Kubernetes Volumes

Wayne Chang (wayne@wyc.io)

May 24th, 2017

Table of Contents

These slides *attempt* to cover:

- Volume Terminology and Life Cycles
- Volume Plugin Codewalk
- Future of Volumes

https://github.com/wyc/k8s-volumes

glhf!

Terminology, Demystified

- Volume
- PersistentVolume
- PersistentVolumeClaim
- StatefulSets (previously, PetSets pre-1.5)

What Are `Volumes`?

`Volumes` are storage facilities tied to the lifetime of a Pod. They die when the Pod dies. They are specified in a Pod definition. Think Docker volumes.

Volume example

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-pd
      name: test-volume
  volumes:
  - name: test-volume
    hostPath:
      # directory location on host
      path: /data
```

What are `PersistentVolumes`?

`PersistentVolumes` are cluster-level networked storage resources that exist on their own. They can be provisioned by an admin or on the fly.

They can be in one of many states:

- Provisioning
- Binding
- Using
- Releasing
- Reclaiming

PersistentVolume Example

```yaml
apiVersion: "v1"
kind: "PersistentVolume"
metadata:
  name: "gce-disk-pv-1"
  annotations:
    volume.beta.kubernetes.io/mount-options: "discard"
spec:
  capacity:
    storage: "50Gi"
  accessModes:
    - "ReadWriteOnce"
  gcePersistentDisk:
    fsType: "ext4"
    pdName: "gce-disk-1
```

What are `PersistentVolumeClaims`?

`PersistentVolumeClaims` are cluster-level metadata that exist on their own. They are matched to a specific `PersistentVolume`, and then bound exclusively to it.

The claims are then used by multiple `Pods` as `Volume`.

PersistentVolumeClaim Example

```
apiVersion: "v1"
kind: "PersistentVolumeClaim"
metadata:
  name: "gce-disk-pvc-1"
spec:
  accessModes:
    - "ReadWriteOnce"
  resources:
    requests:
      storage: "50Gi"
  storageClassName: "slow" -- pd-standard/pd-ssd
  selector:
    matchLabels:
      release: "stable"
    matchExpressions:
      - {key: environment, operator: In, values: [dev]}
```

PersistentVolumeClaim as Volume Example

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: frontend
      image: dockerfile/nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: www
  volumes:
    - name: www
      persistentVolumeClaim:
        claimName: gce-disk-pvc-1
```

Killing `PersistentVolumeClaims`

```
$ kubectl delete pvc -l app=myapp
```

From the docs:

> *You should never assume ability to access a volume after claim deletion.*

Enter reclaim policies: - recycle - retain - delete

With retain, you are expected to either delete and recreate the PV or clear the status block with the API manually.

What are `StatefulSets`?

`StatefulSets` give the following guarantees:

- Ordered deployment, scaling, deletion, and termination
  - Who's on first? (Master Node, obvs.)
- Stable network identifiers (e.g., web-0, web-1) and **persistent storage** (?!?!)

What?! Stable persistent storage?

Yes. It's true.

```
// types.go:84
// The StatefulSet controller is responsible for mapping
// network identities to claims in a way that maintains
// the identity of a pod.

// stateful_set_utils.go:94
func getPersistentVolumeClaimName(
    set *apps.StatefulSet,
    claim *v1.PersistentVolumeClaim,
    ordinal int
) string {
    return fmt.Sprintf(
        "%s-%s-%d", claim.Name, set.Name, ordinal)
}
```

Persistent storage it is.

Use the StatefulSet labels. Not mentioned in docs yet?

```
// stateful_set_utils.go:140
func getPersistentVolumeClaims(
    set *apps.StatefulSet, pod *v1.Pod
) map[string]v1.PersistentVolumeClaim {
    // ...
    for i := range templates {
        claim := templates[i]
        claim.Name = getPersistentVolumeClaimName(
                        set, &claim, ordinal)
        claim.Namespace = set.Namespace
        // USE ME
        claim.Labels = set.Spec.Selector.MatchLabels
        claims[templates[i].Name] = claim
    }
    return claims
}
```

Some current StatefulSets drawbacks

Single point of failure

From 2017 article by @beekhof (RedHat), see references.

