# Broker side quota (V1)

### Design goal:

The goal of this design is to get us started with the problem. It's not meant to be comprehensive and is intended to provide a simple solution to a common problem introduced by the producers.

### Problem:

When the produced data exceeds a certain threshold, I/O performance in the brokers starts to degrade . The means that the latency of all requests will increase. This will impact real time applications that are sensitive to the latency.

### High level design:

One way to address this problem is to set a global bytesIn threshold per broker. If that threshold is exceeded, the broker will start dropping produced data until the observed bytesIn rate falls below the expected threshold. The question remains what data to drop when the threshold is exceeded. In a shared Kafka cluster, it's likely that some topics are more important than others. So, it makes sense to have a per topic bytesIn threshold. When the global bytesIn threshold is exceeded, the broker will start dropped produced data on topics whose observed bytesIn rate exceeds the expected per topic threshold.

### Low level design:

1. Introduce a new broker level config total.bytes.in.per.sec.threshold. Each topic can define a topic level config bytes.in.per.sec.threshold. A change in bytes.in.per.sec.threshold will be rejected if the sum of all specified bytes.in.threshold exceeds total.bytes.in.per.sec.threshold. For topics without an explicitly defined bytes.in.per.sec.threshold, its threshold is computed as: (total.bytes.in.per.sec.threshold - sum(explicitly specified bytes.in.per.sec. threshold)) / (# topics without an explicitly defined bytes.in.per.sec.threshold). Such a value is recomputed every time if bytes.in.per.sec.threshold for a topic is changed or a new topic is added to the broker. This value can probably be maintained inside LogManager.

2. We are already monitoring the global and the topic level bytesIn rate using Meter. Meter exposes a OneMinuteRate, which is an EWMA and is refreshed every 5 seconds by default. This should be good enough for measuring the observed bytesIn rate.

3. Introduce TopicQuota.

```
Object TopicQuota
{
    public bool recordQuota(Meter observedTotalRate, Long expectedTotalRate, Meter observedTopicRate, Long
expectedTopicRate, Long incomingDataSize)
    {
      if (observedTotalRate.getOneMinuteRate() > expectedTotalRate && observedTopicRate.getOneMinuteRate() >
expectedTopicRate)
        return false

      observedTotalRate.record(incomingDataSize)
      observedTopicRate.record(incomingDataSize)
      return true
    }
}
```

4. In Log.append(), we do

```
    if (Topic.Quota.recordQuota(....))
      append to log
    else
      throw QuotaExceededException
```

### Discussions:

1. Currently, messages rejected due to MessageSizeTooLargeException is included in the bytesIn rate. We probably should use bytesIn rate to measure the amount of data actually getting into the log. We can introduce two other meters to measured the error bytesIn rate (e.g. due to MessageSizeTooLargeException) and throttled bytesIn rate (due to quota). We probably don't need topic level rate for the latter two. Instead, we just log in log4j the dropped requests and the associated reason.