


FLIP-144: Native Kubernetes HA for Flink

Status

Discussion thread	https://lists.apache.org/thread.html/r466ad059dda1276fba4fa9a710cbfdfeab6b8a24c4047c6ed5d619e8%40%3Cdev.flink.apache.org%3E
Vote thread	https://lists.apache.org/thread.html/r816a1c557fc197769e2974950d13605406d9fce59a4dce51562b8b33%40%3Cdev.flink.apache.org%3E
JIRA	 FLINK-12884 - FLIP-144: Native Kubernetes HA Service CLOSED
Release	1.12

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

- [Motivation](#)
- [Background](#)
 - [HA components](#)
 - [Kubernetes](#)
 - [ConfigMap](#)
 - [Resource version](#)
 - [LeaderElection](#)
 - [Owner Reference](#)
- [Architecture](#)
- [Implementation](#)
 - [LeaderElection](#)
 - [LeaderRetrieval](#)
 - [Shared and dedicated ConfigMap](#)
 - [HA data clean up](#)
- [How to use](#)
 - [Standalone on K8s](#)
 - [Native K8s](#)
- [Compatibility, Deprecation, and Migration Plan](#)
- [Test Plan](#)
- [Alternative HA implementation](#)
 - [StatefulSet + PV + FileSystemHAResourceService](#)
- [Rejected Alternatives](#)
 - [Lock and release](#)

Motivation

High Availability(aka HA) is a very basic requirement in production. It helps to eliminate the single point of failure for Flink clusters. For Flink HA configuration, it is necessary to have more than one JobManagers in the cluster, known as active and standby JobManagers. Once the active JobManager failed exceptionally, other standby ones could take over the leadership and recover the jobs from the latest checkpoint. Starting more than one JobManager will make the recovery faster. Benefit from Yarn application attempts or Kubernetes(aka K8s) deployment, more than one JobManagers could be easily started successively or simultaneously.

Currently, Flink has provided Zookeeper HA and been widely used in production environments. It could be integrated in standalone cluster, Yarn, Kubernetes deployments. However, using the Zookeeper HA in K8s will take additional cost since we need to manage a Zookeeper cluster. In the meantime, K8s has provided some public API for leader election and configuration storage(i.e. ConfigMap). We could leverage these features and make running HA configured Flink cluster on K8s more convenient.

Note: Both the [standalone on K8s](#) and [native K8s](#) could benefit from the new introduced **KubernetesHaService**.

Background

The first major functionality of Flink high availability is leader election and retrieval(aka service discovery). If we have multiple JobManagers running, they should elect an active one for the resource allocation and task scheduling. Also the RPC endpoint will be stored in a shared storage. Others become standby and wait for taking over. The TaskManagers will retrieve the active JobManager address for registration and offering slots.

A distributed coordination system(e.g. Zookeeper, ETCD) also serves as a distributed key-value data store. The running job ids, job graph meta, checkpoints meta will be persisted in the share store.

Currently, Flink high availability service could be implemented as plugins. In order to enable Flink HA over various distributed coordination systems, interface **HighAvailabilityServices** have already been abstracted which consists of the following five components. In this way, the implementation directly interacting with specific distributed coordination systems is decoupled with flink's internal logic. For example, **ZooKeeperHaServices** is the implementation of HighAvailabilityServices based on Zookeeper, and we need to add a similar one based on K8s APIs.

HA components

The following are key components of interface HighAvailabilityServices.

