Kubernetes Workshop

Kubernetes Training

6: Managing Persistent State

Mass Storage

Why is there mass storage?

- All systems eventually need some capacity to manage data
- A storage system should not add stateful dependencies to nodes

Mass storage

SANs, NAS and filesystems like NFS have been around for a long time.

Same for public clouds, with

- elastic local volumes
- services dedicated to storing data: like AWS S3
- managed database systems like AWS RDS.

But what if we need access to a file system shared among machines?

Horizontally scaling application stacks, that are in some way heavily reliant on local filesystem access, are always difficult.

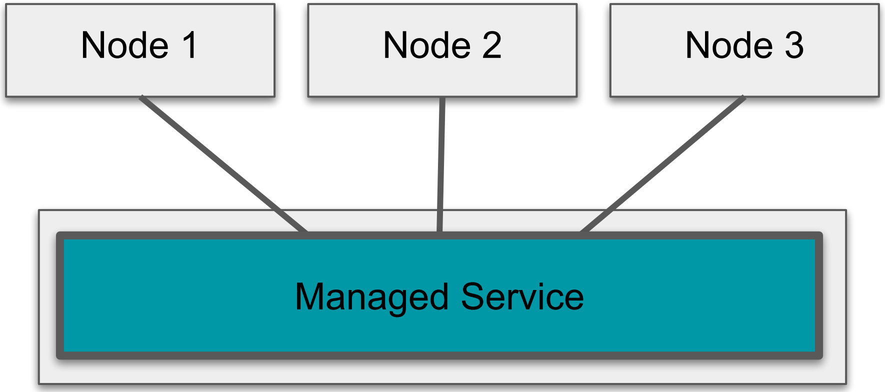
Strategies for Stateful Applications

Strategies for Stateful Applications

- Local disk (?)
- External Services for Storage, f.e.:
  - S3
  - NFS
  - SAN
- Application layer state replication
  (f.e. DB sharding, master/slave replication)
- Cluster File Systems (Container Native Storage)
- Managed DBMS

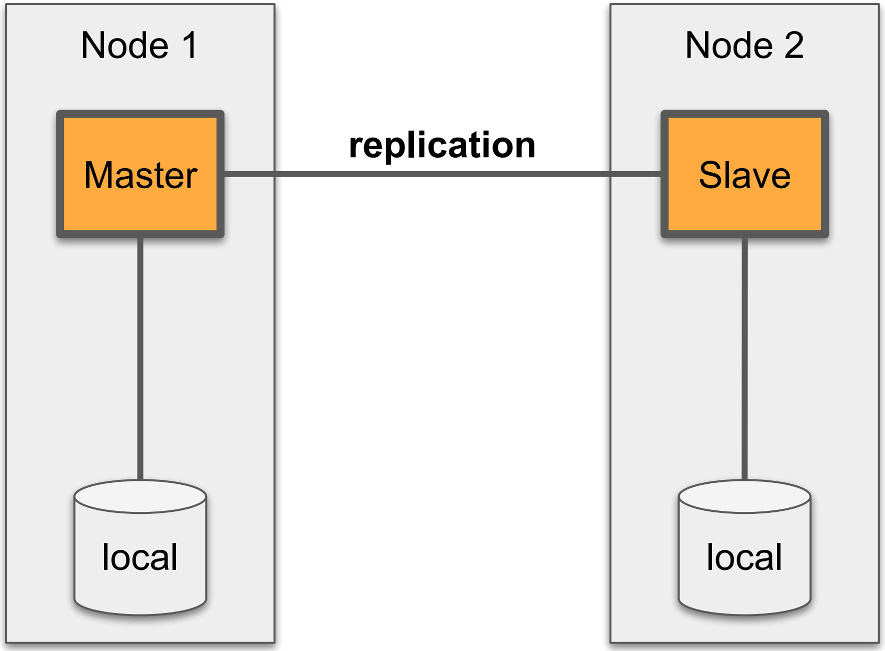
… or combinations

Managed Service (zB RDBMS)



