*** COMMUNITY DOC ***

Multi-Cluster Works API

Authors: jqiu@redhat.com
Updated: 08/17/2020

Objective

Introduce a common API to distribute workload to multiple clusters.

Background

The general idea originates from vllry's blog post and prototype, that we group a set of k8s API resources to be applied to one or multiple clusters together as a concept of "work" or "workload"

As the starting point, we want to have an API that describes "work" as a list of resources deployed to one single cluster. It could be simple enough that it does not relate to cluster registration mechanisms or any workload scheduling on multiple clusters.

Motivation

There are already several different techniques to distribute workload on multiple kubernetes clusters:

Kubefed v1

Kubefed v1 has an architecture that resources needs to be applied on an federated-apiserver and then the controller will push the corresponding resources onto the certain clusters. There are only limited types of resources that are supported since the controller for each resource has to be reimplemented. To select clusters that a resource should be applied to, a cluster selector annotation needs to be added into the resource.

Kubefed v2

Similar to the architecture of Kubefed v1, kubefed v2 has a controller to watch the federated resources and pushes resources to the specified cluster. However, kubefed v2 leverages CRD instead of an additional federation-apiserver. And each resource type needs a mapped

federated type for the federation controller to consume. A federated resource type is basically the original resource spec plus the placement field. Kubefedctl can be used to easily convert a resource type to its related federated resource type.

Gitops

Gitops is to use git as the single source of truth for resource manifests which can be pushed/pulled to certain clusters by a controller. The version control and git workflow can be leveraged to better control the content of the resource manifests.

Common abstractions

All these techniques has some common patterns for the function to deploy workload resource manifests to one or multiple clusters

1. A single source of truth which could be git, cloud storage, kube-apiserver or rpc server.
2. A control loop to apply resources manifests from a single source of truth to a cluster, return apply results and track resource status.
3. A way to decide which clusters the resource manifests should apply to. The blog post has identified some critical placement criteria
    a. Specific levels of redundancy.
    b. Specific kinds of geographic/topological placement or spread.
    c. Specific resource availability.

In addition, the control loop will also face the problem of garbage collecting resources when there is no intent to deploy them on a cluster

All of the above motivates the notion of work api which:

1. Allow developers to easily integrate with other sources of truth, e.g. git, another kube-apiserver. And the control loop of work api could provide a generic way to apply resource manifests to a certain cluster.
2. Easy to integrate with other placement primitives.
3. Track which clusters a particular workload is deployed to
4. Track the deployed resource on a cluster so the control loop could garbage collect these resources.

Terminology

- Work Hub: A place that work API resides. It could be a k8s cluster playing the role as the management plane for other k8s clusters. Or it could also be an RPC server or a cloud API depending on the detailed implementation. Users create work API resources on the work hub. In the rest of the doc, we assume the implementation of work hub is based on k8s cluster for simplicity.

- Managed Cluster: A k8s cluster managed by the work hub. The resources defined in the work API are applied on the managed cluster.
  It was also referred to “Spoke” or “Spoke Cluster”, though we use “Managed Cluster” in this document. We should make an agreement on the terminology.
- Work Controller: a controller that reconciles the work object on hub, and applies resources defined in work to the managed cluster.

Use Cases

Deploy a workload to another cluster

I have 3 clusters. One is the work hub, the other two are managed clusters. I want to declare the workload (deployment/configmap/service etc) on the work hub, and ensure the workload will be deployed on the desired managed cluster. I want to ensure that when I update the workload declaration, the real deployed workload on the managed cluster reflects the update.

