APACHE KAFKA

Apache Kafka Data Access Semantics: Consumers and Membership

STANISLAV KOZLOVSKI

MAY 7, 2019

Every developer who uses Apache Kafka® has used a Kafka consumer at least once. Although it is the simplest way to subscribe to and access events from Kafka, behind the scenes, Kafka consumers handle tricky distributed systems challenges like data consistency, failover and load balancing.

Luckily, Kafka's consuming model is quite easy to understand.

Understanding the details of how Kafka consumers work will make you an all-around better Kafka developer so that you can troubleshoot and build reliable applications more effectively.

Data processing requirements

Let’s first examine what we would want from a typical system that processes a large amount of data for a critical cause.

We will put ourselves in the shoes of a fictional, yet very popular stock trading platform that uses Kafka to process trade orders from financial brokers.

Imagine the following: A user executes a trade order on our website. That order gets propagated to Kafka and processed by an application that registers it on the New York Stock Exchange via the NYSE API.

Here are some characteristics that we would absolutely require from a stock trading platform:

Scalability

Since we’re a widely used fictional platform, we’re working with a huge volume of data and therefore need to be able to support it. There is no way that one computer node will ever be able to ingest and process all the events that get generated in real time. We therefore need a way of splitting up the data ingestion work.

Availability and fault tolerance

As all business-critical use cases require constant availability, their processing should always be on. Any downtime could cost our organization money and our users a lot of hassle. We need to always be able to read the latest data from Kafka.

Consistency

We would need to track what data we have processed to identify missing data or duplicate processing. Consistency should never be breached, even during periods of unavailability or while scaling our workload up/down.

Latency

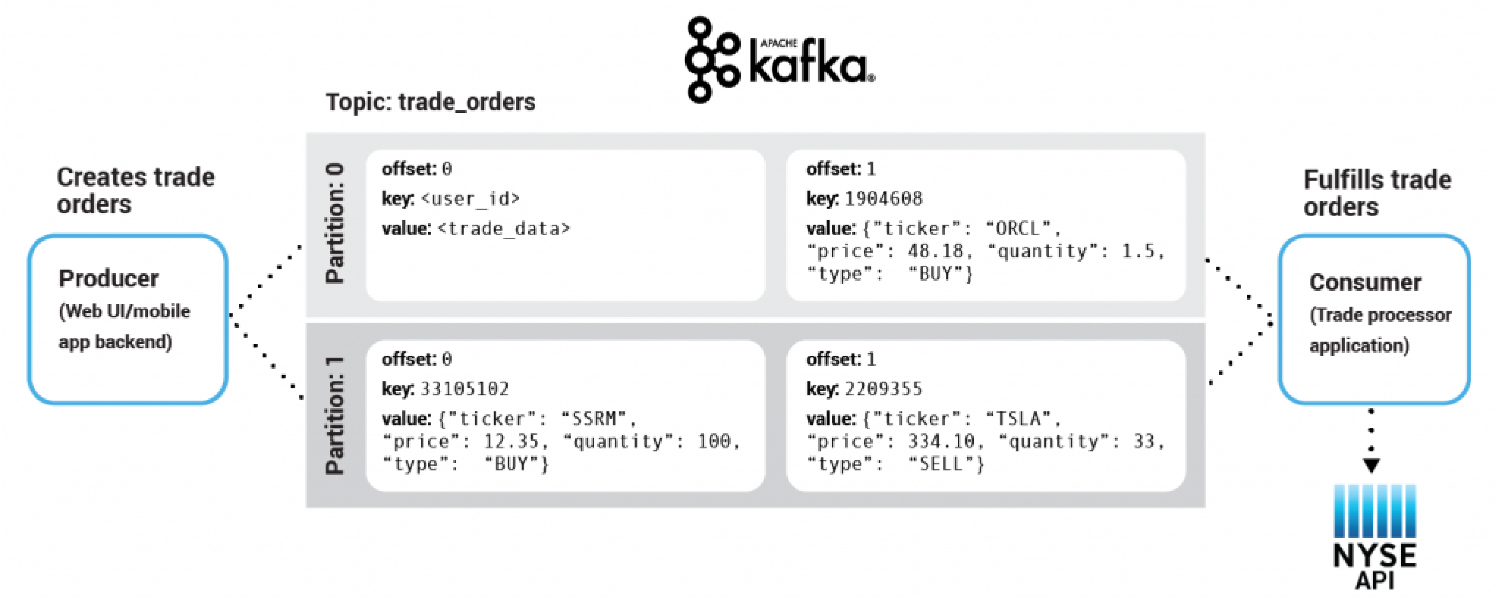
We want very low latency in order to ensure the most accurate prices on orders, but we will not be exploring this requirement in depth as data processing is only partly responsible for this.

