas.

2.2 ORQUESTRAÇÃO DE CONTÊINERES

Orquestração de contêineres é o nome dado para a administração de funcionalidades que os contêineres não possuem por padrão em ambientes de produção. É possível utilizar contêineres localmente, instanciando cada um deles separadamente e administrando-os através de comandos de linha. Mas, quando se faz necessário a execução destes em ambientes de produção algumas funcionalidades são necessárias, como, reinicialização dos contêineres quando estes falham, escalar os contêineres em réplicas, limitar e definir quantidades de recursos de cada contêiner, permitir a comunicação entre diferentes contêineres ou isolar a comunicação para um grupo específico, possibilidade de organizar os contêineres em diversas máquinas operando em modo *cluster*.

Dado o contexto de orquestração de contêineres tudo isto poderia ser feito através de ferramentas distintas que realizam cada uma das tarefas, ou até mesmo por ferramentas escritas pelo usuário. Mas, algumas destas ferramentas estão encapsuladas em aplicações que permitem realizar a orquestração de contêineres. Entre as principais temos:

- docker-swarm: ferramenta desenvolvida pela própria equipe do docker que permite escalar os contêineres, executar contêineres em modo de *cluster*, reinicial-

ização automática dos contêineres, descobrimento de serviços, comunicação de rede entre os contêineres, funciona necessariamente com a *runtime* de contêineres docker (DOCKER, 2023).

- Kubernetes: um *framework*, como se descreve em sua documentação, que permite a orquestração de contêineres, bem completo, permite todas as funcionalidades abordadas no docker-swarm, mas também provê funcionalidades muito interessantes para extensão. Através das extensões podemos utilizar a API do Kubernetes para estender funcionalidades nativas e criar novas funcionalidades, como por exemplo, para *Checkpoint/Restore* de contêineres. Pela alta extensibilidade do *framework*, alta adoção do Kubernetes pela indústria (CORMIER, 2023) e também pela possibilidade de trabalhar com diversas *runtime* de contêineres, desde que estas implementem a CRI (*Container Runtime Interface*), iremos utilizar o Kubernetes neste trabalho.

- OpenShift RedHat: uma ferramenta desenvolvida pela RedHat com base no Kubernetes. A principal diferença é a utilização do cri-o como *runtime* de contêineres. Entretanto, como não é tão adotada pela indústria e constrói ferramentas em cima das funcionalidades do Kubernetes (REDHAT, 2023), foi descartada para a elaboração deste trabalho.

2.2.1 Kubernetes

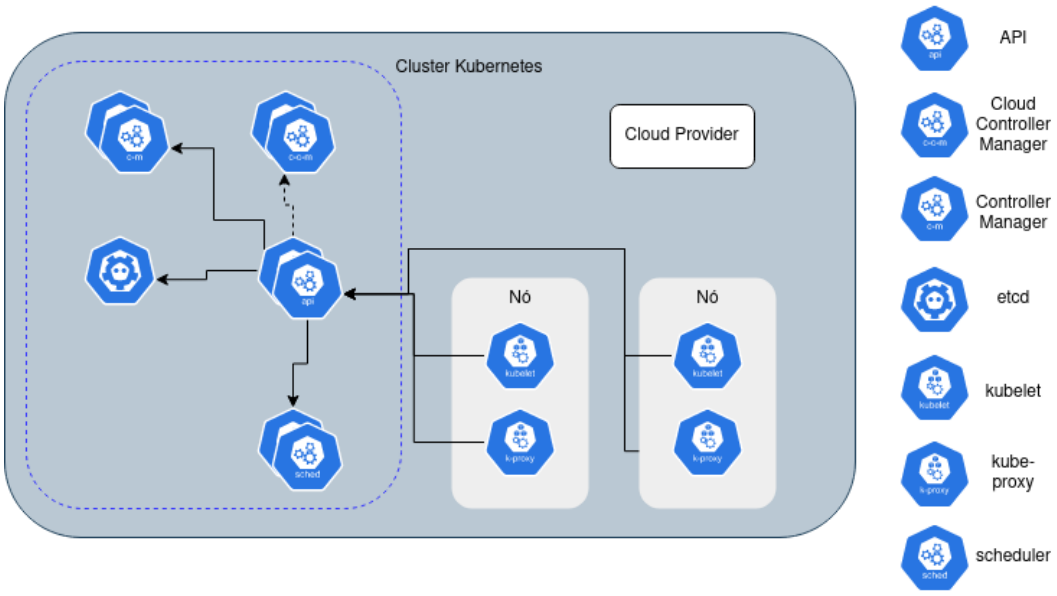
Kubernetes como já dito é um *framework* para a orquestração de contêineres em *cluster*. Permite o escalonamento de contêineres, a administração de recursos utilizados, a reinicialização automática dos contêineres, a comunicação entre diferentes contêineres, escolha de modo de distribuição dos contêineres nos nós do *cluster*, modo de *deployment* das novas versões dos contêineres, escolha da *runtime* de contêiner e extensibilidade das funcionalidades padrões da API do Kubernetes (KUBERNETES, 2023i).

Para tornar o Kubernetes mais extensível possível, ele é composto por diversas aplicações que juntas executam o que chamamos de um *cluster* Kubernetes. Cada uma das aplicações administra uma funcionalidade do *cluester*. Um nó do *cluster* do Kubernetes pode ou não possuir o *control plane*, que permite a administração dos nós e também dos contêineres que executam no nó (KUBERNETES, 2023f). Os componentes do Kubernetes estão ilustrados na Figura 1.

Como visto na figura, temos diversos componentes que juntos trabalham para prover as funcionalidades do *framework* do Kubernetes:

- kube-apiserver: é a API do Kubernetes, onde estão definidos os tipos de recursos que o *cluster* possui, realiza a interface de acesso aos dados que ficam

Figura 1 – Componentes que compõe a arquitetura do Kubernetes e comunicações entre si.



Fonte: Adaptado de (KUBERNETES, 2023i).

armazenados em banco de dados. Permite a criação e administração dos recursos através desta API através de acesso de dados CRUD (*Create*, *Retrieve*, *Update*, *Delete*).

- etcd: não é desenvolvido pelo Kubernetes, mas, é utilizado para armazenar todos os metadados do *cluster*, como por exemplo, em que nó um contêiner está sendo executado e quais outros contêineres podem acessar via HTTP este contêiner. Neste banco de dados de chave/valor também são armazenados o estado desejável dos recursos que o Kubernetes está administrando.

- kube-scheduler: aplicação que cuida da atribuição de um nó para um contêiner. Para escolher o melhor nó se faz uso de diversas políticas, como a definição dos recursos necessários do contêiner, é necessário que o nó possua os recursos e que os recursos gerais sejam utilizados da melhor forma no *cluster*, ou também se o nó possui alguma peculiaridade em anotação, como, por exemplo, uma GPU, o que é chamado de *node affinity*.

- kube-controler-manager: todos os recursos da API do Kubernetes possuem um *controller*(controlador) que controla a criação e administração destes recursos. Nesta aplicação, estão definidos a maioria dos controladores padrões do Kubernetes. Quando necessitamos estender a API, geralmente escrevemos novos controladores que executam separadamente destes.

- cloud-controller-manager: aplicação específica para *clusters* que estejam sendo executados em serviços de nuvem, permitindo que o serviço defina como serão

feitos os roteamentos de dados de rede e outras particularidades de acesso de rede na nuvem, por exemplo, abertura de portas e exposição das mesmas, protocolos de redes, DNS, entre outros.

Nem todo nó precisa executar o *control plane*, mas, todos os nós que executam contêineres do usuário precisam contar com três aplicações, o kubelet, o kube-proxy e a *runtime* de contêiner, como visto na Figura 1. O kubelet é responsável por administrar os contêineres da máquina e assegurar que o contêiner é executado conforme a definição de um Pod. Desta forma, é possível saber quais contêineres não são mais necessários através das definições do estado almejado por cada recurso. O kube-proxy implementa regras de rede para o nó que permitem que os contêineres que este executa sejam acessíveis aos outros nós do *cluster*, através de registros DNS internos ao *cluster*. Por fim, a *runtime* de contêineres é a responsável por de fato executar os contêineres do nó. O kubelet se comunica com a *runtime* de contêineres para que seja possível realizar a administração dos contêineres, realizando ações que seriam feitas manualmente por um usuário fora de um orquestrador, como escalar um contêiner.

2.2.1.1 Pod

O Pod é o conceito fundamental do Kubernetes. Um Pod é a encapsulação de diversos contêineres de uma aplicação com significado lógico para estarem executando juntos. Dentro de um Pod, as ferramentas do Linux, como cgroups e namespaces, isolam os contêineres executando no Pod em um contexto compartilhado, permitindo a comunicação entre os contêineres e compartilhamento de recursos. O Pod é definido, como todo recurso do Kubernetes, através de um arquivo YAML ou JSON, contendo a imagem que será utilizada, os recursos e também os contêineres que serão utilizados (KUBERNETES, 2023j). O próprio Kubernetes executa algumas de suas aplicações internas em Pods, como o próprio kube-apiserver. Toda aplicação que executa no Kubernetes executa através de um Pod, que é o recurso mínimo para que possamos utilizar o Kubernetes.

2.2.1.2 Deployment/StatefulSet

Para administrar Pods no Kubernetes utilizamos o Deployment ou o StatefulSet. A principal diferença entre os dois é que o último permite a utilização de volumes persistentes para armazenar dados entre reinicializações dos Pods. Eles permitem definir os estado que queremos que nosso Pods permaneçam, este estado está relacionado principalmente ao número de réplicas deste Pod e também da versão de imagem de contêiner que eles devem estar. O Deployment ou StatefulSet lida com toda a parte de criar novos Pods e seus contêineres, de reiniciá-los, pará-los ou atualizá-los através de um controlador do kube-controller-manager (KUBERNETES, 2023c). Estes recursos

permitem que o Kubernetes possa administrar as aplicações que ele executa através de outros recursos, como o ReplicaSet, que é a definição da quantidade de réplicas que deve ser mantida para cada um dos serviços.

2.2.1.3 Volumes

Para o armazenamento de dados persistentes, o Kubernetes utiliza o conceito de Volumes, que é uma abstração sobre uma localização de dados persistentes. Existem diferentes tipos de volumes que permitem que sejam utilizadas implementações de diferentes provedores de serviços em nuvem ou até mesmo de discos rígidos das máquinas em que se está executando o *cluster* do Kubernetes (KUBERNETES, 2023k).

Os Volumes do Kubernetes funcionam de maneira diferente dos do Docker, sendo que no último é mais desorganizado, onde, é apenas disponível para o disco local ou outro contêiner. Os Volumes do Kubernetes permitem que seja compartilhando um volume entre diversos Pods através de permissões dos volumes que são solicitadas para os Pods como PersistentVolumeClaim para os PersistentVolumes (KUBERNETES, 2023a). Nas reinicializações de Pods com PersistentVolumeClaims é assegurado que eles receberão acesso ao mesmo volume novamente, persistindo os dados armazenados da aplicação. Isto não permite o salvamento de dados em memória, que não são salvos em persistência de dados, mas, no contexto da aplicação.

2.2.1.4 Controllers

Todos os recursos definidos no Kubernetes são controlados pelos chamados *Controllers*. Estes controladores ouvem a mudanças nos estados dos recursos a partir de monitoramento da API do Kubernetes, kube-apiserver, e executam passos para atingir os objetivos de estado que os recursos na API necessitam. O Kubernetes provê alguns recursos padrões, como Pods, Jobs e Volumes e todos estes possuem um controlador para controlar o estado desejado de cada um dos recursos (KUBERNETES, 2023b).

Também podemos adicionar recursos através de controladores personalizados, estes são criados em qualquer linguagem de programação e também se comunicam com o kube-apiserver. Com isso, é possível estender, por exemplo, a funcionalidade do StatefulSet Controller, a partir de monitoramento dos recursos que ele administra. Com isso, podemos, por exemplo, adicionar novas funcionalidades para monitorar Pods e criar funcionalidades como servidores de *reverse proxy* (KUBERNETES, 2023b).

2.3 SERVIÇOS STATEFUL E STATELESS

De modo geral, quando executamos uma aplicação ou um serviço em produção, temos dois tipos de possibilidade de aplicações. Temos as aplicações que não necessitam saber do estado de execução da aplicação, como, por exemplo, a instrução que estava sendo executada. Aplicações que precisam apenas acessar um banco de dados e servir dados, como uma API HTTP, não possuem necessidade de permanecer o estado da memória ou mesmo de saber a instrução sendo executada em determinado momento. Chamamos este tipo de aplicação/serviço de *Stateless*, ou seja, sem estado. A maioria das aplicações desenvolvidas para microsserviços são desenvolvidas através deste padrão (VAYGHAN et al., 2021).

No caso de aplicações mais críticas, que necessitam de um estado de execução, como, por exemplo, uma aplicação que se comunica com clientes e mantém estado do acesso de cada cliente e suas execuções em memória. Estas aplicações/serviços nós chamamos do tipo *Stateful*, ou seja, com estado. Estas, em geral, tem menos tolerância a falhas, por exemplo, se ela falhar perdemos todo o estado da aplicação e não teremos a mesma aplicação executando quando ela reiniciar, é interessante que possamos de alguma forma salvar este estado nos casos de falha (VAYGHAN et al., 2021).

O estado da aplicação em geral consiste do estado do processo pai e filhos executando na aplicação. Este estado contém, por exemplo, como as páginas de memória estão em determinado momento. Desta forma, temos exatamente o estado da execução, mas, não temos conhecimento de como ela nele. Não seria interessante para a aplicação salvar estes valores em persistência, pois, seria necessária muita instrumentação (OH; KIM, 2018).

2.3.1 etcd

O *etcd* consiste de um banco de dados de chave/valor distribuído. A distribuição e replicação dos dados se dá por um algoritmo de eleição de líderes, logo, ele é tolerante a falhas e consistente. (ETCD, 2023)

Por se tratar de um banco de dados distribuídos, ele é adequado para utilização em *clusters*, e, por isso, é utilizado por padrão no Kubernetes para armazenar metadados dos seus recursos. Também é comum que controladores customizados do Kubernetes utilizem este para armazenamento de seus metadados (KUBERNETES, 2023g).

2.4 CHECKPOINTING/RESTORE

A técnica de*Checkpoint/Restore* permite o salvamento do estado de uma aplicação *Stateful* e, posteriormente, a restauração da aplicação a partir do ponto de

salvamento passado. Em geral a técnica consiste em obter uma cópia da execução em memória da aplicação, salvar o contexto em uma imagem que poderá ser armazenada em armazenamento sólido e persistente. Quando há a necessidade de se recuperar o estado da aplicação a imagem que temos é utilizada para recuperar o estado para o momento em que ocorreu o salvamento da imagem (LAADAN; HALLYN, 2010).

Esta técnica pode ser utilizada para tolerância a falhas no caso de aplicações que quando falham necessitam do estado de recuperação (MÜLLER; MEINHARDT; MENDIZABAL, 2022) (CHEN, 2015/10), aplicações *Stateful*. Nesta técnica temos cópias dos estados anteriores da aplicação e podemos recuperá-la mais rapidamente. Outra possibilidade também é aumentar a disponibilidade dos serviços através do salvamento do estado e posterior recuperação dele (VAYGHAN et al., 2021). Embora, a técnica de *Checkpoint/Restore* tenha o objetivo de salvar um estado e depois recuperá-lo, ela pode ser utilizada para outras finalidades não envolvidas exatamente nessa linha, como migração de aplicações entre nós de um *cluster* (CHEN, 2015/10).

2.4.1 CRIU

CRIU é um projeto que implementa *Checkpoint/Restore* no Linux no userspace, o acrônimo da aplicação se dá justamente por isso, *Checkpoint/Restore in Userspace*. Ele trabalha com o conceito de congelar a execução de um processo, salvar seu estado de execução no disco, que depois pode ser utilizado para recuperar a aplicação exatamente no momento em que ela parou. Como contêineres são isolações em cima das ferramentas do Linux, é possível realizar o salvamento de estado de contêineres utilizando o CRIU (CRIU, 2023).

Algumas *runtimes* de contêineres possuem o suporte de *Checkpoint/Restore* implementado com o auxílio do CRIU, é o caso do containerd. Por usar runc para construção do *backend* da *runtime*, que possui o CRIU integrado, comandos na interface de linha de comando do containerd foram criados para realizar o *Checkpoint/Restore*. Desta forma, utilizando o Kubernetes com containerd também é possível realizar *Checkpoint/Restore* dos contêineres, não de forma nativa, mas se aproveitando das funcionalidades do containerd através do CRIU e a comunicação com as APIs tanto do containerd quanto do Kubernetes.

2.5 EVENT SOURCING

O padrão de projeto *Event Sourcing* cria eventos sobre ações em um sistema e, armazena estes em fluxos ordenados (MICROSOFT, 2023b), através de um versionamento da ordem. Então, quando uma ação é feita no sistema temos a emissão de um novo evento para um determinado fluxo. Estes são ordenados pela versão em que são entregues. Uma vantagem deste modelo é a possibilidade de alcançar o estado da aplicação em um determinado momento apenas reprojetando os eventos, ou seja,

ao emitir os mesmos eventos em ordem até um determinado momento no tempo, e, por momento se entende a última versão desejada (MICROSOFT, 2023b).

Este padrão permite uma grande forma de auditoria ao armazenar as ações feitas pelo sistema e permite encontrar inconsistências com grande facilidade ao reprojetar eventos até o ponto de uma falha. Também é possível adicionar um *snapshotter* que captura o estado do sistema em determinado momento, e, não será necessário reprojetar todos os eventos desde de o início do sistema, economizando tempo de processamento (MICROSOFT, 2023b).

3 TRABALHOS RELACIONADOS

Neste capítulo serão apresentados alguns trabalhos que possuem correlação com esta monografia, envolvendo os conceitos de *Checkpoint/Restore*, tolerância a falhas e contêineres.

3.1 CHECKPOINT/RESTORE PARA CONTÊINERES STATEFUL

Em (MÜLLER; MEINHARDT; MENDIZABAL, 2022), é proposta, como prova de conceito, uma arquitetura transparente para realizar o *Checkpoint/Restore* de contêineres *Stateful* utilizando Kubernetes. O foco primário é em tolerância a falhas com técnicas de *Checkpoint/Restore*, onde é abordada a problemática de se ter uma réplica nova do serviço que seja consistente com o momento atual da falha, não havendo perda de informações de estado da antiga réplica.

Para o *Checkpoint/Restore* o trabalho propõe a utilização de um contêiner *sidecar*, o interceptador, que intercepte as requisições para o contêiner que será feito o *Checkpoint/Restore*. Este contêiner irá redirecionar as requisições ao contêiner que é feito o *checkpoint* e também controla o momento do *checkpoint*. Desta forma, é possível saber quais as requisições que foram processadas pelo contêiner que está sendo monitorado até o momento do último *checkpoint*, já que a imagem de *checkpoint* possui metadados salvos no *etcd*. Quando uma nova réplica deste contêiner for iniciada, teremos a imagem do último *checkpoint* e o interceptador irá refazer as requisições feitas a partir do momento do último *checkpoint*, isto permite obter a imagem mais consistente com o momento da falha do último contêiner que estamos reinicializando.

Na arquitetura também temos um operador, que é responsável por manter o estado das aplicações. Este troca informações com o interceptador para remover contêineres, reiniciá-los ou avisar que eles falharam. Outro ponto da arquitetura é o administrador de estados, que serve para armazenar e categorizar as imagens do *checkpoint* com *checksums* e seus metadados no etcd.

