KIP-257 - Configurable Quota Management

Status

Current state: *Accepted*

Discussion thread: *here*

JIRA: *KAFKA-6576*

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

Motivation

Kafka brokers support quotas that enforce rate limits to prevent clients saturating the network or monopolizing broker resources. `Fetch`/`Produce` quotas can be configured to limit network bandwidth usage and `Request` quotas can be configured to limit CPU usage (network and I/O thread time). Client quotas may be configured at `<user, client-id>`, `<user>` or `<client-id>` levels and defaults may be defined at each level. For any request, the most specific quota configuration that matches the `(user, client-id)` of the request is applied.

Quotas are configured using the tool `kafka-configs.sh`, which persists quotas in ZooKeeper. Brokers watch quota configuration in ZooKeeper and enforce the currently configured quota for each request. All brokers use the same quota configuration.

Kafka currently does not support customization of quota allocation. In some scenarios, customization of quota limits will be useful.

Scenarios

1. Kafka brokers currently group clients based on user principal and/or client-id for quota enforcement. If quotas are configured at `<user, client-id>` level, all requests that share the user principal and client-id will share the quota. If quotas are configured at `<user>` level, all requests that share the user principal but don't have a matching `<user, client-id>` quota configuration share the `<user>` quota (and similarly for `<client-id>` quotas). In some scenarios, it is useful to define a quota group that combines multiple user principals and/or client-ids. All the requests from the group may then share a single quota.
2. Some clients may have access only to a few topics which are hosted on a subset of brokers. The load from these clients will be mostly on the subset of brokers that are leaders of that subset of topic partitions. Rather than allocate a fixed quota for these clients on each broker, it will be useful to have quotas that are proportional to the number of partitions used by the client that are hosted on the broker. Since partition leaders may change dynamically, it will be better to compute quotas at runtime rather than update ZooKeeper with new quotas whenever partition leaders change.

Goals

- Enable quotas to be customized using a configurable callback.
- Ensure that the callback interface will not prevent us from adding new levels of quotas in future. For example, we may want to introduce the concept of user groups. It should be possible to handle groups in a consistent way for ACLs as well as quotas using the Authorizer interface and the new quota callback interface respectively.
- Enable custom callbacks to access quotas configured in ZooKeeper easily so that existing tools can be used to manage persisted quota configuration if required.
- Enable custom callbacks to track partition leaders easily to support partition-based quotas so that callbacks dont need access to ZooKeeper.

Public Interfaces

Broker Configuration Option

A new broker property will be added to configure a callback for determining client quotas (`Fetch/Produce/Request` quotas). This will be a dynamic broker configuration option that can be updated without restarting the broker. This KIP does not propose to add custom callbacks for replication quotas, but we could add one in future if a requirement arises.

- Name: `client.quota.callback.class`
- Type: `CLASS`
- Mode: Dynamically configurable as cluster-default for all brokers in the cluster
- Description: The fully qualified name of a class that implements the `ClientQuotaCallback` interface, which is used to determine quota limits applied to client requests. By default, `<user, client-id>`, `<user>` or `<client-id>` quotas stored in ZooKeeper are applied. For any given request, the most specific quota that matches the user principal of the session and the client-id of the request is enforced by every broker.

New Interfaces

The following new public classes/traits will be introduced in the package `org.apache.kafka.server.quota` (in the Kafka clients project).

The quota types supported for the callback will be `FETCH/PRODUCE/REQUEST.`

Quota types

```
public enum ClientQuotaType {
    PRODUCE,
    FETCH,
    REQUEST
}
```

`ClientQuotaCallback` must be implemented by custom callbacks. It will also be implemented by the default quota callback. Callback implementations should cache persisted configs if necessary to determine quotas quickly since `ClientQuotaCallback.quota()` will be invoked on every request.

