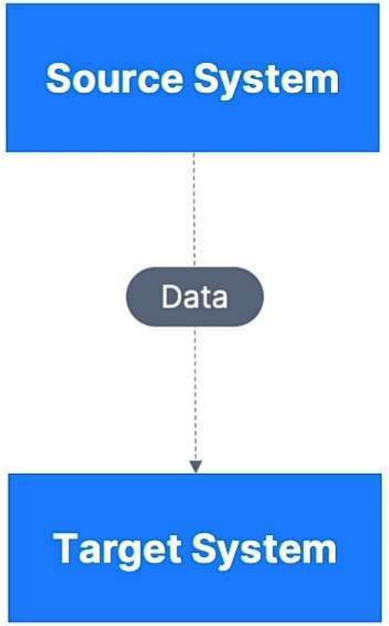
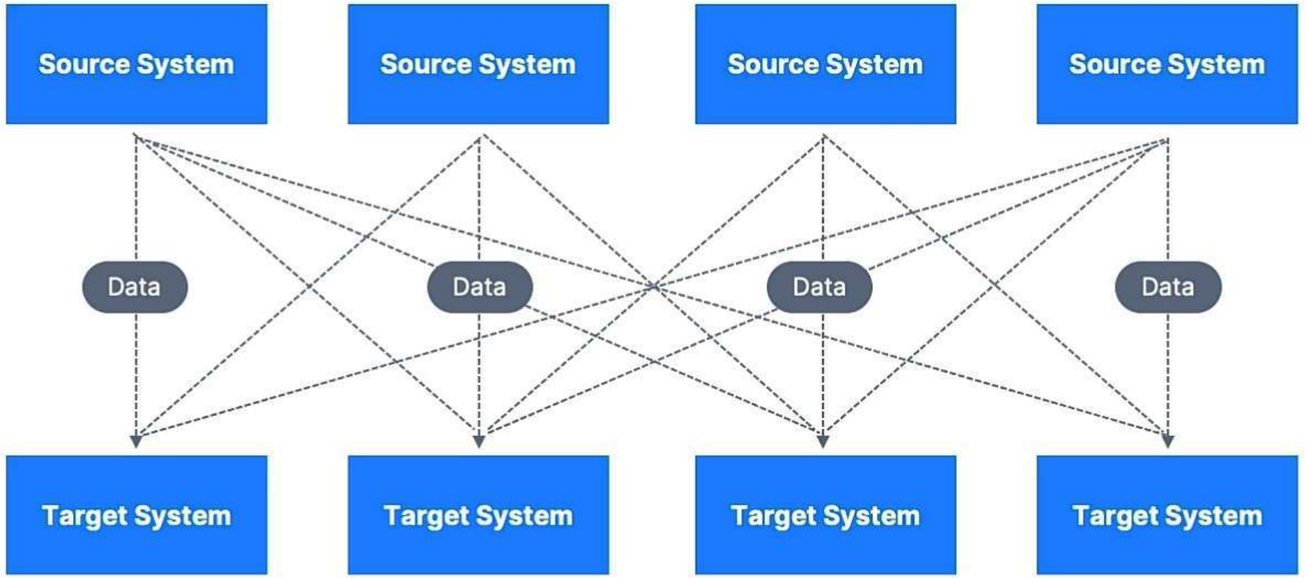
How companies start



Simple at first!

After a while...



Very complicated!

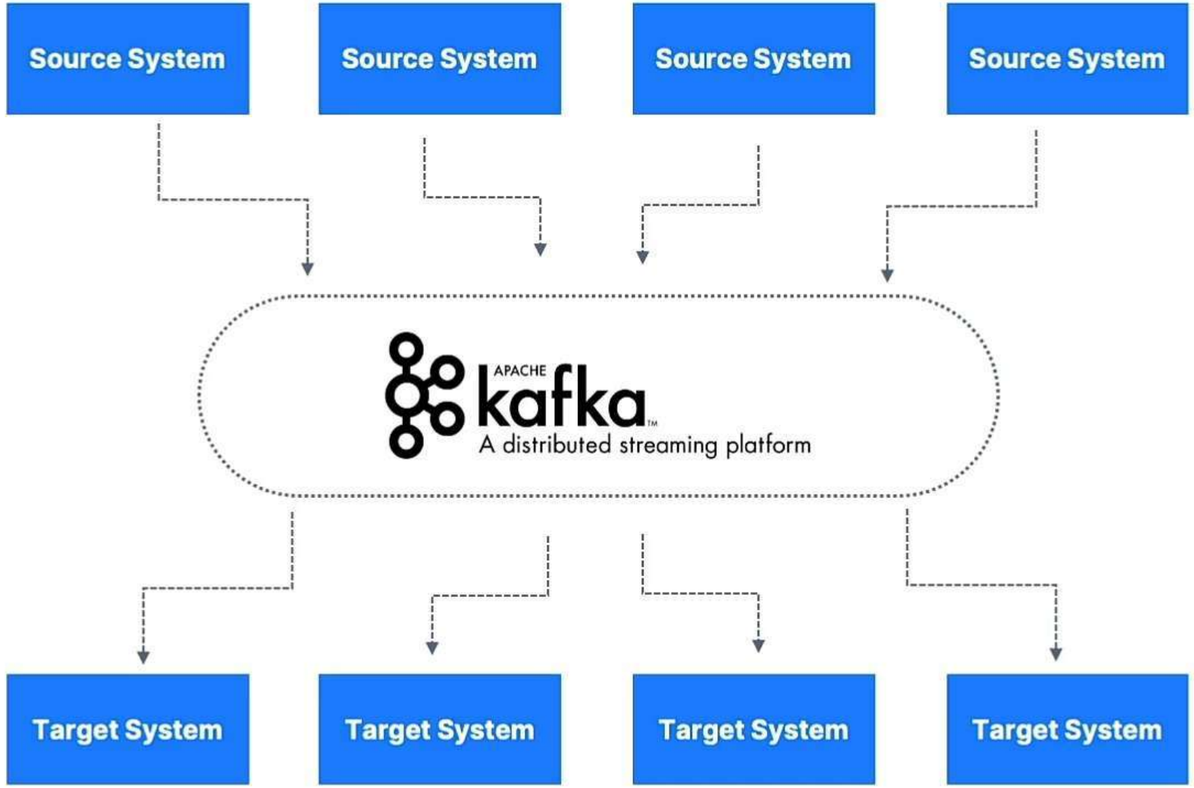
Types of problems organisations are facing with the previous architecture

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

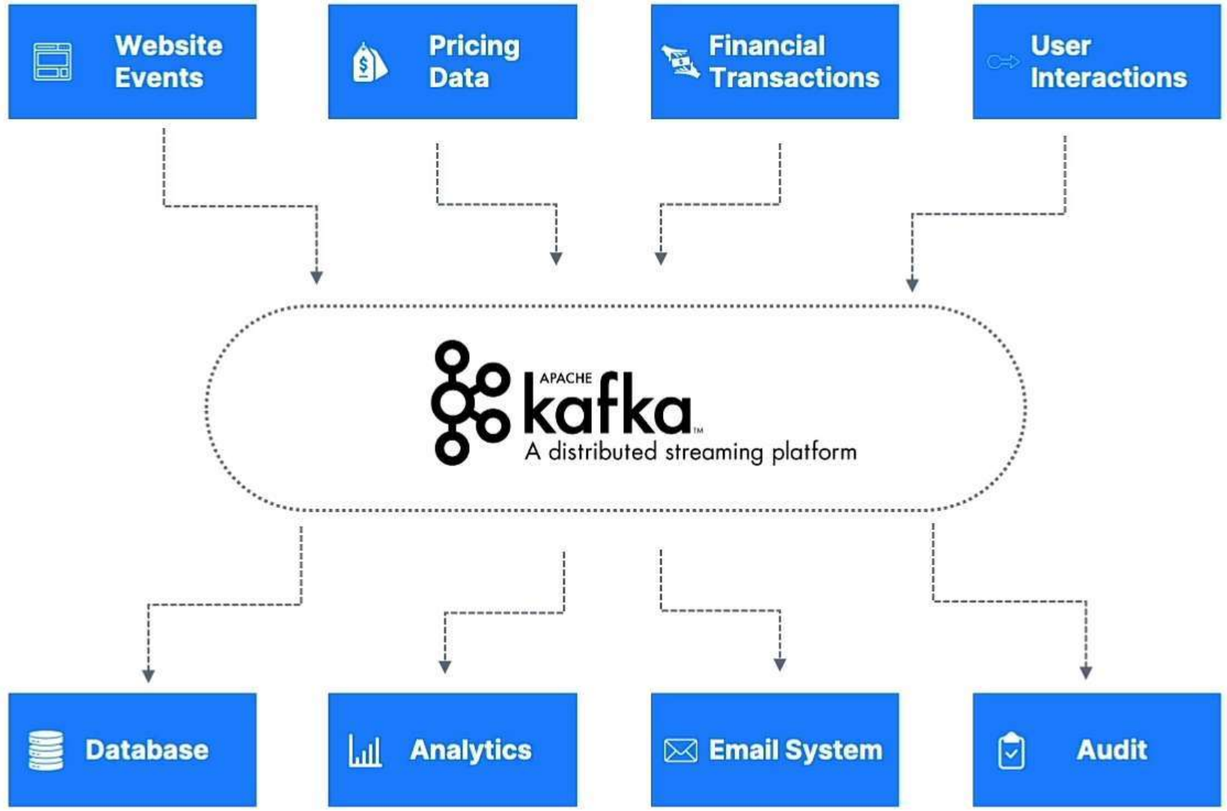
- Each integration comes with difficulties around
  - Protocol – how the data is transported *(TCP, HTTP, REST, FTP, JDBC…)*
  - Data format – how the data is parsed *(Binary, CSV, JSON, Avro, Protobuf…)*
  - Data schema & evolution – how the data is shaped and may change

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

Why Apache Kafka:
Decoupling of data streams & systems



Why Apache Kafka:
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...

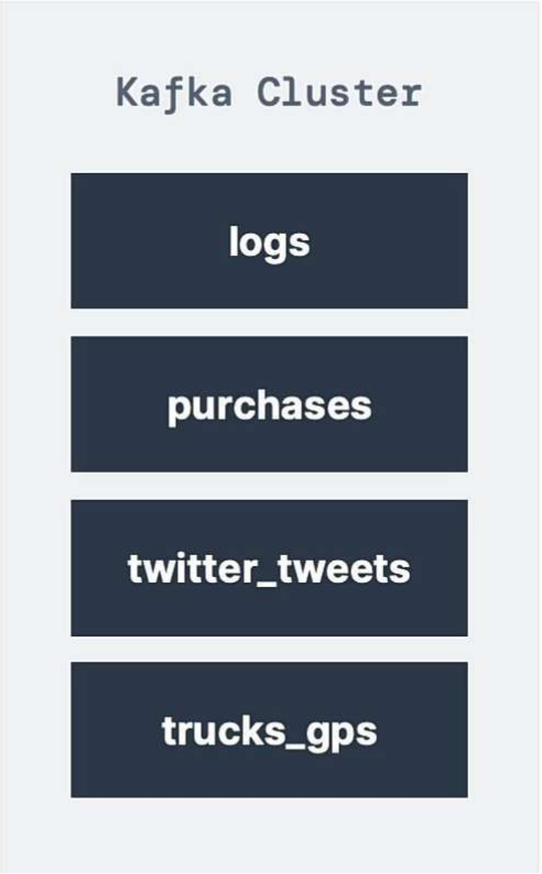
- **Netflix** uses Kafka to apply recommendations in real-time while you're watching TV shows
- **Uber** uses Kafka to gather user, taxi and trip data in real-time to compute and forecast demand, and compute surge pricing in real-time
- **LinkedIn** uses Kafka to prevent spam, collect user interactions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

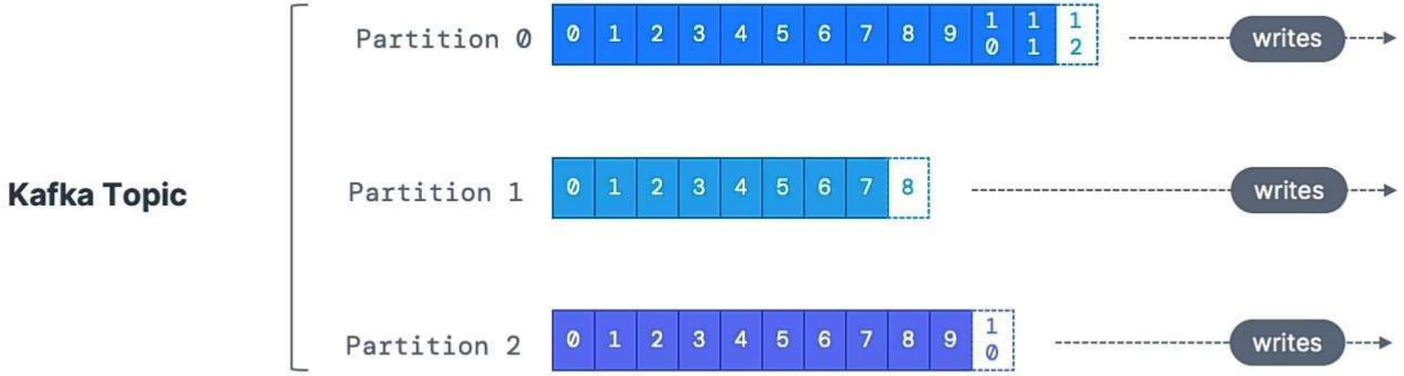
- Like a table in a database (without all the constraints)
- You can have as many topics as you want
- A topic is identified by its name
- Any kind of message format
- The sequence of messages is called a data stream
- You cannot query topics, instead, use Kafka Producers to send data and Kafka Consumers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



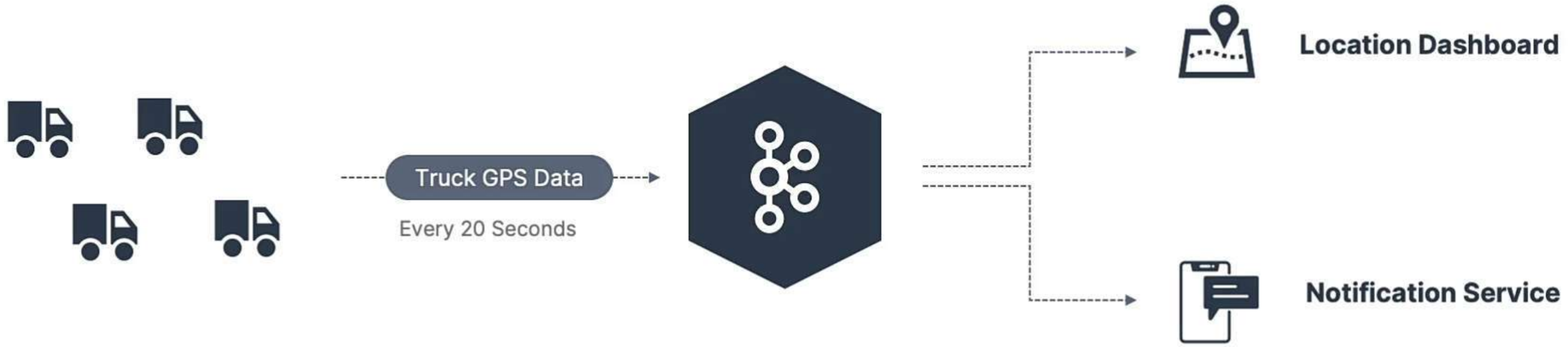
Partitions and offsets

- 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

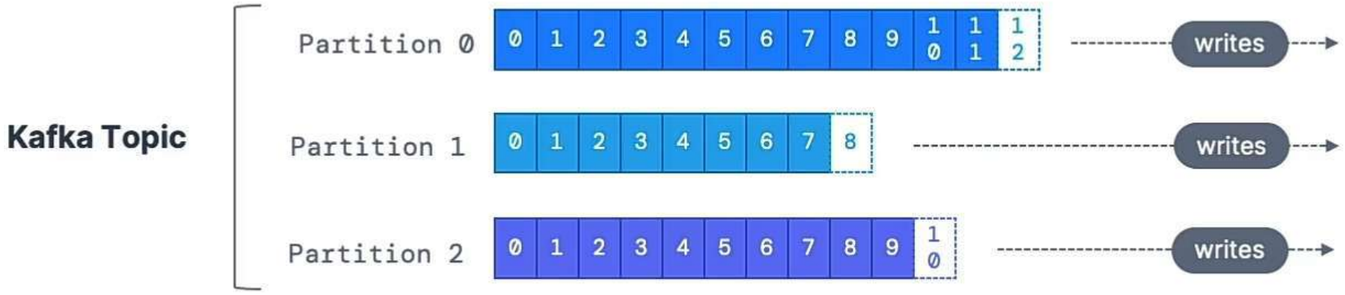


Topic example: `truck_gps`



- 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)

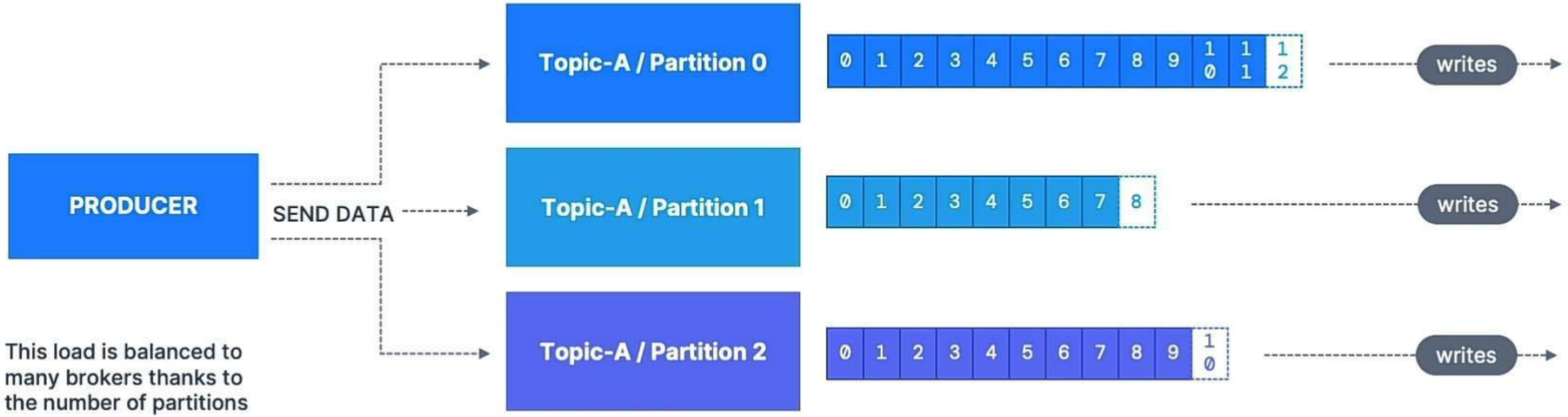
Topics, partitions and offsets – important notes



- 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

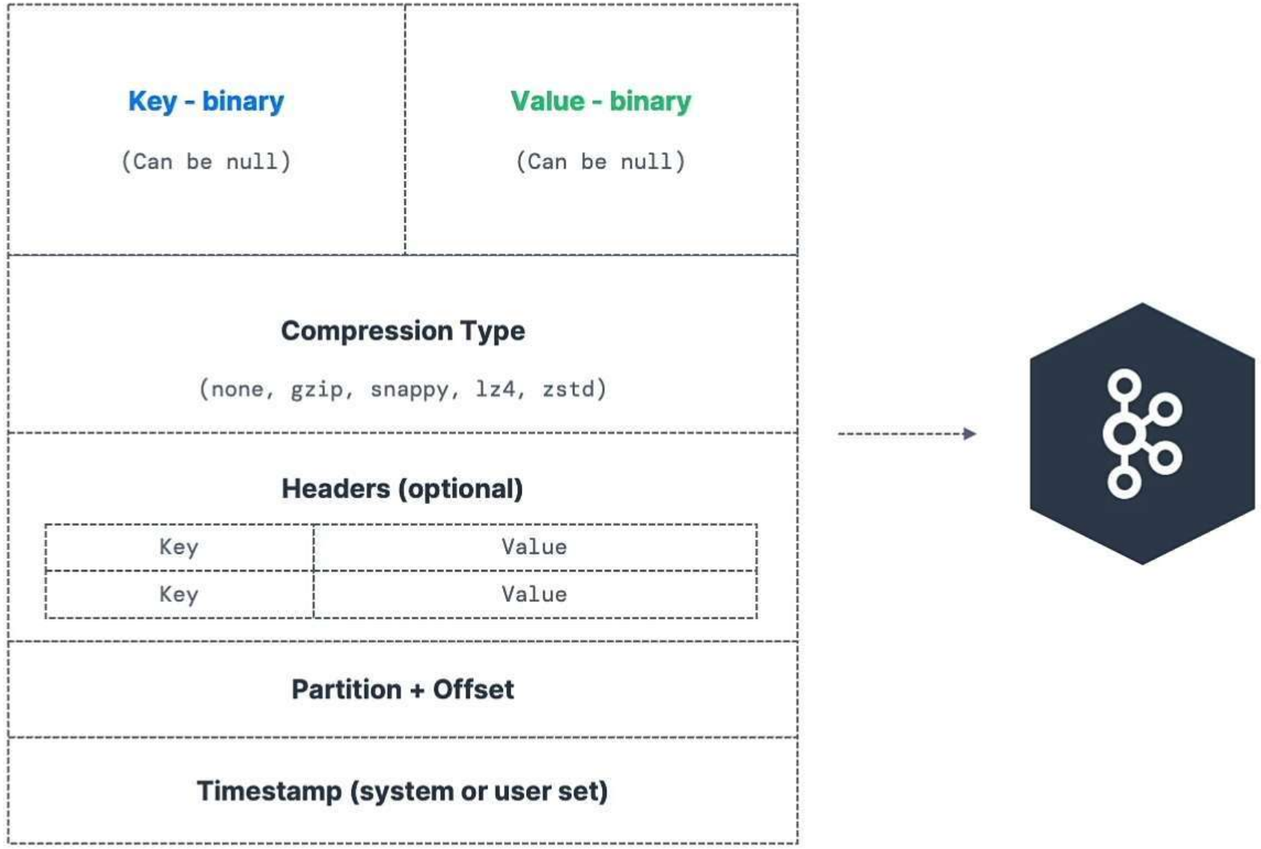


Producers: Message keys

- 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`)

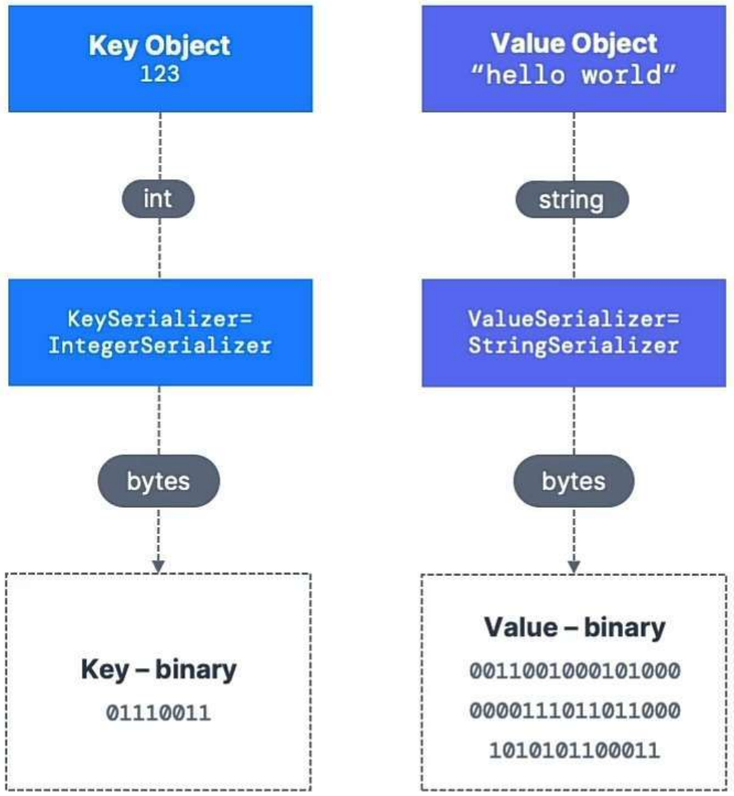
Kafka Messages anatomy...

