VolSync

The VolSync authors

May 12, 2022

CONTENTS

1 Installation **1**
- 1.1 Development . . . 1
- 1.2 RBAC permissions . . . 2
- 1.3 Kubernetes & OpenShift . . . 4

2 Usage **9**
- 2.1 Triggers . . . 9
- 2.2 Metrics & monitoring . . . 11
- 2.3 Rclone-based replication . . . 16
- 2.4 Restic-based backup . . . 24
- 2.5 Rsync-based replication . . . 32
- 2.6 VolSync CLI / kubectl plugin . . . 46
- 2.7 Triggers . . . 56
- 2.8 Metrics . . . 56

3 Enhancement proposals **57**
- 3.1 A case for VolSync . . . 57
- 3.2 Configuration and CRDs . . . 60
- 3.3 Rsync-based data mover . . . 62
- 3.4 Restic-based data mover . . . 64

CHAPTER

ONE

INSTALLATION

1.1 Development

If you are developing VolSync, there are a few options to get up-and-running. All of these options will assume the use of a local kind cluster.

Once you have kind installed, there is a convenient script in the `hack/` directory that will get a cluster running and properly configured.

```
$ ./hack/setup-kind-cluster.sh
```

Once you have a cluster running, you can either build and deploy the operator in the cluster, or you can run the operator locally against the cluster.

Build & deploy

Run locally

The below command will build all containers (operator and movers) from the local source, inject them into the running kind cluster, then use the local helm templates to start the operator.

```
# Build, inject, and run
$ ./hack/run-in-kind.sh
```

The below commands will run the operator binary locally, but the mover containers will be pulled from Quay (`latest` tag). This option is good when developing the operator code since it permits fast rebuilds and easy access to the operator logs.

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

# Run the operator locally
$ make run
```

If you will be working with the Rclone or Restic movers, you may want to deploy Minio in the kind cluster to act as an object repository. It can be started via:

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

1.2 RBAC permissions

Once the VolSync operator has been installed, it is ready for use in the cluster, but only those with cluster administrator privileges have permission to use it.

In order for the operator to be used, it is necessary to have the ability to access VolSync's ReplicationSource and ReplicationDestination custom resource objects. It is recommended that users be allowed to manage data replication within the namespaces that they are assigned. This enables "self-service" data protection for the cluster's users.

The below RBAC rules give users access to VolSync's CRs within the namespaces that they manage. It also grants access to VolumeSnapshot objects so that users can easily "promote" the latest destination snapshot, if necessary, during recovery/fail-over.

Listing 1: volsync-rbac.yaml

```yaml
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: volsync-edit
  labels:
    # Grant access to namespace admins
    rbac.authorization.k8s.io/aggregate-to-admin: "true"
    # Grant access to namespace editors
    rbac.authorization.k8s.io/aggregate-to-edit: "true"
rules:
  # Give users full control of ReplicationSource and ReplicationDestination
  # objects so they can manage data replication
  - apiGroups:
      - volsync.backube
    resources:
      - replicationdestinations
      - replicationsources
    verbs:
      - create
      - delete
      - deletecollection
      - get
      - list
      - patch
      - update
      - watch
  - apiGroups:
      - volsync.backube
    resources:
      - replicationdestinations/status
      - replicationsources/status
    verbs:
      - get
      - list
      - watch
  # Give users the ability to view VolumeSnapshots so they can "promote" the
  # destination snapshots into usable PVCs
  - apiGroups:
```

(continues on next page)

(continued from previous page)

```
    - snapshot.storage.k8s.io
  resources:
    - volumesnapshots
    - volumesnapshots/status
  verbs:
    - get
    - list
    - watch

---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: volsync-view
  labels:
    # Grant access to namespace viewers
    rbac.authorization.k8s.io/aggregate-to-view: "true"
rules:
  # Give users read access to ReplicationSource and ReplicationDestination
  # objects so they can monitor data replication
  - apiGroups:
      - volsync.backube
    resources:
      - replicationdestinations
      - replicationsources
      - replicationdestinations/status
      - replicationsources/status
    verbs:
      - get
      - list
      - watch
  # Give users the ability to monitor (destination) VolumeSnapshots
  - apiGroups:
      - snapshot.storage.k8s.io
    resources:
      - volumesnapshots
      - volumesnapshots/status
    verbs:
      - get
      - list
      - watch
```

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

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

There are several methods for installing VolSync. Choose the option that relates to your situation.

Warning: VolSync 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.

1.3 Kubernetes & OpenShift

The recommended method for deploying VolSync is via its 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 volsync-system volsync backube/volsync
```

Verify VolSync is running by checking the output of `kubectl get deploy`:

```
$ kubectl -n volsync-system get deploy/volsync
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
volsync   1/1     1            1           60s
```

1.3.1 Configuring CSI storage

To make the most of VolSync's capabilities, it's important that the volumes being replicated are using CSI-based storage drivers and that volume snapshotting is properly configured.

The currently configured StorageClasses can be viewed via:

```
$ kubectl get storageclasses
```

And the VolumeSnapshotClasses can be viewed via:

```
$ kubectl get volumesnapshotclasses
```

StorageClasses that carry the `storageclass.kubernetes.io/is-default-class: "true"` and VolumeSnapshotClasses that carry the `snapshot.storage.kubernetes.io/is-default-class: "true"` annotations are marked as the defaults on the cluster, meaning that if the class is not specified, these defaults will be used. However, it is not necessary to set or modify the default on your cluster since the classes can be specified directly in the ReplicationSource and ReplicationDestination objects used by VolSync.

Below are examples of configured CSI storage on a few different cloud platforms. Your configuration may be different.

AWS

Azure

GCP

vSphere

The EBS CSI driver on AWS-based clusters is usually named `gp2-csi` or `gp3-csi`.

```
# List StorageClasses
$ kubectl get storageclasses
NAME            PROVISIONER             RECLAIMPOLICY   VOLUMEBINDINGMODE      ␣
↪ALLOWVOLUMEEXPANSION   AGE
```

(continues on next page)

(continued from previous page)

```
gp2 (default)   kubernetes.io/aws-ebs   Delete          WaitForFirstConsumer   true      ␣
↪             25m
gp2-csi         ebs.csi.aws.com         Delete          WaitForFirstConsumer   true      ␣
↪             25m
gp3-csi         ebs.csi.aws.com         Delete          WaitForFirstConsumer   true      ␣
↪             25m


# View details of the gp2-csi SC
$ kubectl get storageclass/gp2-csi -oyaml
allowVolumeExpansion: true
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
   creationTimestamp: "2022-02-08T14:03:20Z"
   name: gp2-csi
   resourceVersion: "5288"
   uid: 24d2cee6-1346-4c3e-8742-39dec08e3e50
parameters:
   encrypted: "true"
   type: gp2
provisioner: ebs.csi.aws.com
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
```

