Combined Aggregated Logs

YARN-2942

4/24/15 -- Robert Kanter

Problem

Turning on log aggregation allows users to easily store container logs in HDFS and subsequently view them in the YARN web UIs from a central place. Currently, there is a separate log file for each Node Manager per YARN application. This can be a problem for HDFS if you have a cluster with many nodes as you'll slowly start accumulating many (possibly small) files and blocks. To work around this, users are forced to delete the log files (through JHS configs) frequently even though disk space itself is not a concern.

Proposal

Combining the per-node-aggregated log files into a single log file per application would allow users to keep more logs without stressing out the NameNode.

The per-node aggregated log files currently reside all in the same directory, per application and are in a binary format (`AggregatedLogFormat`, which is essentially a `TFile`). Each file consists of some header metadata (ACLs, owner, and version), followed by key-value pairs. Each of the keys is a Container ID and the values are the logs (stdout, stderr, and syslog are all here) of that container and their lengths.

The end goal is to have a procedure (described later) which will combine these aggregated log files into a single combined aggregated log file. The format of this file can be similar to the aggregated log files; we can essentially just "dump" the information from the aggregated log file into it, which gives us the advantage of reusing most of the log parsing code that already exists. However, the JHS currently finds the logs for a specific Container ID in an aggregated log file by searching through it -- this is somewhat inefficient and would be even worse for the larger combined file. The solution to this is to have a second file which stores the index, which can simply be a mapping of the container IDs with their offsets and lengths in the combined log file; we can also put the header metadata in here as well. Here's what the format of the index file would roughly look like:

```
|-----------------------------|
|Version                      |
|ACLs                         |
|Owner                        |
|ContainerID,offset,length    |
|...                          |
|-----------------------------|
```

There isn't currently a practical way to know when all NMs used by an Application are done uploading their aggregated logs from an outside perspective; this is especially true in the case

of a long running application (which will be addressed in YARN-2548). As a result, we need to have a distributed approach to combining the logs, with each NodeManager appending its own log sometime after it's done uploading. We'll have each NM maintain a list of applications for which their logs have finished uploading, but have not been appended yet. A new `LogCombinationService` will be added to each NodeManager that will look at this list and append its log file for that application at some interval. ZooKeeper/Curator will be used to store one lock per Application to ensure that only one NodeManager can append per application at a time. More succinctly, we'd have these procedures:

`LogAggregationService` (modified)

1. Aggregate and upload logs for Application_X (original behavior)
2. Add Application_X to list (new behavior)

`LogCombinationService` (new)

1. For each application in the list
    a. Try to acquire lock for Application_X
        i. Yes → combine logs for Application_X, delete original, remove from list, release lock
        ii. No → skip for now and try again after some interval

As the NodeManager runs through its aggregated log file, it will append the log for a container and then write the Container ID, offset, and length to the index. If an NM dies while appending, that container will be missing from the index. Any incompletely written log data (or log data with a missing index) will essentially be "dead space" because we can't get rid of it or overwrite it in HDFS; this isn't a problem because the reader and writer code can ignore this space. In the JHS/ATS reader, this will trigger a fallback behavior to serve the missing container's logs from the original aggregated log file (which won't have been deleted yet because the NM wouldn't have gotten around to it before dying). Additionally, if that NM later tries again and succeeds in appending its log file, the readers will then be able to find it in the index. The list of applications to append can be persisted in the NMStateStore so that it gets recovered on restart.

Because of the bottleneck at appending, we'll still keep the aggregated log behavior so that the user can get logs sooner. The JHS (and presumably eventually ATS) code for retrieving logs will be updated to check the combined aggregated log's index file for a Container ID and only fallback to the aggregated log if it can't find that ID. This will ensure that the user can get all of the logs as soon as they are aggregated, while they're being combined, and after they're combined.

The combined aggregated log files would be located in the same place as the aggregated log files. Some advantages of this include: these directories are already created by the `LogAggregationService` with the correct permissions, no need to duplicate the directory

structure, and the `AggregatedLogDeletionService` will handle cleaning up combined aggregated log files with no additional code changes.

The work can be split up into 3 tasks:

1. Create the `CombinedAggregatedLogFormat` Reader and Writer
2. Create the `LogCombinationService` and modify the `LogAggregationService`
3. Modify the JHS and CLI to be able to read using either the `CombinedAggregatedLogFormat` or the `AggregatedLogFormat`

We will include some configuration options, such as:

- Enable/Disable combining
- ZooKeeper/Curator configs
- Frequency of combining service

Follow-up Work

- Logs from long running services (which are uploaded on a per container basis instead of per node) are not considered here at this time (YARN-2548 can update it to handle them). For now, they should be left alone by the combining code. Support for this case should be doable by updating the Reader to handle multiple index entries for the same Container ID.