OrgQueue for easy CapacityScheduler queue configuration management

Authors: Min Shen, Ye Zhou, Jonathan Hung

Motivation

YARN's capacity scheduler is currently configured using a file-based mechanism. To change the capacity scheduler's configuration, cluster administrators must manually edit this file, then refresh the queues via CLI. This is both cumbersome and error-prone.

The OrgQueue extension to the capacity scheduler provides a programmatic way to change configurations by providing a REST API that users can call to modify queue configurations. There are a few benefits here:

1. This enables automation of queue configuration management. We can check that the desired queue configuration changes are sane before reinitializing the capacity scheduler. This also enables use cases such as scheduled configuration changes without the need for cluster admin action.
2. We leverage the existing administer_queue ACL configuration to give the admins in this ACL the ability to change their queue's configuration via REST API based on what is best for their queue and their use cases.

Existing Capacity Scheduler Design

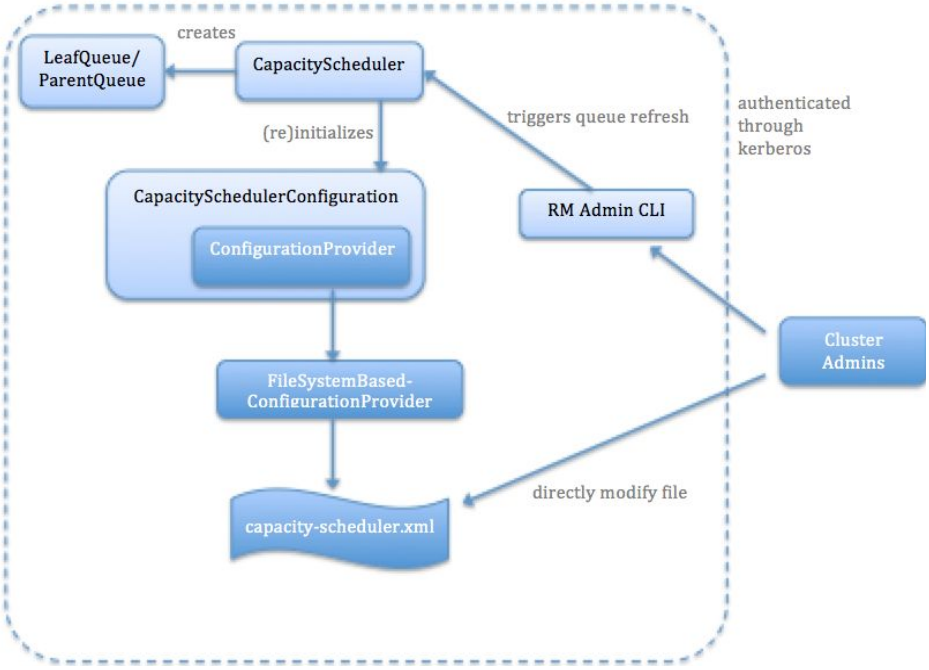

Fig 1. Existing Capacity Scheduler Design

Figure 1 shows the existing architecture of capacity scheduler's configuration. The only way to modify the configuration is by getting cluster admins to directly modify the capacity-scheduler.xml file, then having an admin trigger a queue refresh via the RM admin CLI. Having to go through cluster admins for each configuration change is not scalable, and doesn't make sense for configuration changes in an organization for which cluster admins have no context. Thus we extend the capacity scheduler to allow organizations running within a queue to change their queue's configurations without cluster admin intervention. To allow this, we need to implement a couple of main functionalities:

1. Perform checks to ensure the user-submitted configuration changes are valid before attempting to reinitialize the scheduler.
2. Since there are multiple people able to modify the configuration, the configuration's backing store must be changed to support concurrent modifications.