The CSI driver on Azure-based clusters is usually named `managed-csi`.

```
# List StorageClasses
$ kubectl get storageclasses
NAME                        PROVISIONER                RECLAIMPOLICY   VOLUMEBINDINGMODE␣
↪     ALLOWVOLUMEEXPANSION   AGE
managed-csi                 disk.csi.azure.com         Delete          ␣
↪WaitForFirstConsumer   true                   45m
managed-premium (default)   kubernetes.io/azure-disk   Delete          ␣
↪WaitForFirstConsumer   true                   46m

# View details of the managed-csi SC
$ kubectl get storageclass/managed-csi -oyaml
allowVolumeExpansion: true
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
   creationTimestamp: "2022-02-08T14:57:23Z"
   name: managed-csi
   resourceVersion: "5853"
   uid: 3aeba0d1-6c52-481c-9dc1-786ae84a2f7b
parameters:
   skuname: Premium_LRS
provisioner: disk.csi.azure.com
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
```

The CSI driver on GCP-based clusters is usually named `standard-csi`.

```
# List StorageClasses
$ kubectl get storageclasses
NAME                 PROVISIONER             RECLAIMPOLICY   VOLUMEBINDINGMODE     ␣
↪ALLOWVOLUMEEXPANSION   AGE
standard (default)   kubernetes.io/gce-pd    Delete          WaitForFirstConsumer   true␣
↪                  15m
standard-csi         pd.csi.storage.gke.io   Delete          WaitForFirstConsumer   true␣
↪                  15m


# View details of the standard-csi SC
$ kubectl get storageclass/standard-csi -oyaml
allowVolumeExpansion: true
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
   creationTimestamp: "2022-02-08T13:24:53Z"
   name: standard-csi
   resourceVersion: "5976"
   uid: 066a43fc-798f-49a7-b62a-0350e8946364
parameters:
   replication-type: none
   type: pd-standard
provisioner: pd.csi.storage.gke.io
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
```

The CSI driver on vSphere-based clusters is usually named `thin-csi`.

```
# List StorageClasses
$ kubectl get storageclasses
NAME             PROVISIONER                 RECLAIMPOLICY   VOLUMEBINDINGMODE     ␣
↪ALLOWVOLUMEEXPANSION   AGE
thin (default)   kubernetes.io/vsphere-volume   Delete          Immediate             ␣
↪false                  20m
thin-csi         csi.vsphere.vmware.com         Delete          WaitForFirstConsumer  ␣
↪true                   18m

# View details of the thin-csi SC
$ kubectl get storageclass/thin-csi -oyaml
allowVolumeExpansion: true
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
   creationTimestamp: "2022-02-08T16:48:52Z"
   name: thin-csi
   resourceVersion: "9789"
   uid: 80d45374-8447-47eb-950c-2568af070d6e
parameters:
   StoragePolicyName: openshift-storage-policy-ci-ln-54d2r5t-c1627-jvkws
provisioner: csi.vsphere.vmware.com
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
```

You should also verify the presence of a corresponding VolumeSnapshotClass. Note that the name of the SC and VSC do not need to be the same, but the provisioner/driver should be.

AWS

Azure

GCP

vSphere

```
# List VolumeSnapshotClasses
$ kubectl get volumesnapshotclasses
NAME          DRIVER            DELETIONPOLICY   AGE
csi-aws-vsc   ebs.csi.aws.com   Delete           23m


# View details of the csi-aws-vsc VSC
$ kubectl get volumesnapshotclass/csi-aws-vsc -oyaml
apiVersion: snapshot.storage.k8s.io/v1
deletionPolicy: Delete
driver: ebs.csi.aws.com
kind: VolumeSnapshotClass
metadata:
   annotations:
      snapshot.storage.kubernetes.io/is-default-class: "true"
   creationTimestamp: "2022-02-08T14:03:20Z"
   generation: 1
   name: csi-aws-vsc
   resourceVersion: "5301"
   uid: d990af7b-d2ae-4a49-8cfe-fd5ae93902df
```

Important: The AWS EBS CSI driver does not support volume cloning. When configuring replication with VolSync, be sure to choose a `copyMethod` of `Snapshot` for the source volume. Choosing `Clone` will not work.

```
# List VolumeSnapshotClasses
$ kubectl get volumesnapshotclasses
NAME                DRIVER               DELETIONPOLICY   AGE
csi-azuredisk-vsc   disk.csi.azure.com   Delete           48m

# View details of the csi-azuredisk-vsc VSC
$ kubectl get volumesnapshotclass/csi-azuredisk-vsc -oyaml
apiVersion: snapshot.storage.k8s.io/v1
deletionPolicy: Delete
driver: disk.csi.azure.com
kind: VolumeSnapshotClass
metadata:
   annotations:
      snapshot.storage.kubernetes.io/is-default-class: "true"
   creationTimestamp: "2022-02-08T14:57:23Z"
   generation: 1
   name: csi-azuredisk-vsc
   resourceVersion: "5847"
   uid: 1d105f8c-4e49-48e1-8ead-927f90f4bb2e
```

(continues on next page)

(continued from previous page)

```
parameters:
   incremental: "true"
```

```
# List VolumeSnapshotClasses
$ kubectl get volumesnapshotclasses
NAME             DRIVER                  DELETIONPOLICY   AGE
csi-gce-pd-vsc   pd.csi.storage.gke.io   Delete           17m


# View details of the csi-gce-pd-vsc VSC
$ kubectl get volumesnapshotclass/csi-gce-pd-vsc -oyaml
apiVersion: snapshot.storage.k8s.io/v1
deletionPolicy: Delete
driver: pd.csi.storage.gke.io
kind: VolumeSnapshotClass
metadata:
   annotations:
      snapshot.storage.kubernetes.io/is-default-class: "true"
   creationTimestamp: "2022-02-08T13:24:53Z"
   generation: 1
   name: csi-gce-pd-vsc
   resourceVersion: "5981"
   uid: 886de96d-820c-403b-8570-fcfb37939532
```

At this time (Feb 2022), volume snapshotting is an alpha feature in the vSphere CSI driver and not enabled by default. If you are interested in trying it out, please consult VMware's documentation.

Next, consider *granting users access to VolSync's custom resources* so that they can manage their own data replication.

Continue to the *usage docs*.

CHAPTER

TWO

USAGE

2.1 Triggers

There are three types of triggers in volsync:

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: volsync.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, VolSync 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 VolSync for each replication object (source or destination):

volsync_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.

volsync_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).

volsync_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 VolSync 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 volsync

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

OpenShift

The following steps start a simple Prometheus instance to scrape metrics from VolSync. 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 volsync-system
$ kubectl -n volsync-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: volsync-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: volsync-controller
  resources:
    requests:
      memory: 400Mi
```

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 VolSync

The metrics port for VolSync 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 `volsync-metrics-reader` ClusterRole. This can be accomplished by (substitute in the namespace & SA name of the Prometheus server):

```
$ kubectl create clusterrolebinding metrics --clusterrole=volsync-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 VolSync.

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

Listing 4: VolSync ServiceMonitor

```yaml
---
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: volsync-monitor
  namespace: volsync-system
  labels:
    control-plane: volsync-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: volsync-controller
```

2.3 Rclone-based replication

2.3.1 Rclone Database Example

The following example will use the Rclone replication method to replicate a sample MySQL database.

