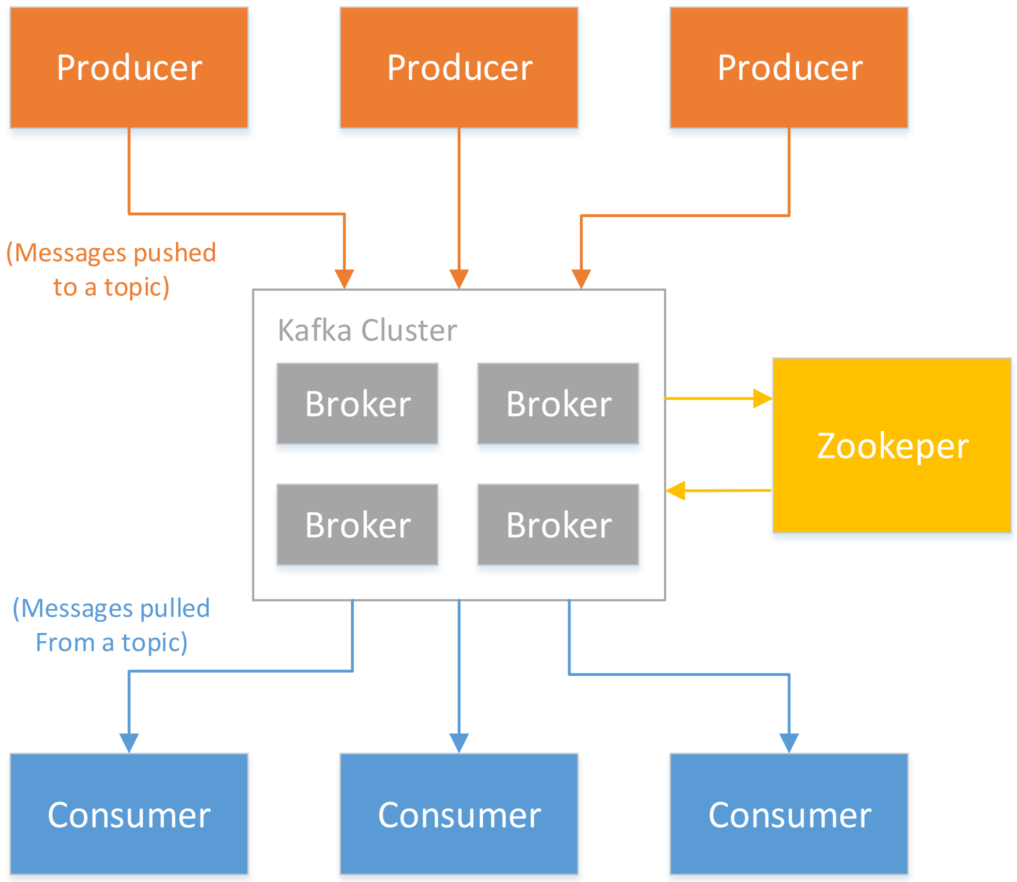
# Overview of Kafka

**Author: Peter Bugaj**

## Table of Contents

## Main Components

- **Zookeeper**, for helping manage brokers for Kafka.
- A **Kafka Cluster**, made up of one or more servers called brokers.
- A **producer**, for publishing messages to a specific topic.
- A **consumer** for reading the messages.
- The **messages** themselves.
- **Brokers** responsible for helping the messages be processed in parallel.



## Brokers

- A server or instance running as a part of a Kafka Cluster. A Kafka Cluster has multiple servers / brokers for scalability and redundancy.

- Takes data from producers, stores it, and then sends it to the consumers on their request.

- Handle replication of partitions.

- Assigns the incremental offsets to new messages. This is done before storing the message.

- Deletes the messages based on a setting specifying a time limit, or by deleting the oldest messages after a specified size limit. Different topics can be configured to have separate settings. Messages can also be log compacted.

- One broker is called the controller broker or the cluster controller, which does administrative functionality like assigning partitions to brokers and keeping track of broker failures.

- Uses Zookeeper for distributed coordination.

- On modern hardware (2017), a broker can handle 2 million messages per second.

- Receives heartbeats from consumers in a consumer group. If no heartbeat is sent after a set timeout, it assumes the consumer is dead and signals the group that it needs to to be rebalanced.

