STRIMZI

Optimizing Kafka producers

October 15, 2020 by Paul Mellor

You can fine-tune Kafka producers using configuration properties to optimize the streaming of data to consumers. Get the tuning right, and even a small adjustment to your producer configuration can make a significant improvement to the way your producers operate.

In this post we'll discuss typical tuning considerations for Kafka producers.

Optimizing Kafka producers

Obviously, we want our producers to deliver data to Kafka topics as efficiently as possible. But what do we mean by this, and how do we quantify it? Do we base this on the number of messages sent over a set period of time? Or on how producers are set up to handle failure?

Before starting your adventure in optimization, think about your destination. What are the results your are hoping to achieve? Think long enough about this, and you might find competing requirements. For example, by maximizing throughput you might also increase latency.

Be prepared to make adjustments to your adjustments.

How is your producer performing?

It's only when you have been monitoring the performance of your producers for some time that you can gauge how best to tune their performance.

To begin with, you might start with a basic producer configuration in development as a benchmark. When you start to analyze producer metrics to see how the producers actually perform in typical production scenarios, you can make incremental changes and make comparisons until you hit the sweet spot.

> “*If you want to read more about performance metrics for monitoring Kafka producers, see Kafka's Producer Sender Metrics.*

When you start investigating how you to tune the performance of your producers, look at how your producers perform on average.

For example, broker restarts will have an outsize impact on very high (99%) percentile latencies impact on very high (99%) percentile latencies. So you might concentrate on tuning your producer to achieve a latency target within a narrower bound under more typical conditions.

Basic producer configuration

Before looking at the properties to use for fine-tuning your producer, let's assume we have a basic configuration.

Something like this.

```
bootstrap.servers=localhost:9092
key.serializer=org.apache.kafka.common.
value.serializer=org.apache.kafka.commo
client.id=my-client
compression.type=gzip
```

This configuration specifies the bootstrap address for connection to the Kafka cluster, and the serializers that transform the key and value of a message from from a String to its corresponding raw byte data representation.

Optionally, it's good practice to add a unique client ID, which is used to identify the source of requests in logs and metrics.

Compression is useful for improving throughput and reducing the load on storage, but might not be suitable for low latency applications where the cost of compression or decompression could be prohibitive. More on compression later.

In our basic configuration, acknowledgments only confirm that messages have reached the broker, and there is no guarantee on message ordering.

We can change that…

Key properties

Let’s look at how you can add or change producer configuration properties for fine-tuning.

Here are the properties we’ll consider:

- acks
- min.insync.replicas (topics)
- enable.idempotence
- max.in.flight.requests.per.connection
- retries
- transactional.id
- transaction.timeout.ms

We won’t discuss the properties in isolation here, as we’ll be concentrating on how to use them to achieve a certain result.

We’ll look at how you can use a combination of these properties to regulate:

- *Data durability* to minimize data loss when passing messages
- *Ordered delivery* of messages
- *Reliability guarantees* on message transactions involving batches of related messages
- *Throughput* measured in the number of messages processed over a specific period
- *Latency* measured in the time it takes for messages reach the broker

It’s quite likely you’ll want to balance throughput and latency targets whilst also minimizing data loss and guaranteeing ordering.

> *“If you want to read more about what each property does, see Kafka’s Producer configs.*

Remember, the producer configuration properties you *can* use will also be driven by the requirements of your application. Avoid any

change that breaks a property or guarantee provided by your application.

Data durability

It you want to reduce the likelihood that messages are lost, use message delivery acknowledgments.

Specify `acks=all` in your producer configuration to force a partition leader to replicate messages to a certain number of followers before acknowledging that the message request was successfully received.

```
# ...
acks=all
# ...
```

Use the `acks=all` producer configuration in conjunction with the `min.insync.replicas` property for topics. You set the `min.insync.replicas` property in the

`KafkaTopic` resource.

The `min.insync.replicas` configuration sets the numbers of brokers that need to have logged a message before an acknowledgment is sent to the producer.

```
apiVersion: kafka.strimzi.io/v1alpha1
kind: KafkaTopic
metadata:
  name: my-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 1
  replicas: 3
  config:
    min.insync.replicas: 2
    #...
```

Using a topic replication factor of 3, and 2 in-sync replicas on other brokers, the producer can continue unaffected if a single broker is unavailable and at least one other broker is in-sync.

