The Design and Implementation of Kafka

Wang Sheying

HuiLongGuan of Beijing

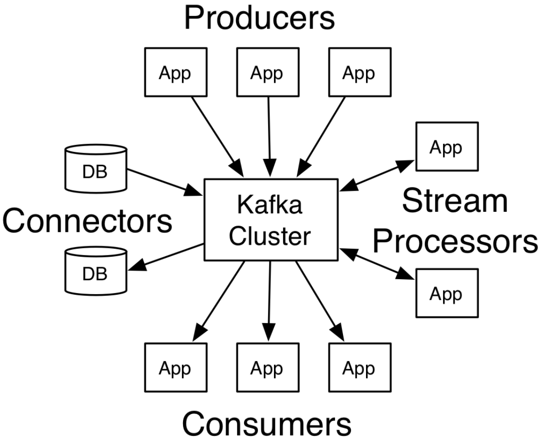
April 18, 2020

Outline

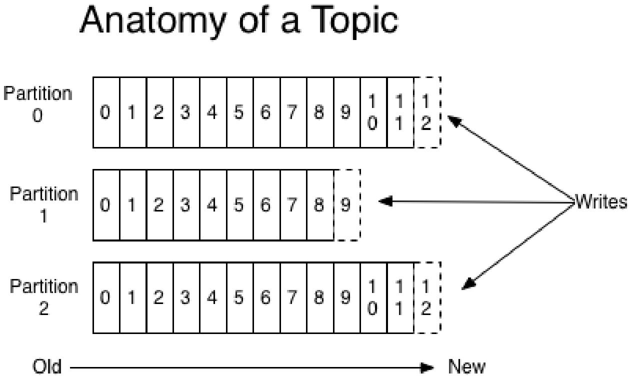
Outline

Introduction

Apache Kafka is used for building real-time data pipelines and streaming apps. It is horizontally scalable, fault-tolerant, wicked fast, and runs in production in thousands of companies.

