Gjallarhorn - A Kubernetes Extension

Bryan Keane

October 17, 2022

Contents

List of Figures

List of Tables

1 Plagiarism Declaration

I declare... yada yada yada

2 Acknowledgements

Tanks lads...

3 Introduction

Atomic Kubernetes Resources is an open-source Kubernetes tool which facilitates the development of less problematic multi-operator containerized applications. Implementing this solution will allow developers to configure the custom controller to watch resources of their choice and connect the controller with their team's slack channel to receive interactive notifications when the resource is being updated by the incorrect operator. Users can go as far as setting an atomic owner of resources and blocking other operators from making changes.

3.1 Background

In 2022 I completed an 8-month internship at Red Hat for my 3rd-year work placement. I worked on the Red Hat OpenShift API Management (RHOAM) team. RHOAM utilises multiple OpenShift Operators (automatically managed and configured applications) to provide a comprehensive API solution to its customers. The API management features provided include

- Limiting the number of API requests based on the customer's purchased quota
- reating security policies to manage API access
- Monitoring API health
- An API portal for sign-up and documentation

While working on this team, I had the opportunity to contribute to applications which utilize various technologies like Kubernetes, OpenShift, and Go Operators.

3.2 Motivation

ternship. One of which was with RHOAM's rate-limiting service and is the motivation for Gjallarhorn. Rate-limiting is done with the Marin3r Operator for OpenShift clusters (Red Hat's container orchestration platform). It works by injecting a rate-limiting container into the Pod being rate-limited. The container works by acting as a middleman for requests. It takes in requests and sends them to the destination container if the rate of requests has not reached the limit. The bug I found was that the port to send requests through the rate-limit container was being overwritten by a rogue Operator and set back by the owner Operator. This meant that customers who had RHOAM installed were not getting consistently rate-limited. As a result, an influx of requests could have overloaded Pod CPU and Memory, causing needless stress to the cluster. It took multiple days of debugging to figure out what was going wrong and another week passed before my code changes were merged. This meant that this problem had been occurring on customer clusters for an unknown amount of time before the fix reached customers. After speaking to developers on various Red Hat teams, it became apparent that this problem was not unique to our team and is an issue that all Kubernetes and OpenShift product teams face. Fixing this will not only benefit Red Hat teams but any team working on a Kubernetes-based application.

3.3 Problem Statement

In Kubernetes (K8s), custom resources consist of YAML files which store the desired state of an object (a resource in Kubernetes like Pods and Deployments). An Operator can become a watcher (observer and/or editor of a resource's state) of an object, where it will compare the current state of that object

with the desired state stored in its YAML file. If the states differ, the operator's controller will reconcile the object's state so that they match. This is an integral part of how Kubernetes functions, but it has its flaws.

