OrgQueue: API-Based Scheduler Configuration Management

Motivation

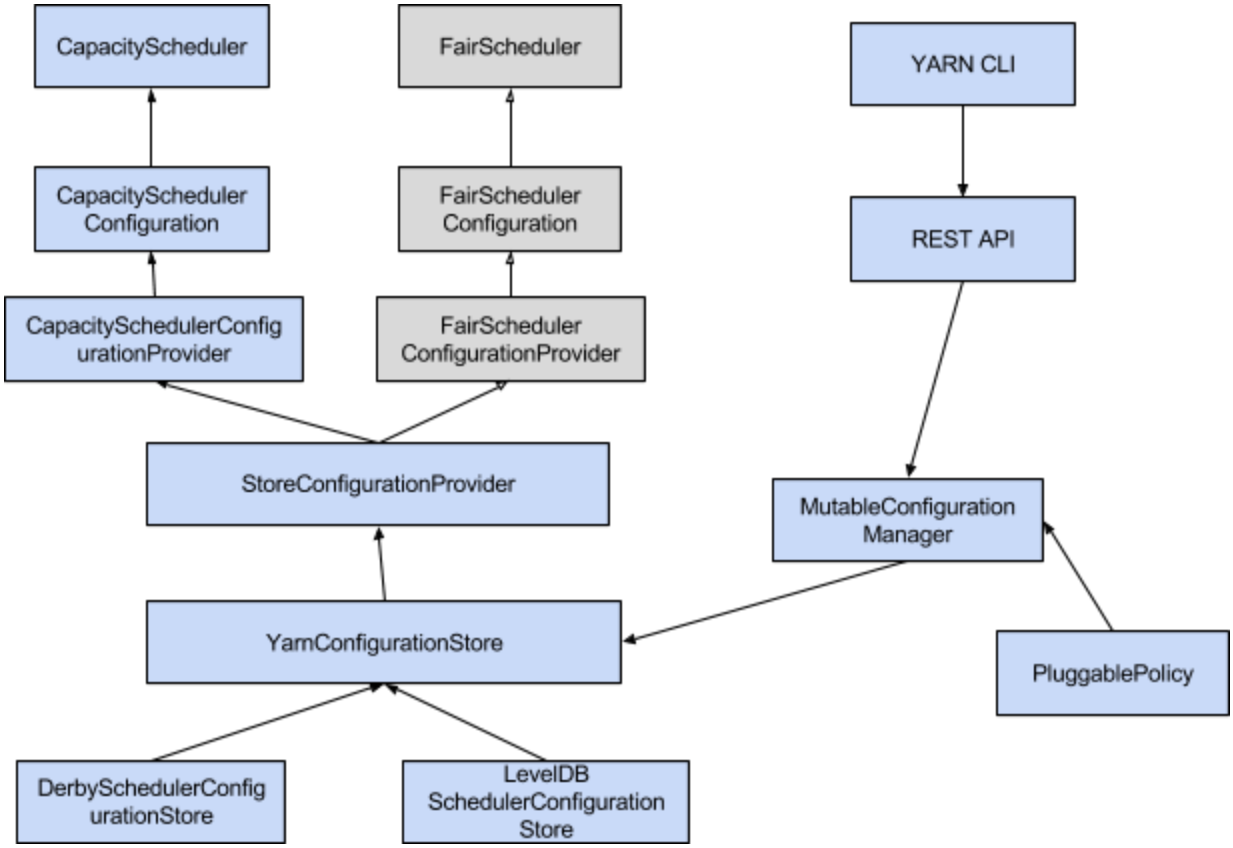
YARN's capacity and fair schedulers are currently configured using a file-based mechanism. To change the capacity scheduler's configuration, cluster administrators must manually edit this file, then refresh the queues via CLI. This is both cumbersome and error-prone.

The OrgQueue extension to the YARN 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. API-based instead of file-based scheduler configuration management allows queue administrators to self-manage their own queues. This frees cluster operators from being involved in every scheduler configuration change request such as sub-queue creation, sub-queue capacity management, and acl changes.
3. Deleting queues right now is not natively supported - the configuration file must be modified in multiple places to delete a queue, which is error-prone.

To solve these issues, we propose an alternative configuration management mechanism, in which scheduler configurations are stored inside a configuration store instead of the scheduler-specific XML config file. We further provide a set of REST APIs for retrieving and changing the scheduler configurations.

