Scribe

The Scribe authors

May 26, 2021

CONTENTS

1 Installation **1**

1.1 Kubernetes & OpenShift 1

1.2 Development 3

2 Usage **5**

2.1 Triggers 5

2.2 Metrics & monitoring 7

2.3 Rclone-based replication 12

2.4 Restic-based backup 20

2.5 Rsync-based replication 28

2.6 Triggers 46

2.7 Metrics 46

3 Enhancement proposals **47**

3.1 A case for Scribe 47

3.2 Configuration and CRDs 50

3.3 Rsync-based data mover 52

3.4 Restic-based data mover 54

CHAPTER

ONE

INSTALLATION

The following directions will walk through the process of deploying Scribe.

Note: Volume snapshot and clone capabilities are required for some Scribe functionality. It is recommended that you use a CSI driver and StorageClass capable of snapshotting and cloning volumes.

There are three options for installing Scribe. Choose the option that relates to your situation.

Warning: Scribe requires the Kubernetes snapshot controller to be installed within a cluster. If the controller is not deployed review the snapshot controller documentation https://github.com/kubernetes-csi/external-snapshotter.

1.1 Kubernetes & OpenShift

While the operator can be deployed via the `make deploy` or `make deploy-openshift` targets, the recommended method for deploying Scribe is via the Helm chart.

```
# Add the Backube Helm repo
$ helm repo add backube https://backube.github.io/helm-charts/

# Deploy the chart in your cluster
$ helm install --create-namespace -n scribe-system scribe backube/scribe
```

Verify Scribe is running by checking the output of `kubectl get pods`:

```
$ kubectl -n scribe-system get pods
NAME                          READY   STATUS    RESTARTS   AGE
scribe-686c8557bc-cr6k9       2/2     Running   0          13s
```

1.1.1 Configure default CSI storage

AWS

```
$ kubectl annotate sc/gp2 storageclass.kubernetes.io/is-default-class="false" --overwrite
$ kubectl annotate sc/gp2-csi storageclass.kubernetes.io/is-default-class="true" --
↪overwrite

# Install a VolumeSnapshotClass
$ kubectl create -f - << SNAPCLASS
---
apiVersion: snapshot.storage.k8s.io/v1beta1
kind: VolumeSnapshotClass
metadata:
  name: gp2-csi
driver: ebs.csi.aws.com
deletionPolicy: Delete
SNAPCLASS

# Set gp2-csi as default VolumeSnapshotClass
$ kubectl annotate volumesnapshotclass/gp2-csi snapshot.storage.kubernetes.io/is-default-
↪class="true"
```

GCE

```
$ kubectl annotate sc/standard storageclass.kubernetes.io/is-default-class="false" --
↪overwrite
$ kubectl annotate sc/standard-csi storageclass.kubernetes.io/is-default-class="true" --
↪overwrite

# Install a VolumeSnapshotClass
$ kubectl create -f - << SNAPCLASS
---
apiVersion: snapshot.storage.k8s.io/v1beta1
kind: VolumeSnapshotClass
metadata:
  name: standard-csi
driver: pd.csi.storage.gke.io
deletionPolicy: Delete
SNAPCLASS

# Set standard-csi as default VolumeSnapshotClass
$ kubectl annotate volumesnapshotclass/standard-csi snapshot.storage.kubernetes.io/is-
↪default-class="true"
```

At this point it is now possible to use the Rsync and Rclone capabilities of Scribe.

Continue on to the *usage docs*.

1.2 Development

If you are developing Scribe, run the following as it will run the operator locally and output the logs of the controller to your terminal.

```
# Install Scribe CRDs into the cluster
$ make install

# Run the operator locally
$ make run
```

CHAPTER

TWO

USAGE

2.1 Triggers

There are three types of triggers in scribe:

1. Always - no trigger, always run.
2. Schedule - defined by a cronspec.
3. Manual - request to trigger once.

See the sections below with details on each trigger type.

2.1.1 Always

```
spec:
  trigger: {}
```

This option is set either by omitting the trigger field completely or by setting it to empty object. In both cases the effect is the same - keep replicating all the time.

When using Rsync-based replication, the destination should be set to always-listen for incoming replications from the source. Therefore, the default configuration for rsync destination is with no trigger, which keeps waiting for the next trigger by the source to connect.

In this case `status.nextSyncTime` will not be set, but `status.lastSyncTime` will be set at the end of every replication.

2.1.2 Schedule

```
spec:
  trigger:
    schedule: "*/6 * * * *"
```

The synchronization schedule, `.spec.trigger.schedule`, is defined by a cronspec, making the schedule very flexible. Both intervals (shown above) as well as specific times and/or days can be specified.

In this case `status.nextSyncTime` will be set to the next schedule time based on the cronspec, and `status.lastSyncTime` will be set at the end of every replication.

2.1.3 Manual

```
spec:
  trigger:
    manual: my-manual-id-1
```

Manual trigger is used for running one replication and wait for it to complete. This is useful to control the replication schedule from an external automation (for example using quiesce for live migration).

To use the manual trigger choose a string value and set it in `spec.trigger.manual` which will start a replication. Once replication completes, `status.lastManualSync` will be set to the same string value. As long as these two values are the same there will be no trigger, and the replication will remain paused, until further updates to the trigger spec.

After setting the manual trigger in spec, the user should watch for `status.lastManualSync` and wait for it to have the expected value, which means that the manual trigger completed. If needed, the user can then continue to update `spec.trigger.manual` to a new value in order to trigger another replication.

Something to keep in mind when using manual trigger - the update of `spec.trigger.manual` by itself does not interrupt a running replication, and `status.lastManualSync` will simply be set to the value from the spec when the current replication completes. This means that to make sure we know when the replication started, and that it includes the latest data, it is recommended to wait until `status.lastManualSync` equals to `spec.trigger.manual` before setting to a new value.

In this case `status.nextSyncTime` will not be set, but `status.lastSyncTime` will be set at the end of every replication.

Here is an example of how to use manual trigger to run two replications:

```
MANUAL=first
SOURCE=source1

# create source replication with first manual trigger (will start immediately)
kubectl create -f - <<EOF
apiVersion: scribe.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: $SOURCE
spec:
  trigger:
    manual: $MANUAL
  ...
EOF

# waiting for first trigger to complete...
while [ "$LAST_MANUAL_SYNC" != "$MANUAL" ]
  do
    sleep 1
    LAST_MANUAL_SYNC=$(kubectl get replicationsource $SOURCE --template={{.status.
↪lastManualSync}})
    echo " - LAST_MANUAL_SYNC: $LAST_MANUAL_SYNC"
  done

# set a second manual trigger
MANUAL=second
```

(continues on next page)

(continued from previous page)

```
kubectl patch replicationsources $SOURCE --type merge -p '{"spec":{"trigger":{"manual":"'
↪$MANUAL'"}}}'

# waiting for second trigger to complete...
while [ "$LAST_MANUAL_SYNC" != "$MANUAL" ]
  do
    sleep 1
    LAST_MANUAL_SYNC=$(kubectl get replicationsource $SOURCE --template={{.status.
↪lastManualSync}})
    echo " - LAST_MANUAL_SYNC: $LAST_MANUAL_SYNC"
  done

# after second trigger is done we delete the replication...
kubectl delete replicationsources $SOURCE
```

2.2 Metrics & monitoring

In order to support monitoring of replication relationships, Scribe exports a number of metrics that can be scraped with Prometheus. These metrics permit monitoring whether volumes are "in sync" and how long the synchronization iterations take.

2.2.1 Available metrics

The following metrics are provided by Scribe for each replication object (source or destination):

scribe_missed_intervals_total This is a count of the number of times that a replication iteration failed to complete before the next scheduled start. This metric is only valid for objects that have a schedule (`.spec.trigger.schedule`) specified. For example, when using the rsync mover with a schedule on the source but not on the destination, only the metric for the source side is meaningful.

scribe_sync_duration_seconds This is a summary of the time required for each sync iteration. By monitoring this value it is possible to determine how much "slack" exists in the synchronization schedule (i.e., how much less is the sync duration than the schedule frequency).

scribe_volume_out_of_sync This is a gauge that has the value of either "0" or "1", with a "1" indicating that the volumes are not currently synchronized. This may be due to an error that is preventing synchronization or because the most recent synchronization iteration failed to complete prior to when the next should have started. This metric also requires a schedule to be defined.

Each of the above metrics include the following labels to assist with monitoring and alerting:

obj_name This is the name of the Scribe CustomResource

obj_namespace This is the Kubernetes Namespace that contains the CustomResource

role This contains the value of either "source" or "destination" depending on whether the CR is a ReplicationSource or a ReplicationDestination.

method This indicates the synchronization method being used. Currently, "rsync" or "rclone".

As an example, the below raw data comes from a single rsync-based relationship that is replicating data using the ReplicationSource `dsrc` in the `srcns` namespace to the ReplicationDestination `dest` in the `dstns` namespace.

Listing 1: Example raw metrics data

```
$ curl -s http://127.0.0.1:8080/metrics | grep scribe

# HELP scribe_missed_intervals_total The number of times a synchronization failed to␣
↪complete before the next scheduled start
# TYPE scribe_missed_intervals_total counter
scribe_missed_intervals_total{method="rsync",obj_name="dest",obj_namespace="dstns",role=
↪"destination"} 0
scribe_missed_intervals_total{method="rsync",obj_name="dsrc",obj_namespace="srcns",role=
↪"source"} 0
# HELP scribe_sync_duration_seconds Duration of the synchronization interval in seconds
# TYPE scribe_sync_duration_seconds summary
scribe_sync_duration_seconds{method="rsync",obj_name="dest",obj_namespace="dstns",role=
↪"destination",quantile="0.5"} 179.725047058
scribe_sync_duration_seconds{method="rsync",obj_name="dest",obj_namespace="dstns",role=
↪"destination",quantile="0.9"} 544.86628289
scribe_sync_duration_seconds{method="rsync",obj_name="dest",obj_namespace="dstns",role=
↪"destination",quantile="0.99"} 544.86628289
scribe_sync_duration_seconds_sum{method="rsync",obj_name="dest",obj_namespace="dstns",
↪role="destination"} 828.711667153
scribe_sync_duration_seconds_count{method="rsync",obj_name="dest",obj_namespace="dstns",
↪role="destination"} 3
scribe_sync_duration_seconds{method="rsync",obj_name="dsrc",obj_namespace="srcns",role=
↪"source",quantile="0.5"} 11.547060835
scribe_sync_duration_seconds{method="rsync",obj_name="dsrc",obj_namespace="srcns",role=
↪"source",quantile="0.9"} 12.013468222
scribe_sync_duration_seconds{method="rsync",obj_name="dsrc",obj_namespace="srcns",role=
↪"source",quantile="0.99"} 12.013468222
scribe_sync_duration_seconds_sum{method="rsync",obj_name="dsrc",obj_namespace="srcns",
↪role="source"} 33.317039014
scribe_sync_duration_seconds_count{method="rsync",obj_name="dsrc",obj_namespace="srcns",
↪role="source"} 3
# HELP scribe_volume_out_of_sync Set to 1 if the volume is not properly synchronized
# TYPE scribe_volume_out_of_sync gauge
scribe_volume_out_of_sync{method="rsync",obj_name="dest",obj_namespace="dstns",role=
↪"destination"} 0
scribe_volume_out_of_sync{method="rsync",obj_name="dsrc",obj_namespace="srcns",role=
↪"source"} 0
```

2.2.2 Obtaining metrics

The above metrics can be collected by Prometheus. If the cluster does not already have a running instance set to scrape metrics, one will need to be started.

Configuring Prometheus

Kubernetes

The following steps start a simple Prometheus instance to scrape metrics from Scribe. Some platforms may already have a running Prometheus operator or instance, making these steps unnecessary.

