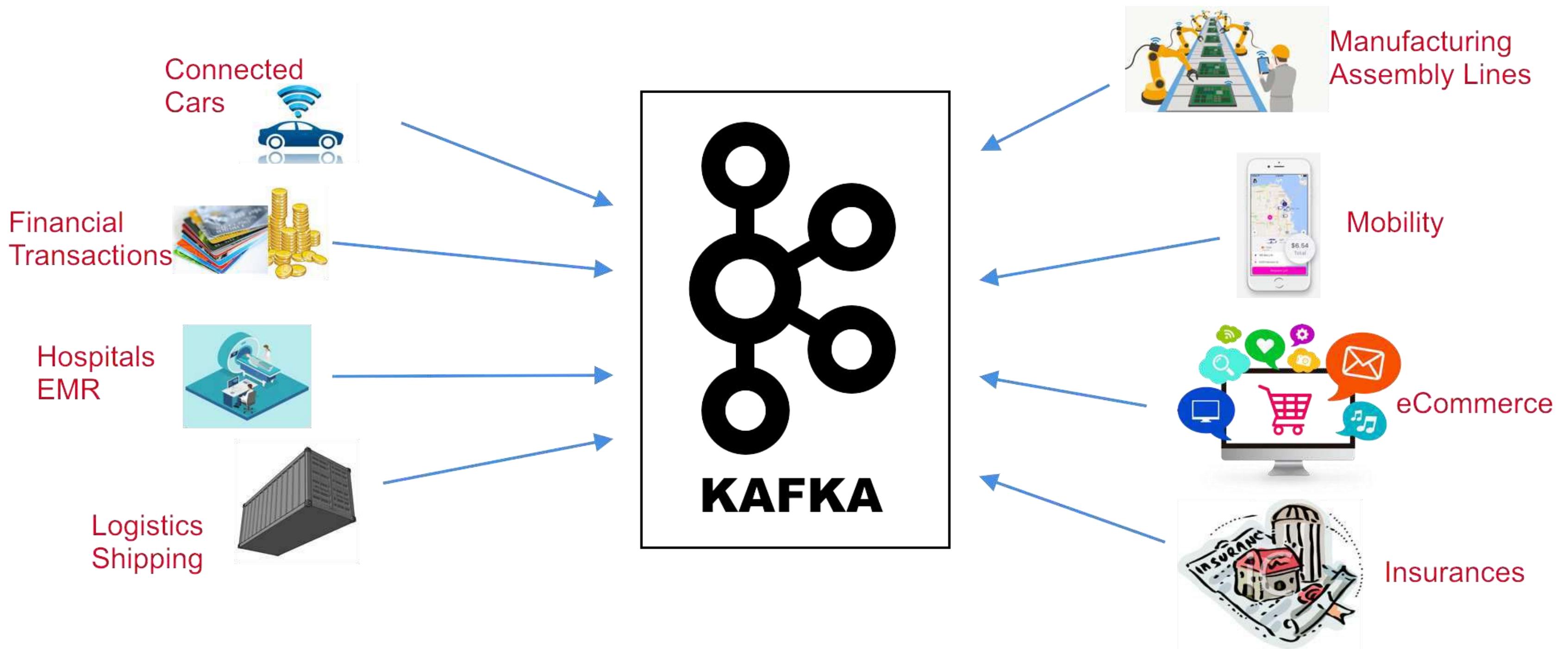




Apache Kafka

Getting Introduced to Kafka and Tips on leveraging it

The World Produces Data



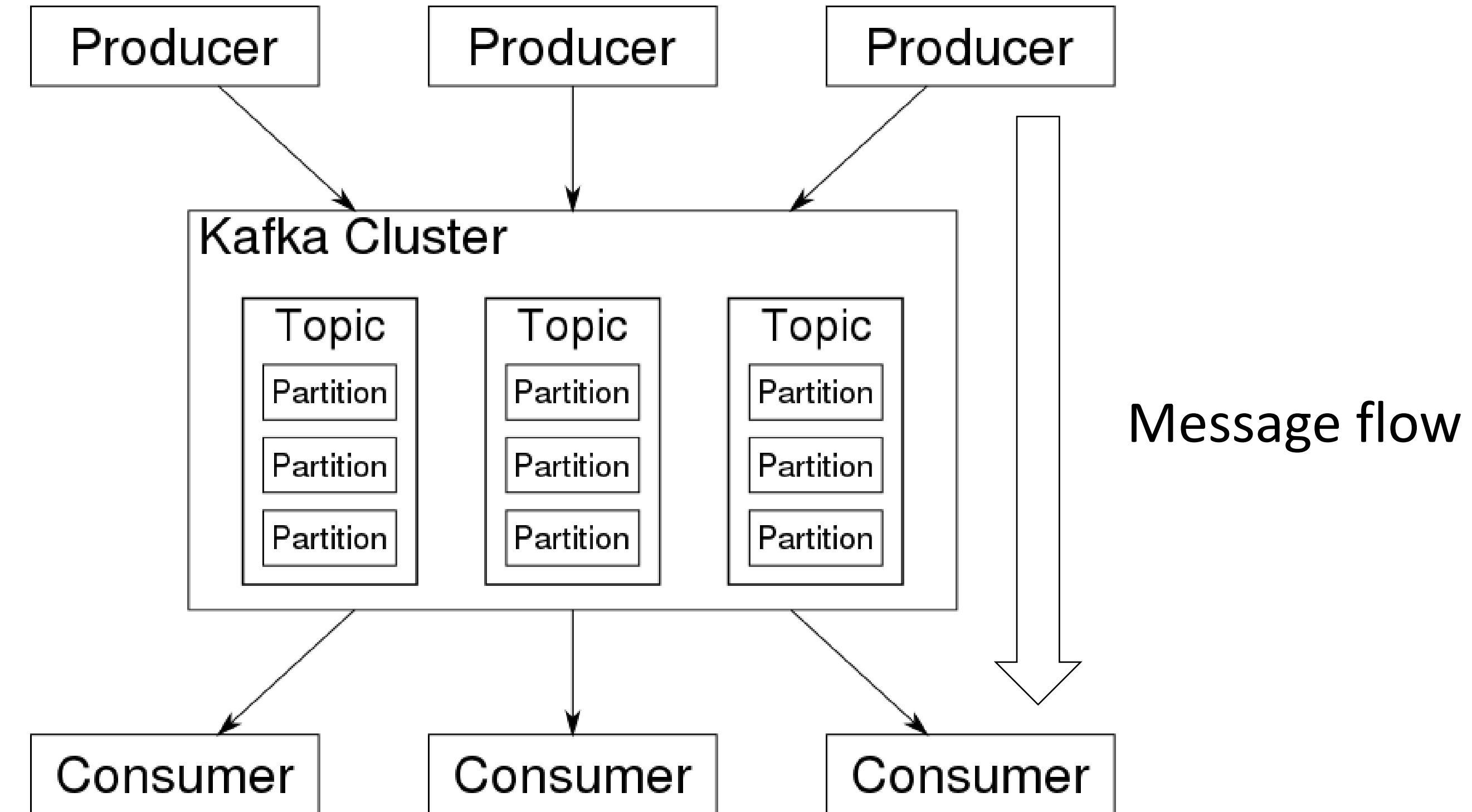
What is Kafka?

Distributed streams
Processing System

Messages sent by
Distributed Producers

to
Distributed Consumers

via
Distributed Kafka Cluster



Kafka benefits

- Fast – high throughput and low latency
- Scalable – horizontally scalable with nodes and partitions
- Reliable – distributed and fault tolerant
- Durable - zero data loss, messages persisted to disk with immutable log
- Open Source – An Apache project
- Available as a Managed Service - on multiple cloud platforms



Why Kafka is Needed?

- Real time streaming data processed for real time analytics
 - Service calls, track every call, IOT sensors
- Apache Kafka is a fast, scalable, durable, and fault-tolerant publish-subscribe messaging system
- Kafka is often used instead of JMS, RabbitMQ and AMQP
 - higher throughput, reliability and replication

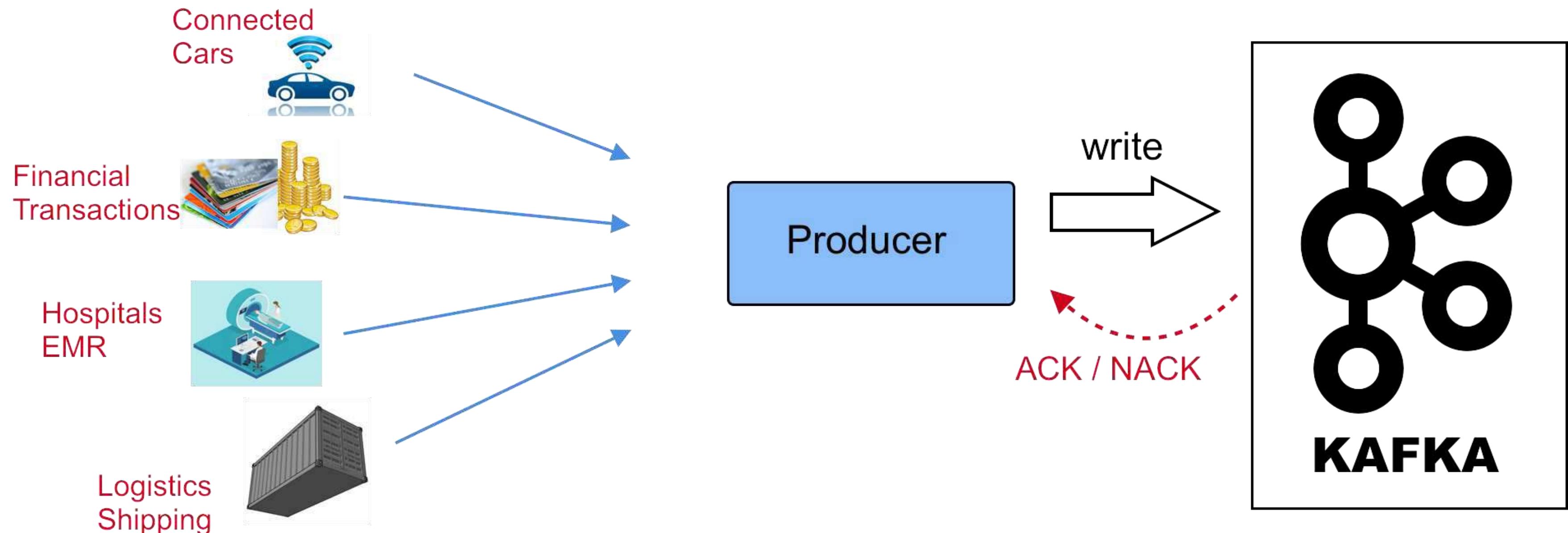
Why is Kafka needed?

- Kafka can work in combination with
 - Flume/Flafka, Spark Streaming, Storm, HBase and Spark for real-time analysis and processing of streaming data
 - Feed your data lakes with data streams
- Kafka brokers support massive message streams for follow-up analysis in Hadoop or Spark
- Kafka Streaming (subproject) can be used for real-time analytics

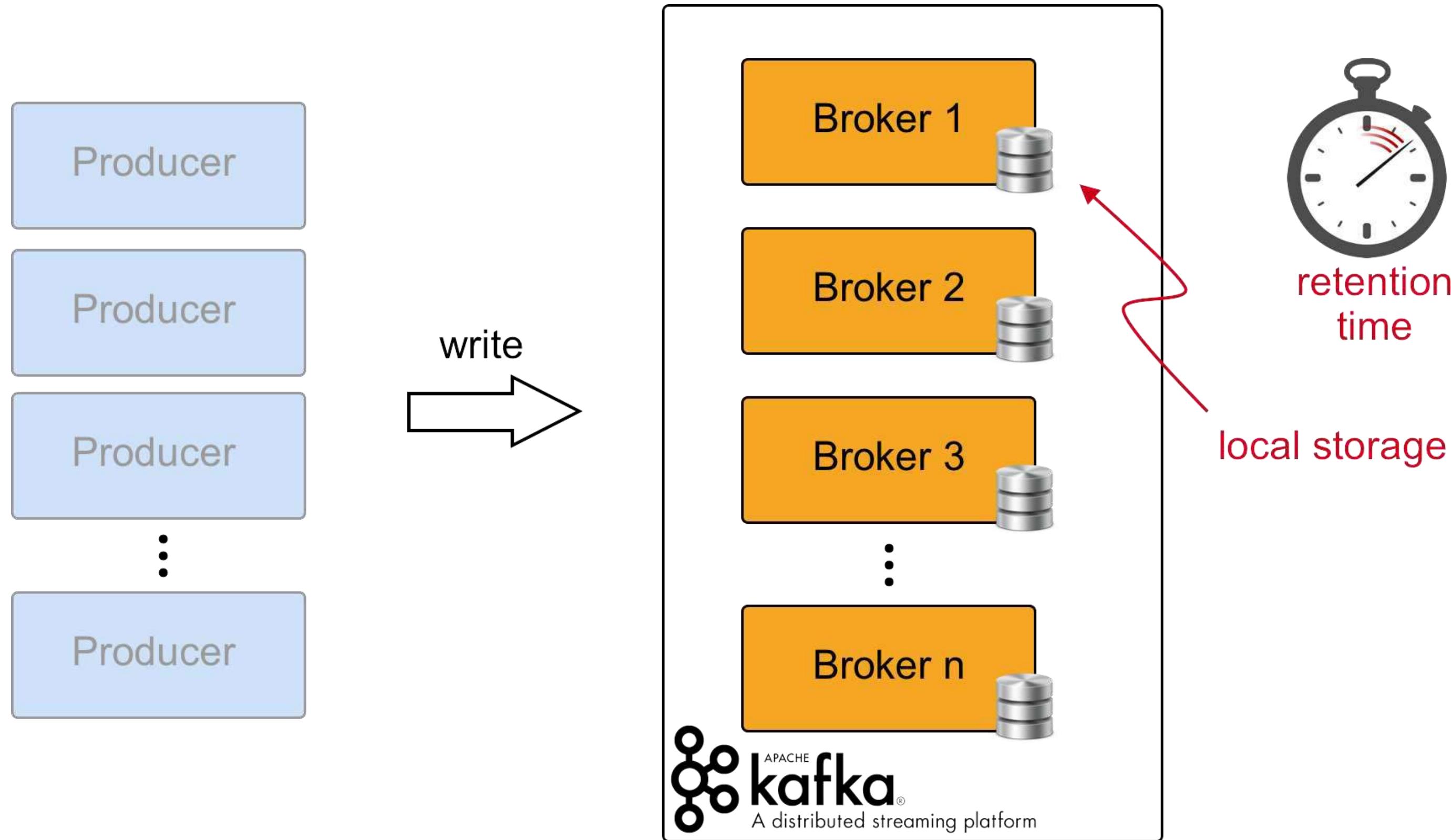
Kafka Use Cases

- Stream Processing
- Website Activity Tracking
- Metrics Collection and Monitoring
- Log Aggregation
- Real time analytics
- Capture and ingest data into Spark / Hadoop
- CRQS, replay, error recovery
- Guaranteed distributed commit log for in-memory computing

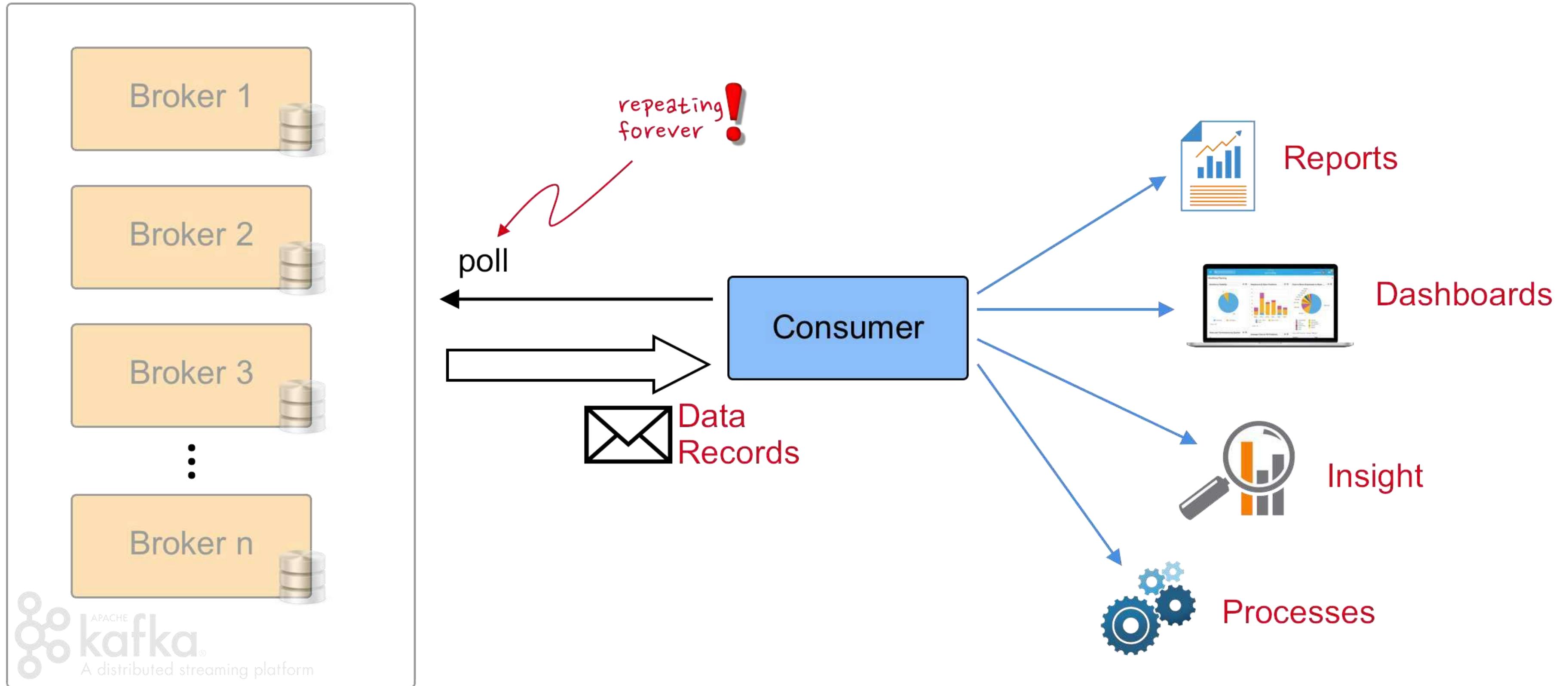
Producers



Brokers

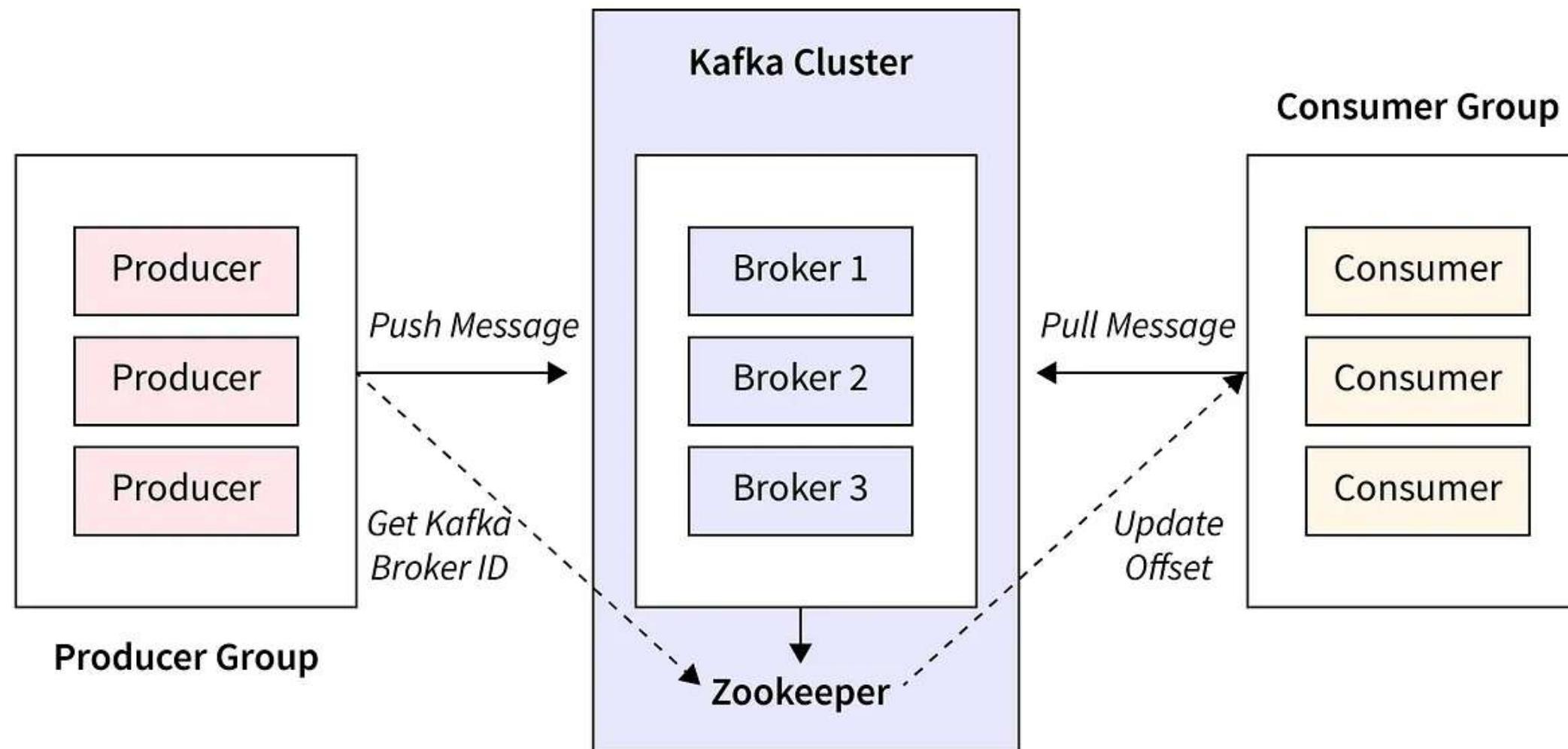


Consumers



Apache KAFKA

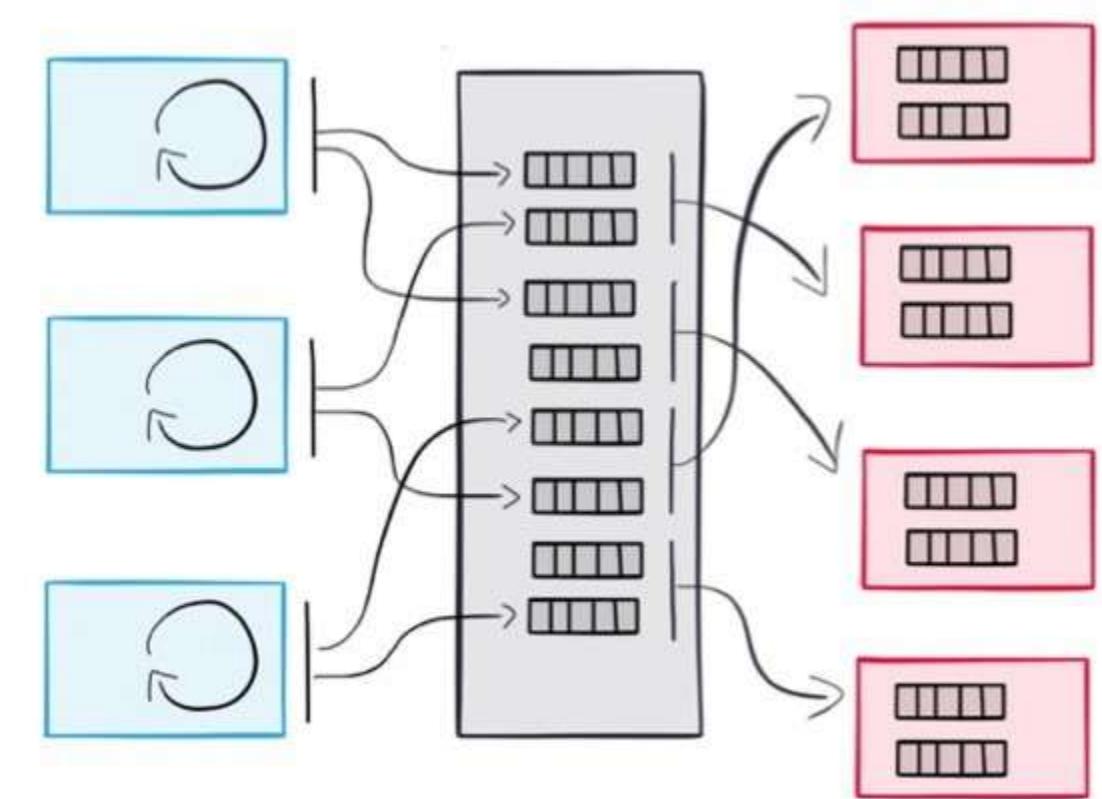
Kafka Ecosystem



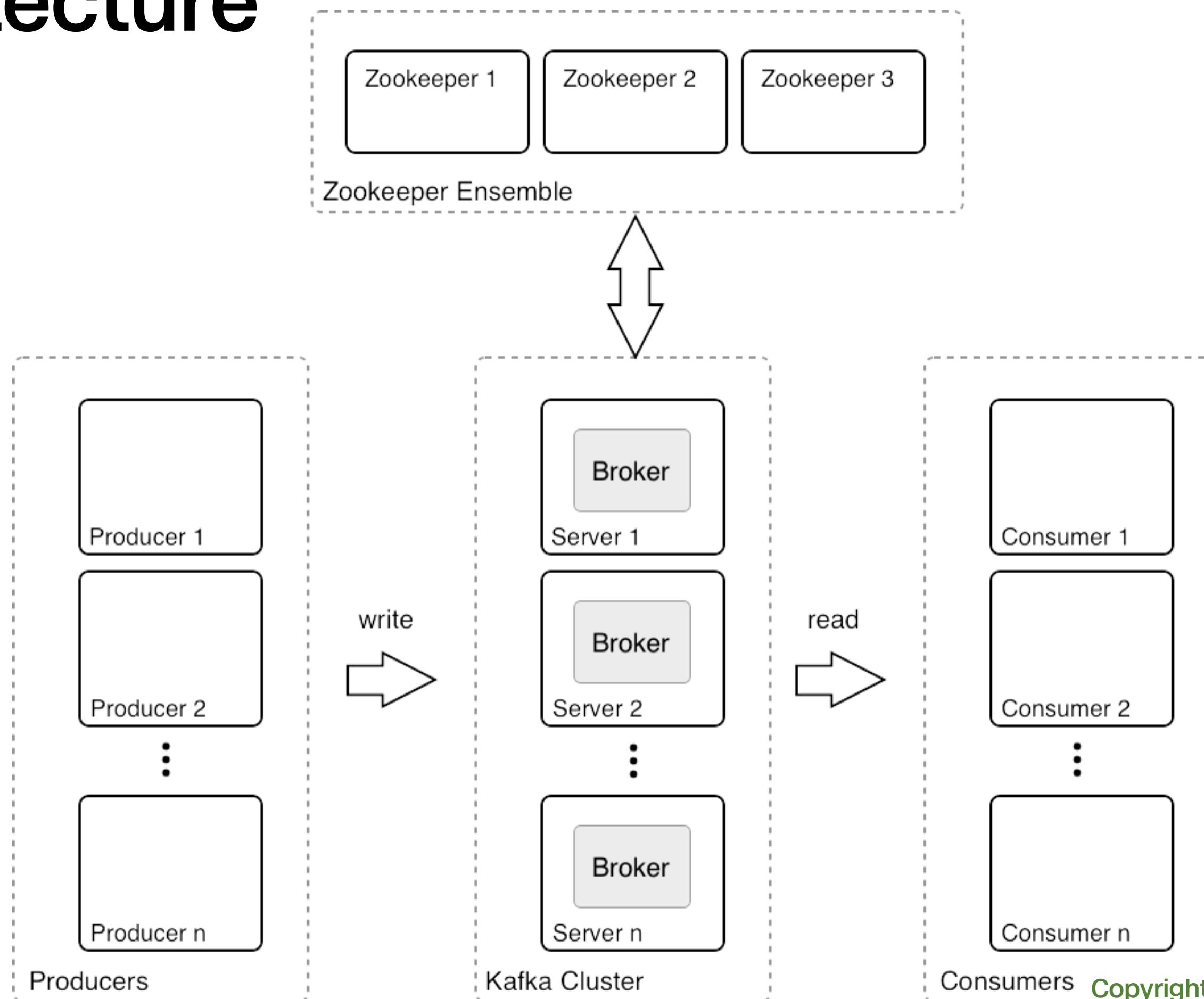
SCALER
Topics

Producers and Consumers are Decoupled

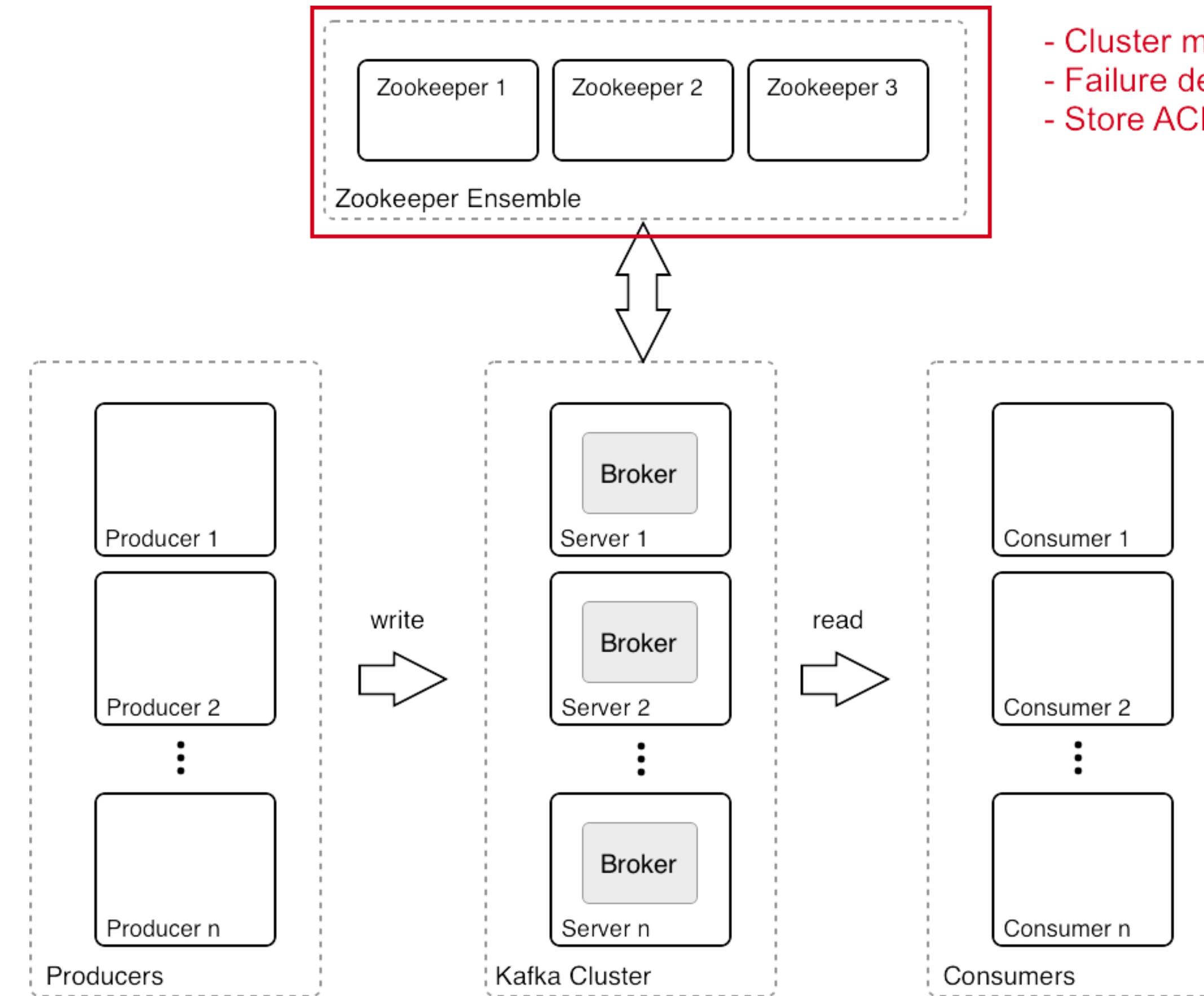
- Producers and Consumers are decoupled
- Slow Consumers do not affect Producers
- Add Consumers without affecting Producers
- Failure of Consumer does not affect System



Architecture



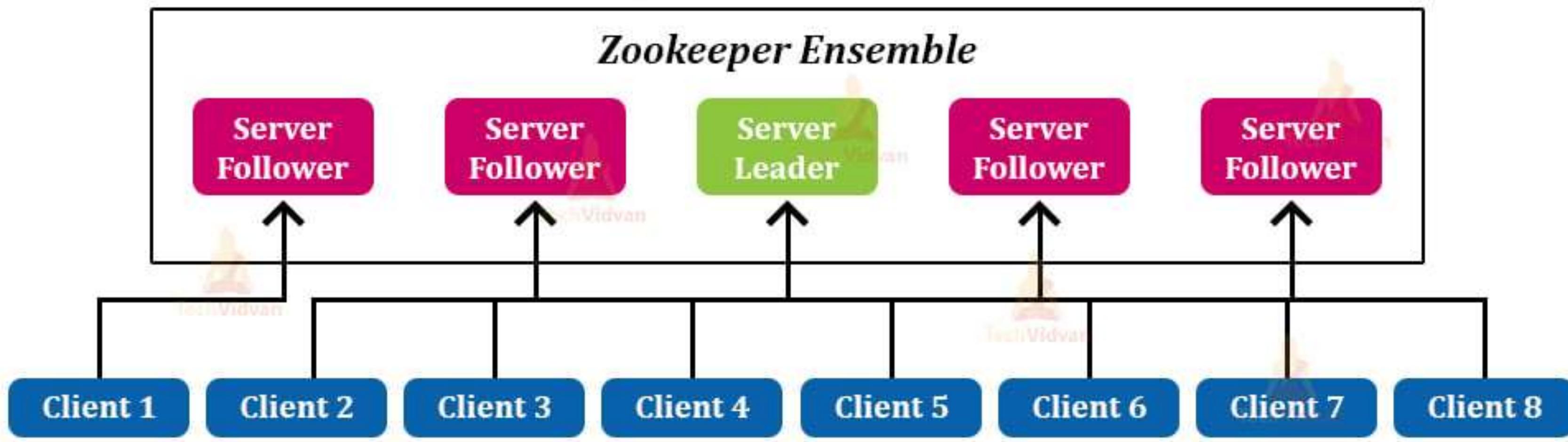
How Kafka Uses ZooKeeper



- Cluster management
- Failure detection & recovery
- Store ACLs & secrets

Apache Kafka

Zookeeper Architecture

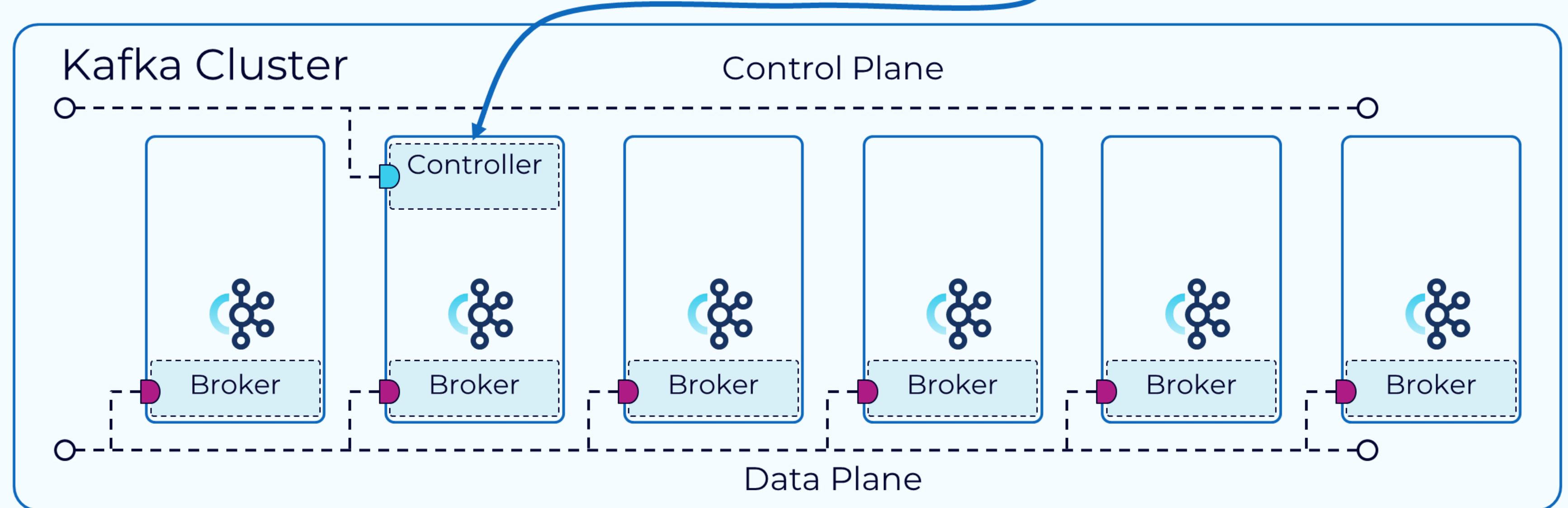
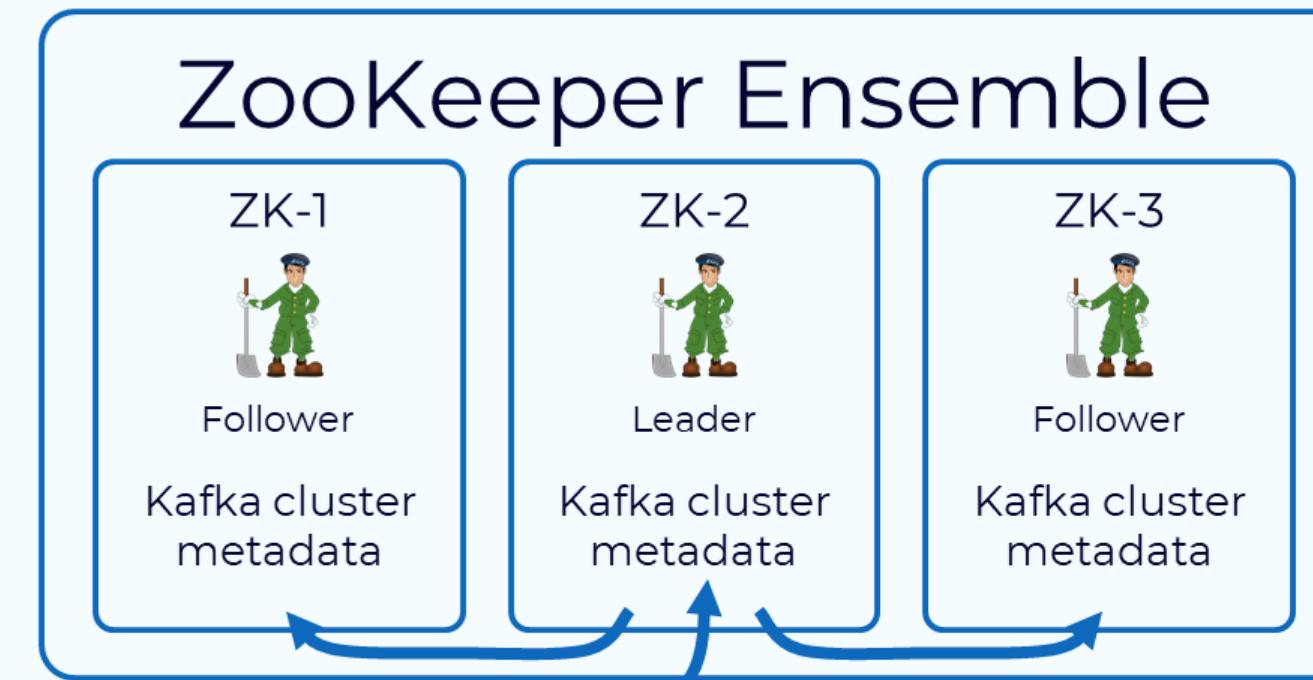


Zookeeper Basics

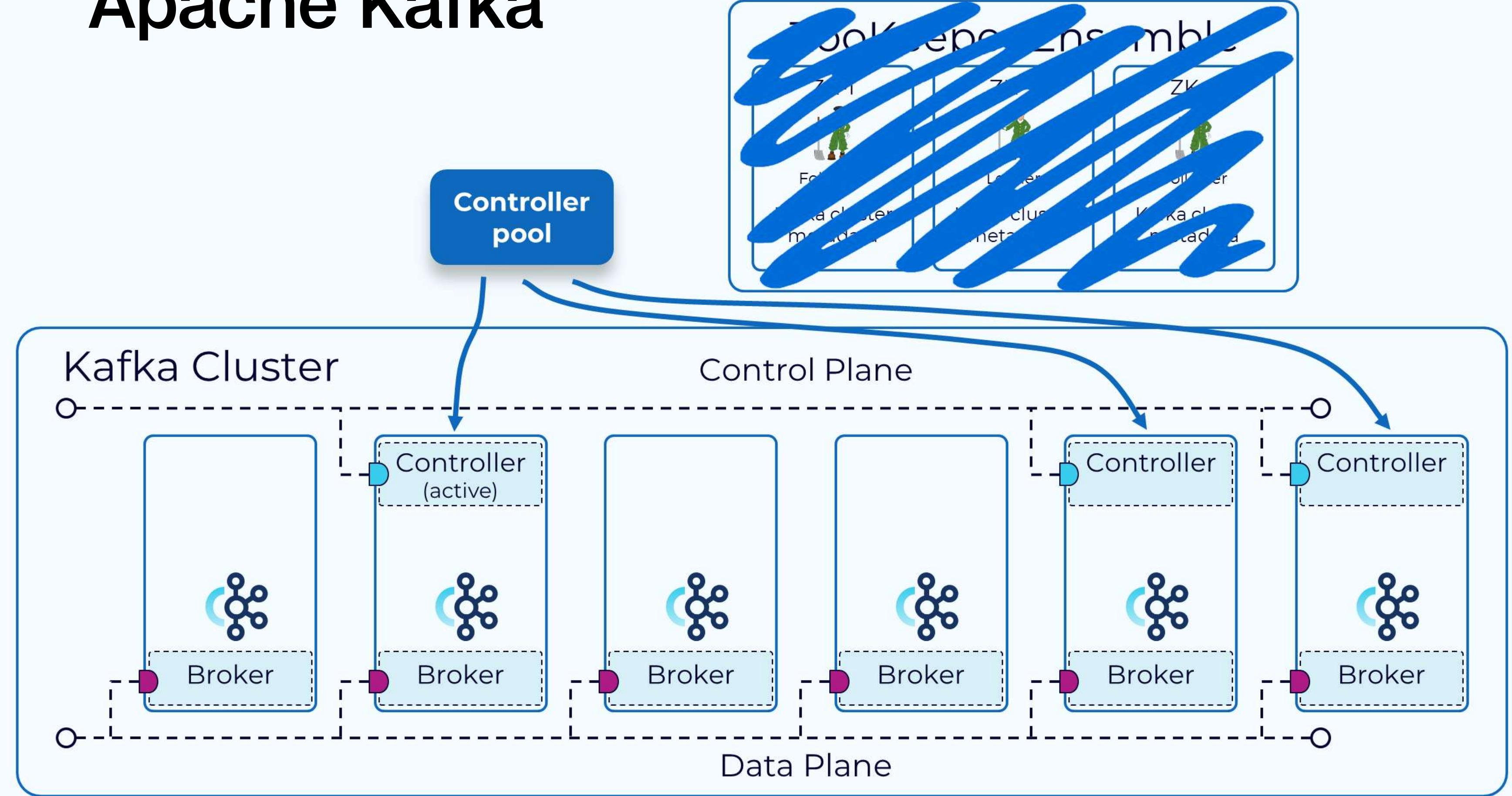
- Open Source Apache Project
- Distributed Key Value Store
- Maintains configuration information
- Stores ACLs and Secrets
- Enables highly reliable distributed coordination
- Provides distributed synchronization
- Three or five servers form an ensemble



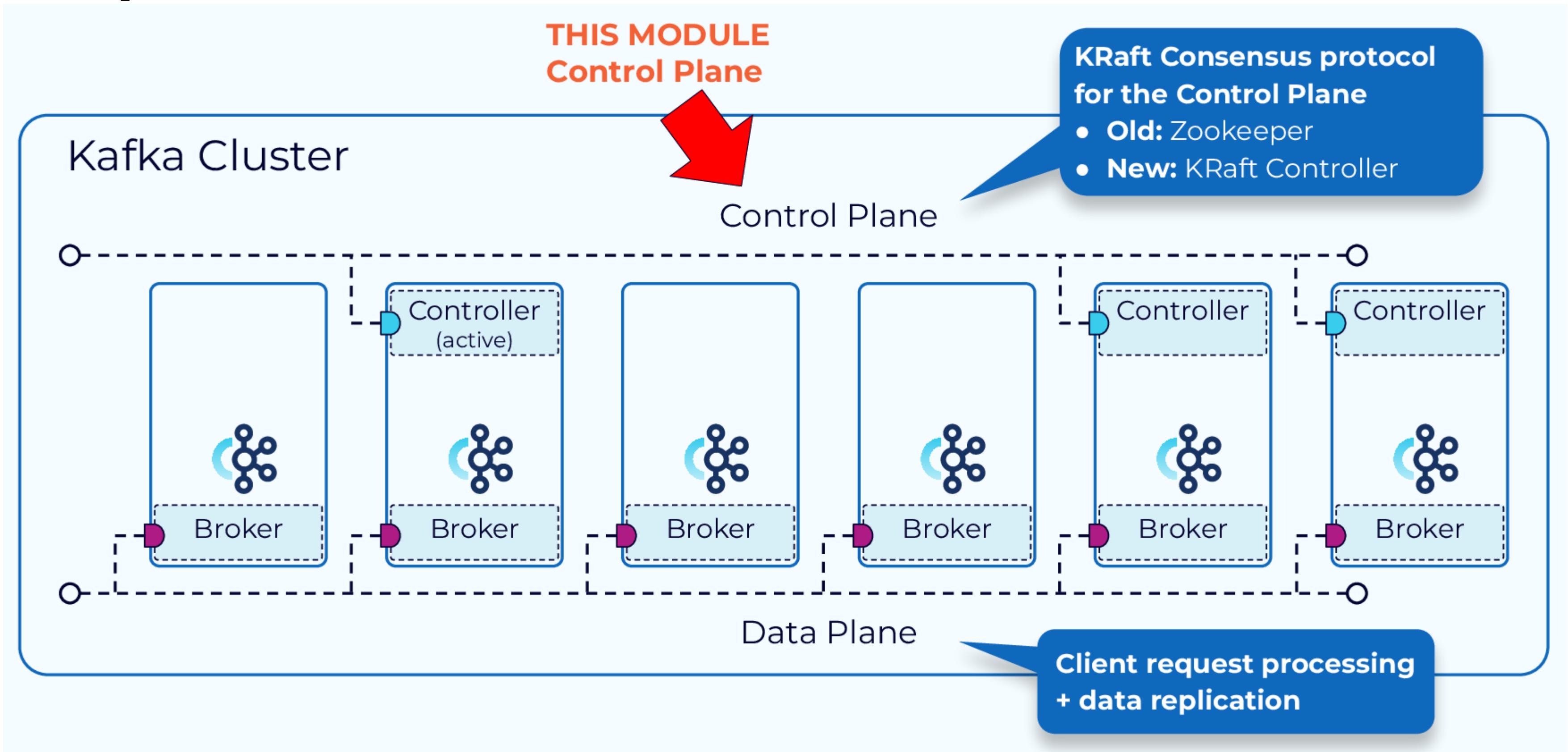
Apache Kafka



Apache Kafka

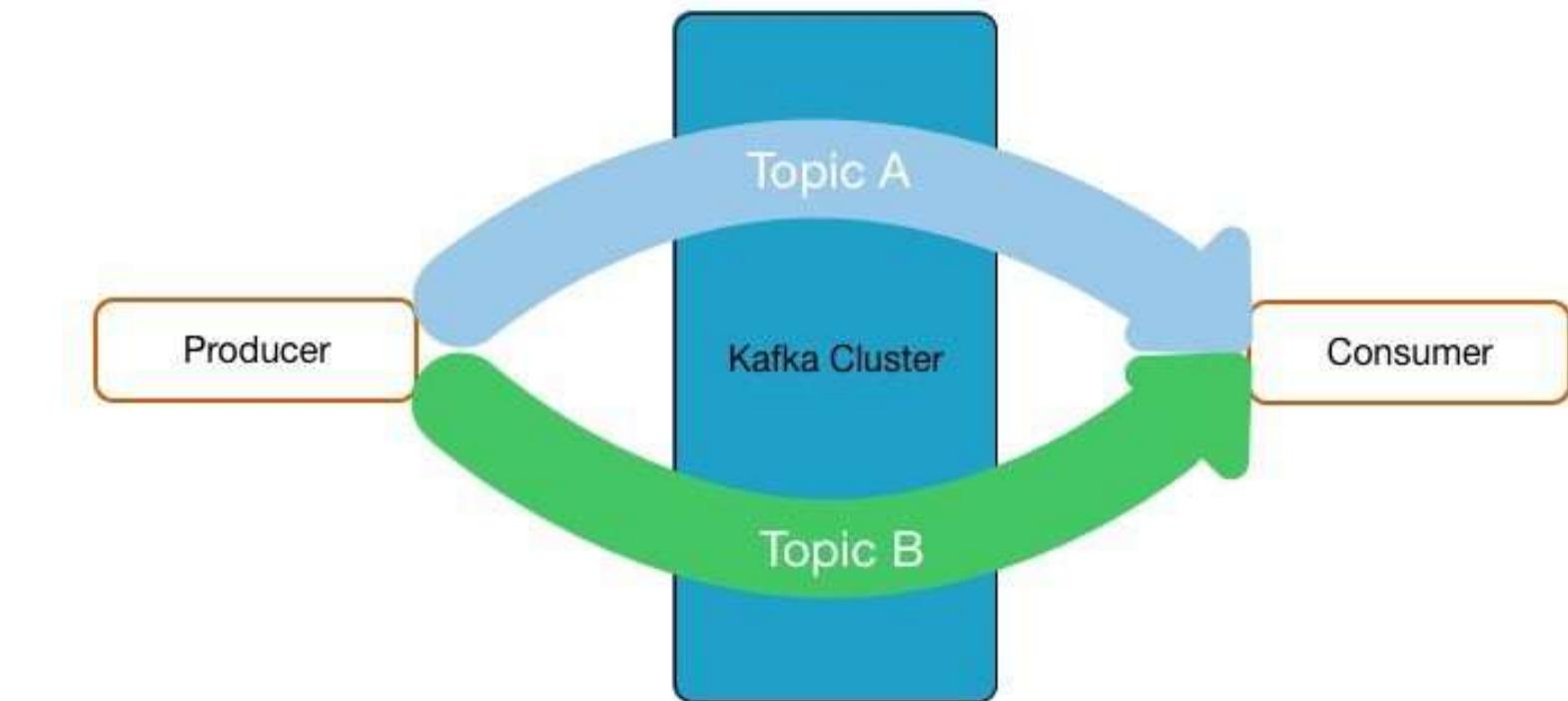


Apache Kafka

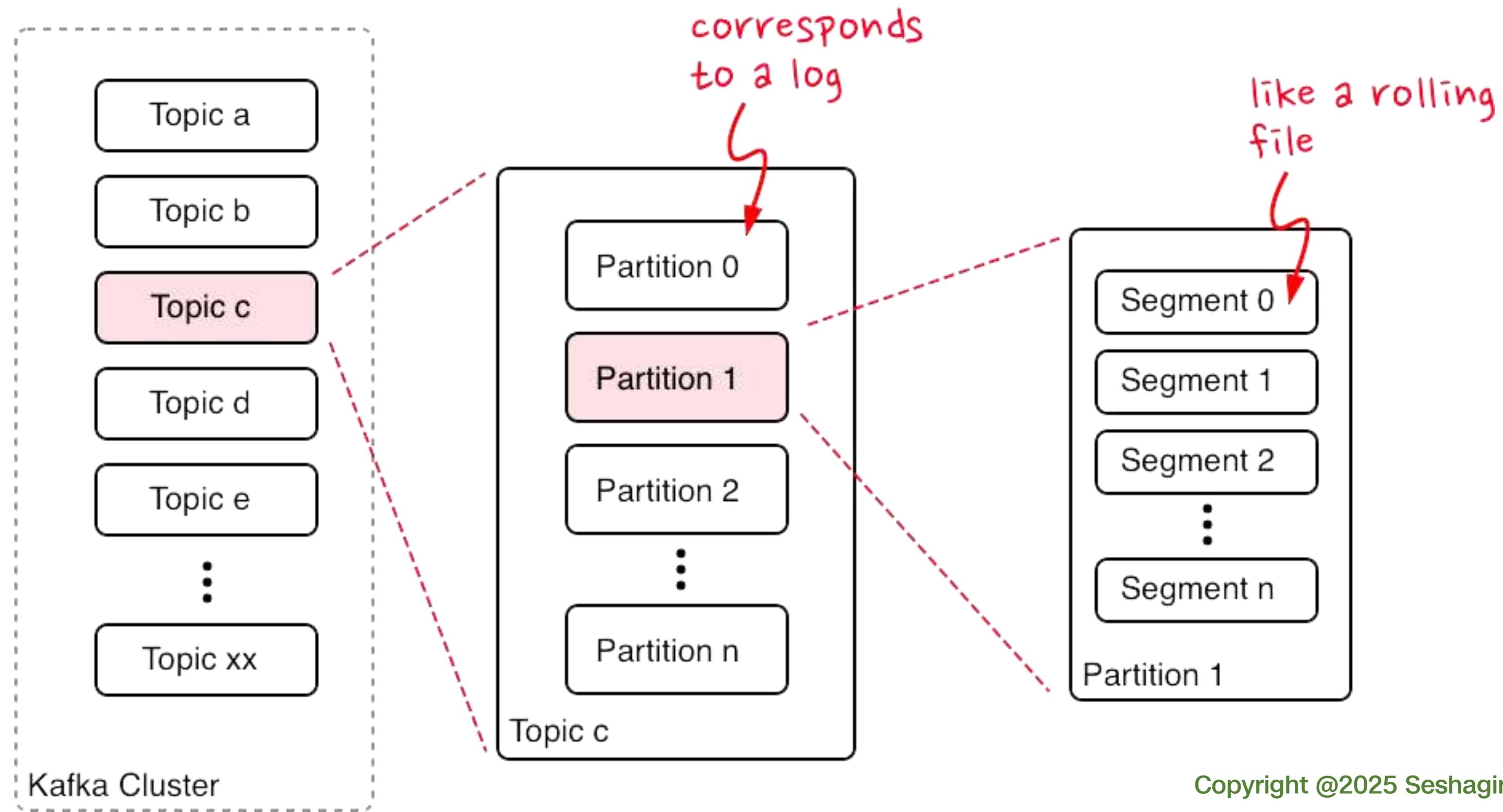


Topics

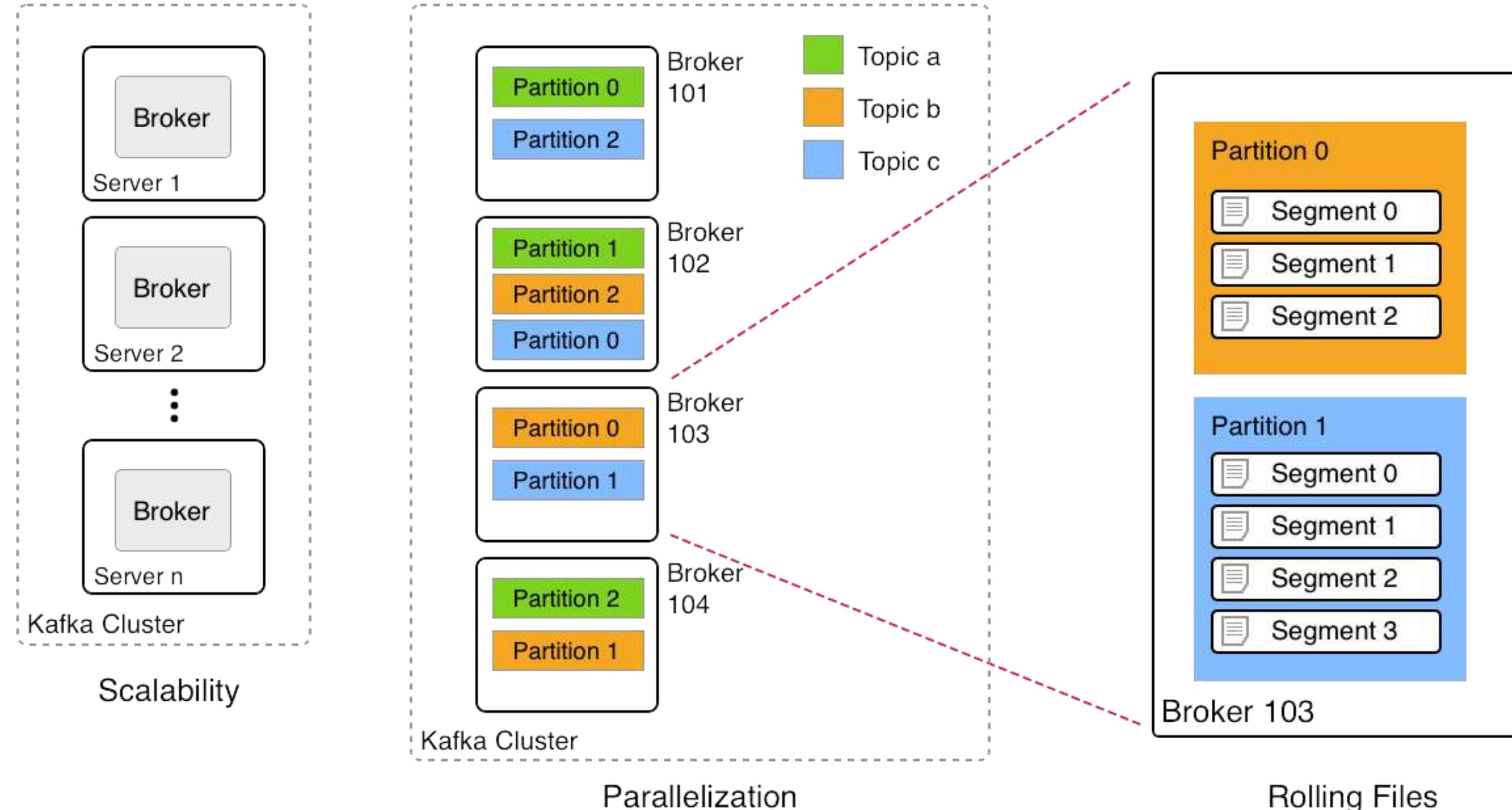
- **Topics:** Streams of “related” Messages in Kafka
 - Is a Logical Representation
 - Categorizes Messages into Groups
- Developers define Topics
- Producer ↔ Topic: N to N Relation
- Unlimited Number of Topics