First, create the source namespace and 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/
↪rclone.conf -n source
$ kubectl create -f examples/rclone/volsync_v1alpha1_replicationsource.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` VolSync 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/
↪rclone.conf -n dest
$ kubectl create -f examples/rclone/volsync_v1alpha1_replicationdestination.yaml -n dest
```

Once the `ReplicationDestination` is deployed, VolSync will create a Rclone data mover job on the destination side. At the end of the each successful iteration, the `ReplicationDestination` is updated with the latest 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 that the next synchronization cycle does not start while the following steps are being completed or VolSync may replace the existing snapshot with a new one before the database starts.

```
# Get the latest snapshot name
$ kubectl get replicationdestination database-destination -n dest --template={{.status.
↪latestImage.name}}
# Substitute that name into the database PVC template
$ sed -i 's/snapshotToReplace/volsync-dest-database-destination-20201203174504/g'␣
↪examples/destination-database/mysql-pvc.yaml
# Start the database
$ 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;
+--------------------+
```

(continues on next page)

(continued from previous page)

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

> exit
$ exit
```

2.3.2 Understanding `rclone-secret`

The Rclone Secret provides the configuration details to locate and access the intermediary storage system. It is mounted as a secret on the Rclone data mover pod and provided to the Rclone executable.

```
[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 the Rclone documentation on how to create an `rclone.conf` configuration file.

Deploy `rclone-secret`

Assuming the above example is placed in a local file, `rclone.conf`, the Secret can be created via:

```
# Create the secret (remember to pass the correct namespace name)
$ kubectl create -n source secret generic rclone-secret --from-file=rclone.conf=rclone.
↪conf
$ kubectl get -n source secrets
NAME                  TYPE                                  DATA   AGE
default-token-g9vdx   kubernetes.io/service-account-token   3      20s
rclone-secret         Opaque                                1      17s
```

This Secret should be created on both the source and the destination locations.

Contents

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, VolSync synchronizes data from a ReplicationSource to a ReplicationDestination using Rclone via an intermediary object storage location like AWS S3.

The Rclone method uses a "push" and "pull" model for the data replication. This requires a schedule or other trigger on both the source and destination sides to trigger the replication iterations.

During each synchronization iteration:

- A point-in-time (PiT) copy of the source volume is created using CSI drivers. This copy will be used as the source data.
- The copy is attached to an Rclone data mover job pod which uses the contents of the `rclone-secret` to connect to the intermediary object storage target (e.g., AWS S3).
- The source pod uses `rclone sync` to copy the data to S3.
- On the destination side, a corresponding Rclone mover pod syncs the data from the intermediate object storage into a volume on the destination.
- At the conclusion of the transfer, the destination creates a snapshot copy to preserve a point-in-time copy of the incoming source data.

VolSync is configured via two CustomResources (CRs), one on the source side and one on the destination side of the replication relationship. While there should only be one ReplicationSource pushing data to the intermediate storage, there may be an arbitrary number of ReplicationDestination instances syncing data from the intermediate storage to destination clusters. This enables the model of high fan-out data distribution.

2.3.3 Source configuration

An example source configuration is shown below:

```
---
apiVersion: volsync.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: database-source
  namespace: source
spec:
  # The PVC to sync
  sourcePVC: mysql-pv-claim
  trigger:
    # Synchronize every 6 minutes
    schedule: "*/6 * * * *"
  rclone:
    # The configuration section of the rclone config file to use
    rcloneConfigSection: "aws-s3-bucket"
    # The path to the object bucket
    rcloneDestPath: "volsync-test-bucket"
    # Secret holding the rclone configuration
    rcloneConfig: "rclone-secret"
    # Method used to generate the PiT copy
    copyMethod: Snapshot
    # The StorageClass to use when creating the PiT copy (same as source PVC if omitted)
    storageClassName: my-sc-name
    # The VSC to use if the copy method is Snapshot (default if omitted)
    volumeSnapshotClassName: my-vsc-name
```

Since the `copyMethod` specified above is `Snapshot`, the Rclone data mover creates a `VolumeSnapshot` of the source pvc `mysql-pv-claim`. Then it converts this snapshot back into a PVC. If `copyMethod: Clone` were used, the temporary, point-in-time copy would be created by cloning the source PVC to a new PVC directly. This is more efficient, but it is not supported by all CSI drivers.

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

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

```
---
apiVersion:  volsync.backube/v1alpha1
kind:         ReplicationSource
#  ... omitted ...
spec:
  rclone:
    copyMethod:               Snapshot
    rcloneConfig:             rclone-secret
    rcloneConfigSection:      aws-s3-bucket
    rcloneDestPath:           volsync-test-bucket
    storageClassName:         my-sc-name
```

(continues on next page)

(continued from previous page)

```
    volumeSnapshotClassName:  my-vsc-name
  sourcePVC:              mysql-pv-claim
  trigger:
    schedule:  "*/6 * * * *"
  status:
    conditions:
      lastTransitionTime:  2021-01-18T21:50:59Z
      message:               Reconcile complete
      reason:                ReconcileComplete
      status:                True
      type:                  Reconciled
    nextSyncTime:          2021-01-18T22:00:00Z
```

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.
- **Direct** - Do no create a PiT copy. The VolSync 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 identify the configuration section within `rclone.conf` to use.

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

rcloneConfig This specifies the name of a secret to be used to retrieve the Rclone configuration. The *content of the Secret* is an `rclone.conf` file.

2.3.4 Destination configuration

An example destination configuration is shown here:

```yaml
---
apiVersion: volsync.backube/v1alpha1
kind: ReplicationDestination
metadata:
  name: database-destination
  namespace: dest
spec:
  trigger:
    # Every 6 minutes, offset by 3 minutes
    schedule: "3,9,15,21,27,33,39,45,51,57 * * * *"
  rclone:
    rcloneConfigSection: "aws-s3-bucket"
    rcloneDestPath: "volsync-test-bucket"
    rcloneConfig: "rclone-secret"
    copyMethod: Snapshot
    accessModes: [ReadWriteOnce]
    capacity: 10Gi
    storageClassName: my-sc
    volumeSnapshotClassName: my-vsc
```

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. It is important that the schedule for the destinations are offset from that of the source to allow the source to finish pushing updates for an iteration prior to the the destination attempting to pull them.

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

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

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

```yaml
---
API Version:  volsync.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:           volsync-test-bucket
```

(continues on next page)

(continued from previous page)

```
    Storage Class Name:          my-sc
    Volume Snapshot Class Name:  my-vsc
  Status:
    Conditions:
      Last Transition Time:  2021-01-19T22:16:02Z
      Message:               Reconcile complete
      Reason:                ReconcileComplete
      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:       volsync-dest-database-destination-20210119221601
```

In the above example,

- `Rclone Dest Path` indicates the intermediary storage system from where data will be transferred to the destination site. In the above example, the intermediary storage system is an 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 `Direct`, this will be the PVC that is used as the destination by VolSync.

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

capacity When VolSync 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:

- **Direct** - 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 VolSync automatically provision the destination volume (using capacity, accessModes, etc.), the name of a pre-existing PVC may be specified here.

storageClassName When VolSync 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. If omitted, the system default VolumeSnapshotClass will be used.

rcloneConfigSection This is used to identify the configuration section 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 an `rclone.conf` file with the configuration and credentials for the object target.

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 |
```

(continues on next page)

(continued from previous page)

```
| 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: volsync.backube/v1alpha1
kind: ReplicationSource
metadata:
   name: database-source
   namespace: source
spec:
   sourcePVC: mysql-pv-claim
   trigger:
      schedule: "*/30 * * * *"
```