Start the Prometheus operator:

```
$ kubectl apply -f https://raw.githubusercontent.com/prometheus-operator/prometheus-
↪operator/v0.46.0/bundle.yaml
```

Start Prometheus by applying the following block of yaml via:

```
$ kubectl create ns scribe-system
$ kubectl -n scribe-system apply -f -
```

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: prometheus
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus
rules:
  - apiGroups: [""]
    resources:
      - nodes
      - services
      - endpoints
      - pods
    verbs: ["get", "list", "watch"]
  - apiGroups: [""]
    resources:
      - configmaps
    verbs: ["get"]
  - nonResourceURLs: ["/metrics"]
    verbs: ["get"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: prometheus
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus
subjects:
  - kind: ServiceAccount
    name: prometheus
    namespace: scribe-system  # Change if necessary!
---
```

(continues on next page)

(continued from previous page)

```
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: prometheus
spec:
  serviceAccountName: prometheus
  serviceMonitorSelector:
    matchLabels:
      control-plane: scribe-controller
  resources:
    requests:
      memory: 400Mi
```

OpenShift

If necessary, create a monitoring configuration in the `openshift-user-workload-monitoring` namespace and enable user workload monitoring:

Listing 2: Example user workload monitoring configuration

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: user-workload-monitoring-config
  namespace: openshift-user-workload-monitoring
data:
  config.yaml: |
    # Allocate persistent storage for user Prometheus
    prometheus:
      volumeClaimTemplate:
        spec:
          resources:
            requests:
              storage: 40Gi
    # Allocate persistent storage for user Thanos Ruler
    thanosRuler:
      volumeClaimTemplate:
        spec:
          resources:
            requests:
              storage: 40Gi
```

Listing 3: Enabling user workload monitoring

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: cluster-monitoring-config
  namespace: openshift-monitoring
data:
  config.yaml: |
```

(continues on next page)

(continued from previous page)

```
    # Allocate persistent storage for alertmanager
    alertmanagerMain:
      volumeClaimTemplate:
        spec:
          resources:
            requests:
              storage: 40Gi
    # Enable user workload monitoring stack
    enableUserWorkload: true
    # Allocate persistent storage for cluster prometheus
    prometheusK8s:
      volumeClaimTemplate:
        spec:
          resources:
            requests:
              storage: 40Gi
```

Monitoring Scribe

The metrics port for Scribe is (by default) protected via kube-auth-proxy. In order to grant Prometheus the ability to scrape the metrics, its ServiceAccount must be granted access to the `scribe-metrics-reader` ClusterRole. This can be accomplished by (substitute in the namespace & SA name of the Prometheus server):

```
$ kubectl create clusterrolebinding metrics --clusterrole=scribe-metrics-reader --
↪serviceaccount=<namespace>:<service-account-name>
```

Optionally, authentication of the metrics port can be disabled by setting the Helm chart value `metrics.disableAuth` to `false` when deploying Scribe.

A ServiceMonitor needs to be defined in order to scrape metrics. If the ServiceMonitor CRD was defined in the cluster when the Scribe chart was deployed, this has already been added. If not, apply the following into the namespace where Scribe is deployed. Note that the `control-plane` labels may need to be adjusted.

Listing 4: Scribe ServiceMonitor

```
---
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: scribe-monitor
  namespace: scribe-system
  labels:
    control-plane: scribe-controller
spec:
  endpoints:
    - interval: 30s
      path: /metrics
      port: https
      scheme: https
      tlsConfig:
        # Using self-signed cert for connection
        insecureSkipVerify: true
```

(continues on next page)

(continued from previous page)

```
  selector:
    matchLabels:
      control-plane: scribe-controller
```

2.3 Rclone-based replication

2.3.1 Rclone Database Example

The following example will use the Rclone replication method and take a Snapshot at the source. A MySQL database will be used as the example application.

First, create the source. Next deploy the source MySQL database.

```
$ kubectl create ns source
$ kubectl create -f examples/source-database/ -n source
```

Verify the database is running.

```
$ kubectl get pods -n source
NAME                       READY   STATUS    RESTARTS   AGE
mysql-8b9c5c8d8-24w6g      1/1     Running   0          17s
```

Add a new database.

```
$ kubectl exec --stdin --tty -n source `kubectl get pods -n source | grep mysql | awk '
↪{print $1}'` -- /bin/bash
$ mysql -u root -p$MYSQL_ROOT_PASSWORD
> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
4 rows in set (0.00 sec)


> create database synced;
> exit
$ exit
```

Now, deploy the `rclone-secret` followed by `ReplicationSource` configuration.

```
$ kubectl create secret generic rclone-secret --from-file=rclone.conf=./examples/rclone.
↪conf -n source
$ kubectl create -f examples/scribe_v1alpha1_replicationsource_rclone.yaml -n source
```

To verify the replication has completed describe the Replication source.

```
$ kubectl describe ReplicationSource -n source database-source
```

From the output, the success of the replication can be seen by the following lines:

```
Status:
 Conditions:
   Last Transition Time:  2021-01-18T21:50:59Z
   Message:               Reconcile complete
   Reason:                ReconcileComplete
   Status:                True
   Type:                  Reconciled
 Next Sync Time:          2021-01-18T22:00:00Z
```

At `Next Sync Time` Scribe will create the next Rclone data mover job.

To complete the replication, create a destination, deploy `rclone-secret` and `ReplicationDestination` on the destination.

```
$ kubectl create ns dest
$ kubectl create secret generic rclone-secret --from-file=rclone.conf=./examples/rclone.
↪conf -n dest
$ kubectl create -f examples/scribe_v1alpha1_replicationdestination_rclone.yaml -n dest
```

Once the `ReplicationDestination` is deployed, Scribe will create a Rclone data mover job on the destination side. At the end of the each successful reconcilation iteration, the `ReplicationDestination` is updated with the lastest snapshot image.

Now deploy the MySQL database to the `dest` namespace which will use the data that has been replicated. First we need to identify the latest snapshot from the `ReplicationDestination` object. Record the values of the latest snapshot as it will be used to create a pvc. Then create the Deployment, Service, PVC, and Secret.

Ensure the Snapshots Age is not greater than 3 minutes as it will be replaced by Scribe before it can be used.

```
$ kubectl get replicationdestination database-destination -n dest --template={{.status.
↪latestImage.name}}
$ sed -i 's/snapshotToReplace/scribe-dest-database-destination-20201203174504/g'␣
↪examples/destination-database/mysql-pvc.yaml
$ kubectl create -n dest -f examples/destination-database/
```

Validate that the mysql pod is running within the environment.

```
$ kubectl get pods -n dest
NAME                                              READY   STATUS    RESTARTS   AGE
mysql-8b9c5c8d8-v6tg6                             1/1     Running   0          38m
```

Connect to the mysql pod and list the databases to verify the synced database exists.

```
$ kubectl exec --stdin --tty -n dest `kubectl get pods -n dest | grep mysql | awk '
↪{print $1}'` -- /bin/bash
$ mysql -u root -p$MYSQL_ROOT_PASSWORD
> show databases;
+--------------------+
| Database           |
```

(continues on next page)

(continued from previous page)

```
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| synced             |
| sys                |
+--------------------+
5 rows in set (0.00 sec)

> exit
$ exit
```

2.3.2 Understanding `rclone-secret`

What is `rclone-secret`?

This file provides the configuration details to locate and access the intermediary storage system. It is mounted as a secret on the Rclone data mover pod.

```
[aws-s3-bucket]
type = s3
provider = AWS
env_auth = false
access_key_id = *******
secret_access_key = ******
region = <region>
location_constraint = <region>
acl = private
```

In the above example AWS S3 is used as the backend for the intermediary storage system.

- `[aws-s3-bucket]`: Name of the remote
- `type`: Type of storage
- `provider`: Backend provider
- `access_key_id`: AWS credentials
- `secret_access_key`: AWS credentials
- `region`: Region to connect to
- `location_constraint`: Must be set to match the `region`

For detailed instructions follow Rclone

Deploy `rclone-secret`

Source side

```
$ kubectl create secret generic rclone-secret --from-file=rclone.conf=./examples/rclone.
↪conf -n source
$ kubectl get secrets -n source
NAME                   TYPE                                  DATA   AGE
default-token-g9vdx    kubernetes.io/service-account-token   3      20s
rclone-secret          Opaque                                1      17s
```

Destination side

```
$ kubectl create secret generic rclone-secret --from-file=rclone.conf=./examples/rclone.
↪conf -n dest
$ kubectl get secrets -n dest
NAME                   TYPE                                  DATA   AGE
default-token-5ngtg    kubernetes.io/service-account-token   3      17s
rclone-secret          Opaque                                1      6s
```

Contents

Rclone-based replication

- *Rclone-based replication*
  - *Source configuration*
    - *Source status*
    - *Additional source options*
  - *Destination configuration*
    - *Destination status*
    - *Additional destination options*

Rclone-based replication supports 1:many asynchronous replication of volumes for use cases such as:

- High fan-out data replication from a central site to many (edge) sites

With this method, Scribe synchronizes data from a ReplicationSource to a ReplicationDestination using Rclone using an intermediary storage system like AWS S3.

The Rclone method uses a "push" and "pull" model for the data replication. A schedule or other trigger is used on the source side of the relationship to trigger each replication iteration.

Following are the sequences of events happening in each iterations.

- A point-in-time (PiT) copy of the source volume is created using CSI drivers. It will be used as the source data.
- A temporary PVC is created out of the PiT copy and mounted on Rclone data mover job pod.
- The Scribe Rclone data mover then connects to the intermediary storage system (e.g. AWS S3) using configurations based on `rclone-secret`. It uses `rclone sync` to copy source data to S3.
- At the conclusion of the transfer, the destination creates a PiT copy to preserve the incoming source data.

Scribe is configured via two CustomResources (CRs), one on the source side and one on the destination side of the replication relationship.

2.3.3 Source configuration

Start by configuring the source; a minimal example is shown below:

```yaml
---
apiVersion: scribe.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: database-source
  namespace: source
spec:
  sourcePVC: mysql-pv-claim
  trigger:
    schedule: "*/6 * * * *"
  rclone:
    rcloneConfigSection: "aws-s3-bucket"
    rcloneDestPath: "scribe-test-bucket"
    rcloneConfig: "rclone-secret"
    copyMethod: Snapshot
```

Since the `copyMethod` specified above is `Snapshot`, the Rclone data mover creates a `VolumeSnapshot` of the source pvc `mysql-pv-claim` using the cluster's default `VolumeSnapshotClass`.

The synchronization schedule, `.spec.trigger.schedule`, is defined by a cronspec, making the schedule very flexible. Both intervals (shown above) as well as specific times and/or days can be specified.

It then creates a temproray pvc `scribe-src-database-source` out of the VolumeSnapshot to transfer source data to the intermediary storage system like AWS S3 using the configurations provided in `rclone-secret`

Source status

Once the `ReplicationSource` is deployed, Scribe updates the `nextSyncTime` in the `ReplicationSource` object.