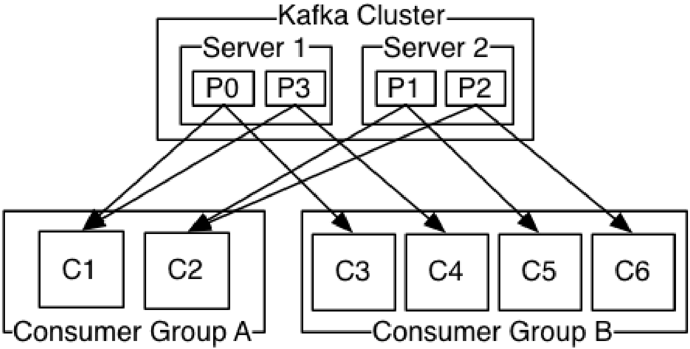


Introduction

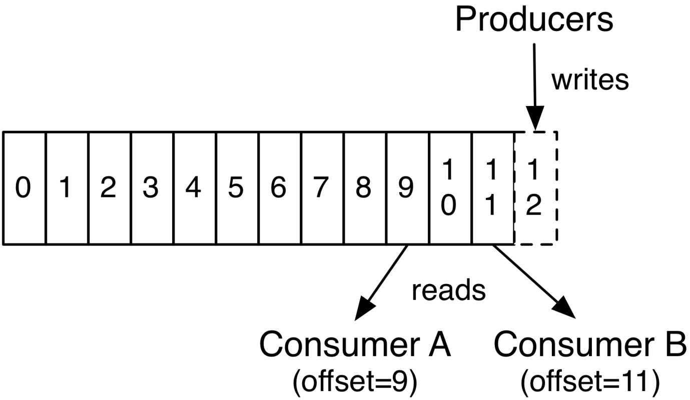


Introduction

Introduction



Introduction

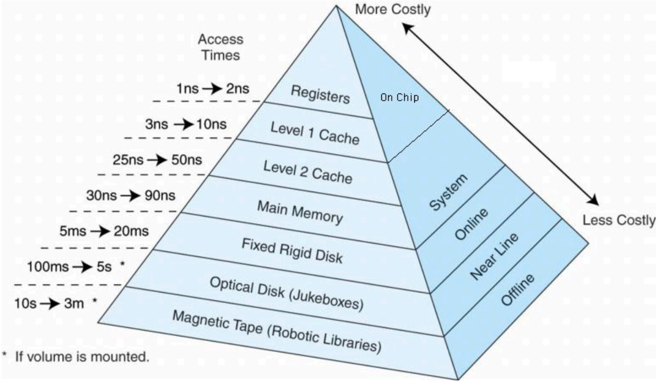


Outline

Design

Persistence

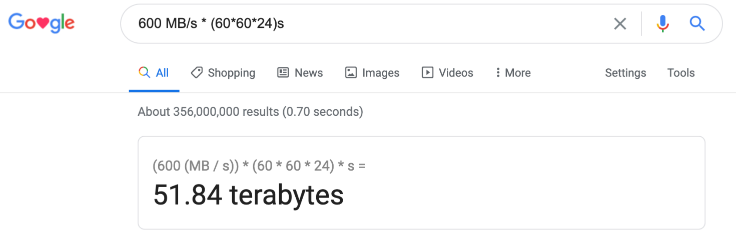
Kafka relies heavily on the filesystem for storing and caching messages.



Design

Efficiency

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.



Design

Efficiency

The killer when working with a disk? Access time. Today, a page fault stalls the CPU for millions of cycles; as clock speeds increase, this penalty will increase to tens of millions of cycles.

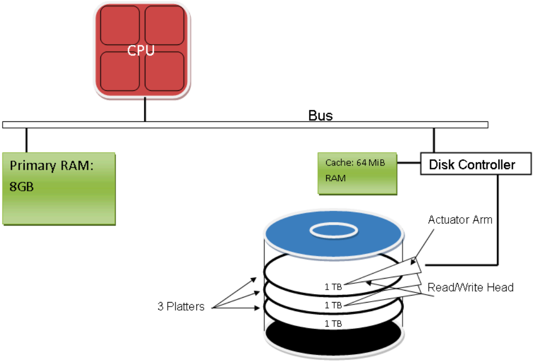
This is why caching data in memory(page cache) is so important for performance –disk is 5 orders of magnitude slower than RAM.

The modern OS predicts future accesses based on past access patterns and then performs a cost-benefit analysis to determine whether the benefit of prefetching a block exceeds its cost.

Design

Efficiency

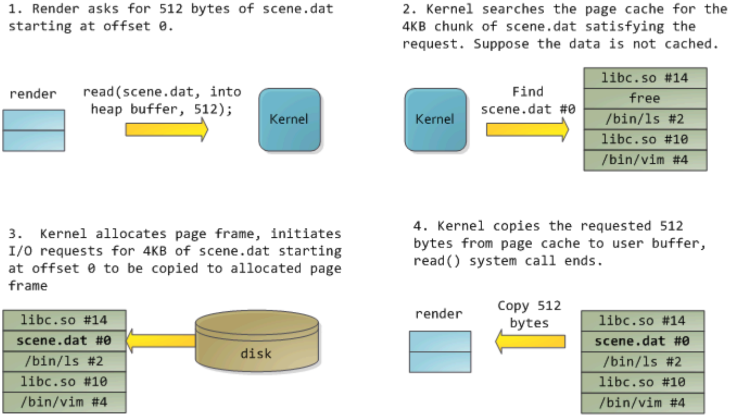
To illustrate the page cache, I'll conjure a Linux program named render, which opens file scene.dat and reads it 512 bytes at a time, storing the file contents into a heap-allocated block.