(continues on next page)

(continued from previous page)

```
  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 examples/restic/source-restic/source-restic.yaml -n source
$ kubectl create -f examples/restic/volsync_v1alpha1_replicationsource.yaml -n source
```

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

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

apiVersion: volsync.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 VolSync 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  volsync                  /data
------------------------------------------------------------
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/restic/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:

```
---
apiVersion: volsync.backube/v1alpha1
kind: ReplicationDestination
metadata:
  name: database-destination
spec:
  trigger:
    manual: restore
  restic:
    destinationPVC: mysql-pv-claim
    repository: restic-config
    copyMethod: Direct
```

```
$ kubectl -n dest create -f examples/restic/volsync_v1alpha1_replicationdestination.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
> show databases;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| synced             |
| sys                |
+--------------------+
5 rows in set (0.00 sec)

> exit
$ exit
```

Contents

Backing up using Restic

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

VolSync 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 VolSync's other movers because it is not meant for synchronizing data between clusters. This mover is specifically designed 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.

```
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
  # 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). The key names in this configuration Secret directly correspond to the environment variable names supported by Restic.

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: volsync.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
```

(continues on next page)

(continued from previous page)

```
    monthly: 2
    yearly: 1
  # Clone the source volume prior to taking a backup to ensure a
  # point-in-time image.
  copyMethod: Clone
  # The StorageClass to use when creating the PiT copy (same as source PVC if omitted)
  #storageClassName: my-sc-name
  # The VSC to use if the copy method is Snapshot (default if omitted)
  #volumeSnapshotClassName: my-vsc-name
```

Backup options

There are a number of additional configuration options not shown in the above example. VolSync'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.
- **Direct** - Do no create a PiT copy. The VolSync 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.

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. Shared backup repositories are not currently supported.

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:

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

Restore the data into `datavol`:

```
---
apiVersion: volsync.backube/v1alpha1
kind: ReplicationDestination
metadata:
  name: datavol-dest
spec:
  trigger:
    manual: restore-once
  restic:
    repository: restic-repo
    # Use an existing PVC, don't provision a new one
    destinationPVC: datavol
    copyMethod: Direct
```

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 `Direct` 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.

The example, shown above, will restore the data from the most recent backup. To restore an older version of the data, the `previous` and `restoreAsOf` fields can be used. See below for more information on their meaning.

Restore options

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

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

capacity When VolSync 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:

- **Direct** - 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 VolSync automatically provision the destination volume (using capacity, accessModes, etc.), the name of a pre-existing PVC may be specified here.

storageClassName When VolSync 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. If omitted, the system default VolumeSnapshotClass 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.

previous Non-negative integer which specifies an offset for how many snapshots ago we want to restore from. When `restoreAsOf` is provided, the behavior is the same, however the starting snapshot considered will be the first one taken before `restoreAsOf`.

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.

restoreAsOf An RFC-3339 timestamp which specifies an upper-limit on the snapshots that we should be looking through when preparing to restore. Snapshots made after this timestamp will not be considered. Note: though this is an RFC-3339 timestamp, Kubernetes will only accept ones with the day and hour fields separated by a `T`. E.g, `2022-08-10T20:01:03-04:00` will work but `2022-08-10 20:01:03-04:00` will fail.

2.5 Rsync-based replication

2.5.1 Rsync Database Example

The following example will use the Rsync replication method to periodically replicate a MySQL database.

First, create the destination Namespace and deploy the ReplicationDestination object.

```
$ kubectl create ns dest
$ kubectl create -n dest -f dest.yaml
```

The ReplicationDestination has the following configuration:

```
---
apiVersion: volsync.backube/v1alpha1
kind: ReplicationDestination
metadata:
  name: database-destination
spec:
  rsync:
    serviceType: LoadBalancer
    copyMethod: Snapshot
    capacity: 2Gi
    accessModes: [ReadWriteOnce]
    storageClassName: standard-csi
    volumeSnapshotClassName: csi-gce-pd-vsc
```

A LoadBalancer Service is created by VolSync which will be used by the ReplicationSource to connect to the destination. Record the service IP address as it will be used in the ReplicationSource configuration. *(More information on LoadBalancer vs ClusterIP)*

```
$ kubectl get replicationdestination database-destination -n dest --template={{.status.
↪rsync.address}}
34.133.219.189
```

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 create the ReplicationSource items. First, we need the ssh secret from the destination namespace. *(SSH Secret copying details)*

```
# Retrieve the Secret from the destination cluster
$ kubectl get secret -n dest volsync-rsync-dest-src-database-destination -o yaml > /tmp/
↪secret.yaml

# Remove unnecessary fields
$ vi /tmp/secret.yaml
# ^^^ change the namespace to "source"
# ^^^ remove the owner reference (.metadata.ownerReferences)

# Insert the Secret into the source cluster
$ kubectl create -f /tmp/secret.yaml
```

Using the IP address that relates to the ReplicationDestination that was recorded earlier. Create a ReplicationSource object:

```
---
apiVersion: volsync.backube/v1alpha1
kind: ReplicationSource
```

(continues on next page)

(continued from previous page)

```
metadata:
  name: database-source
  namespace: source
spec:
  sourcePVC: mysql-pv-claim
  trigger:
    # Replicate every 5 minutes
    schedule: "*/5 * * * *"
  rsync:
    # The name of the Secret we just created
    sshKeys: volsync-rsync-dest-src-database-destination
    # The LoadBalancer address from the ReplicationDestination
    address: 34.133.219.189
    copyMethod: Clone
```

Note: You may need to change the `copyMethod` to `Snapshot` and specify both a `storageClassName` and `volumeSnapshotClassName`, depending on your CSI driver's capabilities.

Once the ReplicationSource is created, the initial synchronization should begin. 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:          2020-12-03T16:10:00Z
```

We will modify the source database by creating an additional 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)
```

(continues on next page)

(continued from previous page)

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

During the next synchronization iteration, these updates will be replicated to the destination.

Now the mysql database will be deployed to the destination 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 that the above steps are completed before a new replication cycle starts or the latest snapshot may be replaced before it can be used.

```
$ kubectl get replicationdestination database-destination -n dest --template={{.status.
↪latestImage.name}}
volsync-dest-database-destination-20201203174504

$ sed -i 's/snapshotToReplace/volsync-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 Moving data into Kubernetes w/ Rsync

While VolSync is typically used to replicate data between Kubernetes clusters, it is sometimes necessary to replicate data into a cluster from outside. For example, when containerizing a previously standalone workload, that application's data needs to be moved into the cluster and onto a PVC.

In this configuration, VolSync manages the destination (via a ReplicationDestination object), but instead of having a VolSync ReplicationSource as the sender, it will be an external program that plays that role. It will transmit the data to the destination by initiating the Rsync over SSH connection directly.

The VolSync CLI incorporates this functionality via its `migration` set of sub-commands. For more information and a walk-through of how to perform synchronization of data into a Kubernetes PVC, please see:

- *CLI/kubectl plugin installation*
- *VolSync migration command*

2.5.3 Manual SSH key generation

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

Two key pairs 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
```

(continues on next page)

(continued from previous page)

```
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 *.. |
|    .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 volsync-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 volsync-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. Since the name of a key Secret is being provided in `.spec.rsync.sshKeys`, the operator will use this Secret instead of generating its own and placing it in the `.status`.

