Apache Kafka

Motivation

The Shift to Event-driven Systems has Already Begun...

From a static snapshot... **...to a continuous stream of events**

Occasional call to a friend

A constant feed about the activities of all your friends

Daily news reports

Real time news feeds, accessible online anytime, anywhere

- Single Platform to connect everyone to every event
- Real Time of Stream of events
- All events Stored for historical view

Messaging System

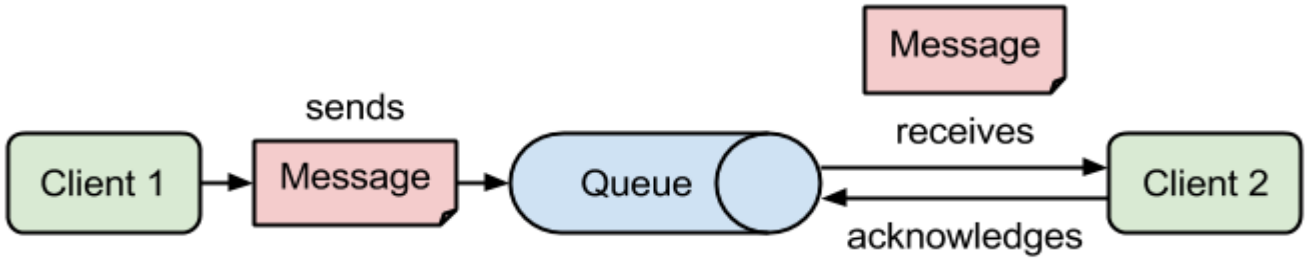
There are two types of Messaging System:

1. **Point to Point System**
2. **Publish-Subscribe System**

1. Point to Point System

Messages are persisted in a queue, but a particular message can be consumed by a maximum of one consumer only. Once a consumer reads a message in the queue, it disappears from that queue.

The typical example of this system is an Order Processing System, where each order will be processed by one Order Processor.

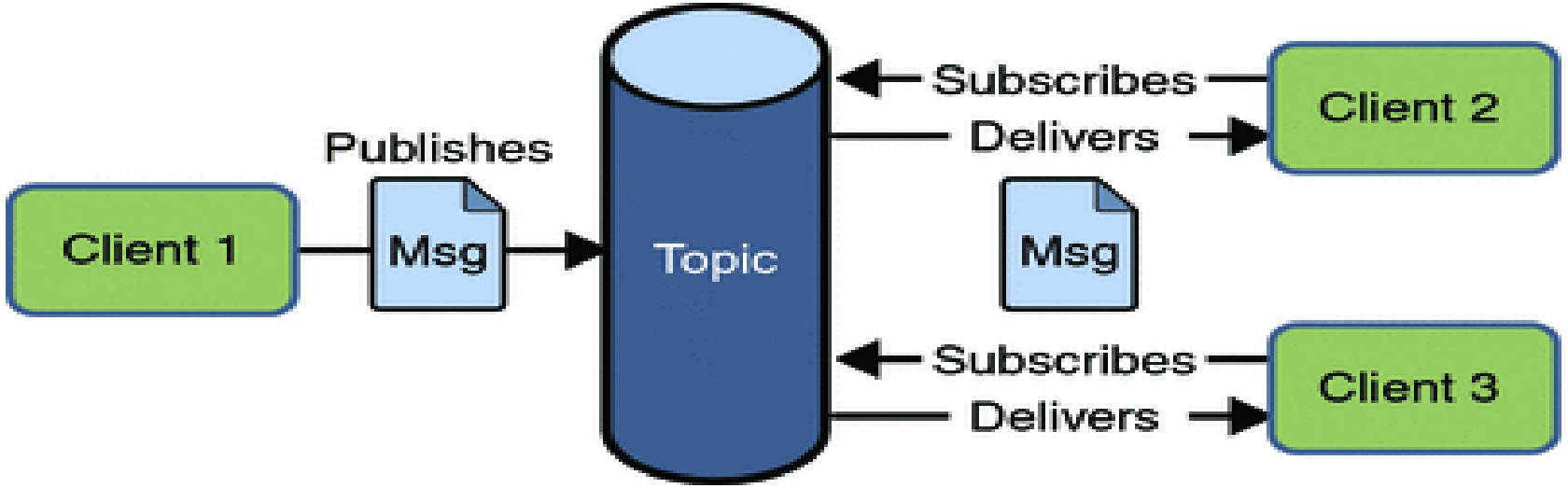


Messaging System

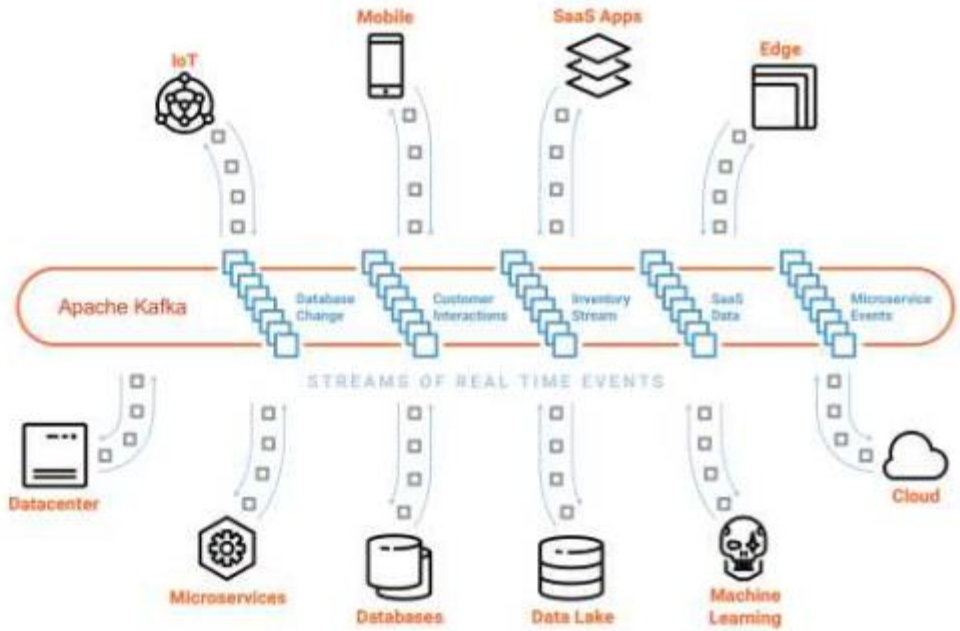
2. Publish-Subscribe System

Messages are persisted in a topic. Unlike point-to-point system, consumers can subscribe to one or more topic and consume all the messages in that topic. In the Publish-Subscribe system, message producers are called publishers and message consumers are called subscribers.

A real-life example is Dish TV, which publishes different channels like sports, movies, music, etc., and anyone can subscribe to their own set of channels and get them whenever their subscribed channels are available.



Motivation



Apache Kafka®: the De-facto Standard for Real-Time Event Streaming

- Global-scale
- Real-time
- Persistent Storage
- Stream Processing

Apache Kafka introduction

Apache Kafka is a software platform which is based on a distributed streaming process.
It is a publish-subscribe messaging system which let exchanging of data between applications, servers, and processors as well.

It is designed and optimized to be a high-throughput, low-latency, fault-tolerant, scalable platform for handling real-time data feeds

Apache Kafka was originally developed by **LinkedIn in 2010**, and later it was donated to the Apache Software Foundation. Currently, it is maintained by **Confluent** under Apache Software Foundation.

In the year **2011** Kafka was made public.

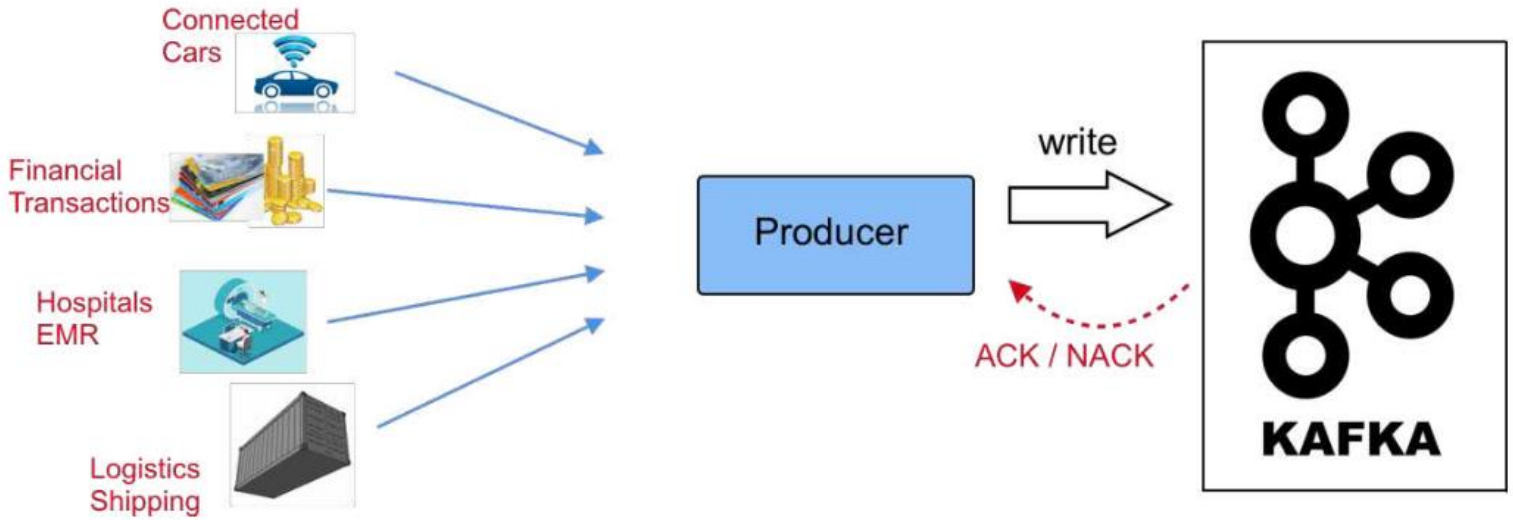
Motivation

Thousands of Companies Worldwide trust Kafka for their Journey towards "**Event-driven**"

Kafka Components

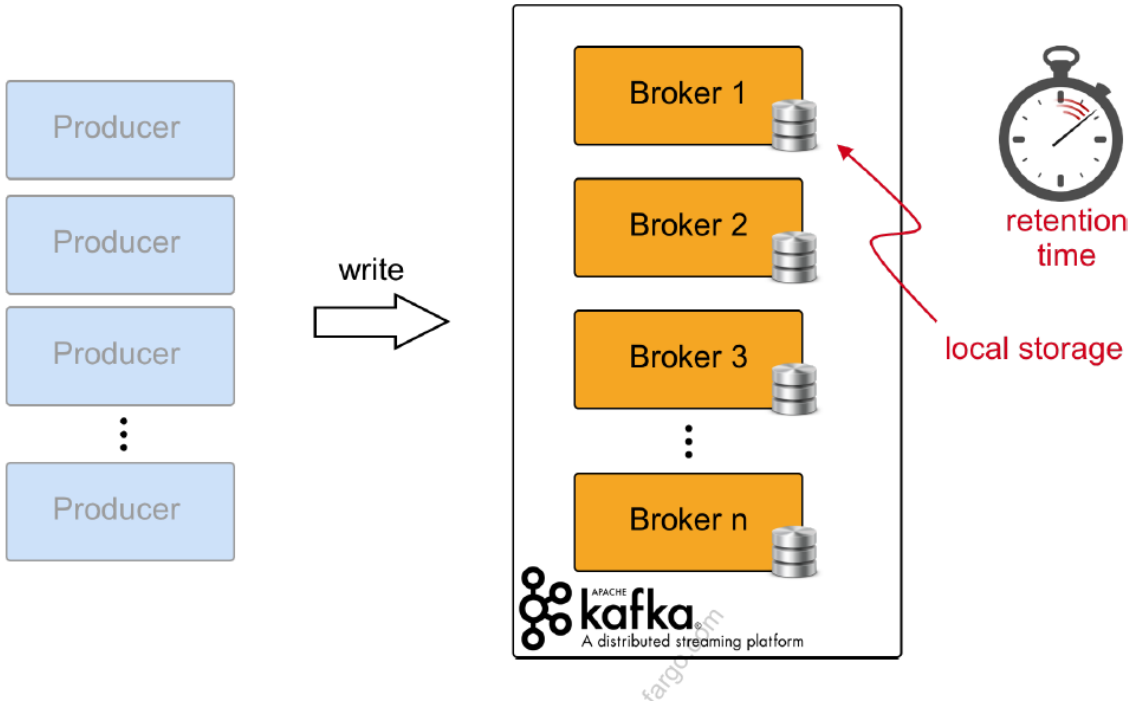
Producer
Brokers
Consumers
Topic
Partitions
Clusters
Zookeeper

Producers



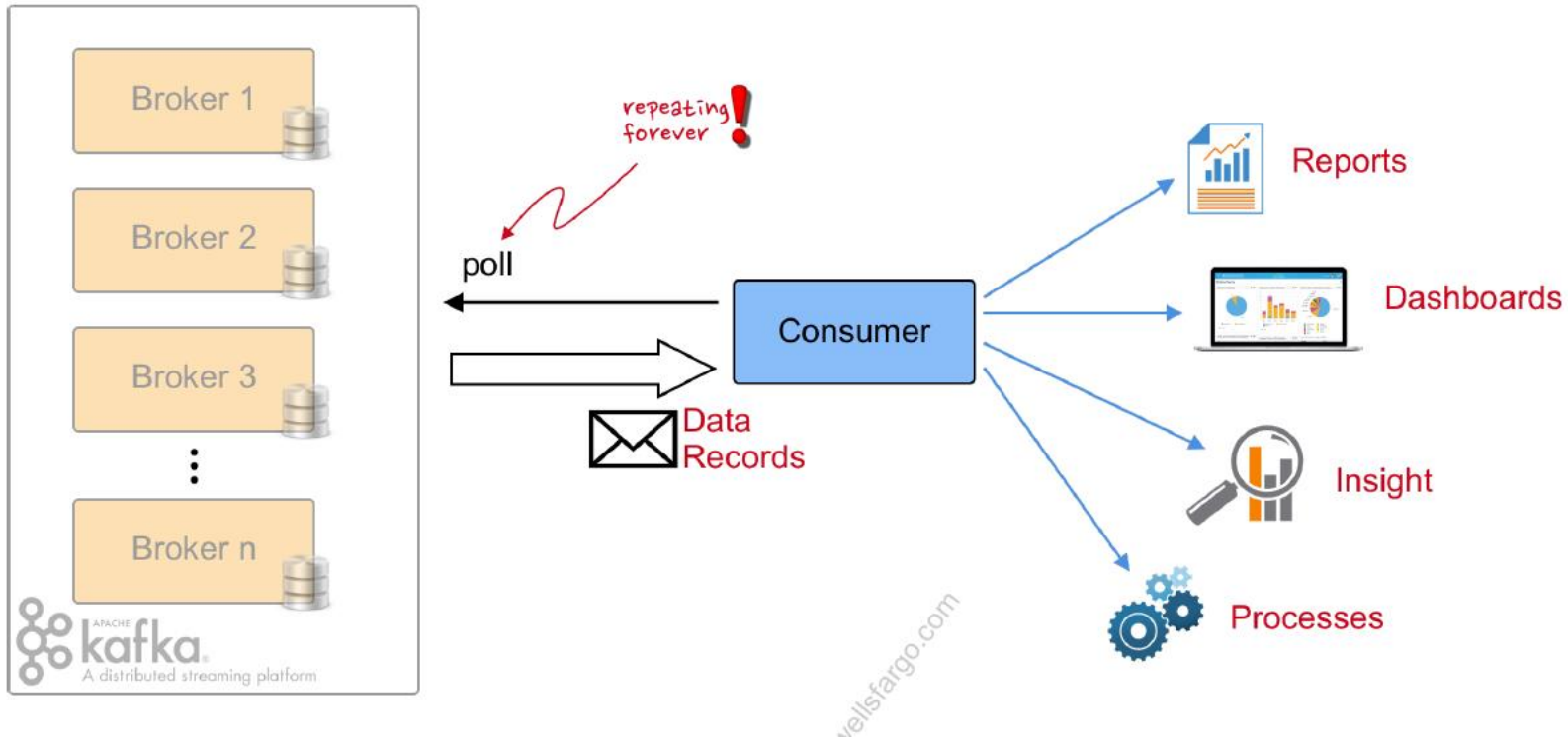
- The application that forward or write all this data to Kafka are called producers
- A Producer Sends data to Kafka
- Producer (optionally) receives ACK or NACK from Kafka
- If an ACK(Acknowledged) => all good
- If a NACK (not acknowledged) is received then the producer knows that Kafka was notable to accept the data for whatever reason. In this case the producer automatically retries to send the data.
- Many producers sends can data to Kafka concurrently

Kafka Brokers



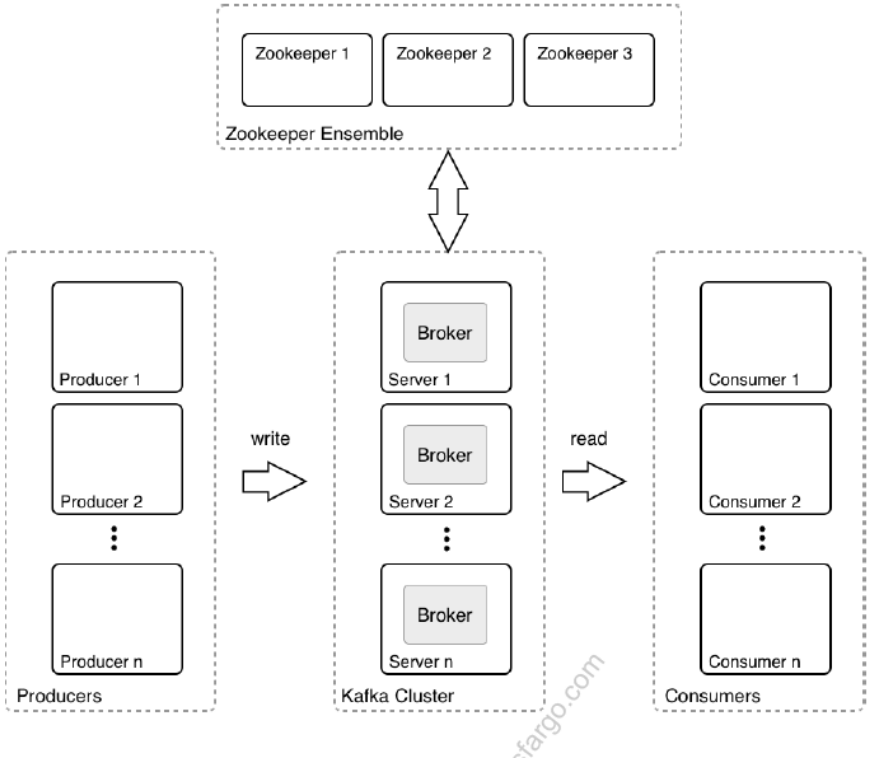
- Brokers receive the data from producers and **store it temporarily** in the page cache, or permanently on disk after the OS flushes the page cache
- Brokers keep the data ready for down stream consumers
- How long the data is kept around is determined by the so called **retention time** (1 week by
- default)

Consumers



- A consumer **polls** data from Kafka
- Each consumer **periodically** asks brokers of the Kafka cluster: "Do you have more data for me?". Normally the consumer does this in an endless loop.
- Many consumers can poll data from Kafka at the same time
- Many different consumers can poll the same data, each **at their own pace**
- To allow for **parallelism**, consumers are organized in consumer groups that **split up the work**

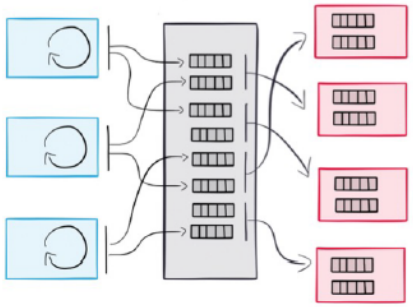
Architecture



- On the left you see producers that **write** data to Kafka
- In the middle you have the Kafka cluster consisting of many brokers that receive and **store** the data
- On the right you see consumers that poll or **read** data from Kafka for downstream Processing
- On top there is a cluster of ZooKeeper instances that form a so called **ensemble**.

Decoupling Producers and Consumers

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

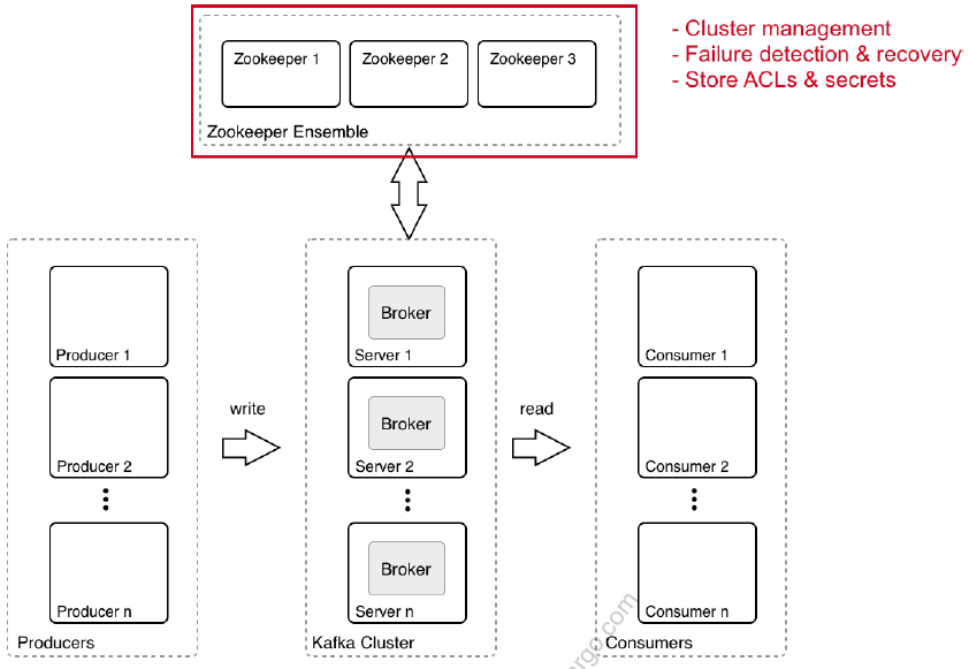


a key feature of Kafka is that Producers and Consumers are **decoupled**, that is,

- They do not need to know about the existence of each other.
- The internal logic of a producer does never depend on any of the downstream consumers
- The internals of a consumer do not depend on the upstream producer

Producers and Consumers simply need to agree on the data format of the records produced and consumed

How Kafka Uses ZooKeeper

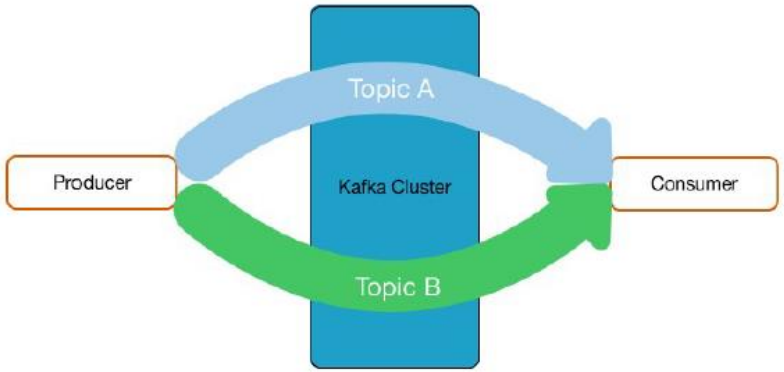


Kafka Brokers use Zookeeper for a number of important internal features such as

- Cluster management
- Failure detection and recovery (e.g. when a broker goes down)
- To store Access Control Lists (ACLs) used for authorization in the Kafka cluster
- Maintains **configuration information**
- Enables highly reliable **distributed coordination**
- Provides **distributed synchronization**
- 3 or 5 servers form an **ensemble**
- It is not recommended to run more than 5 instances in an ensemble for

