KafkaArchitecture

In Kafka producers publish messages to a topic.
These topics are stored on brokers which are physical computers/machines.
A consumer subscribes to topics and pulls messages from the brokers which contain the subscribed topics.
Each kafka cluster is divided into brokers , and each broker is aware of all the topics and partitions even though it may only store some topics and partitions.
Multiple producers and consumers can produce and retrieve messages at the same time.

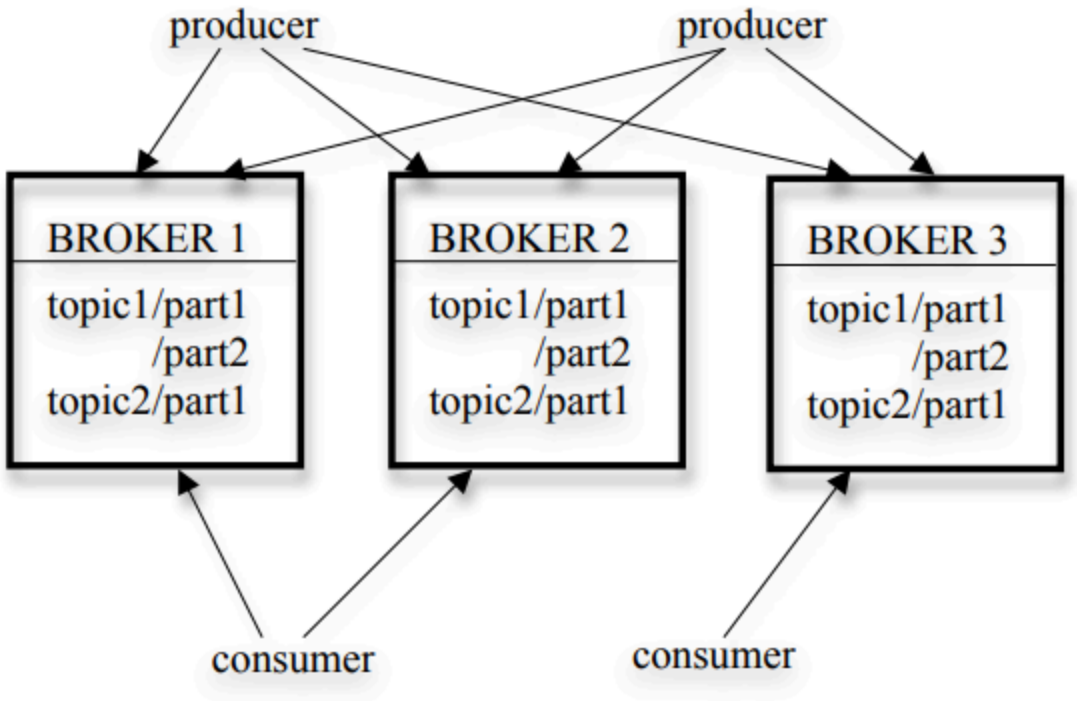


Figure 1. Kafka Architecture

Each partition is basically a log. A log is implemented as a set of segment files of about 1Gb.
Every time we produce a message it is basically appended to the segment file and the segment file will be flushed to the disk and a certain amount of time or after a certain number of messages have been published.
Only when the message is flushed it is exposed to the consumers.
Each message stored in Kafka doesnt have a message id instead it has an offset in the log.This avoids the overhead of maintaining index like structures that map the actual physical location to the logical location.
The message ids are always increasing but are not consecutive.
To compute the id of the next message, we have to add the length of the current message to its id

A consumer always consumes messages from a particular partition sequentially.When a consumer acknowledges a message offset it implies that it has also received all the other message offsets in that partition.
The consumer issues async pull requests to the broker to get a batch of messages.
Each pull request of the consumer contains the message Offset from where the consumption should begin and the acceptable number of bytes to fetch.
Each broker keeps in memory a sorted list of offsets, including the offset of the first message in every segment file. The broker locates the segment file where the requested message resides by searching the offset list, and sends the data back to the consumer. After a consumer receives a message, it computes the offset of the next message to consume and uses it in the next pull request