- **LeaderElectionService**
 - Contends for the leadership of a service in JobManager. There are four components in a JobManager instance that use LeaderElectionService: ResourceManager, Dispatcher, JobManager, RestEndpoint(aka WebMonitor).
 - Once the leader election is finished, the active leader addresses will be stored in the ConfigMap so that other components could retrieve successfully.
- **LeaderRetrievalService**
 - Used by Client to get the RestEndpoint address for the job submission.
 - Used by JobManager to get the ResourceManager address for registration.
 - Used by TaskManagers to retrieve addresses of the corresponding LeaderElectionService(e.g. JobManager address, ResourceManager address) for registration and offering slots.
- **RunningJobsRegistry**
 - Registry for the running jobs. All the jobs in the registry will be recovered when JobManager failover.
- **SubmittedJobGraphStore**
 - JobGraph instances for running JobManagers. Note that only the meta information(aka location reference, DFS path) will be stored in the Zookeeper/ConfigMap. The real data is stored on the DFS.
- **CheckpointRecoveryFactory**
 - Stores meta information to Zookeeper/ConfigMap for checkpoint recovery.
 - Stores the latest checkpoint counter.

Kubernetes

ConfigMap

Kubernetes provides [ConfigMap](#) which could be used as key-value storage. Actually a ConfigMap can store a set of key-value pairs just like a Map in Java. And the values in ConfigMap can be binary data, we can safely serialize/deserialize from java object to/from ConfigMap. However, it is supported after K8s 1.10 version. And in current implementation, i suggest to use base64 to encode the serializedStoreHandle and store in data field.

The size limit of a ConfigMap is 1 MB based on [Kubernetes codes](#) (MaxSecretSize = 1 * 1024 * 1024). We should make sure the total size of all the values (including data and binary data) in a ConfigMap should not be greater than 1 MB. So we could only store metadata or dfs location reference in the ConfigMap. The real data needs to be stored on DFS(configured via `high-availability.storageDir`). We will store job graphs, completed checkpoints, checkpoint counters, and running job registry in the ConfigMap.

Note: Actually we are already using ConfigMap to store flink-conf.yaml, log4j properties and hadoop configurations.

Resource version

In [ZooKeeperCheckpointIDCounter](#), Flink is using a [shared counter](#) to make sure the "get and increment" semantics. Benefit from the [Kubernetes Resource Version](#), we could perform a similar transactional operation using K8s API. It is used to enable optimistic concurrency for atomic read/update/write operations.

The way that it works is like this:

- Client reads value, get resource version N
- Client updates the value client side to represent desired change.
- Client writes back the value with resource version N.

This write only succeeds if the current resource version matches N. This ensures that no one else has snuck in and written a different update while the client was in the process of performing its update.

This will guarantee that Flink state metadata is not updated concurrently and goes into the wrong state in any case.

LeaderElection

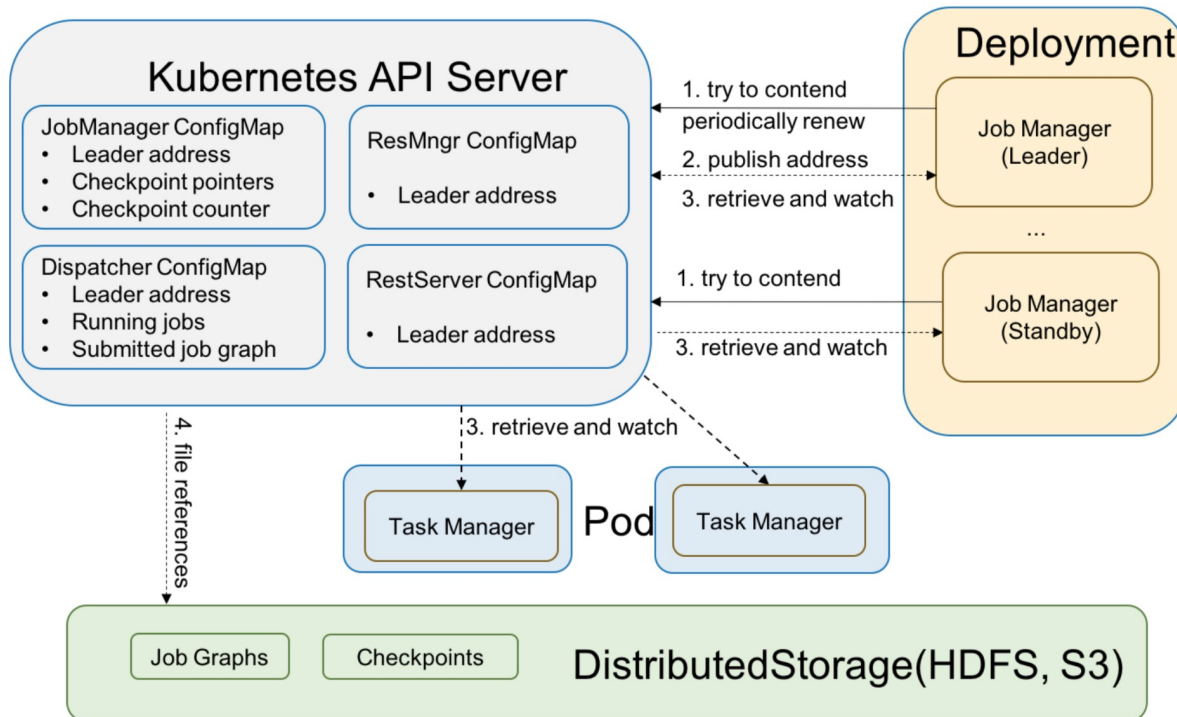
Based on the resource version, we could easily do a compare-and-swap operation for certain K8s objects. So a leader election could be achieved in the following steps.

