The Design and Implementation of Kafka

Wang Sheying

HuiLongGuan of Beijing

April 17, 2020

Outline

Outline

Kafka 简介

Outline

Design

Persistence

Kafka relies heavily on the filesystem for storing and caching messages.

As a result the performance of linear writes on a JBOD configuration with six 7200rpm SATA RAID-5 array is about 600MB/sec but the performance of random writes is only about 100k/sec—a difference of over 6000X.

To compensate for this performance divergence, modern operating systems have become increasingly aggressive in their use of main memory for disk caching.

Design

Persistence

A modern OS will happily divert all free memory to disk caching with little performance penalty when the memory is reclaimed.

All disk reads and writes will go through this unified cache.

A modern operating system provides read-ahead and write-behind techniques that prefetch data in large block multiples and group smaller logical writes into large physical writes.

Design

Persistence

Furthermore, kafka are building on top of the JVM, and anyone who has spent any time with Java memory usage knows two things:

- The memory overhead of objects is very high, often doubling the size of the data stored (or worse).
- Java garbage collection becomes increasingly fiddly and slow as the in-heap data increases.

Design

Persistence

As a result of these factors using the filesystem and relying on pagecache is superior to maintaining an in-memory cache or other structure

Doing so will result in a cache of up to 28-30GB on a 32GB machine without GC penalties.

Furthermore, this cache will stay warm even if the service is restarted, whereas the in-process cache will need to be rebuilt in memory

Design

Persistence

This suggests a design which is very simple: rather than maintain as much as possible in-memory and flush it all out to the filesystem in a panic when we run out of space, we invert that.

All data is immediately written to a persistent log on the filesystem without necessarily flushing to disk.

In effect this just means that it is transferred into the kernel's pagecache.

Design

Persistence

The persistent data structure used in messaging systems are often a per-consumer queue with an associated BTree or other general-purpose random access data structures to maintain metadata about messages.

They do come with a fairly high cost, though: Btree operations are O(log N). Normally O(log N) is considered essentially equivalent to constant time, but this is not true for disk operations.

Since storage systems mix very fast cached operations with very slow physical disk operations, the observed performance of tree structures is often superlinear as data increases with fixed cache.

Design

Persistence

Intuitively a persistent queue could be built on simple reads and appends to files as is commonly the case with logging solutions.

This structure has the advantage that all operations are O(1) and reads do not block writes or each other.

This has obvious performance advantages since the performance is completely decoupled from the data size.

Design

Efficiency

We assume each message published is read by at least one consumer (often many), hence we strive to make consumption as cheap as possible.

Once poor disk access patterns have been eliminated, there are two common causes of inefficiency in this type of system:

- too many small I/O operations
- excessive byte copying

Design

Efficiency

The small I/O problem happens both between the client and the server and in the server's own persistent operations.

To avoid this, our protocol is built around a "message set" abstraction that naturally groups messages together.

The server in turn appends chunks of messages to its log in one go, and the consumer fetches large linear chunks at a time.

Design

Efficiency

This simple optimization produces orders of magnitude speed up.

Batching leads to larger network packets, larger sequential disk operations, contiguous memory blocks, and so on.

All of which allows Kafka to turn a bursty stream of random message writes into linear writes that flow to the consumers.

Design

Efficiency

The other inefficiency is in byte copying.

To avoid this we employ a standardized binary message format that is shared by the producer, the broker, and the consumer,so data chunks can be transferred without modification between them.

The message log maintained by the broker is itself just a directory of files, each populated by a sequence of message sets that have been written to disk in the same format used by the producer and consumer.

Design

Efficiency

Maintaining this common format allows optimization of the most important operation: network transfer of persistent log chunks.

Modern unix operating systems offer a highly optimized code path for transferring data out of pagecache to a socket; in Linux this is done with the sendfile system call.

The Java class libraries support zero copy on Linux and UNIX systems through java.nio.channels.FileChannel.transferTo().