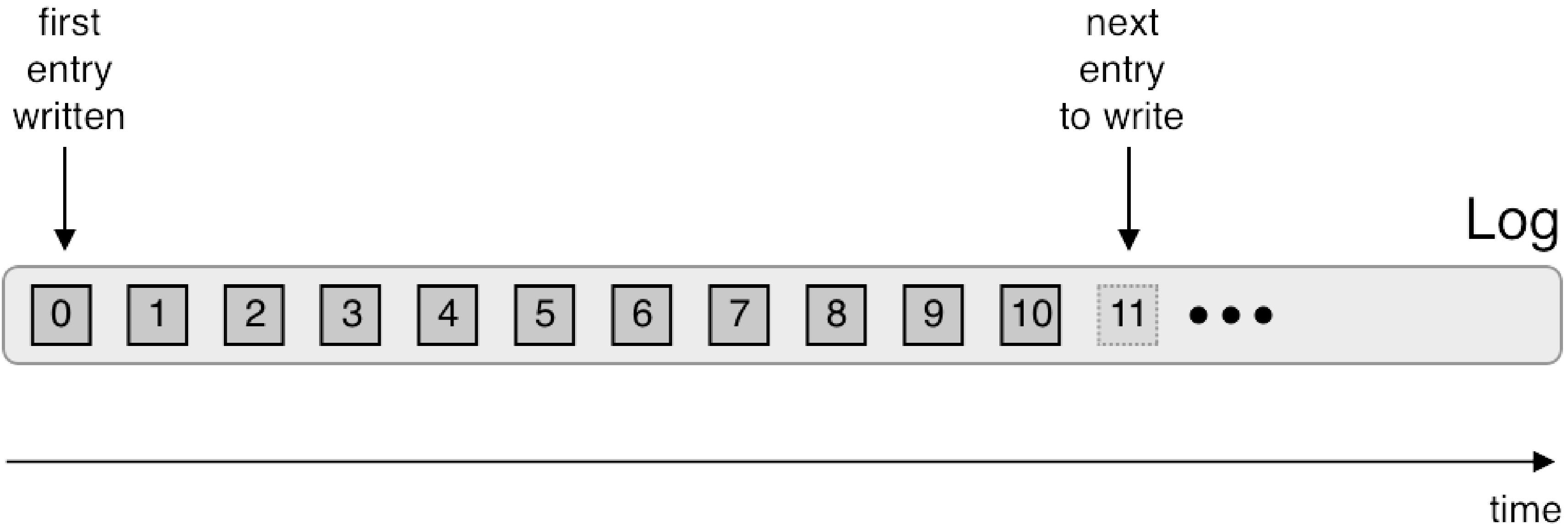
Topics, Partitions and Segments



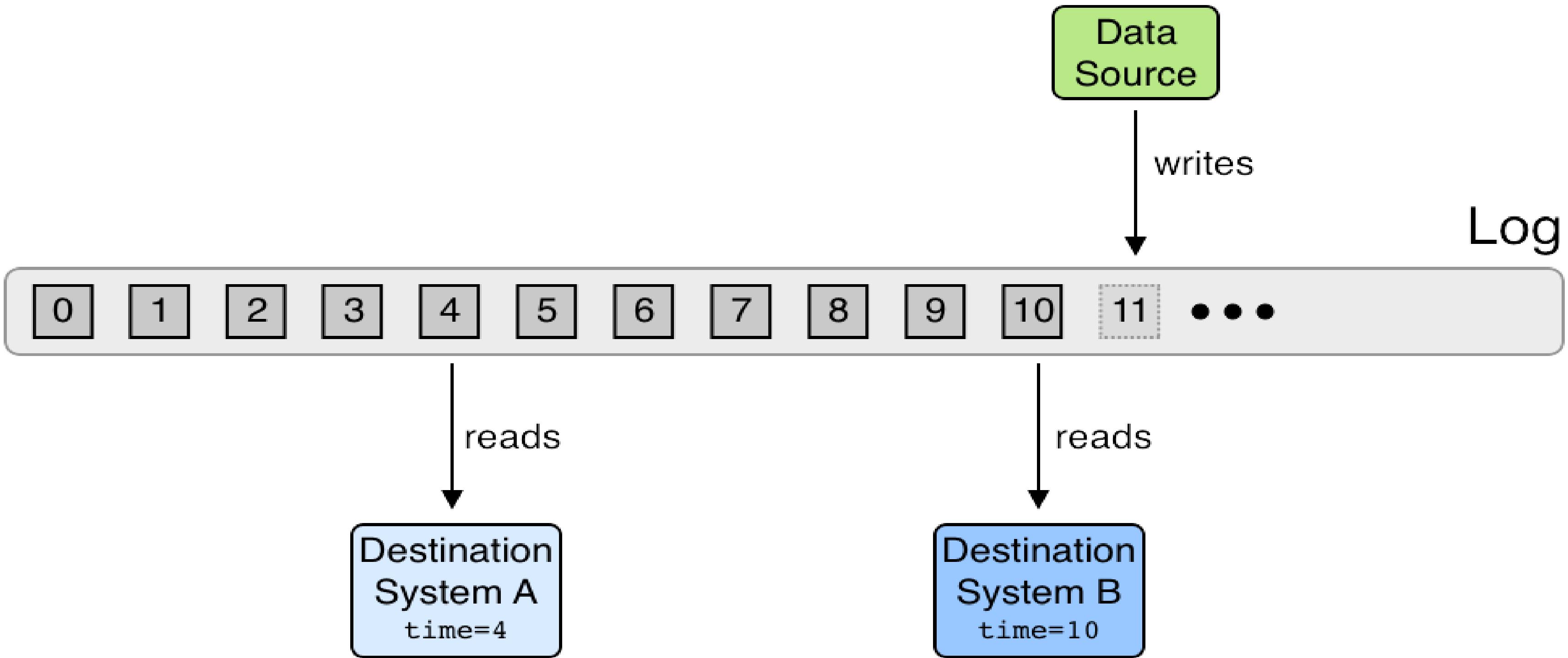
Topics, Partitions and Segments



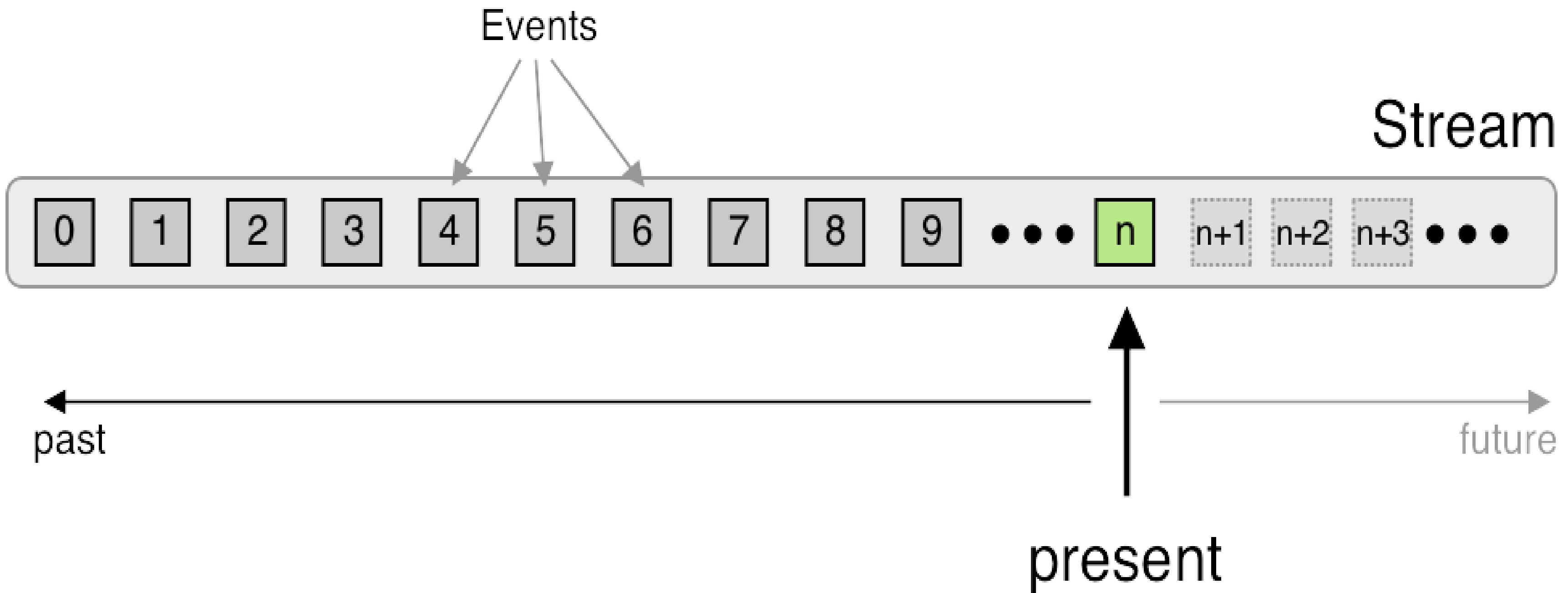
Kafka Logs



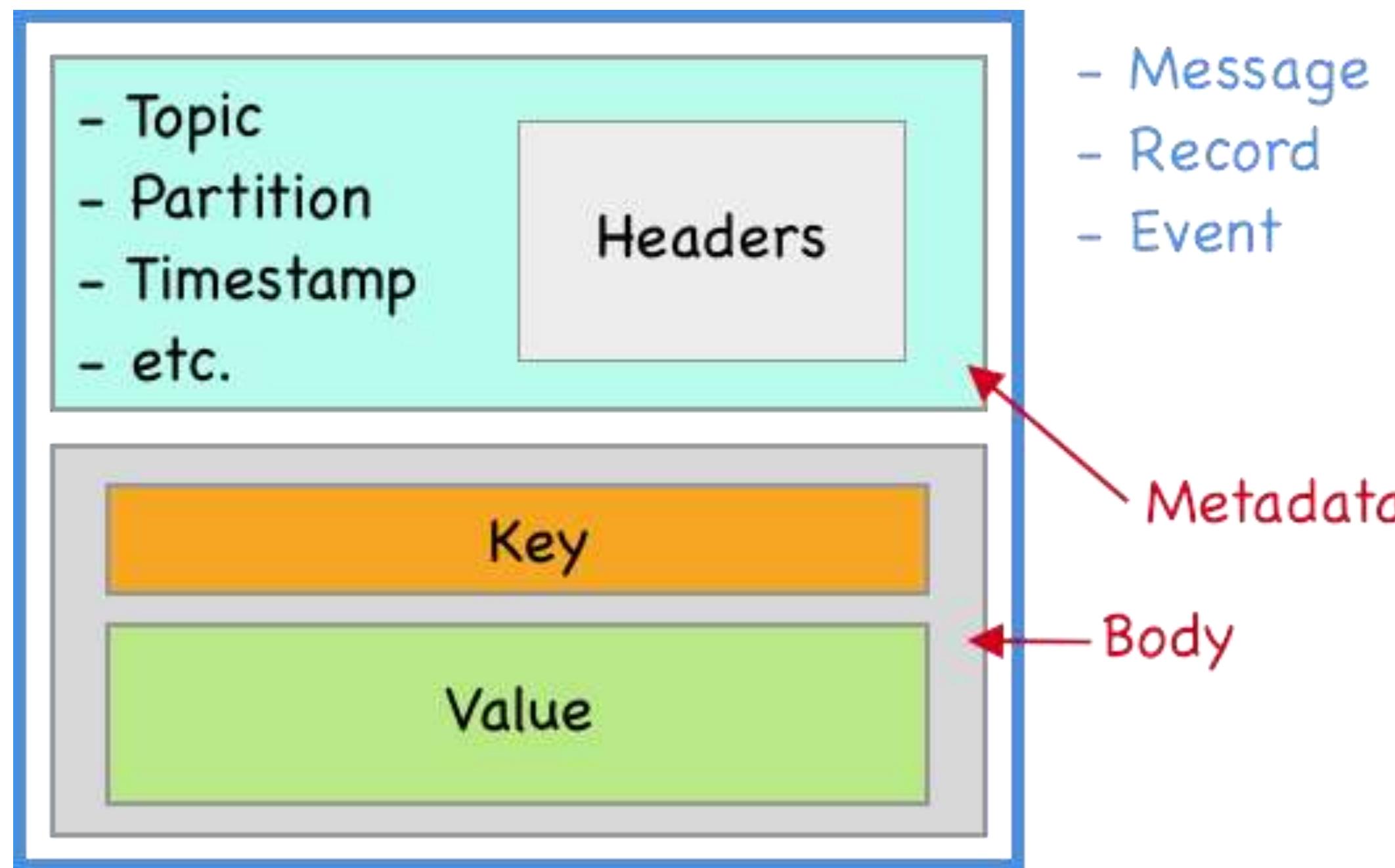
Structured Data Flow



Streams



Data Elements

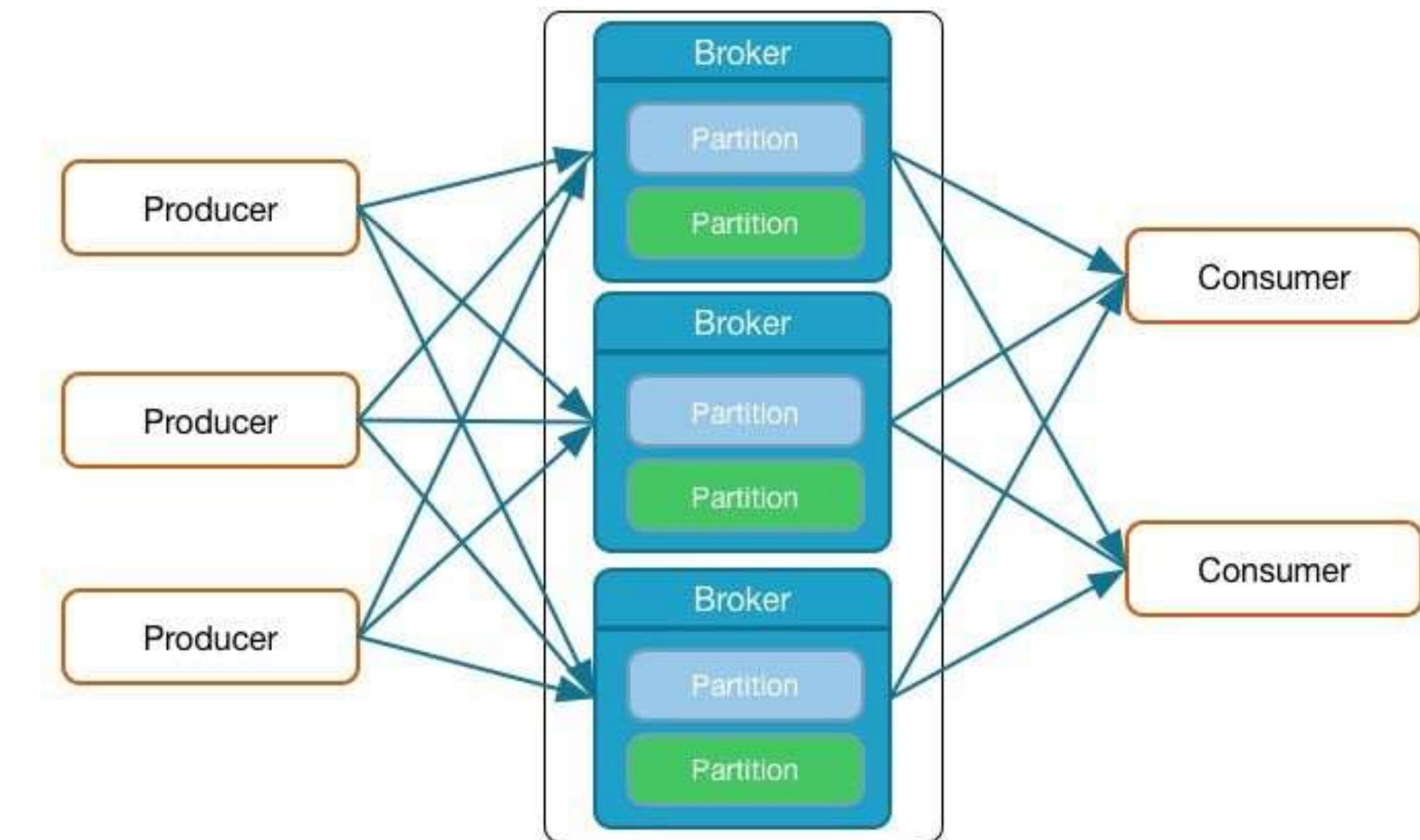


Brokers And Partitions

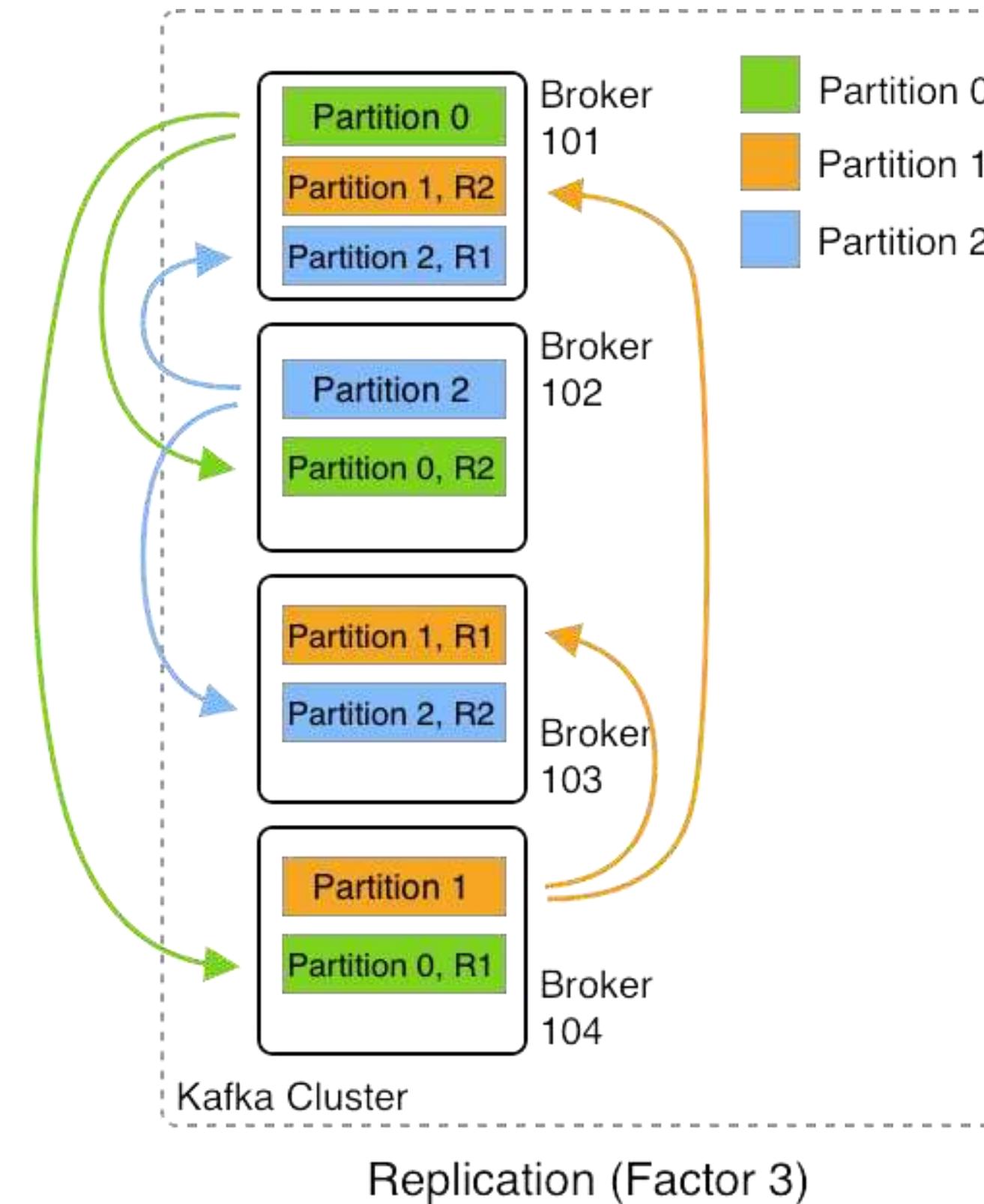
- Messages of Topic spread across Partitions
- Partitions spread across Brokers
- Each Broker handles many Partitions
- Each Partition stored on Broker's disk
- Partition: 1..n log files
- Each message in Log identified by *Offset*
- Configurable Retention Policy

Broker Basics

- Producer sends Messages to Brokers
- Brokers receive and store Messages
- A Kafka Cluster can have many Brokers
- Each Broker manages multiple Partitions



Broker Replication



Producer Basics

- Producers write Data as Messages
- Can be written in any language
 - Native: Java, C/C++, Python, Go,, .NET, JMS
 - More Languages by Community
 - REST Server for any unsupported Language
- Command Line Producer Tool

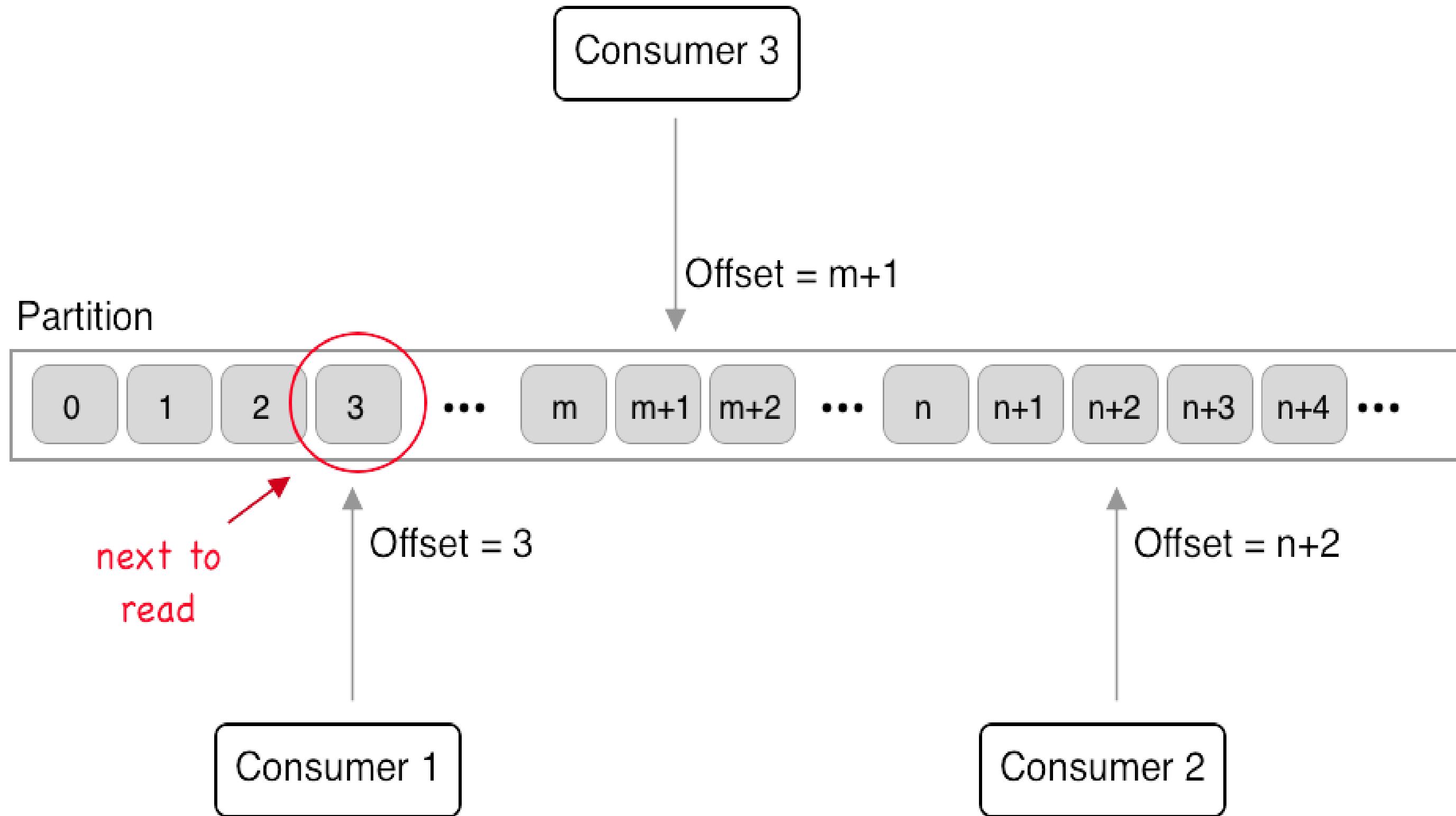
Load Balancing and Semantic Partitioning

- Producers use a Partitioning Strategy to assign each message to a Partition
- Two Purposes:
 - Load Balancing
 - Semantic Partitioning
- Partitioning Strategy specified by Producer
 - Default Strategy: `hash(key) % number_of_partitions`
 - No Key → Round-Robin
- Custom Partitioner possible

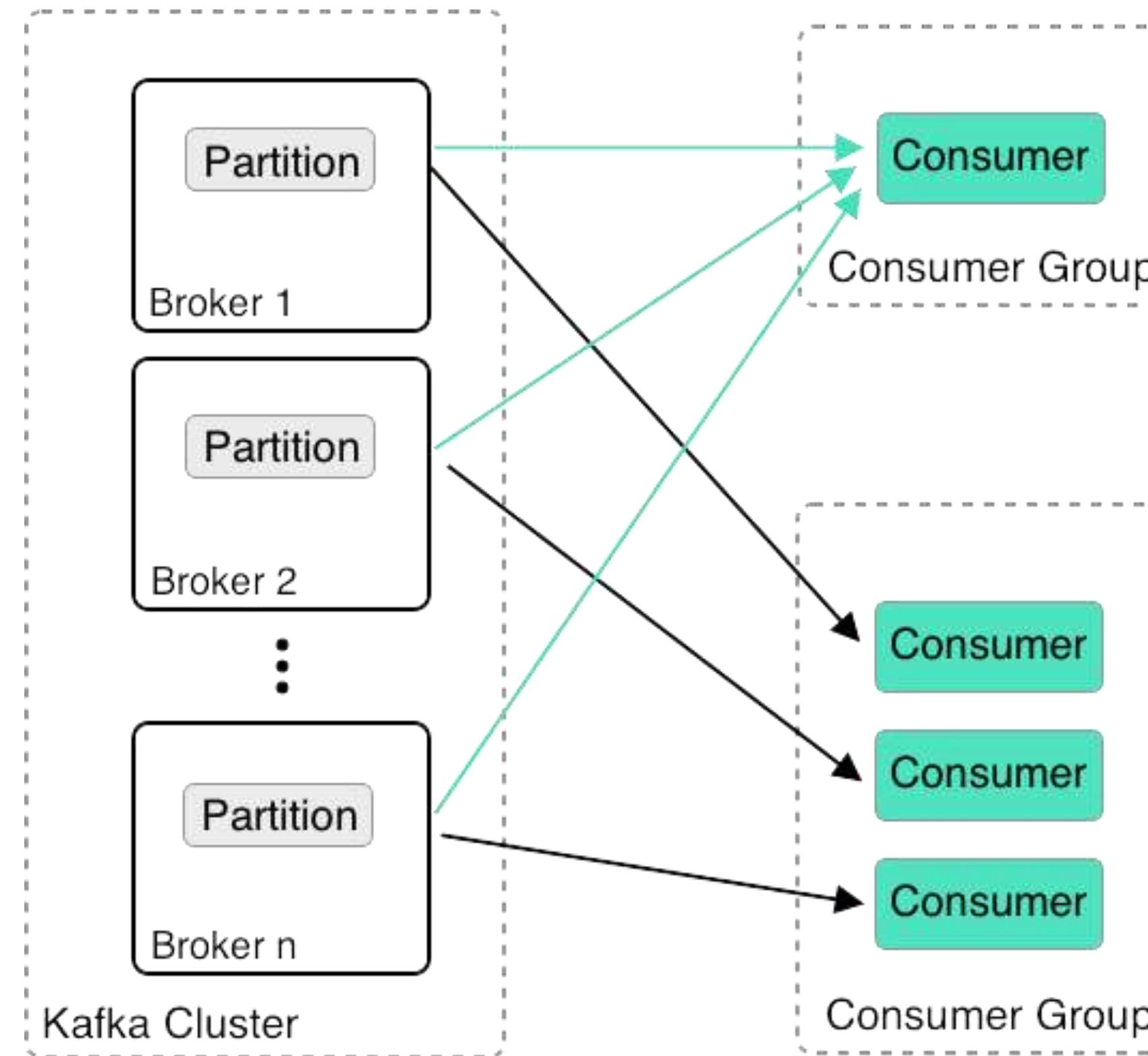
Consumer Basics

- Consumers **pull** messages from 1..n topics
- New inflowing messages are automatically retrieved
- Consumer offset
 - Keeps track of the last message read
 - Is stored in special topic
- CLI tools exist to read from cluster

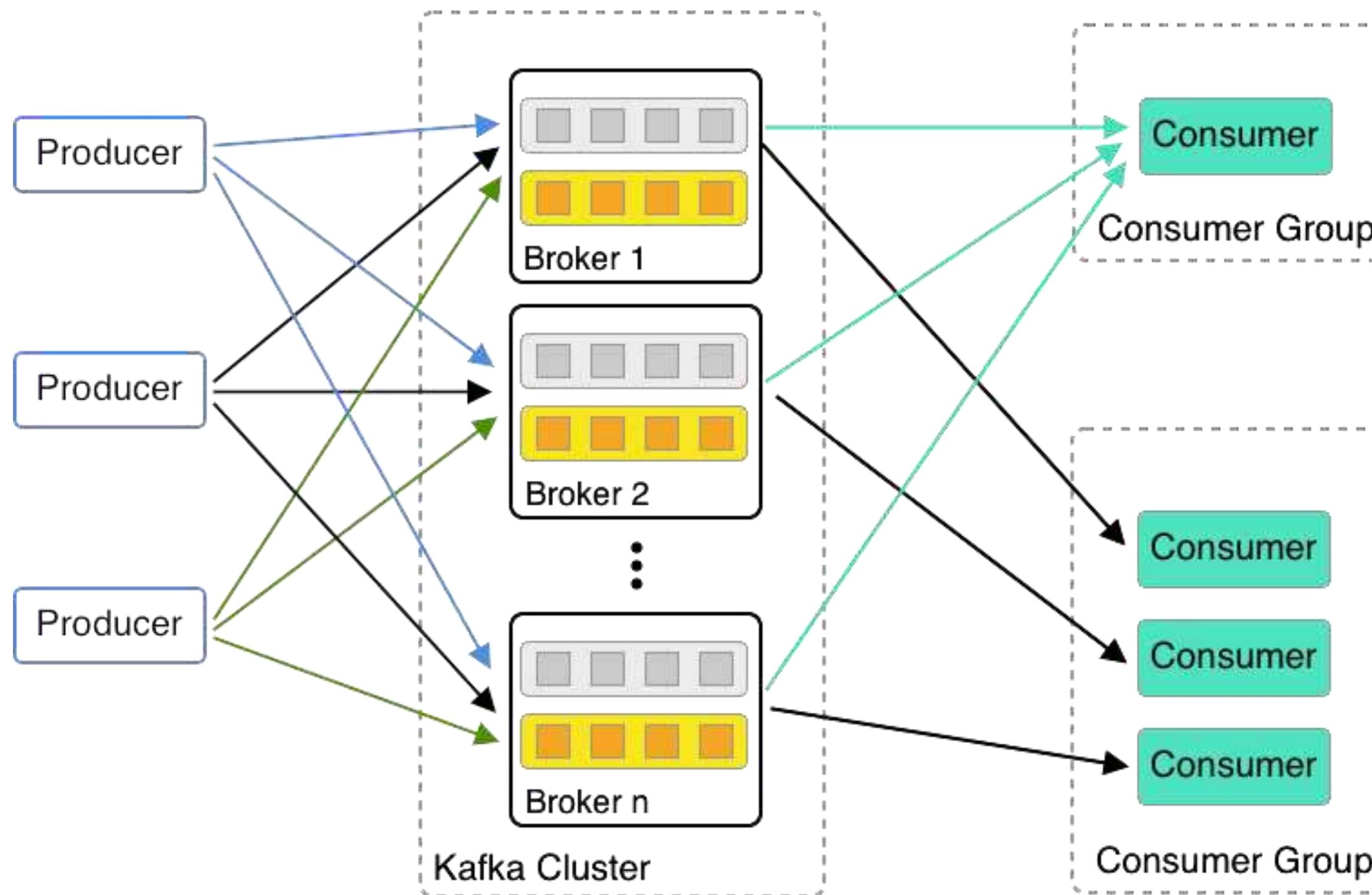
Consumer Offsets



Distributed Consumption



Scalable Data Pipelines



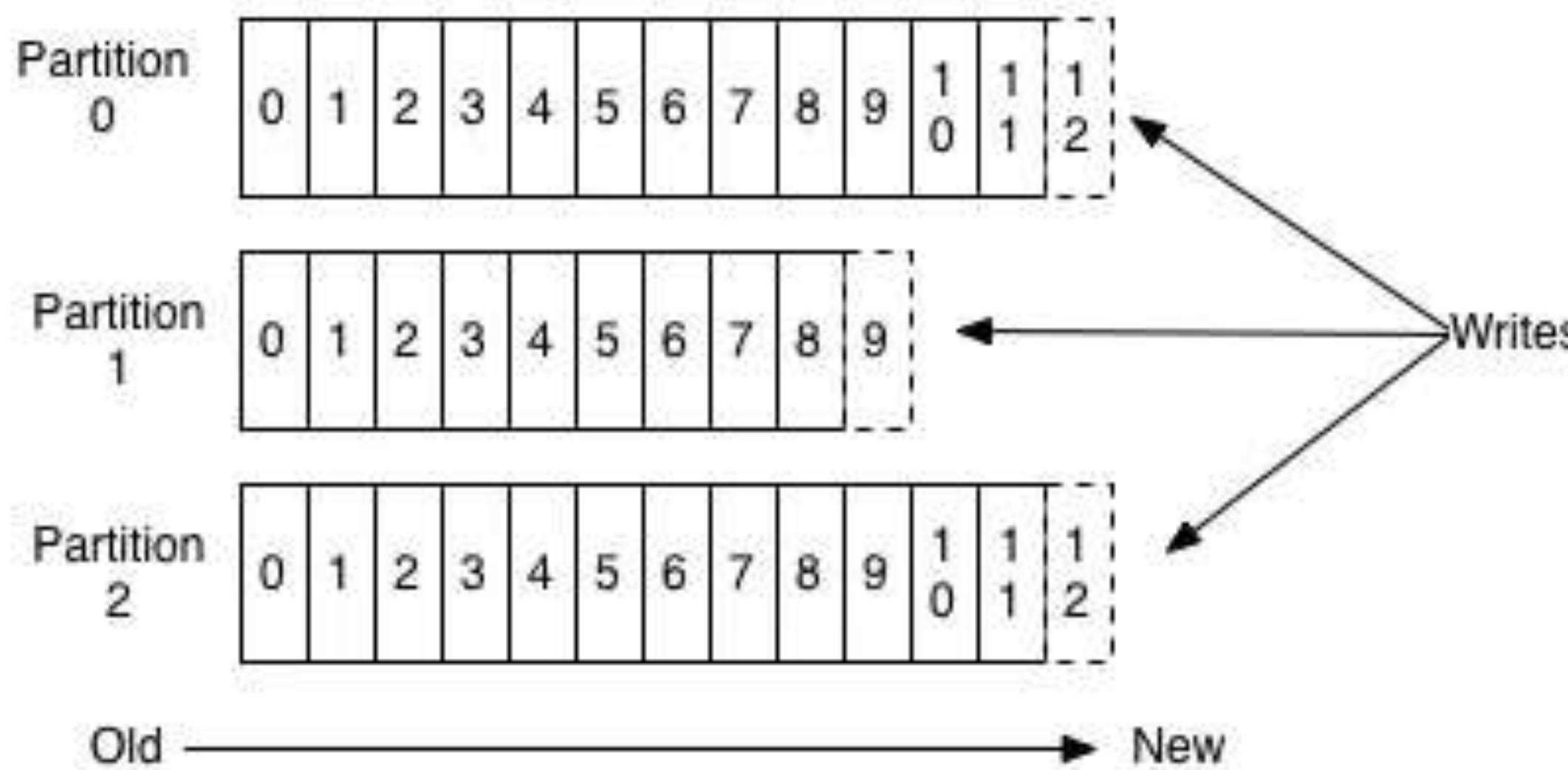
Q & A



- **Questions:**
- Why do we need an odd number of ZooKeeper nodes?
- How many Kafka brokers can a cluster maximally have?
- How many Kafka brokers do you minimally need for high availability?
- What is the criteria that two or more consumers form a consumer group?

KAFKA TOPICS AND PARTITIONS

Anatomy of a Topic



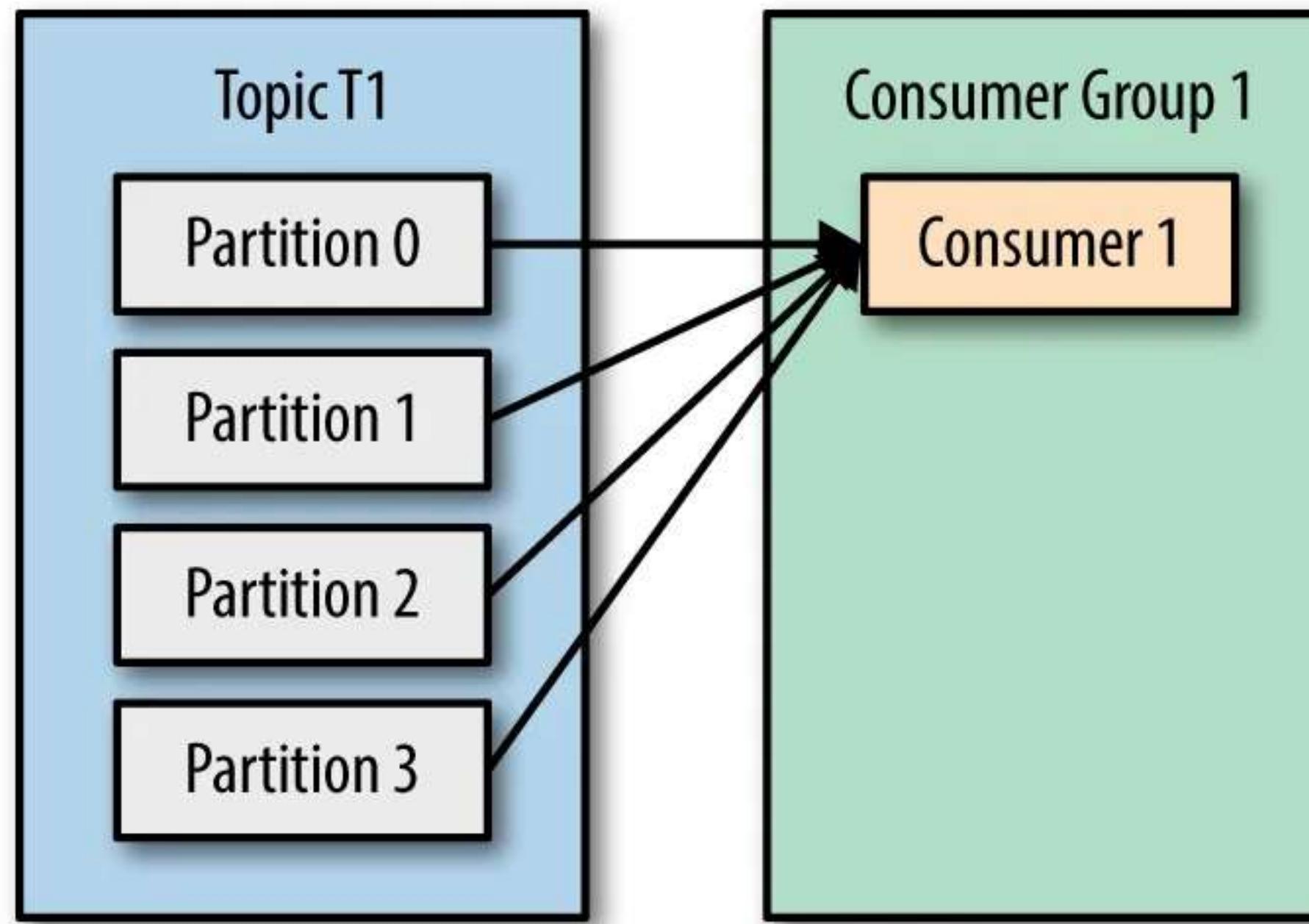
- Every Topic has one or more partitions.
- Default is 1
- No 2 Partitions will have same data
- Every Partition has its own offset.

KAFKA TOPICS AND PARTITIONS

- A bunch of consumers can form a group in order to cooperate and consume messages from a set of topics.
- This grouping of consumers is called a **Consumer Group**.
- If two consumers have subscribed to the same topic and are present in the same consumer group, then these two consumers would be assigned a different set of partitions and none of these two consumers would receive the same messages.

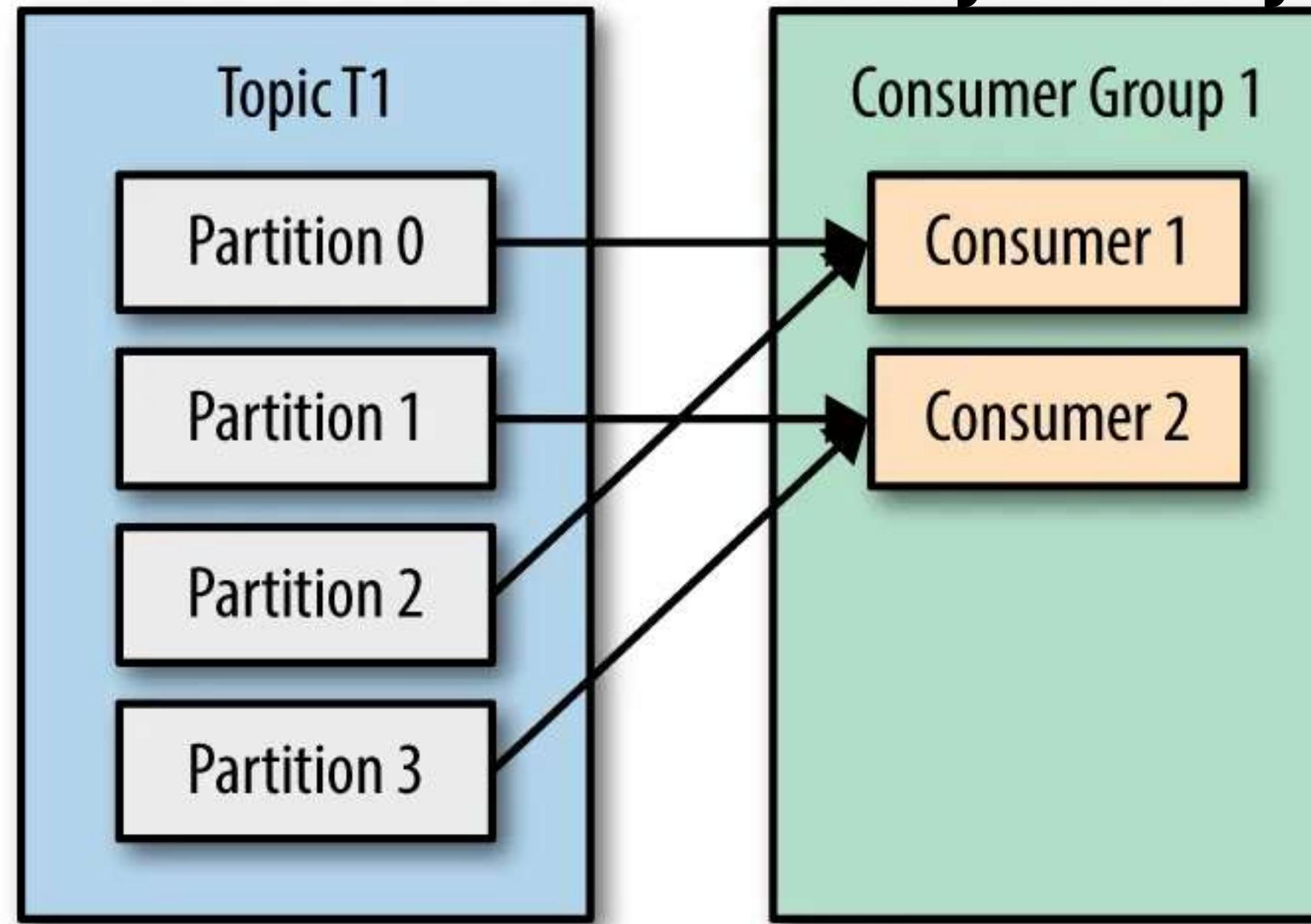
KAFKA TOPICS AND PARTITIONS

- Scenario #1 = 4 Partitions, 1 Consumer Group, 1 Consumer



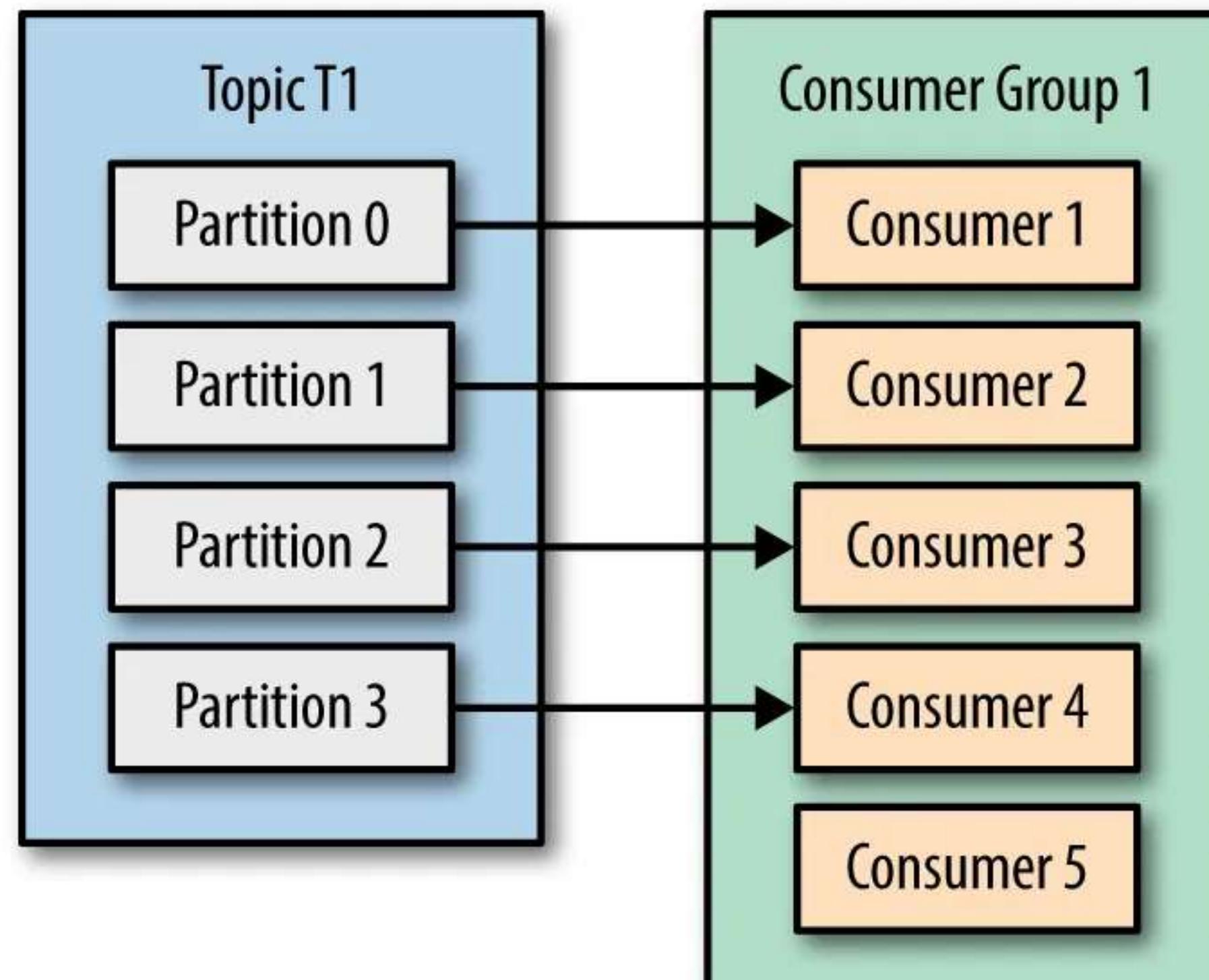
KAFKA TOPICS AND PARTITIONS

- Scenario #2: 4 Partitions, 1 Consumer Group, 2 Consumers
- EACH Partition is consumed by exactly 1 Consumer



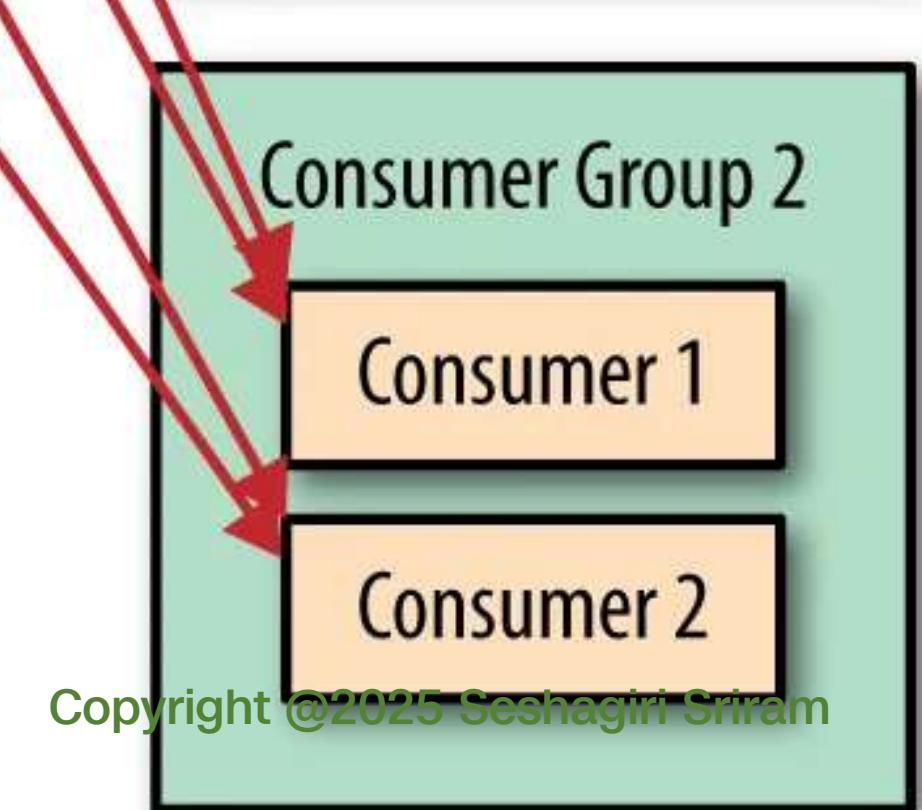
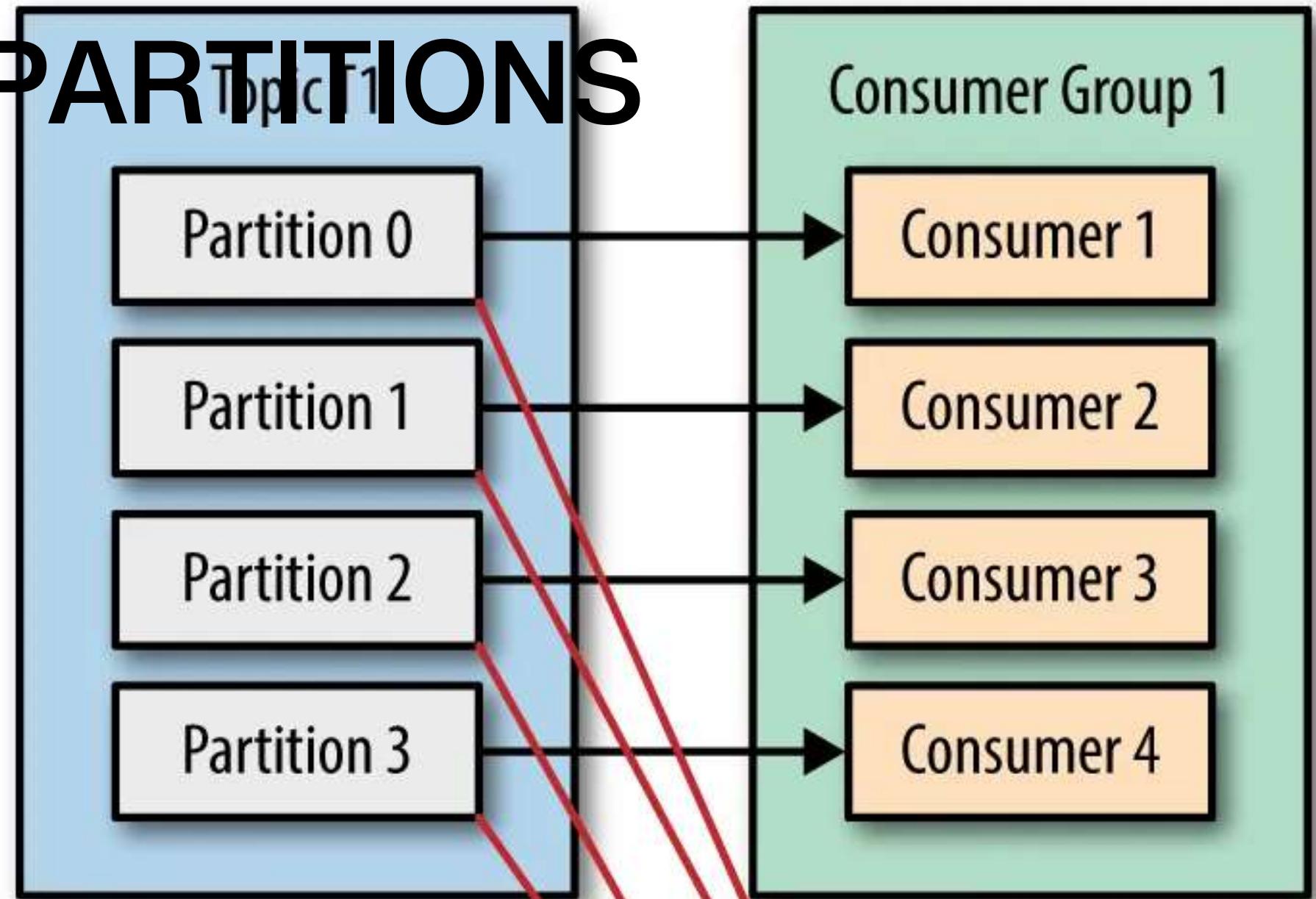
KAFKA TOPICS AND PARTITIONS

- Scenario #3: More Consumers than there are partitions.



KAFKA TOPICS AND PARTITIONS

- SCENARIO #4:
Multiple Consumers
need to Read from
Same Partition.



KAFKA TOPICS AND PARTITIONS

- If your goal is to attain a higher throughput or increase the consumption rate for a particular topic, then adding multiple consumers in the same consumer group would be the way to go.
- You can have at max consumers equal to the number of partitions of a topic in a consumer group, adding more consumers than the number of partitions would cause the extra consumers to remain idle, since Kafka maintains that no two partitions can be assigned to the consumer in a consumer group.
- When the the number of consumers consuming from a topic is equal to the number of partitions in the topic, then each consumer would be reading messages from a single topic, and the message consumption would be happening in parallel, thereby increasing the consumption rate.