Design

Efficiency

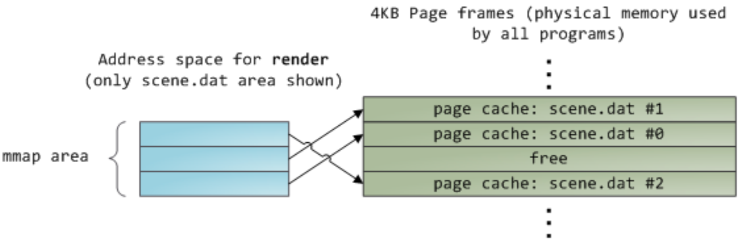
The first read goes like this:



Design

Efficiency

When you use memory-mapped files, the kernel maps your program's virtual pages directly onto the page cache.



Design

Efficiency

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

So long as there's enough free physical memory, the cache should be kept full. It is therefore not dependent on a particular process, but rather it's a system-wide resource.

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

Design

Efficiency

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.
- Java garbage collection becomes increasingly fiddly and slow as the in-heap data increases.

Design

Efficiency

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.

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

Design

Efficiency

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

Efficiency

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

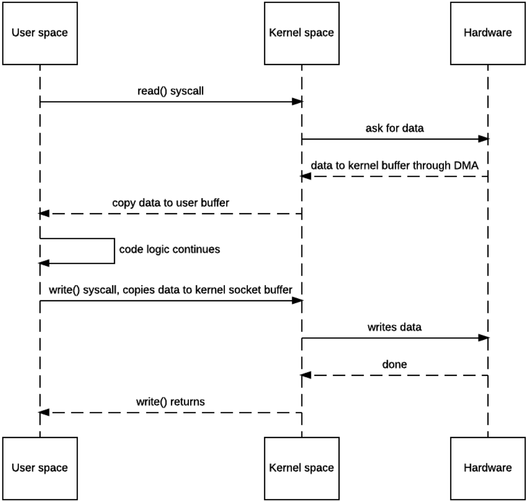
Efficiency

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

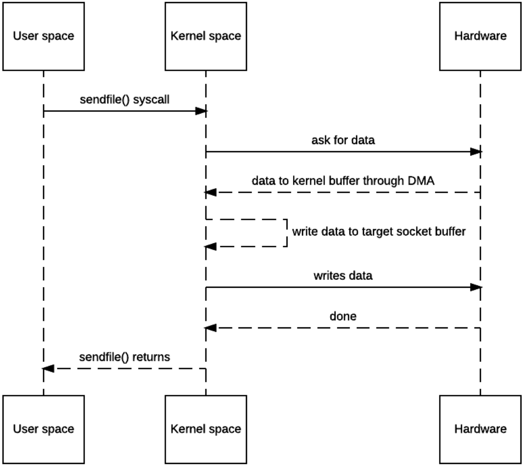
This is clearly inefficient, there are four copies and two system calls.

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

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

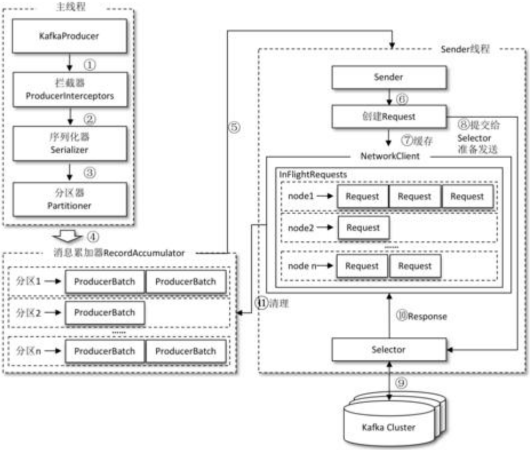
Batching is one of the big drivers of efficiency, and to enable batching the Kafka producer will attempt to accumulate data in memory and to send out larger batches in a single request.