```yaml
---
apiVersion:  scribe.backube/v1alpha1
kind:         ReplicationSource
#  ... omitted ...
spec:
  rclone:
    copyMethod:           Snapshot
    rcloneConfig:         rclone-secret
    rcloneConfigSection:  aws-s3-bucket
    rcloneDestPath:       scribe-test-bucket
  sourcePVC:               mysql-pv-claim
  trigger:
    schedule:  "*/6 * * * *"
  status:
    conditions:
      lastTransitionTime:  2021-01-18T21:50:59Z
      message:               Reconcile complete
```

(continues on next page)

(continued from previous page)

```
      reason:                 ReconcileComplete
      status:                 True
      type:                   Reconciled
    nextSyncTime:           2021-01-18T22:00:00Z
```

In the above `ReplicationSource` object,

- The Rclone configuations are provided via `rclone-secret`
- The PiT copy of the source data `mysql-pv-claim` will be created using cluster's default `VolumeSnapshot`.
- `rcloneDestPath` indicates the location on the intermediary storage system where the source data will be copied
- The synchronization schedule, `.spec.trigger.schedule`, is defined by a cronspec, making the schedule very flexible. Both intervals (shown above) as well as specific times and/or days can be specified.
- No errors were detected (the Reconciled condition is True)
- `nextSyncTime` indicates the time of the upcoming Rclone data mover job

Additional source options

There are a number of more advanced configuration parameters that are supported for configuring the source. All of the following options would be placed within the .spec.rclone portion of the ReplicationSource CustomResource.

accessModes When using a copyMethod of Clone or Snapshot, this field allows overriding the access modes for the point-in-time (PiT) volume. The default is to use the access modes from the source PVC.

capacity When using a copyMethod of Clone or Snapshot, this allows overriding the capacity of the PiT volume. The default is to use the capacity of the source volume.

copyMethod This specifies the method used to create a PiT copy of the source volume. Valid values are:

- **Clone** - Create a new volume by cloning the source PVC (i.e., use the source PVC as the volumeSource for the new volume.
- **None** - Do no create a PiT copy. The Scribe data mover will directly use the source PVC.
- **Snapshot** - Create a VolumeSnapshot of the source PVC, then use that snapshot to create the new volume. This option should be used for CSI drivers that support snapshots but not cloning.

storageClassName This specifies the name of the StorageClass to use when creating the PiT volume. The default is to use the same StorageClass as the source volume.

volumeSnapshotClassName When using a copyMethod of Snapshot, this specifies the name of the VolumeSnapshotClass to use. If not specified, the cluster default will be used.

rcloneConfigSection This is used to idenitfy the object storage configuration within `rclone.conf` to use.

rcloneDestPath This is the remote storage location in which the persistent data will be uploaded.

rcloneConfig This specifies the secret to be used. The secret contains credentials for the remote storage location.

2.3.4 Destination configuration

A minimal destination configuration is shown here:

```
---
apiVersion: scribe.backube/v1alpha1
kind: ReplicationDestination
metadata:
  name: database-destination
  namespace: dest
spec:
trigger:
  schedule: "*/6 * * * *"
  rclone:
    rcloneConfigSection: "aws-s3-bucket"
    rcloneDestPath: "scribe-test-bucket"
    rcloneConfig: "rclone-secret"
    copyMethod: Snapshot
    accessModes: [ReadWriteOnce]
    capacity: 10Gi
```

In the above example, a 10 GiB RWO volume will be provisioned using the default `StorageClass` to serve as the destination for replicated data. This volume is used by the Rclone data mover to receive the incoming data transfers.

Similar to the replication source, a synchronization schedule is defined `.spec.trigger.schedule`. This indicates when persistent data should be pulled from the remote storage location.

Since the `copyMethod` specified above is `Snapshot`, a `VolumeSnapshot` of the incoming data will be created at the end of each synchronization interval. It is this snapshot that would be used to gain access to the replicated data. The name of the current `VolumeSnapshot` holding the latest synced data will be placed in `.status.latestImage`.

Destination status

Scribe provides status information on the state of the replication via the `.status` field in the ReplicationDestination object:

```
---
API Version:  scribe.backube/v1alpha1
Kind:         ReplicationDestination
#  ... omitted ...
Spec:
Rclone:
  Access Modes:
    ReadWriteOnce
  Capacity:                10Gi
  Copy Method:             Snapshot
  Rclone Config:           rclone-secret
  Rclone Config Section:   aws-s3-bucket
  Rclone Dest Path:        scribe-test-bucket
  Status:
    Conditions:
      Last Transition Time:   2021-01-19T22:16:02Z
      Message:                Reconcile complete
      Reason:                 ReconcileComplete
```

(continues on next page)

(continued from previous page)

```
    Status:                True
    Type:                  Reconciled
  Last Sync Duration:      7.066022293s
  Last Sync Time:          2021-01-19T22:16:02Z
  Latest Image:
    API Group:  snapshot.storage.k8s.io
    Kind:       VolumeSnapshot
    Name:       scribe-dest-database-destination-20210119221601
```

In the above example,

- `Rclone Dest Path` indicates the intermediary storage system from where data will be transfered to the destination site. In the above example, the intermediary storage system is S3 bucket
- No errors were detected (the Reconciled condition is True)

After at least one synchronization has taken place, the following will also be available:

- `Last Sync Time` contains the time of the last successful data synchronization.
- `Latest Image` references the object with the most recent copy of the data. If the copyMethod is Snapshot, this will be a VolumeSnapshot object. If the copyMethod is None, this will be the PVC that is used as the destination by Scribe.

Additional destination options

There are a number of more advanced configuration parameters that are supported for configuring the destination. All of the following options would be placed within the `.spec.rclone` portion of the ReplicationDestination CustomResource.

accessModes When Scribe creates the destination volume, this specifies the accessModes for the PVC. The value should be ReadWriteOnce or ReadWriteMany.

capacity When Scribe creates the destination volume, this value is used to determine its size. This need not match the size of the source volume, but it must be large enough to hold the incoming data.

copyMethod This specifies how the data should be preserved at the end of each synchronization iteration. Valid values are:

- **None** - Do not create a point-in-time copy of the data.
- **Snapshot** - Create a VolumeSnapshot at the end of each iteration

destinationPVC Instead of having Scribe automatically provision the destination volume (using capacity, accessModes, etc.), the name of a pre-existing PVC may be specified here.

storageClassName When Scribe creates the destination volume, this specifies the name of the StorageClass to use. If omitted, the system default StorageClass will be used.

volumeSnapshotClassName When using a copyMethod of Snapshot, this value specifies the name of the VolumeSnapshotClass to use when creating a snapshot.

rcloneConfigSection This is used to idenitfy the object storage configuration within `rclone.conf` to use.

rcloneDestPath This is the remote storage location in which the persistent data will be downloaded.

rcloneConfig This specifies the secret to be used. The secret contains credentials for the remote storage location.

For a concrete example, see the *database synchronization example*.

2.4 Restic-based backup

2.4.1 Restic Database Example

Restic backup

Restic is a fast and secure backup program. The following example will use Restic to create a backup of a source volume.

A MySQL database will be used as the example application.

Creating source PVC to be backed up

Create a namespace called `source`, and deploy the source MySQL database.

```
$ kubectl create ns source
$ kubectl -n source create -f examples/source-database/
```

Verify the database is running:

```
$ kubectl -n source get pods,pvc,volumesnapshots

NAME                         READY    STATUS    RESTARTS   AGE
pod/mysql-87f849f8c-n9j7j    1/1      Running   1          58m

NAME                                       STATUS    VOLUME                                     ␣
↪    CAPACITY   ACCESS MODES   STORAGECLASS      AGE
persistentvolumeclaim/mysql-pv-claim   Bound     pvc-adbf57f1-6399-4738-87c9-
↪4c660d982a0f   2Gi        RWO            csi-hostpath-sc   60m
```

Add a new database:

```
$ kubectl exec --stdin --tty -n source `kubectl get pods -n source | grep mysql | awk '
↪{print $1}'` -- /bin/bash

$ mysql -u root -p$MYSQL_ROOT_PASSWORD

> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
4 rows in set (0.00 sec)


> create database synced;
> exit

$ exit
```

Restic Repository Setup

For the purpose of this tutorial we are using minio as the object storage target for the backup.

Start `minio`:

```
$ hack/run-minio.sh
```

The `restic-config` Secret configures the Restic repository parameters:

```
---
apiVersion: v1
kind: Secret
metadata:
  name: restic-config
type: Opaque
stringData:
  # The repository url
  RESTIC_REPOSITORY: s3:http://minio.minio.svc.cluster.local:9000/restic-repo
  # The repository encryption key
  RESTIC_PASSWORD: my-secure-restic-password
  # ENV vars specific to the back end
  # https://restic.readthedocs.io/en/stable/030_preparing_a_new_repo.html
  AWS_ACCESS_KEY_ID: access
  AWS_SECRET_ACCESS_KEY: password
```

ReplicationSource

Start by configuring the source; a minimal example is shown below

```
---
apiVersion: scribe.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: database-source
  namespace: source
spec:
  sourcePVC: mysql-pv-claim
  trigger:
    schedule: "*/30 * * * *"
restic:
  pruneIntervalDays: 15
  repository: restic-config
  retain:
    hourly: 1
    daily: 1
    weekly: 1
    monthly: 1
    yearly: 1
  copyMethod: Clone
```

In the above `ReplicationSource` object,

- The PiT copy of the source data `mysql-pv-claim` will be created by cloning the source volume.

- The synchronization schedule, `.spec.trigger.schedule`, is defined by a cronspec, making the schedule very flexible. In this case, it will take a backup every 30 minutes.
- The restic repository configuration is provided via the `restic-config` Secret.
- `pruneIntervalDays` defines the interval between Restic prune operations.
- The `retain` settings determine how many backups should be saved in the repository. Read more about restic forget.

Now, deploy the `restic-config` followed by `ReplicationSource` configuration.

```
$ kubectl create -f example/source-restic/source-restic.yaml -n source
$ kubectl create -f examples/scribe_v1alpha1_replicationsource_restic.yaml -n source
```

To verify the replication has completed, view the the ReplicationSource `.status` field.

```
$ kubectl -n source get ReplicationSource/database-source -oyaml

apiVersion: scribe.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: database-source
  namespace: source
spec:
  # ... lines omitted ...
status:
  conditions:
  - lastTransitionTime: "2021-05-17T18:16:35Z"
    message: Reconcile complete
    reason: ReconcileComplete
    status: "True"
    type: Reconciled
  lastSyncDuration: 3m10.261673933s
  lastSyncTime: "2021-05-17T18:19:45Z"
  nextSyncTime: "2021-05-17T18:30:00Z"
  restic: {}
```

In the above output, the `lastSyncTime` shows the time when the last backup completed.

The backup created by Scribe can be seen by directly accessing the Restic repository:

```
# In one window, create a port forward to access the minio server
$ kubectl port-forward --namespace minio svc/minio 9000:9000

# An another, access the repository w/ restic via the above forward
$ AWS_ACCESS_KEY_ID=access AWS_SECRET_ACCESS_KEY=password restic -r s3:http://127.0.0.
↪1:9000/restic-repo snapshots
enter password for repository:
repository 03fd0c91 opened successfully, password is correct
created new cache in /home/jstrunk/.cache/restic
ID        Time                 Host        Tags        Paths
------------------------------------------------------------
caebaa8e  2021-05-17 14:19:42  scribe                  /data
```

(continues on next page)

(continued from previous page)

```
------------------------------------------------------------
1 snapshots
```

There is a snapshot in the restic repository created by the restic data mover.

Restoring the backup

To restore from the backup, create a destination, deploy `restic-config` and `ReplicationDestination` on the destination.

```
$ kubectl create ns dest
$ kubectl -n dest create -f examples/source-restic/
```

To start the restore, create a empty PVC for the data:

```
$ kubectl -n dest create -f examples/source-database/mysql-pvc.yaml
persistentvolumeclaim/mysql-pv-claim created
```

Create the ReplicationDestination in the `dest` namespace to restore the data:

```yaml
---
apiVersion: scribe.backube/v1alpha1
kind: ReplicationDestination
metadata:
  name: database-destination
spec:
  trigger:
    manual: restore
  restic:
    destinationPVC: mysql-pv-claim
    repository: restic-config
    copyMethod: None
```

```
$ kubectl -n dest create -f examples/scribe_v1alpha1_replicationdestination_restic.yaml
```

Once the restore is complete, the `.status.lastManualSync` field will match `.spec.trigger.manual`.

To verify restore, deploy the MySQL database to the `dest` namespace which will use the data that has been restored from sourcePVC backup.

