Shared Cache Design Document (YARN-1492)

Authors: Chris Trezzo and Sangjin Lee

Diff from previous versions

1. Cache management logic moved from the client to a shared cache manager and localization service. As a benefit, cached resources are now read-only and managed solely by a trusted source (SCM and NodeManager) running as its own user.
2. Caching of a resource is now off the job submission path and happens asynchronously. If submitting to the cache fails, it does not affect the job.
3. Improved robustness when dealing with clients that don't clean up after themselves correctly.

Introduction

The proposed shared cache YARN feature adds the facility to upload and manage shared application resources to HDFS in a safe and scalable manner. YARN applications will be able to leverage resources uploaded by other applications or previous runs of the same application without having to re-upload identical files multiple times. This will save network resources and reduce YARN application startup time.

Currently, there are already some building blocks that can be leveraged at both the YARN and MapReduce layers. At the YARN layer, the node manager resource localization service maintains a cache of "localized" files (i.e. files downloaded from HDFS) and makes them available to YARN containers. These files can either be public (shared by all users on the node), private (shared by all applications run by the same user), or application specific (shared by all containers running the same application). The proposed design will heavily leverage this resource localization service.

At the MapReduce layer, the distributed cache api allows MapReduce jobs to share the same read-only public resources that have been uploaded to HDFS. This facility is MapReduce specific and requires a MapReduce client to know what resources have already been uploaded to HDFS. The proposed design will create a more generic facility for MapReduce jobs as well as other YARN applications to share resources.

The major design goals of the proposal are as follows:

1. Scalable - The design must be scalable to a large number of cached entries. It should not adversely affect the amount of load put on the namenode or resource manager. This will require a design that can handle a large number of cache entries with an acceptable amount of overhead. In addition, it should be able to tolerate rapid growth in the number of

cached entries.
2. Secure - Resources in the shared cache should be protected from tampering and the users of the cache must be able to trust the integrity of the cache contents.
3. Fault Tolerant - YARN applications should be able to continue running even if the shared cache service is unavailable. The shared cache service should be able to gracefully start and stop. Finally, the shared cache should be able to tolerate YARN client failure (i.e. should not depend on applications cleaning up their resources correctly).
4. Transparent - The management of the shared cache should be transparent to existing MapReduce jobs as well as new YARN applications.

Design Overview

The shared cache service consists of two major components. The shared cache manager (aka. SCM) and the localization service.

Shared Cache Manager (SCM)

The SCM is the major component of the shared cache service. It will be responsible for communicating with the clients about what is in the shared cache and removing (i.e. cleaning up) stale entries from the shared cache. The SCM will run as a separate daemon process that can be placed on any node in the cluster. This allows for administrators to start/stop/upgrade the SCM without affecting other YARN components (i.e. the resource manager or node managers). Clients of the shared cache will only need to communicate with the SCM via the public client api.

SCM Public Client API

```
Path use(String checksum, String appId);
```

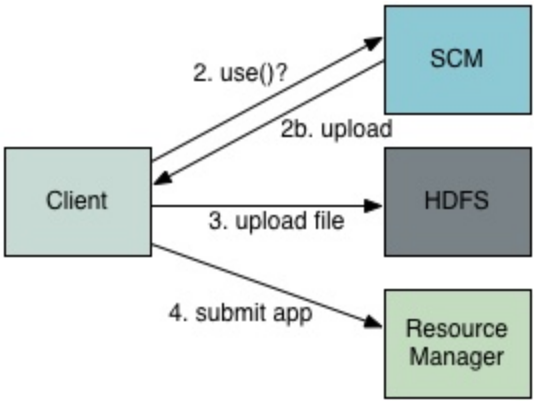
This method enables the client to register its use of a cached resource identified by the given checksum with the SCM. If the resource exists in the shared cache, its path is returned, and this app is recorded as using this resource. If the resource does not exist, null is returned.

```
boolean release(String checksum, String appId);
```

Indicates that an application is no longer using the specified shared resource. If a client uses this method it is helpful for the SCM to clean up the cache, but it is not necessary for the cache to function correctly.

Client to SCM Protocol

This is the client to SCM protocol that is used for a client to submit a resource to the shared cache:



1. Client computes checksum of resource.
2. Notify SCM of the intent to use resource: use(checksum, appId)
    a. If resource is in cache, SCM returns path to cache. Use that path to refer to the resource during job submission. Continue to 4.
    b. SCM returns null. The resource is not in the shared cache. Continue to 3.
3. Upload file to HDFS using the normal yarn application specific mechanism. For example, in MapReduce the file is uploaded to a staging directory via the JobSubmitter.
4. Submit application using the path from either step 3 or step 2a.
5. Optionally notify the SCM that the application is no longer using the resource: release(checksum, appId). This is purely an optimization and is not required for correctness.

SCM State

Checksum	Cleaner Lock	List of AppIds	Access Time
9c34c19d934c7d57cfbeca313c59f58d	Y	1385075571494_444294, 1385075571494_444292, 1385075571494_444291	1392143175
9c34c19a9c4c7d57cfb3ca313c790ab6	N	1385075571494_444294	1392149432
9c34c194984c7d57cfbec4313c590a38	N		1392143586

For every cache entry the SCM will keep track of the checksum, the cleaner lock (i.e. whether or not the cleaner is currently looking at the entry), a list of application ids that are referencing the entry, and the latest access time of the entry.

