

KIP-599: Throttle Create Topic, Create Partition and Delete Topic Operations


- [Status](#)
- [Motivation](#)
- [Proposed Changes](#)
 - [Changes to Kafka Brokers](#)
 - [Throttling Algorithm](#)
 - [Controller Mutation Quota Manager](#)
 - [Handling of new Clients](#)
 - [Handling of old Clients](#)
 - [Co-Existence of Multiple Quotas](#)
 - [Handling of ValidateOnly](#)
 - [Changes to Kafka Admin Client](#)
 - [Automatic Retry](#)
 - [Manual Retry](#)
- [Public Interfaces](#)
 - [Protocol](#)
 - [New Broker Configurations](#)
 - [New Quota Types](#)
 - [New Broker Metrics](#)
 - [New TokenBucket Metric](#)
 - [Admin API](#)
 - [Kafka Topic Command](#)
 - [Kafka Config Command](#)
- [Known Limitations](#)
- [How to Define the Quota?](#)
- [Compatibility, Deprecation, and Migration Plan](#)
 - [Compatibility with Old Clients](#)
 - [Compatibility after Upgrading the Admin Client to the new version](#)
- [Rejected Alternatives](#)
 - [Update our existing Rate to behave like the Token Bucket](#)
 - [Usage based Quota](#)
 - [Throttle the Execution instead of the Admission](#)
 - [Throttle the Admission but without an explicit Error code](#)

Status

Current state: *Accepted*

Discussion thread: [here](#)

JIRA:

 Unable to render Jira issues macro, execution error.

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

Motivation

The public APIs to create topics, create partitions, and delete topics are heavy operations that have a direct impact on the overall load in the Kafka Controller. In the worst case, a misbehaving client could, intentionally or unintentionally, overload the controller which could affect the health of the whole cluster. For instance, electing a new partition leader could be delayed significantly, resulting in a partition which is not available. This issue arises particularly in shared clusters where clients can create and delete partitions at their own goodwill.

We did some basic performance experiments to understand the effect of high amount of concurrent of topics creations and topics deletions in a 9 brokers cluster:

- Our first experiment consisting in creating 5000 topics with 4 partitions and a replication factor of 3, at a rate of 10 requests/sec, containing 50 topics each, showed that the controller took approx. 2 minutes to process all the events queued up due to the topic creations.
- Our second experiments consisting in deleting the 5000 topics with a single request showed that the controller took approx. 2 minutes to process all the events queued up due to the topic deletions.

In both cases, it means that any cluster event queued up after all these events will be waiting for approx. 2 minutes in the queue before being processed.

In order to prevent a cluster from being harmed due to high concurrent topic and/or partition creations or topics deletions, we propose to introduce a new quota limiting the admission of these operations. All operations exceeding the quota will be rejected with an explicit error. The new quota is a resource level quota which complements the existing request quota.

Proposed Changes

Changes to Kafka Brokers

We propose to introduce a resource-level quota for the `CreateTopicsRequest`, the `CreatePartitionsRequest` and the `DeleteTopicsRequest` in the Kafka API layer that limits the number of partition mutations allowed per second. The number of partition mutations is defined as the number of partitions of a topic to be created, or the number of partitions to be added to an existing topic, or the number of partitions of a topic to be deleted. We chose to express the quota in partition mutations because the load in the controller is highly correlated to the number of created or deleted partitions.

The broker hosting the controller receives all the above request (already the case today) and can accept partition mutations with the rate R partitions per second and the maximum burst worth of B partitions. All the operations worth of N partition mutations are accepted until the burst is exhausted. All the remaining are rejected with a `THROTTLING_QUOTA_EXCEEDED` error. The complete algorithm is formalised below.

The reason to explicitly defines a burst B is to allow a burst of several mutations with a small or medium number of partitions within the same request. Typically, applications tend to send one request to create all the topics that they need. This would not work without an explicit burst.

Throttling Algorithm

Our current Rate based quota implementation accepts any burst and then compute the throttle to bring back the rate to the allowed quota. We can effectively define a burst by configuring the number of samples kept appropriately as the average rate is computed based on the sum of all samples divided by the overall time window. While this work well with workloads which generates multiple samples, we have found that it does not work well when the workload is bursty like ours in conjunction with a large window. The issue is that a unique and large sample can hold the average above the quota and this until it is discarded. Let's take an example to illustrate this behavior.

