Worker Decommissioning Design Document

Out Of Date: See updated 2019/2020 Doc @ https://docs.google.com/document/d/1xVO1b6KAwdUhjEJBoIVPl9C6sLj7oOveErwDSYdT-pE/edit

Goals:

Spark's resilient computing mechanism is able to handle executor and driver failure, but in some cases we can know a node will be decommissioned shortly, and avoid schedualing tasks on a node that is about to be removed. Furthermore, if there is sufficient time Spark may be able to avoid recompute of persisted data by copying it to nodes which are not scheduled for decommission.

Motivation:

Decommissioning can come in many flavours, on cloud computing services such as EC2 or GCP, spot-instances or preemptible instances can be removed when there is high demand for instances. Similarly in-house systems such as YARN have the concept of killing containers to make way for higher priority tasks. In a traditional cluster on-prem computing environment sometimes nodes or racks will be decommissioned for service.

Related:

There has been related work (specific to YARN) to allow for shutting down of executors while keeping the shuffle service alive. This is not sufficient for other systems where an entire VM or host may be preempted.

Parts:

In part 1 we will add support for tracking workers which are decommissioning and avoiding scheduling new tasks on workers.

In part 2 we will copy cache/persisted blocks from the decommissioned host.

Optionally, in part 3 we will copy shuffle files from the decommissioned host.

Part 1 Design:

To allow the system to be general to multiple providers, while still useful in its initial state, a separate component will be responsible for notifying the worker that it is being decommissioned. An implementation for EC2 will be provided using polling and passes a **DecommissionSelf** message to the worker process on the host. To simplify interfacing for systems not built on the JVM a shell script interface will also be provided to send a **DecommissionSelf** message.

The worker, when receiving a **DecommissionSelf** message will update its internal state to decommissioned and avoid launching any new executors on the worker. The existing executors will also update their state to decommissioned and avoid launching any new tasks. The worker will also send a **WorkerDecommission** message to the master.

The master, when receiving a **WorkerDecommission** message will move the workers state into **WorkerState.DECOMMISSIONED** which will prevent it from making any resource offers on that worker. The application info for each application with an executor on that worker will be updated remove that executor. For each executor on the worker the master will send an **ExecutorDecommissioned** message with the executor state of **ExecutorState.DECOMMISSIONED**.

On receiving **ExecutorDecommissioned** the driver will notify the scheduler backend which will add the executor to an internal executorsPendingDecommission set and no longer schedule tasks on the decommissioned executor.

Phase 2:

When the BlockManager & Shuffle service are notified by the worker they are being decommissioned they will respectively notify the BlockManagerMaster & ShuffleManagerMaster that they are being decommissioned. A viable alternative to this approach would be treating the BlockManager as being decommissioned when worker sends the message (as currently implicitly happens during executor lost), however in that event it could be more difficult to trigger the logic for block migration.

To prevent executors on the worker being offered to more applications, once the master program receives a decommissioning message from the worker it will set the workers state to DECOMMISSIONED and only ALIVE workers are considered.

For applications which already have executors on the worker, for each executor an executor lost message will be sent to the driver for with the loss reason set to Decommissioned. This will remove the executor from the TaskScheduler's internal data structures. The logic of TaskSchedulerImpl will treat Decommissioned in a similar way as LossReasonPending (e.g. allowing current tasks to return).

Out of scope:

When deployed on Kubernetes, the cluster manager can trigger a script to indicate a pod is being shutdown/moved. Integration with other cluster managers, native cloud provider APIs (e.g. EC2) is optional follow on work which may eventually live inside of Spark or as external libraries.

Avoiding scheduling new tasks is designed to work with the StandaloneScheduler, other schedulers should have their own mechanisms for handling worker decommission. Note: other schedulers can still send a DecommissionSelf message to a specific worker if they wish to the worker to no longer accept new tasks, however they are responsible for ensuring the worker is not offered. Phase 2 & 3 be able to behave the same way - since it is not specific to the scheduler.

Recommissioing is not directly supported, instead the worker (if preempted and restored on the same node) should restart. Since most decommissioning involves loss of local storage, this is viewed as reasonable. Existing Spark mechanisms for temporary loss of communication with worker should be sufficient to handle worker migrations, decommissioning is not intended for strict migrations.