```
$ kubectl create -n dest -f examples/destination-database/
```

Validate that the mysql pod is running within the environment.

```
$ kubectl get pods -n dest
NAME                                              READY   STATUS    RESTARTS   AGE
mysql-8b9c5c8d8-v6tg6                             1/1     Running   0          38m
```

Connect to the mysql pod and list the databases to verify the synced database exists.

```
$ kubectl exec --stdin --tty -n dest `kubectl get pods -n dest | grep mysql | awk '
↪{print $1}'` -- /bin/bash
$ mysql -u root -p$MYSQL_ROOT_PASSWORD
```

(continues on next page)

(continued from previous page)

```
> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| synced             |
| sys                |
+--------------------+
5 rows in set (0.00 sec)

> exit
$ exit
```

Contents

Backing up using Restic

- *Restic-based backup*
  - *Specifying a repository*
  - *Configuring backup*
    - *Backup options*
  - *Performing a restore*
    - *Restore options*

Scribe supports taking backups of PersistentVolume data using the Restic-based data mover. A ReplicationSource defines the backup policy (target, frequency, and retention), while a ReplicationDestination is used for restores.

The Restic mover is different than most of Scribe's other movers because it is not meant for synchronizing data between clusters. This mover is specifically meant for data backup.

2.4.2 Specifying a repository

For both backup and restore operations, it is necessary to specify a backup repository for Restic. The repository and connection information are defined in a `restic-config` Secret.

Below is an example showing how to use a repository stored on Minio.

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: restic-config
type: Opaque
stringData:
  # The repository url
  RESTIC_REPOSITORY: s3:http://minio.minio.svc.cluster.local:9000/restic-repo
  # The repository encryption key
  RESTIC_PASSWORD: my-secure-restic-password
  # ENV vars specific to the chosen back end
```

(continues on next page)

(continued from previous page)

```
  # https://restic.readthedocs.io/en/stable/030_preparing_a_new_repo.html
  AWS_ACCESS_KEY_ID: access
  AWS_SECRET_ACCESS_KEY: password
```

This Secret will be referenced for both backup (ReplicationSource) and for restore (ReplicationDestination).

Note: If necessary, the repository will be automatically initialized (i.e., `restic init`) during the first backup.

2.4.3 Configuring backup

A backup policy is defined by a ReplicationSource object that uses the restic replication method.

```
---
apiVersion: scribe.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: mydata-backup
spec:
  # The PVC to be backed up
  sourcePVC: mydata
  trigger:
    # Take a backup every 30 minutes
    schedule: "*/30 * * * *"
restic:
  # Prune the repository (repack to free space) every 2 weeks
  pruneIntervalDays: 14
  # Name of the Secret with the connection information
  repository: restic-config
  # Retention policy for backups
  retain:
    hourly: 6
    daily: 5
    weekly: 4
    monthly: 2
    yearly: 1
  # Clone the source volume prior to taking a backup to ensure a
  # point-in-time image.
  copyMethod: Clone
```

Backup options

There are a number of additional configuration options not shown in the above example. Scribe's Restic mover options closely follow those of Restic itself.

accessModes When using a copyMethod of Clone or Snapshot, this field allows overriding the access modes for the point-in-time (PiT) volume. The default is to use the access modes from the source PVC.

capacity When using a copyMethod of Clone or Snapshot, this allows overriding the capacity of the PiT volume. The default is to use the capacity of the source volume.

copyMethod This specifies the method used to create a PiT copy of the source volume. Valid values are:

- **Clone** - Create a new volume by cloning the source PVC (i.e., use the source PVC as the volumeSource for the new volume.
- **None** - Do no create a PiT copy. The Scribe data mover will directly use the source PVC.
- **Snapshot** - Create a VolumeSnapshot of the source PVC, then use that snapshot to create the new volume. This option should be used for CSI drivers that support snapshots but not cloning.

storageClassName This specifies the name of the StorageClass to use when creating the PiT volume. The default is to use the same StorageClass as the source volume.

volumeSnapshotClassName When using a copyMethod of Snapshot, this specifies the name of the VolumeSnapshot-Class to use. If not specified, the cluster default will be used.

cacheCapacity This determines the size of the Restic metadata cache volume. This volume contains cached metadata from the backup repository. It must be large enough to hold the non-pruned repository metadata. The default is `1 Gi`.

cacheStorageClassName This is the name of the StorageClass that should be used when provisioning the cache volume. It defaults to `.spec.storageClassName`, then to the name of the StorageClass used by the source PVC.

cacheAccessModes This is the access mode(s) that should be used to provision the cache volume. It defaults to `.spec.accessModes`, then to the access modes used by the source PVC.

pruneIntervalDays This determines the number of days between running `restic prune` on the repository. The prune operation repacks the data to free space, but it can also generate significant I/O traffic as a part of the process. Setting this option allows a trade-off between storage consumption (from no longer referenced data) and access costs.

repository This is the name of the Secret (in the same Namespace) that holds the connection information for the backup repository. The repository path should be unique for each PV.

retain This has sub-fields for `hourly`, `daily`, `weekly`, `monthly`, and `yearly` that allow setting the number of each type of backup to retain. There is an additional field, `within` that can be used to specify a time period during which all backups should be retained. See Restic's documentation on –keep-within for more information.

When more than the specified number of backups are present in the repository, they will be removed via Restic's `forget` operation, and the space will be reclaimed during the next prune.

2.4.4 Performing a restore

Data from a backup can be restored using the ReplicationDestination CR. In most cases, it is desirable to perform a single restore into an empty PersistentVolume.

For example, create a PVC to hold the restored data:

```yaml
---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: datavol
spec:
accessModes:
  - ReadWriteOnce
resources:
  requests:
    storage: 3Gi
```

Restore the data into `datavol`:

```yaml
---
apiVersion: scribe.backube/v1alpha1
kind: ReplicationDestination
metadata:
  name: datavol-dest
spec:
  trigger:
    manual: restore-once
  restic:
    repository: restic-repo
    destinationPVC: datavol
    copyMethod: None
```

In the above example, the data will be written directly into the new PVC since it is specified via `destinationPVC`, and no snapshot will be created since a `copyMethod` of `None` is used.

The restore operation only needs to be performed once, so instead of using a cronspec-based schedule, a manual trigger is used. After the restore completes, the ReplicationDestination object can be deleted.

Note: Currently, Scribe only supports restoring the latest backup. However, older backups may be present in the repository (according to the retain parameters). Those can be accessed directly using the Restic utility plus the connection information and credentials from the repository Secret.

Restore options

There are a number of additional configuration options not shown in the above example.

accessModes When Scribe creates the destination volume, this specifies the accessModes for the PVC. The value should be ReadWriteOnce or ReadWriteMany.

capacity When Scribe creates the destination volume, this value is used to determine its size. This need not match the size of the source volume, but it must be large enough to hold the incoming data.

copyMethod This specifies how the data should be preserved at the end of each synchronization iteration. Valid values are:

- **None** - Do not create a point-in-time copy of the data.
- **Snapshot** - Create a VolumeSnapshot at the end of each iteration

destinationPVC Instead of having Scribe automatically provision the destination volume (using capacity, accessModes, etc.), the name of a pre-existing PVC may be specified here.

storageClassName When Scribe creates the destination volume, this specifies the name of the StorageClass to use. If omitted, the system default StorageClass will be used.

volumeSnapshotClassName When using a copyMethod of Snapshot, this value specifies the name of the VolumeSnapshotClass to use when creating a snapshot.

cacheCapacity This determines the size of the Restic metadata cache volume. This volume contains cached metadata from the backup repository. It must be large enough to hold the non-pruned repository metadata. The default is `1 Gi`.

cacheStorageClassName This is the name of the StorageClass that should be used when provisioning the cache volume. It defaults to `.spec.storageClassName`, then to the name of the StorageClass used by the source PVC.

cacheAccessModes This is the access mode(s) that should be used to provision the cache volume. It defaults to `.spec.accessModes`, then to the access modes used by the source PVC.

repository This is the name of the Secret (in the same Namespace) that holds the connection information for the backup repository. The repository path should be unique for each PV.

2.5 Rsync-based replication

2.5.1 Rsync Database Example

The following example will use the Rsync replication method and take a Snapshot at the destination. A MySQL database will be used as the example application.

First, create the destination and deploy the ReplicationDestination configuration.

```
$ kubectl create ns dest
$ kubectl create -n dest -f examples/scribe_v1alpha1_replicationdestination.yaml
```

A Service is created which will be used by the ReplicationSource to Rsync the data. Record the service IP address as it will be used for the ReplicationSource.

```
$ kubectl get replicationdestination database-destination -n dest --template={{.status.
↪rsync.address}}
10.107.249.72
```

Now it is time to deploy our database.

```
$ kubectl create ns source
$ kubectl create -n source -f examples/source-database
```

Verify the database is running.

```
$ kubectl get pods -n source
NAME                     READY   STATUS    RESTARTS   AGE
mysql-8b9c5c8d8-24w6g    1/1     Running   0          17s
```

Now it is time to create the ReplicationSource items. First, we need the ssh secret from the dest namespace.

```
$ kubectl get secret -n dest scribe-rsync-dest-src-database-destination -o yaml > /tmp/
↪secret.yaml
$ vi /tmp/secret.yaml
# ^^^ change the namespace to "source"
# ^^^ remove the owner reference (.metadata.ownerReferences)
$ kubectl create -f /tmp/secret.yaml
```

Using the IP address that relates to the ReplicationDestination that was recorded earlier. Modify `scribe_v1alpha1_replicationsource.yaml` replacing the value of the address and create the ReplicationSource object.

```
$ sed -i 's/my.host.com/10.107.249.72/g' examples/scribe_v1alpha1_replicationsource.yaml
$ kubectl create -n source -f examples/scribe_v1alpha1_replicationsource.yaml
```

To verify the replication has completed describe the Replication source.

```
$ kubectl describe ReplicationSource -n source database-source
```

From the output, the success of the replication can be seen by the following lines:

```
Status:
  Conditions:
    Last Transition Time:  2020-12-03T16:07:35Z
    Message:               Reconcile complete
    Reason:                ReconcileComplete
    Status:                True
    Type:                  Reconciled
  Last Sync Duration:      4.511334577s
  Last Sync Time:          2020-12-03T16:09:04Z
  Next Sync Time:          2020-12-03T16:12:00Z
```

Create a database in the mysql pod running in the source namespace.

```
$ kubectl exec --stdin --tty -n source `kubectl get pods -n source | grep mysql | awk '
↪{print $1}'` -- /bin/bash
$ mysql -u root -p$MYSQL_ROOT_PASSWORD
> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
4 rows in set (0.00 sec)


> create database synced;
> exit
$ exit
```

Now the mysql database will be deployed to the dest namespace which will use the data that has been replicated. First we need to identify the latest snapshot from the ReplicationDestination object. Record the values of the latest snapshot as it will be used to create a pvc. Then create the Deployment, Service, PVC, and Secret. Ensure the Snapshots Age is not greater than 3 minutes as it will be replaced by Scribe before it can be used.

```
$ kubectl get replicationdestination database-destination -n dest --template={{.status.
↪latestImage.name}}
$ sed -i 's/snapshotToReplace/scribe-dest-database-destination-20201203174504/g'␣
↪examples/destination-database/mysql-pvc.yaml
$ kubectl create -n dest -f examples/destination-database/
```

Validate that the mysql pod is running within the environment.

```
$ kubectl get pods -n dest
NAME                                              READY   STATUS    RESTARTS   AGE
mysql-8b9c5c8d8-v6tg6                             1/1     Running   0          38m
```

Connect to the mysql pod and list the databases to verify the synced database exists.

```
$ kubectl exec --stdin --tty -n dest `kubectl get pods -n dest | grep mysql | awk '
↪{print $1}'` -- /bin/bash
$ mysql -u root -p$MYSQL_ROOT_PASSWORD
> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| synced             |
| sys                |
+--------------------+
5 rows in set (0.00 sec)
```

2.5.2 Rsync Database Plugin Example

This example will sync data from mysql database persistent volumes For this example, sync will happen within a single cluster and 2 namespaces.

