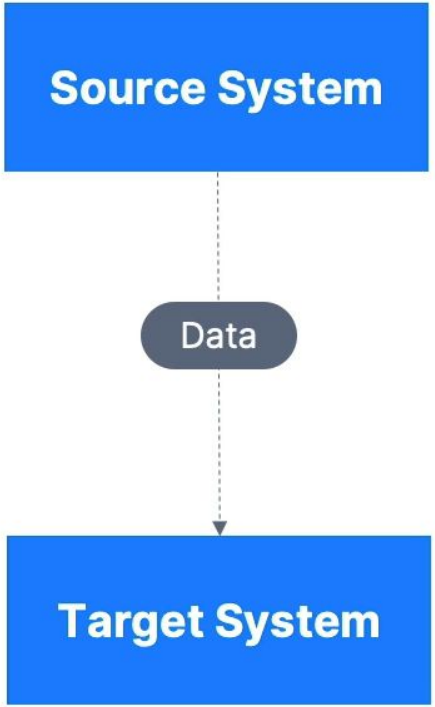
# Kafka

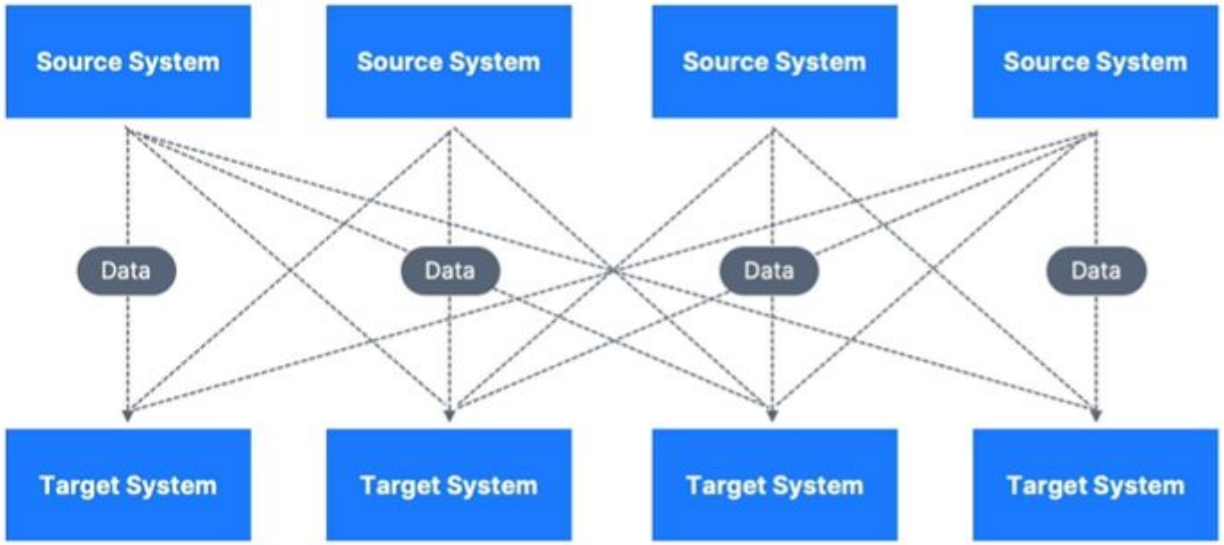
## Distributed Stream Processing System

# How companies start




Simple at first!

# After a while...




Very complicated!

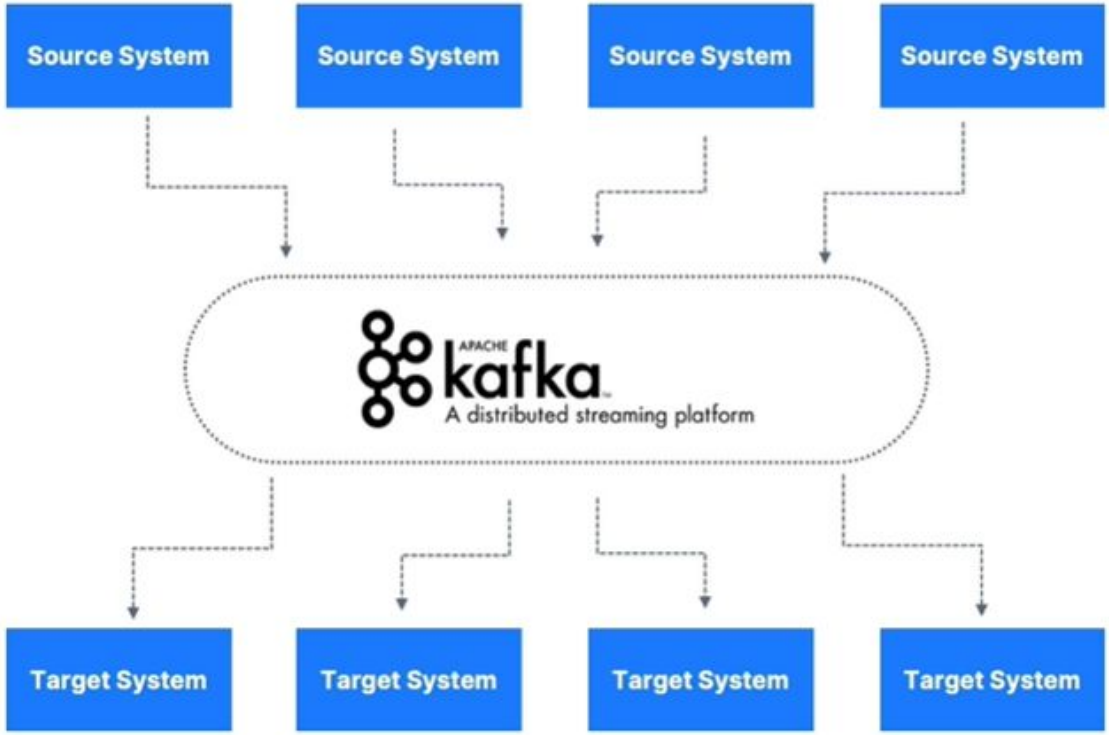
# Types of problems organisations are facing with the previous architecture

- If you have 4 source systems, and 6 target systems, you need to write 24 integrations!

- Each integration comes with difficulties around
  - Protocol – how the data is transported *(TCP, HTTP, REST, FTP, JDBC…)*
  - Data format – how the data is parsed *(Binary, CSV, JSON, Avro, Protobuf…)*
  - Data schema & evolution – how the data is shaped and may change

- Each source system will have an increased load from the connections

# Why Apache Kafka:
# Decoupling of data streams & systems



# Why Apache Kafka

- Created by LinkedIn, now Open-Source Project mainly maintained by Confluent, IBM, Cloudera
- Distributed, resilient architecture, fault tolerant
- Horizontal scalability:
  - Can scale to 100s of brokers
  - Can scale to millions of messages per second
- High performance (latency of less than 10ms) – real time
- Used by the 2000+ firms, **80% of the Fortune 100**





# Apache Kafka: Use cases

- Messaging System
- Activity Tracking
- Gather metrics from many different locations
- Application Logs gathering
- Stream processing (with the Kafka Streams API for example)
- De-coupling of system dependencies
- Integration with Spark, Flink, Storm, Hadoop, and many other Big Data technologies
- Micro-services pub/sub

# For example...

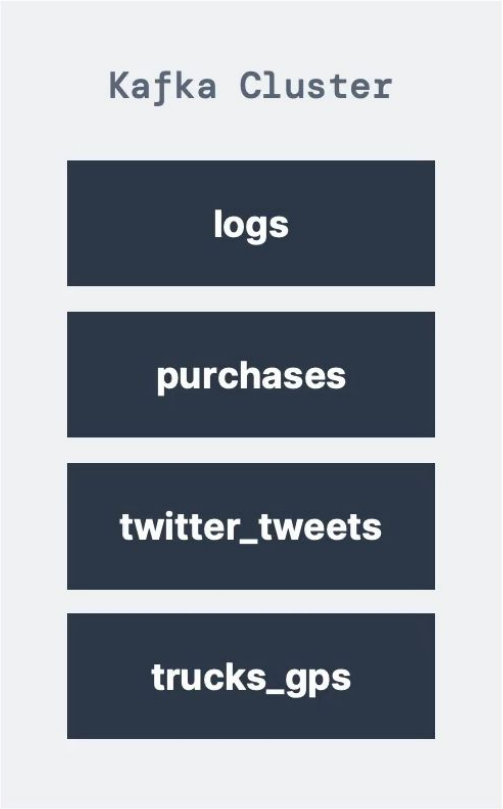
- **Netflix** uses Kafka to apply recommendations in real-time while you're watching TV shows
- **Uber** uses Kafka to gather user, taxi and trip data in real-time to compute and forecast demand, and compute surge pricing in real-time
- **LinkedIn** uses Kafka to prevent spam, collect user interactions to make better connection recommendations in real time.

- Remember that Kafka is only used as a transportation mechanism!

# Kafka Topics

- Topics: a particular stream of data

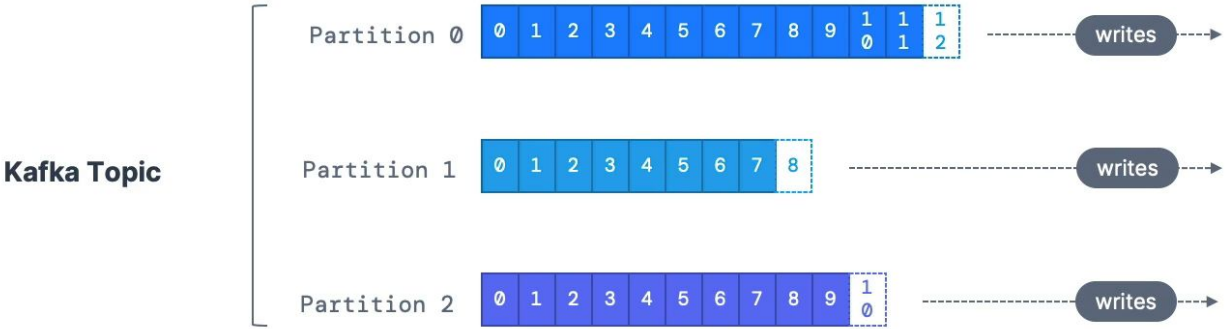
- Like a table in a database (without all the constraints)
- You can have as many topics as you want
- A topic is identified by its name
- Any kind of message format
- The sequence of messages is called a data stream
- You cannot query topics, instead, use Kafka Producers to send data and Kafka Consumers to read the data



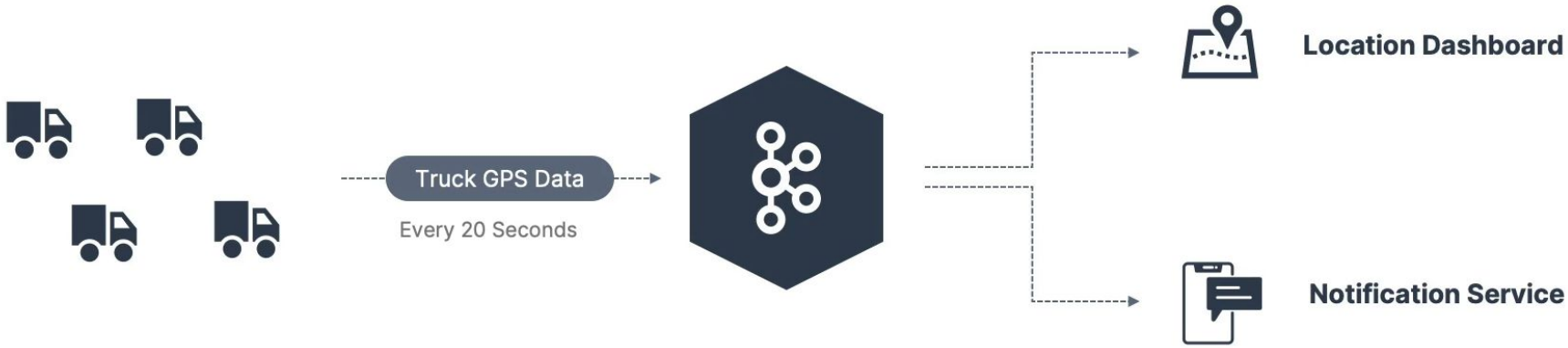
# Partitions and offsets

- Topics are split in partitions (example: 100 partitions)
  - Messages within each partition are ordered
  - Each message within a partition gets an incremental id, called offset

- Kafka topics are **immutable**: once data is written to a partition, it cannot be changed