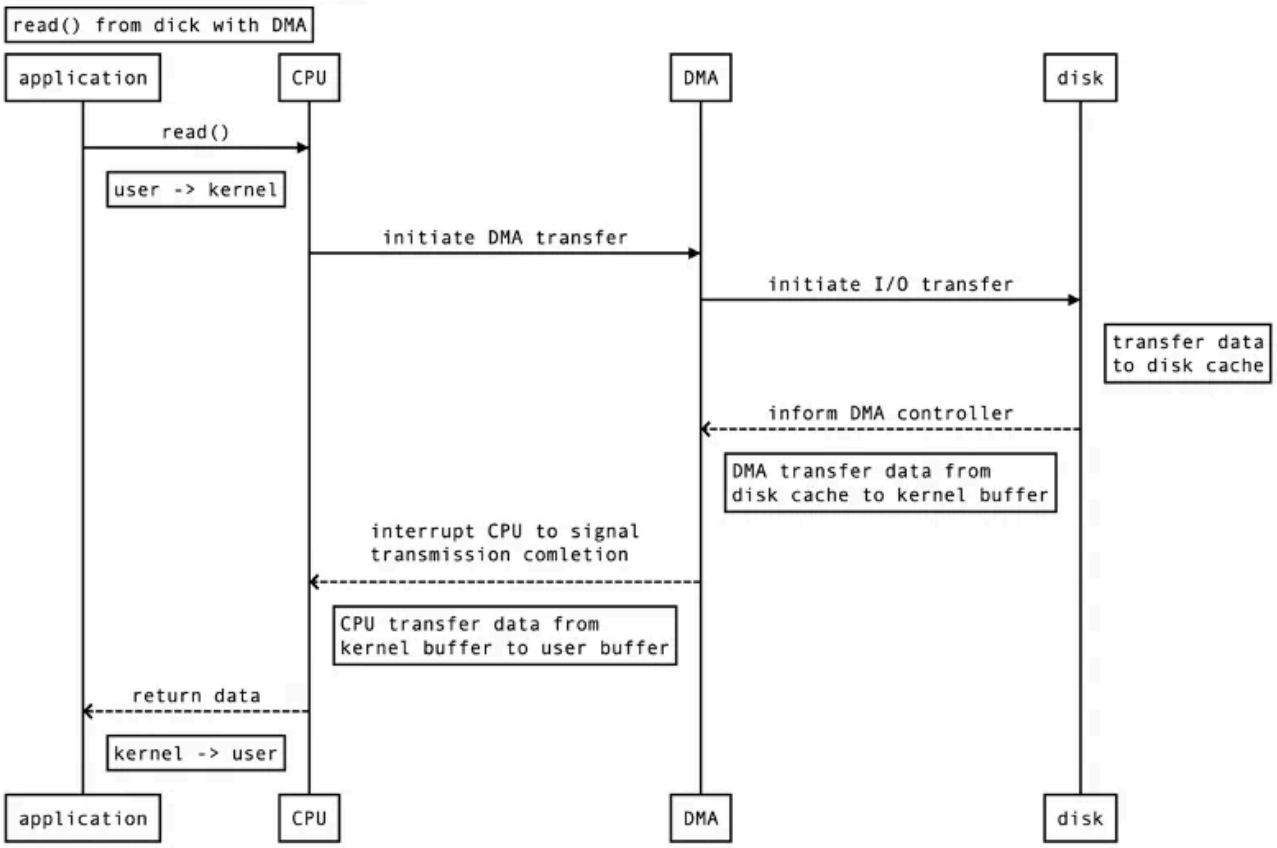
Kafka makes use of the underlying file system page cache, since the producer and the consumer access the segment files sequentially with the consumer lagging behind by a small time, normal operation system caching heuristics are very effective.

Kafka also optimizes networks access for consumers by utilising the sendfile API of Linux and Unix operating systems that can directly transfer bytes from a file channel to a socket channel. Kafka uses **zero-copy transfer**

