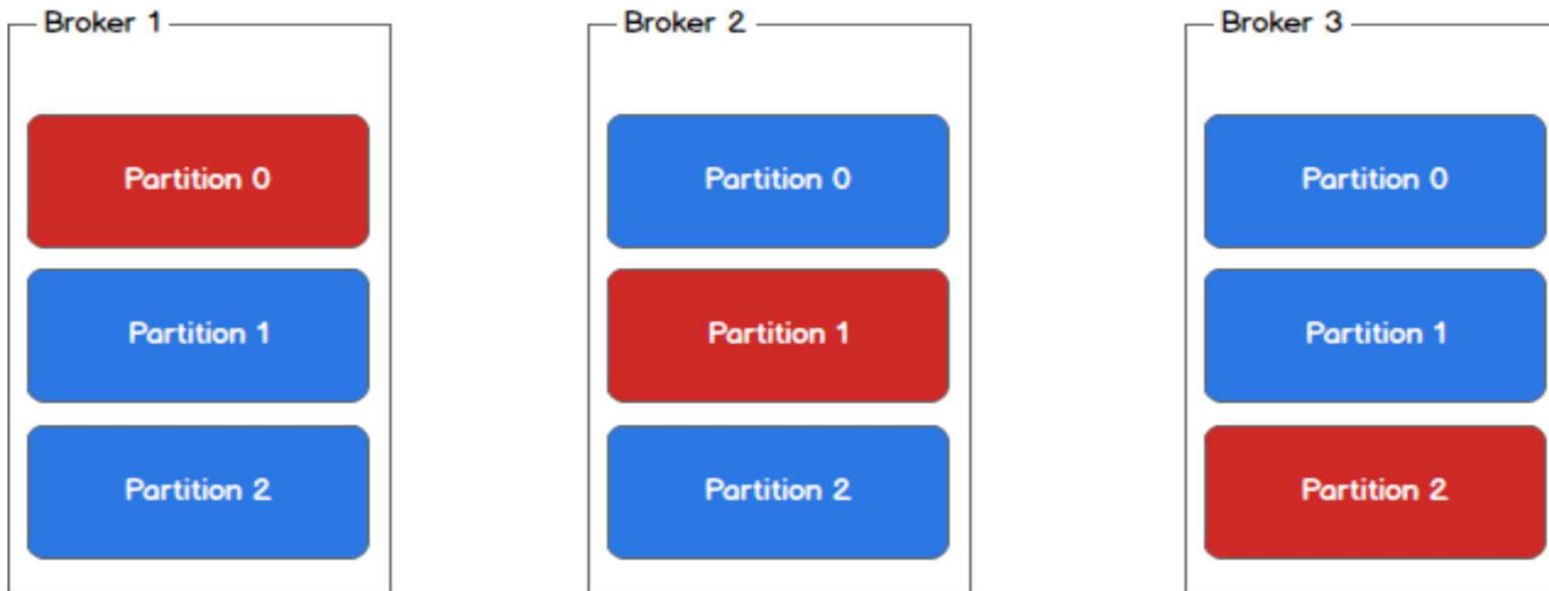
A distributed pub/sub platform: Apache Kafka (Part 2)

Prof. Carlo Ferrari

Michele Stecca, Ph.D.

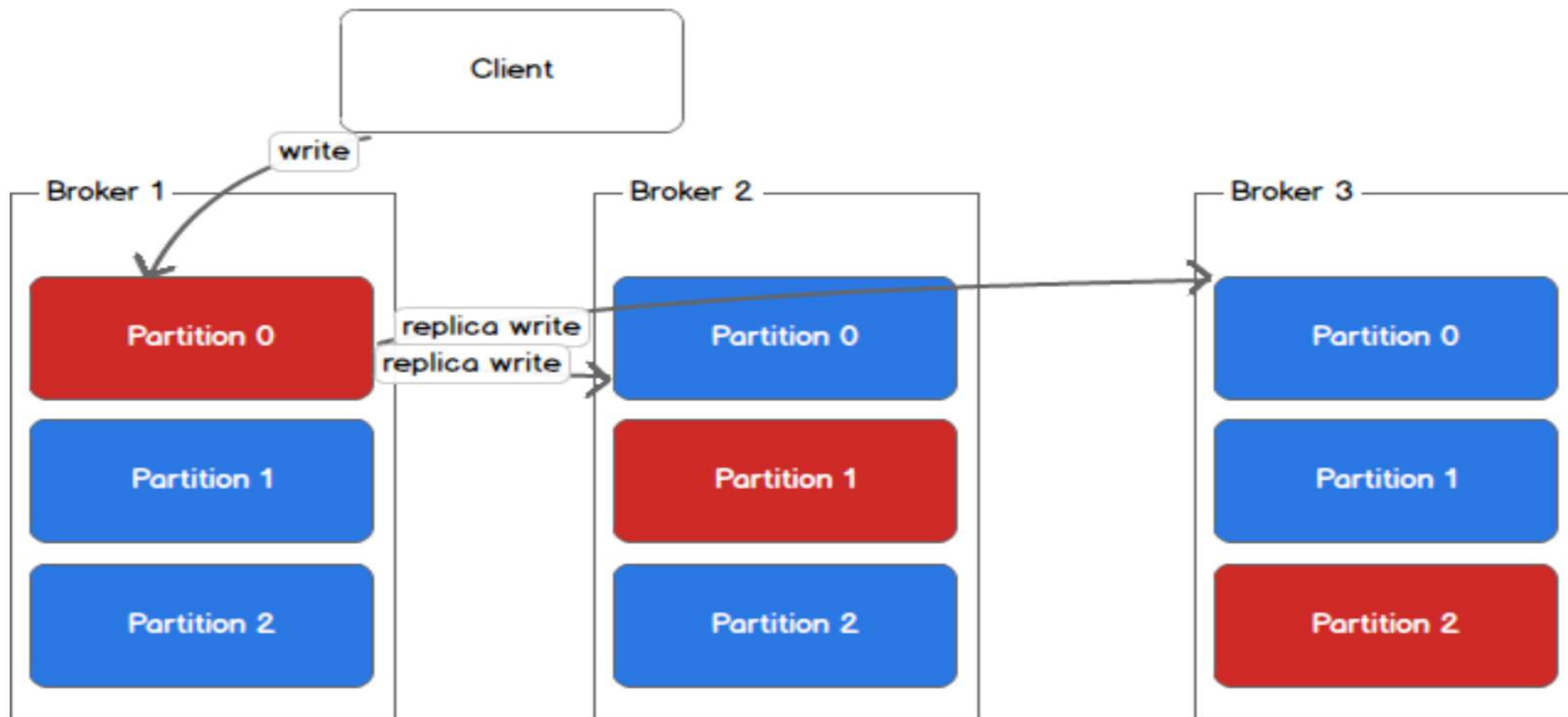
Kafka partitions (Replication)