Let's imagine that we want to guarantee an average rate $R = 5$ mutations/sec while accepting a burst $B = 500$ mutations. This can be achieve by using the following parameters for our current rate based quota:

- Q (Quota) = 5
- S (Samples) = $B / R = 100$
- W (Time Window) = 1s (the default)

With this configuration, if a client sends a request with 7 topics with 80 partitions each at time T , worth a total of 560 mutations, it bring the average rate R to 5.6 ($560 / 100$). As the rate is above the defined quota, any subsequent mutations is rejected until the average rate gets back to the allowed quota. Currently, this is computed as follow for our current quotas: $((R - Q / Q * S * W)) = ((5.6 - 5) / 5 * 100 * 1) = 12$ secs. In practice, the average rate won't go back until the samples at time T is discarded so any new mutations won't be accepted before $S * W$. 100 secs in our case.

To overcome this, we propose to use a variation of the Tokens Bucket algorithm. Let's define:

- K : The number of tokens in the bucket
- B : The maximum number of tokens that can be accumulated in the bucket (burst)
- R : The rate at which we accumulate tokens
- T : The last time K got updated

At the time now, we update the number of tokens in the bucket with:

- $K = \min(K + (\text{now} - T) * R), B)$

A partition mutation with N partitions is admitted iff $K \geq 0$, and updates $K = K - N$. The partition mutation request is rejected otherwise. The variation here is that we allow the number of tokens to become negative to accommodate any number of partitions for a topic. The throttle time is defined with: $-K * R$.

In order to be able to reuse our existing configuration, the Tokens Bucket is configured based on Q , S and W as follow:

- $R = Q$
- $B = Q * S * W$

Our previous example would work as follow: $R = 5$, $B = 500$. The burst of 560 mutations brings the number of tokens to -60. The throttle time is $-(-60) / 5 = 12$ s.

The Tokens Bucket will be implemented as a new `MeasurableStat` that will be used within a `Sensor` alongside the existing `Rate`. The quota will be enforced on the Token Bucket metric only. The `Sensor` will be updated the handle the verification of any Tokens Bucket metrics.

Controller Mutation Quota Manager

We propose to introduce a new quota manager called `ControllerMutationQuotaManager` which implements the algorithm described above. The new quota manager will be configured by the new quota types: `controller_mutation_rate`.

Handling of new Clients

For new clients, the new `THROTTLING_QUOTA_EXCEEDED` error code will be used to inform client that a topic has been rejected. The `throttle_time_ms` field is used to inform the client about how long it has been throttled:

- $\text{throttle_time_ms} = \max(\text{<request throttle time>}, \text{<controller throttle time>} - \text{<waiting time in the purgatory if applied>})$

The channel will be muted as well when `throttle_time_ms > 0`. This is necessary to cope with misbehaving clients which would not honour the throttling time. This logic already exist and is used by the request quota thus we will reuse it.

Handling of old Clients

For old clients, we can't gracefully reject any requests. Therefore, we propose to always accept them and to throttle the client by muting its channel. The `throttle_time_ms` field is used to inform the client about how long it has been throttled:

- $\text{throttle_time_ms} = \max(\text{<request throttle time>}, \text{<controller throttle time>} - \text{<waiting time in the purgatory if applied>})$

The channel will be muted as well when `throttle_time_ms > 0`.

Co-Existence of Multiple Quotas

The new partition mutations quota can be used in conjunction with the existing request quota. When both are used, the client will be throttled by the most constraining of the two.

Handling of ValidateOnly

The `CreateTopicsRequest` and `CreatePartitionsRequest` offer the possibility to only validate their content by setting the flag `ValidateOnly`. As the validation does not generate load on the controller, we plan to not apply the quota in this particular case. We will clarify this in the documentation.

Changes to Kafka Admin Client

Automatic Retry

A new retryable `QuotaViolatedException` will be introduced on the client side. By default, the admin client will retry this error until `default.api.timeout.ms` is reached and also honour the throttling time. We will extend the retry mechanism already in place to support this. Once `default.api.timeout.ms` has been reached, the topics which were throttled will return the `QuotaViolatedException` to the caller. Any other errors will be given back to the caller as today. As the new exception is a retryable exception, it should not break code of users when they upgrade as handling retryable exception is already something that they should do.

Manual Retry

To complement this default behavior, we propose to add a new configuration to the admin client to allow users to opt-out from the automatic retry mechanism. It allows application to have fine grained control.

Public Interfaces

Protocol

The following new error code will be introduced:

- `THROTTLING_QUOTA_EXCEEDED`: It will be used when the defined throttling quota is exceeded but only for the above RPCs.