Na proposta, se utiliza um monitoramento por *ping* de pacotes para identificar se um contêiner falhou ou não. No próprio trabalho o autor deixa claro outros métodos que poderiam ser utilizados, como monitoramento das informações do contêiner, ou mesmo um indicador de saúde provido pelo contêiner. A partir do momento de identificação da falha, outro contêiner com a imagem mais recente de *checkpoint* é criado para substituir o contêiner com falha. Como o interceptador só confirma uma mensagem quando o contêiner responde de volta, então, temos certeza de que o interceptador sabe a partir de qual requisição enviar para recuperar o estado.

Por fim, é apresentada uma prova de conceito realizada em cima do Kubernetes. A partir da prova de conceito o controlador do StatefulSet foi estendido para dar suporte ao Operador da arquitetura, e a aplicação a ser monitorada e o Interceptador

foram colocados no mesmo Pod. Conseguiu-se provar ao utilizar o Kubernetes a possibilidade de utilizar a arquitetura para *Checkpoint/Restore* para tolerância a falhas. Mas, foram levantados pontos de melhoria, como, utilização de *checkpointing* ativa, ou seja, realizando *checkpoint* somente quando uma réplica falhar, para diminuir o problema de *buffer* em cache dos interceptadores, detecção de falhas a partir de processos diferentes, como um algoritmo Bizantino, e outros processos de recuperação pró ativa, como votação para decidir réplicas incorretas.

No contexto do trabalho desenvolvido por nós, este trabalho propõe pontos importantes para nosso desenvolvimento. Como, por exemplo, como recuperar exatamente o estado que precisaríamos do nosso serviço. Também é interessante a abordagem para salvamento de metadados utilizando o etcd que já vem integrado ao Kubernetes. Nota-se, também, que alguns pontos de melhoria podem ser priorizados no desenvolvimento de nosso trabalho para ter uma solução mais abrangente de outras aplicações.

Já em (VAYGHAN et al., 2021), os autores propõem uma extensão das funcionalidades do Kubernetes para contêineres *Stateful* para prover alta disponibilidade (*High Availability*) a réplicas de um mesmo microsserviço. Isto implica, que será necessário manter a aplicação disponível o mais rápido possível depois de ela ter alguma falha.

O trabalho em (VAYGHAN et al., 2021) aborda uma nova forma de se utilizar contêineres *Stateful* sem utilizar o StatefulSet do Kubernetes, mas utilizando o Deployment com um Persistent Volume compartilhado entre todos os Pods que o Deployment controla. Desta forma, todos os contêineres compartilham o mesmo armazenamento e tem acesso ao mesmo estado em disco. Segundo o artigo, nenhum dos dois tipos consegue suprir necessidades reais de aplicações de alta disponibilidade, por isso, se faz necessário criar uma solução que provenha mais velocidade na reinicialização de Pods com falhas.

A solução proposta consiste em ter dois Pods disponíveis provendo o mesmo serviço em determinado tempo. Um deles é classificado como o ativo, enquanto o outro é classificado como o "em espera". O Pod que serve requisições é o ativo, quando este falha, um *controller* é criado, o State Controller, que utiliza a cópia do estado do Pod com falhas para o Pod reserva e cria um novo Service para ele servir as requisições. Quando o Pod ativo é recuperado, se faz o caminho inverso para ele voltar a ser o Pod que recebe as requisições. A replicação de estado no Pod reserva é feito através de um redirecionamento das requisições do Pod ativo. Como o State Controller cria Services separados para os Pods, é possível descobrir o Pod reserva através do nome do Service ao invés de seu IP no *cluster* graças ao componente de kube-proxy do Kubernetes.

Por fim, são apresentados os experimentos e resultados, tendo em mente que a alta disponibilidade dos serviços almejada era de 99,999% anualmente. Os experimentos consistiram em utilizar um serviço de *streaming* para os clientes, em que o estado do contêiner consiste no momento que o cliente está na transmissão de vídeo,

utilizando um cluster de oito máquinas e salvando o estado num PersistentVolume do Kubernetes.

As métricas de disponibilidade usadas foram de tempo de reação, o tempo de reação do Kubernetes para perceber uma falha e iniciar o processo de recuperação do Pod, o tempo de reparo, tempo a partir da percepção do Kubernetes até recuperação do Pod, o tempo de recuperação, o tempo a partir da percepção da falha pelo Kubernetes até a finalização da recuperação e quando o serviço atinge o estado de pronto, e, por fim, o tempo de indisponibilidade, que consiste no tempo que o serviço permanece sem disponibilidade, representado pela soma do tempo de reação com o tempo de recuperação.

Outras métricas obtidas pelo serviço, o State Controller proposto também foram utilizadas. A métrica de tempo de escala, que é o tempo de percepção do controlador de estado de um evento de escala até o escalonamento e disponibilização do serviços escalados para mais ou menos réplicas. A métrica de tempo de configuração do estado de alta disponibilidade, que é o tempo para o State Controller agir de acordo com o evento de réplica e configurar os estados dos Pods para se ter um ativo e outro reserva.

Os experimentos pretendiam responder às cinco perguntas:

1. Qual o impacto do State Controller na disponibilidade provida?
2. Qual o impacto de escalar durante um momento de falha na disponibilidade que o State Controller pode prover aos microsserviços gerenciados?
3. Qual é a sobrecarga gerada pelo State Controller no escalonamento?
4. Qual é o impacto de falhas simultâneas nos Pods ativos no tempo de indisponibilidade de cada Pod com falhas?

Dos experimentos se obteve respostas:

1. Os resultados mostram que é mais rápido para o State Controller configurar os estados aos Pods recuperados do que o Kubernetes recuperar os serviços. Uma pequena diferença na velocidade de recuperação entre StatefulSets e Deployments, onde o último tem velocidade maior e melhorada graças ao State Controller. Apesar do pequeno trabalho a mais que o State Controller gera, o tempo de recuperação diminui devido ao Pod reserva.
2. Os resultados mostram que o State Controller gera um atraso no escalonamento devido ao tempo que se leva para configurar os estados de disponibilidade aos Pods. Como o serviço não consegue realizar as duas operações ao mesmo tempo, ocorre um atraso no escalonamento, com maior atraso para os escalonamento de menos réplicas.

3. Os resultados mostram que gera mais trabalho para escalonamento, mesmo sem falhas, e que demora mais para as reações de escalonamento ocorrerem quando se utiliza o State Controller.

4. Os resultados mostram que quanto mais tarde um Pod falha mais tarde ele será recuperado pelo State Controller. Segundo a autora, isso ocorre pela utilização de uma fila *FIFO* para tratamento da recuperação, o que gera este problema.

Este trabalho demonstra uma outra forma de abordar o problema de serviços *Stateful*, ao invés de se salvar o estado em memória, se salva o estado utilizando os métodos de Persistent Volumes do Kubernetes. Desta forma, demonstra mais um fator que pode ser interessante observar no desenvolvimento do nosso trabalho, já que, para algumas aplicações este estado é interessante. Outro fator importante é a seleção de métricas, que o artigo aponta algumas métricas importantes para se considerar na avaliação da adequação da solução.

Já em (SCHMIDT et al., n.d.), temos uma proposta de uma ferramenta para tolerância a falhas em contêineres *Stateful* no Kubernetes utilizando *Checkpoint/Restore*. Os autores utilizam a técnica de *Checkpoint* já disponível no Kubernetes em versão alfa para gerar um ponto de salvamento do estado do sistema. O Kubernetes utiliza a *runtime* de contêineres cri-o(CRI-O, 2023), que internamente, utiliza o CRIU(CRIU, 2023) para realizar o *Checkpoint*. Como os autores discutem, este ponto de salvamento gerado não pode ser utilizado diretamente para executar outro contêiner como uma imagem.

Para possibilitar a parte do *Restore* os autores desenvolveram uma aplicação que constrói uma imagem compatível com *Open Container Interface* que é possível ser utilizada no cri-o. Para o desenvolvimento tanto do sistema de *Checkpoint* quanto do de *Restore* os autores utilizaram o conceito de operador do Kubernetes para monitorar o estado dos recursos do *cluster* e conciliar para salvar o estado das aplicações monitoradas através de anotações nos manifestos dos recursos e restaurar quando ao monitorar se perceber que houve uma falha na aplicação monitorada. Foram integradas outras funcionalidades ao trabalho, como a possibilidade de integrar diversos nós do *cluster* para distribuição geográfica e possibilidade de criar formas diferentes de se verificar se a aplicação falhou, como um recurso de *liveness*.

Os autores conseguiram prover o serviço, que, como já esperado gera um pouco de perda de performance nos momentos de realizar o salvamento do estado e da recuperação. Também é possível ver que durante momentos de salvamento e de recuperação a latência das aplicações aumenta, pois, durante um *Checkpoint* a aplicação fica não responsiva, e até a recuperação da aplicação falhante há outra latência. Entretanto, se verificou que é possível realizar o *Checkpoint/Restore* de aplicações *Stateful* no Kubernetes.

3.2 CHECKPOINT/RESTORE COM MIGRAÇÃO DE CONTÊINERES STATEFUL

Em (OH; KIM, 2018), os autores desenvolvem uma técnica de *Checkpoint/Restore* para migração de *clusters* com contêineres StatefulSet. Como prova de conceito é construída uma aplicação em cima de Kubernetes e CRIU para realização do *Checkpoint/Restore*. Alguns motivos levam contêineres em um orquestrador de contêineres ter serem movidos de um nó para outro, como manutenção do nó atual necessitando de uma reinicialização, falha do nó atual ou escalonamento e balanceamento de carga das aplicações do *cluster*. Logo, se um contêiner possui estado, nestes casos é interessante que a migração dele para um novo nó seja feita com salvamento do estado passado para reinicializar consistentemente.

Primeiramente, é analisada uma técnica para migração de contêineres *Stateful* através da utilização de volumes, *Persistent Volumes* no Kubernetes. Através da análise do autor, se observa que para realizar uma migração do estado utilizando os volumes, se tem uma complexidade aumentada. Já que, para salvar o estado em um volume é necessário que a aplicação saiba realizar esta tarefa, então, não é transparente ao desenvolvedor. Também o desgaste pelo acesso ao volume que pode ser intermediado por uma API de terceiro, como no caso de um volume de dados armazenado na EC2 API da Amazon. A partir desta análise, os autores propõem uma abordagem através da técnica de *Checkpoint/Restore* com criação de uma imagem com o estado salvo do contêiner.

Para realização da migração eles propõem uma solução que consiste em três estágios: pré migração, migração e pós-migração. Na pré migração é iniciado o processo de migração por qualquer motivo, como escalonamento, depois se elege novos nós para receberem os contêineres do nó antigo, por fim, se criam as imagens dos contêineres para substituir as antigas. Na migração é realizado *checkpoint* do contêiner, gerando uma nova imagem, que é transmitida para o novo nó através da internet e no novo nó é iniciado um novo contêiner a partir desta imagem, que permite que o processo seja recuperado com seu estado passado. Por fim, no estágio de pós migração é realizada a alocação de recursos antigos, como balanceadores de carga ou registros *DNS* para o novo contêiner.

Segundo os autores, a solução proposta ainda possui alguns problemas. No caso de falhas do nó ela não consegue realizar a migração, embora o *checkpoint* periódico resolveria este problema. Problemas com a internet para envio da imagem são ignorados no trabalho. Mas, com a proposta se pode obter uma transparência a nível de aplicação da migração do contêiner, se tem uma migração leve e ágil, já que não se usa um volume e não há o trabalho a mais de comunicação com possíveis APIs externas e leitura/escrita de volumes.

Para testar a solução se utilizou um *cluster* de máquinas com um orquestrador de contêineres e uma aplicação de servidor que incrementava um contador a cada

requisição com o estado salvo em memória. Também se implementou a solução na linguagem Go, se comunicando com a *engine* Docker e utilizando o CRIU para realizar o *checkpoint*. Como observado no trabalho, o tempo de inatividade no processo de migração se mostra baixo, 576.61ms em média, sendo que o tempo de latência da aplicação não foi alterado durante a utilização da imagem. Também não houveram falhas para enviar a imagem de *checkpoint* pela rede para o novo nó.

Embora neste trabalho o conceito abordado de *Checkpoint/Restore* seja utilizado em um contexto diferente, para migração de aplicações em *clusters*. Ainda sim, muitos conceitos se relacionam a nosso trabalho. A utilização de contêineres e aplicações de *Checkpoint/Restore*, diferentemente dos outros trabalhos vistos até o momento, este utiliza uma outra forma de salvar a imagem de *checkpoint*, que é através do envio pela rede. Também é abordada a transparência da aplicação, que também é importante no nosso trabalho, para que não haja adaptação das aplicações para a solução específica. Por fim, os autores apontam para um ponto importante que queremos cobrir, que é o caso de tolerância a falhas no caso de uma falha de nó, como apontado por eles, que por sugestão dos autores poderia ser resolvida por monitoramento do nó ou mesmo *checkpoints* periódicos das aplicações com estado.

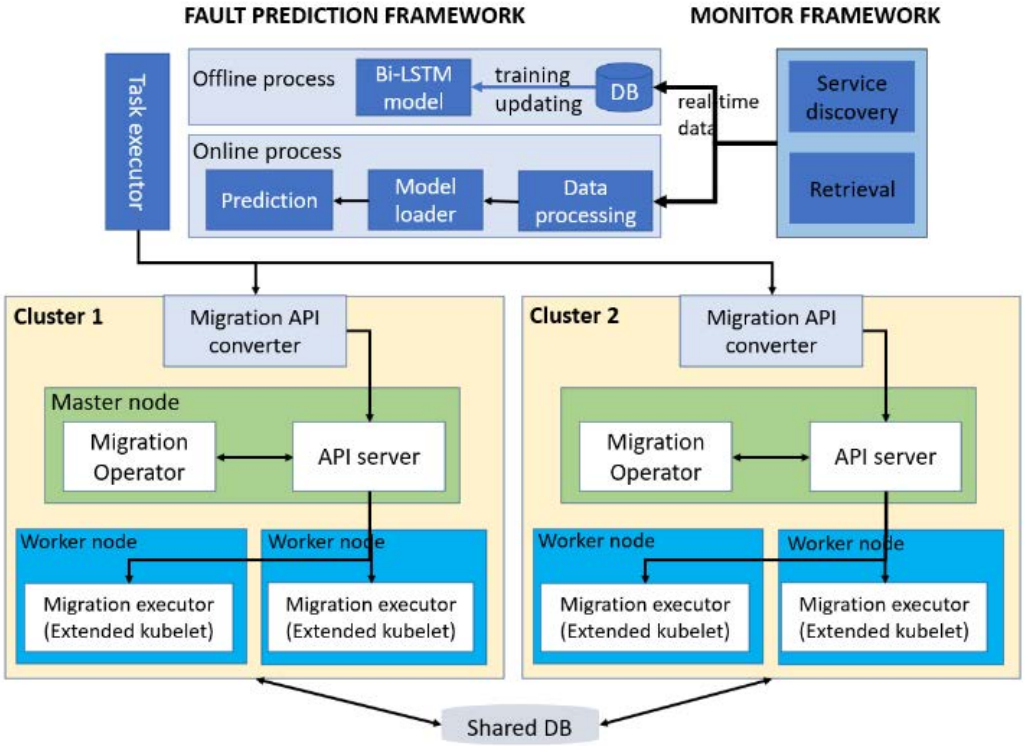
Já em (TRAN; VU; KIM, 2022), os autores propõem uma implementação para migração de serviços *Stateful* que seja proativa, ou seja, que perceba antes de uma falha ocorrer no nó e realize a migração para um novo nó sem que haja a necessidade do nó falhar antes, isto aumenta a tolerância a falhas e a qualidade de serviço provida. Para isso, os autores utilizam um algoritmo de *Bidirectional Long Short-Term Memory* (Bi-LSTM) para predição de falhas. A seleção foi em base do estado da arte em *Machine Learning*, já que o propósito do artigo é avaliar a atividade proativa de *Checkpoint/Restore* para migração dos contêineres. Desta forma, se utilizou o algoritmo para prever a sobrecarga do uso de CPU como falha.

Para realizar o *Checkpoint/Restore* os autores propõem uma solução baseada no StatefulSet do Kubernetes, assim como em (VAYGHAN et al., 2021), e também um salvamento dos dados em memória através do CRIU, como em (MÜLLER; MEINHARDT; MENDIZABAL, 2022). Desta forma, se tem o estado total da aplicação, suportando multi *clusters* e *clusters* de nó único.

Para realização das migrações proativas criaram três aplicações principais. O *Monitor Framework*, responsável por monitorar as métricas do *cluster*, como CPU e memória, também lida com o registro de serviços. O *Fault Prediction Framework*, responsável pelo modelo de predição de falha, que possui dois processos, o processo *offline*, que realiza o treinamento com base nos dados recebidos do *Monitor Framework* sobre a métrica de CPU, e, o processo *online*, que realiza a execução do modelo com a predição da falha do nó. Por último, temos a funcionalidade nativa de migração de *clusters*, que é uma modificação do Kubernetes, componente kubelet, para possibilitar a utilização da predição e da migração entre dois *clusters*. Podemos ver esta arquitetura

na Figura 2.

Figura 2 – Arquitetura de migração proativa de contêineres em *clusters*.



Fonte: (TRAN; VU; KIM, 2022)

Para realizar a migração foi modificado o componente kubelet do Kubernetes e também a *runtime* de contêineres em utilização pelo Kubernetes para possibilitar a comunicação com o CRIU para realizar o *Checkpoint/Restore* dos processos dos contêineres a serem migrados. Desta forma, quando uma migração é solicitada, primeiro se realiza o *Checkpoint/Restore* de todos os contêineres no Pod solicitado para migrar, depois se envia a imagem ao novo nó que ele irá executar. O novo nó utiliza a imagem gerada do *Checkpoint* para restaurar o estado antigo, só então são excluídos o Pod antigo e redirecionado o Service ao novo.

Para o processo de migração se utilizou o conceito de Operator do Kubernetes e *Custom Resource Definitions* (CRDs), que permitem a definição de novos recursos de API e da criação de novos controladores para monitorar a API e alcançar o estado desejado dos recursos. Então, em um processo de migração um novo recurso de migração é adicionado ao Kubernetes e o controlador da migração deve ser capaz de realizar o Checkpoint do Pod, enviar para o novo nó, iniciar o Pod no novo nó, eliminar o Pod antigo e reconfigurar o Service para o novo Pod. Também alteraram a definição da interface de *runtime* de contêiner do Kubernetes para incluir métodos para realizar *Checkpoint* e *Restore* com CRIU, que possibilita realizar as chamadas diretamente pela API do Kubernetes. Por fim, foi utilizado um servidor de arquivos para armazenar as imagens de *Checkpoint* entre os nós de migração. Assim, basta montar o volume

nos dois nós e eles estariam compartilhando as imagens, removendo a necessidade de envio por internet da imagem entre os nós e diminui as comunicações.

Para realizar a validação da solução se utilizou quatro *clusters* de Kubernetes, onde seriam executados aplicações de longo período de inicialização, como *Mongo-DB* e *Redis*, bem como aplicações de estado baseado na execução, como FFMPEG, todas executando através de StatefulSets. Para realizar a migração proativa, o estresse de CPU foi simulado utilizando uma ferramenta que permite sobrecarregar os nós, *Stress-ng*. Assim, foram verificadas métricas para resultados, no caso o erro da média da raiz quadrada e o erro de média absoluta para o modelo de predição e o tempo de recuperação do serviço migrado e a latência de qualidade de serviço evitada pela migração.

