Kafka

Javier Cacheiro

Basic Concepts

Kafka

A distributed real-time streaming platform

Kafka

Originally developed at LinkedIn

Programmed in Scala like Spark

Kafka

A distributed and replicated publish-subscribe messaging system

Kafka

Programmed as a distributed commit log

Kafka

Records/Messages

Data is stored as key/value pairs with *timestamp* called Records or Messages

Messages are immutable: Once a message is written to a partion it can not be changed

Messages can contain any value (binary data)

Topics

A sequence of messages is called a data stream

A topic is a specific data stream to which consumers can subscribe

Similar to a table in a database

Partitions

Topics are split into partitions

A Partition is a structured commit log

It’s an ordered, immutable sequence of messages that are continually appended to.

It can’t be divided across brokers or even disks.

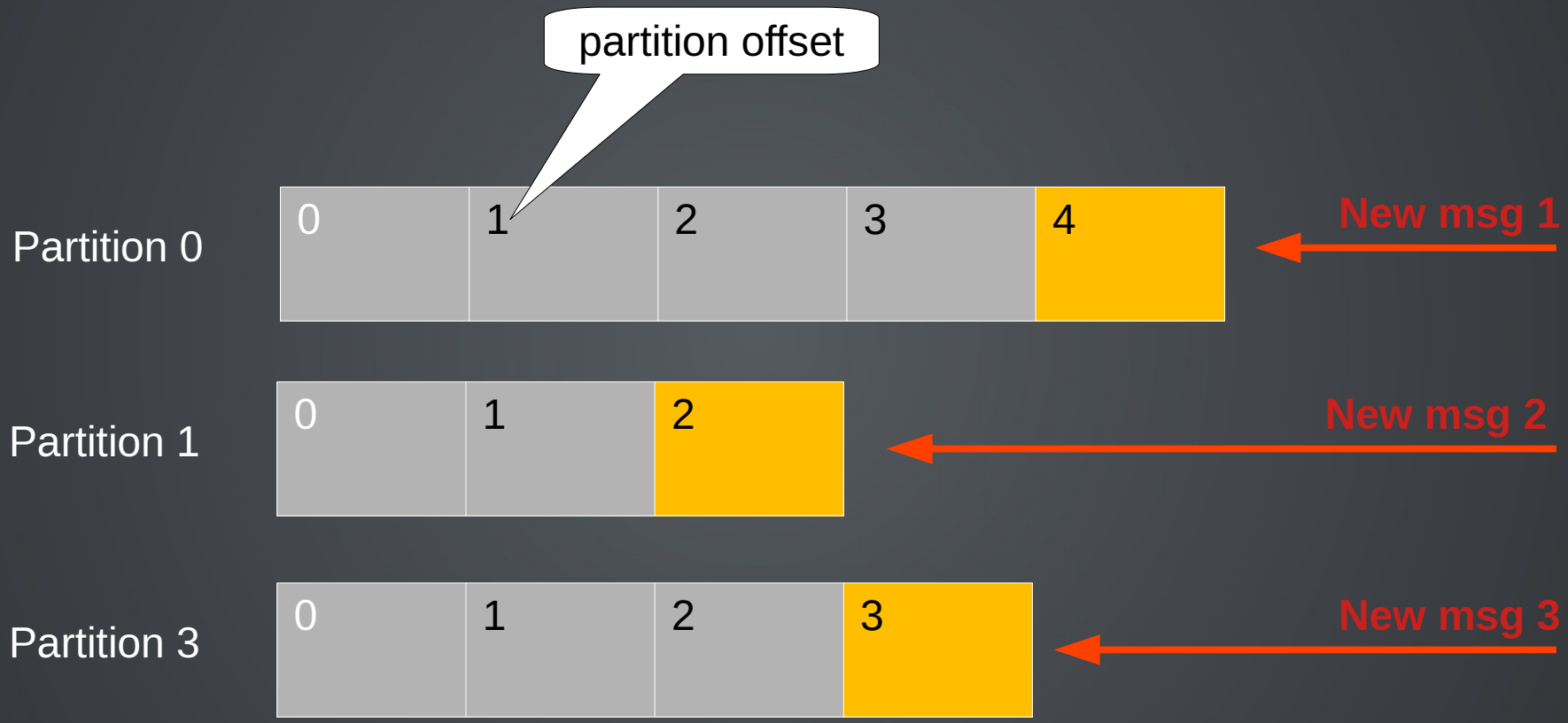
Offsets

Messages in the partitions are assigned a sequential id number called the offset

The offset uniquely identifies each message within a partition

The offset is unique (per partion) and sequential

Topic Partitions



Segments

Each partition is sub-divided into segments

Instead of storing all the messages of a partition in a single file, the partition is splitted into chunks called segments

Brokers

- The servers of the Kafka cluster are called Brokers
- Each broker hosts topics
- Receives messages from producers and stores them
- Allows consumers to fetch messages

Zookeeper

An open-source server which enables highly reliable distributed coordination

Maintains configuration data

Maintains the leader-follower relationship across all the partitions

Based on the Paxos consensus algorithm

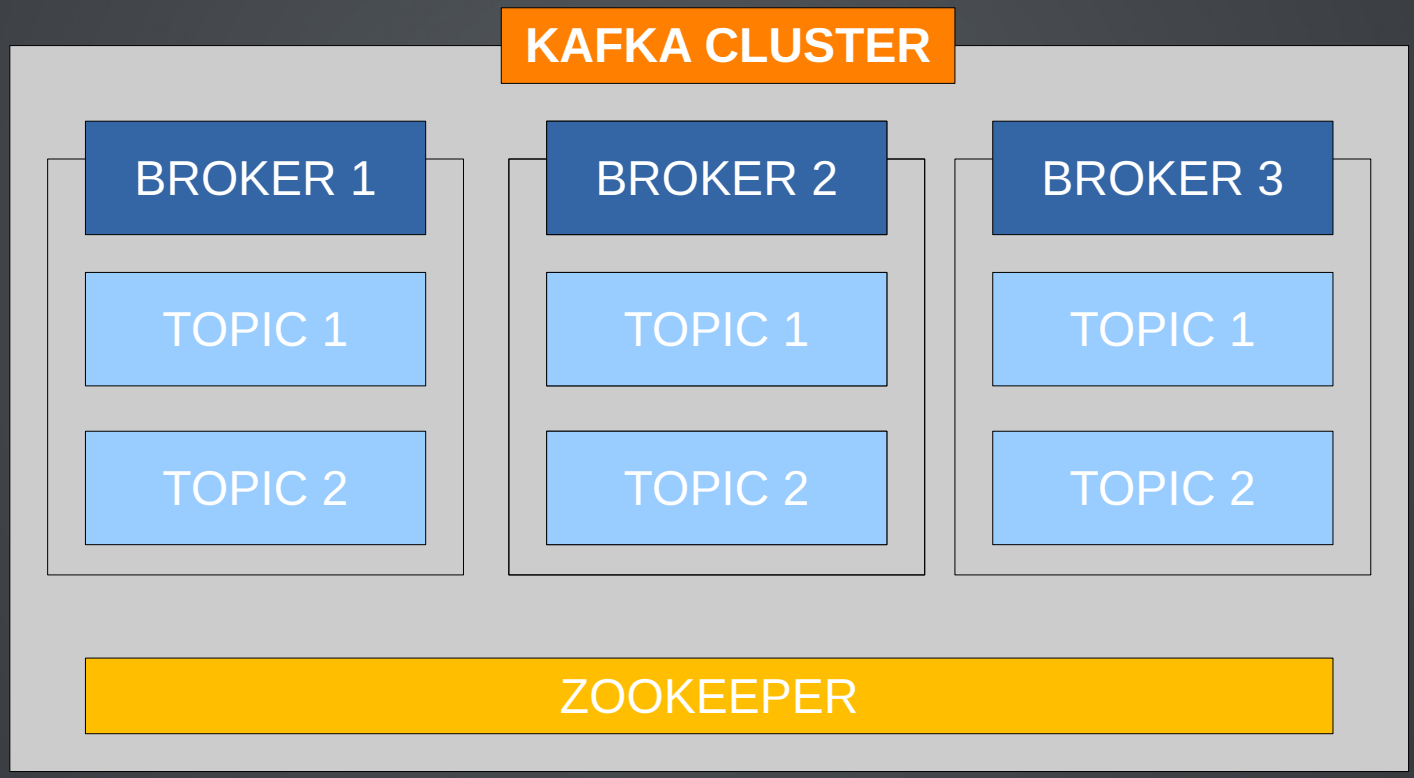
To be replaced by Kraft (based on Raft)