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

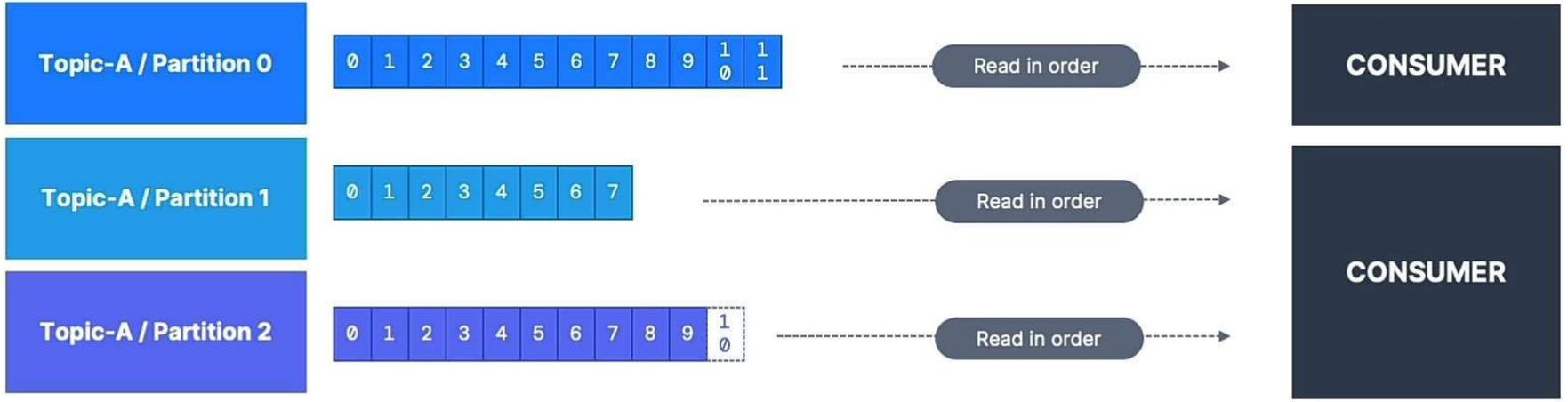
- A Kafka partitioner is a code logic that takes a record and determines to which partition to send it into.

