KIP-117: Add a public AdminClient API for Kafka admin operations

Contents

Status

Current state: Accepted

Discussion thread: here

JIRA: KAKFA-3265

Released: 0.11.0

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

Systems that interface with Kafka, such as management systems and proxies, often need to perform administrative actions. For example, they might need to be able to create or delete topics. Currently, they can't do this without relying on internal Kafka classes, or shell scripts distributed with Kafka. We would like to add a public, stable `AdminClient` API that exposes this functionality to JVM-based clients in a well-supported way.

The `AdminClient` will use the KIP-4 wire protocols to communicate with brokers. Because we avoid direct communication with ZooKeeper, the client does not need a ZooKeeper dependency. In fact, once this KIP is implemented, we will be able to lock down Zookeeper security further, by enforcing the invariant that only brokers need to communicate with ZK.

By using the `AdminClient` API, clients will avoid being tightly coupled to the implementation details of a particular Kafka version. They will not need to access internal Kafka classes, or parse the output of scripts. They will also gain the benefits of cross-version client compatibility as implemented in KIP-97.

Proposed Changes

The `AdminClient` will be distributed as part of `kafka-clients.jar`. It will provide a Java API for managing Kafka.

The `AdminClient` interface will be in the `org.apache.kafka.clients.admin` namespace. The implementation will be in the `KafkaAdminClient` class, in the same namespace. The separation between interface and implementation is intended to make the difference between public API and private implementation clearer, and make developing mocks in unit tests easier. This is similar to the divide between `Producer` and `KafkaProducer`, and `Consumer` and `KafkaConsumer`.

Users will configure the `AdminClient` the same way they configure the `Producer` and `Consumer`: by supplying a map of keys to values to its constructor. As much as possible, we should reuse the same configuration key names, such as bootstrap.servers, client.id, etc. We should also offer the ability to configure timeouts, buffer sizes, and other networking settings.

The `AdminClient` will provide `CompletableFuture`-based APIs that closely reflect the requests which the brokers can handle. The client will be multi-threaded; multiple threads will be able to safely make calls using the same `AdminClient` object. When a future fails, its `get()` method will throw an `ExecutionException` which wraps the underlying exception.

We want to handle errors fully and cleanly in `AdminClient`. APIs that require communication with multiple brokers should allow for the possbiility that some brokers will respond and others will not. Any possible return value from the API should be handled. Note that API functions do not throw exceptions. Instead, the `CompletableFuture` objects contain the exceptions when necessary.

In general, we want to avoid using internal Kafka classes in the `AdminClient` interface. For example, most RPC classes should be considered internal, such as `MetadataRequest` or `MetadaResponse`. We should be able to change those classes in the future without worrying about breaking users of `AdminClient`. Inner classes such as `MetadataResponse#TopicMetadata` should also be considered internal, and not exposed in the API of `AdminClient`.

Implementation

As mentioned earlier, the `AdminClient` will use the KIP-4 wire protocol. This mainly means using `NetworkClient` and related RPC classes for the implementation.

This KIP will add only APIs that can be implemented with the existing server-side RPCs. See "New or Changed Public Interfaces" for details. The intention is that we will continue to extend `AdminClient` with further KIPs that also add the appropriate server-side functionality is added (such as ACL management.)

