Launching Kafka with Apache Mesos

Apache Mesos Scheduling and Task Execution

In the initial implementation of this, Apache Mesos will be tested to create this abstraction and an implementation of a Kafka-Mesos project that will bridge /glue them together.

The tested code for the kafka.mesos implementation will be under https://issues.apache.org/jira/browse/KAFKA-1207

The development environment for this is active https://github.com/stealthly/kafka-mesos with the goal of having a fully functional version available to contrib back as a patch.

Once it is working outside of the Kafka code base an option is to move all of the changes into core/src/main/scala/kafka/grid/mesos and any other changes to existing Kafka code captured in https://issues.apache.org/jira/browse/KAFKA-1206 ... So 1207 is all new code for Mesos Framework and 1206 are any changes we are making in existing Kafka code to support (introduce risk) for doing that.

kafka.mesos has a few parts to it which is implemented overall

1) The framework

The framework is responsible for managing the scheduled jobs that are running.

2) The scheduler

The scheduler understands what is running on slaves (so what resources are available) and will decide what to-do. This is really awesome because you can wait until you have a certain amount of resources and it is as a certain time before automatically launching Kafka producers, consumers or brokers. There is a lot to benefit from here for folks and we need to provide some hooks for them to operate it (i.e. bin/kafka-mesos-scheduler-start.sh w/ options)

3) The executor gets information from the framework (like environmental variables / configuration information) and launches tasks with it.

The existing implementations I have been looking at for reference are for Hadoop https://github.com/mesos/hadoop/tree/master/src/main/java/org/apache/hadoop/mapred and Storm https://github.com/mesosphere/storm-mesos/tree/master/src/jvm/storm/mesos along with the examples from https://github.com/apache/mesos/tree/master/src/examples, https://github.com/mesosphere/sample_mesos_executor and https://github.com/guenter/mesos-getting-started al so as guide for making the system work together.

I am tending right now to go with the environmental value approach. This uses the mesos.proto structures to pass an array/list of value within the enviornment structure. We can set this in the scheduler and read it the Executor during the call of

```
def registered(executorDriver: ExecutorDriver, executorInfo: ExecutorInfo, frameworkInfo: FrameworkInfo,
slaveInfo: SlaveInfo)
```

`executorInfo.command.environment.variables`

`variables are a list of name/value string pairs`

The specific behind these structures are in the Mesos protobuf file https://github.com/apache/mesos/blob/master/include/mesos/mesos.proto?source=c

The Scheduler and Executor API http://mesos.apache.org/documentation/latest/app-framework-development-guide/

More on the Mesos Architecture

Example of resource offer

The figure below shows an example of how a framework gets scheduled to run a task.

blocked URL

Let's walk through the events in the figure.

1. Slave 1 reports to the master that it has 4 CPUs and 4 GB of memory free. The master then invokes the allocation policy module, which tells it that framework 1 should be offered all available resources.
2. The master sends a resource offer describing what is available on slave 1 to framework 1.
3. The framework's scheduler replies to the master with information about two tasks to run on the slave, using <2 CPUs, 1 GB RAM> for the first task, and <1 CPUs, 2 GB RAM> for the second task.
4. Finally, the master sends the tasks to the slave, which allocates appropriate resources to the framework's executor, which in turn launches the two tasks (depicted with dotted-line borders in the figure). Because 1 CPU and 1 GB of RAM are still unallocated, the allocation module may now offer them to framework 2.

In addition, this resource offer process repeats when tasks finish and new resources become free.

While the thin interface provided by Mesos allows it to scale and allows the frameworks to evolve independently, one question remains: how can the constraints of a framework be satisfied without Mesos knowing about these constraints? For example, how can a framework achieve data locality without Mesos knowing which nodes store the data required by the framework? Mesos answers these questions by simply giving frameworks the ability to **reject** of fers. A framework will reject the offers that do not satisfy its constraints and accept the ones that do. In particular, we have found that a simple policy called delay scheduling, in which frameworks wait for a limited time to acquire nodes storing the input data, yields nearly optimal data locality.

You can also read much more about the Mesos architecture in this technical paper.