ATS Write Pipeline Design Proposal

Robert Kanter

Motivation

A recent discussion involving members of the community concerned with the ATS has brought out several concerns with the current design and additional use cases that we'd like it to support (see Sangin Lee's ATS-meet-up-8-28-2014-notes.pdf on YARN-1530). This document will describe a revised design proposal to enhance the way that clients publish events to the ATS in a way that is more scalable and reliable. This design is pluggable, so it should be able to fit the needs of any user.

Proposal

One of the scalability and reliability issues with the ATS is that all events are going through an unreliable communications channel (i.e. the write pipeline) to a non-scalable single node. One of the ideas discussed during the meeting was making the TimelineClient pluggable, which we're going to expand on. In addition to the pluggable TimelineClient, there would also be a complementary pluggable interface in the ATS for "importing" the data and storing it locally (in other words, we'd have pairs of implementations: one for the TimelineClient and one for the ATS). We can keep the REST implementation as a simple low-load solution, but add additional implementations:

1. HDFS-based implementation: The TimelineClient writes events to a file in HDFS, much like the current behavior with the JHS, with one file per application. The events would simply be serialized and appended to the end of its entity's file. The ATS-side can then periodically read through the files and bring in the new data to its store. It can either keep track of its current location in the file, or it can use another file as a write lock and have it move the existing events file elsewhere before importing (the TimelineClient would write subsequent new events to a new file at the original location). Essentially, we're using HDFS as a scalable, highly available, and distributed communication channel.
    a. Pros:
        i. Scalable because each AM writes to its own file in HDFS
        ii. Reliable because HDFS is reliable
        iii. The ATS is removed from the write path; ATS availability has no effect on the job
    b. Cons:
        i. Redundant data written to HDFS; though we can probably clean this up at some point
        ii. HDFS files have a replication factor of 3 by default; for large jobs, this can end up being a large amount of data which adds additional overhead
        iii. There will be a lag between a history event occurring and it showing up in the ATS; the amount of lag will depend on the frequency of importing.

Higher frequency will lower this lag but would put additional stress on the ATS

A modification of this design could be done where instead of having a separate ATS store that we import into, we could have the ATS read completed data directly from HDFS files and current data directly from the running AM. We can probably make this more efficient by adding in some kind of cache in the ATS.

2. Direct-Writing implementation: The TimelineClient can write directly to the backing store of the ATS. The exact implementations will depend on the store implementation being used.
    a. Pros:
        i. The ATS is removed from the write-path; ATS availability has no effect on the job
        ii. The ATS only has to worry about serving data
        iii. No redundant information
    b. Cons:
        i. The administrator will need to choose a store that can handle the load and reliability requirements
        ii. Configuration may get more complicated; classpath may need additional jars/files

These are two possible approaches that we can either use or not use. Additional implementations can also be created as needed.