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)

$\Downarrow$

ProducerBatch (Sender Thread)

$\Downarrow$

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

$\Downarrow$

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

$\Downarrow$

<Node, Request>(Sender Thread)

$\Downarrow$

Broker(page cache)

$\Downarrow$

Broker(log segment)

Design

The Producer

Kafka provides 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 Consumer

The Kafka consumer works by issuing "fetch" requests to the brokers leading the partitions it wants to consume.

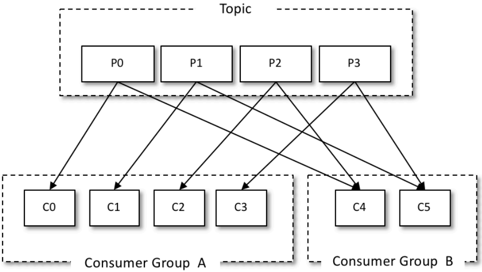
The consumer specifies its offset in the log with each request and receives back a chunk of log beginning from that position.

The consumer thus has significant control over this position and can rewind it to re-consume data if need be.

Design

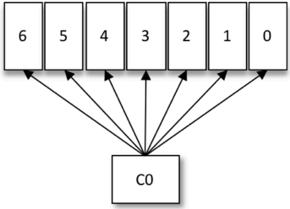
The Consumer

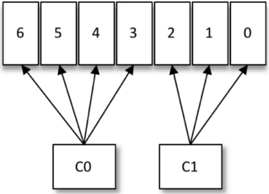
Our topic is divided into a set of totally ordered partitions, each of which is consumed by exactly one consumer within each subscribing consumer group at any given time



Design

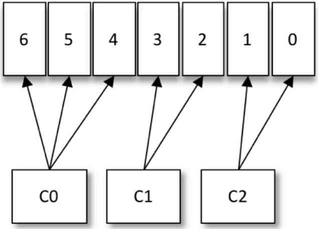
The Consumer

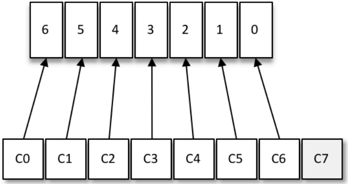




Design

The Consumer





Design

The Consumer

The rebalance protocol relies on the group coordinator to allocate entity ids to group members.

These generated ids are ephemeral and will change when members restart and rejoin.

Motivated by this observation, Kafka's group management protocol allows group members to provide persistent entity ids(2.3+).

Design

Message Delivery Semantics

Let's discuss the semantic guarantees Kafka provides between producer and consumer.

- at most once
- at least once
- exactly once

It's worth noting that this breaks down into two problems: the durability guarantees for publishing a message and the guarantees when consuming a message.

Design

Message Delivery Semantics

When publishing a message we have a notion of the message being "committed" to the log.

Once a published message is committed it will not be lost as long as one broker that replicates the partition to which this message was written remains "alive" (at least once).

Since 0.11.0.0, the Kafka producer also supports an idempotent delivery option which guarantees that resending will not result in duplicate entries in the log (exactly once).

Design

Message Delivery Semantics

The consumer controls its position in this log.

It has several options for processing the messages and updating its position.

- at least once(read, update, process)
- at most once(read, process, update)
- exactly once
  - to topic(producer transactional capabilities)
  - to external system(store offset in the same place as its output)

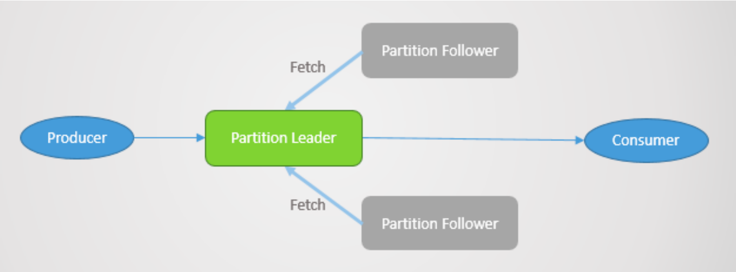
Design

Replication

Kafka replicates the log for each topic's partitions across a configurable number of servers.