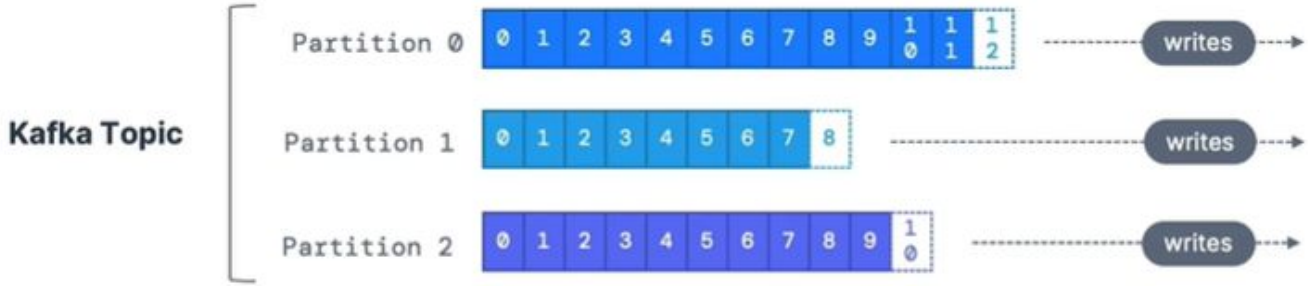


# Topic example: `truck_gps`




- Say you have a fleet of trucks; each truck reports its GPS position to Kafka.
- Each truck will send a message to Kafka every 20 seconds, each message will contain the truck ID and the truck position (latitude and longitude)
- You can have a topic `trucks_gps` that contains the position of all trucks.
- We choose to create that topic with 10 partitions (arbitrary number)

# Topics, partitions and offsets – important notes




- Once the data is written to a partition, it cannot be changed (immutability)
- Data is kept only for a limited time (default is one week - configurable)
- Offset only have a meaning for a specific partition.
  - E.g. offset 3 in partition 0 doesn't represent the same data as offset 3 in partition 1
  - Offsets are not re-used even if previous messages have been deleted
- Order is guaranteed only within a partition (not across partitions)
- Data is assigned randomly to a partition unless a key is provided (more on this later)
- You can have as many partitions per topic as you want

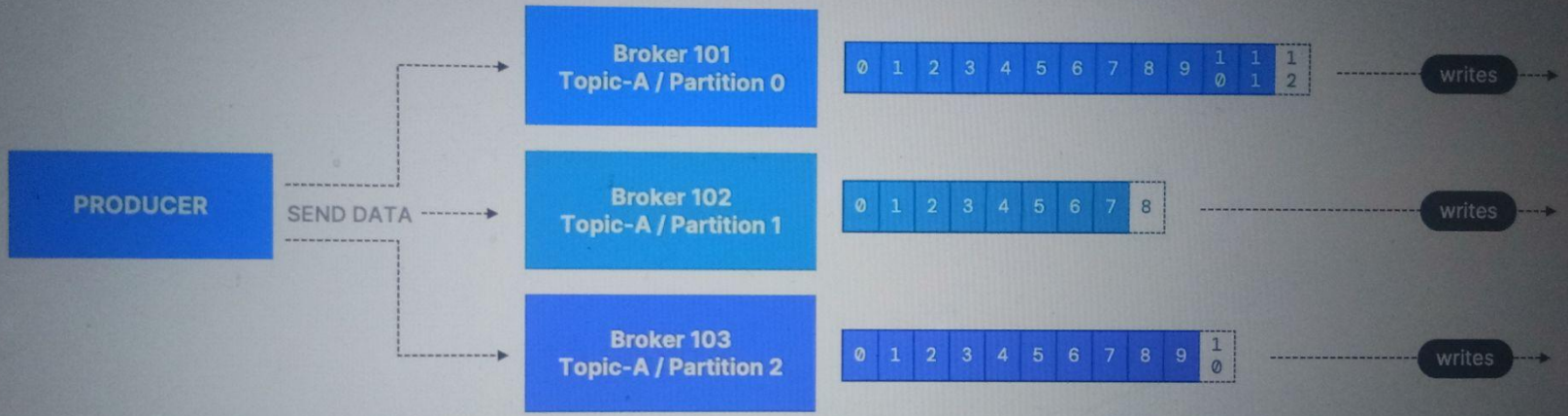
# Producers

- Producers write data to topics (which are made of partitions)
- Producers know to which partition to write to (and which Kafka broker has it)
- In case of Kafka broker failures, Producers will automatically recover

# Producers: Message keys

- Producers can choose to send a **key** with the message (string, number, binary, etc..)
- If `key=null`, data is sent round robin (partition 0, then 1, then 2...)
- If `key!=null`, then all messages for that key will always go to the same partition (hashing)
- A key are typically sent if you need message ordering for a specific field (ex: `truck_id`)

# Producer Acknowledgements (acks)




- Producers can choose to receive acknowledgment of data writes:
  - `acks=0`: Producer won't wait for acknowledgment (possible data loss)
  - `acks=1`: Producer will wait for leader acknowledgment (limited data loss)
  - `acks=all`: Leader + replicas acknowledgment (no data loss)

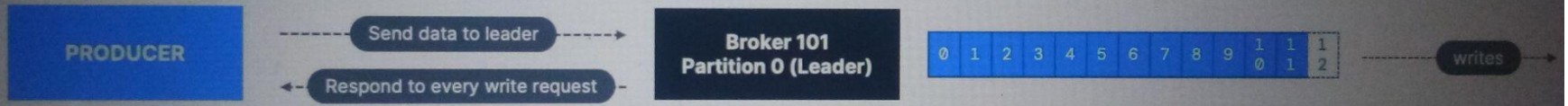
# Producer: acks=0

- When acks=0 producers consider messages as "written successfully" the moment the message was sent without waiting for the broker to accept it at all.


- If the broker goes offline or an exception happens, we won't know and **will lose data**
- Useful for data where it's okay to potentially lose messages, such as metrics collection
- Produces the highest throughput setting because the network overhead is minimized

# Producer acks=1

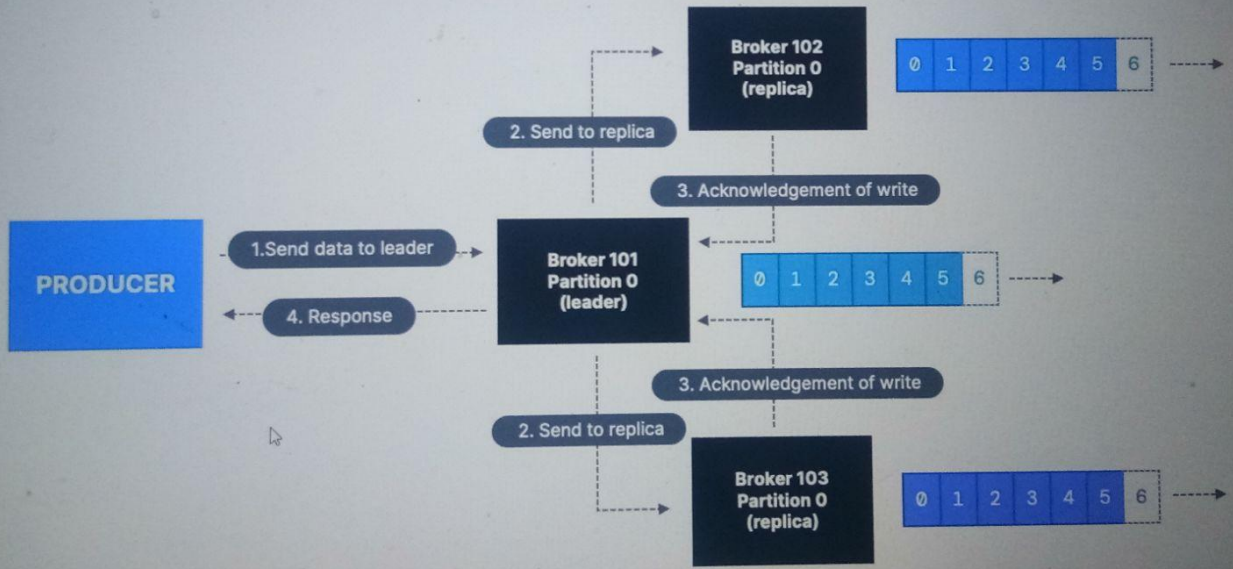
- When `acks=1`, producers consider messages as "written successfully" when the message was acknowledged by only the leader
- Default for Kafka v1.0 to v2.8




- Leader response is requested, but replication is not a guarantee as it happens in the background
- If the leader broker goes offline unexpectedly but replicas haven't replicated the data yet, we have a data loss
- If an ack is not received, the producer may retry the request

# Producer acks=all (acks=-1)

- When `acks=all`, producers consider messages as "written successfully" when the message is accepted by all in-sync replicas (ISR).
- Default for Kafka 3.0+



# Kafka Topic Availability

- **Availability: (considering RF=3)**
  - `acks=0` & `acks=1`: if one partition is up and considered an ISR, the topic will be available for writes.
  - `acks=all`:
    - `min.insync.replicas=1` (default): the topic must have at least 1 partition up as an ISR (that includes the leader) and so we can tolerate two brokers being down
    - `min.insync.replicas=2`: the topic must have at least 2 ISR up, and therefore we can tolerate at most one broker being down (in the case of replication factor of 3), and we have the guarantee that for every write, the data will be at least written twice
    - `min.insync.replicas=3`: this wouldn't make much sense for a corresponding replication factor of 3 and we couldn't tolerate any broker going down.
    - in summary, when `acks=all` with a `replication.factor=N` and `min.insync.replicas=M` we can tolerate N-M brokers going down for topic availability purposes

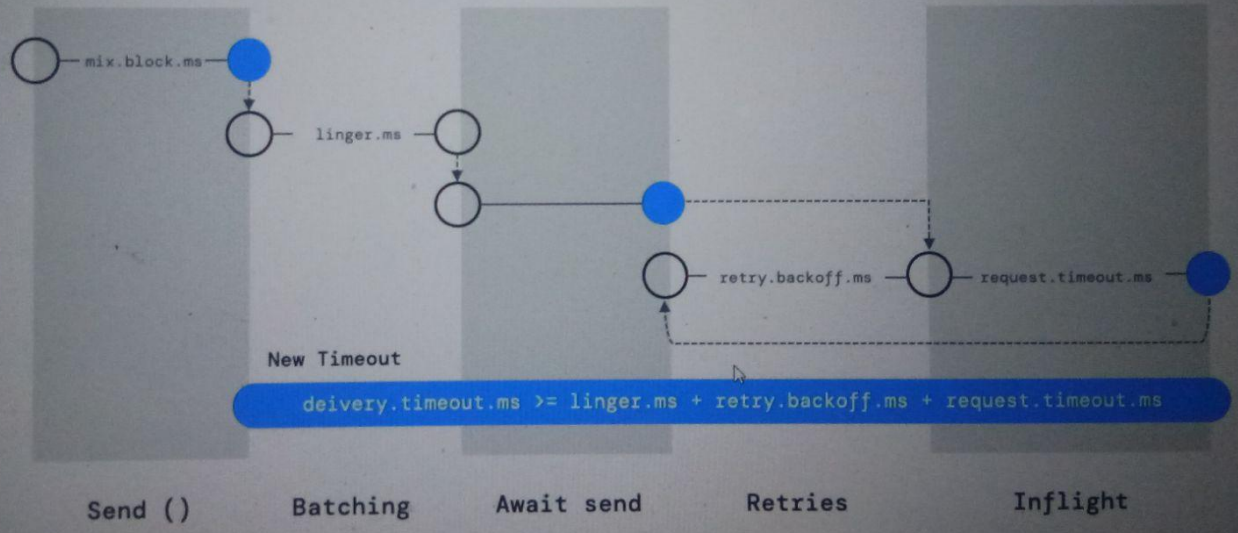
- `acks=all` and `min.insync.replicas=2` is the most popular option for data durability and availability and allows you to withstand at most the loss of **one** Kafka broker

# Producer Retries

- In case of transient failures, developers are expected to handle exceptions, otherwise the data will be lost.
- Example of transient failure:
  - NOT_ENOUGH_REPLICAS (due to min.insync.replicas setting)
- There is a "retries" setting
  - defaults to 0 for Kafka <= 2.0
  - defaults to 2147483647 for Kafka >= 2.1
- The retry.backoff.ms setting is by default 100 ms

# Producer Timeouts

- If retries>0, for example `retries=2147483647`, retries are bounded by a timeout
- Since Kafka 2.1, you can set: `delivery.timeout.ms=120000==2 min`
- Records will be failed if they can't be acknowledged within `delivery.timeout.ms`