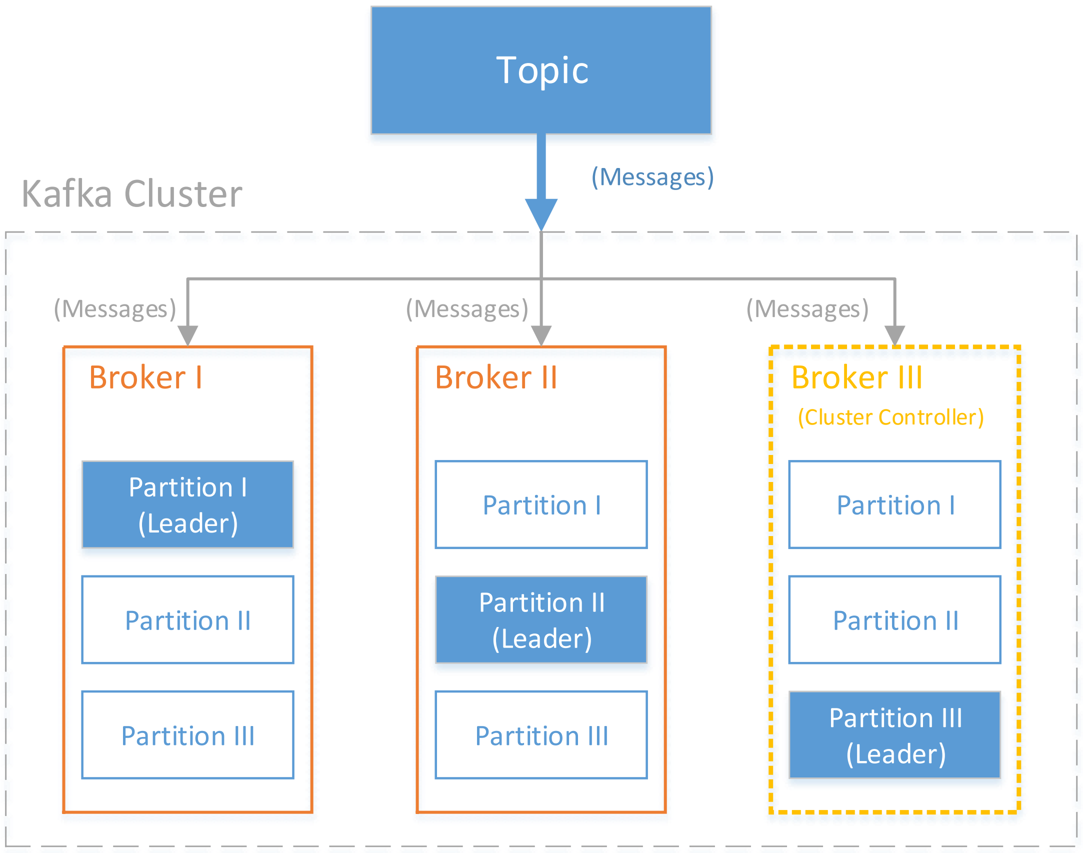
## Producer

- Publishes messages to a topic. A topic is split across multiple partitions to increase scalability. Hence, a publisher writes to multiple partitions at random using a hashing function to help distribute the messages evenly across each partition.

- To write to a specific partition, a publisher specifies a key.

- Usually consumes an existing data feed that arrives to Hadoop. A producer can also distribute data to consumers, which write their data yet again to other producers.

- Runs the client API of the Kafka system.



## Consumer

- Reads messages from the Kafka cluster.
- Part of a consumer group. The group ID of the consumer specifies what group it belongs to.
- One consumer can read from one or more partitions, but a partition can be read from at most one consumer in a consumer group.
- A mapping of a consumer to partition is an ownership of the partition by that consumer.
- Consumers in a consumer group are assigned a partition.
- Once assigned a partition, the consumer begins to poll the partition until there are no more messages. It waits for a specific time until it decides that there are no more messages. This is controlled by the parameter named poll. Alternatively the consumer can wait for a very long time and be controlled by the wakeup API.
- A consumer keeps track of its partition ID and cursor position / message offset in Kafka (previously stored in Zookeeper). This allows the consumer to freely stop and restart its reading.
- A consumer commits the offset it reads up to as well. This allows Kafka to have the information needed to rebalance the messaging system in case the consumer goes down. In such a case, a partition must be reassigned to a new consumer, and that consumer will start reading from the last committed offset.
- Messages with the same key will arrive at the same partition, hence to the same consumer reading that partition.
- Runs the client API of the Kafka system.
- Sends heartbeats to the cluster controller. A heartbeat is embedded in the poll mechanism.

