KIP-936: Throttle number of active PIDs

Status

Current state: Under Discussion

Discussion thread: *here*

JIRA: *here*

Motivation

The motivations here are similar to KIP-854 Separate configuration for producer ID expiry. Idempotent producers became the default in Kafka since KIP-679: Producer will enable the strongest delivery guarantee by default as a result of this all producer instances will be assigned PID. The increase of number of PIDs stored in Kafka brokers by `ProducerStateManager` exposes the broker to OOM errors if it has a high number of producers, rogue or misconfigured client(s). Here are few of use-cases that might cause increase of PIDs:

- Clients fail after produce and enter a repeated restart state for a while.
- Users initialise a new Producer instance every time they encounter any type of error instead of waiting for the producer instance to auto recover.
- Users who initialise new Producer per batch.

As a result of this the broker will hit OOM and become offline. The only way to recover from this is to increase the heap.

KIP-854 added separated config to expire PID from transaction IDs however the broker is still exposed to OOM if it has high number of PID before `producer.id.expiration.ms` is exceeded. And decreasing the value of `producer.id.expiration.ms` will impact all clients which not desired all the time. It would be more better to target only inefficient users and stopping them from crowding the map of PIDs to their `ProducerState` by `ProducerStateManager`.

This KIP proposes to throttle the number PIDs at the leader of the partition by adding a new rating quota that will be applied during handling the `PRODUCE` request. This way the broker can reject only misbehaving users early on in the process and protect itself without impacting good behaving users.

Proposed Changes

We propose adding the new `QuotaManager` called `ProducerIdQuotaManager` on the PRODUCE request level in the Kafka API that limits the number of active PIDs per user (KafkaPrincipal). The number of active PIDs will be defined as a rate within a period of time (similar to ControllerMutation quota).

The new quota will be applied based on

- `/config/users/<user>`
- `/config/users/<default>`

`ProducerIdQuotaManager`

- will be applied per KafkaPrincipal as it's a smaller subset which is known to the administrator of the cluster. It will not be applied to ClientId (which is not enforced by the client config) nor a combination of KafkaPrincipal and ClientId.

- will keep a cache of user (KafkaPrincipal) to unique active PIDs to track active PIDs. The cache is used only to decide if we encountered PID before or not for a given KafkaPrincipal and not as a counter. The cache will be implemented using a simple bloom filter controlled by time to avoid any unwanted growth that might cause OOM. (More details on this is explained in the next section)
- will check if the caching layer contains the PID for this given user or not.
    - If cache layer doesn't contain the PID then the quota manager will add the PID for this user to the cache and increment quota rating metrics.
    - If the cache layer contains the PID then there is no need to update the cache or increment the quota rating metrics.
- will throttle users once it reach the allowed quota. The quota manager will throw `QuotaValidationException` similar to existing quotas in Kafka. And the client will receive `ProduceResponse` with `throttleTimeMs` similar to throttling of bandwidth or request.

Caching layer to track active PIDs per KafkaPrincipal

The cache will be represented as a map of KafkaPrincipal to timed controlled bloom filter (TimedBloomFilter). User's PIDs in the bloom filter in the caching layer will go through the filling steps in its lifecycle:

- Step1: Adding the first PID for user (let's call this user userA) will create an entry to this user in the cache with its first bloom filter in the cached map (let call it bloom_filter_1)
    - Now the cache layer is storing the following entry for the user

```
Map { "UserA" -> TimedBloomFilter {
                                         bloom_filter_1_create_timestamp -> bloom_filter_1
                                        }
        }
```

    - Any new PIDs will be added to this cache for the first half of `producer.id.quota.window.size.seconds`
- Step2: A new bloom filter will be created along side the old one for the user for the second half of `producer.id.quota.window.size.seconds (let's call it bloom_filter_2).`
    - All new PIDs from this point will be added to the new filter.
    - Both bloom filters will be used to check if we came across the same PID before or not.
    - Now the cache layer is storing the following entry for the user