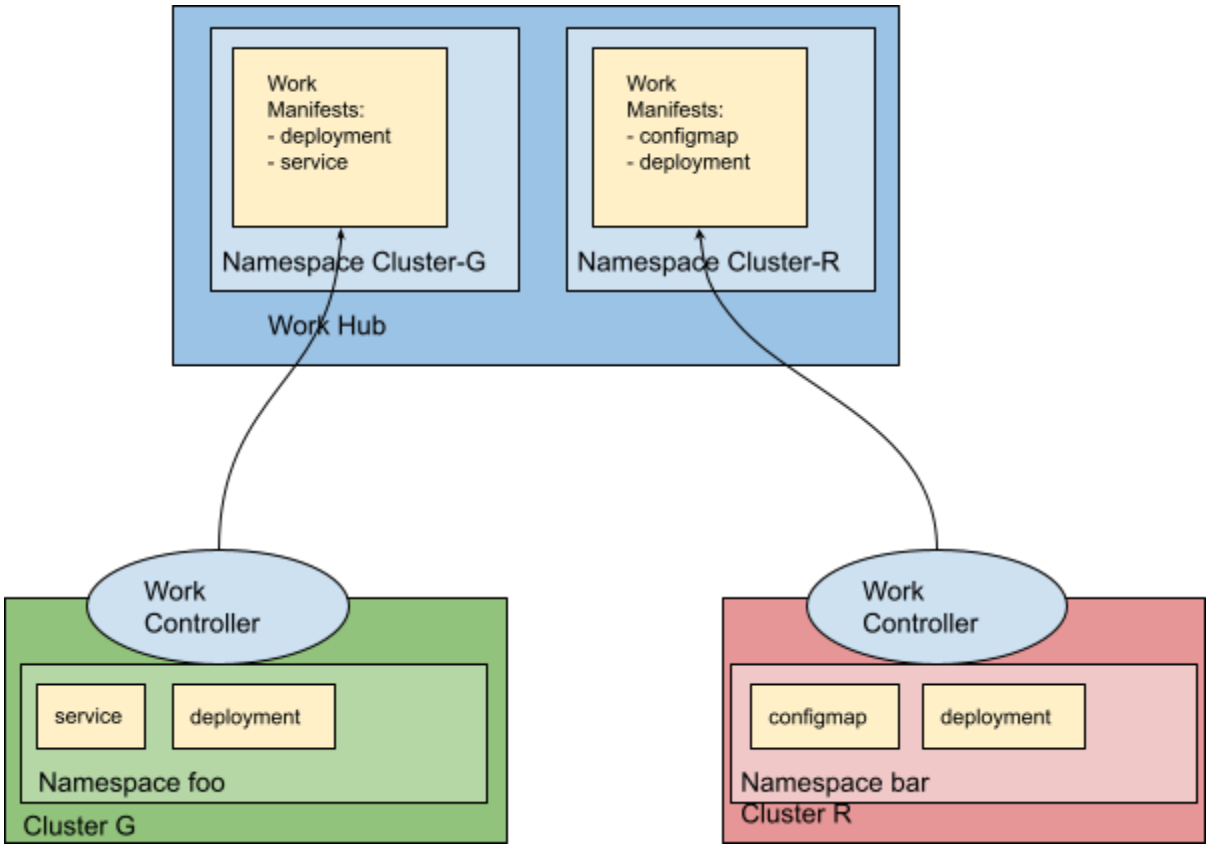
Track the status of workload in another cluster

After I have declared the workload on the work hub, I want to track whether the workload has been successfully deployed on the managed cluster, and the workload is running normally.

Overview

We propose a new CRD called `Work` to represent a list of api resources to be deployed on a cluster. `Work` is created on the work hub, and resides in the namespace that the work controller is authorized to access. Creation of a `Work` on the work hub indicates that resources defined in `Work` will be applied on a certain managed cluster. Update of a `Work` will trigger the resource update on the managed cluster, and deletion of a `Work` will recycle the resources on the managed cluster.

If there are multiple managed clusters, multiple work controllers will be running that monitor the work API in same or different namespaces in the work hub. It is possible that multiple work controllers watch `Works` in one namespace on a work hub and deploy the resources on multiple clusters. It is also possible that multiple work controllers watch `Works` in different namespaces on the work hub, so a `Work` created in one namespace triggers the resource deployment on a certain cluster. An example of architecture diagram is as following:



Resources in the work

The resources in kube could be classified to several categories:

1. Workload related resources: deployments/statefulset, configmaps, ns-scoped custom resources etc.
2. Clusterwide configuration resources: apiservices, CRDs, storageclasses
3. Credentials: secrets

Work api should be mainly used for workload related resources.
Secrets should not be declared in work apis, other techniques (e.g. vault) should be considered as a more secure way to transmit secrets among clusters.