New or Changed Public Interfaces

Clients use the administrative client by creating an instance of class `AdminClient`, via the `AdminClient#Factory#create` function. When the user is done with the `AdminClient`, they must call close to release the network sockets and other associated resources of the client.

```
public interface AdminClient extends AutoCloseable {
    static AdminClient create(Map<String, Object> conf);

    /**
     * Close the AdminClient and release all associated resources.
     */
    void close();

    /**
     * Create a batch of new topics with the default options.
     *
     * @param newTopics         The new topics to create.
     * @return                  The CreateTopicsResults.
     */
    CreateTopicResults createTopics(Collection<NewTopic> newTopics);

    /**
     * Create a batch of new topics.
     *
     * @param newTopics         The new topics to create.
     * @param options           The options to use when creating the new topics.
     * @return                  The CreateTopicsResults.
     */
    CreateTopicResults createTopics(Collection<NewTopic> newTopics, CreateTopicsOptions options);

    /**
     * Similar to #{@link AdminClient#deleteTopics(Collection<String>, DeleteTopicsOptions),
     * but uses the default options.
     *
     * @param topics            The topic names to delete.
     * @return                  The DeleteTopicsResults.
     */
    DeleteTopicResults deleteTopics(Collection<String> topics);

    /**
     * Delete a batch of topics.
     *
     * It may take several seconds after AdminClient#deleteTopics returns
     * success for all the brokers to become aware that the topics are gone.
     * During this time, AdminClient#listTopics and AdminClient#describeTopic
     * may continue to return information about the deleted topics.
     *
     * If delete.topic.enable is false on the brokers, deleteTopics will mark
     * the topics for deletion, but not actually delete them.  The futures will
     * return successfully in this case.
     *
     * @param topics            The topic names to delete.
     * @param options           The options to use when deleting the topics.
     * @return                  The DeleteTopicsResults.
     */
    DeleteTopicResults deleteTopics(Collection<String> topics, DeleteTopicsOptions options);

    /**
     * List the topics available in the cluster with the default options.
```

```
     * @return                 The ListTopicsResults.
     */
    ListTopicsResults listTopics();

    /**
     * List the topics available in the cluster.
     *
     * @param options          The options to use when listing the topics.
     * @return                 The ListTopicsResults.
     */
    ListTopicsResults listTopics(ListTopicsOptions options);

    /**
     * Similar to {@link AdminClient#describeTopic(String, DescribeTopicOptions)},
     * but uses the default options.
     *
     * @param topicName        The topic to describe.
     * @return                 The DescribeTopicResults.
     */
    DescribeTopicResults describeTopic(String topicName);

    /**
     * Descripe an individual topic in the cluster.
     *
     * Note that if auto.create.topics.enable is true on the brokers,
     * AdminClient#describeTopic(topicName) may create a topic named topicName.
     * There are two workarounds: either use AdminClient#listTopics and ensure
     * that the topic is present before describing, or disable
     * auto.create.topics.enable.
     *
     * @param topicName        The topic to describe.
     * @param options          The options to use when describing the topic.
     * @return                 The DescribeTopicResults.
     */
    DescribeTopicResults describeTopic(String topicName, DescribeTopicOptions options);

    /**
     * Describe the cluster information, using the default options.
     *
     * @return                 The ListNodesResults.
     */
    DescribeClusterResults describeCluster();

    /**
     * Describe the cluster information.
     *
     * @param options          The options to use when describing the cluster.
     * @return                 The ListNodesResults.
     */
    DescribeClusterResults describeCluster(DescribeClusterOptions options);

    /**
     * Get information about the api versions of nodes in the cluster with the default options.
     *
     * @param nodes            The nodes to get information about.
     * @return                 The ApiVersionsResults.
     */
    ApiVersionsResults apiVersions(Collection<Node> nodes);

    /**
     * Get information about the api versions of nodes in the cluster.
     *
     * @param nodes            The nodes to get information about.
     * @param options          The options to use when getting api versions of the nodes.
     * @return                 The ApiVersionsResults.
     */
    ApiVersionsResults apiVersions(Collection<Node> nodes, ApiVersionsOptions options);
}

/**
 * The base class for a request to create a new topic.
```

```
 */
abstract class NewTopic {
    public NewTopic(String name, int numPartitions, short replicationFactor);
    public NewTopic(String name, Map<Integer, List<Integer>> replicasAssignments);

    public String name();

    /**
      * Set the configuration to use on the new topic.
      *
      * @param configs               The configuration map.
      * @return                      This NewTopic object.
      */
    public NewTopic setConfigs(Map<String, String> configs);
}

 /**
  * Options for the newTopics call.
  */
class CreateTopicsOptions {
    private Integer timeoutMs = null;
    private boolean validateOnly = false;
    public CreateTopicsOptions setTimeoutMs(int timeoutMs);
    public CreateTopicsOptions setValidateOnly(boolean validateOnly);
    public boolean validateOnly();
}

/**
  * The result of the createTopics call.
  */
class CreateTopicResults {
    /**
      * Return a map from topic names to futures, which can be used to check the status of individual
      * topic creations.
      */
    public Map<String, KafkaFuture<Void>> results();

    /**
      * Return a future which succeeds if all the topic creations succeed.
      */
    public KafkaFuture<Void> all();
}

/**
  * Options for the deleteTopics call.
  */
class DeleteTopicsOptions {
    private Integer timeoutMs = null;
    public DeleteTopicsOptions setTimeoutMs(int timeoutMs);
    public int timeoutMs();
}

/**
  * The result of the deleteTopics call.
  */
class DeleteTopicResults {
    /**
      * Return a map from topic names to futures which can be used to check the status of
      * individual deletions.
      */
    public Map<String, KafkaFuture<Void>> results();

    /**
      * Return a future which succeeds only if all the topic deletions succeed.
      */
    public KafkaFuture<Void> all();
}

class ListTopicsOptions {
    private Integer timeoutMs = null;
    private boolean listInternal = true;
```

```
    public ListTopicsOptions setTimeoutMs(Integer timeoutMs);
    public Integer timeoutMs();

    /**
      * Set whether we should list internal topics.
      *
      * @param listInternal  Whether we should list internal topics.
      * @return              This ListTopicsOptions object.
      */
    public ListTopicsOptions setListInternal(boolean listInternal);

    public boolean listInternal();
}

/**
  * The result of the listTopics call.
  */
class ListTopicsResults {
    /**
      * Return a future which yields a map of topic names to TopicListing objects.
      */
    public KafkaFuture<Map<String, TopicListing>> namesToDescriptions();

    /**
      * Return a future which yields a collection of TopicListing objects.
      */
    public KafkaFuture<Collection<TopicListing>> descriptions();

    /**
      * Return a future which yields a collection of topic names.
      */
    public KafkaFuture<Collection<String>> names();
}

class TopicListing {
    public String name();
    public boolean internal();
}

/**
  * A detailed description of a single topic in the cluster.
  */
class TopicDescription {
    public String name();
    public boolean internal();
    public Map<Integer, TopicPartitionInfo> partitions();
    public String toString();
}

class TopicPartitionInfo {
    public int partition();
    public Node leader();
    public List<Node> replicas();
    public List<Node> isr();
    public String toString();
}

class DescribeTopicsOptions {
    public DescribeTopicOptions setTimeoutMs(int timeoutMs);
    public int timeoutMs();
}

/**
  * The results of the describeTopic call.
  */
class DescribeTopicResults {
    /**
      * Return a future which yields a map of topic names to TopicDescription objects.
      */
    public CompletableFuture<Map<String, TopicDescription>> namesToDescriptions();
```

```
    /**
      * Return a future which yields a collection of TopicDescription objects.
      */
    public CompletableFuture<Collection<TopicDescription>> descriptions();
}

/**
  * Options for the listNodes call.
  */
class DescribeClusterOptions {
    public DescribeClusterOptions setTimeoutMs(Integer timeoutMs);
    public Integer timeoutMs();
}

/**
  * The results of the describeCluster call.
  */
class DescribeClusterOptionsResults {
    public CompletableFuture<Collection<Node>> nodes();
    public CompletableFuture<Node> controller();
    public CompletableFuture<String> clusterId();
}

/**
  * Options for the apiVersions call.
  */
class ApiVersionsOptions {
    public ApiVersionsOptions timeoutMs(Integer timeoutMs);
    public Integer timeoutMs();
}

/**
  * Results of the apiVersions call.
  */
class ApiVersionsResults {
    ApiVersionsResults(Map<Node, KafkaFuture<NodeApiVersions>> futures);
    public Map<Node, KafkaFuture<NodeApiVersions>> results();
    public KafkaFuture<Map<Node, NodeApiVersions>> all();
}
```