Como resultados foi observado que dos modelos abordados o melhor para a predição correta através de dois processos é o Bi-LSTM e ele pode ser utilizado para séries de tempo de diferentes tipos, como CPU, memória e temperatura. Também se observou que nas aplicações com tempo longo de inicialização houve melhoria do tempo de recuperação de serviço apenas em redes de alta velocidade, mas, segundo os autores não é um problema devido às novas redes de alta disponibilidade e velocidade 5G e 6G. Outros dois fatores a se destacar para serviços de tempo de inicialização alto é o que em momentos que o algoritmo de predição demora para prever a falha do nó, em que a migração pode não terminar e temos uma violação da qualidade de serviço em latência. Assim, como no caso de *clusters* muito grandes, com muitos Pods, ocorre degradação da comunicação com os servidores de arquivos das imagens, já que a rede do *cluster* fica sobrecarregado. Para que esta degradação não ocorra é necessário aumentar os servidores de arquivo e utilizar balanceadores de carga. Agora, para as aplicações de estado em execução se observou que o tempo de recuperação foi considerado com base em quanto o serviço demorava para completar suas funções, no caso do padrão do Kubernetes tínhamos mais tempo gasto em recuperar por exemplo todos os passos de uma conversão de vídeo no FFMPEG. Já na migração a conversão reiniciava do ponto do *Checkpoint*, que gerou uma melhoria de 20% a 22% na qualidade de serviço em latência. Desta forma, se demonstrou que a migração proativa com *Checkpoint/Restore* pode melhorar a qualidade de serviço e também tempos de recuperação dos serviços.

As mudanças nas características do Kubernetes abordadas no trabalho através de Operators são de grande importância para este e indica um caminho a se seguir. Também é interessante verificar as medidas feitas para avaliar a eficiência da solução em questão de tempo de recuperação e variação da latência do serviço. Já o modelo de predição não deve ser abordado em nosso trabalho, já que o foco não é realizar a predição, que poderia ser adicionado em um trabalho futuro.

4 SERVIÇO

Neste capítulo, abordamos a conceituação do nosso serviço de *Checkpoint/Restore* de contêineres *Stateful* no Kubernetes. Apresentamos a arquitetura geral do serviço e como os componentes da arquitetura se comunicam entre si. Também apresentamos as duas implementações que foram abordadas no trabalho, uma utilizando técnicas de *Event Sourcing* e a outra utilizando técnicas mais difundidas para *Checkpoint/Restore* através da utilização de CRIU para salvamento do estado de uma aplicação. Serão apresentados ao longo do texto diagramas de sequência que representam o funcionamento da arquitetura.

4.1 ARQUITETURA GERAL

A arquitetura do serviço possibilita o *Checkpoint/Restore* transparente das aplicações monitoradas pelo serviço no Kubernetes. Esta arquitetura é agnóstica sobre a implantação, seja ela em Kubernetes ou localmente em um sistema, é fortemente inspirada pelo trabalho em (MÜLLER; MEINHARDT; MENDIZABAL, 2022). A arquitetura é possível de implementar de diferentes formas, como na utilização de CRIU quanto de técnicas de *Event Sourcing*. Esta arquitetura também é adequada para execução em *cluster* de máquinas e adequada para microsserviços (VAYGHAN et al., 2021) (MÜLLER; MEINHARDT; MENDIZABAL, 2022) (OH; KIM, 2018), embora ainda falte um pouco de compreensão de como poderemos utilizá-la em sistemas distribuídos geograficamente de forma eficiente.

4.1.1 Checkpoint/Restore

O *Checkpoint/Restore* de aplicações *stateful* em contêineres pode ser feito entre duas formas, como abordamos no Capítulo 2, ou utilizando técnicas de salvamento do contexto dos processos, com isolação do contexto provida para contêineres (MÜLLER; MEINHARDT; MENDIZABAL, 2022) com CRIU, ou com as técnicas de *Event Sourcing*. No nosso serviço a parte de recuperação dos estados será administrada pelo componente de Administrador de Estado, já a parte de salvamento de estado será feita ativamente pelo Interceptador. Juntos estes dois componentes fornecem a funcionalidade de salvamento do estado da aplicação e posterior recuperação em caso de falhas.

4.1.2 Armazenamento

Para realizar os salvamentos de estados pelo Interceptador, é interessante que eles sejam feitos de forma periódica, capturando o estado da aplicação em determinado ponto da execução. Desta forma, é necessário armazenar este estado em algum

armazenamento. Para isso teremos a imagem gerada por *Checkpoint* que deverá ser armazenada em algum lugar para ser posteriormente recuperada, em (VAYGHAN et al., 2021) utiliza-se o conceito de volumes para armazenamento dos dados, aqui também iremos utilizar o mesmo conceito, tanto para imagens de *Checkpoint* quanto para armazenamento de eventos.

Ainda na questão do armazenamento, para utilização de *Checkpoint* com CRIU, diferentemente do *Event Sourcing* é interessante sabermos metadados sobre o momento de salvamento da imagem, que nos ajudem a identificá-la no tempo (OH; KIM, 2018) (MÜLLER; MEINHARDT; MENDIZABAL, 2022) (CHEN, 2015/10). Para isso, nos trabalhos de (MÜLLER; MEINHARDT; MENDIZABAL, 2022), (OH; KIM, 2018) e (CHEN, 2015/10) utilizou-se bancos de dados distribuídos de chave/valor como o etcd. Nosso administrador de estado também utilizará o mesmo conceito para armazenar metadados das imagens.

4.1.3 Recuperação ao último estado

Durante um salvamento do estado da aplicação a aplicação permanece indisponível para servir requisições e comandos do usuário, pois, as técnicas de salvamento de estado de contêineres com CRIU congelam o processo da aplicação para salvar o estado (VAYGHAN et al., 2021). Já no *Event Sourcing* não possuímos este problema. Entretanto, no segundo caso precisamos saber a ordem em que as requisições chegam para podermos reprojetar o estado da aplicação a partir da reemissão das requisições à aplicação monitorada. Ainda no caso do CRIU, ainda precisamos saber quais as requisições que chegaram a partir de um salvamento de estado, já que todas requisições posteriores àquele salvamento não estarão no estado salvo, precisaremos replicá-las para o serviço monitorado também. De modo geral, o que temos de diferença é que no caso do CRIU reprojetamos apenas as requisições a partir do último ponto de salvamento, já no caso do *Event Sourcing* replicamos as requisições desde o começo da existência da aplicação.

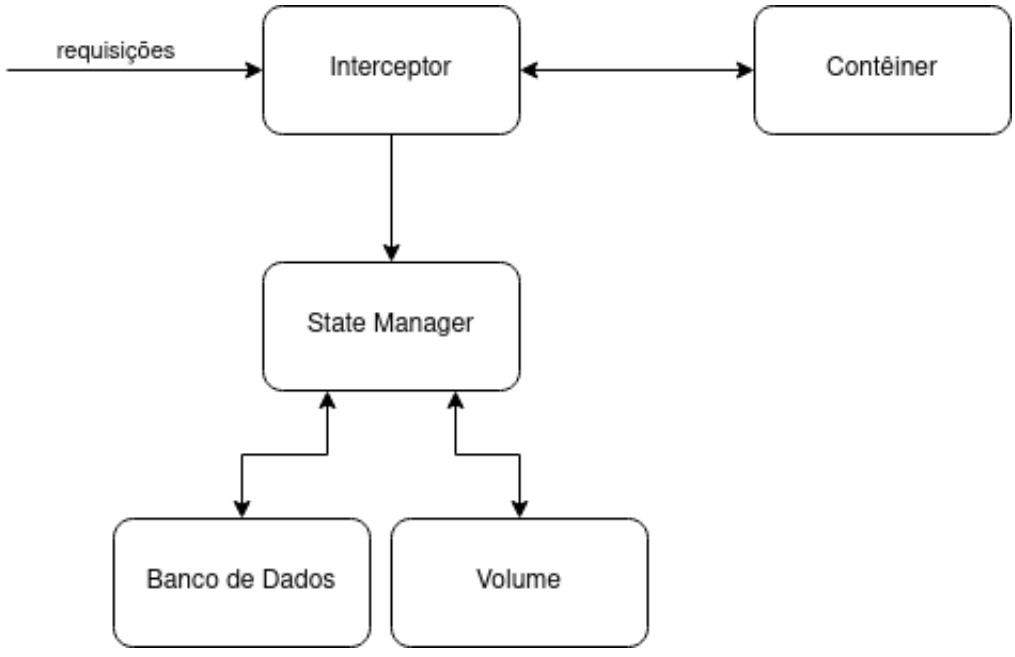
Para solucionar o problema de alcançar o último estado possível da aplicação em uma recuperação para nossos dois casos, iremos utilizar no nosso Interceptador o padrão de projeto de Embaixador(*Ambassador*) como em (MÜLLER; MEINHARDT; MENDIZABAL, 2022). Nosso Interceptador irá interceptar as requisições que chegam a nossa aplicação monitorada, irá salvar cada uma das requisições em um volume, que pode ser em memória ou armazenamento persistente, e encaminhá-las a aplicação. Sempre que formos realizar uma restauração iremos replicar as requisições a partir do último ponto de salvamento para utilização de CRIU, como em (MÜLLER; MEINHARDT; MENDIZABAL, 2022), mas, no caso do *Event Sourcing* faremos uma reprojeção total com todas as requisições feitas desde o momento que a aplicação começou a executar.

Para que novas requisições que chegarem durante o processo de restauração não interfiram na ordem do estado com as requisições sendo refeitas, nosso Interceptador tem dois estados de operação um Ativo que está enviando requisições diretamente para a aplicação monitorada, e outro de Aguardo que ficará esperando a restauração terminar para começar a enviar as requisições. Neste estado as requisições não serão respondidas imediatamente aos clientes, mas, só após a restauração finalizar, quando o estado do Interceptador retornar para Ativo. Em (VAYGHAN et al., 2021) algo semelhante foi feito sobre estados da aplicação sendo recuperada para impedir que requisições cheguem com estado desatualizado da aplicação.

4.1.4 A Arquitetura

As subseções acima trouxeram um pouco dos problemas que temos que enfrentar e como resolvê-los com algum componente do sistema. Nesta subseção, iremos abordar um pouco de como os componentes se comunicam entre a arquitetura e delimitar as funcionalidades de cada um dos componentes sem abordar nenhum tipo de implementação específica. Aqui, elucidamos o que o Administrador de Estado, o Interceptador, o Volume, o Banco de dados e Contêiner são e fazem. A arquitetura de maneira geral está apresentada na Figura 3, e a chamamos de representação geral abstrata da arquitetura.

Figura 3 – Representação da arquitetura abstrata da solução representada através de seus componentes.



Fonte: Elaborado pelo autor (2023).

O Contêiner é a abstração da nossa aplicação com estado, que é o cerne deste trabalho prover tolerância a falha a contêineres através de *Checkpoint/Restore*. Este

Contêiner contém uma imagem executando, onde o endereçamento de memória é importante para o contexto de execução da aplicação (CHEN, 2015/10). A aplicação neste caso pode ser um banco de dados como Redis ou uma aplicação de *streaming* de vídeo como usado em (VAYGHAN et al., 2021).

O Volume é a abstração de um armazenamento persistente ou não para armazenar as imagens e dados da aplicação. Para o nosso serviço é interessante que este Volume seja acessível por diversas máquinas em diferentes localidades (VAYGHAN et al., 2021), pelo foco do trabalho em *clusters* de máquinas. Este volume deve permitir escritas das imagens e recuperação das imagens através de uma interface de acesso.

O Banco de Dados é abstração de um banco de dados específico, que precisa persistir os dados, possibilitar a escrita, leitura e atualização dos dados. Ele também deve ser distribuído pelo *cluster* e tolerante a falhas para que todos as instâncias dele consigam servir as mesmas informações consistentemente.

O Interceptor é a nossa aplicação do padrão de projeto do Embaixador (MICROSOFT, 2023a). Ele deve interceptar as requisições à aplicação aplicação, enviá-las a esta, esperar a resposta deste, salvar a requisição para posterior recuperação das requisições respondidas pelo Contêiner e responder para o cliente da requisição. Outro requisito do Interceptor é que ele consiga replicar as requisições a um Contêiner a partir de uma informação de tempo, para que possamos recuperar um estado do Contêiner apenas repetindo as requisições. O Interceptor também é o responsável por realizar o salvamento do estado em uma imagem do Contêiner e enviar ao Administrador de Estado os metadados da última requisição tratada. Enquanto se faz um salvamento também devemos colocar as requisições que chegarem em um *buffer*, a partir da informação do estado do Interceptador como Aguardo.

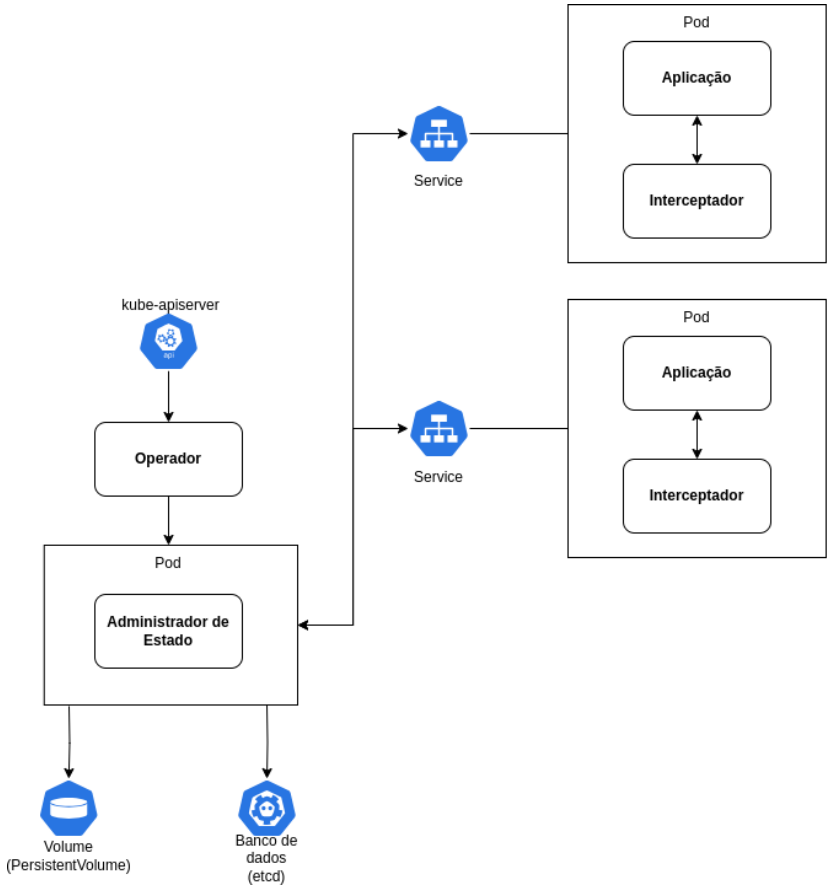
Por fim, o Administrador de Estado é o coordenador de toda a operação de *Checkpoint/Restore*, ele é quem deve receber metadados sobre a aplicação que tem salvamento de estado feito. Também é o Administrador de Estados que realiza a principal funcionalidade sobre o processo de restauração que é a identificação da falha do Contêiner monitorado e a atividade de recuperação pela comunicação com o Interceptador.

4.2 ARQUITETURA COM KUBERNETES

Para estruturação da arquitetura proposta na Subseção passada no Kubernetes devemos fazer algumas modificações. Iremos primeiro revisar alguns conceitos abordados no Capítulo 2. No Kubernetes temos o conceito de Operator, eles permitem estender as funcionalidades do Kubernetes, aliados com a API do Kubernetes podemos monitorar eventos no estado dos recursos e agir de acordo com isso. Já sabemos que toda aplicação no Kubernetes executa através de um Pod, e, a forma de realizar

a criação e administração de um Pod é através de Deployments e ReplicaSets, podemos monitorar estes recursos para criar novos Pods e verificar a falha deles. Também temos PersistentVolumes no Kubernetes que são acessíveis através de todo o *cluster* e o etcd como banco de dados de chave/valor distribuído disponível no Kubernetes. Representamos a arquitetura em Kubernetes na Figura 4.

Figura 4 – Representação da arquitetura da solução no Kubernetes.



Fonte: Elaborado pelo autor (2023).

O Contêiner da aplicação será implantado a partir de um Deployment do Kubernetes, isto permite definir a aplicação para ser administrada pelo Deployment usando ReplicaSets. Com isto, teremos a distribuição do serviço e conseguiremos monitorar a criação, deleção e atualização dos Pods que o Deployment e o ReplicaSet monitoram através da API do Kubernetes. Como em (SCHMIDT, 2023), deveremos monitorar e modificar as funcionalidades do ReplicaSet para possibilitar que nós possamos iniciar novos Pods da aplicação com imagens recuperadas e identificar quais Pods devemos monitorar com o nosso Interceptador.

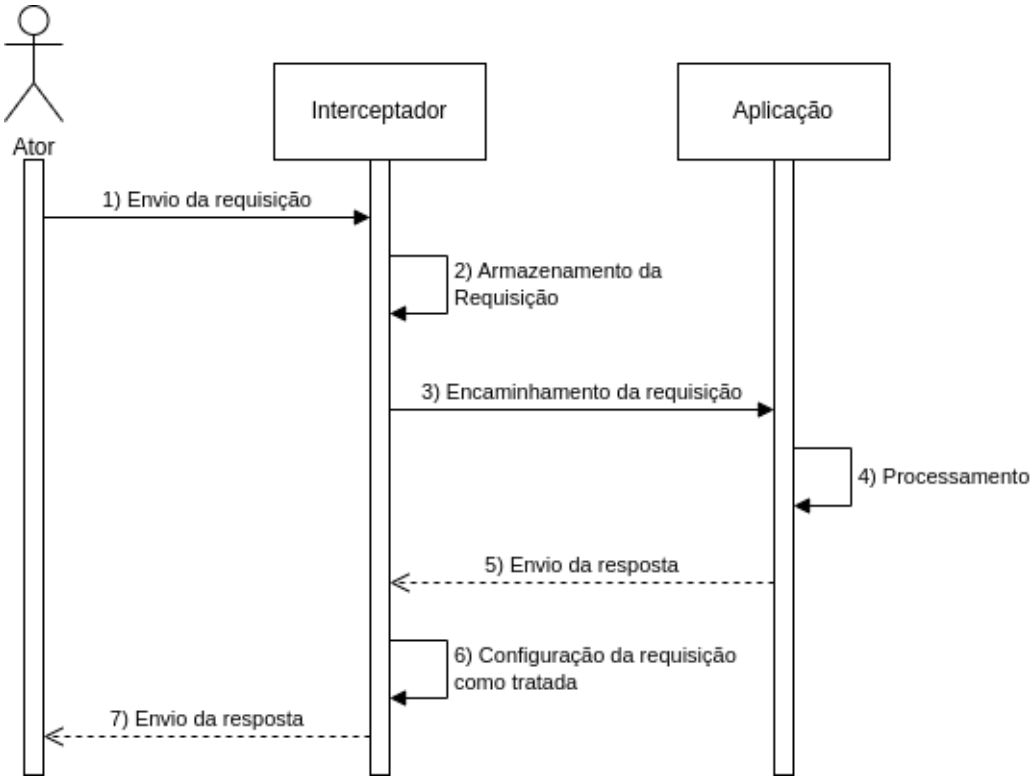
Para implementação do Administrador de Estado iremos utilizar também o conceito de Operator no Kubernetes. O nosso Administrador de Estado será implantado através de um controlador do estado dos Pods, que quando houver uma alteração no seu estado, como a falha, iremos realizar a recuperação do estado. O administrador

irá se comunicar com o Interceptador através do endereço IP do Pod do Kubernetes que pode ser obitdo a partir do momento em que o Pod está executando e pronto.

Nosso Interceptador será implantado através de um *sidecar container* na aplicação monitorada, teremos um Operador do Kubernetes que tentará conciliar o estado dos Deployments, sempre que houver um Deployment com uma anotação do tipo `crsc.io/checkpoint-restore` com valor de *true*, iremos criar um novo contêiner no Deployment que servirá como Interceptador e irá interceptar o tráfego ao Pod. Assim, como em (MÜLLER; MEINHARDT; MENDIZABAL, 2022), nosso Interceptador implementado como um *sidecar* permite economia de tráfego, já que ambos os contêineres operando no Pod tem a mesma interface de rede.