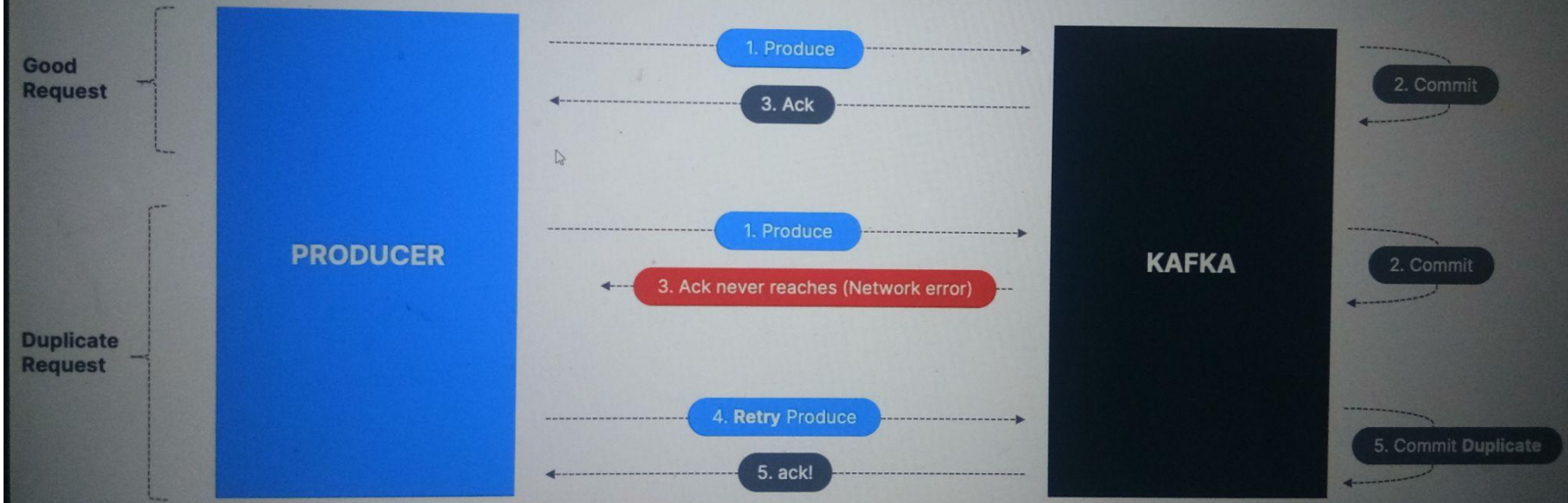


# Producer Retries: Warning for old version of Kafka

- If you are not using an idempotent producer (not recommended – old Kafka):
  - In case of retries, there is a chance that messages will be sent out of order (if a batch has failed to be sent).
  - If you rely on key-based ordering, that can be an issue.
- For this, you can set the setting while controls how many produce requests can be made in parallel: max.in.flight.requests.per.connection
  - Default: 5
  - Set it to 1 if you need to ensure ordering (may impact throughput)
- In Kafka >= 1.0.0, there's a better solution with idempotent producers!

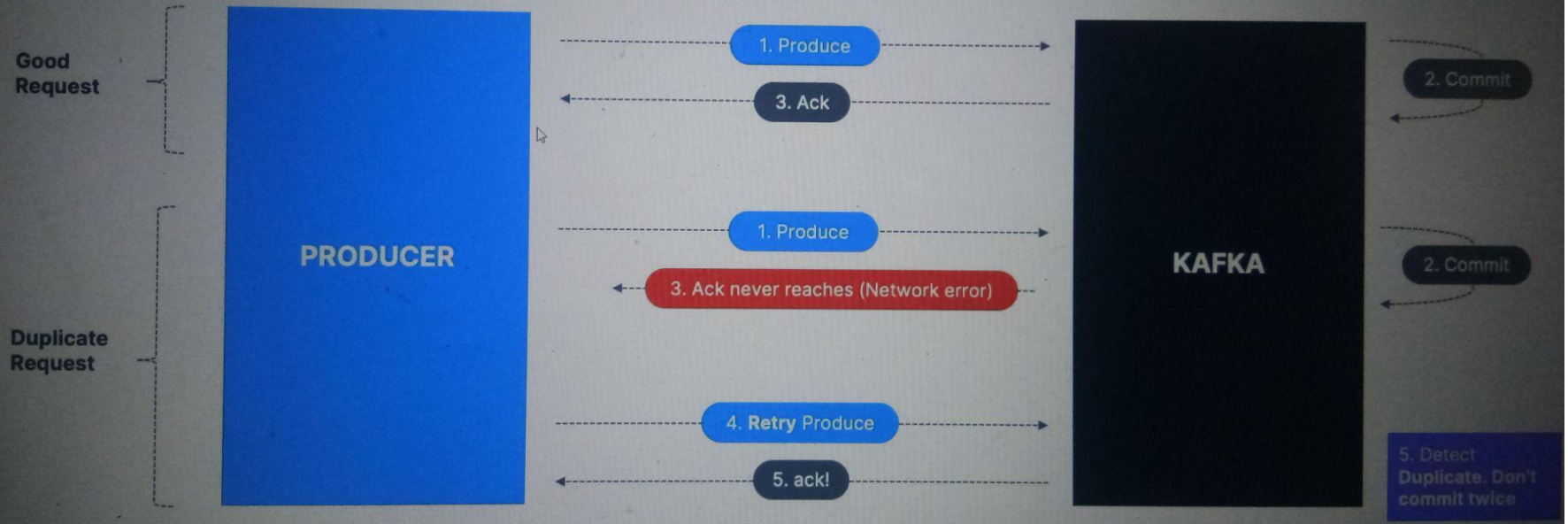
# Idempotent Producer

- The Producer can introduce duplicate messages in Kafka due to network errors



# Idempotent Producer

- In Kafka >= 0.11, you can define a “idempotent producer” which won’t introduce duplicates on network error



# Idempotent Producer

- Idempotent producers are great to guarantee a stable and safe pipeline!
- **They are the default since Kafka 3.0, recommended to use them**

- They come with:
  - `retries=Integer.MAX_VALUE` (2^31-1 = 2147483647)
  - `max.in.flight.requests=1` (Kafka == 0.11) or
  - `max.in.flight.requests=5` (Kafka >= 1.0 – higher performance & keep ordering – KAFKA-5494)
  - `acks=all`

- These settings are applied automatically after your producer has started if not manually set
- Just set:

```
producerProps.put("enable.idempotence", true);
```

# Kafka Producer defaults

- Since Kafka 3.0, the producer is "safe" by default:
  - `acks=all (-1)`
  - `enable.idempotence=true`
- With Kafka 2.8 and lower, the producer by default comes with:
  - `acks=1`
  - `enable.idempotence=false`
- I would recommend using a safe producer whenever possible!
- Super important: **always use upgraded Kafka Clients**

# Safe Kafka Producer - Summary & Demo

- Since Kafka 3.0, the producer is "safe" by default, otherwise, upgrade your clients or set the following settings

- `acks=all`
  - Ensures data is properly replicated before an ack is received
- `min.insync.replicas=2` (broker/topic level)
  - Ensures two brokers in ISR at least have the data after an ack
- `enable.idempotence=true`
  - Duplicates are not introduced due to network retries
- `retries=MAX_INT` (producer level)
  - Retry until `delivery.timeout.ms` is reached
- `delivery.timeout.ms=120000`
  - Fail after retrying for 2 minutes
- `max.in.flight.requests.per.connection=5`
  - Ensure maximum performance while keeping message ordering

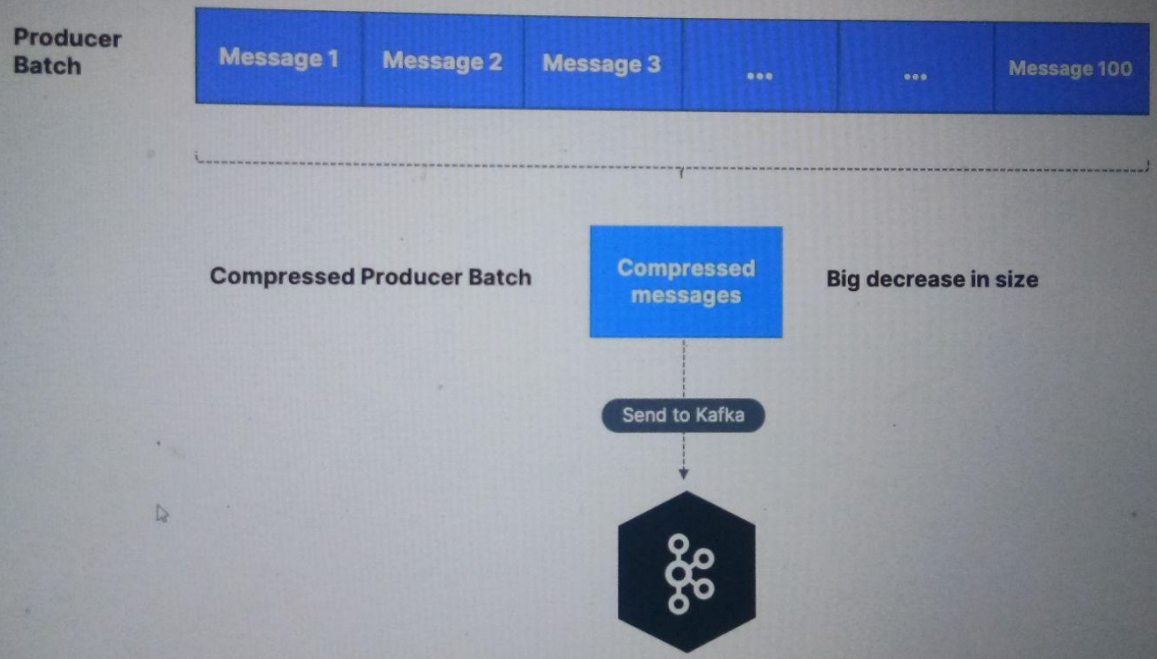
# Message Compression at the Producer level

- Producer usually send data that is text-based, for example with JSON data
- In this case, it is important to apply compression to the producer.

- Compression can be enabled at the Producer level and doesn't require any configuration change in the Brokers or in the Consumers
- `compression.type` can be `none` (default), `gzip`, `lz4`, `snappy`, `zstd` (Kafka 2.1)

- Compression is more effective the bigger the batch of message being sent to Kafka!
- Benchmarks here: https://blog.cloudflare.com/squeezing-the-firehose/

# Message Compression at the Producer level



# Message Compression

- The compressed batch has the following advantage:
  - Much smaller producer request size (compression ratio up to 4x!)
  - Faster to transfer data over the network => less latency
  - Better throughput
  - Better disk utilisation in Kafka (stored messages on disk are smaller)
- Disadvantages (very minor):
  - Producers must commit some CPU cycles to compression
  - Consumers must commit some CPU cycles to decompression
- Overall:
  - Consider testing `snappy` or `lz4` for optimal speed / compression ratio (test others too)
  - Consider tweaking `linger.ms` and `batch.size` to have bigger batches, and therefore more compression and higher throughput
  - Use compression in production

# Message Compression at the Broker / Topic level

- There is also a setting you can set at the broker level (all topics) or topic-level

- `compression.type=producer` (default), the broker takes the compressed batch from the producer client and writes it directly to the topic's log file without recompressing the data
- `compression.type=none`: all batches are decompressed by the broker
- `compression.type=lz4`: (for example)
  - If it's matching the producer setting, data is stored on disk as is
  - If it's a different compression setting, batches are decompressed by the broker and then re-compressed using the compression algorithm specified

- Warning: if you enable broker-side compression, it will consume extra CPU cycles

# Producer Default Partitioner when `key!=null`


- **Key Hashing** is the process of determining the mapping of a key to a partition
- In the default Kafka partitioner, the keys are hashed using the **murmur2 algorithm**

```
targetPartition = Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1)
```

- This means that same key will go to the same partition (we already know this), and adding partitions to a topic will completely alter the formula
- It is most likely preferred to not override the behavior of the partitioner, but it is possible to do so using partitioner.class

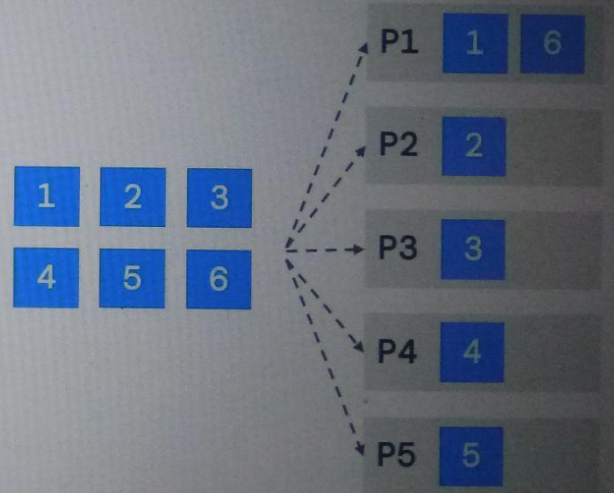
# Producer Default Partitioner when `key=null`


- When `key=null`, the producer has a default partitioner that varies:
  - Round Robin: for Kafka 2.3 and below
  - Sticky Partitioner: for Kafka 2.4 and above

- Sticky Partitioner improves the performance of the producer especially when high throughput when the key is null

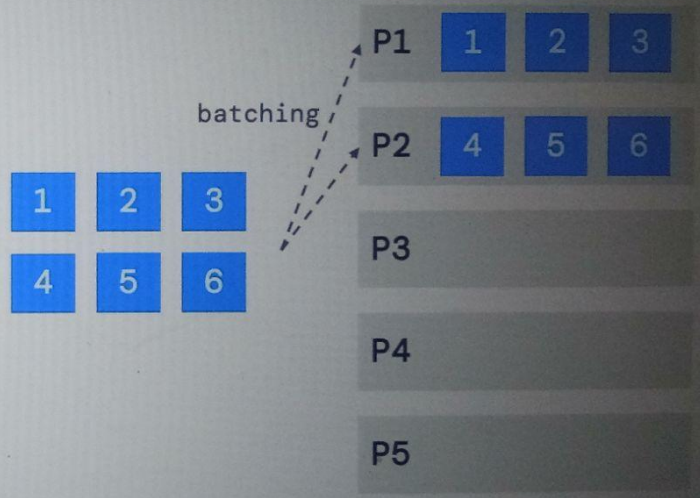
# Producer Default Partitioner Kafka ≤ v2.3
# Round Robin Partitioner

