KIP-201: Rationalising Policy interfaces

- Status
- Motivation
  - Problem 1 - Topic config is governed by both CreateTopicPolicy and AlterConfigPolicy
  - Problem 2 - Creating more partitions is not currently covered by any policy
  - Problem 3 - CreateTopicPolicy can govern partition assignment, but there is no policy for reassignment
  - Problem 4 - There is no policy for topic deletion or record deletion
  - Problem 5 - The existing CreateTopicPolicy doesn't have enough information
- Public Interfaces
- Proposed Changes
  - Add TopicManagementPolicy and supporting interfaces
  - Deprecate existing policies
  - Add new versions of DeleteTopicsRequest and DeleteTopicsResponse
  - Add new versions of DeleteRecordsRequest and DeleteRecordsResponse
- Compatibility, Deprecation, and Migration Plan
- Rejected Alternatives

Status

Current state: *Under Discussion*

Discussion thread: *here*

JIRA: *https://issues.apache.org/jira/browse/KAFKA-5693*

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

Motivation

Kafka already has user configurable policies which can be used by a cluster administrator to limit how the cluster can be modified by non-administrator users (for example by using the AdminClient API):

- `CreateTopicPolicy` can prevent a topic being created based on topic creation parameters (name, number of partitions & replication factor or replica assignments, topic configs)
- `AlterConfigPolicy` can prevent a change to topic config (or, in theory, broker config, but it's current not possible to change broker configs via the AdminClient API)

The existing policies were added in KIP-108 and KIP-133 and at that time there was an expectation that the AdminClient would gain a single API for topic modification. However, discussion about KIP-179, and work on KIP-195 shows that the AdminClient will end up with multiple APIs for modifying topics in different ways. Consequently there isn't and won't be a direct mapping between operations and policies.

Adding new policies ad-hoc is likely to lead to a poor result because:

- Users need to know about a policy to implement it. Its easier to know about 1 thing than half a dozen.
- The more policies there are the more places they need to be configured.

Problem 1 - Topic config is governed by both CreateTopicPolicy and AlterConfigPolicy

Currently the topic config is passed to the `CreateTopicPolicy`, but if a topic config is later modified the `AlterConfigPolicy` is applied. If an administrator wants to use the topic config in their policy decisions they have to implement this logic in two places (or at least multiply inherit both policy implementations and configure the same class name for each policy). If the policy decision depends on both the topic config and another aspect of the topic the `AlterConfigPolicy` interface doesn't provide the necessary information.

Problem 2 - Creating more partitions is not currently covered by *any* policy

Changing the number of partitions in a topic was the subject of KIP-195 and is just one kind of topic modification. Consider two example use cases:

1. It shouldn't be possible to create a topic, but then modify it so that it no longer conforms to the `CreateTopicPolicy`.
2. An administrator who wants to prevent increasing the number of partitions entirely for topics with keys, because of the effect on partitioning.

To solve 1, we could simply apply the existing `TopicCreationPolicy` to modifications, but

- this would obscure whether a particular invocation of the policy was for a topic creation or modification (the second bullet)
- we would be left with a misleadingly named policy

So there needs to be a policy for specifically for *modifying* a topic. But it is confusing and error-prone if there are *different* policy classes for creation and modification (the `CreateTopicPolicy` and a new `ModifyTopicPolicy`, say): It would be easy for the code implementing a user's policies to get out of sync if it needs to be maintained in two places. It would also be easy to configure one policy but not the other.

Multiply inheriting separate the `CreateTopicPolicy` and a new `ModifyTopicPolicy` is a solution, but still requires multiple configuration keys.

It would be better if there were a *single* policy interface which is applied to both topic creation and modification in a more uniform way.

Problem 3 - CreateTopicPolicy can govern partition assignment, but there is no policy for reassignment

Reassigning replicas is another kind of topic modification and the subject of KIP-179. By similar reasoning to example 2 it, too, should be covered by the same policy. This would require that the same request metadata could describe both kinds of modification satisfactorily.

Problem 4 - There is no policy for topic deletion or record deletion

KIP-170 proposes a policy for topic deletion (see that KIP for the motivation behind this) and KIP-204 proposes to add an AdminClient API for the existing network protocol for deleting records from the partitions of a topic.