Cluster

A cluster is formed by multiple Kafka brokers

Kafka brokers communicate between themselves through Zookeeper

Kafka Architecture



Topic Replication Factor

Kafka design focuses on maintaining highly available and strongly consistent replicas

All replicas are byte-to-byte identical

If a broker is down another broker that hosts a replica of the data can serve it

CAP Theorem

All distributed systems must make trade-offs between guaranteeing consistency, availability, and partition tolerance

CAP Theorem

- Consistency
- Availability
- Partition tolerance

Partition Leader

For a given partition just one broker can act as the leader for that partition

The other brokers will replicate the data

This way each partition has one leader and multiple ISR (in-sync replicas)

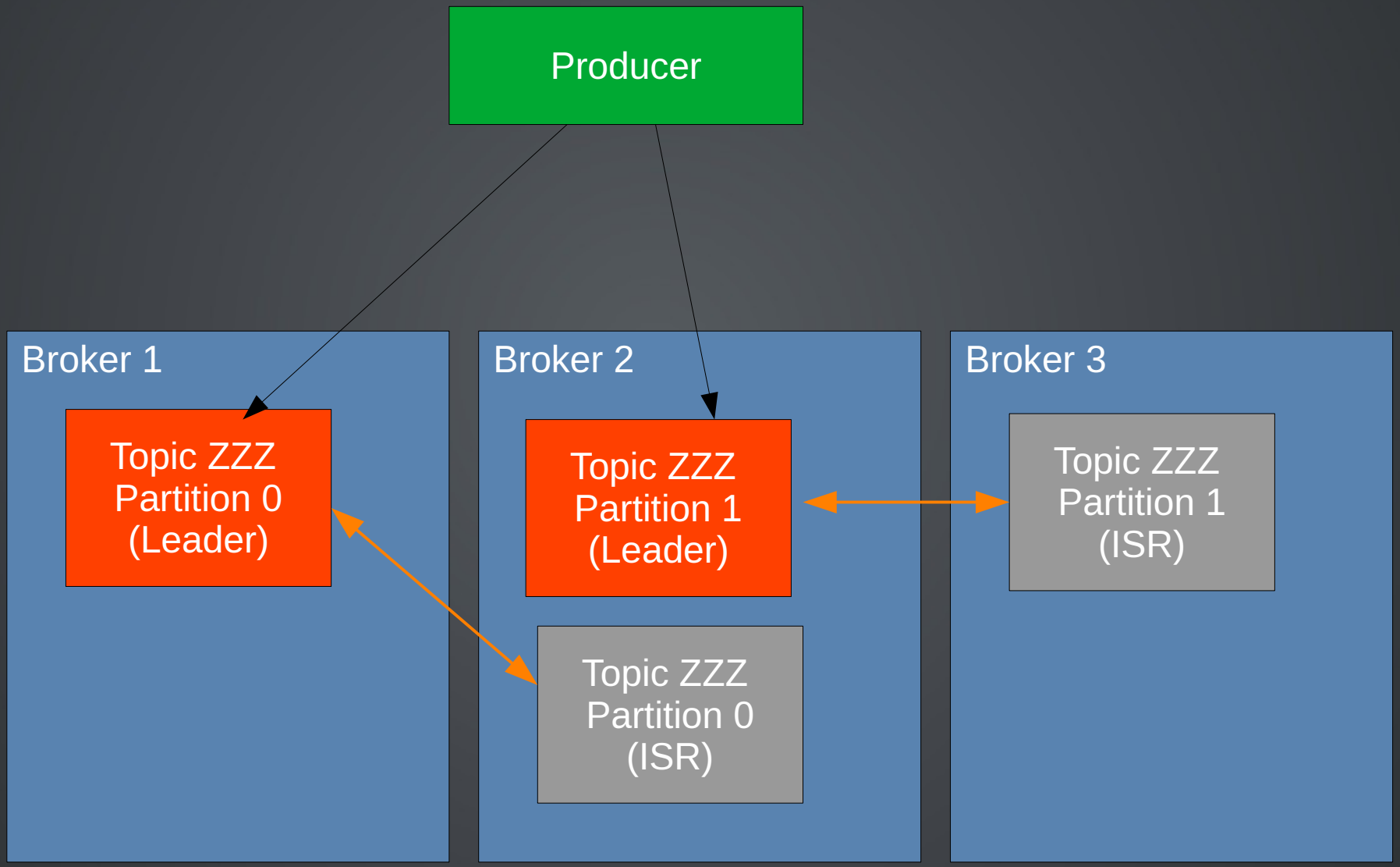
Partition Leader

Producers can only send data to the broker that is the leader of the partition

Consumers by default will read from the leader broker of a partition

But from Kafka 2.4 it is possible to configure consumers to read from the closest replica

Partition Leader



Producers

- Kafka producers send messages to topics
- Messages can be sent asynchronously
- Producers will automatically recover in case of broker failures
- Producers write to a given partition
- Producers decide to which kafka broker to write

