The Design and Implementation of Kafka

Wang Sheying

HuiLongGuan of Beijing

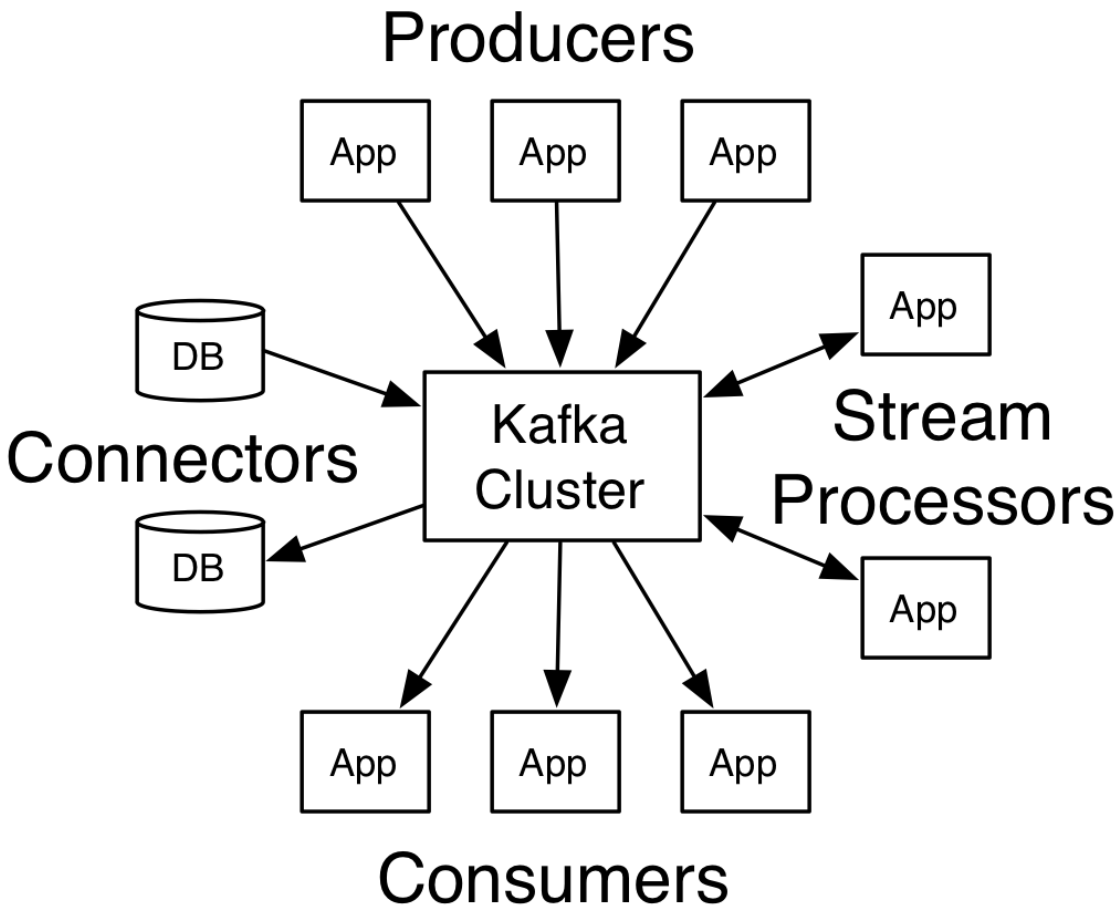
April 21, 2020

Outline

Outline

Introduction

Apache Kafka is used for building real-time data pipelines and streaming apps.



Introduction

The Features of the Kafka

- high throughput
- low latency
- high reliability
- high concurrency
- scalable horizontally

Introduction

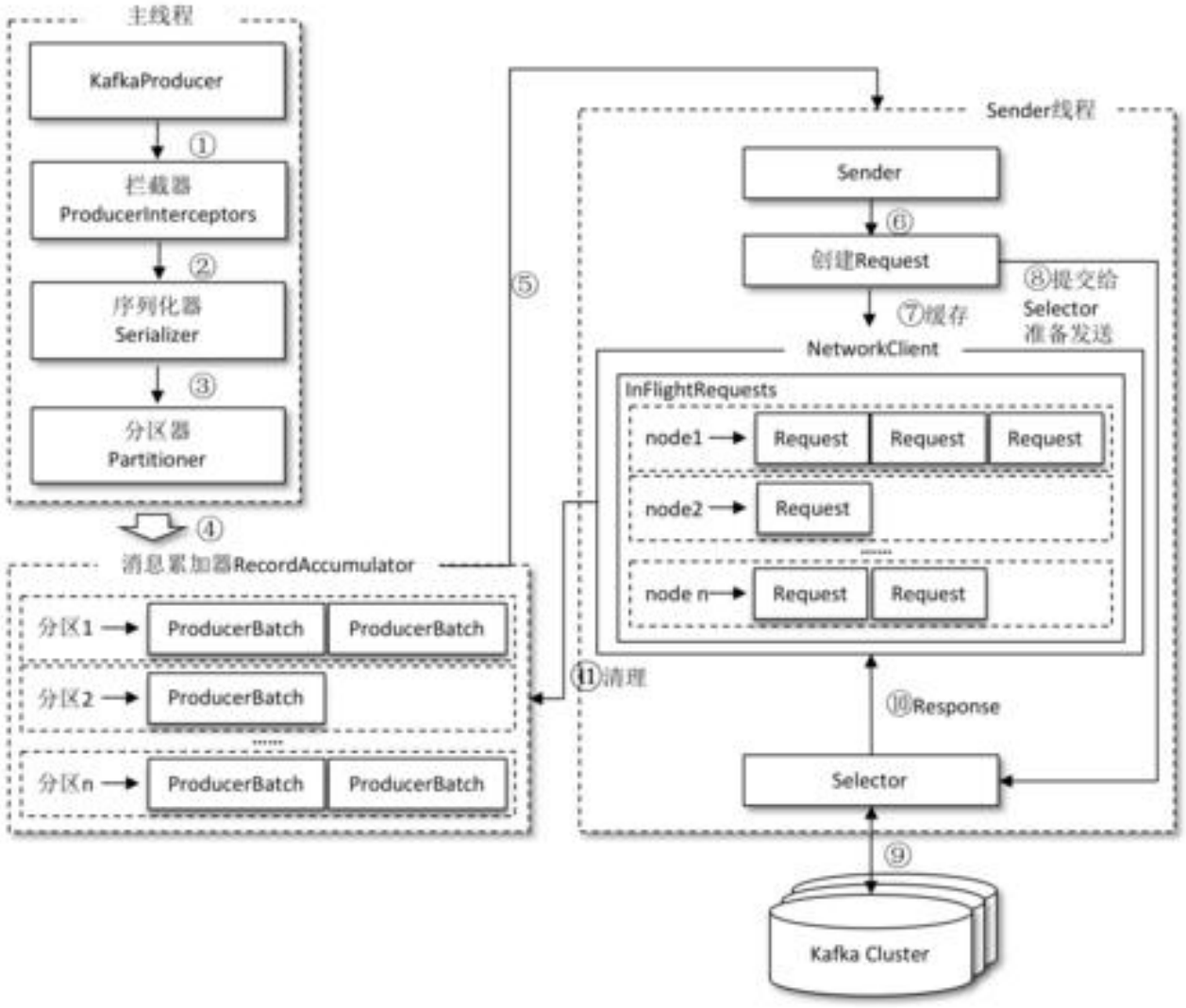
The popular use cases for Apache Kafka

- messaging
- log aggregation
- stream processing
- website activity tracking
- metrics

Outline

Producer

Architecture



Producer

Encapsulation

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)

Producer

Asynchronous Send

The send() method is asynchronous.

