NM Restart Design Overview

This document describes the design of the NodeManager restart work under YARN-1336 and its sub-JIRAs. NM restart is a feature where the NodeManager can be restarted without losing the active containers running on the node.

At a high level, the NM stores any necessary state synchronously to a state store as it processes requests. When the NM restarts it recovers by loading state for various subsystems and those systems perform recovery processing using the loaded state. Initially leveldb was chosen as a state store backend since it has some useful features that apply to NM restart:

- Simple key-value store methodology is easy to map to the needs of a state store
- Log-based storage and atomic individual key or batch commits makes it robust to application crashes
- Random key storage is very fast, so it reduces the impact to normal NM processing
- Already in use as the default backend for the ApplicationTimelineServer

Enabling NM Restart

NM restart is enabled by setting yarn.nodemanager.recovery.enabled to true and yarn.nodemanager.recovery.dir to the local filesystem directory where the state should be stored. If recovery is not enabled then the NM will use a null state store service to minimize the "if (recoveryEnabled)" checks throughout the code. The state store interface also provides a canRecover method as a convenience for code to quickly check if the state store supports recovery. The null state store returns false for this method, and all other stores will return true.

Localized Resource State Storage and Recovery

Local resource state is preserved by calling startResourceLocalization(user, appId, rsrcProto, localPath) on the state store service when the nodemanager starts to download a resource. For public resources the user and appId should be null, and for private resources the appId should be null. localRsrcProto is the original resource request associated with the container request.

Local resource requests are stored in the leveldb database with the following key templates:

Public Resources:

Localization/public/started/<*local filesystem path*>

Private Resources:

Localization/private/<*user*>/filecache/started/<*local filesystem path*>

Application Resources:

Localization/private/<*user*>/appcache/<*applicationId*>/started/<*local filesystem path*>

The value stored for each of the above keys is the resource request protocol buffer from the client that describes the resource to localize.

A successful resource localization is indicated to the state store by calling finishResourceLocalization(user, appId, localizedRsrcProto). For the leveldb store this writes the localizedRsrcProto (which contains the unpacked resource size among other things) to a completed key. Completed keys have the same form as the started keys except "/started/" is replaced with "/completed/". During recovery this allows us to distinguish resources that completed localization from ones that were in-progress but did not complete.

Removal of a localized resource is indicated to the state store by calling removeLocalizedResource(user, appId, localPath). This removes the corresponding started and completed keys from the leveldb database.

During recovery the loadLocalizationState method loads the localized resource state from the database. LocalResource objects are re-created for each successfully completed resource and added to the appropriate LocalResourceTracker, while any in-progress resources are deleted from the local disk and forgotten. Localized resources that did not complete will be re-requested by containers during container recovery. Similarly localized resources that were requested but never started will also be re-requested during container recovery and therefore do not need to be persisted explicitly by the localization service.

Application State Storage and Recovery

As applications are initialized on the NM the storeApplication method can be used to persist information about the application. When the RM indicates an application has finished then this state can be persisted by calling the finishApplication method. When the NM no longer needs to track an application then the removeApplication method can be used to remove the application state from the state store.

During recovery the loadApplicationsState method loads the application states from the state store. The state for each application indicates whether the application has finished, i.e.: no more containers will be launched but it may be undergoing log aggregation processing. As each application is recovered, an ApplicationImpl instance is created and init events are triggered to re-initialize the bookkeeping for the app within the NM. If an application is finished then an ApplicationFinishedEvent is dispatched to the ApplicationImpl after containers are recovered to trigger any log aggregation and cleanup processing for the application.

Container State Storage and Recovery

As container start requests are received the addContainer method can be used to persist information about the container start request to the state store. As the container is launched the setContainerLaunched method should be used to mark the container as launched in the state store. When a container completes the setContainerCompleted method should be called, and similarly when a container is killed the setContainerKilled method should be called. Any updates to the container diagnostics can be persisted by calling the setContainerDiagnostics method.

Finally when the NM no longer needs to track a container the removeContainer method can be called to remove the container state from the state store.

During recovery the loadContainerState method is used to load the state of all containers being tracked by the NM. A container is loaded with a number of state attributes:

- Requested
- Launched
- Killed
- Completed

For each container a ContainerImpl instance is created. If the container is marked completed then the instance transitions to the DONE state and sends appropriate container finished events for log aggregation. If the container is marked killed but not launched then it also transitions to the DONE state.