- Each broker holds a number of partitions and each of these partitions can be either a **leader** or a **replica** for a topic. Thus, each partition has one leader and multiple in-sync replicas **(ISR).**
- All writes and reads to a topic go through the leader and the **leader coordinates updating replicas** with new data.



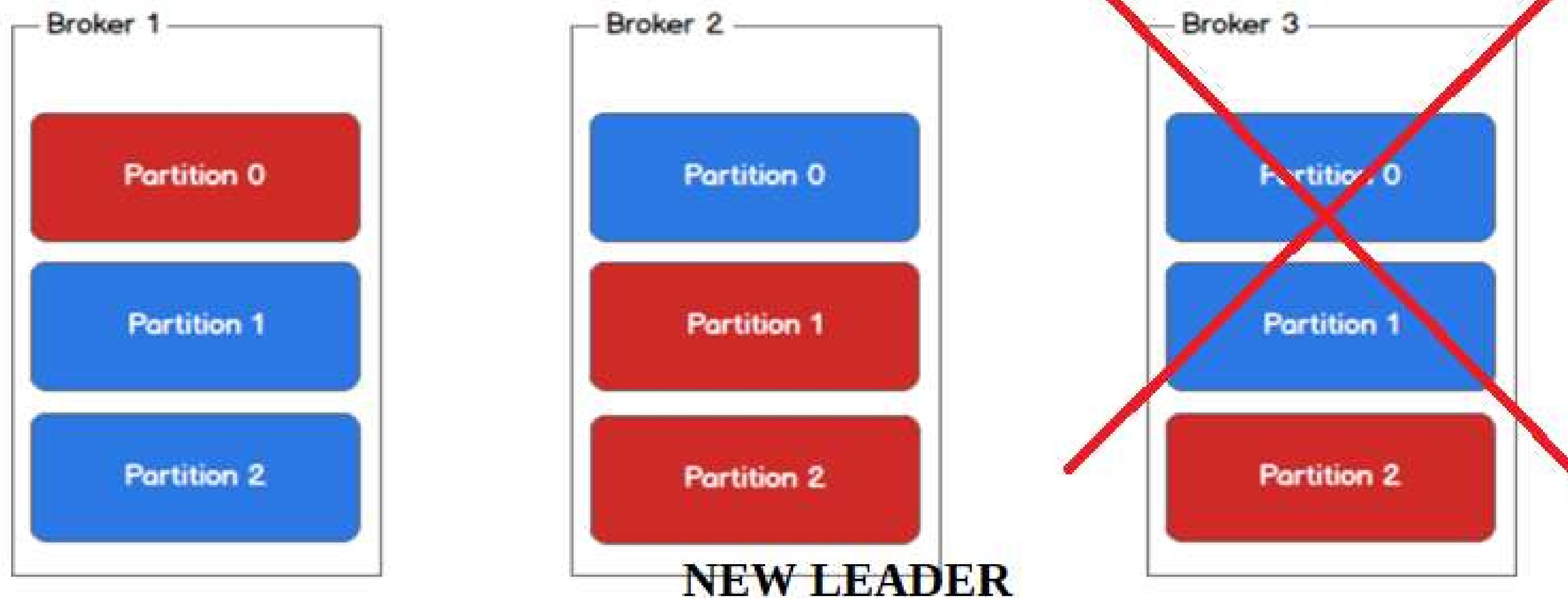
Kafka partitions (Producers)

Producers write to a single leader, this provides a means of load balancing production so that each write can be serviced by a separate broker and machine.



Kafka partitions (Fault Tolerance)

If a leader fails, a replica takes over as the new leader.



Unclean election: What if they all die?