1. Start multiple JobManagers and the instance who firstly creates the ConfigMap will become the leader at the very beginning. Also it will add an annotation([control-plane.alpha.kubernetes.io/leader](#)) to expose leadership information to other followers in the cluster.
2. The leader JobManager should periodically renew its lease time to indicate its existence. Please make sure that the renew interval is greater than leaseDuration.
3. The followers will constantly check the existence of ConfigMap. If it is created already by the leader then the followers will do a lease checking against the current time. If renewTime is outdated, it usually means the old leader JobManager died. Then a new leader election process is started until a follower successfully claims leadership by updating ConfigMap with its own identity and lease duration.

Owner Reference

Kubernetes [Owner Reference](#) is used for garbage collection. When the owner of some K8s resources are deleted, they could be deleted automatically. Benefit from this, in Flink we set owner of the flink-conf configmap, service and TaskManager pods to JobManager Deployment. So when we want to [destroy a Flink cluster](#), we just need to delete the deployment.

Architecture



1. For the leader election, a set of JobManagers for becoming leader is identified. They all race to declare themselves as the leader. One of them wins and becomes the leader. Once the election won, the active JobManager continually "heartbeats" to renew their position as the leader. All other standby JobManagers periodically make new attempts to become the leader. This ensures that the JobManager could failover quickly. The ResourceManager, JobManager, Dispatcher, RestEndpoint have separate leader election services and ConfigMaps.
2. The active leader publishes its address to the ConfigMap. Note that, we use the same ConfigMap for contending lock and store the leader address.
3. The leader retrieval service is used to find the active leader address and then register themselves. For example, TaskManagers retrieve the address of ResourceManager and JobManager for the registration and offering slots. We will use a Kubernetes watcher in the leader retrieval service. Once the content of ConfigMap changed, it usually means the leader has changed, it could get the latest leader address.
4. All other meta information(e.g. running jobs, job graphs, completed checkpoints and checkpoint counter) will be directly stored in different ConfigMaps. It will only be cleaned up when the Flink cluster reaches the global terminal state.

Implementation

LeaderElection

Currently, we already have an embedded [fabric8 Kubernetes client](#) in Flink. It is widely used in many projects and works pretty well in Flink. Now it could also support the [leader election](#). The following is a very simple example of how the leader election could be used.

Note:

- The `run` method is a blocking call. So it should be called in the io executor service, not the main thread.
- When we want to write the leader information to the ConfigMap, we could check the leadership first.
- "Get(check the leader)-and-Update(write back to the ConfigMap)" is a transactional operation.