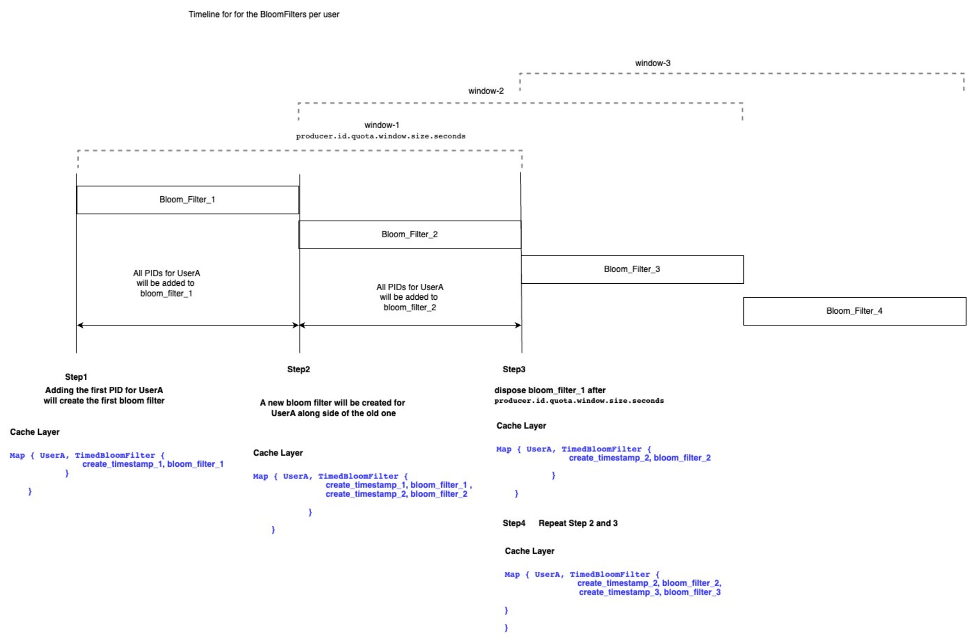
```
Map { "UserA" -> TimedBloomFilter {
                                         bloom_filter_1_create_timestamp -> bloom_filter_1,
                                          bloom_filter_2_create_timestamp -> bloom_filter_2
                                        }
        }
```

- Step3: The old bloom filter (bloom_filter_1) will be disposed once we reach `producer.id.quota.window.size.seconds` .
    - Now the cache layer is storing the following entry for the user

```
Map { "UserA" -> TimedBloomFilter {
                                          bloom_filter_2_create_timestamp -> bloom_filter_2
                                        }
        }
```

- `Step4: Repeat steps 2, and 3 on bloom_filter_2.`
    - Now the cache layer is storing the following entry for the user

```
Map { "UserA" -> TimedBloomFilter {
                                         bloom_filter_2_create_timestamp -> bloom_filter_2,
                                          bloom_filter_3_create_timestamp -> bloom_filter_3
                                        }
        }
```


Timeline for for the BloomFilters per user
window-3
window-2
window-1
producer.id.quota.window.size.seconds
Bloom_Filter_1
Bloom_Filter_2
Bloom_Filter_3
Bloom_Filter_4
All PIDs for UserA will be added to bloom_filter_1
All PIDs for UserA will be added to bloom_filter_2
Step1
Adding the first PID for UserA will create the first bloom filter
Step2
A new bloom filter will be created for UserA along side of the old one
Step3
dispose bloom_filter_1 after producer.id.quota.window.size.seconds
Step4 Repeat Step 2 and 3
Cache Layer
Map { UserA, TimedBloomFilter { create_timestamp_1, bloom_filter_1 } }
Map { UserA, TimedBloomFilter { create_timestamp_1, bloom_filter_1 , create_timestamp_2, bloom_filter_2 } }
Map { UserA, TimedBloomFilter { create_timestamp_2, bloom_filter_2 } }
Map { UserA, TimedBloomFilter { create_timestamp_2, bloom_filter_2, create_timestamp_3, bloom_filter_3 } }

```
This way each user within a window will have 2 bloom filters (both are available for read while only one of them
active for write) and when we need to check if we came across PID before or not for a given user we will check if
any of the bloom filters contains the PID.
```

Note: User entry in the cached map will be entirely removed from the caching layer if it doesn't have any active bloom filters attached to it anymore. Performing a cleanup for inactive users.