`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

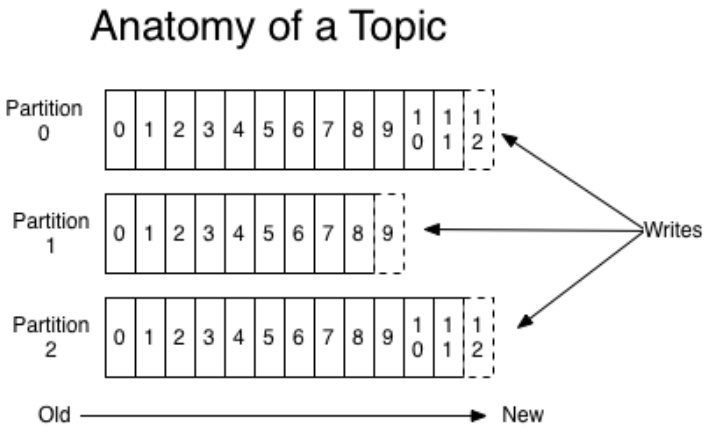
The acks config controls the criteria under which requests are considered complete.

From Kafka 0.11, the KafkaProducer supports two additional modes:

- the idempotent producer
- the transactional producer

Producer

Asynchronous Send



(a) ordering of messages

Producer

Message Format

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]

```

Producer

Message Format

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]
```

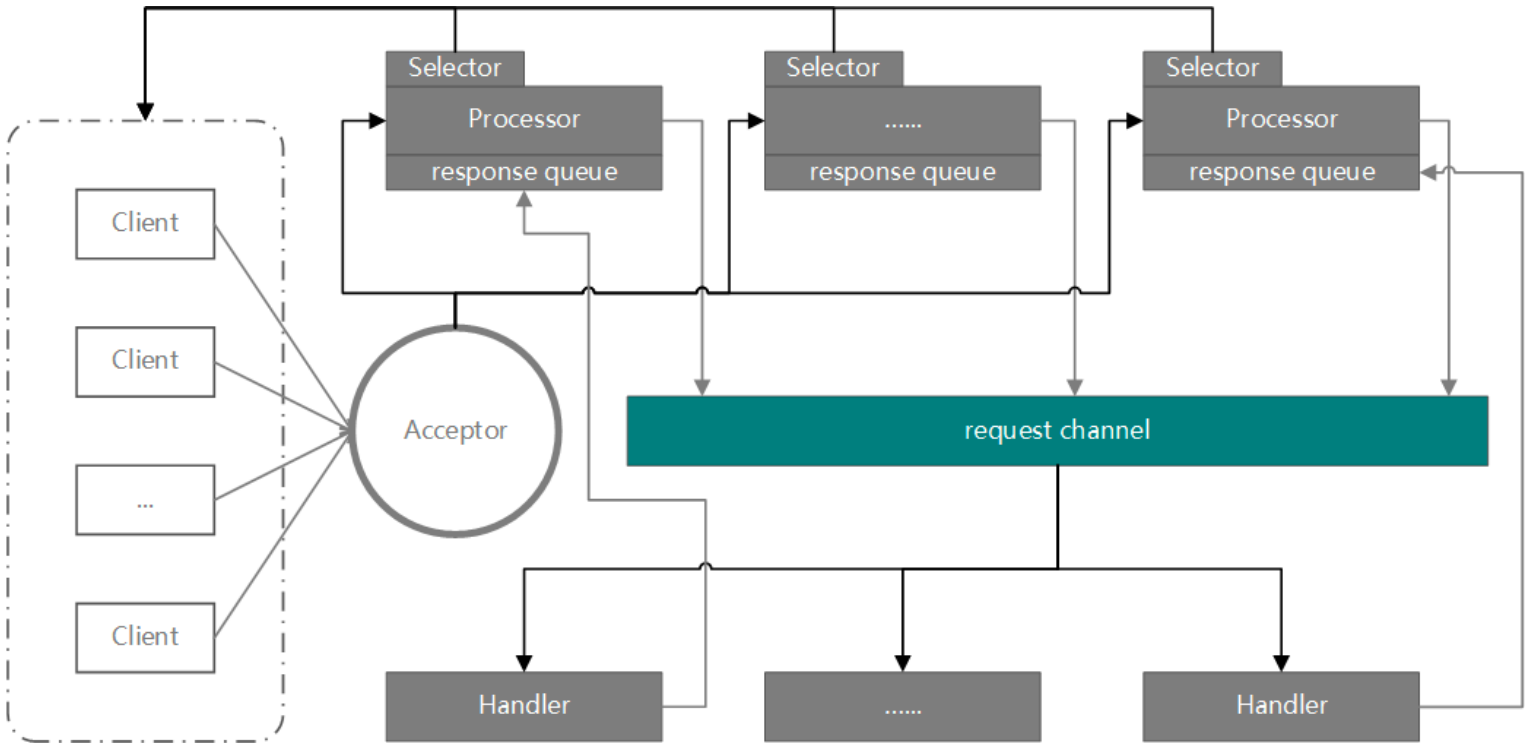
```
headerKeyLength: varint
headerKey: String
headerValueLength: varint
Value: byte[]
```

