Design for Reservation (RAYON) HA

Authors: Anubhav Dhoot, Subru Krishnan

Goals

YARN-1051 added ability to reserve cluster capacity. This design covers adding high availability (HA) support for this feature to handle RM failovers.

Introduction

At a high level **ReservationSystem** accepts user requirements of resources over time and matches them to dynamic queues with varying capacity over time that satisfies the requirements called as a **ReservationAllocation**. Thus the ReservationSystem takes as input the resource requirements over time and translates into a set of reservations, which are saved in a **Plan**. Each reservation maps to a reservation queue in the scheduler whose capacity is adjusted periodically by a **PlanFollower** to match the reservation. Applications are then attached to these reservation queues.

On HA failover we need to recover the original user requirements and the reservations. We need to recover the original user requirements in order to allow re-planning in the future (re-planning is needed for example because of change in available resources). We need to recover the reservation queues before we recover applications in those queues because an application always expects a valid queue.

We make some simplifications for the recovery implementation. Failover time is considered small enough that there are no significant changes in the plan. This can be visualized as freezing the plan follower steps during failover time and postponing any transitions to after the failover is done. Also the times of the ResourceManager are reasonably synchronized such that we do not have to worry about plans follower going backwards in time. More about mitigating this in section about Recovering the state.

Implementation Details

1) Simplifying the state

The in-memory state for reservations is stored per Plan. A Plan corresponds to each reservable queue and is implemented in memory as **InMemoryPlan**. The InMemoryPlan stores the resources allocated over time as an instance of a **ReservationAllocation** for each reservation (see the code snippet below).

```
public interface ReservationAllocation {
  public ReservationId getReservationId();
  public ReservationDefinition getReservationDefinition();
  public long getStartTime();
  public long getEndTime();
  public Map<ReservationInterval, ReservationRequest> getAllocationRequests();
  public String getPlanName();
  public String getUser();
  public boolean containsGangs();
  public long getAcceptanceTime();
}

public abstract class ReservationDefinition {
  public abstract long getArrival();
  public abstract long getDeadline();
  public abstract ReservationRequests getReservationRequests();
  public abstract String getReservationName();
}

public abstract class ReservationRequests {
  public abstract List<ReservationRequest> getReservationResources();
  public abstract ReservationRequestInterpreter getInterpreter();
}

public abstract class ReservationRequest {
  public abstract Resource getCapability();
  public abstract int getNumContainers();
  public abstract int getConcurrency();
  public abstract long getDuration();
}

public enum ReservationRequestInterpreter {
  R_ANY,
  R_ALL,
  R_ORDER,
  R_ORDER_NO_GAP
```

The **ReservationDefinition** represent the original requirements of the user. It has a list of ReservationRequest and an enum representing how to interpret them (ReservationRequestInterpreter). If a reservation is thought of as a skyline, each ReservationRequest indicates the shape and flexibility of each block of the skyline and the ReservationRequestInterpreter describes how to put these blocks together.

The Map<ReservationInterval, ReservationRequest> getAllocationRequests() represents assigned allocation details. This ultimately translates to a specific cluster capacity at any time that is set to the dynamic queue. This means we can simply represent this as Map<ReservationInterval, Resource> instead which simplifies the state to be stored for the allocation.

2) When to store the state

Whenever the Plan is updated to add remove or update reservation we need to store the changes in the state store. This will be achieved by the InMemoryPlan persisting the ReservationAllocation in the RMStateStore.

3) Recovering the state

In ResourceManager the state is recovered at ResourceManager::recover. Before we call RMAppManager::recover to recover applications, we call ReservationSystem::recover to recover the in memory state and the reservation queues for all reservations that have a start time less than the current time plus a safety margin to accommodate the current ResourceManager's time being behind the previous ResourceManager. This will ensure that all applications have their queues pre-created. The PlanFollower needs to delay its running to this point to ensure state is recreated completely. So if HA is enabled, PlanFollower will not actually start until the recovery is complete.

4) Store details

We will implement methods on the RMState to maintain the Reservation State just like RMDTSecretManagerState, RMAppState and AMRMTokenSecretManagerState. Logically the state will be organized as shown

```
/reservationsRootNode/
    /plans/
        planName1/
        planName2/
            reservations/
                id1/
                    ReservationAllocation reservationAllocation(
                        user,
                        ReservationDefinition contract(
                            long arrival,
                            long deadline,
                            repeated<ReservationRequest(
                                    Resource capability, int numContainers, int concurrency, long duration),
                            ReservationRequestInterpreter interpreter,
                            String name),
                        startTime,
                        endTime,
                        repeated<ReservationInterval(long startTime, long endTime), Resource>,
                        hasGang,
                        acceptedAt)
                id2/…
```