- **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)
```

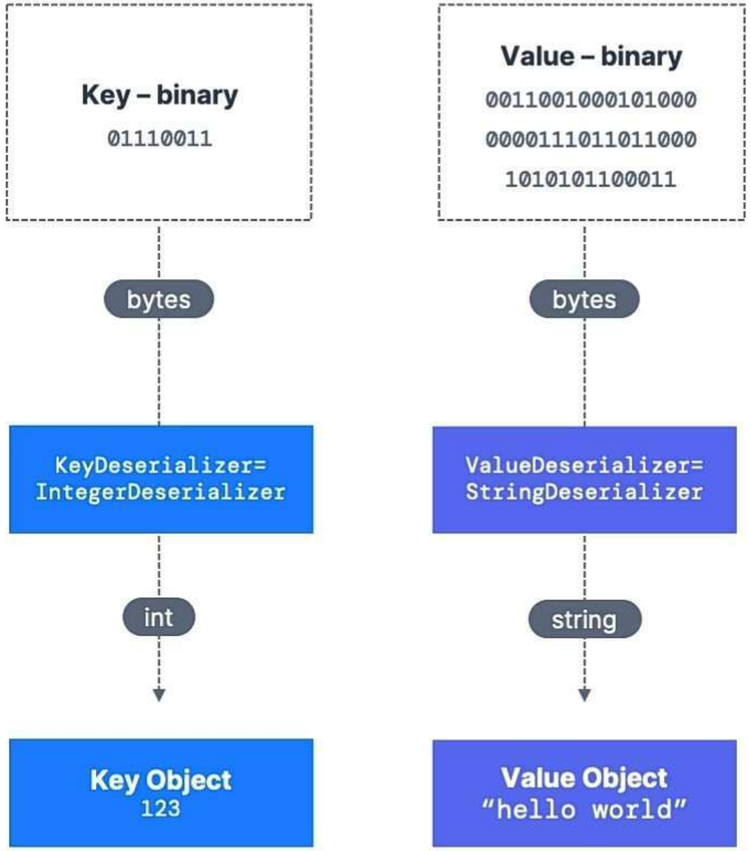
Consumers

- 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**



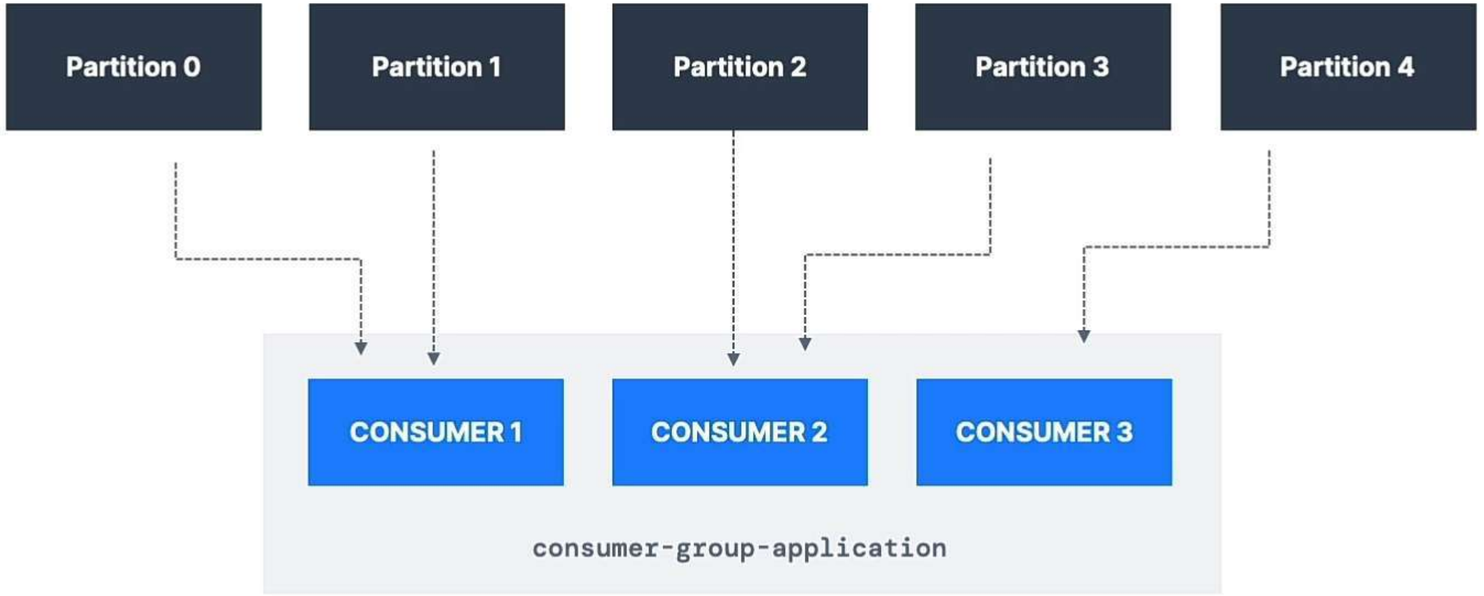
Consumer Deserializer

- 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)



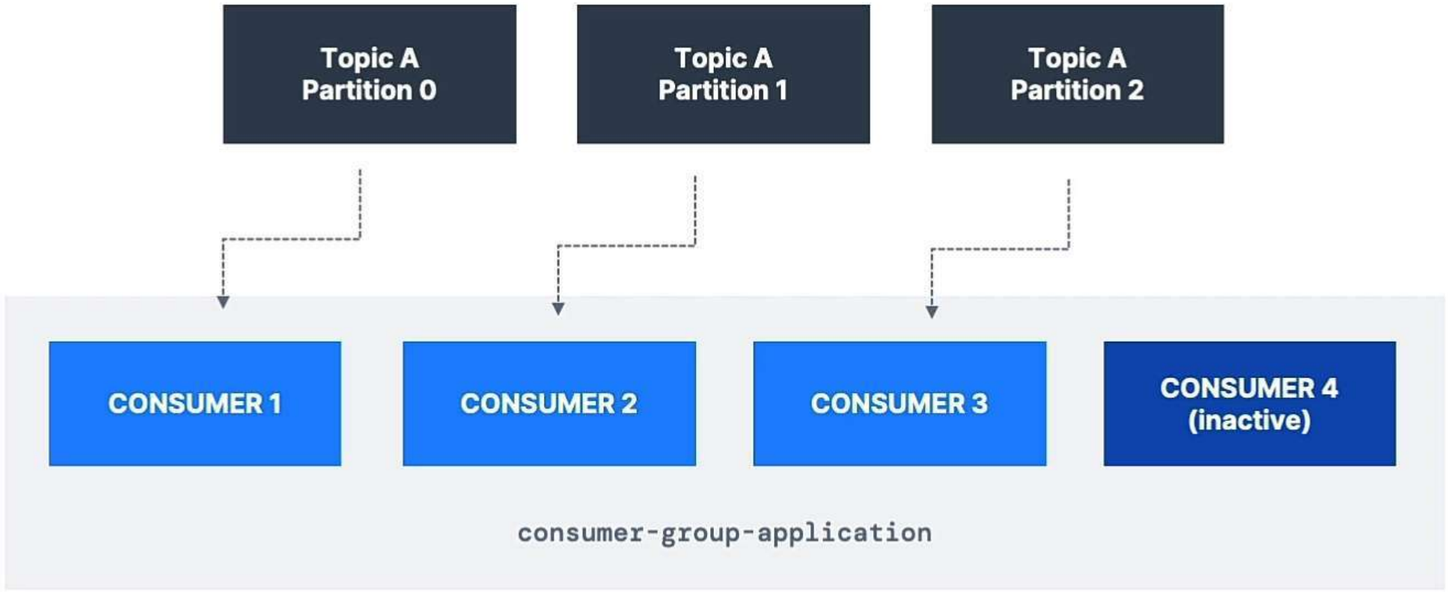
Consumer Groups

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



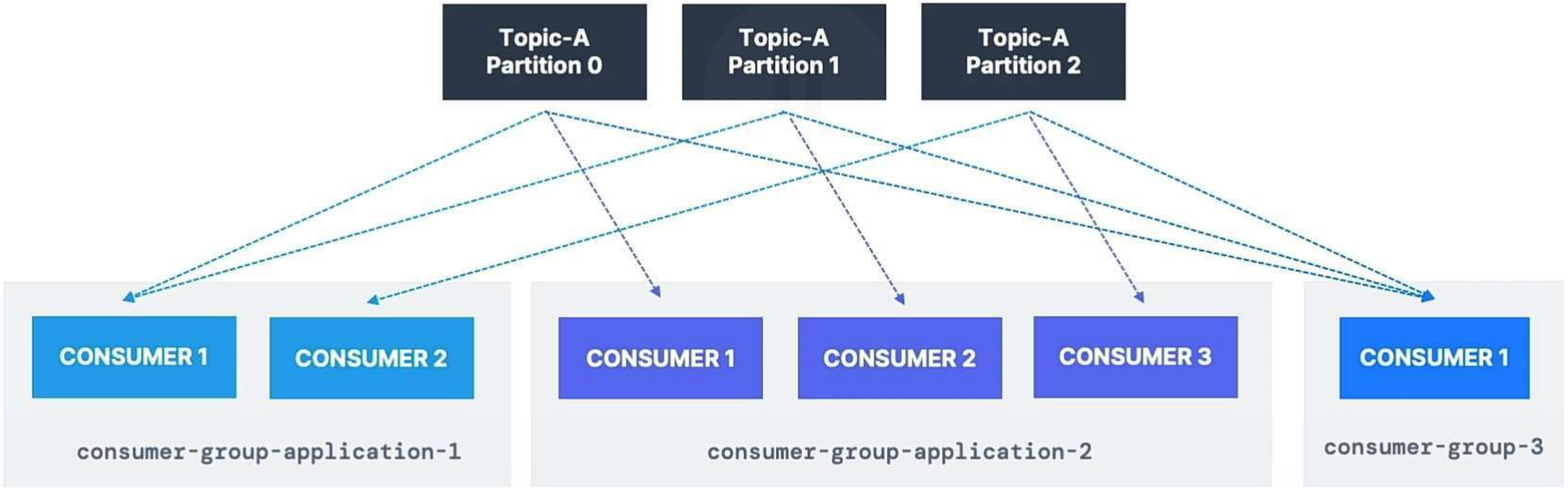
Consumer Groups - What if too many consumers?

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



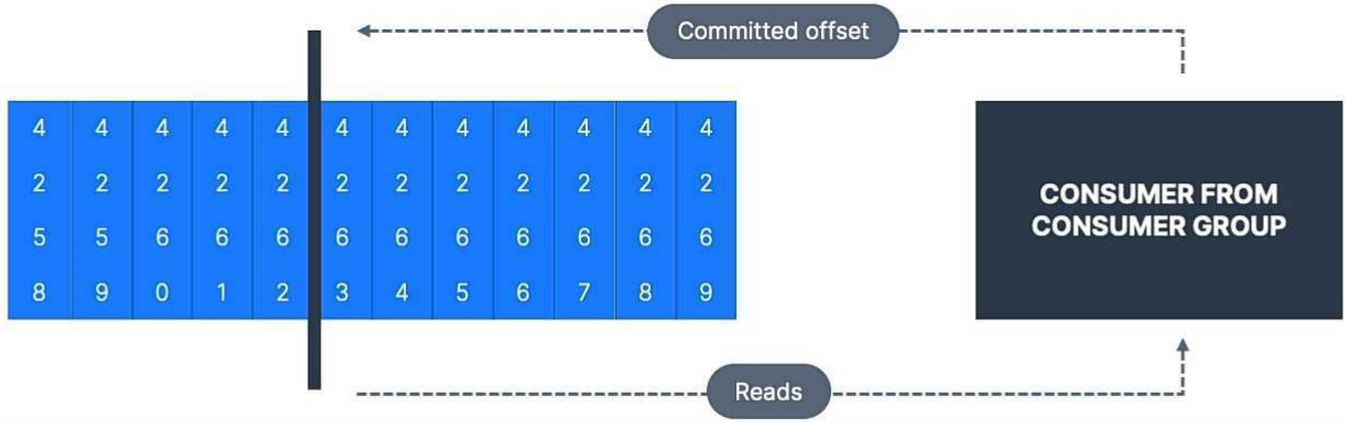
Multiple Consumers on one topic

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

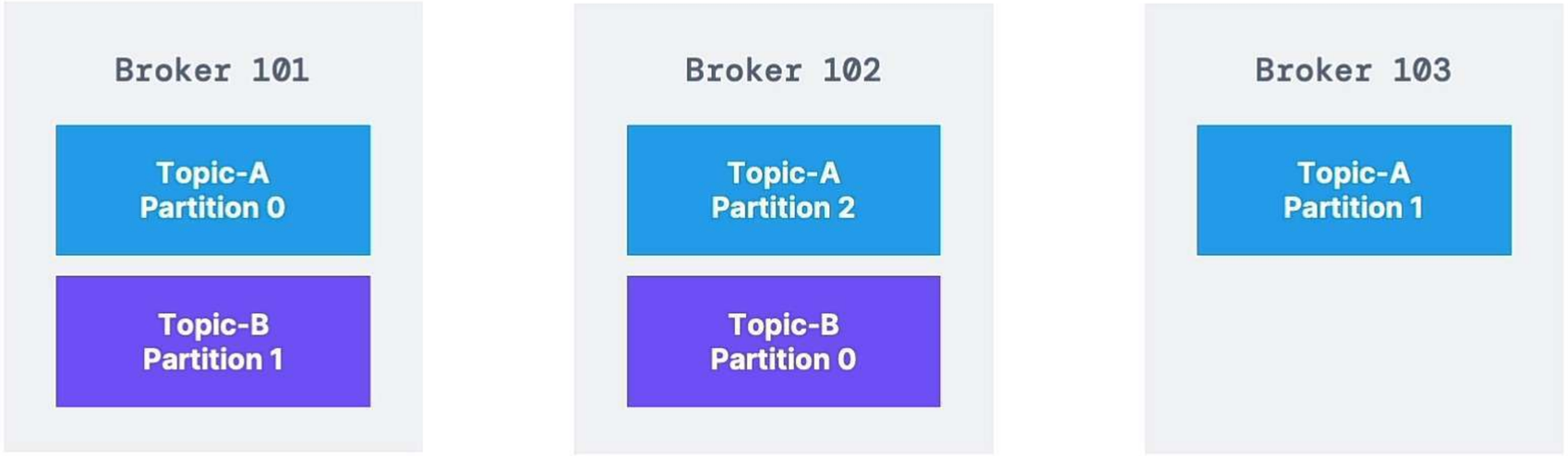
- By default, Java Consumers will automatically commit offsets (at least once)
- There are 3 delivery semantics if you choose to commit manually
- At least once (usually preferred)
  - Offsets are committed after the message is processed
  - If the processing goes wrong, the message will be read again
  - 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

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)

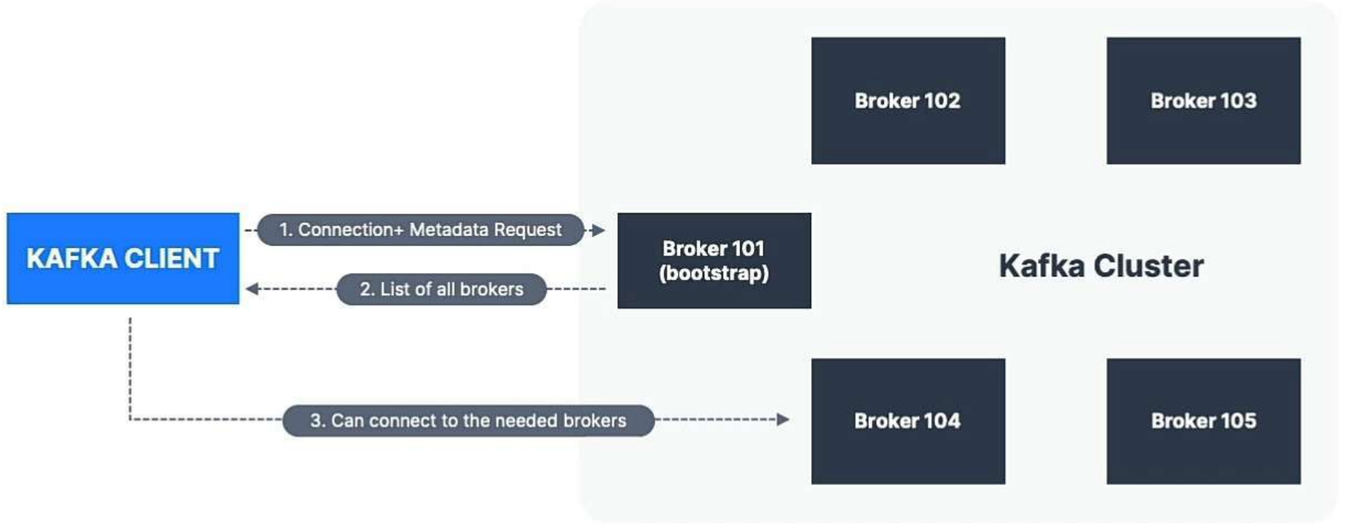
Brokers and topics

- 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



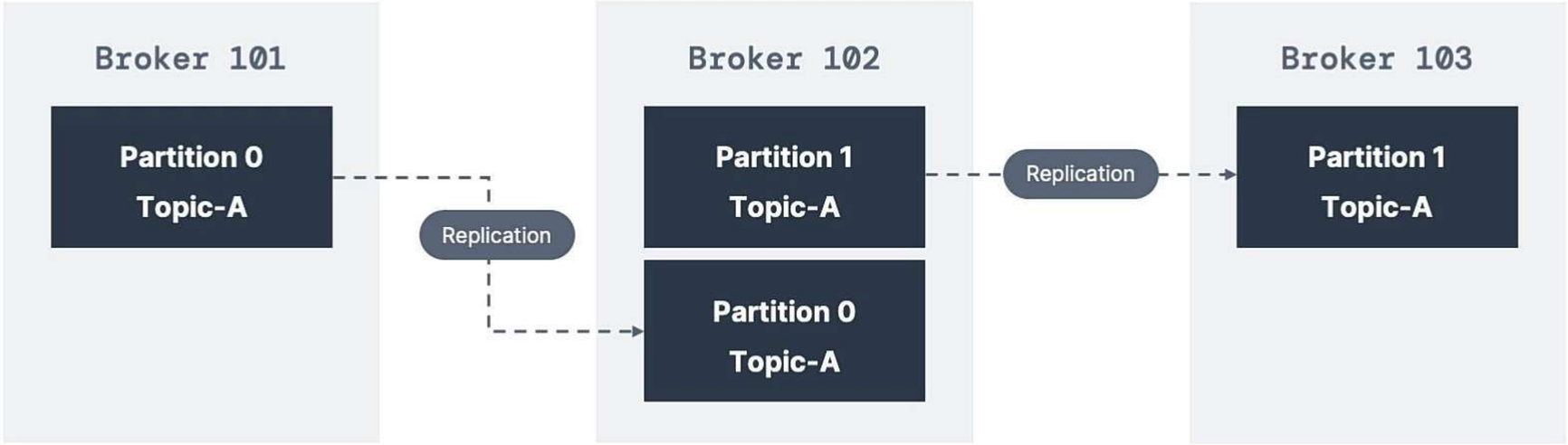
Kafka Broker Discovery

- 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)



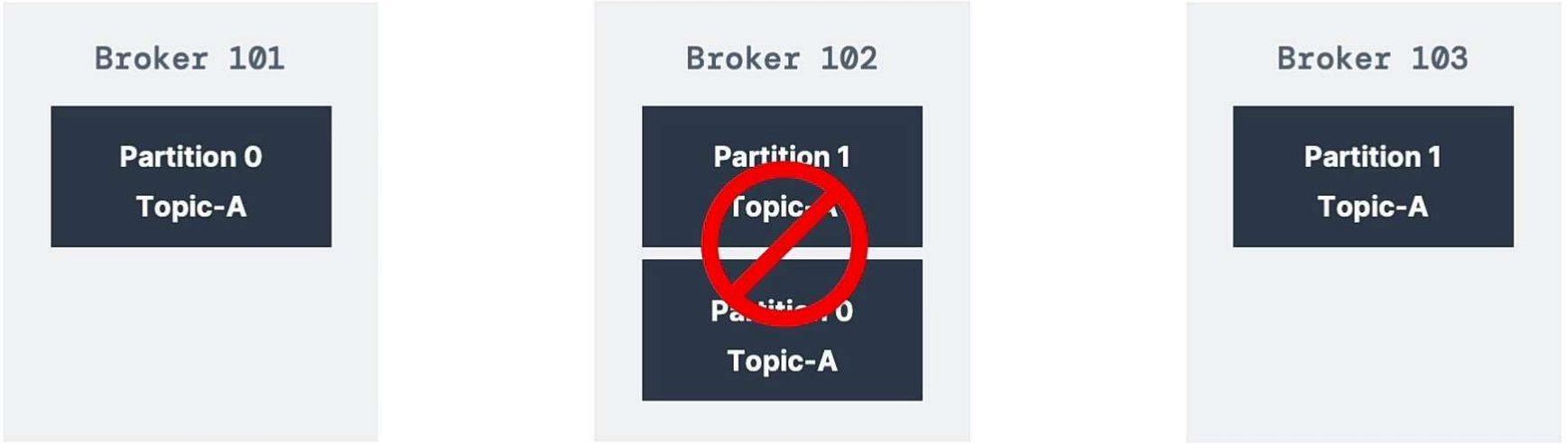
Topic replication factor

- 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



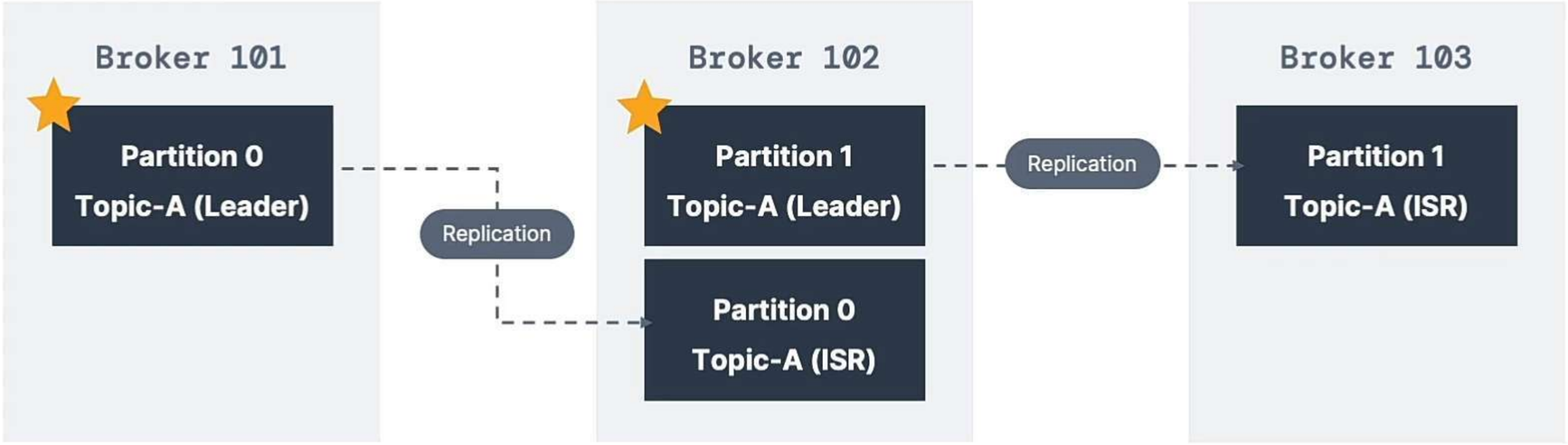
Topic replication factor

- 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)

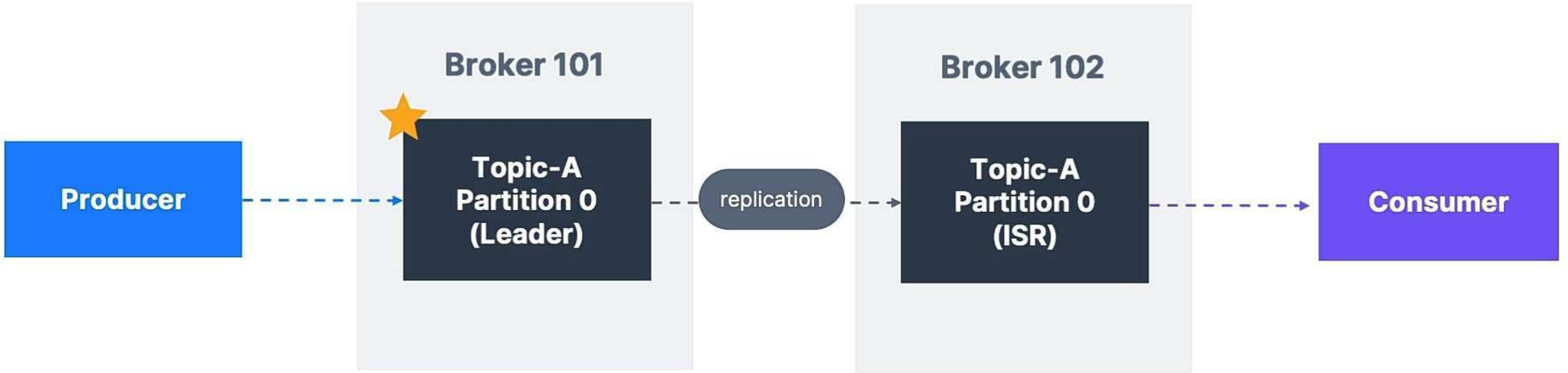


Default producer & consumer behavior with leaders

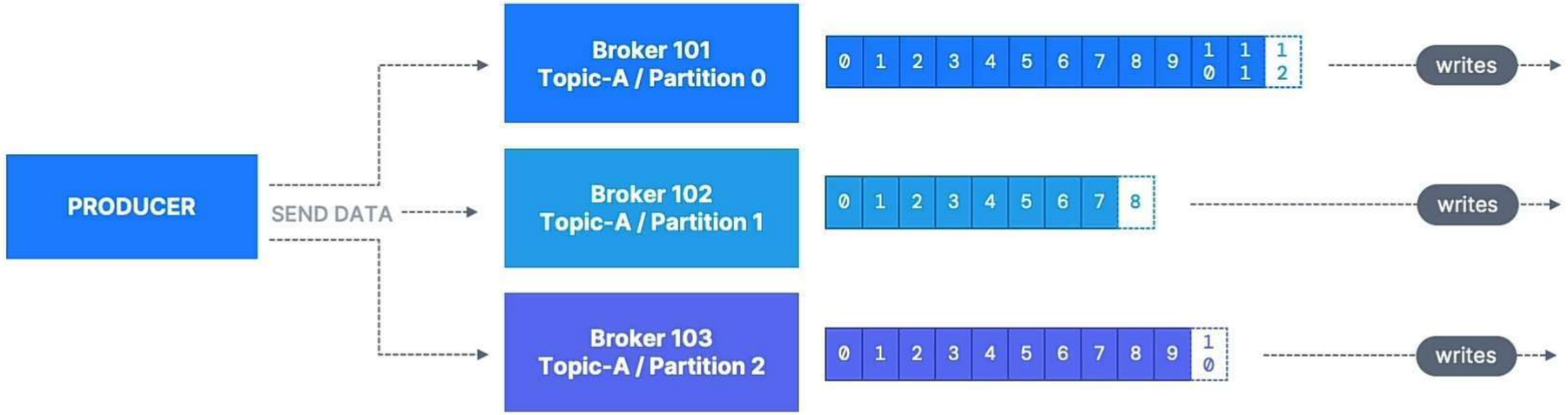
- Kafka Producers can only write to the leader broker for a partition
- Kafka Consumers by default will read from the leader broker for a partition

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



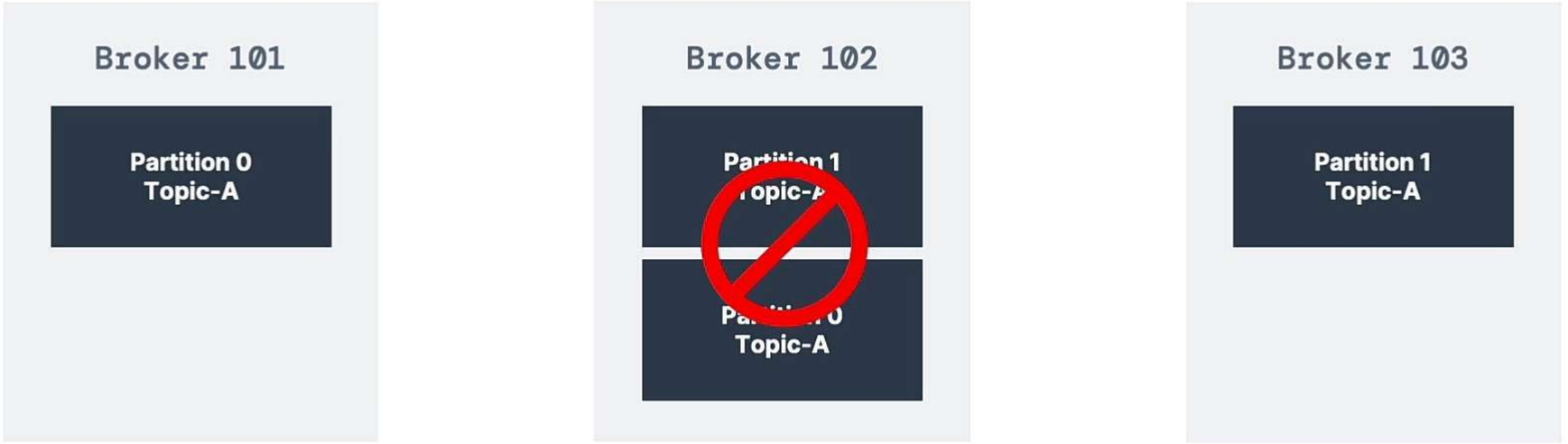
Producer Acknowledgements (acks)



- 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

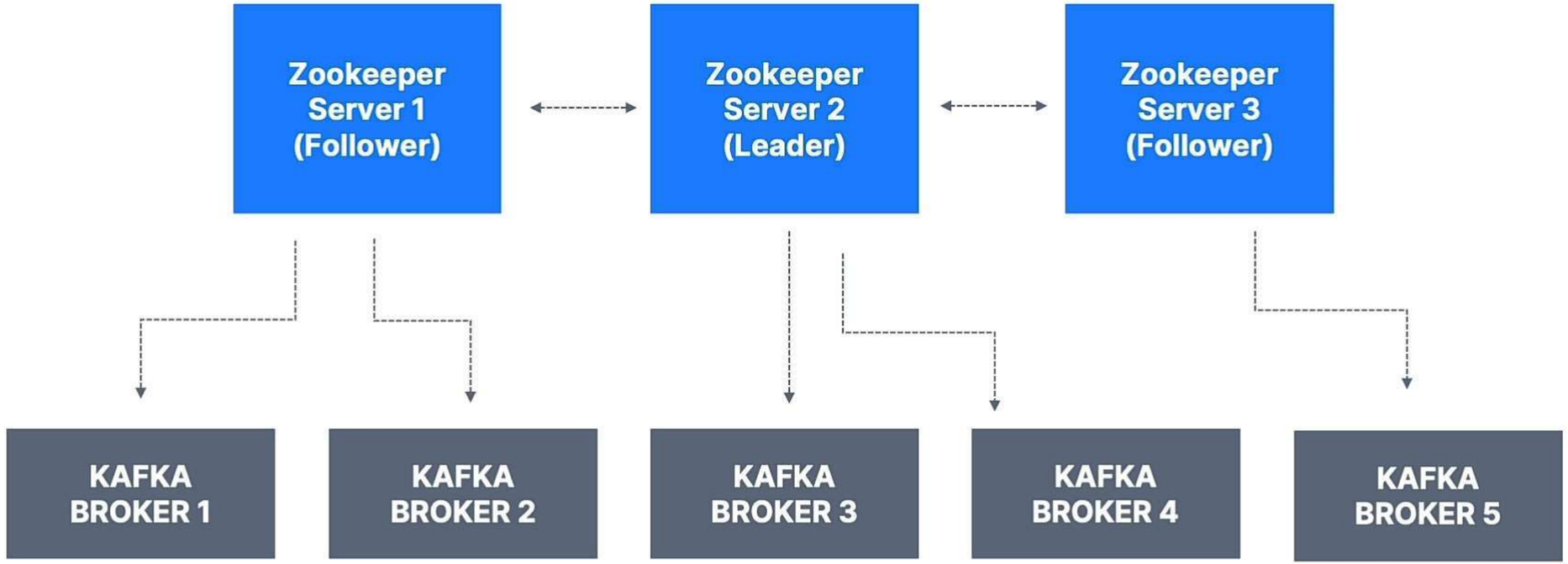
- For a topic replication factor of 3, topic data durability can withstand 2 brokers loss.
- As a rule, for a replication factor of N, you can permanently lose up to N-1 brokers and still recover your data.



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.**

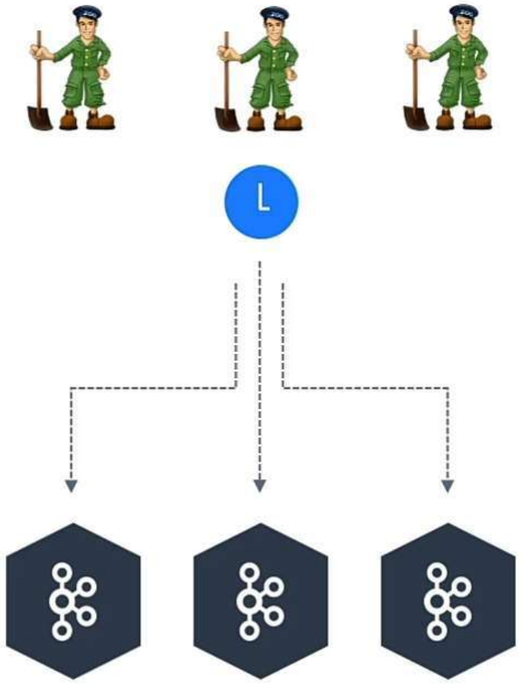
About Kafka KRaft

- 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

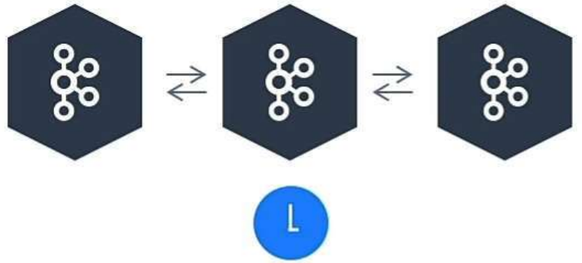
- Kafka 3.X now implements the Raft protocol (KRaft) in order to replace Zookeeper