While discussing the naming of the error. We have also thought about using `LIMIT_QUOTA_EXCEEDED` in the future when a limit quota is exceeded. A limit is final error whereas a throttling error can be retried.

In order to be able to distinguish the new clients from the old ones, we will bump to version of the following requests/responses:

- `CreateTopicsRequest` and `CreateTopicsResponse` to version 6.
- `CreatePartitionsRequest` and `CreatePartitionsResponse` to version 3.
- `DeleteTopicRequest` and `DeleteTopicResponse` to version 5.

Starting from the bumped version, the new `THROTTLING_QUOTA_EXCEEDED` error will be used. It won't be used for older versions.

We will add the `ErrorMessage` field in the `DeleteTopicResponse` as follow:

```

{
  "apiKey": 20,
  "type": "response",
  "name": "DeleteTopicsResponse",
  // Version 1 adds the throttle time.
  //
  // Starting in version 2, on quota violation, brokers send out responses before throttling.
  //
  // Starting in version 3, a TOPIC_DELETION_DISABLED error code may be returned.
  //
  // Version 4 is the first flexible version.
  //
  // Version 5 adds the ErrorMessage field.
  "validVersions": "0-5",
  "flexibleVersions": "4+",
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "1+", "ignorable": true,
      "about": "The duration in milliseconds for which the request was throttled due to a quota violation, or
zero if the request did not violate any quota." },
    { "name": "Responses", "type": "[]DeletableTopicResult", "versions": "0+",
      "about": "The results for each topic we tried to delete.", "fields": [
        { "name": "Name", "type": "string", "versions": "0+", "mapKey": true, "entityType": "topicName",
          "about": "The topic name" },
        { "name": "ErrorCode", "type": "int16", "versions": "0+",
          "about": "The deletion error, or 0 if the deletion succeeded." },
        { "name": "ErrorMessage", "type": "string", "versions": "5+", "nullableVersions": "5+", "ignorable": true,
          "about": "The error message, or null if there was no error." }
      ]
    }
  ]
}

```

New Broker Configurations

We propose to introduce the following new configuration in the Kafka broker:

Name	Type	Default	Description
controller.quota.window.num	Int	11	The number of samples to retain in memory for alter controller mutations replication quotas
controller.quota.window.size.seconds	Int	1	The time span of each sample for controller mutations quotas

New Quota Types

We propose the introduce the following new quota types in the Kafka Broker:

Name	Type	Default	Description
controller_mutations_rate*	Double	Long.MaxValue	The rate at which mutations are accepted for the create topics request, the create partitions request and the delete topics request. The rate is accumulated by the number of partitions created or deleted.

They will be supported for <client-id>, <user> and <user, client-id> similar to the existing quotas. Defaults can be configured using the dynamic default properties at <client-id>, <user> and <user, client-id> levels.

* We keep the name intentionally generic to allow us to extend their coverage in the future.

New Broker Metrics

We propose to expose the following new metric in the Kafka Broker:

Group	Name	Tags	Description
ControllerMutation	rate	(user, client-id) - with the same rules used by existing quota metrics	The current rate.
ControllerMutation	tokens	(user, client-id) - with the same rules used by existing quota metrics	The remaining tokens in the bucket. < 0 indicates that throttling is applied.

ControllerMutation	throttle-time	(user, client-id) - with the same rules used by existing quota metrics	Tracking average throttle-time per user/client-id.
--------------------	---------------	--	--

New TokenBucket Metric

As mentioned, we propose to introduce a new metric named `TokenBucket` which implements the tokens bucket algorithm.

```
/**
 * The {@link TokenBucket} is a {@link MeasurableStat} implementing a token bucket algorithm
 * that is usable within a {@link org.apache.kafka.common.metrics.Sensor}.
 */
public class TokenBucket implements MeasurableStat {
    public TokenBucket() {
        this(TimeUnit.SECONDS);
    }

    public TokenBucket(TimeUnit unit) {
        // Implementation omitted
    }

    @Override
    public double measure(final MetricConfig config, final long timeMs) {
        // Implementation omitted
    }

    @Override
    public void record(final MetricConfig config, final double value, final long timeMs) {
        // Implementation omitted
    }
}
```

