Apache Kafka

- Apache Kafka Introduction
- Playing with Apache Kafka
- Consumer groups

Apache Kafka Introduction

What is Kafka?

Kafka is a data streaming platform, just like Amazon Kinesis it is able to take in message and store messages durably for consumers to read in those messages.

It is very scalable because of the three different components, producers, brokers, and consumers. There is also built-in data replication to be a durable platform.

Core concepts

Messages or a record in Kafka is how you would be sending the data from the producer to the broker. The record consists of key and body. The key is used to identify which partition of a topic to write the record into.

A collection of message or record is referred to as a batch.

Kafka topic

A Kafka topic can be think of as a container that you can deposit your message / record into in the Kafka ecosystem and be read on a later time. Topics themselves are identified by a unique name, and the messages are sent to and read from a specific topic.

Kafka partition

Within a topic, it can have one or more partition, further division of a topic. When a topic is created the controller (head of the cluster) will determine how many partition a topic will have.

So a topic can be think of as a container to deposit your message into, then partition is more smaller boxes within the container to further organize your message.

Records will be stored into a topic based on the provided name, then based on the provided key which will be hashed in order to get the actual partition the record will be placed into.

Example

For example, if we have a topic name **activity-log** with three partition named **activity-log-1, activity-log-2, activity-log-3.**

Then when a source system publishes messages to the topic it will be stored into either of those partition.

Kafka broker and cluster

A Kafka broker is the server that handles the request from both Producer, Consumer, and Metadata queries. They are also responsible for keeping the data replicated within a cluster.

A Kafka cluster just have multiple Kafka broker in it handling requests to keep Kafka platform running, that is it.

Zookeeper

Zookeper is another set of servers that is responsible for metadata maintenance. They are able to direct producer the brokers to contact if there are multiple brokers.

Producer

The producer are processes that publishes records into Kafka topic via broker.

Consumer

A consumer are processes that pulls records off a Kafka topic via broker.

How are replication done?

In Kafka replication is implemented at the partition level. The redundant partition in a topic is called a replica. Each partition (that actually gets message published into) in a topic usually have one or more replicas associated with them.

Within a partition, partition and replica partition, there is the leader which handles all the read-write operations for the specific partition. Then the replicas will be replicating the leader partition.

If the leader fails, then one of the replicas will be promoted as the leader to take over.

What is a bootstrap-server

A bootstrap-server is a url for one of the Kafka brokers that allows you to fetch initial metadata about your Kafka clusters. Which topics are available and the number of partitions within each of the topics, which partition is a leader.

Producer or consumer will use these metadata to produce and consume from the appropriate topic / partition / contact the right broker.

There can be multiple bootstrap-server for failover purposes just in case one of them goes down.

Schema registry

Kafka at its core only transfer data in bytes, the data stored in topics are in raw bytes format, when you publish or consume from topic it is read in as raw bytes format. The consumer needs to know about the type of data the producer is sending in order to deserialize it later on (Get it back as an Object, how to transfer complex objects like a Linked list across network - Use serialization sent the object as array of bytes). Producer serializes the data using library like Avro in order to store it into raw bytes.

This is where Schema registry comes into play, it is an application that lives outside of Kafka cluster and handles distribution of schemas to the producer and consumer by storing the schema (layout of the object how to deserialize) in its local cache.

The producer before sending the data to Kafka, first check with registry to see if the schema is available, if not sent it and registry will cache it. Then it will serialize the data with the schema and send it to Kafka with a schema ID.

When the consumer gets the message, it will first get the schema from registry with the ID, and then deserialize it according to the schema.

The schema basically tells the consumer HOW to deserialize the bytes, what bytes constitute the first field, second bytes, and so on.

Playing with Apache Kafka

Kafka Setup

In this article I'm going to use Kafka with Zookeeper instead of KRaft. This is because Zookeeper has been for around a long time and lots of company has been using Zookeeper instead of upgrading it to KRaft.