Note:

- *Cluster must have the scribe operator installed*.
- *Cluster storage may need configuring*.

Build Scribe CLI

```
$ make scribe
$ mv bin/kubectl-scribe /usr/local/bin (or add to $PATH)
```

Create a Scribe-Config

Create a config file to designate your source and destination options. You can also pass these individually to each command, but they add up so the config file is usually a good option. You can add any, some, or all flags to the config file.

Create the config file at `./config.yaml` *or* `~/.scribeconfig/config.yaml`, scribe will look for that file in the current directory or in `~/.scribeconfig`. For complete list of options for a command, run the following or consult the API:

```
$ kubectl scribe <command> -h
```

```
$ cat config.yaml

dest-access-mode: ReadWriteOnce
dest-copy-method: Snapshot
dest-namespace: dest
source-namespace: source
```

(continues on next page)

(continued from previous page)

```
source-pvc: mysql-pv-claim
source-copy-method: Snapshot
```

Refer to the *example config* that lists plugin options with default values.

Create Source Application

```
$ kubectl create ns source
$ kubectl -n source apply -f examples/source-database/
```

Modify the Mysql Database

```
$ kubectl exec --stdin --tty -n source `kubectl get pods -n source | grep mysql | awk '
↪{print $1}'` -- /bin/bash
# mysql -u root -p$MYSQL_ROOT_PASSWORD
> create database my_new_database;
> show databases;
> exit
$ exit
```

Start a Scribe Replication

```
$ kubectl scribe start-replication
```

The above command: * Creates destination PVC (if dest PVC not provided & if dest CopyMethod=None) * Creates replication destination * Syncs SSH secret from destination to source * Creates replication source

Necessary flags are configured in `./config.yaml` shown above.

Set and Pause a Scribe Replication

Usually the source deployment will be scaled down before pinning a point-in-time image.

```
$ kubectl scale deployment/mysql --replicas=0 -n source
```

```
$ kubectl scribe set-replication
```

The above command: * Sets a manual trigger on the replication source * Waits for final data sync to complete * Creates destination PVC with latest snapshot (if dest PVC not provided & if dest CopyMethod=Snapshot)

Necessary flags are configured in `./config.yaml` shown above.

Create a Destination Application if not already running

Create the destination application from the scribe example:

```
$ kubectl apply -n dest -f examples/destination-database/mysql-deployment.yaml
$ kubectl apply -n dest -f examples/destination-database/mysql-service.yaml
$ kubectl apply -n dest -f examples/destination-database/mysql-secret.yaml
```

Edit the Destination Application with Destination PVC

```
$ kubectl edit deployment/mysql -n dest
```

Replace the value of Spec.Volumes.PersistentVolumeClaim.ClaimName with name of destination PVC created from the source PVC. By default, this will be named *sourcePVCName-date-time-stamp* in destination namespace.

Verify the Synced Database

```
$ kubectl exec --stdin --tty -n dest `kubectl get pods -n dest | grep mysql | awk '
↪{print $1}'` -- /bin/bash
# mysql -u root -p$MYSQL_ROOT_PASSWORD
> show databases;
> exit
$ exit
```

Resume Existing Scribe Replication

It may be desireable to periodically sync data from source to destination. In this case, the *continue-replication* command is available.

```
$ kubectl scribe continue-replication
```

The above command: * Removes a manual trigger on the replication source

It is now possible to set the replication again with the following.

```
$ kubectl scribe set-replication
```

After setting a replication, the destination application may be updated to reference the latest destination PVC. The stale destination PVC will remain in the destination namespace.

Remove Scribe Replication

After verifying the destination application is up-to-date and the destination PVC is bound, the scribe replication can be removed. **Scribe does not delete source or destination PVCs**. Each new destination PVC is tagged with a date and time. It is up to the user to prune stale destination PVCs.

```
$ kubectl scribe remove-replication
```

The above command: * Removes the replication source * Removed the synced SSH Secret from the source namespace * Removes the replication destination

2.5.3 External Rsync

A situation may occur where data needs to be imported into a Kubernetes environment. In the Scribe repository, the script *bin/external-rsync-source* can be executed which will serve as the *replicationsource* allowing data to be replicated to a Kubernetes cluster.

Usage

This binary works by using Rsync and SSH to copy a directory into an endpoint within a Kubernetes cluster created by the *replicationdestination* object. Connectivity must exist from the data source to the Kubernetes cluster. Because of the simplicity the underlying storage does not matter. This means the directory could exist on a NFS share, within GlusterFS, or on the servers filesystem.

The binary requires specific flags to be provided.

- -d Destination address to rsync the data
- -i Path to the SSH Key
- -s Source Directory

An example usage of the script would be to copy the */var/www/html* directory to the LoadBalancer service created by the *replicationdestination*.

```
$ external-rsync-source -s /var/www/html -d a48a38bf6f69c4070831391e8b22e8d5-
↪08027986c9de8c10.elb.us-east-2.amazonaws.com -i /home/user/source-key
```

Migration Example

In this example, a database is running on a RHEL 8 server. It has been decided that this database should move from a server into a Kubernetes environment.

Starting at the Kubernetes cluster create the *Namespace*, *replicationdestination*, and the *PVC* using the examples below.

```
$ kubectl create ns database
$ kubectl create -f examples/external-database/replicationdestination.yaml
$ kubectl create -f examples/external-database/mysql-pvc.yaml
```

This will geneate a LoadBalancer service which will be used by our binary as our destination address.

```
$ kubectl get replicationdestination database-destination -n database --template={{.
↪status.rsync.address}}
```

The *replicationdestination* created an SSH key to be used on the server.

Acquire the private key by running the following.

```
$ kubectl get secret -n database scribe-rsync-dest-src-database-destination --template {
↪{.data.source}} | base64 -d > ~/replication-key
$ chmod 0600 ~/replication-key
```

From the server, run the *external-rsync-source* binary specifying the Loadbalancer, SSH private key, and MySQL directory.

```
$ ./external-rsync-source -i ~/replication-key -s /var/lib/mysql/ -d␣
↪a48a38bf6f69c4070831391e8b22e8d5-08027986c9de8c10.elb.us-east-2.amazonaws.com
```

At the Kubernetes cluster we can now create our database deployment.

```
$ kubectl create -f examples/external-database/mysql-deployment.yaml
```

Now that the MySQL deployment is running, verify the expected databases exist within the Kubernetes cluster. When logging into the database the password and authentication values were copied over from the database running on the RHEL server.

```
$ kubectl exec --stdin --tty -n database `kubectl get pods -n database | grep mysql |␣
↪awk '{print $1}'` -- /bin/bash
$ root@mysql-87c47498d-7rc9m:/# mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 15
Server version: 8.0.23 MySQL Community Server - GPL

Copyright (c) 2000, 2021, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> show databases;
+--------------------+
| Database           |
+--------------------+
| employees          |
| information_schema |
| mysql              |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.01 sec)

mysql> exit
Bye
```

2.5.4 Rsync Database Cross-Cluster Plugin Example

This example will sync data from mysql database persistent volumes For this example, sync will happen between 2 clusters. Data will be synced from cluster-name `api-source-com:6443` to cluster-name `destination123` Snapshot copy method will be used.

Note:

- *Clusters must have the scribe operator installed*.
- *Cluster storage may need configuring*.

Build Scribe

```
$ make scribe
$ mv bin/kubectl-scribe /usr/local/bin (or add to $PATH)
```

Merge Kubeconfigs

If clusters not already in a single kubeconfig, then merge like so:

~/kubeconfig1 with context `destuser` and cluster-name `destination123`

~/kubeconfig2 with context `sourceuser` and cluster-name `api-source-com:6443`

```
$ export KUBECONFIG=~/kubeconfig1:~/kubeconfig2
```

You can view config with the following commands:

```
$ kubectl config view
$ kubectl config get-clusters
$ kubectl config get-contexts
```

You can rename contexts with the following:

```
$ kubectl config rename-context <oldname> <newname>
```

Create Source Application

```
$ kubectl --context sourceuser create ns source
$ kubectl --context sourceuser -n source apply -f examples/source-database/
```

Modify the Mysql Database

```
$ kubectl --context sourceuser exec --stdin --tty -n source `kubectl --context␣
↪sourceuser get pods -n source | grep mysql | awk '{print $1}'` -- /bin/bash
# mysql -u root -p$MYSQL_ROOT_PASSWORD
> create database my_new_database;
> show databases;
> exit
$ exit
```

Create a Scribe-Config

Create a config file to designate your source and destination options. You can also pass these individually to each command, but they add up so the config file is usually a good option. You can add any, some, or all flags to the config file. For multiple clusters, you must pass the source and destination contexts and cluster names.

Create the config file at `./config.yaml` *or* `~/.scribeconfig/config.yaml`, scribe will look for that file in the current directory or in `~/.scribeconfig`. For complete list of options for a command, run the following or consult the API:

```
$ kubectl scribe <command> -h
```

```
$ cat config.yaml

dest-kube-context: destuser
dest-kube-clustername: destination123
dest-service-type: LoadBalancer
dest-access-mode: ReadWriteOnce
dest-copy-method: Snapshot
dest-namespace: dest
source-kube-context: sourceuser
source-kube-clustername: api-source-com:6443
source-namespace: source
source-service-type: LoadBalancer
source-copy-method: Snapshot
source-pvc: mysql-pv-claim
```

Refer to the *example config* that lists plugin options with default values.

Start a Scribe Replication

```
$ kubectl scribe start-replication
```

The above command: * Creates destination PVC (if dest PVC not provided & if dest CopyMethod=None) * Creates replication destination * Syncs SSH secret from destination to source * Creates replication source

Necessary flags are configured in `./config.yaml` shown above.

Set and Pause a Scribe Replication

Usually the source deployment will be scaled down before pinning a point-in-time image.

```
$ kubectl scale deployment/mysql --replicas=0 -n source --context sourceuser
```

```
$ kubectl scribe set-replication
```

The above command: * Sets a manual trigger on the replication source * Waits for final data sync to complete * Creates destination PVC with latest snapshot (if dest PVC not provided & if dest CopyMethod=Snapshot)

Necessary flags are configured in `./config.yaml` shown above.

For the rest of the example, you'll be working from the `destuser context`. So we don't have to pass that to every kubectl command, run this:

```
$ kubectl config use-context destuser
```

Create a Destination Application if not already running

Create the destination application from the scribe example:

```
$ kubectl apply -n dest -f examples/destination-database/mysql-deployment.yaml
$ kubectl apply -n dest -f examples/destination-database/mysql-service.yaml
$ kubectl apply -n dest -f examples/destination-database/mysql-secret.yaml
```

Edit the Destination Application with Destination PVC

```
$ kubectl edit deployment/mysql -n dest
```

Replace the value of Spec.Volumes.PersistentVolumeClaim.ClaimName with name of destination PVC created from the source PVC. By default, this will be named *sourcePVCName-date-time-stamp* in destination namespace.