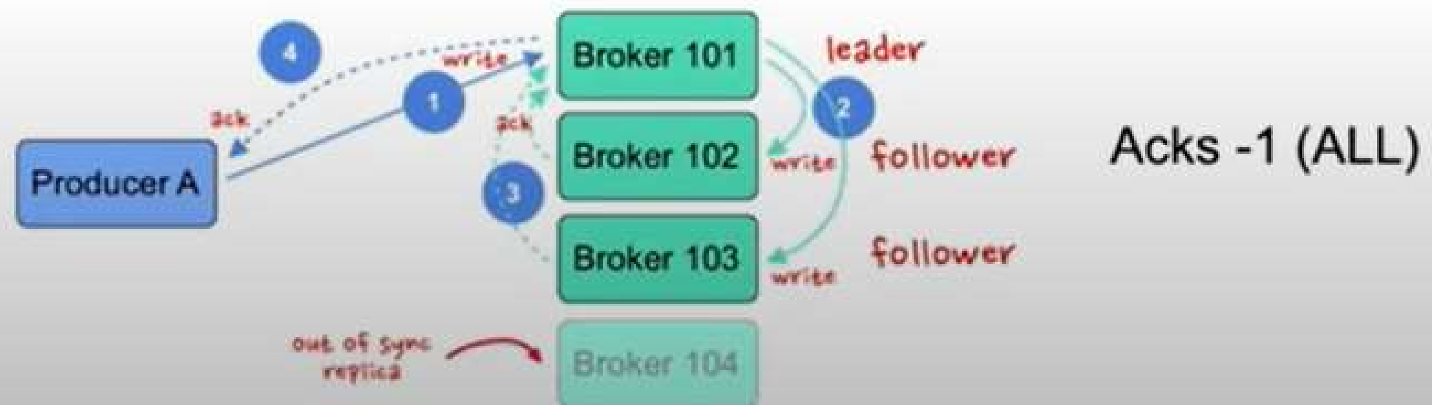
There are two behaviors that could be implemented:

- Wait for a replica in the ISR to come back to life and choose this replica as the leader (hopefully it still has all its data).

- Choose the first replica (not necessarily in the ISR) that comes back to life as the leader.

This is a simple tradeoff between availability and consistency.