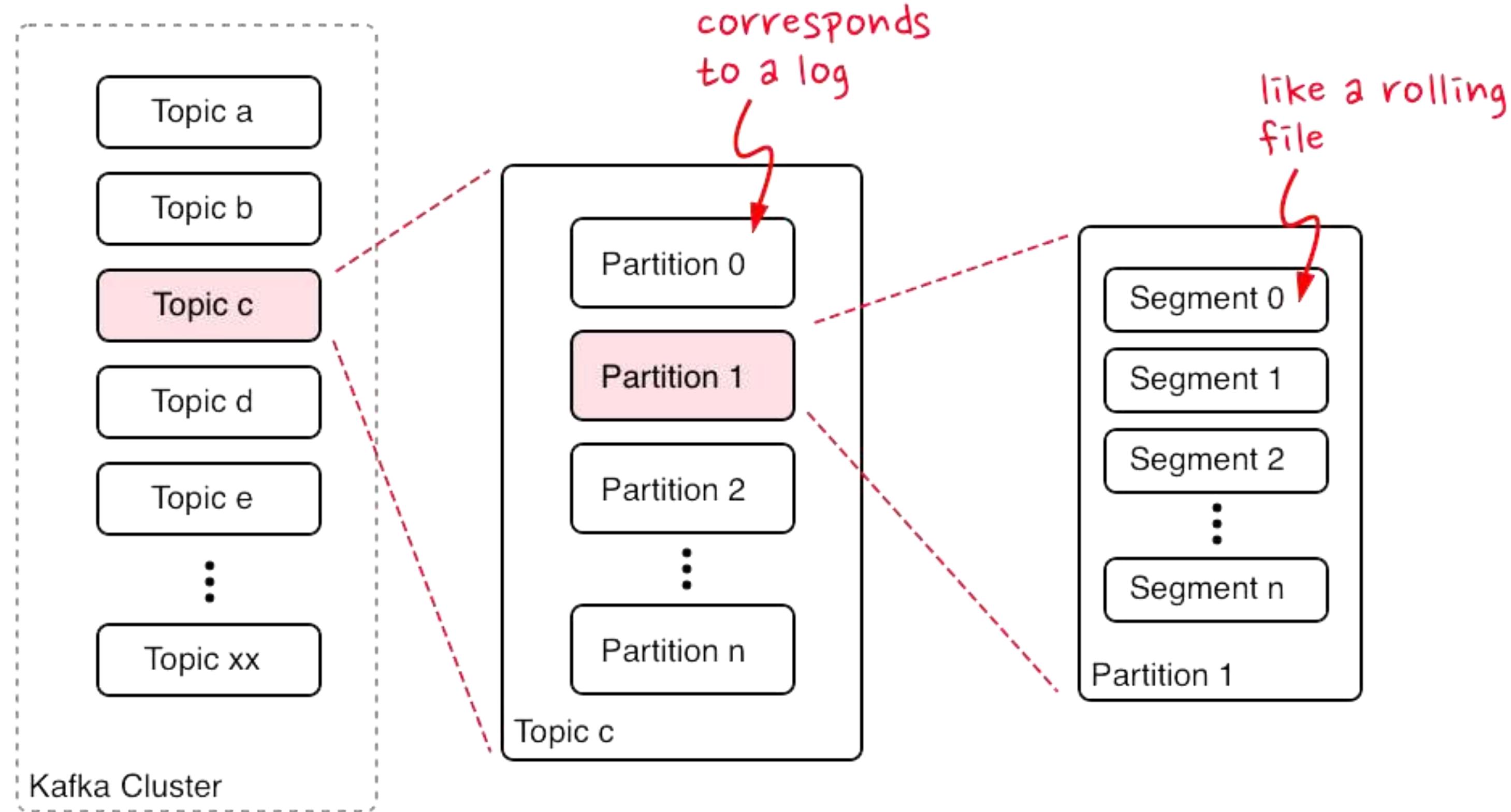
KAFKA TOPICS AND PARTITIONS

- All Consumers are part of a Consumer Group.
- When Consumers are added or removed/stopped, a process of re-balancing occurs.
- Rebalancing happens
 - Consumers are added
 - Consumers go down
 - Consumer is considered Dead by the group Coordinator
 - New Partitions are added.
- Note Partitions cannot be decreased.

Indexes, Retention, Compaction, Logs

- Kafka stores data in **log files**, which are essentially append-only files that hold messages for each partition.
- A Kafka log for a partition isn't a monolithic file but is broken down into **segments**.
- Each segment is a sequence of messages with a monotonically increasing offset.
- Kafka appends new messages to the last segment of a partition.

Indexes, Retention, Compaction, Logs



Indexes, Retention, Compaction, Logs



```
./bin/kafka-run-class.sh kafka.tools.GetOffsetShell --broker-list  
<your_broker_list> --topic <your_topic_name> --time -1
```

See log.dirs folder – there will be 1 folder / topic partition.

AN offset represents the position of a message within a partition.

Indexes, Retention, Compaction, Logs

File Explorer View			
Name	Date modified	Type	Size
configured-topic-0	11/15/2021 4:56 PM	File folder	
configured-topic-1	11/15/2021 4:56 PM	File folder	
configured-topic-2	11/15/2021 4:56 PM	File folder	
.lock	11/15/2021 4:48 PM	LOCK File	0 KB
cleaner-offset-checkpoint	11/15/2021 4:48 PM	File	0 KB
log-start-offset-checkpoint	11/15/2021 7:27 PM	File	1 KB
meta.properties			
recovery-point-offset-checkpoint			
replication-offset-checkpoint			

Path: This PC > OS (C) > tmp > kafka-logs > configured-topic-0

Name	Date modified	Type	Size
00000000000000000000.index	11/15/2021 4:56 PM	INDEX File	10,240 KB
00000000000000000000	11/15/2021 4:56 PM	Text Document	0 KB
00000000000000000000.timeindex	11/15/2021 4:56 PM	TIMEINDEX File	10,240 KB
leader-epoch-checkpoint	11/15/2021 4:56 PM	File	0 KB

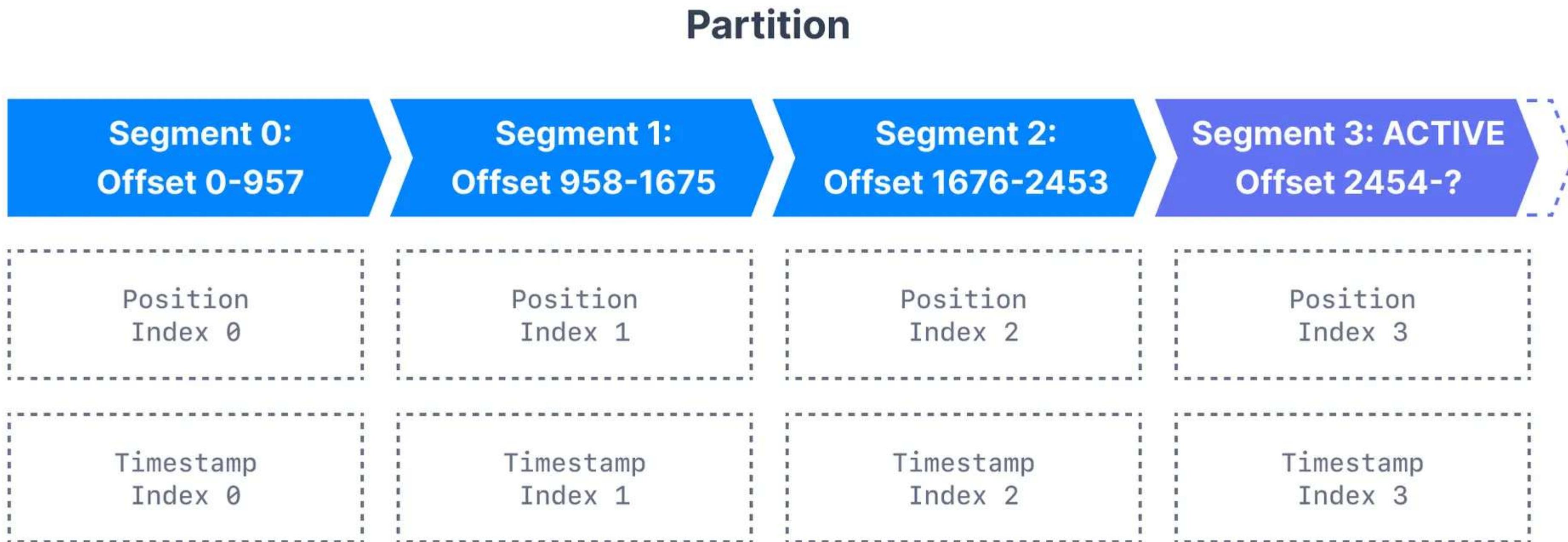
Indexes, Retention, Compaction, Logs

- Each segment is stored as a pair of files
 - .log
 - Actual Messages
 - .index
 - Manages message offsets to their position
- The naming convention is <offset of 1st msg within segment>
- If Base offset of first message is 5000, then files are
 - 00000000000005000.log
 - 00000000000005000.index

Indexes, Retention, Compaction, Logs

- There are actually 2 indices
 - An offset to position index - It helps Kafka know what part of a segment to read to find a message
 - A timestamp to offset index - It allows Kafka to find messages with a specific timestamp

Indexes, Retention, Compaction, Logs



Indexes, Retention, Compaction, Logs

- Kafka will not keep data forever
- It will not keep it in a single file
- It rolls over segment files based on
 - Time
 - Segment size (default 1GB)
- Time is controlled by log.roll.ms
- Size is controlled by log.segment.bytes

Indexes, Retention, Compaction, Logs

- Improved Manageability: Smaller segment files are easier to handle, particularly for deletion and retention.
- More Frequent Compaction: Kafka's compaction process, which removes obsolete data, becomes more frequent with time-based rolling
- Trade-off: Too frequent log rolling increases the number of small files, which may slow down performance due to increased file system overhead. On the other hand, rolling too infrequently creates larger files that are slower to process during reads and compaction.

Indexes, Retention, Compaction, Logs

- Kafka will store messages before they are deleted.
- Time Based retention - `log.retention.hours` – how long before they are deleted (also see `log.retention.ms`)
- Size Based retention – `log.retention.bytes` (default 10 GB)
- Compaction – Only latest version of key will be stored in logs.
- Question: Assume `log.retention.ms` has been set and segment is not closed because `log.segment.bytes` or `log.roll.ms` not set, what should happen? Why?

Indexes, Retention, Compaction, Logs

```
[2020-12-24 15:51:17,808] INFO [Log partition=Topic-2,  
dir=/Folder/Kafka_data/kafka] Found deletable segments with base  
offsets [165828] due to retention time 604800000ms breach  
(kafka.log.Log)
```

```
[2020-12-24 15:51:17,808] INFO [Log partition=Topic-2,  
dir=/Folder/Kafka_data/kafka] Scheduling segments for deletion  
List(LogSegment(baseOffset=165828, size=895454171,  
lastModifiedTime=1608220234000, largestTime=1608220234478))  
(kafka.log.Log)
```

```
$KAFKA_HOME/bin/kafka-configs.sh --bootstrap-server :9092 --entity-type  
topics --entity-name my-topic --describe --all | grep segment.bytes
```

Indexes, Retention, Compaction, Logs

- Broker Configs: `log.segment.bytes` and `log.roll.ms`
- Broker Configs: `log.retention.ms` and `log.retention.bytes`
- Per Topic: `retention.ms`, `retention.bytes`, `segment.bytes`, and `segment.ms`.

Indexes, Retention, Compaction, Logs

Common Modes

(a) 1 Week of Retention

`log.retention.hours=168`

`log.retention.bytes = -1`

`log.retention.ms = 604800000`

(b) Indefinite retention time bounded by 500 MB

`log.retention.hours = -1`

`log.retention.ms = -1`

`log.retention.bytes=524288000`

Indexes, Retention, Compaction, Logs

- **Segment Size**
 - Smaller Segments: Can improve deletion times but increase the number of files, which adds file system overhead.
 - Larger Segments: Reduce the number of files but can slow down deletion and lead to slower recovery times after failures
- **Rolling Frequency**
 - Time-based Rolling: Ensures fresh segments are created at regular intervals, making it easier to manage log compaction and retention.
 - Size-based Rolling: Prevents segments from becoming too large, which improves write and read performance.
- **Retention Policies**
 - Time-based Retention: Suitable for streaming applications where you need to retain data for a fixed period.
 - Size-based Retention: Ideal for environments with limited storage capacity.
 - Compaction: Useful for use cases where only the latest state of each key is needed, such as changelog topics in stateful applications.

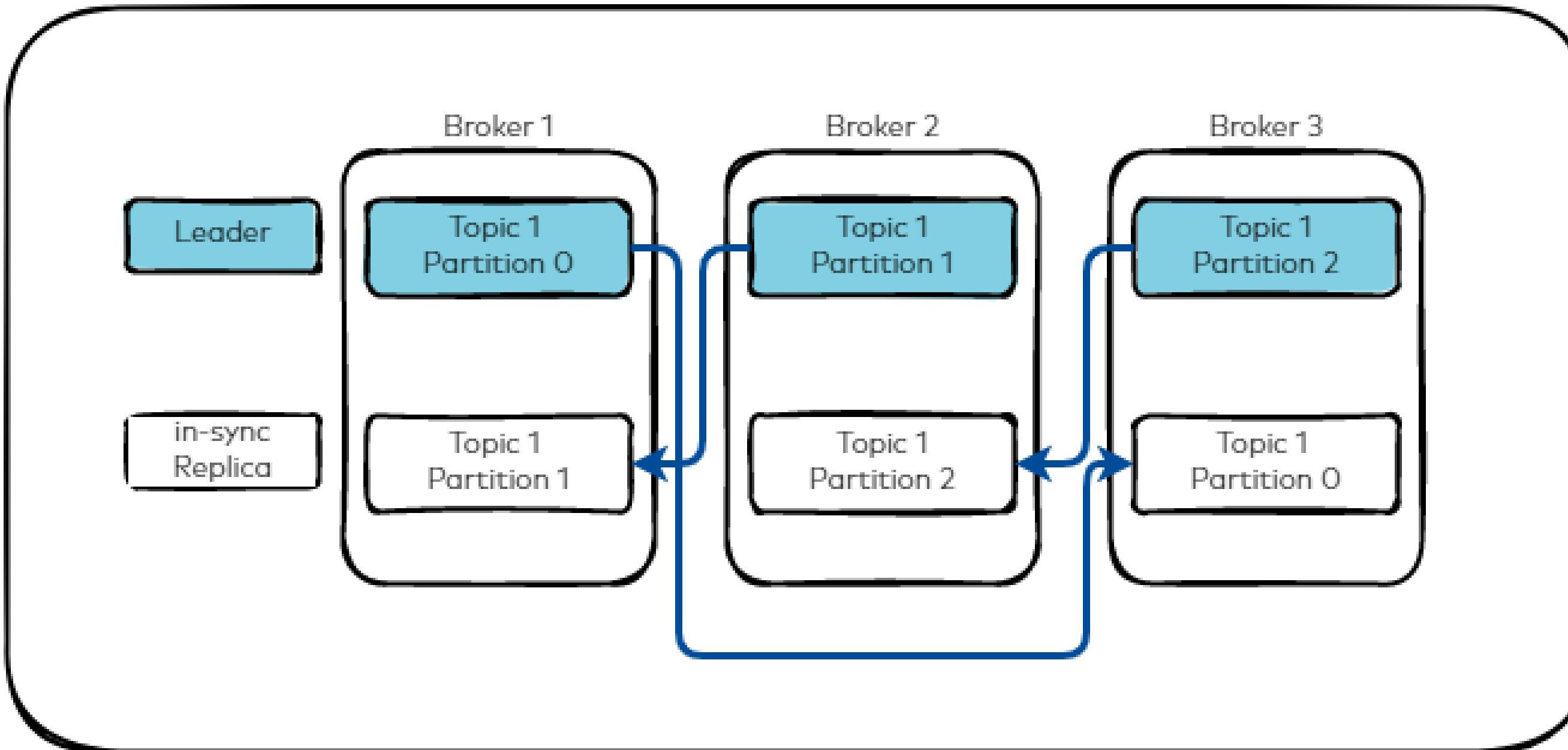
Replicas

- By Default, partitions only have one replica.
- If partition fails, data is lost.
- For High availability, the number of replicas (or replication factor) is set to > 1
- More on this when we talk of producers.

Replicas

- In practice a replication factor of 3 allows for 2 failures.
- It is a via medium between resiliency and performance
- For a given topic/partition, one broker is designated as a leader. (Others are called followe)
- An ISR or In-Sync Replicas, is a replica that's up-to-date with leader broker.
- All reads and writes go to the leader

Replication



Replication

- For Kafka node to be considered alive, it has to meet two conditions
 - A node must be able to maintain its session with the controller
 - If it is a follower it must replicate the writes happening on the leader and not fall “too far” behind
- `min.insync.replicas` Configured both at topic and Broker
- A value of 2 = 2 brokers that are ISR (including leader)
- `default.replication.factor` is set to 1.
- Ref: <https://thelinuxcode.com/change-replication-factor-apache-kafka/>

Replication

- How do you set it?
 - `./usr/bin/kafka-topics --bootstrap-server localhost:9092 --create --topic my-topic-with-short-retention-period --partitions 1 --replication-factor 3 --config retention.ms=10000 --config segment.ms=10000`
- How do you change it?
 - Use JSON files/reassign-partitions.

Recap on Replicas

- By Default, partitions only have one replica.
- If partition fails, data is lost.
- For High availability, the number of replicas (or replication factor) is set to > 1
- We typically set the replication factor to 3 and minimum in sync replicas to be 2.

Producers

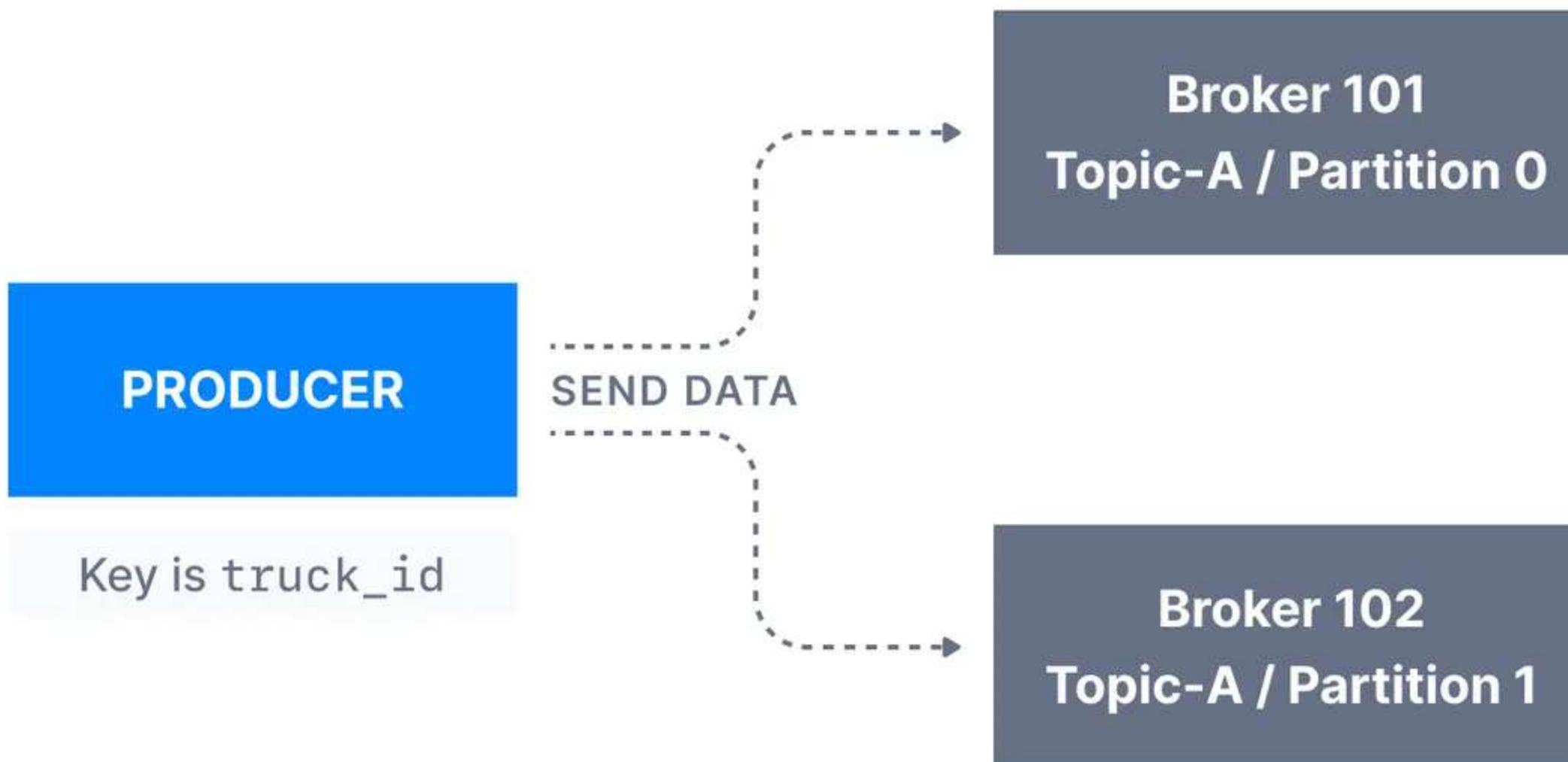
Producers, Consumers, Partitions



Producers, Consumers, Partitions

- For a message to be written into a topic, a producer should specify a level of acknowledgment (ACKS)
- Every Message contains an optional key and a value.
- Key selection is very critical to Kafka Performance and load balancing.
- E.g. Customer_ID, Vehicle_ID

Producers, Consumers, Partitions



Example:

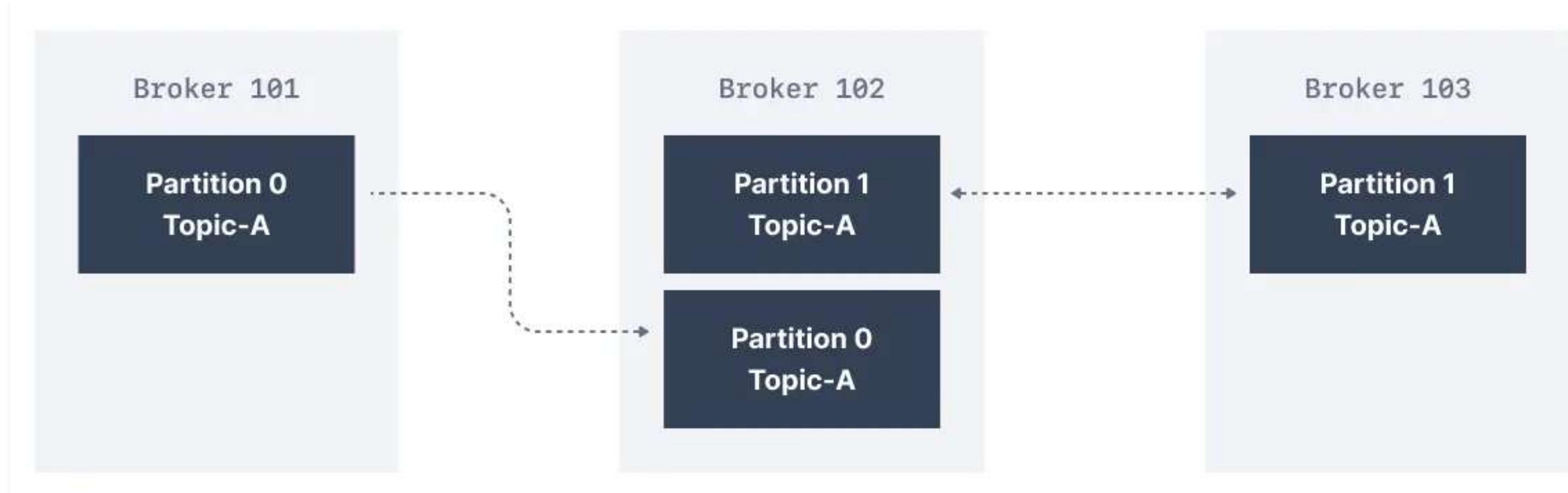
truck_id_123 data will always be in partition 0

truck_id_234 data will always be in partition 0

truck_id_345 data will always be in partition 1

truck_id_456 data will always be in partition 1

Producers, Consumers, Replicas



Producers, Consumers, Replicas

- Kafka Producers will only write to the current leader.
- They must also specify ACK level before message is written
 - essentially to how many replicas should it be written before it's considered good.
- For Kafka ≥ 3.0 , default value is all
- For Kafka < 3.0 , default ack value is 1
- An acks value of 0 – do not wait for the broker to acknowledge.

Producers, Consumers, Replicas

- In practice a replication factor of 3 allows for 2 failures.
- It is a via medium between resiliency and performance
- For a given topic/partition, one broker is designated as a leader. (Others are called replicas)
- An ISR or In-Sync Replicas, is a replica that's up-to-date with leader broker.

Producers

- For a message to be written into a topic, a producer should specify a level of acknowledgment (ACKS)
- Every Message contains an optional key and a value.
- Key selection is very critical to Kafka Performance and load balancing.
- E.g. Customer_ID, Vehicle_ID

Producers

- Default uses an algorithm : murmur2 algorithm,
- `targetPartition = Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1)`
- <https://murmurhash.shorelabs.com/>
- <https://md5hashing.net/hash/murmur3/>

The goal of producer performance tuning

With a given dataset to send:

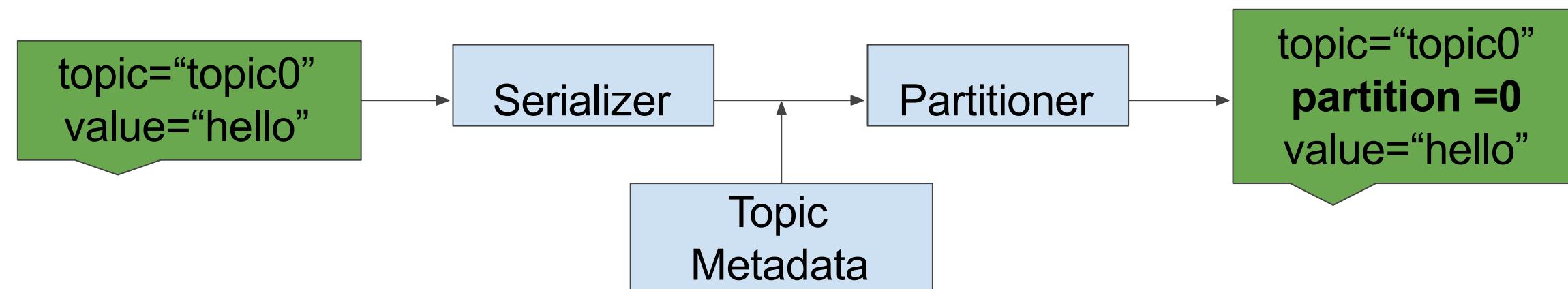
- Achieve the **throughput** and **latency** goals with guarantees of
 - Durability
 - Ordering
- Focus on the average producer performance
 - 99 percentile performance sometimes cannot be “tuned”
 - Tuning average performance also helps 99 percentile numbers
- See additional info
 - broker side recompression ([KIP-31](#))
 - 8 bytes overhead per message due to the introduction of timestamp ([KIP-32](#))

Critical Configurations

- batch.size
- linger.ms
- compression.type
- max.in.flight.requests.per.connection (*affects ordering*)
- acks (*affects durability*)

Understand the Kafka producer

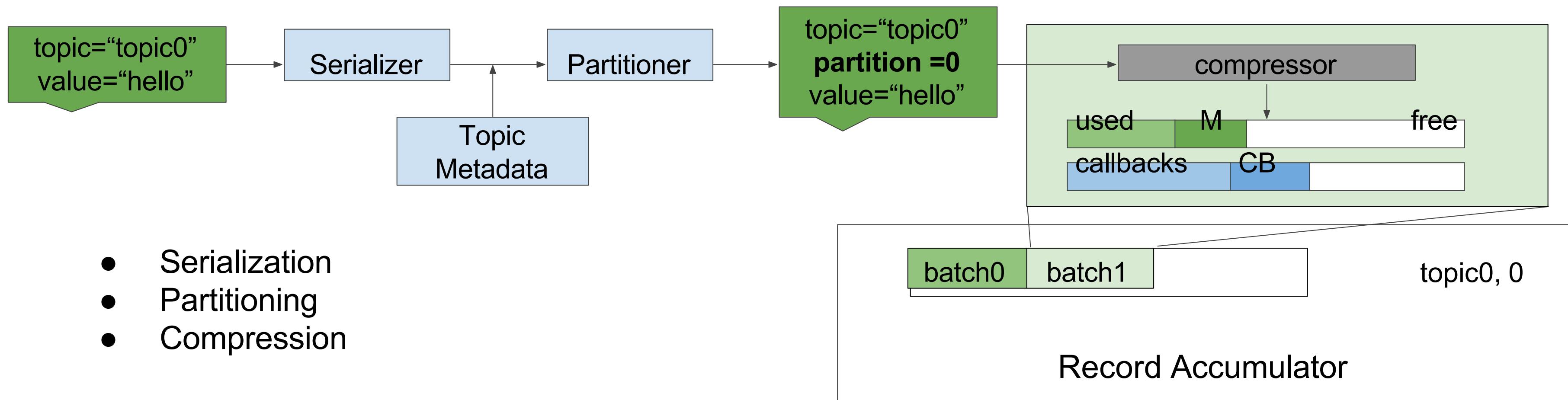
User: `producer.send(new ProducerRecord("topic0", "hello"), callback);`



- Serialization
- Partitioning

Understand the Kafka producer

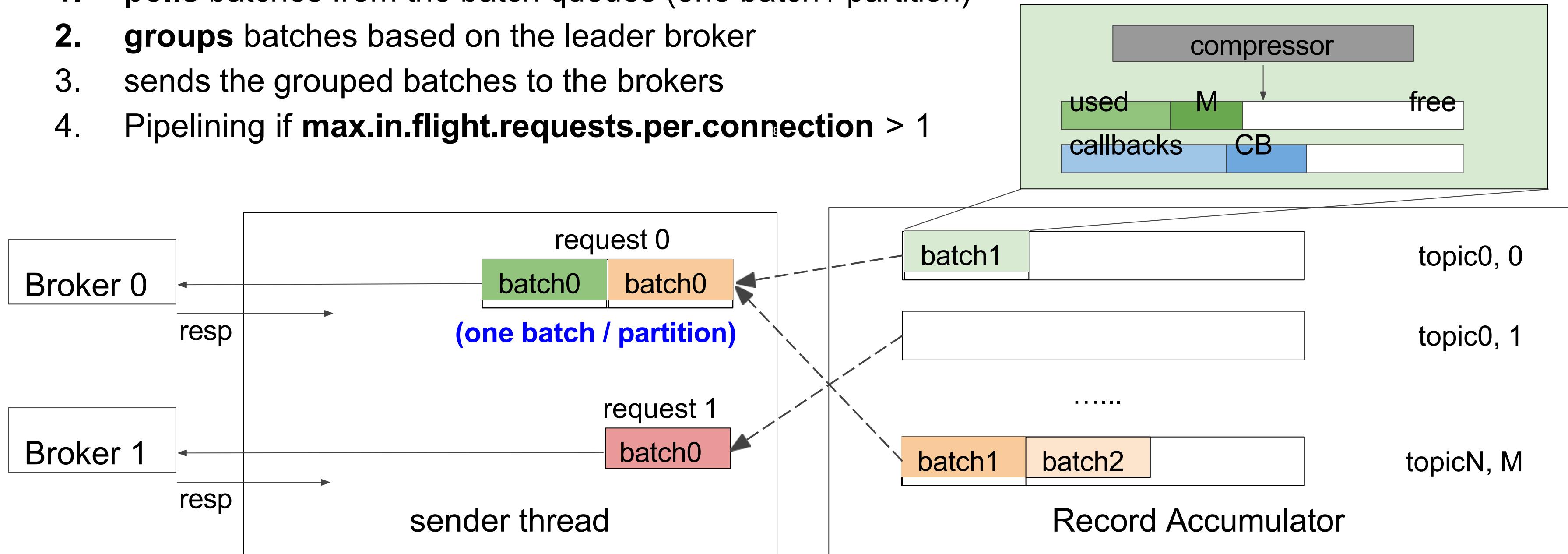
User: `producer.send(new ProducerRecord("topic0", "hello"), callback);`



Understand the Kafka producer

Sender:

1. **polls** batches from the batch queues (one batch / partition)
 2. **groups** batches based on the leader broker
 3. sends the grouped batches to the brokers
 4. Pipelining if **max.in.flight.requests.per.connection** > 1

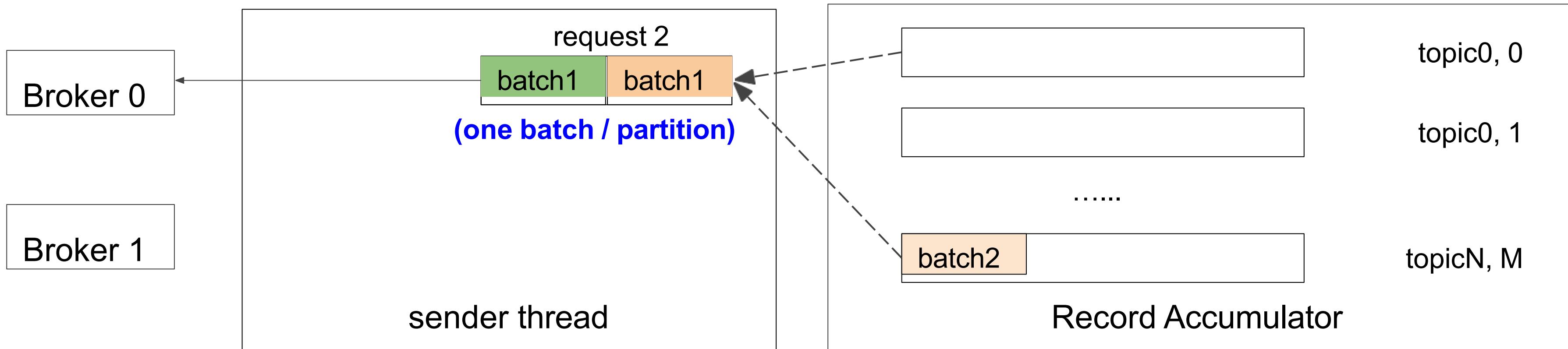
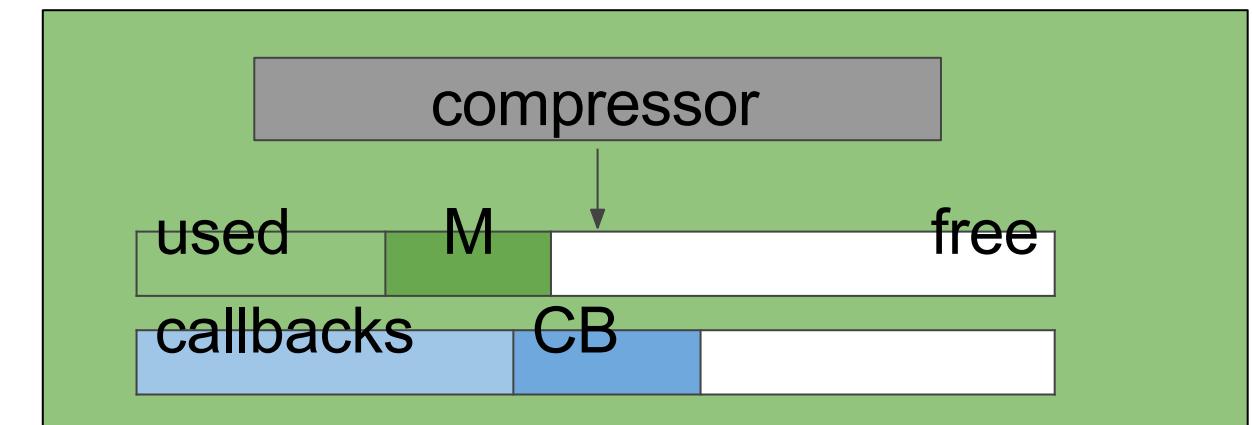


Understand the Kafka producer

Sender:

A batch is ready when one of the following is true:

- `batch.size` is reached
- `linger.ms` is reached
- Another batch to the same broker is ready (piggyback)
- `flush()` or `close()` is called

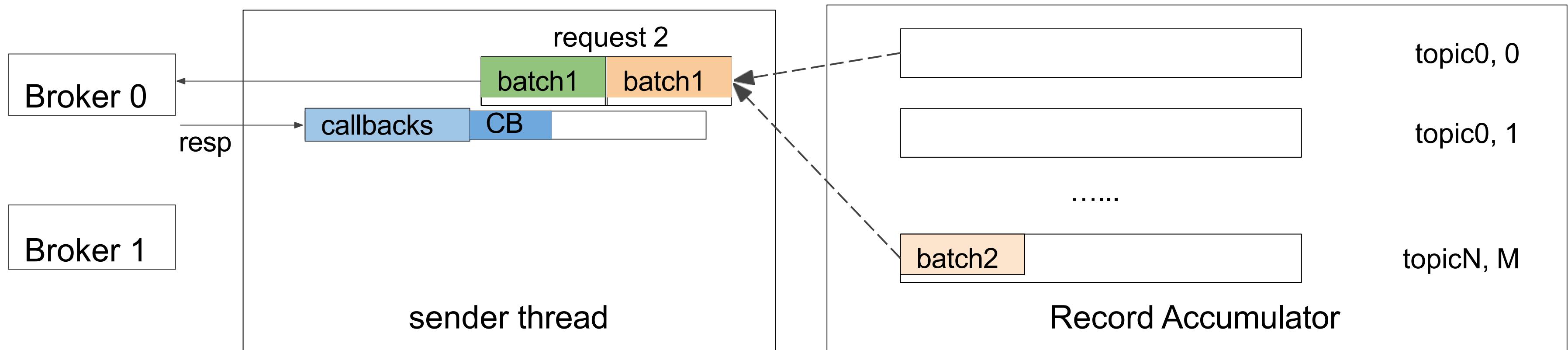
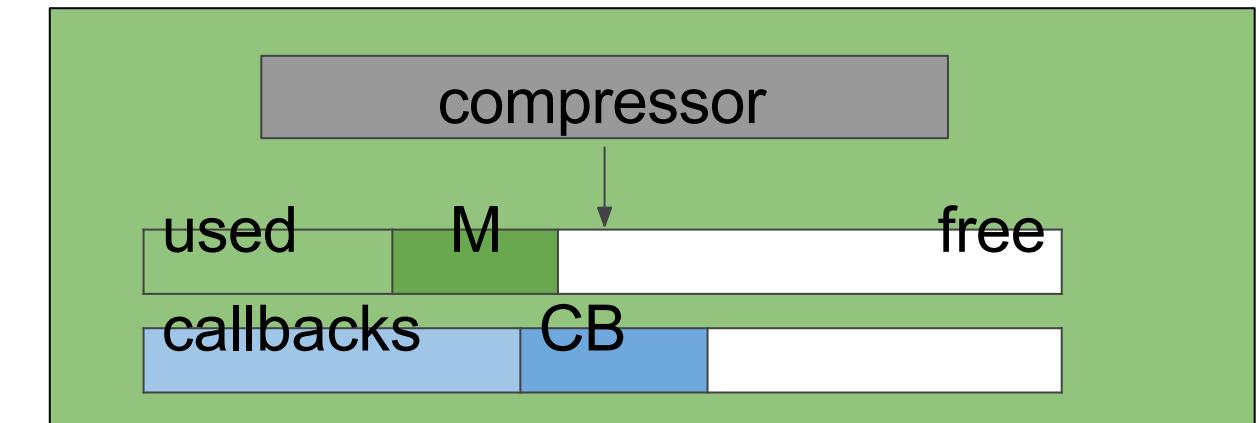


Understand the Kafka producer

Sender:

On receiving the response

- The callbacks are fired in the message sending order



batch.size & linger.ms

batch.size is **size based** batching

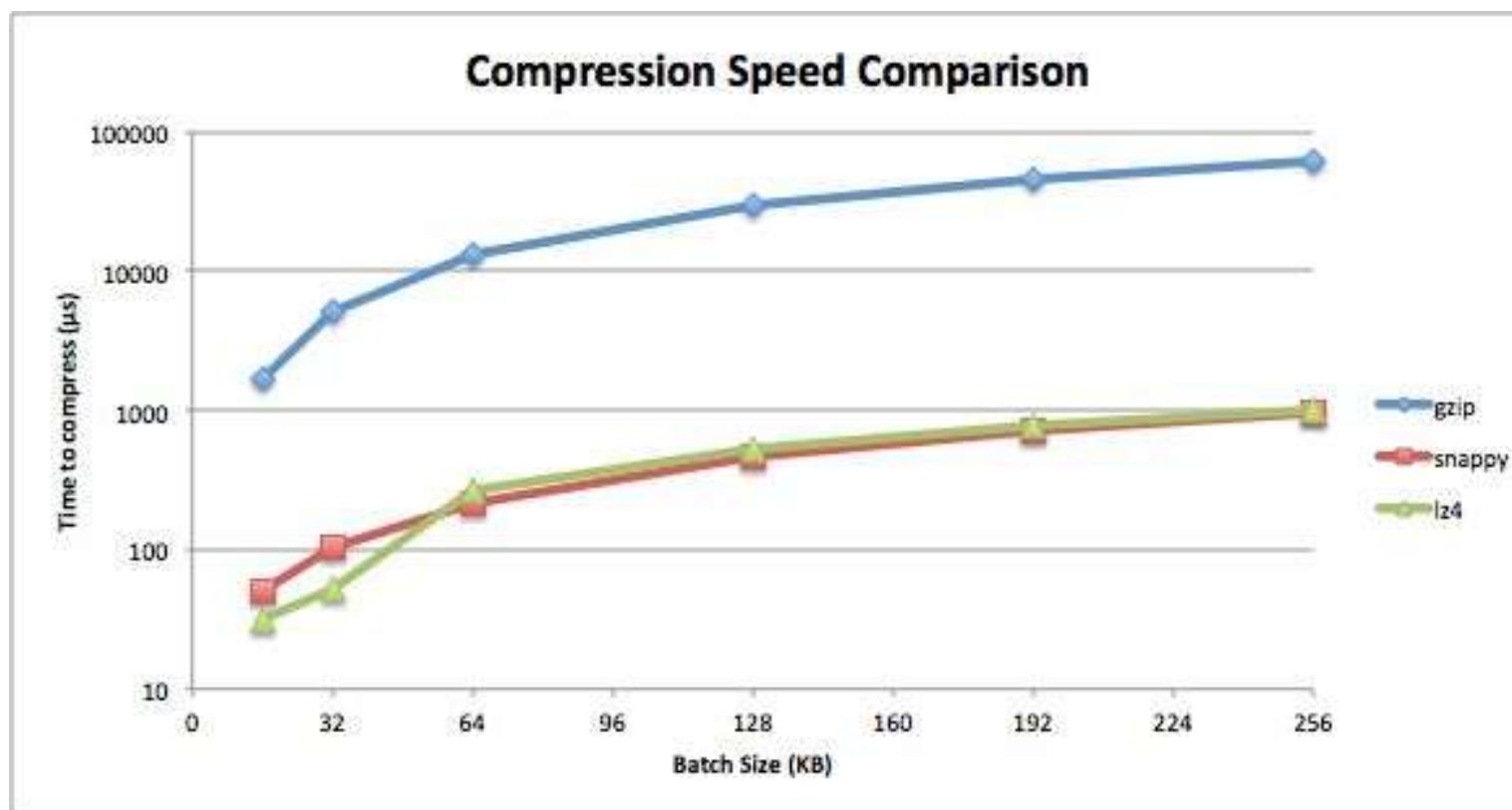
linger.ms is **time based** batching

In general, more batching

- ⇒ **Better** compression ratio ⇒ **Higher** throughput
- ⇒ **Higher** latency

compression.type

- Compression is usually the **dominant** part of the `producer.send()`
- The speed of different compression types differs **A LOT**
- Compression is in user thread, so **adding more user threads** helps with **throughput** if compression is slow



max.in.flight.requests.per.connection

max.in.flight.requests.per.connection > 1 means pipelining.

In general, pipelining

- gives **better throughput**
- may cause **out of order** delivery when retry occurs
- **Excessive pipelining** ⇒ Drop of throughput
 - lock contention
 - worse batching

acks

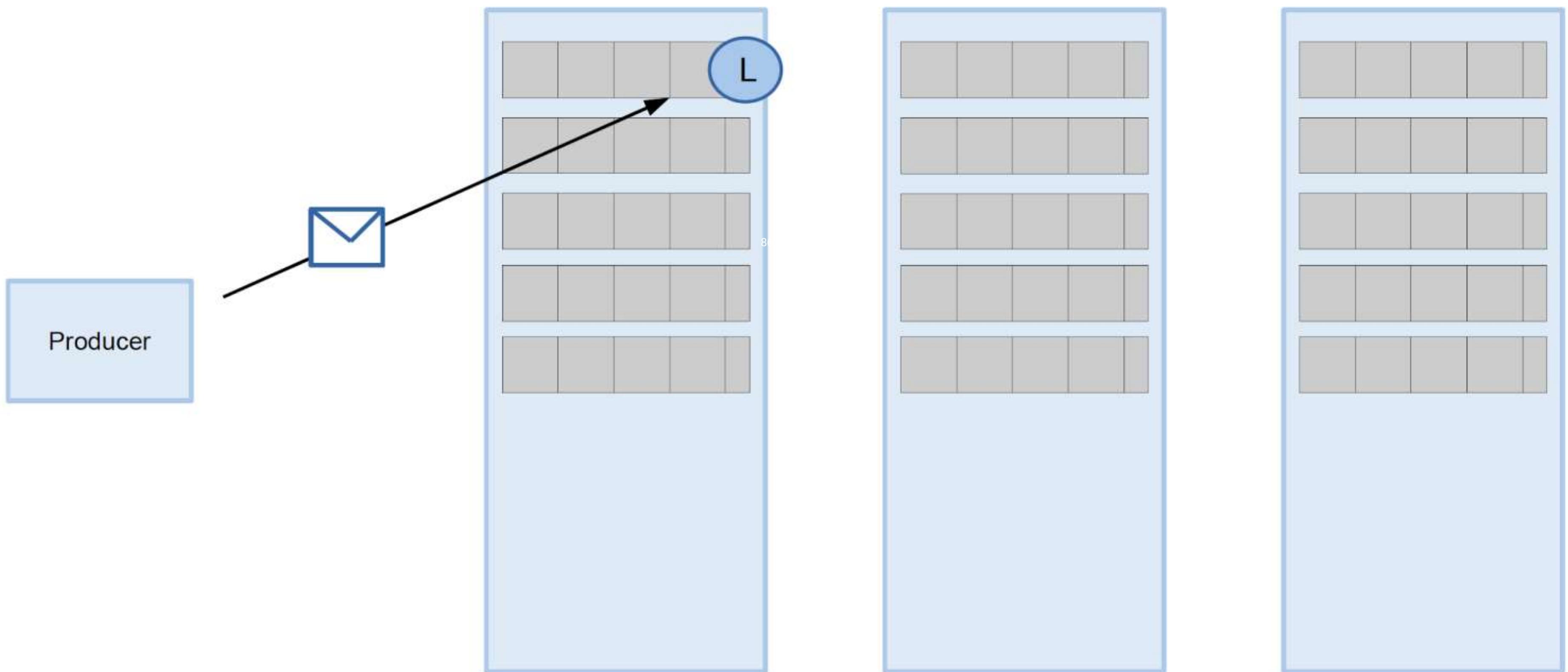
- Kafka Producers will only write to the current leader.
- They must also specify ACK level before message is written
 - essentially to how many replicas should it be written before it's considered good.
- For Kafka ≥ 3.0 , default value is all
- For Kafka < 3.0 , default ack value is 1
- An acks value of 0 – do not wait for the broker to acknowledge.

acks

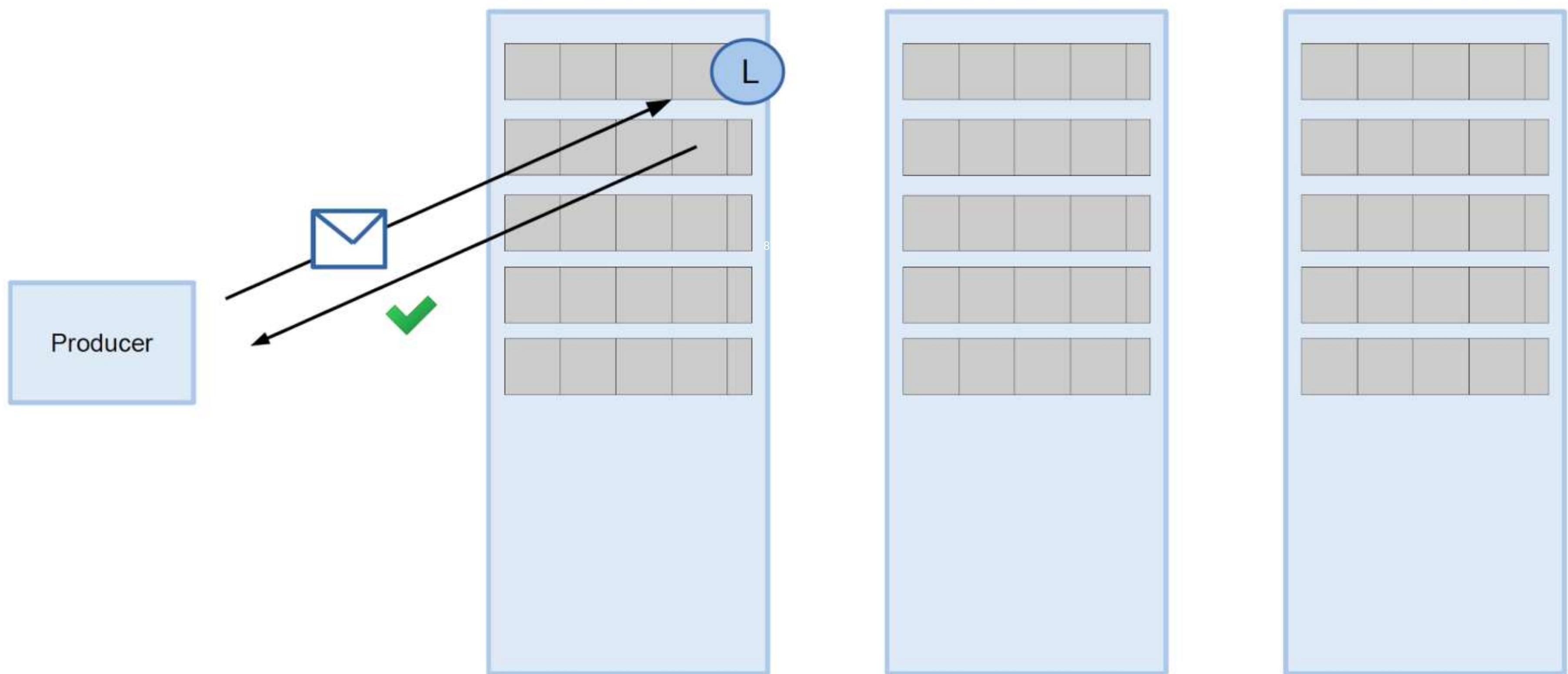
Defines different durability level for producing.

acks	Throughput	Latency	Durability
0	high	low	No guarantee
1	medium	medium	leader
-1	low	high	ISR

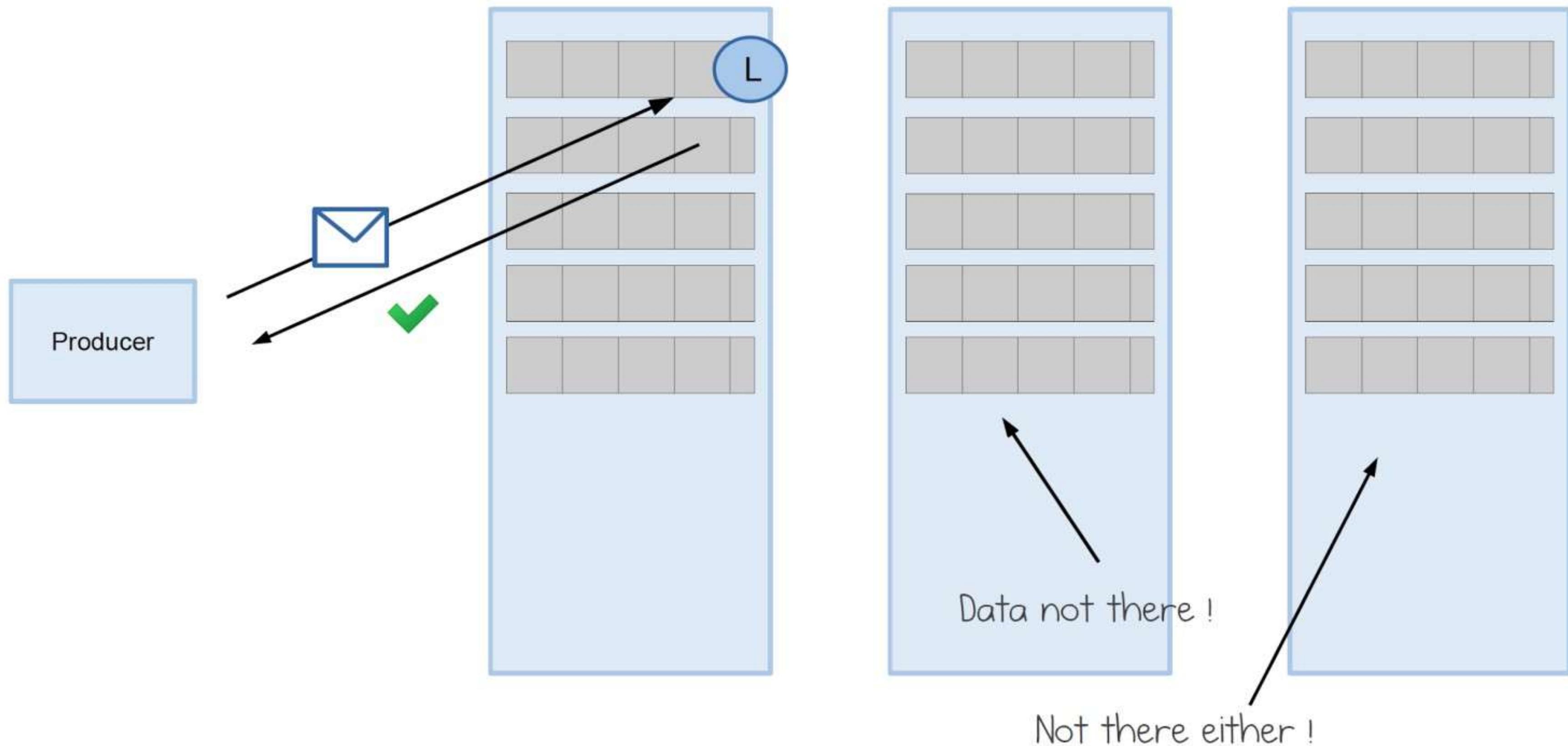
When will the cluster acknowledge ?



As soon as it is on the page cache of
the leader...



As soon as it is on the page cache of
the leader...

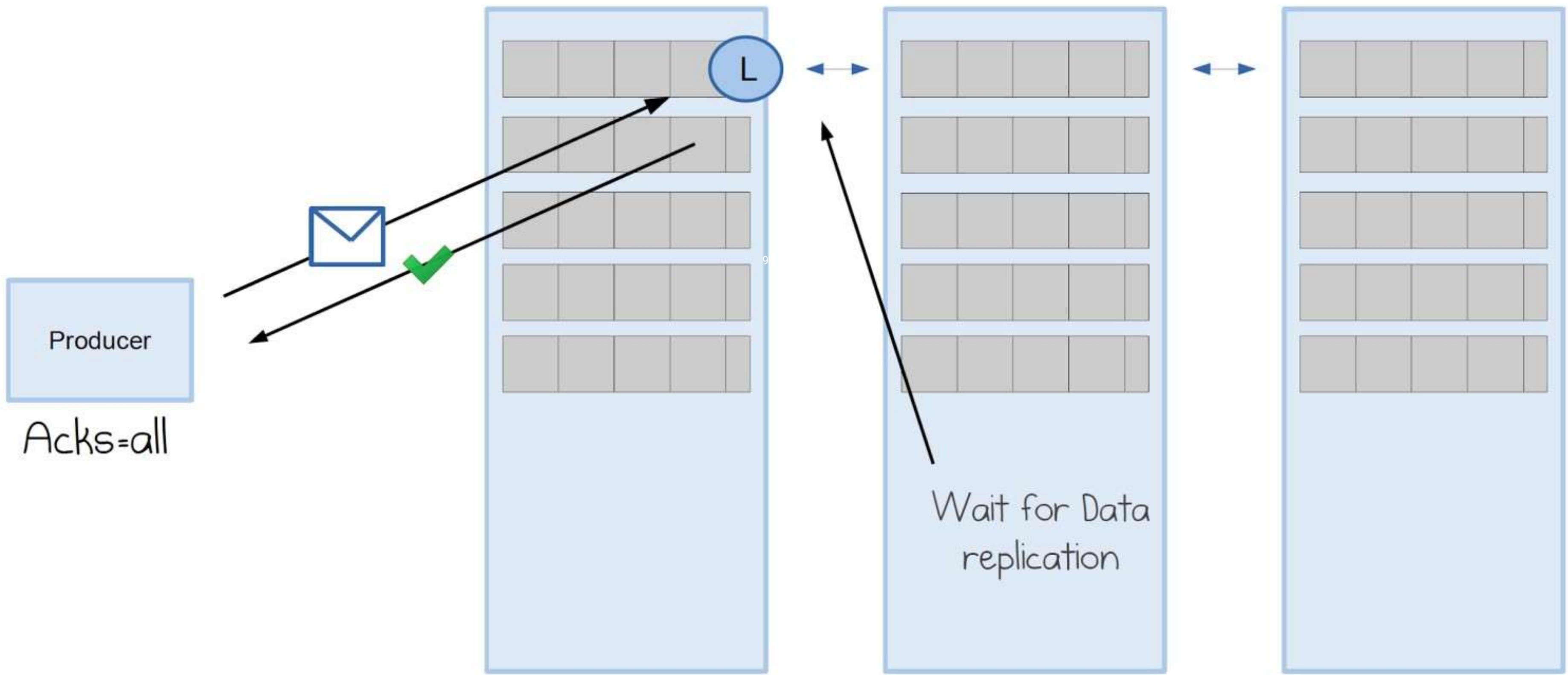


Data durability

acks=1 (default value)
good for **latency**

acks=all
good for **durability**

Replication before acknowledging



Parameter:
acks=all

The leader will wait for
the full set of **in-sync
replicas** to acknowledge
the record.