Zookeeper responsibility

In older version of Kafka, you cannot use Kafka without first installing Zookeeper but this requirements was removed starting with v2.8.0 version of Kafka, it can be run without Zookeeper however, this is not recommended in production.

Zookeeper's responsibility in this distributed system is to coordinate tasks between different Brokers. Remember that Kafka cluster is made up of one or more Kafka Brokers. The brokers are responsible for handling the client's request, for both producer and consumers. The topics are enclosed within the Kafka Brokers and they are also responsible for replicating the partitions within the topics.

A Kafka Broker can have more than one topic enclosed within them, it doesn't just have to be one topic that they are helping to coordinate the writing and consuming from.

Zookeeper's responsibility

1. Controller election: Every Kafka Cluster has a controller broker that is responsible for managing the partitions and replications, basically admin tasks. The Zookeeper will help to pick out one that does the job
2. Cluster membership: Zookeeper keeps the list of functioning brokers in the cluster
3. Topic configuration: Zookeeper also maintains the list of all topics, the number of partition in each topic, the replica partitions, and leader nodes
4. Quotas: Zookeeper can accesses how much data each client is allowed to read/write
5. Access Control List: Zookeeper also can control who and what kind of permission the client can have on each topics.

Basically Zookeeper is used for metadata management, it doesn't really affect producers and consumer.

Offsets

Current offset

Whenever a consumer polls for messages from Kafka, let's assume we have 100 records in a particular partition that the consumer is polling from. The initial current offset will be 0 for the consumer, after we have made our call and we receive 20 messages. The consumer will move the current offset to 20. when we make our next request it will retrieve messages starting at position 20 and then move the offset again forward after receiving the messages.

This "current offset" is just a simple integer that is used by Kafka to maintain the current position of one consumer. That is it. It is just used to maintain the last record that Kafka sent to this particular consumer, so that the consumer doesn't get spammed with the same message twice.

Committed offset

Committed offset is used to confirmed that a consumer has confirmed about the processing of the record after it has received them. The committed offset is a pointer to the last record that any consumer has successfully processed. This offset is used to avoid resending the same record to a new consumer in the event of partition rebalance (This occurs when ownership of a partition changes from one consumer to another at certain events described in section below).

Taking the example of 100 records again let's say 20 records are already processed, and a brand new consumer joins into the group and it wants to help process the message from the partition as well, where should it start? It should process those records that are processed by the previous owner of the partition.

With committed offset you can do auto committing (DEFAULT), or manual committing.

Auto committing you just set a interval say 5 seconds, and everytime your consumer polls for records it will check whether 5 seconds has been and if it has it will commit the offset and poll for records. This option is convenient but it might result in duplicate processing of messages because if a new consumer joins in the group and rebalancing is trigger and the partition goes to a different consumer, as the new owner of the partition it will not see the auto committed offset and thus reprocess the same records that has been processed by the previous owner of the partition.

To solve the previous issue manual commits is used. There is two types of manual commits, synchronous commits and asynchronous commits.