Producer guarantees



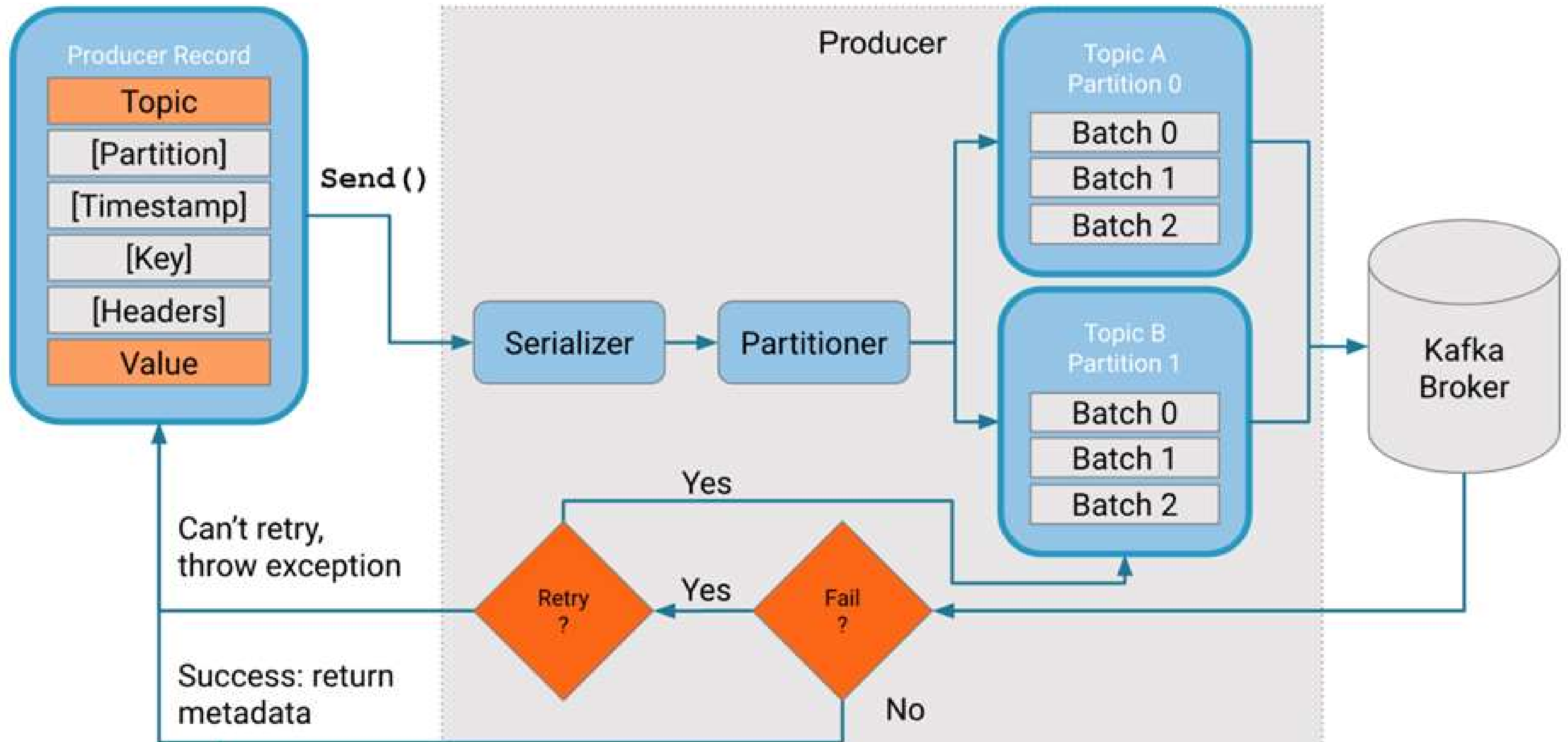
Producer guarantees

'acks=0'
With a value of 0, the producer won't even wait for a response from the broker. It immediately considers the write successful the moment the record is sent out.

'acks=1'
With a setting of 1, the producer will consider the write successful when the leader receives the record. The leader will immediately respond the moment it receives the record.

'acks=all'
When set to all, the producer will consider the write successful when all of the in-sync replicas receive the record. This is achieved by the leader broker being smart as to when it responds to the request — it'll send back a response once all the in-sync replicas receive the record themselves.

The Publish primitive



Some configuration parameters

batch.size

- The producer config property batch.size defaults to 16K bytes.
- This is used by the Producer to batch records.
- Batches are per partition.

linger.ms

- You can set this so that the Producer will wait this long before sending if batch size not exceeded.
- This setting allows the Producer to group together any records that arrive before they can be sent into a batch.

compression.type

- Setting this allows the producer to compresses request data.
- This setting can be set to none, gzip, snappy, or lz4.

ProducerRecord
Topic
Partition
Timestamp
Message Key
Message Value

Kafka Producer

send()
serializer
key.serializer
value.serializer
partitioner
Hash Key Partitioning(Key is not Null)
Round Robin Partitioning(Key is Null)

Buffer
Topic A , Partition 0
batch
Message1
Message2
Message3

Buffer
Topic A , Partition 1
batch
Message1
Message2
Message3

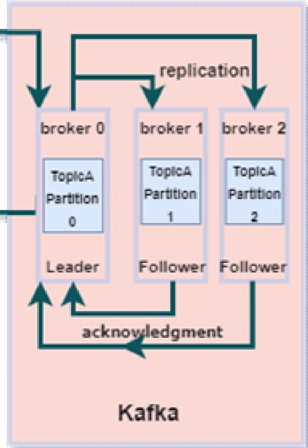
Buffer
Topic A , Partition 2
batch
Message1
Message2
Message3

buffer.memory
max.block.ms
batch.size
compression.type
linger.ms

I/O Threads
max.in.flight.requests.per.connection

retry ?
Yes
No
retries
request.timeout.ms
delivery.timeout.ms
retries.backoff.ms

Failed ?
Yes
No
response
acknowledgment
acks



Exception

Record Metadata
Partition
Offset
Timestamp

Consumers: Pull vs. Push

Push approach

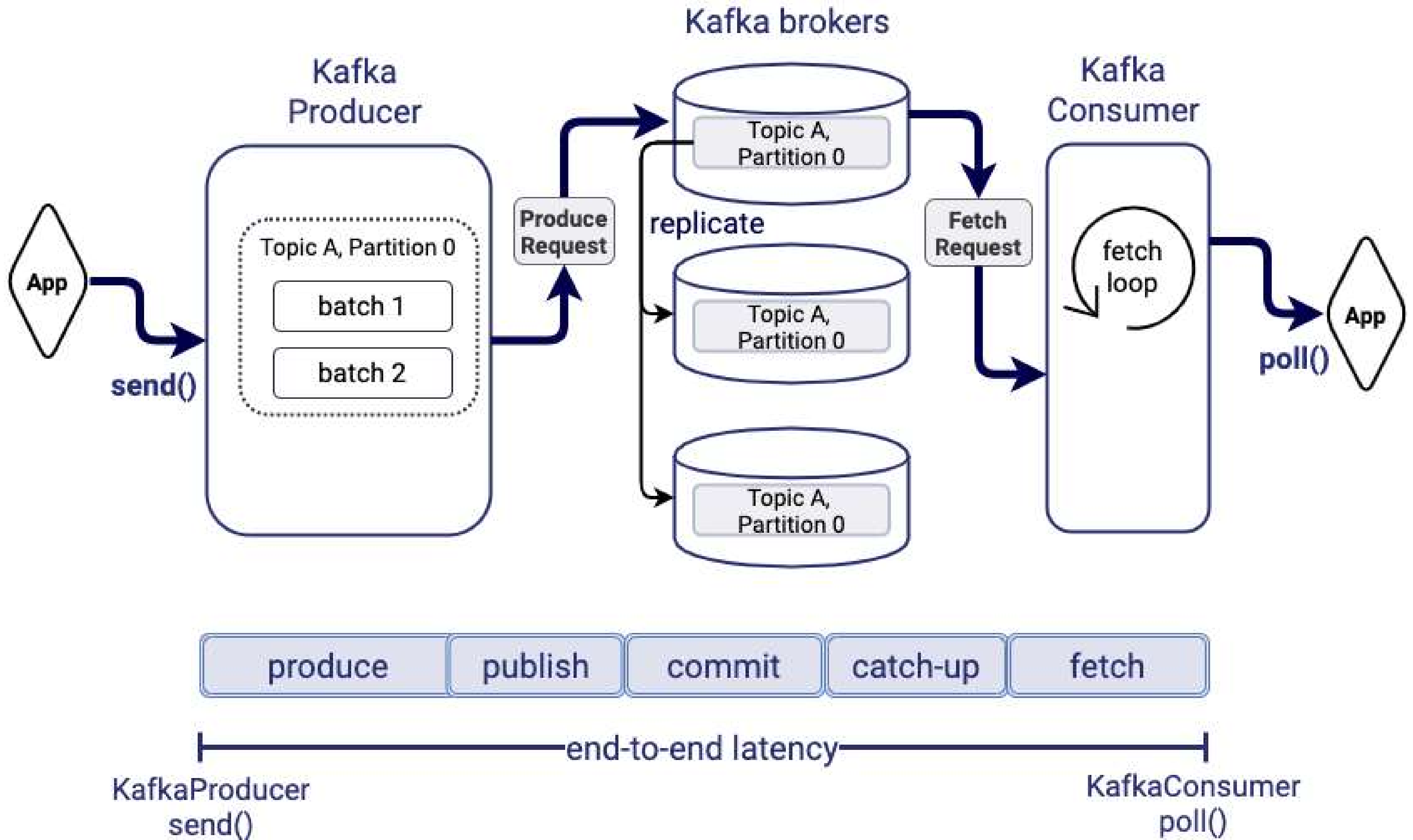
- Flow control needs to be explicit to deal with diverse consumers. Different consumers will consume at different rates, so the broker needs to be aware of this.
- A push-based system must choose to either send a request immediately or accumulate more data and then send it later without knowledge of whether the downstream consumer will be able to immediately process it.
- It is possible to use a **backoff protocol** like additive increase/multiplicative decrease, widely known for its use in TCP congestion control, to optimize utilization.