Push/Pull Model

There have been discussions in the community on whether the workload distribution in multicluster should use a push or pull model.

Push model means that a controller on the hub watches APIs defining workload and “PUSH” the resource manifest to the managed cluster. There are some limitations in the push model:

- It requires apiserver of each managed cluster must be accessed by the work hub where the controller is running. This could be a hard requirement since some managed clusters may hide behind firewalls and do not have a public accessible IP. Exposing apiserver of all the managed clusters also enlarges the surface to be attacked.
- It requires credentials of managed clusters with sufficient permission to be put on the work hub. The credential has to be passed in an out of band secure way.
- Having a centralized controller to distribute workload to many clusters could have scalability limitations.

Pull model means that an agent running in the managed cluster watches APIs defined on hub, fetches them and applies locally on the managed cluster. Compared to the push model, the API exposure surface is reduced since only apiserver of the work hub needs to be publicly accessible. The credential for the agent to talk to the work hub can have very limited permission only on certain APIs.
`Work` API itself is not constrained to be used only on push or pull mode. The work controller could reside in a managed cluster that “PULLS” the API and applies locally on the managed cluster. It could also reside in the work hub that watches the Wok API and “PUSHES” the workload to the managed cluster.

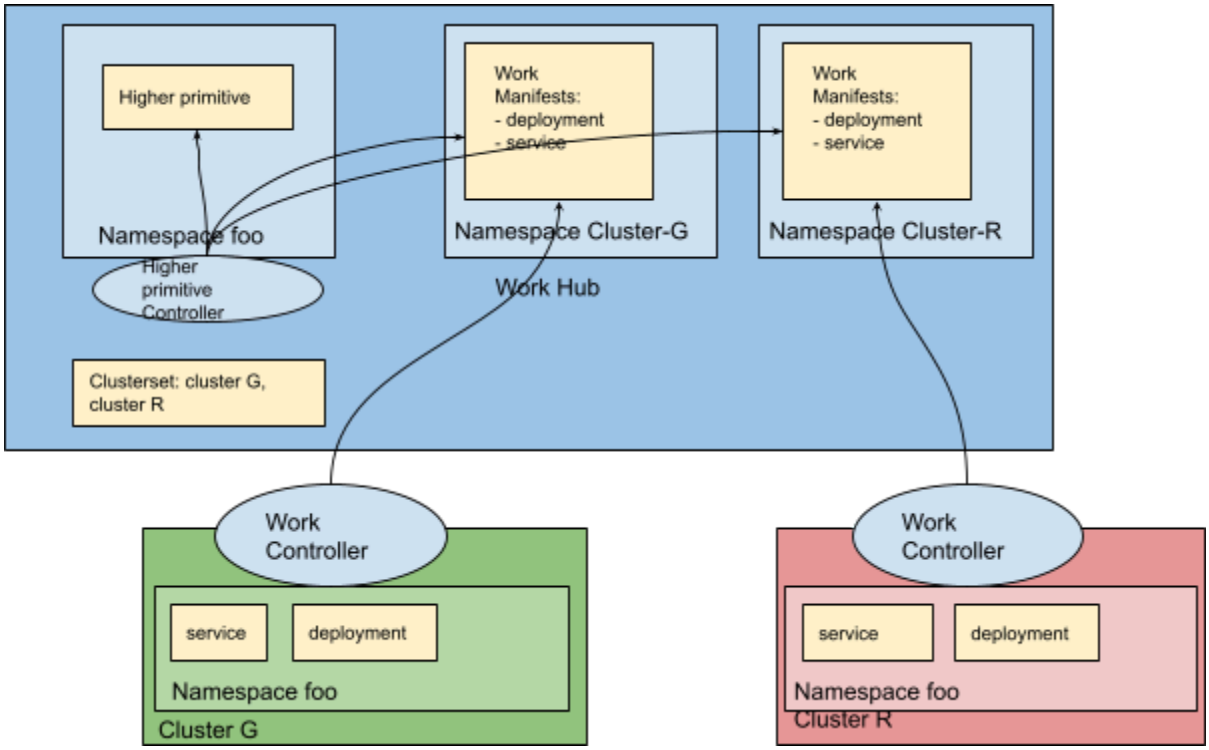
Working with higher primitive

`Work` represents a workload to be deployed in a target namespace on a single managed cluster. Which cluster the work is to be deployed and how the `Work` is scheduled to a certain cluster is not defined in the `Work` api. A higher primitive could be used to generate `Work` based on a scheduling decision and place the workload on a managed cluster. The higher primitive must coordinate with clusterset together to decide which clusters the `Work` should place to.