- With Kafka ≤ v2.3 , when there's no partition and no key specified, the default partitioner sends data in a round-robin fashion
- This results in more batches (one batch per partition) and smaller batches (imagine with 100 partitions)
- Smaller batches lead to more requests as well as higher latency



# Producer Default Partitioner Kafka ≥ 2.4
# Sticky Partitioner

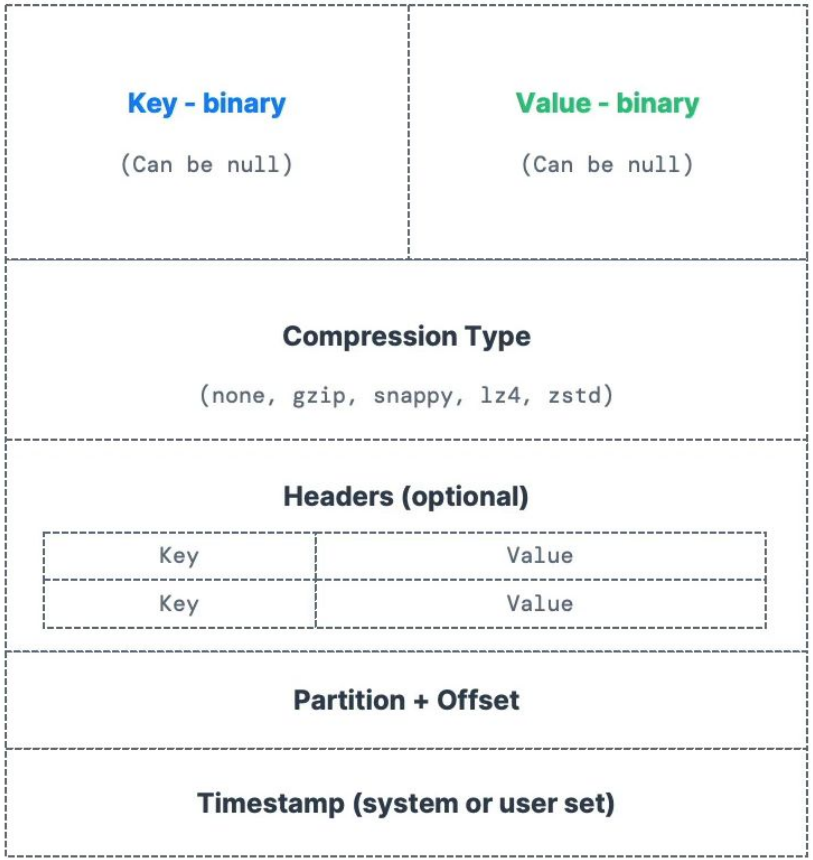
- It would be better to have all the records sent to a single partition and not multiple partitions to improve batching
- The producer sticky partitioner:
  - We "stick" to a partition until the batch is full or `linger.ms` has elapsed
  - After sending the batch, the partition that is sticky changes
- Larger batches and reduced latency (because larger requests, and `batch.size` more likely to be reached)
- Over time, records are still spread evenly across partitions




Sticky Partitioner
(performance improvement)

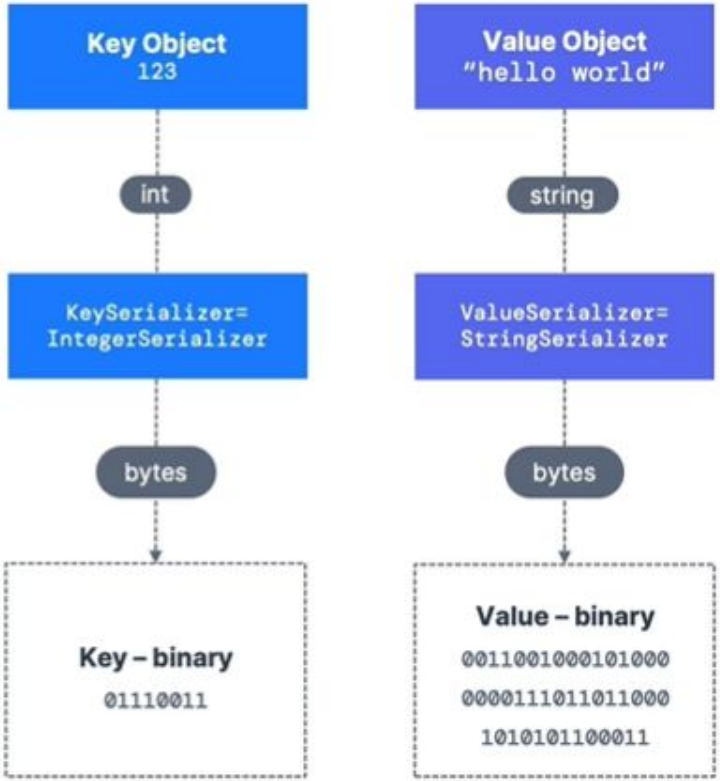
# Kafka Messages anatomy...

Kafka Message Created by the producer




# Kafka Message Serializer

- Kafka only accepts bytes as an input from producers and sends bytes out as an output to consumers
- Message Serialization means transforming objects / data into bytes
- They are used on the value and the key
- Common Serializers
  - String (incl. JSON)
  - Int, Float
  - Avro
  - Protobuf



# For the curious: Kafka Message Key Hashing

- A Kafka partitioner is a code logic that takes a record and determines to which partition to send it into.

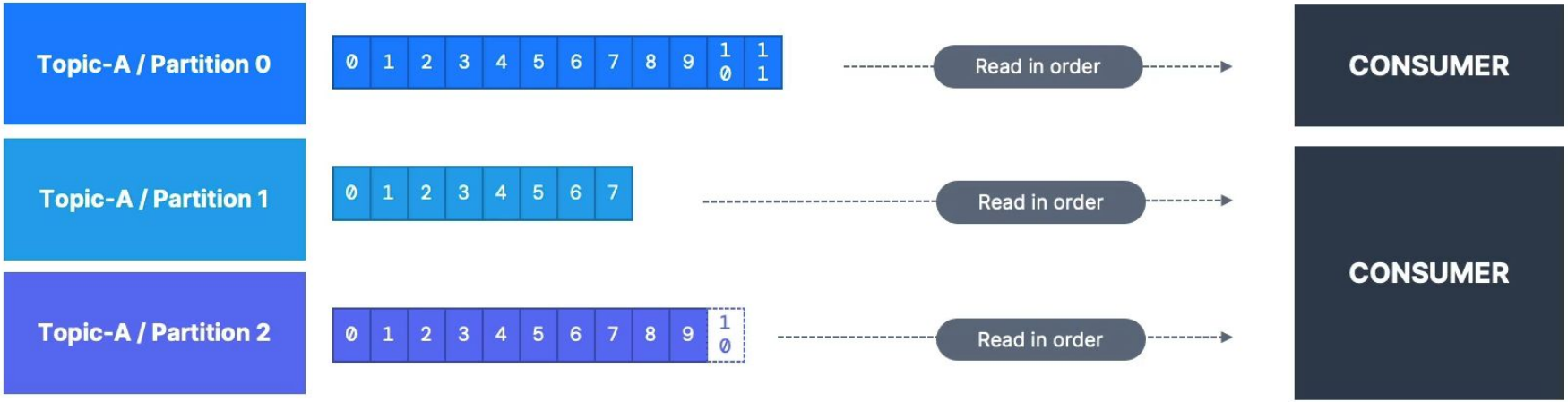

- **Key Hashing** is the process of determining the mapping of a key to a partition
- In the default Kafka partitioner, the keys are hashed using the **murmur2 algorithm,** with the formula below for the curious:

```
targetPartition = Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1)
```

Only the producer determines to which partition of the Kafka Topic this msg should go

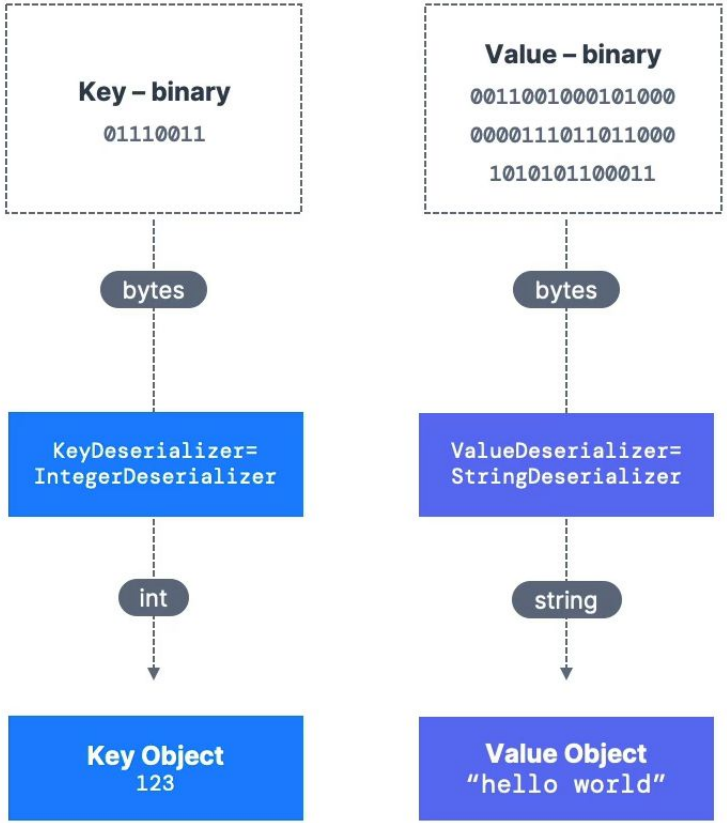
# Consumers

- Consumers read data from a topic (identified by name) – pull model
- Consumers automatically know which broker to read from
- In case of broker failures, consumers know how to recover
- Data is read in order from low to high offset **within each partitions**



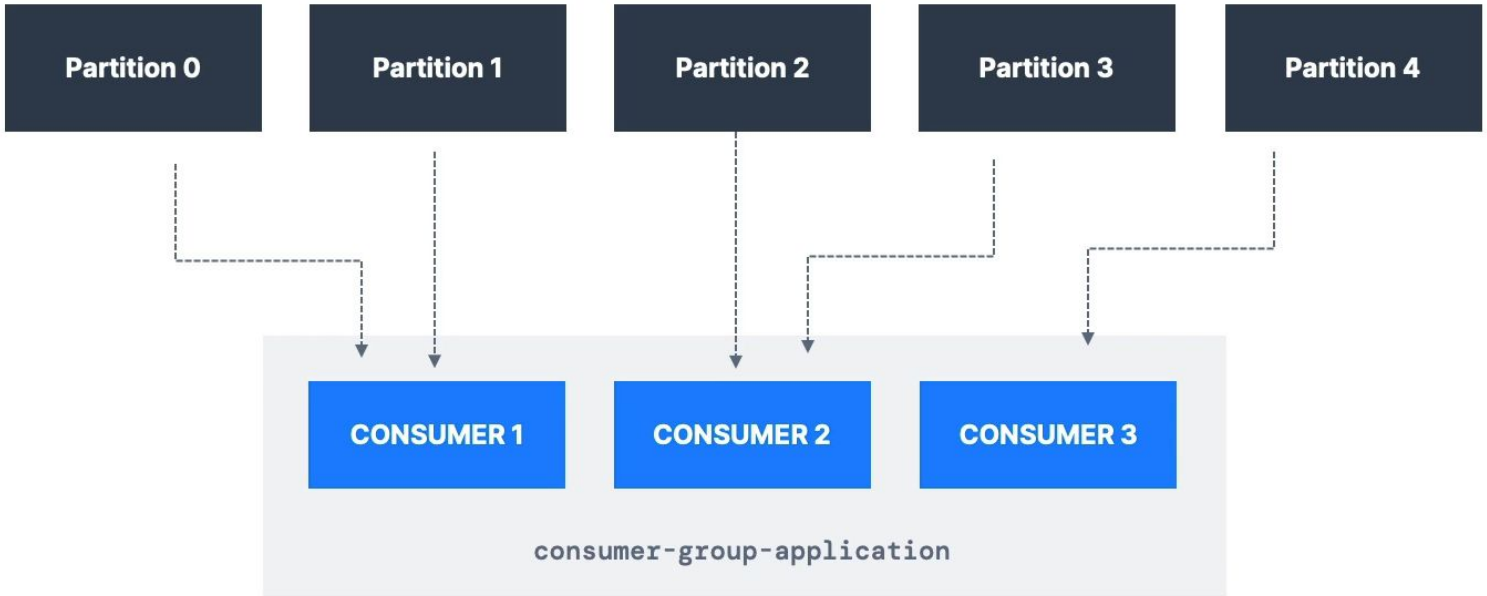
# Consumer Deserializer

- Deserialize indicates how to transform bytes into objects / data
- They are used on the value and the key of the message
- Common Deserializers
  - String (incl. JSON)
  - Int, Float
  - Avro
  - Protobuf
- The serialization / deserialization type must not change during a topic lifecycle (create a new topic instead)