Public Interfaces

New Broker Configurations

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

Name	Type	Default	Description
`producer.id.quota.window.num`	Int	11	The number of samples to retain in memory for alter producer id quotas
`producer.id.quota.window.size.seconds`	Int	3600	The time span of each sample for producer id quotas. Default is 1hr.
`producer.id.quota.cache.cleanup.scheduler.interval.ms`	Int	10	The frequency in ms that the producer id quota manager will check for disposed cached window.

`quota.window.num` and `quota.window.size.seconds` are similar to the rest of Kafka Quota Configs.

New Quota Types

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

Name	Type	Default	Description
`producer_ids_rate`	Double	Long.MaxValue	The rate at which produce request are accepted with new producer id.

The config will be supported for `<user>` only as we are trying to avoid the growth of the caching layer and `<user>` are known number for the operator of the cluster and could be controlled more than the client-id.

- Extend `QuotaConfigs` to handle the new quota type

```
public class QuotaConfigs {
    public static final String PRODUCER_ID_RATE_OVERRIDE_CONFIG = "producer_ids_rate";
         public static final String PRODUCER_ID_RATE_DOC = "A rate representing the upper bound of active
producer ids."

         public static ConfigDef buildProducerIdsConfig(ConfigDef configDef ) {
                configDef.define(PRODUCER_ID_RATE_OVERRIDE_CONFIG, ConfigDef.Type.DOUBLE, Integer.MAX_VALUE,
            Integer.valueOf(Integer.MAX_VALUE).doubleValue(), ConfigDef.Importance.MEDIUM,
PRODUCER_ID_RATE_DOC);
        return configDef;
         }
}
```

- Extends `DynamicConfig` and `ClientQuotaControlManager.configKeysForEntityType` to handle the new quota.

New Broker Metrics

The new metrics will be exposed by the broker:

Group	Name	Tags	Description
`ProducerIds`	rate	user	The current rate
`ProducerIds`	tokens	user	The remaining tokens in the bucket. < 0 indicates that throttling is applied.
`ProducerIds`	throttle-time	user	Tracking average throttle-time per user.

Client Errors

The new quota type will use `QuotaViolationException` similar to `ClientQuotaManager. And the client will receive` ProduceResponse with throttleTimeMs similar to throttling of bandwidth or request.

New TimeControlledBloomFilter

```
class TimeControlledBloomFilter[T](numberOfItems: Int, falsePositiveRate: Double, disposalSchedulerIntervalMs: 
Long, quotaWindowSizeSeconds: Long, scheduler: Scheduler) {
  val bloomFilters: ConcurrentHashMap[Long, SimpleBloomFilter[T]] = new ConcurrentHashMap() // This keep a map 
of create time to bloom filter

 def create(): Unit = {
         // Will create new SimpleBloomFilter with numberOfBits and numberOfHashes driven from falsePositiveRate
 }
 def put(value: T): Unit = {
        // Will choose the right bloom filter to use
        }
  def mightContain(value: T): Boolean = {
        // Will check all available bloom filters
        }

  scheduler.schedule("dispose-old_bloom-filter", ()=> {
                // dispose the bloom filter that older the 1.5 x quotaWindowSizeSeconds.
        }, 0L, disposalSchedulerIntervalMs)
}

class SimpleBloomFilter[T](numberOfBits: Int, numberOfHashes: Int) {
 val bits = mutable.BitSet.empty

 def put(value: T): Unit {
        // Will use MurmurHash3 to has the value
        }
 def mightContain(value: T): Boolean {
        // will check if any of the available bloom filters contains the value
        }
}
```