Under non-failure conditions, each partition in Kafka has a single leader and zero or more followers

All reads and writes go to the leader of the partition.



Design

Replication

As with most distributed systems automatically handling failures requires having a precise definition of what it means for a node to be "alive".

For Kafka node liveness has two conditions

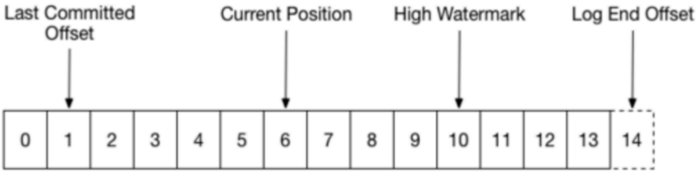
1. A node must be able to maintain its session with ZooKeeper (via ZooKeeper's heartbeat mechanism)
2. If it is a follower it must replicate the writes happening on the leader and not fall "too far" behind

Design

Replication

We refer to nodes satisfying these two conditions as being "in sync" to avoid the vagueness of "alive" or "failed".

The leader keeps track of the set of "in sync" nodes.



The guarantee that Kafka offers is that a committed message will not be lost, as long as there is at least one in sync replica alive, at all times.

Design

Replication

Instead of majority vote, Kafka dynamically maintains a set of in-sync replicas (ISR) that are caught-up to the leader.

Only members of this set are eligible for election as leader.

A write to a Kafka partition is not considered committed until all in-sync replicas have received the write.

Design

Replication

This ISR set is persisted to ZooKeeper whenever it changes.

Because of this, any replica in the ISR is eligible to be elected leader. This is an important factor for Kafka's usage model where there are many partitions and ensuring leadership balance is important.

With this ISR model and f+1 replicas, a Kafka topic can tolerate f failures without losing committed messages.

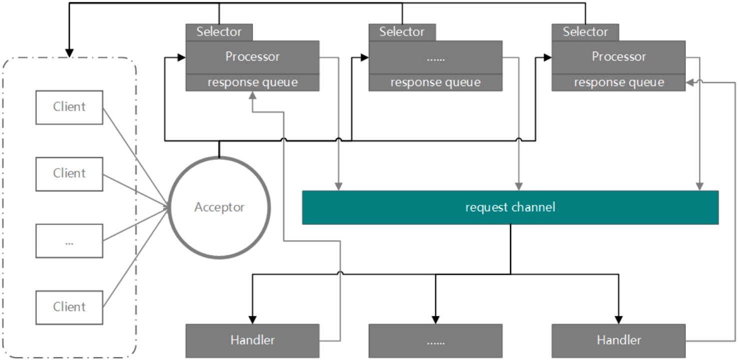
Outline

Implementation

Network Layer

The network layer is a fairly straight-forward NIO server.

The threading model is a single acceptor thread and N processor threads which handle a fixed number of connections each.



Implementation

Messages

Messages consist of a variable-length header, a variable-length opaque key byte array and a variable-length opaque value byte array.

Leaving the key and value opaque enable user to choose a particular serialization type.

The RecordBatch interface is simply an iterator over messages with specialized methods for bulk reading and writing to an NIO Channel.

Implementation

Messages

The following is the on-disk format of a RecordBatch.

```
baseOffset: int64
batchLength: int32
partitionLeaderEpoch: int32
magic: int8 (current magic value is 2)
crc: int32
attributes: int16
    bit 0~2:
        0: no compression
        1: gzip
        2: snappy
        3: lz4
        4: zstd
    bit 3: timestampType
    bit 4: isTransactional (0 means not transactional)
    bit 5: isControlBatch (0 means not a control batch)
    bit 6~15: unused
lastOffsetDelta: int32
firstTimestamp: int64
maxTimestamp: int64
producerId: int64
producerEpoch: int16
baseSequence: int32
records: [Record]

```

Implementation

Messages

Record level headers were introduced in Kafka 0.11.0. The on-disk format of a record with Headers is delineated below.