Consumers: Pull vs. Push

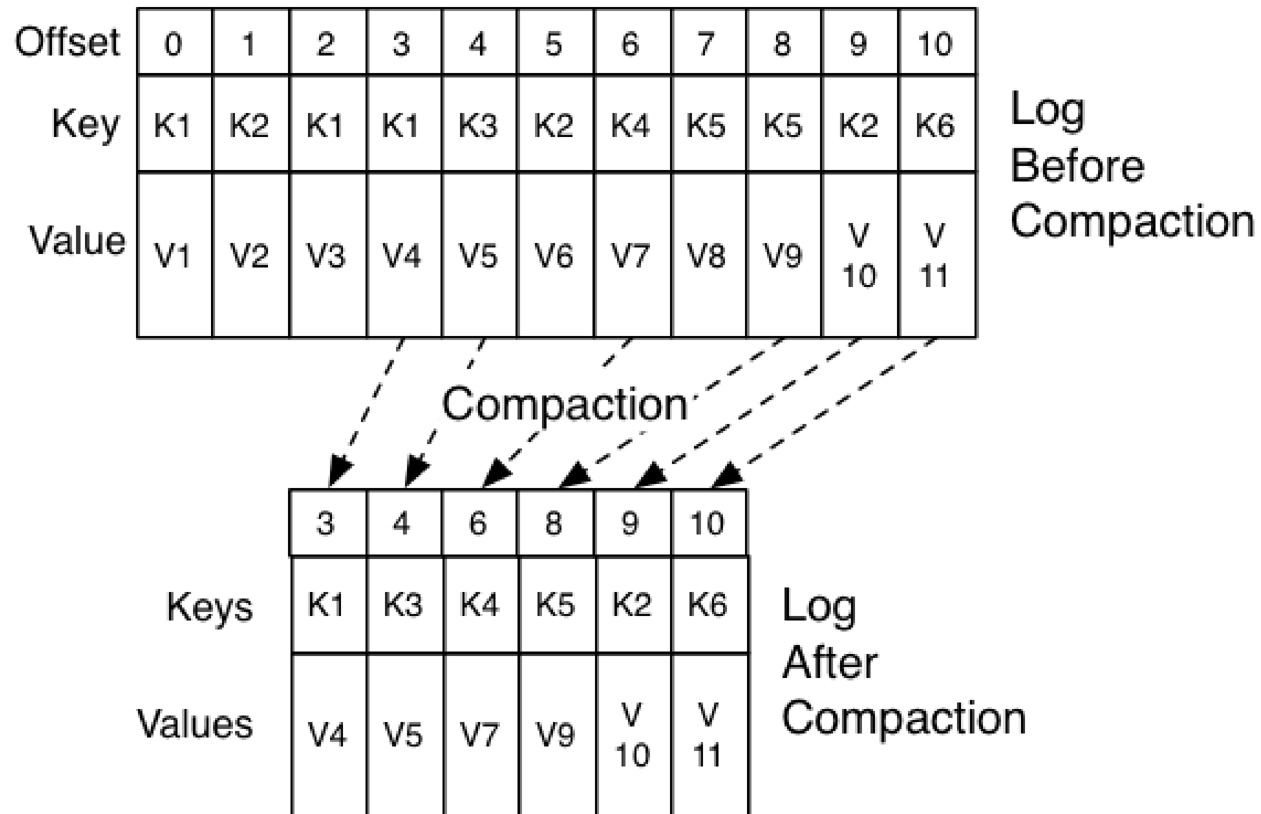
Pull approach

- Flow control is implicit. Consumers simply go at their own pace, and the server doesn’t need to track anything.
- Complex tuning to avoid “busy waiting” and network inefficiency
- **In the end:** Kafka clients tend to be “thick” and have a lot of complexity. That is, they do a lot because the broker is designed to be simple.