ZERO-COPY-TRANSFER

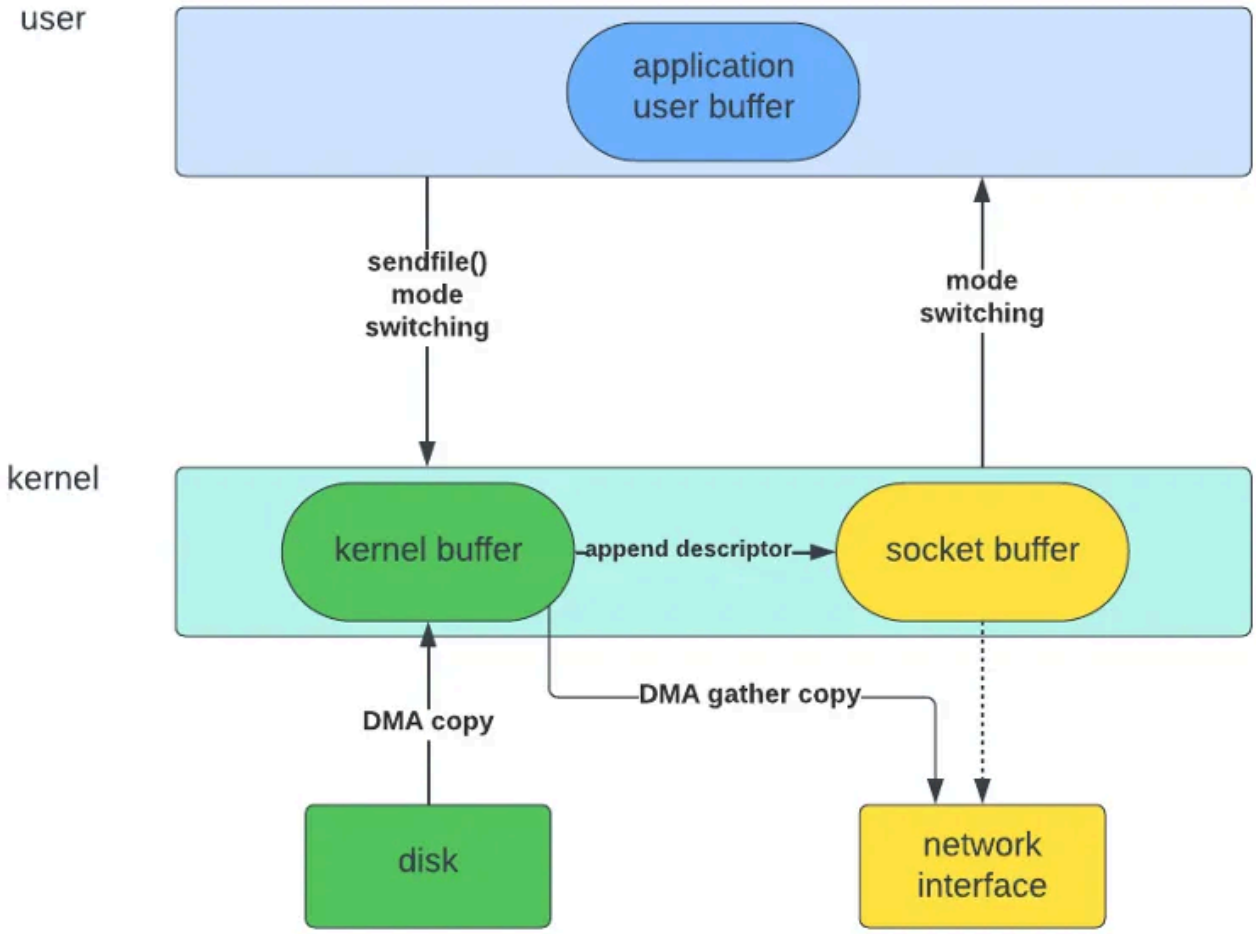
1. From disk data is copy in kernel read buffer . The first copy is performed by the direct memory access (DMA) engine, which reads file contents from the disk and stores them into a kernel address space buffer. As Before this we were in user mode and to do this one context switch required.
2. From kernel read buffer it copy to application read buffer(context switch , switch to user mode).
3. From application buffer it will copy to kernel socket buffer and put the data into a kernel address space buffer again(switched to kernel mode).

4. From socket buffer it will copy to network buffer



The objective of Zero-copy is simple, we want to eliminate or reduce the unnecessary data copy by CPU between kernel buffer and user buffer, so as to reduce the mode switching as well, and

the performance improvement could be then realized.



In Linux kernels 2.4 and later, the socket buffer descriptor was modified to accommodate this requirement.

1. The `transferTo()` method causes the file contents to be copied into a kernel buffer by the DMA engine.
2. No data is copied into the socket buffer. Instead, only descriptors with information about the location and length of the data are appended to the socket buffer. The DMA engine passes data directly from the kernel buffer to the Network buffer, thus eliminating the remaining final CPU copy.
   As application call sendfile(), DMA controller copies data from disk to kernel buffer by DMA scatter, meaning you don’t need a continuous memory space for data storage. Then CPU appends the file descriptor to the socket buffer and DMA controller generates corresponding header and tailer of the network packet. Lastly, DMA controller follows the description on socket buffer, copies data and then makes packets from kernel buffer to network interface for network transmission.

The broker also doesnt maintain information about how much each consumer has consumed.This is mantained by the consumer itself.
Kakfa uses a time-based SLA (Service Level Agreement) for its retention policy. A message is

automatically deleted if it has been retained in the broker longer than a certain period, typically 7 days.

