# KIP-124 - Request rate quotas



## Status

**Current state**: *"Accepted"*

**Discussion thread**: *here*

**JIRA**: *KAFKA-4195*

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

## Motivation

Kafka currently supports quotas by data volume.  Clients that produce or fetch messages at a byte rate that exceeds their quota are throttled by delaying the response by an amount that brings the byte rate within the configured quota. However, if a client sends requests too quickly (e.g., a consumer with `fetch.max.wait.ms`=0), it can still overwhelm the broker even though individual request/response size may be small. It will be useful to additionally support throttling by request rate to ensure that broker resources are not monopolized by some users/clients.

### Scenarios:

1. Clients send requests too quickly - eg. a consumer with `fetch.max.wait.ms`=0 that polls continuously. Fetch byte rate quotas are not sufficient here, either request count quotas or request processing time quotas are required.
2. DoS attack from clients that overload brokers with continuous authorized or unauthorized requests. Either request count quotas or request processing time quotas that limit all unauthorized requests and all non-broker (client) requests is required.
3. Client sends produce requests with compressed messages where decompression takes a long time, blocking the request handler thread. Request processing time quotas are required since neither produce byte rate quotas nor request count quotas will be sufficient to limit the broker resources allocated to users/clients in this case.
4. Consumer group starts with 10 instances and then increases to 20 instances. Number of requests may double, so request counts increase, even though the load on the broker doesn't double since the number of partitions per fetch request has halved. Quotas based on request count per second may not be easy to configure in this case.
5. Some requests may use more of their quota on the network threads rather than the request handler threads (eg. disk read for fetches happen on the network threads). While quotas of processing time on the request handler thread limit the request rate in many cases above, for a complete request rate quota solution, network thread utilization also needs to be taken into account.
6. In clusters that enable both TLS and non-TLS endpoints, the load on the network thread is significantly higher for TLS. Network thread utilization needs to be taken into account to address this.

### Goals

This KIP proposes to control request handler (I/O) thread and network thread utilization using request processing time quotas that limit the amount of time within each quota window that can be used by users/clients.

## Limitations

This KIP attempts to avoid unintended denial-of-service scenarios from misconfigured application (eg. zero heartbeat interval) or a client library with a bug can cause broker overload. While it can reduce the load in some DoS scenarios, it does not completely protect against DoS attacks from malicious clients. A DDoS attack with large numbers of connections resulting in a large amount of expensive authentication or CPU-intensive requests can still overwhelm the broker..

# Public Interfaces

## Request quotas

Request quotas will be configured as the percentage of time a client can spend on request handler (I/O) threads and network threads within each quota window. A quota of *n%* represents *n%* of one thread, so the quota is out of a total capacity of (`(num.io.threads + num.network.threads) * 100`)% . Each request quota will be the total percentage across all request handler and network threads that a user/client may use in a quota window before being throttled. Since the number of threads allocated for I/O and network threads are typically based on the number of cores available on the broker host, request quotas represent the total percentage of CPU that may be used by the user/client. In future, if quotas are implemented for utilization of other types of threads, the same quota configuration can be used to limit the total utilization across all the threads monitored for quotas.

The limits will be applied to the same quota window configuration (`quota.window.size.seconds` with 1 second default) as existing produce/fetch quotas. This approach keeps the code consistent with the existing quota implementation, while making it easy for administrators to allocate a slice of each quota window to users/clients to control CPU utilization on the broker. If a client/user exceeds the request processing time limit, responses will be delayed by an amount that brings the rate within the limit. The maximum delay applied will be the quota window size.

## Default quotas

By default, clients will not be throttled based on request processing time, but defaults can be configured using the dynamic default properties at *<client-id>*, *<user>* and *<user, client-id>* levels. Defaults as well as overrides are stored as dynamic configuration properties in Zookeeper alongside the other rate limits.