Para o Interceptor é importante que alguns fluxos estejam declarados. Temos o fluxo de interceptação das requisições, que é representado através de um diagrama de sequência na Figura 5. Neste diagrama representamos a chegada de requisições de um usuário até o Interceptor, este salva a requisição em *buffer* para o passo de reenvio numa restauração e, posteriormente, encaminha as requisições à aplicação, que, então, serão processadas e uma resposta será enviada ao Interceptor. A resposta recebida pelo Interceptor permitirá sabermos que o estado foi efetivamente atualizado na aplicação, marcando a requisição como processada no Interceptor e enviando a resposta ao usuário que a solicitou.

Figura 5 – Diagrama de sequência da interceptação de requisição pelo Interceptor.



Fonte: Elaborado pelo autor (2023).

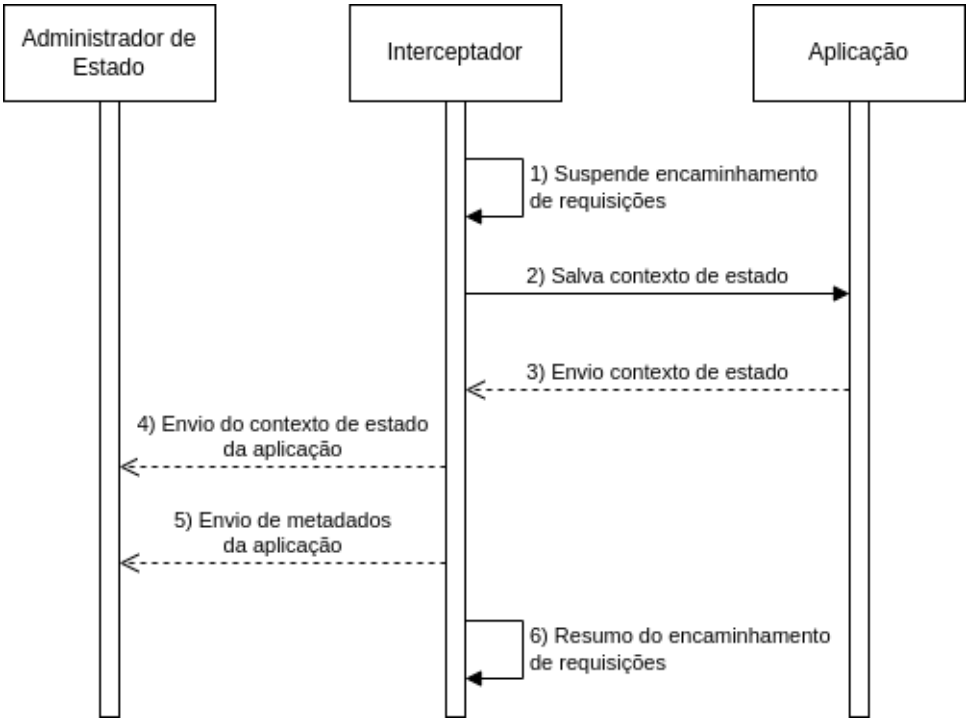
Agora para o fluxo de salvamento de uma imagem do contexto de estado da aplicação temos a Figura 6. Neste fluxo, temos a comunicação do Interceptador com o Administrador de Estado. A partir do momento em que o Interceptador percebe que é um momento para salvamento, seja através de metadados sobre intervalos para coleta, como em (VAYGHAN et al., 2021), ou através de detecção de erro como em (TRAN; VU; KIM, 2022), ele inicia o fluxo. Inicialmente, todas as requisições param de ser encaminhadas à aplicação e ficam apenas no *buffer* do Interceptador, este, então, inicia o processo de salvamento do estado da aplicação utilizando o CRIU. No momento que a imagem estiver pronta, os metadados dela serão enviados ao Administrador de Estado, que contém principalmente a última requisição servida, a identificação da aplicação e o horário da imagem. O Administrador de Estado internamente irá salvar os metadados. Por fim, no Interceptador é retomado o processo de interceptação das mensagens, servindo as mensagens no *buffer* e reencaminhando novas que chegarem. Em (MÜLLER; MEINHARDT; MENDIZABAL, 2022) teríamos um passo entre o 5 e o 6, que seria descartar as requisições no *buffer* anteriores a este *checkpoint*. Não faremos esta abordagem, manteremos o *buffer* com todas as requisições, salvando-as ou não em um banco de dados persistente, para que possamos implementar nossa solução específica com técnicas de *Event Sourcing* para recuperação do estado da aplicação.

As requisições serão mantidas após um *checkpoint*, como mostrado na Figura 6, para que seja possível replicá-las a partir de qualquer imagem de recuperação ou até mesmo da última imagem possível. Através da reprojeção das requisições como se fossem eventos, poderemos utilizar o padrão de projeto de *Event Sourcing* (MICROSOFT, 2023b) para replicar o estado de uma aplicação até o momento da falha apenas realizando todas as requisições novamente. Isto poderá ser lento, mas, iremos utilizar esta técnica para comparar com a perda de latência através de um *checkpoint*, que precisa parar todas as requisições momentaneamente.

Já para nossos Volumes para armazenamento persistente das imagens de recuperação utilizamos os PersistenVolumes do Kubernetes, que possibilitam que os recursos deste armazenamento persistente sejam acessíveis através de todo o *cluster*, possibilitando que a imagem seja acessível em qualquer nó (KUBERNETES, 2023a). Já para o Banco de Dados utilizamos o etcd que já está presente no Kubernetes, que é consistente e distribuído entre os nós do Kubernetes, ele permitirá salvar os metadados das imagens e acessar através de qualquer réplica do State Manager (ETCD, 2023) (KUBERNETES, 2023g).

Por fim, temos o problema de interferir no tráfego da nossa aplicação a ser monitorada. Conseguimos através da API do Kubernetes e do nosso Operator alterar um Service que iria indicar o tráfego para o serviço e colocar nosso Interceptador para servir este Service, pois eles funcionam por padrão como balanceadores de carga. Com isso temos a interceptação através do Interceptador e podemos interromper as requisições à aplicação quando necessário.

Figura 6 – Diagrama de sequência do salvamento de imagem feito pelo Interceptor.



Fonte: Elaborado pelo autor (2023).

5 IMPLEMENTAÇÃO DO SERVIÇO

Neste capítulo iremos trazer a experiência de implementação do Serviço para duas diferentes implantações. A primeira com CRIU e cri-o, que não conseguimos finalizar, mas que possui importantes experiências e resolução de problemas a serem relatadas. Já a segunda sendo a implantação utilizando técnicas de *Event Sourcing* que foi finalizada. Antes de apresentar as duas implementações iremos continuar apresentando mais aplicações, bibliotecas e plataformas que utilizamos durante a implementação do Serviço.

5.1 APLICAÇÕES, BIBLIOTECAS E PLATAFORMAS

Para a implementação tanto do *Checkpoint/Restore* com CRIU quanto com técnicas de *Event Sourcing* foi necessário a utilização de aplicações, bibliotecas e plataformas que facilitaram o nosso desenvolvimento.

Um dos pontos iniciais para a implementação é o local para realizar a implantação do Serviço. Inicialmente escolhemos utilizar a plataforma Emulab (WHITE et al., 2002) que fornece máquinas virtuais sob demanda para computação distribuída. Embora, tudo tenha funcionado bem com estas máquinas inicialmente, houveram dificuldades para fazer funcionar o CRIU juntamente com o Kubernetes nas máquinas. Então, optamos por trocar para outra solução mais comercial, o Google Cloud. Nesta plataforma obtemos uma máquina virtual permanente com as configurações sempre inicializadas, desta forma, não era necessário refazer a configuração da máquina todas as vezes que reinicializássemos *cluster*. De modo a economizar nos custos dos recursos de nossos testes, eles foram feitos sob uma implantação em *cluster* Kubernetes de máquinas com 2vCPU e 4GB de memória, que é suficiente para executar um *cluster* de nó único com Kubernetes.

Para possibilitar a existência da aplicação no Kubernetes nós precisamos adequar a aplicação no fluxo de reconciliação de estado do Kubernetes através do conceito do Operador (KUBERNETES, 2023h). Para isto, criamos três controladores, que são específicos para cada recurso do Kubernetes e realizam o fluxo de reconciliação para os recursos, que juntos implementam partes dos componentes da nossa arquitetura. O Interceptador é o único que não foi implementado a através de um controlador. Teremos o controlador de Deployment, que é responsável por verificar novos Deployments com anotação de monitoramento `crsc.io/checkpoint-restore: true`, que indica que deve ser monitorado para recuperação pelo Serviço, este controlador adiciona no Deployment o Interceptador como um *sidecar container*, que permitirá interceptar as requisições à aplicação alvo. O controlador de Pod, será o responsável por monitorar os Pods e verificar quando houver a anotação `crsc.io/checkpoint-restore: true` e o contêiner alvo da aplicação sofrer uma falha, ou seja, não estiver no estado de Pronto

do Kubernetes, ele ser agendado para uma recuperação. Por fim, temos um outro controlador não totalmente implementado, ele é o responsável por verificar o intervalo de *Checkpoint* da aplicação através da anotação do Deployment `crsc.io/checkpoint-interval`, sempre que um novo Deployment monitorado for identificado, um recurso de Checkpoint é criado, e este é monitorado pelo controlador do Checkpoint, quando se dá o momento do *Checkpoint* o controlador se comunica com o Interceptador e depois cria uma nova imagem a partir do resultado do salvamento de estado do interceptador. Iremos cobrir com mais ênfase cada um deles nas seções posteriores. Para construir todos este Operador, utilizamos o *framework* operator-sdk, que permite realizar a criação de recursos de Kubernetes e dos controladores de maneira ágil e simples, abstraindo as interações com o kube-apiserver e criação dos Custom Resource Definition, que foram necessários para o novo recurso Checkpoint.

Ainda no contexto do *Checkpoint*, como comentamos no capítulo de trabalhos relacionados em (SCHMIDT et al., n.d.), podemos obter um *Checkpoint* da aplicação utilizando o kubelet e as funcionalidades já implementadas para o cri-o através da API do Kubernetes (KUBERNETES, 2023e). Entretanto, isso não nos gera uma imagem no formato do Open Container Initiative, devemos criar essa imagem a partir de todos os passos que fizeram no trabalho em (SCHMIDT et al., n.d.). O caminho que optamos por seguir é um pouco diferente e apresenta uma facilidade maior, na implementação de (SCHMIDT et al., n.d.) temos o problema de dar manutenção na geração da imagem sempre que uma mudança na interface ocorrer. Por isso, optamos por utilizar uma biblioteca e aplicação que já fizesse esta parte a abstraísse o trabalho para nós, a biblioteca que encontramos, que também funciona como uma aplicação de linha de comando, é o buildah(BUILDAH, 2023). O buildah é uma ferramenta especializada para facilitar a criação de imagens de contêineres no formato do OCI.

Para armazenamento persistente dos dados da aplicação, como as requisições, utilizamos o PostgreSQL (POSTGRESQL, 2023) como banco de dados persistente. Ele será utilizado para armazenar as requisições persistentemente quando necessário. Entretanto, a aplicação pode funcionar utilizando memória, mas, nesta implementação existem problemas dependendo da quantidade de memória livre para aplicação ao se armazenar todas as requisições.

5.2 CONFIGURAÇÃO DO CLUSTER

Para realizar a configuração do *cluster* Kubernetes foram necessário realizar algumas configurações e instalação de pacotes que fizessem CRIU, Kubernetes e buildah funcionarem. Para configurar um *cluster* Kubernetes com todas as funcionalidades necessárias utilizamos Ubuntu 20.04, em máquinas com processador Intel Broadwell de arquitetura x86/64 com 2vCPU e 4GB de memória, a partir disso precisamos executar a sequência de comandos no Código 1, neste código utilizamos o cri-o na versão

1.25, que é a versão mínima necessária para fornecer suporte à funcionalidade de *Container Checkpoint* utilizando CRIU no Kubernetes, que será a funcionalidade utilizada na implementação com CRIU para prover salvamento do estado da aplicação.

Código 1 – Comandos de configuração da máquina para CRIU e cri-o.

```
1  sudo sh -c 'echo "deb https://download.opensuse.org/repositories/devel:/kubic:/
       libcontainers:/stable/xUbuntu_20.04/ /" > /etc/apt/sources.list.d/devel:
       kubic:libcontainers:stable.list'
2  sudo sh -c 'echo "deb http://download.opensuse.org/repositories/devel:/kubic:/
       libcontainers:/stable:/cri-o:/1.25/xUbuntu_20.04/ /" > /etc/apt/sources.
       list.d/devel:kubic:libcontainers:stable:cri-o:$CRIO_VERSION.list'
3  curl -L https://download.opensuse.org/repositories/devel:/kubic:/libcontainers
       :/stable:/cri-o:/1.25/xUbuntu_20.04/Release.key | sudo apt-key add -
4  curl -L https://download.opensuse.org/repositories/devel:/kubic:/libcontainers
       :/stable/xUbuntu_20.04/Release.key | sudo apt-key add -
5  sudo apt-get update
6  sudo apt-get install -y criu cri-o cri-o-runc
```

Primeiro adicionamos os repositórios para instalação do cri-o, que será utilizado como *runtime* de contêineres para o Kubernetes, e instalação dele, no mesmo comando de instalação, instalamos o pacote do CRIU. No Código 1 também instalamos o *backend* que o cri-o utiliza para contêineres, o runc, uma versão específica deve ser instalada para que funcione o cri-o no Kubernetes.

Após instalado cri-o e CRIU, alteramos as configurações de ambos para que pudéssemos realizar o *Checkpoint*. Primeiro, no Código 2, alteramos o comportamento da *runtime* adicionando a opção que ativa o suporte para o CRIU no cri-o e uma opção que remove a infraestrutura quando um contêiner não possui uma id de processo privada, este deve ser ativado para mitigar um erro no cri-o. Estas alterações foram feitas no arquivo /etc/crio/crio.conf.

Código 2 – Configuração a ser incluída no arquivo de configurações do cri-o.

```
1 [crio.runtime]
2 enable_criu_support = true # permite suporte a CRIU
3 drop_infra_ctr = false # remove a infra quando um pod não tem uma id de processo
      privada
```

Para que o CRIU possa utilizar o runc corretamente, também devemos editar a configuração do CRIU. No Código 3, editamos o arquivo /etc/criu/runc.conf para incluir configurações para fechar as conexões TCP no caso de uma restauração iniciada. De modo que o cliente que iniciou a conexão TCP deverá refazê-la, para ignorar conexões TCP já feitas no momento do *Checkpoint*, onde o cliente também deverá refazer as requisições. Também desativamos a administração e cgroups feita pelo CRIU para

utilizar o padrão, para que seja possível executar o *runc* no mesmo cgroup que cri-o e kubelet.

Código 3 – Configuração a ser incluída no arquivo de configurações do runc para o CRIU.

```
1 tcp-close # fecha as conexões TCP
2 skip-in-flight # ignora conexões já abertas que devem ser refeitas depois do
      checkpoint
3 manage-cgroups=ignore
```

A próxima parte requer que tenhamos buildah instalado e os componentes principais do nó mestre do Kubernetes, o kubeadm e o kubelet, como também o kubectl para acessar o cluster através da API do Kubernetes. Os comandos para instalação destes pacotes e componentes podem ser instalados através do Código 4. O comando `sudo apt-mark hold` é necessário para que seja possível congelarmos a versão em utilização dos pacotes que especificamos, neste caso os pacotes importantes para funcionamento do Kubernetes, que devem permanecer na mesma versão para possibilitar o funcionamento correto do *cluster*.

Código 4 – Instalação dos pacotes necessários para Kubernetes e buildah.

```
1 sudo apt-get install -y apt-transport-https ca-certificates curl buildah make
2 sudo mkdir /etc/apt/keyrings
3 sudo sh -c "curl -fsSL https://pkgs.k8s.io/core:/stable:/v1.25/deb/Release.key |
      gpg --dearmor -o /etc/apt/keyrings/kubernetes-apt-keyring.gpg"
4 sudo sh -c 'echo "deb [signed-by=/etc/apt/keyrings/kubernetes-apt-keyring.gpg]
      https://pkgs.k8s.io/core:/stable:/v1.25/deb/ /" | tee /etc/apt/sources.list.d/
      kubernetes.list'
5 sudo apt-get update
6 sudo apt-get install -y kubelet kubeadm kubectl
7 sudo apt-mark hold kubelet kubeadm kubectl
```

Como o Checkpoint para contêineres no Kubernetes ainda é uma funcionalidade em alfa para o Kubernetes, ela não é ativada automaticamente para toda nova instância de *cluster* Kubernetes. Precisamos alterar o funcionamento do Kubernetes para utilizar especificamente essa funcionalidade, para isso existem os chamados *Feature Gates* do Kubernetes que permitem nas versões mais novas do Kubernetes ativar funcionalidades ainda não lançadas. Para ativar a funcionalidade de *Container Checkpoint* devemos alterar o arquivo de configuração que determina a execução do kubelet no systemd do Ubuntu, em /usr/lib/systemd/system/kubelet.service.d/10-kubeadm.conf, que deve estar como no Código 5. A parte que ativa o *Feature Gate* é `Environment="KUBELET_FEATURE_GATES_ARGS=--feature-gates=ContainerCheckpoint=true"`. Já a parte `Environment="KUBELET_EXTRA_ARGS=--cgroup-driver=systemd"` configura o kubelet para uti-

lizar o cgroup systemd para executar. Isto coloca o cri-o no mesmo cgroup do kubelet, que permite que o último utilize o primeiro. Caso isto não seja feito, o kubelet não teria acesso ao cri-o.

Código 5 – Configuração do kubelet para executar no systemd com Feature Flag de ContainerCheckpoint e cgroup do systemd.

```
1  [Service]
2  Environment="KUBELET_KUBECONFIG_ARGS=--bootstrap-kubeconfig=/etc/kubernetes/
       bootstrap-kubelet.conf --kubeconfig=/etc/kubernetes/kubelet.conf"
3  Environment="KUBELET_CONFIG_ARGS=--config=/var/lib/kubelet/config.yaml"
4  Environment="KUBELET_EXTRA_ARGS=--cgroup-driver=systemd"
5  Environment="KUBELET_FEATURE_GATES_ARGS=--feature-gates=ContainerCheckpoint=true"
6  # This is a file that "kubeadm init" and "kubeadm join" generates at runtime,
       populating the KUBELET_KUBEADM_ARGS variable dynamically
7  EnvironmentFile=-/var/lib/kubelet/kubeadm-flags.env
8  # This is a file that the user can use for overrides of the kubelet args as a last
       resort. Preferably, the user should use
9  # the .NodeRegistration.KubeletExtraArgs object in the configuration files instead.
        KUBELET_EXTRA_ARGS should be sourced from this file.
10 EnvironmentFile=-/etc/default/kubelet
11 ExecStart=
12 ExecStart=/usr/bin/kubelet $KUBELET_KUBECONFIG_ARGS $KUBELET_CONFIG_ARGS
       $KUBELET_KUBEADM_ARGS $KUBELET_EXTRA_ARGS $KUBELET_FEATURE_GATES_ARGS
```

Por fim, iniciamos o *cluster* Kubernetes através dos comando no Código 6, desativamos o *swap* da máquina para não criar conflitos que podem ocorrer no kubelet. Outra seção do Código 6 é a configuração do KUBECONFIG, que é a configuração que o kubectl utiliza para se comunicar com a API do Kubernetes. Ao final do código nós adicionamos um controlador de rede para os Pods do nosso sistema, qualquer um pode ser utilizado, mas optamos por utilizar o Calico (CALICO, 2023) pela facilidade em instalação como vista na seção do Código 6.