Outline

Broker

Architecture

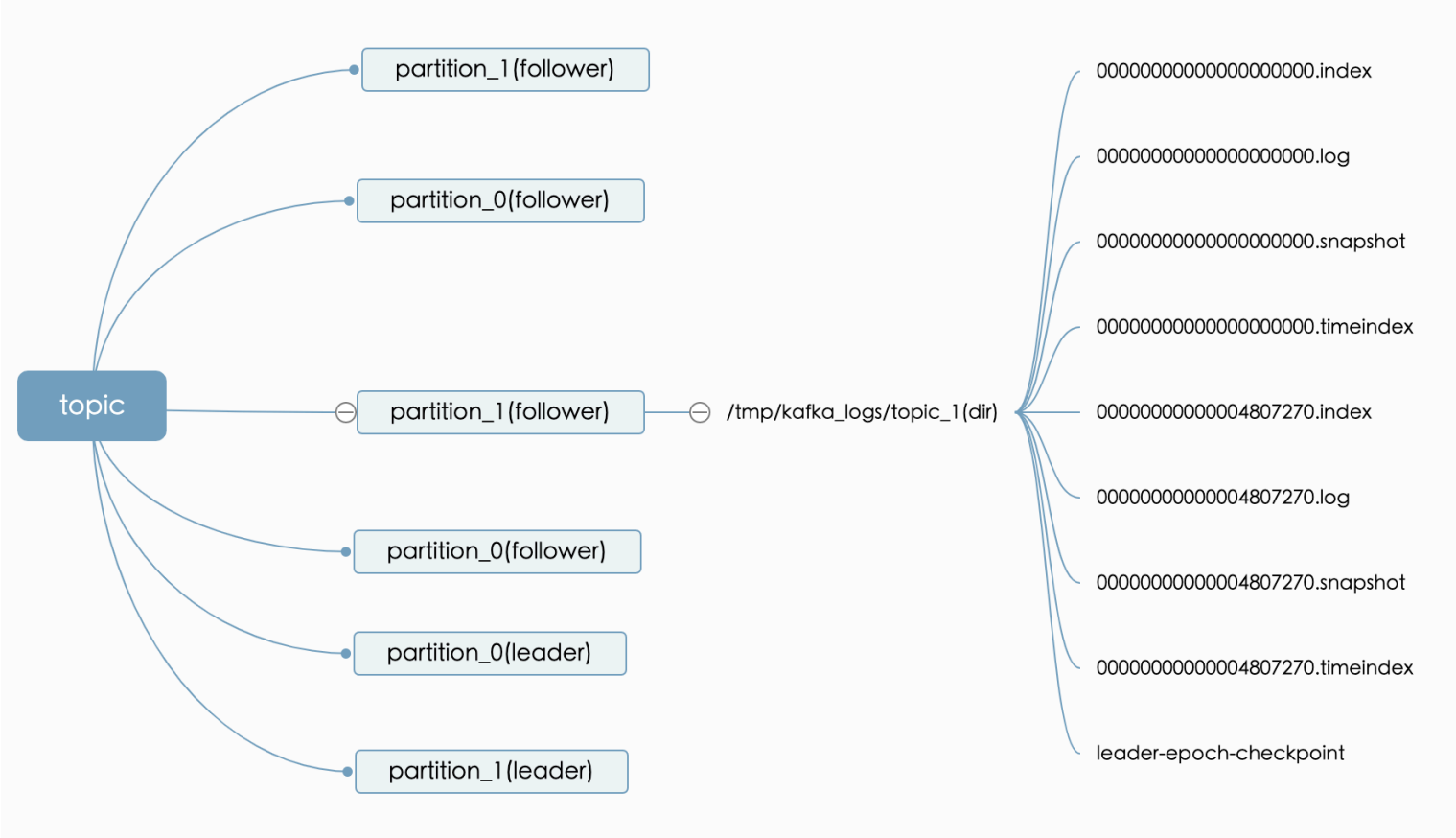
The threading model is a single acceptor thread and N processor threads which handle a fixed number of connections each.



Broker

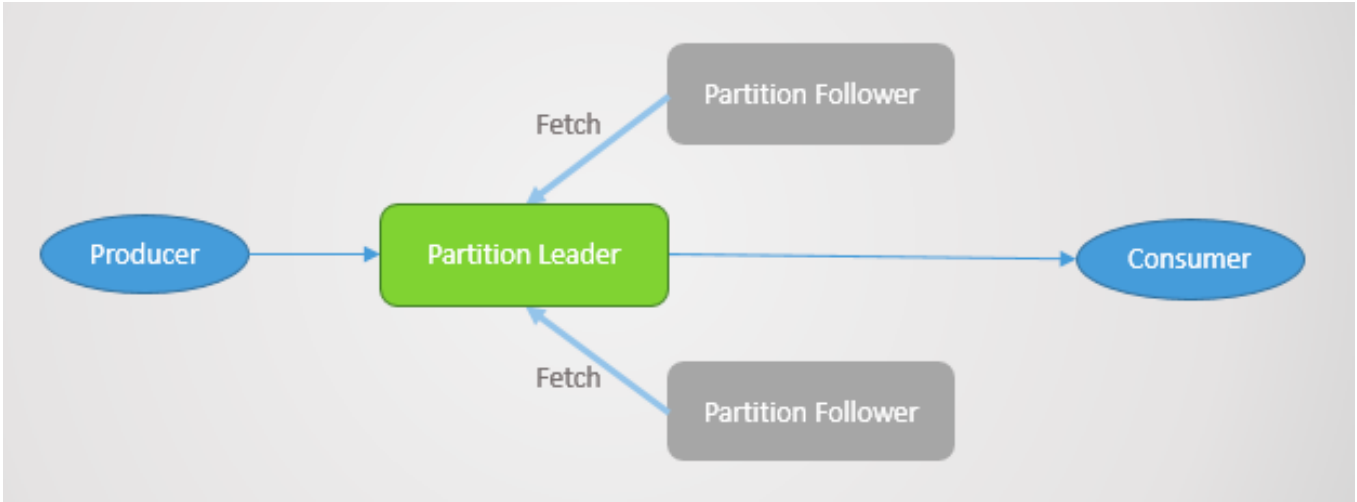
Log

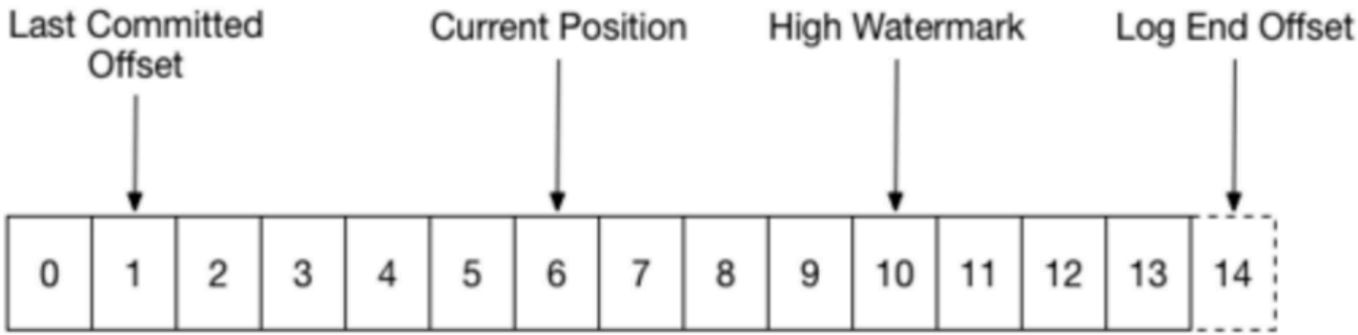
Each log file is named with the offset of the first message it contains.