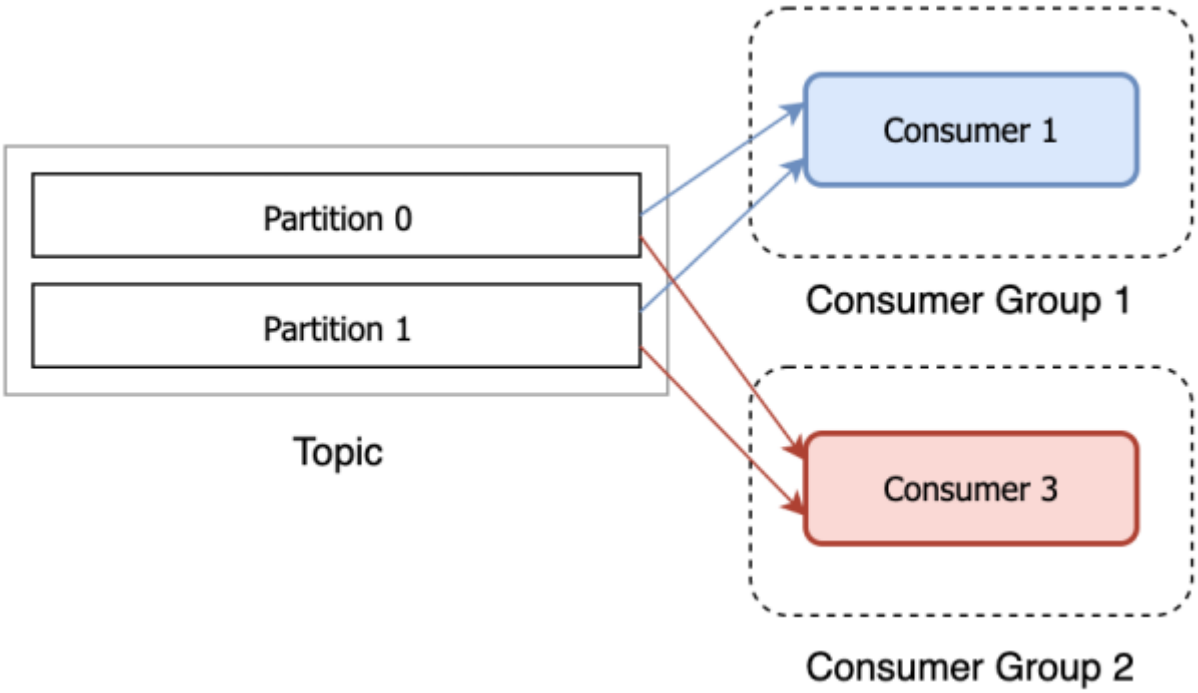
Synchronous commits: It is blocking, it will block until the commits succeeds

Asynchronous commits: It is non-blocking, if it fails it will not be retried. It is purposely designed not to retry otherwise it the ordering problem will occur. If commit to 75 failed but commit to 100 succeed, but 75 is retried and succeed it will cause problem. Therefore, Kafka avoided this problem just by not retrying async commits.

Consumer groups

A consumer group is a group of consumers that shared the same **group id**. Consumers from the consumer group will then be assigned a partition (if available, because again if there are more consumer than the partition, the extra consumers will just sit idling).

This means that if you want to consume the same record (from same partition) from multiple consumers, they MUST be under different consumer groups. Otherwise, the consumer will process the same records.



How does consumers keep track of committed and current offset

Let's start with current offset because this is easy. When a consumer reads messages it will increment the current offset so that IT will not get duplicate messages. This is kept track of by the consumer internally. This is more for the consumer itself, that "oh I got messages and next message I need to get starts at position 20".

Committed offset, this offset is taken care by the group coordinator by producing a message to an internal **__consumer_offsets** topic. The offsets are stored for each `(consumer group, topic, partition)` tuple, **so that you can distinguish different commit offsets for each topic, each partition, and each consumer group.** (Source)

What the hell is `auto.offset.reset`

This property is used to define the behavior of the consumer when there is no committed position.

For example, when the consumer group is first initialized, or when an offset is out of range. They must decide from which point to start to poll the messages.

You can choose to set it to earliest or the latest (default) offset.