Topics

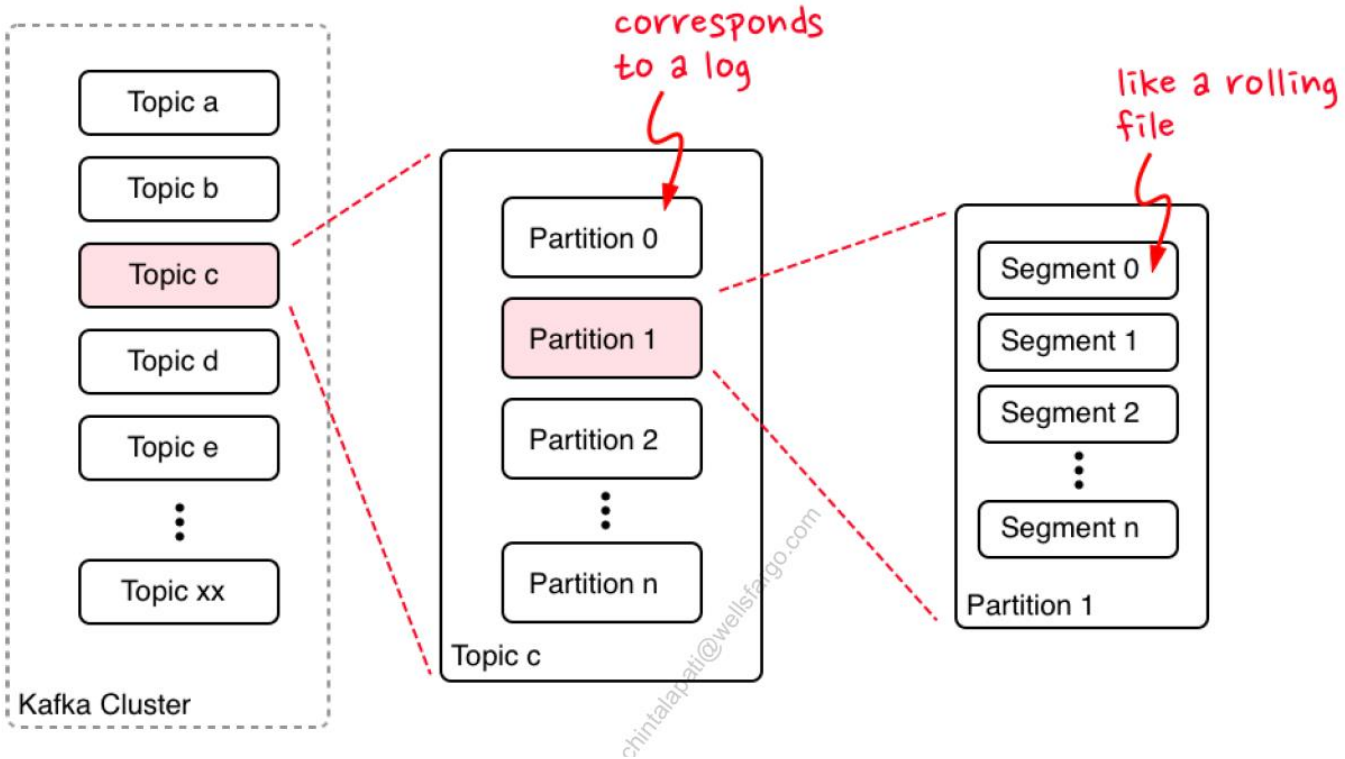
- **Topics**: Streams of "related" Messages in Kafka
  - Is a **Logical Representation**
  - **Categorizes Messages** into Groups
- Developers define Topics
- Producer ←→ Topic: N to N Relation
- Unlimited Number of Topics



The Topic is a logical representation that spans across Producers, Brokers, and Consumers

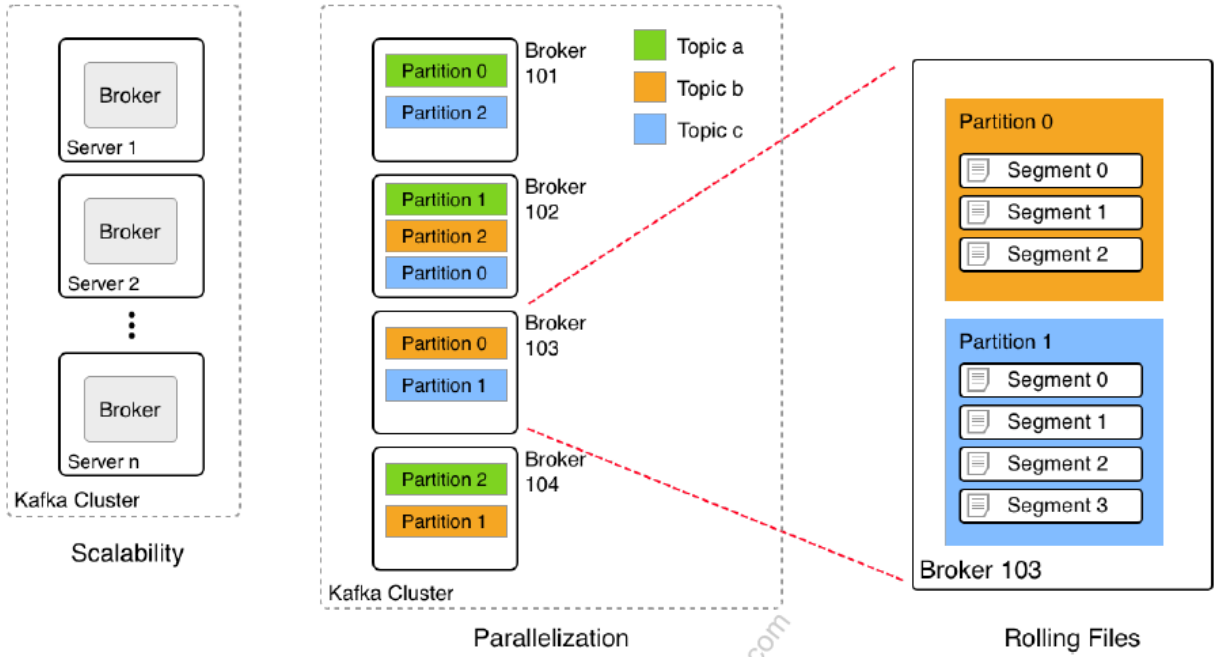
- Developers decide which Topics exist
  - By default, a Topic is auto-created when it is first used
- One or more Producers can write to one or more Topics
- There is no limit to the number of Topics that can be used

Topics, Partitions and Segments



Topic: A topic comprises all messages of a given category.
Partition: To parallelize work and thus increase the throughput Kafka can split a single topic into many partitions. The messages of the topic will then be split between the partitions.
Segment: The broker stores the messages as they come in memory (page cache), then periodically flushes them to a physical file. Since the data can potentially be endless the broker is using a "rolling-file" strategy.

Topics, Partitions and Segments

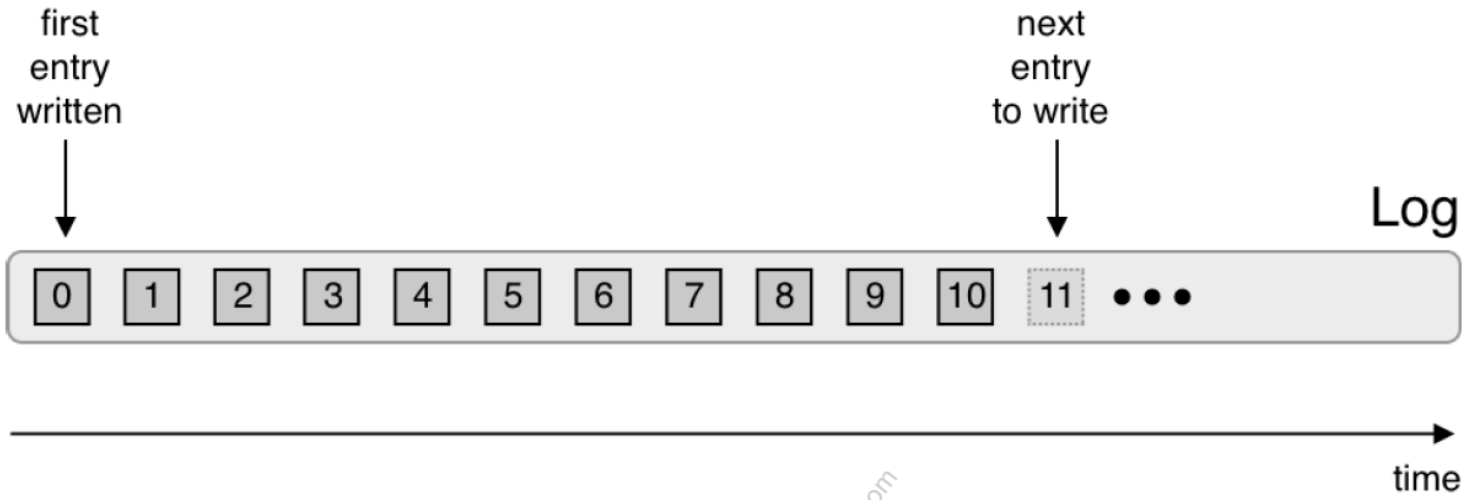


Normally one uses several Kafka brokers that form a cluster to scale out. If we have now three topics (that is 3 categories of messages) then they may be organized physically as shown in the graphic.

- Partitions of each topic are distributed among the brokers
- Each partition on a given broker results in one to many physical files called segments
- Segments are treated in a rolling file strategy. Kafka opens/allocates a file on disk and

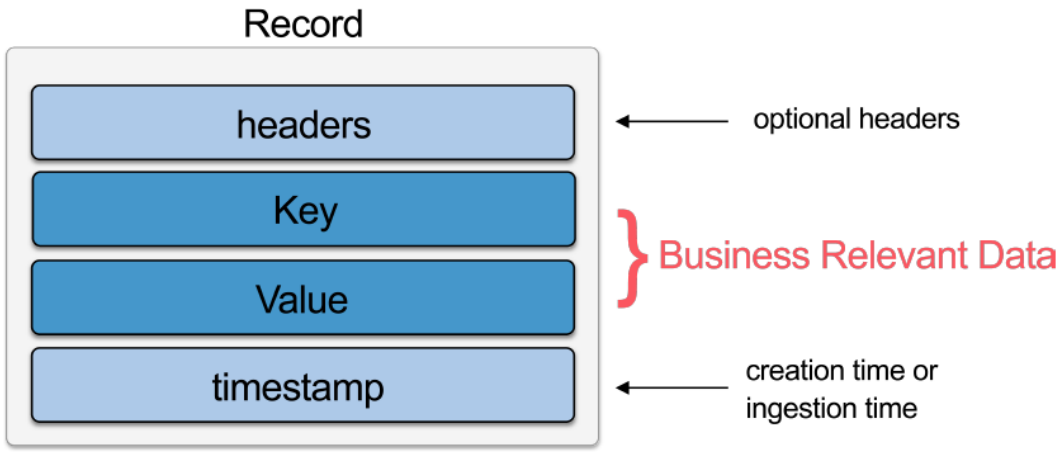
then fills that file sequentially and in append only mode until it is full, where "full" depends on the defined max size of the file

The Log



- A log is a data structure that is like a queue of elements. New elements are always appended at the end of the log and once written they are never changed.
- Elements that are added to the log are strictly ordered in time. The first element added to the log is older than the second one which in turn is older than the third one.
- In the image the time axis reflects that fact. The offset of the elements in that sense can be viewed as a time scale.

Data Elements

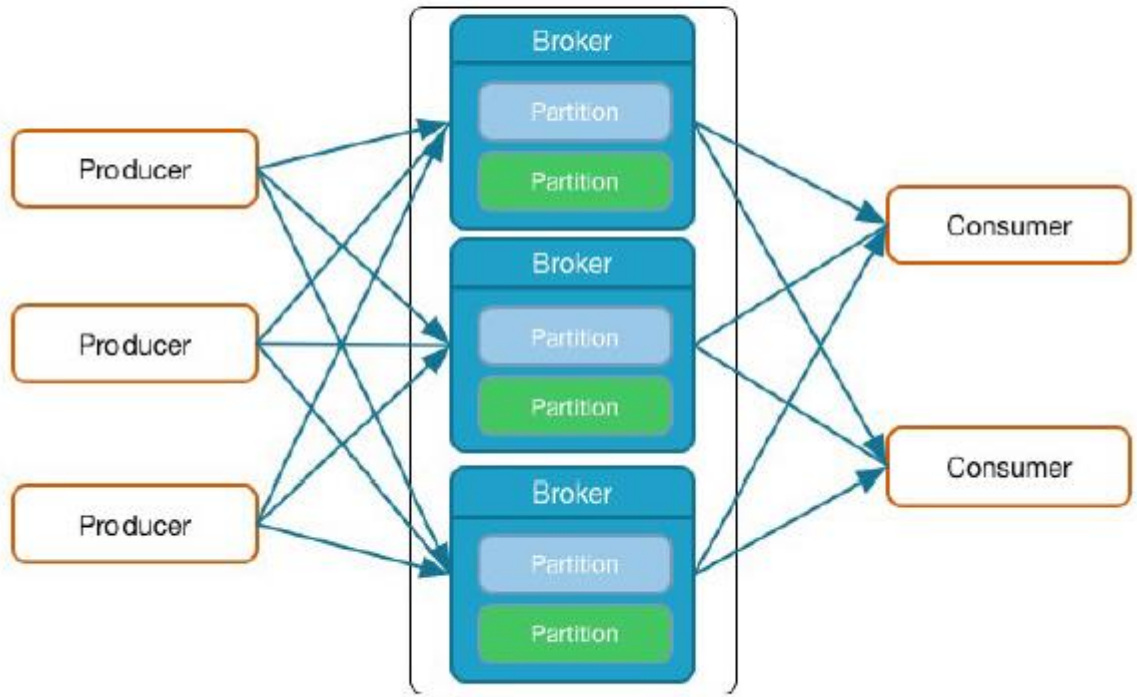


A data element in a log (or topic; to be introduced later) is called a **record** in the Kafka world. Often we also use equivalent words for a record. The most common ones are **message** and **event**.
A record in Kafka consists of **Metadata** and a **Body**.

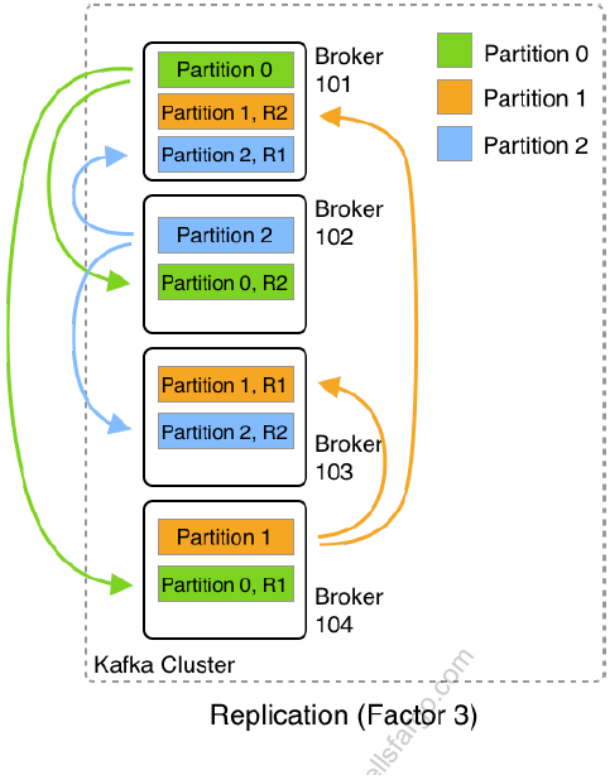
- The metadata contains offset, compression, magic byte, timestamp and an optional **headers** collection of 0 to many key value pairs.
- The body consists of a **Key** and a **Value** part
- The value part is usually containing the **business relevant** data
- The key by default is used to decide into which partition a record is written to. As a consequence all records with identical an key go into the same partition.

Broker Basics

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

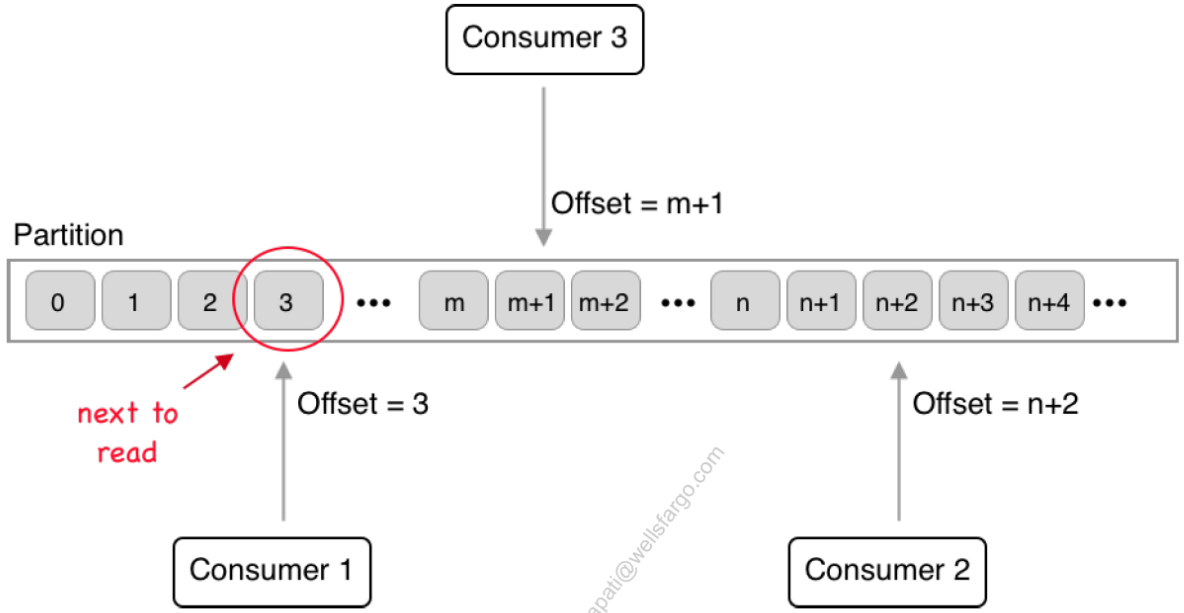


Broker Replication



Kafka can **replicate** partitions across a configurable number of Kafka servers which is used
for **fault tolerance**. Each partition has a leader server and zero or more follower servers.
Leaders handle all read and write requests for a partition.
In this image we have four brokers and three replicated partitions. The replication factor is
also 3. For each partition a different broker is the leader, for **optimal resource usage**.

Consumer Offset



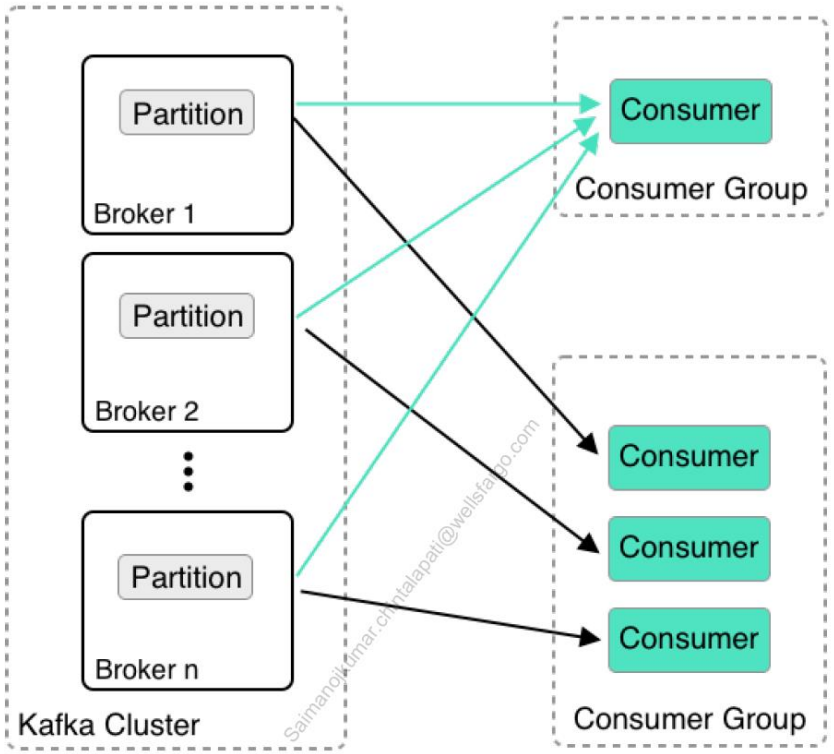
Kafka stores messages in topics for a pre-defined amount of time (by default 1 week), such as that many consumers (or more precisely consumer groups) can access the data. Most consumers will process the incoming data in near-real time, but others may consume the data at a later time.

On the slide we see 3 consumers that all poll data from the same partition of a given topic.

- consumer 1 will be reading from offset 3
- consumer 2 is currently reading from offset n+2
- consumer 3 is reading from offset m+1

Consumers, by default, store their offsets in a special topic called __consumer_offsets in Kafka. The reason being that if a consumer crashes, another consumer can take its place, read the next to read offset from Kafka and proceed the task where the crashed consumer left off.

Consumer Groups



To allow to increase the throughput in downstream consumption of data flowing into a topic, Kafka introduced **Consumer Groups**.

- A consumer group consists of 1 to many consumer instances
- All consumer instances in a consumer group are identical clones of each other
- To be transparently added to a consumer group an instance needs to use the same group.id
- A consumer group can scale out and thus parallelize work until the number of consumer instances is equal to the number of topic partitions

Development: A Basic Producer in Java

BasicProducer.java

```java
package clients;

import java.util.Properties;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;

public class BasicProducer {
    public static void main(String[] args) {
        System.out.println("*** Starting Basic Producer ***");

        Properties settings = new Properties();
        settings.put("client.id", "basic-producer-v0.1.0");
        settings.put("bootstrap.servers", "kafka-1:9092,kafka-2:9092");
        settings.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
        settings.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

        final KafkaProducer<String, String> producer = new KafkaProducer<>(settings);

        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            System.out.println("### Stopping Basic Producer ###");
            producer.close();
        }));

        final String topic = "hello_world_topic";
        for(int i=1; i<=5; i++){
            final String key = "key-" + i;
            final String value = "value-" + i;
            final ProducerRecord<String, String> record = new ProducerRecord<>(topic, key, value);
            producer.send(record);
        }
    }
}
```

