RM HA Phase1: Cold Standby

Motivation

Without RM HA, the ResourceManager is a single-point of failure in a YARN cluster.

1. HA allows tolerating unplanned events such as a machine crashes; otherwise, the cluster would be unavailable until an operator restarts the ResourceManager.
2. HA also enables planned maintenance events such as software or hardware upgrades on the ResourceManager machine would result in windows of cluster downtime.

To address these use cases, we need to add high-availability support for RM, in a way that is consistent with NameNode HA if possible.

High-Level Design

In Phase 1, we focus on the Standby approach to address the above-mentioned cases: the Active RM serves the clients and manages the AMs and NMs, and the Standby takes over in case the Active fails.The Standby approach is also consistent with that of HDFS.
Note: More sophisticated approaches like Active/Active for zero cluster downtime are beyond the scope for Phase 1.

The Standby's implementation determines the failover-time, we consider the following alternatives:

Hot/Cold Standby

1. Hot Standby: Both RMs maintain the state of the cluster (NMs) and applications, so the Standby can immediately take over in case the Active fails. However, this requires either (1) the RMs to receive heartbeats from NMs, and events from the AM and Client in the same order or (2) the Active RM to update the Standby on every state change. Both options increase the overhead in the cluster, and might adversely affect the responsiveness.
2. "Warm" Standby: The Standby can periodically read RMStateStore updates to have the application-related state, and start scheduling as soon as the NM heartbeats come in. Depending on the size of the RMStateStore and the time taken to load it, preloading it could be worthwhile.
3. Cold Standby: The Standby maintains no state and constructs state when it takes over as the Active. During failover, the Standby reconstructs the cluster state through NM heartbeats, and reads the application-related information from the RMStateStore. Also, recovering this state leads to longer cluster downtime compared to the Hot Standby case. Note that most of the work involved to implement the Cold Standby is required to implement warm/hot standby as well.

For Phase 1, we propose Cold Standby as it is less involved and anyways required for warm/hot standby. That said, we keep the more sophisticated approaches in mind as we might want to

support them eventually.

Design/Implementation Details

The following items constitute the HA solution and need careful consideration as well.

RM Restart

RM Restart deals with storing/retrieving the RM state for continued functioning post restart. This also addresses the semantics of (1) how the NMs and AMs interact with the RM post restart, and (2) how the work for the duration the RM is down (or failing-over) is preserved. Given the tight dependence on the Restart, the Restart semantics might have to be revisited as the high-level approach for HA changes.

YARN-128 implements RM Restart, bulk of the functionality is already implemented. The state of the RM (application-specific context) is written to the `RMStateStore`. On restart, the RM reconstructs the state from this store and through NM/AM heartbeats. YARN-128 implements both file-based and ZooKeeper-based state stores.

Leader Election

ZooKeeper-based `ActiveStandbyElector`, as in HDFS, can be used to pick the Active RM from among the RM nodes.

Automatic Failover

In case the Active RM fails, the failover mechanism should elect a new leader and activate it. In the context of a cold standby, by activating an RM, we mean restarting the RM on the new node which automatically loads the RMStateStore. In the context of more sophisticated standby mechanisms (warm, hot, active), it translates to starting certain services on the RM.

For Phase 1, we can use the `ZKFailoverController` (exactly as in HDFS) to monitor the health of the NMs, elect a new Active in case of a failure, and the actual failover.

Fencing

In an Active/passive architecture, to avoid the split-brain situation, only one RM should be able to (1) modify the RMStateStore, (2) handle NM heartbeats, (3) schedule containers on AM's allocate(), and (4) communicate with the clients. To ensure this, we need a fencing mechanism. Possible alternatives include:

1. ***sshfence***: HDFS non-QJM HA provides this where one can SSH to the target node and use fuser to kill the process listening on the service's TCP port.
2. ***shell***: Again HDFS non-QJM HA provides this where one can run an arbitrary shell command to fence the Active ResourceManager.
3. ***zk-store-lock***: While this approach is not directly supported by HDFS, in principle, it is along the lines of QJM's fencing. RM uses RMStateStore much like NN uses QJM. QJM limits the access to the journal to a single (the real Active) NN. Similarly, the ZK-based RMStateStore can limit the access to a single RM. An RM sets itself to Standby mode

when it loses access to the store.

For Phase 1, we focus on zk-store-lock mechanism.

Client-side changes

RPC: The clients need to determine which among the RMs is Active at any point; the client should try to connect to each of the individual RMs and see which one is responsive. We should be able to reuse the HDFS retry policy implementation with minor refactoring and changes.

HTTP: Clients should implement their own retry/try-another-RM logic to figure out the Active RM for HTTP requests (REST API). A load-balancer/Virtual-IP is required to be able to support automatic failover for HTTP, which is beyond the scope of Phase 1.

Wrapper vs Vanilla RMs for HA

We see two alternatives to implement the above design:

1. Vanilla RMs: Add HA-related states to the RM, and include all of the HA logic in the RM implementation itself. While it makes the RM code more complex, it allows fine-grained control of RM services and is consistent with the HDFS approach.
2. Wrapper approach: We can introduce a wrapper (`RMHADaemon`) that participates in the HA election/failover. When a daemon is picked to be the Active, it starts an RM-instance. Advantages:
    a. One major advantage of this approach is that it keeps all of the HA-related code (leader election, failover, etc.) separate from the RM implementation itself, making it easier to reason about and maintain.
    b. It can be extended to support warm/hot configurations - again the logic of what RM services need to be running resides in the wrapper. We will need to add states to the RM and start those services as part of state transitions, initiated from calls by the wrapper.
    c. HTTP redirection: The Standby RMHADaemon can forward client-requests arriving at the RM Web UI port to the Active RM. As long as the Standby HADaemon is running, the clients continue to be able access the Web UI irrespective of whether they talk to the Active/Standby.

    Disadvantage: The wrapper is an extra daemon to support and diverges from the HDFS HA approach.

For Phase 1, we would like to use the Wrapper approach so we can keep the HA logic separate from the RM. Also, it doesn't affect users who don't want to use HA at all.