Verify the Synced Database

```
$ kubectl exec --stdin --tty -n dest `kubectl get pods -n dest | grep mysql | awk '
↪{print $1}'` -- /bin/bash
# mysql -u root -p$MYSQL_ROOT_PASSWORD
> show databases;
> exit
$ exit
```

Resume Existing Scribe Replication

It may be desireable to periodically sync data from source to destination. In this case, the *continue-replication* command is available. If scaled down, then scale back up and resume replications.

```
$ kubectl scale deployment/mysql --replicas=1 -n source --context sourceuser
```

```
$ kubectl scribe continue-replication
```

The above command: * Removes a manual trigger on the replication source

It is now possible to set the replication again with the following.

```
$ kubectl scale deployment/mysql --replicas=0 -n source --context sourceuser
$ kubectl scribe set-replication
```

After setting a replication, the destination application may be updated to reference the latest destination PVC. The stale destination PVC will remain in the destination namespace.

Remove Scribe Replication

After verifying the destination application is up-to-date and the destination PVC is bound, the scribe replication can be removed. **Scribe does not delete source or destination PVCs**. Each new destination PVC is tagged with a date and time. It is up to the user to prune stale destination PVCs.

```
$ kubectl scribe remove-replication
```

The above command: * Removes the replication source * Removed the synced SSH Secret from the source namespace * Removes the replication destination

2.5.5 SSH keys

Scribe generates SSH keys upon the deployment of a ReplicationDestination object but SSH keys can also be provided to Scribe rather than generating new ones. The steps below will describe the process to provide Scribe SSH keys.

Generating keys

`ssh-keygen` can be used to generate SSH keys. The keys that are created will be used to create secrets which will be used by Scribe.

Two keys need to be generated. The first SSH key will called `destination`.

```
$ ssh-keygen -t rsa -b 4096 -f destination -C "" -N ""
Generating public/private rsa key pair.
Your identification has been saved in destination
Your public key has been saved in destination.pub
The key fingerprint is:
SHA256:5gRLpIdeu+3CbkScH7qIsEw6tMNPRdVFUe82ihWw5BU
The key's randomart image is:
+---[RSA 4096]----+
|      ... o*oE.  |
|     +.  .o + .  |
|    oo=.   o . . |
|   ..+++.     o  |
|    .oooS.   . + |
|.o  . o*.   o o .|
|*o.o +..o  . .   |
|+=o . =.         |
| .o. o...        |
+----[SHA256]-----+
```

The second SSH key will be called *source*:

```
$ ssh-keygen -t rsa -b 4096 -f source -C "" -N ""
Generating public/private rsa key pair.
Your identification has been saved in source
Your public key has been saved in source.pub
The key fingerprint is:
SHA256:NEQNMNsgR43Y3c2dWMyit70JagmbCLNRfakWhWORENU
The key's randomart image is:
+---[RSA 4096]----+
|    .+OX*O o *.. |
```

(continues on next page)

(continued from previous page)

```
|    .oo*B E = =  |
|      .o+o o .   |
|      ..o.+ .    |
|     .  S+ . o   |
|    +   +   o .  |
|     = o + o . o |
|    . . o +   o  |
|         .       |
+----[SHA256]-----+
```

Creating secrets

Secrets will be created using the SSH keys that were generated above. These keys must reside on the cluster and namespace that serves as the replication source/destination.

The destination needs access to the public and private destination keys but only the public source key:

```
$ kubectl create ns dest
$ kubectl create secret generic scribe-rsync-dest-dest-database-destination --from-
↪file=destination=destination --from-file=source.pub=source.pub --from-file=destination.
↪pub=destination.pub -n dest
```

The source needs access to the public and private source keys but only the public destination key:

```
$ kubectl create ns source
$ kubectl create secret generic scribe-rsync-dest-src-database-destination --from-
↪file=source=source --from-file=source.pub=source.pub --from-file=destination.
↪pub=destination.pub -n source
```

Replication destination configuration

The last step to use these keys is to provide the value of `sshKeys` to the ReplicationDestination object as a field. As an example we will modify `examples/scribe_v1alpha1_replicationdestination.yaml`.

```
---
apiVersion: scribe.backube/v1alpha1
kind: ReplicationDestination
metadata:
  name: database-destination
  namespace: dest
spec:
  rsync:
    serviceType: ClusterIP
    copyMethod: Snapshot
    capacity: 2Gi
    accessModes: [ReadWriteOnce]
    # This is the name of the Secret we created, above
    sshKeys: scribe-rsync-dest-dest-database-destination
```

The ReplicationDestination object can now be created:

```
$ kubectl create -f examples/scribe_v1alpha1_replicationdestination.yaml
```

The above steps should be repeated to modify set the `sshKeys` field in the ReplicationSource.

2.5.6 Scribe Plugin Options with Defaults

These may be passed as `key: value` pairs in a file, on the command-line, or a combination of both. Command-line values override the config file values.

Scribe Plugin will look for this file in `./config.yaml`, `~/.scribeconfig/config.yaml`, or from the command-line passed `--config` value that is a path to a local file.

```
dest-address: <remote address to connect to for replication>
dest-name: <dest-namespace>-destination
dest-namespace: <current namespace>
dest-kube-context: <kubectl config current-context>
dest-kube-clustername: <current-context clustername>
dest-access-mode: one of ReadWriteOnce|ReadOnlyMany|ReadWriteMany
dest-capacity: <source-capacity>
dest-cron-spec: <continuous>
dest-pvc: <if not provided, scribe will provision one>
dest-service-type: 'ClusterIP'
dest-ssh-user: 'root'
dest-storage-class-name: <default sc>
dest-volume-snapshot-class-name: <default vsc>
dest-copy-method: one of None|Clone|Snapshot
dest-port: 22
dest-provider: <external replication provider, pass as 'domain.com/provider'>
dest-provider-params: <key=value configuration parameters, if external provider; pass as
↪'key=value,key1=value1'>
dest-path: /
source-name: <source-namespace>-source
source-namespace: <current namespace>
source-kube-context: <current-context>
source-kube-clustername: <current-context clustername>
source-access-mode: one of ReadWriteOnce|ReadOnlyMany|ReadWriteMany
source-capacity: "2Gi"
source-cron-spec: "*/3 * * * *"
source-pvc: <name of existing PVC to replicate>
source-service-type: 'ClusterIP'
source-ssh-user: 'root'
source-storage-class-name: <default sc>
source-volume-snapshot-class-name: <default vsc>
source-copy-method: one of None|Clone|Snapshot
source-port: 22
source-provider: <external replication provider, pass as 'domain.com/provider'>
source-provider-params: <key=value configuration parameters, if external provider; pass␣
↪as 'key=value,key1=value1'>
ssh-keys-secret: <scribe-rsync->dest-src-<name-of-replication-destination>
```

Contents

Rsync-based replication

- *Rsync-based replication*
  - *Destination configuration*
    - *Destination status*
    - *Additional destination options*
  - *Source configuration*
    - *Source status*
    - *Additional source options*

Rsync-based replication supports 1:1 asynchronous replication of volumes for use cases such as:

- Disaster recovery
- Mirroring to a test environment
- Sending data to a remote site for processing

With this method, Scribe synchronizes data from a ReplicationSource to a ReplicationDestination using Rsync across an ssh connection. By using Rsync, the amount of data transferred during each synchronization is kept to a minimum, and the ssh connection ensures that the data transfer is both authenticated and secure.

The Rsync method is typically configured to use a "push" model for the data replication. A schedule or other trigger is used on the source side of the relationship to trigger each replication iteration.

During each iteration, (optionally) a point-in-time (PiT) copy of the source volume is created and used as the source data. The Scribe Rsync data mover then connects to the destination using ssh (exposed via a Service or load balancer) and sends any updates. At the conclusion of the transfer, the destination (optionally) creates a VolumeSnapshot to preserve the updated data.

Scribe is configured via two CustomResources (CRs), one on the source side and one on the destination side of the replication relationship.

2.5.7 Destination configuration

Start by configuring the destination; a minimal example is shown below:

```yaml
---
apiVersion: scribe/v1alpha1
kind: ReplicationDestination
metadata:
  name: myDest
  namespace: myns
spec:
  rsync:
    copyMethod: Snapshot
    capacity: 10Gi
    accessModes: ["ReadWriteOnce"]
```

In the above example, a 10 GiB RWO volume will be provisioned using the default StorageClass to serve as the destination for replicated data. This volume is used by the rsync data mover to receive the incoming data transfers.

Since the `copyMethod` specified above is `Snapshot`, a VolumeSnapshot will be created at the end of each synchronization interval. It is this snapshot that would be used to gain access to the replicated data. The name of the current VolumeSnapshot holding the latest synced data will be placed in `.status.latestImage`.

Destination status

Scribe provides status information on the state of the replication via the `.status` field in the ReplicationDestination object:

```yaml
---
apiVersion: scribe/v1alpha1
kind: ReplicationDestination
metadata:
  name: myDest
  namespace: myns
spec:
  rsync:
  # ... omitted ...
status:
  conditions:
    - lastTransitionTime: "2021-01-14T19:43:07Z"
      message: Reconcile complete
      reason: ReconcileComplete
      status: "True"
      type: Reconciled
  lastSyncDuration: 31.333710313s
  lastSyncTime: "2021-01-14T19:43:07Z"
  latestImage:
    apiGroup: snapshot.storage.k8s.io
    kind: VolumeSnapshot
    name: scribe-dest-test-20210114194305
  rsync:
    address: 10.99.236.225
    sshKeys: scribe-rsync-dest-src-test
```

In the above example,

- No errors were detected (the Reconciled condition is True)
- The destination ssh server is available at the IP specified in `.status.rsync.address`. This should be used when configuring the corresponding ReplicationSource.
- The ssh keys for the source to use are available in the Secret `.status.rsync.sshKeys`.

After at least one synchronization has taken place, the following will also be available:

- lastSyncTime contains the time of the last successful data synchronization.
- latestImage references the object with the most recent copy of the data. If the copyMethod is Snapshot, this will be a VolumeSnapshot object. If the copyMethod is None, this will be the PVC that is used as the destination by Scribe.

Additional destination options

There are a number of more advanced configuration parameters that are supported for configuring the destination. All of the following options would be placed within the `.spec.rsync` portion of the ReplicationDestination CustomResource.

accessModes When Scribe creates the destination volume, this specifies the accessModes for the PVC. The value should be ReadWriteOnce or ReadWriteMany.

capacity When Scribe creates the destination volume, this value is used to determine its size. This need not match the size of the source volume, but it must be large enough to hold the incoming data.

copyMethod This specifies how the data should be preserved at the end of each synchronization iteration. Valid values are:

- **None** - Do not create a point-in-time copy of the data.
- **Snapshot** - Create a VolumeSnapshot at the end of each iteration

destinationPVC Instead of having Scribe automatically provision the destination volume (using capacity, access-Modes, etc.), the name of a pre-existing PVC may be specified here.

storageClassName When Scribe creates the destination volume, this specifies the name of the StorageClass to use. If omitted, the system default StorageClass will be used.

volumeSnapshotClassName When using a copyMethod of Snapshot, this value specifies the name of the VolumeS-napshotClass to use when creating a snapshot.

sshKeys This is the name of a Secret that contains the ssh keys for authenticating the connection with the source. If not provided, the destination keys will be automatically generated and corresponding source keys will be placed in a new Secret. The name of that new Secret will be placed in `.status.rsync.sshKeys`.

serviceType Scribe creates a Service to allow the source to connect to the destination. This field determines the type of that Service. Allowed values are ClusterIP or LoadBalancer. The default is ClusterIP.

port This determines the TCP port number that is used to connect via ssh. The default is 22.