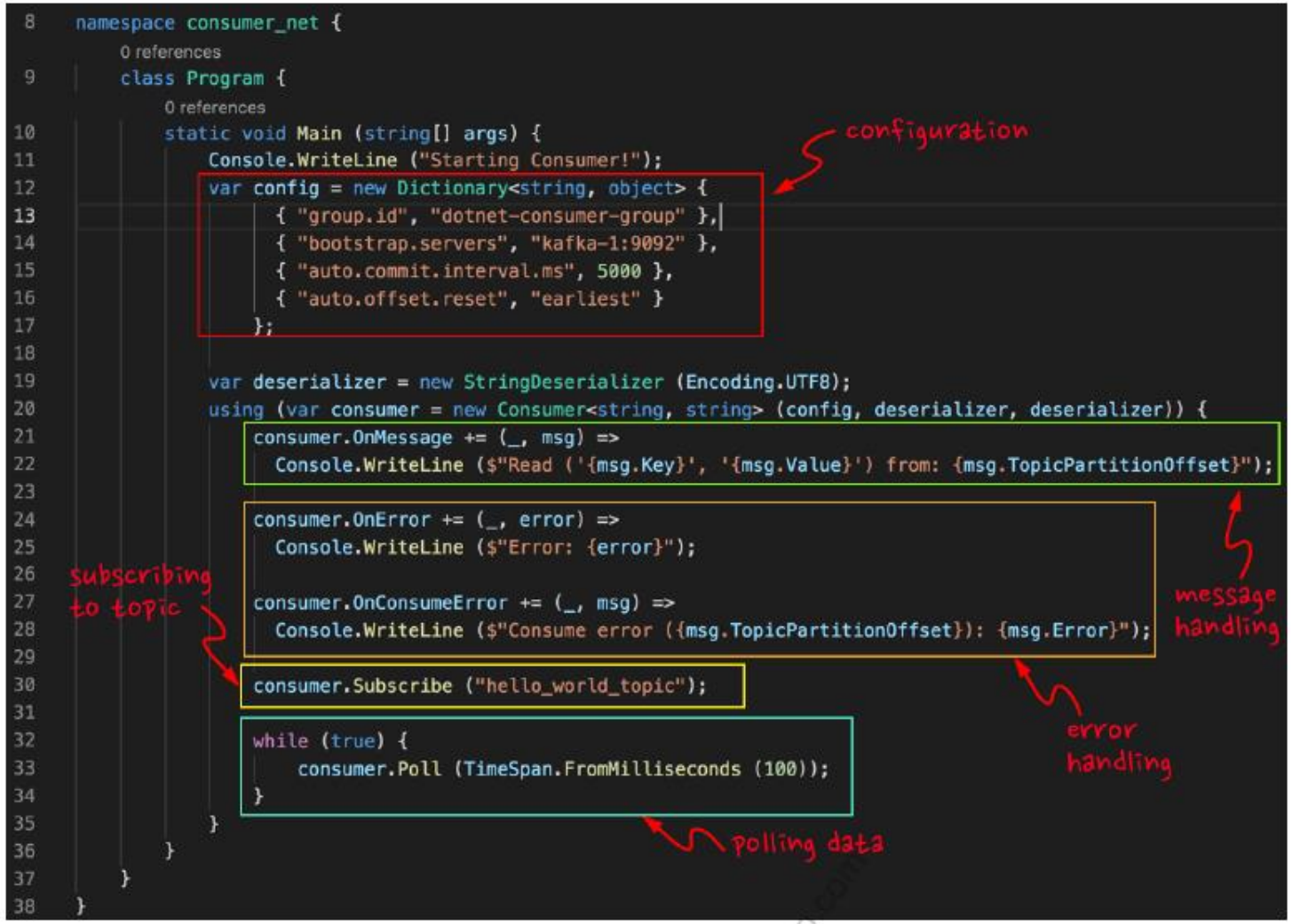
configuration

create producer

shutdown behaviour

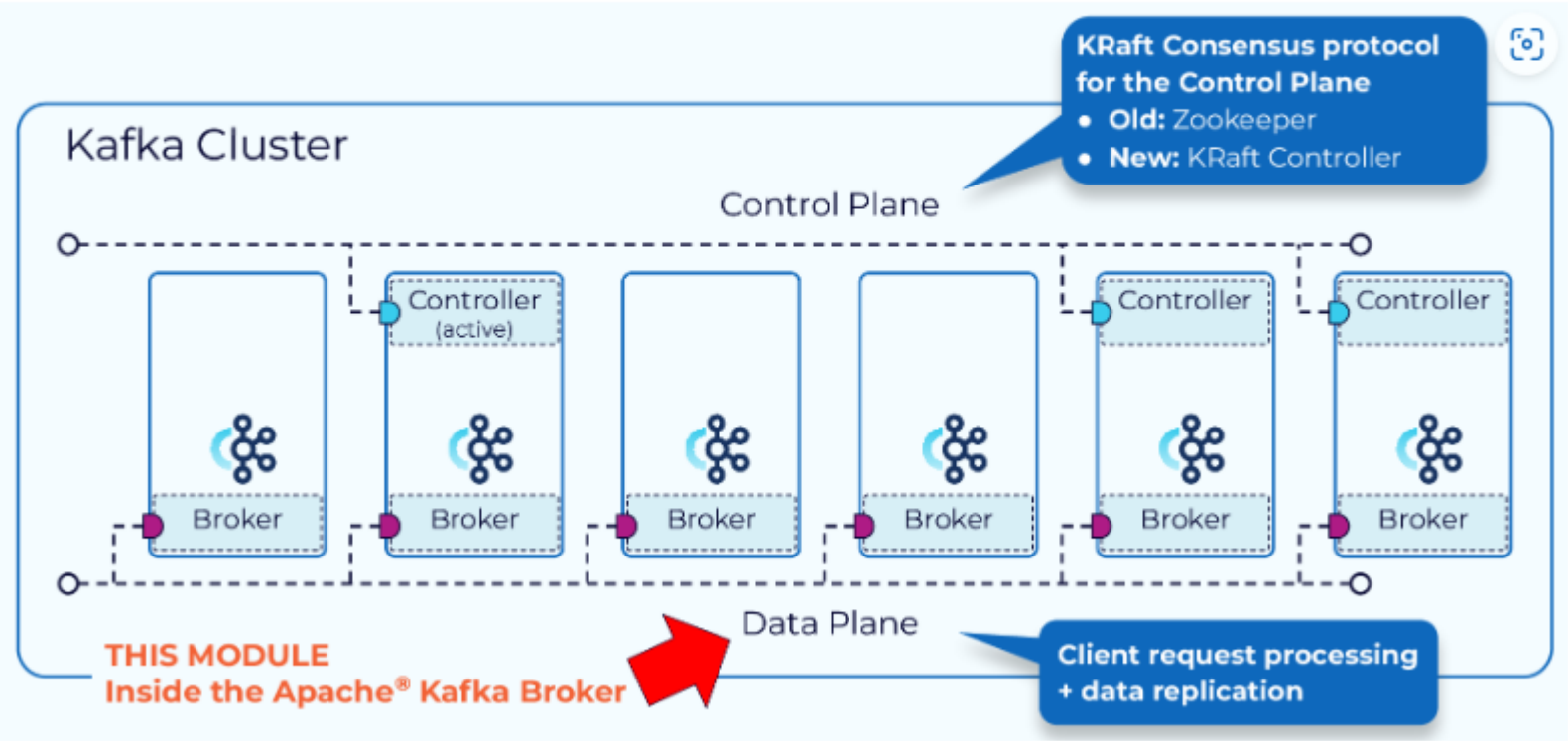
sending data

Development: A Basic Consumer in .NET/C#

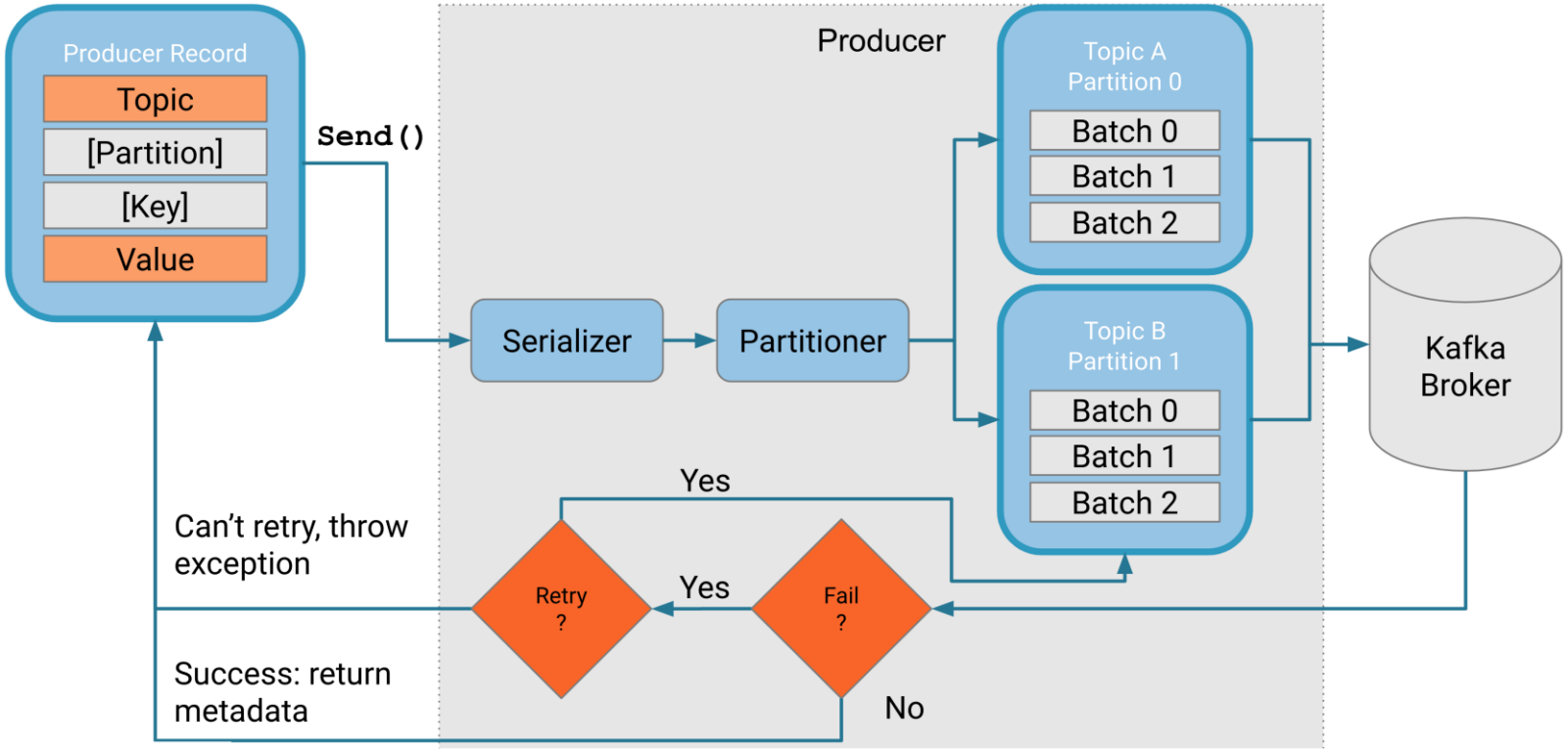



```
namespace consumer_net {
    0 references
    class Program {
        0 references
        static void Main (string[] args) {
            Console.WriteLine ("Starting Consumer!");
            var config = new Dictionary<string, object> {
                { "group.id", "dotnet-consumer-group" },
                { "bootstrap.servers", "kafka-1:9092" },
                { "auto.commit.interval.ms", 5000 },
                { "auto.offset.reset", "earliest" }
            };

            var deserializer = new StringDeserializer (Encoding.UTF8);
            using (var consumer = new Consumer<string, string> (config, deserializer, deserializer)) {
                consumer.OnMessage += (_, msg) =>
                  Console.WriteLine ($"Read ('{msg.Key}', '{msg.Value}') from: {msg.TopicPartitionOffset}");

                consumer.OnError += (_, error) =>
                  Console.WriteLine ($"Error: {error}");

                consumer.OnConsumeError += (_, msg) =>
                  Console.WriteLine ($"Consume error ({msg.TopicPartitionOffset}): {msg.Error}");

                consumer.Subscribe ("hello_world_topic");

                while (true) {
                    consumer.Poll (TimeSpan.FromMilliseconds (100));
                }
            }
        }
    }
}
```

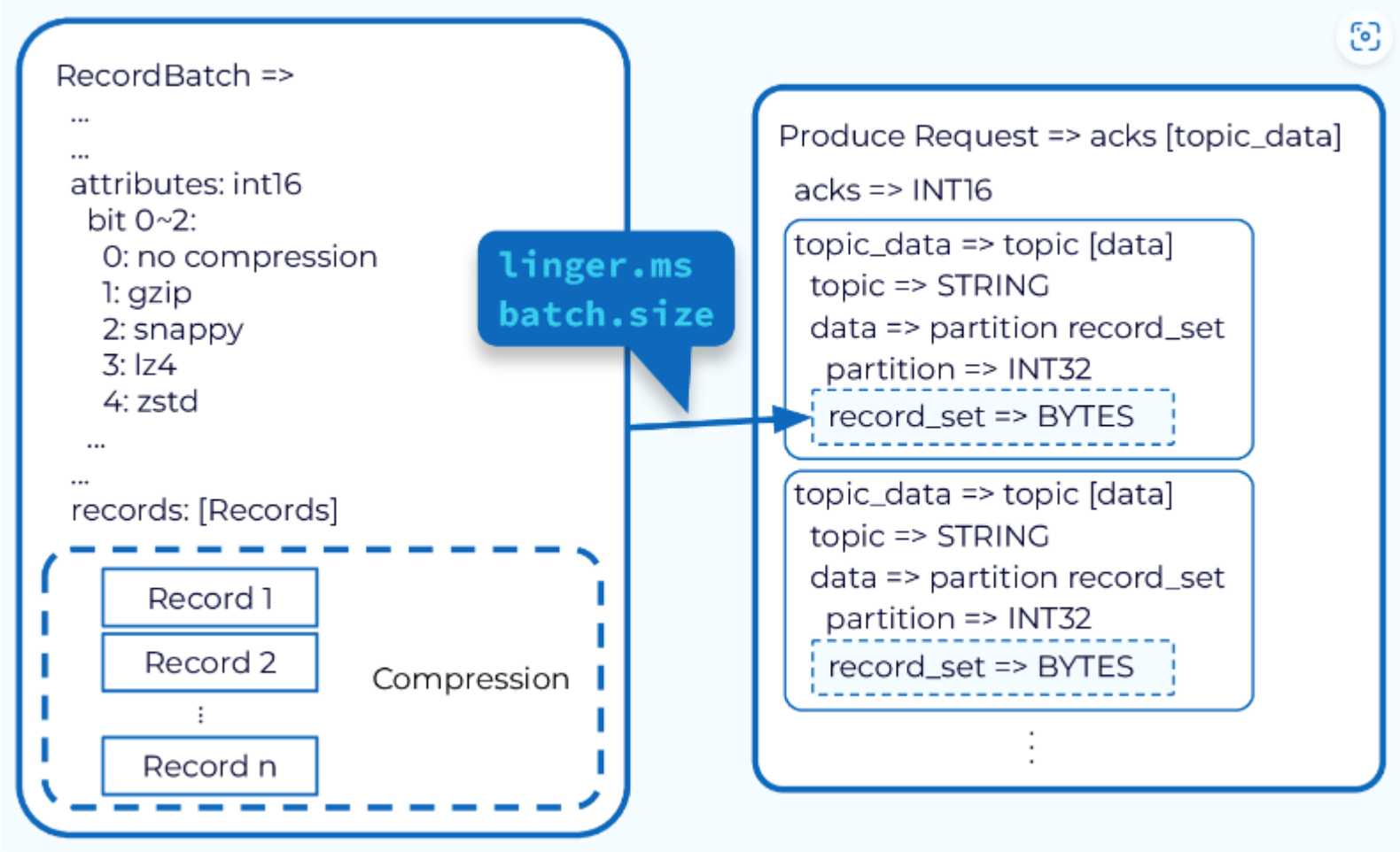
Kafka Manages Data and Metadata Separately



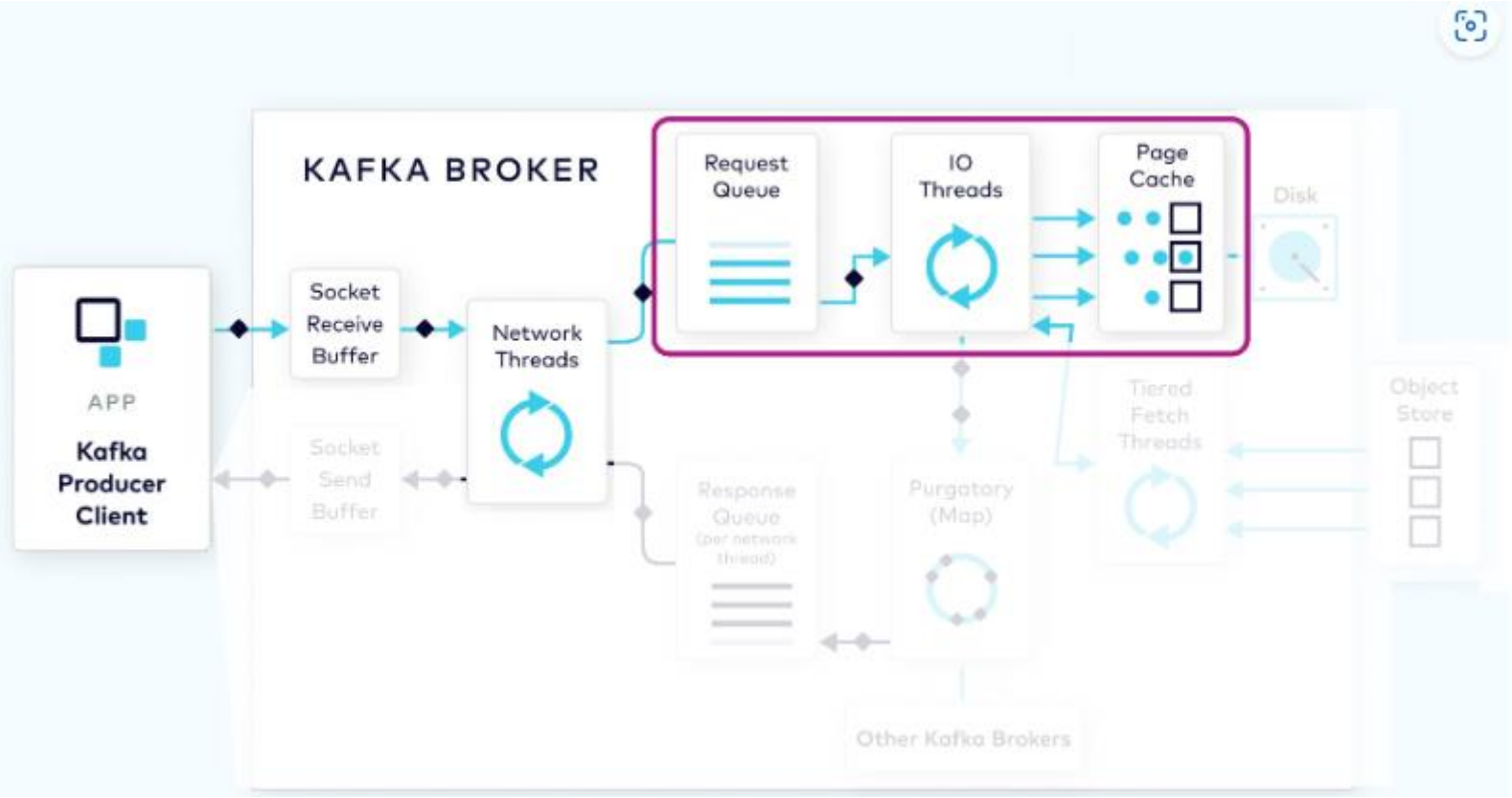
- The functions within a Kafka cluster are broken up into a data plane and a control plane.
- The control plane handles management of all the metadata in the cluster.
- The data plane deals with the actual data that are writing to and reading from Kafka.



When a producer is ready to send an event record, it will use a configurable partitioner to determine the topic partition to assign to the record. If the record has a key, then the default partitioner will use a hash of the key to determine the correct partition. After that, any records with the same key will always be assigned to the same partition.
Sending records one at a time would be inefficient due to the overhead of repeated network requests. So, the producer will accumulate the records assigned to a given partition into batches. Batching also provides for much more effective compression, when compression is used.



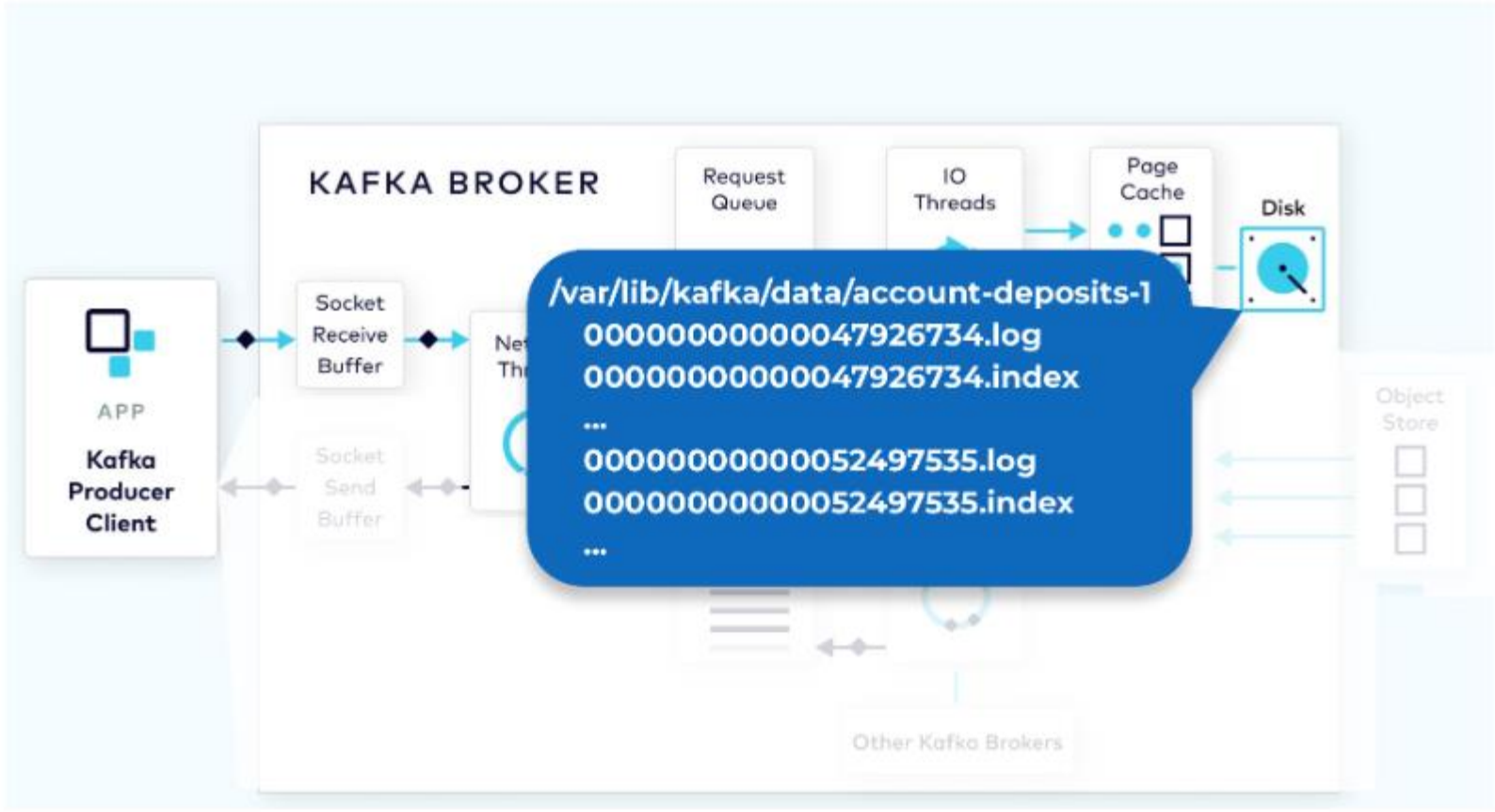
The producer also has control as to when the record batch should be drained and sent to the broker. This is controlled by two properties. One is by time(linger.ms). The other is by size(batch.size). So once enough time or enough data has been accumulated in those record batches, those record batches will be drained, and will form a produce request. And this produce request will then be sent to the broker that is the leader of the included partitions.



The request first lands in the broker's socket receive buffer where it will be picked up by a network thread from the pool. That network thread will handle that particular client request through the rest of its lifecycle. The network thread will read the data from the socket buffer, form it into a produce request object, and add it to the request queue.

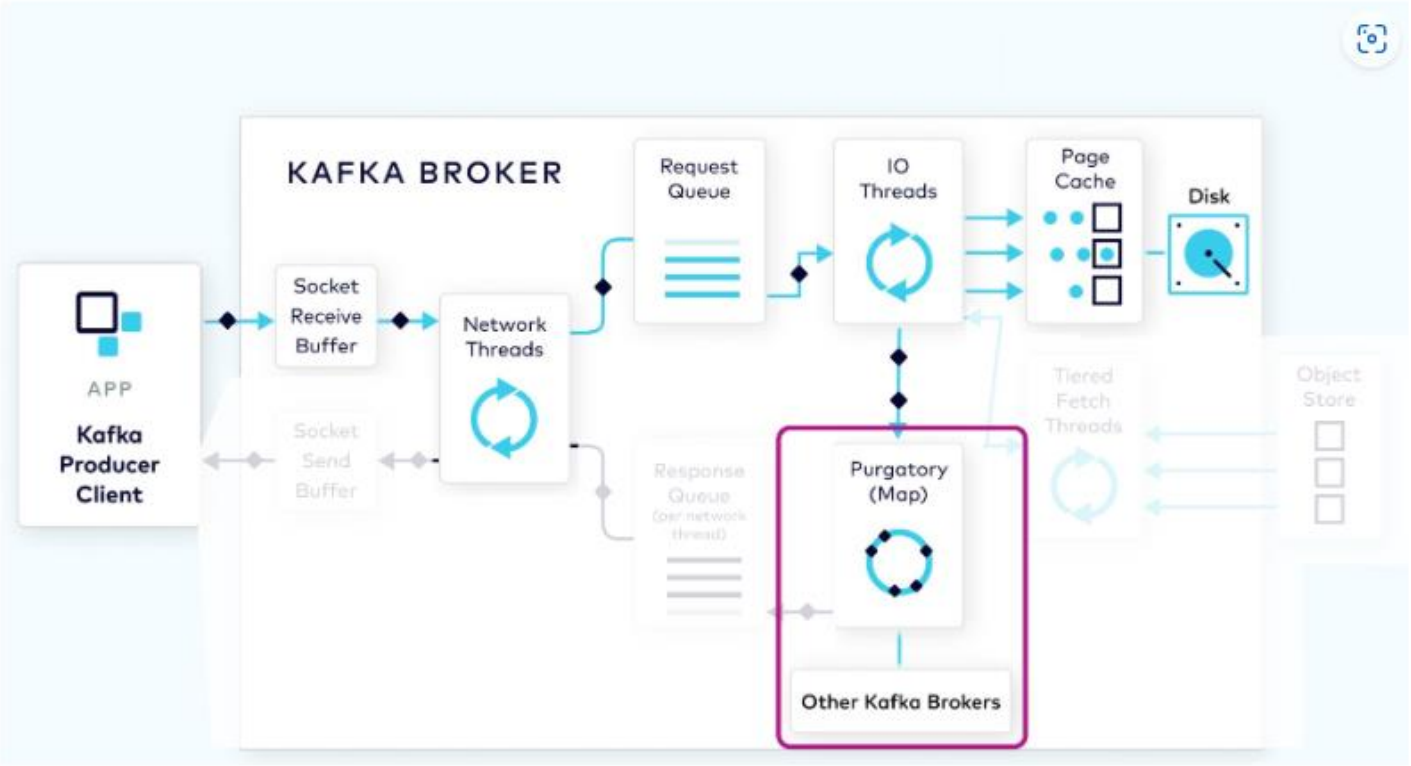
Next, a thread from the I/O thread pool will pick up the request from the queue. The I/O thread will perform some validations, including a CRC check of the data in the request. It will then append the data to the physical data structure of the partition, which is called a commit log.

Kafka Physical Storage



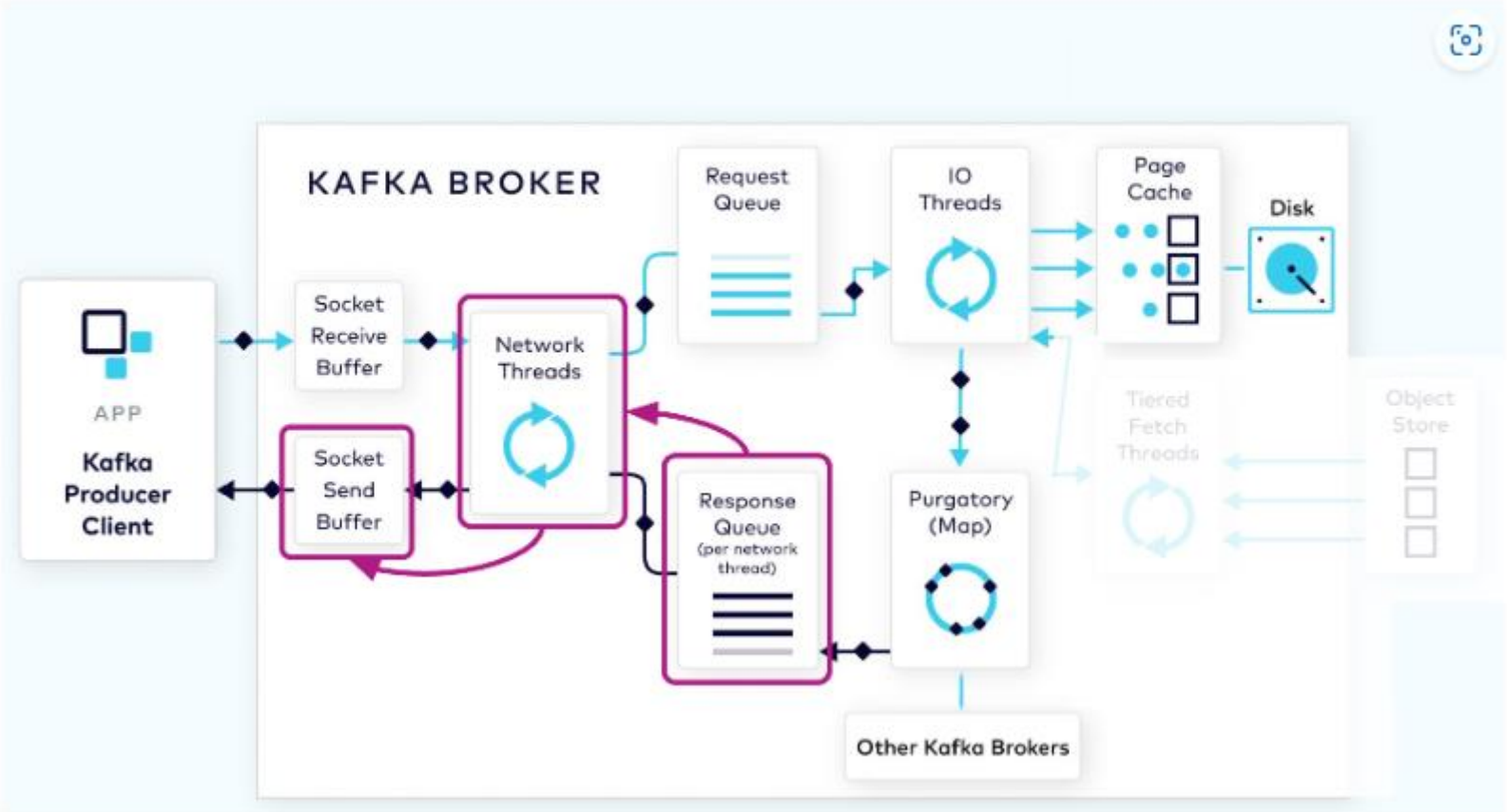
On disk, the commit log is organized as a collection of segments.
Each segment is made up of several files. One of these, a .log file, contains the event data.
A second, a .index file, contains an index structure, which maps from a record offset to the position of that record in the .log file.