Next, let's explore how Kafka addresses these requirements.

Consumers and consumer groups

When it comes to Kafka, there is the notion of consumers and consumer groups (more on groups later).

A consumer is simply an application that reads records from Kafka. Since the data record is in some partition, we can say that consumers simply read from partitions.


APACHE kafka
Topic: trade_orders
Creates trade orders
Producer
(Web UI/mobile app backend)
Partition: 0
offset: 0
key: <user_id>
value: <trade_data>
offset: 1
key: 1904608
value: {"ticker": "ORCL", "price": 48.18, "quantity": 1.5, "type": "BUY"}
Partition: 1
offset: 0
key: 33105102
value: {"ticker": "SSRM", "price": 12.35, "quantity": 100, "type": "BUY"}
offset: 1
key: 2209355
value: {"ticker": "TSLA", "price": 334.10, "quantity": 33, "type": "SELL"}
Fulfills trade orders
Consumer
(Trade processor application)
NYSE API

A sample architecture of our fictional trading site

Kafka consumers use a pull model to consume data. This means that a consumer periodically sends a request to a Kafka broker in order to fetch data from it. This is called a FetchRequest. The consumer application then receives a number of records in a response, processes them and asks for more in the form of another request.

Consumers typically perform multiple fetches in parallel internally and accumulate data. The user receives the records in their application via the `KafkaConsumer#poll()` method call.

```
while (true) {
 ConsumerRecords<String, String> records = consumer.poll(100);
 processTrades(records) // business logic
}
```

Just before exiting the `poll()` call, the consumer sends out another round of FetchRequests so that new records can be fetched while the current ones are being processed by the application.

What is a Kafka offset?

As a brief refresher, let’s remind ourselves what a Kafka offset is! An offset is a special, monotonically increasing number that denominates the position of a certain record in the partition it is in. It provides a natural ordering of records—you know that the record with offset 100 came after the record with offset 99.

Consistency: Keeping track of where we’re at

We want our consumer applications to store the state of their progress. If we have read all the way up to offset 1,000 in a topic and shut down the consumer, we would like to start off from offset 1,000 in the subsequent startup of that very same consumer. This requires that we store this progress somewhere.

If we're being naive, we could simply store that state in a local text file. A big downside to this approach is that it would make our consumers stateful—ultimately making them hard to deploy, manage and scale. Data durability is another problem that this approach would not solve. If our consumer application's disk dies, we may not have a way of recovering the state!

It makes sense to store the offset in the same place the consumer is reading from. For example, a Kafka Connect connector populates data in HDFS along with the offsets of the data it reads so that it is guaranteed that either data and offsets are both updated, or neither is. A similar pattern is followed for many other data systems that require these stronger semantics, and for which the messages do not have a primary key to allow for deduplication.

This is also how Kafka supports exactly once processing in Kafka Streams, and the transactional producer or consumer can be used generally to provide exactly once delivery when transferring and processing data between Kafka topics.

We need an external storage system. The following question arises: What's a good, reliable and practical storage system inside a Kafka deployment? Yup, you guessed it—Kafka itself!