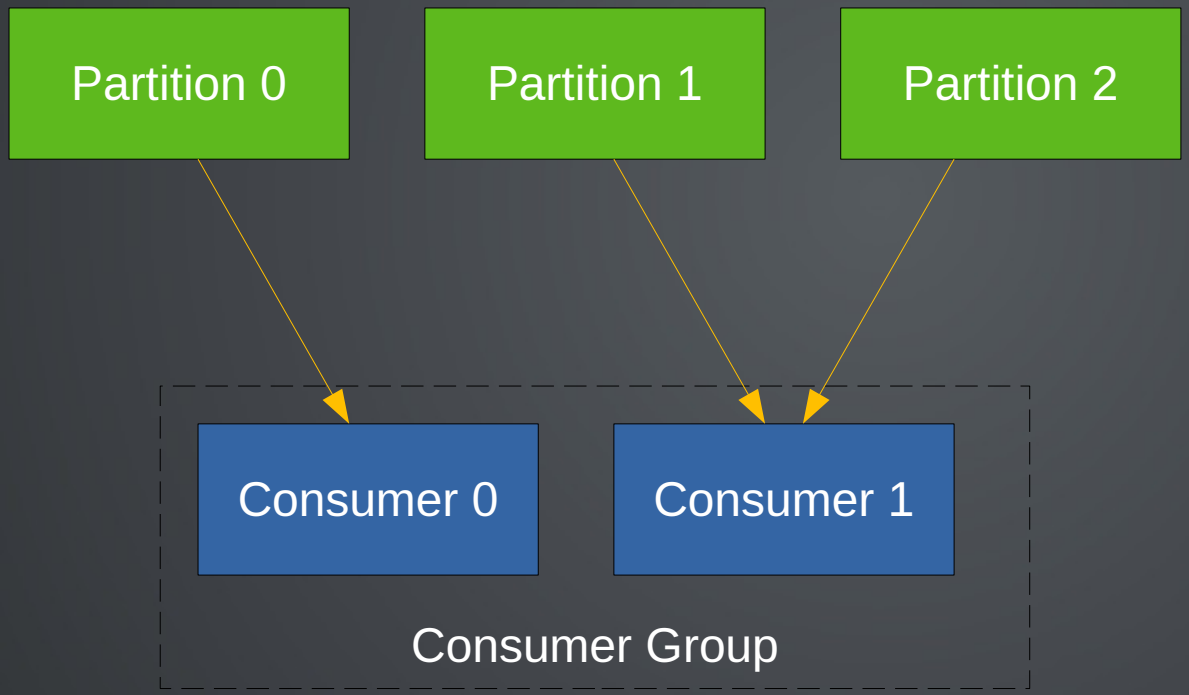
Consumers

- Kafka consumers read messages from topics
- Consumers must keep track of the partition offsets becase Kafka brokers are stateless
- Consumers can rewind or skip to any point in a partition