Broker

Failover

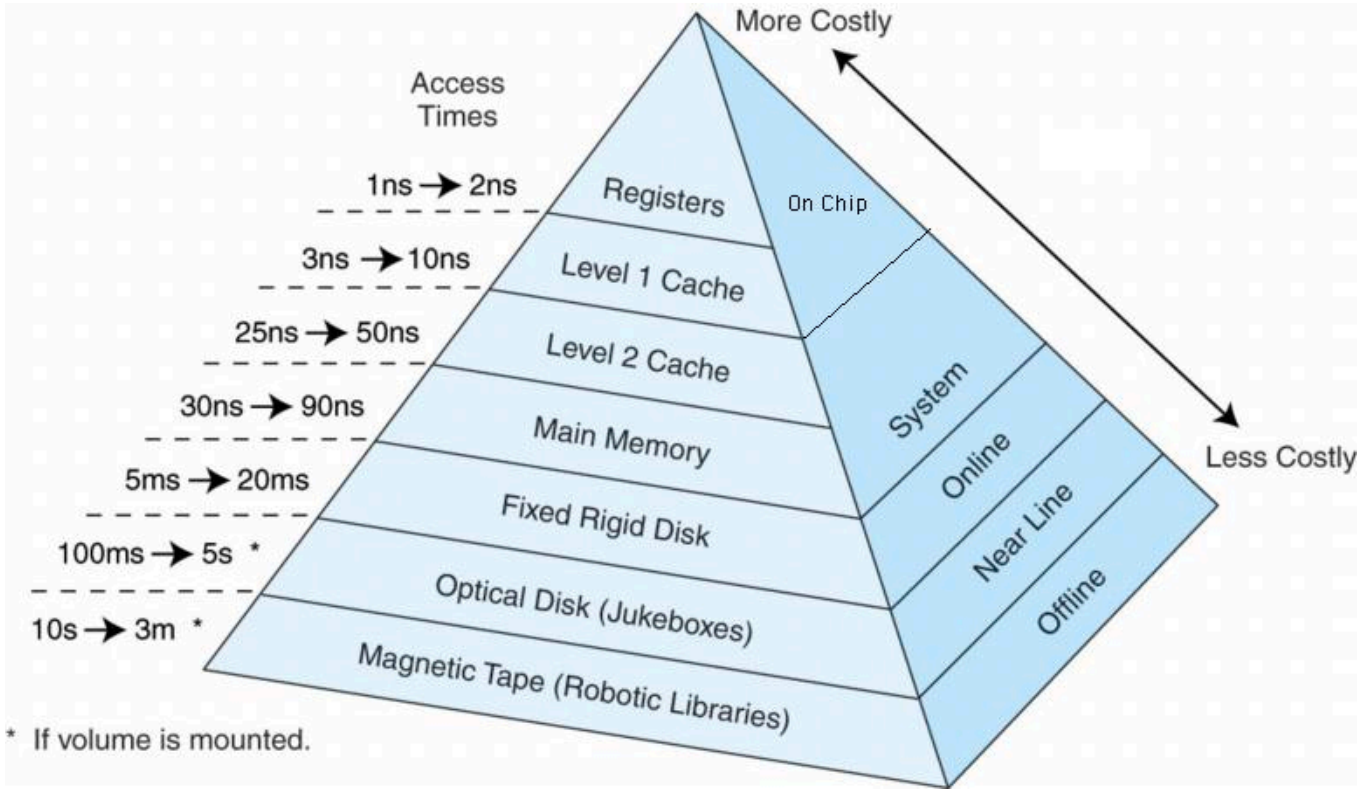




Broker

Why Kafka Is so Fast

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

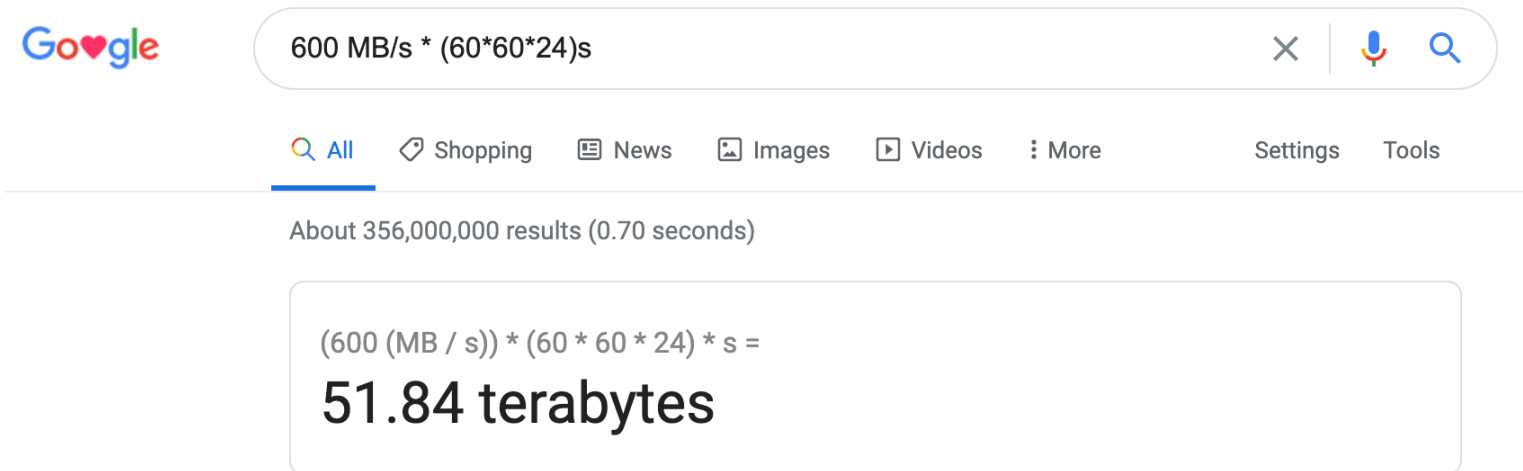


Broker

Why Kafka Is so Fast

As a result the performance of linear writes on a JBOD configuration with six 7200rpm SATA RAID-5 array.