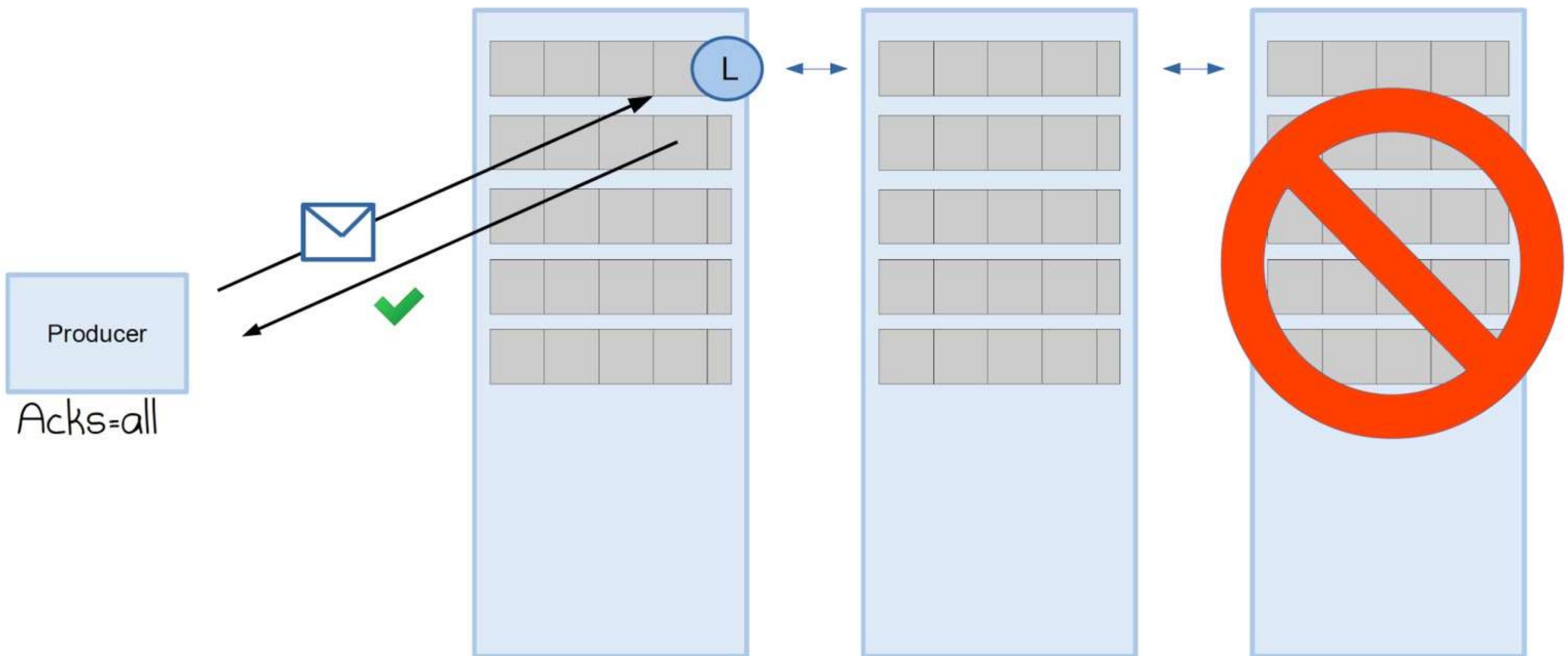
Parameter:

min.insync.replicas

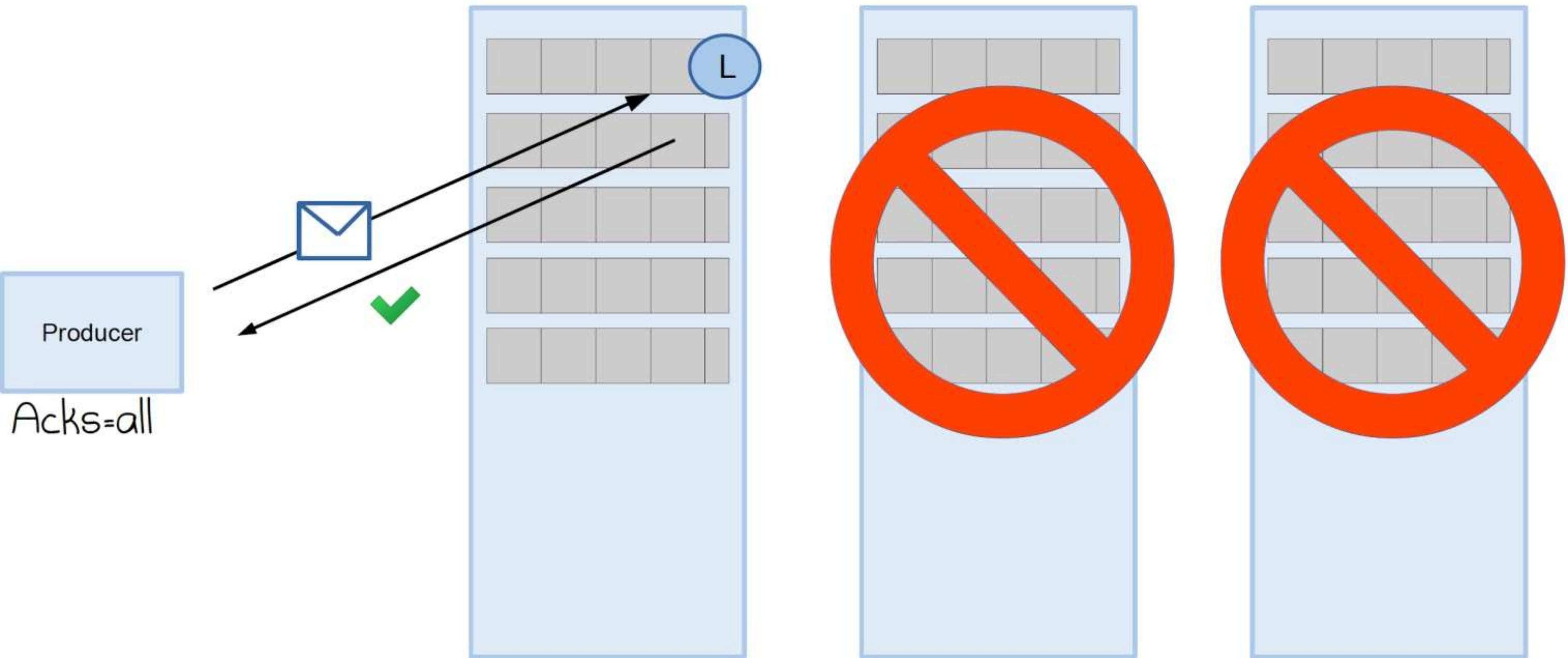
minimum number of replicas that must acknowledge.

Default is 1.

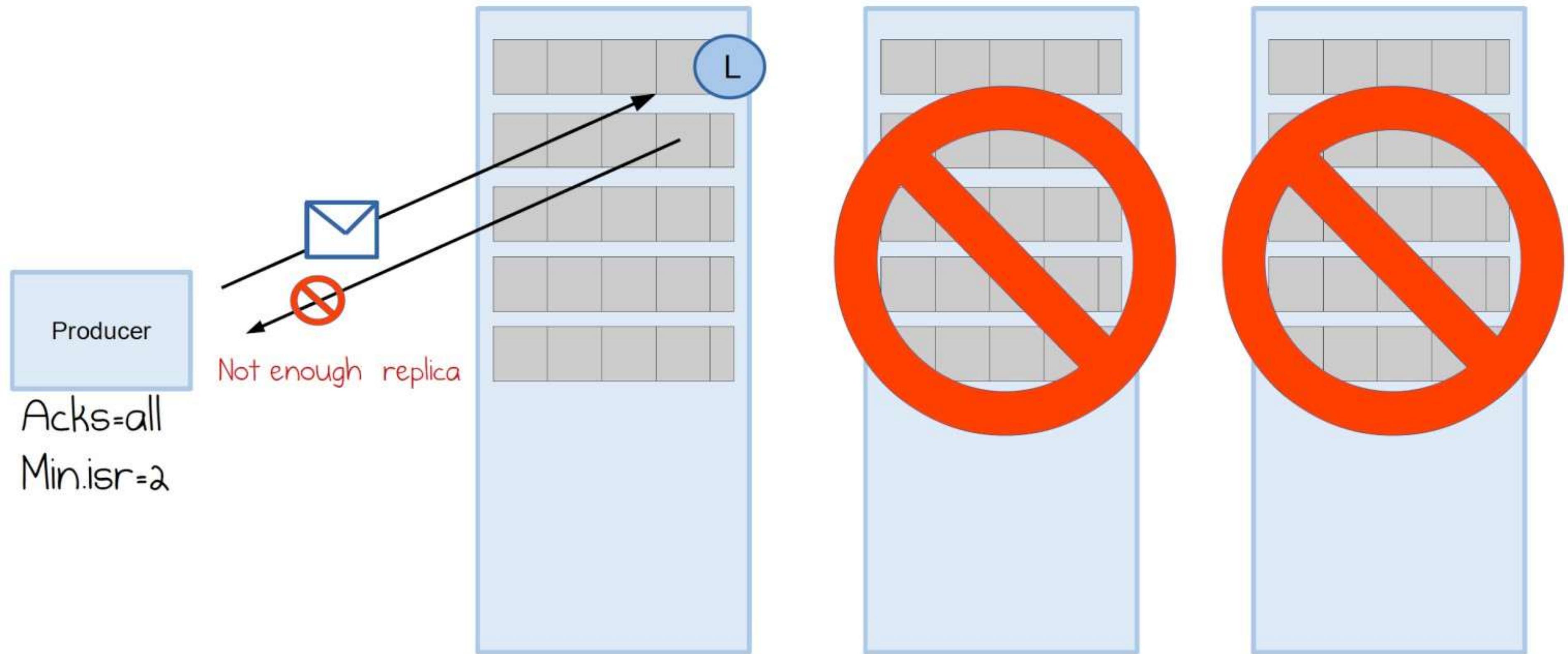
But only to the In-Sync Replicas...



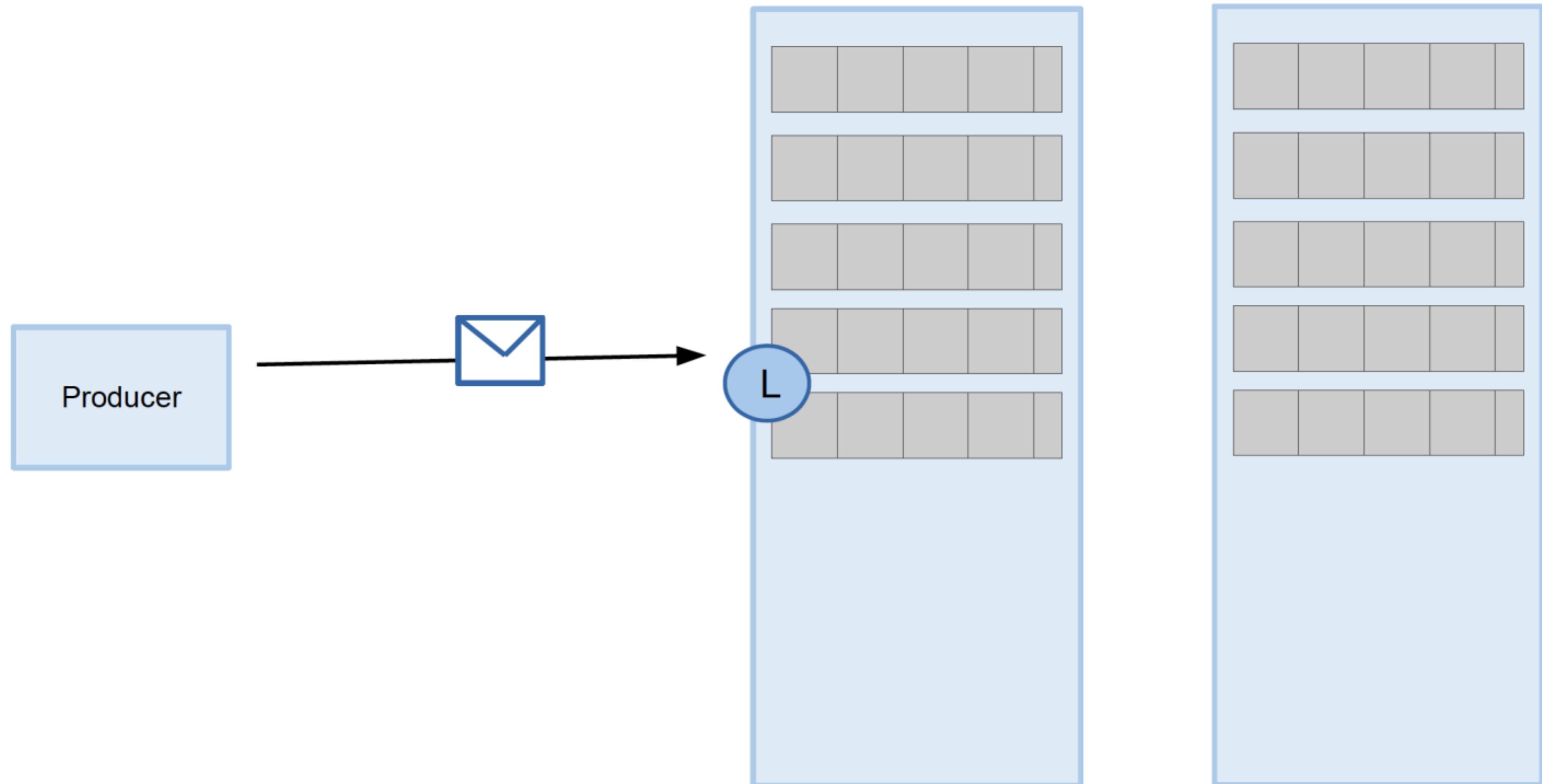
... which could be only one server



... which could be only one server

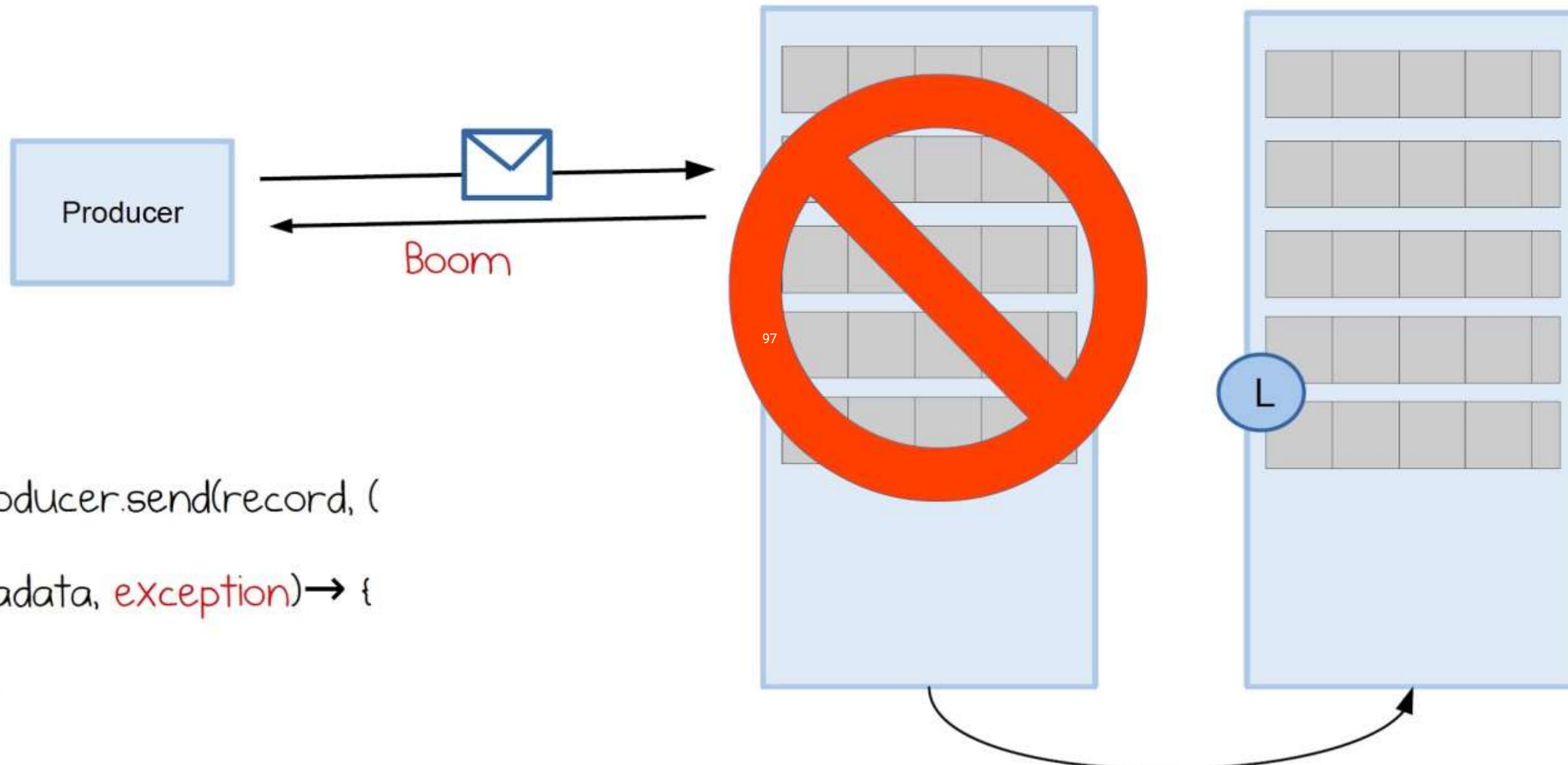


What's happening in case of issue ?



What's happening in case of issue ?

```
kafkaProducer.send(record, (metadata, exception) → {  
    ...  
})  
)
```



The leader moved to a
different broker

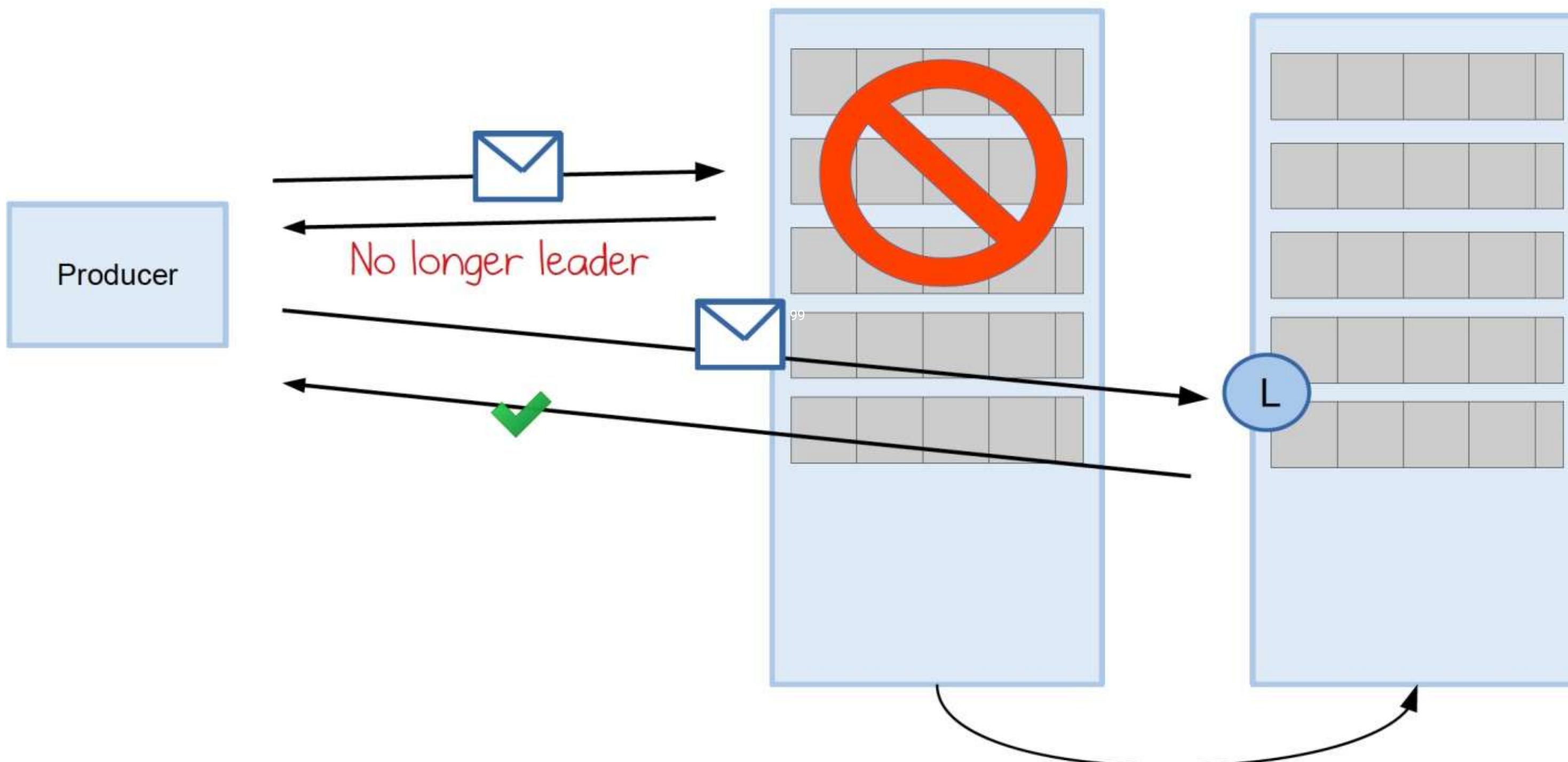
Parameter:
retries

It will cause the client to resend any record whose send fails with a potentially transient error.

98

Default value : 0

What's happening in case of issue with retry ?



The leader moved to a
different broker

Parameter:

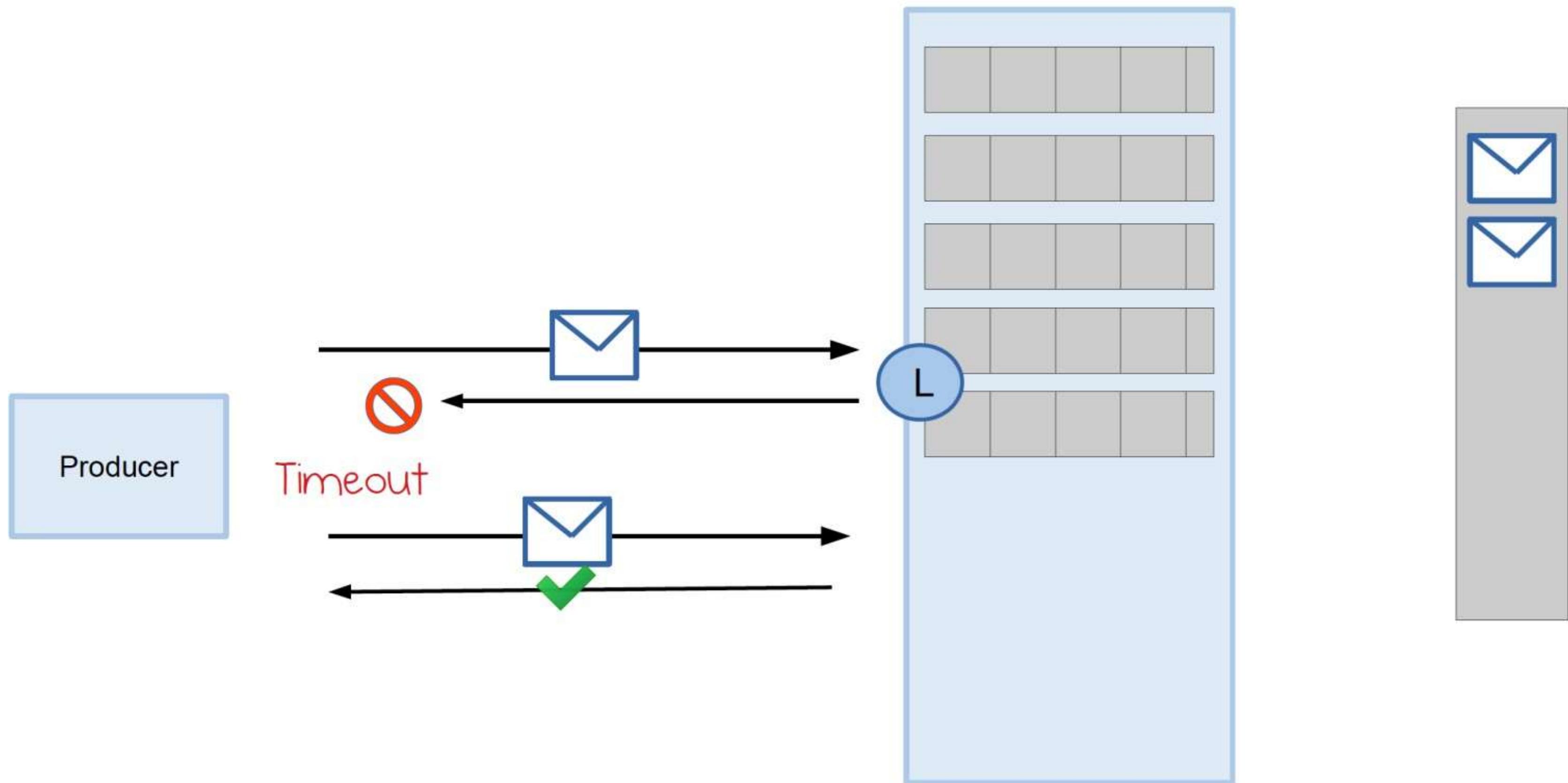
retries

Use built in retries !

Bump it from 0 to **infinity!**

And another set of issues

Message duplication



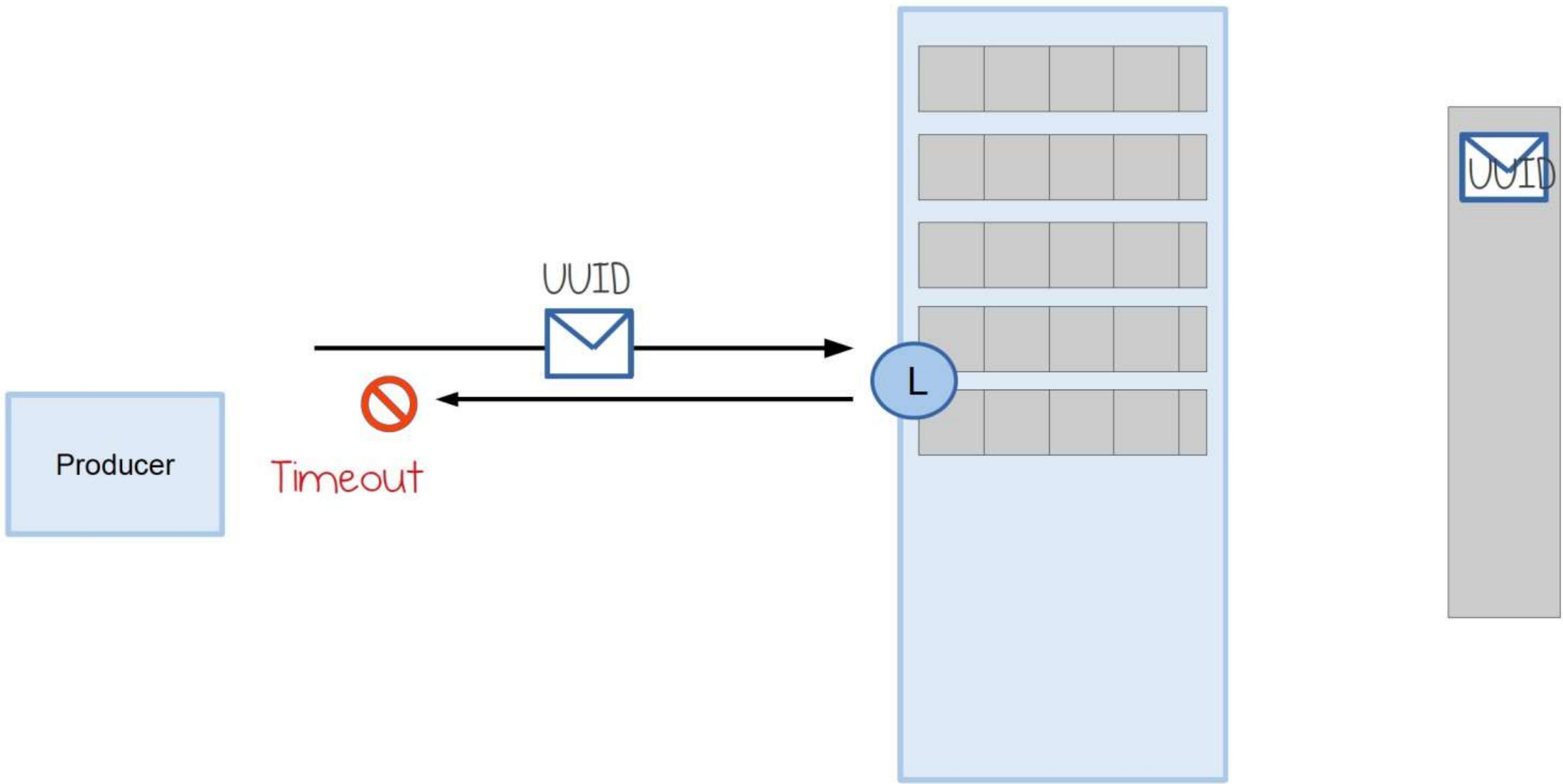
Parameter:
enable.idempotence

10
2

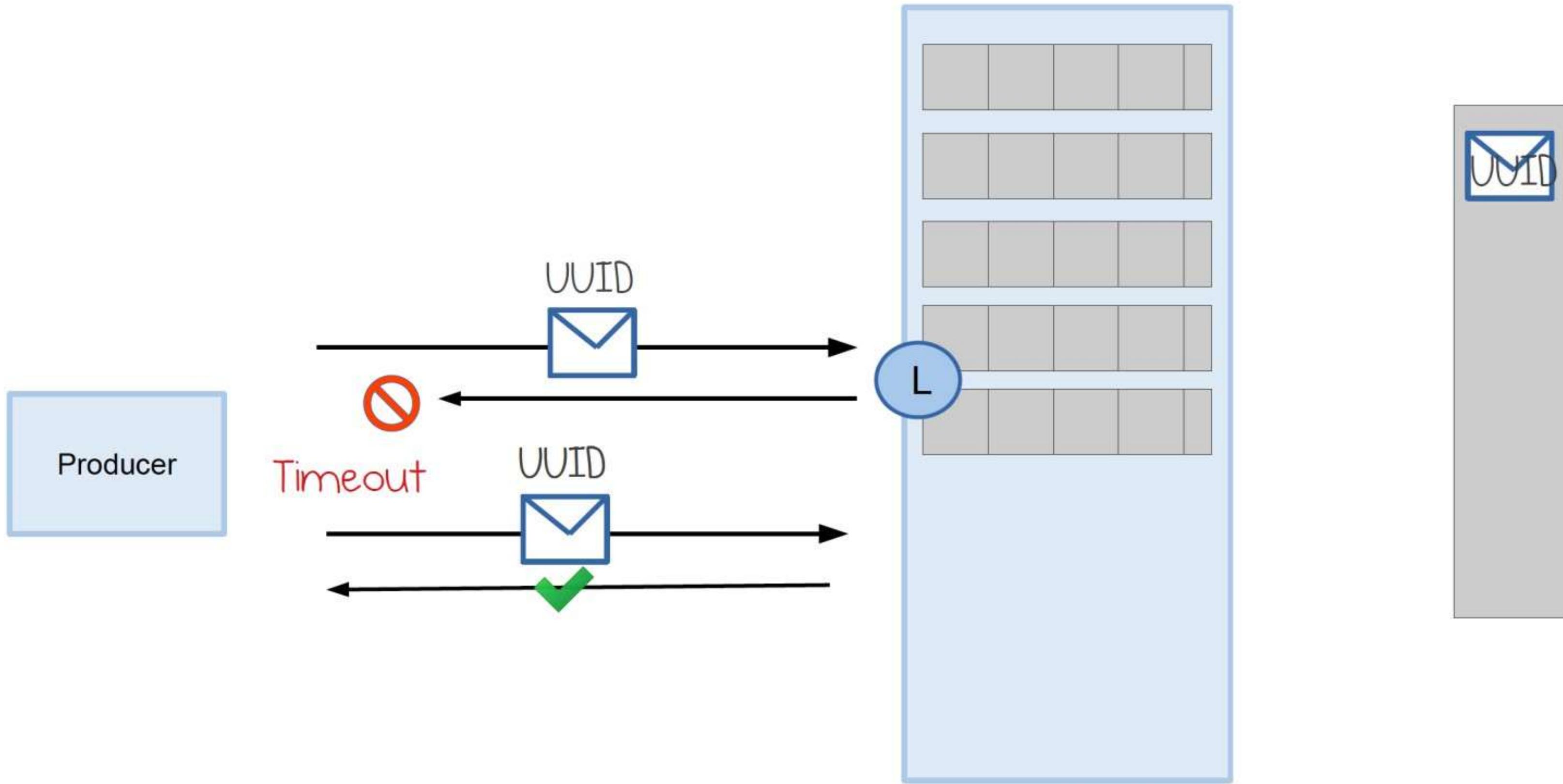
When set to 'true',
the producer will ensure
that exactly
one copy of each
message is written.

Default value: false

With Retries and Idempotency



With Retries and Idempotency



Infinite Retries

```
properties.put(ProducerConfig.RETRIES_CONFIG, Integer.MAX_VALUE);
```

Other Strategies

- Write to DLQ and continue
- Ignore and Continue
- ***Neither is appropriate – Handle all exceptions without fail. Varies from case to case.***
- ***Always Remember – Anything that can go wrong will !***

Test your understanding

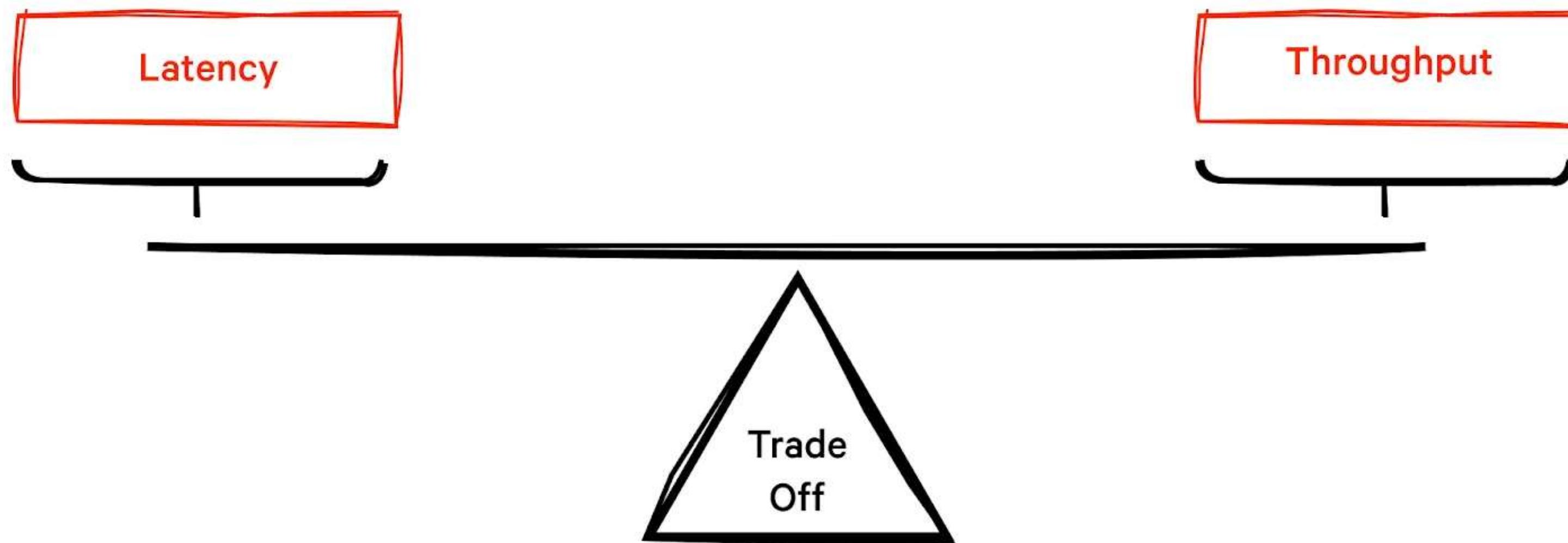
Let us assume your min in sync replica is set to 2

You have only 2 replicas. What's the behavior?

Now answer this when you have only 1 replica.

How does acks setting affect the behavior?

Producers, Consumers, Replicas



defaults

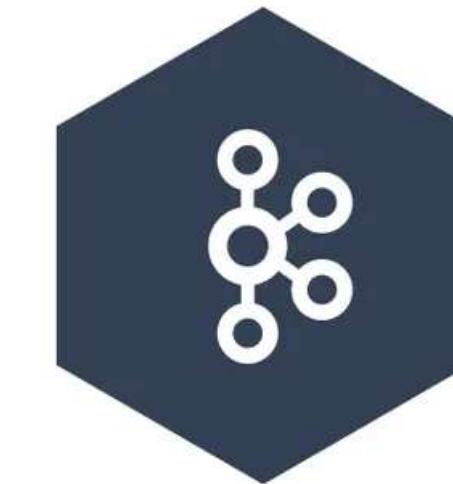
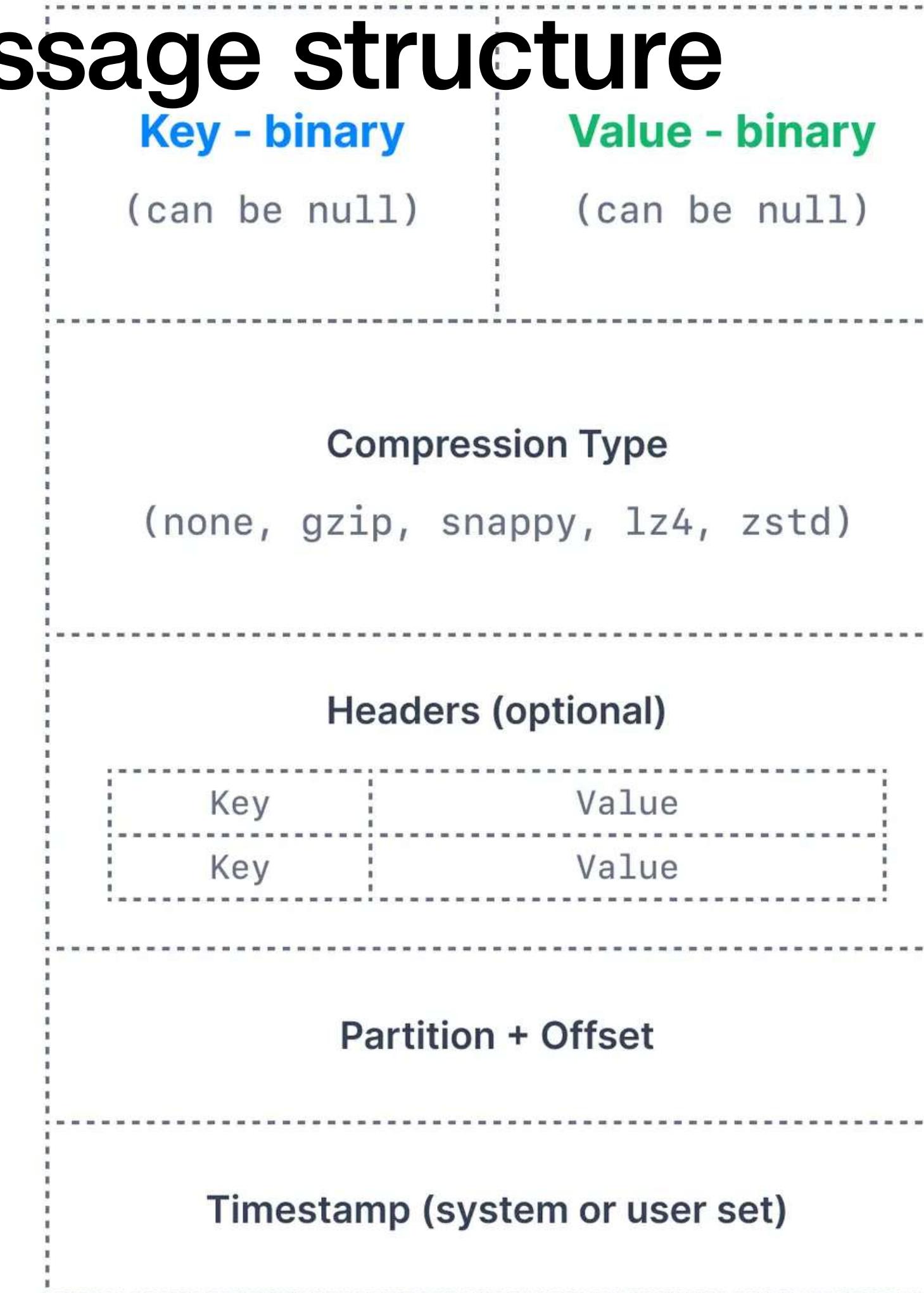
The default values are optimized for availability & latency.

10
9

If durability is more important, tune it!

Sample message structure

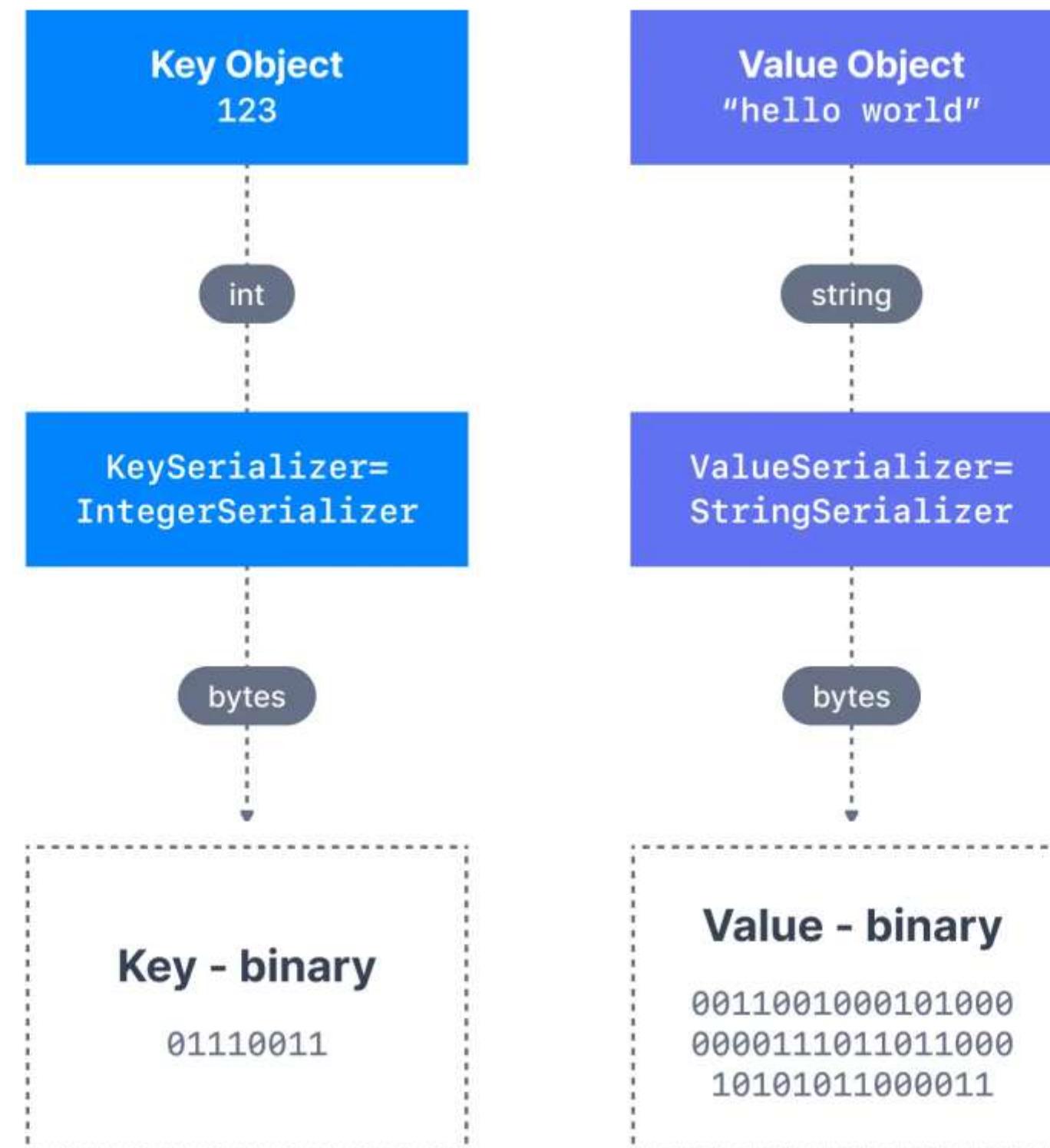
Kafka
Message
Created by
the producer



Messages

- Key+Value are really objects ☺
- They are in essence passed as byte[]
- Serialize/Deserialize

Messages



Messages

- Several serializers already exist, such as string (which supersedes JSON), integer, float.
- Other serializers may have to be written by the users, but commonly distributed Kafka serializers exist and are efficiently written for formats such as
 - JSON-Schema,
 - Apache Avro
 - and Protobuf, thanks to the Confluent Schema Registry.

Messages

- So how is a key Hashed?



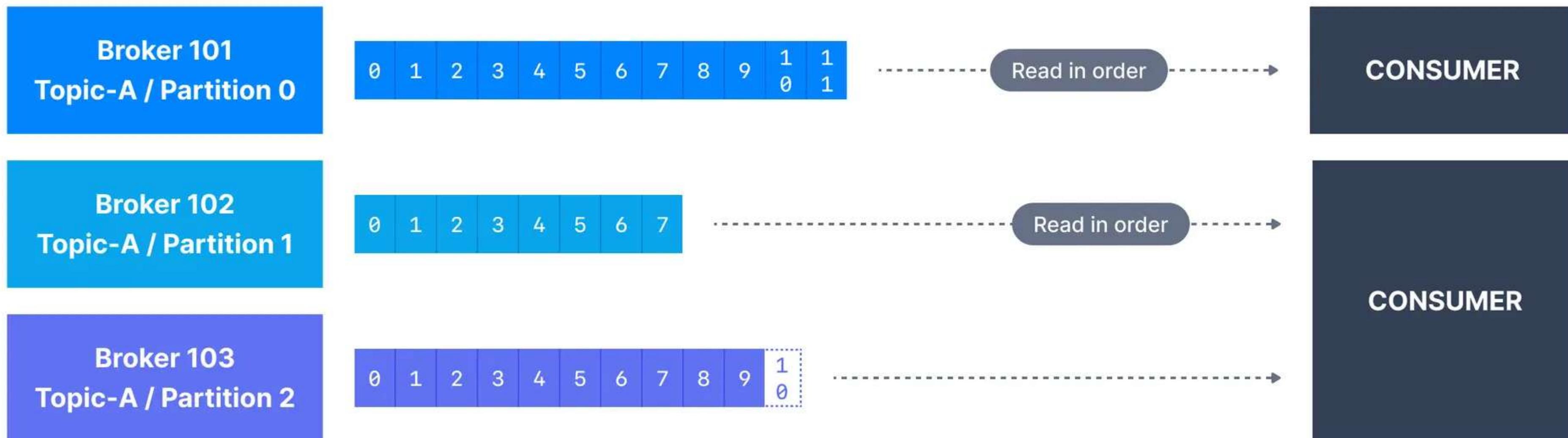
Messages

- Default uses an algorithm : murmur2 algorithm,
- `targetPartition = Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1)`
- <https://murmurhash.shorelabs.com/>
- <https://md5hashing.net/hash/murmur3/>

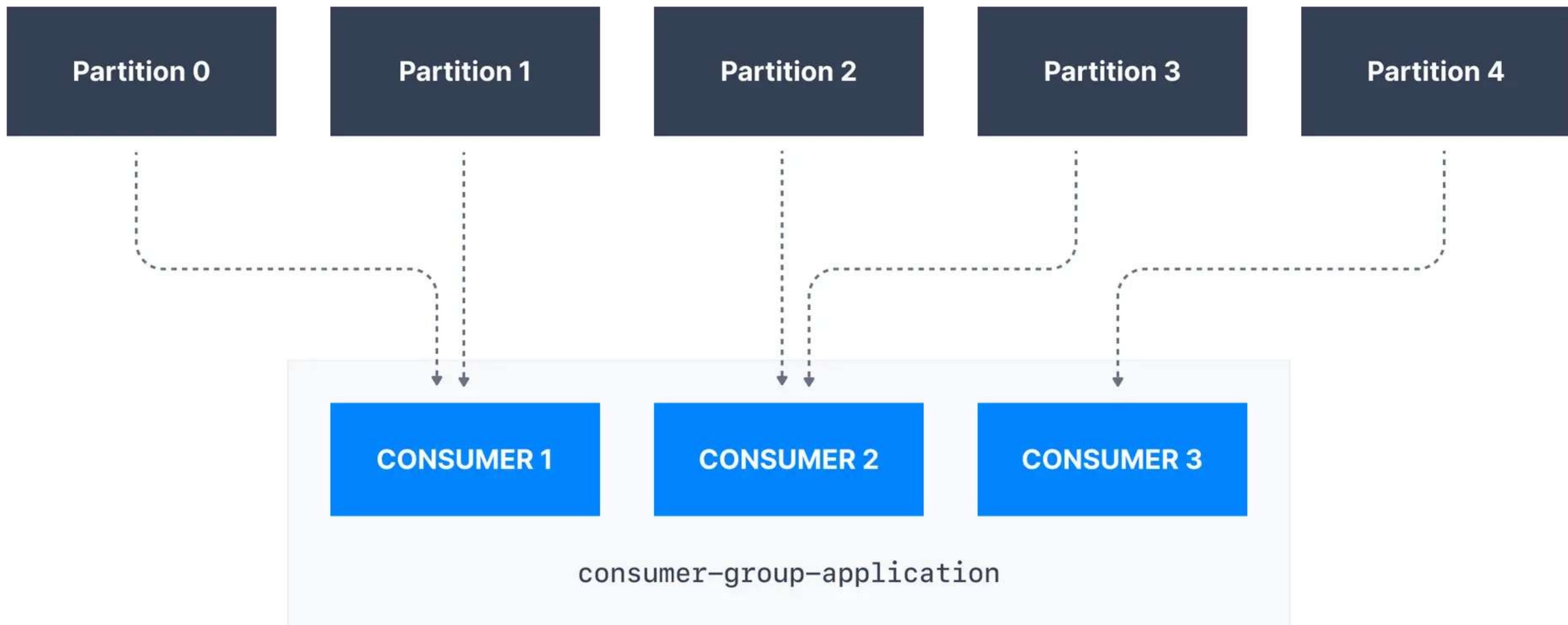
Consumers

Producers, Consumers, Partitions

- Consumers are apps that read from partitions.



Producers, Consumers, Partitions



Consumers

- Consumer offsets (not to be confused with message offsets) are used to “checkpoint” how far the consumer has been reading the partition.
- Consumers frequently commit the latest processed message – consumer offsets
- **The commit is usually turned on by default.**
- Data for this can be found in the `_consumer_offsets` topic.

Producers, Consumers, Partitions

- Consumers cannot read backwards.
- Data read from multiple partitions is not guaranteed to be in orders.
- By default, consumers read only data after they first connect to Kafka.
- Issues – Data serialized by producers have to be deserialized in the same format.
- Format should not change during topic life cycle. If you do need to do that, consider moving to new topics

Producers, Consumers, Partitions

- Consumers that are part of the same application and therefore performing the same "logical job" can be grouped together as a Kafka consumer group.
- A topic usually consists of many partitions. These partitions are a unit of parallelism for Kafka consumers.
- The benefit of leveraging a Kafka consumer group is that the consumers within the group will coordinate to split the work of reading from different partitions.

Consumers

- 2 common issues
 - A. Consumer lag – how far behind the actual message header
 - B. Crash Recovery
- The commit behavior controlled by
 - **enable.auto.commit = true|false**
 - **auto.commit.interval.ms (default 5 s)**

Consumers

- Depending on these, delivery to Customer is
- At Most once – Offsets are committed as soon as message is received. Processing fails, message is gone 😊
- At least once (preferred) – Offsets are committed after message processing. If failure, message is re-read with possible duplicates. → Commit anyways and send to DLQs
- Exactly Once – Topic to Topic Workflow using the **Transactions API**. In Kafka Streams, set ***processing.guarantee=exactly_once_v2***

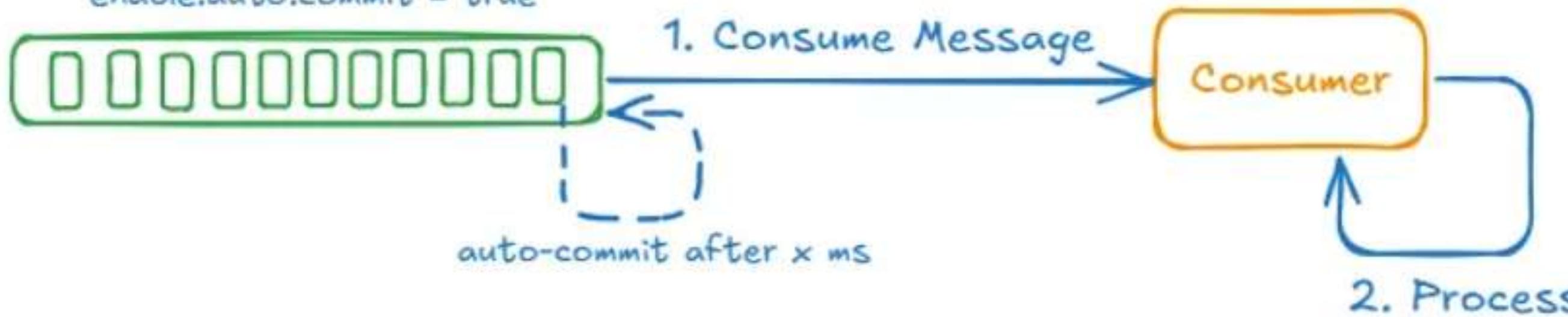
Offset Commits

- 4 ways to commit offsets
- Method #1: Auto-Commit (Default after every 5 seconds)
 - Commits the largest offset returned by poll()
- Method #2: Manual Sync Commit
- Method #3: Manual Async Commit
- Method #4: Commit Specific Offsets

Consumers

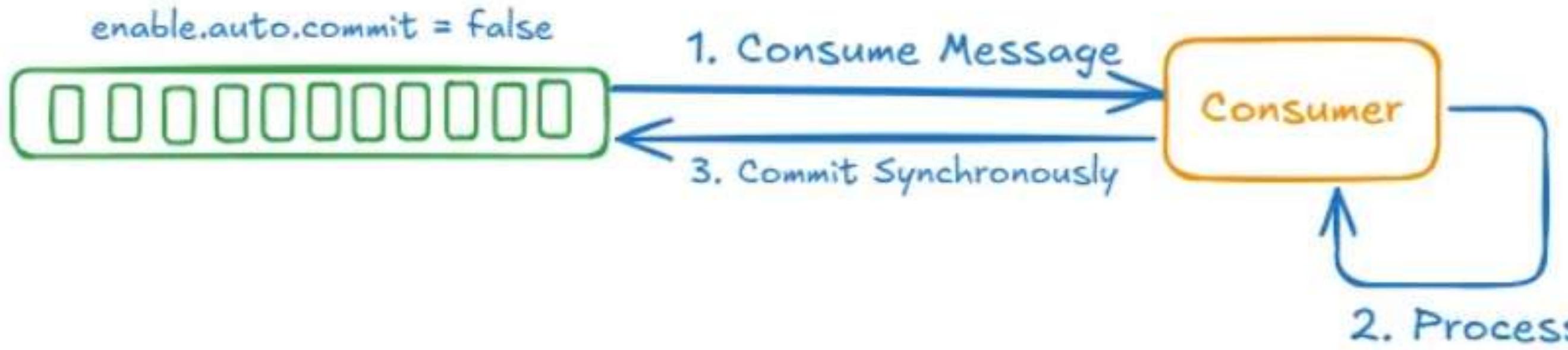
At Most Once (Zero or Once Delivery)

enable.auto.commit = true



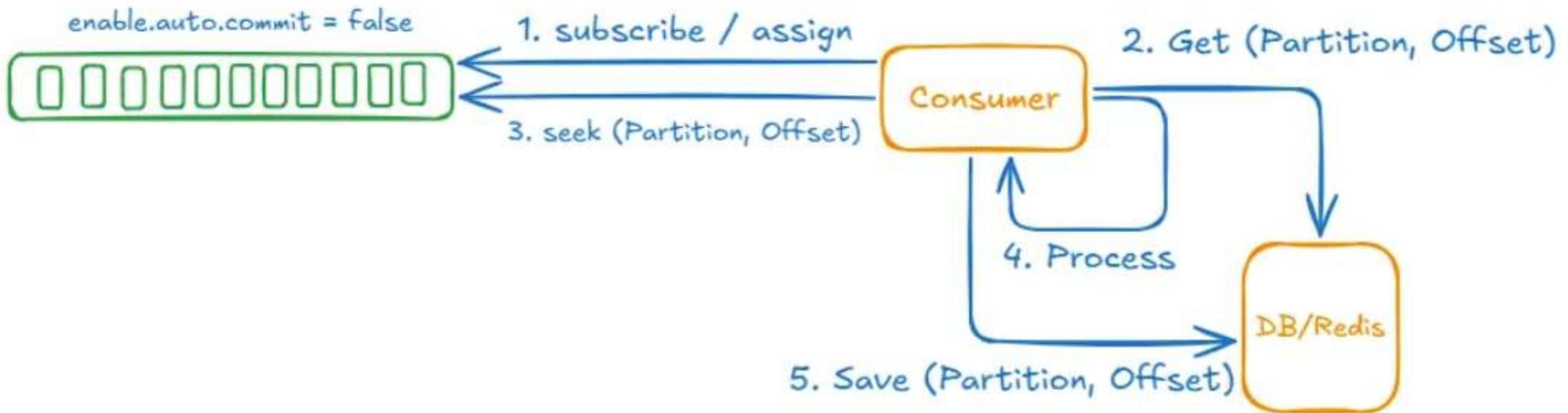
Consumers

At Least Once (One or More Deliveries)



Consumers

Exactly Once (One and Only One Delivery)



Consumers – At Most once (0 or more delivery)

- Messages may be lost but will never be re-delivered
- Use when
 - Losing Data is OK
- How to:

`enable.auto.commit = true`

`auto.commit.interval.ms` → keep it low for frequent auto-commits

Do **NOT** call `consumer.commitSync()` in your app — Kafka handles it.

Consumers – At least once Delivery

- Messages may potentially be delivered more than once.
- Your App. Logic should handle this.
- Use when
 - Data loss is not acceptable
- How to:

`enable.auto.commit = false (OR)`

`Keep enable.auto.commit = true but set auto.commit.interval.ms higher.`

Manually commit offsets after processing each message:

129 `consumer.commitSync();`

Consumers – Exactly once Delivery

- Use when
 - Data loss is not acceptable
 - You need no loss no duplicates
- There are 2 ways to do this
 - Dynamic Group Memberships
 - Static Assignment

Consumers – Exactly once Delivery

- Dynamic Membership

`enable.auto.commit = false`

Register using `consumer.subscribe()` to join a group.

On startup, use `consumer.seek()` to start at a specific offset.

Store the message and offset together atomically in your database or storage — so if you replay, you skip what's already processed.

Do not call `commitSync()` after processing.

Consumers – Exactly once Delivery

- **Static Partition Assignment**

`enable.auto.commit = false`

Register using `consumer.assign()` for exact partitions.

On startup, use `consumer.seek()` to start at a specific offset.

Store the message and offset together atomically in your database or storage — so if you replay, you skip what's already processed.

Do not call `commitSync()` after processing.