Purgatory Holds Requests Until Replicated



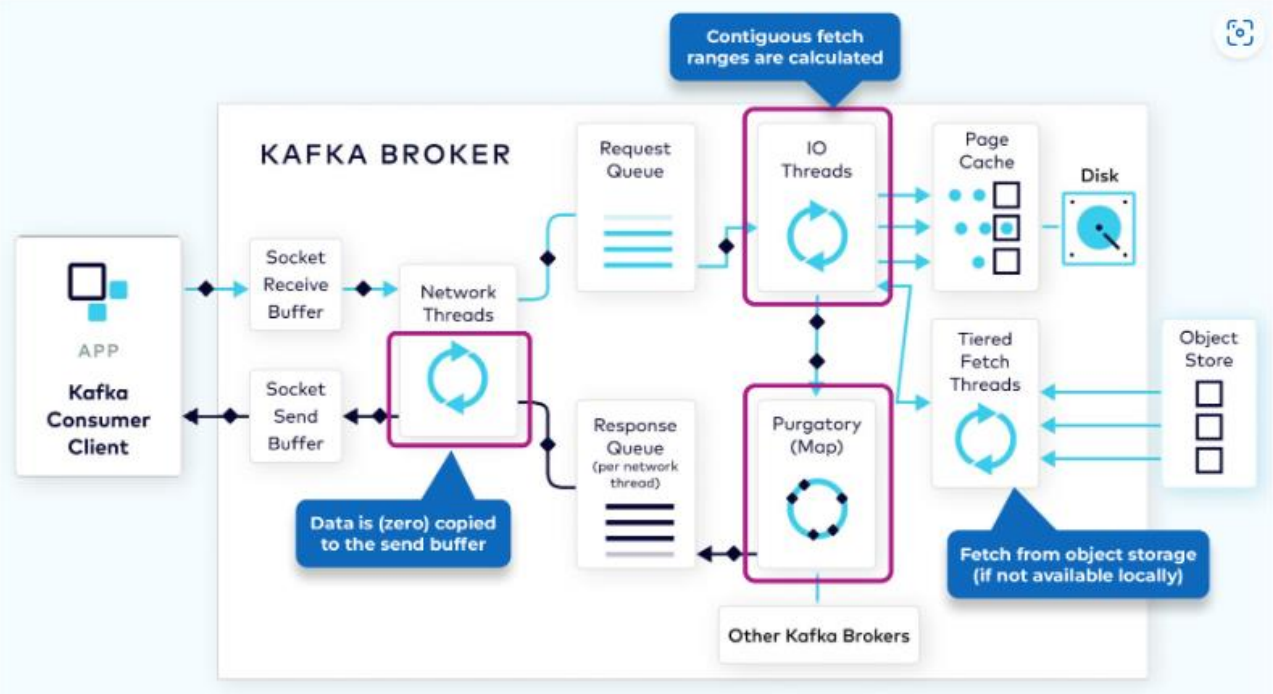
Since the log data is not flushed from the page cache to disk synchronously, Kafka relies on replication to multiple broker nodes, in order to provide durability. By default, the broker will not acknowledge the produce request until it has been replicated to other brokers.
To avoid tying up the I/O threads while waiting for the replication step to complete, the request object will be stored in a map-like data structure called purgatory (it's where things go to wait).
Once the request has been fully replicated, the broker will take the request object out of purgatory, generate a response object, and place it on the response queue.

Response Added to Socket



From the response queue, the network thread will pick up the generated response, and send its data to the socket send buffer. The network thread also enforces ordering of requests from an individual client by waiting for all of the bytes for a response from that client to be sent before taking another object from the response queue.

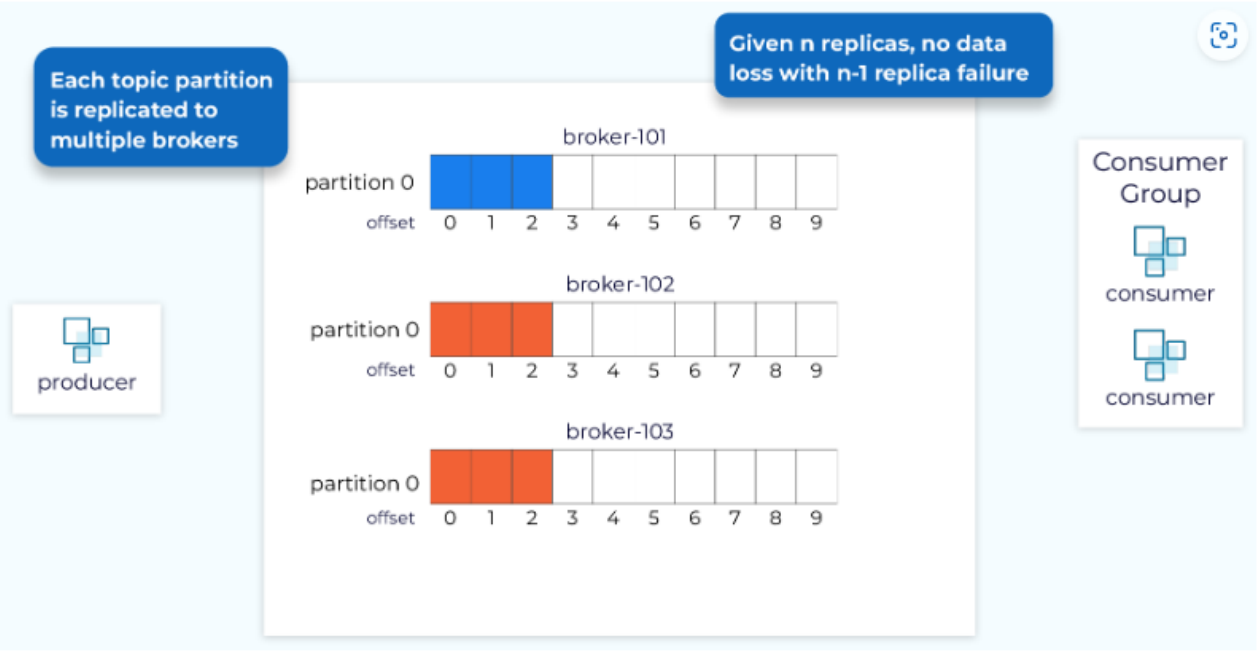
The Fetch Request



In order to consume records, a consumer client sends a fetch request to the broker, specifying the topic, partition, and offset it wants to consume. The fetch request goes to the broker's socket receive buffer where it is picked up by a network thread. The network thread puts the request in the request queue, as was done with the produce request.The I/O thread will take the offset that is included in the fetch request and compare it with the .index file that is part of the partition segment. That will tell it exactly the range of bytes that need to be read from the corresponding .log file to add to the response object.

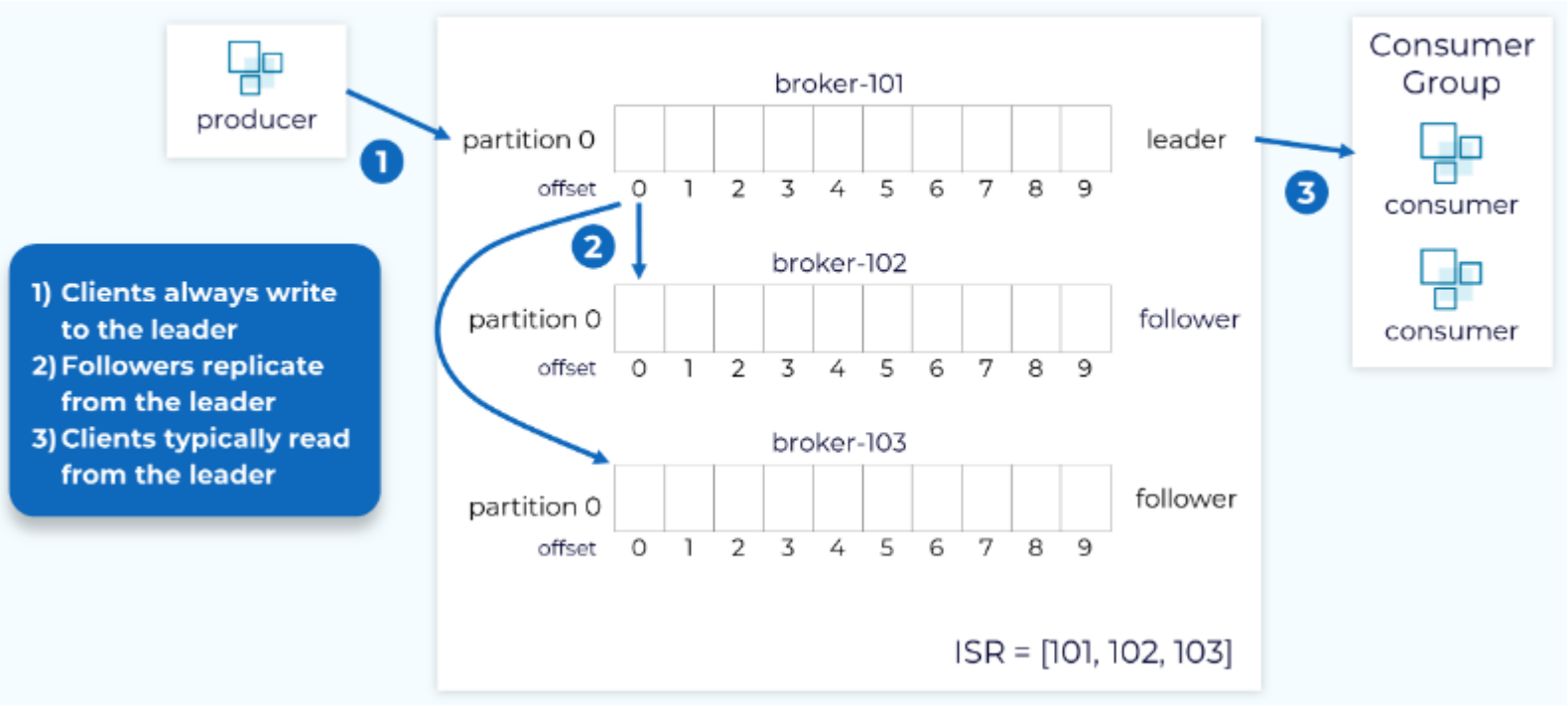
However, it would be inefficient to send a response with every record fetched, or even worse, when there are no records available. To be more efficient, consumers can be configured to wait for a minimum number of bytes of data(fetch.min.bytes
), or to wait for a maximum amount of time(fetch.max.wait.ms) before returning a response to a fetch request. While

Kafka Data Replication



When a new topic is created ,each partition of that topic will be replicated that many times. This number is referred to as the replication factor. With a replication factor of N, in general, can tolerate N-1 failures, without data loss, and while maintaining availability.

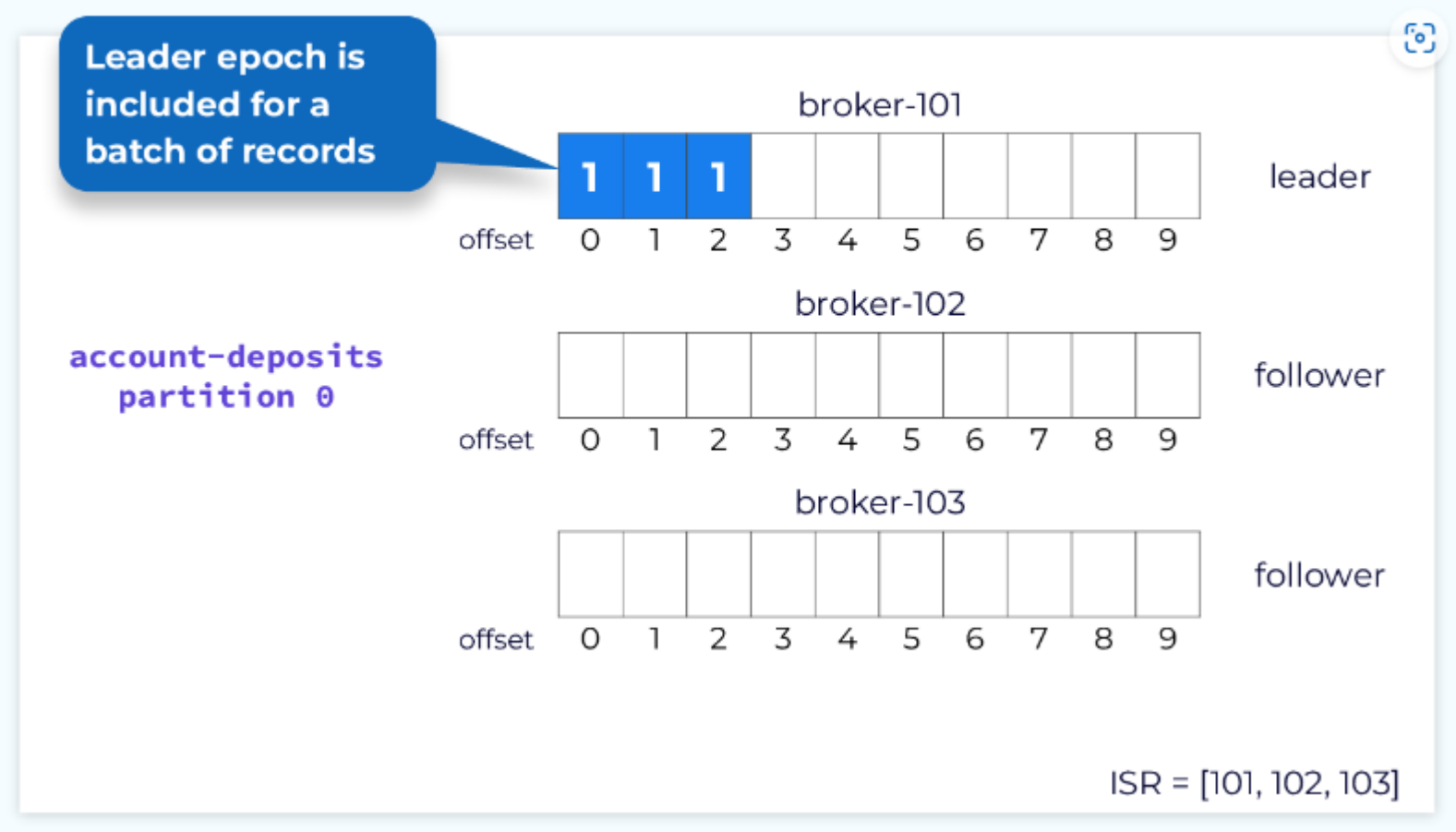
Leader, Follower, and In-Sync Replica (ISR) List



Once the replicas for all the partitions in a topic are created, one replica of each partition will be designated as the leader replica and the broker that holds that replica will be the leader for that partition. The remaining replicas will be followers. Producers will write to the leader replica and the followers will fetch the data in order to keep in sync with the leader. Consumers also, generally, fetch from the leader replica, but they can be configured to fetch from followers.

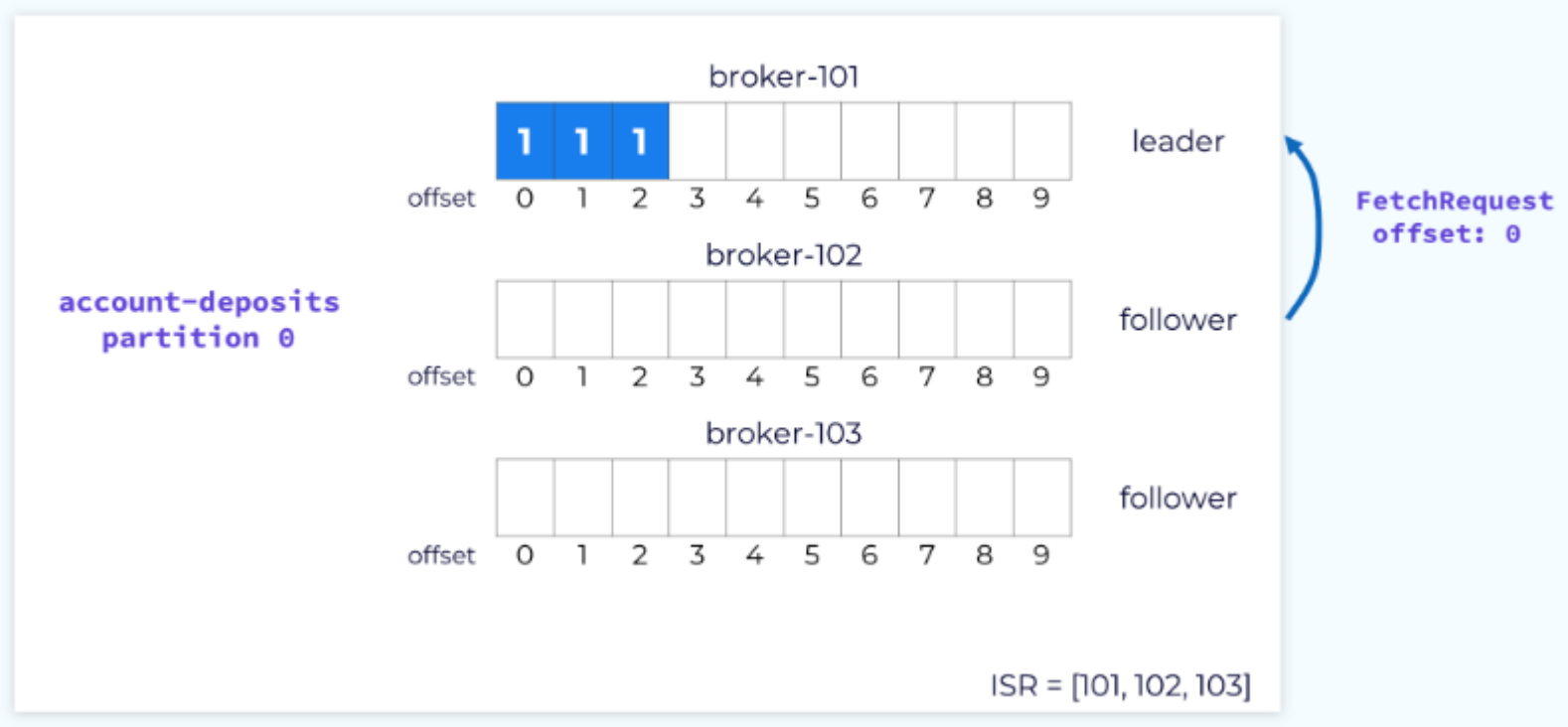
The partition leader, along with all of the followers that have caught up with the leader, will be part of the in-sync replica set (ISR). In the ideal situation, all of the replicas will be part of the ISR.

Leader Epoch



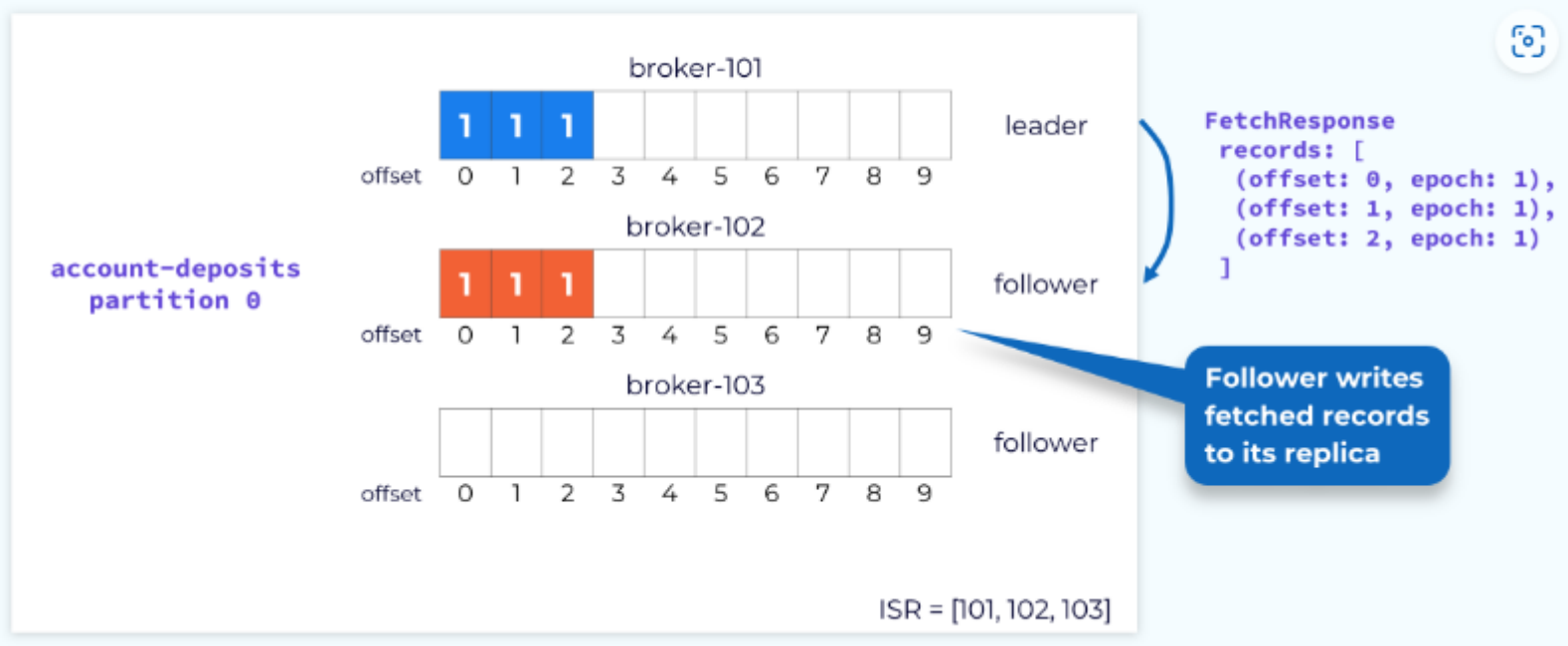
Each leader is associated with a unique, monotonically increasing number called the leader epoch. The epoch is used to keep track of what work was done while this replica was the leader and it will be increased whenever a new leader is elected. The leader epoch is very important for things like log reconciliation

Follower Fetch Request



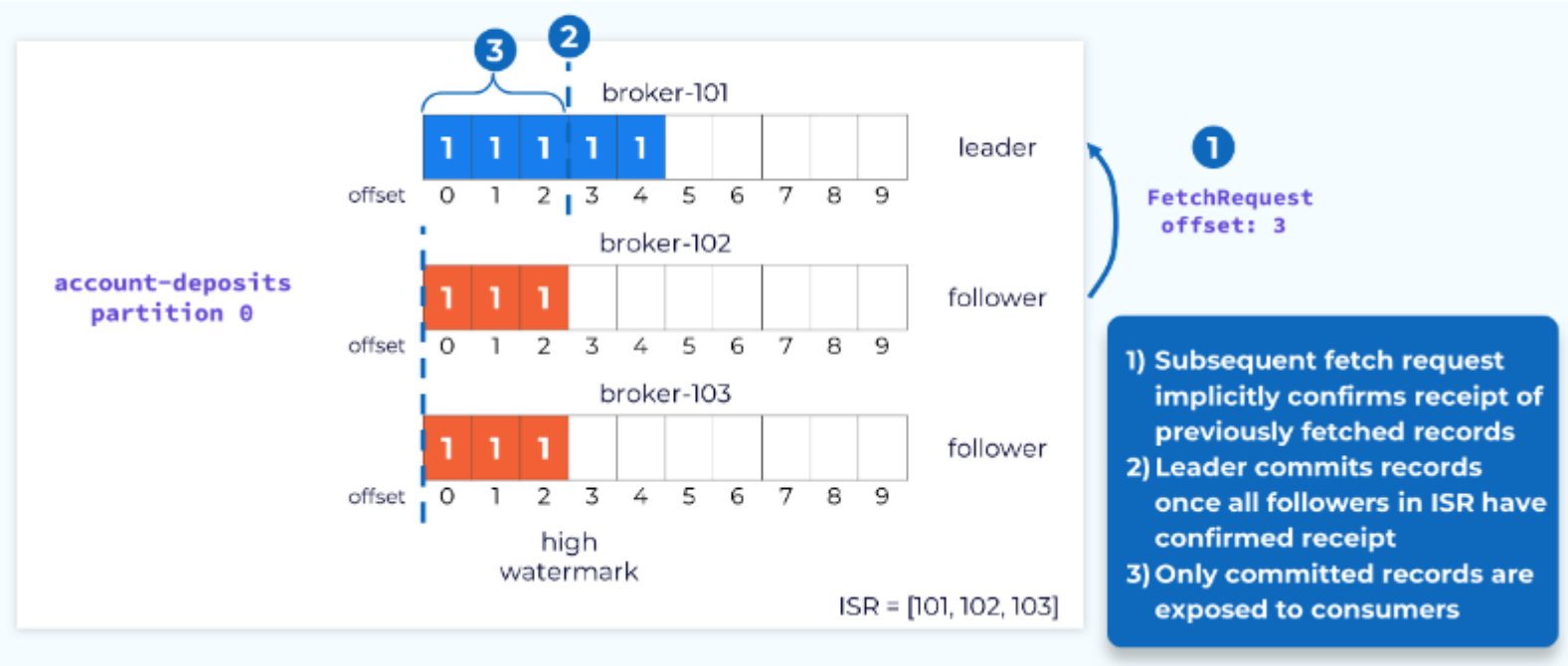
Whenever the leader appends new data into its local log, the followers will issue a fetch request to the leader, passing in the offset at which they need to begin fetching.