Design

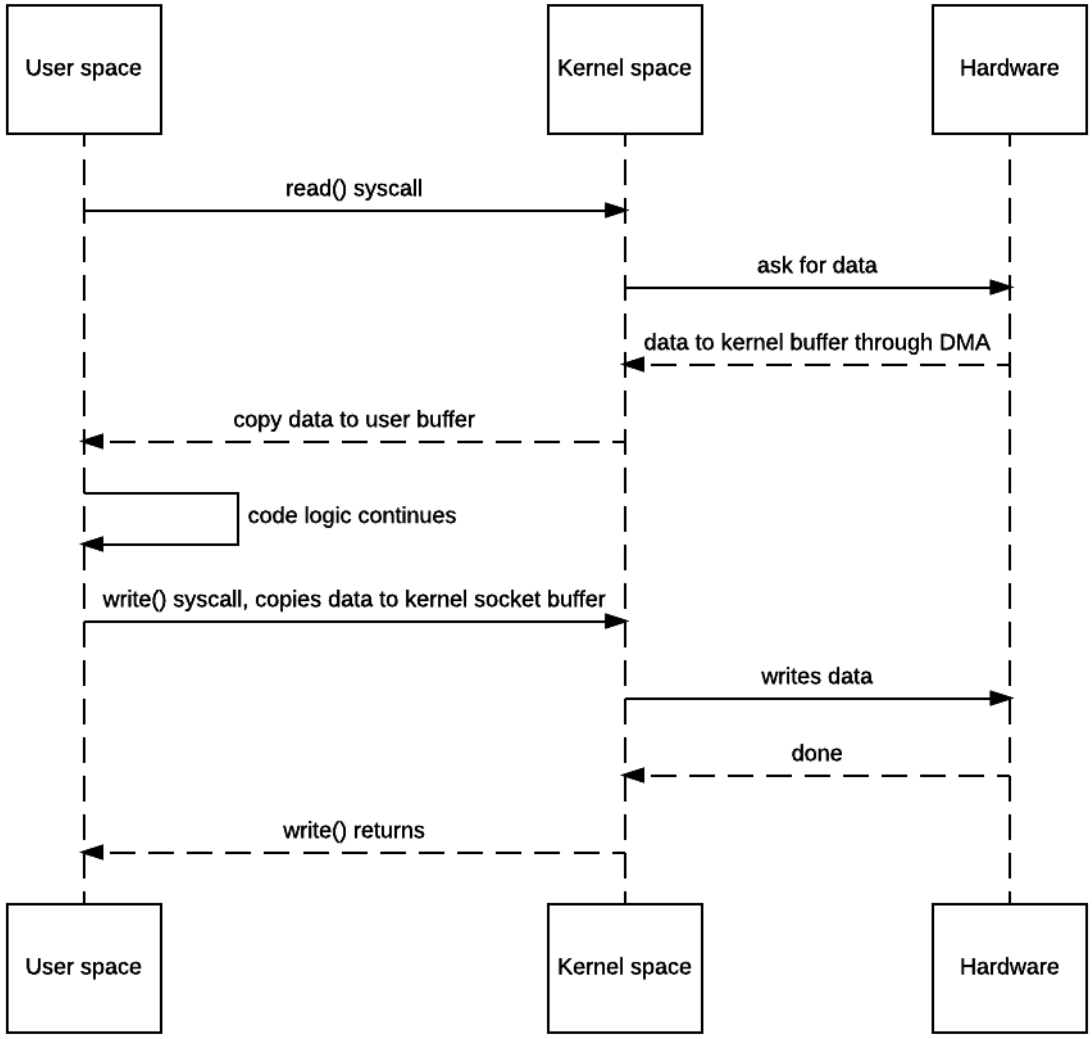
Efficiency

To understand the impact of sendfile, it is important to understand the common data path for transfer of data from file to socket:

1. The operating system reads data from the disk into pagecache in kernel space
2. The application reads the data from kernel space into a user-space buffer
3. The application writes the data back into kernel space into a socket buffer
4. The operating system copies the data from the socket buffer to the NIC buffer where it is sent over the network