```yaml
---
apiVersion: volsync.backube/v1alpha1
kind: ReplicationDestination
metadata:
  name: database-destination
  namespace: dest
spec:
  rsync:
    # ... other fields omitted ...
```

(continues on next page)

(continued from previous page)

```
    # This is the name of the Secret we created, above
    sshKeys: volsync-rsync-dest-dest-database-destination
```

The ReplicationDestination object can now be created:

```
$ kubectl create -f examples/rsync/volsync_v1alpha1_replicationdestination.yaml
```

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

Contents

Rsync-based replication

- *Destination configuration*
  - *Destination status*
  - *Additional destination options*
- *Source configuration*
  - *Source status*
  - *Additional source options*
- *Rsync-specific considerations*
  - *Copying the SSH key secret*
  - *Choosing between Service types (ClusterIP vs LoadBalancer)*

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, VolSync 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 VolSync Rsync data mover then connects to the destination using ssh *(exposed via a Service)* and sends any updates. At the conclusion of the transfer, the destination (optionally) creates a VolumeSnapshot to preserve the updated data.

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

2.5.4 Destination configuration

Start by configuring the destination; an example is shown below:

```
---
apiVersion: volsync/v1alpha1
kind: ReplicationDestination
metadata:
  name: myDest
  namespace: myns
spec:
  rsync:
    copyMethod: Snapshot
    capacity: 10Gi
    accessModes: ["ReadWriteOnce"]
    storageClassName: my-sc
    volumeSnapshotClassName: my-vsc
```

In the above example, a 10 GiB RWO volume will be provisioned using the StorageClass `my-sc` 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, using the VolumeSnapshotClass named `my-vsc`, 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 the ReplicationDestination's `.status.latestImage`.

Destination status

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

```
---
apiVersion: volsync/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: volsync-dest-test-20210114194305
```

(continues on next page)

(continued from previous page)

```
  rsync:
    address: 10.99.236.225
    sshKeys: volsync-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`. This Secret will need to be *copied to the source* so that it can authenticate.

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 Direct, this will be the PVC that is used as the destination by VolSync.

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

capacity When VolSync 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:

- **Direct** - 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 VolSync automatically provision the destination volume (using capacity, accessModes, etc.), the name of a pre-existing PVC may be specified here.

storageClassName When VolSync 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. If omitted, the system default VolumeSnapshotClass will be used.

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 VolSync 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.

2.5.5 Source configuration

An example source configuration is shown here:

```
---
apiVersion: volsync.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: mySource
  namespace: source
spec:
  sourcePVC: mysql-pv-claim
  trigger:
    schedule: "*/5 * * * *"
  rsync:
    sshKeys: volsync-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.

When configuring the source, the user must manually create the Secret referenced in `.spec.rsync.sshKeys` by *copying the contents* from the Secret generated previously on the destination (and made available in the destination's `.status.rsync.sshKeys`).

Additionally, this ReplicationSource specifies a `copyMethod` of `Clone` which will directly generate a point-in-time copy of the source volume. However, not all CSI drivers support volume cloning (most notably the ebs-csi driver). In such cases, the `copyMethod: Snapshot` can be used to indirectly create a copy of the volume by first taking a snapshot, then restoring it. In this case, the user should also provide the `volumeSnapshotClassName: <vsc-name>` option to indicate which VolumeSnapshotClass VolSync should use when creating the temporary snapshot.

Source status

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

```
---
apiVersion: volsync.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: mySource
  namespace: source
spec:
  sourcePVC: mysql-pv-claim
  trigger:
    schedule: "*/5 * * * *"
  rsync:
    # ... omitted ...
status:
```

(continues on next page)

(continued from previous page)

```
  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.
- **Direct** - Do no create a PiT copy. The VolSync 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.

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*.

2.5.6 Rsync-specific considerations

This section explains some additional considerations when setting up rsync-based replication.

Copying the SSH key secret

When setting up the replication, it is necessary for the ReplicationSource to have a copy of the SSH keys so that it can connect to the network endpoint created by the ReplicationDestination. While these keys can be *generated manually*, the recommended method is to allow VolSync to generate the keys when setting up the ReplicationDestination. The resulting Secret should then be copied to the source cluster.

Below is an example of a ReplicationDestination object. The VolSync operator has generated the SSH keys that should be used in the source, and it has provided the name of the Secret containing them in the `.status.rsync.sshKeys` field:

Listing 5: ReplicationDestination with SSH key Secret highlighted

```
apiVersion: volsync.backube/v1alpha1
kind: ReplicationDestination
metadata:
  creationTimestamp: "2022-02-17T13:56:16Z"
  generation: 1
  name: database-destination
  namespace: dest
  resourceVersion: "2307"
  uid: 71f0512b-8a6b-438c-9b9a-0dd2c0f4e7b8
spec:
  rsync:
    accessModes:
    - ReadWriteOnce
    capacity: 2Gi
    copyMethod: Snapshot
    serviceType: ClusterIP
status:
  conditions:
  - lastTransitionTime: "2022-02-17T13:56:30Z"
    message: Reconcile complete
    reason: ReconcileComplete
    status: "True"
    type: Reconciled
  lastSyncStartTime: "2022-02-17T13:56:16Z"
```

(continues on next page)

(continued from previous page)

```
  rsync:
    address: 10.96.150.107
    sshKeys: volsync-rsync-dst-src-database-destination
```

This Secret exists in the same Namespace as the associated Replicationdestination. It has the following contents:

Listing 6: Secret as created by VolSync

```
apiVersion: v1
data:
  destination.pub: c3NoL...
  source: LS0tL...
  source.pub: c3NoLX...
kind: Secret
metadata:
  creationTimestamp: "2022-02-17T13:56:30Z"
  name: volsync-rsync-dst-src-database-destination
  namespace: dest
  ownerReferences:
  - apiVersion: volsync.backube/v1alpha1
    blockOwnerDeletion: true
    controller: true
    kind: ReplicationDestination
    name: database-destination
    uid: 71f0512b-8a6b-438c-9b9a-0dd2c0f4e7b8
  resourceVersion: "2296"
  uid: 61ab5402-318f-46df-b36f-cd209f3d1455
type: Opaque
```

The above Secret contains 3 fields: the source's public, the source's private, and the destination's public keys.

