Rook Deep Dive

Jared Watts
Rook Senior Maintainer
Upbound Founding Engineer

https://rook.io/
https://github.com/rook/rook
Agenda

- Quick introduction to Rook (again)
- Deep dive: Ceph orchestration
- Deep dive: Storage provider integration for Minio
- Demo (if time allows)
- Questions
What is Rook?

- Cloud-Native Storage Orchestrator
- Extends Kubernetes with custom types and controllers
- Automates deployment, bootstrapping, configuration, provisioning, scaling, upgrading, migration, disaster recovery, monitoring, and resource management
- Framework for many storage providers and solutions
- Open Source (Apache 2.0)
- Hosted by the Cloud-Native Computing Foundation (CNCF)
Storage Challenges

- Reliance on external storage
  - Requires these services to be accessible
  - Deployment burden
- Reliance on cloud provider managed services
  - Vendor lock-in
- Day 2 operations - who is managing the storage?
Possible Solutions

- Deploy storage systems INTO the cluster
- Portable abstractions for all storage needs
  - Database, message queue, cache, object store, etc.
- Power of choice: cost, features, resiliency, compliance
- Automated management by smart software
Custom Resource Definitions (CRDs)

- Teaches Kubernetes about new first-class objects
- Custom Resource Definition (CRDs) are arbitrary types that extend the Kubernetes API
  - look just like any other built-in object (e.g. Pod)
  - Enabled native kubectl experience
- A means for user to describe their desired state
Rook Operators

- Implements the **Operator Pattern** for storage solutions
- User defines *desired state* for the storage cluster
- The Operator runs reconciliation loops
  - **Observe** - Watches for changes in desired state and cluster
  - **Analyze** - Determine differences between desired and actual
  - **Act** - Applies changes to the cluster to drive it towards desired
Rook Framework for Storage Solutions

- Rook is more than just a collection of Operators and CRDs
- **Framework** for storage providers to integrate their solutions into cloud-native environments
  - Storage resource normalization
  - Operator patterns/plumbing
  - Common policies, specs, logic
  - Testing effort
- Ceph, CockroachDB, Minio, NFS, Cassandra, Nexenta, and more...
Ceph Deep Dive
General Orchestration Approach

- Operator runs ceph commands to initialize and bootstrap cluster (cephx auth, crush map, etc.)
- Pod template spec is generated from Cluster CRD config
  - Ceph daemon command line arguments
  - Environment variables injected
  - `ceph.conf` generated and written to pod filesystem
- Operator creates a Kubernetes controller primitive (**Deployment**) to manage lifecycle of each Ceph pod
- Health of cluster and components is monitored over time and corrective actions taken
Orchestration of Monitors

- Operator creates a pod for each mon specified in the CRD
- **Deployment** object wraps each mon pod for reliable lifecycle management
- Placement of mons ensures node isolation (1 mon per node)
- **Service** object is created per mon to establish a consistent IP address - important for quorum and mon map

```yaml
apiVersion: ceph.rook.io/v1beta1
kind: Cluster
metadata:
  name: my-cluster
spec:
  mon:
    count: 3
    multiPerNode: false
```
Monitors: Surviving Pod Restarts

- Mon persistent state must survive restarts (pod restart, node reboot, power failure, etc.)
- Mon state is stored in a HostPath mounted by the mon pod
  - user configurable via dataDirHostPath in Cluster CRD
- After a power outage, mon pods start and load state from the persisted data
  - Once mons form quorum, the cluster is healthy again
- If a mon loses its persisted data, it will heal itself after a restart
Monitors: Maintaining quorum

- Mon quorum is critical to cluster health
- Operator regularly checks on mon quorum
- If a mon falls out of quorum for too long, the operator takes action to replace the failed mon
  - A new mon is started (new Deployment and Service IP)
  - Wait for new mon to join quorum
  - Delete the failed mon Deployment and Service
  - Remove the failed mon from the mon map
Orchestration of OSDs