Código 6 – Inicialização do Kubernetes, configuração de acesso para o kubectl e instalação do administrador de rede para Pods Calico.

```
1  # Start Kubernetes
2
3  sudo swapoff -a
4  sudo kubeadm init --pod-network-cidr=192.168.0.0/16 --ignore-preflight-errors='all'
5
6  # Init kubectl
7
8  mkdir -p $HOME/.kube
9  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
```

```
10 sudo chown $(id -u):$(id -g) $HOME/.kube/config
11
12 # Add network manager. We are going to install Calico.
13
14 kubectl create -f https://raw.githubusercontent.com/projectcalico/calico/v3.26.1/
       manifests/tigera-operator.yaml
15 kubectl create -f https://raw.githubusercontent.com/projectcalico/calico/v3.26.1/
       manifests/custom-resources.yaml
```

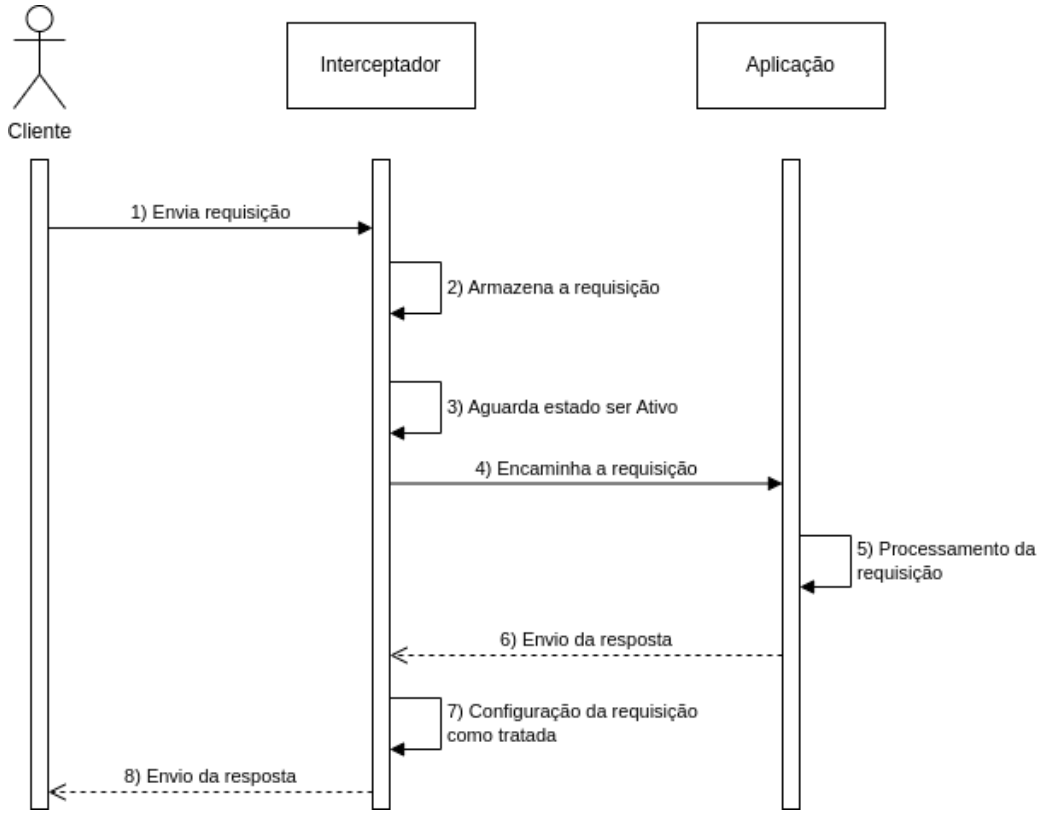
A partir das configurações listadas nesta seção obtivemos um *cluster* Kubernetes funcional pronto para iniciar nossas implementações, que permitem a utilização de CRIU para o *Checkpoint* e de buildah para construção das imagens de recuperação.

5.3 INTERCEPTADOR

O nosso Interceptador como descrito nas seções anteriores é implantado a partir de um *sidecar container*. Ambas as versões do Serviço, a de *Checkpoint/Restore* a partir do CRIU e a do *Checkpoint/Restore* a partir de técnicas de *Event Sourcing* utilizam o mesmo Interceptador que é a funcionalidade comum aos dois. Para criá-lo utilizamos a linguagem de programação Go e realizamos uma construção de sua imagem utilizando Docker. Não é necessário utilizar cri-o para construir a imagem neste caso, pois as imagens geradas também seguem OCI. A aplicação consiste de um servidor HTTP, que possui algumas funcionalidades expostas que serão chamadas de acordo com a necessidade dos outros componentes. O Interceptador possui um estado interno que pode estar em dois valores diferentes, ou em Ativo ou em Aguardo. No Ativo enviamos todas as requisições interceptadas à aplicação alvo do Serviço. Já no estado Aguardo nós realizamos o armazenamento das requisições e esperamos o estado retornar a Ativo para continuar a entrega das requisições. Enquanto seguimos o fluxo do diagrama da Figura 5 para interceptação das requisições durante o estado de Pronto do Interceptador, a Figura 7 representa o fluxo de interceptação quando o estado do Interceptador é Aguardo.

Para realizar um *Checkpoint*, expomos uma rota HTTP de `/checkpoint`, que chama a API do Kubernetes para que o kubelet comunique com o cri-o para realizar um *Checkpoint* da aplicação executando no contêiner dentro do Pod alvo utilizando CRIU. De modo a realizar a comunicação com o kube-apiserver precisamos estar autenticados com o Kubernetes. Deste modo, criamos um Secret no Kubernetes, uma configuração de segredo disponível na API do Kubernetes, com o certificado e chave de assinatura para comunicação com o kube-apiserver. Estes arquivos ficam localizados no *cluster* em `/etc/kubernetes/pki/apiserver-kubelet-client.key` e em `/etc/kubernetes/pki/apiserver-kubelet-client.crt`. Adicionamos estes dois arquivos em um Secret como no Código 7. Então, podemos chamarmos a API do

Figura 7 – Diagrama de sequência para tratamento de requisições durante estado de Aguardo do Interceptador.



Fonte: Elaborado pelo autor (2023).

Kubernetes na rota `<https://ip-do-cluster/checkpoint/namespace/pod/container>` utilizando o método HTTP POST, onde ip-do-cluster é o IP do *cluster* Kubernetes, é o *namespace* do Kubernetes em que se encontra a aplicação do Kubernetes, pod é o Pod que está executando a aplicação alvo e container o contêiner que está executando a aplicação alvo. A partir daí, um *Checkpoint* é salvo na máquina em `/var/lib/kubelet/checkpoints` na forma de um arquivo compactado da forma checkpoint-pod_namespace-container-timestamp.tar, em que os valores são os mesmo, exceto, *timestamp*, que representa a string do momento em que o *Checkpoint* foi feito.

Código 7 – Criação do Secret para comunicação com o kubelet para compartilhamento com nosso Interceptador.

```
1 kubectl create secret generic kubelet-client-certs --from-file=client.crt=/etc/
    kubernetes/pki/apiserver-kubelet-client.crt --from-file=client.key=/etc/
    kubernetes/pki/apiserver-kubelet-client.key
```

Outra rota do Interceptador é a do estado em `/state`, através de uma chamada HTTP ao método POST podemos alterar o estado com o parâmetro na *query* `state`, sendo ou Active, ou Waiting, que coloca o Interceptador nos estados, respectivamente,

de Ativou e Aguardo. O estado inicial do Interceptador é o Ativo.

Por fim, nós temos a rota `/reproject` que é a reprojeção de todas as requisições interceptadas pelo Interceptador e aceita pela aplicação alvo. Esta reprojeção permite que o estado seja alcançado. O ideal nesta reprojeção seria que ela pudesse ser feita a partir de uma versão das requisições. Entretanto, não a fizemos porque não foi possível finalizar a parte do *Checkpoint/Restore* com CRIU que necessitaria desta parte.

5.4 CHECKPOINT/RESTORE COM CRIU

Na implementação de *Checkpoint/Restore* com CRIU criamos os três controladores do nosso Operador, o controlador dos Deployments, o controlador dos Pods e o controlador do nosso recurso customizado Checkpoint. Nesta implementação temos três passos para toda aplicação monitorada, criação e implantação de um manifesto do Deployment da aplicação no Kubernetes. Posterior criação do recurso customizado Checkpoint e monitoramento das falhas do contêiner monitorado e posterior recuperação. Estes passos serão feitos pelos nossos controladores e descrevemos elas na próxima subseção.

5.4.1 Controladores

Todos os controladores do Kubernetes possuem um ciclo de reconciliação que deve verificar o estado do *cluster* e, em caso, de modificação do estado de algum recurso que ele monitora deve agir de acordo com a alteração de estado para alcançar um estado dos outros recursos de acordo com o que ele tem definido através dos manifestos dos recursos.

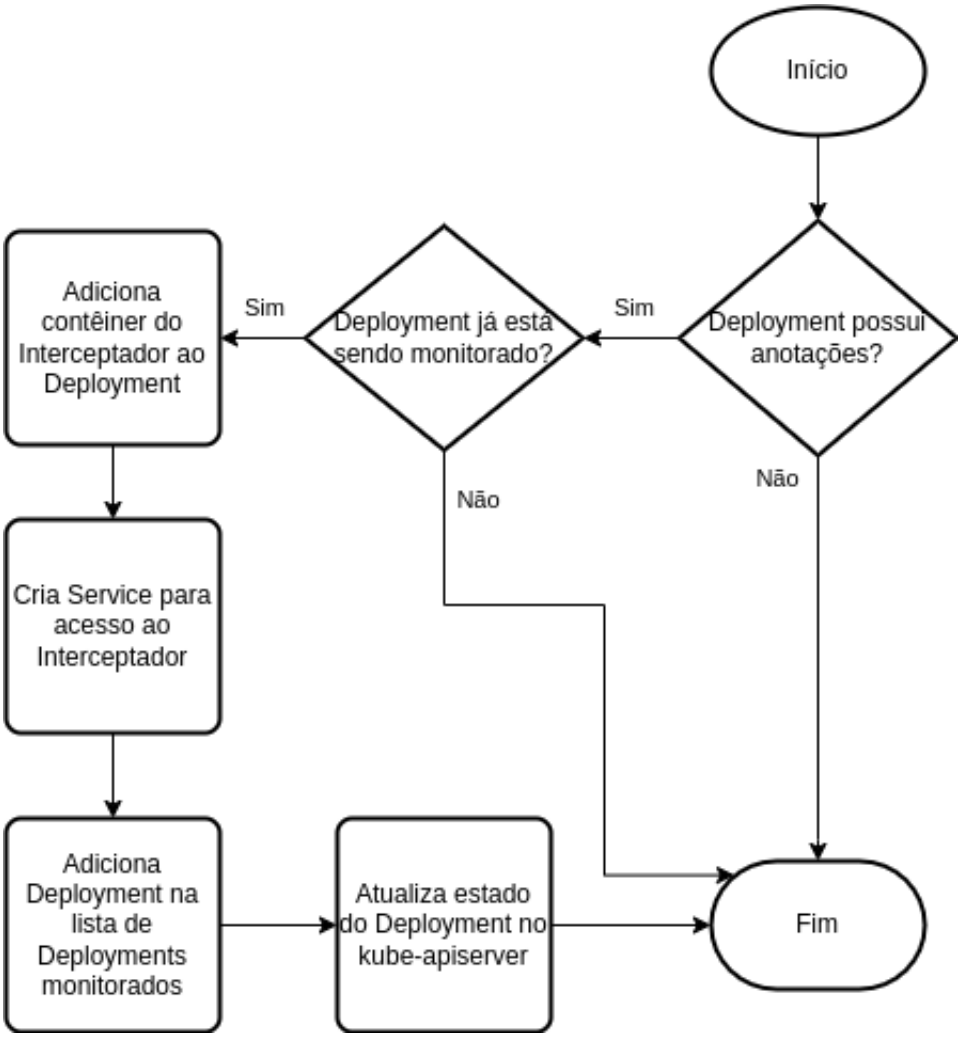
5.4.1.1 Controlador de Deployment

O controlador do Deployment tem como seu recurso monitorado os Deployments do *cluster*. Sempre que uma aplicação é criada ela deve ser criada com anotações no manifesto do Deploment, como `crsc.io/checkpoint-restore` e `crsc.io/checkpoint-interval`, que são respectivamente, o valor booleano que define se a aplicação é monitorada pelo Serviço ou não e o valor do intervalo de *Checkpoint* ativo. O diagrama da Figura 8 define o funcionamento do controlador.

A partir da alteração do estado do *cluster* pela criação de um novo Deployment monitorado o controlador realiza a criação de um novo recurso de Checkpoint. Este novo recurso possui um manifesto com os valores de intervalo de *Checkpoint* e também as informações da aplicação monitorada, como o nome do Deployment. Este recurso será monitorado pelo controlador de Checkpoint que irá executar as ações necessárias para alcançar o estado definido pelo manifesto.

Este controlador também deve injetar o Interceptador como um *sidecar container*

Figura 8 – Diagrama de fluxo para a injeção das configurações do Interceptador na aplicação alvo através de um Controlador de Deployments para o Operador.



Fonte: Elaborado pelo autor (2023).

ao Deployment da aplicação alvo. Sempre que um Deployment novo é adicionado com a anotação `crsc.io/checkpoint-restore` o manifesto do Deployment é alterado para adicionar o contêiner do Interceptador como um *sidecar container*. Um novo ReplicaSet é, então, criado pelo Kubernetes que criará um novo Pod com o nosso Interceptador. Ao mesmo tempo o controlador cria um novo Service para o Interceptador no novo Deployment que permite que o Interceptador seja acessado através de todo o *cluster*. O Kubernetes, então, lida com a parte do DNS e das configurações de rede.

Ao final temos a adição no manifesto do Deployment do conteúdo do contêiner como visto no Código 8. Este contêiner tem montado na imagem um volume que indica os arquivos de assinatura na comunicação com o kube-apiserver que criamos na Seção de configuração do *cluster*. Isto permite que o nosso Interceptador possa se comunicar de forma segura e autenticada com o Kubernetes para solicitar um novo *Checkpoint* ao kubelet.

Código 8 – Configuração do Interceptador para o Deployment da aplicação alvo como sidecar container.

```
image: docker.io/gianaortiz/crsc-interceptor
imagePullPolicy: Always
name: interceptor
ports:
- containerPort: 8001
  hostPort: 8001
  protocol: TCP
resources: {}
terminationMessagePath: /dev/termination-log
terminationMessagePolicy: File
volumeMounts:
- mountPath: /var/run/secrets/kubelet-certs
  name: kubelet-certs
  readOnly: true
```

A partir desse momento nossa aplicação monitorada tem suas requisições interceptadas pelo nosso Interceptador. Inicialmente, o estado é Ativo no Interceptador, então todas as requisições serão primeiro armazenadas e depois encaminhadas à aplicação para que elas sejam tratadas.

5.4.1.2 Controlador de Pod

Para implementar o nosso Administrador de Estado, implementamos parte dele no controlador de Pod. O controlador tem uma administração mais passiva, através do ciclo de reconciliação. Quando um Pod apresenta um problema, como por exemplo, um dos contêineres falhar, o controlador filtra os Pods que possuem a anotação `crsc.io/checkpoint-restore`, adicionando a uma fila de restauração. A restauração aconteceria a partir da configuração de utilização de uma nova imagem à aplicação monitorada. Entretanto, não conseguimos criar a imagem de *Checkpoint* e esta parte não foi feita.

Na Figura 9 temos o diagrama de como funciona esta parte. Primeiro o controlador recebe uma atualização de estado dos Pods, verificando se o Pod é um Pod monitorado. Caso seja e tenha um contêiner com falha, ele será adicionado à fila de restauração. Caso a atualização de estado indique que ele está pronto iremos buscar o Pod na fila de restauração. Então, a imagem da aplicação alvo será editada para que seja a imagem do último *Checkpoint* feito pelo controlador de Checkpoint. Após estabilização da imagem, colocamos o Interceptador no estado Aguardo e é solicitado a ele a reprojeção das requisições a partir da última requisição recebida antes de se realizar o *Checkpoint* atual. Ao final da reprojeção, o estado do Interceptador é alterado para Ativo e as requisições são encaminhadas para a aplicação alvo com o estado

mais atual, fornecendo uma recuperação transparente ao usuário.

Figura 9 – Diagrama de fluxo para a recuperação da aplicação alvo através de um Controlador de Pods para o Operador com a implementação com CRIU.

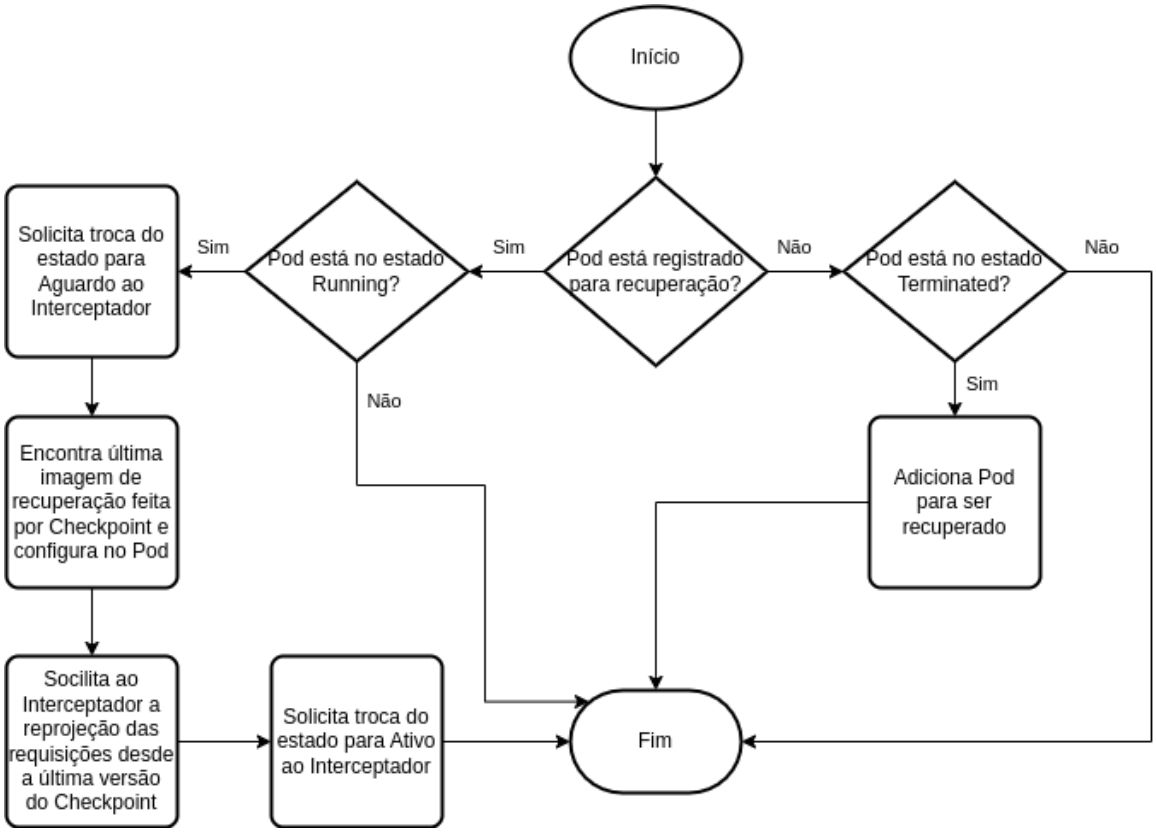


Figura 10 – Fonte: Elaborado pelo autor (2023).