2.5.8 Source configuration

A minimal source configuration is shown here:

```yaml
---
apiVersion: scribe.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: mySource
  namespace: source
spec:
  sourcePVC: mysql-pv-claim
  trigger:
    schedule: "*/5 * * * *"
  rsync:
    sshKeys: scribe-rsync-dest-src-database-destination
    address: my.host.com
    copyMethod: Clone
```

In the above example, the PVC named `mysql-pv-claim` will be replicated every 5 minutes using the Rsync replication method. At the start of each iteration, a clone of the source PVC will be created to generate a point-in-time copy for the iteration. The source will then use the ssh keys in the named Secret (`.spec.rsync.sshKeys`) to authenticate to the destination. The connection will be made to the address specified in `.spec.rsync.address`.

The synchronization schedule, `.spec.trigger.schedule`, is defined by a cronspec, making the schedule very flexible. Both intervals (shown above) as well as specific times and/or days can be specified.

Source status

The state of the replication from the source's point of view is available in the `.status` field of the ReplicationSource:

```yaml
---
apiVersion: scribe.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: mySource
  namespace: source
spec:
  sourcePVC: mysql-pv-claim
  trigger:
    schedule: "*/5 * * * *"
  rsync:
    # ... omitted ...
status:
  conditions:
    - lastTransitionTime: "2021-01-14T19:42:38Z"
      message: Reconcile complete
      reason: ReconcileComplete
      status: "True"
      type: Reconciled
  lastSyncDuration: 7.774288635s
  lastSyncTime: "2021-01-14T20:10:07Z"
  nextSyncTime: "2021-01-14T20:15:00Z"
  rsync: {}
```

In the above example,

- No errors were detected (the Reconciled condition is True).
- The last synchronization was completed at `.status.lastSyncTime` and took `.status.lastSyncDuration` seconds.
- The next scheduled synchronization is at `.status.nextSyncTime`.

Note: The length of time required to synchronize the data is determined by the rate of change for data in the volume and the bandwidth between the source and destination. In order to avoid missed intervals, ensure there is sufficient bandwidth between the source and destination such that `lastSyncTime` remains safely below the synchronization interval (`.spec.trigger.schedule`).

Additional source options

There are a number of more advanced configuration parameters that are supported for configuring the source. All of the following options would be placed within the .spec.rsync portion of the ReplicationSource CustomResource.

accessModes When using a copyMethod of Clone or Snapshot, this field allows overriding the access modes for the point-in-time (PiT) volume. The default is to use the access modes from the source PVC.

capacity When using a copyMethod of Clone or Snapshot, this allows overriding the capacity of the PiT volume. The default is to use the capacity of the source volume.

copyMethod This specifies the method used to create a PiT copy of the source volume. Valid values are:

- **Clone** - Create a new volume by cloning the source PVC (i.e., use the source PVC as the volumeSource for the new volume.
- **None** - Do no create a PiT copy. The Scribe data mover will directly use the source PVC.
- **Snapshot** - Create a VolumeSnapshot of the source PVC, then use that snapshot to create the new volume. This option should be used for CSI drivers that support snapshots but not cloning.

storageClassName This specifies the name of the StorageClass to use when creating the PiT volume. The default is to use the same StorageClass as the source volume.

volumeSnapshotClassName When using a copyMethod of Snapshot, this specifies the name of the VolumeSnapshot-Class to use. If not specified, the cluster default will be used.

address This specifies the address of the replication destination's ssh server. It can be taken directly from the ReplicationDestination's `.status.rsync.address` field.

sshKeys This is the name of a Secret that contains the ssh keys for authenticating the connection with the destination. If not provided, the source keys will be automatically generated and corresponding destination keys will be placed in a new Secret. The name of that new Secret will be placed in .status.rsync.sshKeys.

path This determines the path within the destination volume where the data should be written. In order to create a replica of the source volume, this should be left as the default of `/`.

port This determines the TCP port number that is used to connect via ssh. The default is 22.

sshUser This is the username to use when connecting to the destination. The default value is "root".

For a concrete example, see the *database synchronization example*.

There are two different replication methods built into Scribe. Choose the method that best fits your use-case:

Rclone replication Use Rclone-based replication for multi-way (1:many) scenarios such as distributing data to edge clusters from a central site.

Restic backup Create a Restic-based backup of the data in a PersistentVolume.

Rsync replication Use Rsync-based replication for 1:1 replication of volumes in scenarios such as disaster recovery, mirroring to a test environment, or sending data to a remote site for processing.

2.6 Triggers

Scribe *supports several types of triggers* to specify when to schedule the replication.

2.7 Metrics

Scribe *exposes a number of metrics* that permit monitoring the status of replication relationships via Prometheus.

CHAPTER

THREE

ENHANCEMENT PROPOSALS

3.1 A case for Scribe

Contents

- *A case for Scribe*
  - *Motivation*
  - *Use cases*
  - *Proposed solution*
  - *Initial implementation*

3.1.1 Motivation

As Kubernetes is used in an increasing number of critical roles, businesses are in need of strategies for being able to handle disaster recovery. While each business has its own requirements and budget, there are common building blocks employed across many DR configurations. One such building block is asynchronous replication of storage (PV/PVC) data between clusters.

While some storage systems natively support async replication (e.g, Ceph's RBD or products from Dell/EMC and NetApp), there are many that lack this capability, such as Ceph's cephfs or storage provided by the various cloud providers. Additionally, it is sometimes advantageous to have different storage systems for the source and destination, making vendor-specific replication schemes unworkable. For example, it can be advantageous to have different storage in the cloud vs. on-prem due to resource or environmental constraints.

This project proposes to create a general method for supporting asynchronous, cross-cluster replication that can work with any storage system supporting a CSI-based storage driver. Given a single configuration interface, the controller would implement replication using the most efficient method available. For example, a simplistic CSI driver without snapshot capabilities should still be supported via a best-effort data copy, but a storage system w/ inbuilt replication capabilities should be able to use those mechanisms for fast, efficient data transfer.

3.1.2 Use cases

While disaster recovery is the most obvious use for asynchronous storage replication, there are a number of different scenarios that could benefit.

Case (1) - Async DR

As an application owner, I'd like to ensure my application's data is replicated off-site to a potentially different secondary cluster in case there is a failure of the main cluster. The remote copy should be crash-consistent such that my application can restart at the remote site.

Once a failure has been repaired, I'd like to be able to "reverse" the synchronization so that my primary site can be brought back in sync when the systems recover.

Case (2) - Off-site analytics

As a data warehouse owner, I'd like to periodically replicate my primary data to one or more secondary locations where it can be accessed, read-only, by a scale-out ML or analytics platform.

Case (3) - Testing w/ production data

As a software developer, I'd like to periodically replicate the data from the production environment into an isolated staging environment for continuous testing with real data prior to deploying application updates.

Case (4) - Application migration

As an application owner, I'd like to migrate my production stateful application to a different storage system (either on the same or different Kubernetes cluster) with minimal downtime. I'd like to have the bulk of the data synchronized in the background, allowing for minimal downtime during the actual switchover.

3.1.3 Proposed solution

Using CustomResources, it should be possible for a user to designate a PersistentVolumeClaim on one cluster (the source) to be replicated to a secondary location (the destination), typically on a different cluster. An operator that watches this CR would then initialize and control the replication process.

As stated above, remote replication should be supported regardless of the capabilities of the underlying storage system. To accomplish this, the Scribe operator would have one or more built-in generic replication methods plus a mechanism to allow offloading the replication directly to the storage system when possible.

Replication by Scribe is solely targeted at replicating PVCs, not objects. However, the source and destination volumes should not need to be of the same volume access mode (e.g., RWO, RWX), StorageClass, or even use the same CSI driver, but they would be expected to be of the same volume mode (e.g., Block, Filesystem).

Potential replication methods

For specific storage systems to be able to optimize, the replication and configuration logic must be modular. The method to use will likely need to be specified by the user as there's no standard Kubernetes method to query for capabilities of CSI drivers or vendor storage systems. When evaluating the replication method, if the operator does not recognize the specified method as one internal to the operator, it would ignore the replication object so that an different (storage system-specific) operator could respond. This permits vendor-specific replication methods without requiring them to exist in the main Scribe codebase.

There are several methods that could be used for replication. From (approximately) least-to-most efficient:

1) Copy of live PVC into another PVC
    - This wouldn't require any advanced capabilities of the CSI driver, potentially not even dynamic provisioning
    - Would not create crash-consistent copies. Volume data would be inconsistent and individual files could be corrupted. (Gluster's georep works like this, so it may have some value)
    - For RWO volumes, the copy process would need to be co-scheduled w/ the primary workload
    - Copy would be via rsync-like delta copy
2) Snapshot-based replication
    - Requires CSI driver to support snapshot
    - Source would be snapshotted, the snapshot would be used to create a new volume that would then be replicated to the remote site
    - Copy would be via rsync-like delta copy
    - Remote site would snapshot after each complete transfer
3) Clone-based replication
    - Requires CSI driver to support clone
    - Source would be cloned directly to create the source for copying
    - Copy would be via rsync-like delta copy
    - Remote site would snapshot after each complete transfer
4) Storage system specific
    - A storage system specific mechanism would need to both set up the relationship and handle the sync.
    - Our main contribution here would be a unifying API to provide a more consistent interface for the user.

Built-in replication

With the exception of the storage system specific method, the other options require the replication to be handled by Scribe, copying the data from the source to the destination volume.

It is desirable for Scribe's replication to be relatively efficient and only transfer data that has changed. As a starting point for development, it should be possible to use a pod running rsync, transferring data over an ssh connection.

3.1.4 Initial implementation

The initial Scribe implementation should be focused on providing a minimal baseline of functionality that provides value. As such, the focus will be providing clone-based replication via an rsync data mover, and this implementation will assume both the source and destination are Kubernetes clusters.

3.2 Configuration and CRDs

This document covers the rationale for how Scribe is configured and the structure of the CustomResourceDefinitions.

Contents

- *Configuration and CRDs*
  - *Representation of relationships*
  - *Proposed CRDs*

3.2.1 Representation of relationships

One of the main interaction points between users and Scribe will be centered around configuring the replication relationships between volumes. When looking at the *use cases* presented in the overview of Scribe, there are several commonalities and differences.

Replication triggers

Depending on the use case, the "trigger" for replication may be different. For example, in the case of asynchronous replication for disaster recovery, it is desirable to have the volume(s) replicated at some predictable frequency (e.g., every five minutes). This bounds the amount of data loss that would be incurred during a failover. Some of the other use cases could benefit from scheduled replication (e.g., every day at 3:00am) such as the case of replicating from production to a testing environment. Still other cases may want the replication to be triggered on-demand or via a webhook since it may be desirable to replicate data once a certain action or processing has completed.

Bi-directional vs. uni-directional

Use cases such as disaster recovery naturally desire the replication to be bi-directional (i.e., reversible) so that once the primary site recovers, it can be brought back into sync and the application transitioned back. However, many of the other use cases only desire uni-directional replication— the primary will always remain so.

Further, when volumes are being actively replicated-to (i.e., they are the secondary), they are not in a usable state. Some storage systems actively block their usage until they are "promoted" to an active state, halting or reversing the replication. At best, even if not blocked, the secondary should not be used while replication is ongoing due to the potential of accessing inconsistent data. This has implications on the representation of the "volume" within a Kubernetes environment. For example, it is assumed that a PV/PVC, if bound, is usable by a pod, so exposing a secondary volume as a PV/PVC pair to the user is likely to cause confusion.

Based on the above, a clean interface for the user is likely to be one where a primary PVC is replicated to a destination location as a uni-directional relationship, and the secondary is not visible as a PVC until a "promotion" action is taken.

The lack of a secondary PVC until promotion is what precludes the bi-directional relationship. Instead, two uni-directional relationships could be created. The second, "reverse" relationship would not initially be active since its source PVC would not exist until a secondary volume is promoted.