  - Not production ready, see: https://github.com/apache/kafka/blob/trunk/config/kraft/README.md

Kafka KRaft Architecture

With Zookeeper

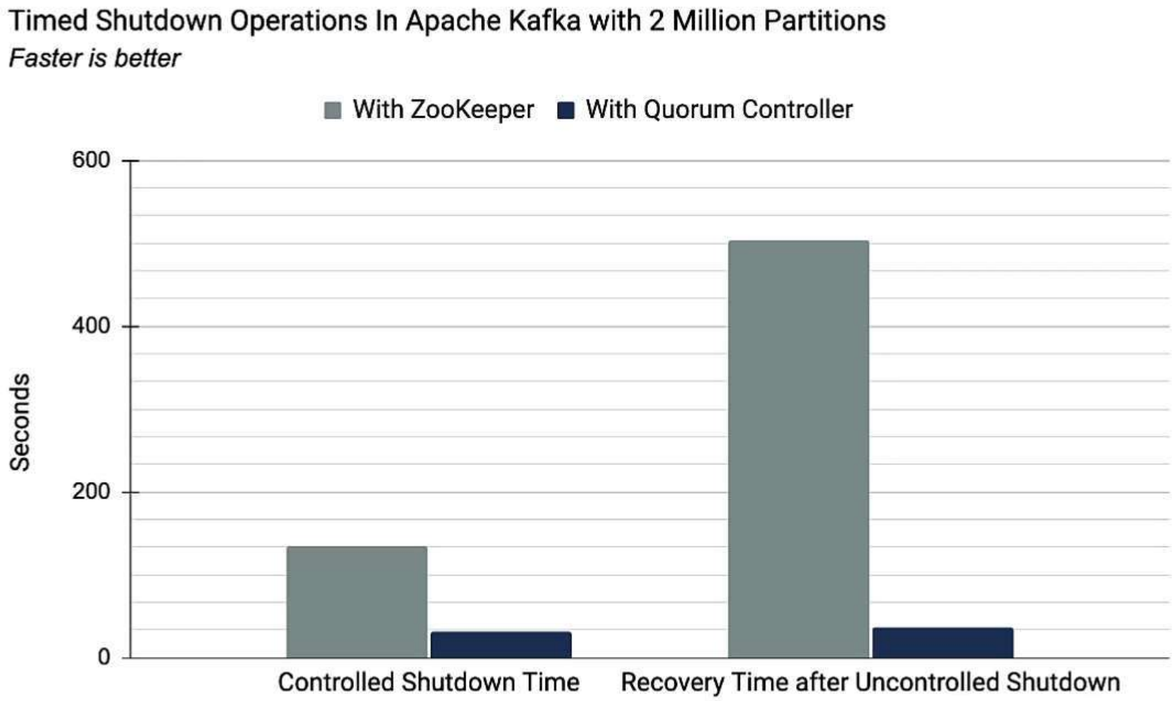


With Quorum Controller



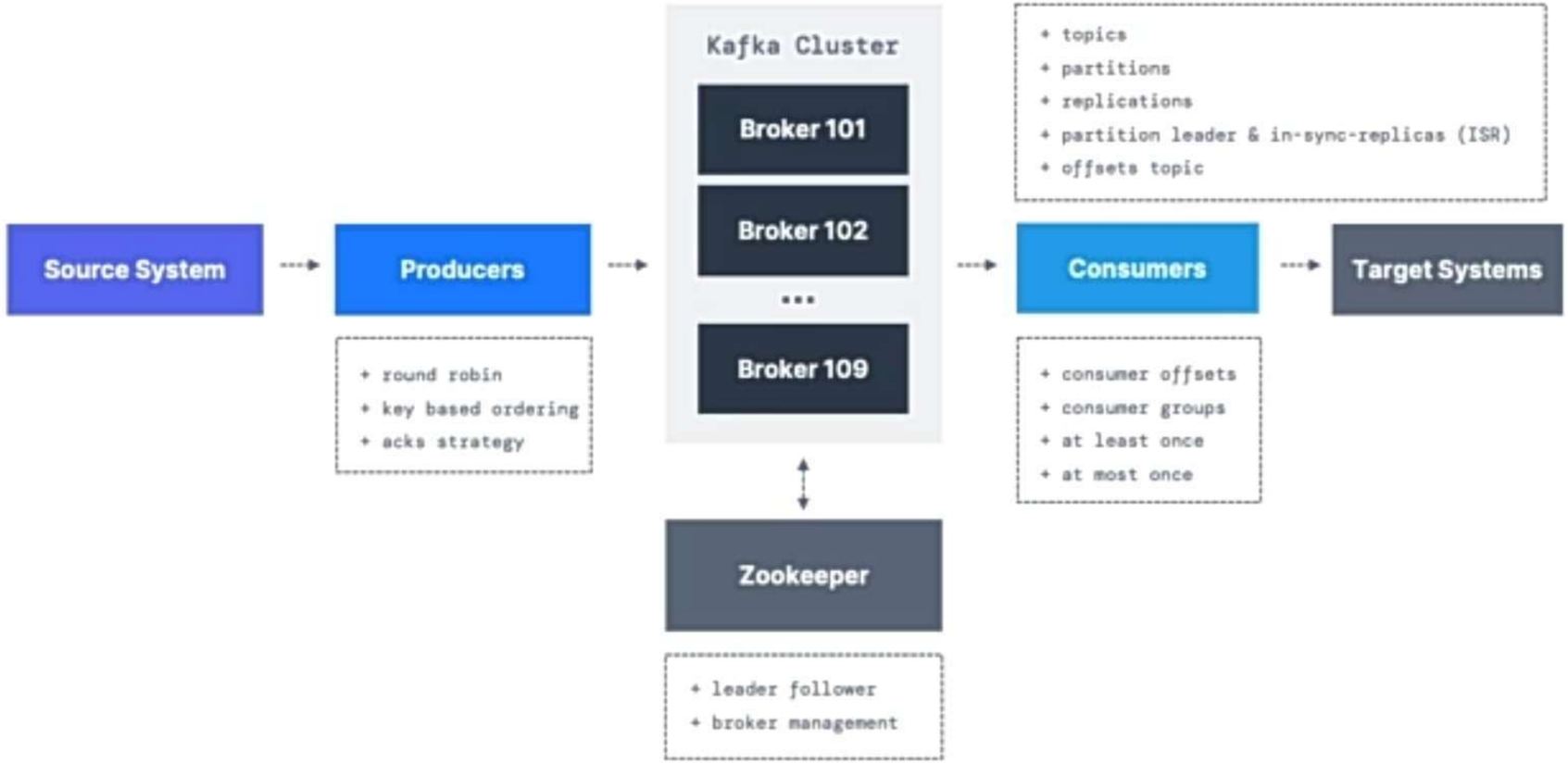
L **Quorum Leader**

KRaft Performance Improvements



https://www.confluent.io/blog/kafka-without-zookeeper-a-sneak-peek/

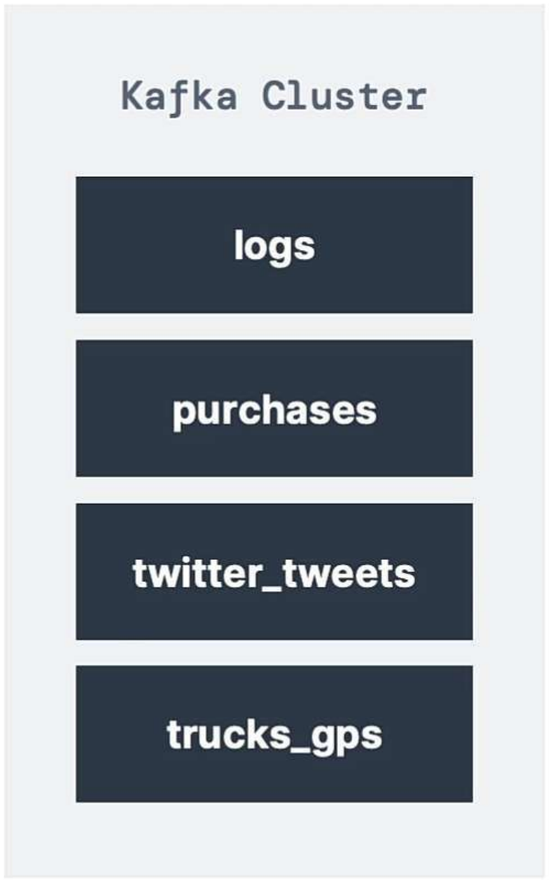
Theory Roundup
We've looked at all the Kafka concepts



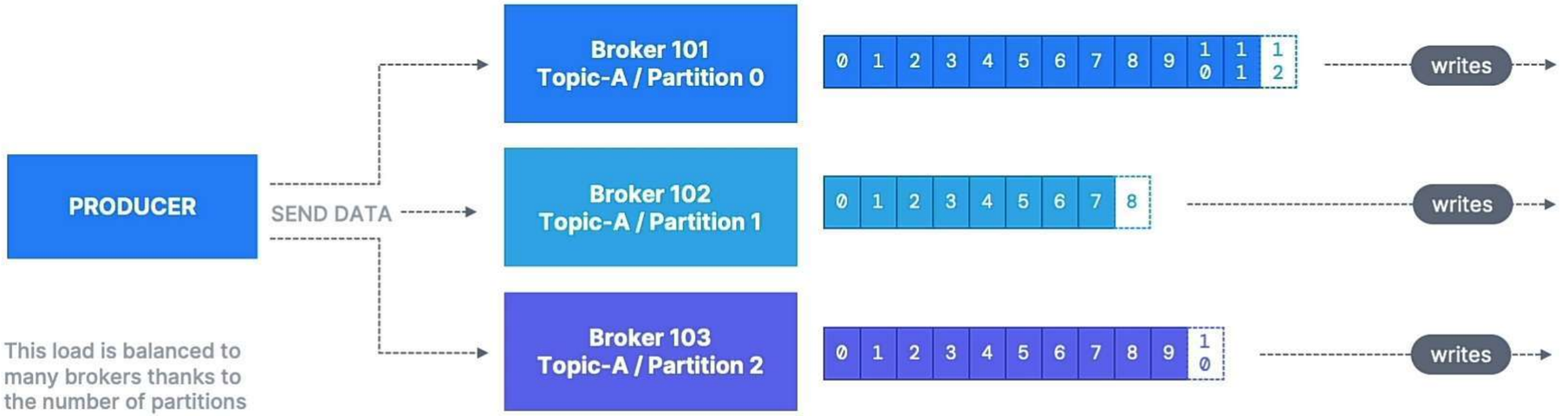
Kafka CLI: `kafka-topics.sh`

- Kafka Topic Management

1. Create Kafka Topics
2. List Kafka Topics
3. Describe Kafka Topics
4. Increase Partitions in a Kafka Topic
5. Delete a Kafka Topic

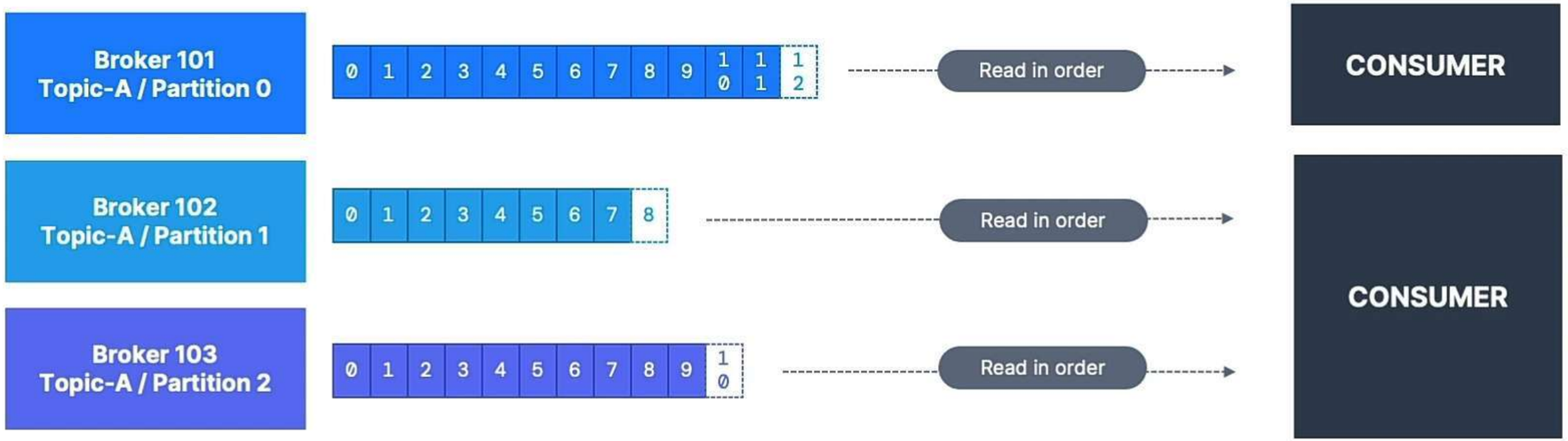


Kafka CLI: kafka-console-producer.sh



1. Produce without keys
2. Produce with keys

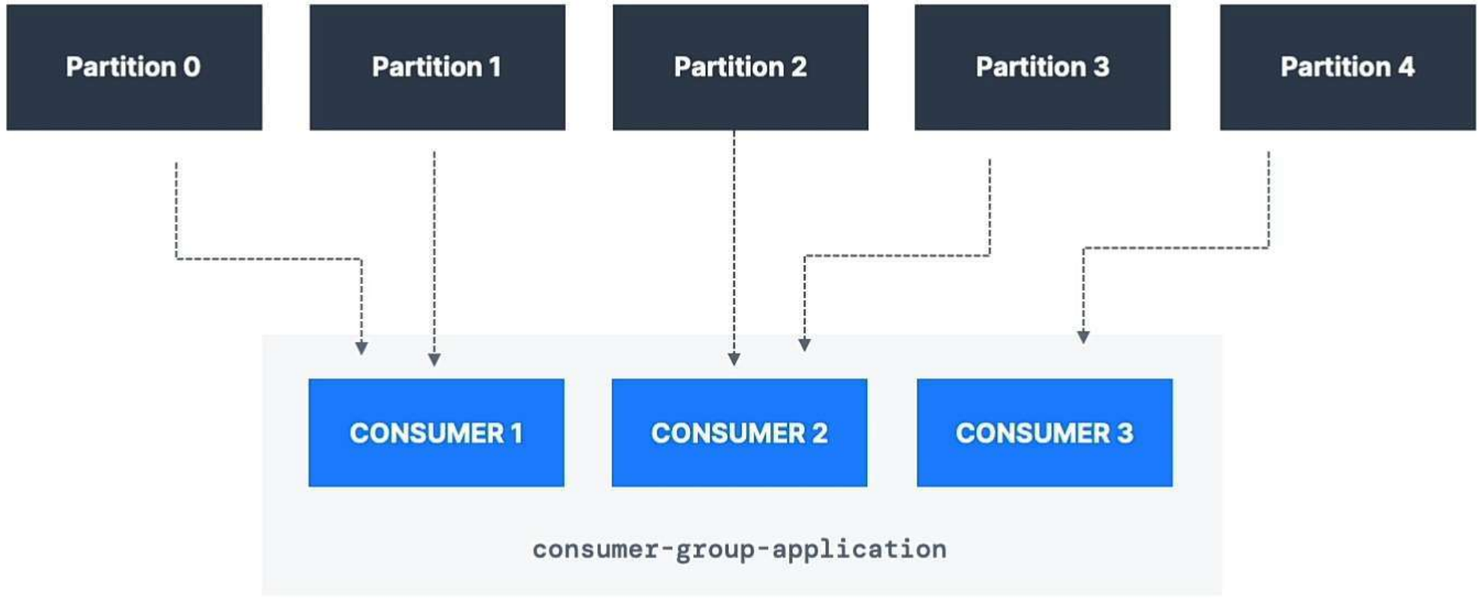
Kafka CLI: `kafka-console-consumer.sh`



1. Consume from tail of the topic
2. Consume from the beginning of the topic
3. Show both key and values in the output

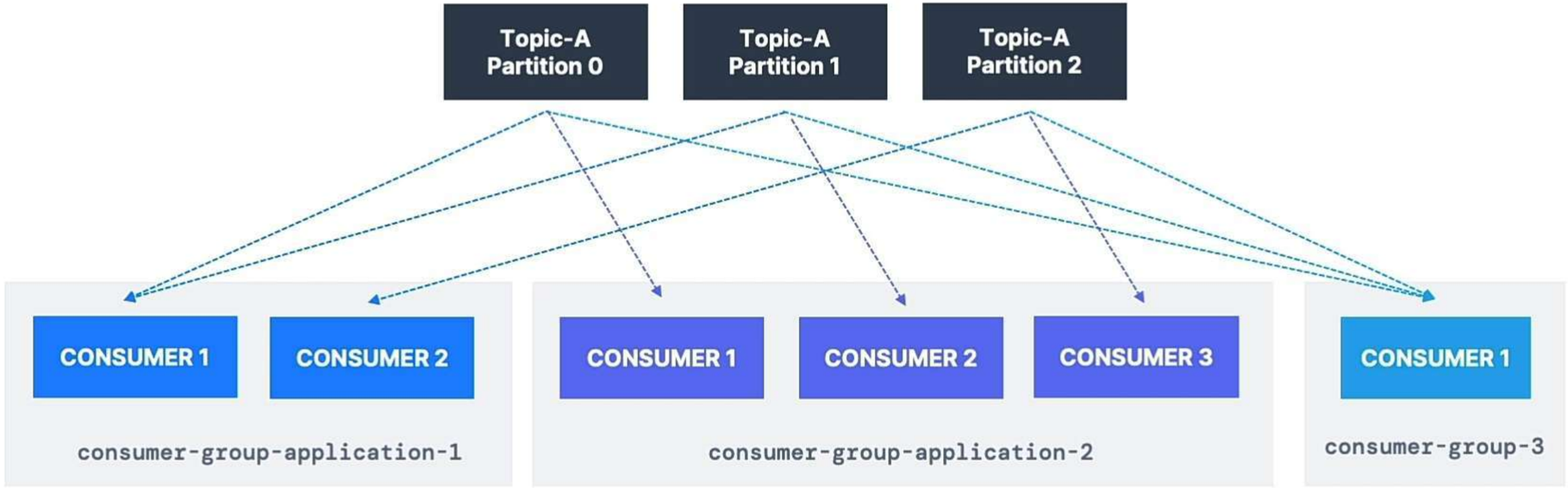
CLI Consumer in Groups with `kafka-console-consumer.sh`

- Learn about --group parameter
- See how partitions read are divided amongst multiple CLI consumers



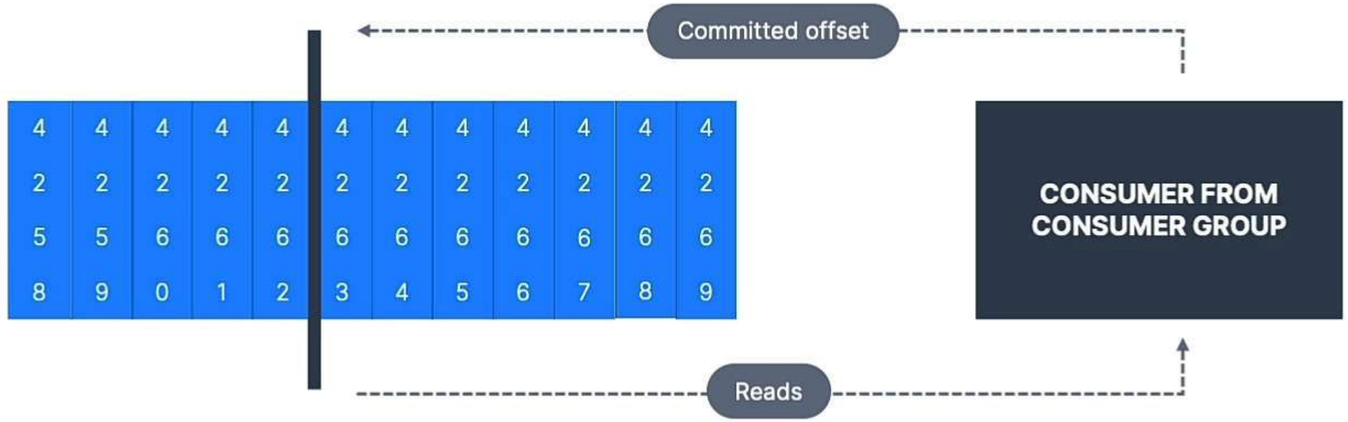
Consumer Group Management CLI
`kafka-consumer-groups.sh`

1. List consumer groups
2. Describe one consumer group
3. Delete a consumer group



Consumer Groups – Reset Offsets
`kafka-consumer-groups.sh`

1. Start / Stop Console Consumer
2. Reset Offsets
3. Start Console Consumer and see the outcome

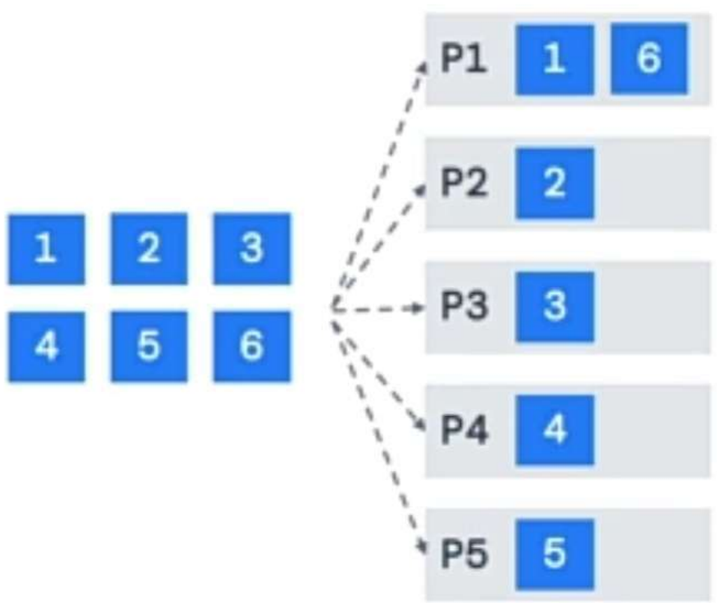


Kafka Producer: Java API - Basics

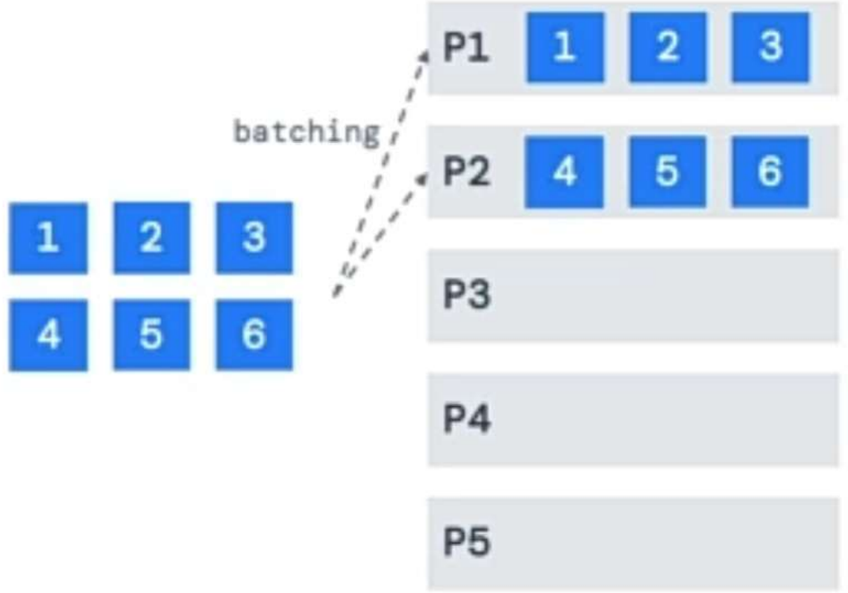
- Learn how to write a basic producer to send data to Kafka
- View basic configuration parameters
- Confirm we receive the data in a Kafka Console Consumer

Kafka Producer: Java API - Callbacks

- Confirm the partition and offset the message was sent to using Callbacks
- We'll look at the interesting behavior of **StickyPartitioner**



Round Robin



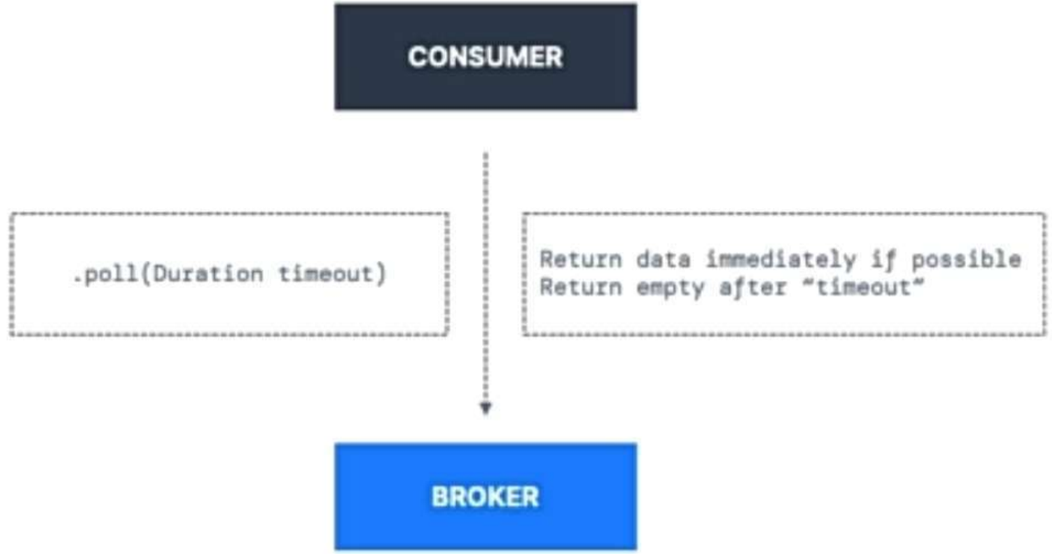
Sticky Partitioner (performance improvement)

Kafka Producer: Java API – with Keys

- Send non-null keys to the Kafka topic
- Same key = same partition, remember?

Kafka Consumer: Java API - Basics

- Learn how to write a basic consumer to receive data from Kafka
- View basic configuration parameters
- Confirm we receive the data from the Kafka Producer written in Java



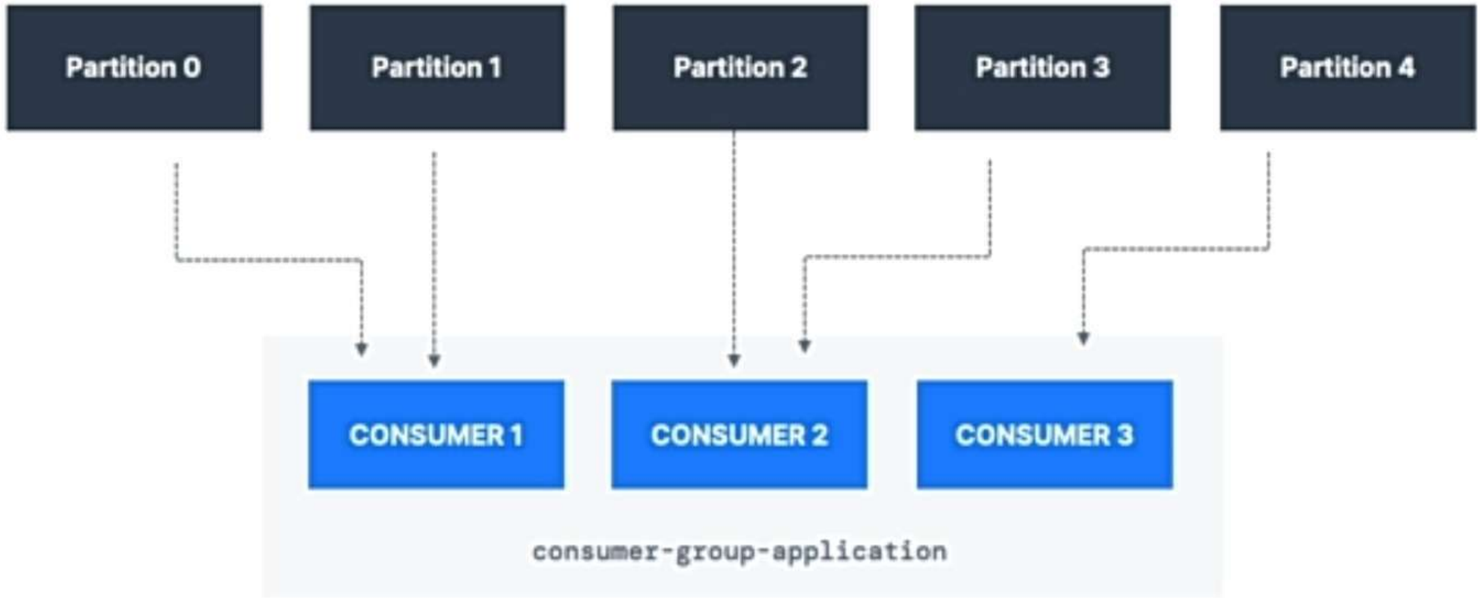
Kafka Consumer – Graceful shutdown

- Ensure we have code in place to respond to termination signals
- Improve our Java code