KubernetesLeaderElectionService.java

```
private final LeaderElector leaderElector = kc.leaderElector()
    .withConfig(
        new LeaderElectionConfigBuilder()
            .withName(leaderKey)
            .withLeaseDuration(Duration.ofSeconds(15L))
            .withLock(new ConfigMapLock(ns, leaseName, lockIdentity))
            .withRenewDeadline(Duration.ofSeconds(10L))
            .withRetryPeriod(Duration.ofSeconds(2L))
            .withLeaderCallbacks(new LeaderCallbacks(
                this::isLeader,
                this::notLeader,
                newLeader -> LOG.info("New leader elected {}. ", newLeader)
            ))
            .build()
    ).build();
executor.execute(leaderElector::run);
```

Each component will have a separate leader election service and [ConfigMap](#) named with “<ClusterID>-<Component>”. The ConfigMap is used to store the leader information. The following is a list of leader ConfigMaps for a typical Flink application with HA enabled.

k8s-ha-appl-00000000000000000000000000000000-jobmanager	2	4m35s
k8s-ha-appl-dispatcher	2	4m38s
k8s-ha-appl-resourcemanager	2	4m38s
k8s-ha-appl-restserver	2	4m38s

A detailed ConfigMap of rest server leader is shown below.

```
$ kubectl get cm k8s-ha-appl-restserver -o yaml

apiVersion: v1
data:
  address: http://172.20.1.36:8081
  sessionId: 25546372-b83f-4cea-96bb-81c97c8cd7df
kind: ConfigMap
metadata:
  annotations:
    control-plane.alpha.kubernetes.io/leader: '{"holderIdentity":"545ble7a-9d7f-4133-a3a0-0cfd19fef14c","leaseDuration":15.000000000,"acquireTime":"2020-09-14T09:41:23.347000Z","renewTime":"2020-09-14T09:46:32.747000Z","leaderTransitions":155}'
  creationTimestamp: "2020-09-14T09:41:24Z"
  name: k8s-ha-appl-restserver
  namespace: default
  resourceVersion: "1365032204"
  selfLink: /api/v1/namespaces/default/configmaps/k8s-ha-appl-restserver
  uid: 0b2b914f-a570-4025-bed8-4cf0aec9a49c
```

LeaderRetrieval

We could create a watcher for the ConfigMap and get the leader address in the callback handler.

KubernetesLeaderRetrievalService.java

```
kubeClient.configMaps().withName(cm).watch(new Watcher<ConfigMap>() {
    @Override
    public void eventReceived(Action action, ConfigMap resource) {
        final String name = resource.getMetadata().getName();
        switch (action) {
            case ADDED:
            case MODIFIED:
                if (resource.getData() != null) {
                    // TODO a new leader has been elected
                }
                break;
            case DELETED:
                listener.handleError(new Exception("Deleted while watching the configMap " + name));
                break;
            case ERROR:
                listener.handleError(new Exception("Error while watching the configMap " + name));
                break;
            default:
                LOG.debug("Ignore handling {} event for configMap {}", action, resource.getMetadata().getName());
                break;
        }
    }
}
```

Shared and dedicated ConfigMap

Unlike the hierarchical structure in Zookeeper, ConfigMap provides a flat key-value map. So we may need to store multiple keys in a specific ConfigMap. Each component(Dispatcher, ResourceManager, JobManager, RestEndpoint) will have a dedicated ConfigMap. All the HA information relevant for a specific component will be stored in a single ConfigMap. For example, the Dispatcher's ConfigMap would then contain the current leader, the running jobs and the pointers to the persisted JobGraphs. The JobManager's ConfigMap would then contain the current leader, the pointers to the checkpoints and the checkpoint ID counter. Since "Get(check the leader)-and-Update(write back to the ConfigMap)" is a transactional operation, we will completely solved the concurrent modification issues and not using the "lock-and-release" in Zookeeper.