If a second broker becomes unavailable, using `acks=all` the producer won't receive acknowledgments and won't be able to produce more messages.

Because of the additional checks, `acks=all` increases the latency between the producer sending a message and receiving acknowledgment. So you will have to consider the trade-off when investigating whether this is the right approach for you.

Ordered delivery

You have two approaches to guaranteeing the order of message delivery from producers. And they depend, to a large degree, on whether or not you are using `acks=all` for data durability.

If you are using `acks=all`, you can (and should) enable idempotence for the producer to ensure that messages are delivered only once. Using idempotence, IDs and sequence numbers are assigned to messages, so that the order is preserved even after a failed delivery.

As idempotence preserves the message ordering, you can speed the process along by increasing the number of in-flight requests you allow at one time using the `max.in.flight.requests.per.connection`.

Here we see an example configuration showing idempotence enabled, and used with `max.in.flight.requests.per.connection` and `acks=all`.

```
# ...
enable.idempotence=true
max.in.flight.requests.per.connection=5
acks=all
retries=2147483647
```

The `retries` property sets the number of retries when resending a failed message request. While that number might look impressive, what we're saying is, in effect, "retry forever".

> **Delivery timeout**
>
> *If you are using* `delivery.timeout.ms` *in your producer configuration, producer requests will fail before the number of retries has been used if the timeout expires before a successful acknowledgment. The* `delivery.timeout.ms` *sets a limit on the time to wait for an acknowledgment of the success or failure to deliver a message. You can choose to leave* `retries` *unset and use*
>
> `delivery.timeout.ms` *to perform a similar function instead.*

There is a performance cost to introducing additional checks to the order of delivery.

If you prefer not to use `acks=all` and idempotency, another option is to set the number of in-flight requests to 1 (the default is 5) to preserve ordering.

```
# ...
enable.idempotence=false
max.in.flight.requests.per.connection=1
retries=2147483647
```

In this way you avoid a situation where *Message-A* fails only to succeed after *Message-B* was already written to the broker.

Reliability guarantees

Idempotence on its own is useful for exactly once writes to a single partition.

But how do we guarantee the reliability of message delivery for exactly once writes for a set of messages across multiple partitions? We use idempotence again, but combine it with a unique transactional ID defined for the producer. Transactions guarantee that messages using the same transactional ID are produced once, and either *all* are successfully written to the respective logs or *none* of them are.

You specify a unique transactional ID in the producer configuration, and also set the maximum allowed time for transactions in milliseconds before a timeout error is returned. The default is `900000` or 15 minutes.

```
# ...
enable.idempotence=true
max.in.flight.requests.per.connection=5
acks=all
retries=2147483647
transactional.id=_UNIQUE-ID_
transaction.timeout.ms=900000
# ...
```

The transactional ID is registered with the Kafka cluster on the first operation, as well as a producer `epoch` checkpoint number, which is used to identify the active producer instance.

Why might we want to identify the active producer? Say an application determines that a producer has failed and creates a new producer instance to restart a transaction. If both producers are now sending messages, duplicate records are being created and we have lost our exactly once integrity.

By specifying a transaction ID, if a new producer instance starts, older instances of the producer are identified by their older `epoch` number and *fenced-off* by Kafka so that their messages are not included. This maintains the integrity of the message passing by ensuring that there is only ever one valid producer with the transactional ID. Each `transactional.id` should be used for a unique set of topic partitions.

You can map topic partition names to transactional IDs, or compute the transactional ID from the topic partition names using a function that avoids collisions.

Optimizing throughput and latency

Depending on your objective, Kafka offers a number of configuration parameters and techniques for tuning producer performance for throughput and latency.

Usually, the requirement of a system is to satisfy a particular throughput target for a proportion of messages within a given latency. For example, targeting 50,000 messages per second with 95% of messages being acknowledged within 2 seconds.

Batching and buffering messages

Message batching delays sending messages so that more messages destined for the same broker are batched into a single request. Without message batching, messages are sent without delay.

You can use two properties to set batch thresholds:

- `batch.size` specifies a maximum batch size in bytes (default 16384)
- `linger.ms` specifies a maximum duration to fill the batch in milliseconds (default 0 or no delay)

Messages are delayed until either of these thresholds are reached.