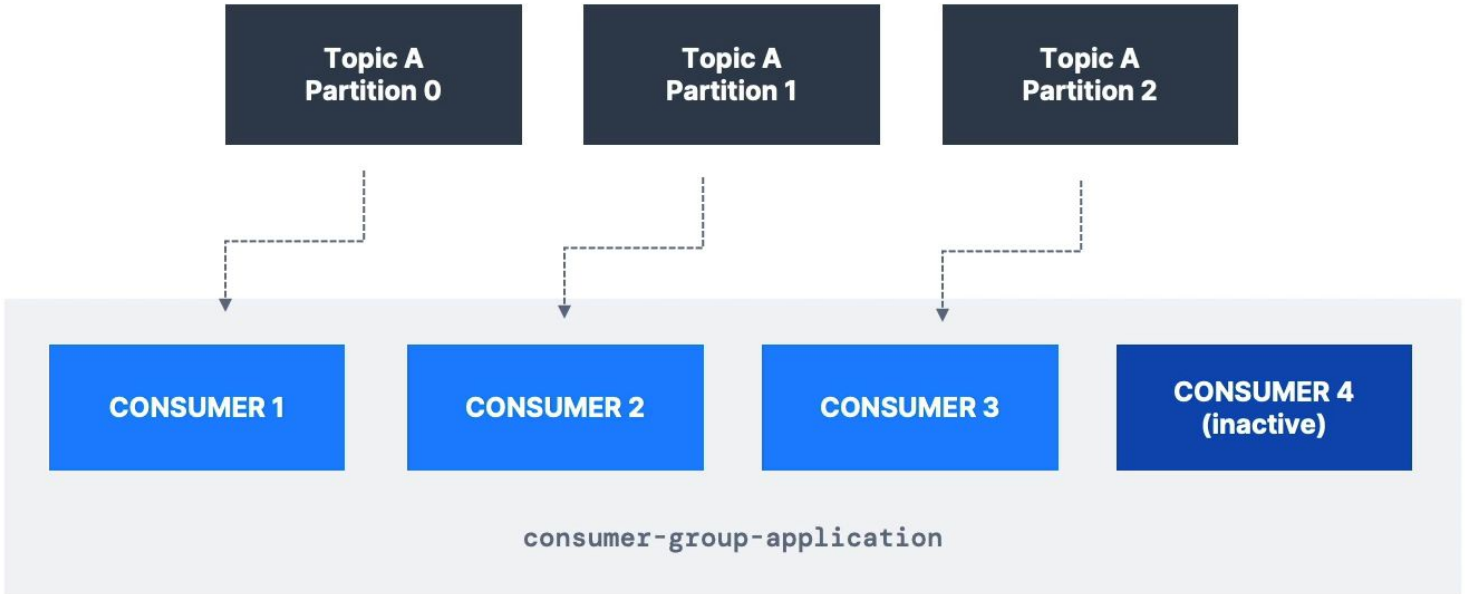
# Consumer Groups

- All the consumers in an application read data as a consumer groups
- Each consumer within a group reads from exclusive partitions



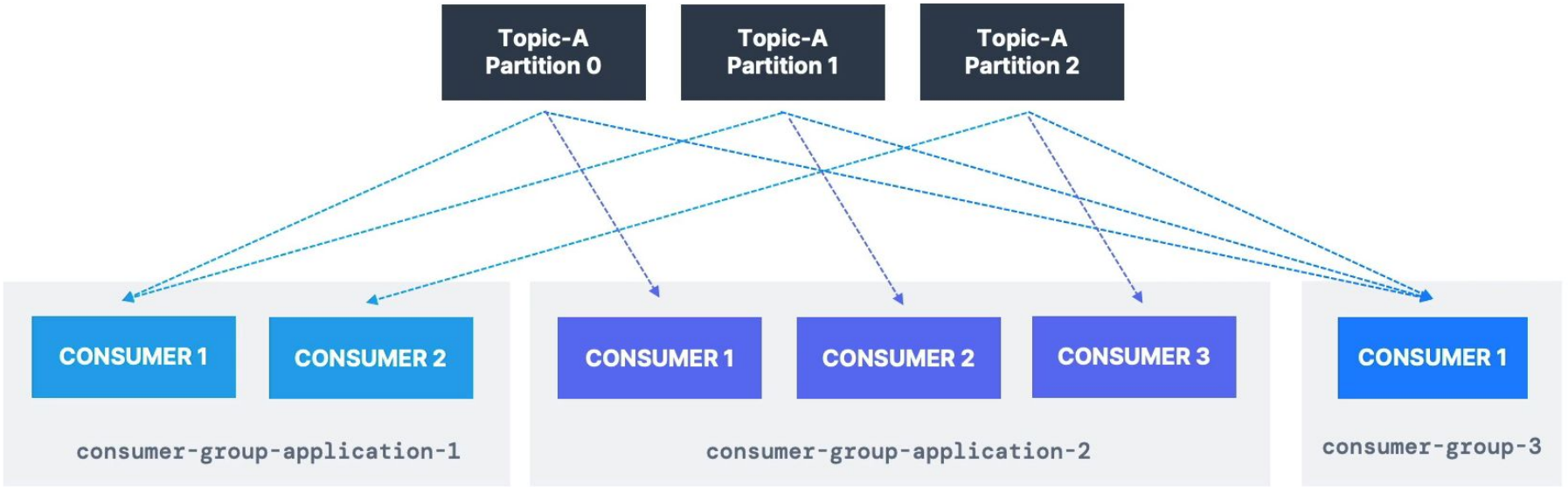
# Consumer Groups - What if too many consumers?

- If you have more consumers than partitions, some consumers will be inactive



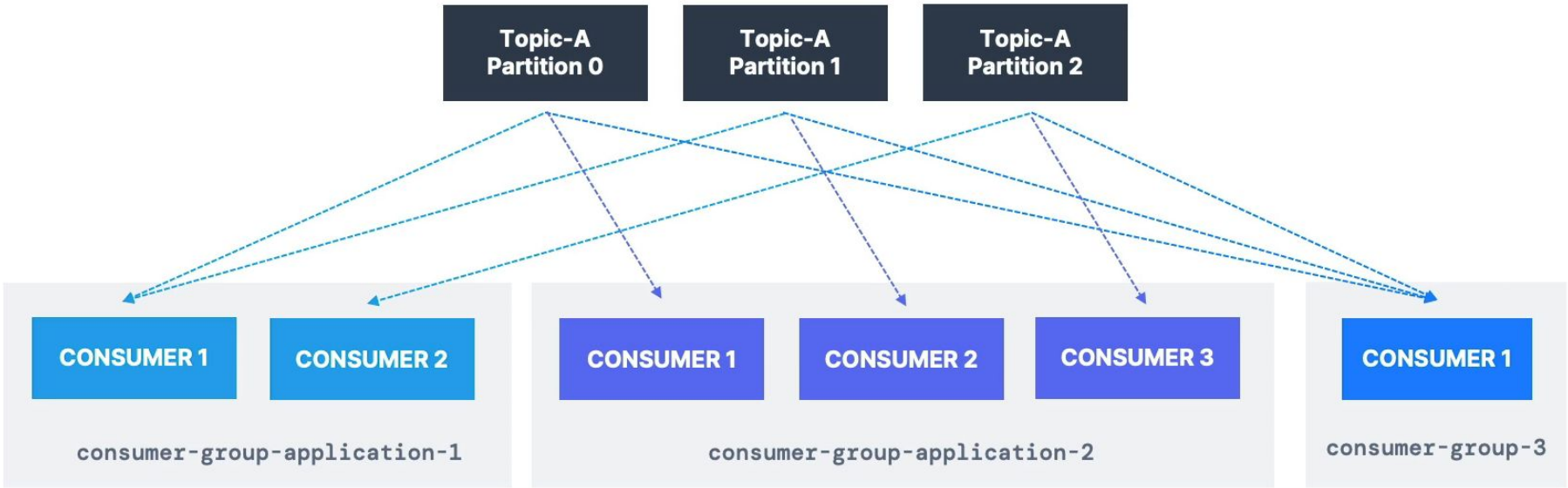
# Multiple Consumers on one topic

- In Apache Kafka it is acceptable to have multiple consumer groups on the same topic



# Multiple Consumers on one topic

- In Apache Kafka it is acceptable to have multiple consumer groups on the same topic
- To create distinct consumer groups, use the consumer property **`group.id`**



# Consumer Offsets

- Kafka stores the offsets at which a consumer group has been reading
- The offsets committed are in Kafka topic named `__consumer_offsets`
- When a consumer in a group has processed data received from Kafka, it should be **periodically** committing the offsets (the Kafka broker will write to `__consumer_offsets`, not the group itself)
- If a consumer dies, it will be able to read back from where it left off thanks to the committed consumer offsets!

# Delivery semantics for consumers

- By default, Java Consumers will automatically commit offsets (at least once)
- There are 3 delivery semantics if you choose to commit manually
- At least once (usually preferred)
  - Offsets are committed after the message is processed
  - If the processing goes wrong, the message will be read again
  - This can result in duplicate processing of messages. Make sure your processing is idempotent (i.e. processing again the messages won't impact your systems)
- At most once
  - Offsets are committed as soon as messages are received
  - If the processing goes wrong, some messages will be lost (they won't be read again)
- Exactly once
  - For Kafka => Kafka workflows: use the Transactional API (easy with Kafka Streams API)
  - For Kafka => External System workflows: use an idempotent consumer

# Consumer group and consumer offset

In the CLI, we can run the Producer and we can have our consumers in a group to read the message for that topic. Say for ex if we have  1 topic with 3 partition and 1 consumer group. In that group if we have 3 consumers each consumer of the group is then each consumer will read data from exclusive partition. If consumer1 in that group reads msg from partition 1 then consumer2 from partition 2 and so on.

If one msg is read by one consumer of the group then other consumers of that group won't receive the msg. Since they are group.

If one consumer group has 2 consumers and if we try to add the 3rd consumer and also we specified that we need to read msgs from beginning, it won't get the previous msgs since all the previous msgs are read by other consumers in that grp. The kafka will have the consumer offset and will send the message from where they left.

Since the consumer grp acts like single consumer with the redundant consumers (kindoff).

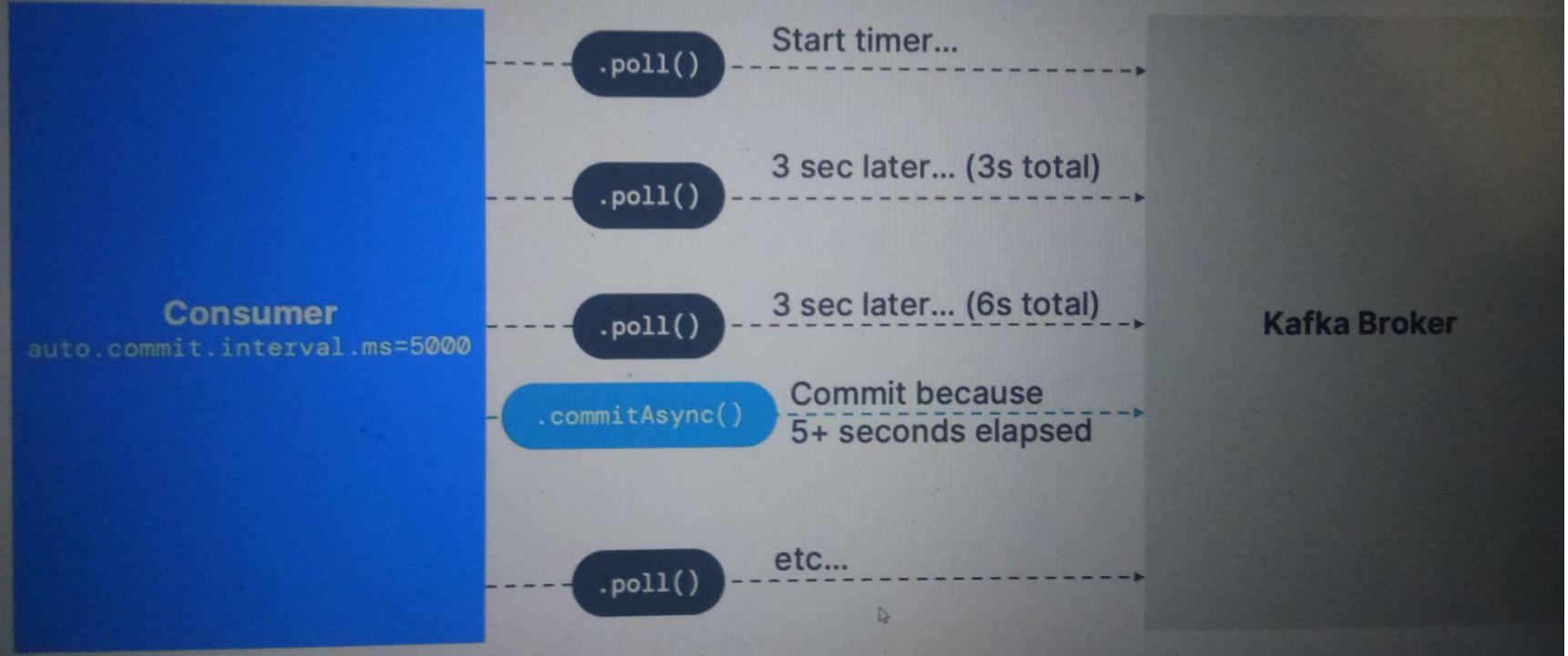
Note: Producers will send the msg to the available partitions based on the round-robin method(sending msgs one by one to all the partition and repeating the cycle)

# Consumer Offset Commit Strategies

- There are two most common patterns for committing offsets in a consumer application.