Client Quota Callback

```
/**
 * Quota callback interface for brokers that enables customization of client quota computation.
 */
public interface ClientQuotaCallback extends Configurable {

    /**
     * Quota callback invoked to determine the quota metric tags to be applied for a request.
     * Quota limits are associated with quota metrics and all clients which use the same
     * metric tags share the quota limit.
     *
     * @param quotaType Type of quota requested
     * @param principal The user principal of the connection for which quota is requested
     * @param clientId  The client id associated with the request
     * @return quota metric tags that indicate which other clients share this quota
     */
    Map<String, String> quotaMetricTags(ClientQuotaType quotaType, KafkaPrincipal principal, String clientId);

    /**
     * Returns the quota limit associated with the provided metric tags. These tags were returned from
     * a previous call to {@link #quotaMetricTags(ClientQuotaType, KafkaPrincipal, String)}. This method is
     * invoked by quota managers to obtain the current quota limit applied to a metric when the first request
     * using these tags is processed. It is also invoked after a quota update or cluster metadata change.
     * If the tags are no longer in use after the update, (e.g. this is a {user, client-id} quota metric
     * and the quota now in use is a {user} quota), null is returned.
     *
     * @param quotaType  Type of quota requested
     * @param metricTags Metric tags for a quota metric of type `quotaType`
     * @return the quota limit for the provided metric tags or null if the metric tags are no longer in use
     */
    Double quotaLimit(ClientQuotaType quotaType, Map<String, String> metricTags);

    /**
     * Quota configuration update callback that is invoked when quota configuration for an entity is
     * updated in ZooKeeper. This is useful to track configured quotas if built-in quota configuration
     * tools are used for quota management.
```

```
     *
     * @param quotaType   Type of quota being updated
     * @param quotaEntity The quota entity for which quota is being updated
     * @param newValue    The new quota value
     */
    void updateQuota(ClientQuotaType quotaType, ClientQuotaEntity quotaEntity, double newValue);

    /**
     * Quota configuration removal callback that is invoked when quota configuration for an entity is
     * removed in ZooKeeper. This is useful to track configured quotas if built-in quota configuration
     * tools are used for quota management.
     *
     * @param quotaType   Type of quota being updated
     * @param quotaEntity The quota entity for which quota is being updated
     */
    void removeQuota(ClientQuotaType quotaType, ClientQuotaEntity quotaEntity);

    /**
     * Returns true if any of the existing quota configs may have been updated since the last call
     * to this method for the provided quota type. Quota updates as a result of calls to
     * {@link #updateClusterMetadata(Cluster)}, {@link #updateQuota(ClientQuotaType, ClientQuotaEntity, double)}
     * and {@link #removeQuota(ClientQuotaType, ClientQuotaEntity)} are automatically processed.
     * So callbacks that rely only on built-in quota configuration tools always return false. Quota callbacks
     * with external quota configuration or custom reconfigurable quota configs that affect quota limits must
     * return true if existing metric configs may need to be updated. This method is invoked on every request
     * and hence is expected to be handled by callbacks as a simple flag that is updated when quotas change.
     *
     * @param quotaType Type of quota
     */
    boolean quotaResetRequired(ClientQuotaType quotaType);

    /**
     * Metadata update callback that is invoked whenever UpdateMetadata request is received from
     * the controller. This is useful if quota computation takes partitions into account.
     * Topics that are being deleted will not be included in `cluster`.
     *
     * @param cluster Cluster metadata including partitions and their leaders if known
     * @return true if quotas have changed and metric configs may need to be updated
     */
    boolean updateClusterMetadata(Cluster cluster);

    /**
     * Closes this instance.
     */
    void close();
}
```

The callback is invoked to obtain the quota limit as well the metric tags to be used. These metric tags determine which entities share the quota.