Consumers – commitAsync()

- **High Throughput Applications:**
 - Systems prioritizing performance, such as analytics or logging.
- **Non-Critical Applications:**
 - Occasional duplicate processing is acceptable.
- **Performance-Critical Scenarios:**
 - Reduces latency caused by blocking calls.
- **Frequent Offset Commit:**
 - Suitable when offsets need frequent updates.
- **Advantages**
- High performance with non-blocking behavior.
- Ideal for low-latency requirements.
- **Disadvantages**
- No guarantee of successful offset commits.
- Requires additional error handling.

Consumers – sync vs async

Criteria	Use commitSync	Use commitAsync
Consistency Requirements	Critical for consistency	Consistency is less critical
Throughput	Lower throughput acceptable	High throughput required
Message Importance	Loss of messages is unacceptable	Occasional duplicate processing okay
Application Type	Financial transactions, critical data	Analytics, logging, non-critical data

Sample code can be found

¹⁰⁴ <https://github.com/seshagirisriram/apache-kafka>

Consumer Configs

- Consumer Tuning
 - Fetch Size
 - Max Poll Records
 - Client Side Buffering
 -
- There are too many to consider. Please read the Manuals 😊
- We will cover only the “more important” ones here.

Consumer Configs

Configuration	Recommended Value	Purpose / Impact
<u>fetch.min.bytes</u>	50000 or higher	Wait until this amount of data is available before fetching. Improves batch efficiency.
<u>fetch.max.bytes</u>	52428800 (50MB)	Max data per fetch request. Larger values reduce network round-trips.
<u>max.poll.records</u>	500 to 1000	Controls batch size per poll. Larger batches improve throughput.
<u>max.poll.interval.ms</u>	300000 (5 minutes)	Must be high enough to allow time for batch processing before consumer is considered dead.
<u>session.timeout.ms</u>	10000	Time to detect consumer failure. Lower values improve rebalancing speed.

Consumer Configs

Configuration	Recommended Value	Purpose / Impact
heartbeat.interval.ms	3000	Frequency of heartbeats. Should be lower than session.timeout.ms .
enable.auto.commit	false	Manual commit gives control over when offsets are committed (after successful processing).
auto.offset.reset	latest	Start from latest offset unless no committed offset exists.
receive.buffer.bytes	65536 or higher	TCP socket buffer size. Larger buffers can improve throughput.
client.id	Unique per consumer	Helps with monitoring and debugging.
request.timeout.ms	30000	Timeout for requests to Kafka brokers.
connections.max.idle.ms	540000	Keep connections alive longer to reduce reconnect overhead.

Consumer Configs

- **Num.consumer.fetchers = Number of threads (usually 4).**
- The above can increase parallelism but use with care.

Some Notes

Why Idempotence?



- **Duplicate-Free Delivery:**
 - Ensures messages are delivered exactly once, even during retries.
- **Simplified Error Handling:**
 - Eliminates the need for custom deduplication logic in applications.
- **Increased Reliability:**
 - Improves consistency and reliability in distributed systems.
- **Foundation for Transactions:**
 - Idempotence is a prerequisite for Kafka's transactional messaging feature, enabling atomic writes across topics and partitions.

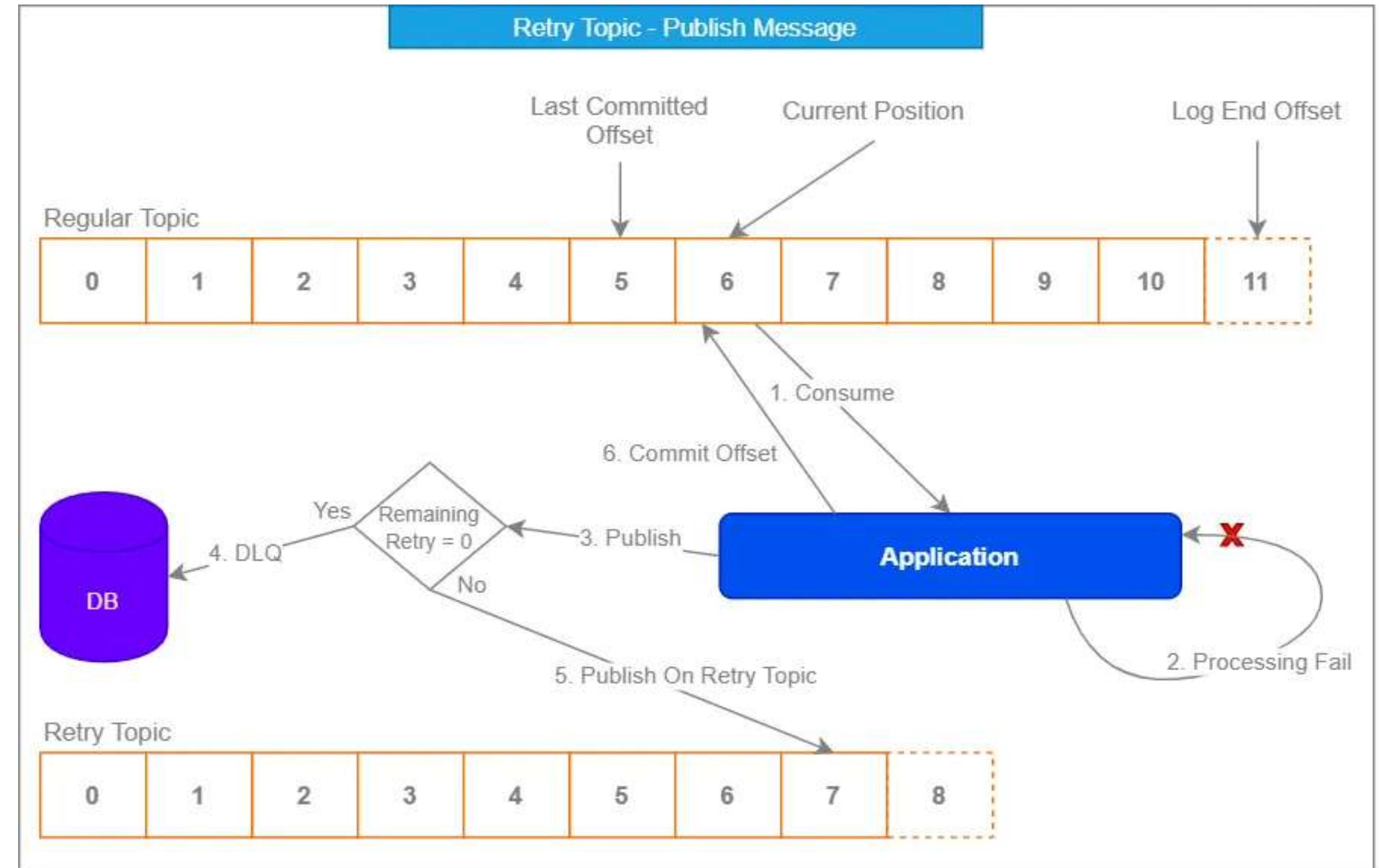
Producers, Consumers, Replicas



- **Partition-Specific Guarantee:**
 - Idempotence works at the partition level. Duplicate messages sent across different partitions are not deduplicated.
- **Does Not Cover Consumers:**
 - Idempotence guarantees apply to producer-to-broker communication. To achieve exactly-once semantics for consumers, you need to use Kafka's transactions feature.
- **Increased Latency:**
 - With `acks=all` and sequence tracking, latency may increase slightly compared to non-idempotent producers.

Resolving errors and duplicates

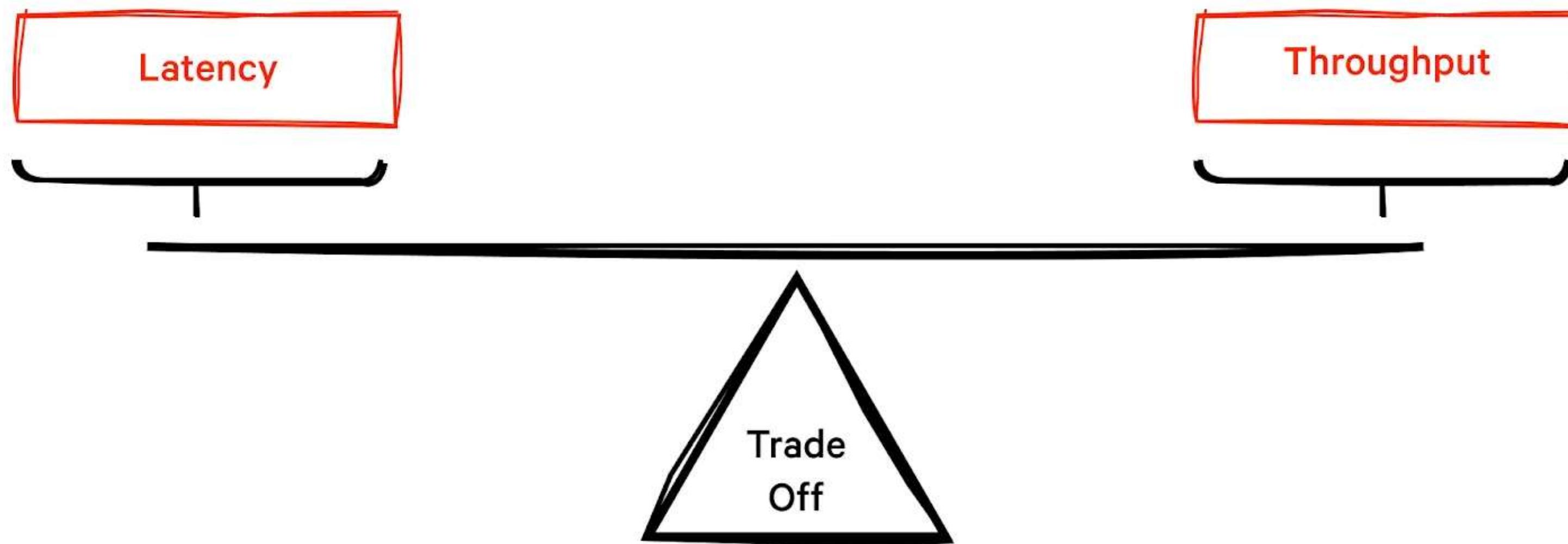
- One Pattern is the use of Retry Logics and DLQs.



Resolving errors and duplicates

- Retries Can be enriched with the following
 - Metadata about number of tries left
 - Metadata about consumer group id
 - Next timestamp
 - Exponential backoffs
 - Retry topic / topic (or single Retry Topic for all)
- Once no more retries are possible, put it into DLQ and be done.

Producers, Consumers, Replicas



Performance Tuning

- Latency measures how long it takes for Kafka to fetch or pull a single message.
- It is the time gap between the producer generating a message and the consumer consuming it.
- Low latency is critical for real-time applications, where delays in processing have significant consequences.
- To reduce latency, you optimize your Kafka configuration to minimize the time it takes for Kafka to process a single event. You can consider strategies like:
 - Tuning the number of partitions and replication factors
 - Optimizing the hardware and network configurations
 - Using compression to reduce the size of the data Kafka processes.

Performance Tuning

- Throughput measures how many messages Kafka can process in a given period.
- High throughput is essential for applications that process large amounts of data quickly.
- To maximize throughput, you optimize your Kafka configuration to handle as many events as possible within a given time frame.
- Strategies include:
 - Increasing the batch size
 - Increasing the number of producer threads
 - Increasing the number of partitions

Performance Tuning

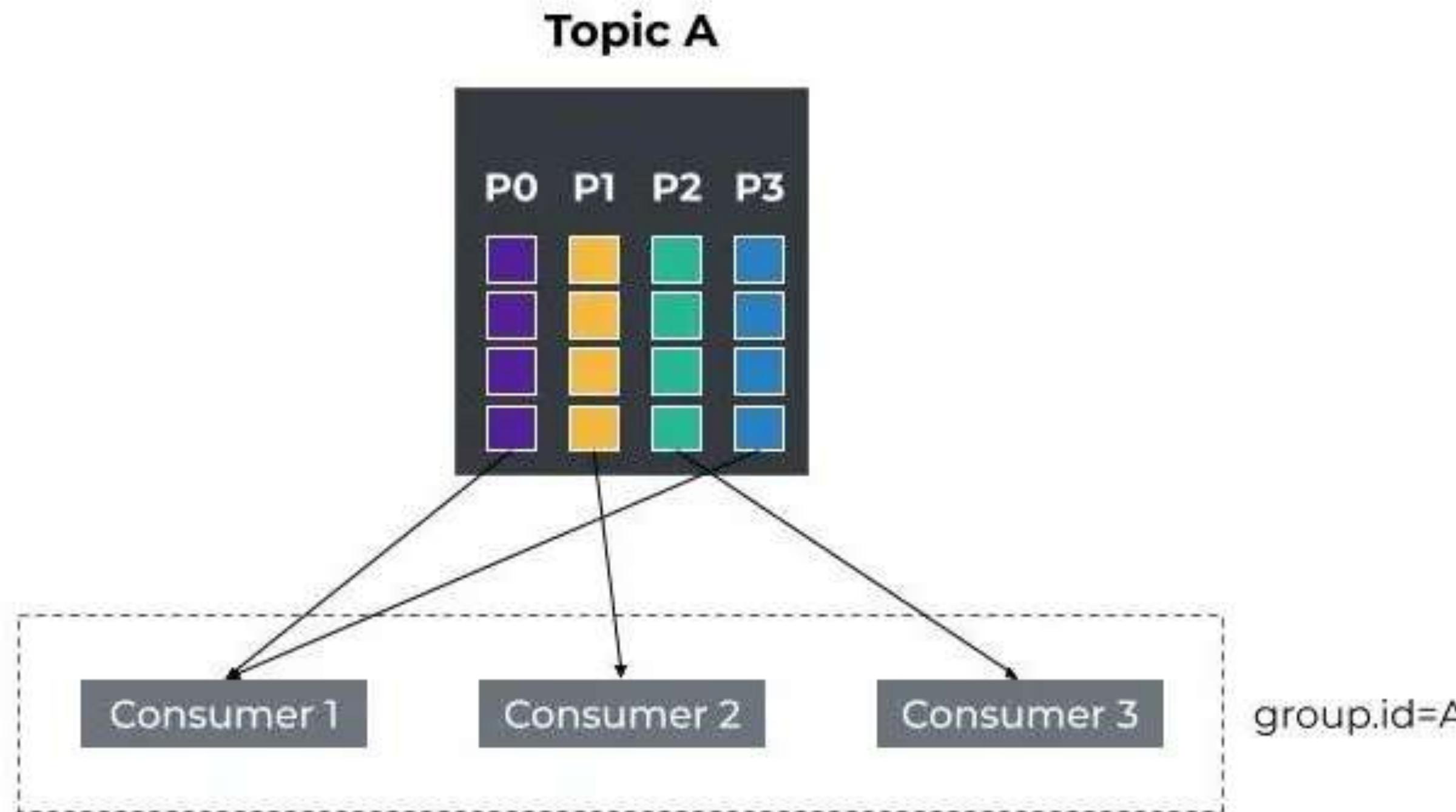
Compression type	Compression ratio	CPU usage	Compression speed	Network bandwidth usage
Gzip	Highest	Highest	Slowest	Lowest
Snappy	Medium	Moderate	Moderate	Medium
Lz4	Low	Lowest	Fastest	Highest
Zstd	Medium	Moderate	Moderate	Medium

Partitioning Strategies

KAFKA PARTITIONING STRATEGIES

- On the producer side, the partitions allow writing messages in parallel.
- If a message is published with a key, then, by default, the producer will hash the given key to determine the destination partition.
- This provides a guarantee that all messages with the same key will be sent to the same partition.
- In addition, a consumer will have the guarantee of getting messages delivered in order for that partition.

KAFKA PARTITIONING STRATEGIES



KAFKA PARTITIONING STRATEGIES

- ***Rebalance/Rebalancing***: the procedure that is followed by a number of distributed processes that use Kafka clients and/or the Kafka coordinator to form a common group and distribute a set of resources among the members of the group

(source : [Incremental Cooperative Rebalancing: Support and Policies](#)).

KAFKA PARTITIONING STRATEGIES

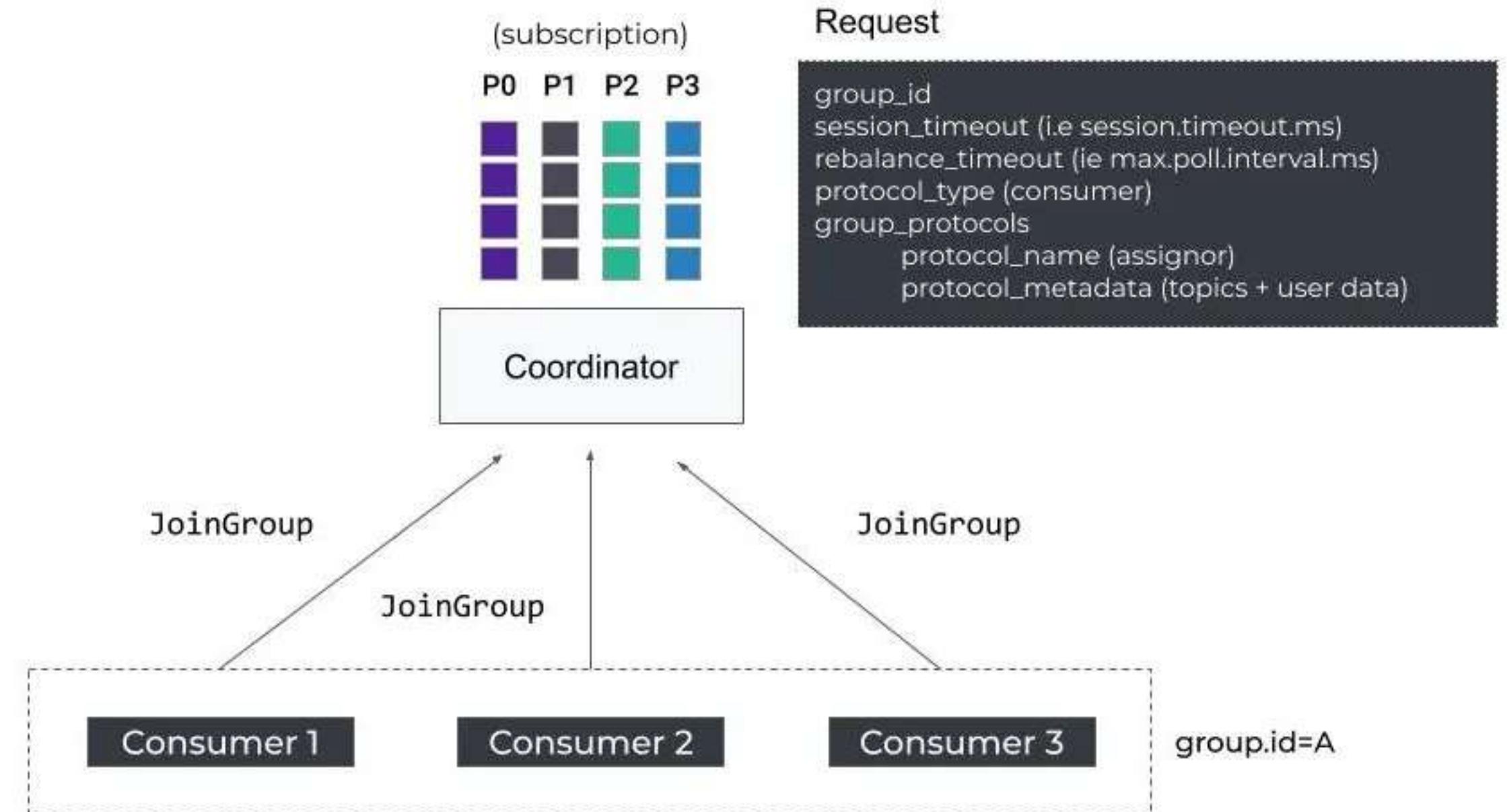


UNDERSTANDING

- The Group Membership Protocol is in charge of the coordination of members within a group.
- The clients participating in a group will execute a sequence of requests/responses with a Kafka broker that acts as coordinator.
- Client Embedded Protocols run on the client side.

KAFKA PARTITIONING STRATEGIES

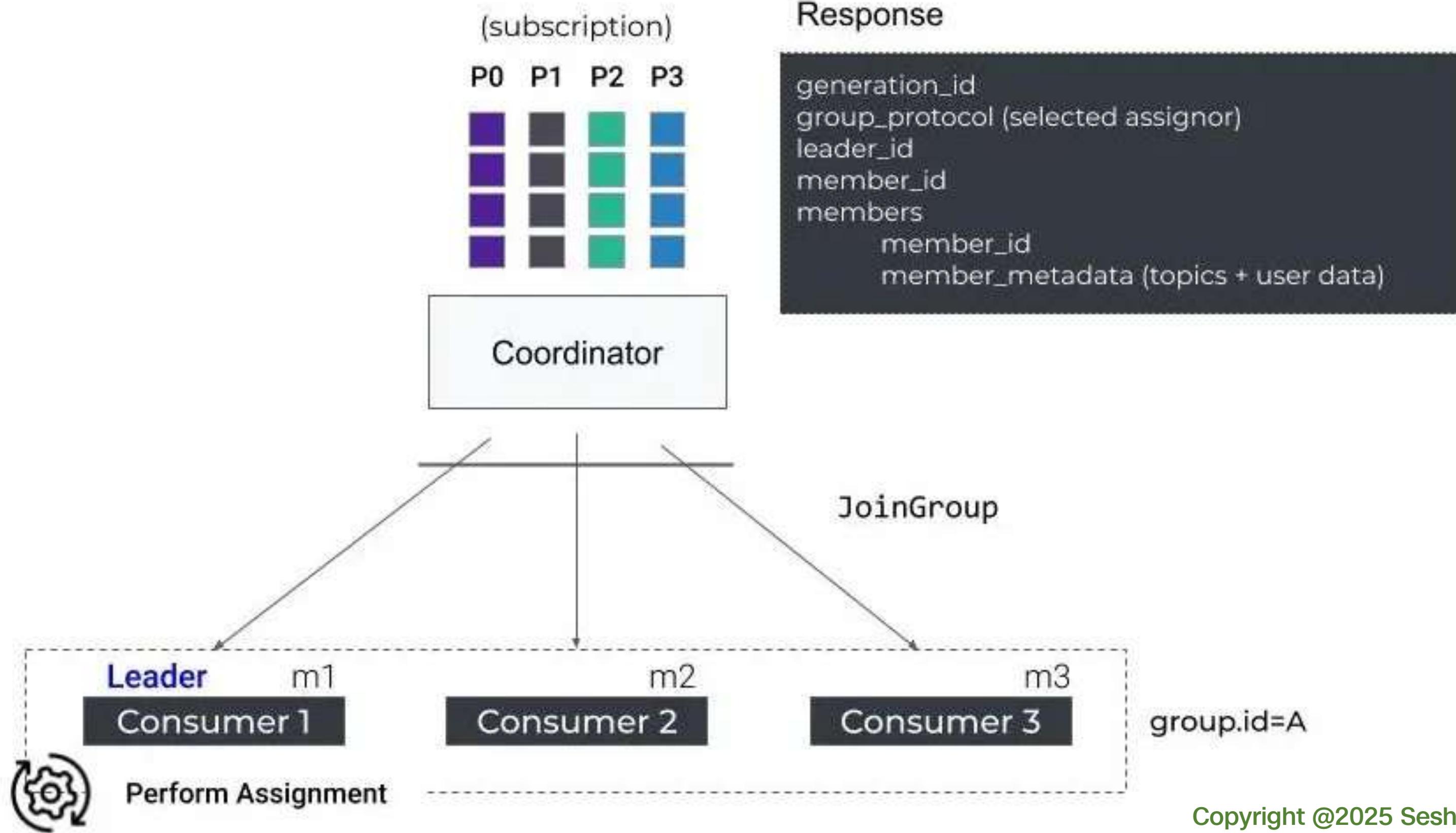
- A consumer sends the **FindCoordinator** request.
- It then sends a **JoinGroup** Request



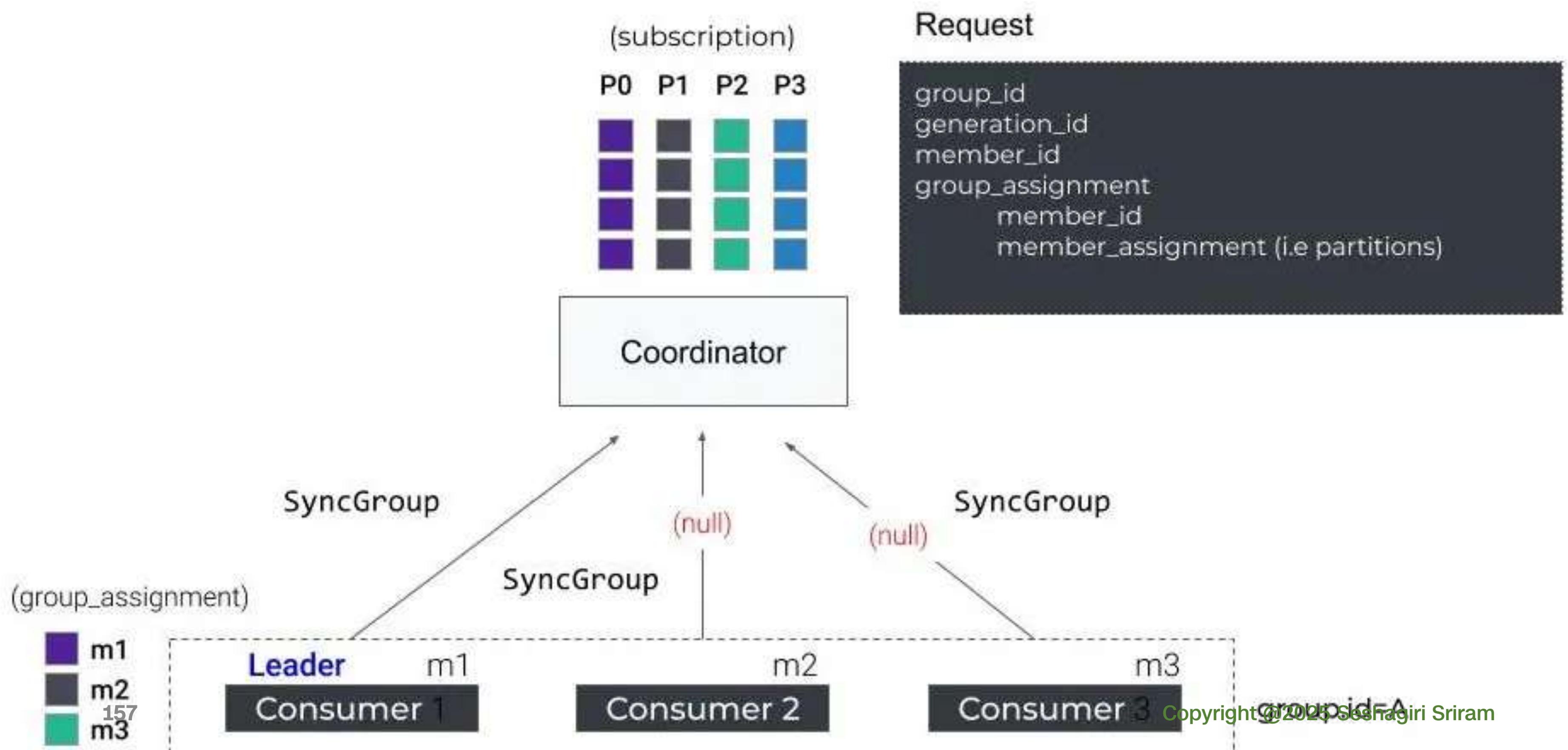
KAFKA PARTITIONING STRATEGIES

- JoinGroup contains Consumer client Configuration like timeout, Max Poll interval etc.
- It also contains
 - List of protocols e.g. Partitioning Strategy
 - Metadata e.g. List of Topics consumer has subscribed to
- Coordinator does not send response immediately.
 - Rebalance timeout (or) group.initial.rebalance.delay has to be exceeded.

KAFKA PARTITIONING STRATEGIES



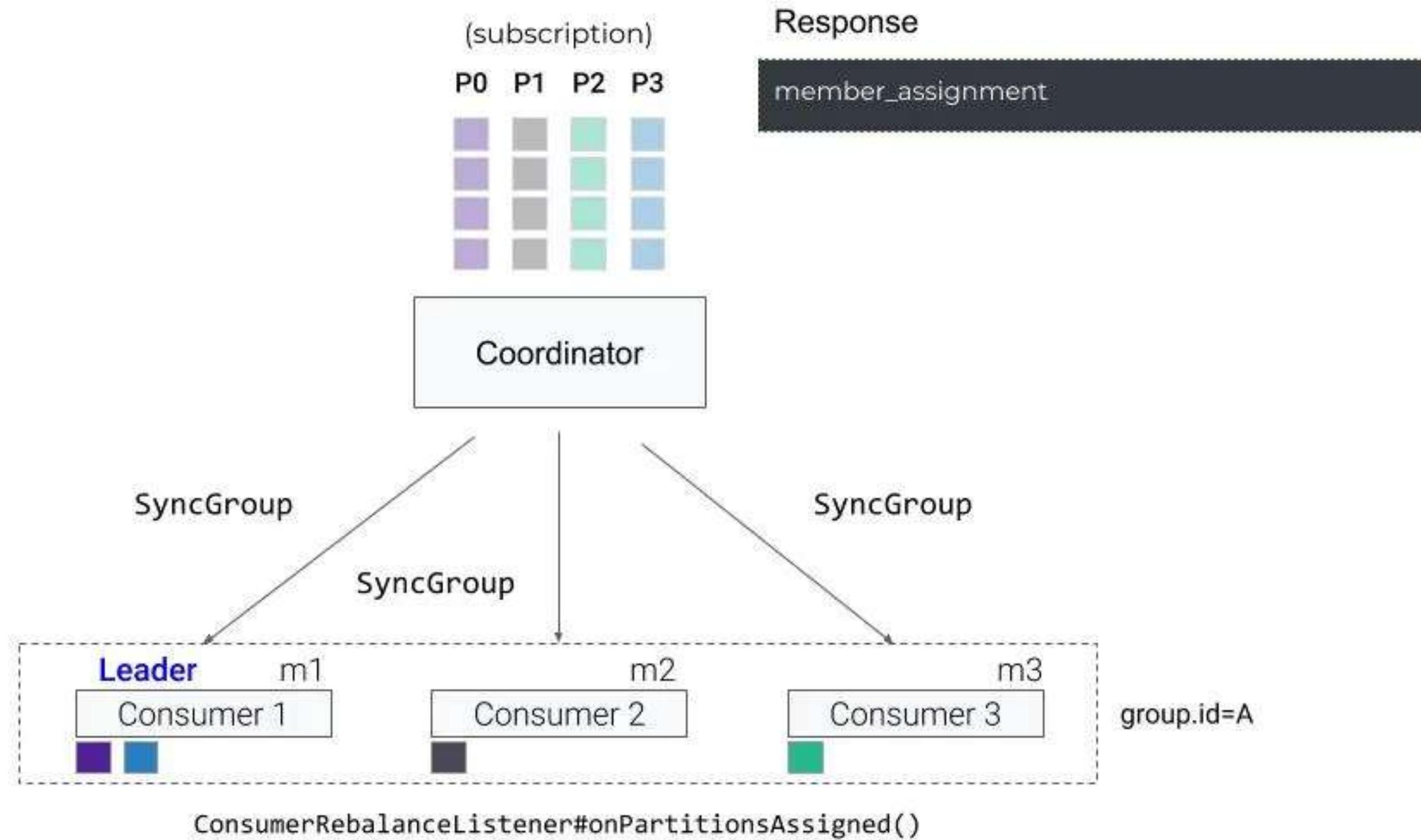
KAFKA PARTITIONING STRATEGIES



KAFKA PARTITIONING STRATEGIES

- Once Coordinator has received and responded to all SyncGroup Requests, each consumer get information on the assigned partition
- They start the `onPartitionAssignedMethod` on configured listener and fetch messages.

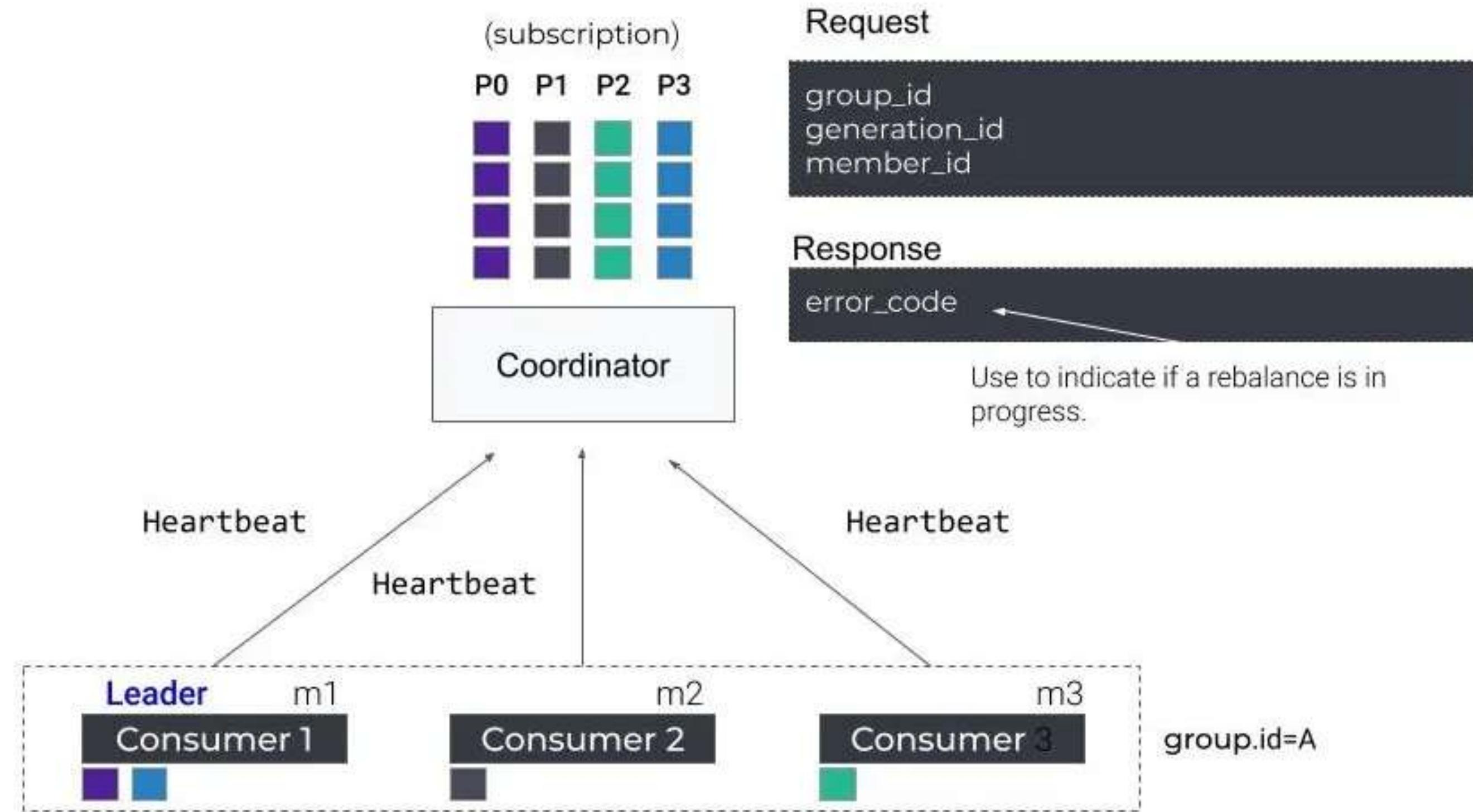
KAFKA PARTITIONING STRATEGIES



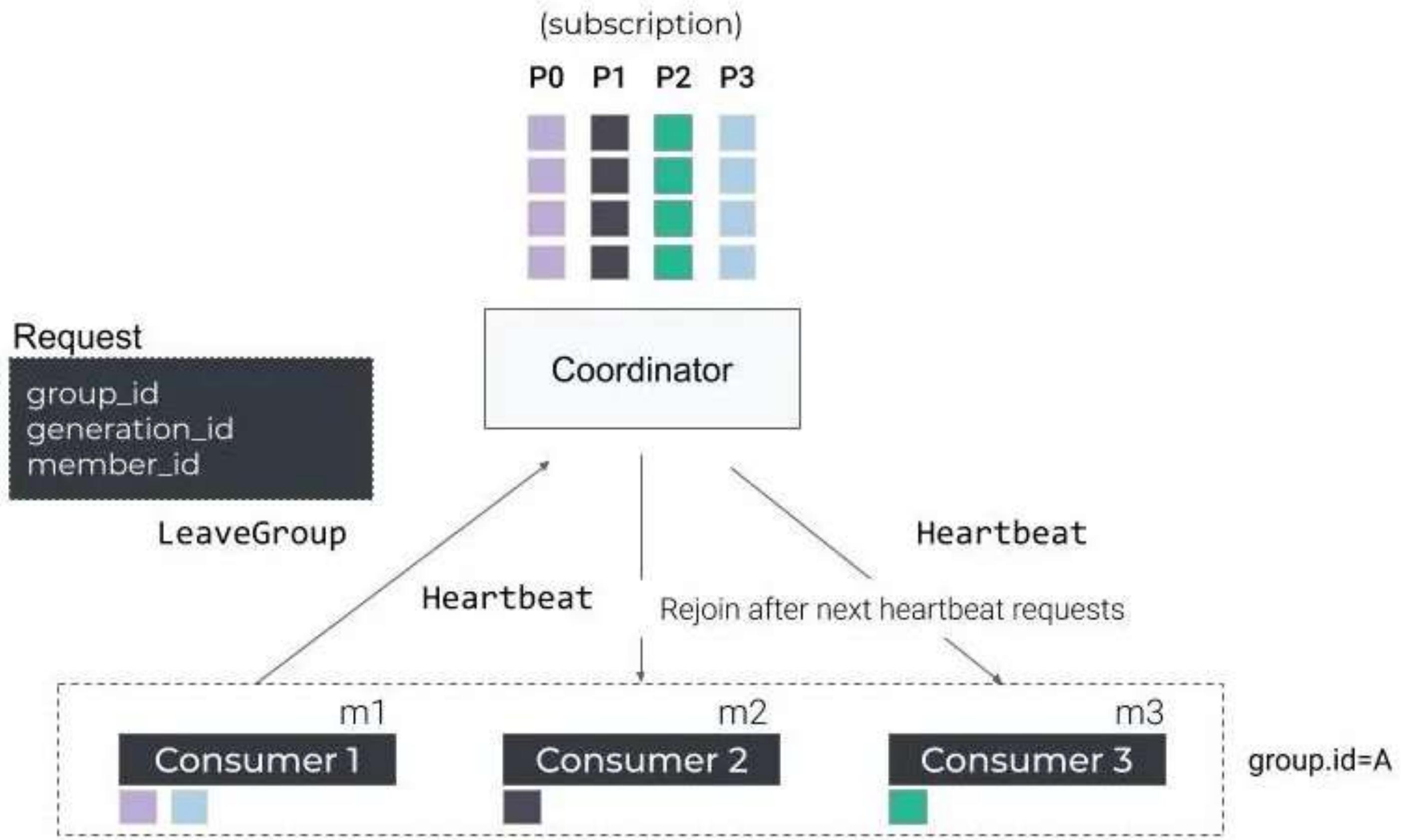
KAFKA PARTITIONING STRATEGIES

- Last but not least, each consumer periodically sends a Heartbeat request to the broker coordinator to keep its session alive (see : `heartbeat.interval.ms`).
- If a rebalance is in progress, the coordinator uses the Heartbeat response to indicate to consumers that they need to rejoin the group.

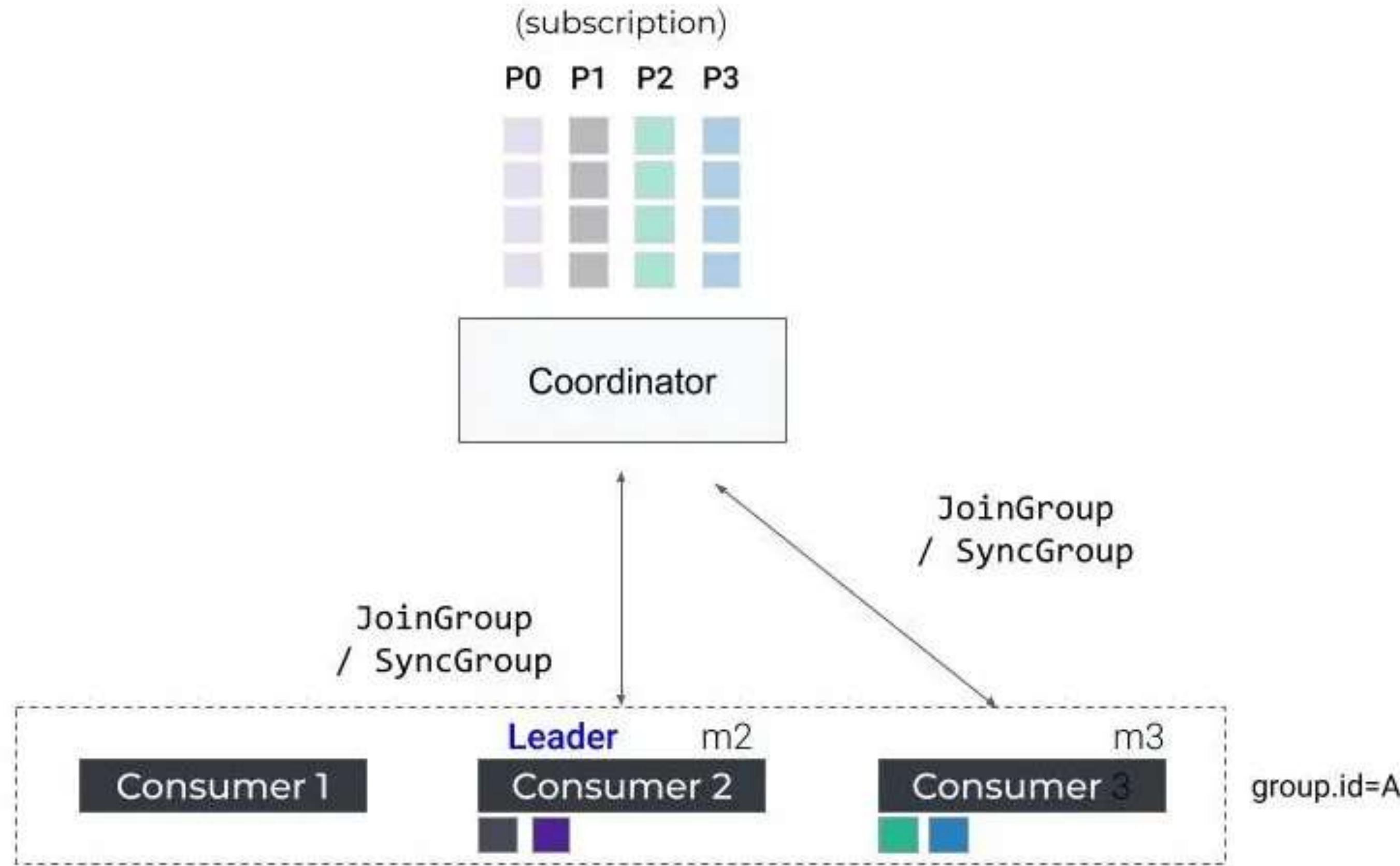
KAFKA PARTITIONING STRATEGIES



KAFKA PARTITIONING STRATEGIES



KAFKA PARTITIONING STRATEGIES



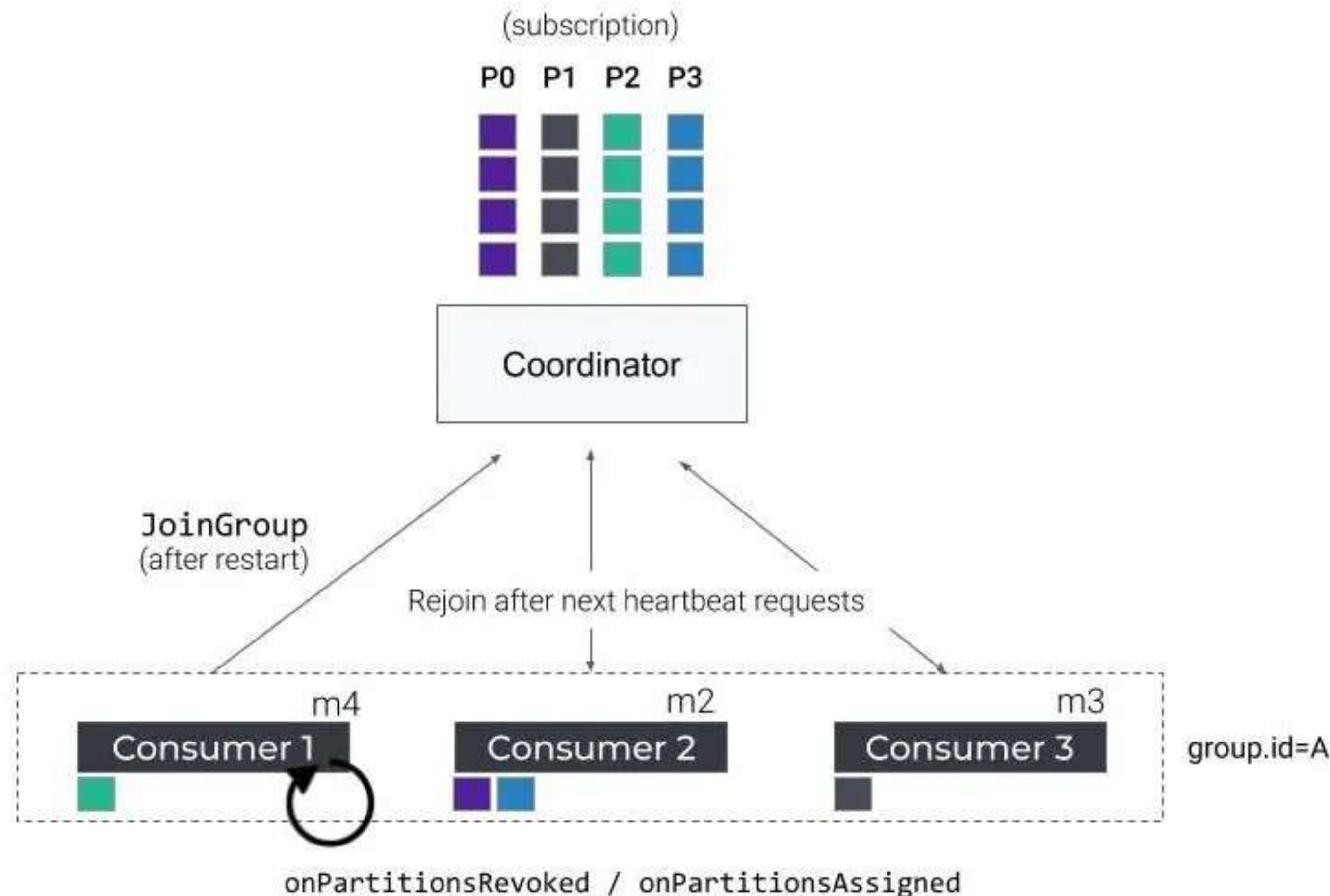
KAFKA PARTITIONING STRATEGIES

- During the entire rebalancing process, i.e. as long as the partitions are not reassigned, consumers no longer process any data.
- By default, the rebalance timeout is fixed to 5 minutes which can be a very long period during which the increasing consumer-lag can become an issue.

KAFKA PARTITIONING STRATEGIES

- Let's just assume it's a transient issue.
- Consumer fails->Restarts.
- Will Trigger Re-balance all over again.
- → Consumers will be stopped all over again.

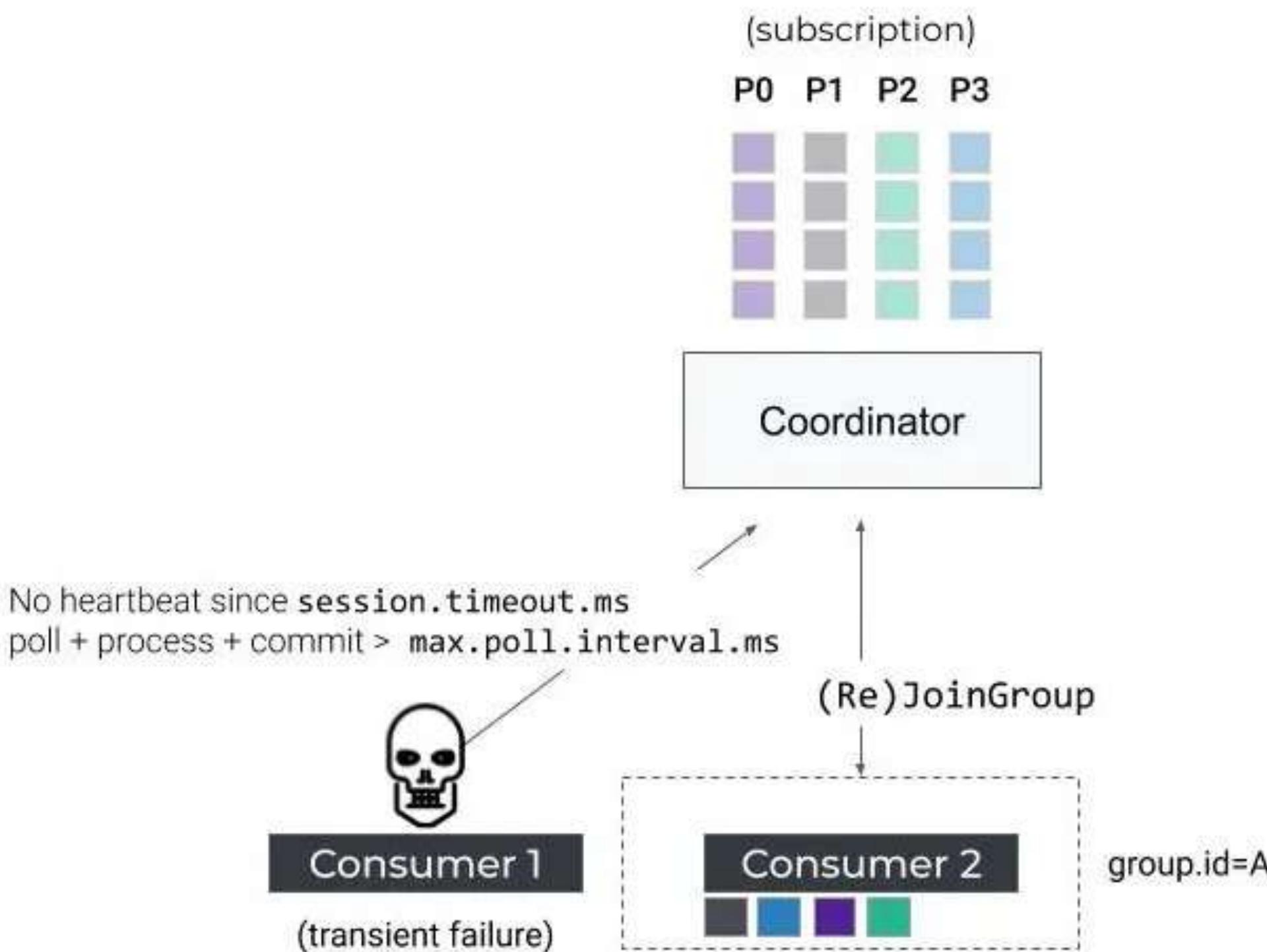
KAFKA PARTITIONING STRATEGIES



KAFKA PARTITIONING STRATEGIES

- A consumer group with n members will trigger 2^*n Rebalance Requests during a rolling upgrade.
- Other issues (in Java)
 - Missing Hearbeat requests
 - Long GC Pauses
 - Network Issues
 - Non-Invoking of KafkaConsumer#Poll() due to long processing times.
 - See: `session.timeout.ms` and `max.poll.interval.ms`

KAFKA PARTITIONING STRATEGIES



KAFKA PARTITIONING STRATEGIES

- Can be very useful for limiting the number of undesirable rebalances and thus minimizing stop-the-world effect.
- On the other hand, this has the disadvantage of increasing the unavailability of partitions because the coordinating broker may only detect a failing consumer after a few minutes (depending on `session.timeout.ms`).
- Unfortunately, this is the eternal trade-off between availability and fault-tolerance you have to make in a distributed system.

KAFKA PARTITIONING STRATEGIES

- Strategies for partitioning = = `partition.assignment.strategy`
- E.g.

```
Properties props = new Properties();
...
props.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,
StickyAssignor.class.getName());
```

```
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
//...
```

KAFKA PARTITIONING STRATEGIES

- All Consumers in group must have same strategy
- This property accepts a comma-separated list of strategies.
- For example, it allows you to update a group of consumers by specifying a new strategy while temporarily keeping the previous one.
- As part of the Rebalance Protocol the broker coordinator will choose the protocol which is supported by all members.

KAFKA PARTITIONING STRATEGIES

- 4 Built in

- Range
- Round Robin
- StickyAssignor
- CooperativeStickyAssignor balancing

Default if none is specified.

Like StickyAssignor but with incremental

Cooperative Balancing

KIP-415 – Apache Kafka 2.3

KAFKA-5505 – Adopted in KafkaConnect

Kafka Connect and Stream already have it by default. (Turned on by default using StreamsPartitionAssignor)

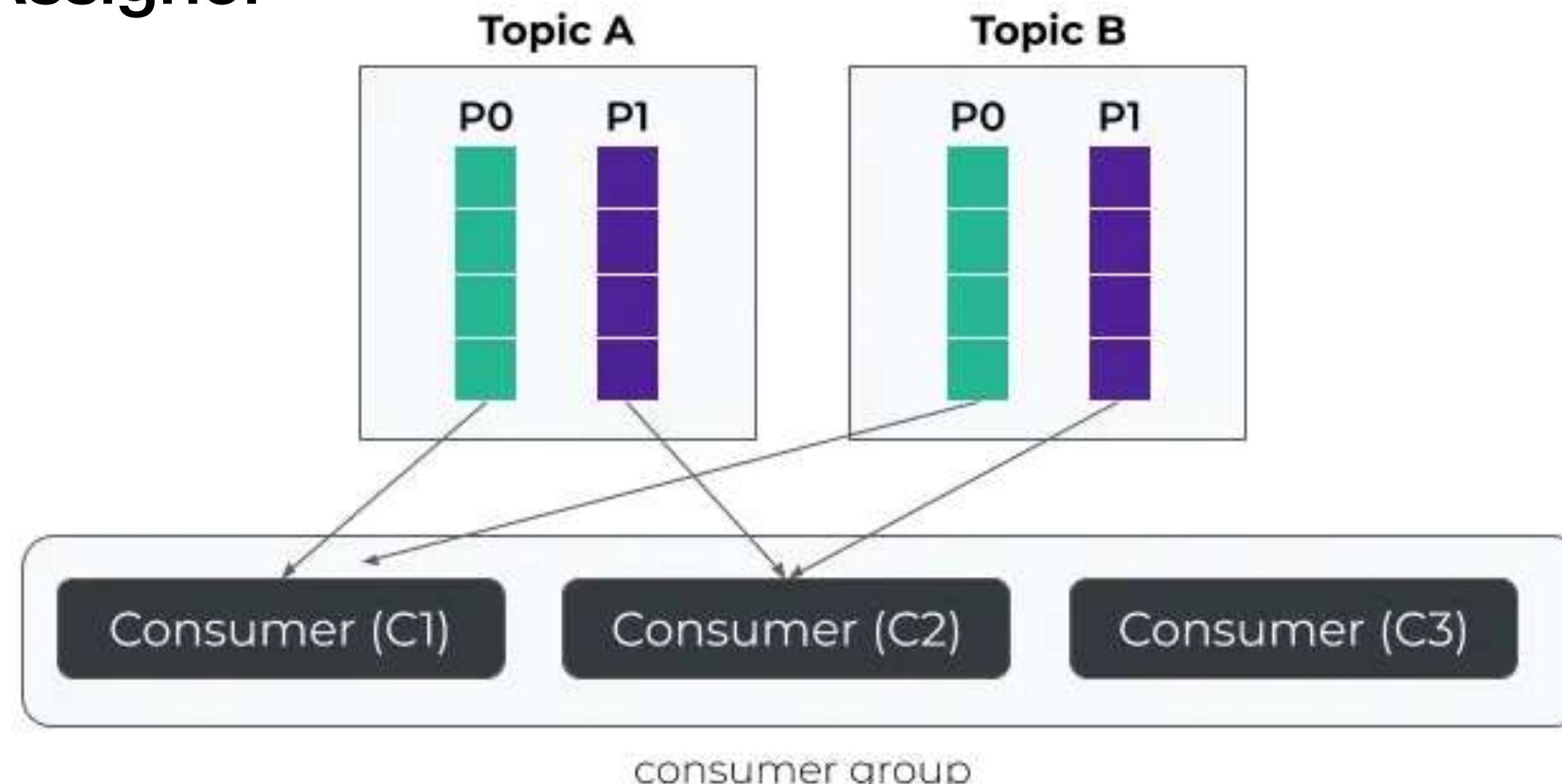
See also: connect.protocol

KAFKA PARTITIONING STRATEGIES

- RangeAssignor
- First put all consumers in lexicographic order using the *member_id* assigned by the broker coordinator.
- Then, it will put available topic-partitions in numeric order.
- Finally, for each topic, the partitions are assigned starting from the first consumer .