5.4.1.3 Controlador de Checkpoint

O controlador de Checkpoint realiza o ciclo de reconciliação para os recursos que nosso Operador criou do tipo Checkpoint. O recurso Checkpoint possui o intervalo que é passado através da anotação `crsc.io/checkpoint-interval` e o nome do Deployment que está sendo monitorado. Através dessa anotação nosso controlador cria um processo que será executado no intervalo dado para pedir um *Checkpoint* ao Interceptador da aplicação ao chamar a rota `/checkpoint` no Service do Interceptador. O Interceptador irá pedir um novo *Checkpoint* a API do kubelet e retornar sucesso. Após o sucesso, este controlador irá obter esse *Checkpoint* e criará uma imagem de contêiner no formato OCI com buildah.

Como este controlador também implementa uma parte do nosso Administrador de Estado da arquitetura geral, ele também seria o responsável por salvar informações de metadados sobre a imagem de salvamento criada. Entretanto, não foi possível criar uma aplicação que conseguisse utilizar as bibliotecas do buildah para criação da imagem, muitos problemas foram encontrados ao se utilizar overlayfs como comu-

nicação de disco e não houve tempo para realizar no nosso Serviço a geração da imagem. Porém, conseguimos criar uma imagem com buildah através da sequência de comandos no Código 9, que podem ser refeitos através da biblioteca e agregados ao Serviço.

Código 9 – Comandos do buildah para construir a imagem de recuperação a partir de um Checkpoint feito pelo CRIU através do kubelet.

```
1 newcontainer=$(buildah from scratch)
2 buildah add $newcontainer /var/lib/kubelet/checkpoints/checkpoint-<pod-name>_<
      namespace-name>-<container-name>-<timestamp>.tar /
3 buildah config --annotation=io.kubernetes.cri-o.annotations.checkpoint.name=<
      container-name> $newcontainer
4 buildah commit $newcontainer checkpoint-image:latest
5 buildah rm $newcontainer
```

Embora o *Checkpoint* impeça as requisições de chegarem à aplicação alvo diretamente, ele ocorre de forma transparente ao usuário da aplicação. O desenvolvedor não necessita configurar nada na aplicação e não há percepção para o usuário que um *Checkpoint* está ocorrendo além de um possível aumento de latência como visto em (SCHMIDT et al., n.d.). Estes *Checkpoints* periódicos seriam utilizados para recuperar a aplicação ao último estado de forma mais rápida e eficiente como em (VAYGHAN et al., 2021) (MÜLLER; MEINHARDT; MENDIZABAL, 2022) (OH; KIM, 2018) e (SCHMIDT et al., n.d.).

5.5 CHECKPOINT/RESTORE COM TÉCNICAS DE EVENT SOURCING

Na nossa implementação a partir de técnicas de Event Sourcing tínhamos o objetivo de prover uma forma de agilizar a restauração da aplicação sem o problema do tempo que leva para realizar o *Checkpoint* da imagem. Já esperávamos que a performance para aplicações com muitas requisições fosse pior, pois, necessitaríamos reenviar todas as requisições desde o início do estado de Pronto da aplicação. Mas, a partir dela poderíamos unir com a técnica de *Checkpoint* com CRIU para realizar ou uma ou outra dependendo do estado atual do sistema, eliminando o tempo gasto em um *Checkpoint* no começo do estado de vida da aplicação. Assim, teremos algumas modificações para alguns controladores que mostramos na Seção passada, já que alguns passos não são realizados para nossa implementação com técnicas de *Event Sourcing*.

5.5.1 Controladores

5.5.1.1 Controlador de Deployment

O controlador de Deployment para a implementação com técnicas de *Event Sourcing* é a mesma que para a implementação com CRIU, exceto, que, não criamos o recurso de Checkpoint neste caso, pois, não há necessidade. Devido ao fato de sempre estarmos salvando as requisições interceptadas pelo Interceptador, sempre temos a ordem de requisições que precisamos executar para obter o mesmo estado na aplicação alvo. Não precisamos realizar um *Checkpoint* periódico como no outro caso, pois, o salvamento de estado está nos eventos que são gerados pelas requisições. Neste controlador, também adicionamos o *sidecar container* ao Deployment e criamos o Service que redireciona requisições ao Interceptador. O Interceptador tem sempre a mesma implementação, o que permite ser utilizado em diferentes sistemas e diferentes contextos, já que tudo que ele provê é uma API.

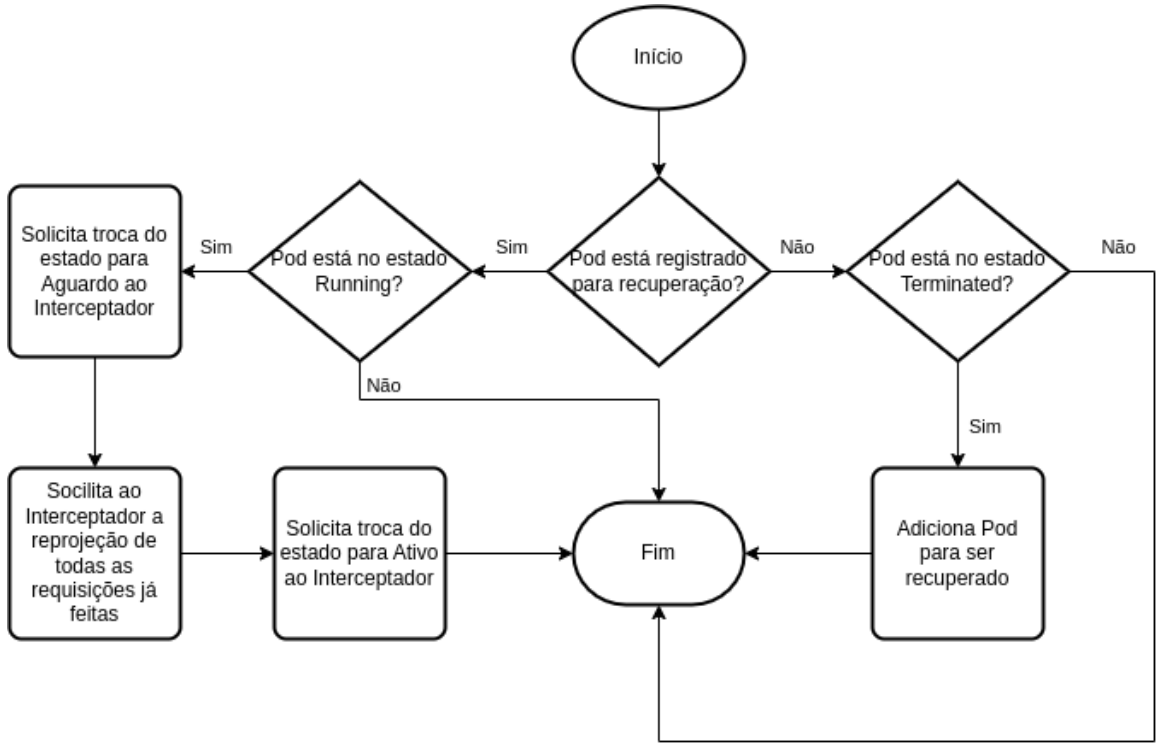
5.5.1.2 Controlador de Pod

O controlador de Pod para a implementação com técnicas de Event Sourcing funciona diferente da implementação com CRIU. Neste caso não precisamos alterar a imagem da aplicação alvo, ela permanece a mesma sempre, tudo que precisamos realizar é a sequência na Figura 11. Primeiro identificamos um contêiner falhante a partir da atualização do kube-apiserver, adicionamos ele a uma fila de restauração caso ele seja monitorado através das anotações. Posteriormente a isso, quando o Pod atinge o estado de Pronto, recebemos mais uma atualização. Caso o Pod esteja na fila de recuperação o Interceptador será colocado no estado Aguardo e será solicitada, então, a reprojeção das requisições. Após o fim da reprojeção das requisições, que pode demorar, já que devemos fazer todas desde o início do funcionamento da aplicação alvo, alteramos o estado do Interceptador para Pronto novamente. Novamente, de forma transparente recuperamos o contêiner da aplicação alvo para um estado consistente.

Desta forma, teríamos uma implementação que supostamente perderia performance dado o número das requisições. Quanto mais requisições o sistema recebe, mais demorado seria a reprojeção de todas as requisições, iremos validar este ponto na próxima Seção. Entretanto, vale notar que esta é uma implementação não tão abordada em outros trabalhos, exceto em (MÜLLER; MEINHARDT; MENDIZABAL, 2022) que utiliza o conceito até certo ponto para replicar as requisições a partir do último ponto de salvamento anterior à recuperação da aplicação, o que diminui a latência em se recuperar a aplicação.

Como inicialmente pretendíamos adicionar as duas implementações, uma com CRIU e outra com *Event Sourcing*, pretendíamos aproveitar os pontos positivos de cada

Figura 11 – Diagrama de fluxo para a recuperação da aplicação alvo através de um Controlador de Pods para o Operador na implementação com técnicas de *Event Sourcing*.



Fonte: Elaborado pelo autor (2023).

uma. Entretanto, apenas a versão da implementação com técnicas de *Event Sourcing* foi finalizada e, pretendemos, na próxima Seção, investigar se essa implementação realmente sofre de todas as partes negativas que apontamos nesta Seção.

6 ANÁLISE EXPERIMENTAL

Este capítulo apresenta a análise experimental sobre o Serviço criado. Para que seja possível validar o Serviço uma aplicação de testes foi criada em Go, que é um acrescentador de valores em memória. Sempre que uma requisição chega a ele, este incrementa um no seu valor em memória. Esta é uma aplicação *Stateful* e, portanto, é passível de ser utilizada no nosso Serviço. Este capítulo pretende responder a algumas perguntas a partir da análise dos experimentos:

- Existe impacto ao se utilizar o Interceptador para interceptar requisições à aplicação alvo?
- Qual o impacto de tempo para a aplicação ser restaurada com a implementação do *Checkpoint/Restore* com técnicas de *Event Sourcing*?
- Qual a latência da aplicação durante a recuperação da aplicação pelas técnicas de *Event Sourcing*?

Para responder estas perguntas fizemos primeiro uma instrumentação no Serviço para que pudessemos coletar pontos importante para análise, como, por exemplo, momentos em que a aplicação alterou o estado do Interceptador, momento da introdução da falha do Interceptador, momento da detecção pelo Administrador de Estado e tempo até a aplicação retornar ao estado de Pronto. A análise de teste de carga para determinar a vazão e latência das requisições com diferentes cargas no sistema. A partir disto pretendemos responder as perguntas para esta análise. É importante destacar, que como a aplicação executa no mesmo nó, não há perdas significativas de latência para comunicação entre serviços.

Para realizar o teste de carga, utilizamos a aplicação bombardier (BOMBARDIER, 2023), que permite realizar um número arbitrário de requisições concorrentes, definindo a quantidade de requisições e o tamanho da concorrência, que nos permite testar a aplicação sob uma carga mais real. Para coleta de outras métricas utilizamos horário de início e fim de cada uma das modificações. O Serviço e a aplicação executam na mesma máquina o horário é igual para todos e não é necessário sincronização de relógios entre máquinas.

6.1 ANÁLISE DA INTERFERÊNCIA DO INTERCEPTADOR NA APLICAÇÃO

Para fins de entender o quanto o Interceptador interfere no serviço que ele monitora, instrumentamos a aplicação e utilizamos um teste de carga para verificar a performance em três casos, da aplicação *standalone*, isto é, sem a interferência do Interceptador, da aplicação com o Interceptador utilizando memória para armazenamento das requisições e da aplicação com o Interceptador utilizando um banco de

dados SQL para armazenamento das requisições. Realizamos os mesmos testes de carga utilizando o bombardier para realizar requisições com N clientes paralelamente durante dez segundos. A partir da finalização das requisições obtivemos o valor de latência para o p95 da distribuição das requisições pudemos construir um gráfico de vazão por latência para os três casos, que é mostrado na Figura 12, onde tomamos um acréscimo de cinquenta clientes a cada novo ponto para a construção das curvas até dez mil clientes. Alguns pontos destacados nas curvas facilitam a comparação entre elas, mas, também são destacados pontos importantes em momentos diferentes da curva, pois, nem todos os testes puderam ser finalizados com cada implementação.

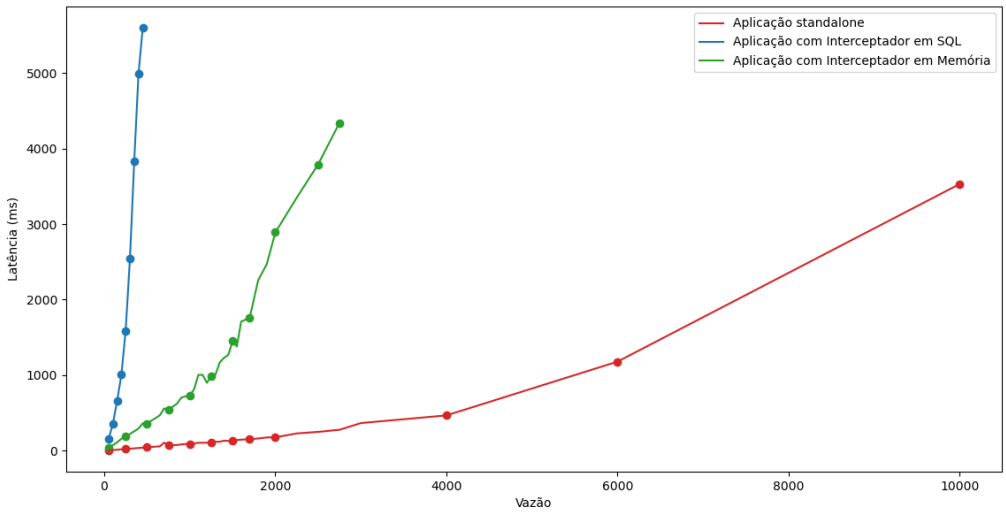
O gráfico da Figura 12 nos provê informações que já esperávamos, mas serão confirmadas por ele. Na aplicação *standalone*, linha em vermelho, podemos ver que demora muito para a aplicação atingir seu máximo de sobrecarga, somente entre seis mil e dez mil clientes acabamos tendo uma performance que só piora a partir do aumento da utilização do serviço. Inicialmente o serviço tem uma latência muito pequena, por exemplo, com cinquenta clientes, temos apenas 5.11ms de latência.

A partir do momento que introduzimos o Interceptador temos uma piora, tanto da latência inicial, quanto uma piora na questão da máxima vazão da aplicação. No caso da aplicação utilizando uma implementação de armazenamento em memória, linha verde na Figura 12, temos que o máximo de carga do serviço se dá entre dois mil a dois mil e setecentos clientes simultâneos. Há uma piora direta logo nos primeiros cinquenta clientes, onde a latência p95 tem seu valor 38.55ms. Isto mostra que o impacto do nosso Interceptador e pelas operações que ele faz, como, por exemplo, processamento da requisição obtendo apenas as partes necessárias para o armazenamento, acréscimo da versão de ordem da requisição, envio da requisição para a aplicação e posterior retorno da resposta para o cliente, geram uma sobrecarga que influencia na performance do sistema como um todo. É importante notar que a implementação em memória que utilizamos não tem possibilidade de realizar o *Checkpoint*, já que não há memória infinita e não poderíamos armazenar todas as requisições em memória que seriam necessárias para a reprojeção. Uma possibilidade de melhoria dessa abordagem, que não foi feita, seria armazenar parte em memória e parte em um armazenamento fixo, como um banco de dados ou em um arquivo.

Já na implementação do Interceptador com armazenamento em um banco de dados SQL, linha azul na Figura 12, temos a pior performance de todas. Isto se deve principalmente por dois fatores, da sobrecarga do sistema que realizamos os experimentos, já que este estava executando o Kubernetes, um banco de dados, nosso Serviço e a aplicação em uma máquina de 2vCPU e 4GB de memória, e também pela latência na inserção das entradas de requisições no banco de dados. Como podemos ver na Figura 12, a capacidade máxima do sistema é atingida muito rapidamente, entre trezentos e quinhentos clientes simultâneos. A aplicação também apresenta a pior performance em latência já no começo dos experimentos com cinquenta clientes si-

multâneos, com uma latência de 151.94ms, quase 400% a mais que na implementação com o Interceptador com armazenamento em memória e quase 3000% a mais que na aplicação *standalone*. Para utilizar essa solução em um ambiente de produção seria necessário deixar a aplicação mais performática, através da possibilidade de utilização de bancos mais rápidos, como de chave/valor ou de documentos.

Figura 12 – Gráfico de sobreposição de vazão X latência da aplicação alvo de forma *standalone*, da aplicação alvo com o Interceptador com armazenamento em memória e da aplicação alvo com o Interceptador com armazenamento em banco de dados SQL.



Fonte: Elaborado pelo autor (2023).

Para nível de comparação no impacto do banco de dados na aplicação, instrumentamos o código do Interceptador para capturar métricas sobre a latência em escrever a requisição no banco de dados. Nas requisições com trezentos clientes simultâneos que obtemos uma latência p95 de 1590ms, tivemos uma latência p95 para inserção da requisição no banco de 1117ms, um impacto de cerca de 70.25% na latência da aplicação. Uma estratégia seria combinar a estratégia de implementação do armazenamento em memória com o armazenamento em banco de dados SQL. Teríamos uma lista com tamanho fixo de dados, que armazenaria as requisições em memória, onde a partir do momento em que ela excedesse sem máximo, iriamos escrever os dados no banco de dados uma única vez, e continuaríamos sobrescrevendo a lista. Com isto, iríamos diminuir a latência de escrita, já que seria feita menos vezes e aumentar a eficiência pelo armazenamento em memória.

6.2 ANÁLISE DO PERÍODO DE TEMPO DE CHECKPOINT DO INTERCEPTADOR

Nesta Seção, pretendemos discutir e analisar qual o tempo gasto pelo *Checkpoint* do Interceptador na implementação com CRIU. Na implementação com técnicas de *Event Sourcing* não temos uma alteração, por exemplo, da latência das requisições ao se fazer um *Checkpoint*, pois este ocorre passivamente. Já na implementação com CRIU a aplicação deve ser pausada por um momento para salvamento do seu estado e isto gera uma piora na qualidade de serviço para o usuário de maneira momentânea. Para simular este estado, iniciamos realizando um teste de carga na aplicação para criar um estado, a partir daí, iniciamos ao mesmo tempo um *Checkpoint* a partir do Interceptador e um outro teste de carga, que deve durar mais tempo que o *Checkpoint*.

Coletamos dados para a latência das requisições que atingem o Interceptador em um período de tempo que inclui antes de uma falha na aplicação alvo, durante a recuperação da aplicação alvo e posteriormente da recuperação no mesmo teste. Estes testes nos proveram uma latência p95 de 7.3s para as requisições do período de tempo, onde já existiam mil requisições acumuladas para que ocorresse a reprojeção durante a recuperação e os tínhamos cem clientes realizando mil requisições simultâneas para a aplicação durante o período de tempo.

Novamente, tivemos uma piora de latência do serviço, que se deve principalmente pelo fator da nossa aplicação ter que realizar todas as requisições já feitas para a aplicação atingir o estado antes da falha na recuperação. Na proposta de (MÜLLER; MEINHARDT; MENDIZABAL, 2022) a arquitetura realiza a poda das requisições, onde, menos requisições teriam que ser refeitas a partir de uma falha para a recuperação através da criação de imagens de *Checkpoint* utilizando o CRIU.