While deleting records from a topic is not the same as deleting the topic itself, both result in records being deleted, and it is those records which have value to the company or organisation operating the cluster. Thus if topic deletion is deserving of a policy, it should also be possible to apply a similar policy to record deletion, otherwise a user might be able to apply business rules to on the one kind of deletion, but not the other. If there were a *separate* `TopicDeletePolicy` and `MessageDeletePolicy` we have the similar problem as described above for separate topic creation and modification policies: It's unnecessarily difficult and tedious to keep the policies consistent and correctly configured.

Problem 5 - The existing `CreateTopicPolicy` doesn't have enough information

As noted in KIP-170, for some use cases, the existing `CreateTopicPolicy` doesn't get passed enough information for the operators desired rules to be enfored. For example, while providing Kafka-as-a-Service there is the need to ensure that the new topic require more resources than the cluster can support (e.g. number of partitions won't exceed some maximum).

Public Interfaces

A new policy interface will be added, TopicManagementPolicy will apply to topic creation, topic alteration, topic deletion and message deletion. It will be configured by the new `topic.management.policy.class.name` config.

The existing policy interfaces `CreateTopicPolicy` and `AlterConfigPolicy` will be deprecated, but will continue to be applied where they are currently applied until they are removed.

New versions of existing network protocol `DeleteTopicsRequest` and `DeleteRecordsRequest` will be added, to add a validate_only flag.

New versions of existing network protocol `DeleteTopicsResponse` and `DeleteRecordsResponse` will be added to include an error message.