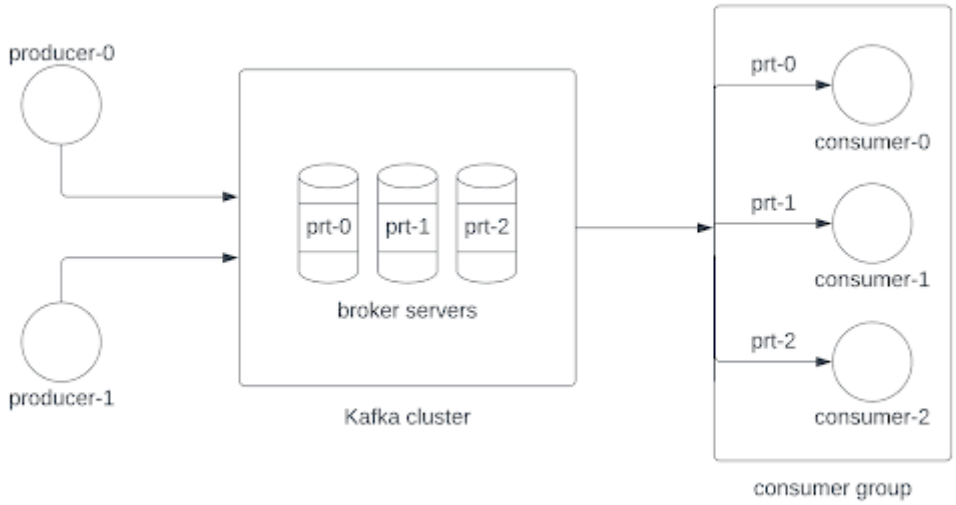
Earliest: Read messages from the beginning of the partition, processing every message present in the partition.

Latest: Only read new messages written to the topic once the consumer has begun listening (so ignoring all the existing messages)

Partition rebalancing

The process of changing partition ownership across a consumer group is called the partition rebalancing.

Let's say for example, each partition has it's consumer in a consumer group laid out like below:

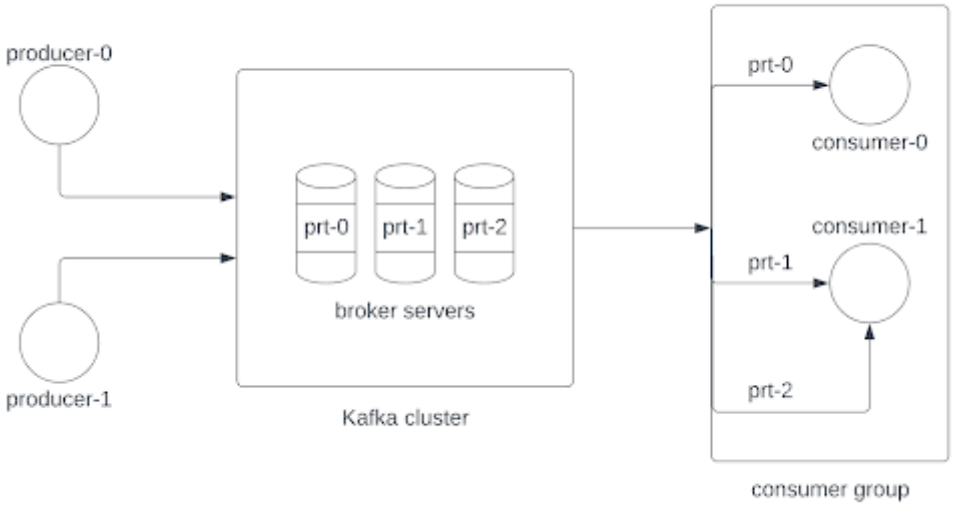


If one of the consumer in the consumer group goes bad due to variety of reasons:

1. Consumer unsubscribed from the topic
2. Consumer doesn't respond to heart beats
3. Consumer hasn't polled the topic for a while
4. Topic is modified (new partition is added)

They will be removed from the consumer group and a partition rebalancing will occur, where the partition will be redistributed with the remaining healthy consumers. Note that this doesn't affect producers because they can still produce to the topic without any issue, only the consumers will be blocked until the partition rebalancing is completed.

The result of the rebalancing will may result in:



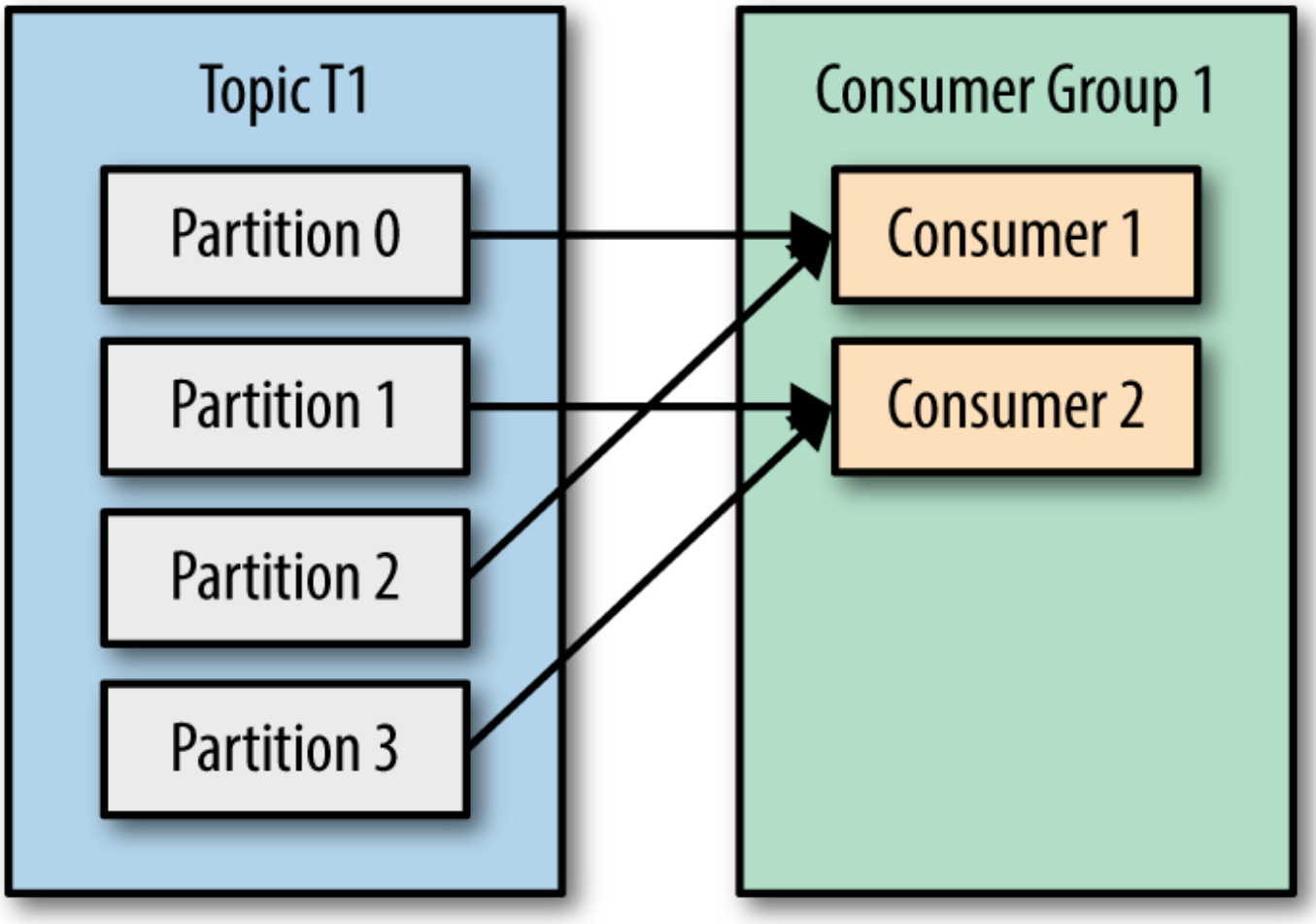
> Note that in the case of more partition than consumers, the consumer can be assigned multiple partitions.

Partition rebalancing is needed to make sure the consumer group is high availability and is scalable.