## Requests exempt from throttling

Requests that update cluster state will be throttled only if authorization for `ClusterAction` fails. These are infrequent requests for cluster management, typically not used by clients:

1. StopReplica
2. ControlledShutdown
3. LeaderAndIsr
4. UpdateMetadata

To ensure that these exempt requests cannot be used by clients to launch a DoS attack, these requests will be throttled on quota violation if `ClusterActi` on authorization fails. `SaslHandshake` request will not be throttled when used for authentication, but will be throttled on quota violation if used at any other time.

All other requests may be throttled if the rate exceeds the configured quota. Produce/Fetch requests will return the total throttling time reflecting both bandwidth and utilization based throttling in the response. All other requests that may be throttled will have an additional field `throttle_time_ms` to indicate to the client that the request was throttled. The versions of these requests will be incremented.

Fetch and produce requests will continue to be throttled based on byte rates and may also be throttled based on thread utilization. Fetch requests used for replication will not be throttled based on request times since it is possible to configure `replica.fetch.wait.max.ms` and use the existing replication byte rate quotas to limit replication rate.

## Metrics and sensors

Two new metrics and corresponding sensors will be added to the broker for tracking `request-time` and `throttle-time` of each quota entity for the new quota type *Request.* These will be handled similar to the metrics and sensors for *Produce/Fetch.* A delay queue sensor with `queue-size` for the new quota type *Request* will also be added similar to the delay queue sensor for *Produce/Fetch*. All the metrics and sensors for request time throttling will be of similar format to the existing produce/fetch metrics and sensors for consistency, but with new group/name indicating the new quota type *Request,* keeping these separate from existing metrics/sensors.

An additional metric `exempt-request-time` will also be added for the quota type *`Request`* to track the time spent processing requests which are exempt from throttling. This will capture the total time for requests from all users/clients that are exempt from throttling so that administrators can view the CPU utilization of exempt requests as well.

Producers and consumers currently expose average and maximum producer/fetch request throttle time as JMX metrics. These metrics will be updated to reflect total throttle time for the producer or consumer including byte-rate throttling and request time throttling for all requests of the producer/consumer. Similar metrics may be added for the admin client in future.

## Tools

`kafka-configs.sh` will be extended to support request quotas.  A new quota property will be added, which can be applied to <*client-id*>, <*user*> or <*user , client-id*>:

- `request_percentage`: The percentage per quota window (out of a total of `(num.io.threads + num.network.threads) * 100%`) for requests from the user or client, above which the request may be throttled.

For example:

```
bin/kafka-configs  --zookeeper localhost:2181 --alter --add-config 'request_percentage=50' --entity-name user1 --entity-type users
```

Default quotas for <*client-id*>, <*user*> or <*user, client-id*> can be configured by omitting entity name. For example:

```
bin/kafka-configs  --zookeeper localhost:2181 --alter --add-config 'request_percentage=200' --entity-type users
```

## Protocol changes

All client requests which are not exempt from request throttling will have a new field containing the time in milliseconds that the request was throttled for.

**Offsets Response**

```
Offsets Response (Version: 2) => throttle_time_ms [responses]
  throttle_time_ms => INT32 (new)
  responses => topic [partition_responses]
    topic => STRING
    partition_responses => partition error_code timestamp offset
      partition => INT32
      error_code => INT16
      timestamp => INT64
      offset => INT64
```

**Metadata Response**

```
Metadata Response (Version: 3) => throttle_time_ms [brokers] cluster_id controller_id [topic_metadata]
  throttle_time_ms => INT32 (new)
  brokers => node_id host port rack
    node_id => INT32
    host => STRING
    port => INT32
    rack => NULLABLE_STRING
  cluster_id => NULLABLE_STRING
  controller_id => INT32
  topic_metadata => topic_error_code topic is_internal [partition_metadata]
    topic_error_code => INT16
    topic => STRING
    is_internal => BOOLEAN
    partition_metadata => partition_error_code partition_id leader [replicas] [isr]
      partition_error_code => INT16
      partition_id => INT32
      leader => INT32
      replicas => INT32
      isr => INT32
```