- Consumers will automatically recover in case of broker failures

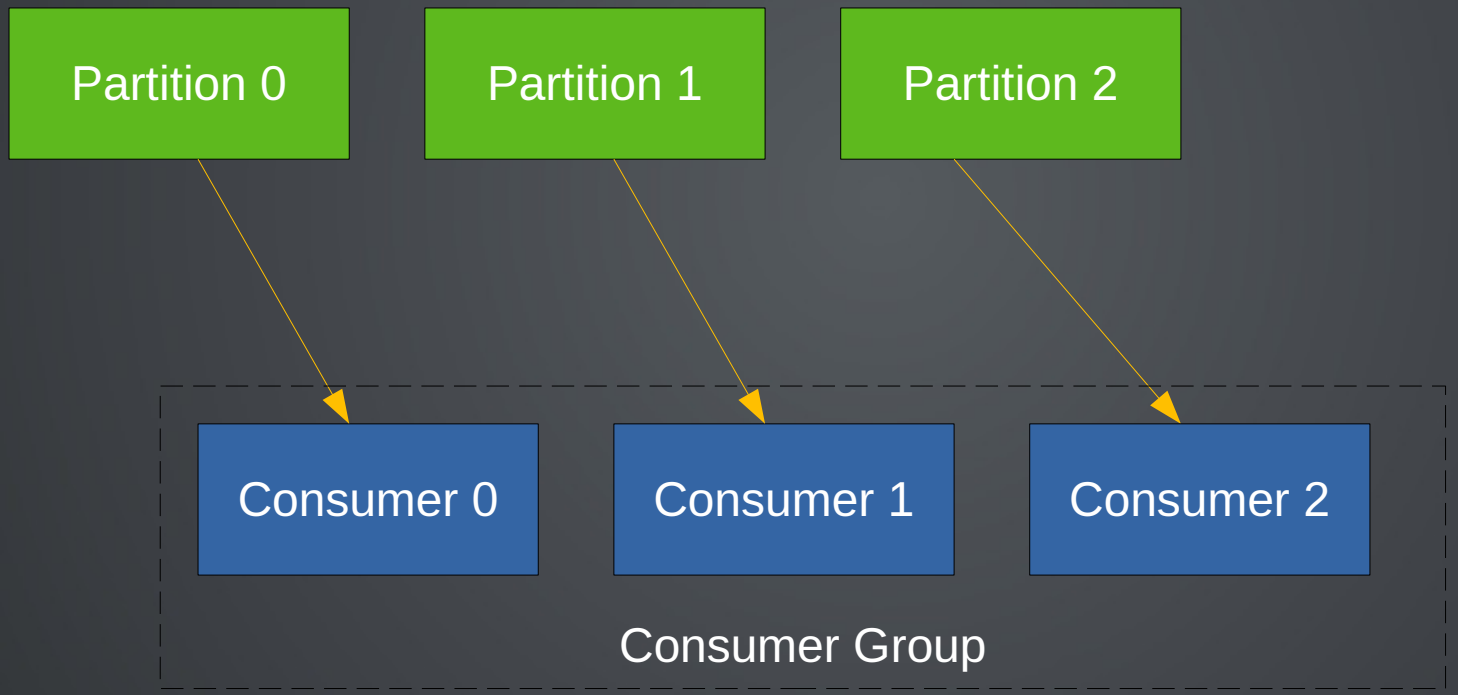
Consumer Groups

A Consumer Group is a set of multiple consumers coordinated to consume data together from the same topic