Let’s assume a scenario that a user would like to place a work including deployment and service in namespace foo to a clusterset containing cluster G and cluster R. Two possible flows could be used by leveraging `Work`

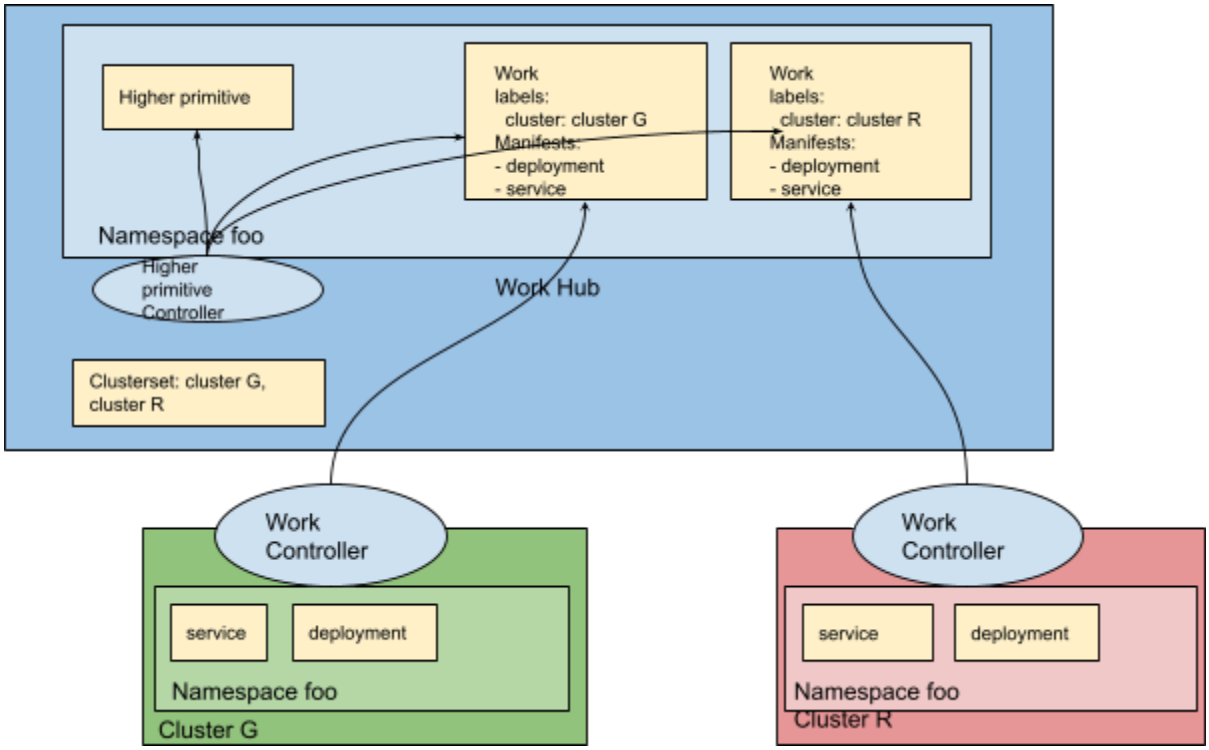
Flow 1:

1. User creates higher primitive in namespace foo in work hub, which indicate the user intends to place the workload in target namespace foo of all clusters in the clusterset (cluster R and cluster G)
2. Controller of higher primitive identifies the clusters in cluster R and cluster G
3. Controller of higher primitive creates `Work` on the work hub, in the namespace cluster-G and cluster-R.
4. Work controller in cluster-G who only watches the ns cluster-G on work hub gets the work, and deploy the resources in the work to namespace foo in cluster-G.