```
} catch (WakeupException e) {
    log.info("Wake up exception!");
    // we ignore this as this is an expected exception when closing a consum
} catch (Exception e) {
    log.error("Unexpected exception", e);
} finally {
    consumer.close(); // this will also commit the offsets if need be.
    log.info("The consumer is now gracefully closed.");
}
```

Kafka Consumer: Java API – Consumer Groups

- Make your consumer in Java consume data data as part of a consumer group
- Observe partition rebalance mechanisms



Consumer Groups and Partition Rebalance

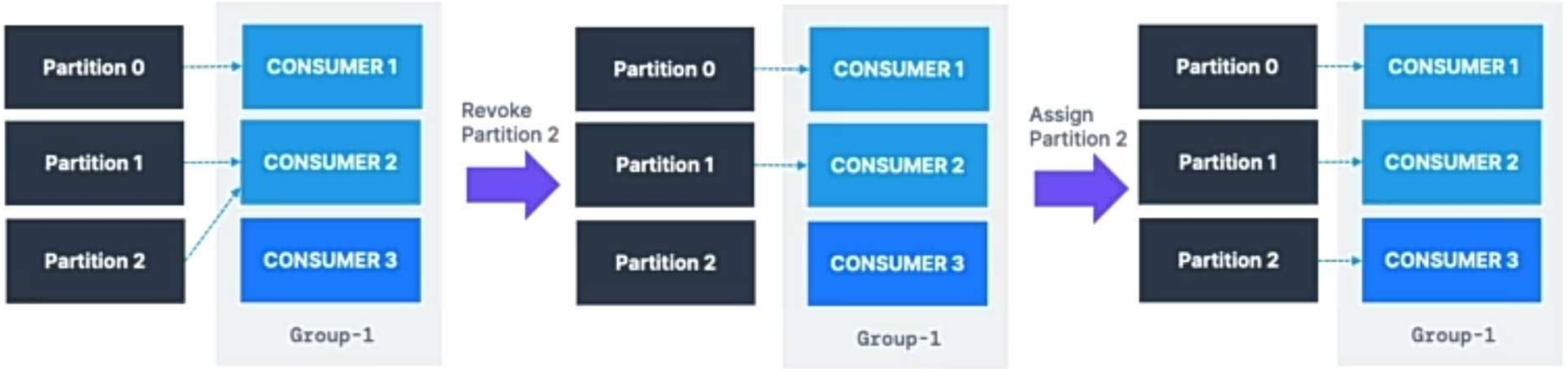
- Moving partitions between consumers is called a rebalance
- Reassignment of partitions happen when a consumer leaves or joins a group
- It also happens if an administrator adds new partitions into a topic

Eager Rebalance

- All consumers stop, give up their membership of partitions
- They rejoin the consumer group and get a new partition assignment
- During a short period of time, the entire consumer group stops processings
- Consumers don't necessarily "get back" the same partitions as they used to

Cooperative Rebalance (Incremental Rebalance)

- Reassigning a small subset of the partitions from one consumer to another
- Other consumers that don't have reassigned partitions can still process uninterrupted
- Can go through several iterations to find a "stable" assignment (hence "incremental")
- Avoids "stop-the-world" events where all consumers stop processing data

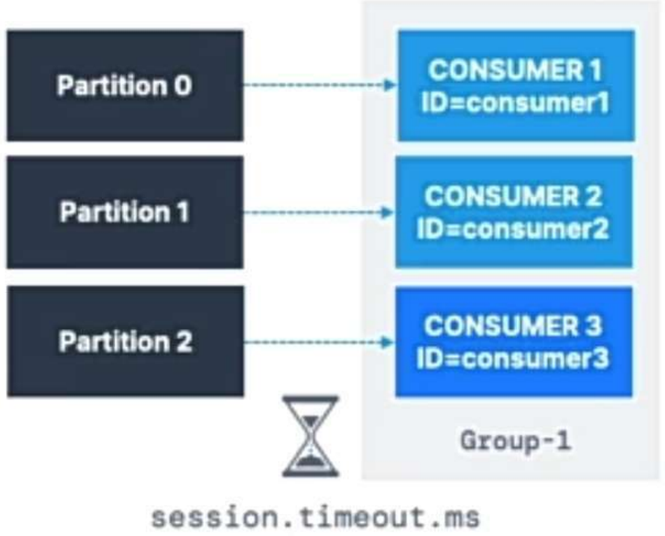


Cooperative Rebalance, how to use?

- **Kafka Consumer**: `partition.assignment.strategy`
  - RangeAssignor: assign partitions on a per-topic basis (can lead to imbalance)
  - RoundRobin: assign partitions across all topics in round-robin fashion, optimal balance
  - StickyAssignor: balanced like RoundRobin, and then minimises partition movements when consumer join / leave the group in order to minimize movements
  - CooperativeStickyAssignor: rebalance strategy is identical to StickyAssignor but supports cooperative rebalances and therefore consumers can keep on consuming from the topic
  - The default assignor is [RangeAssignor, CooperativeStickyAssignor], which will use the RangeAssignor by default, but allows upgrading to the CooperativeStickyAssignor with just a single rolling bounce that removes the RangeAssignor from the list.
- **Kafka Connect:** already implemented (enabled by default)
- **Kafka Streams**: turned on by default using StreamsPartitionAssignor

Static Group Membership

- By default, when a consume leaves a group, its partitions are revoked and re-assigned
- If it joins back, it will have a new “member ID” and new partitions assigned
- If you specify group.instance.id it makes the consumer a **static member**
- Upon leaving, the consumer has up to session.timeout.ms to join back and get back its partitions (else they will be re-assigned), without triggering a rebalance
- This is helpful when consumers maintain local state and cache (to avoid re-building the cache)

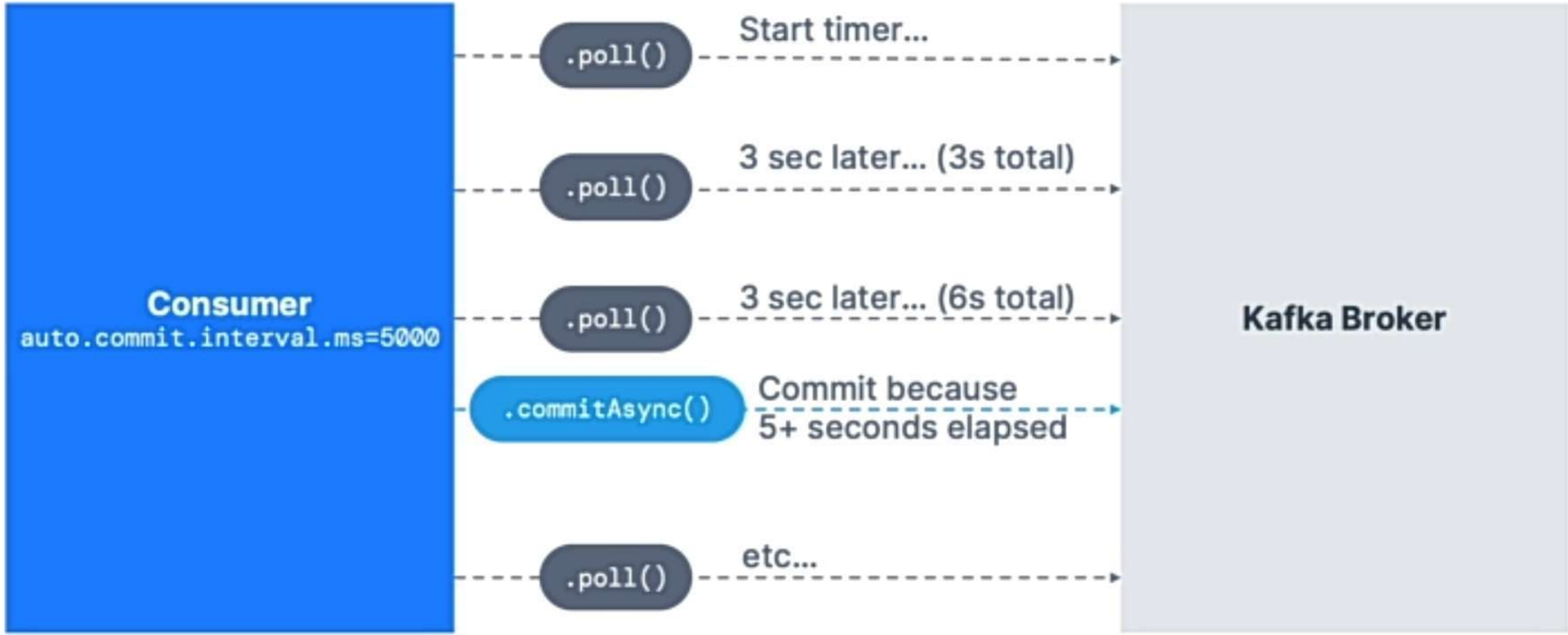


Kafka Consumer – Auto Offset Commit Behavior

- In the Java Consumer API, offsets are regularly committed
- Enable at-least once reading scenario by default (under conditions)
- Offsets are committed when you call `.poll()` and `auto.commit.interval.ms` has elapsed
- Example: `auto.commit.interval.ms=5000` and `enable.auto.commit=true` will commit

- Make sure messages are all successfully processed before you call `poll()` again
  - If you don't, you will not be in at-least-once reading scenario
  - In that (rare) case, you must disable `enable.auto.commit`, and most likely most processing to a separate thread, and then from time-to-time call `.commitSync()` or `.commitAsync()` with the correct offsets manually (advanced)

Kafka Consumer – Auto Offset Commit Behavior



Project goal: Wikimedia Stream to OpenSearch

- Wikimedia:
  - Recent change stream: https://stream.wikimedia.org/v2/stream/recentchange
  - Demo at: https://codepen.io/Krinkle/pen/BwEKgW?editors=1010
  - Demo at: https://esjewett.github.io/wm-eventsource-demo/

- Try implementing this on your own
- The next text lecture will include all the libraries and tips you'll need to be using

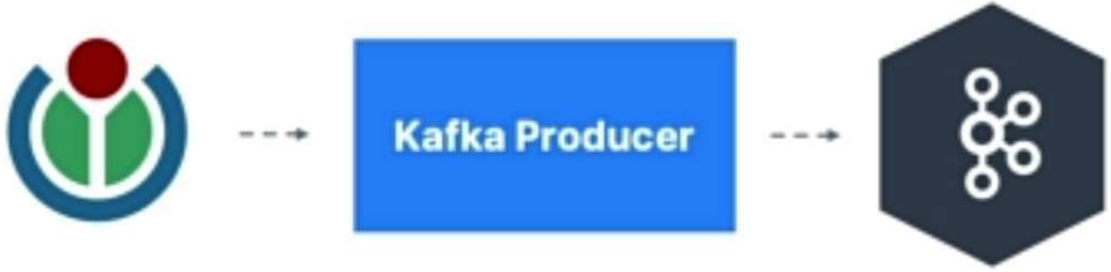
Teaser: next sections

- After implementing this with programming, we'll run the following architecture with advanced concepts (Kafka Connect, Kafka Streams) in the following sections

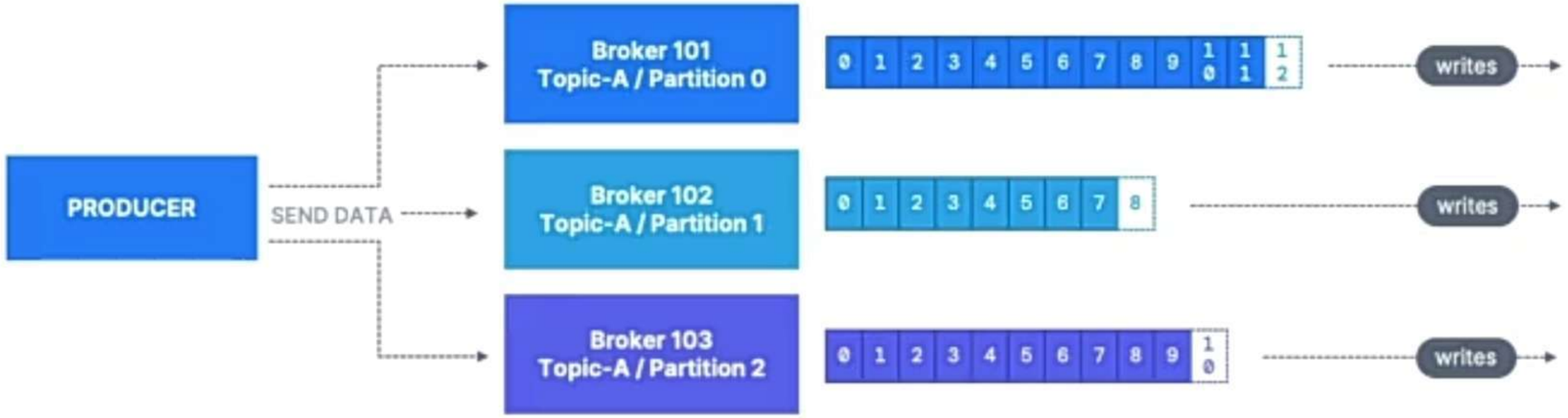
Wikimedia Producer – Project Setup

- Recent change stream: https://stream.wikimedia.org/v2/stream/recentchange
- Demo at: https://codepen.io/Krinkle/pen/BwEKgW?editors=1010
- Demo at: https://esjewett.github.io/wm-eventsource-demo/

- Use Java libraries:
  - OKhttp3
  - Okhttp-eventsource



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

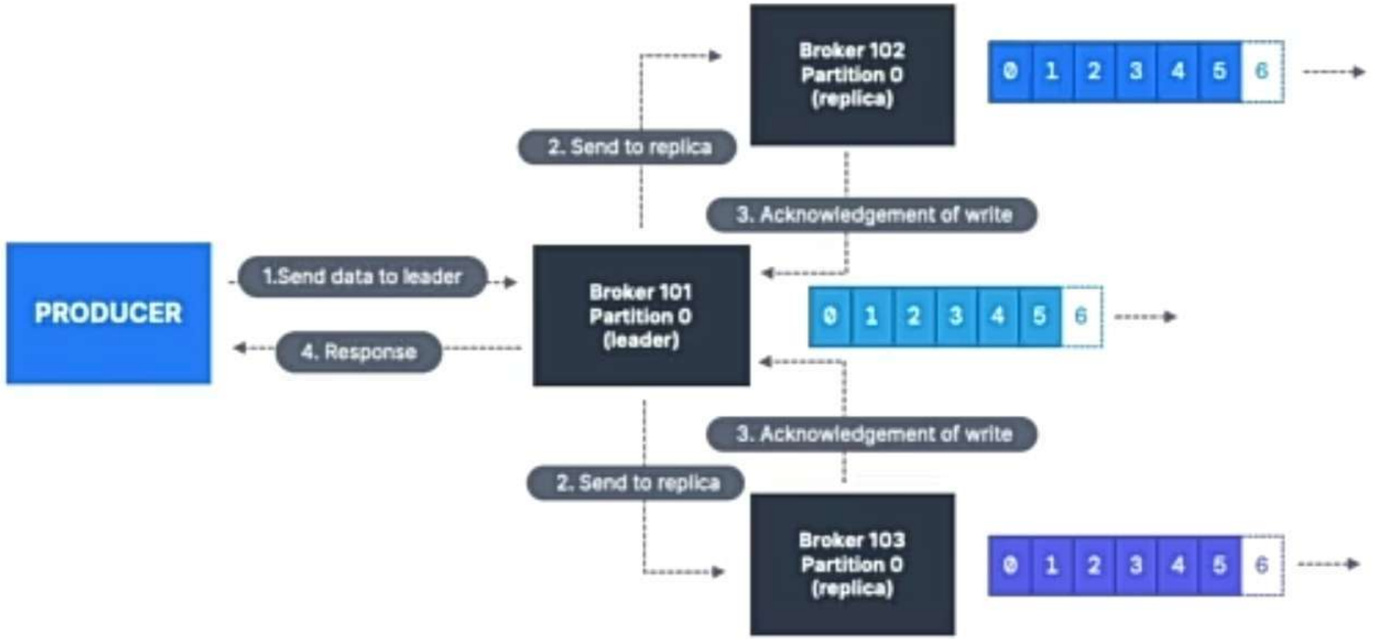
Producer acks=1

- 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

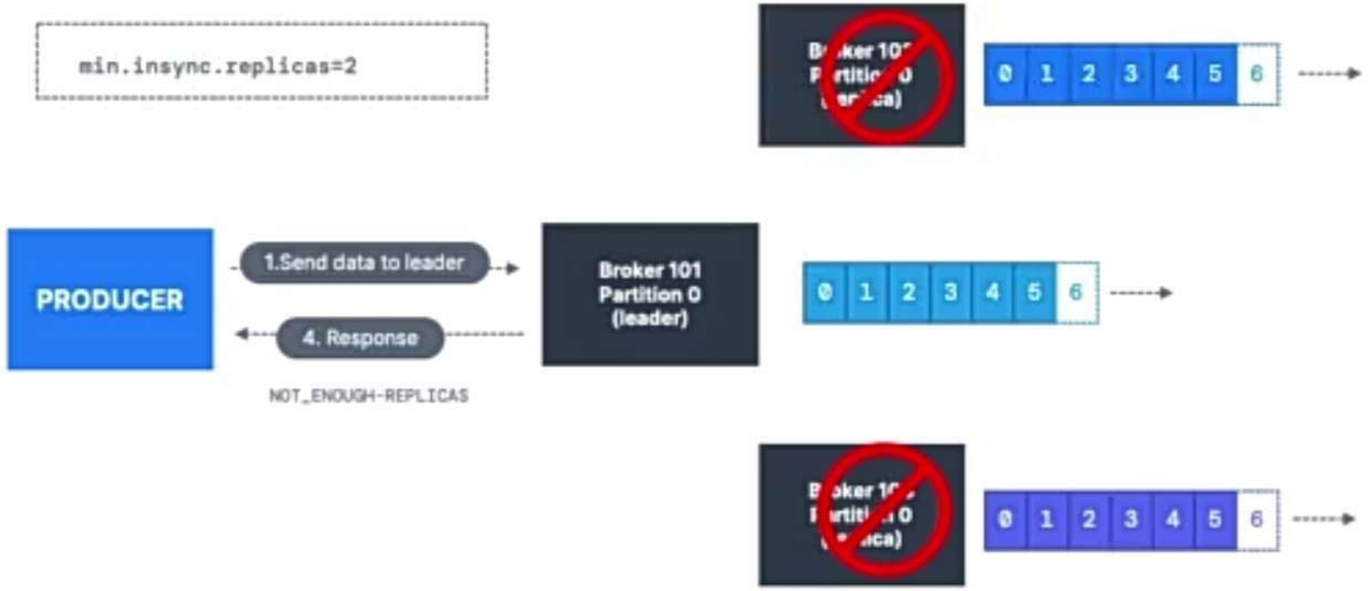
Producer `acks=all` (`acks=-1`)

- 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+



Producer `acks=all` & `min.insync.replicas`

- The leader replica for a partition checks to see if there are enough in-sync replicas for safely writing the message (controlled by the broker setting `min.insync.replicas`)
  - `min.insync.replicas=1`: only the broker leader needs to successfully ack
  - `min.insync.replicas=2`: at least the broker leader and one replica need to ack



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

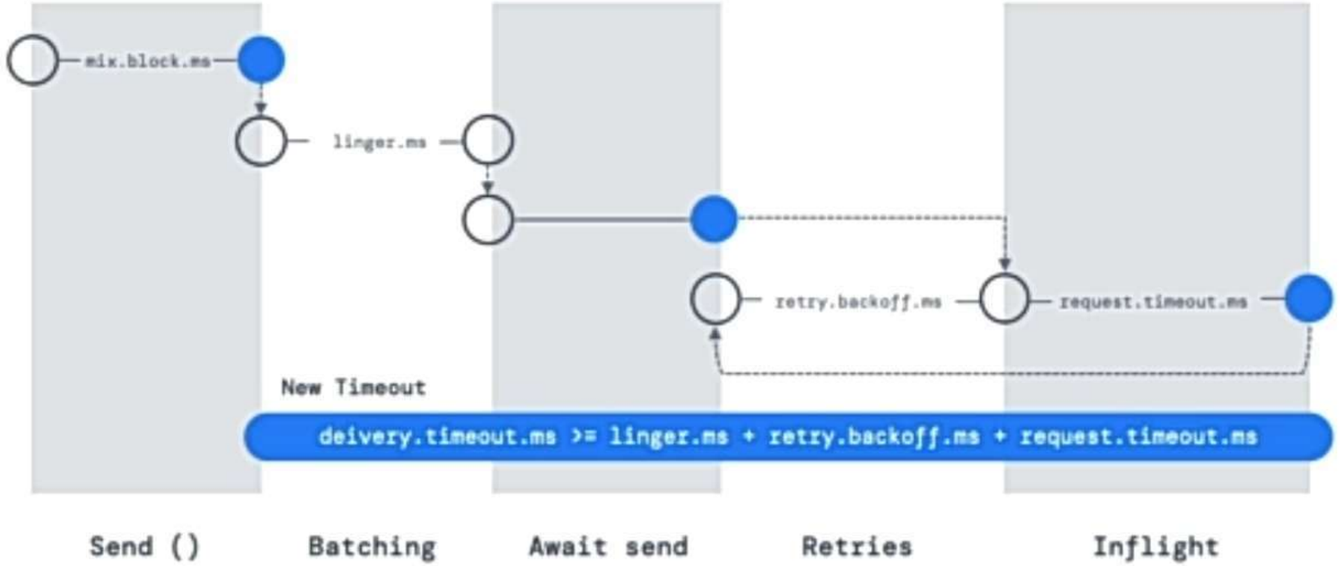
- `acks=all` and `min.insync.replicas=2` is the most popular option for data durability and availability and allows you to withstand at most the loss of **one** Kafka broker

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

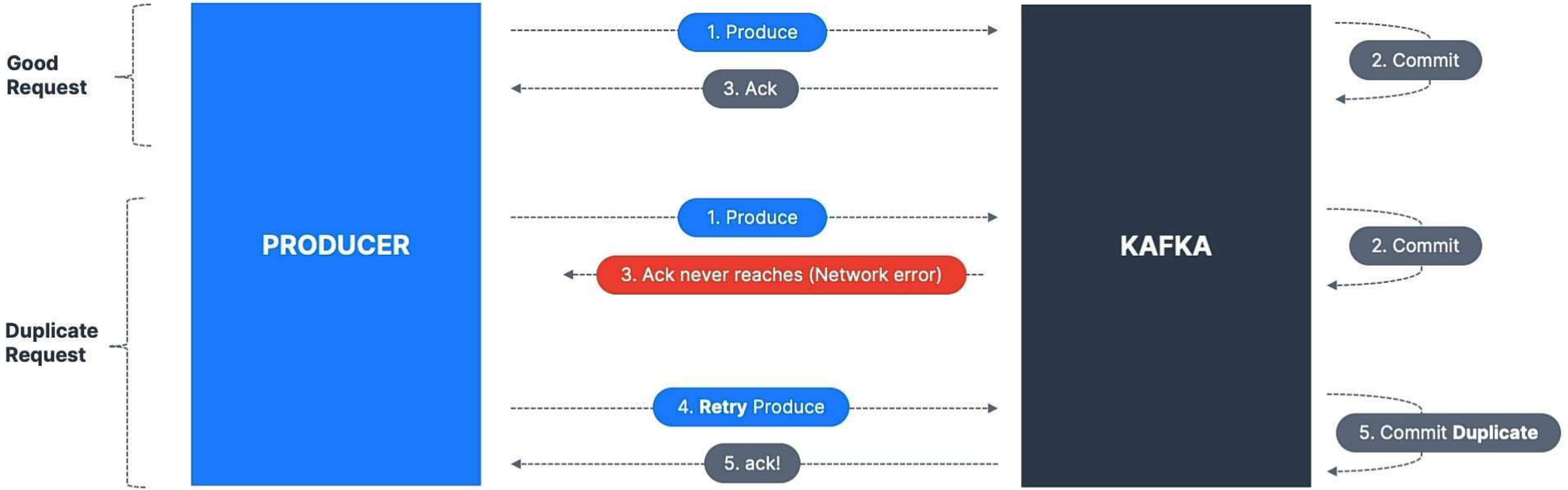
- If you are not using an idempotent producer (not recommended – old Kafka):
  - In case of retries, there is a chance that messages will be sent out of order (if a batch has failed to be sent).
  - If you rely on key-based ordering, that can be an issue.

- 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!

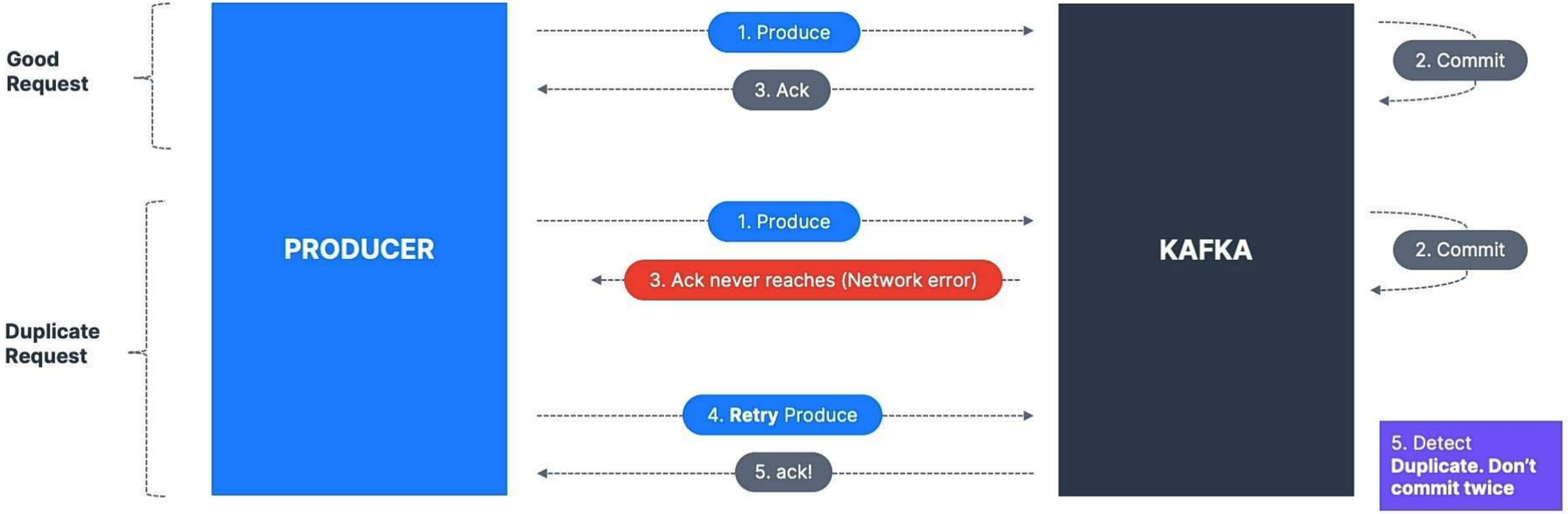
Idempotent Producer

- 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

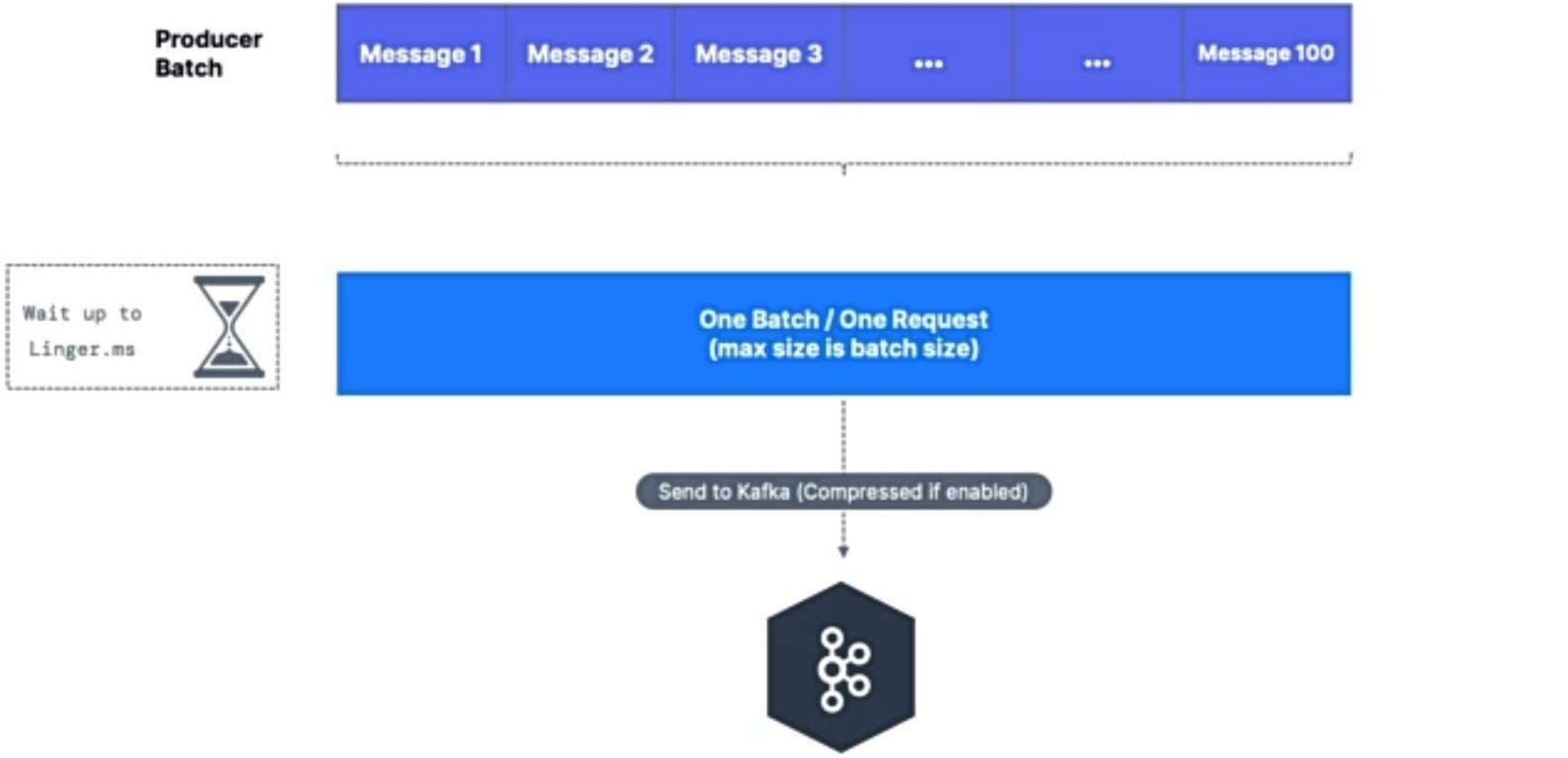
- Warning: if you enable broker-side compression, it will consume extra CPU cycles

`linger.ms` & `batch.size`

- By default, Kafka producers try to send records as soon as possible
  - It will have up to `max.in.flight.requests.per.connection=5`, meaning up to 5 message batches being in flight (being sent between the producer in the broker) at most
  - After this, if more messages must be sent while others are in flight, Kafka is smart and will start batching them before the next batch send
- This smart batching helps increase throughput while maintaining very low latency
  - Added benefit: batches have higher compression ratio so better efficiency

- Two settings to influence the batching mechanism
  - `linger.ms`: (default 0) how long to wait until we send a batch. Adding a small number for example 5 ms helps add more messages in the batch at the expense of latency
  - `batch.size`: if a batch is filled before before `linger.ms`, increase the batch size

linger.ms & batch.size



batch.size (default 16KB)

- Maximum number of bytes that will be included in a batch

- Increasing a batch size to something like 32KB or 64KB can help increasing the compression, throughput, and efficiency of requests
- Any message that is bigger than the batch size will not be batched
- A batch is allocated per partition, so make sure that you don't set it to a number that's too high, otherwise you'll run waste memory!
- (Note: You can monitor the average batch size metric using Kafka Producer Metrics)

High Throughput Producer

- Increase `linger.ms` and the producer will wait a few milliseconds for the batches to fill up before sending them.
- If you are sending full batches and have memory to spare, you can increase `batch.size` and send larger batches
- Introduce some producer-level compression for more efficiency in sends