Proposed Changes

Add TopicManagementPolicy and supporting interfaces

The following policy interfaces and supporting classes will be added

```
/**
 * Represents the state of a topic.
 */
interface TopicState {
    /**
     * The number of partitions of the topic.
     */
    int numPartitions();

    /**
     * The replication factor of the topic. More precisely, the number of assigned replicas for partition 0.
     * // TODO what about during reassignment
     */
    short replicationFactor();

    /**
     * A map of the replica assignments of the topic, with partition ids as keys and
     * the assigned brokers as the corresponding values.
     * // TODO what about during reassignment
     */
    Map<Integer, List<Integer>> replicasAssignments();
```

```
    /**
     * The topic config.
     */
    Map<String,String> configs();

    /**
     * Returns whether the topic is marked for deletion.
     */
    boolean markedForDeletion();

    /**
     * Returns whether the topic is an internal topic.
     */
    boolean internal();

}

/**
 * Represents the requested state of a topic.
 */
interface RequestedTopicState extends TopicState {
    /**
     * True if the {@link TopicState#replicasAssignments()}
     * in this request we generated by the broker, false if
     * they were explicitly requested by the client.
     */
    boolean generatedReplicaAssignments();

    /**
     * The topic config as it will be if the request is successful.
     * This is effectively the same as the value of {@code configs}
     * after the following computation:
     * <pre><code>
     *   Map<String, String> configs = currentState.configs();
     *   configs.putAll(requestedState.requestedConfigs();
     * </code></pre>
     */
    @Override
    Map<String,String> configs();

    /**
     * The topic configs present in the request.
     */
    Map<String,String> requestedConfigs();
}

/** The current state of the topics in the cluster, before the request takes effect. */
interface ClusterState {
    /**
     * Returns the current state of the given topic, or null if the topic does not exist.
     */
    TopicState topicState(String topicName);

    /**
     * Returns all the topics in the cluster, including internal topics if
     * {@code includeInternal} is true, and including those marked for deletion
     * if {@code includeMarkedForDeletion} is true.
     */
    Set<String> topics(boolean includeInternal, boolean includeMarkedForDeletion);

    /**
     * The number of brokers in the cluster.
     */
    int clusterSize();

    /**
     *  Returns the current state of the broker in which the method is called.
     */
    BrokerState brokerState();

}
```

```
/**
 * A policy that is enforced on topic creation, alteration and deletion,
 * and for the deletion of messages from a topic.
 *
 * An implementation of this policy can be configured on a broker via the
 * {@code topic.management.policy.class.name} broker config.
 * When this is configured the named class will be instantiated reflectively
 * using its nullary constructor and will then pass the broker configs to
 * its <code>configure()</code> method. During broker shutdown, the
 * <code>close()</code> method will be invoked so that resources can be
 * released (if necessary).
 *
 */
interface TopicManagementPolicy  extends Configurable, AutoCloseable {

    static interface AbstractRequestMetadata {

        /**
         * The topic the action is being performed upon.
         */
        public String topic();

        /**
         * The authenticated principal making the request.
         */
        public KafkaPrincipal principal();
    }


    static interface CreateTopicRequest extends AbstractRequestMetadata {
        /**
         * The requested state of the topic to be created.
         */
        public RequestedTopicState requestedState();
    }

    /**
     * Validate the given request to create a topic
     * and throw a <code>PolicyViolationException</code> with a suitable error
     * message if the request does not satisfy this policy.
     *
     * Clients will receive the POLICY_VIOLATION error code along with the exception's message.
     * Note that validation failure only affects the relevant topic,
     * other topics in the request will still be processed.
     *
     * @param requestMetadata the request parameters for the provided topic.
     * @param clusterState the current state of the cluster
     * @throws PolicyViolationException if the request parameters do not satisfy this policy.
     */
    void validateCreateTopic(CreateTopicRequest requestMetadata, ClusterState clusterState) throws
PolicyViolationException;

    static interface AlterTopicRequest extends AbstractRequestMetadata {
        /**
         * The state the topic will have after the alteration.
         */
        public RequestedTopicState requestedState();
    }

    /**
     * Validate the given request to alter an existing topic
     * and throw a <code>PolicyViolationException</code> with a suitable error
     * message if the request does not satisfy this policy.
     *
     * The given {@code clusterState} can be used to discover the current state of the topic to be modified.
     *
     * Clients will receive the POLICY_VIOLATION error code along with the exception's message.
     * Note that validation failure only affects the relevant topic,
     * other topics in the request will still be processed.
     *
```

```
     * @param requestMetadata the request parameters for the provided topic.
     * @param clusterState the current state of the cluster
     * @throws PolicyViolationException if the request parameters do not satisfy this policy.
     */
    void validateAlterTopic(AlterTopicRequest requestMetadata, ClusterState clusterState) throws
PolicyViolationException;

    /**
     * Parameters for a request to delete the given topic.
     */
    static interface DeleteTopicRequest extends AbstractRequestMetadata {
    }

    /**
     * Validate the given request to delete a topic
     * and throw a <code>PolicyViolationException</code> with a suitable error
     * message if the request does not satisfy this policy.
     *
     * The given {@code clusterState} can be used to discover the current state of the topic to be deleted.
     *
     * Clients will receive the POLICY_VIOLATION error code along with the exception's message.
     * Note that validation failure only affects the relevant topic,
     * other topics in the request will still be processed.
     *
     * @param requestMetadata the request parameters for the provided topic.
     * @param clusterState the current state of the cluster
     * @throws PolicyViolationException if the request parameters do not satisfy this policy.
     */
    void validateDeleteTopic(DeleteTopicRequest requestMetadata, ClusterState clusterState) throws
PolicyViolationException;

    /**
     * Parameters for a request to delete records from the topic.
     */
    static interface DeleteRecordsRequest extends AbstractRequestMetadata {

        /**
         * Returns a map of topic partitions and the corresponding offset of the last message
         * to be retained. Messages before this offset will be deleted.
         * Partitions which won't have messages deleted won't be present in the map.
         */
        Map<Integer, Long> deletedMessageOffsets();
    }

    /**
     * Validate the given request to delete records from a topic
     * and throw a <code>PolicyViolationException</code> with a suitable error
     * message if the request does not satisfy this policy.
     *
     * The given {@code clusterState} can be used to discover the current state of the topic to have records
deleted.
     *
     * Clients will receive the POLICY_VIOLATION error code along with the exception's message.
     * Note that validation failure only affects the relevant topic,
     * other topics in the request will still be processed.
     *
     * @param requestMetadata the request parameters for the provided topic.
     * @param clusterState the current state of the cluster
     * @throws PolicyViolationException if the request parameters do not satisfy this policy.
     */
    void validateDeleteRecords(DeleteRecordsRequest requestMetadata, ClusterState clusterState) throws
PolicyViolationException;
}


/**
 * Represents the state of a broker
 */
interface BrokerState {

    /**
```

```
     * The broker config.
     */
    Map<String,String> configs();
}

interface RequestedBrokerState extends BrokerState {

    /**
     * The topic config as it will be if the request is successful.
     * This is effectively the same as the value of {@code configs}
     * after the following computation:
     * <pre><code>
     *   Map<String, String> configs = currentState.configs();
     *   configs.putAll(requestedState.requestedConfigs();
     * </code></pre>
     */
    @Override
    Map<String,String> configs();


    /**
     * The broker configs present in the request.
     */
    Map<String,String> requestedConfigs();
}


/**
 * A policy that is enforced on broker alteration.
 *
 * An implementation of this policy can be configured on a broker via the
 * {@code broker.management.policy.class.name} broker config.
 * When this is configured the named class will be instantiated reflectively
 * using its nullary constructor and will then pass the broker configs to
 * its <code>configure()</code> method. During broker shutdown, the
 * <code>close()</code> method will be invoked so that resources can be
 * released (if necessary).
 *
 * TODO: Fully define the lifecycle since the policy is configured by broker config which changes, so a means
of reconfiguration is required.
 */
interface BrokerManagementPolicy extends Configurable, AutoCloseable {
    static interface AbstractRequestMetadata {

        /**
         * The id of the broker the action is being performed upon.
         * This is always the same as the id of the broker in which the
         * broker management policy is executing.
         */
        public int brokerId();

        /**
         * The principal making the request.
         */
        public KafkaPrincipal principal();
    }


    static interface AlterBrokerRequest extends AbstractRequestMetadata {
        /**
         * The requested state of the broker to be altered.
         */
        public RequestedBrokerState requestedState();
    }

    /**
     * Validate the given request to alter a broker
     * and throw a <code>PolicyViolationException</code> with a suitable error
     * message if the request does not satisfy this policy.
     *
     * Clients will receive the POLICY_VIOLATION error code along with the exception's message.
```

```
     * Note that validation failure only affects the relevant broker,
     * other topics in the request will still be processed.
     *
     * @param requestMetadata the request parameters for the provided broker.
     * @param clusterState the current state of the cluster
     * @throws PolicyViolationException if the request parameters do not satisfy this policy.
     */
    void validateAlterBroker(AlterBrokerRequest requestMetadata,
                             ClusterState clusterState)
            throws PolicyViolationException;
}
```