- F.e. AWS RDS

Application Layer



- Tool based replication (f.e. Master/Slave)
- Simply writes to local disk

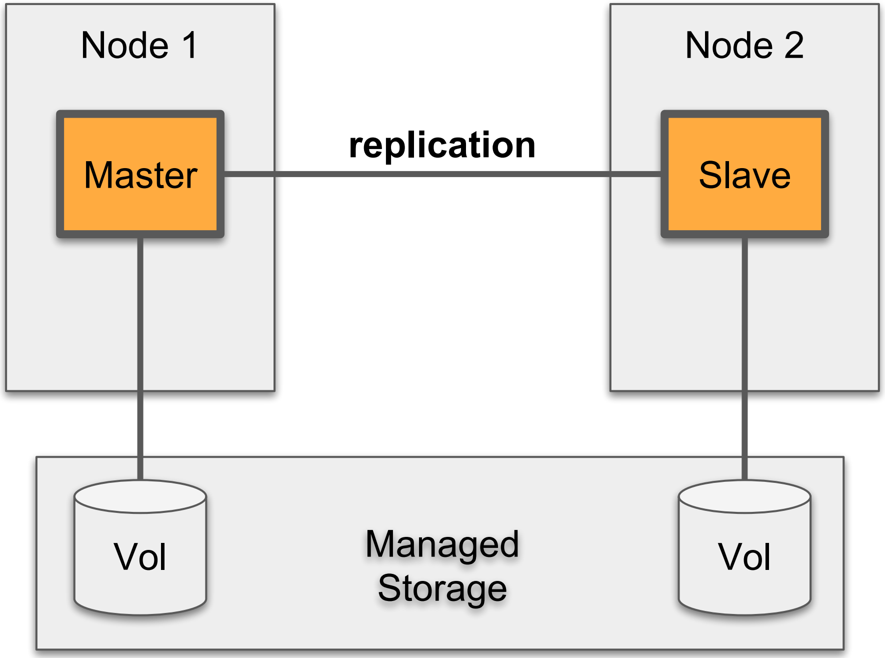
Fixes app to node
One to one relationship (app - node)

Application Layer (local Filesystem)

- **emptyDir**
  - per-pod storage strategy, sandboxed, COW
  - Initiated during the creation of a pod
  - Visible to all Containers in a Pod
  - Survives (only) individual container crashes and restarts
  - for storing intermediary data, sharing configuration settings and/or data

- **hostPath**
  - mount a file or directory from the host node’s filesystem directly
  - directories created on the underlying hosts are only writable by root