Design

Efficiency



Design

Efficiency

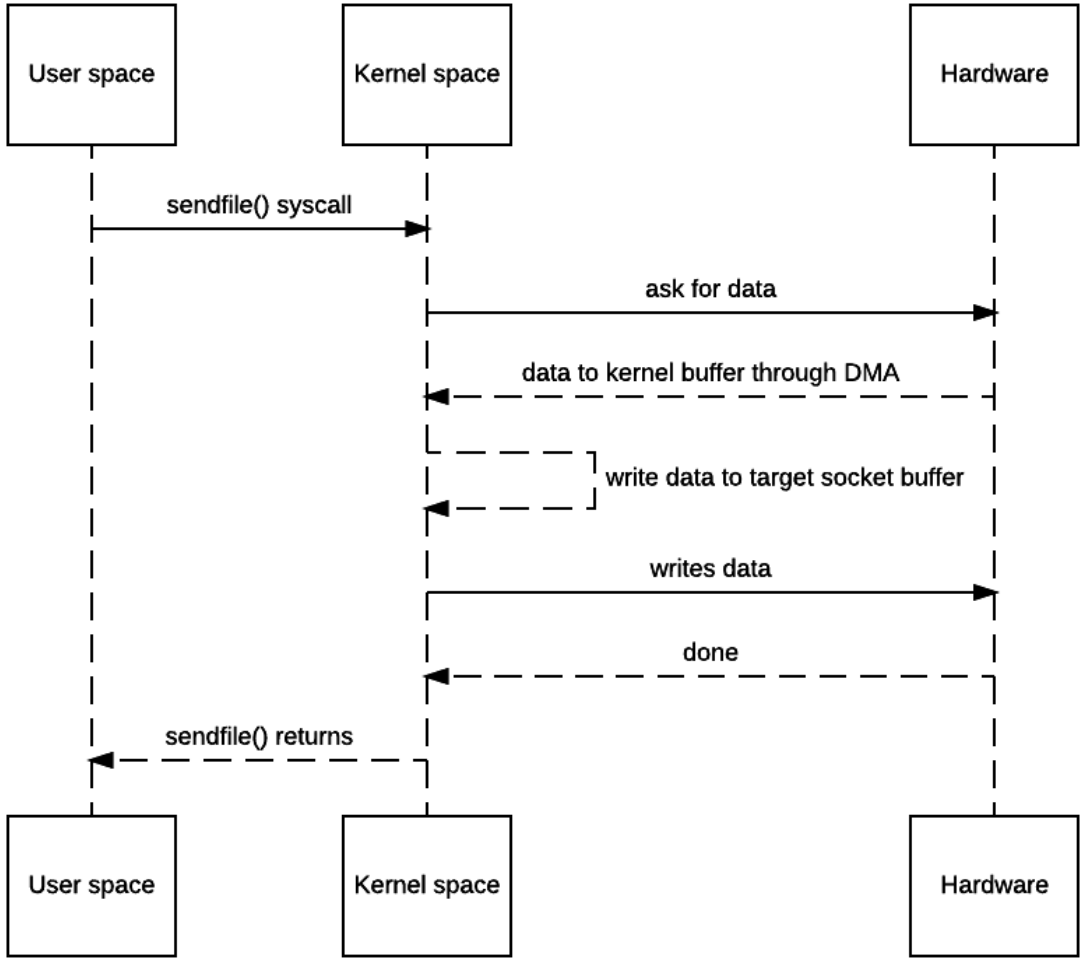
This is clearly inefficient, there are four copies and two system calls.

Using sendfile, this re-copying is avoided by allowing the OS to send the data from pagecache to the network directly.

So in this optimized path, only the final copy to the NIC buffer is needed.

Design

Efficiency



Design

Efficiency

Using the zero-copy optimization above, data is copied into pagecache exactly once and reused on each consumption.

This allows messages to be consumed at a rate that approaches the limit of the network connection.

This combination of pagecache and sendfile means that on a Kafka cluster you will see no read activity on the disks whatsoever as they will be serving data entirely from cache.

Design

Efficiency

In some cases the bottleneck is actually not CPU or disk but network bandwidth.

Efficient compression requires compressing multiple messages together rather than compressing each message individually.

Kafka supports this with an efficient batching format.

Design

Efficiency

A batch of messages can be clumped together compressed and sent to the server in this form.

This batch of messages will be written in compressed form and will remain compressed in the log and will only be decompressed by the consumer.

Kafka supports GZIP, Snappy, LZ4 and ZStandard compression protocols.

Design

Efficiency

Design

The Producer

The producer sends data directly to the broker that is the leader for the partition without any intervening routing tier.

To help the producer do this all Kafka nodes can answer a request for metadata at any given time to allow the producer to appropriately direct its requests.

The client controls which partition it publishes messages to.