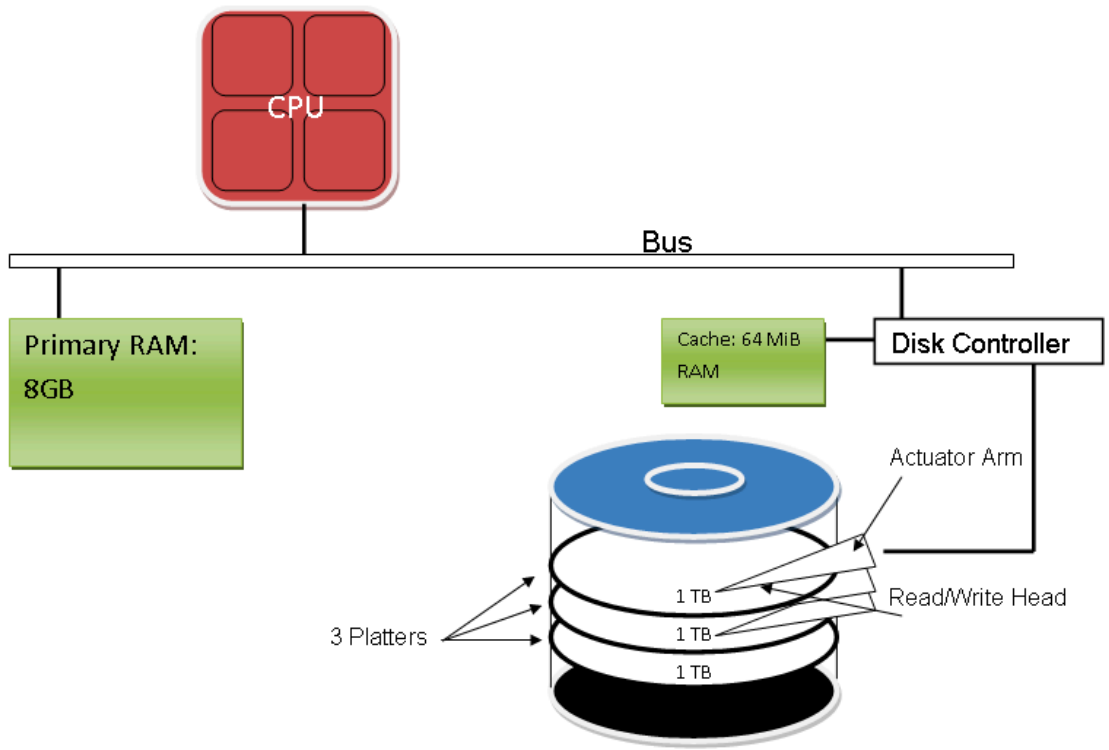
- linear writes 600MB/sec
- random writes 100k/sec



Broker

Why Kafka Is so Fast

To illustrate the page cache, a Linux program named render, which opens file scene.dat and reads it 512 bytes at a time.

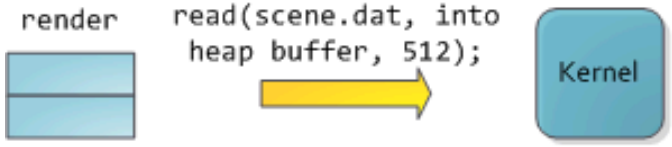


Broker

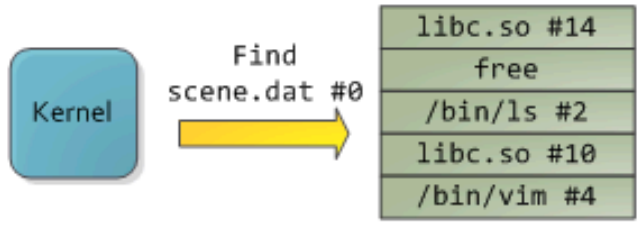
Why Kafka Is so Fast

The first read goes like this:

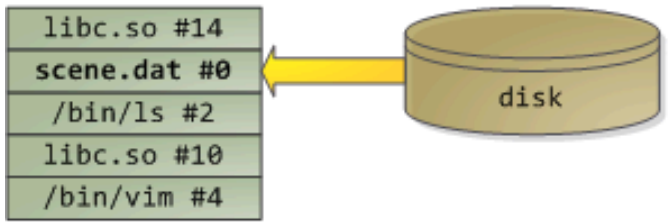
1. Render asks for 512 bytes of scene.dat starting at offset 0.



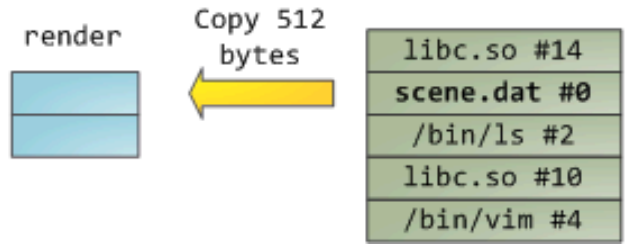
2. Kernel searches the page cache for the 4KB chunk of scene.dat satisfying the request. Suppose the data is not cached.



3. Kernel allocates page frame, initiates I/O requests for 4KB of scene.dat starting at offset 0 to be copied to allocated page frame



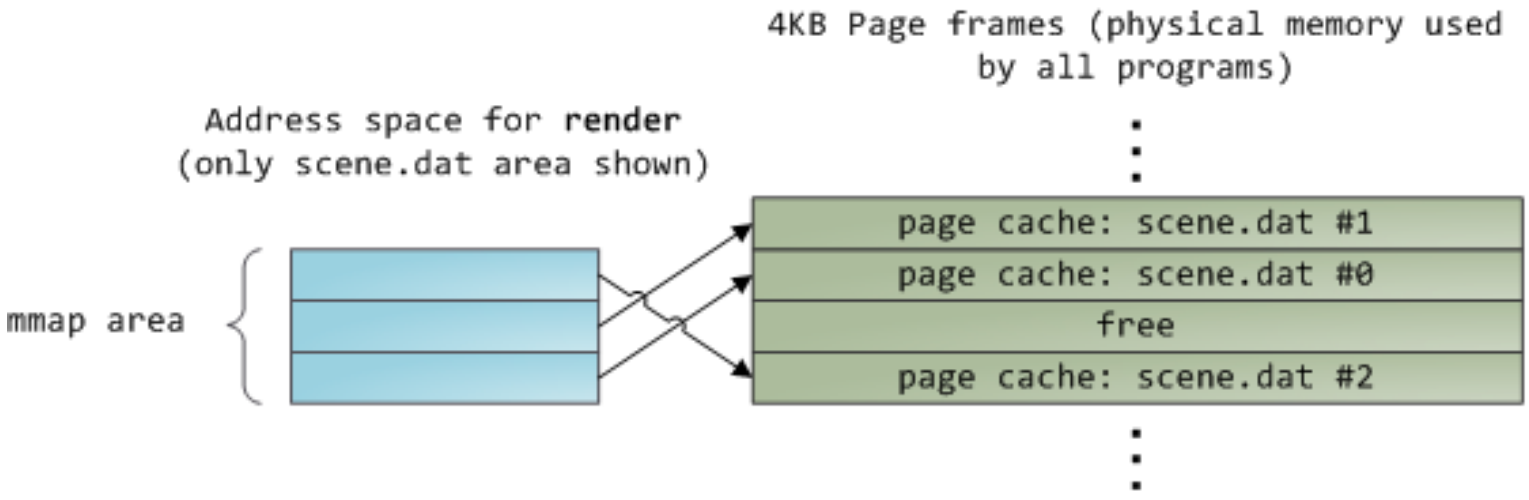
4. Kernel copies the requested 512 bytes from page cache to user buffer, read() system call ends.