```
// high throughput producer (at the expense of a bit of latency and CPU usage)
properties.setProperty(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy");
properties.setProperty(ProducerConfig.LINGER_MS_CONFIG, "20");
properties.setProperty(ProducerConfig.BATCH_SIZE_CONFIG, Integer.toString(32*1024));
```

High Throughput Producer - Demo

- We'll add `snappy` message compression in our producer
- `snappy` is very helpful if your messages are text based, for example log lines or JSON documents
- `snappy` has a good balance of CPU / compression ratio
- We'll also increase the `batch.size` to 32KB and introduce a small delay through `linger.ms` (20 ms)
- We'll check which partitioner is being used

```
// high throughput producer (at the expense of a bit of latency and CPU usage)
properties.setProperty(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy");
properties.setProperty(ProducerConfig.LINGER_MS_CONFIG, "20");
properties.setProperty(ProducerConfig.BATCH_SIZE_CONFIG, Integer.toString(32*1024));
```

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

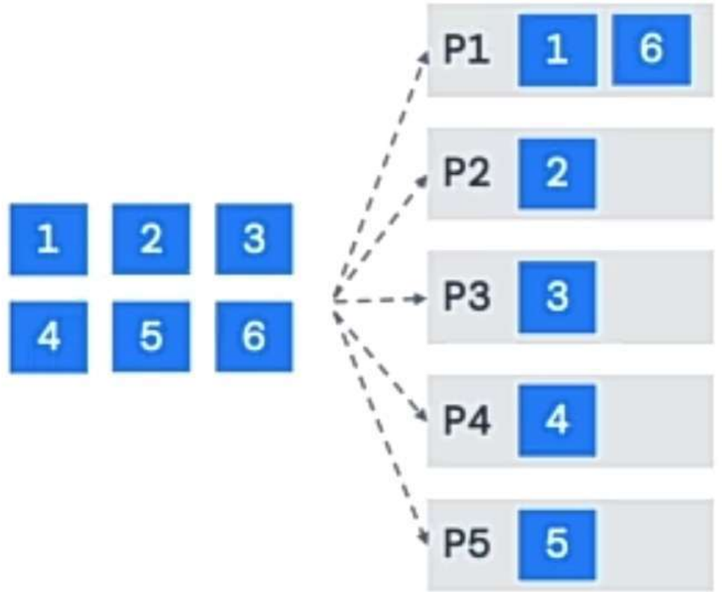
Producer Default Partitioner when `key=null`

- 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

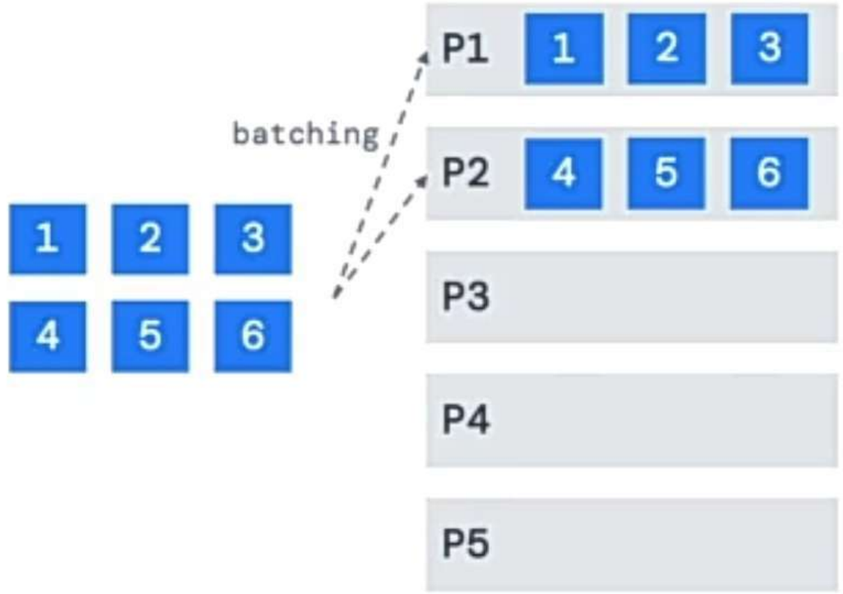
Producer Default Partitioner Kafka ≤ v2.3 Round Robin Partitioner

- 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

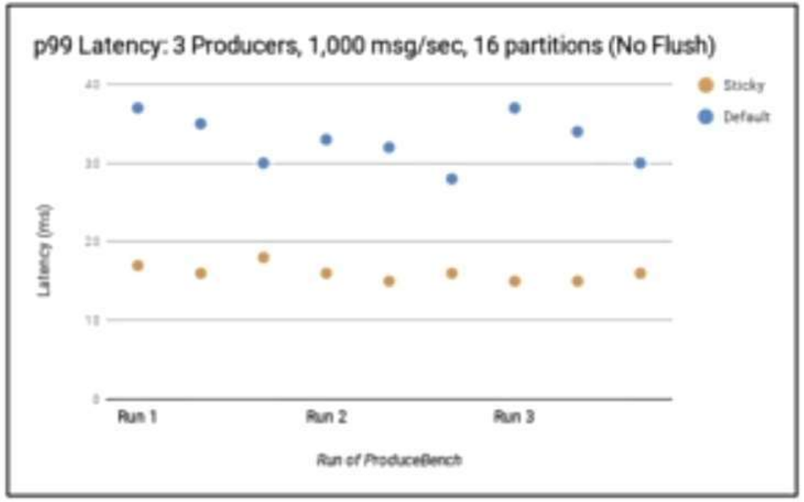
- It would be better to have all the records sent to a single partition and not multiple partitions to improve batching
- The producer sticky partitioner:
  - We "stick" to a partition until the batch is full or `linger.ms` has elapsed
  - After sending the batch, the partition that is sticky changes
- 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



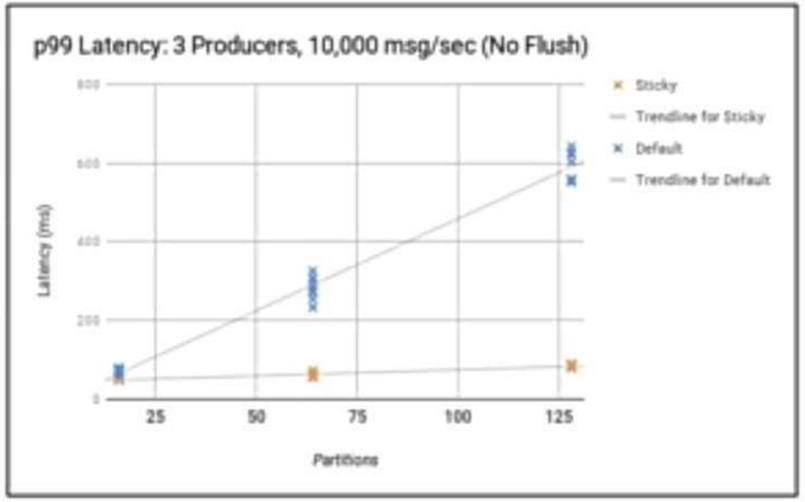
Sticky Partitioner
(performance improvement)

Sticky Partitioner - Performance Improvement

Latency is noticeably lower



Latency is noticeably lower the more partitions



https://cwiki.apache.org/confluence/display/KAFKA/KIP-480%3A+Sticky+Partitioner

max.block.ms & buffer.memory

- If the producer produces faster than the broker can take, the records will be buffered in memory
- **buffer.memory=33554432 (32MB):** the size of the send buffer

- That buffer will fill up over time and empty back down when the throughput to the broker increases

max.block.ms & buffer.memory

- If that buffer is full (all 32MB), then the `.send()` method will start to block (won't return right away)

- **`max.block.ms=60000`**: the time the `.send()` will block until throwing an exception. Exceptions are thrown when:
  - The producer has filled up its buffer
  - The broker is not accepting any new data
  - 60 seconds has elapsed

- If you hit an exception hit that usually means your brokers are down or overloaded as they can't respond to requests

Kafka to OpenSearch Consumer – Project Setup

- Setup an OpenSearch (open-source fork of ElasticSearch) cluster:
  - Free tier / managed OpenSearch available at bonsai.io
  - Run locally using Docker for example

- Use Java libraries:
  - OpenSearch REST High Level Client

OpenSearch 101 (e.g. ElasticSearch)

- Use the REST API (using OpenSearch Dashboards) or online console at bonsai.io

- Tutorial at: https://opensearch.org/docs/latest/#docker-quickstart (version 1.X)

Delivery Semantics - At Most Once

- At most once: offsets are committed as soon as the message batch is received. If the processing goes wrong, the message will be lost (it won't be read again).

Delivery Semantics - At Least Once

- At least once: 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)

Delivery semantics for consumers - Summary

- At most once: offsets are committed as soon as the message is received. If the processing goes wrong, the message will be lost (it won't be read again).
- At least once (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)
- Exactly once: Can be achieved for Kafka => Kafka workflows using the Transactional API (easy with Kafka Streams API). For Kafka => Sink workflows, use an idempotent consumer.

Bottom line: for most applications you should use at least once processing (we'll see in practice how to do it) and ensure your transformations / processing are idempotent

Consumer Offset Commit Strategies

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

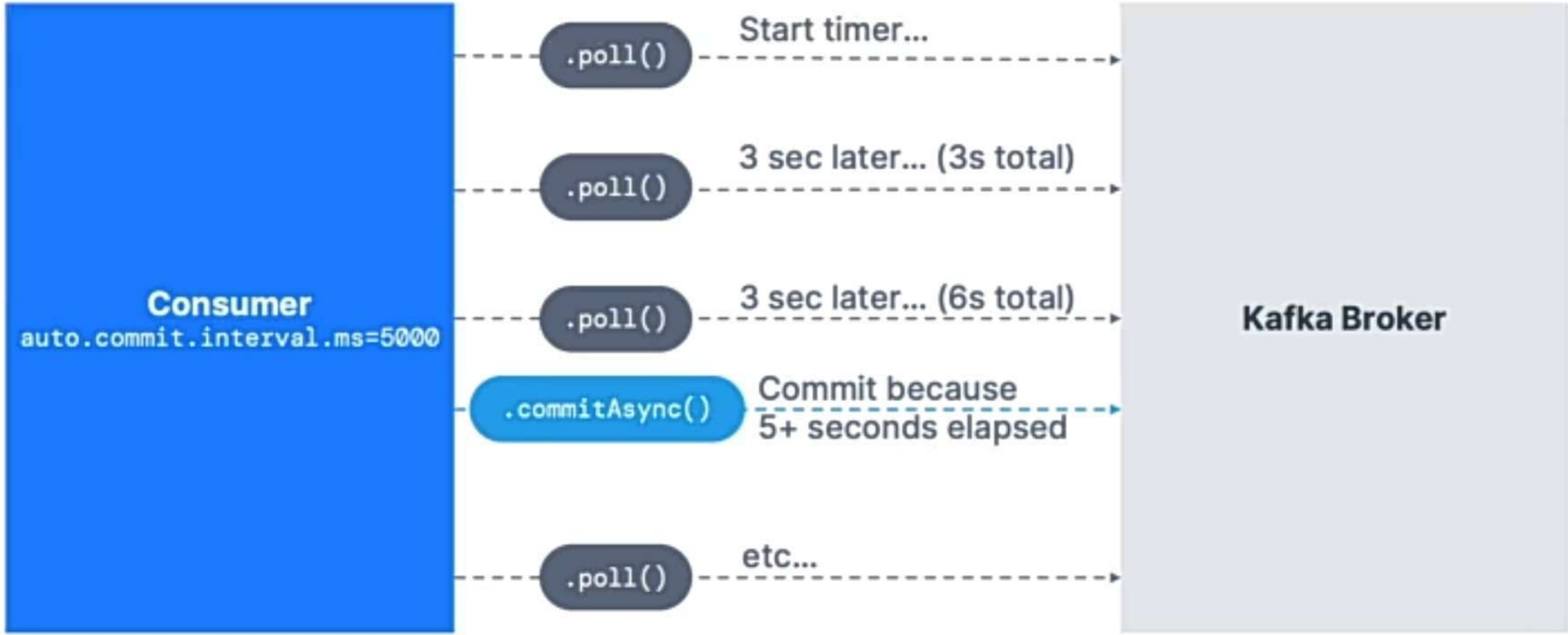
- **2 strategies:**
  - (easy) enable.auto.commit = true & synchronous processing of batches
  - (medium) enable.auto.commit = false & manual commit of offsets

Kafka Consumer – Auto Offset Commit Behavior

- In the Java Consumer API, offsets are regularly committed
- Enable at-least once reading scenario by default (under conditions)
- Offsets are committed when you call `.poll()` and `auto.commit.interval.ms` has elapsed
- Example: `auto.commit.interval.ms=5000` and `enable.auto.commit=true` will commit

- Make sure messages are all successfully processed before you call `poll()` again
  - If you don't, you will not be in at-least-once reading scenario
  - In that (rare) case, you must disable `enable.auto.commit`, and most likely most processing to a separate thread, and then from time-to-time call `.commitSync()` or `.commitAsync()` with the correct offsets manually (advanced)

Kafka Consumer – Auto Offset Commit Behavior



Consumer Offset Commits Strategies

- `enable.auto.commit=true` & synchronous processing of batches

```
while(true){
    List<Records> batch = consumer.poll(Duration.ofMillis(100))
    doSomethingSynchronous(batch)
}
```

- With auto-commit, offsets will be committed automatically for you at regular interval **(`auto.commit.interval.ms=5000` by default)** every-time you call `.poll()`
- If you don't use synchronous processing, you will be in "at-most-once" behavior because offsets will be committed before your data is processed

Consumer Offset Commits Strategies

- `enable.auto.commit=false` & synchronous processing of batches

