KIP-13 - Quotas

Status

Current state: *Accepted*

Discussion thread: http://markmail.org/thread/hpp4ucysifucwmgd

JIRA: KAFKA-2083

Motivation

Currently, the Kafka cluster does not have the ability to throttle/rate limit producers and consumers. It is possible for a consumer to consume extremely fast and thus monopolize broker resources as well as cause network saturation. It is also possible for a producer to push extremely large amounts to data thus causing memory pressure and large IO on broker instances. We need a mechanism to enforce quotas on a per-client basis.

In this KIP, we will discuss a proposal to implement quotas in Kafka. We are proposing an approach that can used for both producer and consumer side quotas.

Public Interfaces

1. Metrics - The Quota Metrics which will be captured on a per-clientId basis, will be exposed to JMX. These are new metrics and do not use codahale. Instead, they use KM (Kafka Metrics) which is a new metrics library written for Kafka. More details in the Metrics section below.
2. Response format - The Fetch and Produce response formats will evolve to include a new field called *'throttleTimeMs'* which will convey the amount of time a request was throttled.

Proposed Changes

The changes are mainly focussed around Quota Policy, Distribution, Metrics, Quota Actions and config management.

Quota Policy

This section basically explains what dimensions we throttle on. The proposal is to throttle based on client IDs. Any client using the system presents a client id or consumer group (producer or consumer). Each client will receive a default quota (for e.g. 10MBps read, 5MBps write) which can be overridden on a per-client basis dynamically. In addition, there will be a quota reserved for clients not presenting a client id (for e.g. simple consumers not setting the id). This will default to an empty client id ("") and all such clients will share the quota for that empty id (which should be the default quota). Producer side quotas are defined in terms of bytes written per second per client id. Consumer quotas as defined in terms of bytes read per second per client id. We do expect that there will be some high volume clients that require more than the default quota. In short, we are proposing fixed quotas for everyone but the top k outliers which can justify custom quotas. If clients violate their quota, we will throttle fetch/produce requests for them.

For example: if a client deletes their consumer offsets and "bootstraps" quickly, this should cause a "read bytes per second" violation and will throttle (slow down) that consumer group. It should have no effect on any other clients.

Quota Distribution

We need a mechanism to distribute a client quota across all the brokers in a cluster. Our proposal is to define quota on a per-broker basis**.** Each client can publish/fetch a maximum of X MBps (configurable) per broker before it gets throttled. This approach assumes that client requests are properly distributed among all the broker instances which may not always be the case. However, this is much better than having a fixed cluster wide bandwidth per client because that would require a mechanism to share current quota usage per-client among all the brokers. This can be very tricky to implement and is outside the scope of this proposal.

Quota Actions

How do the Kafka Server react when it detects a quota violation? In our proposal, the broker does not return an error rather it attempts to slow down a client exceeding it's quota. The slowdown will not be return an error to the client but will be visible through metrics. The behavior is subtly different for producer and consumer.

In case of producer, the broker will append the request batch to the log but will not return a response immediately. Instead, the response is inserted into a DelayedOperationPurgatory for a certain period of time before returning to the client. It is important that the append be done before delaying the client to avoid holding large batches in the broker heap and running out of memory.

In case of consumer, the request needs to be delayed before performing the read to avoid memory pressure.

Here's how the code might look inside the Broker (in ReplicaManager or helper class). This is just for illustration purposes.

```
def appendMessages(...) {
  log.appendToLocalLog()
  ...
  val metric : Sensor = metrics.get(sensorName)
 try {
    metric.record(produceSize)
  } catch (QuotaViolationException ex) {
    val delayedProduce = new DelayedProduce(ex.getDelayTime(), ....)
    delayedProducePurgatory.tryCompleteElseWatch(delayedProduce, producerRequestKeys)
  }
}

def fetchMessages(...) {
  val metric : Sensor = metrics.get(sensorName)
  try {
    metric.record(fetchSize)
  } catch (QuotaViolationException ex) {
    val delayedFetch = new DelayedFetch(ex.getDelayTime(), ....)
    delayedFetchPurgatory.tryCompleteElseWatch(delayedFetch, fetchRequestKeys)
  }
}
```