**OffsetCommit Response**

```
OffsetCommit Response (Version: 3) => throttle_time_ms [responses]
  throttle_time_ms => INT32 (new)
  responses => topic [partition_responses]
    topic => STRING
    partition_responses => partition error_code
      partition => INT32
      error_code => INT16
```

**OffsetFetchResponse**

```
OffsetFetch Response (Version: 3) => throttle_time_ms [responses] error_code
  throttle_time_ms => INT32 (new)
  responses => topic [partition_responses]
    topic => STRING
    partition_responses => partition offset metadata error_code
      partition => INT32
      offset => INT64
      metadata => NULLABLE_STRING
      error_code => INT16
  error_code => INT16
```

**GroupCoordinator Response**

```
GroupCoordinator Response (Version: 1) => throttle_time_ms error_code coordinator
  throttle_time_ms => INT32 (new)
  error_code => INT16
  coordinator => node_id host port
    node_id => INT32
    host => STRING
    port => INT32
```

**JoinGroup Response**

```
JoinGroup Response (Version: 2) => throttle_time_ms error_code generation_id group_protocol leader_id member_id
[members]
  throttle_time_ms => INT32 (new)
  error_code => INT16
  generation_id => INT32
  group_protocol => STRING
  leader_id => STRING
  member_id => STRING
  members => member_id member_metadata
    member_id => STRING
    member_metadata => BYTES
```

**Heartbeat Response**

```
Heartbeat Response (Version: 1) => throttle_time_ms error_code
  throttle_time_ms => INT32 (new)
  error_code => INT16
```

**LeaveGroup Response**

```
LeaveGroup Response (Version: 1) => throttle_time_ms error_code
  throttle_time_ms => INT32 (new)
  error_code => INT16
```

**SyncGroup Response**

```
SyncGroup Response (Version: 1) => throttle_time_ms error_code member_assignment
  throttle_time_ms => INT32 (new)
  error_code => INT16
  member_assignment => BYTES
```

**DescribeGroups Response**

```
DescribeGroups Response (Version: 1) => throttle_time_ms [groups]
  throttle_time_ms => INT32 (new)
  groups => error_code group_id state protocol_type protocol [members]
    error_code => INT16
    group_id => STRING
    state => STRING
    protocol_type => STRING
    protocol => STRING
    members => member_id client_id client_host member_metadata member_assignment
      member_id => STRING
      client_id => STRING
      client_host => STRING
      member_metadata => BYTES
      member_assignment => BYTES
```

**ListGroups Response**

```
ListGroups Response (Version: 1) => throttle_time_ms error_code [groups]
  throttle_time_ms => INT32 (new)
  error_code => INT16
  groups => group_id protocol_type
    group_id => STRING
    protocol_type => STRING
```

**ApiVersions Response**

```
ApiVersions Response (Version: 1) => throttle_time_ms error_code [api_versions]
  error_code => INT16
  api_versions => api_key min_version max_version
    api_key => INT16
    min_version => INT16
    max_version => INT16
  throttle_time_ms => INT32 (new)
```

**CreateTopics Response**

```
CreateTopics Response (Version: 2) => throttle_time_ms [topic_errors]
  throttle_time_ms => INT32 (new)
  topic_errors => topic error_code error_message
    topic => STRING
    error_code => INT16
    error_message => NULLABLE_STRING
```

**DeleteTopics Response**

```
DeleteTopics Response (Version: 1) => throttle_time_ms [topic_error_codes]
  throttle_time_ms => INT32 (new)
  topic_error_codes => topic error_code
    topic => STRING
    error_code => INT16
```

# Proposed Changes

## Quota entity

Request quotas will be supported for <client-id>, <user> and <user, client-id> similar to the existing produce/fetch byte rate quotas. In addition to produce and fetch rates, an additional quota property will be added for throttling based on thread utilization. As with produce/fetch quotas, request quotas will be per-broker. Defaults can be configured using the dynamic default properties at <*client-id*>, <*user*> and <*user, client-id*> levels.