For example, if we use `linger.ms` to add a a 500ms delay, all the messages accumulated in that time are sent in a single request.

```
linger.ms=500
batch.size=16384
buffer.memory=33554432
```

If a maximum `batch.size` is also used, a request is sent when messages are accumulated up the maximum batch size, or messages have been queued for longer than `linger.ms` — whichever comes sooner.

Use the `buffer.memory` to configure a buffer memory size that must be at least as big as the batch size, and also capable of accommodating buffering, compression and in-flight requests.

Size is important. If the batch threshold is too big for the frequency of the messages produced, you're adding unnecessary delay to the messages you're adding unnecessary delay to the messages waiting in the send buffer. You're also allocating more buffer memory than you need. If the batch threshold is too small, larger messages can be delayed.

What you gain in higher throughput you concede with the buffering that adds higher latency to the message delivery. It's a compromise, so you will need to consider how to strike the right balance.

> ***send() blocking and latency***
>
> *Batching and buffering also mitigates the impact of `send()` blocking on latency.*
>
> *When your application calls `KafkaProducer.send()`, the messages produced are:*
>
> - *Processed by any interceptors*
> - *Serialized*
> - *Assigned to a partition*
> - *Compressed*
> - *Added to a batch of messages in a per-partition queue*
>
> *At which point the `send()` method returns. So the time `send()` is blocked is determined by:*
>
> - *The time spent in the interceptors, serializers and partitioner*
> - *The compression algorithm used*
> - *The time spent waiting for a buffer to*

> *use for compression*
>
> *Batches will remain in the queue until one of the following occurs:*
>
> - *The batch is full (according to* `batch.size` *)*
> - *The delay introduced by* `linger.ms` *has passed*
> - *The sender is about to send message batches for other partitions to the same broker, and it is possible to add this batch too*
> - *The producer is being flushed or closed*

Compressing message batches

Compressing data batches using the `compression.type` property improves throughput and reduces the load on storage, but might not be suitable for low-latency applications where the cost of compression or decompression is prohibitive.

Message compression adds latency in the producer (CPU time spent compressing the messages), but makes requests (and potentially disk writes) smaller, which can increase throughput.

Use the `compression.type` property to specify a valid compression codec. You can choose `gzip` , `snappy` , `lz4` , or `zstd` , each of which have varying compression speeds.

```
compression.type=gzip
batch.size=32000
```

If you think compression is worthwhile, the best type of compression to use will depend on the messages being sent.

> ***Adding threads***
>
> *Compression is handled on the thread calling `KafkaProducer.send()`, so if the latency of this method matters for*
>
> *your application you can add more more threads.*

Pipelining messages

Pipelining might sound like it has something to do with surfing the famous *Pipeline* reef in Hawaii, but it's actually the sending of more requests from producers before the response to a previous request has been received.

For pipelining we use our old friend the `max.in.flight.requests.per.connection` property. You might recall its contribution to ordered delivery earlier in this post.

Moving message requests along more frequently will obviously improve throughput. But there is a point at which you might see less beneficial effects, such as less efficient batching.

Adjusting message delivery waiting time

Improve throughput of your message requests by adjusting the maximum time to wait before a message is delivered and completes a send request.

Use the `delivery.timeout.ms` property to specify the maximum time in milliseconds to wait for a complete send request. You can set the value to `MAX_LONG` to delegate to Kafka an indefinite number of retries.

You can also direct messages to a specified partition by writing a custom partitioner to replace Kafka's default, and specify the class name using the `partitioner.class` property. A custom partitioner allows you to choose how you map messages to partitions, based on the data in the message.

```
# ...
delivery.timeout.ms=120000
partitioner.class=my-custom-partitioner
# ...
```

Adapt to survive

Fine-tuning your producers helps alleviate performance issues. But don't be tempted to make a few adjustments and think your work is done.

You should consider fine-tuning as part of a continual optimization process. Monitor Kafka regularly. Look for changing trends in usage and investigate how fine-tuning can help your Kafka deployment adapt. But, as you'll know, this is only one half of the story.

Next time we'll look at how you can optimize your consumers.

Be sure to check back for the next installment.

Share this:

Strimzi provides a way to run an Apache Kafka cluster on Kubernetes in various deployment configurations.

Strimzi
strimziio
Strimzi