```
while(true){
    batch += consumer.poll(Duration.ofMillis(100))
    if isReady(batch) {
        doSomethingSynchronous(batch)
        consumer.commitAsync();
    }
}
```

- You control when you commit offsets and what's the condition for committing them.
- *Example: accumulating records into a buffer and then flushing the buffer to a database + committing offsets asynchronously then.*

Consumer Offset Commits Strategies

- `enable.auto.commit=false` & storing offsets externally

- **This is advanced:**
  - You need to assign partitions to your consumers at launch manually using `.seek()` API
  - You need to model and store your offsets in a database table for example
  - You need to handle the cases where rebalances happen (`ConsumerRebalanceListener` interface)

- *Example: if you need exactly once processing and can't find any way to do idempotent processing, then you "process data" + "commit offsets" as part of a single transaction.*
- *Note: I don't recommend using this strategy unless you exactly know what and why you're doing it*

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)

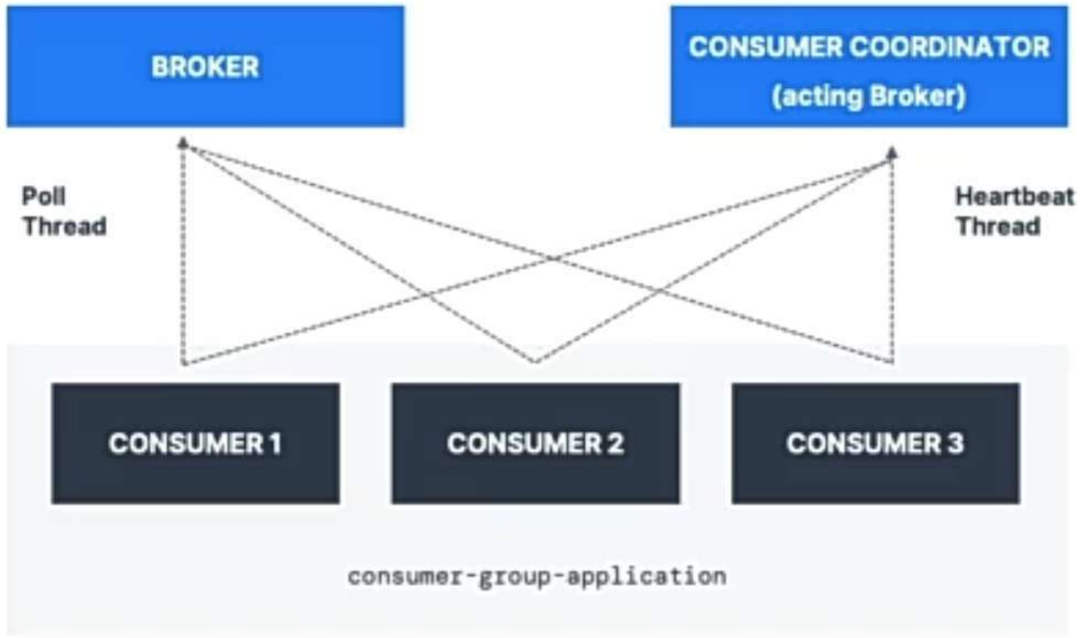
- This can be controlled by the broker setting `offset.retention.minutes`

Replaying data for Consumers

- To replay data for a consumer group:
  - Take all the consumers from a specific group down
  - Use `kafka-consumer-groups` command to set offset to what you want
  - Restart consumers

- Bottom line:
  - Set proper data retention period & offset retention period
  - Ensure the auto offset reset behavior is the one you expect / want
  - Use replay capability in case of unexpected behavior

Controlling Consumer Liveliness



- Consumers in a Group talk to a Consumer Groups Coordinator
- To detect consumers that are “down”, there is a “heartbeat” mechanism and a ”poll” mechanism
- To avoid issues, consumers are encouraged to process data fast and poll often

Consumer Heartbeat Thread

- `heartbeat.interval.ms` **(default 3 seconds):**
  - How often to send heartbeats
  - Usually set to 1/3rd of `session.timeout.ms`

- `session.timeout.ms` **(default 45 seconds Kafka 3.0+, before 10 seconds):**
  - Heartbeats are sent periodically to the broker
  - If no heartbeat is sent during that period, the consumer is considered dead
  - Set even lower to faster consumer rebalances

- Take-away: This mechanism is used to detect a consumer application being down

Consumer Poll Thread

- **`max.poll.interval.ms` (default 5 minutes):**
  - Maximum amount of time between two `.poll()` calls before declaring the consumer dead
  - This is relevant for Big Data frameworks like Spark in case the processing takes time

- Take-away: This mechanism is used to detect a data processing issue with the consumer (consumer is "stuck")

- **`max.poll.records` (default 500):**
  - Controls how many records to receive per poll request
  - Increase if your messages are very small and have a lot of available RAM
  - Good to monitor how many records are polled per request
  - Lower if it takes you too much time to process records

Consumer Poll Behavior

- `fetch.min.bytes` **(default 1):**
  - Controls how much data you want to pull at least on each request
  - Helps improving throughput and decreasing request number
  - At the cost of latency

- `fetch.max.wait.ms` **(default 500):**
  - The maximum amount of time the Kafka broker will block before answering the fetch request if there isn't sufficient data to immediately satisfy the requirement given by fetch.min.bytes
  - This means that until the requirement of fetch.min.bytes to be satisfied, you will have up to 500 ms of latency before the fetch returns data to the consumer (e.g. introducing a potential delay to be more efficient in requests)

Consumer Poll Behavior

- `max.partition.fetch.bytes` **(default 1MB):**
  - The maximum amount of data per partition the server will return
  - If you read from 100 partitions, you'll need a lot of memory (RAM)

- `fetch.max.bytes` **(default 55MB):**
  - Maximum data returned for each fetch request
  - If you have available memory, try increasing fetch.max.bytes to allow the consumer to read more data in each request.

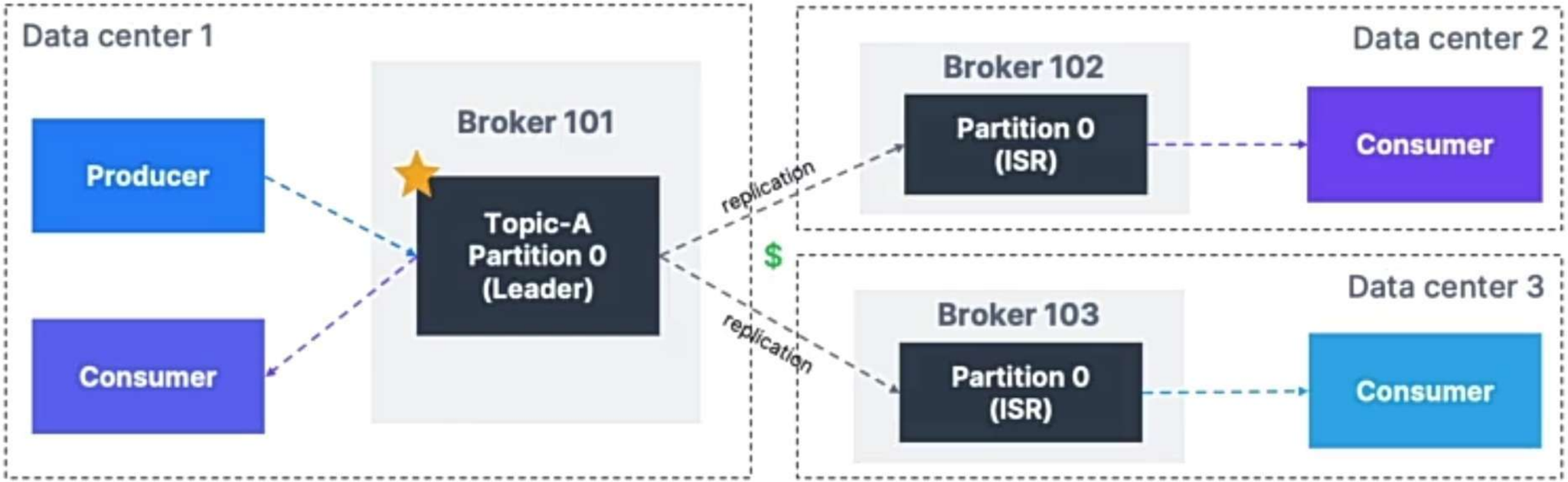
- Advanced: Change these settings only if your consumer maxes out on throughput already

Default Consumer behavior with partition leaders

- Kafka Consumers by default will read from the leader broker for a partition
- Possibly higher latency (multiple data centre), + high network charges ($$$)
- Example: Data Centre === Availability Zone (AZ) in AWS, you pay for Cross AZ network charges

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



Consumer Rack Awareness (v2.4+) - How to Setup

- Broker setting:
  - Must be version Kafka v2.4+
  - `rack.id` config must be set to the data centre ID (ex: AZ ID in AWS)
  - Example for AWS: AZ ID `rack.id=usw2-az1`
  - `replica.selector.class` must be set to `org.apache.kafka.common.replica.RackAwareReplicaSelector`
- Consumer client setting:
  - Set `client.rack` to the data centre ID the consumer is launched on

What will we see in this section?

- Kafka Consumers and Producers have existed for a long time, and they're considered **low-level**

- Kafka & ecosystem has introduced over time some new API that are higher level that solves specific sets of problems:
  - Kafka Connect solves External Source => Kafka and Kafka => External Sink
  - Kafka Streams solves transformations Kafka => Kafka
  - Schema Registry helps using Schema in Kafka

- This is an introduction meant to give you an idea of the depth of the Kafka Ecosystem

Kafka Connect Introduction

- Do you feel you're not the first person in the world to write a way to get data out of Twitter?
- Do you feel like you're not the first person in the world to send data from Kafka to PostgreSQL / ElasticSearch / MongoDB ?
- Additionally, the bugs you'll have, won't someone have fixed them already?

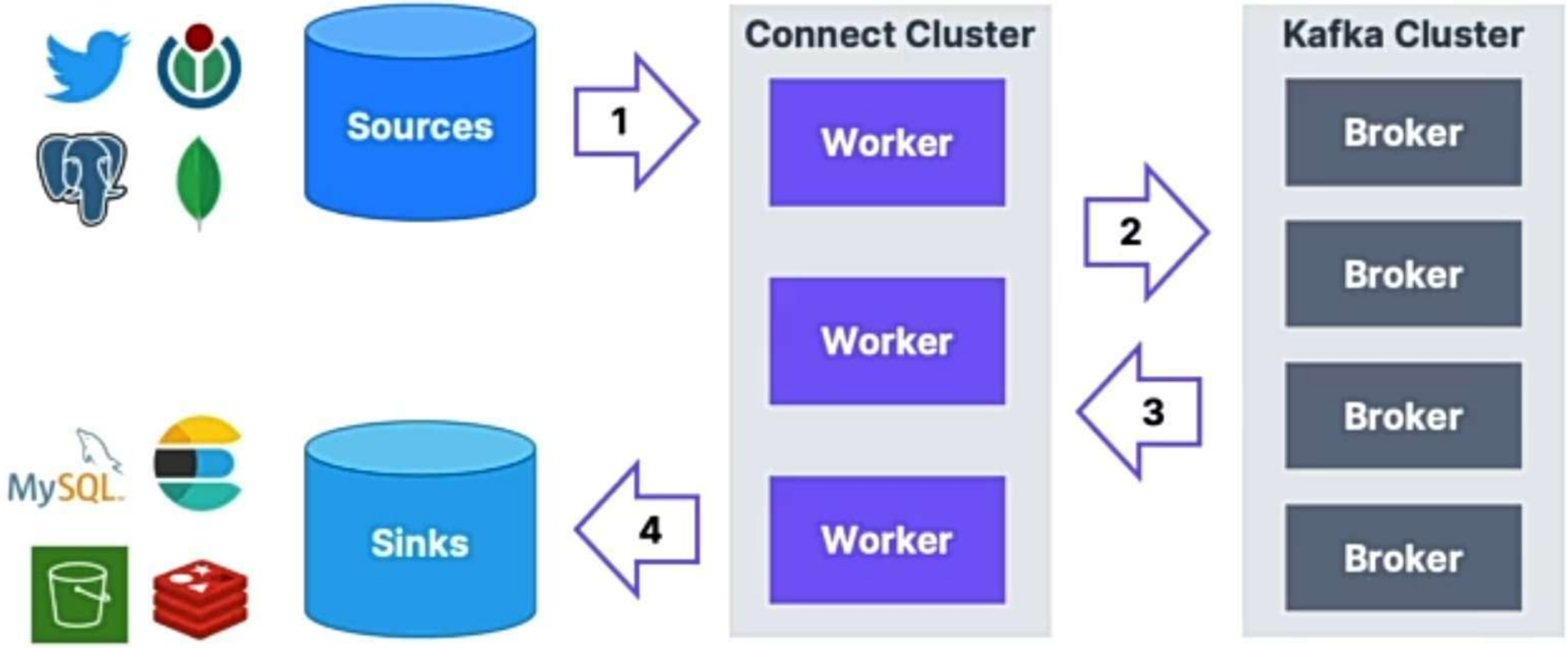
Kafka Connect Introduction

- Do you feel you're not the first person in the world to write a way to get data out of Twitter?
- Do you feel like you're not the first person in the world to send data from Kafka to PostgreSQL / ElasticSearch / MongoDB ?
- Additionally, the bugs you'll have, won't someone have fixed them already?
- Kafka Connect is all about code & connectors re-use!

Why Kafka Connect

- Programmers always want to import data from the same sources:
  - Databases, JDBC, Couchbase, GoldenGate, SAP HANA, Blockchain, Cassandra, DynamoDB, FTP, IOT, MongoDB, MQTT, RethinkDB, Salesforce, Solr, SQS, Twitter…
- Programmers always want to store data in the same sinks:
  - S3, ElasticSearch, HDFS, JDBC, SAP HANA, DocumentDB, Cassandra, DynamoDB, HBase, MongoDB, Redis, Solr, Splunk, Twitter…

Kafka Connect - Architecture Design

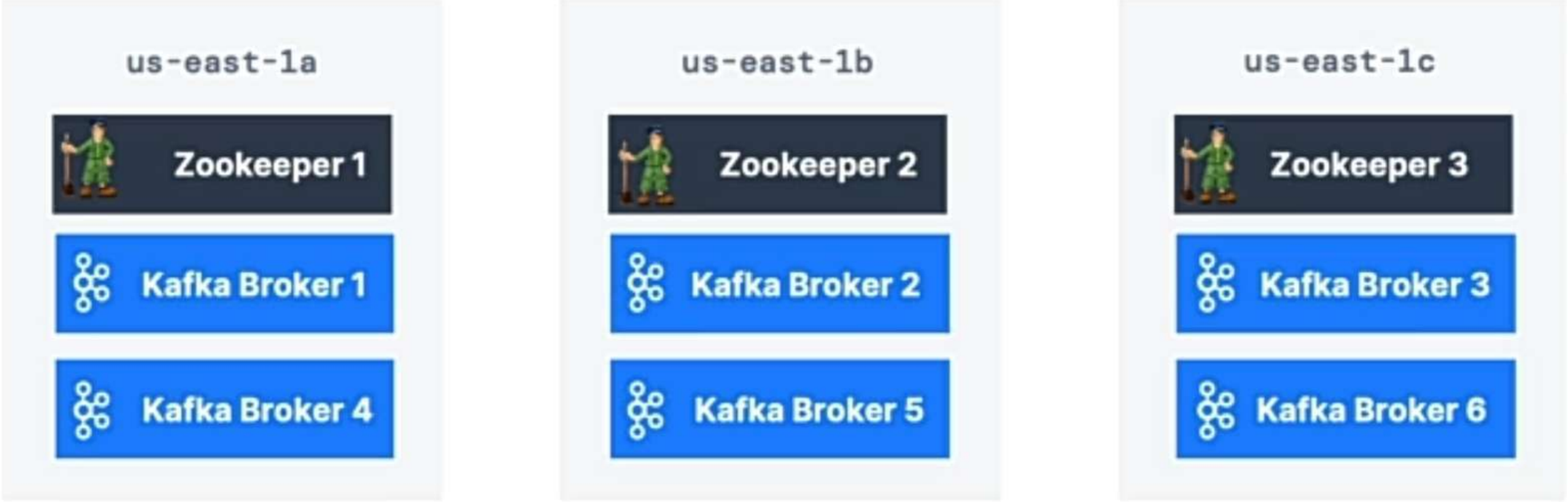


Kafka Connect – High level

- **Source Connectors** to get data from Common Data Sources
- **Sink Connectors** to publish that data in Common Data Stores
- Make it easy for non-experienced dev to quickly get their data reliably into Kafka
- Part of your ETL pipeline
- Scaling made easy from small pipelines to company-wide pipelines
- Other programmers may already have done a very good job: re-usable code!
- Connectors achieve fault tolerance, idempotence, distribution, ordering

Kafka Cluster Setup - High Level Architecture

- You want multiple brokers in different data centers (racks) to distribute your load. You also want a cluster of at least 3 Zookeeper (if using Zookeeper)
- In AWS:



Kafka Cluster Setup Gotchas

- It's not easy to setup a cluster
- You want to isolate each Zookeeper & Broker on separate servers
- Monitoring needs to be implemented
- Operations must be mastered
- You need an excellent Kafka Admin
- Alternative: managed "Kafka as a Service" offerings from various companies
  - Amazon MSK, Confluent Cloud, Aiven, CloudKarafka, Instaclustr, Upstash, etc...)
  - No operational burdens (updates, monitoring, setup, etc...)
- How many brokers?
  - Compute your throughput, data retention, and replication factor
  - Then test for your use case

Other components to properly setup

- Kafka Connect Clusters
- Kafka Schema Registry: make sure to run two for high availability
- UI tools for ease of administration
- Admin Tools for automated workflows

- Automate as much as you can your infrastructure when you've understood how processes work!

Kafka Monitoring and Operations

- Kafka exposes metrics through JMX.
- These metrics are highly important for monitoring Kafka, and ensuring the systems are behaving correctly under load.
- Common places to host the Kafka metrics:
  - ELK (ElasticSearch + Kibana)
  - Datadog
  - NewRelic
  - Confluent Control Centre
  - Promotheus
  - Many others…!

Kafka Monitoring

- Some of the most important metrics are:

- Under Replicated Partitions: Number of partitions are have problems with the ISR (in-sync replicas). May indicate a high load on the system

- Request Handlers: utilization of threads for IO, network, etc… overall utilization of an Apache Kafka broker.

- Request timing: how long it takes to reply to requests. Lower is better, as latency will be improved.

Kafka Operations

- Kafka Operations team must be able to perform the following tasks:
  - Rolling Restart of Brokers
  - Updating Configurations
  - Rebalancing Partitions
  - Increasing replication factor
  - Adding a Broker
  - Replacing a Broker
  - Removing a Broker
  - Upgrading a Kafka Cluster with zero downtime

- It is important to remember that managing your own cluster comes with all these responsibilities and more.

The need for security in Apache Kafka

- Currently, any client can access your Kafka cluster (authentication)
- The clients can publish / consume any topic data (authorisation)
- All the data being sent is fully visible on the network (encryption)

- Someone could intercept data being sent
- Someone could publish bad data / steal data
- Someone could delete topics

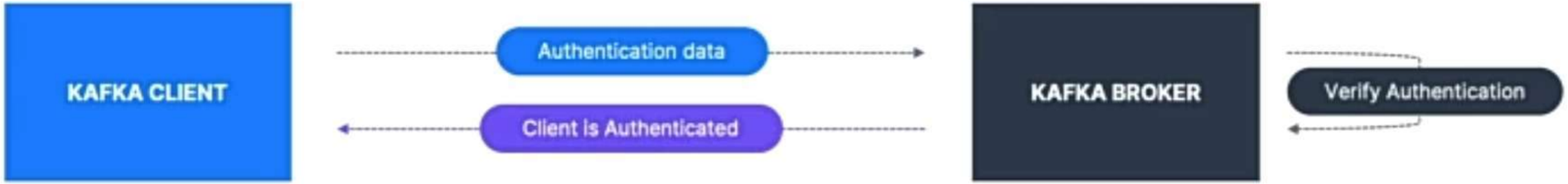
- All these reasons push for more security and an authentication model

In-flight encryption in Kafka

- Encryption in Kafka ensures that the data exchanged between clients and brokers is secret to routers on the way
- This is similar concept to an https website
- Performance improvement but negligible if using Java JDK 11

Authentication (SSL & SASL) in Kafka

- Authentication in Kafka ensures that only clients that can prove their identity can connect to our Kafka Cluster
- This is similar concept to a login (username / password)



Authentication (SSL & SASL) in Kafka

- **SSL Authentication**: clients authenticate to Kafka using SSL certificates
- **SASL/PLAINTEXT**
  - clients authenticate using username / password (weak – easy to setup)
  - Must enable SSL encryption broker-side as well
  - Changes in passwords require brokers reboot (good for dev only)
- **SASL/SCRAM**
  - Username/password with a challenge (salt), more secure
  - Must enable SSL encryption broker-side as well
  - Authentication data is in Zookeeper (until removed) – add without restarting brokers
- **SASL/GSSAPI (Kerberos)**
  - Kerberos: such as Microsoft Active Directory (strong – hard to setup)
  - Very secure and enterprise friendly
- **SASL/OAUTHBEARER**
  - Leverage OAUTH2 tokens for authentication.

Kafka Security - Putting it all together

- You can mix Encryption, Authentication & Authorisation
- This allows your Kafka clients to:
  - Communicate securely to Kafka
  - Clients would authenticate against Kafka
  - Kafka can authorise clients to read / write to topics
- It's hard to setup security in Kafka and requires significant understanding of security

```
security.protocol=SASL_SSL
sasl.kerberos.service.name=kafka
sasl.jaas.config=com.sun.security.auth.module.Krb5LoginModule required useKeyTab=true
storeKey=true keyTab="/etc/security/keytabs/kafka_client.keytab"
principal="kafkaclient1@EXAMPLE.COM";
ssl.truststore.location=/home/gerd/ssl/kafka.client.truststore.jks
ssl.truststore.password=clientpass
```

+

```
kafka-consumer-groups --bootstrap-server host.server.com:9092 \
--describe \
--command-config client.properties
```

- Best support for Kafka Security is with the Java clients although other clients based on librdkafka have good support for security now

Kafka Multi Cluster & Replication

- Kafka can only operate well in a single region
- Therefore, it is very common for enterprises to have Kafka clusters across the world, with some level of replication between them

Kafka Multi Cluster & Replication

- A replication application at its core is just a consumer + a producer

- There are different tools to perform it:
  - Mirror Maker 2 – open-source Kafka Connector connector that ships with Kafka
  - Netflix uses Flink – they wrote their own application
  - Uber uses uReplicator – addresses performance and operations issues with MM
  - Comcast has their own open-source Kafka Connect Source
  - Confluent has their own Kafka Connect Source (paid)

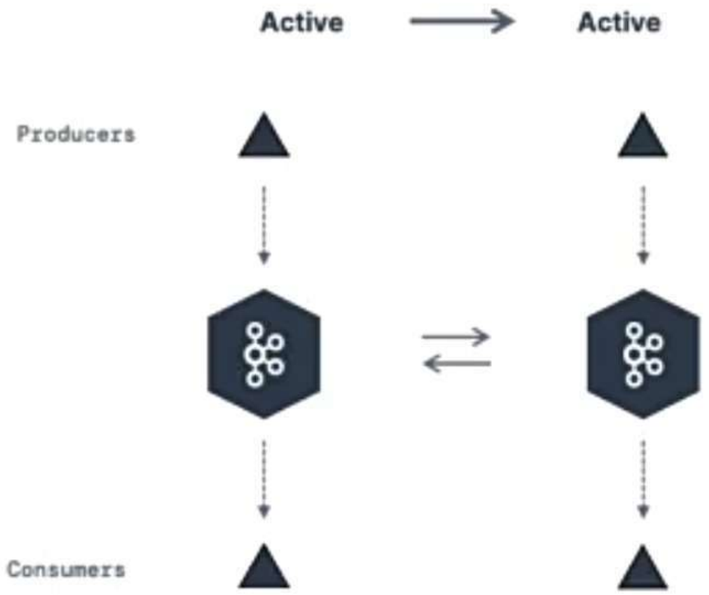
- Overall, try these and see if it works for your use case before writing your own

- **Replicating doesn't preserve offsets, just data! Data at an offset in one cluster is not the same as the data at same offset in another cluster.**

Kafka Multi Cluster & Replication – Active / Active

- **Advantages**
  - The advantages of this architecture are:
  - Ability to serve users from a nearby data center, which typically has performance benefits
  - Redundancy and resilience. Since every data center has all the functionality, if one data center is unavailable you can direct users to a remaining data center.

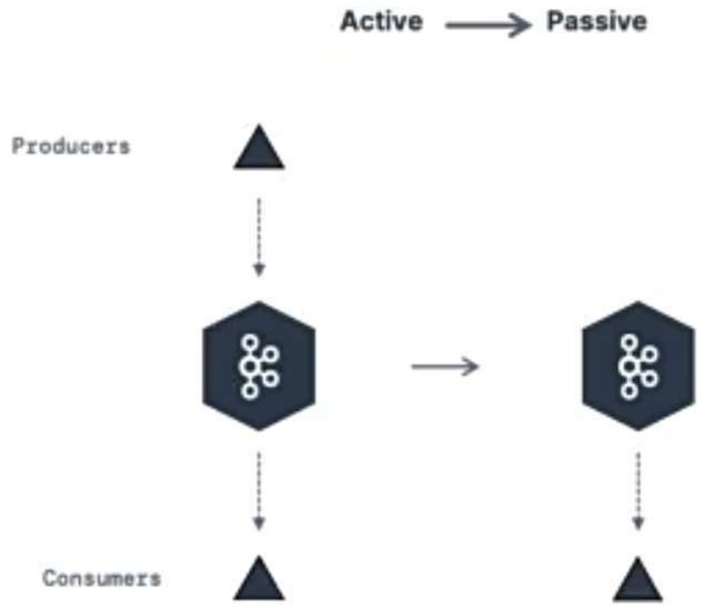
- **Disadvantages**
  - The main drawback of this architecture is the challenges in avoiding conflicts when data is read and updated asynchronously in multiple locations.



Kafka Multi Cluster & Replication – Active / Passive

- **Advantages**
  - Simplicity in setup and the fact that it can be used in pretty much any use case
  - No need to worry about access to data, handling conflicts, and other architectural complexities.
  - Good for cloud migrations as well

- **Disadvantages**
  - Waste of a good cluster
  - The fact that it is currently not possible to perform cluster failover in Kafka without either losing data or having duplicate events.

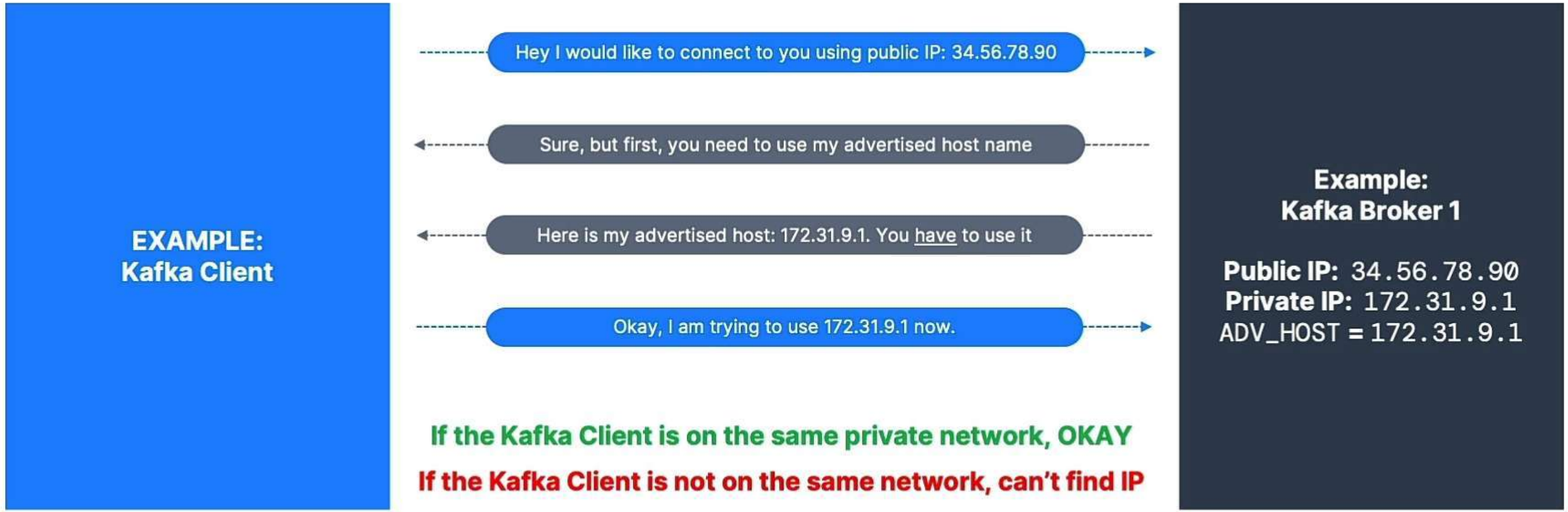


Kafka Multi Cluster & Replication – Resources

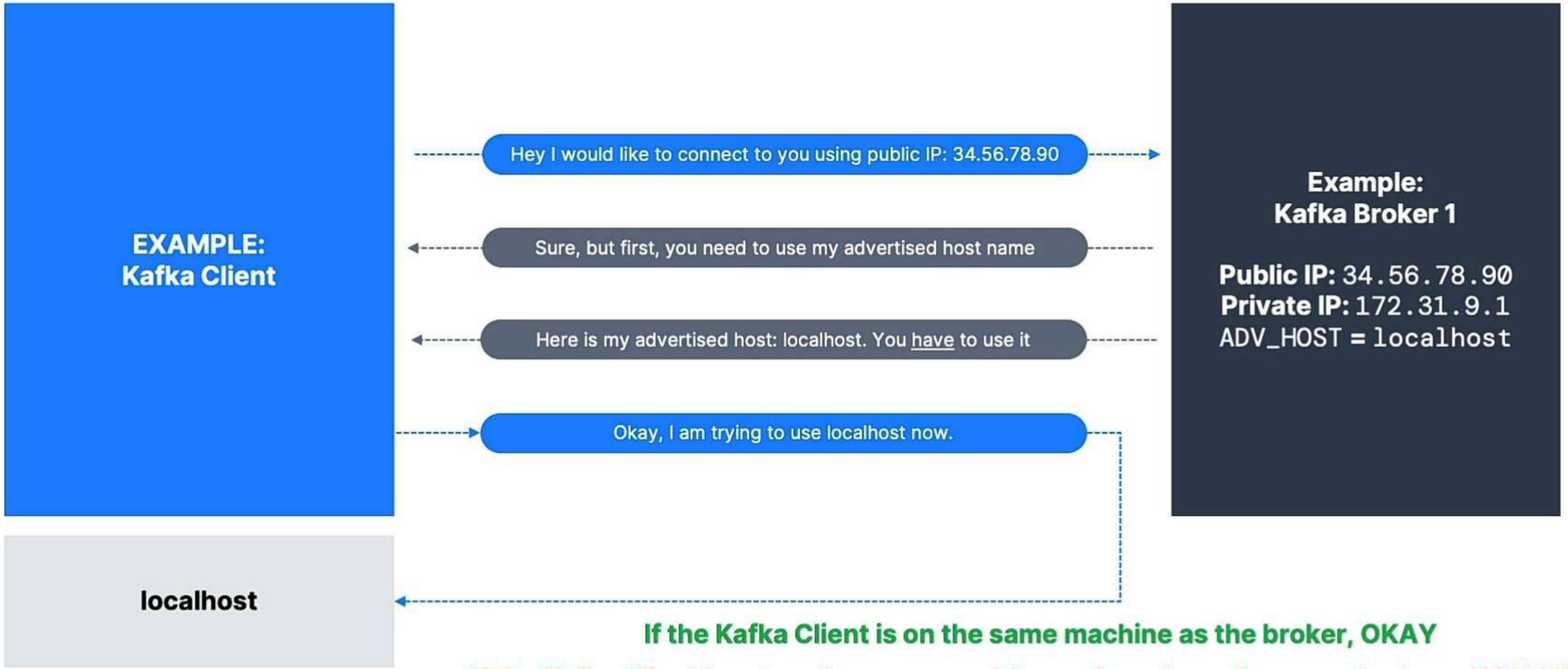
- https://cwiki.apache.org/confluence/display/KAFKA/KIP-382%3A+MirrorMaker+2.0
- https://engineering.salesforce.com/mirrormaker-performance-tuning-63afaed12c21
- https://docs.confluent.io/current/multi-dc/replicator-tuning.html#improving-network-utilization-of-a-connect-task
- https://community.hortonworks.com/articles/79891/kafka-mirror-maker-best-practices.html
- https://www.confluent.io/kafka-summit-sf17/multitenant-multicluster-and-hieracrchical-kafka-messaging-service
- https://eng.uber.com/ureplicator/
- https://www.altoros.com/blog/multi-cluster-deployment-options-for-apache-kafka-pros-and-cons/
- https://github.com/apache/kafka/blob/trunk/connect/mirror/README.md

Understanding communications between Client and with Kafka

- Advertised listeners is the most important setting of Kafka



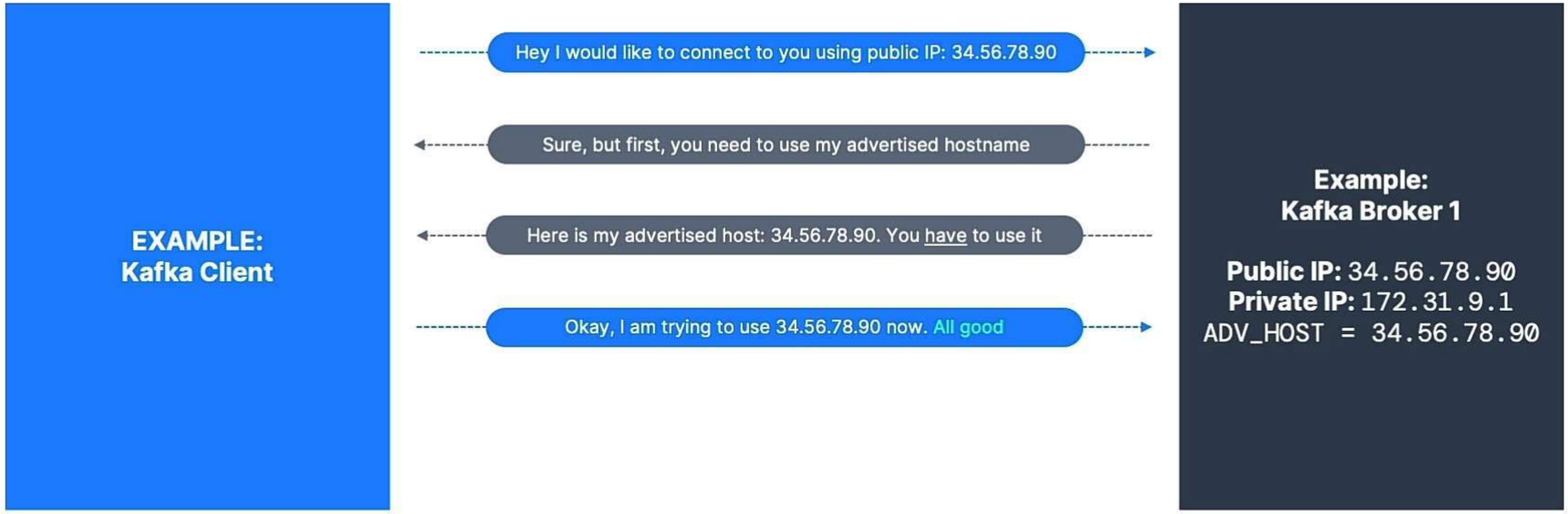
But what if I put localhost? It works on my machine!



If the Kafka Client is on the same machine as the broker, OKAY

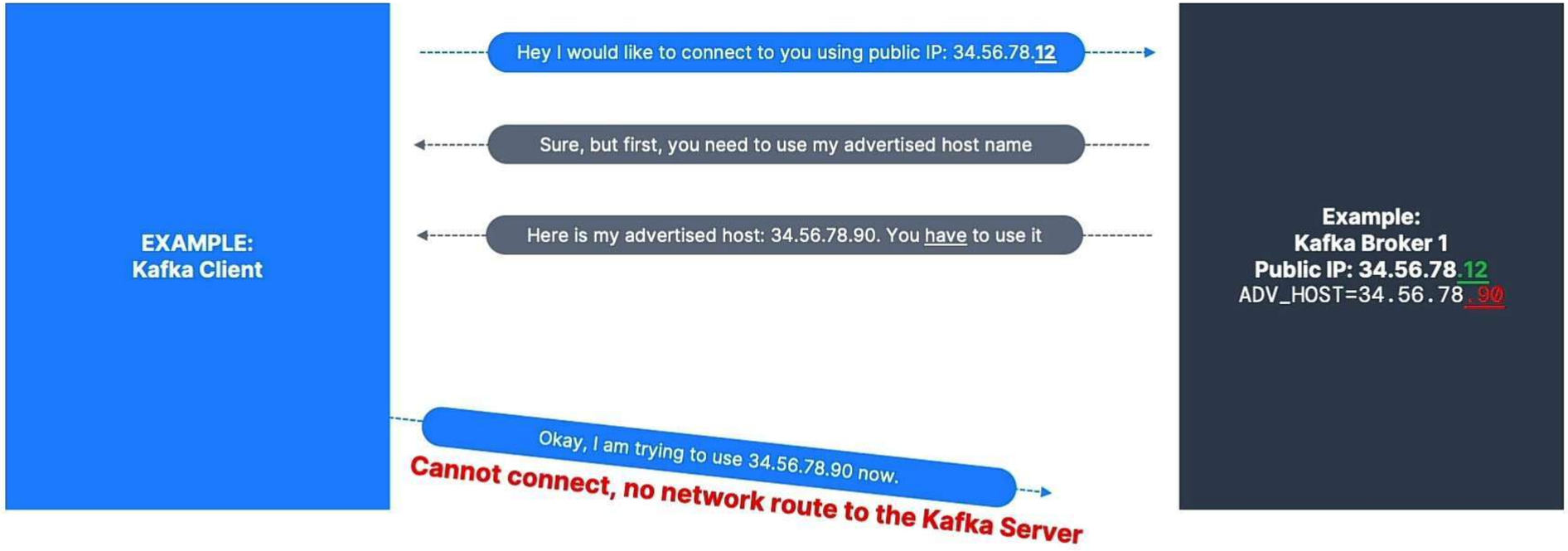
If the Kafka Client is not on the same machine and you have 2 or more brokers, NOT OKAY

What if I use the public IP?



What if I use the public IP...
But after a reboot Kafka public IP changed!

- Assume the IP of your server has changed:
  - FROM 34.56.78.90 => TO 34.56.78.12



So, what do I set for `advertised.listeners`?

- If your clients are on your private network, set either:
  - the internal private IP
  - the internal private DNS hostname
- Your clients should be able to resolve the internal IP or hostname

- If your clients are on a public network, set either:
  - The external public IP
  - The external public hostname pointing to the public IP
- Your clients must be able to resolve the public hostname

Why should I care about topic configuration?

- Brokers have defaults for all the topic configuration parameters
- These parameters impact **performance** and **topic behavior**

- Some topics may need different values than the defaults
  - Replication Factor
  - # of Partitions
  - Message size
  - Compression level
  - Log Cleanup Policy
  - Min Insync Replicas
  - Other configurations

- A list of configuration can be found at: https://kafka.apache.org/documentation/#brokerconfigs