Our little state store: The consumer offsets topic

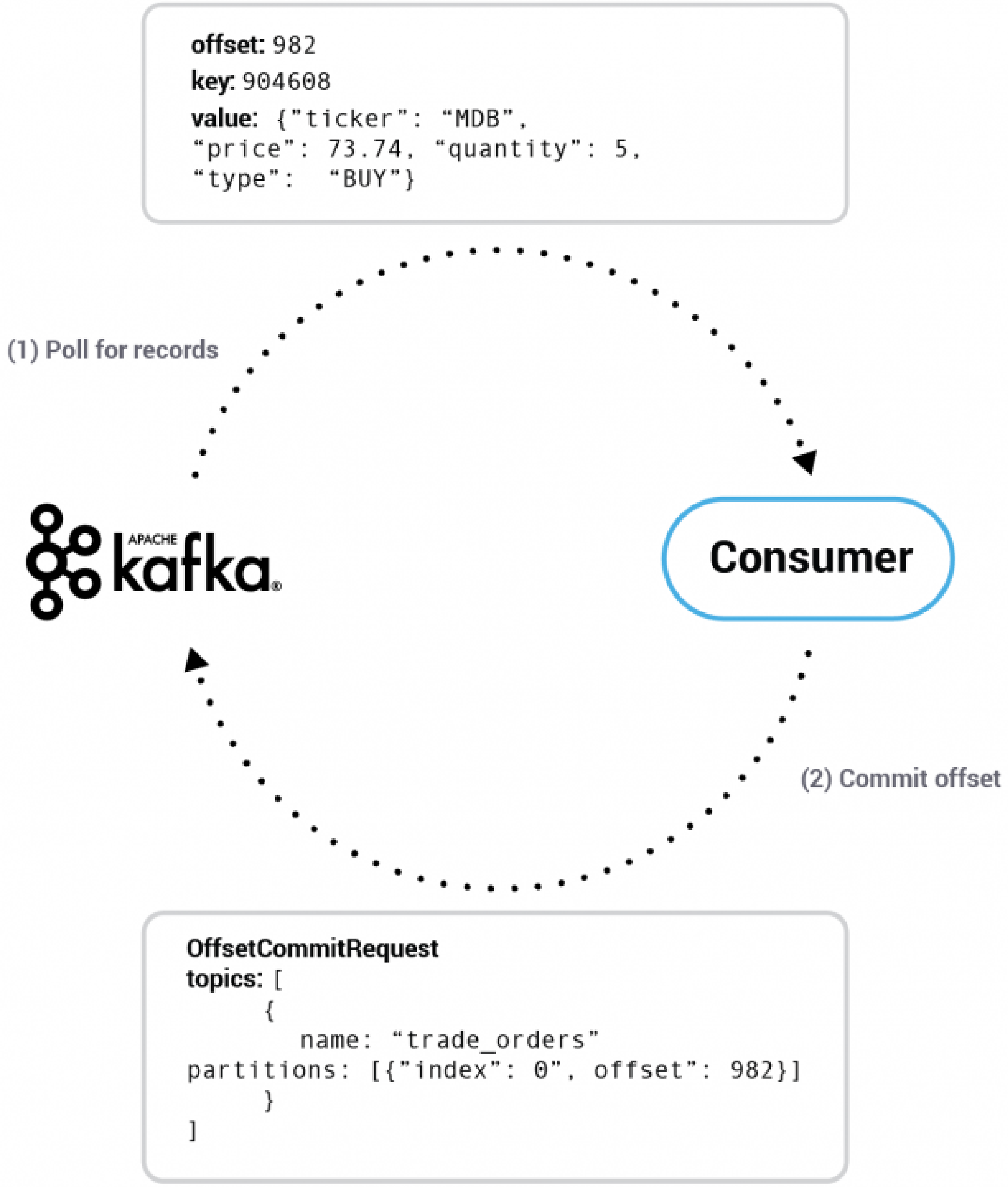
Consumers store their progress inside a Kafka topic called `__consumer_offsets`. Since we only ever need the latest state for a given consumer, and not its past states, this remains a compacted topic. Its retention period is seven days by default and can be configured by the `offsets.retention.minutes` setting.

It is now worth noting that the consumer client applications are responsible for storing, keeping track of and operating their own state. The Kafka brokers, where possible, are agnostic to a consumer's state. This results in a better separation of concerns and less server load.

Consumers send an `OffsetCommitRequest` to Kafka brokers whenever they want to save their progress. In some sense, you can look at the consumer as a part-time producer, as this request is essentially a normal Kafka `Produce` request with `acks=all`.

Consumer can send this request manually when either one of the `KafkaConsumer#commitSync()` or `KafkaConsumer#commitAsync()` methods are called. For good performance, it is essential to balance the frequency with which these methods get called. Calling them after every processed record incurs big overhead, whereas calling them too sparingly will result in more records being processed twice under a failure scenario where the consumer fails in between commits (e.g., right before calling a commit).