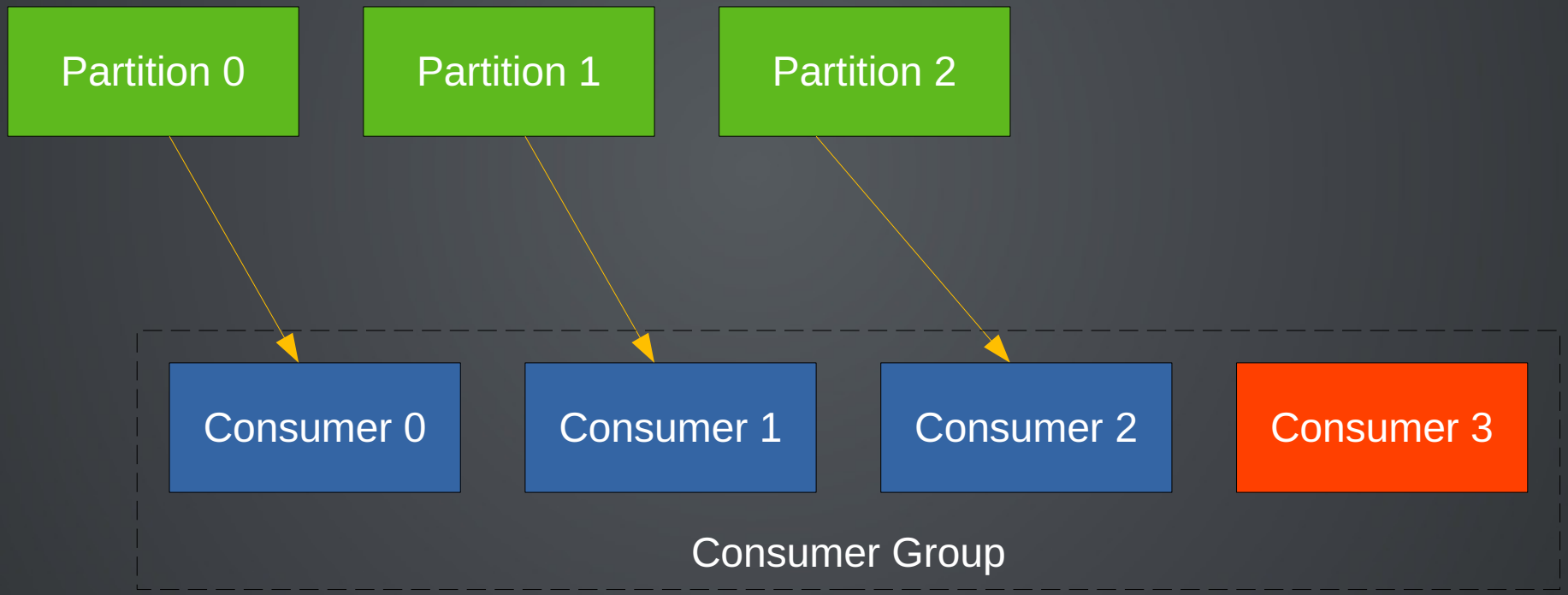
Consumer Group



Consumer Group



Consumer Group



Failure Recovery

If a consumer dies it will be able to read back from where it left using its consumer offsert

Consumer groups will automatically rebalance in case a consumer in the group dies or if new consumers are added

Delivery Semantics

At least once (default)

- Offsets committed after the message is processed
- If processing goes wrong then offsets are not committed and the message is read again
- Can result in duplicate processing of messages

