Kubernetes operator for XRootD

Basic Information

Name: Shivansh Saini
Major: Computer Science and Engineering
Degree: Bachelor of Technology
Year: Junior
Institute: Indian Institute of Technology (BHU), Varanasi
Github: shivanshs9
Website: **shivanshs9's Blog**
Email Address: shivansh.saini.cse17@iitbhu.ac.in
Phone number: (+91) 9717477205
Resume: https://drive.google.com/file/d/1XsZ-coaklQNZrXsdGmNkp3B-t52t_nmv/view
Timezone: Indian Standard Time (UTC +5:30)

Project

Abstract

This project aims to develop a Kubernetes Operator for XRootD, along with its related documentation, in order to ease and fully automate deployment and management of XRootD clusters. This Operator targets the whole field of existing infrastructure where XRootD can run: development workstations, continuous integration platforms, bare-metal clusters in academic datacenters, and also public cloud platforms.

Motivation

XRootD protocol enables high performance, scalable fault-tolerant access to data repositories of various kinds, including EOS - a disk-based, low-latency storage service with a highly scalable hierarchical namespace - which is responsible for the vast majority of physics and infrastructure data at CERN. LSST also uses XRootD protocol in order to build a system for user query access, called QServ, to store galaxies and stars catalogues.

XRootD is supported on any existing infrastructure, however, deploying and managing it at scale is a non-trivial task. Skilled human operators are needed to manage the XRootD services. These operators have deep knowledge of how the system ought to behave, how to deploy it, and how to react if there are problems.

For ease of managing the XRootD services with minimal human intervention, a Kubernetes Operator could be developed. The XRootD operator would at least enable **Basic Install** and **Seamless Upgrades** features and would be used by the XRootD community in order to ease and scale-up worldwide XRootD clusters management.

Why HEP Software Foundation?

I am only applying for HEP Software Foundation for GSoC'20 and have no plans to contribute to any other organization. The experiments at CERN, such as the Large Hadron Collider, have always intrigued me and I'd take up any chance to contribute even a little to such exciting projects. I think the biggest reason would be that I'd love to contribute to the birthplace of the World Wide Web!
Moreover, my field of interest is DevOps and I like to work with Kubernetes and Golang so this project is perfect for me.

Proposed Deliverables

1. Provide a **XRootD operator**, which will demonstrate how to deploy and manage an XRootD service at scale, using Kubernetes.
2. Implement Kubernetes operator's advanced features like **seamless upgrades**, **full lifecycle**, **deep insights** and **auto-pilot**.
3. Write **E2E tests** for the operator to ensure the operator works as intended in real-world scenarios.
4. Write **documentation** for the CRDs and configurations of the operator.
5. Promote **best practices** in the operator by ensuring high score in the **scorecard**.
6. Explain to the **XRootD community** how to leverage this operator in order to ease and scale-up worldwide XRootD clusters management.

Background Info

The Operator pattern captures how you can write code to automate a task beyond what Kubernetes itself provides. Kubernetes' Controller, the core component for automation, is a control loop that watches the shared state of the cluster through the apiserver and then makes or requests changes moving the current cluster state to the desired state. A Controller is responsible for tracking at least one Kubernetes resource type, providing a true Declarative API for those resources.

Operators are clients of the Kubernetes API that act as controllers for a Custom Resource. These Custom Resources are part of the Kubernetes native application that one wants to package, deploy and manage using the Operator. These types of applications are both

deployed on Kubernetes and managed using Kubernetes APIs. Operator runtime is deployed just as any containerized application outside of the Kubernetes Control Plane.
Operator logic can be implemented in variety of technologies:

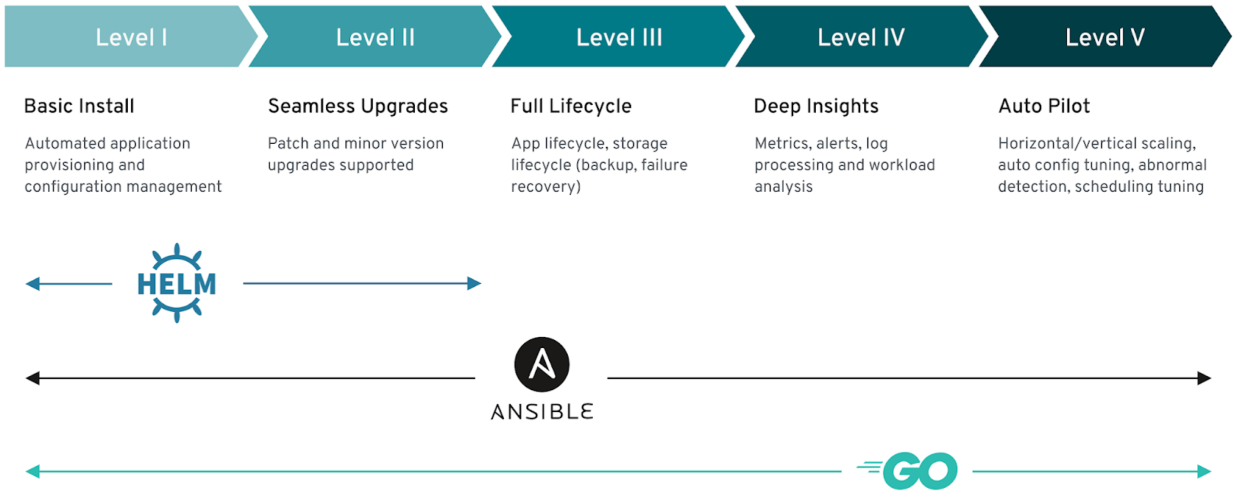
Level I
Level II
Level III
Level IV
Level V
Basic Install
Automated application provisioning and configuration management
Seamless Upgrades
Patch and minor version upgrades supported
Full Lifecycle
App lifecycle, storage lifecycle (backup, failure recovery)
Deep Insights
Metrics, alerts, log processing and workload analysis
Auto Pilot
Horizontal/vertical scaling, auto config tuning, abnormal detection, scheduling tuning
HELM
ANSIBLE
GO

Fig. 1 - Operator Maturity Model

While Helm is sufficient for a “no-code” operator with basic features, using Golang-based Operator-SDK is recommended to get a fully-autopilot operator, implementing all the Kubernetes’ advanced features.

Plan of Action

I’ll review how QServ-operator is implemented to get an idea of XRootD architecture in a working Kubernetes cluster. In any XRootD cluster, there are two types of nodes (from *Fig. 2*):

- **Redirectors -** Redirector coordinates the function of the cluster. It is responsible for querying the server nodes whether they have the file and, if yes, tells the client to connect to the particular server to get the file.
- **Servers/Workers -** Server nodes are responsible to provide the file from its exported directories. They also respond to the metadata query by the redirector for existence of a file.

Focusing on the smallest unit of deployment, each XRootD process will run along with the CMSD process in one pod, with a unified configuration for both manager and server instances. Each cluster can have multiple redirectors and servers with dynamic IPs, so XRootD will consider a **Dynamic DNS** network.

Following the guide, I’ll create the operator using operator-sdk and create a required **Custom Resource Definition** for XRootD which will define a spec template for both worker and redirector replicas. To avoid pre-provisioning storage, the operator spec would take Storage Class name for **Dynamic Provisioning of PersistentVolumes**. I'll create the required StatefulSet object since our pods need to have stable, persistence storage for servers.