Flow 2:

1. User creates higher primitive in namespace foo in work hub, which indicate the user intends to place the workload in target namespace foo of all clusters in the clusterset (cluster R and cluster G)
2. Controller of higher primitive identifies the clusters in cluster R and cluster G.
3. Controller of higher primitive creates two `Work` on the work hub, in the namespace foo with cluster labels.
4. Work controller on the managed cluster uses a label selector to select the `Work` applied on this cluster.



Proposed Design

To deploy a workload, user will create a `Work` on the work hub

```
// Work defines a list of resources to be deployed on the managed cluster
type Work struct {
   metav1.TypeMeta   `json:",inline"`
   metav1.ObjectMeta `json:"metadata,omitempty"`

   Spec   WorkSpec   `json:"spec,omitempty"`
   Status WorkStatus `json:"status,omitempty"`
}

// WorkSpec defines the desired state of Work
type WorkSpec struct {
   // Workload represents the manifest workload to be deployed on managed cluster
   Workload WorkloadTemplate `json:"workload,omitempty"`
}

// WorkloadTemplate represents the manifest workload to be deployed on managed cluster
type WorkloadTemplate struct {
```

```
  // Manifests represents a list of kuberenetes resources to be deployed on the managed
cluster.
  // +optional
  Manifests []Manifest `json:"manifests,omitempty"`
}

// Manifest represents a resource to be deployed on managed cluster
type Manifest struct {
  runtime.RawExtension `json:",inline"`
}

// WorkStatus defines the observed state of Work
type WorkStatus struct {
  // Conditions contain the different condition statuses for this work.
  // Valid condition types are:
  // 1. Applied represents workload in Work is applied successfully on a managed cluster.
  // 2. Progressing represents workload in Work is being applied on a managed cluster.
  // 3. Available represents workload in Work exists on the managed cluster.
  // 4. Degraded represents the current state of workload does not match the desired
  // state for a certain period.
  Conditions []metav1.Condition `json:"conditions"`

  // ManifestConditions represents the conditions of each resource in work deployed on
  // managed cluster.
  // +optional
  ManifestConditions []ManifestCondition `json:"manifestConditions,omitempty"`
}

// ResourceIdentifier provides the identifiers needed to interact with any arbitrary object.
type ResourceIdentifier struct {
  // Ordinal represents an index in manifests list, so the condition can still be linked
  // to a manifest even though manifest cannot be parsed successfully.
  Ordinal int `json:"ordinal,omitempty"`

  // Group is the group of the resource.
  Group string `json:"group,omitempty"`

  // Version is the version of the resource.
  Version string `json:"version,omitempty"`

  // Kind is the kind of the resource.
  Kind string `json:"kind,omitempty"`

  // Resource is the resource type of the resource
  Resource string `json:"resource,omitempty"`

  // Namespace is the namespace of the resource, the resource is cluster scoped if the value
```

```
  // is empty
  Namespace string `json:"namespace,omitempty"`

  // Name is the name of the resource
  Name string `json:"name,omitempty"`
}

// ManifestCondition represents the conditions of the resources deployed on
// managed cluster
type ManifestCondition struct {
  // resourceId represents the identity of a resource linking to manifests in spec.
  // +required
  Identifier ResourceIdentifier `json:"identifier,omitempty"`

  // Conditions represents the conditions of this resource on the managed cluster
  // +required
  Conditions []metav1.StatusCondition `json:"conditions"`
}
```

```
apiVersion: multicluster.x-k8s.io/v1alpha1
kind: Work
metadata:
 name: work-sample
 namespace: cluster
spec:
 workload:
   manifests:
   - apiVersion: v1
     kind: ConfigMap
     metadata:
       name: cm
       namespace: default
     data:
       ui.properties: |
         color=purple
```

User creates a `Work` in the namespace on the hub that the work controller is authorized to access. The work controller then accesses the managed cluster and applies the resources defined in `Work` in its reconcile loop. Work controller also tracks the status of applied resources by updating the manifest conditions in `Work` status.