KAFKA PARTITIONING STRATEGIES

- RangeAssignor



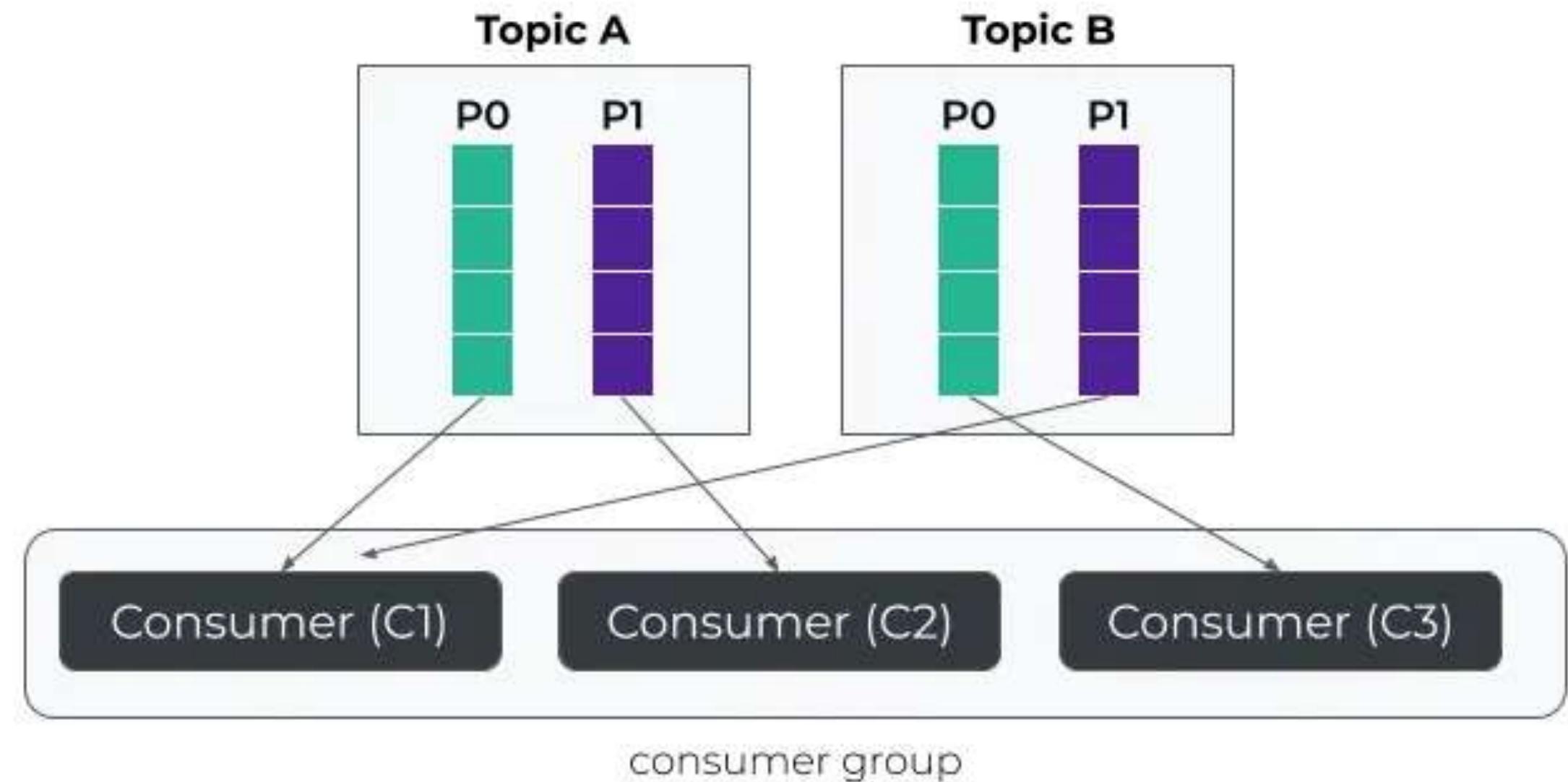
Assignment —> C1 = {A-0, B-0}, C2 = {A-1, B-1}, C3 = {}

KAFKA PARTITIONING STRATEGIES

- **Breaking it down step by step**
- Consumers = C1,C2,C3
- Partitions = A0, A1 (2 partitions)
- Each Consumer gets $\text{Floor}(2/3) = 0$ partitions.
- Reminder partitions = $2 - 0 = 2$
- So C1 get A0, C2 gets A1. C3 gets 0
- Now Partitions = B0,B1 (2 Partitions)
- Each Consumer gets $\text{Floor}(2/3) = 0$ partitions
- Reminder partitions = $2 - 0 = 2$
- So C1 get B0, C2 gets B1, C3 gets 0

KAFKA PARTITIONING STRATEGIES

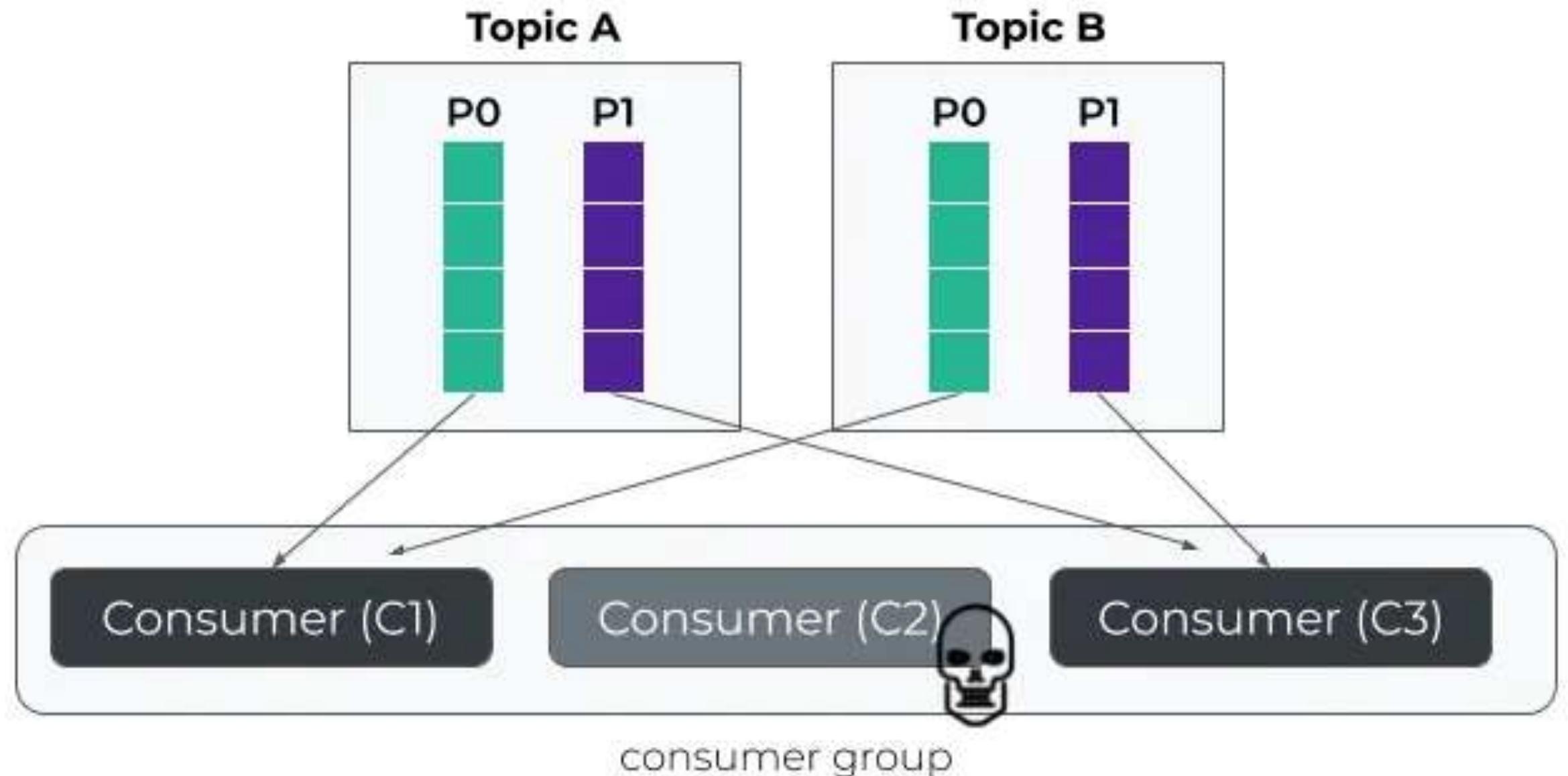
- RoundRobinAssignor
- Distribute evenly across Consumers



Assignment \rightarrow C1 = {A-0, B-1}, C2 = {A-1}, C3 = {B-0}

KAFKA PARTITIONING STRATEGIES

- RoundRobinAssignor does not attempt to reduce partition movements when re-balancing occurs.

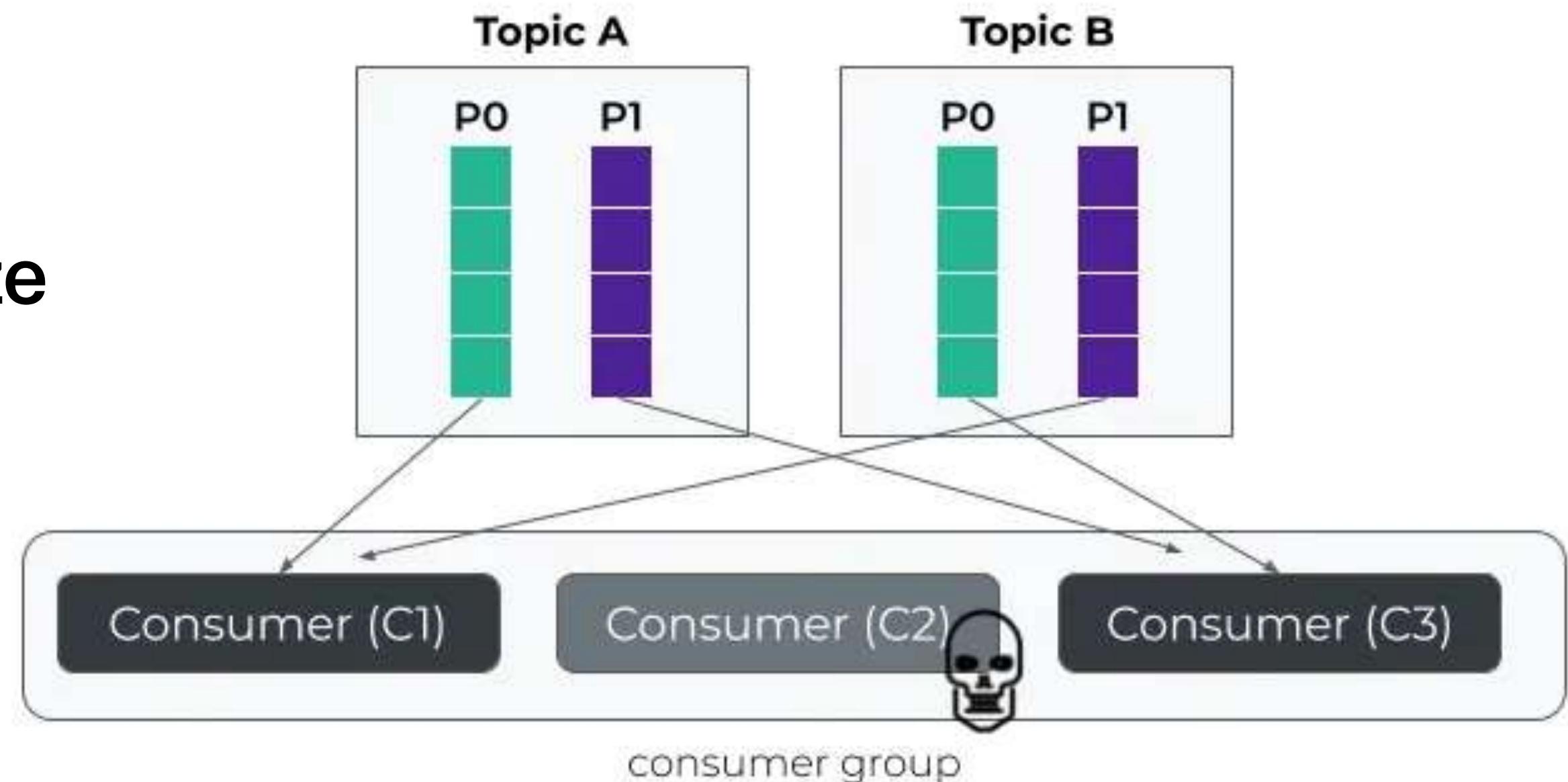


before —> C1 = {A-0, B-1}, C2 = {A-1}, C3 = {B-0}

after —> C1 = {A-0, B-0}, C3 = {A-1, B-1}

KAFKA PARTITIONING STRATEGIES

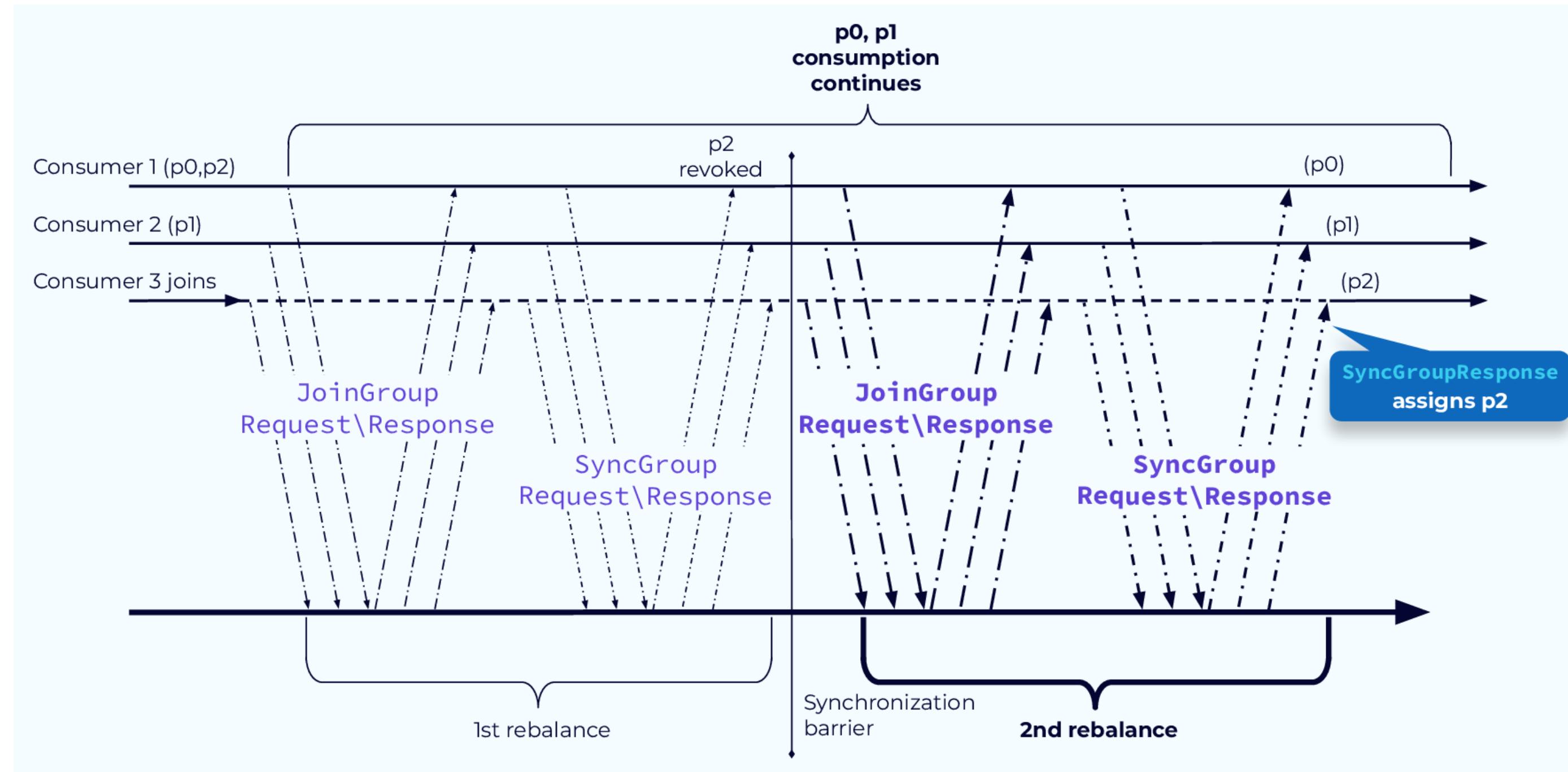
- **StickyAssignor**
Similar to
RoundRobin but
will try to minimize
partition
movement.



KAFKA PARTITIONING STRATEGIES

- The `CooperativeStickyAssignor` assignor works in a two-step process.
- In the first step the determination is made as to which partition assignments need to be revoked.
- Those assignments are revoked at the end of the first rebalance step.
- The partitions that are not revoked can continue to be processed.

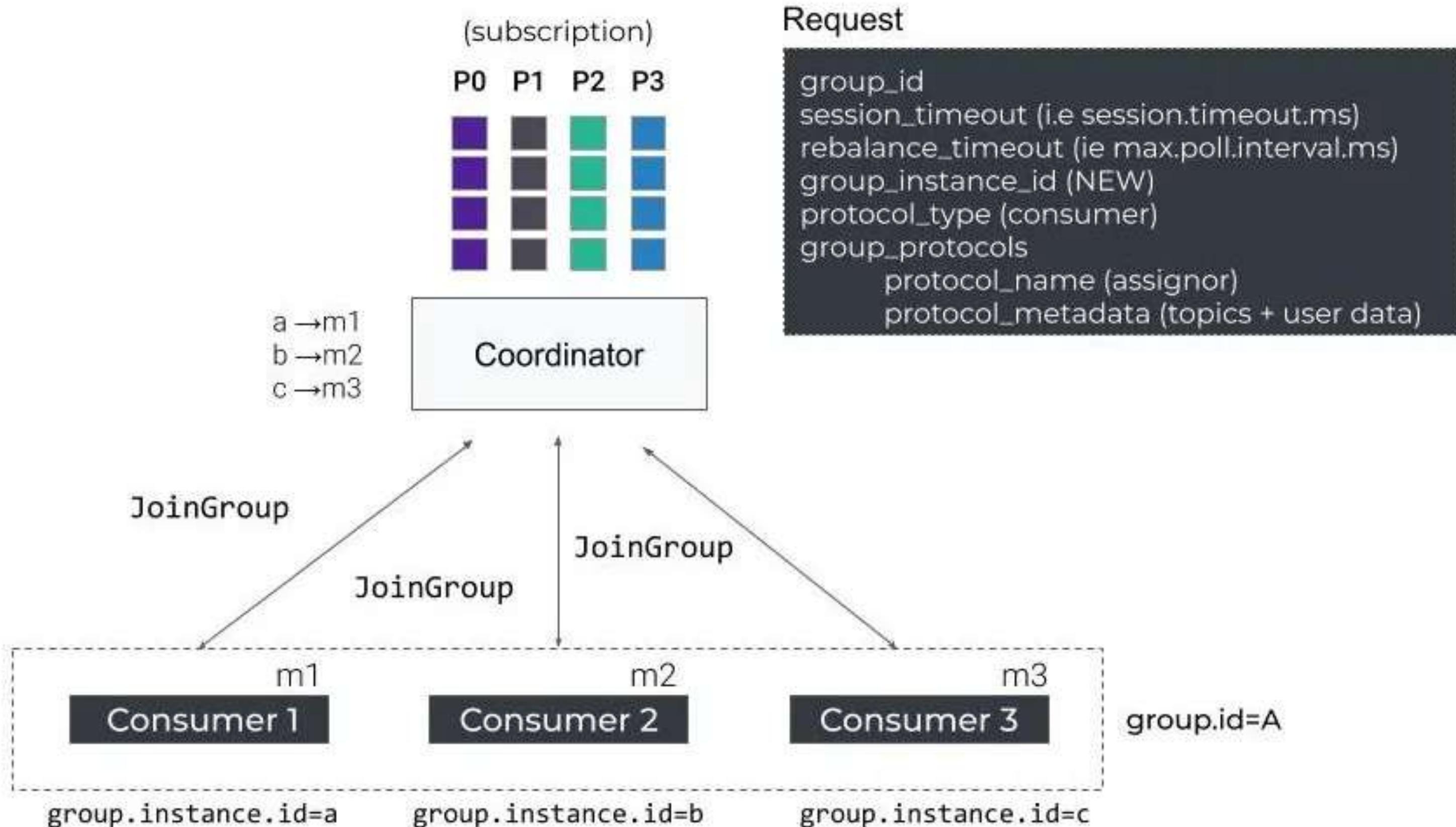
KAFKA PARTITIONING STRATEGIES



KAFKA PARTITIONING STRATEGIES

- Consider Static Membership (since Apache Kafka 2.3)
- Unique identifier along with group.instance.id

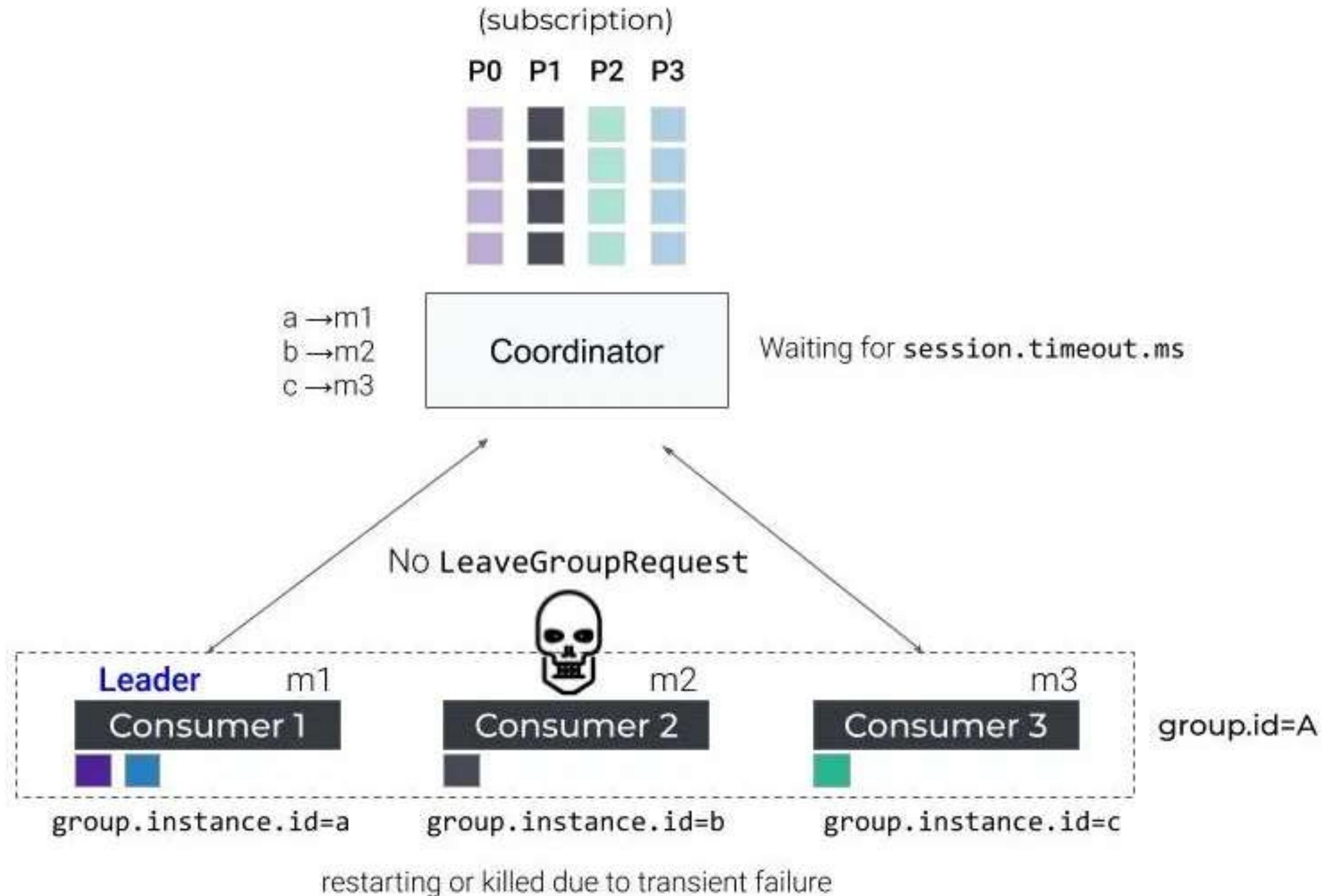
KAFKA PARTITIONING STRATEGIES



KAFKA PARTITIONING STRATEGIES

- When a consumer is killed or restarted, others will not be notified for a re-balance until `session.timeout.ms` is reached.
- Notes that Consumers will not send `LeaveGroup` when they are stopped.

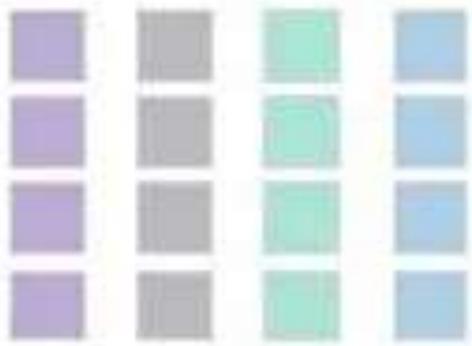
KAFKA PARTITIONING STRATEGIES



KAFKA PARTITIONING STRATEGIES

(subscription)

P0 P1 P2 P3



a → m1
b → m4
c → m3

Coordinator

no rebalance

Leader

m1

Consumer 1



group.instance.id=a

m4

Consumer 2



group.instance.id=b

m3

Consumer 3



group.instance.id=c

group.id=A

Is restarting

No additional re-balance

Adjust session.timeout.ms

KAFKA PARTITIONING STRATEGIES

- All Consumers act as if in Active/Active Mode.
- For some production scenarios, it might be useful to have an Active/Passive mode.
- Similar to a FailoverAssignor. (A Custom Implementation)
- **KAFKA Does not natively support it.**
- One way is to have multi-cluster Replication
 - AWS MKS Replicator
 - Mirror Maker 2
 - Confluent Replicator

KAFKA PARTITIONING STRATEGIES

- You could implement `PartitionAssignor` interface (or) extend the abstract class `AbstractPartitionAssignor`.
- `assign()` method is already implemented here 😊
- Left as an exercise to the class to add active/pассив or priority based assignors. 😊

KAFKA PARTITIONING STRATEGIES

- For consumers, setting the strategy is similar to

```
props.put(ConsumerConfig.PARTITION_ASSIGNMENT_STRATEGY_CONFIG,  
"org.apache.kafka.clients.consumer.CooperativeStickyAssignor");
```

- For producers, (yes there is something similar)

```
props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG,"com.example  
.CustomPartitioner");
```

KAFKA PARTITIONING STRATEGIES

- Producer strategies can be

Strategy Name	Class Name
Default Partitioner	Uses the hash of the key to assign a partition. If the key is null, it round-robin.
Round-Robin Partitioner	Ignores the key and sends messages evenly across partitions.
Uniform Sticky Partitioner	Sends messages to one partition until batch is full or linger.ms expires—reduces latency.

KAFKA PARTITIONING STRATEGIES

- Producer strategies can be

Strategy Name	Class Name
Default Partitioner	org.apache.kafka.clients.producer.internals.DefaultPartitioner
Round-Robin Partitioner	org.apache.kafka.clients.producer.RoundRobinPartitioner
Uniform Sticky Partitioner	org.apache.kafka.clients.producer.UniformStickyPartitioner
Custom Partitioner	<i>(Your own class goes in here)</i>

Notes on Commit styles

Offset Commits

- **Concerns**
 - Which messages should be read?
 - Duplicate Message Reading and Processing
 - Message Loss
- **Kafka keeps track of the messages that consumers read.**
- **This is done by offsets**
- Offsets are integers starting from zero that increment by one as the message gets stored.

Offset Commits

- 4 ways to commit offsets
- **Method #1:** Auto-Commit (Default after every 5 seconds)
- Commits the largest offset returned by poll()

```
KafkaConsumer<Long, String> consumer = new  
KafkaConsumer<>(props);consumer.subscribe(KafkaConfigProperties.getTopic());  
ConsumerRecords<Long, String> messages =  
consumer.poll(Duration.ofSeconds(10));  
for (ConsumerRecord<Long, String> message : messages) { // processed message}
```

Offset Commits

- Assume 100 records came in.
- 60 have been processed.
- Consumer crashes/stop/fails
- It comes back live – records 61 to 100 are lost

Offset Commits

- **Method #2: Manual Sync Commit**
- First off, turn auto Commit to be false

```
Properties props = new Properties();props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "false");
```

```
KafkaConsumer<Long, String> consumer = new KafkaConsumer<>(props);consumer.subscribe(KafkaConfigProperties.getTopic());ConsumerRecords<Long, String> messages = consumer.poll(Duration.ofSeconds(10));//process the messagesconsumer.commitSync();
```

Offset Commits

- Commits only after processing the messages.
- Consumer can still crash before commit is called => Duplicate Processing
- Impacts consumer performance badly.
- Blocking code and will retry => Low throughput.

Offset Commits

- **Method #3: Manual Async Commit**

```
KafkaConsumer<Long, String> consumer = new  
KafkaConsumer<>(props);consumer.subscribe(KafkaConfigProperties.get  
Topic());  
ConsumerRecords<Long, String> messages =  
consumer.poll(Duration.ofSeconds(10));  
//process the messages  
consumer.commitAsync();  
// We do assume that Auto Commit has been disabled
```

Offset Commits

- Suppose 300 is the largest offset, but *commitAsync()* fails due to some issue.
- It could be possible that before it retries, another call of *commitAsync()* commits the largest offset of 400 as it is asynchronous.
- When failed *commitAsync()* retries and if it commits offsets 300 successfully, it will overwrite the previous commit of 400, resulting in duplicate reading.
- That is why *commitAsync()* doesn't retry

Offset Commits

- **Method #4: Commit Specific Offset**
- Processing the messages in small batches
- Commit the offsets as soon as messages are processed.

Offset Commits

```
KafkaConsumer<Long, String> consumer = new KafkaConsumer<>(props);
consumer.subscribe(KafkaConfigProperties.getTopic());

Map<TopicPartition, OffsetAndMetadata> currentOffsets = new HashMap<>();

int messageProcessed = 0;

// So now we have a Hashmap
```

Offset Commits

```
while (true)
{
    ConsumerRecords<Long, String> messages =
    consumer.poll(Duration.ofSeconds(10));
    // our logic will go here...
}
```

Offset Commits

Our logic -→

```
for (ConsumerRecord<Long, String> message : messages)
{
    // processed one message
    messageProcessed++;
}
```

```
currentOffsets.put(
    new TopicPartition(message.topic(), message.partition()),
    new OffsetAndMetadata(message.offset() + 1));
```

```
if (messageProcessed%50==0)
{
    consumer.commitSync(currentOffsets);
}
```

Consumer Tuning

- **Fetch size**
- The fetch size directly impacts the number of messages a consumer fetches from the broker in a single request. Configuring fetch size based on the expected message size optimizes throughput.
fetch.min.bytes
- This configuration defines the minimum amount of data, in bytes, that the broker should return for a fetch request.
- Increasing this value leads to fewer fetch requests, reducing the overhead of network communication and I/O operations.
- However, it may also increase latency as the consumer waits for enough messages to accumulate before making a fetch request.
- You should experiment with different fetch sizes to find the ideal balance between throughput and latency for your specific use case.

Consumer Tuning

- **Max poll records**
- Controlling the maximum number of records fetched in a single poll request helps balance the processing time and consumer lag.
max.poll.records
- This configuration defines the maximum number of records a consumer fetches in a single poll.
- By adjusting this value, you control the trade-off between the time spent processing records in the application and the potential for consumer lag.
- A smaller value leads to more frequent polls and lower consumer lag but also increases the overhead of processing records.
- Conversely, a larger value improves throughput but results in higher consumer lag if the consumer cannot process records fast enough.
- You should set this value based on your application's processing capabilities/resources and tolerance for consumer lag.

Consumer Tuning

- **Client-side buffering**
- You use buffering to reduce the impact of network latency on consumer processing.

fetch.max.bytes

- This configuration defines the maximum amount of data, in bytes, that the consumer buffers from the broker before processing it.
- Increasing this value helps the consumer absorb temporary spikes in network latency, but it may also expand the memory footprint of the consumer.
- You should choose a buffer size that balances the trade-off between network latency and memory usage for your specific use case.

Consumer Tuning

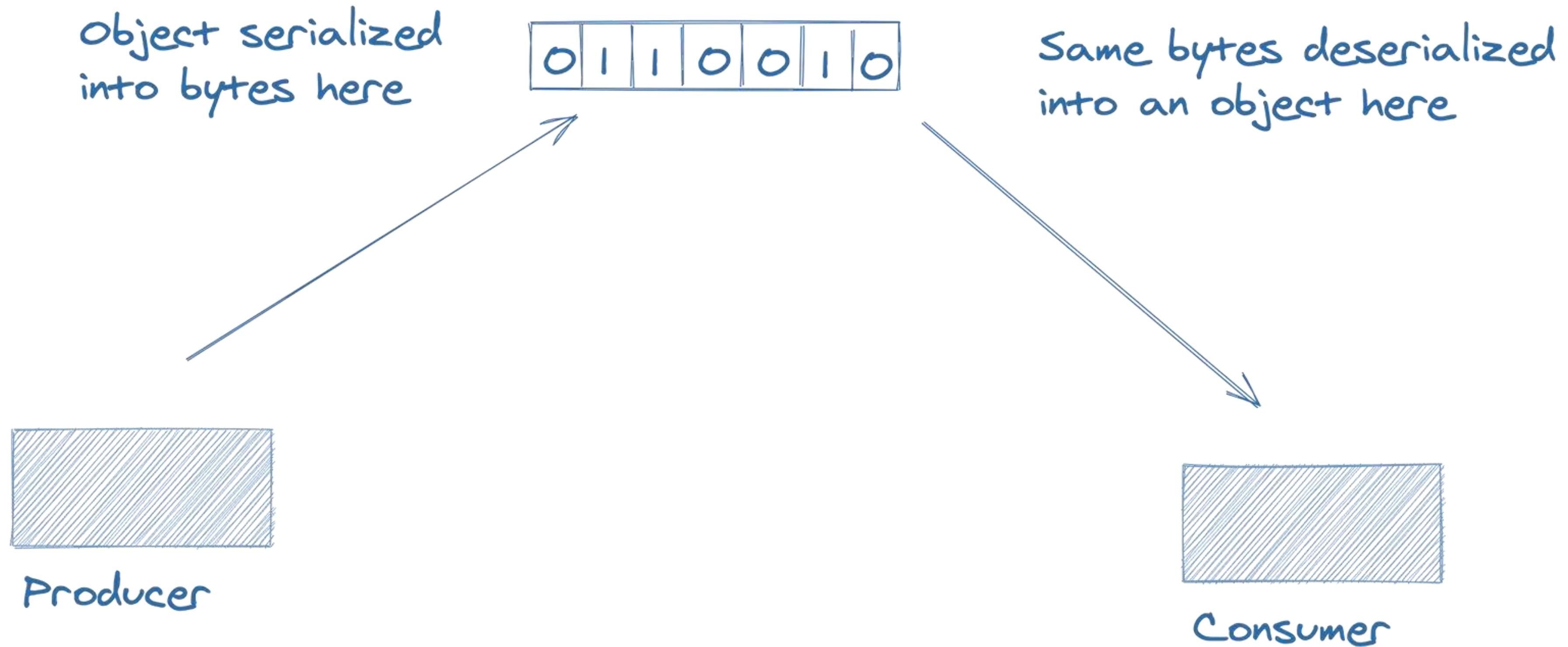
- **Consumer group rebalancing**
 - Configuring session and heartbeat timeouts optimizes group rebalancing, ensuring consumer groups maintain a stable membership and quickly detect failed consumers.
- session.timeout.ms**
- This configuration defines the maximum amount of time, in milliseconds, that a consumer can be idle without sending a heartbeat to the group coordinator.
 - If no heartbeat is received within this time, the consumer is considered failed, and the group triggers a rebalance.
 - Rebalancing in Kafka can impact performance, as it may cause temporary disruption in message processing while consumer groups reassign partitions to ensure even distribution of workload across consumers.
 - During this process, the overall throughput and latency may be affected, making it essential to monitor and manage rebalancing events carefully. You should set this value based on your application's processing capabilities and the desired frequency of rebalances.

heartbeat.interval.ms

- This configuration defines the interval, in milliseconds, at which the consumer sends heartbeats to the group coordinator.
- A shorter interval helps the group coordinator detect failed consumers more quickly but also increases the coordinator's load. You should choose a heartbeat interval that balances the trade-off between failure detection and coordinator load for your specific use case.

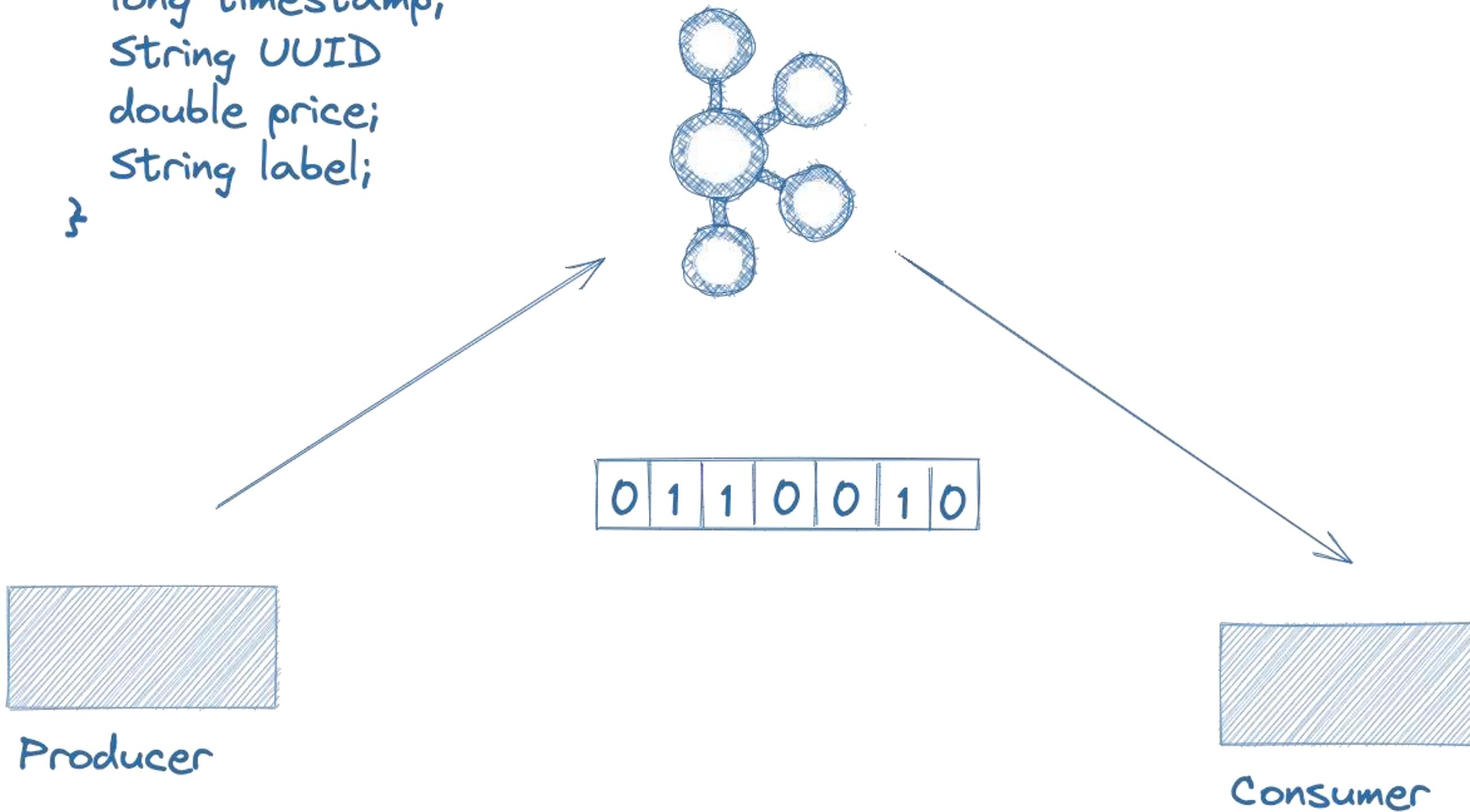
Schemas...

An implicit contract



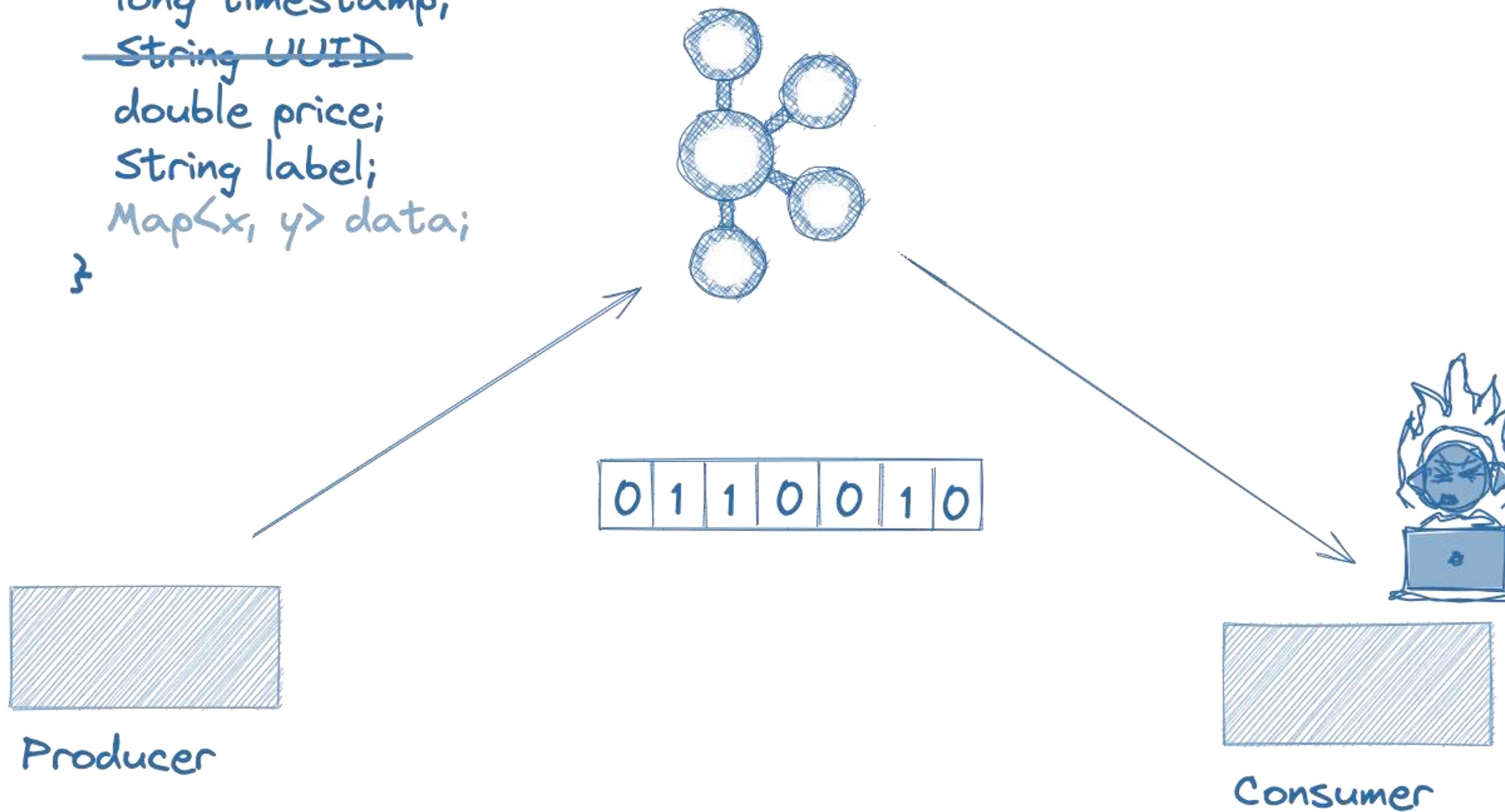
Expectation of object structure

```
public class Foo {  
    long timestamp;  
    String UUID;  
    double price;  
    String label;  
}
```

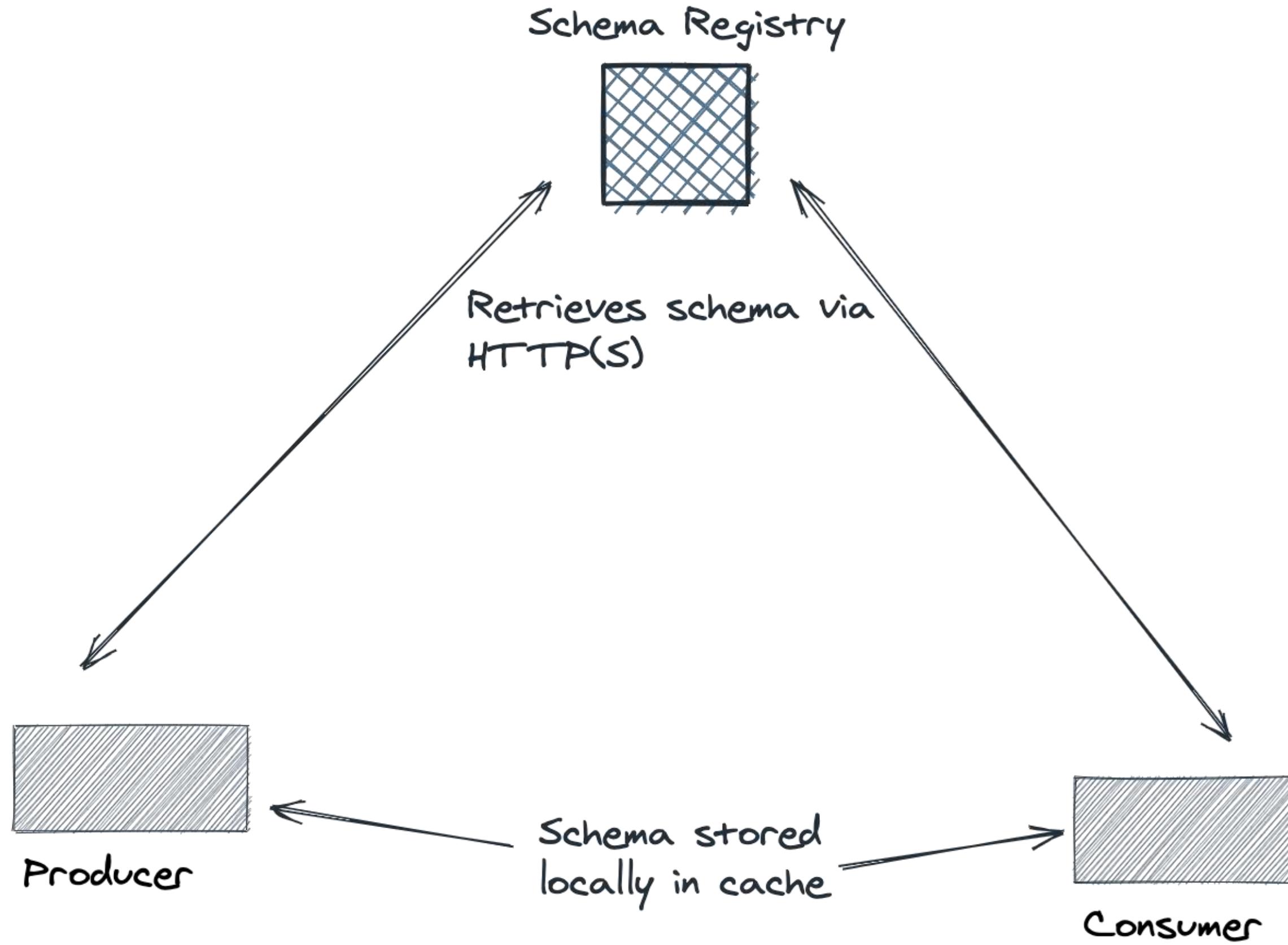


Potentially make unexpected changes

```
public class Foo {  
    long timestamp;  
String UUID  
    double price;  
    String label;  
    Map<x, y> data;  
}
```



Use Schema Registry.



Working with Schema Registry

1. Write/download a schema
2. Test and upload the schema
3. Generate the objects
4. Configure clients (Producer, Consumer, Kafka Streams)
 - Avro
 - Protocol Buffers
 - JSON Schema
5. Write your application!

Build a Schema - AVRO

```
{  
  "type": "record",  
  "namespace": "io.confluent.developer.avro",  
  "name": "Purchase",  
  "fields": [  
    {"name": "item", "type": "string"},  
    {"name": "amount", "type": "double", "default": 0.0},  
    {"name": "customer_id", "type": "string"}  
  ]  
}
```

Build a Schema (Protobuf)

```
syntax = "proto3";  
  
package io.confluent.developer.proto;  
option java_outer_classname = "PurchaseProto";  
  
message Purchase {  
    string item = 1;  
    double amount = 2;  
    string customer_id = 3;  
}
```

How it works

1 Create and register a schema. Any number of tools (largely CLI Based ☺)

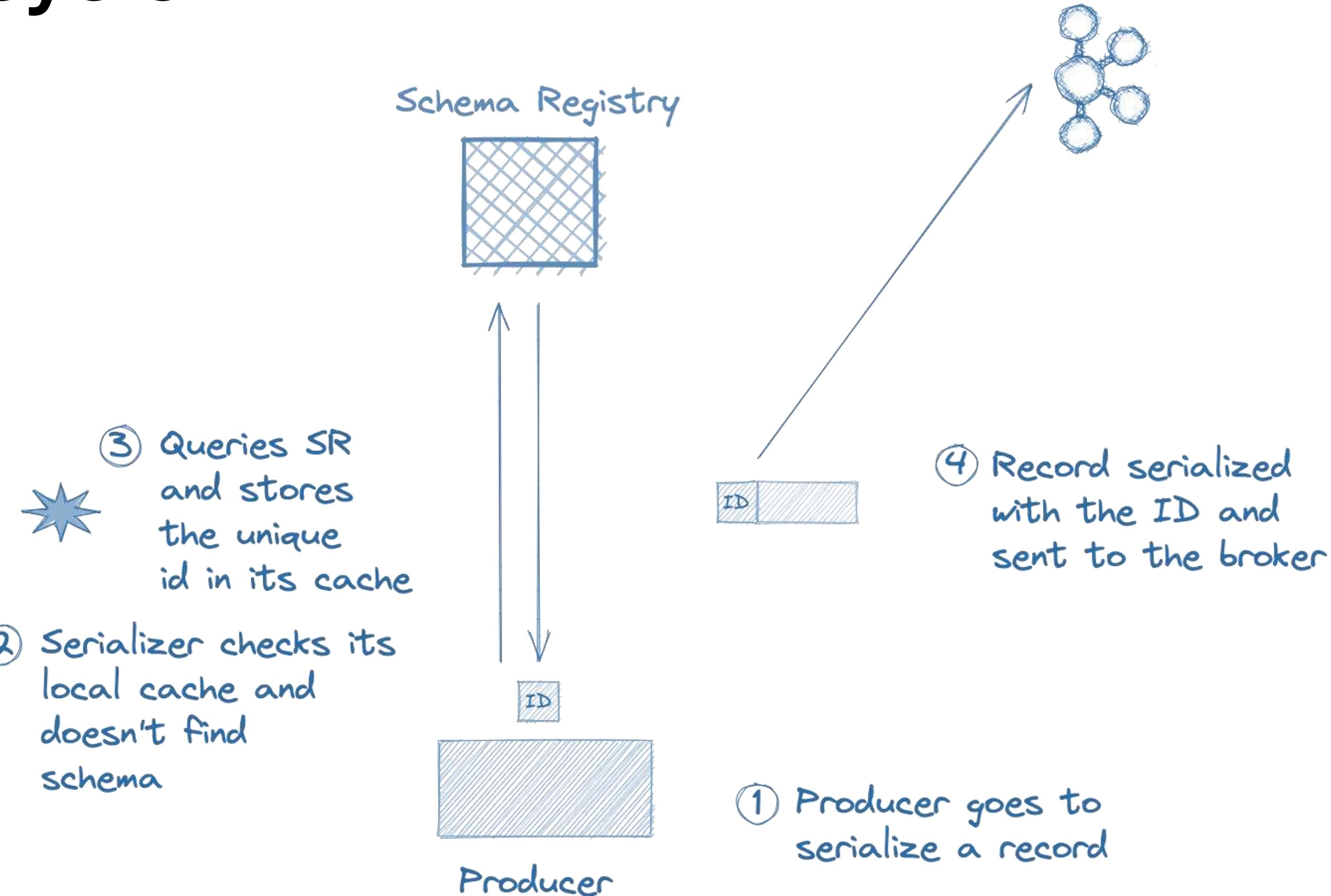
Producers can auto register – JUST DON'T !!

e.g. `producerConfigs.put("auto.register.schemas", true)`

2. View and Retrieve the schema

BTW, in Kafka, set `schema.registry.url`. There are more parameters ☺

Lifecycle



Clients – Producer/Consumers

```
producerConfigs.put("value.serializer",?);
```

- . **KafkaAvroSerializer.class**
- . **KafkaProtobufSerializer.class**
- . **KafkaJsonSchemaSerializer.class**

```
consumerConfigs.put("value.deserializer",?);
```

- . **KafkaAvroDeserializer.class**
- . **KafkaProtobufDeserializer.class**
- . **KafkaJsonSchemaDeserializer.class**

Clients – Consumer

- . **specific.avro.reader = true|false**
 - . *true - Kafka uses SpecificDatumReader to deserialize Avro messages into your generated POJOs (classes generated from .avsc schemas).*
 - . *false – (default) Kafka uses GenericDatumReader, and you get a GenericRecord—a flexible but less type-safe structure*
- . **specific.protobuf.value.type = fully qualified proto class name**
 - . *- Tells the Kafka Protobuf deserializer which specific generated Protobuf class to use when deserializing the message.*
- . **json.value.type = class name**
 - . *- Purpose: Specifies the Java class to deserialize JSON messages into.*
 - . *- Type: Fully qualified class name of the target POJO*

Clients – Consumer Returned Types

Avro

- . SpecificRecord – `myObj.getFoo()`
- . *Access fields via generated Getters*
- . GenericRecord - `generic.get("foo")`
Access fields dynamically

Protobuf

- . Specific
 - . A **specific Protobuf return type** refers to a **generated Java class** that corresponds to a Protobuf message definition—used when deserializing Kafka messages into strongly typed objects.
- . Dynamic

Clients – Kafka Streams

- `SpecificAvroSerde`
- `GenericAvroSerde`
- `KafkaProtobufSerde`
- `KafkaJsonSchemaSerde`

Clients - Testing

For testing you can use a mock schema registry

- . **schema.registry.url=“mock://scope-name”**

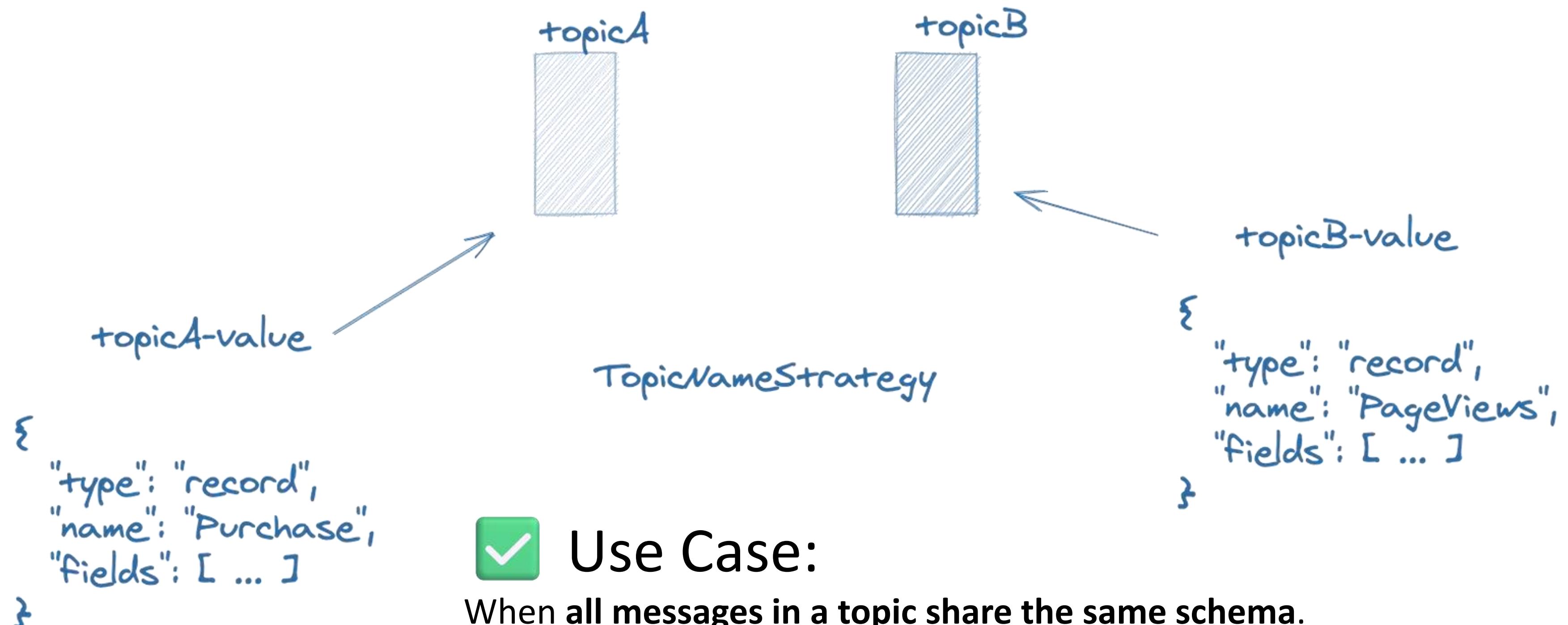
What is a Subject ?

- Defines a namespace for a schema
- Compatibility checks are per subject
- Different approaches – subject name strategies

Subject Name Strategies

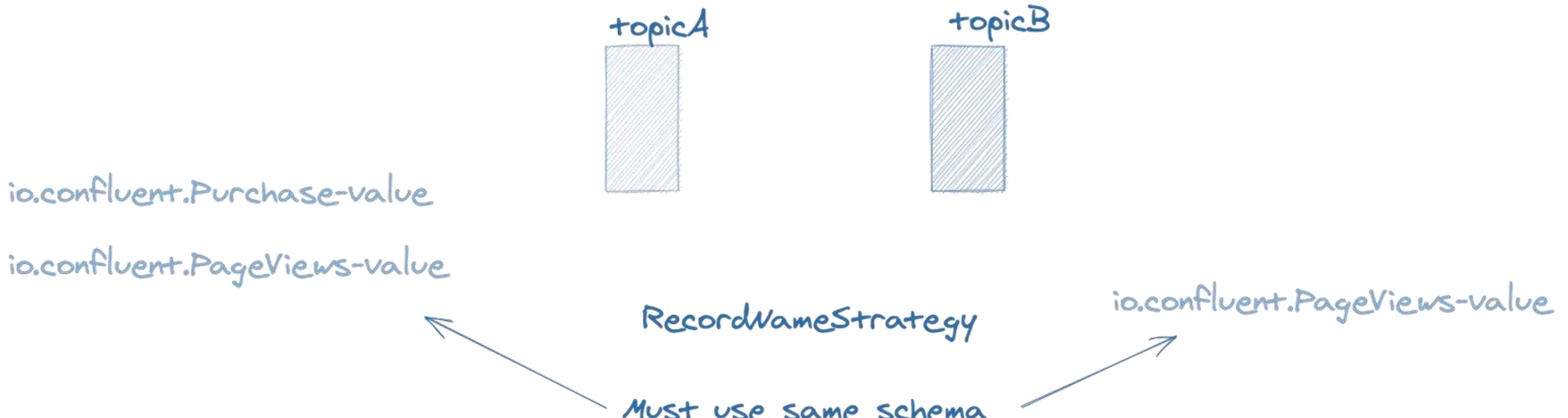
- In Kafka, when using Avro (or Protobuf/JSON Schema) with Schema Registry, every schema is registered under a **subject name**. This subject name determines:
 - How schemas are grouped
 - How compatibility checks are performed
 - How consumers/producers retrieve schemas
- Kafka supports three main strategies for naming these subjects:

Subjects - TopicNameStrategy (Default)



Simple and efficient for homogeneous data.