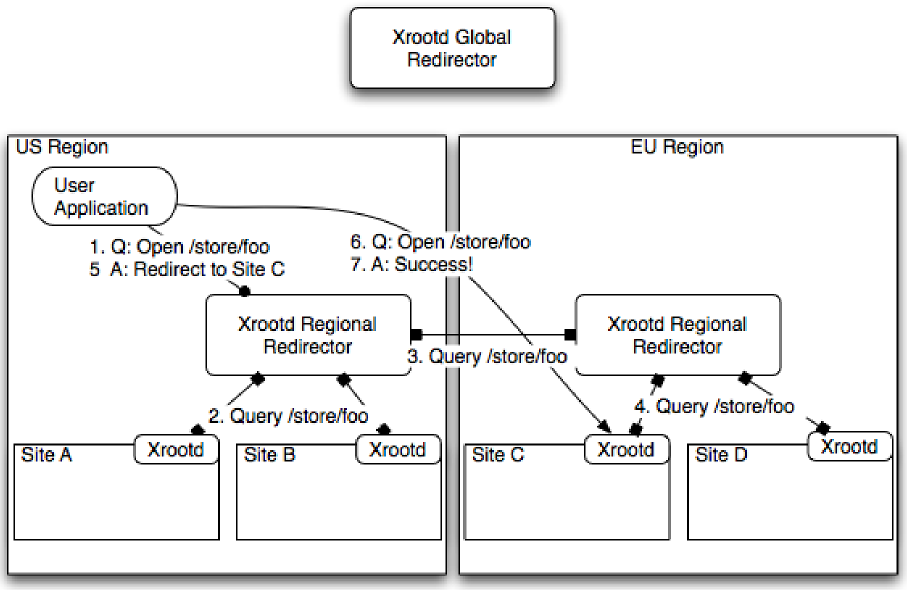

Fig. 2 - XRootD architecture

I'll create a **XRootD controller** to manage the cluster by reconciling the current cluster state to the desired state (described in the spec) using a set of sync operations for each K8s object needed for the working XRootD cluster. If it's possible to mutate the existing object to reach the desired state, then it is done so otherwise it is created or updated by the K8s API.

Once I'm done with the cluster actions for the operator, the operator needs to be deployed on a Kubernetes cluster, which will watch the correct namespaces with a ServiceAccount with required permissions bound to it.

I'll then use the **Operator Lifecycle Manager** to enable a robust deployment model, automating advanced features such as updates, backup and scaling. For that, I'll create a Cluster Service Version (CSV) manifest describing the operator requirements. This will ensure easier installation of the XRootD operator in any cluster, consisting of many other installed operators, and a powerful dependency resolution support.

To get **deeper insights** into service usage and metrics, I'll develop optional components, such as ReportDataSources, for Operator Metering. This can be separately installed via OLM whenever needed by the users of XRootD operator.

Testing is important for any piece of software but it's especially very important for Operators since any failure or unintended behavior by the operator can prove catastrophic for a cluster running in a production environment. However, running functional tests on an actual cluster will be quite expensive, that's why operator-sdk provides a testing framework to run end-to-end (e2e) tests as classic go tests. I'll be using this framework to write e2e tests to ensure the operator works as intended in real-world scenarios.

Finally, I'll be exploring KQueen or Rancher to actually test the XRootD operator in a multi-cloud scenario since XRootD protocol is meant to solve the **Any Data, Anytime, Anywhere (AAA)** requirement to access the remote files regardless if they are present in your region or halfway around the world! So XRootD redirectors across clusters in multiple regions should be able to communicate with each other via the **Cross-region redirection** described in *Fig. 2*. However, it'll be a challenge to implement the **Intercluster Discovery**, which is the capability of automatically configuring DNS servers and load balancers with backends supporting all clusters across many public clouds. Hence, this track will be a long-term one because of its complexity.

Timeline

Community Bonding Period — May 04 - Jun 01

- I'll review the implementation of qserv-operator and experiment with it to better understand how xrootd protocol works.
- I'll be discussing with my mentor to get inputs on what more can I add to the Xrootd resource Specification and Status.
- I'll scaffold a new operator-sdk project and study best practices of operator development in Golang.

Week 1 — Jun 01 - Jun 07

- I'll start implementing the required CRD and the Spec structs with proper validation.
- I'll be studying the godocs of the K8s API, so that I can start writing the controller and the interface needed to sync the cluster from current state to desired state.
- I'll finish the StatefulSet syncers with empty configmaps and no PVCs.

Week 2 & 3 — Jun 8 - Jun 21

- I'll use go templates to finish writing configmap syncers and also finish writing the headless services for the required pods in StatefulSet.
- I'll get the required Dockerfile and docker image for XRootD ready to start testing the operator.