Overall Design

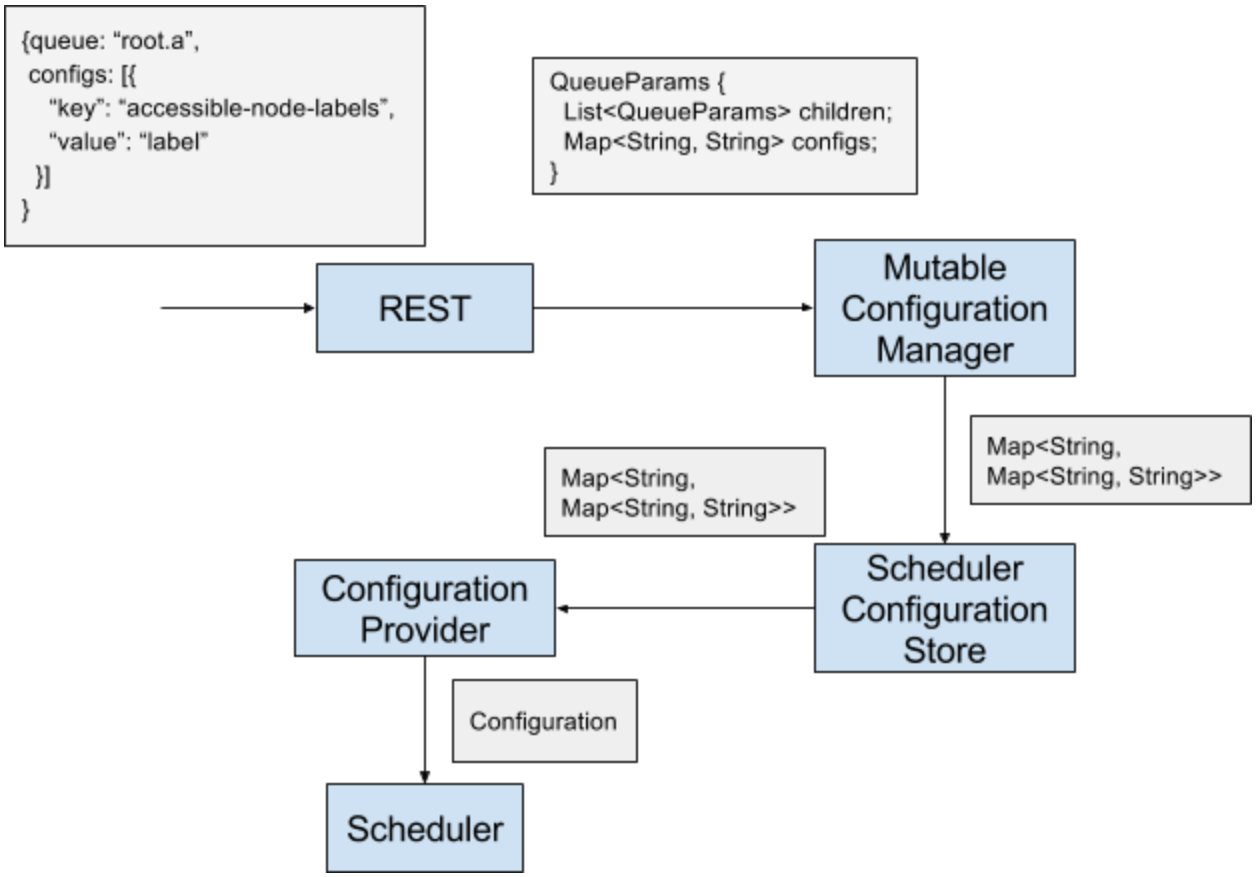


A few highlights for this design are listed below:

- The REST APIs are designed to be scheduler agnostic, in which user provides a payload that describes a portion of the queue hierarchy and the included queues' configurations inside a JSON object. This allows users to specify multiple configuration changes with one REST call. The proposed configuration change is validated for authorization and correctness before committed to the config store. In addition, we allow specifying a queue's fully qualified name (root.qA.qB) so updating a queue local configuration does not require specifying the entire hierarchy of queues.
- A scheduler configuration specific store is proposed on the RM side to manage storing the queue configurations in a non-file based store. This allows the configuration store and the RM state store to use separate backing implementations and keeps the application state store API and configuration store API separate.
- The MutableConfigurationManager component is proposed which takes care of the initial conversion of file based scheduler configuration to the config store based mechanism as well as handling scheduler configuration updates. This decouples backend configuration store from the scheduler that is currently being used.

- We further designed a pluggable policy component which handles queue configuration authorization and validation. The initial policy will simply be based on queue ACLs, but it should be able to support more complicated policies (I want to give certain group the permission to kill applications in a queue but not really modifying queue configurations). We can also have scheduler specific policies which contain logics for validating if a scheduler specific configuration update is valid or not - for example, if the client requests a queue's max capacity be set to greater than 100, the policy should reject this change.
- To delete a queue, the REST client can change the state of a queue to DELETED. The feature proposed in YARN-5724 will then take care of stopping new requests to the queue and eventually deleting the queue gracefully.

The typical flow from user issuing REST invocations to scheduler reinitializing itself to apply the updated configuration is demonstrated below:



- User supplied JSON is parsed into QueueParams objects and given to MutableConfigurationManager
- MutableConfigurationManager handles authentication and verification of configuration changes. It then sends a flattened Map to YarnConfigurationStore, which maps queue path to a map of key/value pairs representing the list of configuration parameters to be updated for a set of queues. MutableConfigurationManager also handles triggering scheduler reinitialization once config changes are committed.

- YarnConfigurationStore stores queue configuration parameters as key/value pairs and handles updating a list of config changes atomically.
- When scheduler reinitializes, the configuration provider will fetch a flattened map of queue configurations and construct a Configuration object to be given to the scheduler.