3.2.2 Proposed CRDs

Since one of the main objectives in the design is to allow storage system specific replication methods, this must be considered when designing the CRDs that will control replication. In order to accommodate separate release timelines and licensing models, it is also desirable for those replication methods to be external to the main Scribe operator. Only a baseline, general replication method needs to be directly integrated.

To achieve the desired flexibility, the CRDs can be structured similar to the Kubernetes StorageClass object which defines a "provisioner" and permits a set of provisioner-specific parameters passed as an arbitrary set of key/value strings.

With the above considerations in mind, the primary side of the replication relationship could be defined as:

Listing 1: CRD defining the source volume to replicate

```
apiVersion: scribe/v1alpha1
kind: Source
metadata:
  name: myVolMirror
  namespace: myNamespace
spec:
  # Source PVC to replicate
  source: my-pvc
  # When/how often to replicate
  trigger:
    # Cronspec for mirroring frequency or schedule
    schedule: "*/10 * * * *"
  # Method of replication. Either built-in "rsync" or an external method
  # (e.g., "ceph.io/rbd-async")
  replicationMethod: Rsync
  # Method-specific configuration parameters
  parameters:  # map[string]string
    param1: value2
status:
  # Method-specific status
  methodStatus:  # map[string]string
    status1: value2
  conditions:  # general conditions
```

The secondary side is configured similarly to the primary, but without the trigger specification:

Listing 2: CRD defining the replication destination

```
apiVersion: scribe/v1alpha1
kind: Destination
metadata:
  name: myVolMirror
  namespace: myNamespace
spec:
  replicationMethod: Rsync
```

(continues on next page)

(continued from previous page)

```
  parameters:
    param1: value2
status:
  methodStatus:
    status1: value2
  conditions:
```

3.3 Rsync-based data mover

This document covers the design of the rsync-based data mover.

Contents

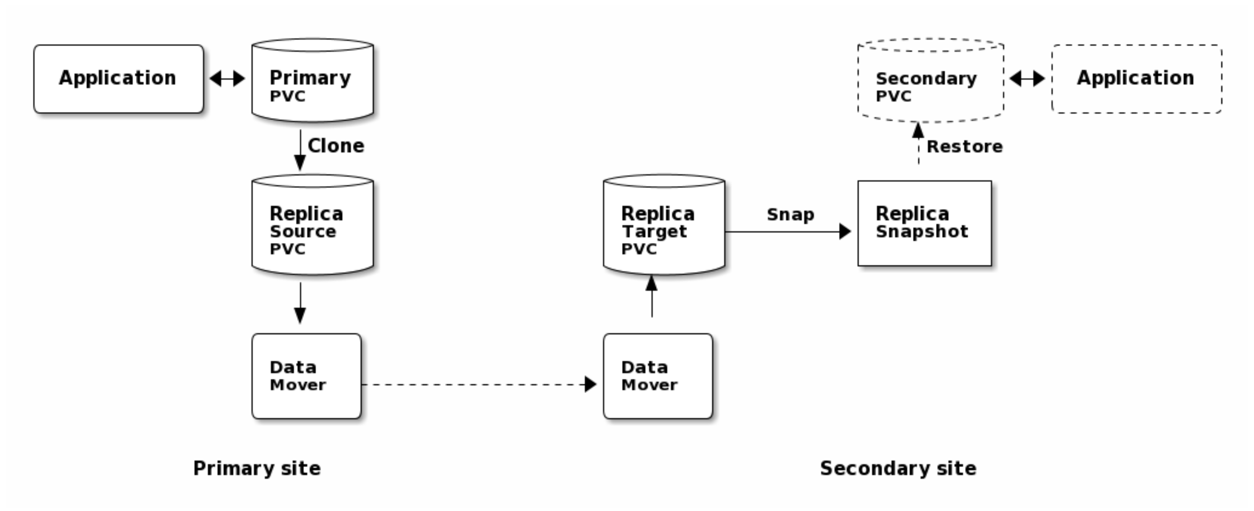
- *Rsync-based data mover*
  - *Overview*
  - *Replication flow*
  - *Setup*

3.3.1 Overview

To meet the goal of being able to replicate arbitrary volumes, Scribe must have a built-in, baseline replication method. Rsync is a well-known and reasonably efficient method to synchronize file data between two locations. It supports both data compression as well as differential transfer of data. Further, its support of ssh as a transport allows the data to be transferred securely, authenticating both sides of the communication.

3.3.2 Replication flow

1) A point-in-time image of the primary application's data is captured by cloning the application's PVC. This new "replica source" PVC serves as the source for one iteration of replication.
2) A data mover pod is started on the primary side that syncs the data to a data mover pod on the secondary side. The data is transferred via rsync (running in the mover pods) over ssh. A shared set of keys allows mutual authentication of the data movers.
3) After successfully replicating the data to a target PVC on the secondary, the secondary PVC is snapshotted to create a point-in-time copy that is identical to the image captured in step 1.
4) The process can be repeated, beginning again with step 1. Subsequent transfers will only need to transfer changed data since the target PVC on the secondary is re-used with each iteration.



Failover

When the primary application has failed, the secondary site should take over. In order to start the application on the secondary site, the synchronized data must be made accessible in a PVC.

As a part of bringing up the application, its PVC is created from the most recent "replica snapshot". This promotion of the snapshot to a PVC is only necessary during failover. The majority of the time (i.e., while the primary is properly functioning), old replica snapshots will be replaced with a new snapshot at the end of each round of synchronization.

Resynchronization

After the primary site recovers, its data needs to be brought back in sync with the secondary (currently the active site). This is accomplished by having a reverse synchronization path identical to the flow above but with data flowing from the secondary site to the primary.

The replication from secondary to primary can be configured a priori, with the data movement only happening after failover. For example, the reverse replication would use "Secondary PVC" from the above diagram as the volume to replicate. In normal operation, this volume would not exist, idling the reverse path. Once the secondary site becomes the active site, that PVC would exist, allowing the reverse synchronization to flow, resulting in replicated snapshots on the primary side. These can later be used to recreate the "Primary PVC", thus restoring the application to the primary site.

3.3.3 Setup

As a part of configuring the rsync replication, a CustomResource needs to be created on both the source and destination cluster. This configuration must contain:

Connection information Synchronization is handled via a push model— the source creates the connection to the destination cluster. As such, the source must be provided with the host/port information necessary to contact the destination.

Authentication credentials An ssh connection is used to carry the rsync traffic. This connection is made via shared public keys between both sides of the connection. This allows the destination (ssh server) to authenticate the source (client) as well as allowing the source to validate the destination (by checking an associated ssh host key).

In order to make the configuration as easy as possible, the destination CR should be created first. When reconciling, the operator will generate the appropriate ssh keys and connection information in a Kubernetes Secret, placing a reference to that secret in the Destination CR's `status.methodStatus` map.

This Secret will then be copied to the source cluster and referenced in `spec.parameters` when creating the Source CR.

3.4 Restic-based data mover

Enhancement status

Status: Proposed

This is a proposal to add Restic as an additional data mover within Scribe. Restic is a data backup utility that copies the data to an object store (among other options).

While the main purpose of Scribe is to perform asynchronous data replication, there are some use cases that are more "backup oriented" but that don't require a full backup application (such as Velero). For example, some users may deploy and version control their application via GitOps techniques. These users may be looking for a simple method that allows preserving (off-cluster) snapshots of their storage so that it can be restored if necessary.

3.4.1 Considerations

The ReplicationSource and ReplicationDestination CRs of Scribe would correspond to the `backup` and `restore` operations, respectively, of Restic. Furthermore, there are repository maintenance operations that need to be addressed. For example, Restic manages the retention of old backups (via its `forget` operation) as well as freeing objects that are no longer used (via its `prune` operation).

While both Restic and Rclone read/write to object storage, their strengths are significantly different. The Rclone data mover is primarily designed for managing 1-to-many replication relationships, using the object store as an intermediary. On each sync, Rclone updates the object bucket to be identical to the current version of the source volume, making no attempt to preserve previous images. This works well for replication scenarios, but it may not be desirable when protection from accidental data deletion is desired. On the other hand, Restic is well suited for maintaining a series of historical versions in an efficient manner, but it is not designed for syncing data. The restore operation makes no allowance for small delta transfers.

3.4.2 CRD for Restic mover

In the normal case, the expected usage would be to have a ReplicationSource that controls the periodic backups of the data. It would use the same "common volume options" that Rsync and Rclone use to create a point-in-time image prior to copying the data.

Backup

Given that in the normal case, only the ReplicationSource would be used, the repository maintenance options should be set there.

```yaml
---
apiVersion: scribe/v1alpha1
kind: ReplicationSource
metadata:
  name: source
  namespace: myns
spec:
  sourcePVC: pvcname
  trigger:
    schedule: "0 * * * *"  # hourly backups
  restic:
    ### Standard volume options
    # ReplicationSourceVolumeOptions

    ### Restic-specific options
    pruneIntervalDays:  # How often to prune the repository (*int)
    repository:  # Secret name containing repository info (string)
    # Retention policy for the backups
    retain:
      last:  # Keep the last n snapshots (*int)
      hourly:  # Keep n hourly (*int)
      daily:  # Keep n daily (*int)
      weekly:  # Keep n weekly (*int)
      monthly:  # Keep n monthly (*int)
      yearly:  # Keep n yearly (*int)
      within: # Keep all within this duration (e.g., "3w15h") (*string)
```

The `.spec.restic.repository` Secret reference in the above structure refers to a Secret in the same Namespace of the following format. The Secret's "keys" correspond directly to the environment variables supported by Restic.

```yaml
---
apiVersion: v1
kind: Secret
metadata:
  name: resticRepo
type: Opaque
data:
  # The repository url
  RESTIC_REPOSITORY: s3:s3.amazonaws.com/bucket_name
  # The repository encryption key
  RESTIC_PASSWORD: XXXXX
  # ENV vars specific to the back end
  # https://restic.readthedocs.io/en/stable/030_preparing_a_new_repo.html
  AWS_ACCESS_KEY_ID: (access key)
  AWS_SECRET_ACCESS_KEY: (secret key)
```

Restore

For now, with Scribe, the intention is to only support restoring the latest version of the backed-up data. For retrieving previous backups (that are still retained), Restic can be directly run against the repository, using the same information as in the Secret, above.

Restore would be handled by the following ReplicationDestination:

```yaml
---
apiVersion: scribe.backube/v1alpha1
kind: ReplicationDestination
metadata:
  name: dest-sample
spec:
  trigger:
    schedule: "30 * * * *"
  restic:
    ### Standard volume options
    # ReplicationDestinationVolumeOptions

    ### Restic-specific options
    repository:  # Secret name containing repository info (string)
```

There are comparatively few configuration options for Restore.

3.4.3 Open issues

The following items are open questions:

- Should ReplicationDestination support scheduling or should it be based on a single restore (i.e., it "syncs" once then never again)? This could also be simulated by having an arbitrarily long schedule since the 1st sync is immediate.
- Are Restic operations fast enough to make this viable?
  - The `prune` operation is documented as being rather slow
  - How long does it take to scan the storage to determine what needs to be backed up?
- Restic uses locks on the repository. Does the lack of concurrency present a problem for us? (Some can be done w/o locks… which ones?)
- What is the right way to expose `prune`?
  - It is the method for freeing space in the repo, but may be too expensive to run frequently

Asynchronous volume replication for Kubernetes CSI storage

Scribe is a Kubernetes operator that performs asynchronous replication of persistent volumes within, or across, clusters. The replication provided by Scribe is independent of the storage system. This allows replication to and from storage types that don't normally support remote replication. Additionally, it can replicate across different types (and vendors) of storage.

The project is still in the early stages, but feel free to give it a try.

To get started, see the *installation instructions*.

Check us out on GitHub https://github.com/backube/scribe