The `TopicManagementPolicy` will be applied:

- On topic creation, i.e. when processing a `CreateTopicsRequest`
- On topic modification
    - Change in topic config, ie. when processing `AlterConfigsRequest`, for topic configs (this change done as part of this KIP).
    - Adding partitions to topics, i.e. when processing a `CreatePartitionsRequest` (see KIP-195, but this change done as part of this KIP)
    - Reassigning partitions to brokers, and/or changing the replication factor when processing `ReassignPartitionsRequest` (as part of KIP-179)
- On topic deletion, i.e. when processing a `DeleteTopicsRequest` (this change done as part of this KIP).
- On message deletion, i.e. when processing a `DeleteRecordsRequest` (this change done as part of this KIP).

The `BrokerManagementPolicy` will be applied:

- On broker startup
    - This is to ensure that brokers start in a valid state; without this it would be possible for a later alter broker request to be denied even though the request itself was not the cause of the policy violation.
- On broker modification, ie. when processing a `AlterConfigsRequest` for broker configs.

Deprecate existing policies

The existing `CreateTopicPolicy` and `AlterConfigPolicy` will be deprecated, but will continue to be applied when they are configured.

Using `create.topic.policy.class.name` or `alter.config.policy.class.name` will result in an deprecation warning in the broker logs.

It will be a configuration-time error if both `create.topic.policy.class.name` and `topic.management.policy.class.name` are used at the same time, or both `alter.config.policy.class.name` and `topic.management.policy.class.name` are used at the same time.

Internally, an adapter implementation of `TopicManagementPolicy` will be used when `CreateTopicPolicy` and `AlterConfigPolicy` are configured, so policy use sites won't be unnecessarily complicated.

If, in the future, `AdminClient.alterConfigs()`/`AlterConfigsRequest` is changed to support changing broker configs a separate policy interface can be applied to such changes.