Week 4 — Jun 22 - Jun 28

- I'll study more on Dynamic Provisioning.
- I'll finish implementing PersistentVolumeClaimSpec for the worker pods.

Phase 1 Evaluation **Jun 29 - Jul 03**

- The operator can be successfully deployed on a K8s with working configuration and persistent volumes.

Week 5 Jun 29 - Jul 05

- I'll start writing basic documentation.
- I'll write necessary Makefile commands to ease the development workflow, like deploying, testing and undeploying of the operator.

Week 6 & 7 Jul 06 - Jul 19

- I'll generate the Cluster Service Version manifest needed for Operator Lifecycle Manager.
- I'll test it using the operator-sdk provided bundle validator and by running using the manifest.
- I'll start writing unit tests for the internal functions.

Week 8 Jul 20 - Jul 26

- I'll be studying the godocs of testing-framework.
- I'll structure the code for best testing practices.

Phase 2 Evaluation **Jul 27 - Jul 31**

- Go tests can be run on the codebase.
- Bundle can be built and validated by operator-sdk.
- Documentation can be read for proper use of the operator.

Week 9 Jul 27 - Aug 02

- I'll start writing code for the E2E test scenarios.
- I'll take care to extract out repeated code and reduce the boilerplate code overall.

Week 10 & 11 Aug 03 - Aug 16

- I'll finish writing all the E2E tests and write a basic script to automate running tests by the Github CI.
- For best development practices, I'll also run the operator through scorecard and goreportcard, aiming for A+ code rating.
- If possible, I'll start exploring Rancher too and document ways to use XrootD cluster in a multi-cloud architecture.

Ultimate Week Aug 17 - Aug 23

- For the final week, I'll focus on the reviews given by my mentors and discuss with the Xrootd community on what more things can be added in the operator.
- I'll clean up the documentation and code.
- I'll possibly write a blog post on how to quickly get started using the operator.

Final Evaluation Aug 24 - Aug 31

- The XRootD operator can be successfully deployed and tested on any cluster with OLM installed.
- All unit and E2E tests can be run and they all will pass.
- The XRootD community can review the documentation and use it in their projects.
- The operator will implement all 5 levels described in Operator Maturity Model, making it a complete auto-pilot operator to be used in production clusters.
- The codebase will follow best Golang practices.

Detailed Plan and Implementation

Basic Operator

I'll use operator-sdk CLI to create and scaffold a new operator project:

```
$ operator-sdk new xrootd-operator
```

I'll add a new CRD API with group and version as **xrootd.org/v1alpha1** and kind as **Xrootd**, with the following specification (may possibly change):

```
type XrootdSpec struct {
       Storage             StorageSettings `json:"storage,omitempty"`
       Worker              WorkerSettings `json:"worker,omitempty"`
       Redirector          RedirectorSettings `json:"redirector,omitempty"`
       Config              ConfigSettings `json:"config,omitempty"`
}
type StorageSettings struct {
       StorageClass    string `json:"storageClass,omitempty"`
       StorageCapacity string `json:"storageCapacity,omitempty"`
}
type WorkerSettings struct {
       Replicas            int32 `json:"replicas,omitempty"`
       // Image must have a tag
       // +kubebuilder:validation:Pattern=.+:.+
       Image               string `json:"image,omitempty"`
}
```

```
type RedirectorSettings struct {
	Replicas		int32 `json:"replicas,omitempty"`
	// Image must have a tag
	// +kubebuilder:validation:Pattern=.+:.+
	Image			string `json:"image,omitempty"`
}
type ConfigSettings struct {
}
```

If needed to know the observed cluster state as seen by the controller, I'll also add the Status subresource and enable it using the annotation `// +kubebuilder:subresource:status` on the Xrootd struct.