OrgQueue Design

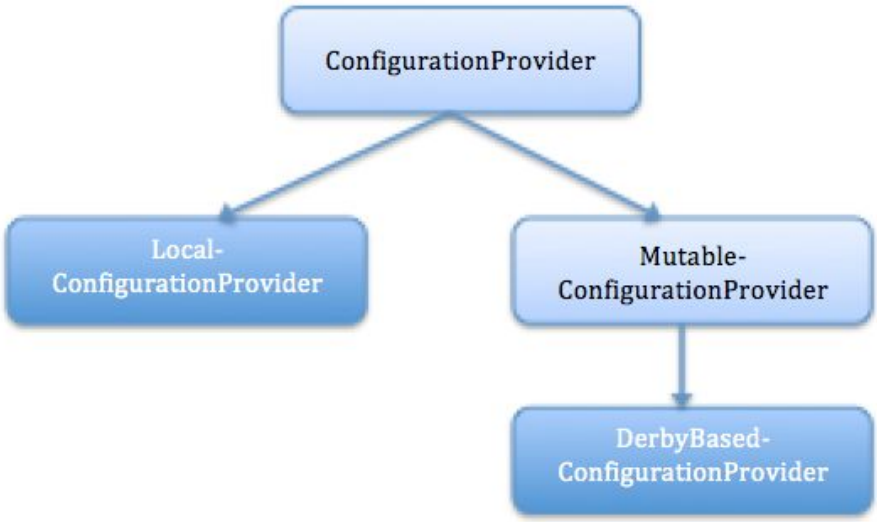

Fig 2. Configuration Provider Extensions

One of the requirements for OrgQueue is to provide a user-accessible way to update CapacityScheduler configurations so that queue administrators can change their queue's configurations without help of the cluster admins. To support this, we provide a MutableConfigurationProvider abstract class which exposes a list of methods which do the configuration modification. There is one method for each supported configuration change. For example, there is a method for changing the capacities of a queue's children, a method for adding a user-to-queue mapping, a method for adding a user/group to a queue's ACLs, etc.

Infrastructure providers can subclass this abstract class based on the way they want to store the capacity scheduler configuration.

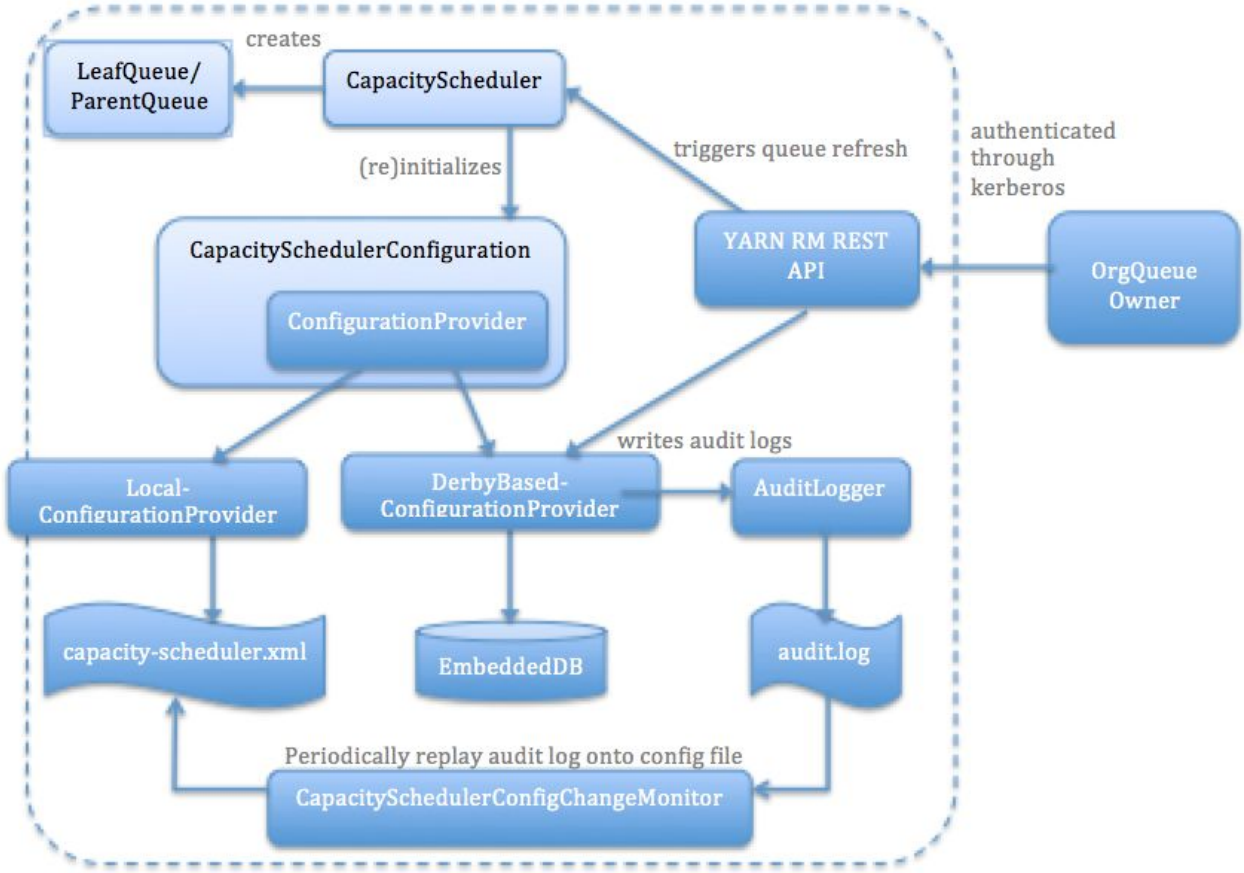

Fig 3. OrgQueue modifications on top of capacity scheduler

Embedded Database

Figure 3 shows the components of OrgQueue which we have implemented. We chose an embedded database as the backend store for capacity scheduler configuration since it automatically gives us support for transactional configuration changes. This is useful for two reasons:

1. Some configuration changes may modify multiple configuration values, and the DB allows us to modify them atomically.
2. Any preliminary checks we do before modifying the configuration and reinitializing the scheduler must occur in the same transaction as the modification and reinitialization.

Our extension uses Apache Derby's RDBMS as the database implementation. It contains the most up-to-date configuration values for the capacity scheduler. When RM starts up, if the embedded DB contains no data, it will load configurations using LocalConfigurationProvider (from capacity-scheduler.xml in the original way) and populate the embedded database using these configurations.

Derby Based Configuration Provider

The DerbyBasedConfigurationProvider class extends ConfigurationProvider to return the configuration stored in the DB. Note that the ConfigurationProvider only allows returning the configuration as an input stream. Since we can't return the configuration from the DB as an input stream, we add a method in ConfigurationProvider called getConfigurationObject, which DerbyBasedConfigurationProvider implements to return the full Configuration object from the DB.

There are two main components to the Derby based configuration provider:

1. **Providing the configuration to capacity scheduler** - When the capacity scheduler requests the Configuration object from the Derby based configuration provider, the Derby based configuration provider queries the DB to get values for each capacity scheduler configuration key and constructs a Configuration object based on these values. If a configuration value does not exist in the DB, it falls back to the value in capacity-scheduler.xml (using the existing LocalConfigurationProvider) and loads this value in the DB so it exists for the next time Derby based configuration provider queries the DB for this configuration key.

2. **Ability to modify configuration in embedded DB** - For each configuration modification method exposed by MutableConfigurationProvider, DerbyBasedConfigurationProvider's implementation calls the corresponding stored procedures containing the SQL queries doing the database modification. For each method, the same process occurs:
    a. Do preliminary checks, such as verifying the calling user has permission to make this configuration change, making sure the configuration change makes sense (e.g. reject if they request to change a subqueue's max capacity to 110), etc.
    b. Call the stored procedure which modifies the DB to make the configuration change.
    c. Reinitialize the capacity scheduler to make this change effective immediately.
    d. Log the change to the audit log.

As mentioned in the embedded database section, parts 2a-2c occur in the same transaction to support multiple queue admins making concurrent requests.

The Derby based configuration provider is also responsible for initializing the embedded DB schema via a provided initialize.sql script, as well as recreating the embedded DB schema if the embedded DB becomes corrupt.

Audit Log / Configuration Change Monitor

OrgQueue adds an additional AuditLogger component which writes every configuration change done by the Derby based configuration provider to both a DB audit and DB edit log stored locally on the RM.

1. The DB audit log's purpose is the same as any other audit log - to keep history of all operations: who, what, and when.

2. The DB edit log is the same but only stores operations done in the past N minutes (where N is some configurable number). The DB edit log is used by the configuration change monitor component.

The configuration change monitor is a service running as part of the capacity scheduler. It runs every N minutes (N as defined above). At each interval, the service will read capacity-scheduler.xml, read the DB edit log which contains all configuration changes from the past interval, apply the changes in-memory, then write the updated configuration back out to capacity-scheduler.xml. This ensures the capacity-scheduler.xml is up-to-date (within the past N minutes).

REST API

The REST API exposes the configuration change methods in MutableConfigurationProvider to users - there is one REST call for each method. We add to the existing REST API in RMWebServices. This is just a thin layer on top of the MutableConfigurationProvider.

The REST APIs we support (with some examples):