Add new versions of `DeleteTopicsRequest` and `DeleteTopicsResponse`

The `DELETE_TOPICS` protocol have a 3rd version added (version 2). The `DeleteTopicsRequest` will get a `validate_only` flag. When this is set the request will be validated for correctness, including that it satisfies the `TopicManagementPolicy.validateDeleteTopic()` method, but the topic won't actually be deleted.

```
DeleteTopics Request (Version: 2) => [topics] timeout validate_only
  topics => STRING
  timeout => INT32
  validate_only => BOOLEAN
```

The `DeleteTopicsResponse` will get the ability to include error messages in addition to error codes:

```
DeleteTopics Response (Version: 2) => throttle_time_ms [topic_error_codes]
  throttle_time_ms => INT32
  topic_error_codes => topic error_code error_message
    topic => STRING
    error_code => INT16
    error_message => NULLABLE_STRING
```

Old versions of the `DeleteTopicsResponse` will use a UNEXPECTED_SERVER_ERROR `error_code` instead of POLICY_VIOLATION so as to not break clients.

The documentation for `AdminClient.deleteTopics()` will be updated mention the possibility of `PolicyViolationException` from the `DeleteTopicsResult` methods.

Add new versions of `DeleteRecordsRequest` and `DeleteRecordsResponse`

The `DELETE_RECORDS` protocol have a 2nd version added (version 1). The `DeleteRecordsRequest` will get a `validate_only` flag. When this is set the request will be validated for correctness, including that it satisfies the `TopicManagementPolicy.validateDeleteRecords()` method, but no records will be deleted.

```
DeleteRecords Request (Version: 1) => [topics] timeout validate_only
  topics => topic [partitions]
    topic => STRING
    partitions => partition offset
      partition => INT32
      offset => INT64
  timeout => INT32
  validate_only => BOOLEAN
```

The `DeleteRecordsResponse` will get the ability to include error messages in addition to error codes:

```
DeleteRecords Response (Version: 0) => throttle_time_ms [topics]
  throttle_time_ms => INT32
  topics => topic [partitions]
    topic => STRING
    partitions => partition low_watermark error_code error_message
      partition => INT32
      low_watermark => INT64
      error_code => INT16
      error_message => NULLABLE_STRING
```

Existing versions of the `DeleteRecordsResponse` will use a UNEXPECTED_SERVER_ERROR `error_code` instead of POLICY_VIOLATION so as to not break clients.

The documentation for `AdminClient.deleteRecords()` (being added by KIP-204) will be updated mention the possibility of `PolicyViolationException` from the `DeleteRecordsResult` methods.

Compatibility, Deprecation, and Migration Plan

Existing users will have to reimplement their policies in terms of the new `TopicManagementPolicy` interface, and reconfigure their brokers accordingly. Since the `TopicManagementPolicy` contains a superset of the existing information used by the deprecated policies such reimplementation should be trivial.

The deprecated policy interfaces and configuration keys will be removed in a future Kafka version. If this KIP is accepted for Kafka 1.1.0 this removal could happen in Kafka 2.0.0 or a later major release.

This KIP proposes to retrospectively apply policies to two APIs (delete topics and delete records) exposed via the network protocol. Existing clients might not be expecting a POLICY_VIOLATION error code in the responses. To mitigate this, only the new version of these network protocols will actually return POLICY_VIOLATION. If a client is using an old version of either of these protocols, the policy violation will be returned an UNEXPECTED_SERVER_ERROR, and a message logged to explain that the error is in fact a policy violation.

Rejected Alternatives

The objectives of this KIP could be achieved without deprecating the existing policy classes, but that:

- incurs ongoing maintenance and testing costs on the project for not overall benefit
- If two policies were in force it would be more confusing to users when a request was rejected (which policy rejected it?) possibly exacerbated if users didn't know two policies were in force.
- If it were possible to have two policies in force administrators have not been relieved of the burden of maintaining two policies in sync.

Having separate policy interfaces was considered, but rejected because there would need to be several of them, making it harder for people to discover, understand, implement and configure policies. It would also be easy for users to miss if a new policy interface was added.

Having separate policy interfaces for creation/modification and deletion and retaining the existing single-method-per-policy-interface design was considered, but rejected because it was a half way house between having multiple policies and having a single policy.