Once the resources are defined, I'll implement API for them by writing a XrootdController to create the following K8s objects during reconciliation:

1. **Configmaps** for a unified configuration file and an executable shell script, which will be volume mounted under both xrootd and cmsd containers. The configmap source file will be generated from a template and may also be modified by the *.spec.Config*.
   Here's a minimal working configuration:

```
############################
# if: manager node
############################
if named manager
    # Use manager mode
    all.role manager

############################
# else: server nodes
############################
else
    # Use server mode
    all.role server
fi

########################################
# Shared directives (manager and server)
########################################

# This specifies valid virtual paths that can be accessed.
all.export /

# This causes hostname resolution to occur at run-time not configuration time
# This is required by k8s
xrd.network dyndns
```

```
# Hostnames of all XRootD redirectors are provided as managers at configuration
time (compiled using Go templates)
# Assuming cmsd uses the port 2131
set xrootddn = {{.XrootdRedirectorDn}}
{{- range $val := Iterate .XrootdRedirectorReplicas}}
all.manager ${xrootddn}-{{$val}}.${xrootddn}:2131
{{- end}}

# - cmsd redirector runs on port 2131
# - cmsd server does not open server socket
#   but only client connection to cmsd redirector
# - xrootd default port is 1094
if exec cmsd
    xrd.port 2131
fi
```

2. Headless **Services** for Xrootd redirectors and workers to expose them internally for the communication inside the cluster (e.g. for the metadata). For Redirectors, it'll export two ports, one for xrootd process, and other for cmsd process. For Workers, it'll export only the port for xrootd process.

3. **StatefulSet** for both redirectors and workers, in which containers will be defined and volumes mounted for the given number of replicas. For Redirector StatefulSet, respective ports for both xrootd and cmsd containers will be opened with a liveness and readiness probe check on the opened ports. For Worker StatefulSet, only xrootd ports will be opened and data volume will be mounted too, along with a PersistentVolumeClaim for the given Storage Class.

This controller will also watch for create/update/delete events over the Xrootd resource type and the above-mentioned resources created by Xrootd. Kubernetes provides a **level-based API** for the Controller to batch multiple events together and/or skip obsolete events such that the controller makes changes from the actual system state to match the state of the spec *at the time reconcile is called*.

Before deploying the Xrootd cluster, I should first have a built Xrootd docker image which has both working Xrootd and Cmsd softwares. I found an unofficial Xrootd image and it works fine in my trials. I need to modify it a bit to allow exposing the desired ports and configurations before I can use it for the operator.

Finally the operator's **Manager** needs to be built as a docker image and deployed as a container into the desired namespace with a ServiceAccount and RBAC permissions on the appropriate resources. I would be writing configs and shell scripts for the operator installation and deletion procedure.

I would ensure proper validation of CRDs using kubebuilder **validation** annotations. I would also be writing documentation on how to use the operator and basic examples with manifests.

Operator Lifecycle

I'll document steps to install OLM in the desired cluster and then generate and update the CSV manifest for xrootd operator:

```
$ operator-sdk generate csv --csv-version 0.0.1 --update-crds
```

CSV will document all the technical information needed to run the operator, like the RBAC rules and the custom resources it manages or depends on. All the owned/required CRDs and their templates, all the owned/required API services, Metadata and Install spec will be documented too.

Finally, the operator bundle images and bundle metadata will be built to manage the operator via Operator Lifecycle Manager. Each operator release will be listed in the multi-version aggregation file, *xrootd-operator.package.yaml*. I will then validate the bundle using **operator-sdk bundle validate**, which will make sure the labels of built bundle image and *metadata/annotations.yaml* match.

With **Operator Metering** integrated, the operator will have cleared all levels till Level 4 of Operator Maturity Model, as described in *Fig. 1*. I'll be experimenting to use this operator with Prometheus to get basic insights like CPU/IO load and network calls, however for actual application-relevant metrics and insights, the XRootD process needs to trigger events to collect metrics data. So this task is beyond the scope of operator and needs to be dealt in XRootD itself.

Testing

It's vital to write automated tests at different levels:

- **Unit Tests -** To test each component or unit of code individually.
- **End-to-End Tests -** To test the behavior of the entire operator, just as it would behave when running on a live cluster.