This Secret must be copied to the source cluster, into the same Namespace where the source PVC and ReplicationSource will reside. That can be accomplished as follows:

```
$ kubectl -n dest get secret volsync-rsync-dst-src-database-destination -oyaml > secret.
↪yaml
```

Once saved to the local file, prepare it for the new cluster/namespace by removing the following fields from the `metadata` area:

- `creationTimestamp`
- `namespace`
- `ownerReferences`
- `resourceVersion`
- `uid`

After removing the above fields, the Secret is as follows:

Listing 7: Prepared `secret.yaml`

```
apiVersion: v1
data:
```

(continues on next page)

(continued from previous page)

```
  destination.pub: c3NoL...
  source: LS0tL...
  source.pub: c3NoLX...
kind: Secret
metadata:
  name: volsync-rsync-dst-src-database-destination
type: Opaque
```

Assuming the source objects will be in Namespace `source`, this Secret can be added to the source cluster via:

```
$ kubectl -n source create -f secret.yaml
secret/volsync-rsync-dst-src-database-destination created
```

This Secret should then be referenced when creating the corresponding ReplicationSource. For example:

Listing 8: ReplicationSource showing reference to SSH key Secret

```
apiVersion: volsync.backube/v1alpha1
kind: ReplicationSource
metadata:
  name: database-source
  namespace: source
spec:
  sourcePVC: mysql-pv-claim
  trigger:
    schedule: "*/10 * * * *"
  rsync:
    sshKeys: volsync-rsync-dest-src-database-destination
    address: my.host.com
    copyMethod: Clone
```

Choosing between Service types (ClusterIP vs LoadBalancer)

When using Rsync-based replication, the ReplicationSource needs to be able to make a network connection to the ReplicationDestination. This requires network connectivity from the source to the destination cluster.

When a ReplicationDestination object is created, VolSync creates a corresponding Service object to serve as the network endpoint. The type of Service (LoadBalancer or ClusterIP) should be specified in the ReplicationDestination's `.spec.rsync.serviceType` field.

Listing 9: ReplicationDestination with service type highlighted

```
apiVersion: volsync.backube/v1alpha1
kind: ReplicationDestination
metadata:
  name: database-destination
  namespace: dest
spec:
  rsync:
    accessModes:
    - ReadWriteOnce
    capacity: 2Gi
```

(continues on next page)

(continued from previous page)

```
    copyMethod: Snapshot
    serviceType: ClusterIP
```

The clusters' networking configuration between the two clusters affects the proper choice of Service type.

If `ClusterIP` is specified, the Service will receive an IP address allocated from the "cluster network" address pool. By default, this collection of addresses are not accessible from outside the cluster, making it a poor choice for cross-cluster replication. However, various networking addons such as Submariner bridge the cluster networks, making this a good option.

If `LoadBalancer` is specified, an externally accessible IP address will be allocated. This requires cluster support for load balancers such as those provided by the various cloud providers or MetalLB in the case of physical clusters. While this is the easiest method for allocating an accessible address in cloud environments, load balancers tend to incur additional costs and be limited in number.

To summarize the above trade-offs, when running on one of the public clouds, using a LoadBalancer is a quick way to get started and will work for replicating small numbers of volumes. If replicating a large number of volumes, an overlay network solution such as Submariner in combination with ClusterIP addresses will likely be more scalable.

2.6 VolSync CLI / kubectl plugin

2.6.1 Migrating data into Kubernetes

```
$ kubectl volsync migration
Copy data from an external file system into a Kubernetes PersistentVolume.

This set of commands is designed to help provision a PV and copy data from a
directory tree into that newly provisioned volume.

Usage:
kubectl-volsync migration [command]

Available Commands:
create      Create a new migration destination
delete      Delete a new migration destination
rsync       Rsync data from source to destination
```

Example Usage

Example steps

- *Create the migration destination*
- *Copy the data into the PVC*
- *Clean up*
- *Use the data in-cluster*

The following example uses the `kubectl volsync migration` subcommand to migrate data from a stand-alone storage system into a Kubernetes PersistentVolumeClaim.

External storage A locally mounted directory tree (could be local disk or network-attached storage such as NFS or GlusterFS)

Destination cluster OpenShift running on GCP with their CSI driver. Note: The VolSync operator must be installed in the destination cluster.

Create the migration destination

Begin by creating a Namespace to hold the PVC (and eventually the application that will use the data).

```
$ kubectl create ns destination
namespace/destination created
```

Create a target for the data migration. If a capacity and accessModes are provided and the PVC does not already exist, the VolSync CLI will create the PVC. Otherwise, it will use the existing PVC.

```
$ kubectl volsync migration create -r mig-example --capacity 2Gi --accessmodes␣
↪ReadWriteOnce --storageclass standard-csi --pvcname destination/mydata
I0302 12:50:42.498947  168200 request.go:665] Waited for 1.007067079s due to client-side␣
↪throttling, not priority and fairness, request: GET:https://api.ci-ln-72rwmxb-72292.
↪origin-ci-int-gce.dev.rhcloud.com:6443/apis/project.openshift.io/v1?timeout=32s
I0302 12:50:43.925309  168200 migration_create.go:329] pvc: "mydata" not found, creating␣
↪the same
I0302 12:50:43.974092  168200 migration_create.go:267] Namespace: "destination" is found,
↪ proceeding with the same
I0302 12:50:44.021410  168200 migration_create.go:314] Created Destination PVC: "mydata"␣
↪in NameSpace: "destination" and Cluster: ""
I0302 12:50:44.073745  168200 migration_create.go:357] Created ReplicationDestination:
↪"destination-mydata-migration-dest" in Namespace: "destination" and Cluster: ""

$ kubectl get -n destination pvc/mydata
NAME     STATUS   VOLUME                                     CAPACITY   ACCESS MODES  ␣
↪STORAGECLASS   AGE
mydata   Bound    pvc-c9040e1f-e3dd-49e4-aa5d-194079181f55   2Gi        RWO           ␣
↪standard-csi   3m6s
```

Copy the data into the PVC

Once the destination has been created, we can use the CLI to transfer data into the cluster.

The data currently resides in the `/tmp/data` directory:

```
$ ls /tmp/data
./  ../  linux-4.1.51/

$ du -sh /tmp/data
643M        /tmp/data
```

Sync this data into the cluster:

```
$ kubectl volsync migration rsync -r mig-example --source /tmp/data/
...
```

(continues on next page)

(continued from previous page)

```
Number of files: 52,680 (reg: 49,453, dir: 3,213, link: 14)
Number of created files: 52,680 (reg: 49,453, dir: 3,213, link: 14)
Number of deleted files: 0
Number of regular files transferred: 49,453
Total file size: 556.98M bytes
Total transferred file size: 556.97M bytes
Literal data: 556.97M bytes
Matched data: 0 bytes
File list size: 524.26K
File list generation time: 0.001 seconds
File list transfer time: 0.000 seconds
Total bytes sent: 150.77M
Total bytes received: 961.29K

sent 150.77M bytes  received 961.29K bytes  10.46M bytes/sec
total size is 556.98M  speedup is 3.67
```

Incremental changes can also be transferred:

```
$ echo "hello" > /tmp/data/hi.txt

$ kubectl volsync migration rsync -r mig-example --source /tmp/data/
I0302 13:37:37.698258  174966 request.go:665] Waited for 1.004977118s due to client-side␣
↪throttling, not priority and fairness, request: GET:https://api.ci-ln-72rwmxb-72292.
↪origin-ci-int-gce.dev.rhcloud.com:6443/apis/snapshot.storage.k8s.io/v1beta1?timeout=32s
I0302 13:37:39.093025  174966 migration_rsync.go:132] Extracting ReplicationDestination␣
↪secrets
I0302 13:37:39.177009  174966 migration_rsync.go:190] Migrating Data from "/tmp/data/"␣
↪to "\destination\mydata"
.d..t...... ./
<f+++++++++ hi.txt

Number of files: 52,681 (reg: 49,454, dir: 3,213, link: 14)
Number of created files: 1 (reg: 1)
Number of deleted files: 0
Number of regular files transferred: 1
Total file size: 556.98M bytes
Total transferred file size: 6 bytes
Literal data: 6 bytes
Matched data: 0 bytes
File list size: 0
File list generation time: 0.001 seconds
File list transfer time: 0.000 seconds
Total bytes sent: 806.41K
Total bytes received: 3.60K

sent 806.41K bytes  received 3.60K bytes  147.28K bytes/sec
total size is 556.98M  speedup is 687.61
```