Broker

Why Kafka Is so Fast

For reuse, the kernel will use memory-mapped files to map your program's virtual pages directly onto the page cache.



Broker

Why Kafka Is so Fast

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

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

Broker

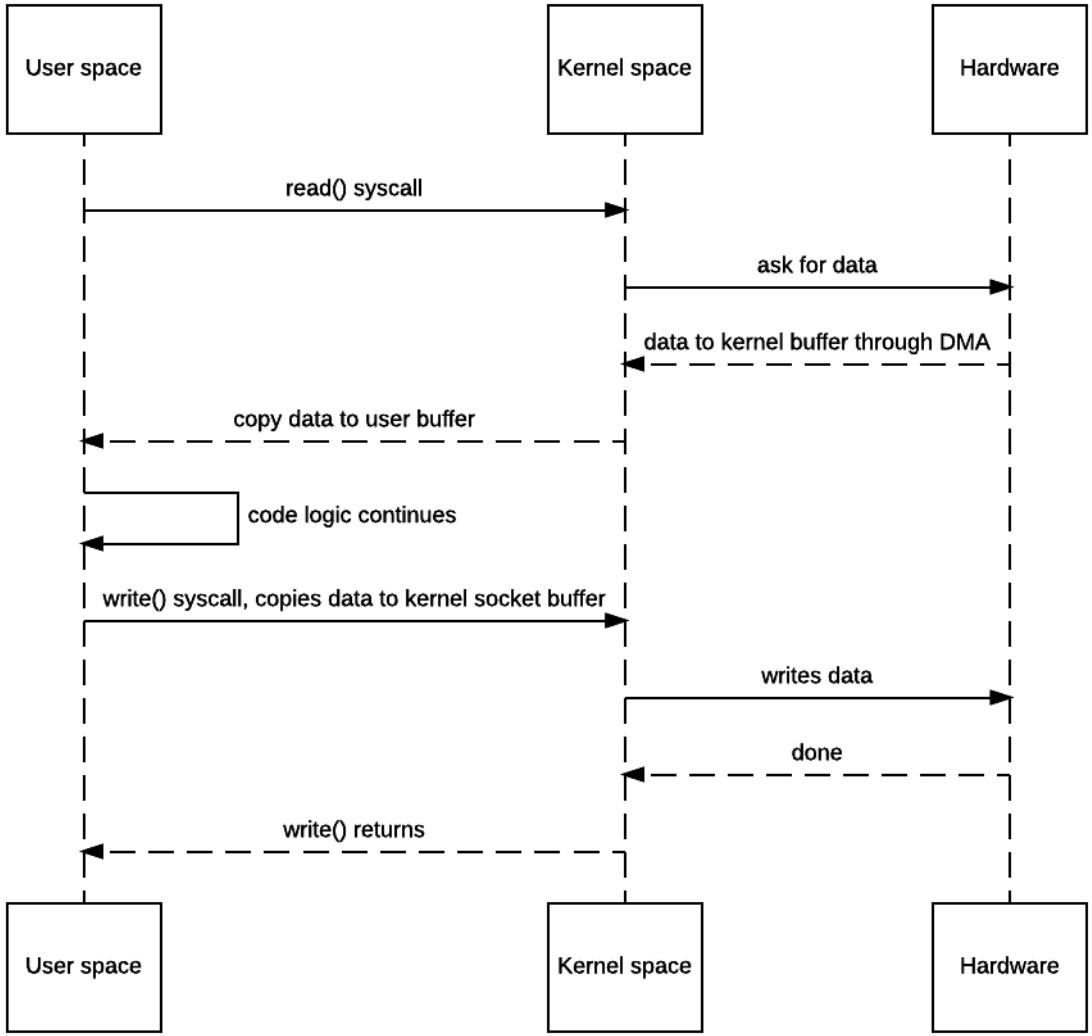
Why Kafka Is so Fast

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

1. too many small I/O operations
   - batch
   - compression
2. excessive byte copying
   - a standardized binary message format
   - zero copy