Admin API

As mentioned, we propose to introduce a new retryable `ThrottlingQuotaExceededException` exception which will be given back to the caller when a topic is rejected due to throttling. The new exception maps to the new `THROTTLING_QUOTA_EXCEEDED` error.

```
/**
 * Exception thrown if an operation on a resource exceeds the throttling quota.
 */
public class ThrottlingQuotaExceededException extends RetryableException {
    private int throttleTimeMs;

    public ThrottlingQuotaExceededException(int throttleTimeMs, String message) {
        super(message);
        this.throttleTimeMs = throttleTimeMs;
    }

    public ThrottlingQuotaExceededException(int throttleTimeMs, String message, Throwable cause) {
        super(message, cause);
        this.throttleTimeMs = throttleTimeMs;
    }

    public int throttleTimeMs() {
        return this.throttleTimeMs;
    }
}
```

The `CreateTopicsOptions`, `CreatePartitionsOptions`, and `DeleteTopicsOptions` will be extended to include a flag indicating if `ThrottlingQuotaExceededException` should be automatically retried by the AdminClient or not.

```
/**
 * Options for {@link Admin#createTopics(Collection)}.
 *
 * The API of this class is evolving, see {@link Admin} for details.
```

```

    */
@InterfaceStability.Evolving
public class CreateTopicsOptions extends AbstractOptions<CreateTopicsOptions> {

    private boolean retryOnQuotaViolation = true;

    /**
     * Set to true if quota violation should be automatically retried.
     */
    public CreateTopicsOptions retryOnQuotaViolation(boolean retryOnQuotaViolation) {
        this.retryOnQuotaViolation = retryOnQuotaViolation;
        return this;
    }

    /**
     * Returns true if quota violation should be automatically retried.
     */
    public boolean shouldRetryOnQuotaViolation() {
        return retryOnQuotaViolation;
    }
}

/**
 * Options for {@link Admin#createPartitions(Map)}.
 *
 * The API of this class is evolving, see {@link Admin} for details.
 */
@InterfaceStability.Evolving
public class CreatePartitionsOptions extends AbstractOptions<CreatePartitionsOptions> {

    private boolean retryOnQuotaViolation = true;

    /**
     * Set to true if quota violation should be automatically retried.
     */
    public CreatePartitionsOptions retryOnQuotaViolation(boolean retryOnQuotaViolation) {
        this.retryOnQuotaViolation = retryOnQuotaViolation;
        return this;
    }

    /**
     * Returns true if quota violation should be automatically retried.
     */
    public boolean shouldRetryOnQuotaViolation() {
        return retryOnQuotaViolation;
    }
}

/**
 * Options for {@link Admin#deleteTopics(Collection)}.
 *
 * The API of this class is evolving, see {@link Admin} for details.
 */
@InterfaceStability.Evolving
public class DeleteTopicsOptions extends AbstractOptions<DeleteTopicsOptions> {

    private boolean retryOnQuotaViolation = true;

    /**
     * Set to true if quota violation should be automatically retried.
     */
    public DeleteTopicsOptions retryOnQuotaViolation(boolean retryOnQuotaViolation) {
        this.retryOnQuotaViolation = retryOnQuotaViolation;
        return this;
    }

    /**
     * Returns true if quota violation should be automatically retried.
     */
    public boolean shouldRetryOnQuotaViolation() {

```

```
        return retryOnQuotaViolation;
    }
}
```

Kafka Topic Command

We propose to disable the automatic try of the `QuotaViolatedException` for the ``kafka-topics.sh`` command in order to not have the command blocked until the retry period is exhausted.

Kafka Config Command

The new name works similarly to the already existing [quota](#). It can be set or changed by using the ``kafka-configs.sh`` tool. When the new quota API is used, an old broker that does not support the new name will reject it with a `INVALID_REQUEST` error.

For instance, the above command defines a quota of 10 controller mutations per secs for the (user=user1, client-id=clientA):

```
> bin/kafka-configs.sh --bootstrap-server ... --alter --add-config 'controller_mutations_rate=10' --entity-type
users --entity-name user1 --entity-type clients --entity-name clientA
Updated config for entity: user-principal 'user1', client-id 'clientA'.
```

The above commands list the quota of (user=user1, client-id=clientA):

```
> bin/kafka-configs.sh --bootstrap-server ... --describe --entity-type users --entity-name user1 --entity-type
clients --entity-name clientA
Configs for user-principal 'user1', client-id 'clientA' are controller_mutations_rate=10,... (other quotas are
listed as well)
```

Known Limitations

Auto Topic Creation

As of writing this proposal, the auto topic creation feature uses Zookeeper to create topics. Therefore, using the feature is not compatible with this proposal at the moment. [KIP-590](#) (not adopted yet) will change this to use a regular `CreateTopicsRequest` instead of writing to Zookeeper directly. The request will be sent as a "cluster admin" therefore auto topic creations will be accounted for that user. We could improve the accounting in the future KIP if the need arise.