Metrics

Kafka server has decided to migrate it's metrics to Kafka Metrics (KM). For more details read this. Kafka metrics maintains a Sensor for each measured metric. This can be configured with a "Quota" which is a bound of the min and max value of a metric. If recording a value causes a bound to get exceeded, a QuotaViolationException is thrown. We will add a bytes-in/bytes-out rate sensor per clientId configured with 'N' 1 second windows. This is already supported in MetricConfig. We can make 'N' configurable and use a good default (10). This lets us precisely track the per-client-broker rate per second. Having small windows makes the quota system very responsive. A quota will also be configured when creating these metrics. For default clients, the quota used will simply be the default quota value. For clients with overridden quotas, we can use the custom value.

In order to fully support quotas, we need to make some changes to the metrics package as defined below.

Delay time in QuotaViolationException

Upon Quota violation, we need to identify how long we should delay requests for the client. It seems natural to have the QuotaViolationException return this information. The great thing about this is that we can catch any QuotaViolationException in the Kafka server and it will have the delay time built into it. This gives us the ability to enforce quotas over any metric we track.

Here's an example of computing the delay time. Assume a quota of 5MBps with a 10 second window. There is a client producing at exactly 5MBps. If it suddenly produces a 15MB batch in the next second, this causes a quota violation because it will have produced

60MB over the last 10 seconds (5*9 + 15 = 60MB).

```
Delay Time = (overall produced in window - quotabound)/Quota limit per second

In the example above, we will delay for 2 seconds.
Delay Time = (60 - 50)/5 = 2 second delay.
```

Configuration Management

We need a mechanism to configure the default quotas and the per-client overrides. There is general agreement that we eventually need dynamic configs per-client to fully operationalize quotas but for the purposes of this proposal, we will proceed with static configs.

```
// Default bytes-out per consumer.
quota.consumer.default=2M
quota.producer.default=2M

// Overrides
quota.producer.override="clientA:4M,clientB:10M"
quota.consumer.override="clientC:3M,clientD:5M"
```

There is a separate discussion for dynamic configs per-client that isn't fully fleshed out. If it makes sense and people agree on a final design, we can model quotas using it.

Tooling/Monitoring Changes

Upon implementation, we will start exposing the following metrics to JMX. Since these are new metrics there are no backward compatibility issues. These metrics will be documented prior to the release.

- per-client byte rate metrics
- per-client metric to indicate throttle times

Client Status Code

In the current response protocol, there is no way to return the quota status back to the client. How do clients know if they are being throttled or not? Our proposed solution is to add a new field in the response that indicates the quota status called *'throttleTimeMs'*. This will require us to increment the protocol version for both producer and consumer. Only clients sending version (1) of those requests will receive the quota status flag in the response. Version 1 of the request has the same format as version 0. On the client side (producer and consumer), we can derive and expose the following metrics over a time window:

- Max request throttle time - The largest throttle time across requests in the time window. If the client is not throttled, it will simply expose 0.
- Avg request throttle time - The average throttle time across requests in the time window.

The response protocol can define a top-level field called "throttleTime".

```
// Current fetch response
public static final Schema FETCH_RESPONSE_V0 = new Schema(new Field("responses", new ArrayOf
(FETCH_RESPONSE_TOPIC_V0)));


// Proposed fetch response
public static final Schema FETCH_RESPONSE_V1 = new Schema(new Field("responses", new ArrayOf
(FETCH_RESPONSE_TOPIC_V0)), new Field("throttle_time_ms", INT32, "Amount of time in milliseconds the request
was throttled if at all"));

// Current produce response
public static final Schema PRODUCE_RESPONSE_V0 = new Schema(new Field("responses",
                                                            new ArrayOf(new Schema(new Field("topic",
STRING),
                                                                                   new Field
("partition_responses",
                                                                                             new ArrayOf
(new Schema(new Field("partition",

INT32),

new Field("error_code",

INT16),

new Field("base_offset",

INT64)))))))));

// Proposed produce response
public static final Schema PRODUCE_RESPONSE_V1 = new Schema(new Field("responses",
                                                            new ArrayOf(new Schema(new Field("topic",
STRING),
                                                                                   new Field
("partition_responses",
                                                                                             new ArrayOf
(new Schema(new Field("partition",

INT32),

new Field("error_code",

INT16),

new Field("base_offset",

INT64)))))),
                                                                new Field("throttle_time_ms", INT32, "Amount of
time in milliseconds the request was throttled if at all"));
```