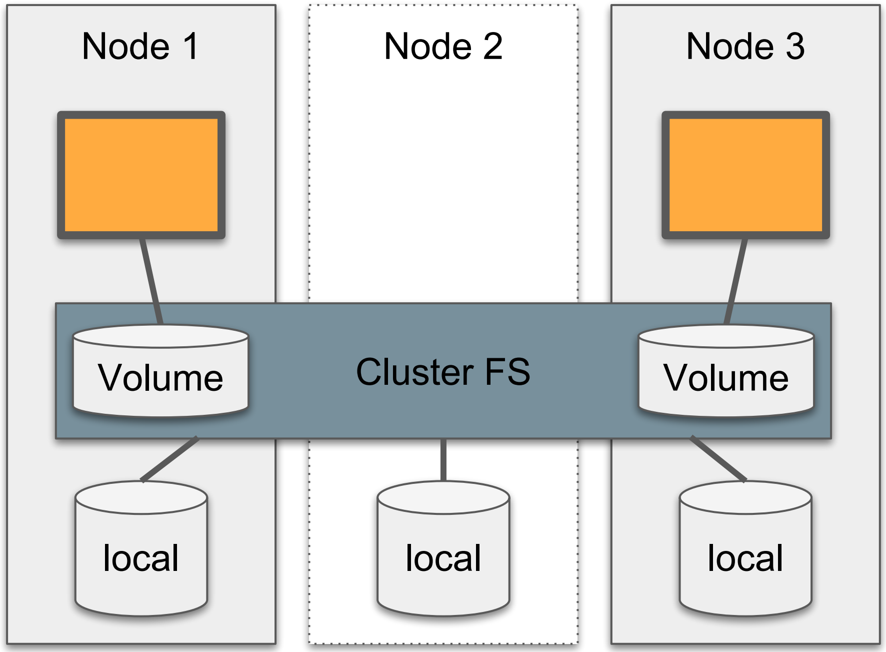
External Services for Storage



- External shares like
  - GCE Persistent Disk
  - AWS Elastic Block Store
  - NFS
  - iSCSI
  - . . .
- Handled by PaaS provider

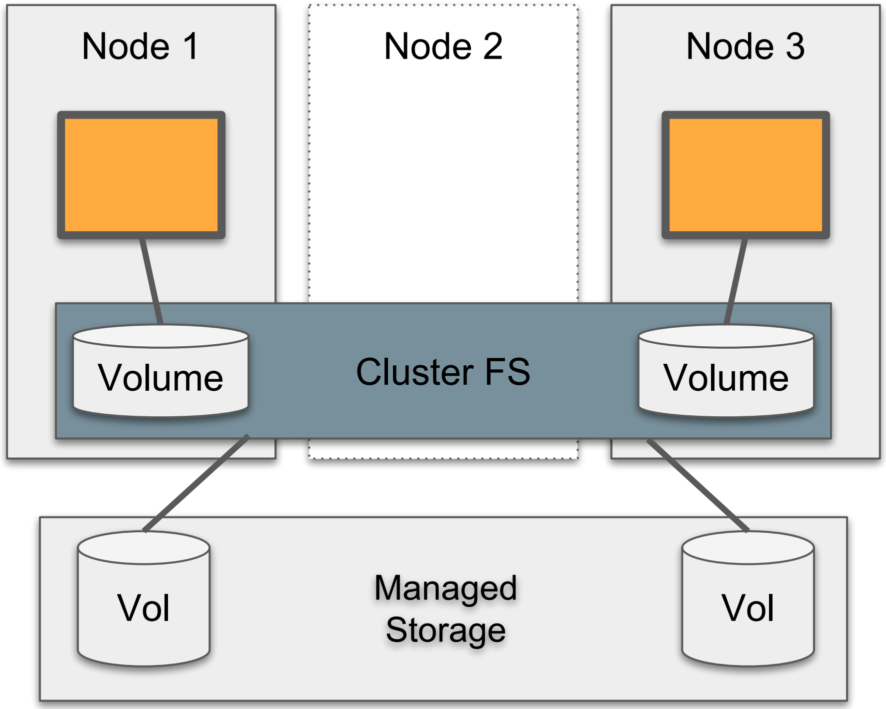
Fixes app to managed storage
One to one relationship (storage - app)

Cloud Native Container Storage 1



- Same shares on every node
- Handles replicas
- Persists to local disk
- Parallel file system
- Dynamic provisioning
- Distributed file locks
- Permission and ACL support
- NFS, S3, ...access

Cloud Native Container Storage 2



- Striping?
- Additional abstraction?

Cluster File System Candidates

Should be Fault Tolerant and Split Brain safe

- Rook / CephFS
- Quobyte
- Heketi / GlusterFS
- Minio
- . . .

Cluster File System Candidates (GlusterFS)