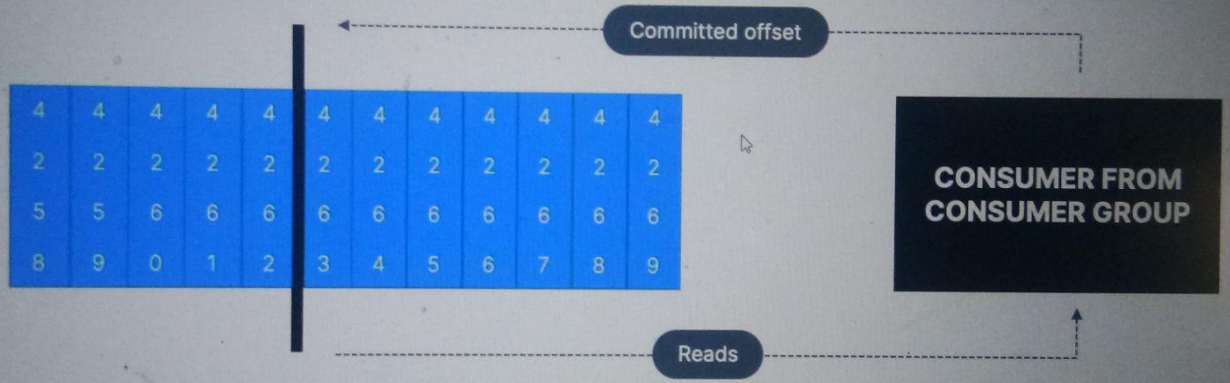
- **2 strategies:**
  - (easy) enable.auto.commit = true & synchronous processing of batches
  - (medium) enable.auto.commit = false & manual commit of offsets

# Kafka Consumer – Auto Offset Commit Behavior



# Consumer Offset Reset Behavior

- A consumer is expected to read from a log continuously.




- But if your application has a bug, your consumer can be down
- If Kafka has a retention of 7 days, and your consumer is down for more than 7 days, the offsets are “invalid”

# Consumer Offset Reset Behavior

- The behavior for the consumer is to then use:
  - `auto.offset.reset=latest`: will read from the end of the log
  - `auto.offset.reset=earliest`: will read from the start of the log
  - `auto.offset.reset=none`: will throw exception if no offset is found

- Additionally, consumer offsets can be lost:
  - If a consumer hasn't read new data in 1 day (Kafka < 2.0)
  - If a consumer hasn't read new data in 7 days (Kafka >= 2.0)

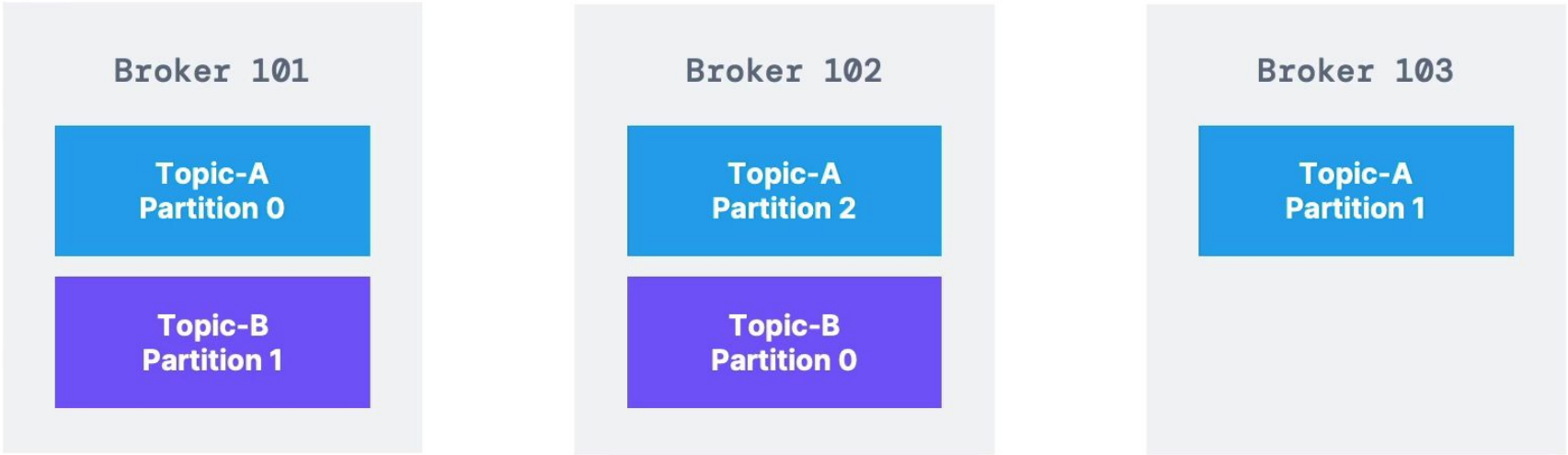
- This can be controlled by the broker setting `offset.retention.minutes`

# Kafka Brokers

- A Kafka cluster is composed of multiple brokers (servers)
- Each broker is identified with its ID (integer)
- Each broker contains certain topic partitions
- After connecting to any broker (called a bootstrap broker), you will be connected to the entire cluster (Kafka clients have smart mechanics for that)
- A good number to get started is 3 brokers, but some big clusters have over 100 brokers
- In these examples we choose to number brokers starting at 100 (arbitrary)

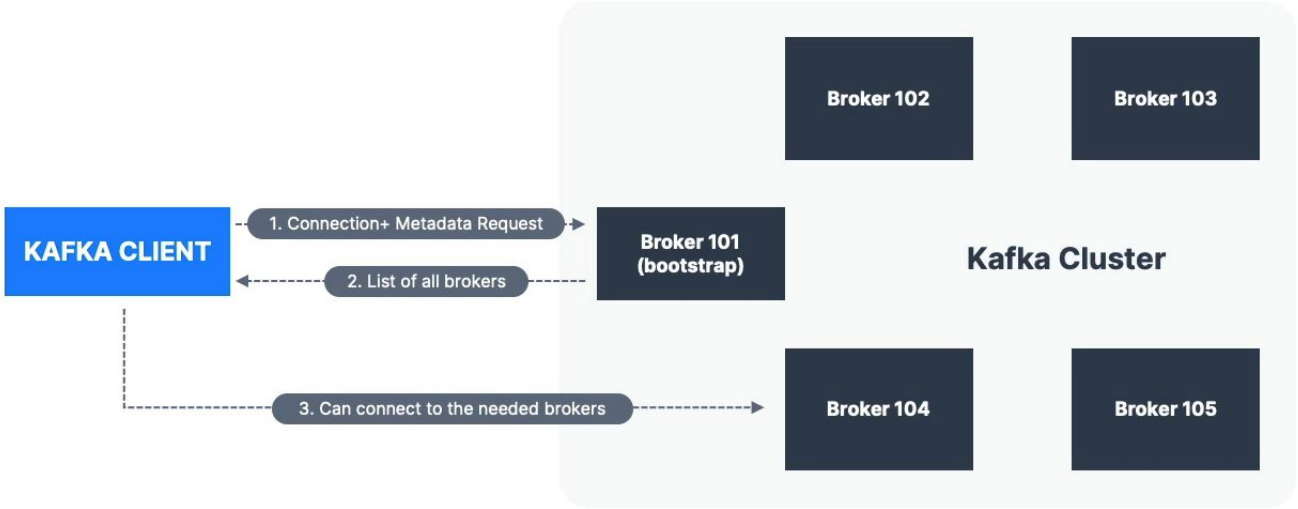
# Brokers and topics

- Example of **Topic-A** with **3 partitions** and **Topic-B** with **2 partitions**
- Note: data is distributed, and Broker 103 doesn't have any **Topic B** data



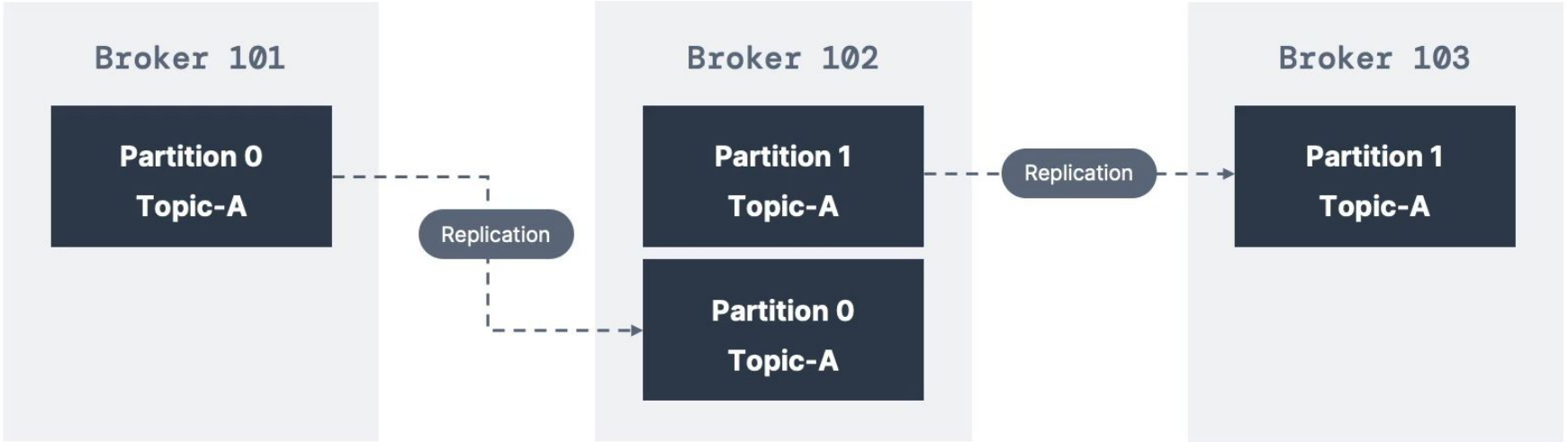
# Kafka Broker Discovery

- Every Kafka broker is also called a “bootstrap server”
- That means that **you only need to connect to one broker**, and the Kafka clients will know how to be connected to the entire cluster (smart clients)
- Each broker knows about all brokers, topics and partitions (metadata)



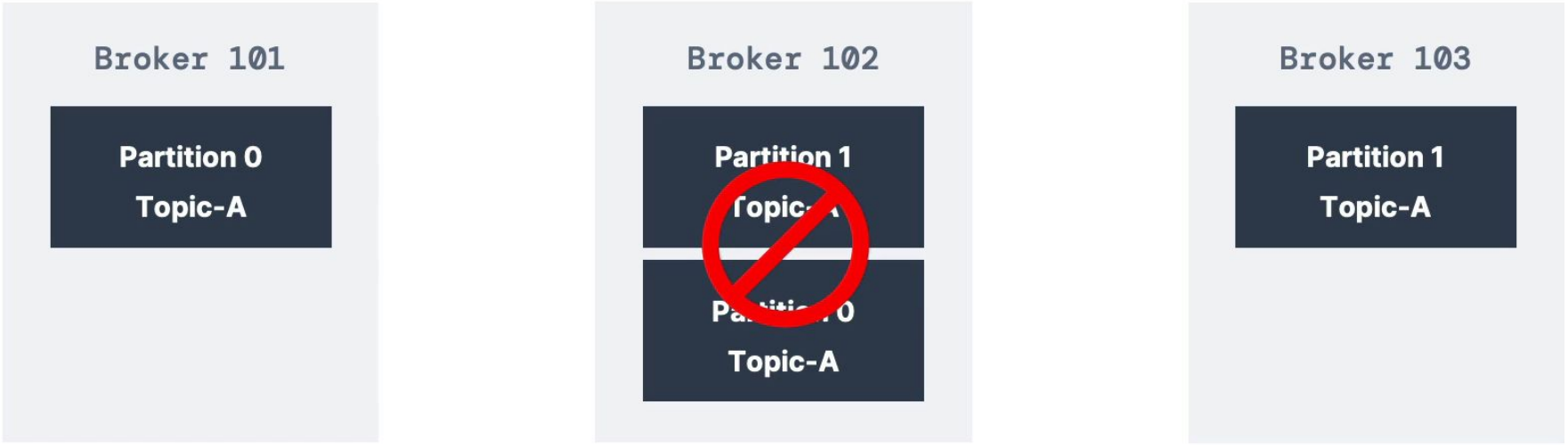
# Topic replication factor

- Topics should have a replication factor > 1 (usually between 2 and 3)
- This way if a broker is down, another broker can serve the data
- Example: Topic-A with 2 partitions and replication factor of 2