Subjects - RecordNameStrategy

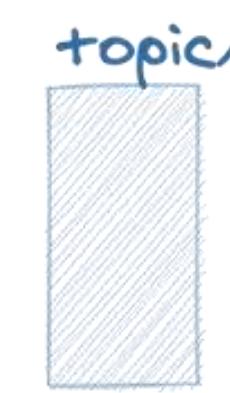


Use Case:

When **same record type is used across multiple topics.**
Schema is shared globally across topics.

Subjects - TopicRecordNameStrategy

topicA-io.confluent.Purchase-value



topicA-io.confluent.PageViews-value

version 1

TopicRecordNameStrategy

version 2

Schemas scoped to the topic level, so now they can evolve individually



Use Case:

- When multiple record types are published to the same topic.
- Allows schema evolution per topic-record pair.

Schema Compatibility

- . Schema Registry provides a mechanism for safe changes
- . Evolving a schema
- . Compatibility checks are per subject
- . When a schema evolves, the subject remains, but the schema gets a new ID and version

Schema Compatibility

```
curl -X PUT \  
-H "Content-Type: application/vnd.schemaregistry.v1+json" \  
--data '{"compatibility": "BACKWARD"}' \  
http://localhost:8081/config/my-topic-value
```

Schema compatibility Example

```
{  
  "type": "record",  
  "name": "Purchase",  
  "fields": [  
    { "name": "item", "type": "string" },  
    { "name": "amount", "type": "double" }  
  ]  
}
```

Schema compatibility Example

```
{ "name": "customer_id", "type": "string", "default": "" }
```

- This is backward compatible since a new field is being added with a default.
- If you remove a field that does not have default, it will fail.

Schema Compatibility

Compatibility Type	Changes allowed	Check against which schemas	Upgrade first
BACKWARD	<ul style="list-style-type: none"> •Delete fields •Add optional fields 	Last version	Consumers
BACKWARD_TRANSITIVE	<ul style="list-style-type: none"> •Delete fields •Add optional fields 	All previous versions	Consumers
FORWARD	<ul style="list-style-type: none"> •Add fields •Delete optional fields 	Last version	Producers
FORWARD_TRANSITIVE	<ul style="list-style-type: none"> •Add fields •Delete optional fields 	All previous versions	Producers
FULL	<ul style="list-style-type: none"> •Add optional fields •Delete optional fields 	Last version	Any order
FULL_TRANSITIVE	<ul style="list-style-type: none"> •Add optional fields •Delete optional fields 	All previous versions	Any order
NONE	<ul style="list-style-type: none"> •All changes are accepted 	Compatibility checking disabled	Depends

Schema Compatibility

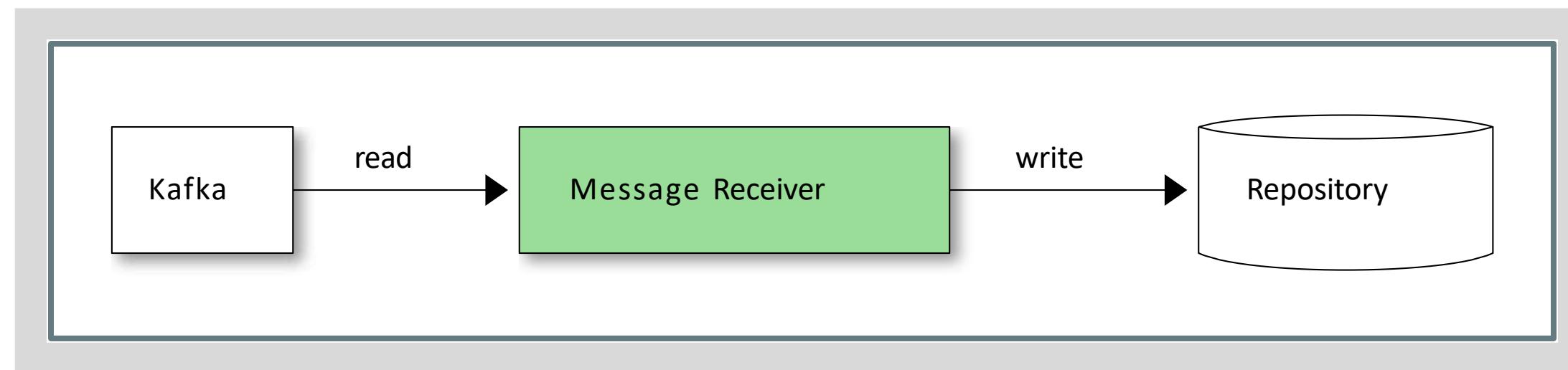
- Resources:
- <https://docs.confluent.io/platform/current/schema-registry/fundamentals/schema-evolution.html>
- <https://github.com/davidwmartines/schema-compatibility>
- <https://www.codestudy.net/blog/kafka-schema-compatibility/>

Testing.....

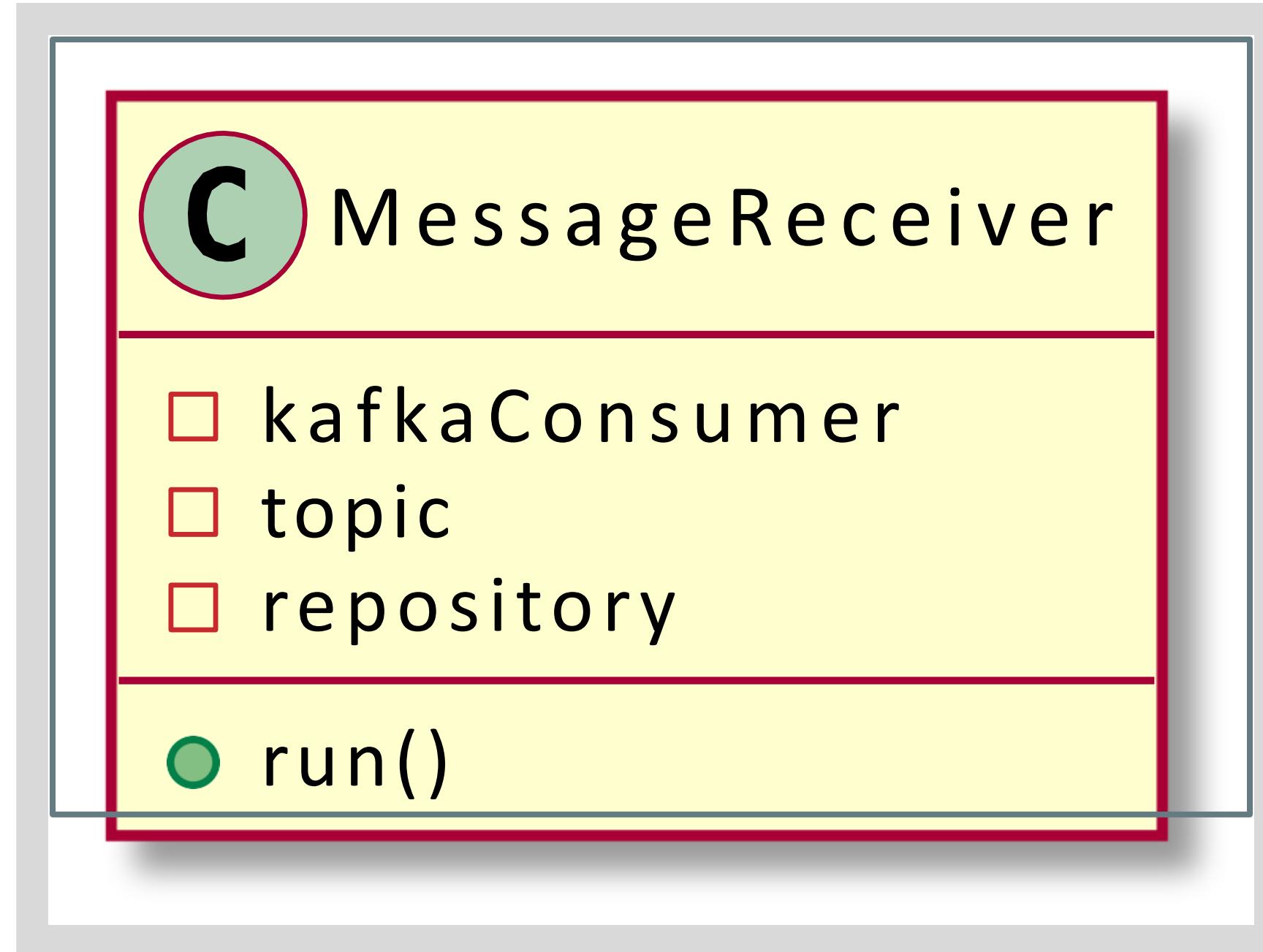
THE TEXTBOOK KAFKA CONSUMER

```
1 var props = new Properties();      1
2 props.put("bootstrap.servers", "kafka-server:9092");
3 props.put("key.deserializer",
4   "org.apache.kafka.common.serialization.StringDeserializer");
5 props.put("value.deserializer",
6   "org.apache.kafka.common.serialization.StringDeserializer");
7
8 var consumer = new KafkaConsumer<String, String>(props);
9 consumer.subscribe(List.of("my-topic"));          2
10
11 while (true) {    3
12   var records = consumer.poll(
13     Duration.ofMillis(Long.MAX_VALUE));           4
14   for (var record : records)
15     System.out.printf("offset = %d, key = %s, value = %s%n",
16       record.offset(), record.key(), record.value());
17 }
```

OVERVIEW OF MESSAGE RECEIVER



CLASS OVERVIEW

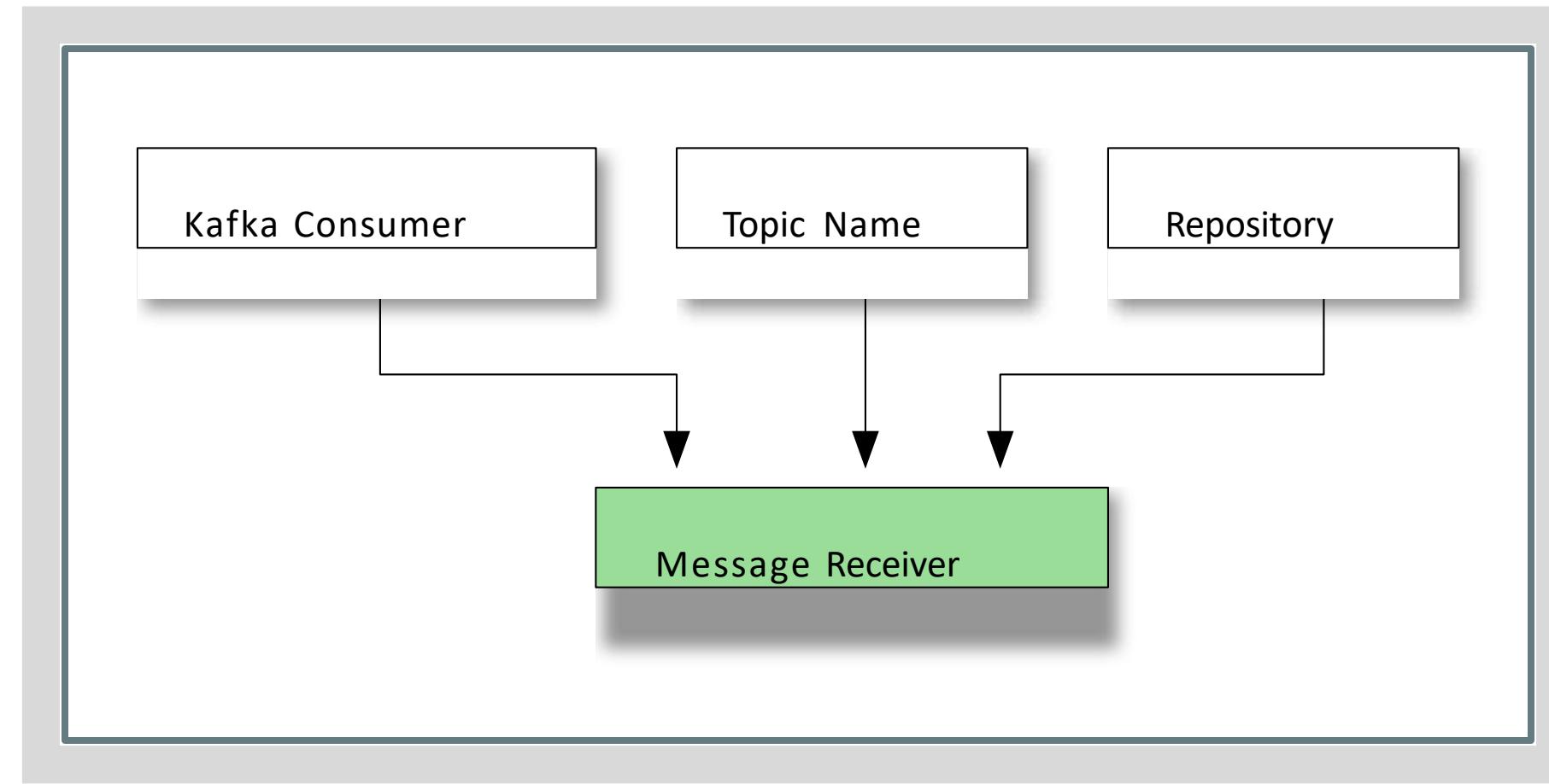


- pass everything needed into the object

USAGE

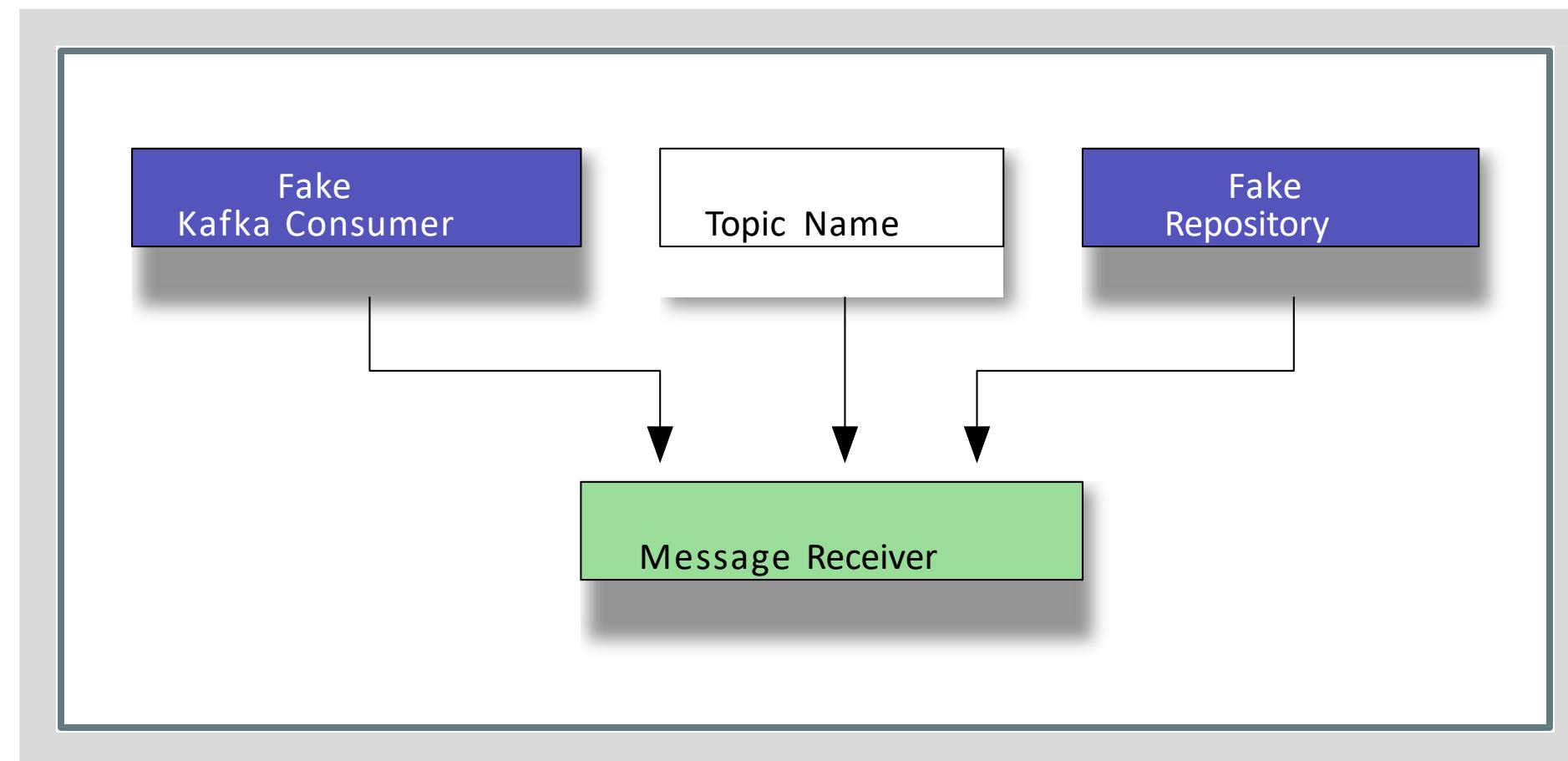
```
public class Main {  
    public static void main(String[] args){  
  
        var consumer = new KafkaConsumer<String, String>(props); 1  
        var topic = "my-topic";  
        var repository = new Repository("jdbc:...");  
  
        var messageReceiver =  
            new MessageReceiver(consumer, topic, repository); 2  
        messageReceiver.run(); 3  
    }  
}
```

CONSTRUCTOR INJECTION



- providing everything the object requires via the constructor
- enables testability

SIMPLE TO REPLACE DEPENDENCIES



FIRST IMPLEMENTATION

```
public MessageReceiver(Consumer<String, String> kafkaConsumer,
                      String topic, Repository repository) {
    this.kafkaConsumer = kafkaConsumer; this.topic = topic;
    this.repository = repository;
}

public void run() {
    kafkaConsumer.subscribe(List.of(topic)); while (true) { // this is an endless loop
        var records = kafkaConsumer.poll(Duration.ofMillis(Long.MAX_VALUE))
        for (var record : records) {
            repository.saveMessage(record.value());
        }
    }
}
```

1

DIVIDE AND CONQUER

```
public MessageReceiver(KafkaConsumer<String, String> kafkaConsumer,
                      String topic, Repository repository) {
    this.kafkaConsumer = kafkaConsumer; this.topic
    = topic;
    this.repository = repository; }

public void run() {
    kafkaConsumer.subscribe(List.of(topic)); while
    (true) {
        processRecords();
    }
}

protected void processRecords() {
    var records = kafkaConsumer.poll(Duration.ofMillis(Long.MAX_VALUE));
    for (var record : records) {
        repository.saveMessage(record.value());
    }
}
```

A POSSIBLE TEST

```
@Mock Repository mockRepository;
@Mock KafkaConsumer<String, String> mockConsumer;

@Test
public void writesReceivedMessage() {
    // arrange
    var records = buildConsumerRecords("my-topic", "HELLO          WORLD")
    when(mockConsumer.poll(any())).thenAnswer(notUsed-> records);

    // act (sut = System Under Test)
    var sut = new MessageReceiver(mockConsumer, mockRepository, "my-topic")
    // we don't call run() directly
    sut.processRecords(); ①

    // assert
    verify(mockRepository, times(1)).saveMessage("HELLO WORLD");
}
```

TEST COVERAGE

```
public MessageReceiver(KafkaConsumer<String, String> kafkaConsumer,
                      String topic, Repository repository) {
    this.kafkaConsumer = kafkaConsumer; this.topic
    = topic;
    this.repository = repository; }

public void run() {
    kafkaConsumer.subscribe(List.of(topic)); while
    (true) {
        processRecords();
    }
}

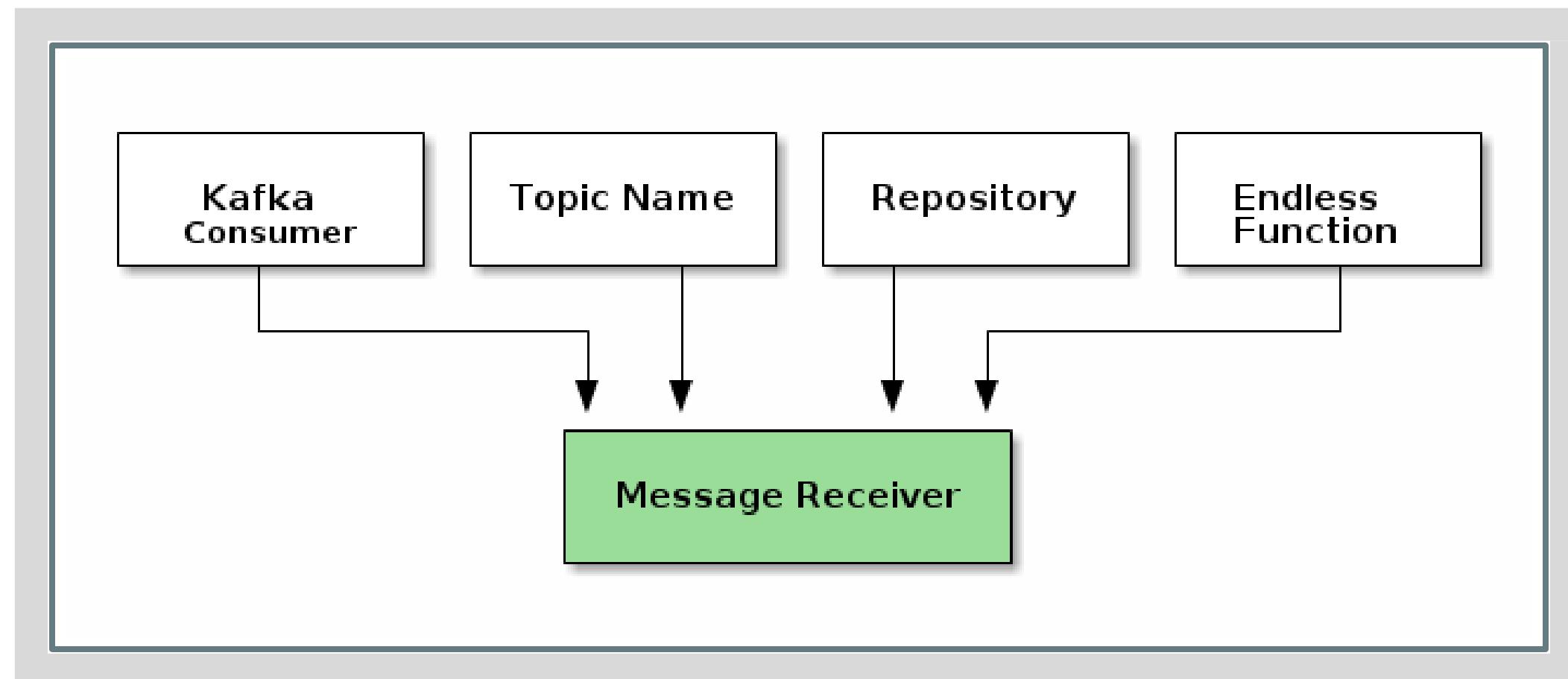
protected void processRecords() {
    var records = kafkaConsumer.poll(Duration.ofMillis(Long.MAX_VALUE));
    for (var record : records) {
        repository.saveMessage(record.value());
    }
}
```

LET'S INJECT "ENDLESS"

```
public MessageReceiver(Consumer<String, String> kafkaConsumer,
                      String topic, Repository repository,
                      BooleanSupplier whileFunc ) {
    this.kafkaConsumer = kafkaConsumer;
    this.topic = topic; this.repository
    = repository; this.whileFunc =
    whileFunc; }

public void run() {
    kafkaConsumer.subscribe(List.of(topic)); while
    (whileFunc.getAsBoolean()) {
        var records = kafkaConsumer.poll(Duration.ofMillis(Long.MAX_VALUE))
        for (var record : records) {
            repository.saveMessage(record.value()); }}}
```

LET'S INJECT "ENDLESS"



USAGE

```
public class Main {  
    public static void main(String[] args){  
  
        var consumer = new KafkaConsumer<String, String>(props);  
        var topic = "my-topic";  
        var repository = new Repository("jdbc:...");  
  
        var messageReceiver = new MessageReceiver(  
            consumer, topic, repository,  
            () -> true );  
        messageReceiver.run();  
    }  
}
```

A POSSIBLE TEST

```
@Mock Repository mockRepository;
@Mock KafkaConsumer<String, String> mockConsumer;

@Test
public void writesReceivedMessage() {
    // arrange
    var records = buildConsumerRecords("my-topic", "HELLO      WORLD")
when(mockConsumer.poll(any())).thenAnswer(notUsed-> records);

    // act
    var sut = new MessageReceiver(
        mockConsumer, mockRepository, "my-topic",
        this::runOnce);
    sut.run();

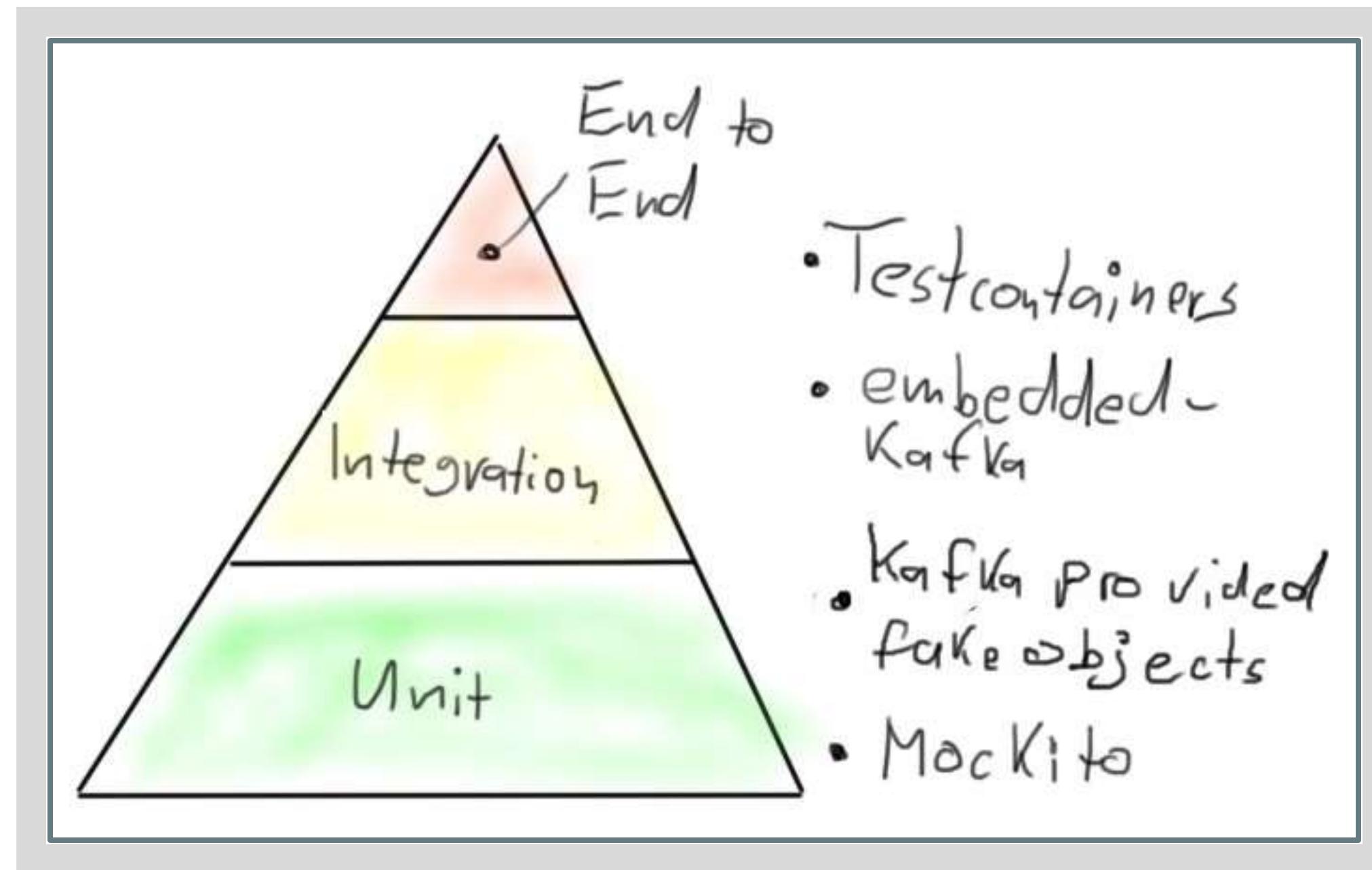
    // assert
    verify(mockRepository, times(1)).saveMessage("HELLO WORLD");
}
```

ONCE IS ENOUGH

```
private boolean firstTimeRun = true;

private boolean runOnce() {
    if (firstTimeRun) {
        firstTimeRun = false;
        return true;
    }
    return false;
}
```

TESTING KAFKA AT DIFFERENT INTEGRATION LEVELS

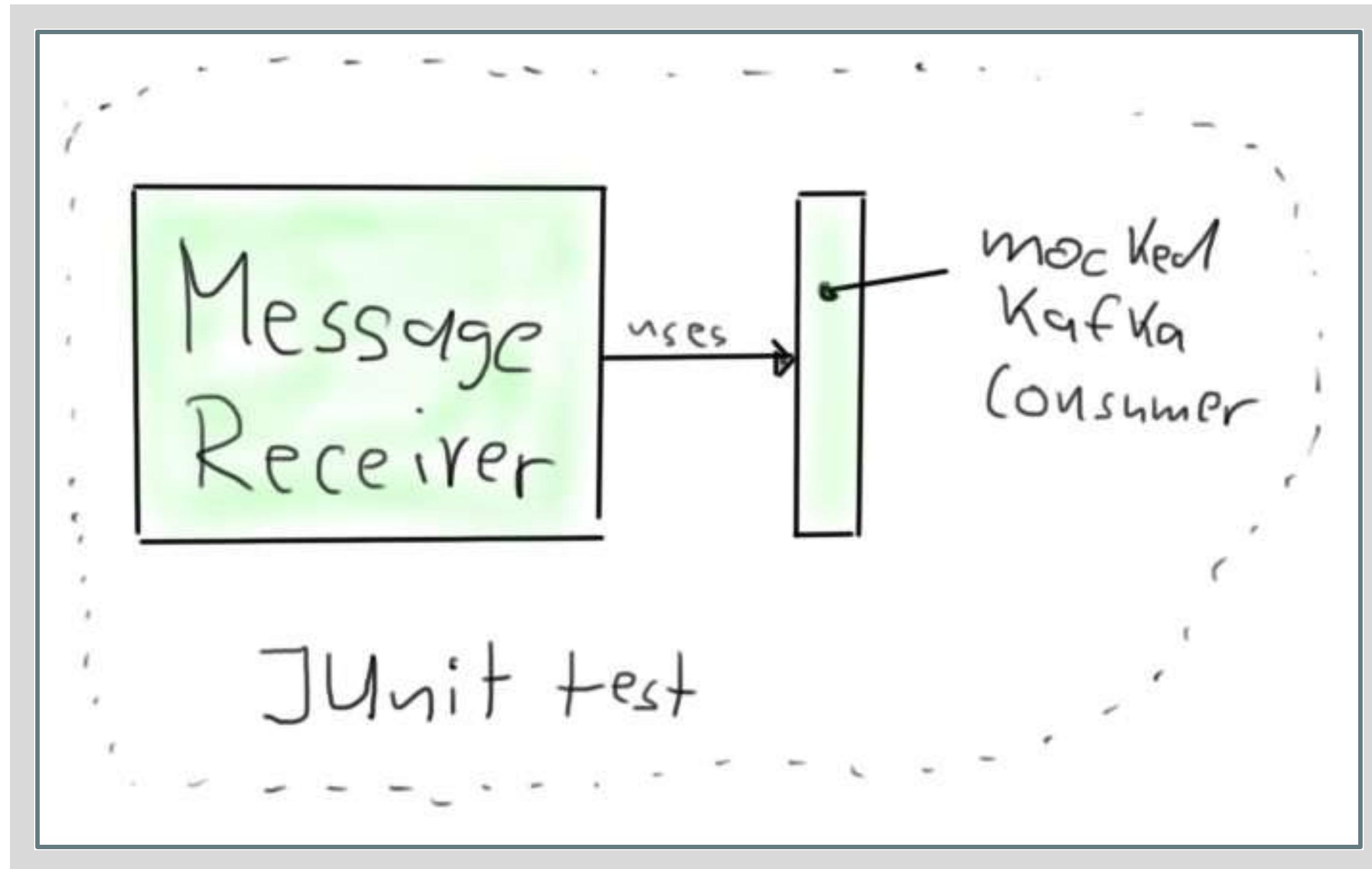


MOCKITO



defacto standard for *mocking* in Java

MOCKITO - CONTEXT



MOCKITO - EXAMPLE TEST

```
@Mock Repository mockRepository;
@Mock KafkaConsumer<String, String> mockConsumer;

@Test
public void writesReceivedMessage() {
    // arrange
    var records = buildConsumerRecords("my-topic", "HELLO WORLD");
    when(mockConsumer.poll(any())).thenAnswer(notUsed-> records);

    // act
    var sut = new MessageReceiver(
        mockConsumer, mockRepository, "my-topic",
        this::runOnce);
    sut.run();

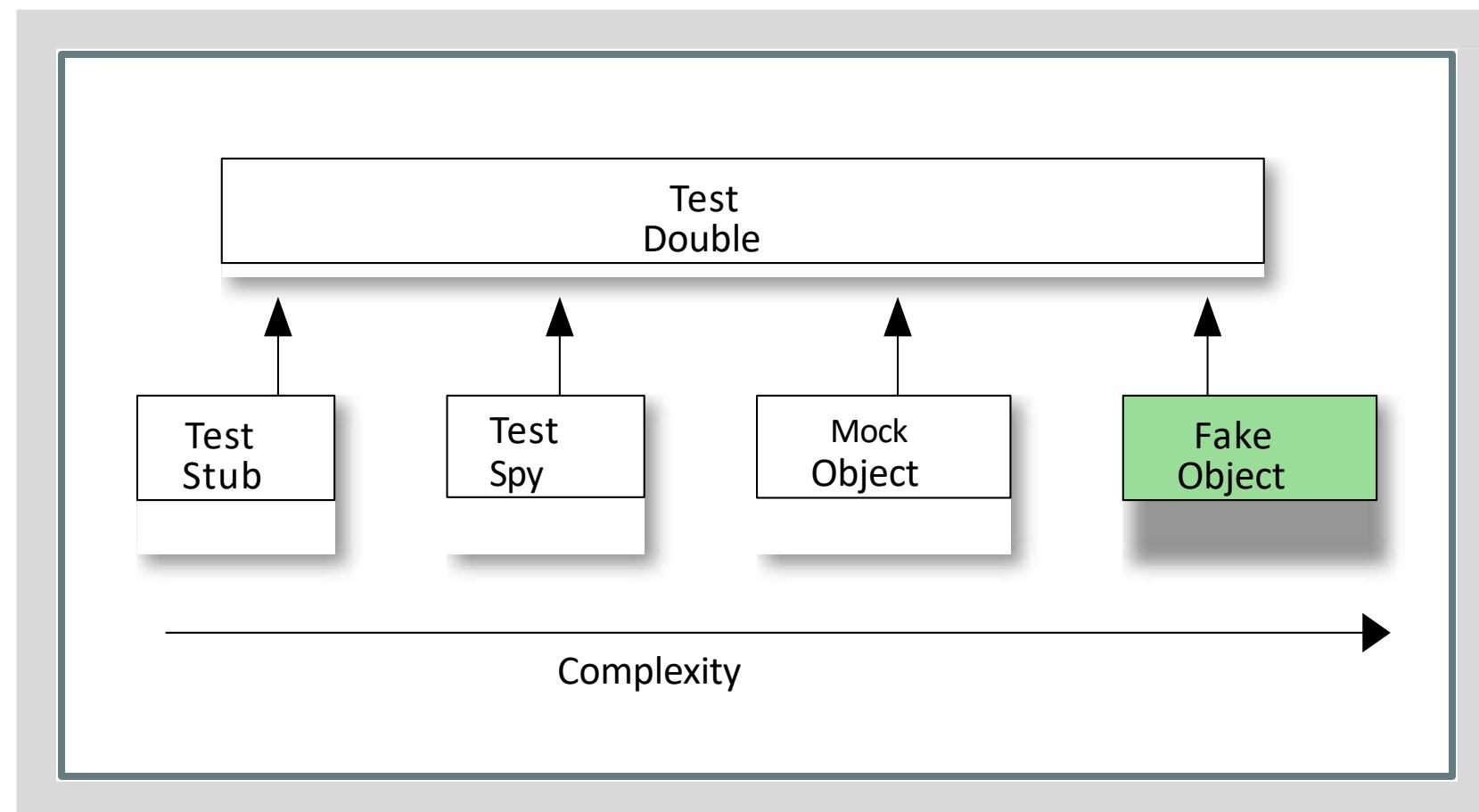
    // assert
    verify(mockRepository, times(1)).saveMessage("HELLO WORLD"); }
```

KAFKA FAKES OBJECTS

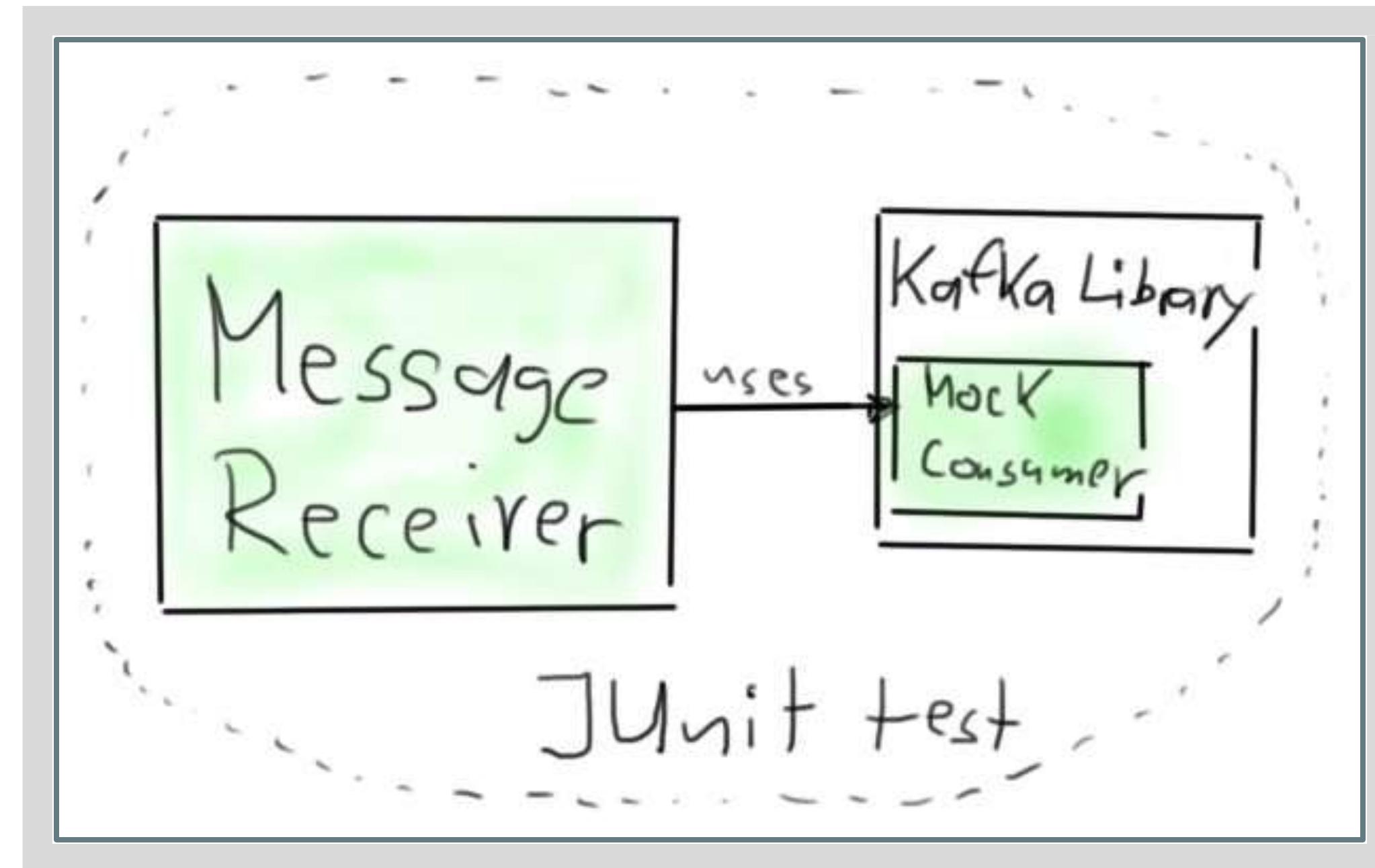


- fake objects
- close to the actual prod implementation Kafka
- internally used for testing

FAKE OBJECT ?



KAFKA FAKE OBJECTS - CONTEXT



USING THE PLAIN MOCKCONSUMER

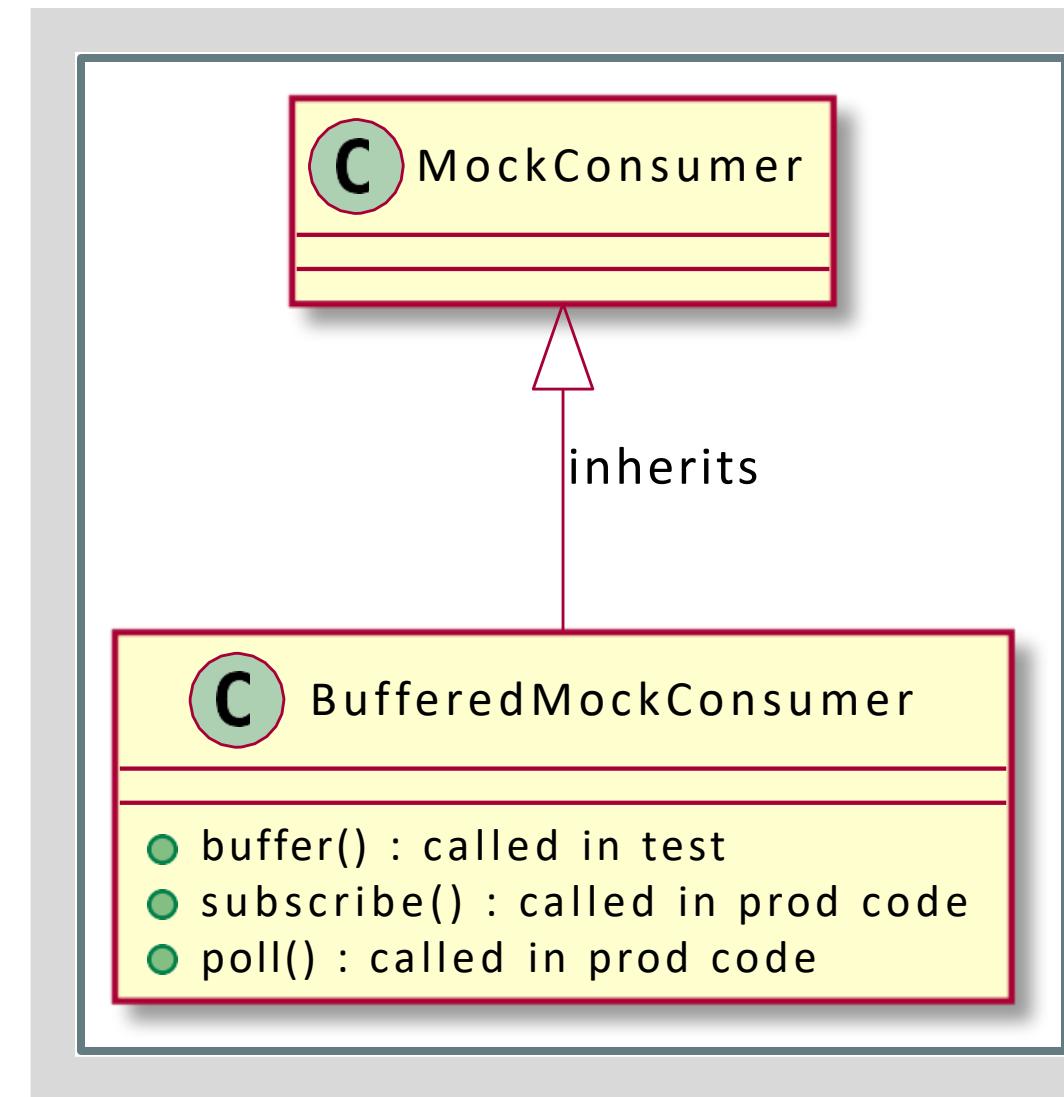
```
import org.apache.kafka.clients.consumer.MockConsumer;
...
@Test
public void plainMockConsumer() {
    // arrange
    // preparing mock consumer
    var mockConsumer =
        new MockConsumer<String, String>(OffsetResetStrategy.EARLIEST);

    // need to subscribe here, we'd like it in prod code 😞
    mockConsumer.subscribe(List.of("my-topic"));

    // voodoo part
    var topicPartition=new TopicPartition("my-topic", 0);
    mockConsumer.rebalance(List.of(topicPartition));
    mockConsumer.updateBeginningOffsets(Map.of(topicPartition,
                                                0L));

    // this what we understand
    mockConsumer.addRecord(new ConsumerRecord<>("my-topic",
```

BUFFERED MOCK CONSUMER



A custom Mock Consumer

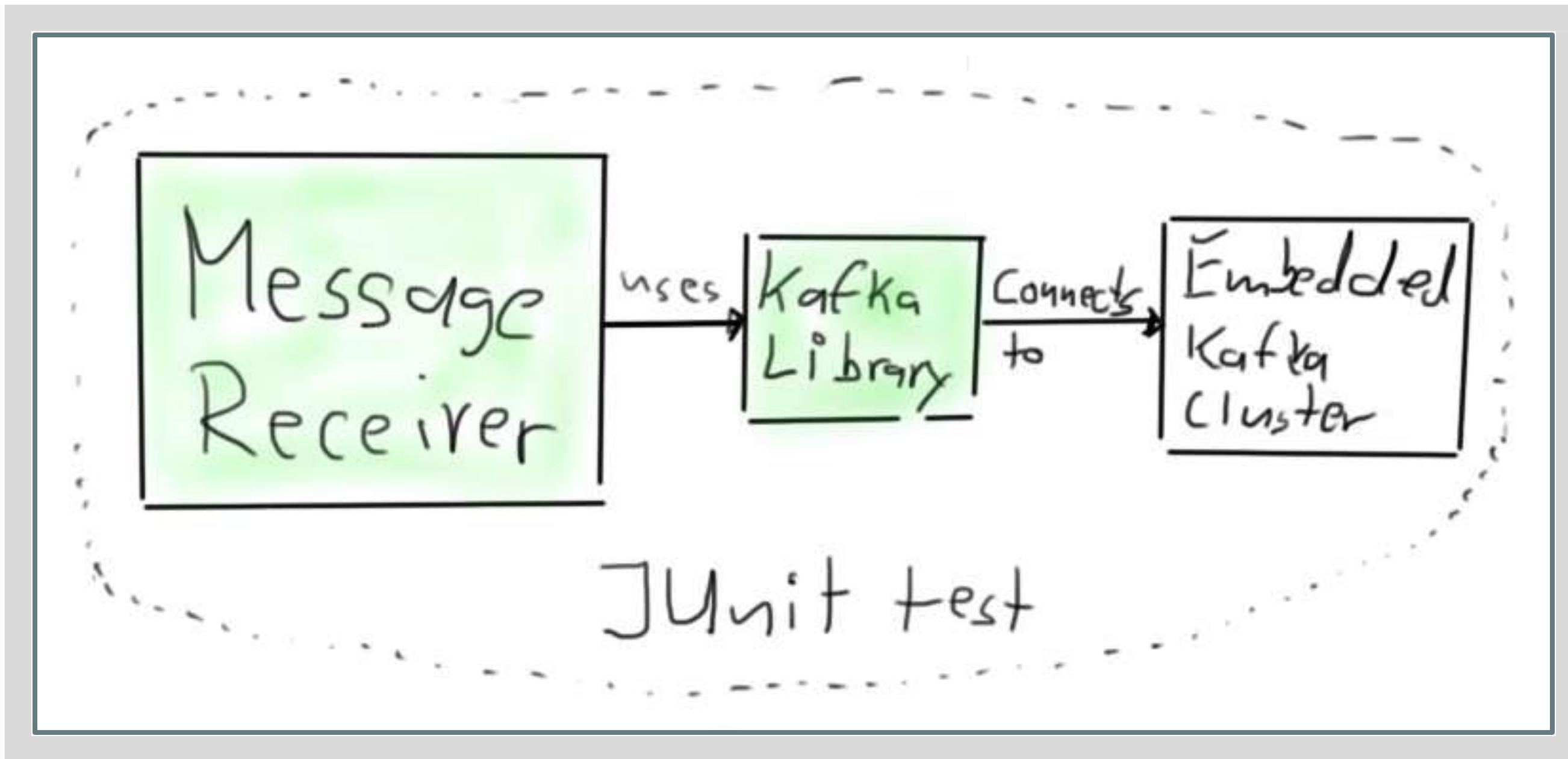
UPDATED TEST

```
@Test
public void withBufferedMockConsumer() {
    // arrange
    var mockConsumer =
        new BufferedMockConsumer<String, String>("my-topic");
    mockConsumer.buffer("HELLO WORLD");

    // act
    var sut = new MessageReceiver(mockConsumer, "my-topic",
        mockRepository, this::runOnce);
    sut.run();

    // assert
    verify(mockRepository, times(1)).saveMessage("HELLO WORLD");
}
```

EMBEDDED KAFKA - CONTEXT



EMBEDDED KAFKA - EXAMPLE TEST

```
@Mock Repository mockRepository;

@RegisterExtension
public static final SharedKafkaTestResource sharedKafkaTestResource =
    new SharedKafkaTestResource();

@Test
public void withEmbeddedKafkaCluster() {
    // arrange
    var kafkaTestUtils = sharedKafkaTestResource.getKafkaTestUtils(); var
    testRecord = Map.of("".getBytes(), "HELLO WORLD".getBytes());
    kafkaTestUtils.produceRecords(testRecord, "my-topic", 0);

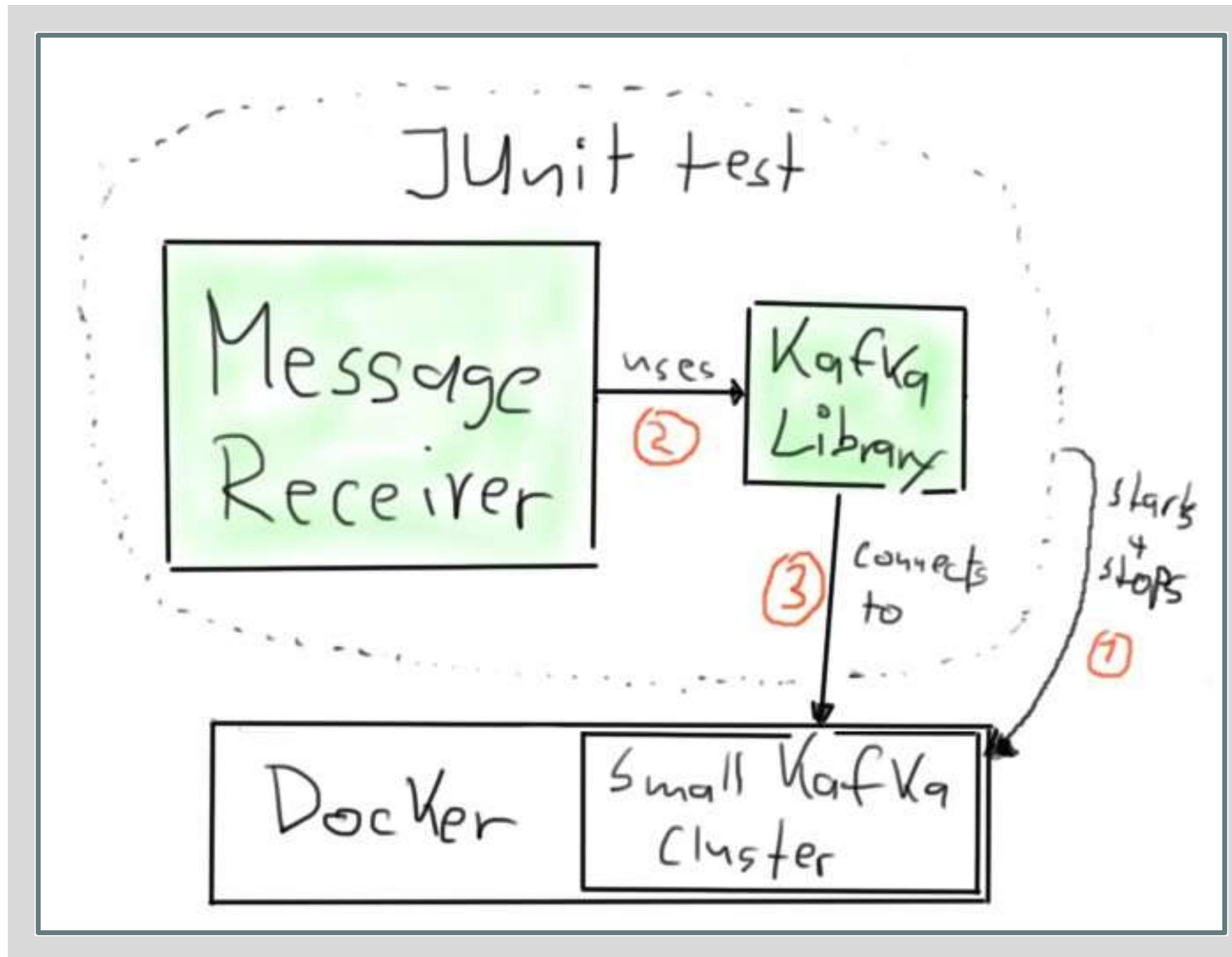
    var props = new Properties();
    props.put("group.id", "mygroup");
    props.put("auto.offset.reset", "earliest"); // this one is needed
    var consumer = kafkaTestUtils.getKafkaConsumer(
        StringDeserializer.class, StringDeserializer.class, props);
```

TESTCONTAINERS



- *...a Java library that supports JUnit tests, providing lightweight, throwaway instances of... anything else that can run in a Docker container.*

TESTCONTAINERS - CONTEXT



TESTCONTAINERS - EXAMPLE TEST

```
@Mock Repository mockRepository;

@Container
private static final KafkaContainer KAFKA = new KafkaContainer();

@Test
public void withTestcontainers() {
    // arrange
    var producerProps = new Properties(); producerProps.put("bootstrap.servers",
KAFKA.getBootstrapServers()); producerProps.put("key.serializer",
StringSerializer.class); producerProps.put("value.serializer",
StringSerializer.class);

    var consumerProps = new Properties(); consumerProps.put("bootstrap.servers",
KAFKA.getBootstrapServers()); consumerProps.put("group.id", "mygroup");
consumerProps.put("auto.offset.reset", "earliest");
consumerProps.put("key.deserializer", StringDeserializer.class);
consumerProps.put("value.deserializer", StringDeserializer.class);
```

A POSSIBLE APPROACH

- majority: **Kafka Fake objects** combined with an own *BufferedMockConsumer*
- a couple of integration tests based on Testcontainers

Load Testing

- **CLI**
 - *kafka-producer-perf-test.sh*
 - *kafka-consumer-perf-test.sh*
- **Measure**
 - MB/sec
 - nMsg/Sec
- **JMETER**
-

PARTITION SIZING

An Example – How many partitions?

- How many events / second (throughput) do you want?
- Start with measuring write throughput with a single partition
- Then measure read throughput with a single partition.

The process

```
kafka-producer-perf-test --producer.config config.properties --throughput  
2000 --num-records 100000 --record-size 1024 --topic <TOPIC_NAME>
```

7501 records sent, 1499.3 records/sec (1.46 MB/sec), 2124.3 ms avg latency, 1307.0 ms max latency.
9870 records sent, 1273.6 records/sec (1.23 MB/sec), 2343.6 ms avg latency, 1452.0 ms max latency.
8805 records sent, 1358.9 records/sec (1.32 MB/sec), 2713.4 ms avg latency, 1982.0 ms max latency.
8355 records sent, 1160.7 records/sec (1.62 MB/sec), 2426.3 ms avg latency, 2783.0 ms max latency.
8925 records sent, 1284.6 records/sec (1.24 MB/sec), 3103.8 ms avg latency, 3353.0 ms max latency.
8820 records sent, 1361.5 records/sec (1.32 MB/sec), 3543.1 ms avg latency, 3921.0 ms max latency.
10290 records sent, 1054.7 records/sec (1.01 MB/sec), 3829.9 ms avg latency, 3943.0 ms max latency.
9000 records sent, 1398.2 records/sec (1.36 MB/sec), 3872.3 ms avg latency, 4291.0 ms max latency.
10125 records sent, 2022.6 records/sec (1.98 MB/sec), 4317.7 ms avg latency, 4490.0 ms max latency.
9660 records sent, 1231.6 records/sec (1.29 MB/sec), 4302.4 ms avg latency, 4430.0 ms max latency.

How does it work?