Follower Fetch Response



The leader will respond to the fetch request with the records starting at the specified offset.
The fetch response will also include the offset for each record and the current leader epoch.
The followers will then append those records to their own local logs.

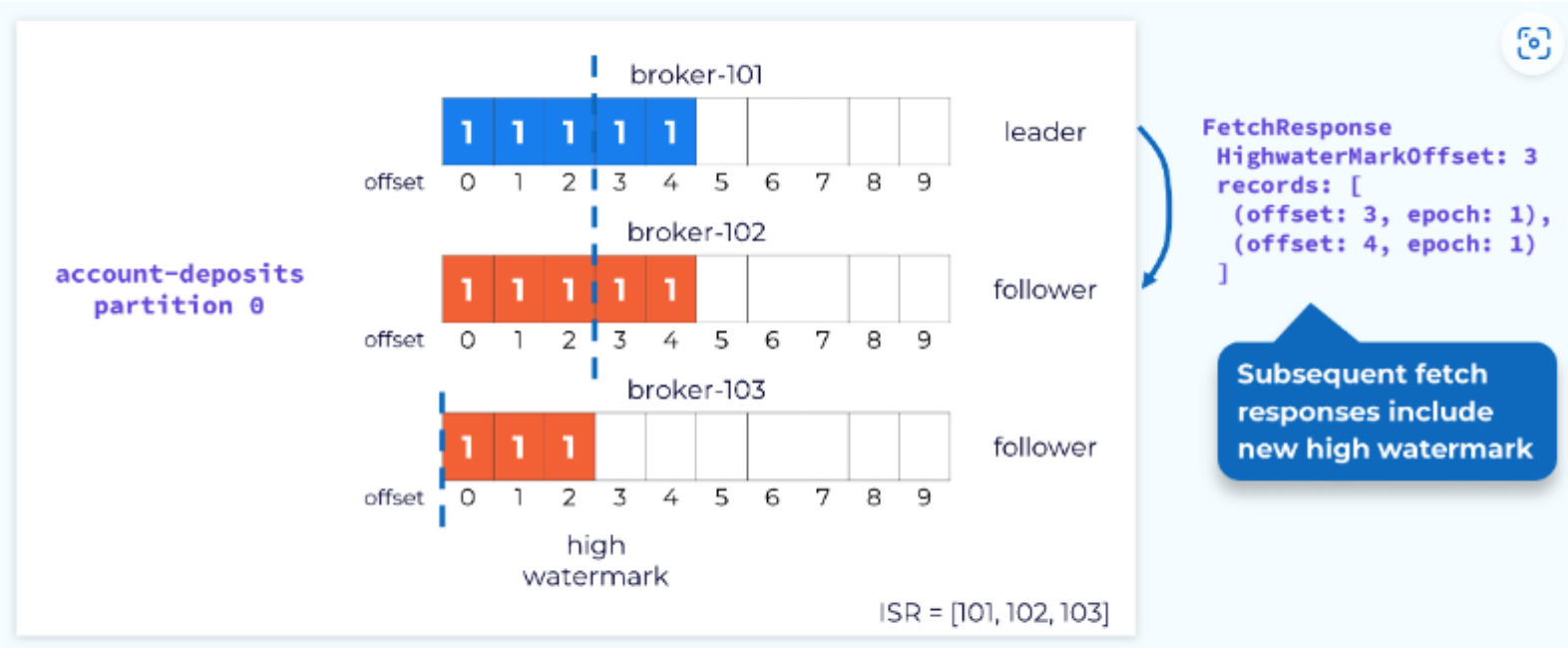
Committing Partition Offsets



Once all of the followers in the ISR have fetched up to a particular offset, the records up to that offset are considered committed and are available for consumers. This is designated by the high watermark.

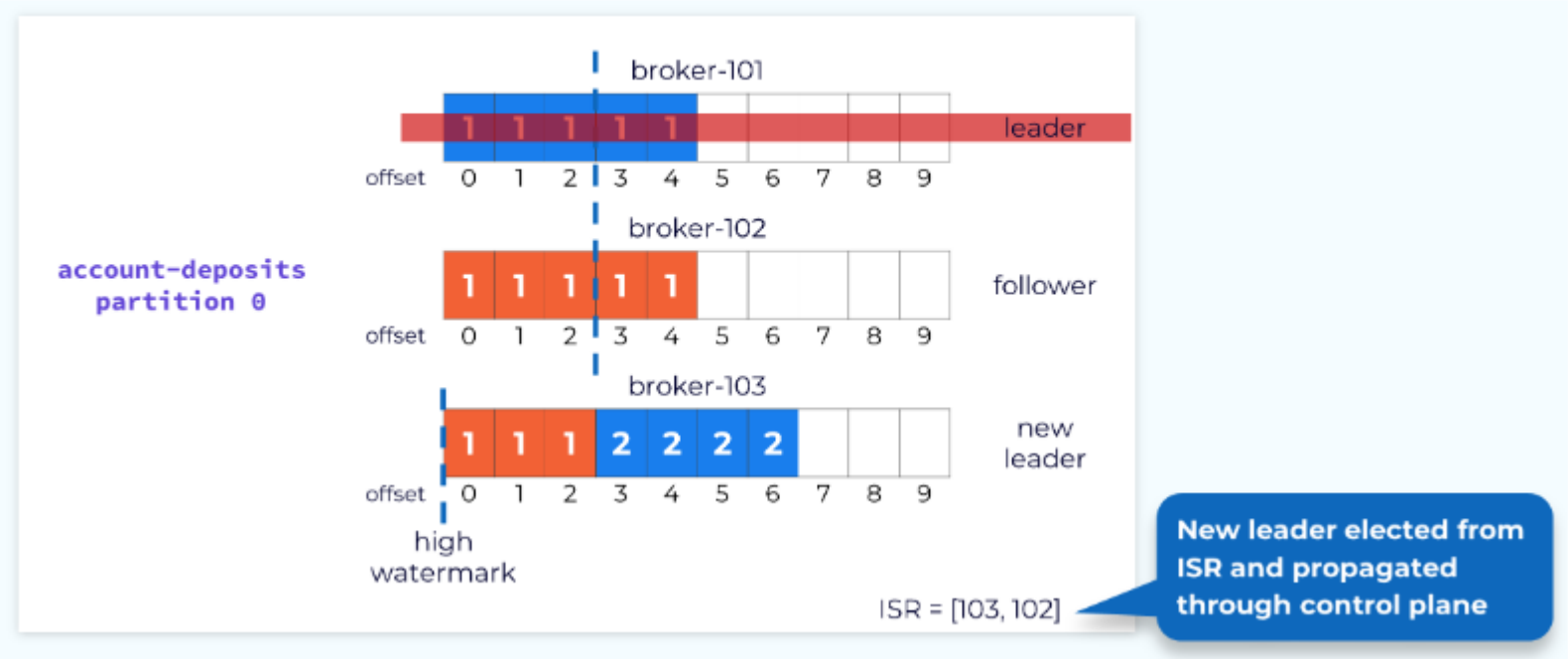
The leader is made aware of the highest offset fetched by the followers through the offset value sent in the fetch requests. For example, if a follower sends a fetch request to the leader that specifies offset 3, the leader knows that this follower has committed all records up to offset 3. Once all of the followers have reached offset 3, the leader will advance the high watermark accordingly.

Advancing the Follower High Watermark



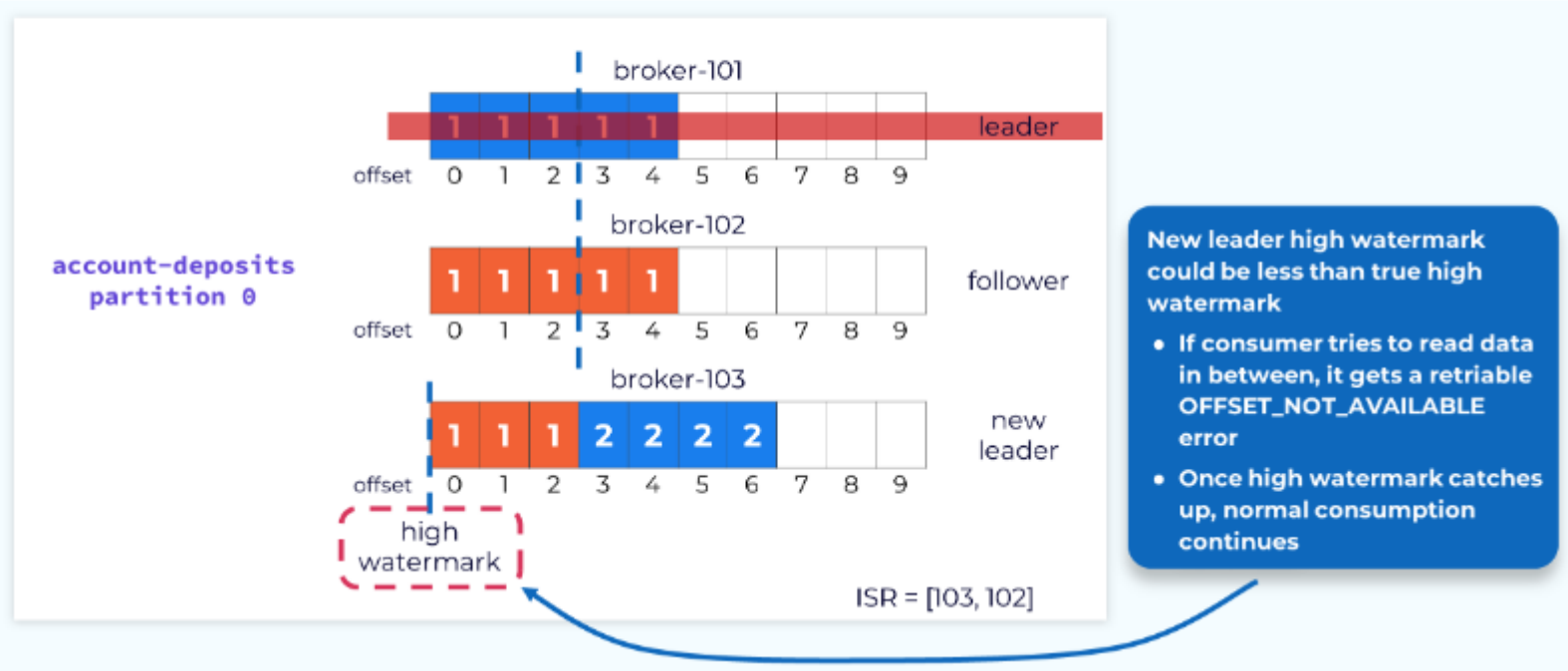
The leader, in turn, uses the fetch response to inform followers of the current high watermark. Because this process is asynchronous, the followers' high watermark will typically lag behind the actual high watermark held by the leader.

Handling Leader Failure



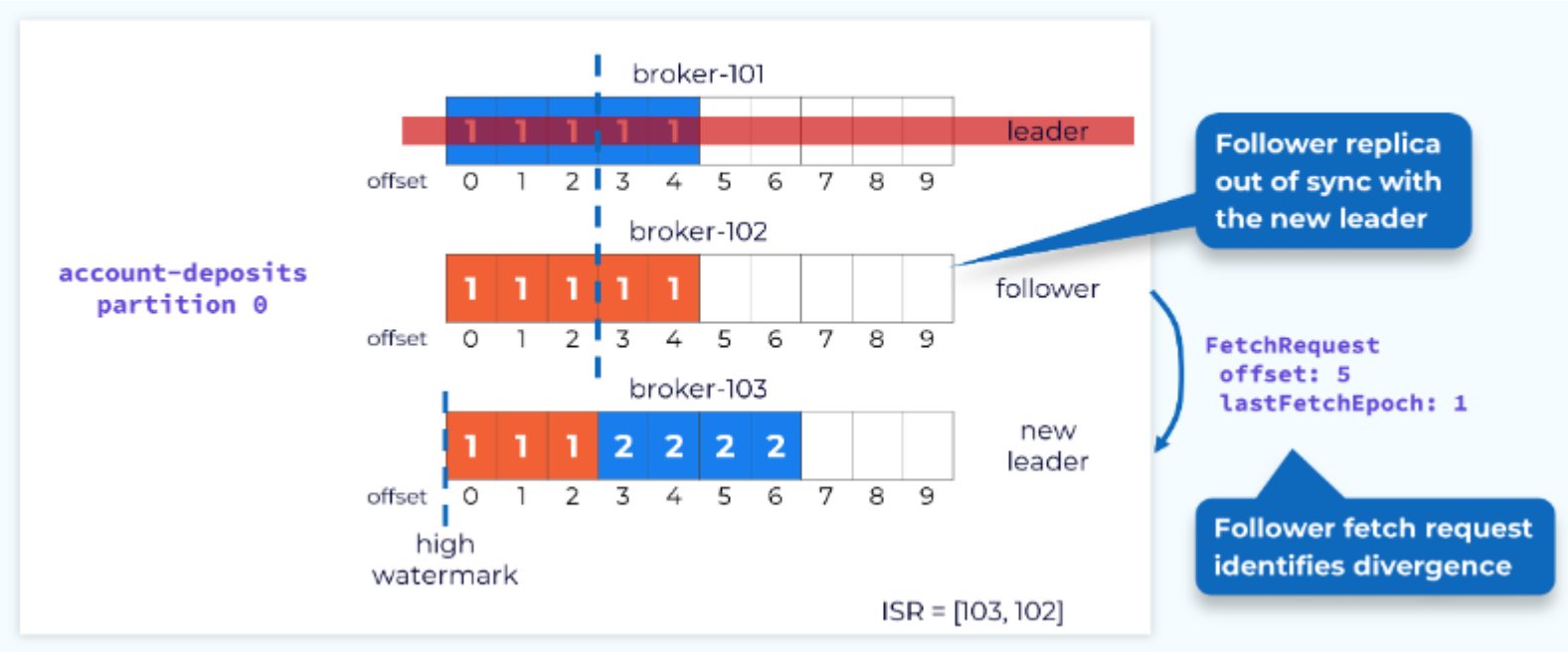
If a leader fails, or if for some other reason ,need to choose a new leader, one of the brokers in the ISR will be chosen as the new leader. The process of leader election and notification of affected followers is handled by the control plane. The important thing for the data plane is that no data is lost in the process. That is why a new leader can only be selected from the ISR, unless the topic has been specifically configured to allow replicas that are not in sync to be selected. We know that all of the replicas in the ISR are up to date with the latest committed offset.
Once a new leader is elected, the leader epoch will be incremented and the new leader will begin accepting produce requests.

Temporary Decreased High Watermark



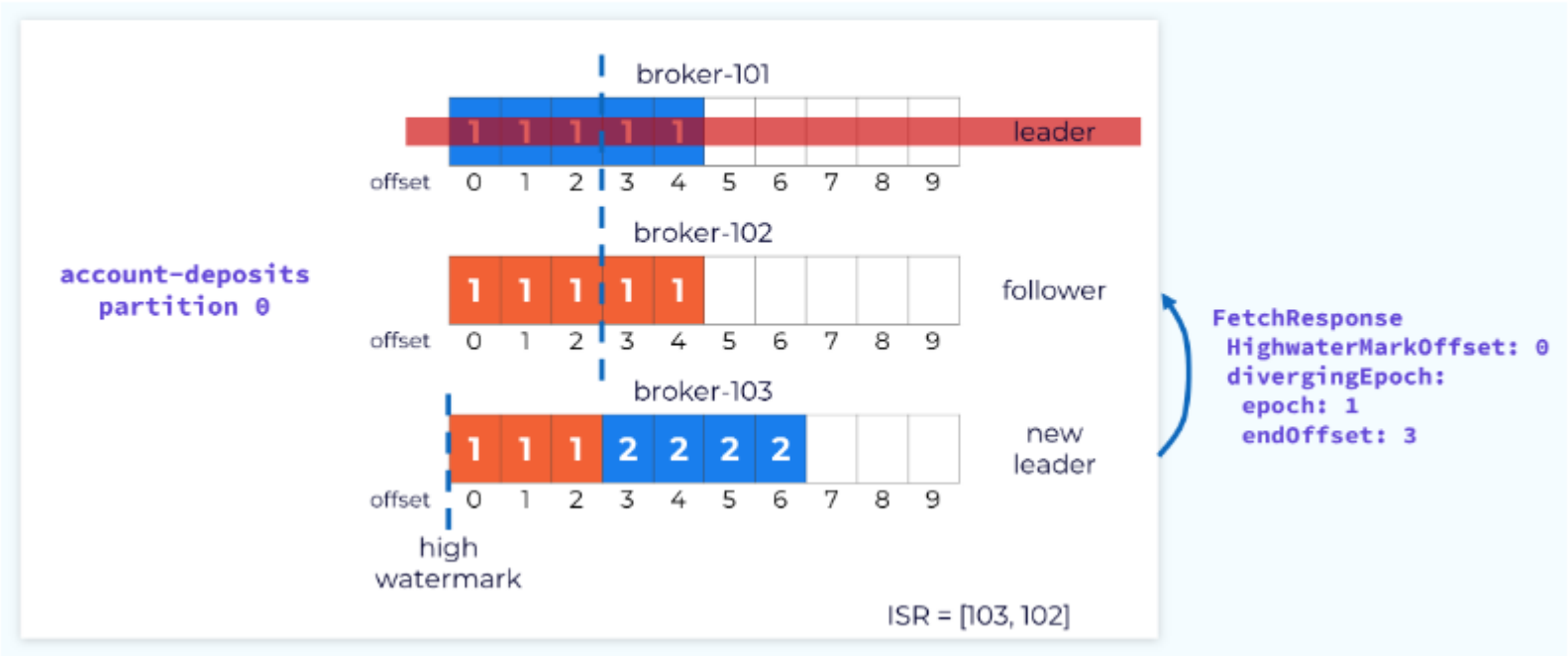
When a new leader is elected, its high watermark could be less than the actual high watermark. If this happens, any fetch requests for an offset that is between the current leader's high watermark and the actual will trigger a retriable OFFSET_NOT_AVAILABLE error. The consumer will continue trying to fetch until the high watermark is updated, at which point processing will continue as normal.

Partition Replica Reconciliation



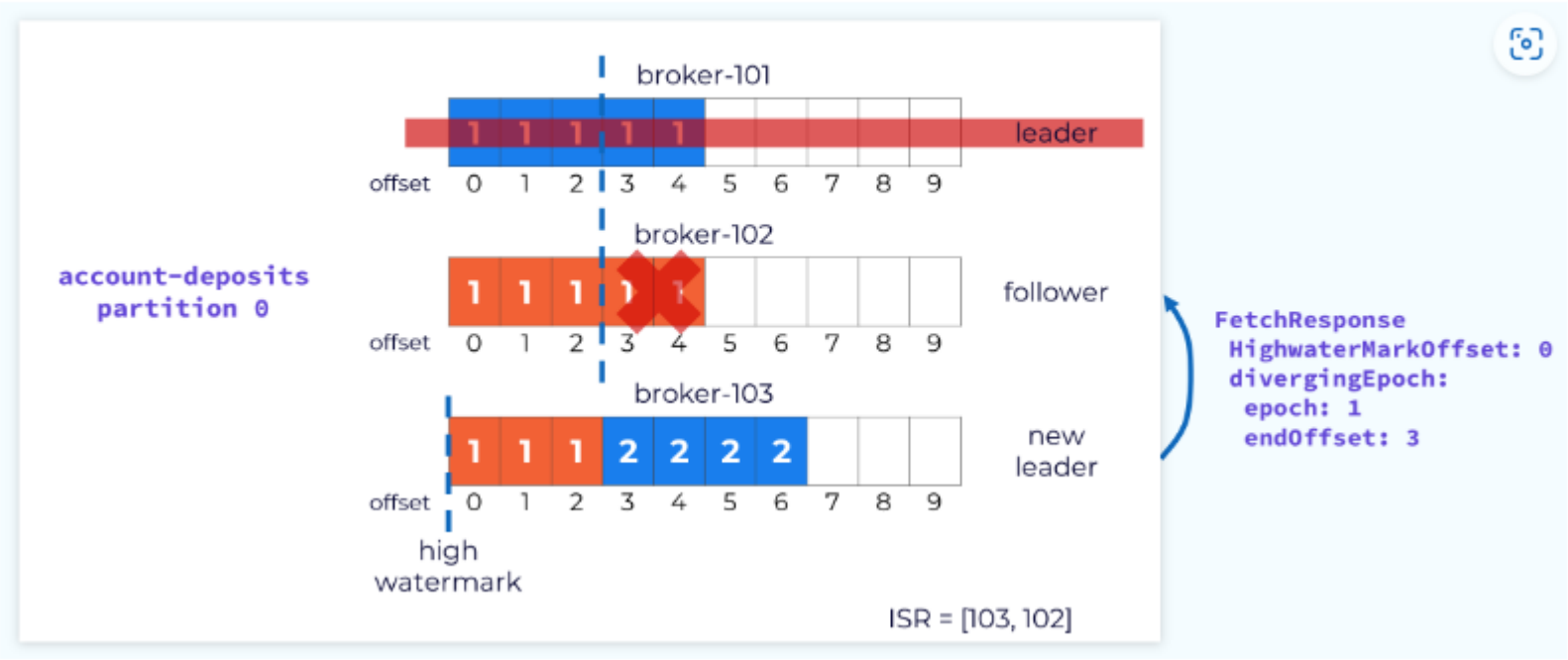
Immediately after a new leader election, it is possible that some replicas may have uncommitted records that are out of sync with the new leader. This is why the leader's high watermark is not current yet. It can't be until it knows the offset that each follower has caught up to. We can't move forward until this is resolved. This is done through a process called replica reconciliation. The first step in reconciliation begins when the out-of-sync follower sends a fetch request. In our example, the request shows that the follower is fetching an offset that is higher than the high watermark for its current epoch.

Fetch Response Informs Follower of Divergence



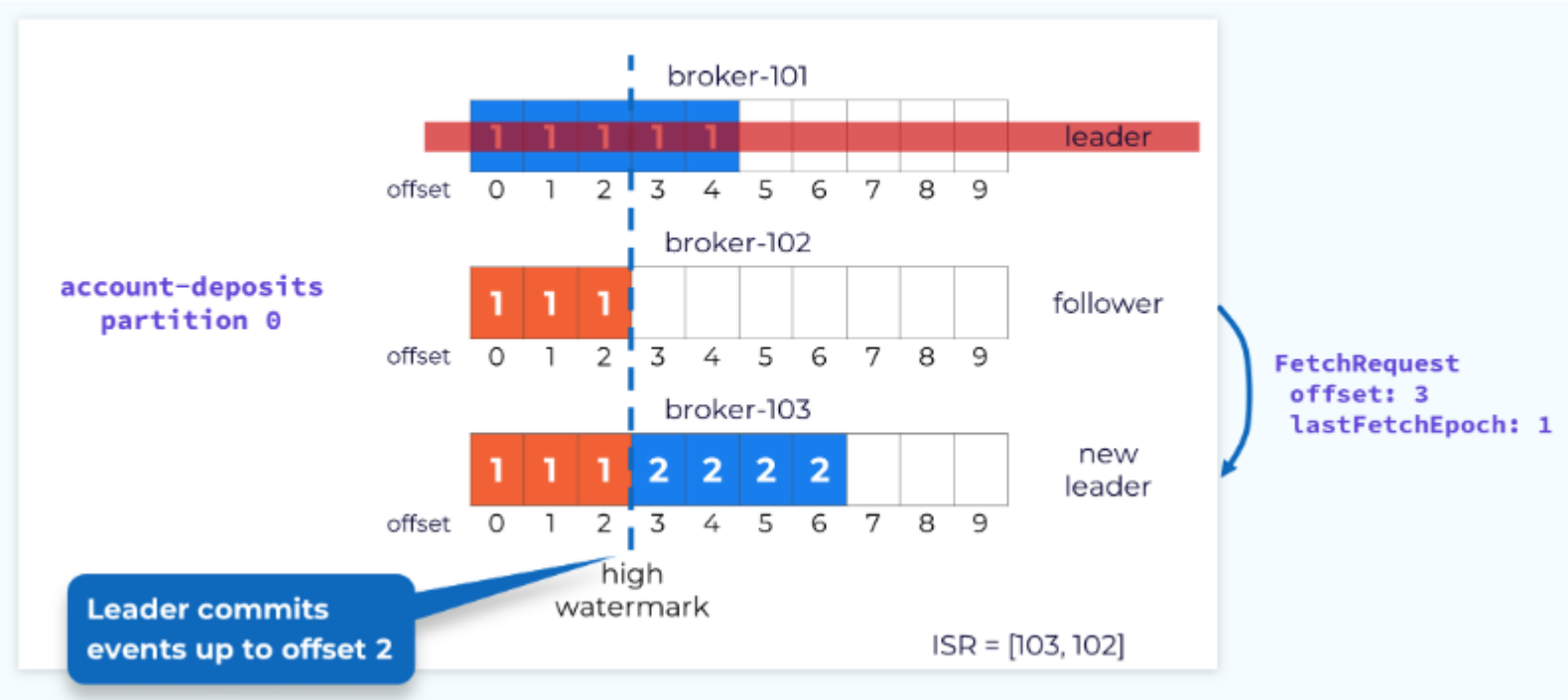
When the leader receives the fetch request it will check it against its own log and determine that the offset being requested is not valid for that epoch. It will then send a response to the follower telling it what offset that epoch should end at. The leader leaves it to the follower to perform the cleanup.