Compatibility, Deprecation, and Migration Plan

- *What impact (if any) will there be on existing clientss?*
  Once the brokers are upgraded, existing clients will receive a default quota. A client's produce and fetch latency may increase if he is throttled. Clients that require more than the default quota will need to have overrides in advance to avoid getting throttled.
- *If we are changing behavior how will we phase out the older behavior?*
  1. Set "inter.broker.protocol.version" to that of the previous release (0.8.2)
  2. Perform a rolling upgrade on the brokers so that they can all decipher the latest version of the Fetch response as described above.
  3. Change to "inter.broker.protocol.version" config to that of current release (0.9?).
  4. Rolling upgrade of the brokers

Rejected Alternatives

Topic Based *Quotas* - We initially considered doing topic based quotas. This was rejected in favor of the client based quotas since it seemed more natural to throttle clients than topics. Multiple producers/consumers can publish/fetch from a single topic. This makes it hard to eliminate interference of badly behaved clients. Topic based quotas also are harder to reason about when clients are accessing multiple topics (e.g. wildcard consumer). If quota for 1 topic is violated, how do we handle requests for the other topics?

Quota Distribution

A) Instead of throttling by client id, we can consider throttling by a combination of "clientId, topic". The default quota is also defined on this tuple i.e. 10Mbps for each "clientId, topic". The obvious downside here is that a wildcard consumer can allocate almost infinite quota for itself if it subscribes to all topics. Producers can also be allocated very large quotas based on the number of topics they publish to. In practice, this approach needs to be paired with a cap on the maximum amount of data a client can publish/consume. But this cluster-wide cap again requires global co-ordination which is we mentioned is out of scope.

B) If we instead model the quotas on a per-topic basis, provisioned quota can be split equally among all the partitions of a topic. For example: If a topic T has 8 partitions and total configured write throughput of 8MBps, each partition gets 1Mbps. If a broker hosts 3 leader partitions for T, then that topic is allowed 3MBps on that broker regardless of the partitions that traffic is directed to. In terms of quota distribution, this is the most elegant model. However, since there can be multiple producers/consumers of a topic, a single misbehaving client can cause multiple clients to get throttled which is undesirable behavior. This is why we have chosen to model quotas on a per-client basis. Since a client can consume from/produce to any number of topic-partitions, it is difficult to track accurately a client's quota usage across the entire cluster. This would require some sort of a gossip mechanism we don't currently have. This is why we choose to model quota distribution a per-broker basis.

Quota Actions

A) Immediately return an error: This is the simplest possible option. However, this requires clients to implement some sort of a back off mechanism since retrying immediately will likely make things worse.

B) Delay the request and return error: If a client is currently exceeding its quota, we can park the request in purgatory for 'n' milliseconds. After the request expires, the broker can return an error to the client. This is a nice feature because it effectively throttles the client up to the request timeout and makes it less critical for them to implement backoff logic. After the error is caught by the client, they can retry immediately. Note that in case of producers, we should be careful about retaining references to large Message bodies because we could easily exhaust broker memory if parking hundreds of requests. The downside of this approach is that it requires the clients to catch quota violation errors and retry their requests.

We have decided against (A) because it relies on clients to correctly implement back off behavior. A badly behaved client can simply retry in a loop and cause network saturation. In case of (B), clients have to handle an additional error code for quota and also build retry logic. In addition, there is a possibility of producer data loss if the request does not succeed after a certain number of retries.

There is more context of the email thread on this particular issue