Consumers can also send this request automatically. When the configurable `auto.commit.interval.ms` setting is set, `KafkaConsumer#poll()` calls will issue a commit request for all the records that were processed in the previous `poll()` call.



```
offset: 982
key: 904608
value: {"ticker": “MDB”,
“price”: 73.74, “quantity”: 5,
“type”:  “BUY”}
```

```
OffsetCommitRequest
topics: [
        {
           name: “trade_orders”
partitions: [{"index”: 0”, offset”: 982}]
        }
]
```

A sample communication between the Kafka broker and the consumer application, in which the protocol is simplified for clarity

The `OffsetCommitRequest` consists of a map that denotes the latest processed offset for any given partition (`TopicPartition->OffsetAndMetadata`).

The broker uses that information to construct records whose key is the `<consumer_group>,<topic>,<partition>` and value is `<offset>,<partition_leader_epoch>,<metadata>,<timestamp>` .

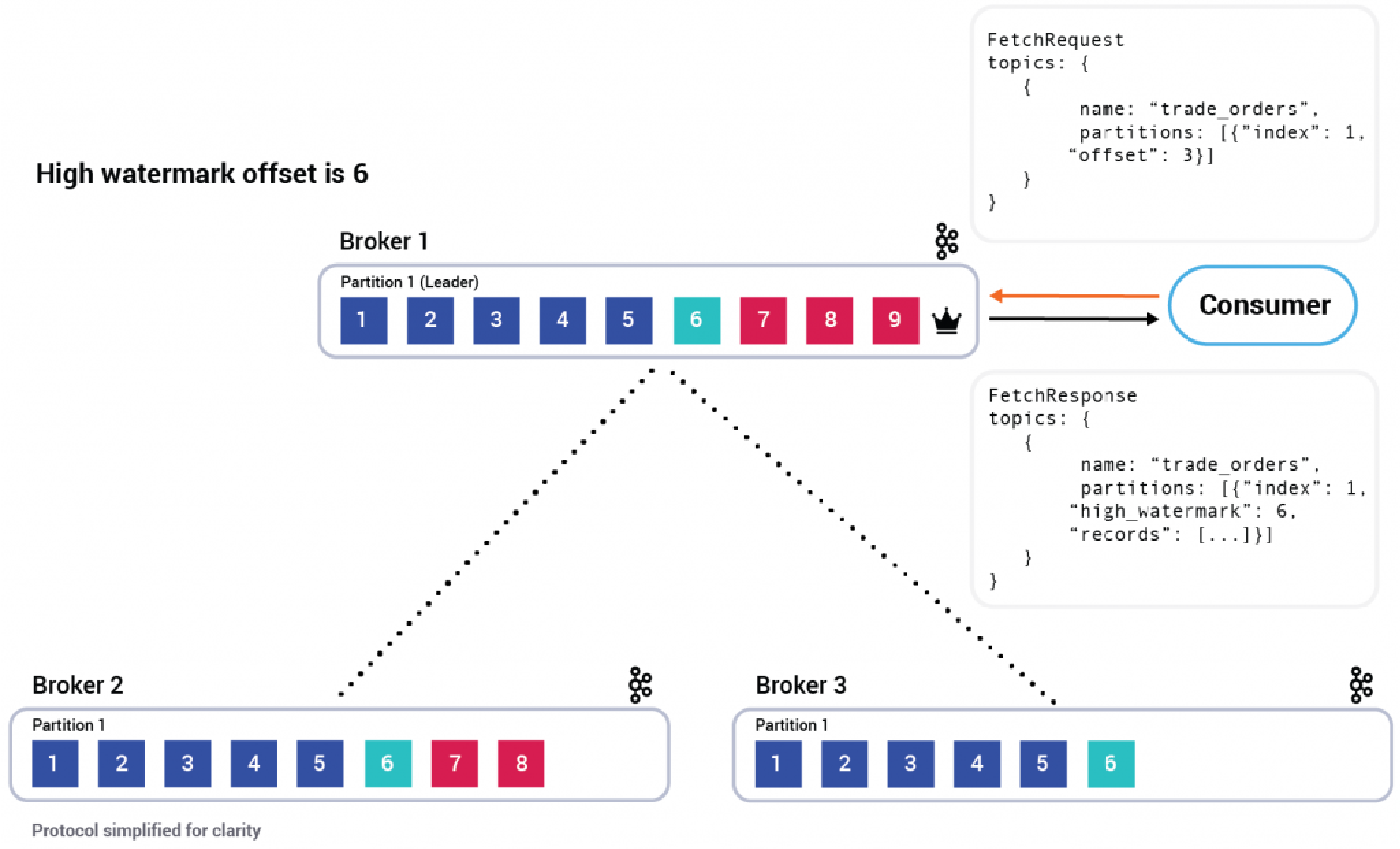
It appends these records to its local log. The broker then waits until that specific `__consumer_offsets` topic's partition data gets replicated to all its followers. After all the replicas have saved the new offsets, a response is returned to the consumer. This way, Kafka ensures data durability, guaranteeing that a consumer cannot lose its progress once committed.