Follower Truncates Log to Match Leader Log



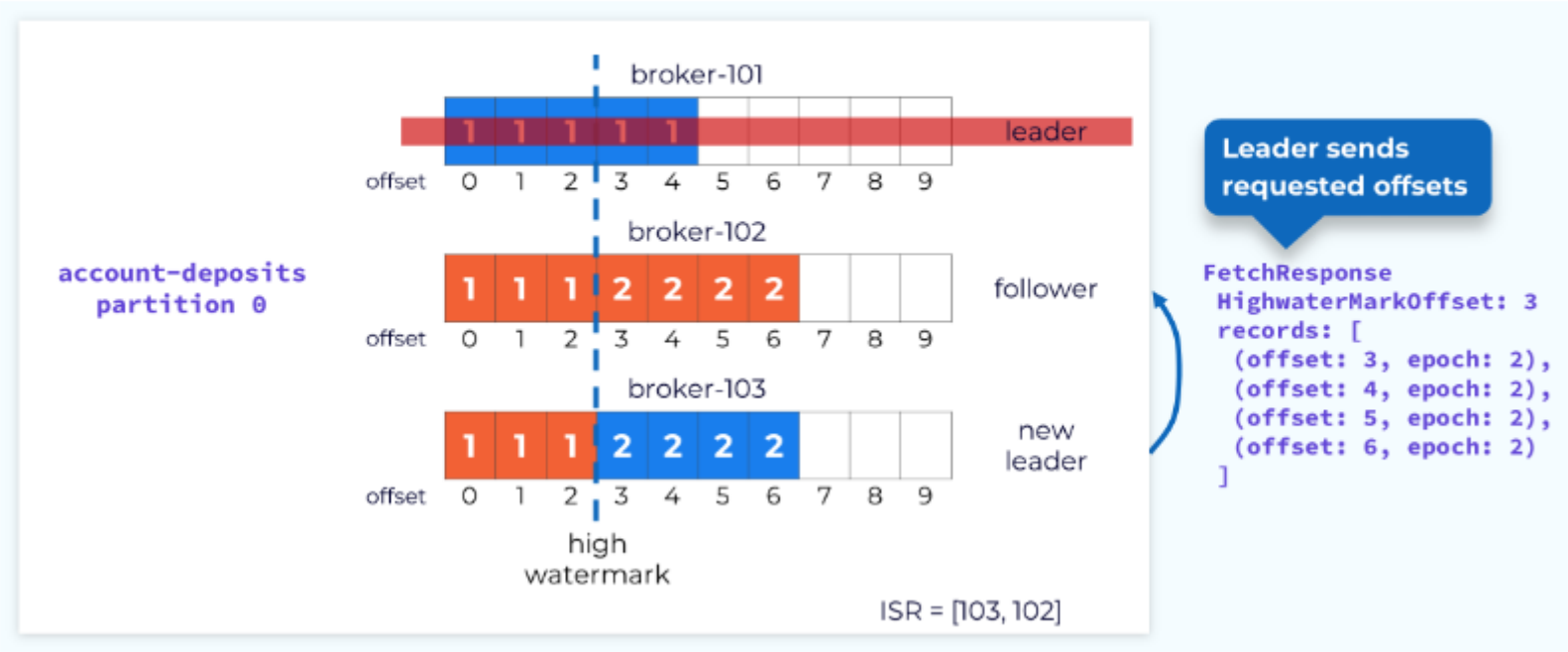
The follower will use the information in the fetch response to truncate the extraneous data so that it will be in sync with the leader.

Subsequent Fetch with Updated Offset and Epoch



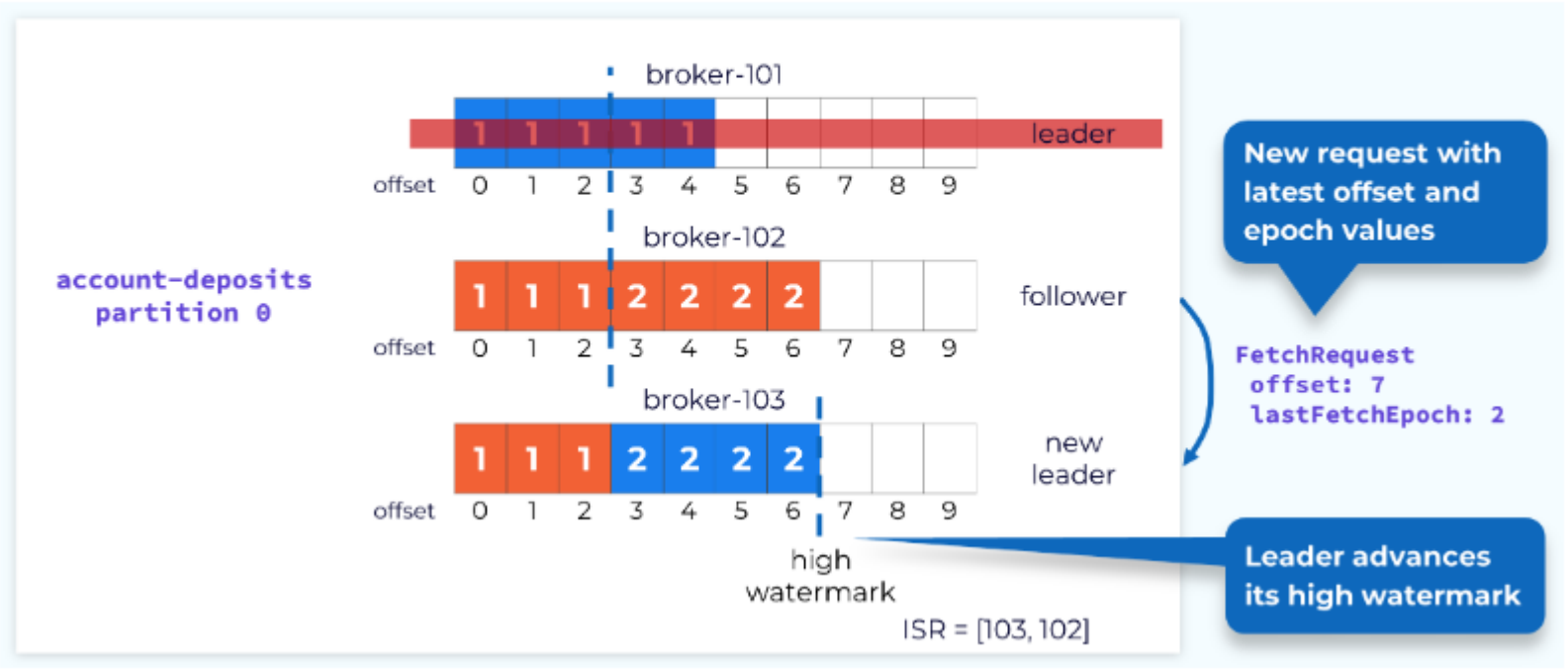
Now the follower can send that fetch request again, but this time with the correct offset.

Follower 102 Reconciled



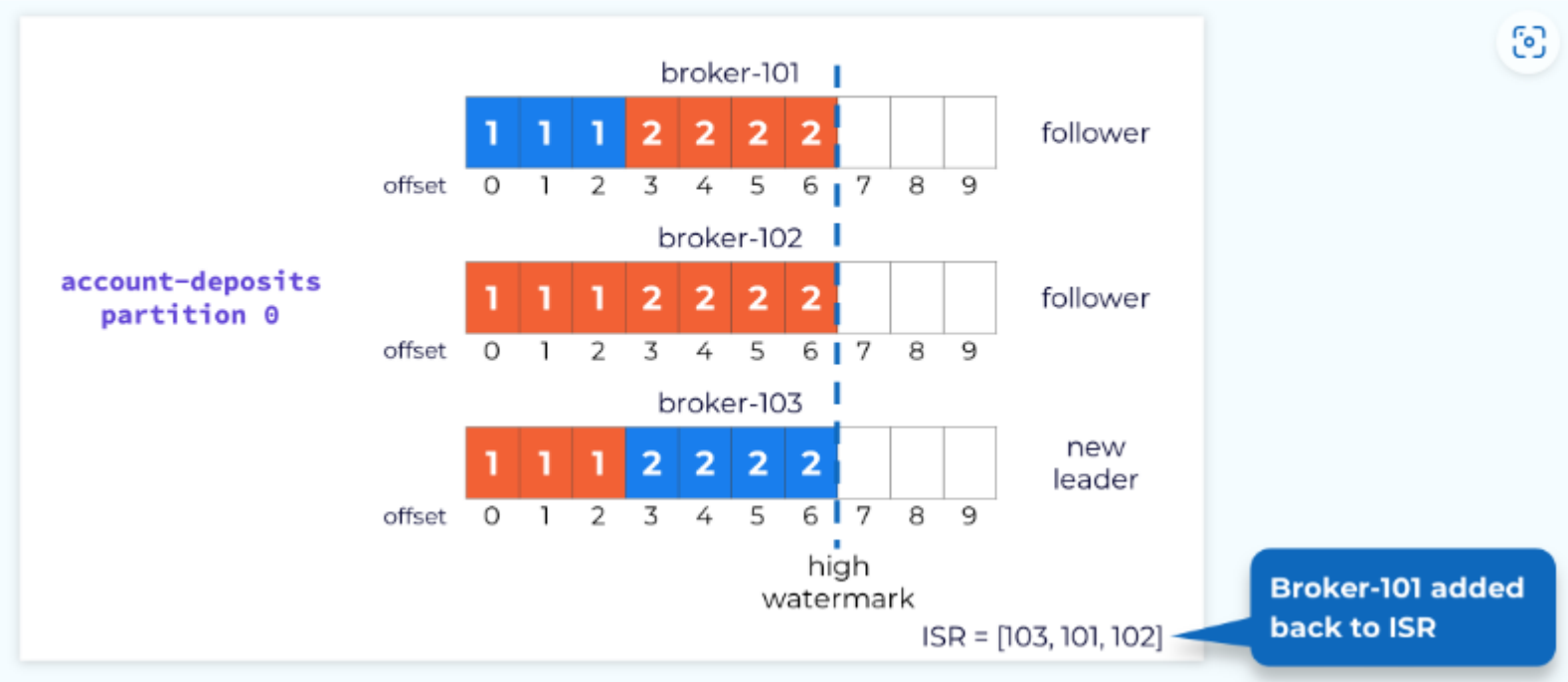
The leader will then respond with the new records since that offset includes the new leader epoch.

Follower 102 Acknowledges New Records



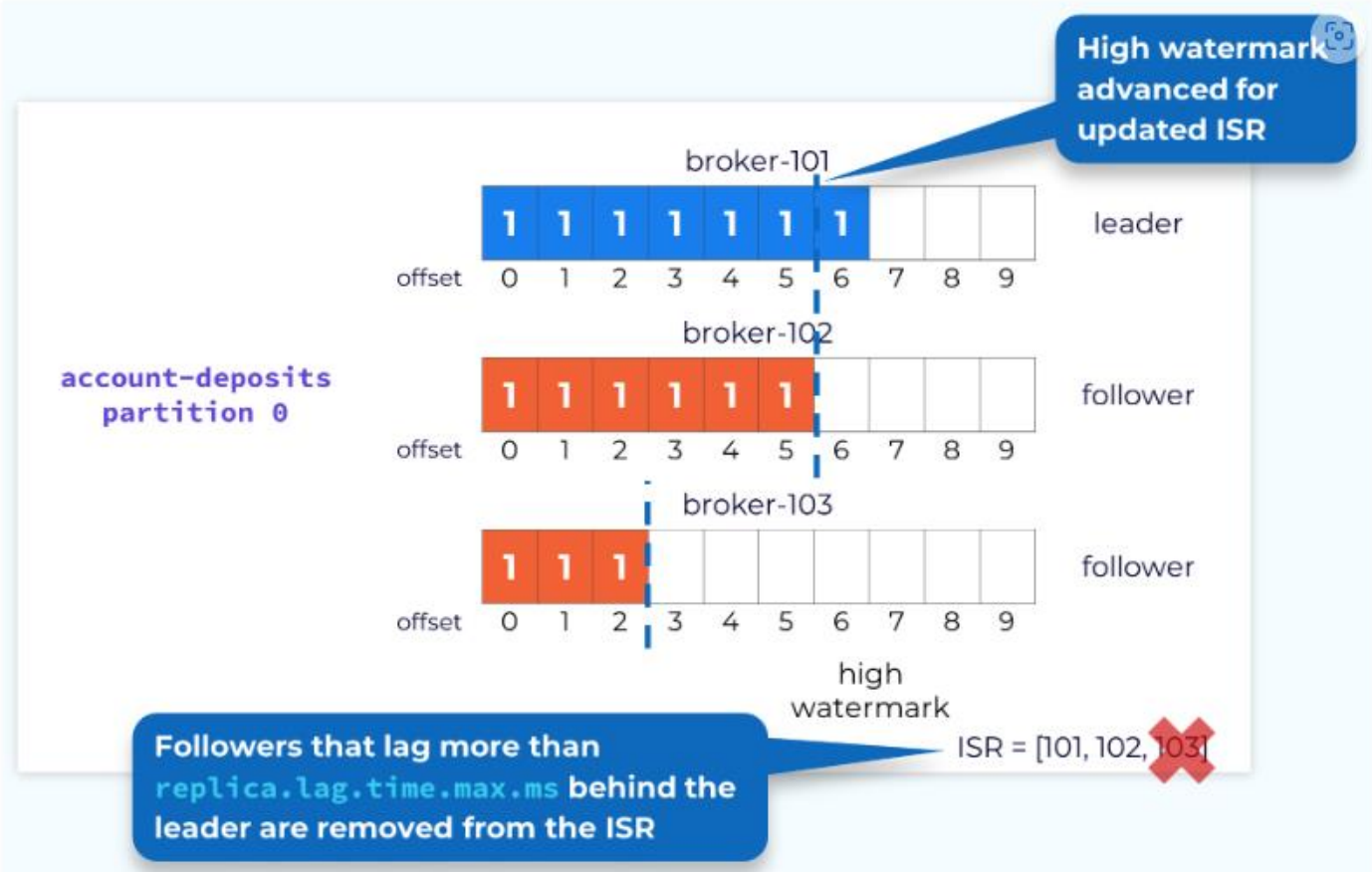
When the follower fetches again, the offset that it passes will inform the leader that it has caught up and the leader will be able to increase the high watermark. At this point the leader and follower are fully reconciled, but we are still in an under replicated state because not all of the replicas are in the ISR. Depending on configuration, we can operate in this state, but it's certainly not ideal.

Follower 101 Rejoins the Cluster



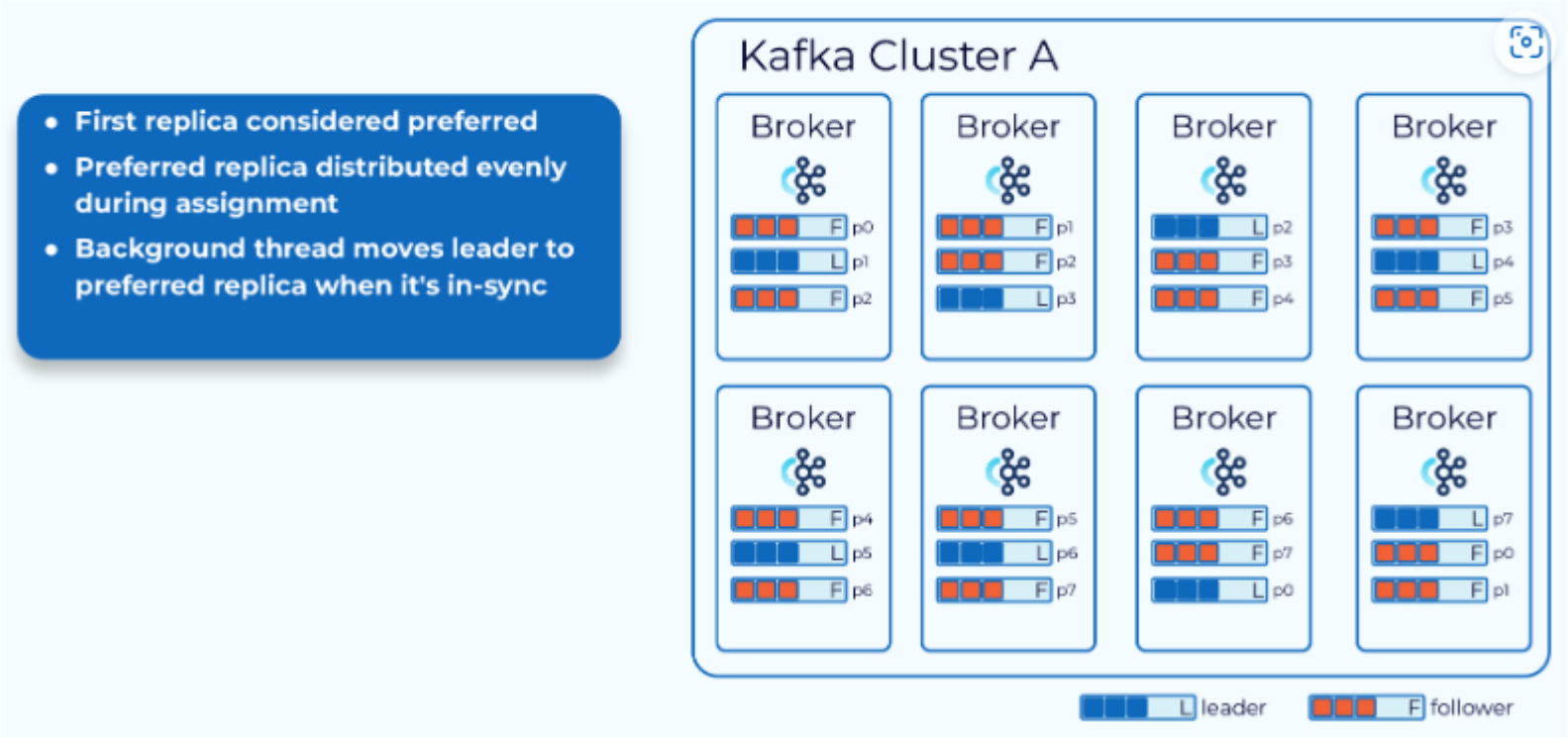
At some point, hopefully soon, the failed replica broker will come back online. It will then go through the same reconciliation process that we just described. Once it is done reconciling and is caught up with the new leader, it will be added back to the ISR and we will be back in our happy place.

Handling Failed or Slow Followers



Obviously when a leader fails, it's a bigger deal, but we also need to handle follower failures as well as followers that are running slow. The leader monitors the progress of its followers. If a configurable amount of time elapses since a follower was last fully caught up, the leader will remove that follower from the in-sync replica set. This allows the leader to advance the high watermark so that consumers can continue consuming current data. If the follower comes back online or otherwise gets its act together and catches up to the leader, then it will be added back to the ISR.

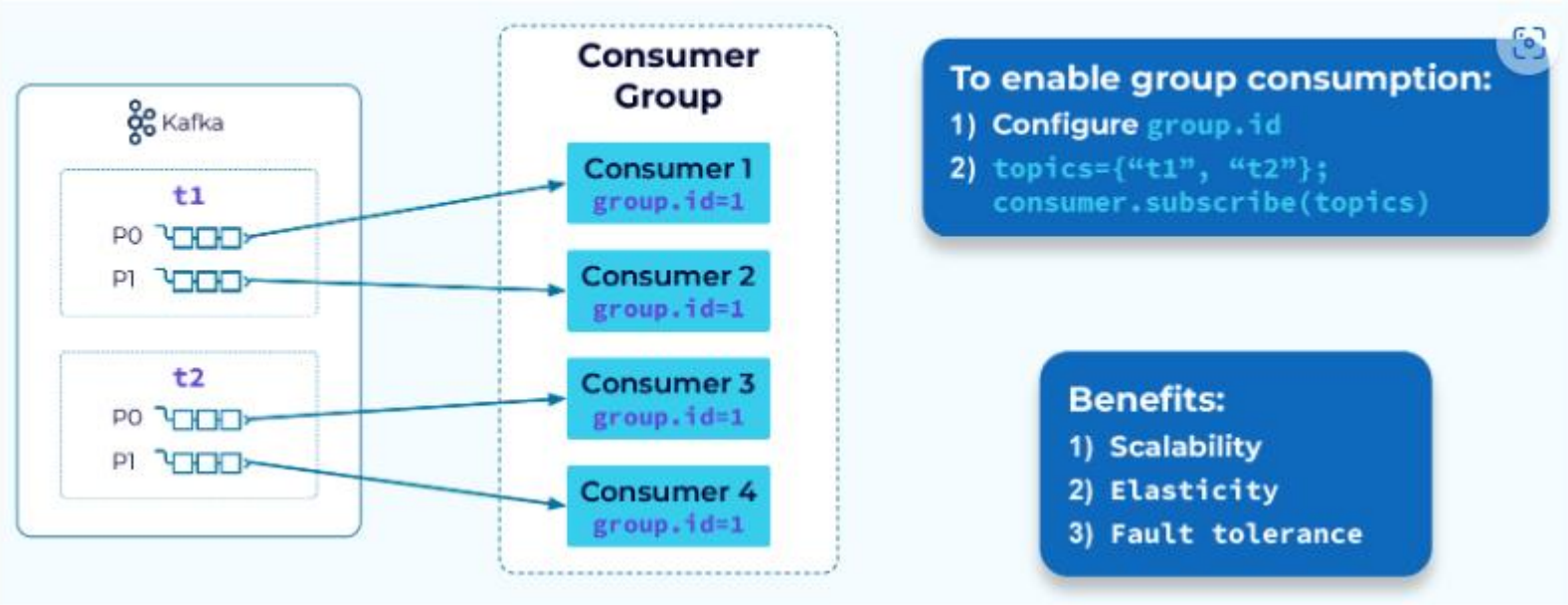
Partition Leader Balancing



The broker containing the leader replica does a bit more work than the follower replicas. Because of this it's best not to have a disproportionate number of leader replicas on a single broker. To prevent this Kafka has the concept of a preferred replica. When a topic is created, the first replica for each partition is designated as the preferred replica. Since Kafka is already making an effort to evenly distribute partitions across the available brokers, this will usually result in a good balance of leaders.

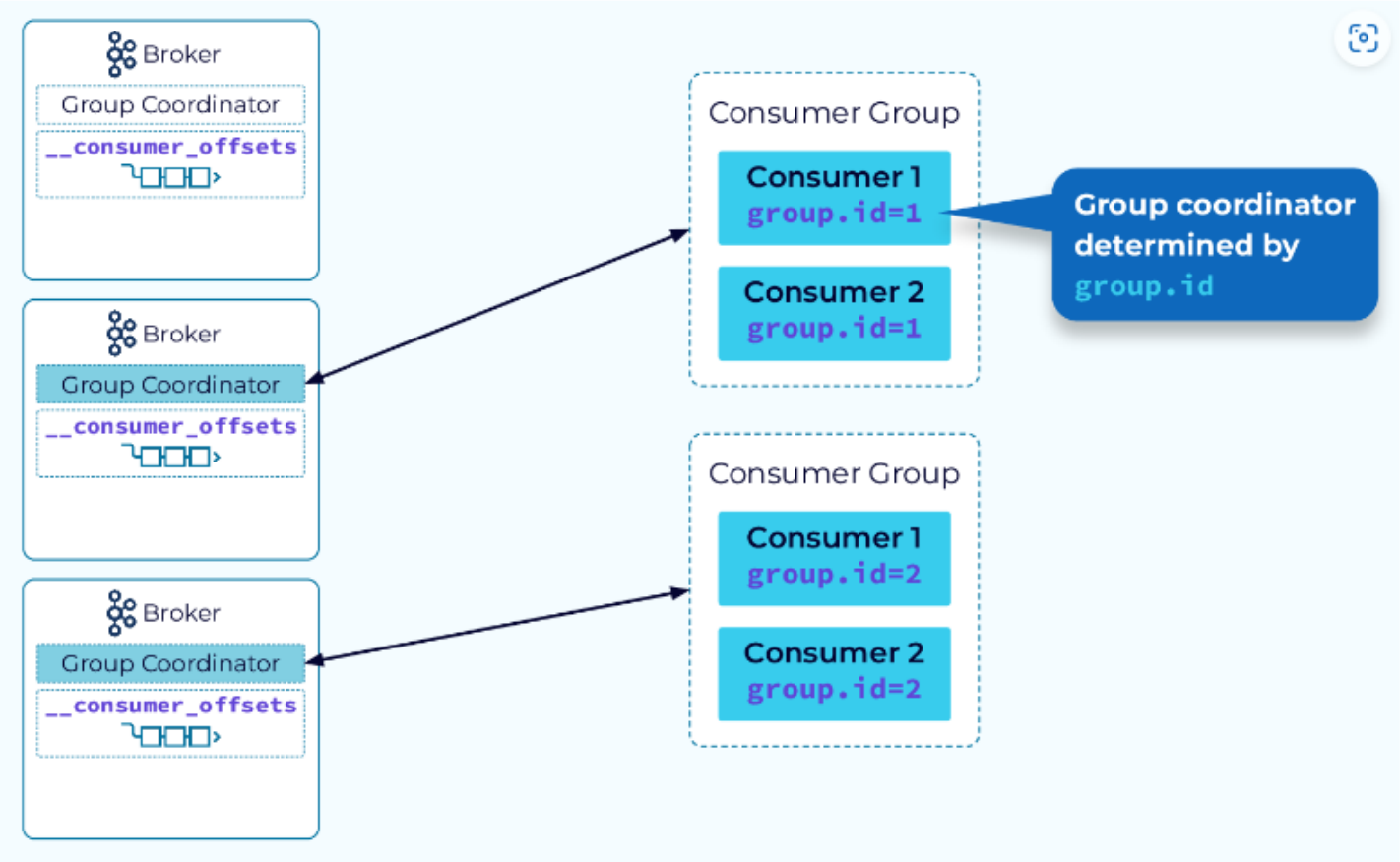
As leader elections occur for various reasons, the leaders might end up on non-preferred replicas and this could lead to an imbalance. So, Kafka will periodically check to see if there is an imbalance in leader replicas. It uses a configurable threshold to make this determination. If it does find an imbalance it will perform a leader rebalance to get the leaders back on their preferred replicas.