Clean up

Once all the data has been transferred, the VolSync destination objects can be cleaned up:

```
$ kubectl volsync migration delete -r mig-example
```

Use the data in-cluster

We can now start a pod attached to the PVC and view the data:

Listing 10: pod.yaml

```
---
kind: Pod
apiVersion: v1
metadata:
  name: busybox
spec:
  containers:
    - name: busybox
      image: busybox
      command: ["/bin/sh", "-c"]
      args: ["sleep 999999"]
      volumeMounts:
        - name: data
          mountPath: "/mnt"
  volumes:
    - name: data
      persistentVolumeClaim:
        claimName: mydata
```

```
$ kubectl -n destination apply -f pod.yaml
pod/busybox created

$ kubectl -n destination exec -it pod/busybox -- ls -al /mnt
total 12
drwx--x--x    3 101587   101587        4096 Mar  2 18:37 .
dr-xr-xr-x    1 root     root            73 Mar  2 18:39 ..
-rw-------    1 101587   101587           6 Mar  2 18:37 hi.txt
drwx--x--x   23 101587   101587        4096 Mar 27  2018 linux-4.1.51

$ kubectl -n destination exec -it pod/busybox -- du -sh /mnt
655.4M      /mnt
```

2.6.2 Asynchronous replication

```
$ kubectl volsync replication
Replicate the contents of one PersistentVolume to another.

This set of commands is designed to set up and manage a replication
relationship between two different PVCs in the same Namespace, across
Namespaces, or in different clusters. The contents of the volume can be
replicated either on-demand or based on a provided schedule.

Usage:
  kubectl-volsync replication [command]

Available Commands:
  create          Create a new replication relationship
  delete          Delete an existing replication relationship
  schedule        Set replication schedule for the relationship
  set-destination Set the destination of the replication
  set-source      Set the source of the replication
  sync            Run a single synchronization
```

Example usage

Example steps

- *Kubectl configuration*
- *Deploy the application*
- *Set up replication*
- *Examining VolSync resources*
- *Manual synchronization*
- *Removing the replication*

The following example uses the `kubectl volsync replication` subcommand to set up and manage cross-cluster asynchronous replication of a PVC.

Application A simple busybox pod that has a PVC attached

Source cluster Kind cluster running on a local laptop using the hostpath CSI driver

Destination cluster OpenShift running on GCP with their CSI driver

Kubectl configuration

The following steps assume that you have a kubeconfig defined that will allow access to both clusters (source and destination) by switching between contexts.

```
$ kubectl config get-contexts
CURRENT   NAME        CLUSTER                 AUTHINFO     NAMESPACE
*         gcp         ci-ln-nm6319k-72292     admin
          kind        kind-kind               kind-kind
```

A configuration like the above will allow directing requests to the different clusters via `kubectl --context <name>`. Likewise, some of the VolSync CLI commands will refer to this context name (e.g., `<context>/<namespace>/<resource>`).

Please see the Kubernetes documentation for details on how to set up your kubeconfig to access multiple clusters.

Deploy the application

The application is a simple busybox pod and an attached PVC.

Listing 11: pvc.yaml

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

Listing 12: pod.yaml

```yaml
---
kind: Pod
apiVersion: v1
metadata:
  name: busybox
spec:
  containers:
    - name: ubi
      image: busybox
      command: ["/bin/sh", "-c"]
      args: ["sleep 999999"]
      volumeMounts:
        - name: data
          mountPath: "/mnt"
  volumes:
    - name: data
```

(continues on next page)

(continued from previous page)

```
      persistentVolumeClaim:
        claimName: datavol
```

Create the namespace and application objects:

```
$ kubectl --context kind create ns source
namespace/source created

$ kubectl --context kind -n source create -f pvc.yaml
persistentvolumeclaim/datavol created

$ kubectl --context kind -n source create -f pod.yaml
pod/busybox created
```

Set up replication

Create a replication relationship. We are naming the relationship "example":

```
$ kubectl volsync replication -r example create
```

Set the source of the replication:

- The hostpath CSI driver supports volume cloning, so we'll use "Clone" as our method to create a point-in-time copy
- The name of the PVC to replicate is given as <cluster-context>/<namespace>/<name>

```
$ kubectl volsync replication -r example set-source --copymethod Clone --pvcname kind/
↪source/datavol
```

Set the destination:

```
# Create a namespace on the destination cluster
$ kubectl --context gcp create ns destns
namespace/destns created

$ kubectl volsync replication -r example set-destination --copymethod Snapshot --
↪storageclass standard-csi --volumesnapshotclass csi-gce-pd-vsc --servicetype␣
↪LoadBalancer --destination gcp/destns/datavol
```

Begin replicating on a 5 minute schedule:

```
$ kubectl volsync replication -r example schedule --cronspec '*/5 * * * *'
I0216 13:51:22.165811  275823 replication.go:381] waiting for keys & address of␣
↪destination to be available
I0216 13:51:32.296465  275823 replication.go:406] creating resources on Source
```

Examining VolSync resources

The above commands deployed a ReplicationSource and ReplicationDestination object on the two clusters:

```
$ kubectl --context kind -n source get replicationsource -oyaml
apiVersion: v1
items:
- apiVersion: volsync.backube/v1alpha1
  kind: ReplicationSource
  metadata:
    creationTimestamp: "2022-02-16T20:07:30Z"
    generation: 1
    labels:
      volsync.backube/relationship: 90d56bef-551d-4ede-b6a7-0783cabdafb6
    name: datavol-87srf
    namespace: source
    resourceVersion: "13695"
    uid: 7511b291-b768-4a2e-96cf-2eafd3854469
  spec:
    rsync:
      address: 34.121.93.205
      copyMethod: Clone
      sshKeys: datavol-87srf
    sourcePVC: datavol
    trigger:
      schedule: '*/5 * * * *'
  status:
    conditions:
    - lastTransitionTime: "2022-02-16T20:07:58Z"
      message: Waiting for next scheduled synchronization
      reason: WaitingForSchedule
      status: "False"
      type: Synchronizing
    - lastTransitionTime: "2022-02-16T20:07:30Z"
      message: Reconcile complete
      reason: ReconcileComplete
      status: "True"
      type: Reconciled
    lastSyncDuration: 28.732770544s
    lastSyncTime: "2022-02-16T20:07:58Z"
    nextSyncTime: "2022-02-16T20:10:00Z"
    rsync: {}
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""

$ kubectl --context gcp -n destns get replicationdestination -oyaml
apiVersion: v1
items:
- apiVersion: volsync.backube/v1alpha1
  kind: ReplicationDestination
  metadata:
```

(continues on next page)