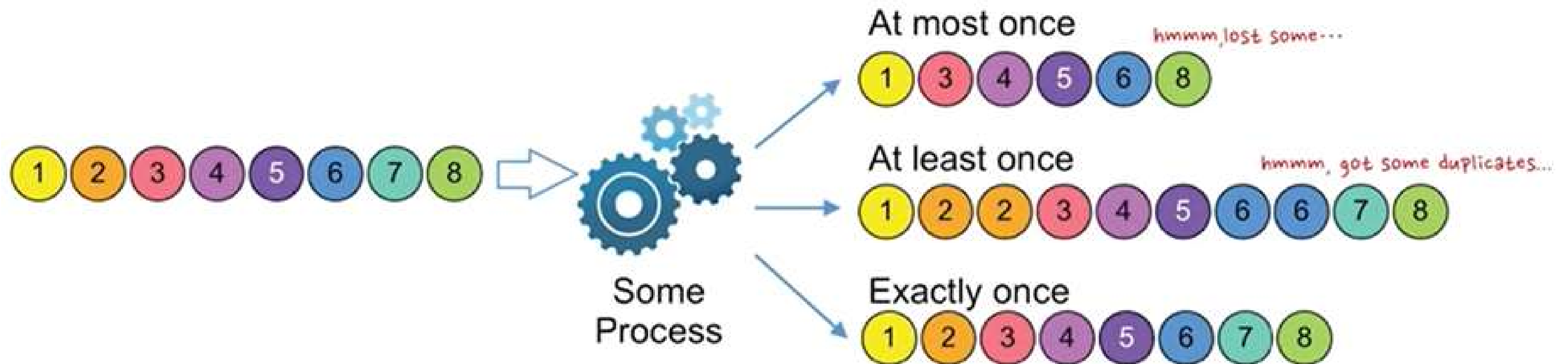
The End-2-End path



Log Compaction



Consumers: Delivery guarantees



Consumers: Updating the Offset

Auto commit, using property *enable.auto.commit*. In this case, Kafka shifts offset as soon as it sends batched messages to Consumer and doesn't take care of whether Consumer handled messages or not. ***It may lead to missing messages.***

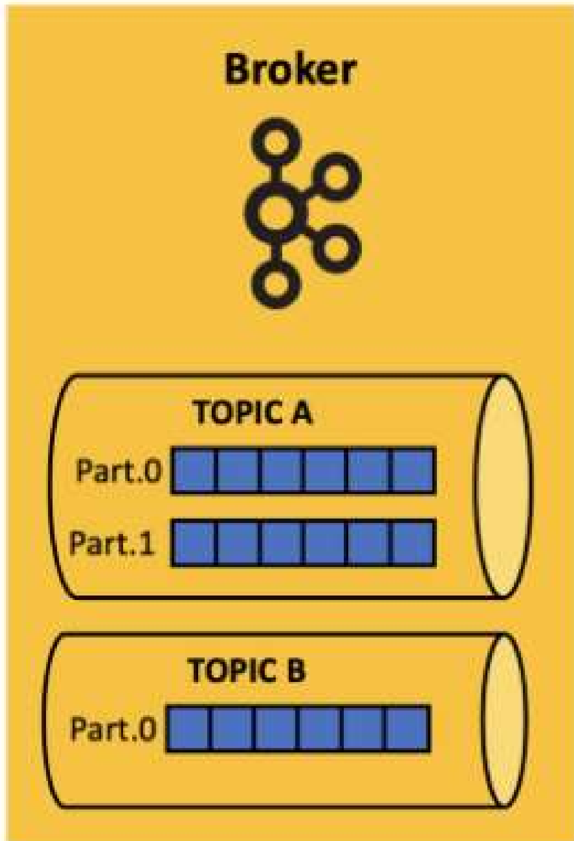
Manual commit, we ask Kafka to change offset explicitly as soon as we are sure that Consumer handles all income messages. In this system, we may get duplicate messages, but if our Consumers handle all messages in an idempotent way it's not an issue at all.