## Consumer Group

- Used for subscribing to a topic. A consumer must be part of a consumer group.

- Kafka guarantees that a message is read by a single consumer within the group and by no more than one if there is no consumer failure. Based on the commit policy of a consumer, messages might be read more than once during such a failure.

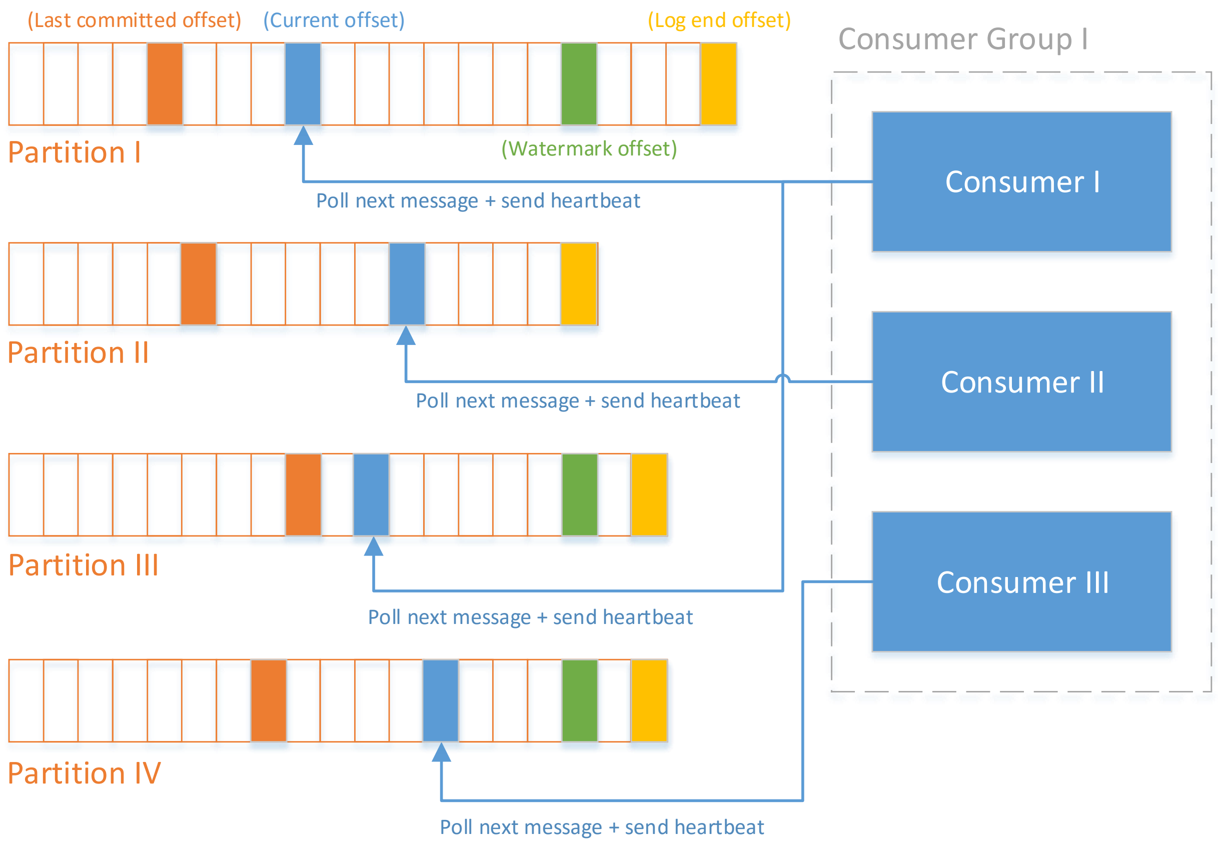
- Consumers within the group are mapped to one or more partitions, but a partition is mapped to only one group.

- Multiple consumer groups can subscribe to one topic.

- If a consumer goes offline, Kafka reassigns the partition to an existing consumer with the offset last committed by the dead consumer. For example, if a consumer commits offset 10 and then reads up to offset 20 before dying, the next offset would pick up from offset 10, reading the last 10 messages again.

- If a new consumer is added to the group, Kafka rebalances.

- Identified by a Group ID.



## Offsets

- Includes the last committed offset by the consumer.
- The current offset read by the consumer not yet committed.
- The high water maker offset marking the position in the log where everything has been successfully replicated to each of the replicas.
- The log end offset marking the highest offset committed so far by the producer.