- Operator starts OSDs according to config from Cluster CRD
- “Discover” DaemonSet identifies available devices on all nodes in the cluster
- Operator schedules a BatchJob on each node to initialize/provision its OSDs
- One Deployment (ReplicaSet/Pod) is created for each OSD
  - OSDs run independently
- Horizontal scaling: Operator automatically add OSDs to new nodes and devices
OSDs: Device selection

- **Mode 1**: Automatically consume available devices on all nodes
  - Safety checks ensure devices aren’t already in use

- **Mode 2**: Only consume the devices specified in the CRD
  - Full admin control for which devices will run OSDs

```yaml
spec:
  storage:
    useAllNodes: true
    useAllDevices: true

spec:
  storage:
    useAllNodes: false
    nodes:
    - name: "node1"
      devices:
      - name: "sdb"
      - name: "node2"
    deviceFilter: "^sd."
```
Orchestration of RGW

- Creates an object gateway according to settings in the **ObjectStore** CRD
- Required Ceph pools are created
  - 5 metadata pools
  - 1 data pool (can be erasure coded)
- RGW pods are started via **Deployment** for HA/reliability
- **Service** created for client access and load balancing

```yaml
apiVersion: ceph.rook.io/v1beta1
kind: ObjectStore
metadata:
  name: my-store
spec:
  metadataPool:
    replicated: true
    size: 3
  dataPool:
    erasureCoded: true
    dataChunks: 2
    codingChunks: 2
  gateway:
    port: 80
    instances: 1
```
Orchestration of CephFS

- Creates a shared file system according to settings in the **Filesystem** CRD
- Required Ceph pools are created
  - 1 metadata pool
  - 1 data pool (can be erasure coded)
- MDS pods are started via **Deployment** for HA/reliability
  - Standby MDS pods for quick failover

```yaml
apiVersion: ceph.rook.io/v1beta1
kind: Filesystem
metadata:
  name: my-filesystem
spec:
  metadataPool:
    replicated:
      size: 3
  dataPools:
    - replicated:
        size: 3
  metadataServer:
    activeCount: 1
    activeStandby: true
```
Rook Agent

- Dynamically attaches/mounts Ceph storage for pod consumption
- Runs as **DaemonSet** on all schedulable nodes in cluster
- Block: `rbd map`
- File: `mount -t ceph`
- Fencing and locking for ReadWriteOnce
- Detach and reattach if pod scheduled onto another node
- Currently a Kubernetes FlexVolume, will be replaced by CSI driver in the near future (work ongoing)
Automated Stateful Upgrades

- Partially implemented in 0.8, more support coming in 0.9
- Operator controls and manages software upgrade flow
- Upgrade is simply applying/reconciling desired state
- Leverages built-in functionality of K8s resources like Deployments to update components in a rolling fashion
- Health checks to ensure cluster health is maintained
- Separation of Rook and Ceph versioning to isolate impact
- Special upgrade and migration steps between major versions of Ceph (Mimic -> Nautilus) will be implemented as necessary
Developer Deep Dive:
Storage Provider Integration
Minio Operator
Operator Frameworks

**Current**: Register CRDs, watch events and invoke handler functions

- Rook operator-kit: [https://github.com/rook/operator-kit](https://github.com/rook/operator-kit)

**Future**: Auto-generate APIs, CRDs, controllers, reconciliation, boilerplate code, unit tests, deployment, etc.

- Kubebuilder: [https://github.com/kubernetes-sigs/kubebuilder](https://github.com/kubernetes-sigs/kubebuilder)
Minio ObjectStore CRD

```yaml
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: objectstores.minio.rook.io
spec:
  group: minio.rook.io
  names:
    kind: ObjectStore
    listKind: ObjectStoreList
    plural: objectstores
    singular: objectstore
  scope: Namespaced
  version: v1alpha1
```
Minio ObjectStore Custom Object

```yaml
apiVersion: minio.rook.io/v1alpha1
kind: ObjectStore
metadata:
  name: my-store
  namespace: rook-minio
spec:
  scope:
    nodeCount: 4
```
Using the Object Store CRD

```
$ kubectl create -f object-store-crd.yaml
customresourcedefinition "objectstores.minio.rook.io" created

$ kubectl get crds
NAME          AGE
objectstores.minio.rook.io  9s

$ kubectl create -f object-store.yaml
objectstore "my-store" created

$ kubectl get objectstores
NAME  AGE
my-store  19s
```
Revisiting the ObjectStore