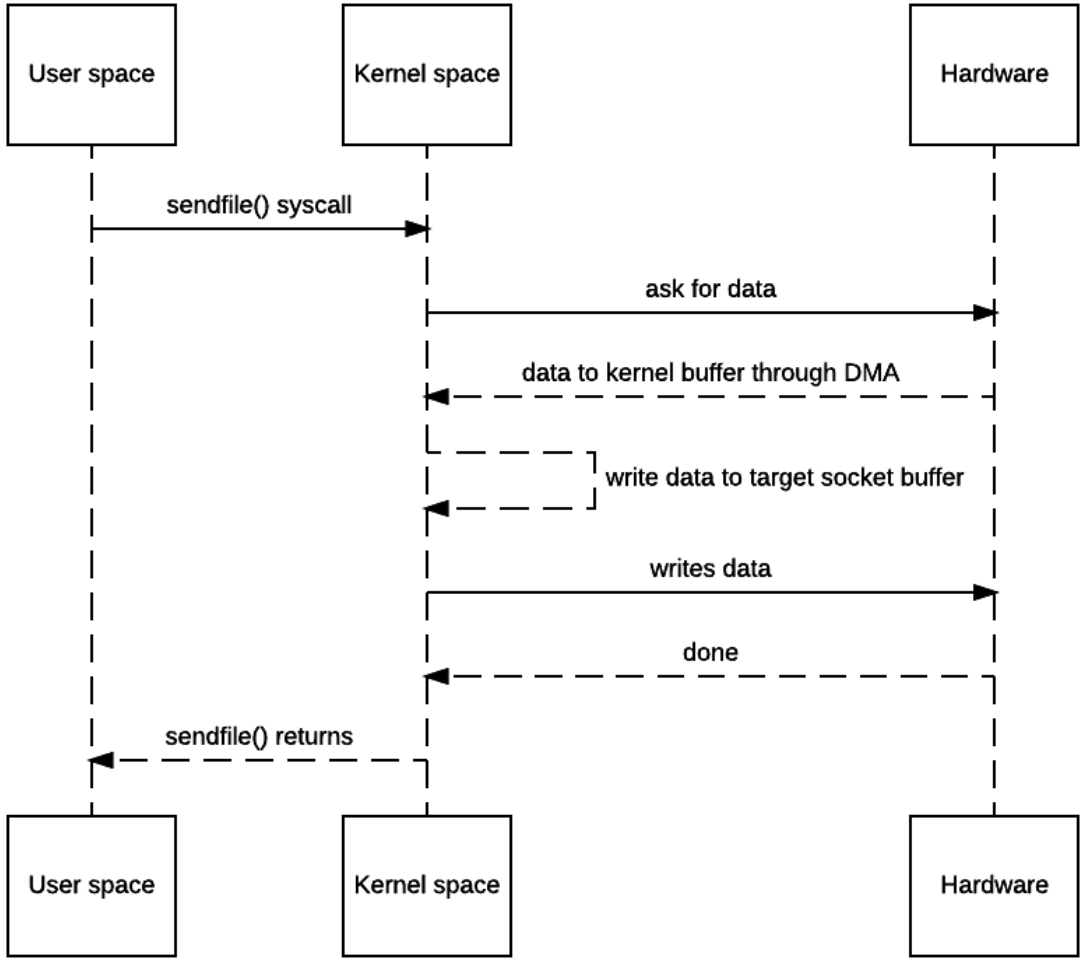
Broker

Why Kafka Is so Fast



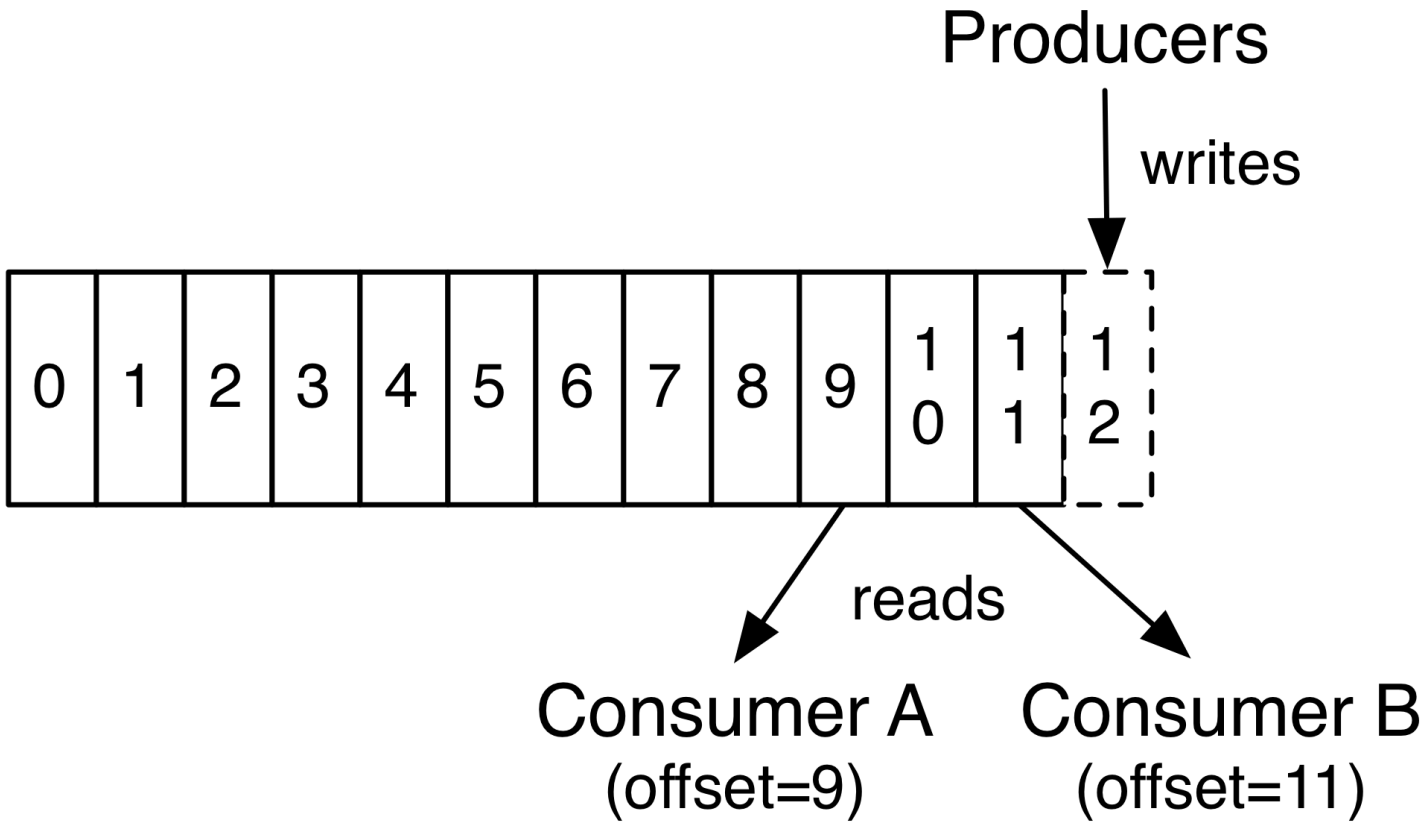
Broker

Why Kafka Is so Fast



Broker

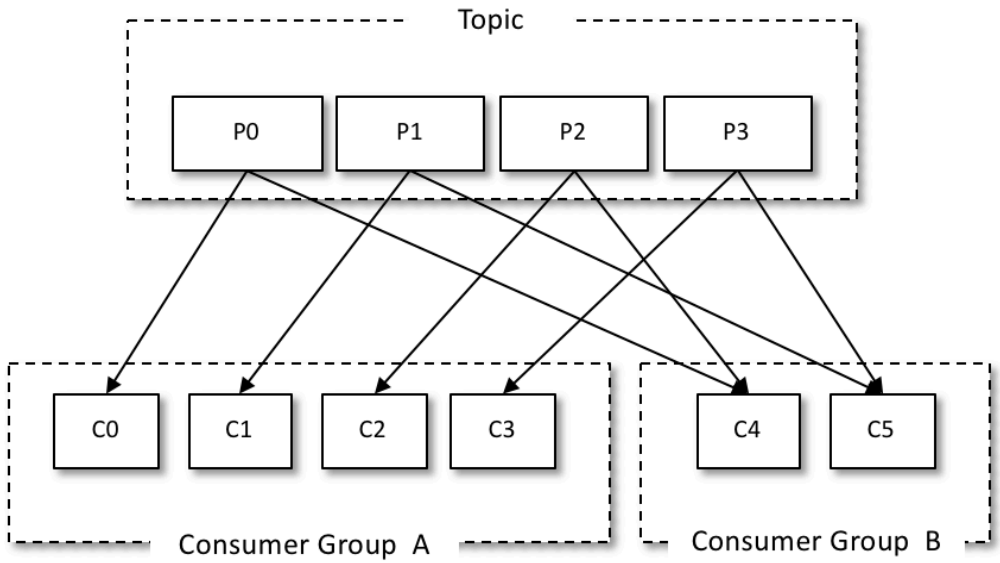
Why Kafka Is so Fast



Outline

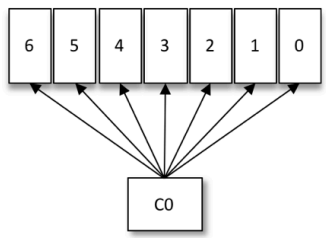
Consumer

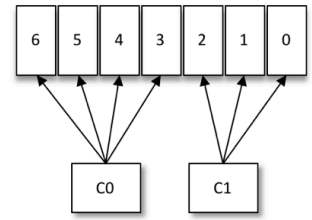
Consumer Groups



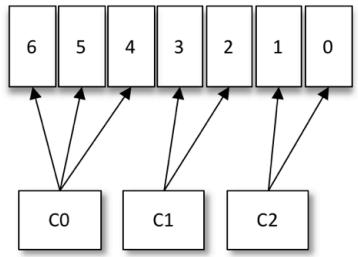
Consumer

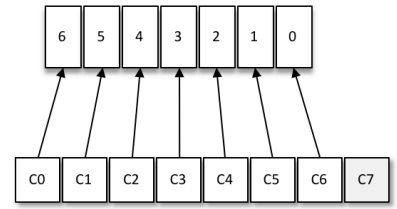
Consumer Groups





(a) group rebalance

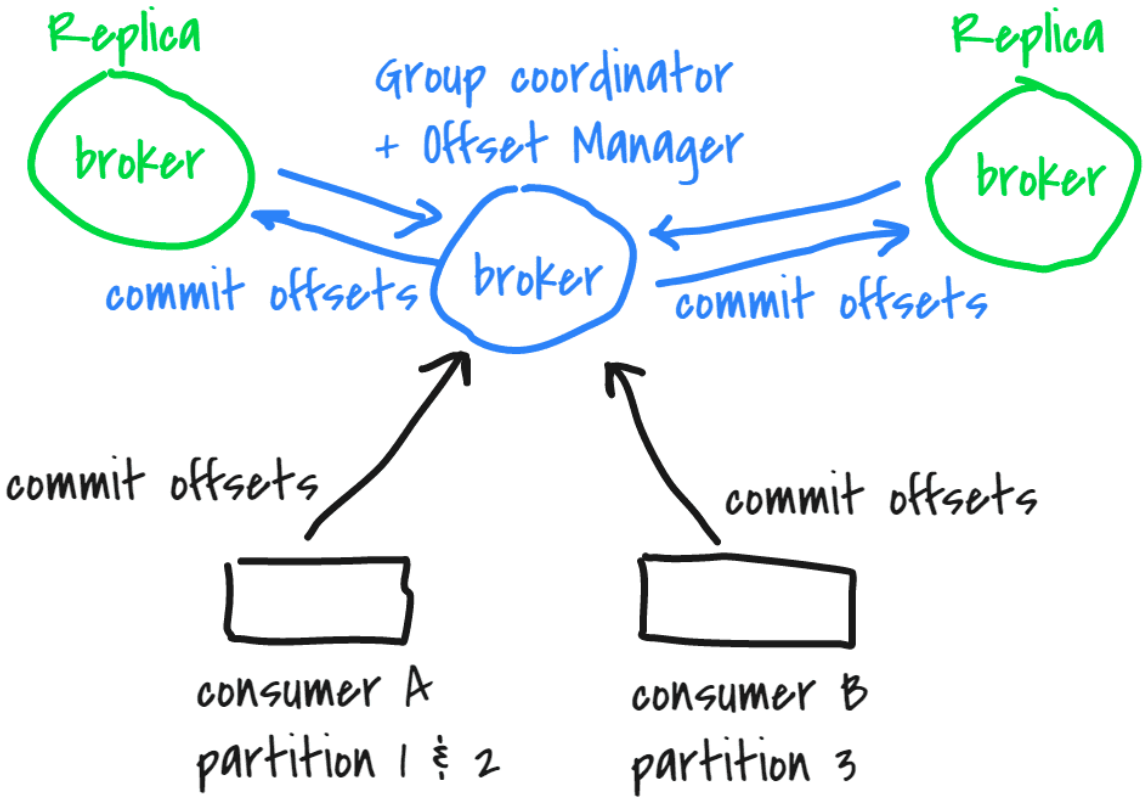




(b) group rebalance

Consumer

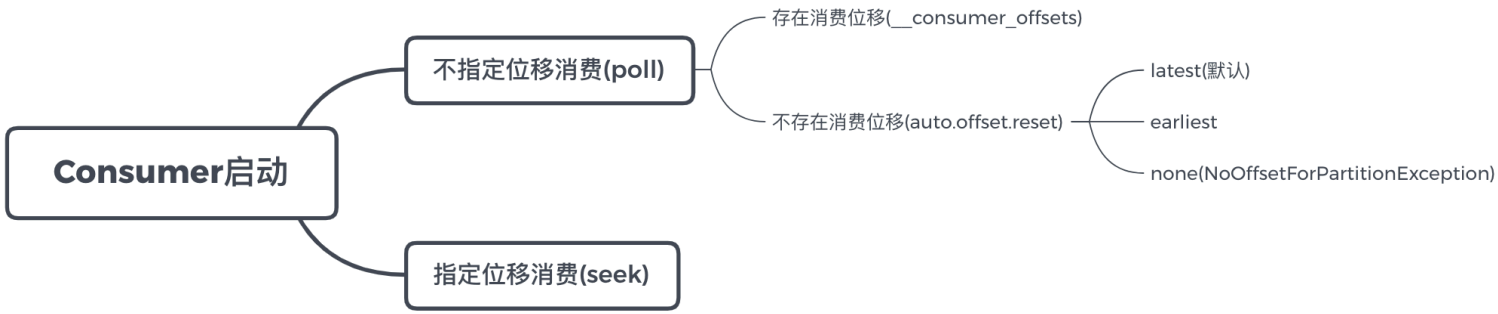
Offsets and Consumer Position



Consumer

Offsets and Consumer Position

```
[mygroup1,mytopic1,11]::[OffsetMetadata[55166421,NO_METADATA],CommitTime 1502060076305,ExpirationTime 1502146476305]
[mygroup1,mytopic1,13]::[OffsetMetadata[55037927,NO_METADATA],CommitTime 1502060076305,ExpirationTime 1502146476305]
[mygroup2,mytopic2,0]::[OffsetMetadata[126,NO_METADATA],CommitTime 1502060076343,ExpirationTime 1502146476343]
```



Consumer

Storing Offsets Outside Kafka

The application can store both the offset and the results of the consumption in the same system.

It will make the consumption fully atomic and give "exactly once" semantics.

Each record comes with its own offset, so to manage your own offset:

- configure enable.auto.commit=false
- use each offset to save your position
- on restart restore the position of the consumer using seek()

Questions and Answers?

Questions and Answers?

Thank You!

References I