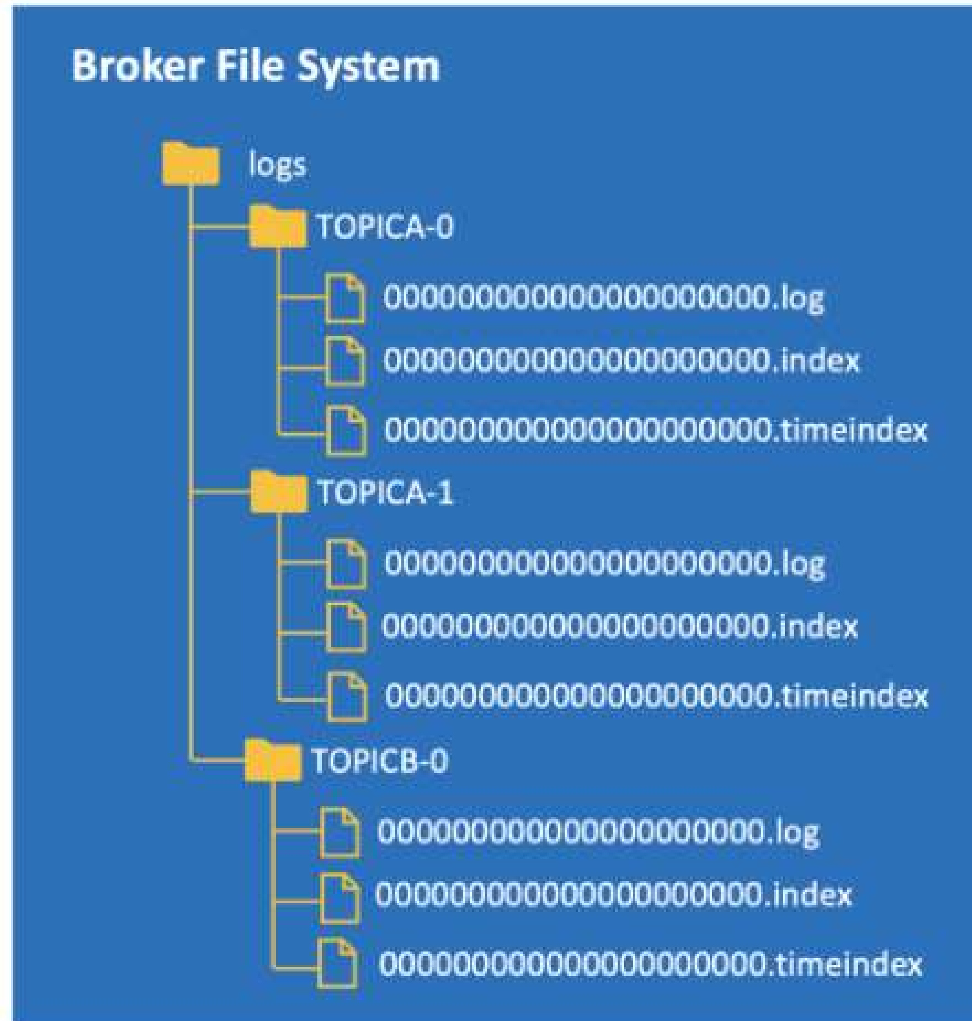
Data Retention Policy

How long do I want to store my data?

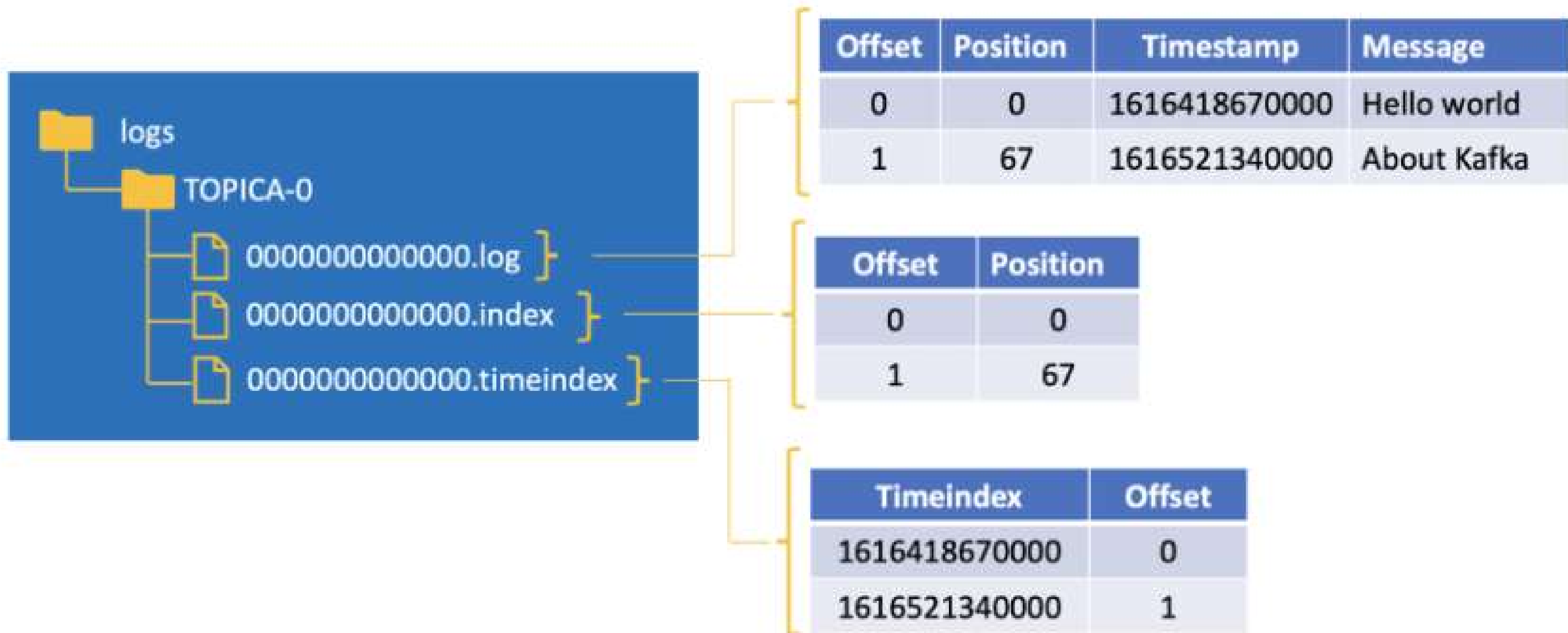
- How long (default: 1 week)
- Set **globally** vs **per topic**
- Business decision (cost factor)