If the container is marked as launched then the container proceeds through the normal container startup transitions (i.e.: requesting local resources, launching, etc.). We process local resource requests as normal to fixup the reference counting for local resources. When it comes time to launch a recovered container a RecoveredContainerLaunch is used instead of a normal ContainerLaunch. This launcher does not re-run the container but rather attempts to reacquire the previously launched container. It does this by locating the PID file created by the container executor and asking the executor to reacquire the running process. If the process is running then RecoveredContainerLaunch will periodically poll to see if the process has exited, and once it has it looks for the exitcode file created by the container executor to obtain the exit code from the container. If the process is no longer running then it searches for an exitcode file to obtain the exit code for the container. If no exit code exists then the reacquisition fails and the container is reported as LOST (exit code -154). Once reacquired if the recovered container is also marked as killed then the container is killed.

To support recovery of exit codes from containers that completed while the NM was restarting a change was made to the way containers are launched by the container executor. Previously the container executor would perform some pre-processing and then exec a bash shell script to start the container process. The container executor now runs the shell script in a subprocess and then echos the exit code of the container process to an exitcode file. This allows the NM to recover the exit code of containers that have completed while it was down or in the process of restarting.

NM Token State Storage and Recovery

The master keys for NM tokens are persisted by calling the storeNMTokenCurrentMasterKey and storeNMTokenPreviousMasterKey methods to store the current and previous master keys, respectively. The master key currently associated with a particular application attempt can be persisted by calling the storeNMTokenApplicationMasterKey method.

NM tokens are stored under the NMTokens/ key hierarchy in leveldb. The master keys are stored under NMTokens/CurrentMasterKey and NMTokens/PreviousMasterKey and the master keys associated with individual application attempts stored at NMTokens/<*appAttemptId*>.

During recovery the loadNMTokenState method loads all of the NM token master key states. This state is then used to update the NMTokenSecretManagerInNM instance with the appropriate master key state and repopulate the map of application attempts to master keys.

Container Token State Storage and Recovery

The master key for container tokens are persisted by calling the storeContainerTokenCurrentMasterKey and storeContainerTokenPreviousMasterKey methods to store the current and previous mater keys, respectively. The expiration time for a particular container token can be persisted by calling the storeContainerToken, and the state for a particular container token is removed from the store by calling removeContainerToken.

Container tokens are stored under the ContainerTokens/ key hierarchy in leveldb. The master keys are stored under ContainerTokens/CurrentMasterKey and ContainerTokens/PreviousMasterKey, respectively. The expiration time for a particular container token is stored under ContainerTokens/<*containerId*>.

During recovery the loadContainerTokenState method loads all of the container token state. The recovered state is then used to update the NMContainerTokenSecretManager instance to repopulate the master keys and rebuild the map of expiration times and container IDs to track container tokens that have been used.

Deletion Service State Storage and Recovery

The deletion service tracks deletion tasks that are scheduled to execute at various times. These are persisted to the store by calling the storeDeletionTask method and passing a protocol buffer that describes the deletion task and any successor task IDs that should be triggered after it executes. Once executed deletion tasks can be removed from the store by calling the removeDeletionTask method.

During recovery the loadDeletionServiceState method loads all of the persisted deletion tasks. This state is then used to recreate the deletion tasks in the DeletionService instance and reconnect the deletion task dependencies based upon the stored successor task IDs.

Log Aggregation Recovery

There isn't an explicit state storage for the log aggregation service. As applications are recovered, the application init/finished events are propagated to the log aggregation service. As containers are recovered, container completion events are propagated to the log aggregation

service for any containers that were recovered as already finished. When the log aggregation service receives the application finished event it proceeds to upload the logs as normal. Any log aggregation that was in-progress when the NM restarted will resume from the start, overwriting any existing .tmp file.

Auxiliary Service State Storage and Recovery

There is rudimentary support for auxiliary services to support state storage and recovery. If recovery is supported, the NM will create a subdirectory in the NM state storage directory specific to that aux service, and it will call the setRecoveryPath method on the aux service before initializing the service. During initialization the aux service can call its getRecoveryPath method to determine if recovery is supported and where it should store/recover its state. If the recovery path is null then recovery is not enabled.

Additional References

Rolling upgrade talk at Hadoop Summit 2014 in San Jose:
http://www.youtube.com/watch?v=O4Q73e2ua9Y