```
~ kafka-topics --create --bootstrap-server localhost:9092 --topic configured-topic --replication-factor 1 --partitions 3
Created topic configured-topic.
~ kafka-topics --bootstrap-server localhost:9092 --topic configured-topic --describe
Topic: configured-topic TopicId: Fxlr1MRkS8ajHj8jVrl6WA PartitionCount: 3       ReplicationFactor: 1    Configs:
        Topic: configured-topic Partition: 0    Leader: 0       Replicas: 0     Isr: 0
        Topic: configured-topic Partition: 1    Leader: 0       Replicas: 0     Isr: 0
        Topic: configured-topic Partition: 2    Leader: 0       Replicas: 0     Isr: 0
~ kafka-configs
This tool helps to manipulate and describe entity config for a topic, client, user, broker or ip
Option                                  Description
------                                  -----------
--add-config <String>                   Key Value pairs of configs to add.
                                          Square brackets can be used to group
                                          values which contain commas: 'k1=v1,
                                          k2=[v1,v2,v2],k3=v3'. The following
                                          is a list of valid configurations:
                                          For entity-type 'topics':
                                        cleanup.policy
                                        compression.type
                                        delete.retention.ms
                                        file.delete.delay.ms
                                        flush.messages
                                        flush.ms
                                        follower.replication.throttled.
                                         replicas
                                        index.interval.bytes
                                        leader.replication.throttled.replicas
                                        local.retention.bytes
                                        local.retention.ms
```

```
~ kafka-configs --bootstrap-server localhost:9092 --entity-type topics --entity-name configured-topic --describe
Dynamic configs for topic configured-topic are:
~
```

```
~ kafka-configs --bootstrap-server localhost:9092 --entity-type topics --entity-name configured-topic --describe
Dynamic configs for topic configured-topic are:
~ kafka-configs --bootstrap-server localhost:9092 --entity-type topics --entity-name configured-topic --alter --add-config min.insync
.replicas=2
Completed updating config for topic configured-topic.
~ kafka-configs --bootstrap-server localhost:9092 --entity-type topics --entity-name configured-topic --describe
Dynamic configs for topic configured-topic are:
  min.insync.replicas=2 sensitive=false synonyms={DYNAMIC_TOPIC_CONFIG:min.insync.replicas=2, DEFAULT_CONFIG:min.insync.replicas=1}
~
```

```
~ kafka-topics --bootstrap-server localhost:9092 --topic configured-topic --describe
Topic: configured-topic TopicId: Fxlr1MRkS8ajHj8jVrl6WA PartitionCount: 3       ReplicationFactor: 1    Configs: min.insync.replicas=2
        Topic: configured-topic Partition: 0    Leader: 0       Replicas: 0     Isr: 0
        Topic: configured-topic Partition: 1    Leader: 0       Replicas: 0     Isr: 0
        Topic: configured-topic Partition: 2    Leader: 0       Replicas: 0     Isr: 0
~ kafka-configs --bootstrap-server localhost:9092 --entity-type topics --entity-name configured-topic --alter --delete-config min.ins
ync.replicas
Completed updating config for topic configured-topic.
~ kafka-topics --bootstrap-server localhost:9092 --topic configured-topic --describe
```

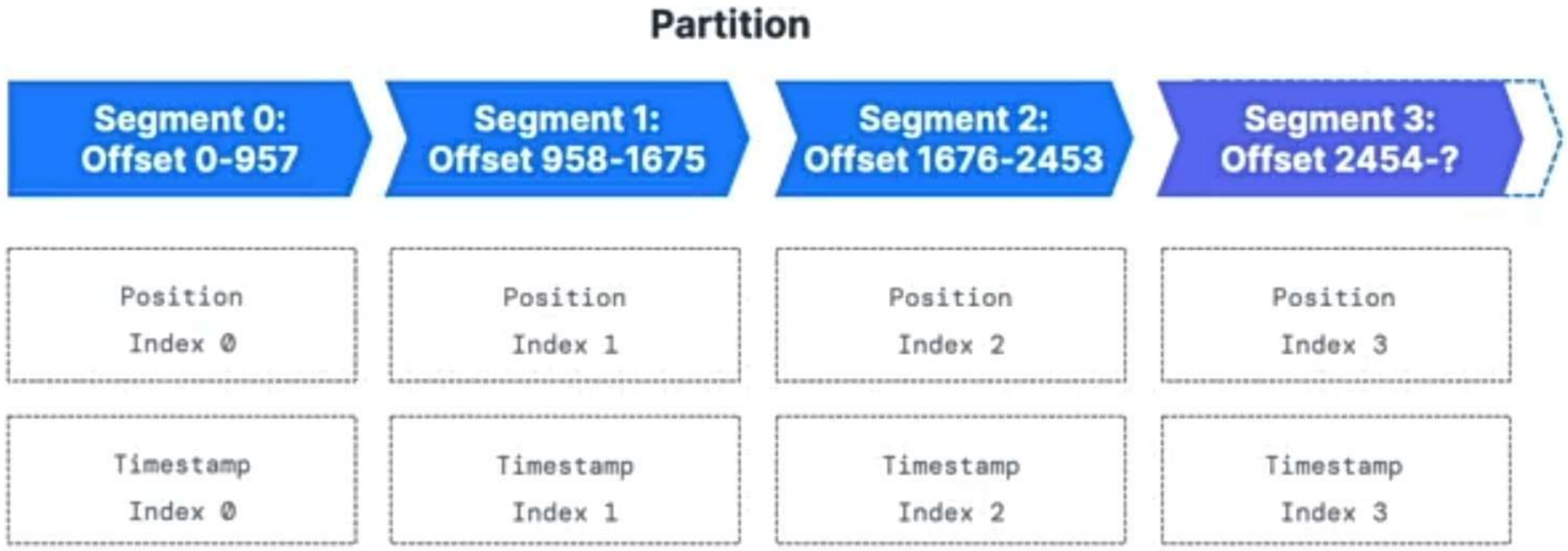
Partitions and Segments

- Topics are made of partitions (we already know that)
- Partitions are made of… segments (files)!

- Only one segment is ACTIVE (the one data is being written to)
- Two segment settings:
  - log.segment.bytes: the max size of a single segment in bytes (default 1GB)
  - log.segment.ms: the time Kafka will wait before committing the segment if not full (1 week)

Segments and Indexes

- Segments come with two indexes (files):
  - An offset to position index: helps Kafka find where to read from to find a message
  - A timestamp to offset index: helps Kafka find messages with a specific timestamp



Segments: Why should I care?

- A smaller **log.segment.bytes** (size, default: 1GB) means:
  - More segments per partitions
  - Log Compaction happens more often
  - BUT Kafka must keep more files opened (*Error: Too many open files*)
- Ask yourself: how fast will I have new segments based on throughput?

- A smaller **log.segment.ms** (time, default 1 week) means:
  - You set a max frequency for log compaction (more frequent triggers)
  - Maybe you want daily compaction instead of weekly?
- Ask yourself: how often do I need log compaction to happen?

Log Cleanup Policies

- Many Kafka clusters make data expire, according to a policy
- That concept is called log cleanup.

Policy1: log.cleanup.policy=delete (Kafka default for all user topics)

- Delete based on age of data (default is a week)
- Delete based on max size of log (default is -1 == infinite)

Policy 2: log.cleanup.policy=compact (Kafka default for topic *__consumer_offsets*)

```
~ clear
~ kafka-topics --bootstrap-server localhost:9092 --topic configured-topic --describe
Topic: configured-topic TopicId: Fxlr1MRkS8ajHj8jVrl6WA PartitionCount: 3       ReplicationFactor: 1    Configs: min.insync.replicas=2
        Topic: configured-topic Partition: 0    Leader: 0       Replicas: 0     Isr: 0
        Topic: configured-topic Partition: 1    Leader: 0       Replicas: 0     Isr: 0
        Topic: configured-topic Partition: 2    Leader: 0       Replicas: 0     Isr: 0
~ kafka-configs --bootstrap-server localhost:9092 --entity-type topics --entity-name configured-topic --alter --delete-config min.ins
ync.replicas
Completed updating config for topic configured-topic.
~ kafka-topics --bootstrap-server localhost:9092 --topic configured-topic --describe
Topic: configured-topic TopicId: Fxlr1MRkS8ajHj8jVrl6WA PartitionCount: 3       ReplicationFactor: 1    Configs:
        Topic: configured-topic Partition: 0    Leader: 0       Replicas: 0     Isr: 0
        Topic: configured-topic Partition: 1    Leader: 0       Replicas: 0     Isr: 0
        Topic: configured-topic Partition: 2    Leader: 0       Replicas: 0     Isr: 0
~ clear
~ kafka-topics --bootstrap-server localhost:9092 --describe --topic __consumer_offsets
Topic: __consumer_offsets       TopicId: 5pzL6ZPSQgKg-YZZ7g2DKw PartitionCount: 50      ReplicationFactor: 1    Configs: compression.typ
e=producer,cleanup.policy=compact,segment.bytes=104857600
        Topic: __consumer_offsets       Partition: 0    Leader: 0       Replicas: 0     Isr: 0
        Topic: __consumer_offsets       Partition: 1    Leader: 0       Replicas: 0     Isr: 0
        Topic: __consumer_offsets       Partition: 2    Leader: 0       Replicas: 0     Isr: 0
        Topic: __consumer_offsets       Partition: 3    Leader: 0       Replicas: 0     Isr: 0
        Topic: __consumer_offsets       Partition: 4    Leader: 0       Replicas: 0     Isr: 0
        Topic: __consumer_offsets       Partition: 5    Leader: 0       Replicas: 0     Isr: 0
        Topic: __consumer_offsets       Partition: 6    Leader: 0       Replicas: 0     Isr: 0
        Topic: __consumer_offsets       Partition: 7    Leader: 0       Replicas: 0     Isr: 0
        Topic: __consumer_offsets       Partition: 8    Leader: 0       Replicas: 0     Isr: 0
        Topic: __consumer_offsets       Partition: 9    Leader: 0       Replicas: 0     Isr: 0
        Topic: __consumer_offsets       Partition: 10   Leader: 0       Replicas: 0     Isr: 0
```

Log Cleanup Policies

- Many Kafka clusters make data expire, according to a policy
- That concept is called log cleanup.

Policy1: log.cleanup.policy=delete (Kafka default for all user topics)

- Delete based on age of data (default is a week)
- Delete based on max size of log (default is -1 == infinite)

Policy 2: log.cleanup.policy=compact (Kafka default for topic *__consumer_offsets*)

- Delete based on keys of your messages
- Will delete old duplicate keys **after** the active segment is committed
- Infinite time and space retention

Log Cleanup: Why and When?

- Deleting data from Kafka allows you to:
  - Control the size of the data on the disk, delete obsolete data
  - Overall: Limit maintenance work on the Kafka Cluster

- How often does log cleanup happen?
  - Log cleanup happens on your partition segments!
  - Smaller / More segments means that log cleanup will happen more often!
  - Log cleanup shouldn't happen too often => takes CPU and RAM resources
  - The cleaner checks for work every 15 seconds (log.cleaner.backoff.ms)

Log Cleanup Policy: Delete

- `log.retention.hours`:
  - number of hours to keep data for (default is 168 – one week)
  - Higher number means more disk space
  - Lower number means that less data is retained (if your consumers are down for too long, they can miss data)
  - Other parameters allowed: `log.retention.ms`, `log.retention.minutes` (smaller unit has precedence)

- `log.retention.bytes`:
  - Max size in Bytes for each partition (default is -1 – infinite)
  - Useful to keep the size of a log under a threshold

Log Cleanup Policy: Delete

Use cases - two common pair of options:

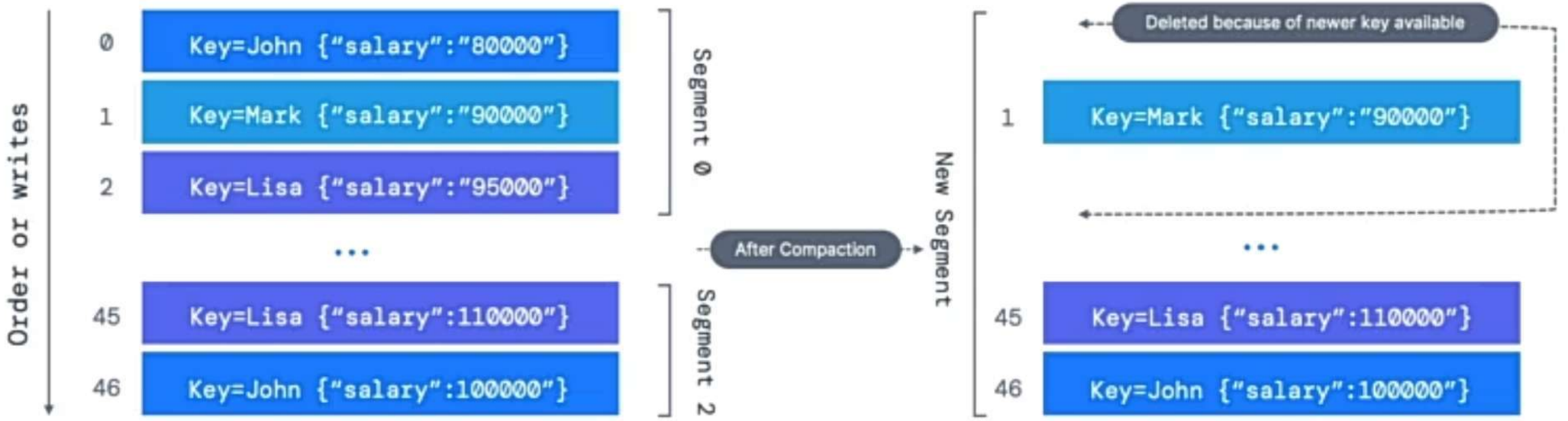
- One week of retention:
  - `log.retention.hours=168` and `log.retention.bytes=-1`
- Infinite time retention bounded by 500MB:
  - `log.retention.ms=-1` and `log.retention.bytes=524288000`

Log Cleanup Policy: Compact

- Log compaction ensures that your log contains at least the last known value for a specific key within a partition
- Very useful if we just require a SNAPSHOT instead of full history (such as for a data table in a database)
- The idea is that we only keep the latest “update” for a key in our log

Log Compaction: Example

- Our topic is: employee-salary
- We want to keep the most recent salary for our employees



Log Compaction Guarantees

- **Any consumer that is reading from the tail of a log (most current data) will still see all the messages sent to the topic**
- Ordering of messages it kept, log compaction only removes some messages, but does not re-order them
- The offset of a message is immutable (it never changes). Offsets are just skipped if a message is missing
- Deleted records can still be seen by consumers for a period of `delete.retention.ms` (default is 24 hours).

Log Compaction Myth Busting

- It doesn't prevent you from pushing duplicate data to Kafka
  - De-duplication is done after a segment is committed
  - Your consumers will still read from tail as soon as the data arrives
- It doesn't prevent you from reading duplicate data from Kafka
  - Same points as above
- Log Compaction can fail from time to time
  - It is an optimization and it the compaction thread might crash
  - Make sure you assign enough memory to it and that it gets triggered
  - Restart Kafka if log compaction is broken
- You can't trigger Log Compaction using an API call (for now…)

Log Compaction – How it works

- Log compaction log.cleanup.policy=compact is impacted by:
  - segment.ms (default 7 days): Max amount of time to wait to close active segment
  - segment.bytes (default 1G): Max size of a segment
  - min.compaction.lag.ms (default 0): how long to wait before a message can be compacted
  - delete.retention.ms (default 24 hours): wait before deleting data marked for compaction
  - min.cleanable.dirty.ratio (default 0.5): higher => less, more efficient cleaning. Lower => opposite

unclean.leader.election.enable

- If all your In Sync Replicas go offline (but you still have out of sync replicas up), you have the following option:
  - Wait for an ISR to come back online (default)
  - Enable `unclean.leader.election.enable=true` and start producing to non ISR partitions
- If you enable `unclean.leader.election.enable=true`, you improve availability, but you will lose data because other messages on ISR will be discarded when they come back online and replicate data from the new leader.
- Overall, this is a very dangerous setting, and its implications must be understood fully before enabling it.
- Use cases include metrics collection, log collection, and other cases where data loss is somewhat acceptable, at the trade-off of availability.

Large Messages in Apache Kafka

- Kafka has a default of 1 MB per message in topics, as large messages are considered inefficient and an anti-pattern.

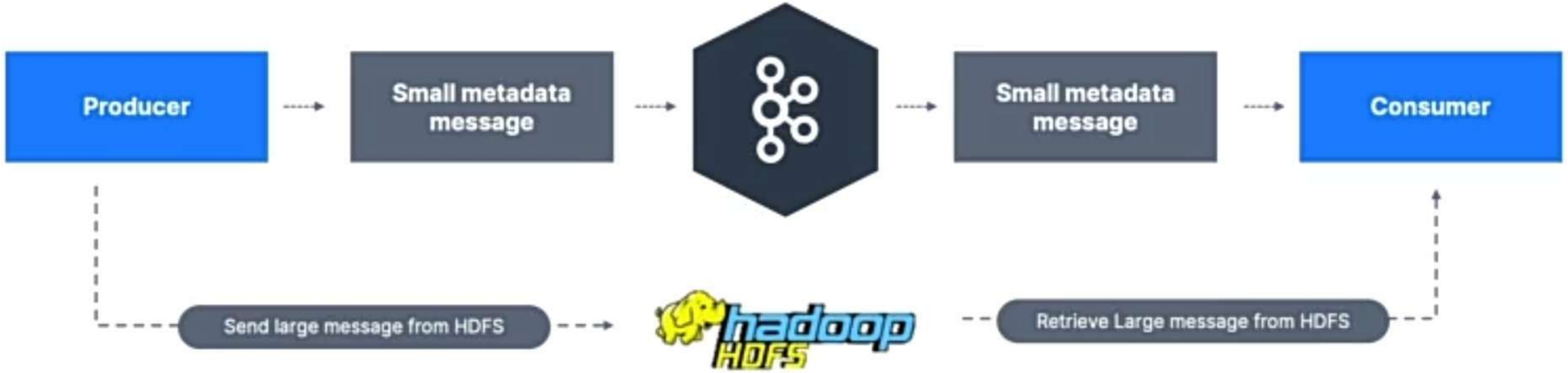
- Two approaches to sending large messages:

1. Using an external store: store messages in HDFS, Amazon S3, Google Cloud Storage, etc… and send a reference of that message to Apache Kafka

2. Modifying Kafka parameters: must change broker, producer and consumer settings

Option 1: Large Messages using External Store

- Store the large message (e.g. video, archive file, etc…) outside of Kafka
- Send a reference of that message to Kafka
- Write custom code at the producer / consumer level to handle this pattern



Option 2: Sending large messages in Kafka (ex: 10MB)

- **Topic-wise, Kafka-side**, set max message size to 10MB:
  - Broker side: modify `message.max.bytes`
  - Topic side: modify `max.message.bytes`
  - Warning: the settings have similar but different name; this is not a typo!
- **Broker-wise**, set max replication fetch size to 10MB
  - `replica.fetch.max.bytes=10485880` (in `server.properties`)
- **Consumer-side**, must increase fetch size of the consumer will crash:
  - `max.partition.fetch.bytes=10485880`
- **Producer-side**, must increase the max request size
  - `max.request.size=10485880`