HA data clean up

Currently, when a Flink cluster reached the terminal state(**FAILED**, **CANCELED**, **FINISHED**), all the HA data, including Zookeeper and HA storage on DFS, will be cleaned up in **'HighAvailabilityServices#closeAndCleanupAllData'**. For the KubernetesHService, we should have the same clean-up behavior.

So the following command will only shut down the Flink session cluster and leave all the HA related ConfigMaps, state untouched.

```
echo 'stop' | ./bin/kubernetes-session.sh -Dkubernetes.cluster-id=<ClusterId> -Dexecution.attached=true
```

The following commands will cancel the job in application or session cluster and effectively remove all its HA data.

```
# Cancel a Flink job in the existing session
$ ./bin/flink cancel -t kubernetes-session -Dkubernetes.cluster-id=<ClusterID> <JobID>
# Cancel a Flink application
$ ./bin/flink cancel -t kubernetes-application -Dkubernetes.cluster-id=<ClusterID> <JobID>
```

If the user wants to keep the HA data and restart the Flink cluster, he/she could simply delete the deploy(via `kubectl delete deploy <ClusterID>`). All the Flink cluster related resources will be destroyed(e.g. JobManager Deployment, TaskManager pods, services, Flink conf ConfigMap) so that it will not occupy the K8s cluster resources. For the HA related ConfigMaps, we do not set the owner so that they could be retained. Then he/she could use `kubernetes-session.sh` or `flink run-application` to start the session/application again. All the previous running jobs could recover from the latest checkpoint successfully.

How to use

Standalone on K8s

Both session and job/application clusters could use the new introduced **KubernetesHaService**. We just need to add the following Flink config options to [flink k-configuration-configmap.yaml](#). All other yamls do not need to be updated.

flink-configuration-configmap.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: flink-config
  labels:
    app: flink
data:
  flink-conf.yaml: |+
  ...
  kubernetes.cluster-id: standalone-k8s-ha-app1
  high-availability: org.apache.flink.kubernetes.highavailability.KubernetesHaServicesFactory
  high-availability.storageDir: oss://flink/flink-ha
  restart-strategy: fixed-delay
  restart-strategy.fixed-delay.attempts: 10
  ...
```

Native K8s

The following is a simple sample how to start a Flink application with native HA enabled. The config options are same for the Flink session cluster.

```
./bin/flink run-application -p 10 -t kubernetes-application -Dkubernetes.cluster-id=k8s-ha-app1 \
-Dkubernetes.container.image=flink:k8s-ha \ -Dkubernetes.container.image.pull-policy=Always \
-Djobmanager.heap.size=4096m -Dtaskmanager.memory.process.size=4096m \
-Dkubernetes.jobmanager.cpu=1 -Dkubernetes.taskmanager.cpu=2 -Dtaskmanager.numberOfTaskSlots=4 \
-Dhigh-availability=org.apache.flink.kubernetes.highavailability.KubernetesHaServicesFactory \
-Dhigh-availability.storageDir=oss://flink/flink-ha \
-Drestart-strategy=fixed-delay -Drestart-strategy.fixed-delay.attempts=10 \
-Dcontainerized.master.env.ENABLE_BUILT_IN_PLUGINS=flink-oss-fs-hadoop-1.12.jar \
-Dcontainerized.taskmanager.env.ENABLE_BUILT_IN_PLUGINS=flink-oss-fs-hadoop-1.12.jar \
local:///opt/flink/examples/streaming/StateMachineExample.jar
```

Compatibility, Deprecation, and Migration Plan

This is a complete new feature. We will not have any compatibility, deprecation, migration issues. Just like the command in the previous section, users could use this new feature via simply adding a few Flink config options.

Test Plan

Unlike Zookeeper, we do not have a TestingServer in the fabric8 Kubernetes client. So we just need to mock the dependency component and test the contract.

Fortunately, we could use minikube for the E2E tests. Start a Flink session/application cluster on K8s, kill one TaskManager pod or JobManager Pod and wait for the job recovered from the latest checkpoint successfully.