Kafka Consumer Group



To define a consumer group we just need to set the `group.id` in the consumer config. Once that is set, every new instance of that consumer will be added to the group. Then, when the consumer group subscribes to one or more topics, their partitions will be evenly distributed between the instances in the group. This allows for parallel processing of the data in those topics.
The unit of parallelism is the partition. For a given consumer group, consumers can process more than one partition but a partition can only be processed by one consumer

Group Coordinator

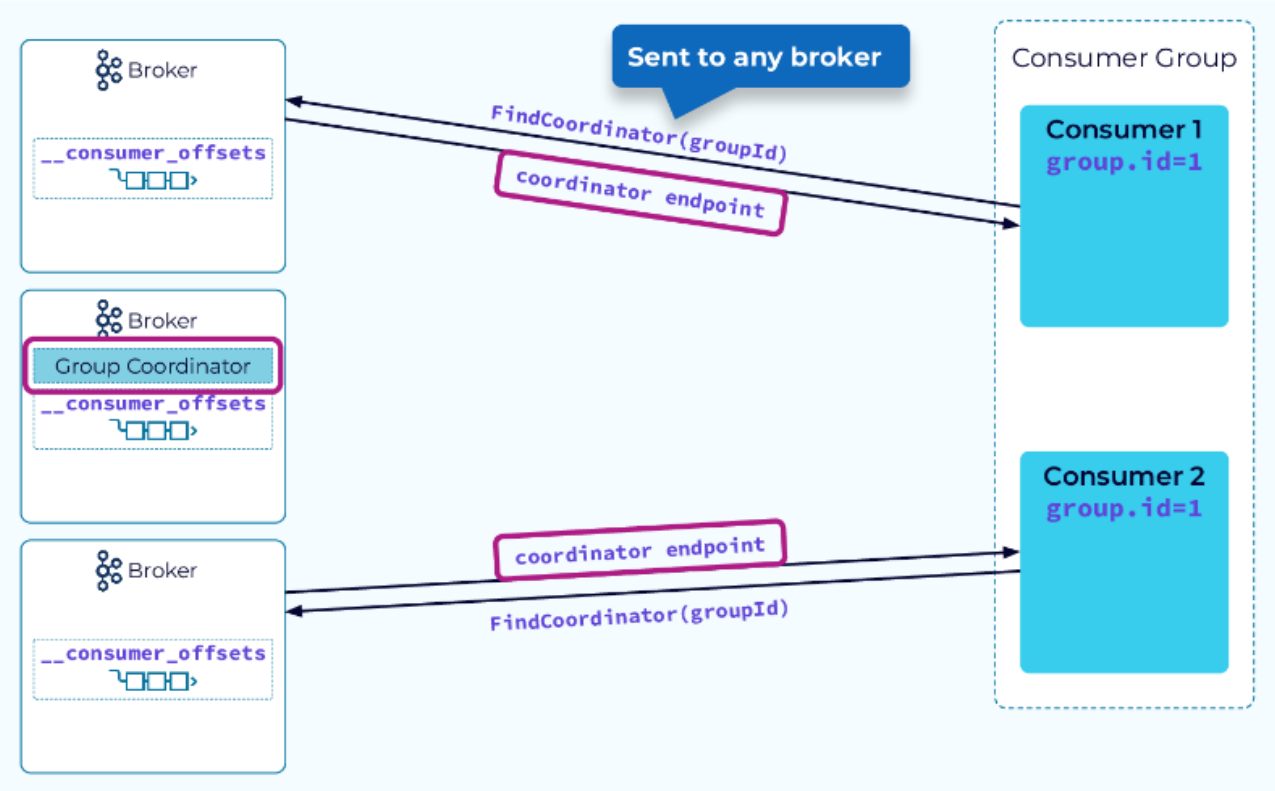


The magic behind consumer groups is provided by the group coordinator. The group coordinator helps to distribute the data in the subscribed topics to the consumer group instances evenly and it keeps things balanced when group membership changes occur. The coordinator uses an internal Kafka topic to keep track of group metadata.

Group Startup

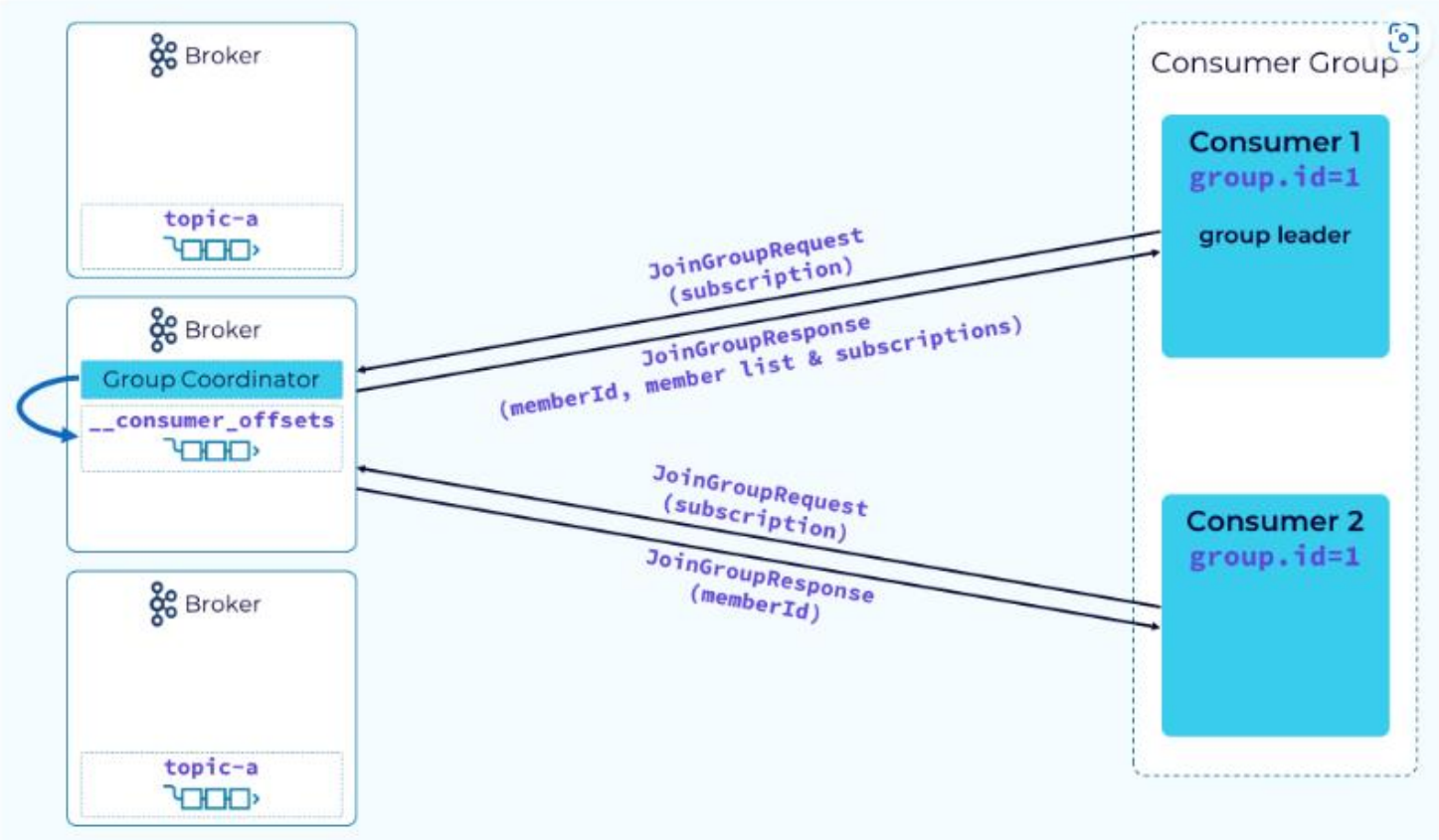
Let's take a look at the steps involved in starting up a new consumer group.

Step 1 – Find Group Coordinator



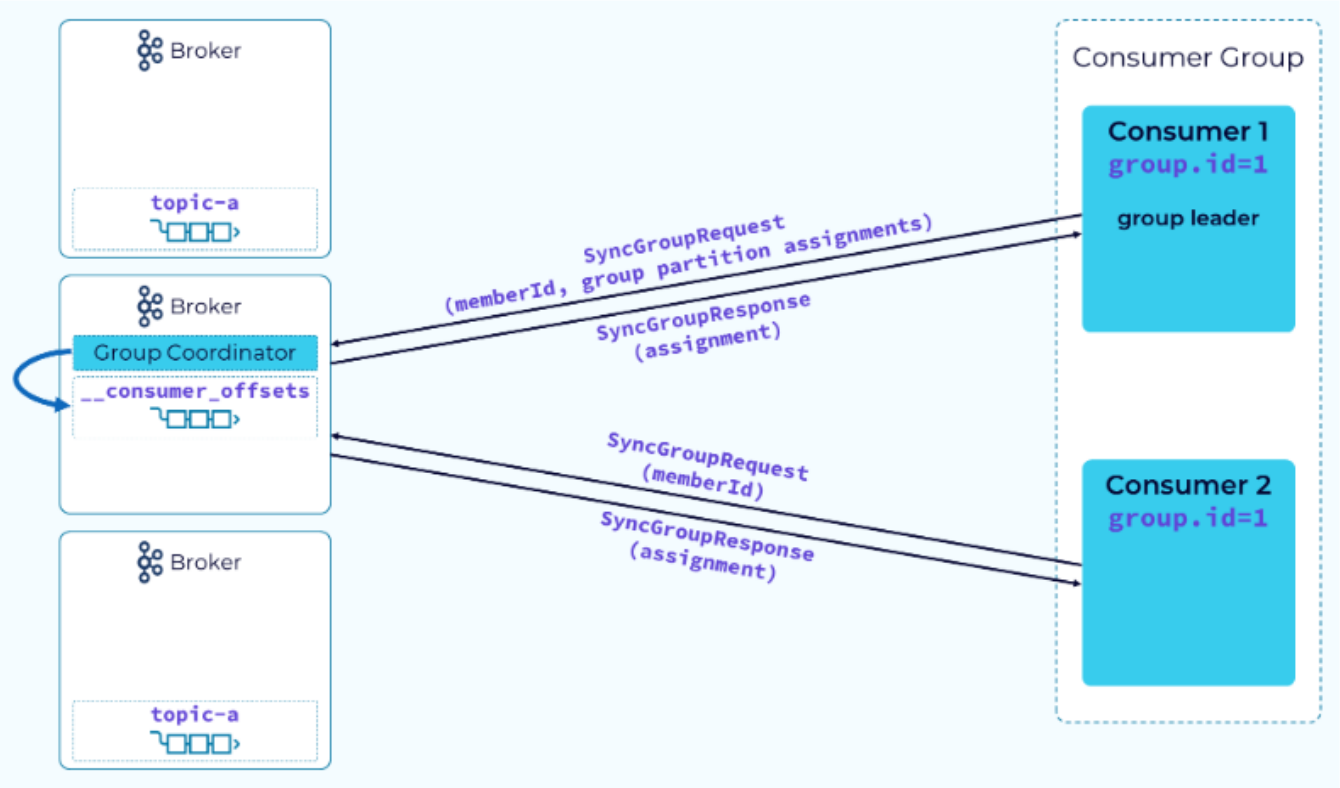
When a consumer instance starts up it sends a FindCoordinator request that includes its group.id to any broker in the cluster. Th broker will create a hash of the group.id and modulo that against the number of partitions in the internal __consumer_offsets topic. That determines the partition that all metadata events for this group will be written to. The broker that hosts the leader replica for that partition will take on the role of group coordinator for the new consumer group. The broker that received the FindCoordinator request will respond with the endpoint of the group coordinator.

Step 2 – Members Join



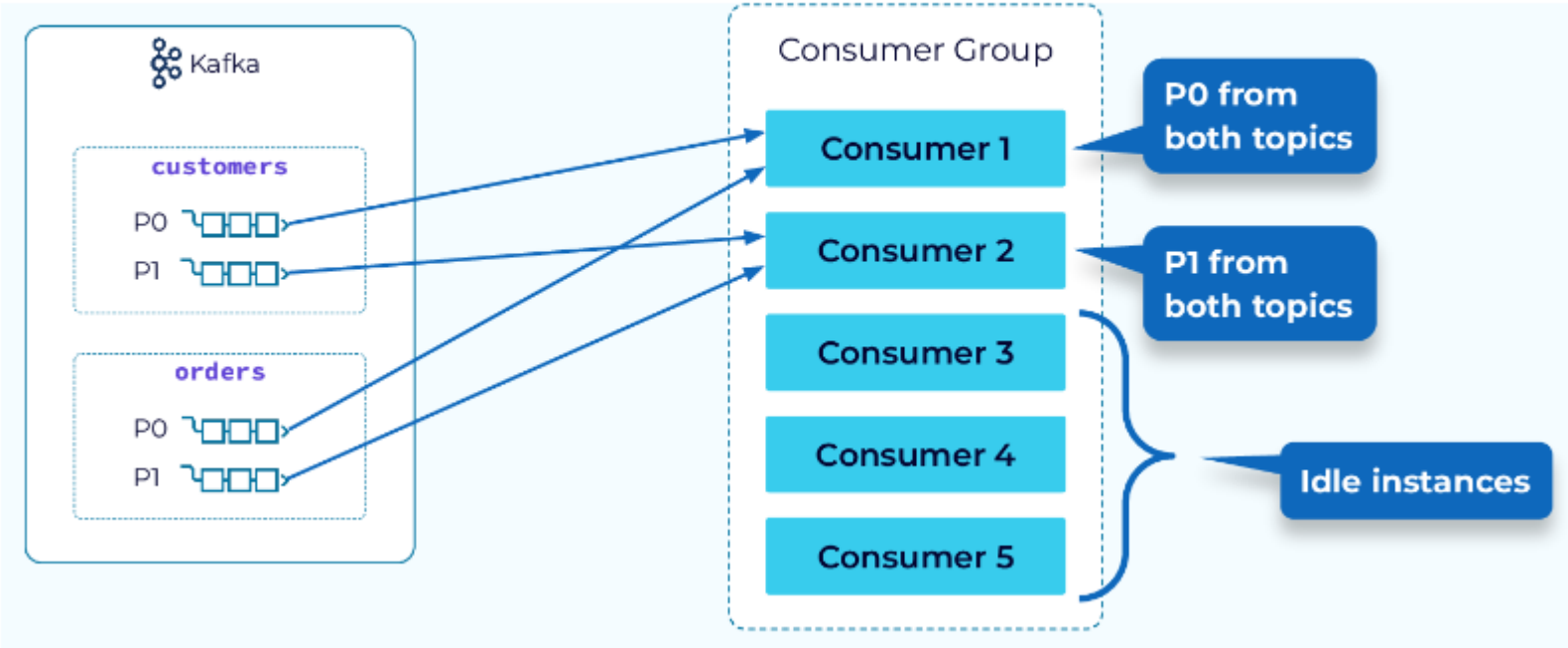
Next, the consumers and the group coordinator begin a little logistical dance, starting with the consumers sending a JoinGroup request and passing their topic subscription information. The coordinator will choose one consumer, usually the first one to send the JoinGroup request, as the group leader. The coordinator will return a memberId to each consumer, but it will also return a list of all members and the subscription info to the group leader. The reason for this is so that the group leader can do the actual partition assignment using a configurable partition assignment strategy.

Step 3 – Partitions Assigned



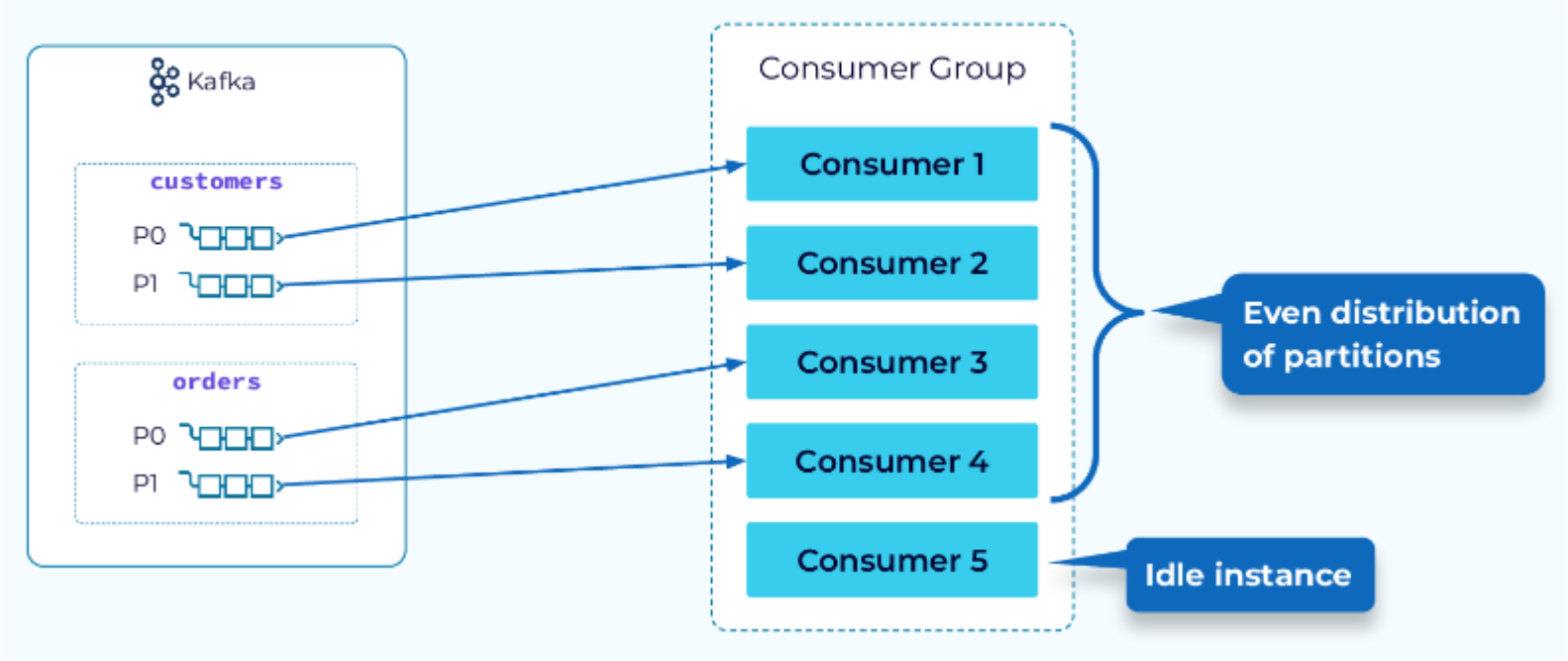
After the group leader receives the complete member list and subscription information, it will use its configured partitioner to assign the partitions in the subscription to the group members. With that done, the leader will send a SyncGroupRequest to the coordinator, passing in its memberId and the group assignments provided by its partitioner. The other consumers will make a similar request but will only pass their memberId. The coordinator will use the assignment information given to it by the group leader to return the actual assignments to each consumer. Now the consumers can begin their real work of consuming and processing data.

Range Partition Assignment Strategy



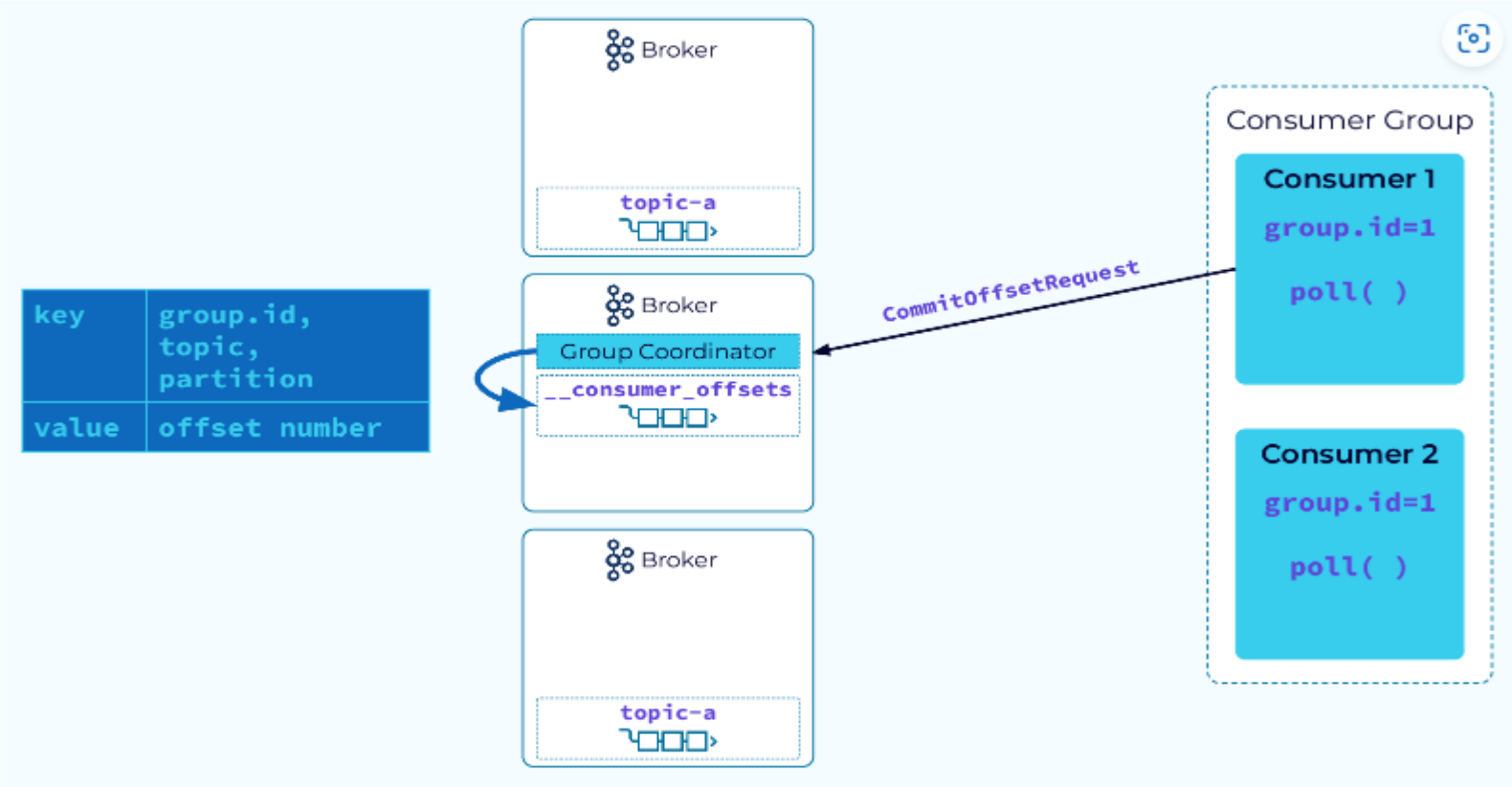
First up is the range assignment strategy. This strategy goes through each topic in the subscription and assigns each of the partitions to a consumer, starting at the first consumer. What this means is that the first partition of each topic will be assigned to the first consumer, the second partition of each topic will be assigned to the second consumer, and so on. If no single topic in the subscription has as many partitions as there are consumers, then some consumers will be idle.

