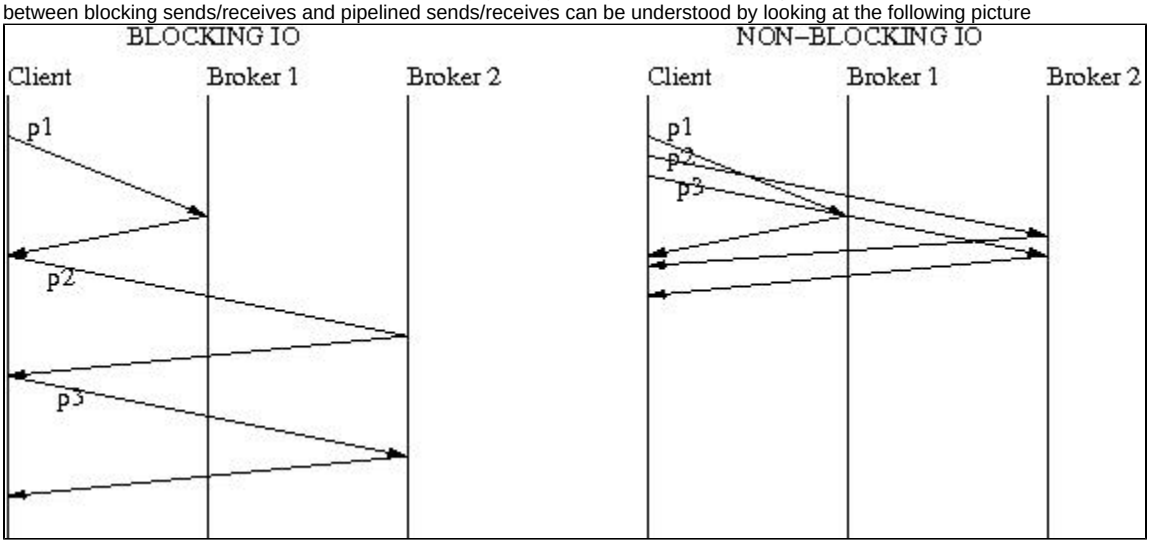
Producer throughput improvement

What is this project about?

In 0.8, the producer throughput is limited due to the synchronous interaction of the producer with the broker. At a time, there can only be one request in flight per broker and that hurts the throughput of the producer severely. The problem with this approach also is that it doesn't take advantage of parallelism on the broker where requests for different partitions can be handled by different request handler threads on the broker, thereby increasing the throughput seen by the producer.

Key idea behind the solution

Request pipelining allows multiple requests to the same broker to be in flight. The idea is to do I/O multiplexing using select(), epoll(), wait() and wait for at least one readable or writable socket instead of doing blocking reads and writes. So if a producer sends data to 3 partitions on 2 brokers, the difference between blocking sends/receives and pipelined sends/receives can be understood by looking at the following picture



Design requirements

1. Ordering should be maintained per partition, if there are no retries.
2. Existing producer API should remain completely asynchronous.
3. No APIs changes are necessary to roll this out.

Proposed design

There are 2 threads, depending on how metadata requests are handled.

1. Client thread - partitions
2. Send thread - refreshes metadata, serializes, collates, batches, sends data and receives acknowledgments

Event Queuing

There are 2 choices here -

One queue per partition

There will be one queue with partition "-1" per topic. The events with null key will enter this queue. There will be on "undesignated" partition queue per topic. This is done to prevent fetching metadata on the client thread when the producer sends the first message for a new topic.

Pros:

1. More isolation within a topic. Keys that are high traffic will not affect keys that are lower traffic within a topic. This will protect important low throughput topics like audit data.
2. Easier to handle the most common error which is leader transition. It is easier with one queue per partition since you can just ignore the queues for partitions that don't have leaders yet.

Cons:

1. More queues means more memory overhead, especially for tools like MirrorMaker.
2. Handling time based event expiration per partition queue complicates the code to some extent.

One queue for all topics and partitions.

Pros:
Less memory overhead since there is only one queue data structure for all events

Cons:

1. Less isolation. One high throughput topic or partition can cause data for other topics to be dropped. This especially hurts audit data and can be easily avoided by having multiple queues.
2. Complicates batching since you will have to find a way to skip over partitions that don't have leaders and avoid dequeueing their data until a leader can be discovered.

Partitioning

Partitioning can happen before the event enters the queue. The advantage is that the event does not require re-queuing if one queue per partition approach is used. Events with null key will be queued to the <topic>-"-1" queue.

Metadata discovery

When a producer sends the first message for a new topic, it enters the <topic>-undesignated queue. The metadata fetch happens on the event thread. There are 2 choices on how the metadata fetch request will work -

The metadata fetch is a synchronous request in the event loop.

Pros:
Simplicity of code

Cons:
If leaders only for a subset of partitions have changed, a synchronous metadata request can potentially hurt the throughput for other topics/partitions.

Metadata fetch is non blocking

Pros:
More isolation, better overall throughput

Serialization

One option is doing this before the event enters the queue and after partitioning. Downside is potentially slowing down the client thread if compression or serialization takes long. Another option is to just do this in the send thread, which seems like a better choice.

Batching and Collation

Producer maintains a map of broker to list of partitions that the broker leads. Batch size is per partition. For each broker, if the key is in write state, the producer's send thread will poll the queues for the partitions that the broker leads. Once the batch is full, it will create a ProducerRequest with the partitions and data, compress the data, and writes the request on the socket. This happens in the event thread while handing new requests. The collation logic gets a little complicated if there is only one queue for all topics/partitions.

Compression

When the producer send thread polls each partition's queue, it compresses the batch of messages that it dequeues.

Event loop

```
while(isRunning)
{

// configure any new broker connections

// select readable keys

// handle responses

// select writable keys

// handle topic metadata requests, if there are non-zero partitions in error

// handle incomplete requests

// handle retries, if any

// handle new requests

}
```