Delivery Semantics

At most once

- Offset committed as soon as message is received
- If processing goes wrong, message will be lost

Delivery Semantics

Exactly once

- From Kafka to Kafka workflows: You have to use the Transactional API
- From Kafka to External system workflow: you have to implement an idempotent consumer

Producer Acknoledgements (Acks)

Producers can choose to receive acknowledgements of data writes

- In case acks=0 producer will not wait for ack (possible data loss)
- In case acks=1 producer will wait for leader ack (limited data loss)
- In case acks=all producer will wait for leader and replicas acks (no data loss)

Topic Durability

Keep in mind that the topic replication factor indicates the total number of replicas (included the leader), not the additional replicas

For example for a topic replication factor of 2 we can withstand 1 broker failure without data loss

In general, for a replication factor of N we could lose up to N-1 brokers

Kafka Broker Discovery

Each Kafka broker is also called a bootstrap server

Kafka clients (producers or consumers) only need to know how to connect to one bootstrap server

They will receive the information of how to connect to the entire cluster

Clients must be able to resolve the hostnames anounced by the cluster

Message Keys

Producers can send a key with the message

The key can be of a string, a number, binary data, etc.

Messages that have the same key will always go to the same partition (hashing)

If the key is null then data is sent round robin between partitions

Message Ordering

- Messages in the same partition are kept in order but not between partitions
- Data is read in order from lower to higher offset within each partition
- If you want all your messages ordered then you have to use a topic with just one partition

Message Anatomy

Key Type: binary (can be null)	**Value** Type: binary (can be null)
Compression Type None, snappy, gzip, ...	
Headers (optional) Key/Value pairs	
Partition + Offset	
Timestamp	

Data Serialization

Kafka only accepts binary data (bytes) in the messages (key and value)

So we have to serialize our data before sending them in our producer: ie. convert them to bytes before sending them to kafka

Data Deserialization

Kafka returns binary data (bytes) in the messages (key and value)

So we have to deserialize the data in our consumer: ie. convert it back from bytes into objects/variables

Best Practices

Keep the same serialization/deserialization mechanism during the lifecycle of a topic

If you have to change the mechanism create a new topic

Best Practices

Do not re-invent the wheel, in production use existing Serializers/Deserializers:

- JSON
- Avro

Best Practices

When using the default delivery semantic of *at least once* (default) make sure the application code is prepared to detect and discard duplicated messages

Best Practices

Topics should have a replication factor greater than 1 for fault tolerance

When using consumer groups check the balance between the number of partitions and the number of consumers in the group

Final notes: Zookeeper

As we have seen Zookeeper is used to keep a list of the brokers in the cluster

Helps in performing leader election for partitions

Sends notifications to Kafka in case of changes (ie. broker comes up, broker dies, new topic, topic deleted, etc.)

Final notes: Zookeeper

In old versions consumer offsets where stored in Zookeeper

But since Kafka 0.10+ consumer offsets are no longer stored in Zookeeper

Now consumers store offsets in Kafka itself, in a topic named __consumer__offsets

If a consumer dies it will be able to read this topic and then recover reading back from where it left

Final notes: Zookeeper

Kafka 2.x and older can’t work without Zookeeper

Final notes: KRaft

- Kafka 3.x can work without Zookeeper using Kafka Raft (KRaft)
- KRaft marked production ready in 3.3
- ZooKeeper planned to be removed in 4.0

Final notes: Kafka Message Key Hashing

A Kakfa partitioner is a piece of code that takes a message and determines which partition to send it to

Key Hashing is the process of determining the mapping of a key to a partition

Final notes: Kafka Message Key Hashing

DefaultPartitioner is a Partitioner that uses a 32-bit murmur2 hash to compute the partition for a record (with the key defined) or chooses a partition in a round-robin fashion (per the available partitions of the topic).

Further Information

- Kafka: The Definitive Guide
- Apache Kafka Series - Learn Apache Kafka for Beginners v3
- All about Apache Kafka – An evolved Distr ibuted commit log