Log implementation on File System





Log implementation on File System

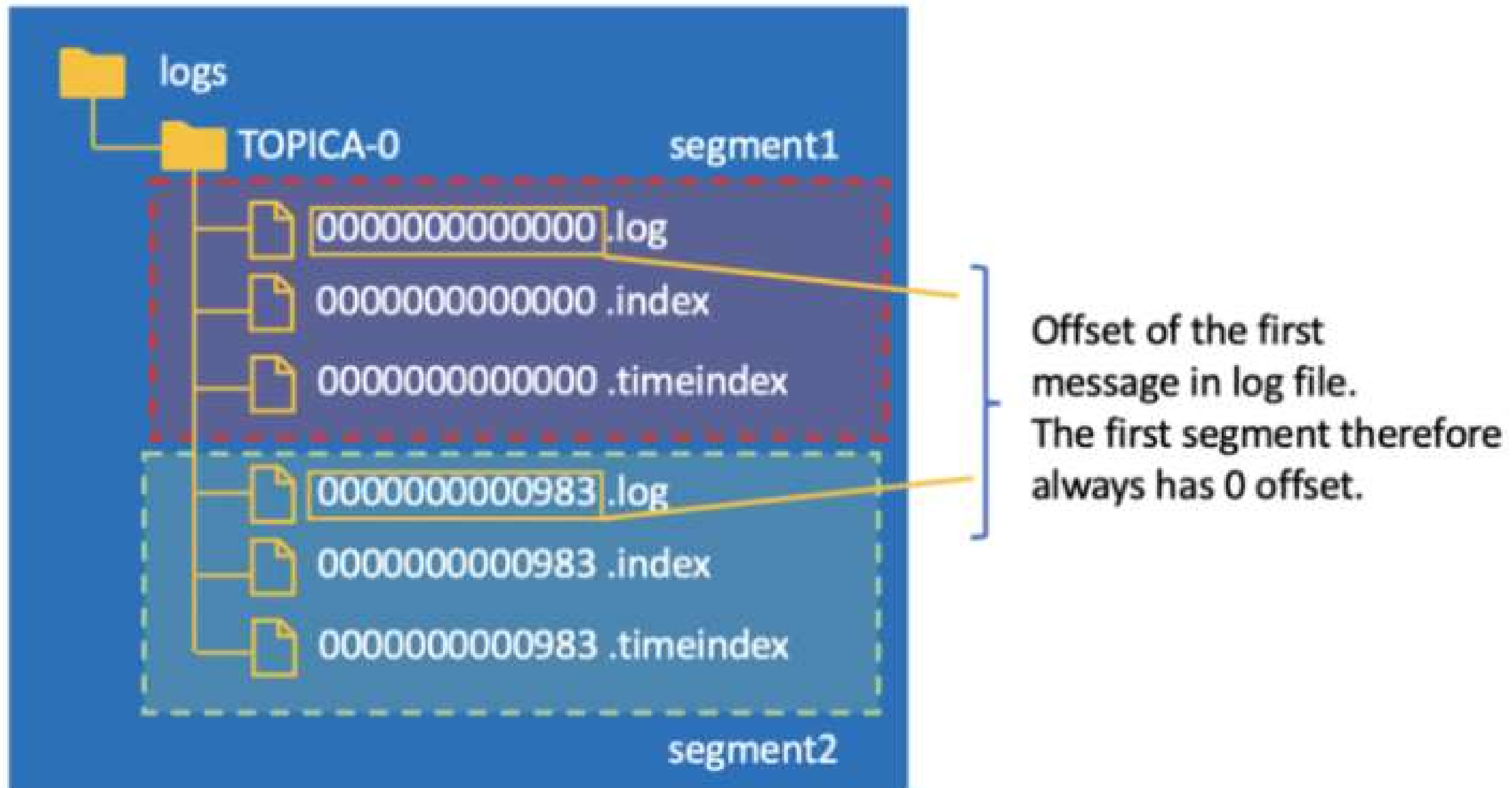


Offset	Position	Timestamp	Message
0	0	1616418670000	Hello world
1	67	1616521340000	About Kafka

Offset	Position
0	0
1	67

Timeindex	Offset
1616418670000	0
1616521340000	1

Log implementation on File System



References

https://medium.com/event-driven-utopia/understanding-kafka-topic-partitions-ae40f80552e8

https://sookocheff.com/post/kafka/kafka-in-a-nutshell/

https://medium.com/geekculture/essential-kafka-overview-with-pictures-bffd84c7f6ac

https://www.baeldung.com/ops/kafka-docker-setup

http://cloudurable.com/blog/kafka-tutorial-kafka-producer-advanced-java-examples/index.html