- Rook knows how to work with common information in storage object specs (networking, node counts, etc.)
- Only the credentials are Minio-specific
- We can use this information to deploy a Minio cluster

```
apiVersion: minio.rook.io/v1alpha1
kind: ObjectStore
metadata:
  name: my-store
  namespace: rook-minio
spec:
  scope:
    nodeCount: 4
  resources:
    - name: objectserver
      limits:
        cpu: "500m"
        memory: "2Gi"
  network:
    hostNetwork: false
    port: 9000
  credentials:
    accessKey: "TEMP_DEMO_ACCESS_KEY"
    secretKey: "TEMP_DEMO_SECRET_KEY"
```
Minio Operator

- We specify the container that the Minio operator will reside in.
- Args are provided to inform the Rook binary that it needs to operate on Minio.
- We include the CRD in the same file as this operator description.
Minio Operator Container Image

- Contains both Minio server/tools and Rook libraries
- Optimized Docker build to collapse layers and minify image
- Base image is Alpine Linux

```
FROM minio/minio:RELEASE.2018-04-19T22-54-58Z
COPY rook /usr/local/bin/
ENTRYPOINT ["/usr/local/bin/rook"]
CMD ["""]
```
Minio ObjectStore Golang Types

- ObjectStoreSpec struct defines the config properties exposed to the user in `object-store.yaml`
- Notice the spec takes advantage of the common types/specs from the Rook framework
Minio Operator Watching for Events

- We create a new watcher to watch for **add**, **update**, or **delete** events.
- Event handler functions are passed to the Rook operator-kit.
Watching with Informers

- We use an Informer to watch for k8s events, which prevents excessive polling on the API server.
- The informer keeps a cache of objects to limit GETs.

```go
func (w *ResourceWatcher) Watch(objType runtime.Object, done <-chan struct{}) error {
    source := cache.NewListWatchFromClient(
        w.client,
        w.resource.Plural,
        w.namespace,
        fields.Everything())
    controller := cache.NewInformer(
        source,
        // The object type.
        objType,
        // resyncPeriod
        // Every resyncPeriod, all resources in the cache will retrigger events.
        // Set to 0 to disable the resync.
        0,
        // Your custom resource event handlers.
        w.resourceEventHandler
    )
    go controller.Run(done)
    <-done
    return nil
}
```
The onAdd handler implementation uses the K8s API to create services, stateful sets, etc. We programmatically follow the deployment procedure for the Minio cluster.
ObjectStore Update Handler

```go
func (c *MinioController) onUpdate(oldObj, newObj interface{}) {
    oldStore := oldObj.(*miniov1alpha1.ObjectStore).DeepCopy()
    newStore := newObj.(*miniov1alpha1.ObjectStore).DeepCopy()

    // Analyze differences between old cluster and new cluster,
    // perform operations to make actual state match the desired state
}
```
How to get involved?

- Contribute to Rook
  - [https://github.com/rook/rook](https://github.com/rook/rook)
  - [https://rook.io/](https://rook.io/)
- Slack - [https://rook-io.slack.com/](https://rook-io.slack.com/)
  - #conferences now for Kubecon China
- Twitter - @rook_io
- Forums - [https://groups.google.com/forum/#!forum/rook-dev](https://groups.google.com/forum/#!forum/rook-dev)
- Community Meetings
Questions?

https://github.com/rook/rook

https://rook.io/
Thank you!

https://github.com/rook/rook

https://rook.io/