- Only once in the previous one, do you see it meet the 2K events/second threshold.
- The next step is to change the target throughput until you reach a number of events per second that will be constant throughout all batches sent.
- This will be your measured write throughput.
- If you know where the contention is; and if you can do something about, instead of changing your target throughput you can simply work towards fixing the issues.
- **These two ifs are usually mutually exclusive.**

The process

```
kafka-consumer-perf-test --bootstrap-server <BROKER_ENDPOINT> --  
messages 100000 --topic <TOPIC_NAME>
```

start.time, end.time, data.consumed.in.MB, MB.sec,
data.consumed.in.nMsg, nMsg.sec, rebalance.time.ms, fetch.time.ms,
fetch.MB.sec, fetch.nMsg.sec

2022-06-29 17:26:05:891, 2022-06-29 17:26:21:816, 98.1252, 6.1617,
100489, 6310.1413, 1767, 14158, 6.9307, 7097.6833

The process

- MB.sec **Throughput in MB/s**
- nMsg.sec **Number of events fetched**
- In our example, throughput is 6.1617 Mb/s and 6310.1413 events per second
- If Payload Size * Number of events ~ MB/sec, you are good to go.

Just like write throughput, you will need to adjust so that different executions give similar results

A Thumb rule = Read Throughput ~ 3* Write Throughput

The calculation

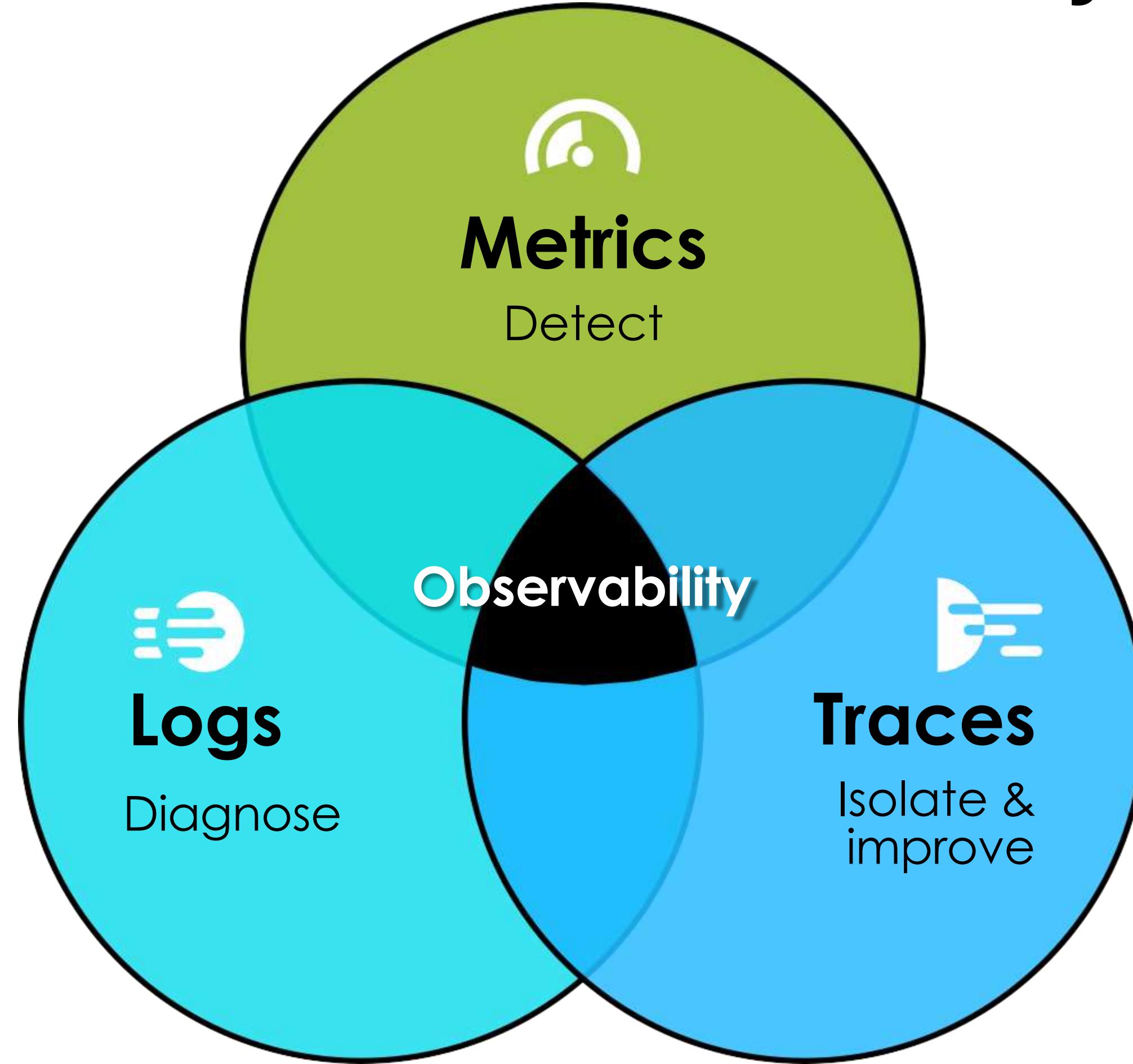
- **NUMBER_PARTITION = MAX (T/W, T/R)**
- Where
 - T = Throughput
 - W = Measured Write Throughput
 - R = Measured Read Throughput

Partitions - Durability

- Each partition of a topic has a replica of 1 on another broker. (Default replication is enabled with replica factor of 1)
- Number of partitions in
$$\sum_{i=1}^N Partitions_i \times ReplicaFactor_i$$
- Where
 - i = total number of topics
 - $Partitions_i$ = Number of partitions for topic i
 - $ReplicaFactor_i$ = Replication Factor for topic i

Observability

The vision: unified observability



Pillars of Observability

Logs/events



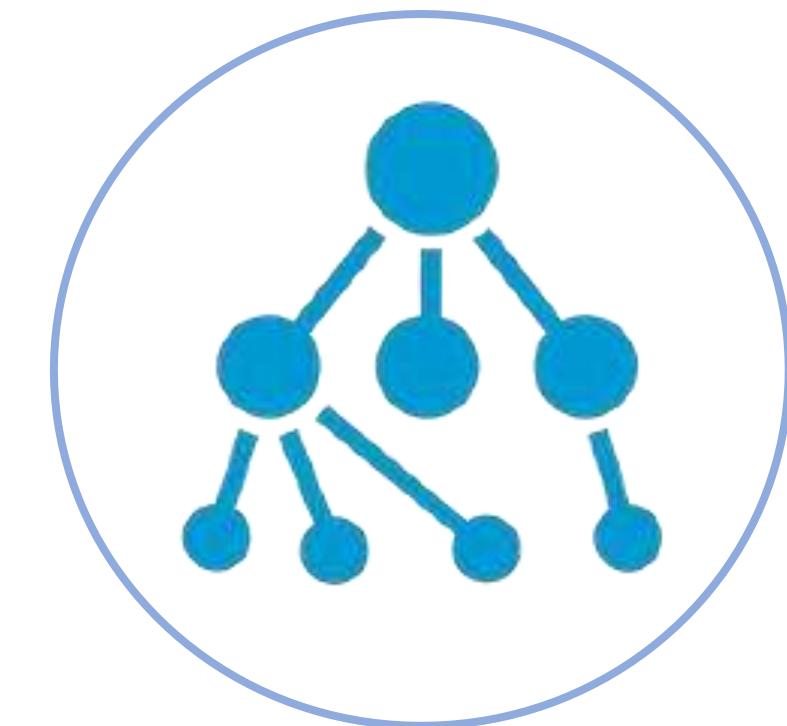
Immutable records of discrete events that happen over time

Metrics



Numbers describing a particular process or activity measured over intervals of time

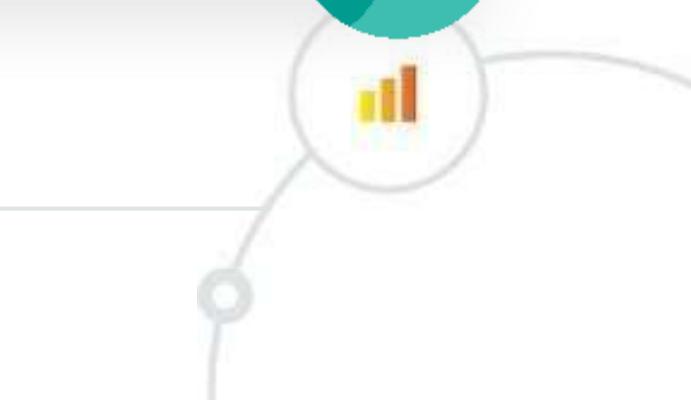
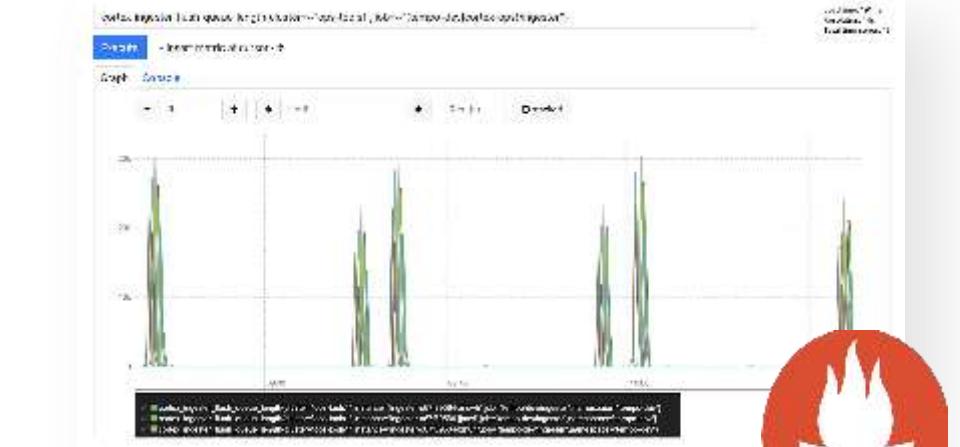
Traces

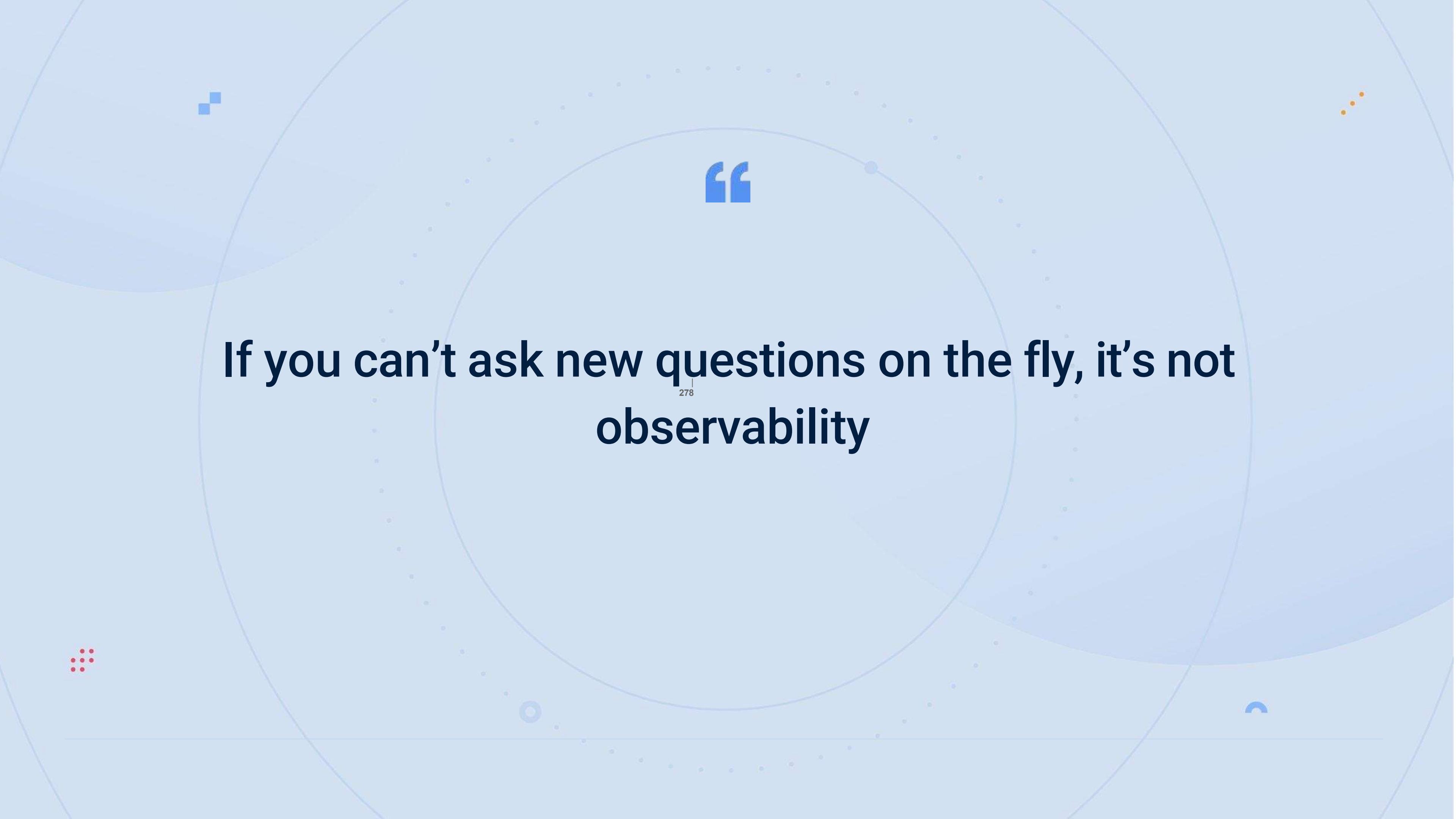


Data that shows, for each invocation of each downstream service, which instance was called, which method within that instance was invoked, how the request performed, and what the results were

Source: A Beginners guide to Observability by Splunk

Disparate systems. Disparate data.





If you can't ask new questions on the fly, it's not
observability

278

LOGS

- Server logs
- Application logs
- **NOT TO BE CONFUSED WITH KAFKA MESSAGE LOGS**
- How do we get logs in a single place?
 - Multiple Brokers
 - Multiple clusters
 - Multiple producers
 - Multiple consumers

What should be logged?

- *Usually you filter out and only look for understandable logs*
 - *Searching for needle in a barn is not a fun activity*
- *Logs are not a good place to store sensitive data*
 - *“user %s authorised with password %s bought item SKU#%s” is my favourite.*
- *You want to see patterns, not individual errors*
 - *“%s caught for %s user, while in %s” is hard to find, when you are not 100% certain what are you looking for*
 - *“exception caught in purchase pipeline” is better, I’ll cover the details in next topic.*

Application Logging

- *Understanding/predicting necessary information*
 - *Do you need a particular field?*
 - *Is it useful now?*
 - *How about future?*
 - *What can you aggregate from a field?*
- *Think about aggregations*
 - *Average over time? You need a number.*
 - *Histogram on field? You need limited set of possible values.*
 - *Pie chart? Limited set of values.*
 - *... (yes, I don't mention geo based aggregations, because you can google that one out)*
- *Use Libraries for Tracing*
 - *OpenTelemetry is only one*
 - *Context should be propagated to allow for easy analysis*

Log Appenders

- Apache Kafka brokers and controllers use **Log4j 2** for logging
- Can configure a variety of appenders to route logs to different destinations.

Log Appenders

Appender Name	Purpose
KafkaAppender	General broker logs (INFO level and above)
StateChangeAppender	Logs state transitions in the Kafka cluster
RequestAppender	Captures client requests handled by the broker
CleanerAppender	Logs compaction and cleanup activity for log segments
ControllerAppender	Logs controller-specific events and decisions
AuthorizerAppender	Logs authorization decisions (useful for debugging ACLs)
MetadataServiceAppender	Tracks metadata changes across the cluster
AuditLogAppender	Captures audit logs, especially when audit messages fail to reach Kafka
DataBalancerAppender	Logs activity related to Kafka's data balancing mechanisms
ConsoleAppender	Outputs logs to stdout or stderr
RollingFileAppender	Writes logs to files with time-based rotation

Log Appenders

- Logging options can be overridden in config files using
- ***KAFKA_LOG4J_OPTS="-Dlog4j.configuration=file:/path/to/log4j2.yaml"***
- Each Appender can be tuned with own layout, file pattern, log level, Rolling levels
- The additivity property should be set to false to avoid recursive logging. E.g. kafka.request.logger.

LOG APPENDERS

- **Audit & Authorizer Logging:** Enable finer-grained access logs by setting `kafka.authorizer.logger` level to DEBUG or TRACE.
- **Network Traffic Inspection:** For debugging SASL/mTLS setups, add logging for `org.apache.kafka.common.network`.
- **DR & Replication Events:** Include `kafka.server.ReplicaFetcherThread` logger to observe replica lag and recovery.
- **Loop Prevention:** Log headers and interceptors via `org.apache.kafka.common.header` for custom routing logic.

Logging - common patterns

Leader Elections

```
INFO [Controller id=1] New leader for partition [topic-name,0] is 2
```

```
WARN [Controller id=1] Error while electing leader for partition [topic-name,0]
```

```
ERROR [Controller id=1] Failed to elect leader for partition [topic-name,0] due to ZkException
```

- Look for **Controller id** and **New leader** messages in controller.log.
- Frequent elections may indicate unstable brokers or ZooKeeper flaps

Logging - common patterns

Rebalance

```
INFO [GroupCoordinator 1]: Stabilized group consumer-group with generation 5
```

```
INFO [Consumer clientId=consumer-1] Revoke previously assigned partitions [topic-1-0, topic-2-1]
```

```
INFO [Consumer clientId=consumer-1] Assigned new partitions [topic-1-1, topic-2-0]
```

```
WARN [GroupCoordinator 1]: Delayed rebalance due to pending state
```

- Use **group.id**, **generation**, and **clientId** to trace rebalance cycles.
- Delayed or frequent rebalances can impact latency and throughput.

Logging – common patterns

Auth – SASL, SSL or ACL misconfiguration

```
ERROR [SocketServer brokerId=1] Connection to node 2 failed authentication due to: SSL handshake failed  
WARN [KafkaServer id=1] Principal = User:kafka is not authorized to access Topic: topic-name  
ERROR [KafkaServer id=1] SASL authentication failed using mechanism GSSAPI
```

- Check `server.log` for SASL, SSL, or Principal errors.
- Match against ACLs and JAAS configs.
- Use `kafka-acls.sh` for verification

Logging - common patterns

General errors - broker crashes, fetch failures, and replication issues.

```
ERROR [KafkaServer id=1] Fatal error during startup. Exiting.  
ERROR [ReplicaFetcherThread-0-1] Error fetching data from leader for partition topic-1-0  
WARN [LogCleaner] Unexpected exception during log cleaning  
ERROR [ZooKeeperClient] Session expired. Reconnecting...
```

- Use **ERROR** and **WARN** levels to filter critical failures.
- Correlate with `state-change.log`, `controller.log`, and `server.log`.
- Keep Separate Log files.

Metrics

- What should be measured?
- Why are we measuring this?
- How do we aggregate data from multiple sources?
- What should trigger an alarm?
- Who should be notified?

Traces

- Where are the bottlenecks?
- We have existing code. How do we measure it?

Other challenges

- How do I Correlate

Logs

Metrics

Traces

JUST AN ASIDE

- Correlation is hard and is an ideal candidate for machine learning and AI
- This helps predict failures before they happen.
- **THIS IS THE FUTURE.** SRE/DEVOPS have now totally adopted this.

Secure Kafka

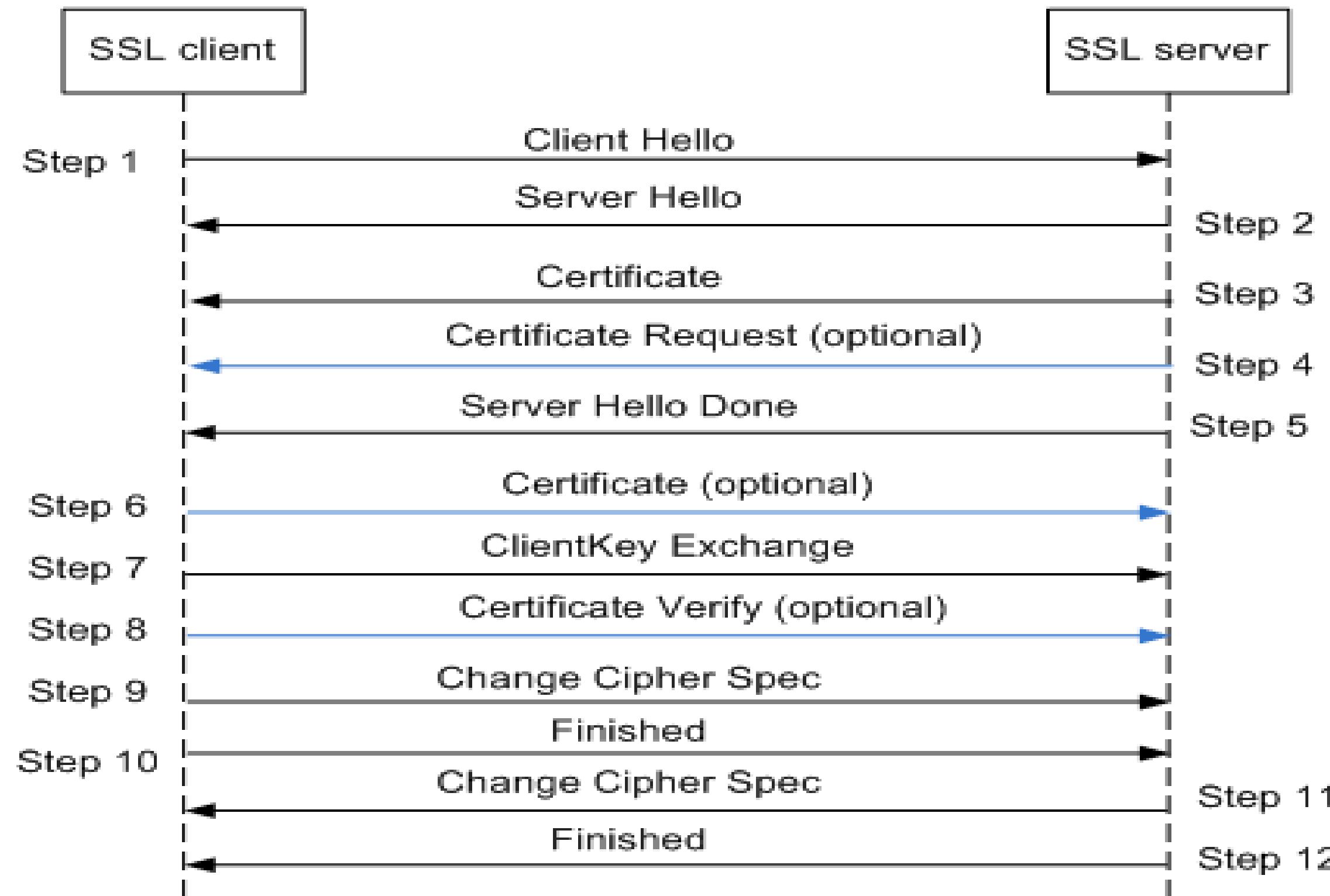
KAFKA Security

- How can we prevent rogue agents to publishing/consuming data from Kafka
- How can we encrypt the data that's flowing through the network
- How can we give permissions to a topic to specific group or users

KAFKA

- By Default, we use PLAINTEXT
 - No Encryption
 - All in plain text
 - No Authentication or Authorization

Kafka Security - SSL



Kafka Security

- **Simple Authentication and Security Layer, or SASL**
 - Provides flexibility in using Login Mechanisms
 - One can use Kerberos , LDAP or simple passwords to authenticate.
- **JAAS Login**
 - Before client & server can handshake , they need to authenticate with Kerberos or other Identity Provider.
 - JAAS provides a pluggable way of providing user credentials. One can easily add LDAP or other mechanism just by changing a config file.
-

Kafka Security

```
KafkaServer {  
    com.sun.security.auth.module.Krb5LoginModule required  
    useKeyTab=true  
    storeKey=true  
    serviceName="kafka"  
    keyTab="/vagrant/keytabs/kafka1.keytab"  
    principal="kafka/host@EXAMPLE.COM";  
};
```

```
KafkaClient {  
    com.sun.security.auth.module.Krb5LoginModule required  
    useKeyTab=true  
    storeKey=true  
    serviceName="kafka"  
    keyTab="/vagrant/keytabs/client1.keytab"  
    principal="client/host@EXAMPLE.COM";  
};
```

Let's Get our hands dirty

- **listeners=PLAINTEXT://:9092**
- **advertised.listeners=PLAINTEXT://your.host.name:9092**
- **listener.security.protocol.map=PLAINTEXT:PLAINTEXT**
- **inter.broker.listener.name=PLAINTEXT**

THIS IS PLAIN SIMPLE CONFIGURATION - No Authentication Every thing in plain text

Let's Get our hands dirty

- Let's now add a single user (Still no Encryption)
- We need to change the broker properties and add a JAAS Config file
- Also clients can be changed.

Let's Get our hands dirty

- **listeners=SASL_PLAINTEXT://:9093**
- **advertised.listeners=SASL_PLAINTEXT://your.host.name:9093**
- **listener.security.protocol.map=SASL_PLAINTEXT:SASL_PLAINTEXT**
- **inter.broker.listener.name=SASL_PLAINTEXT**
- **sasl.mechanism.inter.broker.protocol=PLAIN**
- **security.inter.broker.protocol=SASL_PLAINTEXT**

Let's Get our hands dirty

```
KafkaServer {  
    org.apache.kafka.common.security/plain.PlainLoginModule  
    required  
        username="admin"  
        password="admin-secret"  
        user_admin="admin-secret"  
        user_alice="alice-password";  
};
```

Let's Get our hands dirty

- **KAFKA_OPTS="-Djava.security.auth.login.config=/path/to/kafka_server_jaas.conf"**

Let's Get our hands dirty

- KafkaClient {
org.apache.kafka.common.security.plain.PlainLoginModule
required username="alice" password="alice-password";};
- export KAFKA_OPTS="-
Djava.security.auth.login.config=/path/to/client_jaas.conf"
bin/kafka-console-producer.sh \ --broker-list
your.host.name:9093 \ --topic test \ --producer.config
client.properties

Let's Get our hands dirty

- The PlainLoginModule is never ever to be used in Public
-
- What we have done so far is to use SASL/PLAIN and the broker will create an in-memory mapping
- E.g. user_alice=“alice-password”
- The user alice has a password “password”

Kafka Security

- Kafka doesn't natively support external databases for password verification in SASL/PLAIN.
- Can integrate a database-backed credential store by implementing a custom JAAS login module, which Kafka supports.

Let's Get our hands dirty

- Alternatively
 - Put a proxy service (e.g., Envoy, OAuth2 sidecar, or custom REST gateway) in front of Kafka
 - Handle auth via database or identity provider (IdP)
 - Only route verified clients to Kafka

This isolates your auth logic and can support:

- OAuth
- LDAP
- JWT
- Database credential validation

Kafka Security

- Kafka's JAAS setup is designed for simplicity and JVM-local credential mapping:
- SASL/PLAIN: In-memory
- SASL/SCRAM: Credential hashes stored in ??

Kafka Security

- With SASL/PLAIN,
 - There's no call to a file, LDAP, or database.
 - Everything stays inside the JVM memory via the user_<username> entries.
 - It's simple, but risky for production unless network is restricted.

Recommended to move to SCRAM or SSL-based transport

Avoid hardcoding secrets (or wrap them with Docker secrets/env vars/vaults)

Kafka Security

- With SASL/PLAIN,

Concern	Detail
 Passwords in clear	Sent over the wire in plaintext
 Stored unencrypted	Broker reads passwords from JAAS as raw strings
 No hashing	Passwords are not salted or hashed—just compared verbatim
 Memory-resident	User creds are held in JVM memory—visible via JMX/thread dumps

KAFKA Security

- With SASL/SCRAM, credentials are hashed using PBKDF2
- These are stored in ---???
- Raw Passwords are not transmitted.

KAFKA Security

- On server side.

```
listeners=SASL_PLAINTEXT://:9093
advertised.listeners=SASL_PLAINTEXT://your.host.name:9093
listener.security.protocol.map=SASL_PLAINTEXT:SASL_PLAINTEXT
inter.broker.listener.name=SASL_PLAINTEXT
sasl.mechanism.inter.broker.protocol=SCRAM-SHA-512
security.inter.broker.protocol=SASL_PLAINTEXT
```

KAFKA Security

- On server side.

```
KafkaServer {  
    org.apache.kafka.common.security.scram.ScramLoginModule required  
    username="admin"  
    password="admin-secret";  
};
```

A few points to note here

User Credentials are not stored here.
These users will need to be provisioned first.

KAFKA Security

- On server side

```
# Create user 'alice' with password 'alice-password'  
bin/kafka-configs.sh --zookeeper localhost:2181 \  
--alter --add-config 'SCRAM-SHA-512=[password=alice-password]' \  
--entity-type users --entity-name alice
```

```
# (Optional) Add inter-broker user  
bin/kafka-configs.sh --zookeeper localhost:2181 \  
--alter --add-config 'SCRAM-SHA-512=[password=admin-secret]' \  
--entity-type users --entity-name admin
```

```
# if in KRAFT mode, replace --zookeeper with --bootstrap-server and add  
--admin
```

KAFKA Security

- On client side.

```
KafkaClient {  
    org.apache.kafka.common.security.scram.ScramLoginModule required  
    username="alice"  
    password="alice-password";  
};
```

```
# Client Properties  
security.protocol=SASL_PLAINTEXT  
sasl.mechanism=SCRAM-SHA-512
```

Kafka Security

Component	Status
Broker JAAS	SCRAM module
ZooKeeper/KRaft	User credentials stored securely
Client JAAS	Password passed to broker
Wire Protocol	Still PLAINTEXT (add SSL later if needed)

Kafka Security - Review

- We started with PLAINTEXT – No Authentication
- Add a simple User WITH PLAINTEXT (SASL_PLAIN)
- Then moved to SASL_SCRAM
- Now Moving to SASL_SSL

KAFKA Security

- On server side.

```
listeners=SASL_SSL://:9093
advertised.listeners=SASL_SSL://your.host.name:9093
listener.security.protocol.map=SASL_SSL:SASL_SSL
inter.broker.listener.name=SASL_SSL
sasl.mechanism.inter.broker.protocol=SCRAM-SHA-512
security.inter.broker.protocol=SASL_SSL
ssl.keystore.location=/path/to/keystore.jks
ssl.keystore.password=keystore-password
ssl.key.password=key-password
ssl.truststore.location=/path/to/truststore.jks
ssl.truststore.password=truststore-password
```

KAFKA Security

- On server side.

```
KafkaServer {  
    org.apache.kafka.common.security.scram.ScramLoginModule required  
    username="admin"  
    password="admin-secret";  
};
```

A few points to note here

User Credentials are not stored here.
These users will need to be provisioned first.

KAFKA Security

- On client side.

```
KafkaClient {  
    org.apache.kafka.common.security.scram.ScramLoginModule required  
    username="alice"  
    password="alice-password";  
};
```

```
security.protocol=SASL_SSL  
sasl.mechanism=SCRAM-SHA-512  
ssl.truststore.location=/path/to/truststore.jks  
ssl.truststore.password=truststore-password
```

Kafka Security

Component	Status
Broker JAAS	SCRAM module
ZooKeeper/KRaft	User credentials stored securely
Client JAAS	Password passed to broker
Wire Protocol	Encrypted with SSL

Kafka Security

- As you can see this is very cumbersome with one cluster.
- Factor this for a large number of clusters/brokers and you will see the need for automation

Kafka Security

- Step #1 Generate Keystores and Truststore (You do need keytool – Unix based ☺)
- Step #2 These need to be propagated to all the brokers
- Step #3 Cert Rotation Scripts (NEAR ZERO DOWNTIME)

Kafka Security

- Step #2
 - Symlink cert paths to `server.properties` via volume mounts
 - Inject broker-specific env vars (e.g. `BROKER_ID`, `CN`) to template configs
 - Optionally render JAAS and `server.properties` using `Jinja2` or shell

Kafka Security

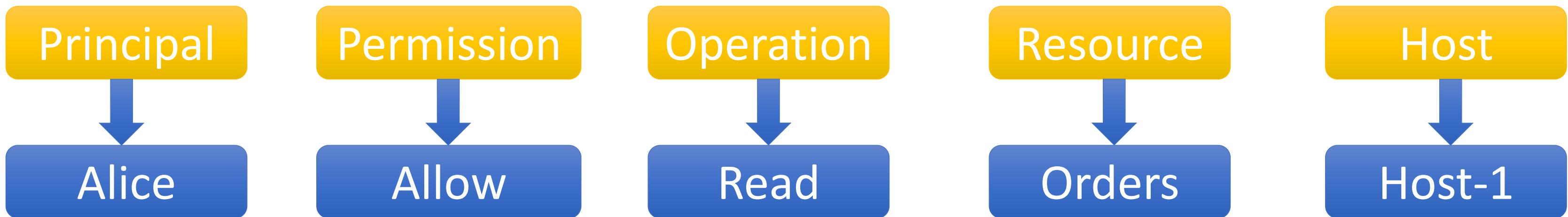
- Step #3
 - To rotate certs safely:
 - Generate new certs in parallel (with a different alias or CN)
 - Update keystore/truststore atomically
 - Bounce brokers one at a time to avoid cluster unavailability

Kafka Security

- So far we have covered Authentication.
- Kafka allows to set authorization rules
 - Who can do what on which Objects
 - Pluggable Authorizers
 - Access Control List (ACL) based approach

Access Control Lists

- Alice is Allowed to Read from Orders-topic from Host-1



Principal

- PrincipalType:Name
- Supported types: User
- Extensible so users can add their own types
- Wild Card User:*

Operation

- **Read, Write, Create, Delete, Alter, Describe, ClusterAction, All**

Resource

- ResourceType:ResourceName
- Topic, Cluster and ConsumerGroup
- Wild card resource ResourceType:*

Permissions

- Allow and Deny
- Anyone without an explicit Allow ACL is denied
- Then why do we have Deny?
- Deny works as negation
- Deny takes precedence over Allow Acls

Hosts

- Why provide this granularity?
- Allows authorizer to provide firewall type security even in non secure environment.
- * as Wild card.

Configuration

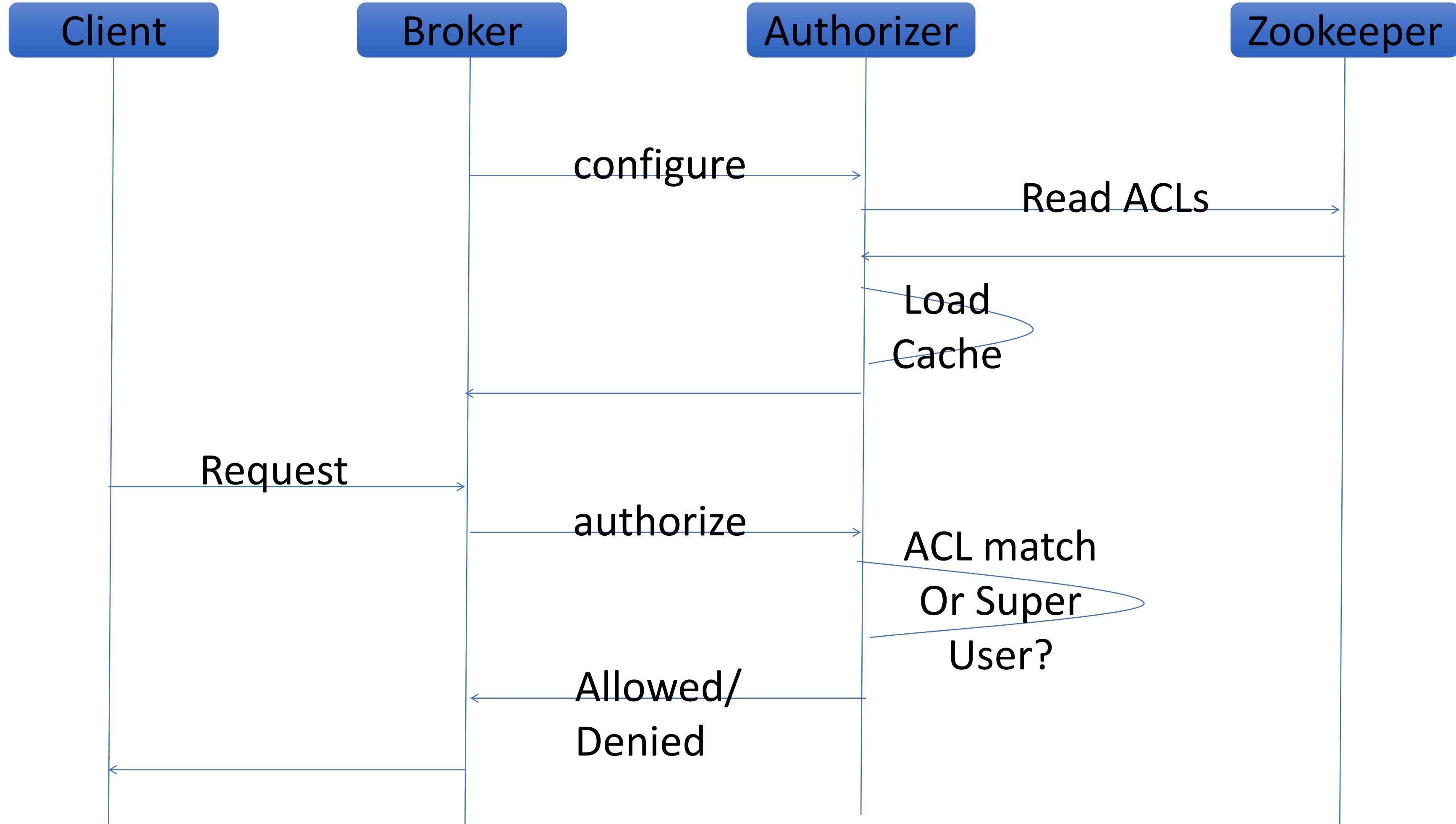
- Authorizer class
- Super users
- Authorizer properties
- Default behavior for resources with no ACLs

SimpleAclAuthorizer

- Out of box authorizer implementation.
- Stores all of its ACLs in zookeeper.
- In built ACL cache to avoid performance penalty.
- Provides authorizer audit log.

What about KRAFT mode?

- **StandardAuthorizer**
 - `org.apache.kafka.metadata.authorizer.StandardAuthorizer`
- `authorizer.class.name=org.apache.kafka.metadata.authorizer.StandardAuthorizer`



CLI

- Add, Remove and List acls
- Convenience options:
 - producer and --consumer.

Zookeeper

- Kafka's metadata store
- Has its own security mechanism that supports SASL and MD5-DIGEST for establishing identity and ACL based authorization
- Create , Delete directly interacts with zookeeper

FAQs

- **How do you do end-to-end encryption in Kafka?**
 - Apache Kafka does not support end-to-end encryption natively, but you can achieve this by using a combination of TLS for network connections and disk encryption on the brokers.
- **Is data encrypted at rest in Kafka?**
 - By default, Apache Kafka data is not encrypted at rest. Encryption can be provided at the OS or disk level using third-party tools.

FAQs

- **SASL Support in KAFKA**
 - Apache Kafka supports various SASL mechanisms, such as GSSAPI (Kerberos), OAUTHBEARER, SCRAM, LDAP, and PLAIN.
- **How do I set up ACLs in Kafka?**
 - To configure ACLs in Apache Kafka, first set the `authorizer.class.name` property in `server.properties`:
`authorizer.class.name=kafka.security.authorizer.AclAuthorizer.`
 - Add and remove ACLs using the `kafka-acls.sh` script.

FAQs

- **Does Kafka support Role-Based Access Control (RBAC)?**
 - Apache Kafka does not support Role-Based Access Control (RBAC) by default.
 - Confluent adds RBAC support to Kafka, allowing you to define group policies for accessing services (reading/writing to topics, accessing Schema Registry, etc.) and environments (dev/staging/prod, etc.), across all clusters.

KAFKA Security Summary

- Apache Kafka Security Models
 - PLAINTEXT
 - SSL
 - SASL_PLAINTEXT
 - SASL_SSL
- Apache Kafka Security Models - A Ready Reckoner
- How to Troubleshoot security issues
- Most Common Errors
- Apache Kafka Security - Dos and Don'ts

What Are We Securing?

- End-user Authentication (Is user who he/she claims to be?)
 - User Authorization (Does authenticated user have access to this resource?)
 - In-flight data i.e. Communication between
 - Kafka Broker <--> Kafka Clients (Consumer/Producers)
 - Kafka Broker <--> Kafka Broker
 - Kafka Broker <--> Zookeeper
- What Are We NOT Securing?
- Data persisted on-disk, e.g. security through data encryption

Apache Kafka Security Models

PLAINTEXT

- No Authentication / No Authorization / insecure channel => ZERO security
- Default security method
- To be used only for Proof-of-Concept
- Absolutely NOT recommended for use in Dev/Test/Prod environment

Apache Kafka Security Models

SSL

- X.509 Certificate based model - only secures the HTTP channel
- Performs certificate based host authorization
- No User Authentication / Authorization
- How to configure
 - Setup per-node certificate truststore/keystore for brokers & clients

Broker-side:	Client-side:
listeners=SSL://127.0.0.1:6667	security.protocol = SSL
inter.broker.protocol=SSL	

Apache Kafka Security Models

SASL_PLAINTEXT (or PLAINTEXTSASL in older version)

- Supports user authentication via
 - Username / Password
 - GSSAPI (Kerberos Ticket)
 - SCRAM (Salted Password)
- Supports User authorization via Kafka ACLs or **Apache Ranger**
- Sends secrets & data over the wire in "**Plain**" format
- How to configure
 - Pre-configure authentication mechanism

Broker-side:	Client-side:
listeners=SASL_PLAINTEXT://127.0.0.1:6667	security.protocol = SASL_PLAINTEXT
inter.broker.protocol=SASL_PLAINTEXT sasl.enabled.mechanism=PLAIN GSSAPI SCRAM	sasl.mechanism = PLAIN GSSAPI SCRAM-SHA-256 SCRAM-SHA-512

Apache Kafka Security Models

SASL_SSL

- Supports user authentication via
 - Username / Password
 - GSSAPI (Kerberos Ticket)
 - SCRAM (Salted Password)
- Supports User authorization via Kafka ACLs or **Apache Ranger**
- Sends secrets & data over the wire in **"Plain" Encrypted**
- How to configure
 - Pre-configure authentication mechanism
 - Setup per-node certificate truststore/keystore for broker(s) & client(s)

Broker-side:	Client-side:
listeners=SASL_SSL://127.0.0.1:6667	security.protocol = SASL_SSL
inter.broker.protocol=SASL_SSL sasl.enabled.mechanism=PLAIN GSSAPI SCRAM	sasl.mechanism = PLAIN GSSAPI SCRAM-SHA-256 SCRAM-SHA-512

Apache Kafka Security Models - A Ready Reckoner

security.protocol	User Authentication	Authorization	Encryption Over Wire
PLAINTEXT	✗	✗	✗
SSL	✗	Host Based (via SSL certificates)	✓
SASL_PLAINTEXT	PLAIN KRB5 SCRAM	Kafka ACLs / Ranger	✗
SASL_SSL	PLAIN KRB5 SCRAM	Kafka ACLs / Ranger	✓

* Available in Apache Kafka 0.9.0 and above

How to troubleshoot security issues?

- Enable Krb debug for SASL clients (consumer/producer)
 - `export KAFKA_OPTS="-Dsasl.security.krb5.debug=true"`
- Enable SSL debug for clients
 - `export KAFKA_OPTS="-Djavax.net.debug=ssl"`
- Enable Krb / SSL debug for Kafka Broker (**AMBARI-24151**)
 - Enable this in console as 'kafka' user & start the Broker from command line:
 - `export KAFKA_KERBEROS_PARAMS="$KAFKA_KERBEROS_PARAMS -Dsasl.security.krb5.debug=true -Djavax.net.debug=ssl"`
 - `/usr/hdp/current/kafka-broker/bin/kafka-server-start.sh -daemon /etc/kafka/conf/server.properties`

* Disable debug properties once you are done with troubleshooting, otherwise it's going to bloat the log files

How to troubleshoot security issues?

- Enable Kafka Broker log4j debug
 - Set `log4j.logger.kafka=DEBUG, kafkaAppender` in `/etc/kafka/conf/log4j.properties`
- Enable Kafka Ranger log4j debug
 - Set `log4j.logger.org.apache.ranger=DEBUG, rangerAppender` in `/etc/kafka/conf/log4j.properties`
- Enable Kafka Client debug
 - Set `log4j.rootLogger=DEBUG, stderr` in `/etc/kafka/conf/tools-log4j.properties`

* Disable debug properties once you are done with troubleshooting, otherwise it's going to bloat the log files

Troubleshoot Using Kafka Console Consumer/Producer

- **Create a Kafka topic**
 - Should be run only on Kafka Broker node as 'kafka' user
- **Use Kafka Console Producer to write messages to above Kafka topic**
 - Can be run from any Kafka client node as any user
 - Make sure that authentication token is acquired and user has permission to 'Describe' & 'Publish'
- **Use Kafka Console Consumer to read messages from the Kafka topic**
 - Can be run from another or same Kafka client as the same or different user
 - Make sure that authentication token is acquired and user has permission to 'Consume'

Most Common Errors

- **javax.security.auth.login.LoginException**
 - Check JAAS configuration
- **Could not login: the client is being asked for a password**
 - Again, issue with JAAS configuration - either Ticket not found or Bad / inaccessible user keytab
- **PKIX path building failed - unable to find valid certification path to requested target**
 - Issue with SSL truststore; most likely truststore not present or readable
- **No User Authentication / Authorization**

Apache Kafka Security - Dos and Don'ts

- No Kerberos = No Security
 - All the pain is well worth it !
- Enabling SSL is only half the story
 - Having SSL without Authentication is meaningless
- Using any SASL (i.e. Authentication) without SSL is dangerous
- Use Apache Ranger for large deployments with many users

Back to KAFKA security

- We have SSL traffic and JAAS configuration.
- We do not yet have mTLS (mutual TLS)
- Here clients pass Certificates to the broker and both sides verify each other (client verifies broker and broker verifies Client)

Kafka Security

- Each Client should generate their own certificates

- Client.properties

ssl.keystore.location=/path/to/client.keystore.jks

ssl.keystore.password=client-keystore-pass

ssl.key.password=client-key-pass

- On the broker side, client authentication is required.

ssl.client.auth=required

- Optionally sign both broker and client certs via a shared CA for centralized trust.

Kafka Security

- In TLS, only server authenticates itself with the client
- In MTLS, both server and client authenticate each other.
- This may seem an overkill – if so read this.
- <https://www.thehindubusinessline.com/info-tech/data-breach-at-pine-labs-exposes-500000-records/article35962296.ece>
- Whenever sensitive information esp. in finance domain is concerned this is a must

KAFKA Security

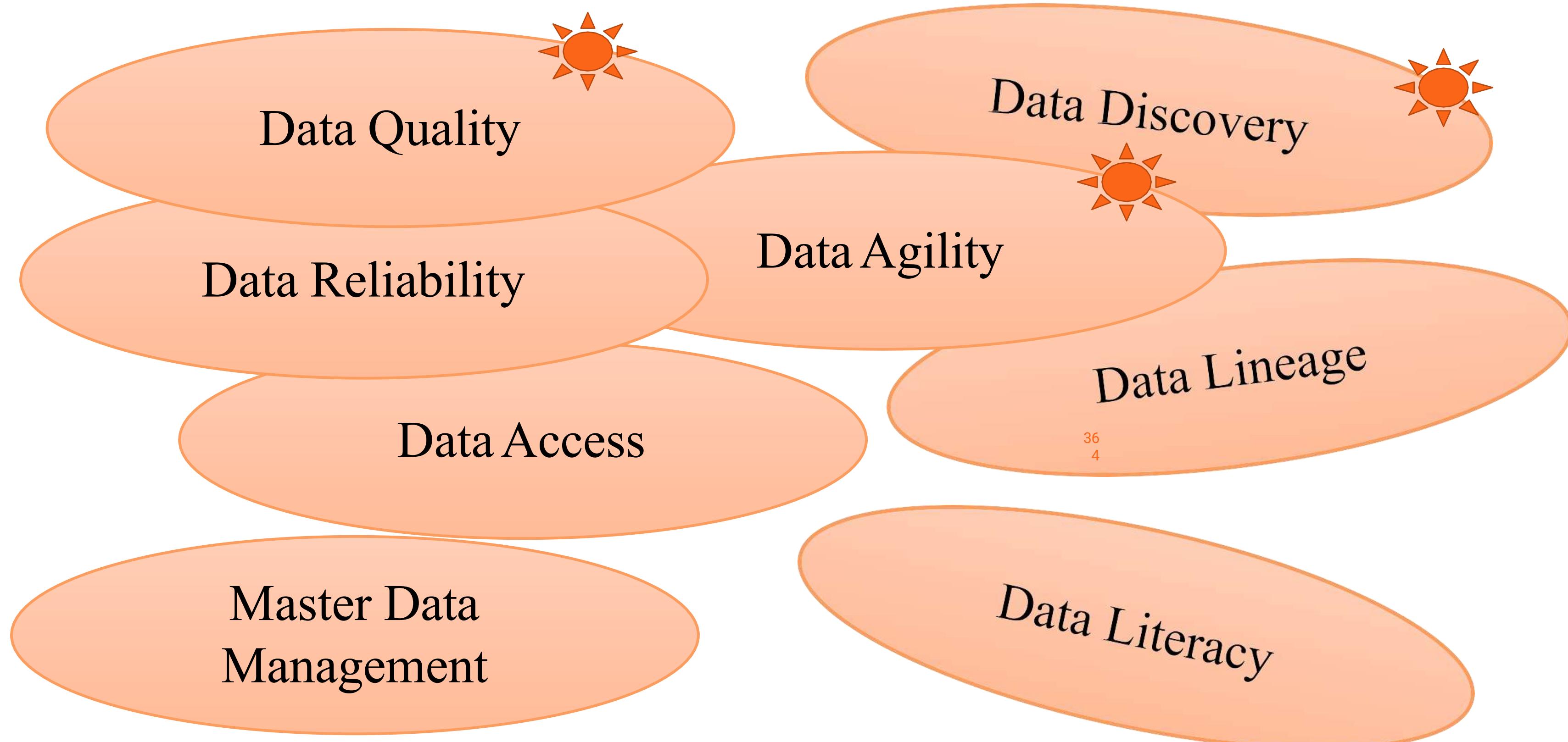
- One way to secure KAFKA end to end with MTLS (almost end to end 😊) is to use 3rd party tools/components.
- This is needed as certificates can and will expire on both ends.
- Hashicorp Vault is one way of doing this. We are not covering this 😊

KAFKA Security

- Workflows will change (Esp. with Certificate Revocation Lists) e.g.
 - Change KAFKA_HOSTS to include
 - -Dcom.sun.security.enableCRLDP=true
Dcom.sun.net.ssl.checkRevocation=true
- Components in your SW and infra architecture will change
- How you deploy software (+updates) will change

Governance

What is Data Governance?

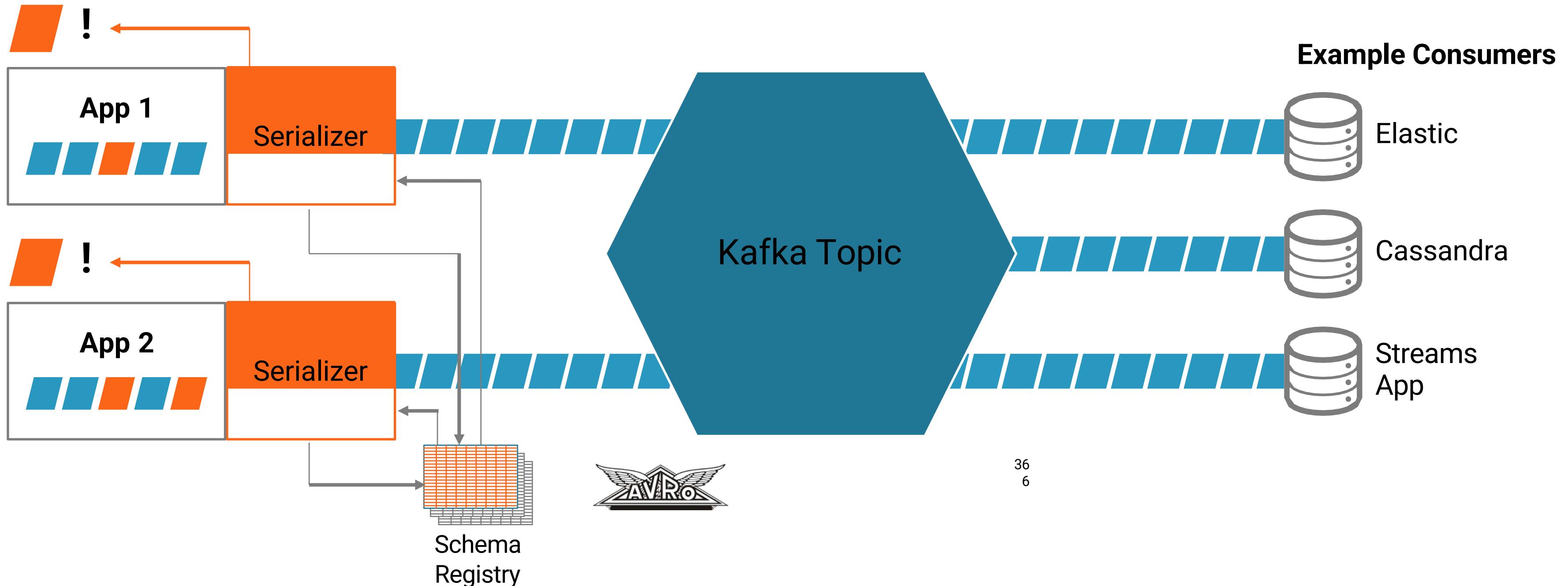


Schemas Are a Key Part of This

•



Schema Registry



Key Points:

- Schema Registry is efficient – due to caching
- Schema Registry is transparent
- Schema Registry is Highly Available
- Schema Registry prevents bad data at the source

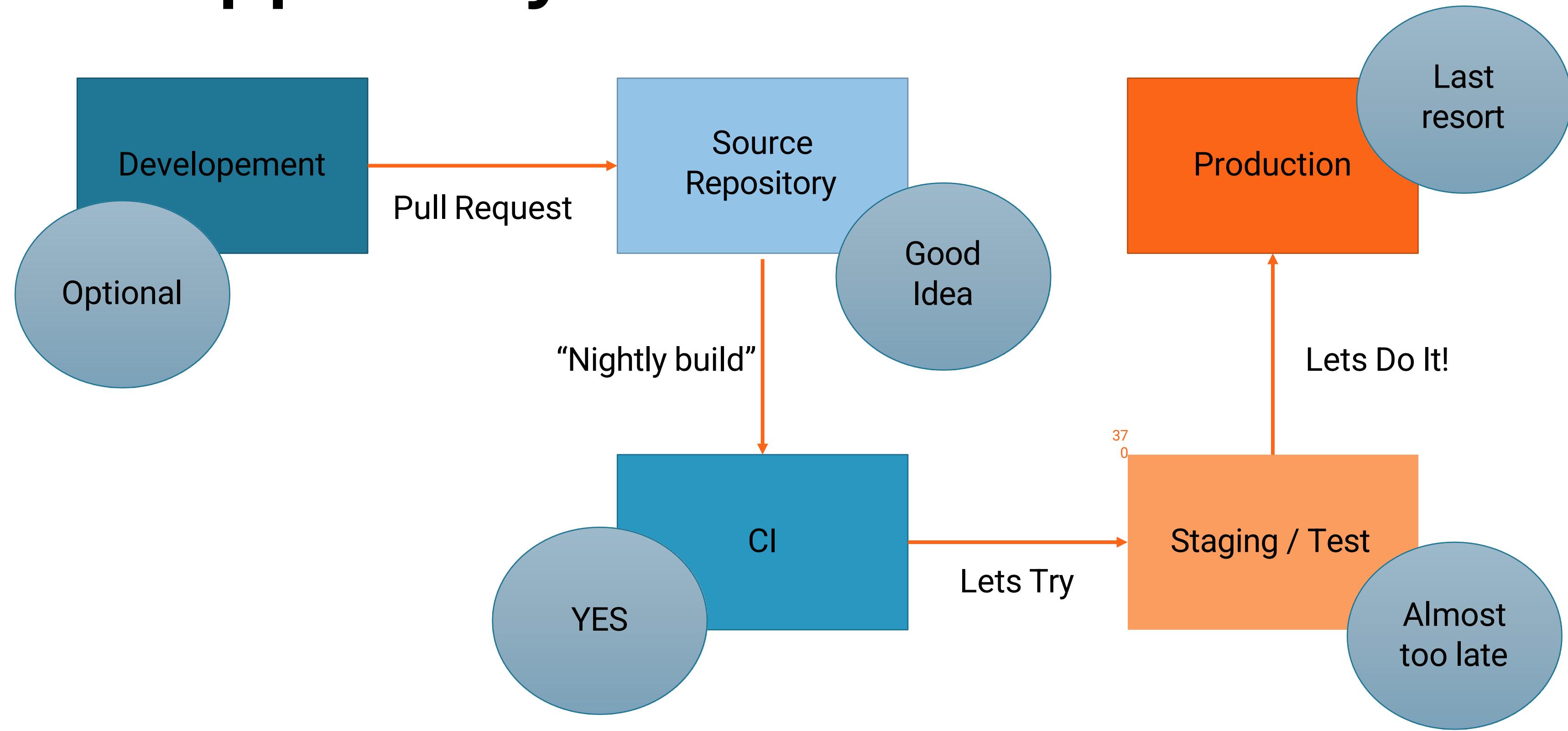
Big Benefits

- Schema Management is part of your entire app lifecycle – from dev to prod
- Single source of truth for all apps and data stores
- No more “Bad Data” randomly breaking things
- Increased agility! add, remove and modify fields with confidence
- Teams can share and discover data
- Smaller messages on the wire and in storage

Compatibility tips!

- **Do you want to upgrade producer without touching consumers?**
 - You want **forward** compatibility.
 - You can add fields.
 - You can delete fields with defaults
- **Do you want to update schema in a DWH but still read old messages?**
 - You want **backward** compatibility
 - You can delete fields
 - You can add fields with defaults
- **Both?**
 - You want **full** compatibility.
 - Default all things!
 - Except the primary key which you never touch.

App Lifecycle



Topics in Durability

- Data Durability
- Tradeoff between latency, throughput , security, costs, benefits
- Idempotency
- Design Considerations – retries, error handling
- Schema updates