## Request quotas

Quotas for requests will be configured as the percentage of time within a quota window that a client is allowed to use across all of the I/O and network threads. The total thread capacity of `((num.io.threads + num.network.threads) * 100%)` can be distributed between clients/users. For example, with the default configuration of 1 second quota window size, if user *alice* has a request quota of 1%, the total time all clients of *alice* can spend in the request handler and network threads in any one second window is 10 milliseconds. When this time is exceeded, a delay is added to the response to bring *alice*'s usage within the configured quota. The maximum delay added to any response will be the window size. The calculation of delay will be the same as the current calculation used for throttling produce/fetch requests:

- If O is the observed usage and T is the target usage over a window of W, to bring O down to T, we need to add a delay of X to W such that: `O * W / (W + X) = T`.
- Solving for X, we get `X = (O - T)/T * W`.
- The response will be throttled by `min(X, W)`

Network thread time will be recorded for each request without performing throttling when the time is recorded. When I/O thread time is recorded, throttling will be performed, taking into account the total processing time of the user/client in network threads and I/O threads in the quota window. This simplifies the handling of network thread utilization without integrating the throttling mechanism into the network layer.

The maximum throttle time for any single request will be the quota window size (one second by default). This ensures that timing-sensitive requests like heartbeats are not delayed for extended durations. For example, if a user has a quota of 0.1% and a stop-the-world GC pause takes 100ms during the processing of the user's request, we don't want all the requests from the user to be delayed by 100 seconds. By limiting the maximum delay, we reduce the impact of GC pauses and single large requests. To exploit this limit to bypass quota limits, clients would need to generate requests that take significantly longer than the quota limit. If R is the amount of time taken process one request and the user has C active connections, the maximum amount of time a user/client can use per quota window is `max(quota, C * R)`. In practice, quotas are expected to be much larger than the time taken to process individual requests and hence this limit is sufficient. Byte rate quotas will also additionally help to increase throttling in the case where large produce/fetch requests result in larger per-request time. DoS attacks using large numbers of connections is not addressed in this KIP.

## Sample configuration in Zookeeper

**Sample quota configuration**

```
// Quotas for user1
// Zookeeper persistence path /config/users/<encoded-user1>
{
    "version":1,
    "config": {
        "producer_byte_rate":"1024",
        "consumer_byte_rate":"2048",
                "request_percentage" : "50"
    }
}
```

## Co-existence of multiple quotas

Produce and fetch byte rate quotas will continue to be applied as they are today. Request processing time throttling will be applied after applying byte rate throttling if necessary. For example, if a large number of small produce requests are sent followed by a very large one, both request time quota and produce byte rate quota may be violated by the same request. The produce byte rate delay is applied first. Request time quota is checked only after the produce delay is applied. The request quota is perhaps no longer violated (or the delay may be lower due to the first byte-rate violation delay that has already been applied). If the request quota check still fails with quota violation, the delay for request quota violation is applied to the response.

## Metrics and sensors

Two new metrics and corresponding sensors will be added to track `request-time` and `throttle-time` of each quota entity for quota type *Request.* Th e `request-time` sensor will be configured with the quota for the user/client so that quota violations can be used to add delays to the response. Quota window configuration (`quota.window.size.seconds`) will be the same as the existing configuration for produce/fetch quotas: 1 second window with 11 samples retained in memory by default. A new delay queue sensor will also be added for quota type *Request*. All the new sensor names (`Request-<quota-entity>`, `RequestThrottleTime-<quota-entity>` and `Request-delayQueue`) are prefixed by the quota type, making these sensors consistent with existing sensors for *Produce/Fetch*. The new metrics will be in the metrics group *`Request`*, distinguishing these from similar metrics for *Pro duce/Fetch* byte rates.

Metrics and sensors will be expired as they are today for *Produce/Fetch* quotas.