## Commit Policy

- Auto commit for offsets can be set to true of false. If enable.auto.commit is set to true, a consumer in will commit an offset after auto.commit.interval.ms milliseconds.

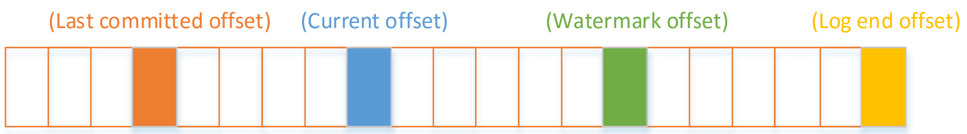
- How often a consumer commits an offset determines the number of duplicates created during a crash.

- An “at least once” policy is enforced by waiting for messages to be processed before committing an offset. This ensures that if the consumer crashes, those messages can be processed again.

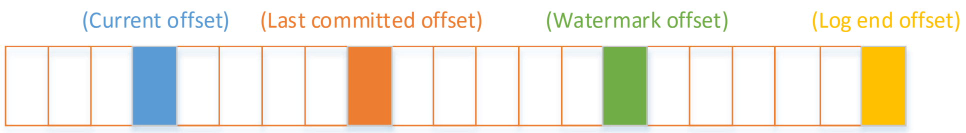
- An “at most once” policy is enforced by committing the offset first. If the consumer crashes before reading the messages, all of them might potentially be lost. This however avoids duplicates.

- An “exactly once” policy is when an offset is committed after each message is read. This guarantees that no messages are lost and that there are no duplicates. However, this has a much heavier overhead as a commit might take time.

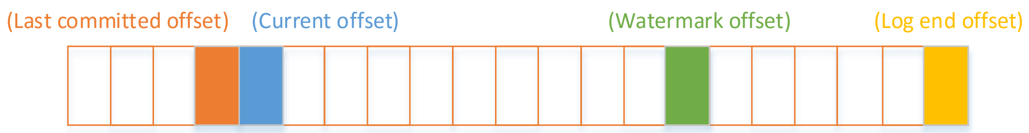
- Another policy is to commit only after all messages are read from one partition.



"At least once" policy – current offset is ahead of the last committed offset



"At most once" policy – offset is committed first before data is read



"Exactly once" policy – offset is committed immediately after a message is read

## Messages

- Saved in disk as a byte array.

- Removed based on a configured time, or after a certain size is reached, where the oldest messages will then start being removed.

- Is a key value pair.

- A message can only be sent to one topic.

- Written in batches for efficiency. Batches are written to the same topic and partition. The batches can be increased or decreased. A larger batch increases the performance, but takes no longer for messages to be sent as Kafka puts them on hold until there is a full batch. Small batches or the transfer of individual messages is a lot of overhead per message. Batches can also be compressed and typically, compression improves for larger data.

- Each message has a unique offset.

## Partition

- Splits the messages in a topic into multiple components. Partitions are replicated across multiple brokers in a Kafka cluster for redundancy.

- A partition has a leader server and one or more follower servers.

- Consumers and producers only communicate to the partition leader.

- A publisher publishes a message to a topic by having the message sent to one of the brokers holding the partitions that partition that topic. The broker is chosen using a hashing algorithm.
- By default, only the consumers are aware of the partitions. Producers do not know about the number of partitions or where they send the messages to, but can use a key to specify that a message should always be sent to the same partition from that producer.

- Messages are stored in the order that they are received, and are consumed in that same order.

- Since a topic usually has multiple partitions, there is no order guarantee for messages throughout the entire topic.

- There can be more than one partition for the same topic on one server.

- A partition exists as a leader only on one broker. Other brokers can have a copy of this partition, but these partitions are referred to as replicas or as followers of the partition leader.

## Zookeeper

- Stores configurations for Kafka, like the topics existing per broker and offsets for consumers (now stored in Kafka directly).

- Monitors the health of the brokers.

- Checks when new brokers are added or when brokers go down. In the case of a new broker, Zookeeper will start utilizing it by giving it partitions for a topic.

- Zookeeper used to keep track of the offsets per consumer. Now this has been moved to directly into Kafka.

## Mirror Maker

- Replicate Kafka clusters.

- Makes the system resilient against data center disasters.

## Exercises

### Pushing and consuming data

1. Setup Kafka to read data from one centralized location in Hadoop to another centralized location.

2. Read data from location into another location, which again is used as the source for another producer that sends that data and has it pull by a consumer into a tertiary location.