New `ProducerIdQuotaManagerCache`

```
class ProducerIdQuotaManager[K, V](disposalSchedulerIntervalMs: Long, cleanupScheduler: Scheduler) {
        protected val concurrentMap: ConcurrentHashMap[KafkaPrincipal, TimedBloomFilter[V]] = new 
ConcurrentHashMap()

        protect val schedulerIntervalMs = new AtomicLong(disposalSchedulerIntervalMs)

    cleanupScheduler.schedule("cleanup-keys", () => {
                        // Cleanup Keys that have empty TimedBloomFilter
    }, 0L, schedulerIntervalMs)


         def disposalSchedulerIntervalMs(intervalMs: Long): Unit = {
                disposalSchedulerIntervalMs(intervalMs)
          }

        def add(key: K, value: V): ControlledCachedMap[K, V] = {
                // Add value to the key bloom filter
        }

    def containsKeyValuePair(key: K, value: V): Boolean = {
                // Check if key, value exist in the cache
        }
}
```

Tools

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

- `producer_ids_rate`: The number of active PIDs per quota window.

For example:

```
bin/kafka-configs --bootstrap-server localhost:9092 --alter --add-config 'producer_ids_rate=50' --entity-name
user1 --entity-type users
```

Default quotas for <*user*> can be configured by omitting entity name. For example:

```
bin/kafka-configs --bootstrap-server localhost:9092 --alter --add-config 'producer_ids_rate=200' --entity-type
users
```

Known Limitations

- As we are using BloomFilter we might get false positives.
- Throttling based on User will punish any client is used by the same user. However, this is similar risk like existing quotas that got applied only on KafkaPrincipal level.
- Similar to other Quotas in Kafka, all throttling is against individual brokers. Which means if leadership changed the new leader will start throttle from zero if it never had this KafkaPrincipal producing to it before.
- Some producers might get throttled for long time depends on the configuration instead of crashing. Which may go unnoticed for some producers specially if they don't alert on the throttle or other metrics to get notified when the producer stopped producing.

Compatibility, Deprecation, and Migration Plan

Compatibility with Old Clients

- None, since we are using the same throttling from ClientQuota which the client knows how to handle.

Rejected Alternatives

1. Limit the total active producer ID allocation number

This solution is the simplest however as stated in the motivation the OOM is always caused by rough or misconfigured client this solution will punish good client along side the rough one.

2. Having a limit to the number of active producer IDs

The idea here is if we had a misconfigured client, we will expire the older entries This solution will risk the idempotency guarantees. Also there is risk that we may end up expiring the PIDs for good clients as the there is no way to link back PID to specific client at this point.

3. Allow clients to "close" the producer ID usage

Part of the root cause of the OOM problem is that we keep PIDs metadata in the broker even if the producer is "closed". This solution would provide a closed API (for example END_PRODUCER_ID) and the broker will remove the PID metadata from its side. In the client side, we can send it when the producer closing. This solution is better however

- it only improves the situation with new clients leaving the broker exposed to OOM because of old producers.
- It doesn't address producers that enter repeated restart cycle as these producer will be crashing and will not call `producer.close` method.

We may need to consider improving the Producer Client anyway to include this at some point but it is not as part of the scope of this KIP.

4. Throttle INIT_PRODUCER_ID requests

This solution might look simple however throttling the INIT_PRODUCER_ID doesn't guarantee the OOM wouldn't happened as

a. INIT_PRODUCER_ID for idempotent producer request PIDs from random controller every time so if a client got throttled on one controller doesn't guarantee it will not go through on next controller causing OOM at the leader later.
b. The problem happened on the activation of the PID when it produce and not at the initialisation. Which means Kafka wouldn't have OOM problem if the producer got assigned PID but crashed before producing anything.
c. Throttling producers that crash between initialisation and producing could slow them down when they recover/fix the problem that caused them to crash right after initialising PID.

5. Throttle PIDs based on IPs

Similar solution#1 we will end up punishing good users specially if the misbehaving producer is deployed on K8S cluster that has other usecase.

6. Use HashSet to track PIDs in the caching layer instead of BloomFilter

HashSet provide 100% correctness however the growth of the caching layer with HashSet will create a risk of OOM. While it is not as bad as the original OOM as the broker wouldn't rebuild this cache on the start time none the less. To control the memory of cache using HashSet will be bit tricky and will need more configuration to keep it under control.
On the other hand BloomFilter is more efficient when it come to memory cost while providing a reasonable correctness that will be good enough for this usecase. And if we want to improve the correctness we can always improve the false positive rates in the bloom filter.