On the client side, the existing produce and fetch sensors will track total throttle time of all requests from producers and consumers respectively. This will include both bandwidth as well as utilization based throttling. Throttle time recording will be moved to `NetworkClient` with appropriate sensor parameters so that produce or fetch sensors are updated based on whether the client corresponds to a producer or consumer. This will also enable addition of similar sensors/metrics for admin clients in future.

# Compatibility, Deprecation, and Migration Plan

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

- None, since by default clients will not be throttled on request processing time.

*If we are changing behavior how will we phase out the older behavior?*

- Quota limits for request processing time can be configured dynamically if required. Older versions of brokers will ignore request time quotas.
- If request quotas are configured on the broker, throttle time will be returned in the response to clients only if the client supports the new version of requests being throttled.
- If request quotas are configured, client produce/fetch throttle-time metrics will reflect total throttle time including bandwidth and utilization based throttling of these requests. The throttle time returned in produce/fetch responses will include this total throttle time.

# Test Plan

One set of integration and system tests will be added for request throttling. Since most of the code can be reused from existing producer/consumer quota implementation and since quota tests take a significant amount of time to run, one test for testing the full path should be sufficient.

# Rejected Alternatives

### Use request rate instead of request processing time for quota bound

Produce and fetch quotas are configured as byte rates (e.g. 10 MB/sec) and enable throttling based on data volume. Requests could be throttled based on request rate (e.g. 10 requests/sec), making request quotas consistent with produce/fetch quotas. But the time taken for processing different requests can vary significantly and since the goal of the KIP is to enable fair allocation of broker resources between users/clients, request processing time is a better metric suited to this quota.

### Use request time percentage across all threads instead of per-thread percentage for quota bound

The KIP proposes to use a quota that specifies the percentage of time as allocated to a client/user as per-thread value, out of a total capacity of `((num.io.threads+num.network.threads) * 100%)`, with the total request processing time measured across all I/O and network threads. An alternative would be to configure relative percentage out of a fixed total capacity of 100. Absolute quota was chosen to avoid automatic changes to client quota values when `num.io.threads or num.network.threads` is modified. Since threads are typically based on the number of cores on the broker host, the per-thread quota percentage reflects the % of cores allocated to client/user. This is consistent with other CPU quotas like *cgroup* and the way CPU usage is reported by commands like *top.*

### Use fractional units of threads instead of percentage for quota bound

The KIP proposes to use a quota that specifies the total percentage of time within each quota window allocated to a client/user, out of a total capacity of `((num.io.threads+num.network.threads) * 100%)`. An alternative would be to model each thread as one unit and configure quota as a fraction of the total number of available units. Percentage was chosen instead of fractional units of threads so that the same `request_percentage` representing CPU utilization can be continue to be applied even if other threads are added in future.

### Allocate percentage of request handler pool as quota bound

An alternative to monitoring request rate will be to model the request handler pool as a shared resource and allocate a percentage of the pool capacity to each user/client. But since only one request is read into the pool from each connection, this would be a measure of the number of concurrent connections per user/client rather than the rate of usage (a single or small number of connections can still overload the broker with a continuous sequence of requests). And it will be harder to compute the amount of time to delay a request when the bound is violated.

### Exempt timing-sensitive requests such as consumer heartbeat

Consumer requests such as heartbeat are timing sensitive and it is possible that throttling these requests could result in the consumer falling out of the consumer group. However, if we don't throttle some requests, a misbehaving application or a misconfigured application with low heartbeat interval or a client library with a bug can cause broker overload. To protect the cluster from DoS attacks and avoid these overload scenarios, all client requests will be throttled. In most deployments, quotas will be set to a large enough value to provide a safety net rather than very small values that throttle well-behaved clients, so this shouldn't be an issue.

### Use separate quotas for timing-sensitive requests such as consumer heartbeat

Two different quotas can configured to ensure that timing-sensitive requests like consumer heartbeat are not impacted by throttling of other requests. But this will be adding more configuration for administrators and more metrics etc. and a single quota will be simpler to use. Since coordinator requests are on their own connection and throttling is performed only after the request is processed, the impact of request quotas should be limited.