6.3 ANÁLISE DA RECUPERAÇÃO COM TÉCNICAS DE EVENT SOURCING

Nesta Seção queremos investigar o impacto das técnicas de *Event Sourcing* na recuperação do estado de uma aplicação que falhou. Para isso, primeiro utilizamos nosso teste de carga para levar a aplicação até um estado, com mil requisições(caso A), dez mil requisições(caso B) e vinte mil requisições(caso C), com mais requisições acumuladas o serviço degrada a ponto de não possibilitar testes com as configurações de *cluster* que utilizamos. Depois repetimos o mesmo teste com cada uma delas, ao alcançar o estado a aplicação terá uma falha ocasionada pela interrupção da execução do contêiner pela ferramenta crictl. Após a falha, a recuperação deve ocorrer, neste momento, passamos a realizar um teste de carga igual para todos os casos, cem requisições serão feitas de maneira concorrente entre dez clientes. Todos os testes foram realizados em um conjunto de dez tentativas, onde se tomou como valor para análise o resultado do p95 entre eles.

Os resultados do teste estão na Tabela 1. Verificamos, que, temos uma

Tabela 1 – Latência durante recuperação de requisições acumuladas.

Número de requisições acumuladas	**Latência**
1000(A)	74.69ms
10000(B)	122.76ms
20000(C)	10010ms

Fonte: Elaborado pelo autor (2023).

degradação relevante entre dez mil requisições acumuladas e vinte mil requisições acumuladas, indo de 122.76ms de latência para 10010ms de latência para as requisições. Novamente, quanto mais requisições acumuladas o Interceptador possui, mais demorada se torna uma recuperação ao realizar todas as requisições novamente para a aplicação alvo. Esta análise mostra uma premissa que já tínhamos, de que existe um limite para utilização das técnicas de *Event Sourcing* sobre o *Checkpoint/Restore*, já que, conforme mais tempo a aplicação vive, mais se torna necessário utilizar outras técnicas para ser possível reduzir a replicação das requisições na recuperação, em (MÜLLER; MEINHARDT; MENDIZABAL, 2022) a diminuição da pilha foi feita através da implementação da recuperação com CRIU.

Agora, pretendemos também investigar o tempo que levamos entre a detecção, a recuperação e atingir o estado de pronto em cada caso, como em (VAYGHAN et al., 2021), adicionamos métricas ao controlador de Pods, para definir a detecção, e no nosso *script* identificamos o início da falha, a recuperação é definida nos momentos em que se altera o estado do Interceptador e o estado de Pronto também é instrumentado pelo controlador de Pods. Na Tabela 2 temos os valores para o início da recuperação após uma falha e o tempo até a aplicação estar disponível novamente após a recuperação. No caso C temos um tempo muito superior aos casos A e B, isto se deve por vários fatores, desde a configuração do nosso *cluster* que não permite escalabilidade e também da necessidade de se obter todas as requisições de um banco de dados para replicação das requisições. A Tabela 2 demonstra que a implementação com técnicas de *Event Sourcing* degrada conforme temos mais requisições acumuladas no Interceptador para realizar a recuperação.

Tabela 2 – Tempos de identificação de recuperação e de recuperação final de um contêiner alvo.

Número de requisições acumuladas	**Tempo de detecção (segundos)**	**Tempo de Recuperação (segundos)**
1000(A)	0.991011s	0.990702s
10000(B)	0.998153s	0.975395s
20000(C)	1.013021s	856.67s

Fonte: Elaborado pelo autor (2023).

Na tabela 2 percebemos que nosso limite da restauração quanto ao número de requisições armazenadas está entre dez mil requisições e vinte mil requisições. O au-

mento de cerca de 1s no tempo de restauração para cerca de 14min mostra uma perda de performance muito grande. Seria essencial integrar outros *Checkpoints* durante a interceptação das requisições com o método de técnicas de *Event Sourcing* para que fosse possível diminuir o tempo de latência na recuperação a partir da dimunição da quantidade de requisições armazenadas antes do último *Checkpoint*.

7 CONSIDERAÇÕES FINAIS

O trabalho desenvolvido nesse Trabalho de Conclusão de Curso, embora, não tenha sido completamente finalizado, possibilitou uma análise sobre as técnicas de *Checkpoint/Restore* para aplicações *Stateful* no Kubernetes. Através da ferramenta CRIU, propusemos uma forma de implementar o *Checkpoint/Restore*, que embora não tenha sido possível ser implementada no nosso Serviço, provou-se útil para realizar o salvamento do estado e posterior recuperação de uma aplicação. Já na implementação com técnicas de *Event Sourcing* conseguimos realizar o Serviço de *Checkpoint/Restore* no Kubernetes com transparência, provendo uma cópia utilizável da aplicação na tolerância a falhas a partir da reprojeção das requisições feitas a aplicação.

O Serviço desenvolvido ainda possui algumas falhas, além de não conseguir implementar o salvamento e recuperação com CRIU. Este só funciona com aplicações de estado que possuam uma réplica, mais réplicas geram estados inconsistentes entre as aplicações. Também não houve muito foco na segurança do Serviço, já que temos acesso às chaves de comunicação com a API do Kubernetes em nosso Interceptador, qualquer um que tenha acesso ao Interceptador, por exemplo, através de uma exposição pela internet HTTP poderia obter acesso às chaves da API do Kubernetes e obter e alterar o estado do Kubernetes, expondo inclusive Secrets que devem funcionar como segredos no *cluster*.

A performance do nosso Serviço fica em evidência no Capítulo 6, onde verificamos que a implementação necessita de melhorias na performance para que seja possível ser utilizada em aplicações em ambientes de produção. Como mostramos no capítulo existe uma evidência de que o armazenamento persistente através de um banco de dados SQL gera muita sobrecarga no sistema. Precisamos trabalhar em cima deste ponto, com modificações unindo implementação com armazenamento em memória e *Checkpoint* com CRIU. Esta perda de performance já era esperada no início do trabalho.

Por fim, o trabalho cumpriu com o seu objetivo, que era implementar um Serviço de *Checkpoint/Restore* para aplicações *Stateful* no Kubernetes. Conseguimos acoplar uma aplicação ao nosso Serviço a partir da utilização de anotações no Deployment, permitindo que o usuário do Serviço não tenha que realizar configurações na sua aplicação, realizar *Checkpoints* passivos com técnicas de *Event Sourcing* e restaurar a aplicação no caso de uma falha de forma transparente, onde a aplicação é restaurada para seu último estado de funcionamento. Também conseguimos entregar uma parte clara de como integrar o *Checkpoint* com CRIU no nosso Serviço através da utilização do buildah para construção das imagens de recuperação.

7.1 TRABALHOS FUTUROS

Como trabalhos futuros podemos listar alguns pontos que faltaram fechar nestes trabalho. O primeiro é a implementação com CRIU para o *Checkpoint/Restore*. Este trabalho mostrou que é possível utilizar a arquitetura proposta para realizar o *Checkpoint/Restore*, mas não houve a finalização da implementação com CRIU, porém listamos as partes e problemas que faltam para ser efetivada a implementação. Com uma implementação com CRIU, podemos unir a implementação com técnicas de *Event Sourcing* para criar um serviço adaptativo que permita trocar o tipo de restauração e definir o momento do início dos salvamentos de estado ativos através do CRIU com base na análise da performance das duas implementações, seguindo os trabalhos de (MÜLLER; MEINHARDT; MENDIZABAL, 2022), onde nossa arquitetura foi inspirada.

Neste trabalho em momento algum utilizamos técnica preditivas para detectar o momento que a aplicação poderia falhar, então, em determinados momentos poderíamos ter uma requisição interceptada pelo Interceptador e que não tenha sido enviada à aplicação com falhas. Em (TRAN; VU; KIM, 2022) houve uma implementação de predição, que poderia ser adicionada a nossa implementação para prover um Serviço mais ativo na recuperação, que não espere a falha acontecer para agir.

Nosso trabalho não agiu com aplicações distribuídas geograficamente; Isto mostra uma nova perspectiva, pois, será necessário espalhar os conteúdos tanto do Interceptador quanto de imagens geradas para outros nós geograficamente distribuídos o que gera latência de rede e tempo de restauração diferente dependendo da localização geográfica do nó que falha. No trabalho de (VAYGHAN et al., 2021), tivemos uma análise da distribuição geográfica e como poderíamos fazer para diminuir a latência através da distribuição dos dados armazenados persistentemente.

O trabalho também não cobre aplicações que tenham mais de uma réplica com estado da mesma aplicação. Neste Serviço cobrimos apenas aplicações com uma réplica. Aplicações com diversas réplicas necessitam de coordenação entre os interceptadores para acordar em uma ordem das requisições com algoritmos de consenso para resolver estes problemas. Aliado ao problema de aplicações geograficamente distribuídas também temos o consenso entre estas os diferentes nós geograficamente distribuídos.

Por fim, é possível aproveitar ainda mais de conceitos de *Event Sourcing*. A aplicação alvo se instrumentada também com este padrão de projeto pode prover uma forma de obtermos o estado do sistema dela e salvarmos temporariamente no nosso Serviço. Esta cópia pode ser o estado de cada objeto armazenado no sistema, no caso de um banco de dados chave/valor, e a partir da finalização de uma requisição poderíamos salvar esta cópia no Administrador de Estados e excluir requisições anteriores a ela, reprojetando apenas as requisições a partir desta cópia.

REFERÊNCIAS

BOMBARDIER. **codesenberg/bombardier: Fast cross-platform HTTP benchmarking tool written in Go**. [S.l.: s.n.], 2023. https://github.com/codesenberg/bombardier. [Online; acessado em 11/08/2023].

BUILDAH. **Buildah | buildah.io**. [S.l.: s.n.], 2023. https://buildah.io/. [Online; acessado em 01/11/2023].

CALICO. **Project Calico**. [S.l.: s.n.], 2023. https://www.tigera.io/project-calico/. [Online; acessado em 11/08/2023].

CHEN, Yang. Checkpoint and Restore of Micro-service in Docker Containers. In: PROCEEDINGS of the 3rd International Conference on Mechatronics and Industrial Informatics. [S.l.]: Atlantis Press, 2015/10. P. 915–918. DOI: 10.2991/icmii-15.2015.160. Available from: <https://doi.org/10.2991/icmii-15.2015.160>.

CONTAINERD. **containerd - An industry-standard container runtime with an emphasis on simplicity, robustness and portability**. [S.l.: s.n.], 2023. https://containerd.io/. [Online; acessado em 03/05/2023].

CORMIER, Paul. **The State of Enterprise Open Source: A Red Hat report**. [S.l.: s.n.], 2023. https://www.redhat.com/en/resources/state-of-enterprise-open-source-report-2022. [Online; acessado em 09/04/2023].

CRI-O. **cri-o**. [S.l.: s.n.], 2023. https://cri-o.io/. [Online; acessado em 03/05/2023].

CRIU. **CRIU**. [S.l.: s.n.], 2023. https://criu.org/Main_Page. [Online; acessado em 03/05/2023].

DOCKER. **Swarm mode overview | Docker Documentation**. [S.l.: s.n.], 2023. https://docs.docker.com/engine/swarm/. [Online; acessado em 02/05/2023].

ETCD. **etcd**. [S.l.: s.n.], 2023. https://etcd.io/. [Online; acessado em 08/05/2023].

KUBERNETES. [S.l.: s.n.], 2023. https://kubernetes.io/docs/concepts/storage/persistent-volumes/. [Online; acessado em 09/05/2023].

KUBERNETES. **Controllers | Kubernetes**. [S.l.: s.n.], 2023. https://kubernetes.io/docs/concepts/architecture/controller/. [Online; acessado em 09/05/2023].

KUBERNETES. **Deployments Kubernetes**. [S.l.: s.n.], 2023. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. [Online; acessado em 09/04/2023].

KUBERNETES. **Kubelet Checkpoint API | Kubernetes**. [S.l.: s.n.], 2023. https://kubernetes.io/docs/reference/node/kubelet-checkpoint-api/. [Online; acessado em 27/06/2023].

KUBERNETES. **Kubelet Checkpoint API | Kubernetes**. [S.l.: s.n.], 2023. https://kubernetes.io/docs/reference/node/kubelet-checkpoint-api/. [Online; acessado em 09/11/2023].

KUBERNETES. **Kubernetes Components | Kubernetes**. [S.l.: s.n.], 2023. https://kubernetes.io/docs/concepts/overview/components/. [Online; acessado em 03/05/2023].

KUBERNETES. **Operating etcd clusters for Kubernetes | Kubernetes**. [S.l.: s.n.], 2023. https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/. [Online; acessado em 08/05/2023].

KUBERNETES. **Operator pattern | Kubernetes**. [S.l.: s.n.], 2023. https://kubernetes.io/docs/concepts/extend-kubernetes/operator/. [Online; acessado em 04/11/2023].

KUBERNETES. **Overview Kubernetes**. [S.l.: s.n.], 2023. https://kubernetes.io/docs/concepts/overview/. [Online; acessado em 27/03/2023].

KUBERNETES. **Pods Kubernetes**. [S.l.: s.n.], 2023. https://kubernetes.io/docs/concepts/workloads/pods/. [Online; acessado em 09/04/2023].

KUBERNETES. **Volumes | Kubernetes**. [S.l.: s.n.], 2023. https://kubernetes.io/docs/concepts/storage/volumes/. [Online; acessado em 09/05/2023].

LAADAN, Oren; HALLYN, Serge E. Linux-CR: Transparent application checkpoint-restart in Linux. In: CITESEER. LINUX Symposium. [S.l.: s.n.], 2010.

MICROSOFT. **Ambassador pattern - Azure Architecture Center | Microsoft Learn**. [S.l.: s.n.], 2023. `https://learn.microsoft.com/en-us/azure/architecture/patterns/ambassador`. [Online; acessado em 17/06/2023].

MICROSOFT. **Event Sourcing pattern - Azure Architecture Center | Microsoft Learn**. [S.l.: s.n.], 2023. `https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing`. [Online; acessado em 25/06/2023].

MÜLLER, Rodrigo H; MEINHARDT, Cristina; MENDIZABAL, Odorico M. An architecture proposal for checkpoint/restore on stateful containers. In: PROCEEDINGS of the 37th ACM/SIGAPP Symposium on Applied Computing. [S.l.: s.n.], 2022. P. 267–270.

OH, SeungYong; KIM, JongWon. Stateful container migration employing checkpoint-based restoration for orchestrated container clusters. In: IEEE. 2018 International Conference on Information and Communication Technology Convergence (ICTC). [S.l.: s.n.], 2018. P. 25–30.

POSTGRESQL. **PostgreSQL: The world's most advanced open source database**. [S.l.: s.n.], 2023. `https://www.postgresql.org/`. [Online; acessado em 04/11/2023].

REDHAT. **Red Hat OpenShift enterprise Kubernetes container platform**. [S.l.: s.n.], 2023. `https://www.redhat.com/en/technologies/cloud-computing/openshift`. [Online; acessado em 03/05/2023].

SCHMIDT, Henri. **State-preserving container orchestration in failover scenarios**. 2023. MA thesis.

SCHMIDT, Henri et al. Transparent Fault Tolerance for Stateful Applications in Kubernetes with Checkpoint/Restore.

TRAN, Minh-Ngoc; VU, Xuan Tuong; KIM, Younghan. Proactive Stateful Fault-Tolerant System for Kubernetes Containerized Services. **IEEE Access**, IEEE, v. 10, p. 102181–102194, 2022.

VAYGHAN, Leila Abdollahi et al. A Kubernetes controller for managing the availability of elastic microservice based stateful applications. **Journal of Systems and Software**, Elsevier, v. 175, p. 110924, 2021.

WHITE, Brian et al. An Integrated Experimental Environment for Distributed Systems and Networks. In: USENIX ASSOCIATION. PROC. OF the Fifth Symposium on Operating Systems Design and Implementation. Boston, MA: [s.n.], Dec. 2002. P. 255–270.

Anexo

ANEXO A – ARTIGO

Inclusão de um artigo científico no padrão da Sociedade Brasileira de Computação sobre o trabalho desenvolvido no Trabalhao de Conclusão de Curso.

Implementação de Serviço de Checkpoint/Restore para Stateful Containers no Kubernetes

Gian A. Ortiz[1]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Florianópolis – SC – Brazil

Abstract. *The world of distributed systems is seeing the increase in the usage of lightweight virtualization, also known as containers, to create applications, like microservices distributed in cloud systems. But, the usage of containers for distributed system has many challenges on the high availability of the service. Container orchestrators, like Kubernetes, solve this challenges by providing tools to manage containers, scale them and distribute them. Although, the high availability of Stateful containers has other challenges than Stateless containers, like the fault tolerance on achieving the same state that the previous failed application, this is the focus of this work. To achieve the same state as the previous failed container, we use Checkpoint/Restore techniques and the Event Sourcing pattern to checkpoint the state of the application and when it fails we restore it to the previous state. The usage of the Event Sourcing pattern is to investigate a new way to achieve the Checkpoint/Restore, also we show how to achieve the same result using traditional Checkpoint/Restore on Linux with CRIU. This work creates a framework for Kubernetes for fault tolerance with Checkpoint/Restore using Event Sourcing.*

Resumo. *Em sistemas distribuidos, percebemos a alta difusão da utilização de virtualização leve, através de contêineres, para criação de implantação de novas aplicações, principalmente entre microsserviços e computação distribuída em serviços em nuvem. Entretanto, este método possui alguns desafios para prover alta disponibilidade dos serviços. Orquestradores de contêneires resolvem muitos dos desafios, como a escalabilidade, a distribuições e administração de contêneires em clusters. Porém, alguns desafios persistem, como o tema deste trabalho que é a alta disponibilidade para serviços em contêineres do tipo Stateful. Estes tipos de serviços necessitam que o estado seja persistido em memória e, em uma falha, seja possível reiniciar o serviço para o mesmo estado anterior à falha. Neste trabalho, utilizamos técnicas de Checkpoint/Restore para realizar o salvamento do estado de um serviço e posteriormente realizar a restauração do serviço, em caso de uma falha. Juntamente com estas técnicas, também aplicamos as técnicas do padrão de projeto de Event Sourcing para prover diferentes tipos de recuperação no trabalho. Ainda, focamos em prover uma solução que funcione de maneira transparente no orquestrador de contêneires Kubernetes, que é atualmente o mais utilizado. Por fim, conseguimos prover um serviço de Checkpoint/Restore para Kubernetes em contêineres do tipo Stateful através de técnicas do padrão de projeto Event Sourcing utilizando uma arquitetura agnóstica a tecnologias. Também conseguimos abordar uma forma de realizar o mesmo salvamento do estado e*

posterior recuperação utilizando CRIU e buildah para construção de imagens de recuperação do estado seguindo o padrão do Open Container Initiative.

1. Introdução

Com o crescimento de técnicas de utilização de microsserviços e computação em nuvem, se criou uma alta necessidade para a alta disponibilidade dos serviços [Vayghan et al. 2021]. A virtualização leve através da utilização de contêineres, desenvolvidos em cima das funcionalidades do Linux, como *cgroups* e *namespaces*, que possibilitam a limitação e priorização de recursos sem máquinas virtuais e a isolação completa de processos, redes e arquivos, respectivamente, possibilitou uma melhoria no processo de disponibilização destes microsserviços na nuvem [Laadan and Hallyn 2010]. Desta forma, para facilitar o processo de gerenciamento e comunicação entre diversos contêineres de microsserviços na nuvem, aplicações com este intuito foram criadas [Vayghan et al. 2021], chamadas de orquestradores de contêineres, entre elas a mais popular é o Kubernetes[Kubernetes 2023b].