API

- REST
    - As referenced above, the REST API takes desired configuration changes and passes it to the mutable configuration manager. The mutable configuration manager will parse the object into a standard format to store in the YarnConfigurationStore as described above.
        - Example for changing a set of configurations:

```
<add>
        <queue=”root.a”>
                <child>a2</child>
                <capacity>30</capacity>
        </queue>
        <queue=”root.b”>
                <child>b2</child>
                <capacity>30</capacity>
        </queue>
</add>
<remove>
        <queue=”root.a”>
                <child>a1</child>
        </queue>
</remove>
<update>
        <queue=”root.a”>
                <accessible-node-labels>label</accessible-node-labels>
                <user-limit-percent>5</user-limit-percent>
        </queue>
</update>
```

This adds a child of root.a “a2” with capacity 30, and a child of root.b “b2” with capacity 30, removes the “a1” child of root.a, and updates the specified queue configurations of root.a. (Note that this REST call will fail if any queue ends up with the sum of its children having capacity not equal to 100)

Also note that the queue name is flattened, but the configuration change is hierarchical. The configuration update can also be passed as:

```
<queue=”root”>
        <queue=”a”>
```

<accessible-node-labels>label</accessible-node-labels>
<user-limit-percent>5</user-limit-percent>
</queue>
</queue>
In both cases, the configuration manager will pass the arguments queue=”root.a”, key=”accessible-node-labels”, and value=”label” to the configuration store, as well as queue=”root.a”, key=”user-limit-percent”, and value=”5”.

- CLI
  - The user gives a list of operations to the CLI command grouped by type (add/remove/update).
    - Example: yarn schedulerconf --update queue=root.a,capacities=a3:10,a4:40;queue=root.a.a1,accessible-node-labels=test --add parent=root.a,queue=a5,capacity=30;parent=root.b;queue=b2 --remove parent=root.b,queue=b1
    - This example updates root.a.a3’s capacity to 10, root.a.a4’s capacity to 40, as well as root.a.a1’s’ accessible node labels to “test”. Also it adds queue a5 under root.a with capacity 30, queue b2 under root.b with 0 capacity, and removes queue b1 from root.b. The configuration changes within the same type are delimited by semicolons, and the parameters within a configuration change are delimited by commas. Again the sum of children for each queue must be 100, or the CLI command will fail.

Configuration Storage

- Use a YarnConfigurationStore (analogous to RMStateStore) to store scheduler configuration.
  - Extend this abstract class for different implementations of backing store. For example, we will provide a DerbySchedulerConfigurationStore which uses embedded Derby DB. This will be the default implementation (we have tested this version which has been running in production for over a year now). Other options are LevelDBSchedulerConfigurationStore, etc.
- StoreConfigurationProvider will get the scheduler-agnostic configuration key/values from YarnConfigurationStore, and translate it to a Configuration object. Then, each scheduler will extend the StoreConfigurationProvider abstract class for retrieving its scheduler specific configurations. For example capacity scheduler has a CapacityStoreConfigurationProvider which queries YarnConfigurationStore for things like capacity, max capacity, …
  - Once CapacityStoreConfigurationProvider gets all the configuration key/values and constructs the Configuration, it passes it to CapacityScheduler.
- Implement a new MutableConfigurationManager which takes care of initializing and updating the YarnConfigurationStore.

- The MutableConfigurationManager should be able to initialize the YarnConfigurationStore with a user provided capacity-scheduler.xml. It parses the config file and insert the relevant content to the backend configuration store.
- The MutableConfigurationManager should also be able to update the relevant part of parameters stored in YarnConfigurationStore according to the QueueParams object handed over from the REST APIs.
- The MutableConfigurationManager has a pluggable Policy mechanism which checks if a given configuration change by a certain user is allowed, as well as if the configuration change is valid. Initially, we would have a CapacitySchedulerPolicy which performs basic checks, such as if a max capacity change is within the range [0-100]. This policy also will do authorization, checking if the caller is part of the ADMINISTER_QUEUE ACL for the configuration changes they are requesting.

Queue State Management

The API-based scheduler configuration management (YARN-5734) is focused on removing the file-based configuration management system and supporting configuration changes via API for generic queue configurations. The focus of YARN-5724 is to support queue configuration and state consistency on queue addition and deletion. This also has a configuration management component, but since YARN-5734 is for general configuration changes, the objectives of YARN-5734 and YARN-5724 are orthogonal.

Audit Logging

The audit log serves two purposes; first as a write-ahead log for durability, second as a way to track what changes are made to the scheduler configurations at which time by whom. We create a new AuditLogger for scheduler configuration changes, and the audit entries are stored in the same backing store as the scheduler configuration. For bulk updates, the storage implementation takes care of logging the bulk update atomically (since the client expects the bulk update to be atomic). Then the configuration store is updated with the configuration changes and the scheduler is refreshed to apply the changes.

If there is failover after the log is written and before the change is applied, the StoreConfigurationProvider will replay the logs that have not been applied to the store (e.g. by giving each log entry and the configuration store a version number, and applying the log entry iff its version number is greater than that of the configuration store).