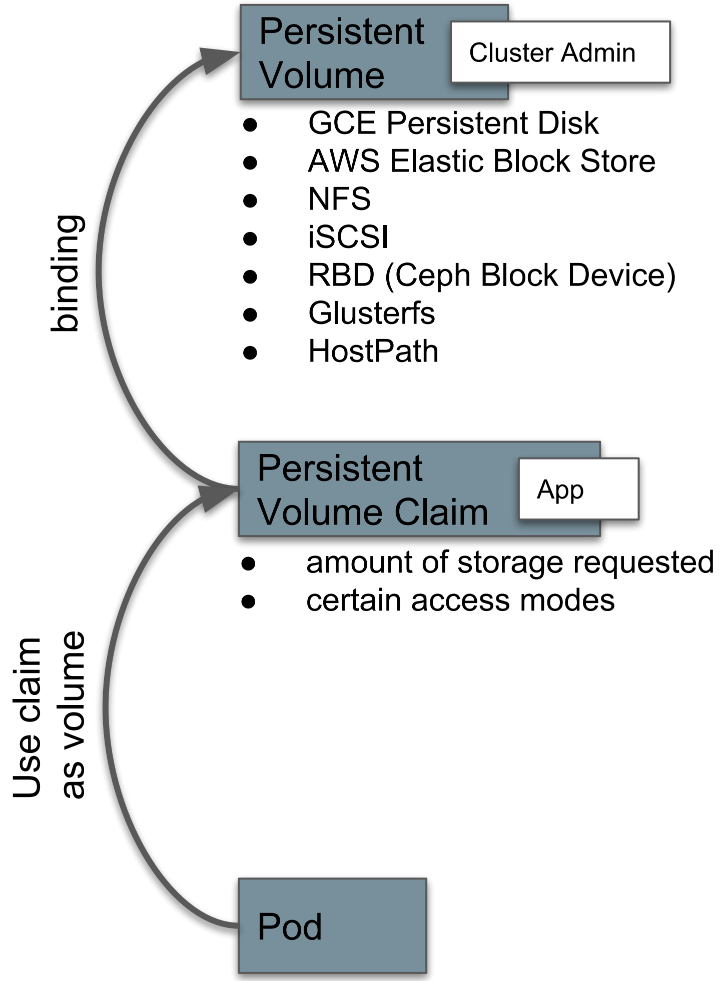
- **Distributed** - Distributed volumes distributes files throughout the bricks in the volume. You can use distributed volumes where the requirement is to scale storage and the redundancy is either not important or is provided by other hardware/software layers.
- **Replicated** – Replicated volumes replicates files across bricks in the volume. You can use replicated volumes in environments where high-availability and high-reliability are critical.
- **Striped** – Striped volumes stripes data across bricks in the volume. For best results, you should use striped volumes only in high concurrency environments accessing very large files.
- **Distributed Striped** - Distributed striped volumes stripe data across two or more nodes in the cluster. You should use distributed striped volumes where the requirement is to scale storage and in high concurrency environments accessing very large files is critical.
- **Distributed Replicated** - Distributed replicated volumes distributes files across replicated bricks in the volume. You can use distributed replicated volumes in environments where the requirement is to scale storage and high-reliability is critical. Distributed replicated volumes also offer improved read performance in most environments.
- **Distributed Striped Replicated** – Distributed striped replicated volumes distributes striped data across replicated bricks in the cluster. For best results, you should use distributed striped replicated volumes in highly concurrent environments where parallel access of very large files and performance is critical. In this release, configuration of this volume type is supported only for Map Reduce workloads.
- **Striped Replicated** – Striped replicated volumes stripes data across replicated bricks in the cluster. For best results, you should use striped replicated volumes in highly concurrent environments where there is parallel access of very large files and performance is critical. In this release, configuration of this volume type is supported only for Map Reduce workloads.
- **Dispersed** - Dispersed volumes are based on erasure codes, providing space-efficient protection against disk or server failures. It stores an encoded fragment of the original file to each brick in a way that only a subset of the fragments is needed to recover the original file. The number of bricks that can be missing without losing access to data is configured by the administrator on volume creation time.
- **Distributed Dispersed** - Distributed dispersed volumes distribute files across dispersed subvolumes. This has the same advantages of distribute replicate volumes, but using disperse to store the data into the bricks.

Kubernetes
Volumes, Claims, Classes, Provisioner

Persistent Volumes

provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed.