Internal code, without dependencies on K8s services and context, will be ideal for Unit Testing. Examples include the **applyTemplate** function for configmap source files, or other utility functions. These tests will run fast and can be written as standard go tests.

However, the most crucial, and yet slow, tests will be E2E tests since they need to run on a cluster. It'll be simpler and efficient to use the **testing framework** provided by operator-sdk.

Following the official guide, I'll first call the main entry function of the testing framework, register the CRDs and setup the test context.

The test specific code will involve testing the following scenarios (may consider adding more later):

- Creating Xrootd object
- Updating Xrootd object
- Deleting Xrootd object
- Deleting Xrootd worker pods
- Deleting Xrootd redirector pods
- Deleting Xrootd configmaps
- Deleting Xrootd services
- Deleting Xrootd StatefulSets
- Adding Persistent Volumes after cluster is ready

OLM is also integrated in operator-sdk CLI to test the operator by running it using the generated CSV. Testing using OLM ensures that everything works fine from the point when a user downloads the operator bundle to the point it's actually up and running on a K8s cluster.

Finally, the operator can be tested against the operator SDK scorecard to make sure all the development best-practices are applied on the operator. The scorecard runs static checks on operator manifests and runtime tests to ensure an operator is using cluster resources correctly, like recording calls to the API server.

Personal Information

Personal Details

I'm Shivansh Saini, an undergraduate at Indian Institute of Technology (BHU), Varanasi. I mainly started with coding 3 years back and have grown to like open source, because of its vibrant and engaging community. I actively use open source software and also love to share my software solutions with the people who may need it.

I'm the Joint Secretary of my college's Club of Programmers to promote open source culture among the students. I've also been working in a food-tech startup, called Checkin, planning the architecture and leading the technical development.

For my technical skills, I've had practical experience in backend REST development, JAMstack frontend development, Android development and **Cloud DevOps** in container orchestration. I've worked with Kubernetes and have deployed microservices behind Ingress on a GCP cluster. I've also explored a bit of **Mainframe DevOps** via IBM tutorials and contributed some code to run on IBM s390x.

I’m proficient in C, C++, Python, Java, Kotlin, Javascript and Shell scripting. I’ve basic knowledge of **Golang**, but insufficient experience in it. I am an eager learner and ready to add more skills to my toolbox.
Most of my projects are polished to work in a production environment in the hands of regular users. I am a fan of semantic versioning and am a CI diplomat, such that I don't let any production code pass without a CI/CD pipeline.

I am a power user at heart and love to hack any technology I get my hands on, until I have used that in something of my own. I believe in not reinventing the wheel, but rather collecting tools in my portable toolbox.
Unfortunately, no matter what I do, there's still lots of space for new tools in my toolbox! I use EndeavorOS (Arch-based Linux), with zsh as my favoured shell and vim as my go-to editor.

Communication

I’m flexible with my schedule and have inculcated the habit of working at night, so time zone difference shouldn’t be an issue.
I can work full-time on weekdays and am usually available between 3 PM IST to 2 AM IST. On weekends, I would love to spend time communicating with the team to learn from them, while working on whatever issues occur at that time.

Due to the COVID-19 situation, our college activities aren’t finalized yet but I’ll keep my mentor updated with any new happenings or if there’s a conflict of this project with any changes in my academic schedule.
I may be travelling for a few days, and I'll inform those plans beforehand to my mentor. I’ll also responsibly keep my mentor updated in case of any emergency that occurs with suitable details.

Post GSoC

If there are things left unimplemented, I’ll try to complete them post-GSoC and will keep contributing to maintain the XrootD operator for its community. Implementing this operator in the Multi-cloud architecture will be a long-term project so I’ll be looking out for its solutions even after the GSoC is over.

Regardless of GSoC, I would love to engage in discussions with the XrootD community to get exposure over technical challenges at the scale of Peta-bytes and brainstorm over new ideas. If HSF permits, I would also love to showcase the XRootD operator in the next Kubecon.

Contributions

Prerequisite task - Added support for spec.xrootd.replicas