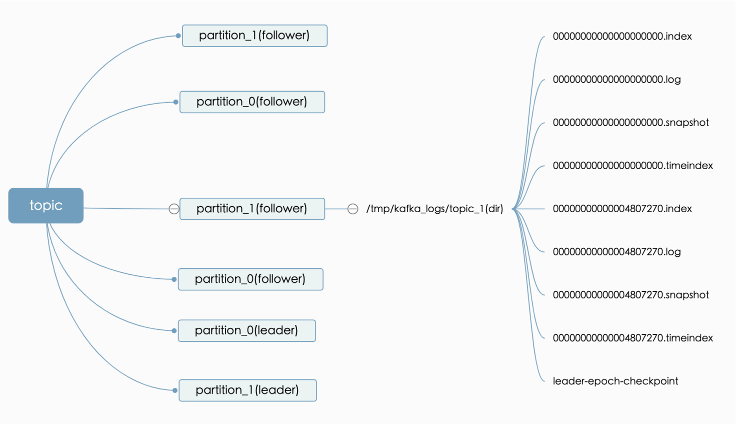
```
length: varint
attributes: int8
    bit 0~7: unused
timestampDelta: varint
offsetDelta: varint
keyLength: varint
key: byte[]
valueLen: varint
value: byte[]
Headers => [Header]
```

```
headerKeyLength: varint
headerKey: String
headerValueLength: varint
Value: byte[]
```

Implementation

Log

Each log file is named with the offset of the first message it contains.



Implementation

Log

The format of the log files is a sequence of "log entries""; each log entry is a 4 byte integer N storing the message length.

Each message is uniquely identified by a 64-bit integer offset giving the byte position of the start of this message in the stream of all messages ever sent to that topic on that partition.

The exact binary format for records is versioned and maintained as a standard interface so record batches can be transferred between producer, broker, and client without recopying or conversion when desirable.

Implementation

Log

The log allows serial appends which always go to the last file.

This file is rolled over to a fresh file when it reaches a configurable size M or a number of seconds S.

This gives a durability guarantee of losing at most M messages or S seconds of data in the event of a system crash.

Implementation

Log

Reads are done by giving the 64-bit logical offset of a message and an S-byte max chunk size

S is intended to be larger than any single message, but in the event of an abnormally large message, the read can be retried multiple times, each time doubling the buffer size, until the message is read successfully.

It is likely that the read buffer ends with a partial message, this is easily detected by the size delimiting.

Implementation

Log

The actual process of reading from an offset requires first locating the log segment file, calculating the file-specific offset from the global offset value, and then reading from that file offset.

The search is done as a simple binary search variation against an in-memory range maintained for each file.

Implementation

Consumer Offset Tracking

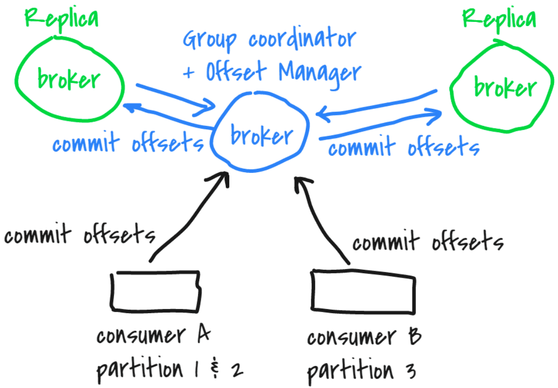
Kafka consumer tracks the maximum offset it has consumed in each partition and has the capability to commit offsets .

Offset commits can be done automatically or manually by consumer instance.

When the group coordinator receives an OffsetCommitRequest, it appends the request to a special compacted Kafka topic named __consumer_offsets.

Implementation

Consumer Offset Tracking



Questions and Answers?

Questions and Answers?

Thank You!

References I