> *Writers are told to connect to pod 0 explicitly rather than use the mysql-read service.*

From the docs:

> *As each Pod is created, it gets a matching DNS subdomain, taking the form: (podname).(governing service domain), where the governing service is defined by the serviceName field on the StatefulSet.*

This spells downtime, but could be fixed if there was an abstraction that would let us specify a "subservice" that always pointed to the writing pod.

Some current StatefulSets drawbacks

Scaling woes

> *Pod N cannot be recovered, created or destroyed until all pods 0 to N-1 are active and healthy.*

You can't scale up if something's wrong with an existing pod. This is unfortunate for HA.

Volume Plugins

Volume plugins allow us to abstract various local, LAN, and cloud storage providers to use as `Volumes` and `PersistentVolumes`. Some examples:

- AWS Elastic Block Storage
- GCE Persistent Disk
- NFS
- GlusterFS
- iSCSI
- Empty Dir
- Git Repo

Flex Volumes

- Custom driver executable installed on worker and master nodes, by you, of course.
- Outputs JSON.

```
# Initializes the driver
$ myexe init
# Cluster-wide unique name
$ myexe getvolumename <json options>
# Attach to node
$ myexe attach <json options> <node name>
# Detach from node
$ myexe detach <json options> <node name>
# ... others, including mount/unmount, waitforattach
```

Plugin Implementation

Two main “styles”:

- Cloud-style: RESTful Interfaces and OS play
  - AWS EBS
  - GCE PD

```
import "k8s.io/kubernetes/pkg/cloudprovider/providers/aws"
devicePath, err := attacher.awsVolumes.AttachDisk(
    volumeID, nodeName, readOnly)
```

- LAN-style: Farmed out to executables (less-meta FlexVolumes!)
  - GlusterFS
  - NFS

```
cmdOut, err := b.exe.Command(
    linuxGlusterMountBinary, "-V")
```

Plugin Interfaces

Who uses Volume Plugins? Kublets and sometimes the control plane.

Found in `pkg/volume/volume.go` and `pkg/volume/plugin.go`.

- VolumePlugin: Main interface, `mount` and `unmount`
  - PersistentVolumePlugin: Can persist data, support access modes
    - ReadWriteOnce
    - ReadOnlyMany
    - ReadWriteMany
  - RecyclableVolumePlugin: Can reclaim resource automatically
  - DeletableVolumePlugin: Can be deleted from the cluster
  - ProvisionableVolumePlugin: Can be provisioned to the cluster
  - AttachableVolumePlugin: Can attach and detach to worker nodes

Plugin Interface Example

Notice how these don't actually do any attaching or detaching. Instead, they interfaces that actually do the work.

```go
type AttachableVolumePlugin interface {
    VolumePlugin
    NewAttacher() (Attacher, error)
    NewDetacher() (Detacher, error)
    GetDeviceMountRefs(deviceMountPath string
        ) ([]string, error)
}
```

Plugin Interfaces

The following are interfaces that get returned from plugins.

- MetricsProvider
    - Volume
        - Mounter
        - Unmounter
        - Deleter
- Provisioner
- Attacher
- Detacher
- BulkVolumeVerifier

Plugin Interface Example

```go
// Detacher can detach a volume from a node.
type Detacher interface {
    Detach(deviceName string,
        nodeName types.NodeName) error
    UnmountDevice(deviceMountPath string) error
}
```

GCE PD Implementation Highlights

This is the “reference implementation,” found in `pkg/volume/gce_pd/`

```go
// struct def
type gcePersistentDiskAttacher struct {
    host     volume.VolumeHost
    gceDisks gce.Disks
}

// attaching
if err := attacher.gceDisks.AttachDisk(
           pdName, nodeName, readOnly); err != nil {
    glog.Errorf("Error attaching PD %q to node %q: %+v",
        pdName, nodeName, err)
    return "", err
}
```

GCE PD Implementation Highlights

After it has been attached + mounted to the host, bind mount is used to the pod.

