Shared Cache Design Document (YARN-1492)

Authors: Chris Trezzo and Sangjin Lee

Diff from previous version

1. Added protocol diagram.
2. Updated ClientSCMProtocol.
3. Updated SCMStore.
4. Updated Upload/Notify protocol.

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.

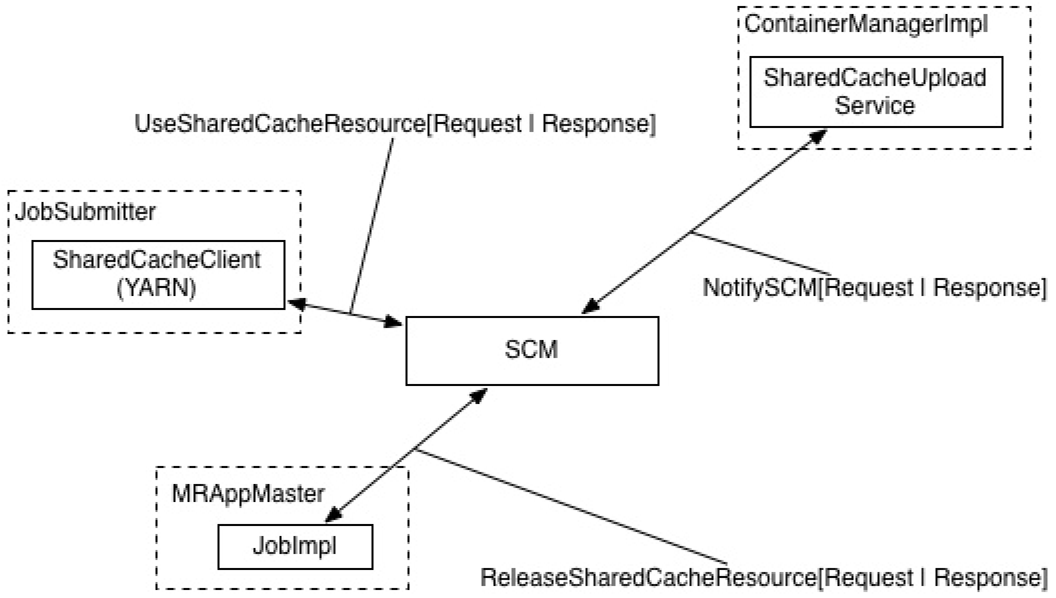
The major design goals of the proposal are as follows:

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.

Here is a diagram of the major components and protocols:


ContainerManagerImpl
SharedCacheUpload Service
UseSharedCacheResource[Request | Response]
JobSubmitter
SharedCacheClient (YARN)
NotifySCM[Request | Response]
SCM
MRAppMaster
JobImpl
ReleaseSharedCacheResource[Request | Response]

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);
```

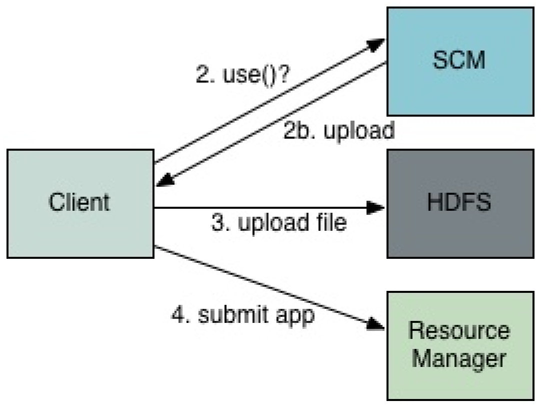
This method enables the client to register its use of a cached resource identified by the given checksum with the SCM. If the resource exists in the shared cache, its path is returned, and this app is recorded as using this resource. If the resource does not exist, null is returned.

```
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 5.
    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. When creating a LocalResource, set the policy for caching that resource (i.e. true if the resource should be uploaded to the cache, false otherwise). The decision of which containers are responsible for uploading resources to the cache is made by the YARN application developer. For example, in MapReduce the MRAppMaster uploads resources to the cache. All other containers do not. The decision to give application developers the ability to choose how resources are cached was done to allow for applications where resources might not be the same across all containers in the application.
5. Submit application using the path from either step 3 or step 2a.
6. After the application has terminated, 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 Store

The SCMStore is the piece of the SCM responsible for maintaining the shared cache metadata. For every cache entry the SCM will keep track of the checksum, the HDFS filename associated with the entry, a list of resource references associated with the entry, and the latest access time of the entry. A resource reference is an ApplicationID and a user name. These are used to keep track of which applications are currently using a cached resource and what user issued the use command.

The design will be agnostic to the storage mechanism used and will have a clean interface with the rest of the SCM. Initially the SCM will support a simple in-memory store. On startup, the store will recreate it’s state using currently active applications and their resources. Once all initial applications have terminated, the cleaner service knows it is safe to delete resources according to the normal cleaning policy.

As follow up work, there will be a local persistent store to handle stateful restarts (necessary for long running applications) and a Zookeeper-based store for HA.

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.

As follow up work, there will be 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 if an entry exists in the SCM. If no entry, check the HDFS modification time of the cache entry directory. If it is older than the staleness period, delete the entry. Otherwise, exit.
2. Check the access time of the cache entry in SCM. If it is stale, set the cleaner lock to true for this entry.
3. Delete appIds from the list of appIds for apps that are not running.
4. If there are no appIds left in the list and the cleaner lock is set, continue to 5. Otherwise, exit.
5. Delete the row in the SCM table.
6. Delete the cached entry in HDFS.

Localization Service

This design will leverage the existing node manager localization service. Once a resource is localized, 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.

Upload/Notify Protocol

1. Localize resource onto NodeManager (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/LocalResource), 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. If the rename fails (the file already exists), delete the temporary file and exit.
6. Notify the SCM that the resource (identified by the checksum) is now in the shared cache. If the SCM rejects the upload (it might already be in the cache under a different filename), delete the file.

Note: There exists a race between the NodeManager uploading the localized resource to the shared cache and the ResourceLocalizationService cache cleanup removing the local resource file. Since uploading a resource to the shared cache is asynchronous and not required for job submission/execution to succeed, adding the resource to the cache can be best effort. If the localization service loses the race with the cache cleaner then the resource simply won't make it into the cache. We feel that this is a reasonable design choice. If it turns out that this is indeed a common condition that is limiting the utility of the shared cache, then the solution would be to introduce additional coordination within the localization service.

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 either be globally readable or owned by the user that is submitting them to the cache. This prevents sharing private resources. The latter condition requires strong authentication to run securely. The localization service will use the existing security facilities that are part of the container and ContainerLaunchContext.

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.

MapReduce Application-Level Changes

The MapReduce application will support the use of the shared cache. The bulk of the changes will be in the job submission code. The JobSubmitter will interact with both the distributed cache api and the shared cache client. MapReduce will use the default shared cache behavior and let the NodeManager that is running the Application Master container submit resources to the cache.

Configuration

Clients will be able to control what is submitted to the shared cache at the job-level using a new job config parameter:

- mapreduce.job.sharedcache.mode (default: disabled) - A comma delimited list of resource categories to submit to the shared cache. The valid categories are: jobjar, libjars, files, archives. If “disabled” is specified then the job submission code will not use the shared cache.

Some example values are:

- jobjar,libjars - only the jobjar and libjars will be submitted to the shared cache
- files - only files will be submitted to the shared cache
- disabled - nothing will be submitted to the shared cache

Note: If permissions are not set correctly for any of the resources (i.e. not world readable or owned by the job submission user), the NodeManager will log a warning message but will continue without submitting resources to the shared cache.