Design

The Producer

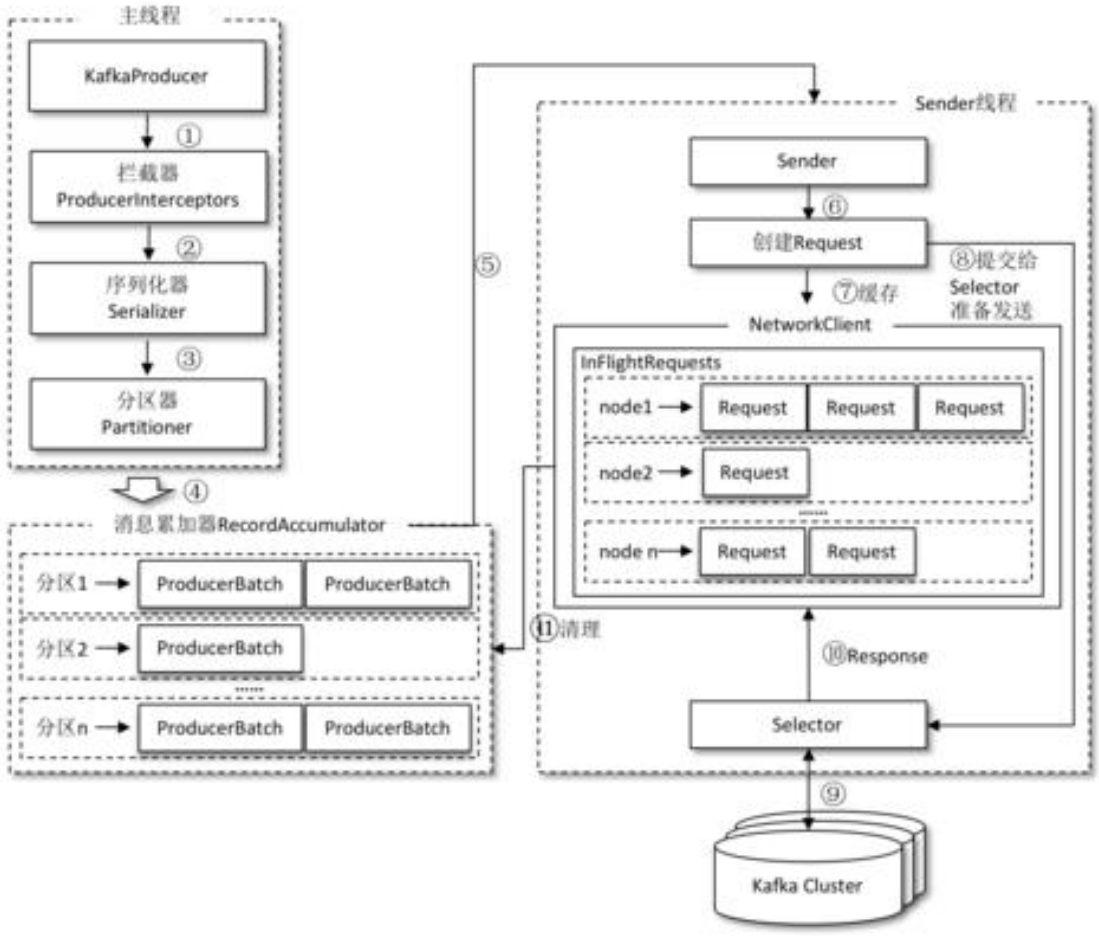
Batching is one of the big drivers of efficiency, and to enable batching the Kafka producer will attempt to accumulate data in memory and to send out larger batches in a single request.

The batching can be configured to accumulate no more than a fixed number of messages and to wait no longer than some fixed latency bound (say 64k or 10 ms).

This buffering is configurable and gives a mechanism to trade off a small amount of additional latency for better throughput.

Design

The Producer



Design

The Producer

ProducerRecord(Main Thread)

⇓

ProducerBatch (Sender Thread)

⇓

<Partition, Deque<ProducerBatch>>(Sender Thread)

⇓

<Node, List<ProducerBatch>>(Sender Thread)

⇓

<Node, Request>(Sender Thread)

⇓

Broker(page cache)

⇓

Broker(log segment)

Design

The Producer

Kafka provides both an asynchronous send method to send a record to a topic.

`Future<RecordMetadata>`	`send(ProducerRecord<K,V> record)` Asynchronously send a record to a topic.
`Future<RecordMetadata>`	`send(ProducerRecord<K,V> record, Callback callback)` Asynchronously send a record to a topic and invoke the provided callback

- synchronous
  - block
  - Future<RecordMetadata>.get()
- asynchronous
  - unblock
  - the callback interface

Design

The Producer

Design

The Producer

Design

The Producer

Outline

Implementation

Questions and Answers?

Questions and Answers?

Thank You!

References I