3. Have a producer push data to one topic with multiple partitions, for each broker, and have consumers consume that data. Have each consume print to a log the partition it is reading from, along with its offset and last committed offset, to show what each consumer is doing.

4. Push data from one location using one producer, and have two separate consumer groups, consuming the source data into two separate locations. Have each consumer within each consumer group log its output, including the offsets it is handling, the group it is part of, and the partition it is reading.

5. Have two producers reading data from two separate locations and have one consumer group subscribed to those two topics, outputting the data to one centralized location. Have each consumer log the data its reading from, including the partition and offset it's keeping track of.

6. Use a key to push data to a specific partition and show the same consumer reading data from that partition.

7. Demonstrate an example of a pub system (one publisher to multiple consumers) and a message queue system (one publisher to one consumer).

8. Implement a data type for each message, so that one consumer can process messages from the same producer differently, based on the data type.

## Scaling and optimization

9. Have a producer push data to a topic and show the runtime comparison for a different number of partitions, from one partition, to where the number of partition equates to the number of brokers, to more partitions than brokers.

10. Have a producer push data to a topic and show the runtime comparison for a different number of consumers in a consumer group, from one consumer, to where the number of consumers equates to the number of brokers and the invalid case where there are more consumers than there are partitions.

11. Add an additional VM in real-time while Kafka is running and show how the new VM starts being utilized. Show if this is even possible while in the process of a large amount of data being streamed.

## Spark integration

12. Push data and consume it with spark. Have spark run a map reduce to transform the output before saving it to disk. For example, if the input is a list of words, write each word in reverse.

13. Push data and consume the output with spark. Have spark sort and group the data by category. Sorting will force Spark to read everything into RAM, and partially onto the disk if needed.

## Downtime handling

14. Demonstrate an “at least once” commit policy. Start transferring a text file of a certain size. Kill one of the running VMs utilized by Kafka. The “at least once” policy results in some messages being read more than once. Show that the output data is larger than the input.

15. Demonstrate an “at most once” commit policy. Transfer a text file of a known size, kill a VM in the process and show that due to the recovery, the final data written by the consumers is smaller.

16. Demonstrate an “exactly once” commit policy. Transfer data of a known size from one location to the other and show that both the output and input data is of the same size, even after one VM is taken down during the transfer process.

17. Tell consumers to sleep between each message the consume. Show that data can no longer be transferred if consumers sleep for a longer time then the poll setting used for specifying how long the controller cluster is to wait for a heart beat.

18. Give an example of a “wakeup” call being used on a consumer.

## Disasters and miscellaneous

19. Kill the admin instance for Kafka and see what happens.

20. Run Cloudera on a different VM from the set of VMs it is supposed to manage.

21. Kill the instance running Zookeeper and see what happens.

22. Show ways to utlize space between Kafka and Spark.

23. Show what Kafka Connect and Kafka Stream APIs are.

## Online References

### Installation

- https://sookocheff.com/post/kafka/kafka-quick-start/
- https://www.tutorialspoint.com/apache_kafka/apache_kafka_basic_operations.htm
- https://youtu.be/iJID-8XTcAU

### Architectural

- http://blogs.quovantis.com/learn-how-to-start-working-with-kafka/
- https://anturis.com/blog/apache-kafka-an-essential-overview/
- https://thenewstack.io/apache-kafka-primer/
- http://spideropsnet.com/site1/blog/2015/06/19/real-time-big-data/
- https://www.safaribooksonline.com/library/view/kafka-the-definitive/9781491936153/ch04.html

### Informative

- https://www.confluent.io/blog/how-to-choose-the-number-of-topicspartitions-in-a-kafka-cluster/

### Examples

- https://www.infoq.com/articles/apache-kafka
- https://www.confluent.io/blog/tutorial-getting-started-with-the-new-apache-kafka-0-9-consumer-client/

**Questions**

- http://stackoverflow.com/questions/35561110/can-multiple-kafka-consumers-reading-same-message-in-the-partition
- https://www.codementor.io/tips/0443378291/in-apache-kafka-why-can-t-there-be-more-consumer-instances-than-partitions
- http://databasefaq.com/index.php/answer/204727/apache-kafka-multiple-consumer-groups-in-kafka
- http://stackoverflow.com/questions/23136500/how-kafka-broadcast-to-many-consumer-groups
- http://stackoverflow.com/questions/30899163/how-do-i-use-multiple-consumers-in-kafka
- http://stackoverflow.com/questions/40162370/heartbeat-failed-for-group-because-its-rebalancing