The SCM state will have to be stored in some manner. The design will be agnostic to the storage mechanism used. Initially the SCM will support a Zookeeper storage mechanism, but the state could easily be stored using another facility. Using Zookeeper has several advantages: it provides an HA storage mechanism, it allows for quick SCM failover, and it is aligned with other HA efforts in the YARN/HDFS ecosystem. Using Zookeeper (along with any other remote store) will increase network traffic and the complexity of the design, but we believe the advantages outweigh these issues.

Note: We also considered keeping the SCM state in-memory and recreating the state on SCM startup using currently active applications and their resources. The SCM would get a list of currently running applications on start-up and the cleaner service would not run until all of these initial applications have terminated. Once all initial applications have terminated, the cleaner service knows it is safe to delete resources according to the normal cleaning policy. Unfortunately this method does not work with long running applications. We deemed this to be too much of a downside considering these types of YARN applications will become more and more prominent in the future.

SCM Cleaner service

The SCM cleaner service is responsible for scanning the shared cache and removing stale cache entries. An entry is stale if the cached resource has remained unused for more than the staleness time period (a configurable setting). One can run the cleaner service on a periodic basis (e.g. weekly), or on demand, or both.

There is a natural tension between the frequency at which the cleaner service runs and the amount of additional load placed on the resource manager. This will be something that requires tuning and the intent is to provide enough knobs to make the cleaner service work effectively.

In the future, one could envision more pluggable cache eviction policies (e.g. size based or frequency based), but for the first version, the staleness policy will be used.

SCM Staleness based Cleaner Protocol for a cache entry

This protocol is run for each entry in the cache (i.e. each cache directory in HDFS).

1. Check the access time of the cache entry. If it is stale, set the cleaner lock to true in the SCM table.
2. Delete appIds from the list of appIds for apps that are not running.
3. If there are no appIds left in the list and the cleaner lock is set, continue to 4. Otherwise, exit.
4. Delete the row in the SCM table.

5. Delete the cached entry in HDFS.

Localization Service

This design will leverage the existing node manager localization service. Once a resource is localized with a PUBLIC visibility, the NodeManager will optionally add the resource to the shared cache. By default, the NodeManager that is running the Application Master container will submit resources to the shared cache. This minimizes the number of NodeManagers that submit resources for a given application. YARN application developers can override this behavior using the ContainerLaunchContext (e.g. if different containers localize different resources) and explicitly choose which containers submit resources to the cache.

Localization Protocol

1. Localize resource onto NodeManger (i.e. the LocalizedResource should be in a LOCALIZED state).
2. Continue to 3 if the resource should be added to the cache (according to the ContainerLaunchContext), otherwise exit.
3. Compute the checksum of the localized resource.
4. Upload the localized resource to HDFS in the proper shared cache directory (see Cache Structure section) as a temporary file.
5. Move temporary file (i.e. an atomic operation) to the proper name in the same directory.
6. Notify the SCM that the resource (identified by the checksum) is now in the shared cache.

Security and Coordination

The SCM and NodeManager runs as its own super-user and acts as a strong gatekeeper for the shared cache. These are the only two entities that manipulate the shared cache. All resources in the cache are read-only to anyone else. In addition, all resources in the shared cache are placed based on the trusted checksum generated by the localization service.

A checksum based on a cryptographically strong hash function (e.g. SHA-256) will be used. This will ensure that validation and identification of resources in the shared cache are done in a safe way.

Finally, all resources added to the shared cache are required to have a PUBLIC visibility. This prevents unintentionally sharing private resources.

Cache Structure

The shared cache will be in a configured root directory and use a nested layout with a configured depth. The resources in the cache will be identified by their checksum and will determine the path for a given cache entry. The checksum will be divided into components of the path depending on the depth specified in configuration. For example, suppose we are dealing with a file named foo.jar with the SHA-256 sum 9c34c194984c7d57cfbeca313c590a36, the root cache directory is configured to /sharedcache, and the depth is set to 3 levels. Then the path to this jar in the cache will be:

/sharedcache/9/c/3/9c34c194984c7d57cfbeca313c590a36/foo.jar

This nested structure will allow an administrator to tune the maximum number of files per single directory in the shared cache.

Metrics

There will be additional metrics added around resource localization and shared cache management. There is an existing JIRA (YARN-1529) that adds metrics around the existing localization service to the node manager. These metrics include the following:

- LocalizationDownloadNanos
- LocalizedBytesCached
- LocalizedBytesCachedRatio
- LocalizedBytesMissed
- LocalizedFilesCached
- LocalizedFilesCachedRatio
- LocalizedFilesMissed

MAPREDUCE-5696 exposes these metrics as MapReduce job counters.

In addition, there will be new metrics added to the cleaner service in the SCM. These metrics will cover things like number of entries cleaned per run, number of total entries, size of cache, etc. etc. etc.

Administration

Proper administrative commands will be added to the yarn CLI script as well as the yarn-daemon scripts. This will give an administrator the ability to start/stop the proper shared cache services and manually trigger a run of the SCM cleaner service.