Em geral, os orquestradores de contêineres provêem um grande ferramental para a replicação, recuperação e manutenção de alta disponibilidade para os contêineres, suprindo a maioria das necessidades que aplicações sem estado de memória interna, do tipo *Stateless*, possuem [Vayghan et al. 2021]. Entretanto, quando precisamos de aplicações que possuam estado em memória, aplicações *Stateful*, e, que este estado seja importante para as futuras execuções da aplicação, falta suporte nos orquestradores de contêineres, principalmente em adoção das práticas de *Checkpoint/Restore* [Müller et al. 2022] para o melhoria do tempo da recuperação de contêineres com falhas. Este tipo de técnica permite salvar o estado de memória em uma nova imagem para recuperá-la mais tarde. Embora, orquestradores de contêineres possuam suporte para aplicações com estado em armazenamento de dados persistente, como o StatefulSet no Kubernetes, não existe suporte nativo para o *Checkpoint* de estado em memória da aplicação[Tran et al. 2022].

Desta forma, este trabalho desenvolve um serviço transparente para realizar o *Checkpoint/Restore* de aplicações *Stateful* executando em contêineres no Kubernetes. Este serviço possibilita a tolerância a falhas das aplicações com estado, onde, quando uma falha ocorrer a aplicação será reiniciada ao último ponto de execução antes da falha.

2. Trabalhos Relacionados

Existem alguns trabalhos relacionados na área de *Checkpoint/Restore* em *Stateful Containers*. Selecionamos os principais, onde podemos agrupá-los em orientados à migração de contêineres e orientados à tolerância a falhas. No primeiro grupo, temos o trabalho [Müller et al. 2022] que propõe uma arquitetura de uso geral para *Checkpoint/Restore* de *Stateful Containers*, o nosso trabalho se inspirou nesta arquitetura para implementação da nossa. A arquitetura proposta consiste de cinco componentes, que realizam funções diferentes e juntos compõe um *framework* que realiza o *Checkpoint/Restore*. Por fim, é demonstrado uma implementação básica, sem dar muita ênfase na análise de dados. Já em [Vayghan et al. 2021], que também está no grupo de tolerância a falhas, temos uma implementação de uma aplicação para *Checkpoint/Restore* no Kubernetes utilizando CRIU. O trabalho consiste em manter sempre duas réplicas da aplicação monitorada,

chamadas de Ativa e Inativa. Periodicamente o serviço implementado realiza um *Checkpoint* utilizando CRIU, através da implementação do padrão de projeto de Operador com controladores do Kubernetes da aplicação Ativa e recupera a aplicação Inativa com este. Quando a aplicação Ativa falha, a aplicação Inativa está com uma cópia próxima do último estado da Ativa, é então, que a aplicação Inativa se torna a Ativa e vice-versa. Por fim, no grupo de tolerância a falhas, temos o trabalho [Schmidt 2023], que implementa um serviço de *Checkpoint/Restore* com CRIU no Kubernetes. O trabalho realiza apenas a implementação de um Operador com controladores para controlar os *Checkpoints* periódicos e detectar falhas para realizar a restauração. Diferentemente do feito neste trabalho, que realiza, além do *Checkpoint*, também o armazenamento das requisições que podem ser utilizadas para recuperação.

Já no grupo de trabalhos relacionados com ênfase em migração de contêineres *Stateful*, temos o trabalho [Tran et al. 2022], que além de implementar uma aplicação para *Checkpoint/Restore* no Kubernetes, utiliza algoritmos de *machine learning* para prever quando um nó ou contêiner irá falhar para realizar o *Checkpoint* e restauração antes da falha. Também se implementa os mesmos algoritmos para descobrir falhas de nós que possivelmente irão falhar e desligar um conjunto de aplicações. Neste caso, todas as aplicações são migradas para um novo nó saudável. Por fim, neste grupo, temos o trabalho [Oh and Kim 2018], que cria uma aplicação de *Checkpoint/Restore* para realizar migração de aplicações do tipo *Stateful* em nós de *clusters* Kubernetes. Esta implementação pode ser utilizada caso exista uma manutenção programada do nó ou caso ele venha a apresentar problemas de lentidão e precise ser desligado.

3. Implementação

Para realizar a implementação primeiro descrevemos uma arquitetura geral para *Checkpoint/Restore* de *Stateful Containers* que é inspirada no trabalho em [Müller et al. 2022] e pode ser vista na 1. Nesta arquitetura temos cinco componentes que realizam o trabalho total de *Checkpoint/Restore*. Cada um possui um papel bem definido de sua funcionalidade.

O Contêiner é a representação da aplicação alvo do usuário para *Checkpoint/Restore*, que será monitorada pelo nosso serviço. Como nossa aplicação é um contêiner, definimos aqui o componente como Contêiner. Já o Interceptador é a aplicação que interceptará requisições vinda de clientes à aplicação alvo. Este funciona através do padrão de projeto Embaixador, interceptando as requisições, realizando possíveis operações sobre ela, encaminhando a requisição à aplicação alvo e reencaminhando a resposta ao cliente. Com isso, possibilitamos realizar a funcionalidade de armazenamento das requisições que recebemos, o que nos permite reprojetá-las futuramente. Também é papel do Interceptador realizar o *Checkpoint* e enviar metadados juntamente com o *Checkpoint* quando solicitado ao Administrador de Estado.

O Administrador de Estados é o componente central, que administra informações sobre aplicações e *Checkpoints*. Este deve identificar os momentos para solicitação de um novo *Checkpoint* e também para detecção de falhas para realizar uma restauração. Por fim, o Administrador de Estado, a partir do momento que recebe informações de *Checkpoint* pelo Interceptador, se comunica com o Volume e o Banco de Dados para armazenar a imagem de *Checkpoint* no primeiro e os metadados deste no último. Desta forma, com as

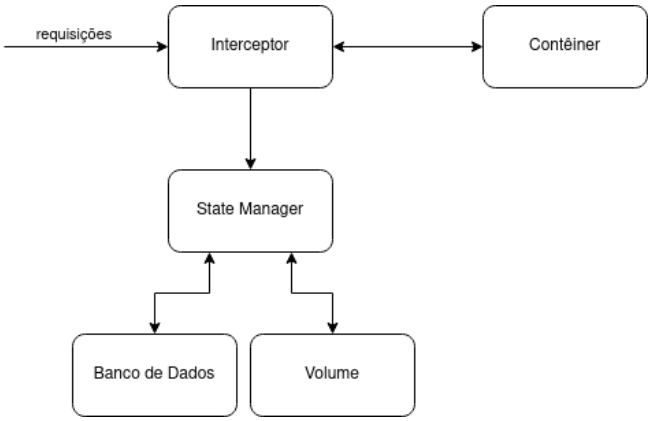


Figure 1. Arquitetura geral definida em cinco componentes, Contêiner, Interceptador, Administrador de Estado, Volume e Banco de Dados.

comunicações entre os componentes temos uma arquitetura geral que implementaremos no Kubernetes.

Para realizar a implementação da arquitetura no Kubernetes, utilizamos o padrão de projeto do Kubernetes, o Operador [Kubernetes 2023a]. Através deste, criamos extensões sobre a API do Kubernetes para adicionar novas funções ao Kubernetes através de controladores de recursos. Na nossa implementação de Operador para *Checkpoint/Restore* com técnicas de *Event Sourcing* utilizamos dois controladores, um para o recurso de Deployment e outro para o de Pod. Juntos os dois controladores fornecem as funcionalidades que são necessárias para o Administrador de Estado da arquitetura geral. Já o Interceptador é implementado através de uma aplicação padrão que é implantado através de contêineres. Na Figura 2 temos a implantação no Kubernetes.

Na Figura 2, o Interceptador é implementado como um *sidecar container* no mesmo Pod do contêiner da aplicação alvo. Isto possibilita a diminuição de latência entre comunicação, pois executam no mesmo nó, diminuindo latência de comunicação entre máquinas e diminui a comunicação da pilha de rede pela isolação de rede em um Pod. O Interceptador também consegue se comunicar com o Administrador de Estado através do recurso de Service do Kubernetes que provê um identificador para o Interceptador. Nosso Interceptador também possui um armazenamento para todas as requisições, que, posteriormente serão utilizadas para reprojetar um estado da aplicação através da técnica de Event Sourcing [Microsoft 2023].

Para implementação do controlador de Deployment, realizamos uma verificação de uma anotação no recurso. A anotação, `crsc.io/checkpoint-restore`, com valor de *true*, indica que o Deployment deve ser monitorado pelo nosso Serviço. Quando o controlador identifica esta anotação, ele adiciona este Deployment aos monitorados e também todos seus Pods que serão criados. Desta forma, conseguimos diferenciar aplicações que devem ser monitoradas das que não devem.

O controlador de Pod realiza toda a parte de recuperação. Já que não precisamos realizar um *Checkpoint* ativo com a implementação, onde este ocorre toda vez que uma nova requisição chega e é armazenada pelo Interceptador. Com isso, nosso controlador

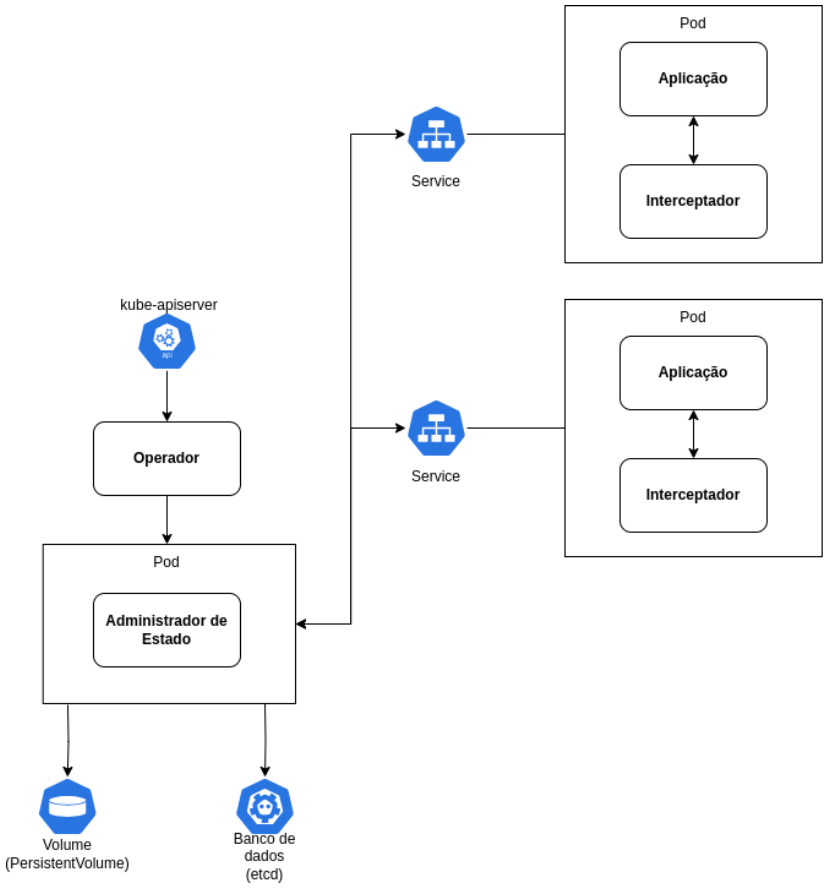


Figure 2. Adaptação da arquitetura geral para execução no Kubernetes.

utiliza o ciclo de reconcialiação do Kubernetes para identificar uma falha em Pods monitorados. Quando o Pod falha, ele é adicionado a fila de recuperação e, após o Kubernetes detectar a falha e recuperar a aplicação, colocamos nosso Interceptador em um modo de Aguardo, onde, não são servidas requisições à aplicação alvo. Então, pedimos a reprojeção de todas as requisições a ele, e, posteriormente, alteramos seu estado novamente para Ativo para continuar a servir requisições normalmente. Desta forma, possibilitamos a recuperação da aplicação ao seu último estado antes da falha através da técnica de Event Sourcing, que pode ser vista através da Figura 3.

4. Experimentos

Para avaliarmos a adição do Serviço em aplicações monitorados, bem como, analisar a eficiência do mesmo, configuramos um ambiente de testes e realizamos alguns experimentos para validação. O ambiente utilizado é hospedado na Google Cloud em uma máquina de 2vCPU e 4GB de memória. Instalamos Kubernetes, cri-o e CRIU na máquina monitorada, onde o Kubernetes deve ser executado com a *feature gate* de *Container Checkpoint* habilitada. Com estas modificações configuramos uma aplicação alvo que consiste de um somador em memória, onde, cada chamada HTTP à aplicação o valor armazenado em memória sofre um acréscimo.

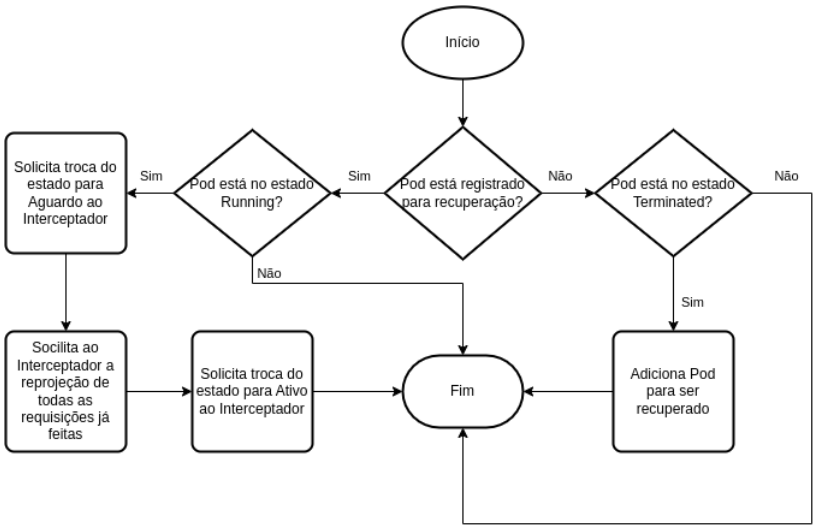


Figure 3. Diagrama de sequência da restauração de um Pod com falhas pelo Serviço.

O primeiro experimento consiste em avaliar o impacto da adição do Interceptador na aplicação alvo. Para isto realizamos testes de carga em três cenário. No cenário 1, temos a aplicação executando sem o Interceptador, o que chamamos de aplicação *standalone*. No cenário 2, temos a aplicação executando com o Interceptador em memória, que não é possível ser utilizado em produção pela disponibilidade finita de memória para armazenamento das requisições. Por fim, no cenário 3, temos a implementação que realiza *Checkpoint/Restore* com a versão do Interceptador com armazenamento em banco de dados SQL. Na Figura **??** temos o cenário 1 representado em vermelho, o cenário 2 representado em verde e o cenário 3 representado em azul. Os testes feitos foram de incrementos de cinquenta clientes simultâneos, ou threads, para avaliar o ponto de saturação da latência da aplicação. Todas as medidas de latência são percentil 95 dos testes realizados.

Pela análise da Figura **??**, percebemos que a aplicação *standalone* tem o melhor desempenho, como já era esperado. A aplicação com Interceptador em memória, verde, possui o segundo melhor desempenho, mas, já possui um ponto de saturação menor que o da aplicação *standalone*, demonstrando um impacto da presença do Interceptador. Já na aplicação com Interceptador com armazenamento em SQL temos o pior dos casos. Ao se analisar causas para perda de eficiência, percebemos que a adição de um banco de dados, dos controladores e do nosso Interceptador na máquina do Kubernetes gerava uma sobrecarga grande. Somente a inserção na tabela SQL resultava em uma parcela de 70% da latência no ponto de saturação da aplicação com Interceptador em armazenamento do banco SQL.

Considerando o impacto a partir do momento de detecção de falha e da recuperação através da reprojeção adicionamos duas tabelas com dados comparativos em três cenários, com mil requisições acumuladas para reprojeção, com dez mil requisições e com vinte mil requisições. Na Tabela 4, temos a comparação para a latência das requisições durante o período de recuperação. Na Tabela 4 temos a comparação para

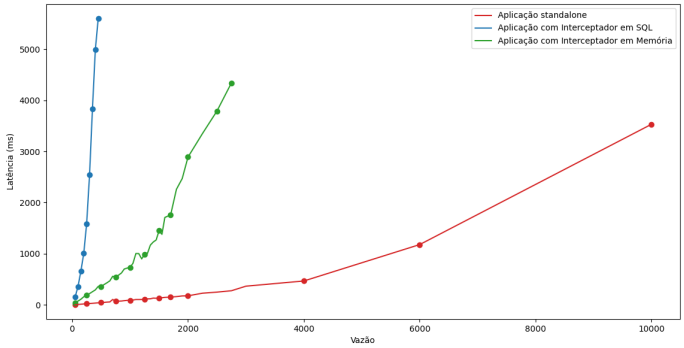


Figure 4. Gráfico de vazão por latência dos experimento. Em vermelho, a aplicação *standalone*, em verde a aplicação com Interceptador em memória e em vermelho, a aplicação com Interceptador com armazenamento em banco de dados SQL.

o tempo necessário para se realizar a recuperação. Percebemos através das tabelas que no momento de vinte mil requisições nossa aplicação tem uma grande sobrecarga, sendo necessário 14min para recuperação da aplicação, o que gera uma latência de 10010ms, maior que os outros dois casos em três ordens de grandeza. Logo, há um grande impacto na recuperação pela adição do nosso Serviço no sistema.

Número de requisições acumuladas	Latência
1000(A)	74.69ms
10000(B)	122.76ms
20000(C)	10010ms

Table 1. Latência durante recuperação de requisições acumuladas.

Número de requisições acumuladas	Tempo de detecção (segundos)	Tempo de Recuperação (segundos)
1000(A)	0.991011s	0.990702s
10000(B)	0.998153s	0.975395s
20000(C)	1.013021s	856.67s

Table 2. Tempos de identificação de recuperação e de recuperação final de um contêiner alvo.

5. Conclusão

Através da implementação do Serviço com técnicas de *Event Sourcing*, conseguimos obter um Serviço que cumprisse seu papel de realizar *Checkpoints* e recuperar aplicações que tivessem falhas durante a execução. Entretanto, nossa análise na Seção 4 mostra que nosso Serviço infere uma carga grande no sistema para realizar o *Checkpoint/Restore*. Analisamos pontos de melhorias no sistema que indicam motivos para a perda de performance. Desta forma, embora pudéssemos concluir com um Serviço apto para realizar

Checkpoint/Restore em *Stateful Containers* no Kubernetes, este não pode ser utilizado em sistemas em produção por causa de sua baixa performance, havendo necessidade de melhorias para adequá-lo ao uso real.

6. Referências

References

Kubernetes (2023a). Operator pattern — kubernetes. `https://kubernetes.io/docs/concepts/extend-kubernetes/operator/`. [Online; acessado em 04/11/2023].

Kubernetes (2023b). Overview kubernetes. `https://kubernetes.io/docs/concepts/overview/`. [Online; acessado em 27/03/2023].

Laadan, O. and Hallyn, S. E. (2010). Linux-cr: Transparent application checkpoint-restart in linux. In *Linux Symposium*, volume 159. Citeseer.

Microsoft (2023). Event sourcing pattern - azure architecture center — microsoft learn. `https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing`. [Online; acessado em 25/06/2023].

Müller, R. H., Meinhardt, C., and Mendizabal, O. M. (2022). An architecture proposal for checkpoint/restore on stateful containers. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 267–270.

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

Schmidt, H. (2023). State-preserving container orchestration in failover scenarios. Master's thesis.

Tran, M.-N., Vu, X. T., and Kim, Y. (2022). Proactive stateful fault-tolerant system for kubernetes containerized services. *IEEE Access*, 10:102181–102194.

Vayghan, L. A., Saied, M. A., Toeroe, M., and Khendek, F. (2021). A kubernetes controller for managing the availability of elastic microservice based stateful applications. *Journal of Systems and Software*, 175:110924.

ANEXO B – CÓDIGO FONTE

O código fonte utilizado para elaboração deste Trabalho de Conclusão de Curso encontra-se disponível em um repositório do Git acessível através do endereço https://github.com/GianOrtiz/checkpoint-restore-in-kubernetes.