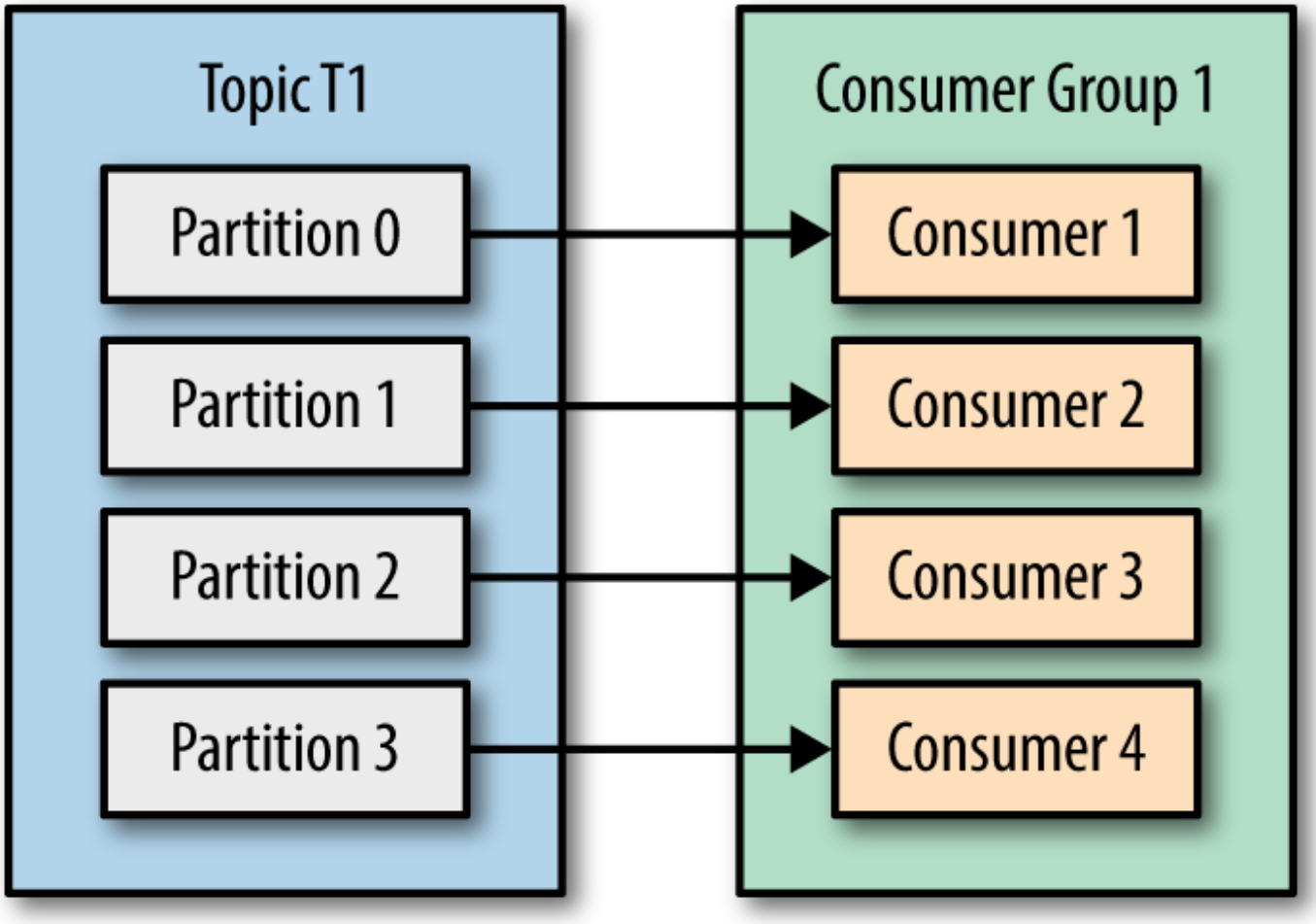
Consumer groups

This page just details out the possible scenarios where the number of partitions and consumers in a consumer group may differ.