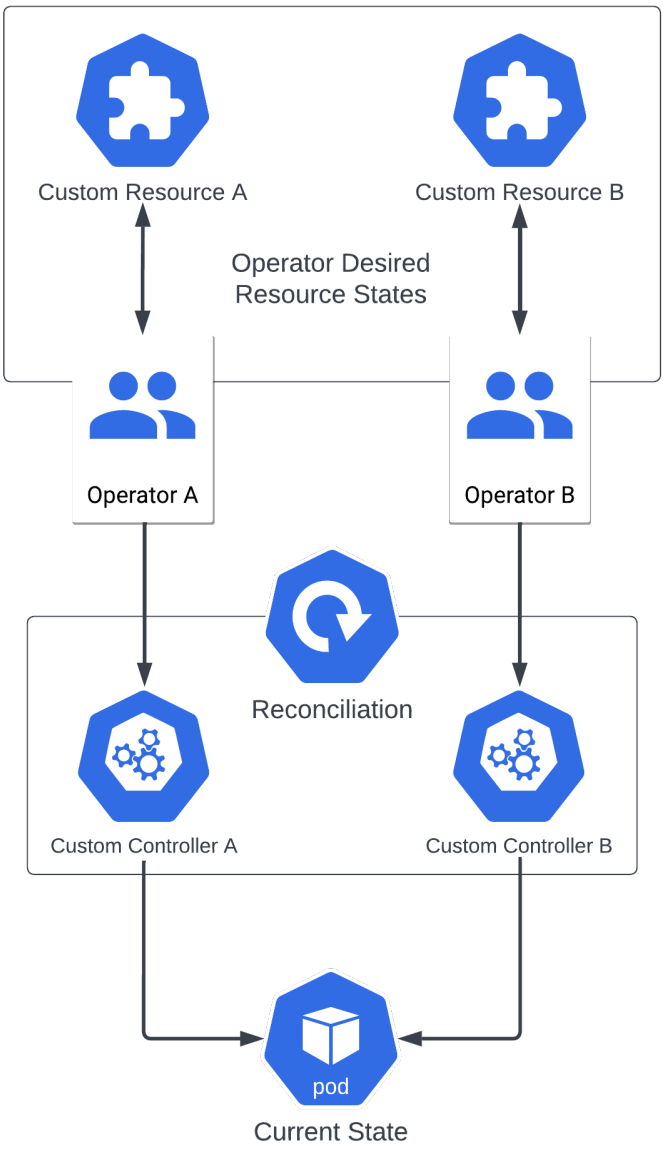

Figure 1: Model of a common problem in multi-operator Kubernetes clusters

Kubernetes natively supports multiple operators to watch the state of a single resource. This can cause problems like the one described in Figure 1 This can cause issues with object states and is the foundation of this project. During my Internship, I was working on a ticket which involved implementing pod count auto-scaling where the number of pods scales up based on the amount of load each pod is experiencing. During this work, I observed some peculiar behaviour. When I forced the pod's load to increase the number of pods increased as expected, but shortly after the pod count returned to its original number. After observing for some time, it became apparent that something was scaling the pods up and down continuously. I discovered it was two OpenShift Operators fighting over the state of the pod. Each operator had a desired state for the pod's deployment, where one operator wanted three replicas of the pod and the other operator wanted two replicas so they underwent a

sort of tug-of-war, fighting over how many pods should be running. This is a common problem where two operators can watch the same resource and have conflicting resource definitions, so the operators' controllers fight to keep the pod in their preferred state. It is a waste of resources and can cause interruptions for the end user.

3.4 Industry Example

TODO: Split up the Problem statement into how the problem works exactly and an example in a context different to the one I encountered

3.5 Aims and Objectives

The solution is to create a custom Kubernetes controller which will monitor resource states. A Kubernetes Operator can create a resource, become its owner, and will set the controller to watch that resource for changes. If the resource is changed by another operator the controller will fire an alert, notify the developer via a slack integration and allow the developer to fix the problem without the need for time-consuming debugging.

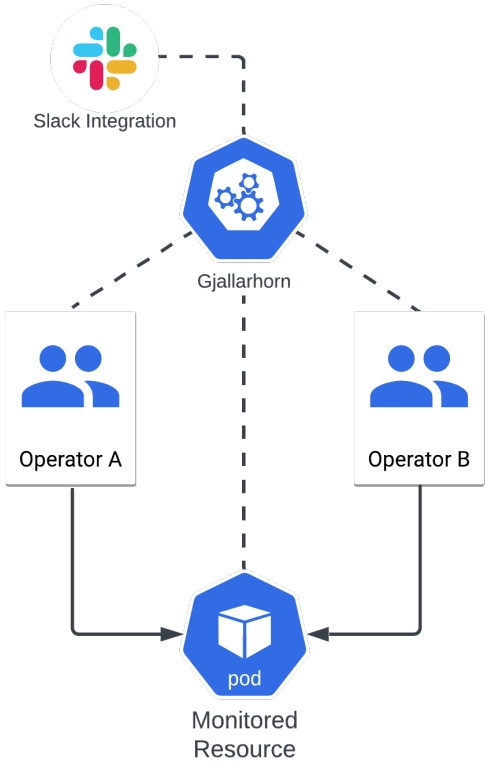

Figure 2: Model of the solution: Gjallarhorn

Figure 2 models the proposed solution where Operator A represents the owner operator of the Monitored Resource, and Operator B represents the rogue operator which is causing the resource to change from the owner's desired state. Once Operator A is installed, it creates the resource and instructs Gjallarhorn to monitor the resource's state. Once Operator B is installed and changes the resource. Gjallarhorn sees this occurring and will send the developers a notification through slack which will remove the need for debugging and allow the developer to get a fix pushed as soon as possible. The stretch goal for this product would be to completely block Operator B from changing the resource so the developers have an interim solution to the bug while they push a fix to the relevant operator, preventing any customer downtime.

3.6 Risks

TODO(): this is a risky project as it has never been done before