```
kubectl exec -it {pod_name} -- /bin/sh -c "kill 1"
```

Moreover, we need to test the new introduced **KubernetesHaService** in a real K8s clusters.

- Running with multiple JobManagers.
- Kill the active one and the job should recover from latest checkpoint.
- Cancel or fail the job, all the HA data should be cleaned up.
- Delete JobManager deployment, the HA data should be retained. Then start the Flink cluster again, the Flink job should recover.

Alternative HA implementation

StatefulSet + PV + FileSystemHaService

[Kubernetes StatefulSet](#) could guarantee that there will never be more than 1 instance of a pod at any given time, which is different from a deployment. So it is quite appropriate to replace the leader election/retrieval. For the TaskManagers, the unique pod name "**<ClusterID>-jobmanager-0**" could always be used to reach to the JobManager.

Note: An except is [manually force-deletion](#). If a Kubernetes node is down, and the user perform a force-deletion for the StatefulSet pod. We may have two running JobManagers then.

[Kubernetes Persistent Volume](#)(PV) has a lifecycle independent of any individual Pod that uses the PV. It could make Flink JobManager keep the local data after failover. So we just need to mount a PV as local path(e.g. `/flink-ha`) for the JobManager pod and set the high availability storage to the local directory. The job graph, running job registry, completed checkpoint and checkpoint counter also need to be stored in the local directory.

[FileSystemHAResource](#) is a new added simple high availability service implementation. It does not provide leader election/retrieval functionality. Unlike **ZooKeeperHAResource** and **KubernetesHAResource**, it directly stores/recovers the HA data to/from local directory.

This HA solution is quite straightforward and could be implemented easily. However, the limitations are also quite obvious. First, it requires the Kubernetes cluster should have pre-configured PV, which is not always true especially in the unmanaged(self-built) cluster. Second, we could not support multiple JobManagers instances since we do not have an embedded leader election/retrieval service. Third, we need to change the current JobManager Deployment to StatefulSet.

In the future we could have both solutions for deploying Flink on K8s with HA enabled. And they could be used to different scenarios.

Rejected Alternatives

Lock and release

Concurrent modification could happen on job graph store and checkpoint store. So in current **ZooKeeperJobGraphStore** and **ZooKeeperCompletedCheckpointStore** implementation, we are using lock and release to avoid concurrent add/delete of job graphs and checkpoints. The job graph and completed checkpoint could only be deleted by the owner or the owner has died. We add an ephemeral node under the persistent node to lock the node. And remove the ephemeral node when we want to release the lock. The following is an ephemeral node of lock.

```
[zk: hostxxxx:2181(CONNECTED) 10] get /flink/application_1597378016174_0573/jobgraphs
/e96a119a188e014e3c660d076d3fc73e/e39e15f9-11d7-4ba5-98d1-bfc44d36c243

11.164.87.180
cZxid = 0x30034fcf6
ctime = Thu Sep 10 13:53:59 CST 2020
mZxid = 0x30034fcf6
mtime = Thu Sep 10 13:53:59 CST 2020
pZxid = 0x30034fcf6
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x173e74341abb276
dataLength = 13
numChildren = 0
```

However, we could not find an existing similar mechanism in Kubernetes. The ETCD does not support [ephemeral key](#). So we need to do this in Flink. First, when we want to lock a specific key in ConfigMap, we will put the owner identify, lease duration, renew time in the ConfigMap annotation. The annotation will be cleaned up when releasing the lock. When we want to remove a job graph or checkpoints, it should satisfy the following conditions. If not, the delete operation could not be done.

- Current instance is the owner of the key.
- The owner annotation is empty, which means the owner has released the lock.
- The owner annotation timed out, which usually indicates the owner died.