With a bind mount, instead of mounting a device to a path you are mounting one path to another path.

```
// Perform a bind mount to the full path
// to allow duplicate mounts of the same PD.
options := []string{"bind"}
if b.readOnly {
    options = append(options, "ro")
}
// globalPDPath is the mount for the worker node
err = b.mounter.Mount(globalPDPath, dir, "", options)
```

GCE PD Implementation Highlights

Global mount path on the worker node:

```
func makeGlobalPDName(host volume.VolumeHost,
    devName string) string {
    return path.Join(host.GetPluginDir(
            gcePersistentDiskPluginName),
        mount.MountsInGlobalPDPath, devName)
}
```

AWS EBS Implementation Highlights

REST helper packages found in `pkg/cloudprovider/providers/` and rely on available Go libraries for those services.

This example found in `pkg/volume/aws_ebs/`.

```
// struct def
type awsElasticBlockStoreAttacher struct {
    host       volume.VolumeHost
    awsVolumes aws.Volumes
}

// attaching
devicePath, err := attacher.awsVolumes.AttachDisk(
        volumeID, nodeName, readOnly)
```

AWS EBS Implementation Highlights

Bind mount, again.

```go
// copy-and-pasted from GCE PD
options := []string{"bind"}
if b.readOnly {
    options = append(options, "ro")
}
err = b.mounter.Mount(globalPDPath, dir, "", options)
```

AWS EBS Implementation Highlights

Hack necessary on AWS to get correct device names, `pkg/cloudprovider/providers/aws/aws.go:1304`:

```
// Find the next unused device name
deviceAllocator := c.deviceAllocators[i.nodeName]
if deviceAllocator == nil {
    // we want device names with two significant
    // characters, starting with /dev/xvdbb
    // the allowed range is /dev/xvd[b-c][a-z]
    // http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/
    deviceAllocator = NewDeviceAllocator()
    c.deviceAllocators[i.nodeName] = deviceAllocator
}
// We need to lock deviceAllocator to prevent
// possible race with Deprioritize function
deviceAllocator.Lock()
defer deviceAllocator.Unlock()
```

AWS EBS Implementation Highlights

pkg/cloudprovider/providers/aws/device_allocator.go:77

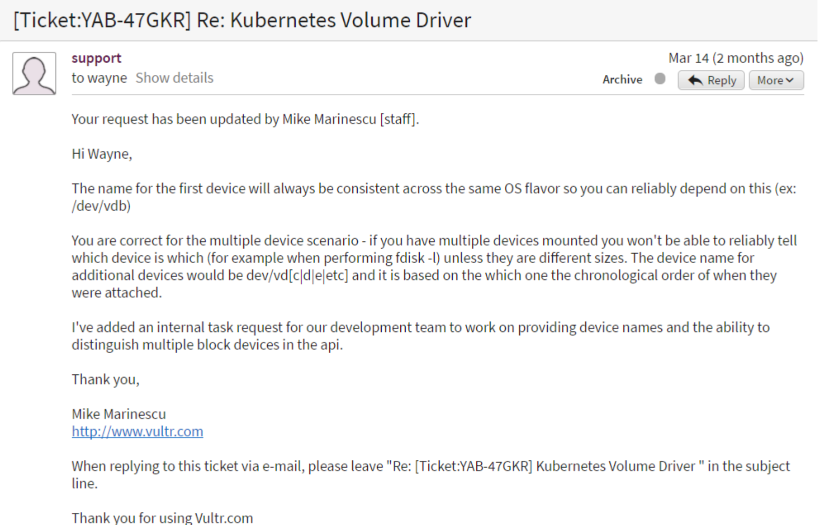
```go
// Allocates device names according to scheme ba..bz, ca..
// it moves along the ring and always picks next device un
// device list is exhausted.
func NewDeviceAllocator() DeviceAllocator {
    possibleDevices := make(map[mountDevice]int)
    for _, firstChar := range []rune{'b', 'c'} {
        for i := 'a'; i <= 'z'; i++ {
            dev := mountDevice([]rune{firstChar, i})
            possibleDevices[dev] = 0
        }
    }
    return &deviceAllocator{
        possibleDevices: possibleDevices,
        counter:         0,
    }
}
```