Manifest Conditions

Manifest conditions represent the status conditions of a certain manifest to be applied on a managed cluster. The structure of manifest conditions will include an identifier to link to the resources defined in work.spec field, and a list of conditions showing the current status of the resource applied. An example of manifest conditions as below that a configmap default/cm1 has been applied on the managed cluster.

```
manifestConditions:
    - conditions:
      - lastTransitionTime: "2020-07-02T03:16:26Z"
        message: Apply manifest complete
        reason: AppliedManifestComplete
        status: "True"
        type: Applied
      identifier:
        group: ""
        kind: ConfigMap
        name: cm1
        namespace: default
        ordinal: 0
        resource: configmaps
        version: v1
```

Condition type in a manifest conditions

Applied indicates that the manifest with the identifier is applied successfully in the managed cluster.
Degraded indicates that the manifest applied on the managed cluster does not match the desired status. Example is the running replica in deployment does not fit the desired replica in deployment spec.

Work Conditions

In addition to track status of each manifest with manifest condition, a work should have summarized conditions based on manifest conditions. An example of work and manifest conditions together as below:

```
conditions:
   - lastTransitionTime: "2020-07-02T03:16:26Z"
     message: Apply manifest work complete
     reason: AppliedManifestWorkComplete
     status: "True"
     type: Applied
 manifestConditions
     - conditions:
       - lastTransitionTime: "2020-07-02T03:16:26Z"
         message: Apply manifest complete
         reason: AppliedManifestComplete
         status: "True"
         type: Applied
       identifier:
         group: ""
         kind: ConfigMap
         name: cm1
         namespace: default
         ordinal: 0
         resource: configmaps
         version: v1
```

We can build a summarization logic to set work conditions from manifest conditions such as

1. If “Applied==true” for all manifest conditions, then set Applied=true for work condition
2. If “Degraded==false” for all manifest conditions, then set Degraded=false for work condition

Edge Cases

This is an attempt to enumerate known edge cases, where naive attempts to reconcile would not work as intended.

Work controller tries to revert the field which is updated by other automation.

- Work controller tries to remove the service.spec.ClusterIP since it is not defined in the `Work` spec.
- Work controller tries to remove the serviceaccount.secrets since it is not defined in the `Work` spec.
- Work controller tries to update fields of APIs that are set by a defaulter.
- Work controller tries to revert replica of deployment when HPA scales up the deployment

Work controller tries to update an immutable field.

- Work controller tries to update secret.Type but failed since the field is immutable

Multiple work objects desire the same API resources.

Solutions for edge cases

Leverage server apply

Server-side apply tracks which actor has changed each field of the object. The reconciler should be able to declare the owner of fields in the resources defined in work. Possible logic would be like:

1. If the reconciler detects that itself is not a fieldManager of the resource on the managed cluster, declares itself as fieldManager for any field of resource defined in the work with force-conflict.
2. If the reconciler finds itself is already a fieldManager ot the resource, use server side apply without force conflict. It is possible that the owner of some field on the resource has been transferred to another fieldmanager, and conflict would happen. In this case, reports the conflict in the work's manifest condition

Explicit define reconcile actions in work

We could have a section in the work API to define what reconcile action should be done in each field in the manifest defined in this work. We could add field specifiers for the following:

- Don't remove: don't remove a value absent in the desired state
- Don't update: don't reconcile 1 value with another

An example of such work with service, service account in its manifest would look like as below:

```
apiVersion: multicluster.x-k8s.io/v1alpha1
kind: Work
metadata:
 name: work-sample
 namespace: cluster
spec:
 workload:
   manifests:
   - apiVersion: v1
     kind: Service
     metadata:
```

```
    name: svc
    namespace: default
  spec:
    selector:
      app: demo
    ports:
    - protocol: TCP
      port: 80
      targetPort: 80
- apiVersion: v1
    kind: ServiceAccount
    metadata:
      name: sa
      namespace: default
workMeta:
  specialKeys:
    notRemove:
    - v1.service.spec.clusterIP
    - v1.serviceaccount.secrets
```

Special logic for known types

We could program special reconcile logic for known types such as service, secret. serviceAccount.

References

[1] https://timewitch.net/post/2020-03-31-multicluster-workloads/
[2] https://github.com/vllry/cluster-reconciler
[3] https://docs.google.com/document/d/1fpUQ-Bru1OUK_p4j6q58EHfi8E827dzLNSspgB6lHc4/edit#