Round Robin and Sticky Partition Assignment Strategies



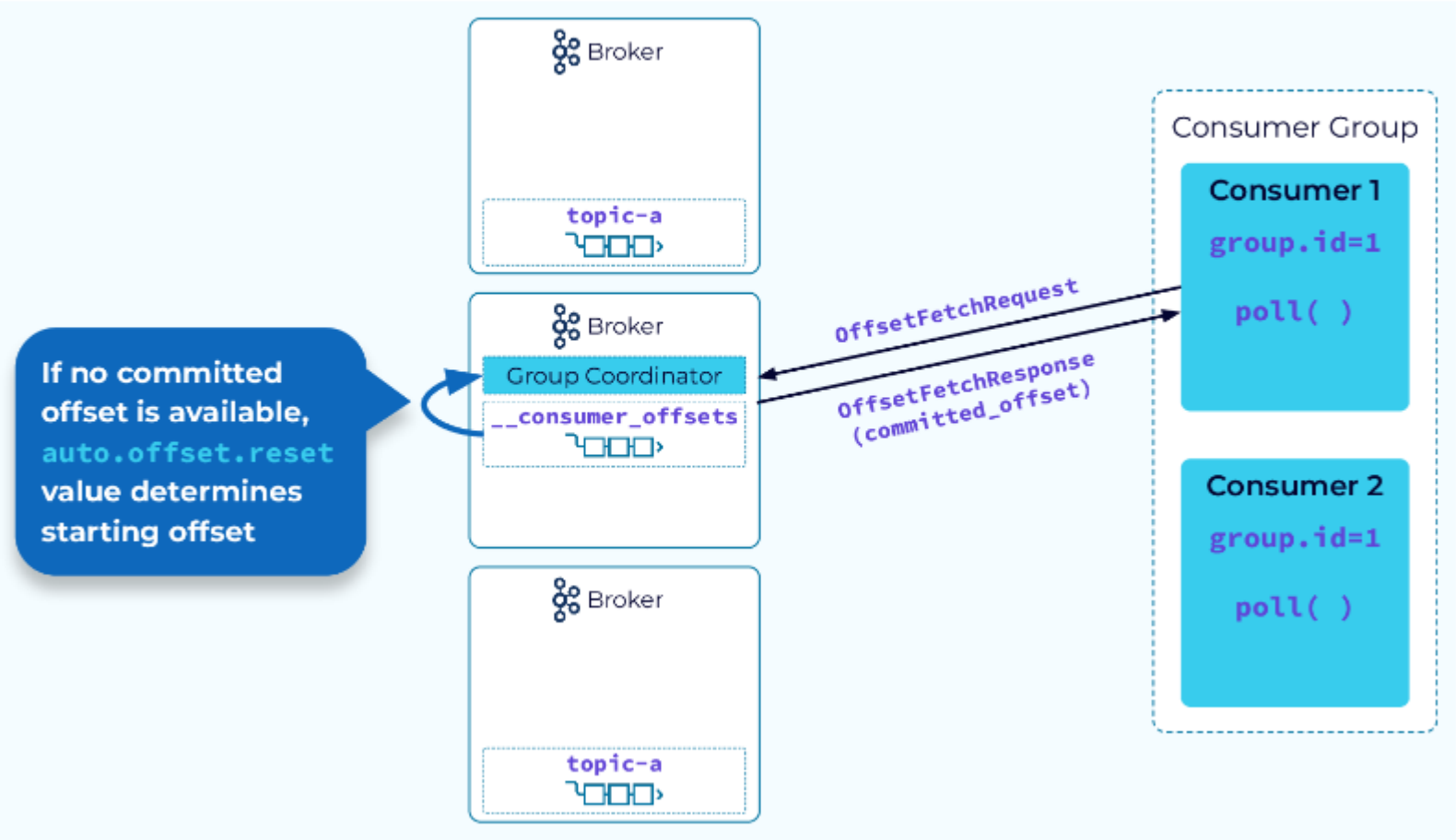
With this strategy, all of the partitions of the subscription, regardless of topic, will be spread evenly across the available consumers. This results in fewer idle consumer instances and a higher degree of parallelism.
A variant of Round Robin, called the Sticky Partition strategy, operates on the same principle but it makes a best effort at sticking to the previous assignment during a rebalance. This provides a faster, more efficient rebalance.

Tracking Partition Consumption



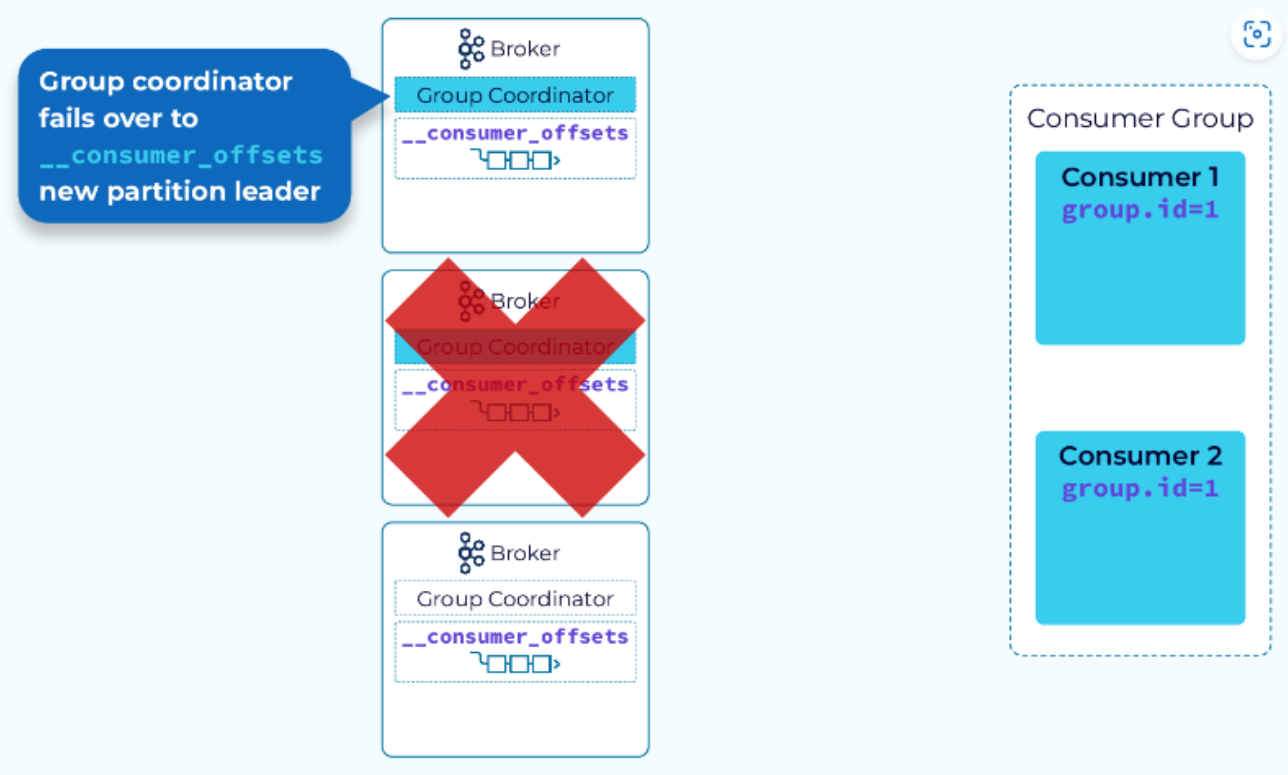
In Kafka, keeping track of the progress of a consumer is relatively simple. A given partition is always assigned to a single consumer, and the events in that partition are always read by the consumer in offset order. So, the consumer only needs to keep track of the last offset it has consumed for each partition. To do this, the consumer will issue a CommitOffsetRequest to the group coordinator. The coordinator will then persist that information in its internal __consumer_offsets topic.

Determining Starting Offset to Consume



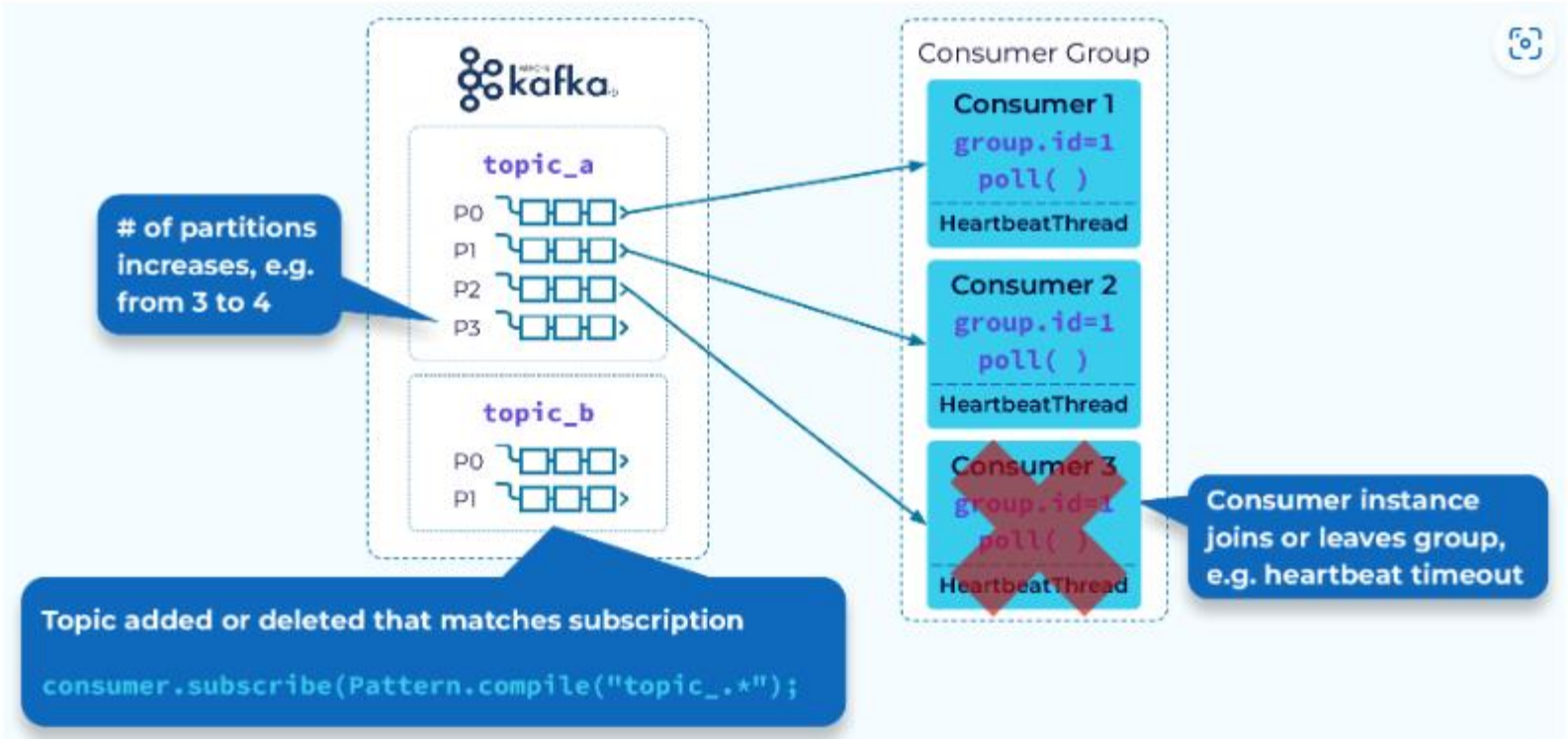
When a consumer group instance is restarted, it will send an OffsetFetchRequest to the group coordinator to retrieve the last committed offset for its assigned partition. Once it has the offset, it will resume the consumption from that point. If this consumer instance is starting for the very first time and there is no saved offset position for this consumer group, then the auto.offset.reset configuration will determine whether it begins consuming from the earliest offset or the latest.

Group Coordinator Failover



The internal __consumer_offsets topic is replicated like any other Kafka topic. Also, recall that the group coordinator is the broker that hosts the leader replica of the __consumer_offsets partition assigned to this group. So if the group coordinator fails, a broker that is hosting one of the follower replicas of that partition will become the new group coordinator. Consumers will be notified of the new coordinator when they try to make a call to the old one, and then everything will continue as normal.

Consumer Group Rebalance Triggers



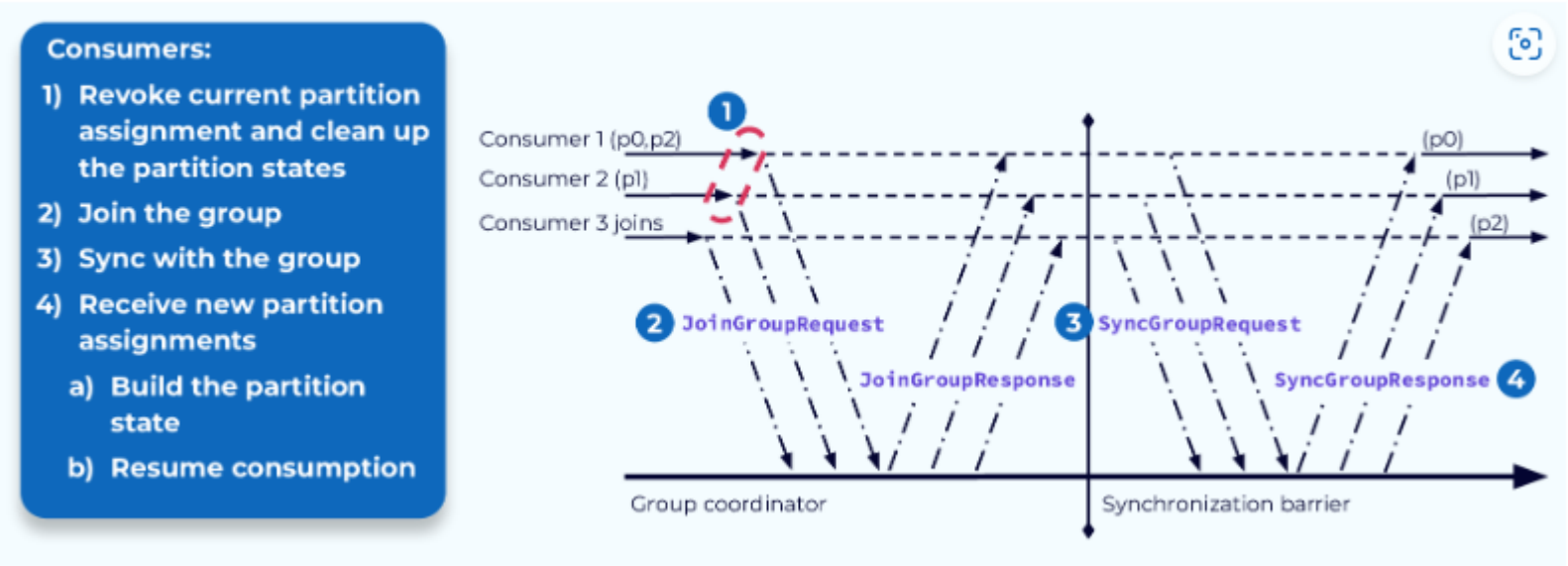
One of the key features of consumer groups is rebalancing. some of the events that can trigger a rebalance:

- An instance fails to send a heartbeat to the coordinator before the timeout and is removed from the group
- An instance has been added to the group
- Partitions have been added to a topic in the group's subscription
- A group has a wildcard subscription and a new matching topic is created
- And, of course, initial group startup

Consumer Group Rebalance Notification

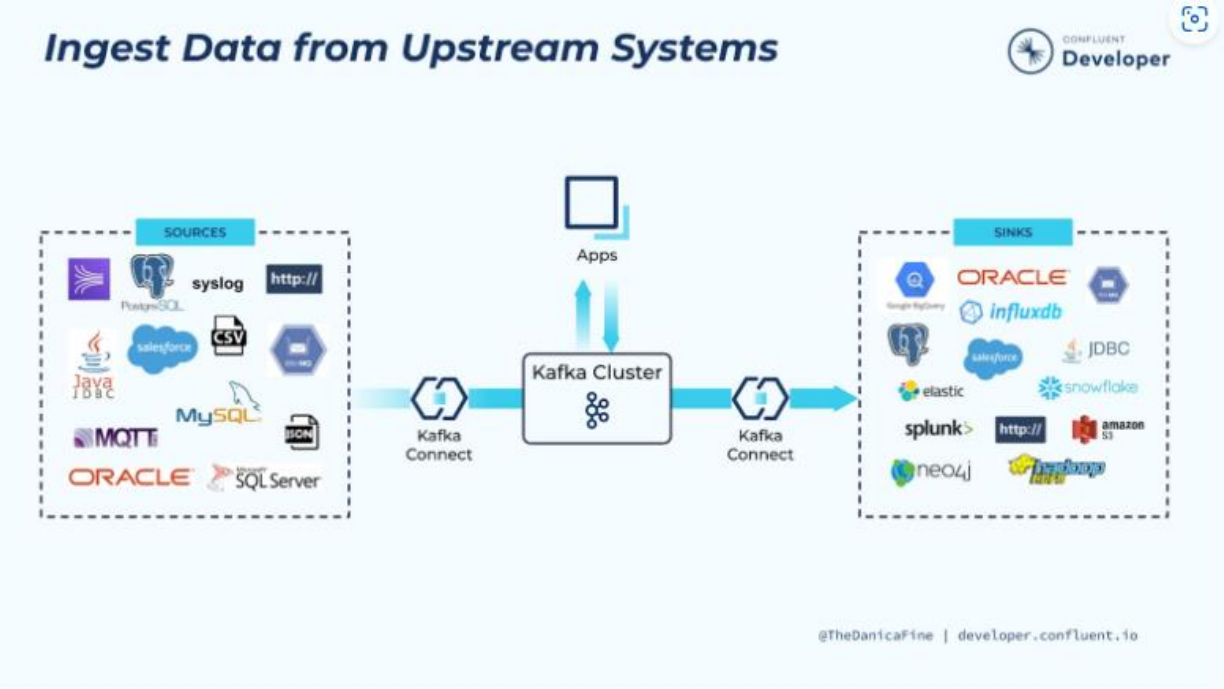
The rebalance process begins with the coordinator notifying the consumer instances that a rebalance has begun. It does this by piggybacking on the HeartbeatResponse or the OffsetFetchResponse.

Stop-the-World Rebalance



The traditional rebalance process is rather involved. Once the consumers receive the rebalance notification from the coordinator, they will revoke their current partition assignments. If they have been maintaining any state associated with the data in their previously assigned partitions, they will also have to clean that up. Now they are basically like new consumers and will go through the same steps as a new consumer joining the group.

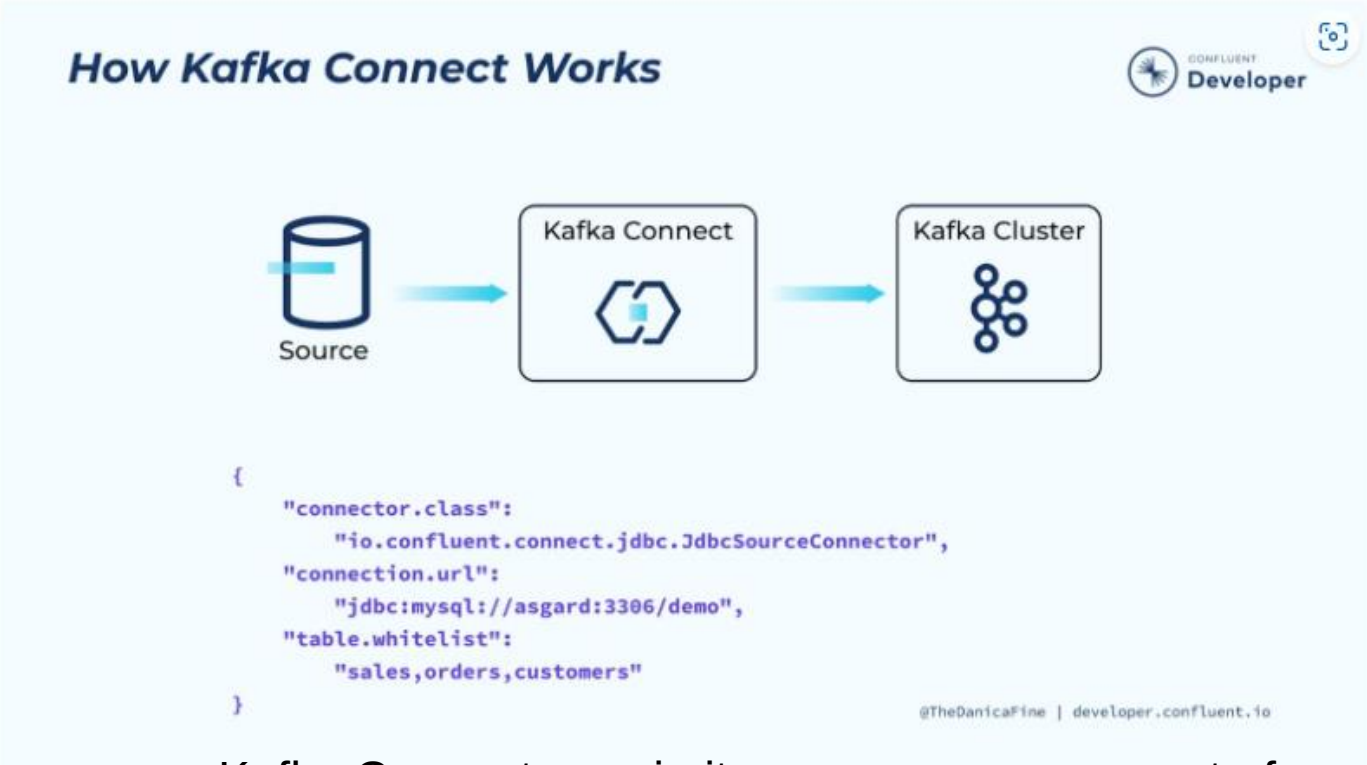
They will send a JoinGroupRequest to the coordinator, followed by a SyncGroupRequest. The coordinator will respond accordingly, and the consumers will each have their new assignments.



Kafka Connect is a component of Apache Kafka® that's used to perform streaming integration between Kafka and other systems such as databases, cloud services, search indexes, file systems, and key-value stores. Kafka Connect makes it easy to stream data from numerous sources into Kafka, and stream data out of Kafka to numerous targets. Some of the most popular ones include:

- RDBMS (Oracle, SQL Server, Db2, Postgres, MySQL)
- Cloud object stores (Amazon S3, Azure Blob Storage, Google Cloud Storage)
- Message queues (ActiveMQ, IBM MQ, RabbitMQ)
- NoSQL and document stores (Elasticsearch, MongoDB, Cassandra)
- Cloud data warehouses (Snowflake, Google BigQuery, Amazon Redshift)

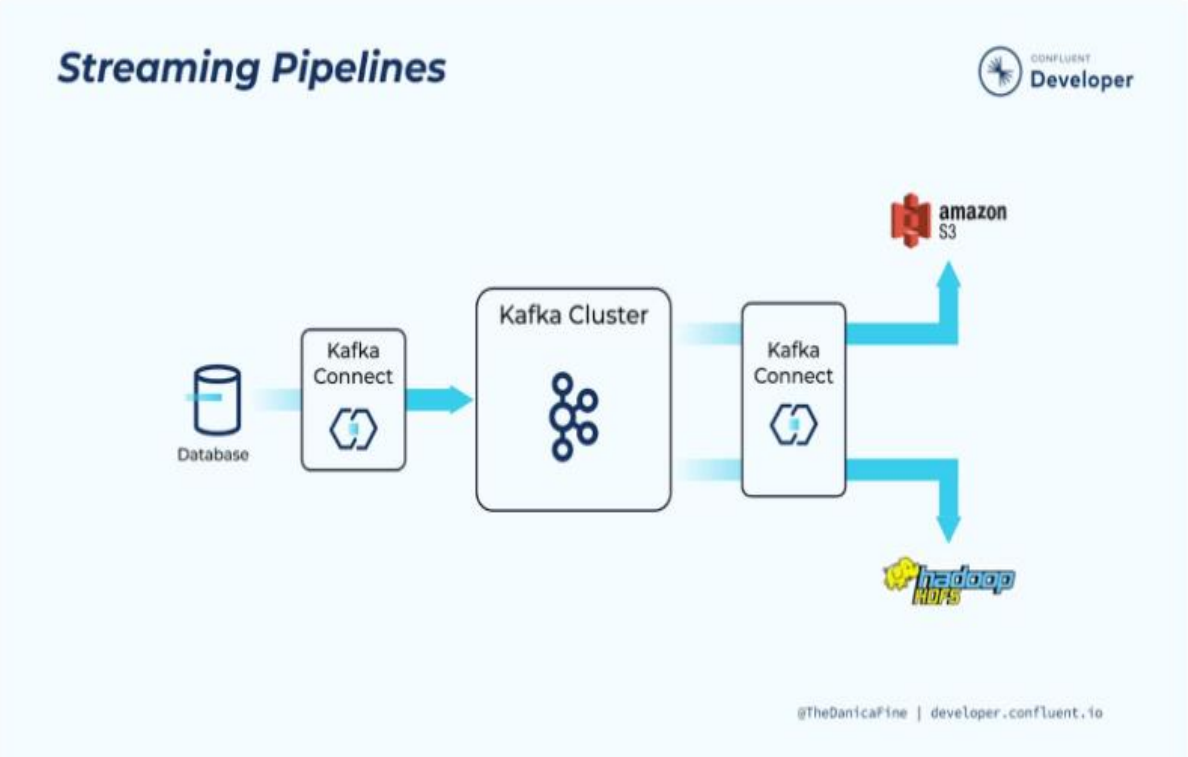
How Kafka Connect Works



Kafka Connect runs in its own process, separate from the Kafka brokers. It is distributed, scalable, and fault tolerant, giving you the same features you know and love about Kafka itself. It's completely configuration-based, making it available to a wide range of users—not just developers. In addition to ingest and egress of data, Kafka Connect can also perform lightweight transformations on the data as it passes through.

Anytime you are looking to stream data into Kafka from another system, or stream data from Kafka to elsewhere, Kafka Connect should be the first thing that comes to mind. Let's take a look at a few common use cases where Kafka Connect is used.

Streaming Pipelines



Kafka Connect can be used to ingest real-time streams of events from a data source and stream them to a target system for analytics. In this particular example, our data source is a transactional database.
We have a Kafka connector polling the database for updates and translating the information into real-time events that it produces to Kafka.

- First of all, having Kafka sits between the source and target systems means that building a loosely coupled system. In other words, it's relatively easy to change the source or target without impacting the other.
- Additionally, Kafka acts as a buffer for the data, applying back pressure as needed.
- And also, Kafka system is scalable and fault tolerant.