Alter Topics via Zookeeper

The proposal do not support the now deprecated way to create, expand or delete topics via Zookeeper because it is not possible to control nor reject changes written to Zookeeper directly.

How to Define the Quota?

While we can provide an exact formula to compute the ideal quota for a cluster, here are some general idea to find it.

- The first step consists in finding the actual real limits for your cluster. To achieve this, the best is to run workloads to create topics, create partitions and delete topics in isolation with different rate and partition size and to measure the impact on the controller. One can use the number of events queued in the controller to see how busy it is. Ideally, the number of events must remain small.
- Choose the quota based on the worst case and based on the extra load the cluster can sustain.
- Once the overall quota is defined, it can be divided among the tenant in the cluster.

Usually, all the tenants won't do operations at the same time so we believe that it is sage to over allocate the quota for each tenant.

It is also important to note the capacity of the controller does not scale with the number of node in the cluster. In fact, it is more the opposite as the controller will have more work to do with more nodes so quota may need to be slightly decrease when the cluster grow.

Compatibility, Deprecation, and Migration Plan

Compatibility with Old Clients

- By default, there is no impact on existing users since the throttling must be enabled.
- If the throttling is enabled in a cluster used by old clients, the old clients will be transparently throttled. If the client is throttled, its request will timeout and thus will be retry by the existing mechanism. If the retries fail, the application will receive a `TimeoutException` which is a retryable

exception. Handling `TimeoutException` and more generally retryable Exceptions is something that clients should already do thus impact is minimal.

Compatibility after Upgrading the Admin Client to the new version

- By default, the upgrade should be transparent since the Admin Client will automatically retry `QuotaViolationException` and return it to the caller only if the retry timeout is reached. In this case, the caller must at minimum handle the `RetryableException` and retry. Handling retryable Exceptions is something that we can safely expect from clients.

Rejected Alternatives

Update our existing Rate to behave like the Token Bucket

We have tried to modify our existing Rate to behave like the Token Bucket. That works but would not provide the same observability guarantees anymore. The Rate would behave like it does today and that is counter intuitive from an operations perspective. Therefore, we have preferred using a new metric along side the Rate to enforce the quota.

Usage based Quota

We have considered using a usage based quota similarly to the request quota. The usage would mainly be measured as the CPU time taken by the operations. While this would have the benefits to cover all the operations in the controller, we have decided against it for the following reasons:

- In order to make it work, we must have a way to associate the cost of an operation in the controller to a principal and/or a client id. Conceptually, this sounds a bit weird because topics are not associated to a unique principal or client id during their lifetime. A topic can be created by Principal A, then partitions can be added by Client 1, and finally it can be deleted by Principal B. This makes the accounting strange. That would work better if we would have a notion of tenant in Kafka.
- Measuring the CPU does not take into account the cost associated to all the requests sent out to achieve an operation.

Throttle the Execution instead of the Admission

The major diametrically opposed alternative consists in throttling the internal execution in the controller of the requests instead of throttling and rejecting them during their admission in the API layer. Requests would already be accepted.

While this alternative would benefit from having zero client side changes, we believe that it is a sub-optimal solution for the clients for the following reasons:

- By default, clients use a 30s request timeout meaning that throttling the execution for longer period would result in `TimeoutException` in the client which would automatically retry them. As it stands, the retry request would be rejected with a `TopicAlreadyExistsException` which is weird. This is due to the fact that topics are registered in Zookeeper to notify the controller and thus are already created. Even without this, it feels wrong with our current API. This could work if we would change our API to be asynchronous. For instance, we could provide an API which allows clients to get the status of a topic (e.g. creating, ready, deleting, etc.) and continue to use the existing API to only trigger the creation. This has its own compatibility challenges.
- Similarly, when users use the ``kafka-topics.sh`` tool, they would be blocked until the retry time is exhausted and also end up with a weird message.
- This approach would keep the amount of pending requests in the system unbound. We believe that it is better having explicit limit and reject the extra work in order to really protect the system well.

Throttle the Admission but without an explicit Error code

Another alternative would be to method used for old clients for the new clients as well. We believe that it is a sub-optimal long term solution for the following reasons:

- Muting the socket to limit the client to send more requests works fine for one client. The AdminClient is used from several places (different instances of an application, command line tools, etc) and thus does not use the same connection all the time. In this case, muting the socket is a sub-optimal solution.