AWS EBS Implementation Highlights

GCE PD has predictable block device paths such as `/dev/gce-disk-1` and therefore doesn't need to do this.

AWS EBS Implementation Highlights

Some cloud providers don't support any kind of predictable naming, but others will have this order guarantee(?).



[Ticket:YAB-47GKR] Re: Kubernetes Volume Driver

support
to wayne Show details

Mar 14 (2 months ago)

Your request has been updated by Mike Marinescu [staff].

Hi Wayne,

The name for the first device will always be consistent across the same OS flavor so you can reliably depend on this (ex: /dev/vdb)

You are correct for the multiple device scenario - if you have multiple devices mounted you won't be able to reliably tell which device is which (for example when performing fdisk -l) unless they are different sizes. The device name for additional devices would be dev/vd[c|d|e|etc] and it is based on the which one the chronological order of when they were attached.

I've added an internal task request for our development team to work on providing device names and the ability to distinguish multiple block devices in the api.

Thank you,

Mike Marinescu
http://www.vultr.com

When replying to this ticket via e-mail, please leave "Re: [Ticket:YAB-47GKR] Kubernetes Volume Driver " in the subject line.

Thank you for using Vultr.com

GlusterFS Implementation Highlights

Thin exec wrapper that expects the executables on all worker nodes. Found in pkg/volume/glusterfs/

```
cmdOut, err := b.exe.Command(
    linuxGlusterMountBinary, "-V").CombinedOutput()
if err != nil {
    return fmt.Errorf(
        "Failed to get binary %s version",
        linuxGlusterMountBinary)
}
```

GlusterFS Implementation Highlights

The struct def:

```go
type glusterfsMounter struct {
    *glusterfs
    hosts        *v1.Endpoints
    path         string
    readOnly     bool
    exe          exec.Interface
    mountOptions []string
}
```

GlusterFS Implementation Highlights

The mounting

```
errs = b.mounter.Mount(ip+":"+b.path, dir,
        "glusterfs", mountOptions)
if errs == nil {
    glog.Infof("glusterfs: successfully mounted %s", dir)
    return nil
}
```

External Provisioners

> *A StorageClass provides a way for administrators to describe the “classes” of storage they offer.*

```yaml
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: slow
provisioner: kubernetes.io/aws-ebs
parameters:
  type: io1
  zone: us-east-1d
  iopsPerGB: "10"
```

You can choose your own provisioner! Seen External Storage reference.

The Future: Container Storage Interface

Motivation

- We know FlexVolume is a hack
- Adding a new drivers per cloud provider is unsustainable
- Then you have to re-implement all drivers for every new container orchestrator

Current Status

- Inspired by standardization via Container Networking Interface
- Based on gRPC endpoints implementing CSI services, aka defined in protobuf
- Wants to specify lifecycle and local storage
- Not on the plate: grading storage, auth/authz, POSIX

References

- Flex Volumes
- Containerizing Databases with Kubernetes and Stateful Sets
- External Storage
- Kubernetes Volume Guide
- Kubernetes Persistent Volume Guide
- Container Storage Interface Announcement
- Google Doc: Container Storage Interface (CSI)
- Stateful Containers on Kubernetes

EOF