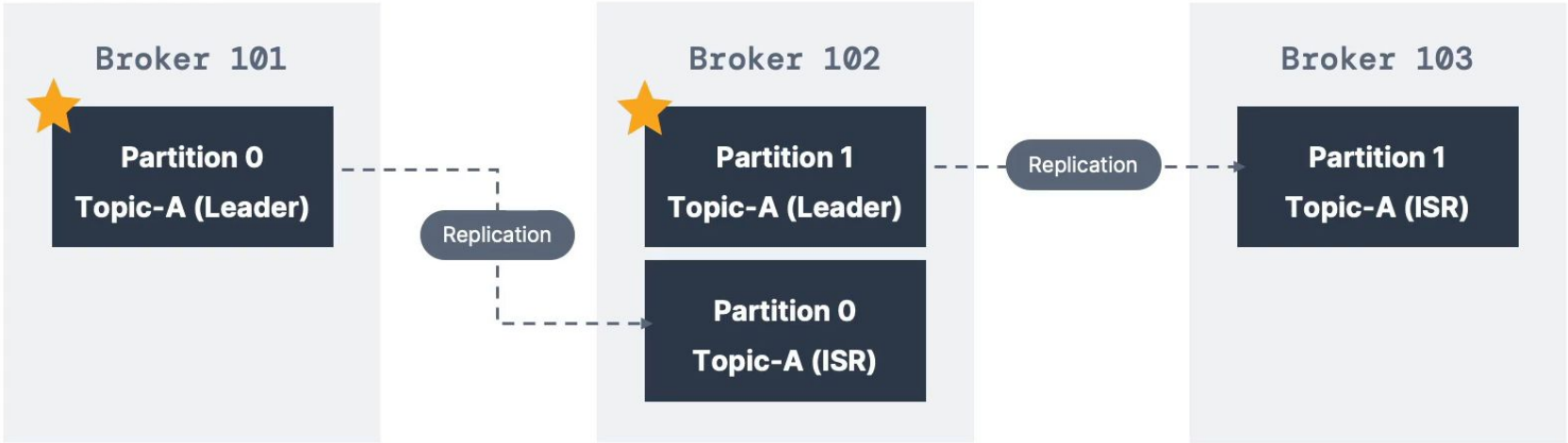
# Topic replication factor

- Example: we lose Broker 102
- Result: Broker 101 and 103 can still serve the data



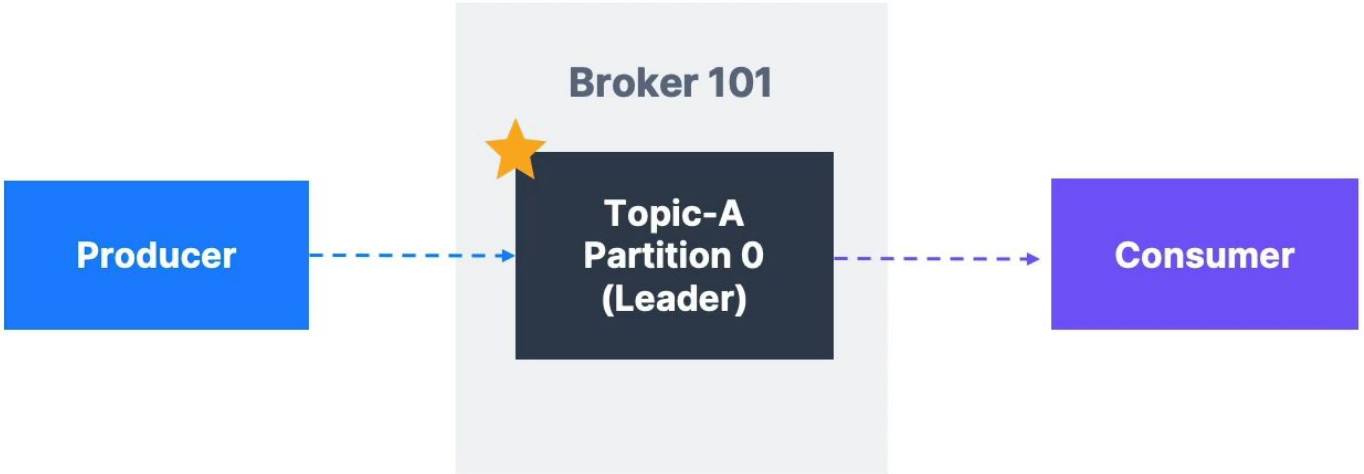
# Concept of Leader for a Partition

- **At any time only ONE broker can be a leader for a given partition**
- **Producers can only send data to the broker that is leader of a partition**
- The other brokers will replicate the data
- Therefore, each partition has one leader and multiple ISR (in-sync replica)



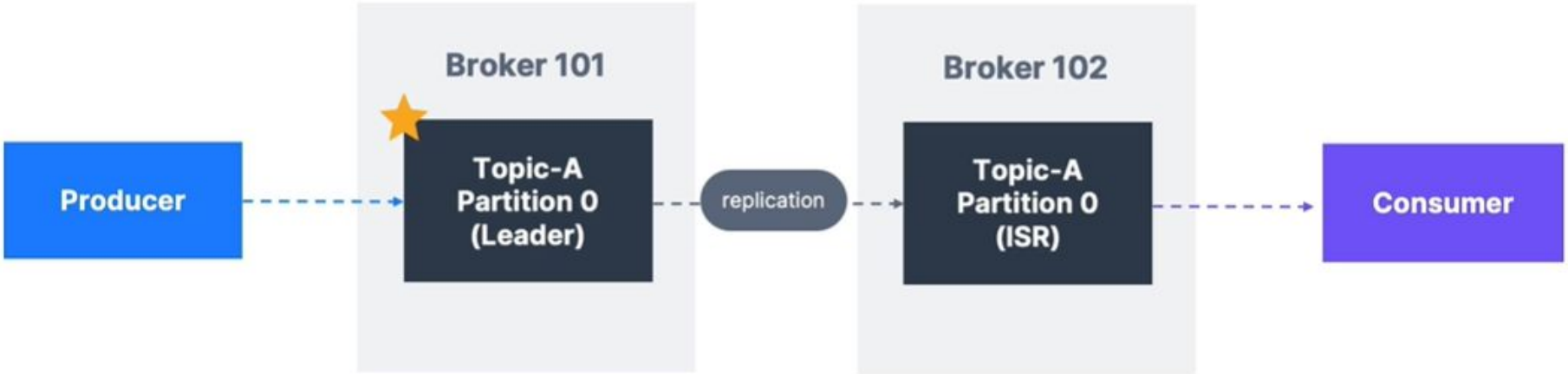
# Default producer & consumer behavior with leaders

- Kafka Producers can only write to the leader broker for a partition
- Kafka Consumers by default will read from the leader broker for a partition



# Kafka Consumers Replica Fetching (Kafka v2.4+)

- Since Kafka 2.4, it is possible to configure consumers to read from the closest replica
- This may help improve latency, and also decrease network costs if using the cloud

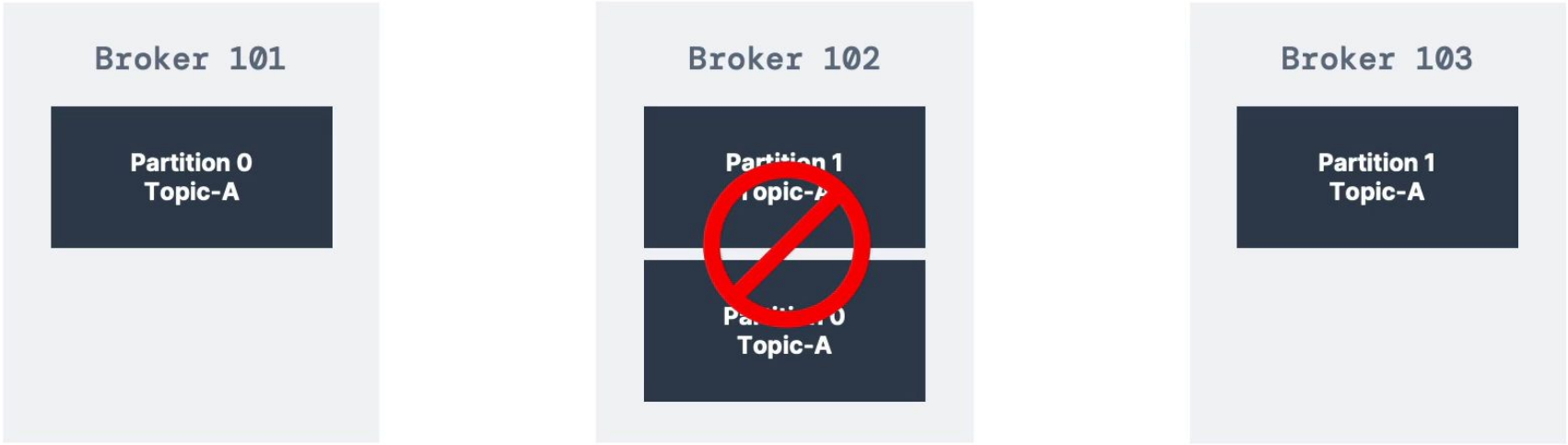


# Producer Acknowledgements (acks)


- Producers can choose to receive acknowledgment of data writes:
  - `acks=0`: Producer won't wait for acknowledgment (possible data loss)
  - `acks=1`: Producer will wait for leader acknowledgment (limited data loss)
  - `acks=all`: Leader + replicas acknowledgment (no data loss)

# Kafka Topic Durability

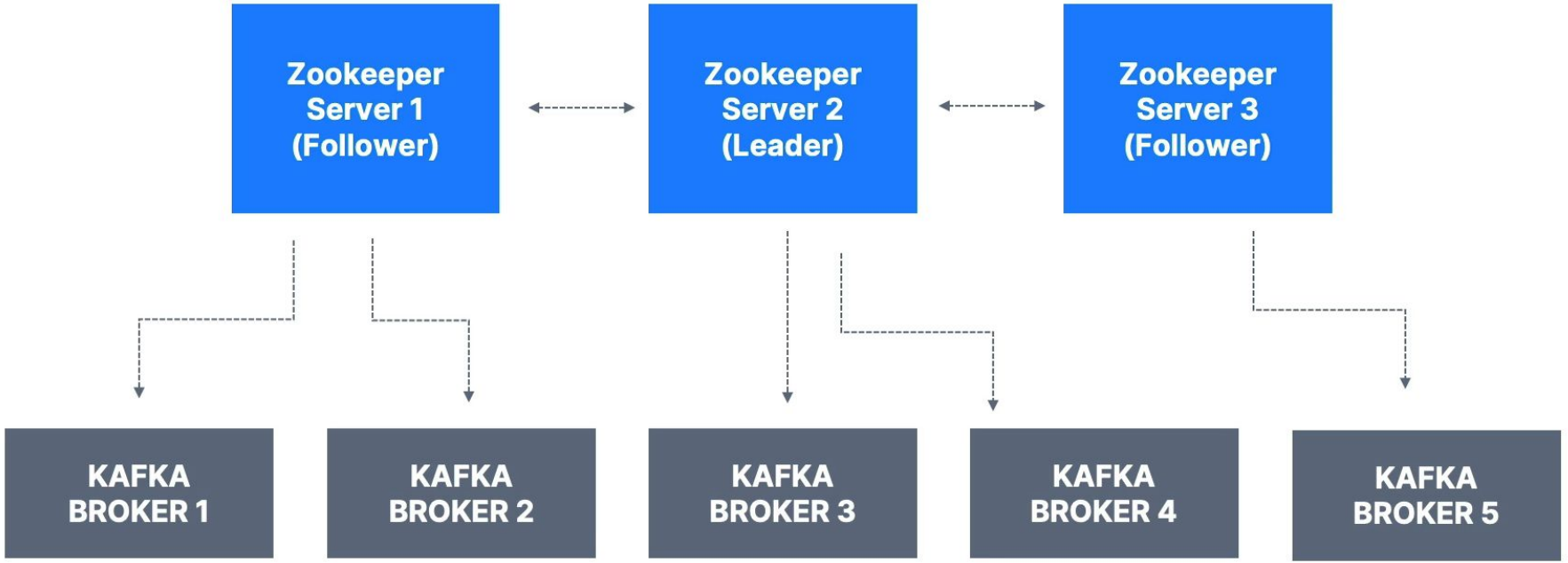
- For a topic replication factor of 3, topic data durability can withstand 2 brokers loss.
- As a rule, for a replication factor of N, you can permanently lose up to N-1 brokers and still recover your data.