(continued from previous page)

```
    creationTimestamp: "2022-02-16T20:06:10Z"
    generation: 1
    labels:
      volsync.backube/relationship: 90d56bef-551d-4ede-b6a7-0783cabdafb6
    name: datavol
    namespace: destns
    resourceVersion: "42743"
    uid: 040dc4ad-6f37-43f1-9da4-b28d956f2bb7
  spec:
    rsync:
      accessModes:
      - ReadWriteOnce
      capacity: 3Gi
      copyMethod: Snapshot
      serviceType: LoadBalancer
      storageClassName: standard-csi
      volumeSnapshotClassName: csi-gce-pd-vsc
  status:
    conditions:
    - lastTransitionTime: "2022-02-16T20:06:10Z"
      message: Reconcile complete
      reason: ReconcileComplete
      status: "True"
      type: Reconciled
    - lastTransitionTime: "2022-02-16T20:08:00Z"
      message: Synchronization in-progress
      reason: SyncInProgress
      status: "True"
      type: Synchronizing
    lastSyncDuration: 1m50.209297869s
    lastSyncStartTime: "2022-02-16T20:08:00Z"
    lastSyncTime: "2022-02-16T20:08:00Z"
    latestImage:
      apiGroup: snapshot.storage.k8s.io
      kind: VolumeSnapshot
      name: volsync-datavol-dst-20220216200800
    rsync:
      address: 34.121.93.205
      sshKeys: volsync-rsync-dst-src-datavol
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```

When creating the resources, the CLI:

- Created the ReplicationDestination
- Waited for the LoadBalancer address and SSH keys to become available
- Copied the SSH keys from the destination cluster to a Secret in the source cluster
- Created the ReplicationSource referencing the Secret, the remote address, and having the supplied cronspec schedule

Manual synchronization

The above steps establish a replication schedule wherein the source is periodically replicated to the destination. During planned migration events, it is desirable to force a synchronization and synchronously wait for completion.

Assuming the CLI has been used as described above, a manual synchronization can be triggered via:

```
$ kubectl volsync replication -r example sync
I0216 15:19:19.832648  290779 replication.go:381] waiting for keys & address of␣
↪destination to be available
I0216 15:19:19.954913  290779 replication.go:406] creating resources on Source
I0216 15:19:19.988886  290779 replication_sync.go:90] waiting for synchronization to␣
↪complete
```

When this command returns, a new synchronization (and VolumeSnapshot) will have been completed. To resume periodic synchronization, re-issue the `kubectl volsync replication schedule` command.

Removing the replication

When the replication relationship is no longer needed, it can be removed via:

```
$ kubectl volsync replication -r example delete
```

The above command removes the VolSync CRs and the SSH key Secret.

VolSync provides a CLI interface to assist in performing common operations using the VolSync operator.

All the tasks that can be accomplished via this CLI can also be performed by directly manipulating VolSync's ReplicationSource and ReplicationDestination objects. It is meant as a simple shortcut for common operations:

- *Setting up asynchronous data replication*
- *Migrating data into Kubernetes*

2.6.3 Installation

The plugin can installed via:

```
$ go install github.com/backube/volsync/kubectl-volsync@main
go: downloading github.com/backube/volsync v0.3.1-0.20220214161039-2a78c57773a4

$ which kubectl-volsync
~/go/bin/kubectl-volsync
```

Assuming that the above installation directory is in your `PATH`, the VolSync CLI will be available as a sub-command of `kubectl` or `oc`:

```
$ kubectl volsync --help
This plugin can be used to configure replication relationships using the
VolSync operator.

The plugin has a number of sub-commands that are organized based on common
data movement tasks such as:
```

(continues on next page)

(continued from previous page)

```
  *  Creating a cross-cluster data replication relationship
  *  Migrating data into a Kubernetes cluster
  *  Establishing a simple PV backup schedule

Usage:
  kubectl-volsync [command]

Available Commands:
  completion  generate the autocompletion script for the specified shell
  help        Help about any command
  migration   Migrate data into a PersistentVolume
  replication Replicate data between two PersistentVolumes
```

There are three different replication methods built into VolSync. 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.

2.7 Triggers

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

2.8 Metrics

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

CHAPTER

THREE

ENHANCEMENT PROPOSALS

3.1 A case for VolSync

Contents

- *A case for VolSync*
  - *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 VolSync 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 VolSync 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 VolSync 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 VolSync, copying the data from the source to the destination volume.

It is desirable for VolSync'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 VolSync 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 VolSync 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 VolSync will be centered around configuring the replication relationships between volumes. When looking at the *use cases* presented in the overview of VolSync, 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 VolSync 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: volsync/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: volsync/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

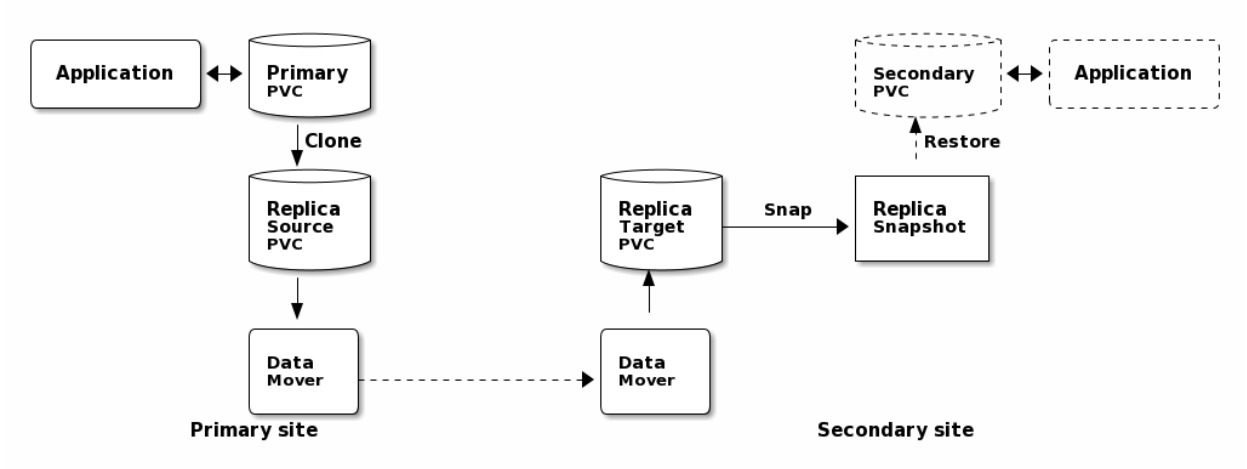
- *Rsync-based data mover*
  - *Overview*
  - *Replication flow*
  - *Setup*

3.3.1 Overview

To meet the goal of being able to replicate arbitrary volumes, VolSync 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 VolSync. Restic is a data backup utility that copies the data to an object store (among other options).

While the main purpose of VolSync 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 VolSync 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.

```
---
apiVersion: volsync/v1alpha1
kind: ReplicationSource
metadata:
  name: source
  namespace: myns
```

(continues on next page)

(continued from previous page)

```yaml
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 VolSync, 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: volsync.backube/v1alpha1
kind: ReplicationDestination
metadata:
```

(continues on next page)

(continued from previous page)

```
  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

VolSync is a Kubernetes operator that performs asynchronous replication of persistent volumes within, or across, clusters. The replication provided by VolSync 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/volsync