By default the tags "`user`" and "`client-id`" will be used for all quota metrics. When `<user, client-id>` quota config is used, *user* tag is set to user principal of the session and *client-id* tag is set to the client-id of the request. If `<user>` quota config is used, *user* tag is set to user principal of the session and *client-id* tag is set to empty string. Similarly, if `<client-id>` quota config is used, the *user* tag is set to empty string. This ensures that the same quota sensors and metrics are shared by all requests that match each quota config.

When quota configuration is updated in ZooKeeper, quota callbacks are notified of configuration changes. Quota configuration entities can be combined to define quotas at different levels.

ClientQuotaEntity

```
/**
 * The metadata for an entity for which quota is configured. Quotas may be defined at
 * different levels and `configEntities` gives the list of config entities that define
 * the level of this quota entity.
 */
public interface ClientQuotaEntity {
    /**
     * Entity type of a {@link ConfigEntity}
     */
    public enum ConfigEntityType {
        USER,
        CLIENT_ID,
        DEFAULT_USER,
        DEFAULT_CLIENT_ID
    }

    /**
     * Interface representing a quota configuration entity. Quota may be
     * configured at levels that include one or more configuration entities.
     * For example, {user, client-id} quota is represented using two
     * instances of ConfigEntity with entity types USER and CLIENT_ID.
     */
    public interface ConfigEntity {
        /**
         * Returns the name of this entity. For default quotas, an empty string is returned.
         */
        String name();

        /**
         * Returns the type of this entity.
         */
        ConfigEntityType entityType();
    }

    /**
     * Returns the list of configuration entities that this quota entity is comprised of.
     * For {user} or {clientId} quota, this is a single entity and for {user, clientId}
     * quota, this is a list of two entities.
     */
    List<ConfigEntity> configEntities();
}
```

When partition leaders change, controller notifies brokers using `UpdateMetadata` request. Quota callbacks are notified of metadata changes so that callbacks that base quota computation on partitions have access to the current metadata. The existing public interface `org.apache.kafka.common.Cluster` will be used for metadata change notification.

Proposed Changes

`ClientQuotaManager` and `ClientRequestQuotaManager` will be updated to move quota configuration management into a new class `DefaultQuotaCallback` that implements `ClientQuotaCallback`. If a custom callback is not configured, `DefaultQuotaCallback` will be used.

If a custom callback is configured, it will be instantiated when the broker is started. `DynamicBrokerConfig` will be updated to handle changes to the callback. `KafkaApis` will invoke `quotaCallback.updateClusterMetadata` when `UpdateMetadata` request is received from the controller. This will be ignored by the default quota callback. When `ConfigHandler` invokes `ClientQuotaManager.updateQuota` to process dynamic quota config updates, `quotaCallback.updateQuota` will be invoked. The existing logic to process quota updates will be moved to the default quota callback.

Compatibility, Deprecation, and Migration Plan

- *What impact (if any) will there be on existing users?*

  None, the current behaviour will be retained as default.

Rejected Alternatives

Introduce new quota management options instead of a callback

We could implement different quota algorithms in Kafka and support quota groups, partition-based quotas etc. But this would require Kafka to manage these groups, mapping of users to partitions etc, increasing the complexity of the code. Since it will be hard to include support for all possible scenarios into the broker code, it will be simpler to make quota computation configurable. This also enables the computation to be altered dynamically without restarting the broker since the new option will be a dynamic broker config.

Enable management of client quotas and replication quotas using a single callback interface

The configuration and management of replication quotas are completely separate from client quota management in the broker. Since the configuration entities are different, it will be simpler to keep them separate. It is not clear if there are scenarios that require custom replication quotas, so this KIP only addresses client quotas.

Use Scala traits for public interfaces similar to Authorizer

For compatibility reasons, we are now using Java rather than Scala for all pluggable interfaces including those on the broker. There is already a KIP to move `Authorizer` to Java as well. As we will be removing support for Java 7 in the next release, we can also use default methods in Java when we need to update pluggable Java interfaces. So the plan is to use Java for all new pluggable interfaces.