# Zookeeper


- Zookeeper manages brokers (keeps a list of them)
- Zookeeper helps in performing leader election for partitions
- Zookeeper sends notifications to Kafka in case of changes
  (e.g. new topic, broker dies, broker comes up, delete topics, etc.…)
- **Kafka 2.x can't work without Zookeeper**
- **Kafka 3.x can work without Zookeeper (KIP-500) - using Kafka Raft instead**
- **Kafka 4.x will not have Zookeeper**
- Zookeeper by design operates with an odd number of servers (1, 3, 5, 7)
- Zookeeper has a leader (writes) the rest of the servers are followers (reads)
- (Zookeeper does NOT store consumer offsets with Kafka > v0.10)

# Zookeeper Cluster (ensemble)



# Should you use Zookeeper?

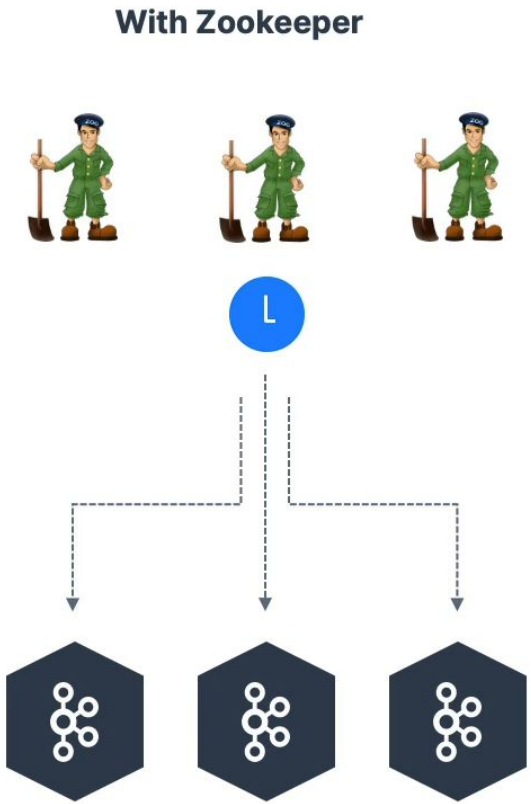
- With Kafka Brokers?
  - Yes, until Kafka 4.0 is out while waiting for Kafka without Zookeeper to be production-ready
- With Kafka Clients?
  - Over time, the Kafka clients and CLI have been migrated to leverage the brokers as a connection endpoint instead of Zookeeper
  - Since Kafka 0.10, consumers store offset in Kafka and Zookeeper and must not connect to Zookeeper as it is deprecated
  - Since Kafka 2.2, the `kafka-topics.sh` CLI command references Kafka brokers and not Zookeeper for topic management (creation, deletion, etc...) and the Zookeeper CLI argument is deprecated.
  - All the APIs and commands that were previously leveraging Zookeeper are migrated to use Kafka instead, so that when clusters are migrated to be without Zookeeper, the change is invisible to clients.
  - Zookeeper is also less secure than Kafka, and therefore Zookeeper ports should only be opened to allow traffic from Kafka brokers, and not Kafka clients
  - **Therefore, to be a great modern-day Kafka developer, never ever use Zookeeper as a configuration in your Kafka clients, and other programs that connect to Kafka.**

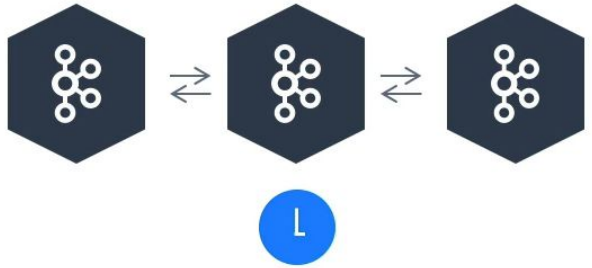
# About Kafka KRaft

- In 2020, the Apache Kafka project started to work to remove the Zookeeper dependency from it (KIP-500)
- Zookeeper shows scaling issues when Kafka clusters have > 100,000 partitions
- By removing Zookeeper, Apache Kafka can
  - Scale to millions of partitions, and becomes easier to maintain and set-up
  - Improve stability, makes it easier to monitor, support and administer
  - Single security model for the whole system
  - Single process to start with Kafka
  - Faster controller shutdown and recovery time

- Kafka 3.X now implements the Raft protocol (KRaft) in order to replace Zookeeper
  - Not production ready, see: https://github.com/apache/kafka/blob/trunk/config/kraft/README.md

# Kafka KRaft Architecture




L Quorum Leader

**Good job!**
by default Kafka automatically creates topics

Question 3:

If I produce to a topic that does not exist, by default

- ○ I will see an ERROR and my producer will not work
- ◉ **I will see a WARNING and Kafka will auto create the topic**

**Good job!**

by default it's 1 & 1, but these can be controlled by the settings num.partitions and default.replication.factor

Question 4:

When a topic is auto-created, how many partitions and replication factor does it have by default?

- ○ partitions: 3
  replication factor: 1
- ◉ **partitions: 1**
  **replication factor: 1**
- ○ partitions: 3
  replication factor: 3

**Good job!**

try running kafka-consumer-groups --list to see!

Question 5:

kafka-console-consumer

- ○ does not use a group id
- ○ always uses the same group id
- ◉ **uses a random group id**

**Good job!**

Question 6:

I should override the group.id for `kafka-console-consumer` using

- (●) **--group mygroup**
- ( ) **--property group.id=mygroup**

# Kafka in Diagrams

# Basic Kafka commands

Start Broker

**bin/kafka-server-start.sh** config/server.properties

Start Zookeeper

**bin/zookeeper-server-start.sh** config/zookeeper.properties

**bin/kafka-topics.sh** \
--bootstrap-server localhost:9092 \
--create \
--replication-factor 1 \
--partitions 3 \
--topic test

Create new topic

**bin/kafka-topics.sh** \
--bootstrap-server localhost:9092 \
--list

List all topics

**bin/kafka-topics.sh** \
--bootstrap-server localhost:9092 \
--describe \
--topic test

Details about the topic

**bin/kafka-console-producer.sh** \
--broker-list localhost:9092 \
--topic test

Start console producer

**bin/kafka-console-consumer.sh** \
--bootstrap-server localhost:9092 \
--topic test \
--from-beginning

Start console consumer

**bin/kafka-consumer-groups.sh** \
--bootstrap-server localhost:9092 \
--list

List all consumer groups

**bin/kafka-consumer-groups.sh** \
--bootstrap-server localhost:9092 \
--group test \
--describe

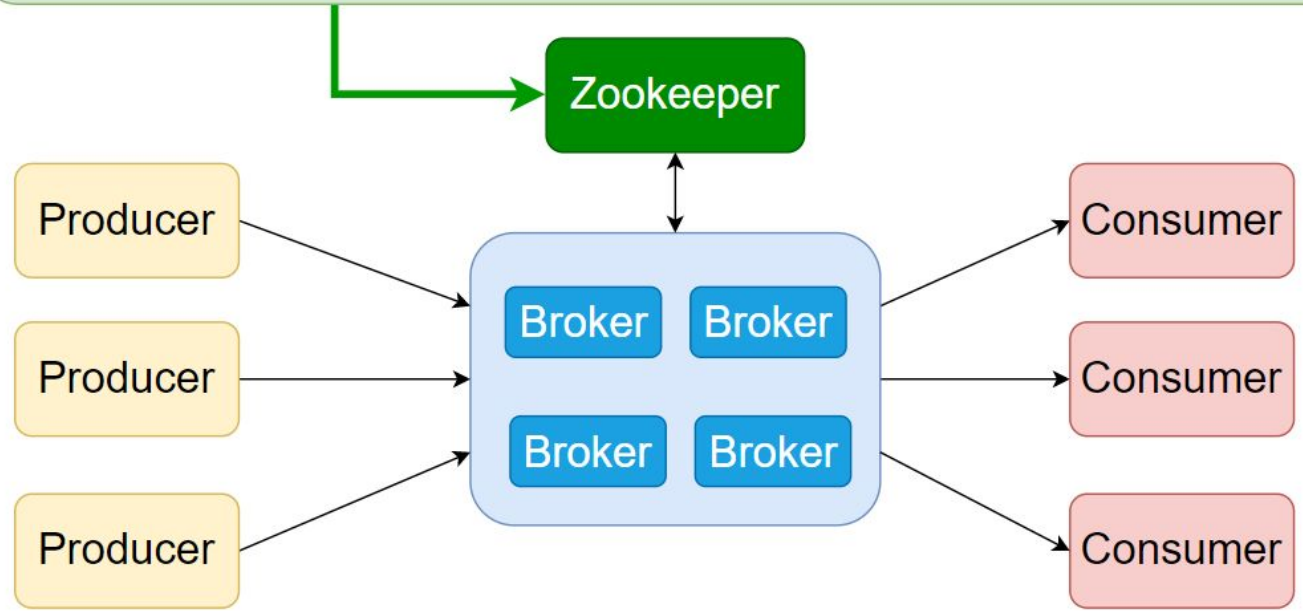
Consumer group details

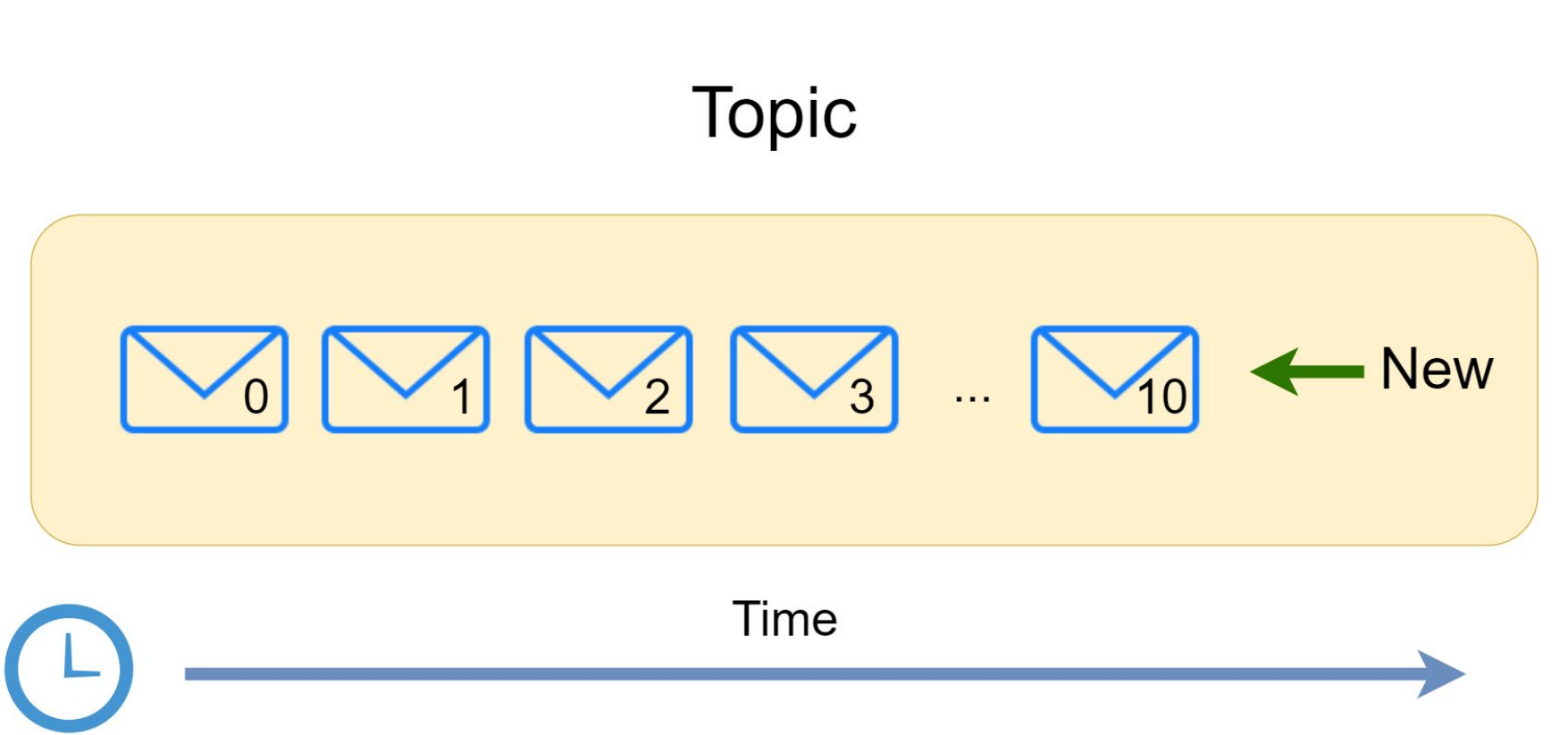
# Zookeeper

Maintains list of active brokers

Elects controller

Manages configuration of the topics and partitions




Topic
0
1
2
3
...
10
New
Time

# Message structure

- Timestamp
- Offset number (unique across partition)
- Key (optional)
- Value (sequence of bytes)