Configuration

Just like the consumer and the producer, the admin client will have its own `AdminClientConfig` configuration class which extends `AbstractConfig.`

Initially, the supported configuration keys will be:

- `bootstrap.servers`
  - The boostrap servers as a list of host:port pairs.
- `client.id`
  - An ID string to pass to the server when making requests.
- `metadata.max.age.ms`
  - The period of time in milliseconds after which we force a refresh of metadata even if we haven't seen any partition leadership changes to proactively discover any new brokers or partitions.
- `send.buffer.bytes`
  - The size of the TCP send buffer (SO_SNDBUF) to use when sending data. If the value is -1, the OS default will be used
- `receive.buffer.bytes`
  - The size of the TCP receive buffer (SO_RCVBUF) to use when reading data. If the value is -1, the OS default will be used.
- `reconnect.backoff.ms`
  - The amount of time to wait before attempting to reconnect to a given host. This avoids repeatedly connecting to a host in a tight loop. This backoff applies to all requests sent by the consumer to the broker.
- `try.backoff.ms`
  - The amount of time to wait before attempting to retry a failed request.
- `request.timeout.ms`
  - The configuration controls the maximum amount of time the client will wait for the response of a request. If the response is not received before the timeout elapses the client will resend the request if necessary or fail the request if retries are exhausted.
- `connections.max.idle.ms`

    - Close idle connections after the number of milliseconds specified by this config
- `security.protocol`
    - The security protocol used to communicate with brokers

Migration Plan and Compatibility

The `AdminClient` will use KIP-97 API version negotiation to communicate with older or newer brokers. In cases where an API is not available on an older or newer broker, we will throw an `UnsupportedVersionException.`

We should avoid making incompatible changes to the `AdminClient` function and class signatures. So for example, if we need to add an additional argument to an API, we should add a new function rather than changing an existing function signature.

Test Plan

We should have an `AdminClientTest` integration test which tests creating topics, deleting topics, and listing topics through the API. We can use the `KafkaServerTestHarness` to test this efficiently with multiple brokers. For methods which support batches, we should test passing a batch of items to be processed. We should test error conditions as well. We should test the node listing and version getting APIs as well.

Rejected Alternatives

Synchronous API

Instead of having a futures-based API, we could have a synchronous API. In this API, each function would block rather than returning a Future. However, the Futures-based API can easily be used as a blocking API, simply by calling get() on the Futures which get returned.

Future Work

We would like to add more functionality to the `AdminClient` as soon as it becomes available on the brokers. For example, we would like to add a way of altering topics that already exist. We would also like to add management APIs for ACLs, or the functionality of `GetOffsetShell`.