1. /cluster/queues/{queue}/update?key={key}&value={value} - updates a queue local configuration (e.g. user limit percent, maximum applications,...)
    a. /cluster/queues/root.a/update?key=ULP&value=10
2. /cluster/queue/{queue}/capacity?queues={queues}&capacities={capacities} - given a queue {queue} and all its children {queues}, set the capacities of the children to {capacities}
    a. /cluster/queue/root.a/capacity?queues=a1,a2&capacities=70,30
3. /cluster/queue/{queue}/maxCapacity?subqueue={subqueue}&maxcapacity={maxcapacity} - changes {queue}.{subqueue}'s max capacity to {maxcapacity}
4. /cluster/queue/{queue}/addQueue?subqueue={subqueue}&nested={isNested} - adds {subqueue} as a child of {queue}. If {isNested}, then {subqueue} is a parent queue with one child called {subqueue}_default.
5. /cluster/queue/{queue}/addUser?permission={permission}&isGroup={isGroup}&user={user} - adds a user to a queue's ACLs. isGroup denotes whether {user} is a group or not. permission is either SUBMIT_APPLICATIONS or ADMINISTER_QUEUE, adding the user to the specified ACL.
6. /cluster/queue/{queue}/removeUser?permission={permission}&user={user} - removes user from queue's ACL (specified by permission)
7. /cluster/queue/{queue}/getACL?permission={permission} - get ACL of queue specified by permission. (This is not a configuration setter API, but a configuration getter that might be useful for queue admins)
8. /cluster/queue/addMapping?queue={queue}&isGroup={isGroup}&user={user}&srcQueue={srcQueue}&delete={isDelete} - creates/deletes a mapping from {srcQueue} to {queue} for {user}.
    a. /cluster/queue/addMapping?queue=root.a&isGroup=false&user=foo&srcQueue=root.b - maps foo's application submissions to queue root.b to root.a

    b. /cluster/queue/addMapping?queue=root.a&isGroup=false&user=foo&srcQueue=root.b&isDelete=true - deletes the above mapping
    c. /cluster/queue/addMapping?queue=root.a&isGroup=false&srcQueue=&user=foo - maps foo's applications submitted to one of the scheduler's default queues (configured by **yarn.resourcemanager.scheduler.default.queues**) to root.a
9. /cluster/queue/reset - wipes the database and reloads it from capacity-scheduler.xml
10. /cluster/queue/{queue}/addLabelToQueue?label={label} - called for a label which no queue has existing capacity for. This gives {queue} (and all of its ancestors) 100% capacity for label {label}
    a. /cluster/queue/root.a.a1/addLabelToQueue?label=bar gives root, root.a, and root.a.a1 100% capacity for bar
11. /cluster/queue/{queue}/labelCapacity?queues={queues}&capacities={capacities}&label={label} - same as /cluster/queue/{queue}/capacity, but for partition {label}
12. /cluster/queue/{queue}/labelMaxCapacity?subqueue={subqueue}&maxcapacity={maxcapacity}&label={label} - same as /cluster/queue/{queue}/maxCapacity, but for partition {label}

Configuration Parameters for OrgQueue

This feature introduces a few new configuration parameters to yarn-site.xml:

1. yarn.scheduler.capacity.config.provider - this overrides yarn.resourcemanager.configuration.provider-class. The former is the configuration provider class, falling back to the latter if none is specified. In addition, the latter is used as the configuration provider on startup, when the DB is empty (as specified in Embedded Database section)
2. yarn.scheduler.capacity.config.change.monitoring_interval_msecs - the interval, in ms, which the configuration change monitor runs.
3. yarn.scheduler.capacity.config.path - the location on the RM where the embedded DB and audit/edit logs are stored. In addition, this is where the configuration change monitor expects to find capacity-scheduler.xml.

Alternative Configuration Providers

The configuration provider we implemented is Derby-based. However, we designed this to be pluggable. Here are a couple of other possible implementations:

1. LevelDB-based - leveraging existing Hadoop dependencies on LevelDB
2. MutableFile-based - similar to FileSystemBasedConfigurationProvider, but extends the MutableConfigurationProvider class

End-to-End Flow

To summarize, here is an example of the typical end-to-end sequence of events from the perspective of a queue admin:

1. An admin of organization "queueA" (specifically, a user whose username is specified in yarn.scheduler.capacity.root.queueA.acl_administer_queue) calls the "/cluster/queue/root.queueA/maxCapacity?subqueue=queueA2&maxcapacity=100"

REST API to change the max capacity of subqueue “queueA2” to 100 (maybe because they realize root.queueA.queueA1, another one of their subqueues, is under-utilized for most of the day - or some other reason).

2. The REST API passes these parameters to the updateQueueMaxCapacity method in MutableConfigurationProvider.
3. The MutableConfigurationProvider checks various conditions, such as the calling user having admin rights to root.queueA, and that max capacity is in the range [0-100].
4. The MutableConfigurationProvider changes root.queueA’s max capacity in the embedded DB to 100.
5. The capacity scheduler is reinitialized, which asks for the CapacitySchedulerConfiguration from MutableConfigurationProvider.
6. The MutableConfigurationProvider constructs the configuration object from the values in the embedded DB and returns it to the scheduler.
7. The capacity scheduler reinitialization proceeds normally, validating the returned configuration and reinitializing the queues.