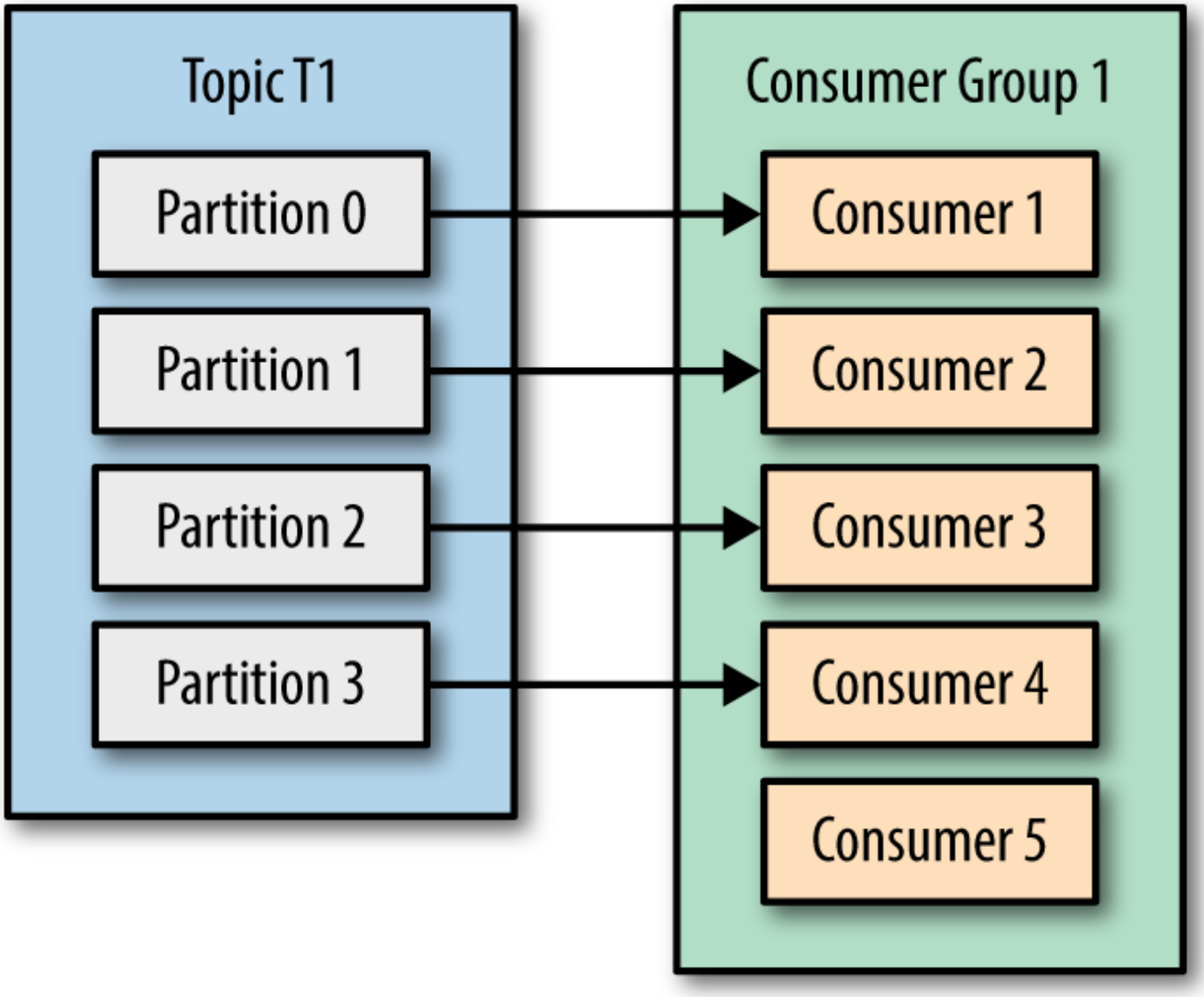
- If the number of consumers is less than the number of topic partitions, then multiple partitions can be assigned to one of the consumers in the group



- If the number of consumers is the same as the number of topic partitions, then partition and consumer will be mapped one to one, one possible mapping can be



- If the number of consumers is higher than the number of topic partitions, then partition and consumer will be mapped one to one as well, except those consumers without a partition will just be idling without a partition to consume from.


Topic T1
Partition 0
Partition 1
Partition 2
Partition 3
Consumer Group 1
Consumer 1
Consumer 2
Consumer 3
Consumer 4
Consumer 5