In case the replication fails within the `offsets.commit.timeout.ms` config setting, the broker will consider this commit as failed. It will return an error to the consumer, and the consumer will typically retry after a backoff period.

This offset data is also cached in memory inside the broker to enable faster responses for `OffsetFetchRequests` .

High watermark offset

Consistency is also ensured by the so-called *high watermark offset*. This is the largest offset, which all in-sync broker replicas of a particular partition share.



```
FetchRequest
topics: {
    {
        name: "trade_orders",
        partitions: [{"index": 1,
      "offset": 3}]
    }
}
```

```
FetchResponse
topics: {
    {
        name: "trade_orders",
        partitions: [{"index": 1,
      "high_watermark": 6,
      "records": [...]}]
    }
}
```

If a consumer were to read the record with offset 9 from the leader (broker 1), and if a leader election had happened before the record could be replicated to the followers, the consumer would have experienced a non-repeatable read. Because of this possibility, Kafka brokers only return records up to the high watermark offset for consumers' fetch requests.

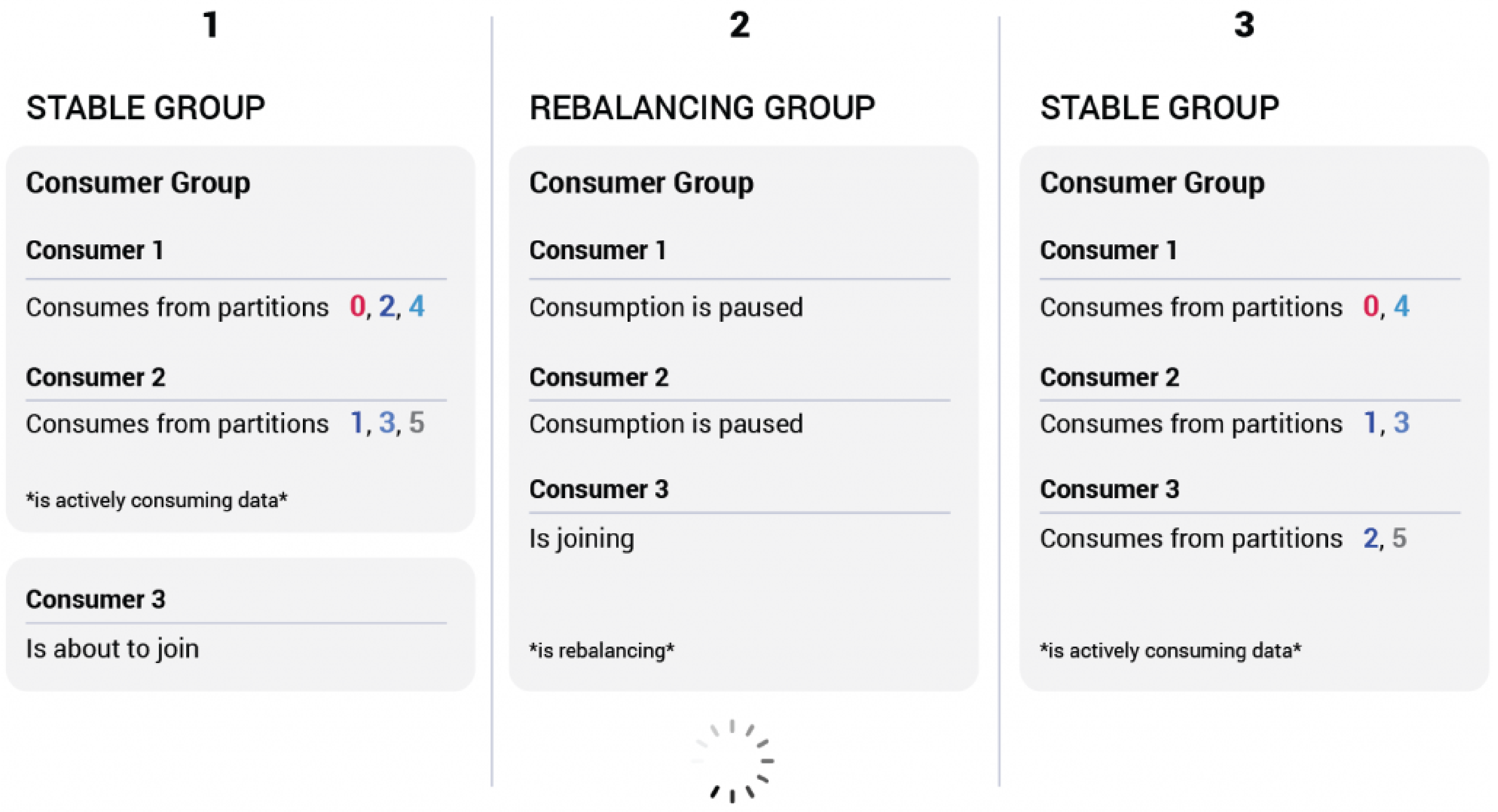
Consumer group mechanics

So far, we have examined data consumption through the lens of a single consumer. In the vast majority of real-life use cases, you will always have more data than what a single consumer application would be able to process. Not only that, but you may also have multiple, separate groups of applications that need to independently process the same Kafka topics. As a result, we need a scalable and fault-tolerant way to split data consumption.

Enter *consumer groups*.

A consumer group is a set of consumer applications working together. They are identified by the `group.id` client config value. Consumers that are part of a group may subscribe to multiple topics. All of the said topics' partitions then get split between the consumers in a customizable way. To ensure consistency, the default configuration ensures that only one consumer inside a consumer group can read from a particular partition. In order to be scalable and elastic, we need to support adding and removing consumers from a group on the fly. Kafka allows a group to shift sizes through the notion of *rebalances*.

A consumer group rebalance is an infrequent process where every consumer in the group pauses its data consumption in order to update its metadata, synchronize with its group and continue processing. Rebalances are a very useful tool, but in the normal course of events are fairly undesirable—they're basically a short window of unavailability.



A consumer group rebalancing when a new member joins the group

This interruption is needed in order to ensure consistency. If we ever want to add a new member to the consumer group, we will need to have that member take some partitions off of the existing consumers. If we ever want to remove a member from the group, we will need to have some members take in more partitions. Both processes involve changing the partitions that some consumers are reading from and therefore entail a slight pause in data processing.

Membership API

It is worth noting that we are currently describing what is called Kafka's membership API. Although we are discussing this in the context of consumer applications, the protocol is intentionally created in a generic way to support additional use cases:

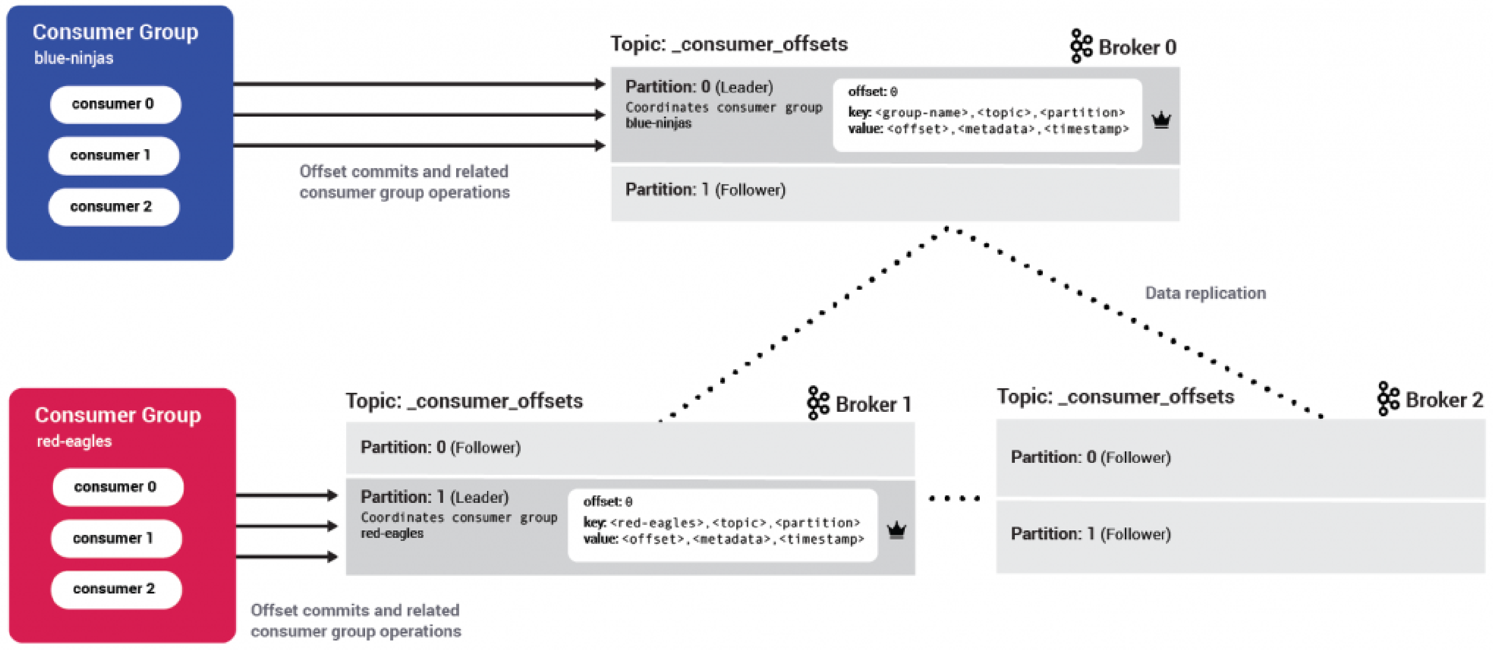
- Kafka Connect: Workers share and assign their role (connector/task) with system-specific metadata
- Kafka Streams: Topic partitions sharing and offset commits
- Confluent Schema Registry: Electing a Schema Registry master, in case of a multi-node Schema Registry deployment

Managing membership

Somebody needs to fill the shoes of managing a membership rebalance. This is the responsibility of the so-called *consumer group coordinator*. The group coordinator is a single Kafka broker that is responsible for managing a particular consumer group.

Remember the `__consumer_offsets` topic? Each consumer group stores the state of all its consumers inside a single partition of that topic. The group coordinator broker is the leader for that consumer group's partition. This way, every broker can be a group coordinator. With enough groups, every broker will work as a coordinator for a different subset of consumer groups.

If a broker holding the metadata of a particular consumer group ever dies, the leader of the partition holding the metadata will be moved. The new designated leader would then rebuild the state of the consumer group by reading the records in the log.



As mentioned earlier, a single consumer will not be able to read from all the data (partitions) we have in a Kafka cluster. We need to split up this data in between all the consumers in a group, preferably in a fair way.

This is the responsibility of the *group leader*.

The consumer group leader is an arbitrary consumer client application that is part of the group. In practice, it is the first consumer to join the group. Apart from being a normal consumer, it also makes the decision about which consumer should ingest from which partition.

To control this assignment, users can either write an implementation of the `PartitionAssignor` interface or use one of the three provided implementations (configured through the `partition.assignment.strategy` config):

- `RangeAssignor` : For each topic, divide its partitions by the number of consumers subscribed to it and assign X to each (lexicographically sorted) consumer. If it does not evenly divide, the first consumers will have more partitions.
- `RoundRobinAssignor` : Assign all partitions from all the subscribed topics to each consumer sequentially, one by one.
- `StickyAssignor` : Assign partitions so that they are distributed as evenly as possible. During rebalances, partitions stay with their previously assigned consumers as much as possible.

Additionally, the `PartitionAssignor` interface exposes a `metadata()` method. Every consumer in the group can use this method to send generic metadata about itself to the broker when joining a group. Once a rebalance is in the works, every consumer's metadata is propagated to the group leader. This enables the leader to make a well-informed decision about assigning partitions (e.g., by considering a consumer application's datacenter rack).

Assignments are defined inside the consumer for convenience. Consequently, you do not have to restart your Kafka brokers every time you come up with a new assignment for a particular consumer group.

Fault tolerance

Our data consumption story is almost complete. Thus far, we've shown we are tolerant to broker failures. Consumer application failures, though, are more common. To have our data consumption be fault tolerant against consumer application failures, we need a way to recognize when (and whether) a particular consumer application has stopped working, so we can act accordingly.