- A **Persistent Volume** (PV) is a piece of networked storage that has been provisioned by an administrator
- A **Persistent Volume Claim** (PVC) is a request for storage by a user



Persistence: Examples

```
apiVersion: v1
  kind: PersistentVolume
  metadata:
    name: pv0003
  spec:
    capacity:
      storage: 5Gi
    accessModes:
      - ReadWriteOnce
    persistentVolumeReclaimPolicy: Recycle
    storageClassName: slow
    nfs:
      path: /tmp
      server: 172.17.0.2
```

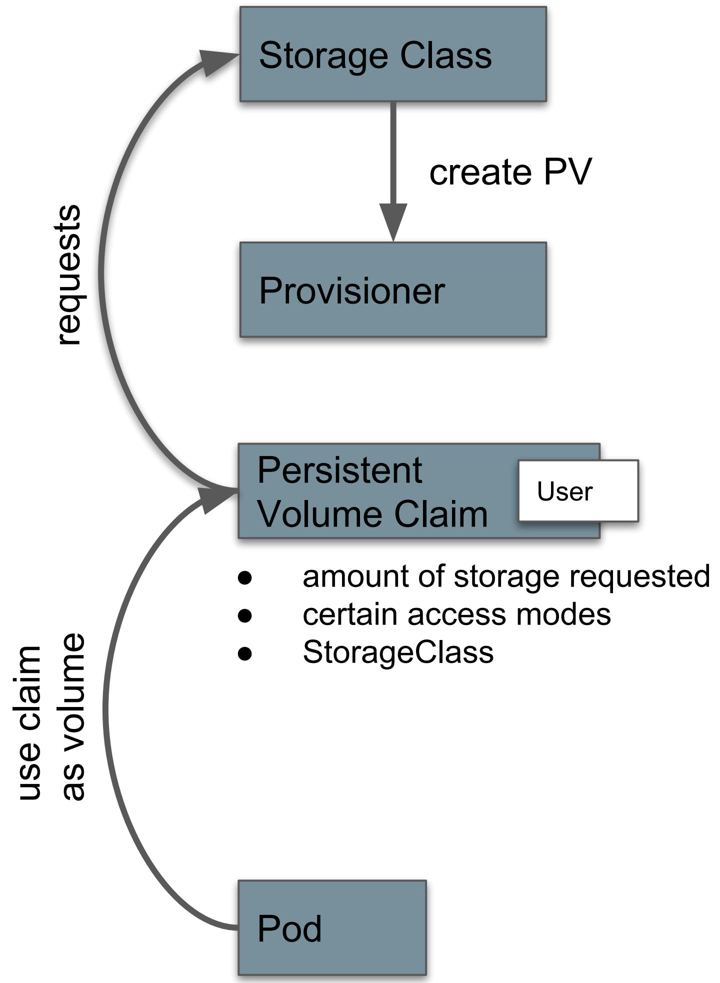
```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: dockerfile/nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: myclaim
```

Persistence Provisioning

provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed.

- A **StorageClass** provides a way to describe the "classes" of offered storage
- A **Provisioner** determines what volume plugin is used for provisioning PVs



Stateful Set

Problem?

With Services and Deployment / ReplicaSet / Pod ...

How would you create a cluster of

- Kafka?
- Mongodb?
- ...?

How would you scale them? What about state / persistence

Stateful Sets

Stateful Sets are valuable for applications that require one or more of the following.

- Stable, unique network identifiers
- Stable, persistent storage
- Ordered, graceful deployment and scaling
- Ordered, graceful deletion and termination

Stable means: Stable across Pod (re)schedulings

Otherwise: Controllers such as Deployment or ReplicaSet may be better suited

Stateful Sets: Components

- A **Headless Service**, is used to control the network domain
- The **StatefulSet**, has a Spec that indicates that replicas of a container will be launched in unique Pods
- The **VolumeClaimTemplates** will provide stable storage using **PersistentVolumes** provisioned by a **PersistentVolume Provisioner**

StatefulSet example: Redis

Creates:

- 3 Redis Pods
- 3 PVs
- 3 PVCs

redis.yaml

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis-persistence
spec:
  serviceName: redis
  replicas: 3
  . . .
    spec:
      containers:
      - name: master
        image: redis:4.0.1-alpine
        command: ["redis-server", "--appendonly",  "yes"]
        volumeMounts:
        - name: data
          mountPath: /data
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

StatefulSet example: Redis

Headless Service:

Does not do anything except creating multiple DNS **A records**

redis.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: redis
  labels:
    app: redis-persistence
spec:
  ports:
    - port: 6379
  clusterIP: None
  selector:
    app: redis-persistence
```

Exercise Stateful Set

1. Review / Create the Provisioner and StorageClass (view sources?)

```go
func (p *hostPathProvisioner) Provision(options controller.VolumeOptions) (*v1.PersistentVolume, error) {
	. . .
	pv := &v1.PersistentVolume{
		ObjectMeta: metav1.ObjectMeta{
			Name: options.PVName,
			Annotations: map[string]string{
				"hostPathProvisionerIdentity": p.identity,
			},
		},
		Spec: v1.PersistentVolumeSpec{. . .},
			PersistentVolumeSource: v1.PersistentVolumeSource{
				HostPath: &v1.HostPathVolumeSource{
					Path: path,
				},
			},
		},
	}
	return pv, nil
}
```

Exercise Stateful Set

1. Review / Create the StatefulSet with `redis.yaml`

2. Try to change replicas (f.e. kubectl edit …). What happens to PV and PVC?

3. Try to review the DNS entries for Service nginx (busybox) (`redis`, `redis.<ns>.svc.cluster.local`, `redis-persistence-0.redis.<ns>.svc.cluster.local`)

4. Recreate Service redis as “non-headless” Service and try 3. again

Exercise Stateful Set

1. *Are there any other Storage Classes?*
2. *What do I need to do to use them?*

Recap: Pod Identity

is comprised of

- an unique ordinal
- a stable network identity
- a stable storage

regardless of which node it’s (re)scheduled on

Pod and DNS Name: `$(statefulset name)-$(ordinal)`

Domain: `$(service name).$(namespace).svc.cluster.local`

Example: DNS with Stateful Sets

```
$ kubectl exec busybox -- nslookup redis
Server:    10.100.0.10
Address 1: 10.100.0.10 kube-dns.kube-system.svc.cluster.local

Name:      redis
Address 1: 10.244.3.68 redis-persistence-2.redis.60mk8s-zw0vlb6mv7ig.svc.cluster.local
Address 2: 10.244.5.78 redis-persistence-0.redis.60mk8s-zw0vlb6mv7ig.svc.cluster.local
Address 3: 10.244.6.62 redis-persistence-1.redis.60mk8s-zw0vlb6mv7ig.svc.cluster.local

$ kubectl exec busybox -- nslookup redis.60mk8s-zw0vlb6mv7ig.svc.cluster.local
Server:    10.100.0.10
Address 1: 10.100.0.10 kube-dns.kube-system.svc.cluster.local

Name:      redis.60mk8s-zw0vlb6mv7ig.svc.cluster.local
Address 1: 10.244.3.68 redis-persistence-2.redis.60mk8s-zw0vlb6mv7ig.svc.cluster.local
Address 2: 10.244.5.78 redis-persistence-0.redis.60mk8s-zw0vlb6mv7ig.svc.cluster.local
Address 3: 10.244.6.62 redis-persistence-1.redis.60mk8s-zw0vlb6mv7ig.svc.cluster.local

$ kubectl exec busybox -- nslookup redis-persistence-0.redis
Server:    10.100.0.10
Address 1: 10.100.0.10 kube-dns.kube-system.svc.cluster.local

Name:      redis-persistence-0.redis
Address 1: 10.244.5.78 redis-persistence-0.redis.60mk8s-zw0vlb6mv7ig.svc.cluster.local
```

Exercise ZooKeeper

1. Review and try Zookeeper.yaml

- Why two services?
- Is online scaling possible?