Kafka is **persistent** in the sense that it stores messages on disk (not just in memory), ensuring durability even if the broker crashes or restarts. However, persistence does not mean that data is stored **forever**. Kafka is designed to balance durability with efficient storage management. By default, Kafka uses retention policies to automatically clean up old data that is no longer needed.

If you want to store data forever, you can configure Kafka to do so by:

- Setting the retention time to a very large value (e.g., `retention.ms=-1`).
- Using a **log compaction** feature, which retains the latest value for each key in a topic indefinitely.

Consumer Group

One of the most important concepts related to consumers is consumer group. I wanted to talk about it in detail because the consumer group logic must be fully established in order to write an efficient consumer application.

A topic generally consists of more than one partition. These partitions are members of parallelism for Kafka consumers. Consumers become part of a consumer group by consuming a partition. There may be more than one consumer group consuming a topic. Each consumer group has a unique id. This id is assigned by users.

How Producers and Consumers interact?

Each producer can publish a message to a partition randomly or by using a partitioning key and a partitioning function.
Kafka has consumer groups , each consumer group has one or more consumer that consumes from a set of subscribed topics within a group,
each message is delivered to only one set of consumers within the group.
The consumers within the same group can be on different processes or on different machines.We need to divide the messages evenly among the consumers.

All messages from one partition are consumed only by a single consumer in a single consumer group. If we had allowed multiple consumers to consumer from a single partition then we would have to have coordination between the consumers and this would lead to some overhead.
Now we only need to co-ordinate for re-balancing the load which is an infrequent event.
The second decision that we made is to not have a central “master” node, but instead let consumers coordinate among themselves in a decentralised fashion.

Kafka uses Zookeeper for the following tasks: (1) detecting the addition and the removal of brokers and consumers, (2) triggering a re balance process in each consumer when the above events happen, and (3) maintaining the consumption relationship and keeping track of the consumed offset of each partition.
When each broker or consumer starts up, it stores its information in a broker or consumer registry in Zookeeper. The broker registry contains the broker's host name and port, and the set of topics and partitions stored on it. The consumer registry includes the consumer group to which a consumer belongs and the set of topics that it subscribes to.

Each consumer group is associated with an ownership registry and an offset registry in Zookeeper. The offset registry stores for each subscribed partition, the offset of the last consumed message in the partition.

The paths created in Zookeeper are ephemeral for the broker registry, the consumer registry and the ownership registry, and persistent for the offset registry. If a broker fails, all partitions on it are automatically removed from the broker registry. The failure of a consumer causes it to lose its entry in the consumer registry and all partitions that it owns in the ownership registry. Each consumer registers a Zookeeper watcher on both the broker registry and the consumer registry, and will be notified whenever a change in the broker set or the consumer group occurs.

During the initial startup of a consumer or when the consumer is notified about a broker/consumer change through the watcher, the consumer initiates a rebalance process to determine the new subset of partitions that it should consume from.
When there are multiple consumers within a group, each of them will be notified of a broker or a consumer change. However, the notification may come at slightly different times at the consumers. So, it is possible that one consumer tries to take ownership of a partition still owned by another consumer. When this happens, the first consumer simply releases all the partitions that it currently owns, waits a bit and retries the rebalance process. In practice, the rebalance process often stabilizes after only a few retries.
When a new consumer group is created, no offsets are available in the offset registry. In this case, the consumers will begin with either the smallest or the largest offset (depending on a configuration) available on each subscribed partition, using an API that we provide on the brokers

Delivery Guarantees

Kafka delivers at-least once delivery.Exactly once delivery typically requires two-phase commits. A message is delivered exactly once to each consumer group. If a consumer crashes , the new consumer which takes over the partitions of the failed consumers may get some duplicate messages. So the application should add its own duplication logic.

Kafka guarantees that messages from a single partition are delivered to a consumer in order. However, there is no guarantee on the ordering of messages coming from different partitions.

Partitions are sequential within themselves. However, there is no order between the partitions. For example, there are 2 partitions: partition0 and partition1. First message0 was written to partition0, then message1 was written to partition1, and then message2 was written to partition0. In this scenario, message0 is always read before message2. However, Kafka does not guarantee that message1 will be read before or after message0. If you do not care in which order the messages you produce are read, you can produce the messages with different keys. However, if it is important for you to have 2 or more messages in their own order, you should produce these messages with the same key. Because Kafka writes the messages produced with the same key to the same partition, so that the order is observed. However, if the order is not important to you, sending all messages with the same key will have a disadvantage. Because Kafka will write all of these to a single partition and the load will not be distributed.

If a broker goes down, any message stored on it not yet consumed becomes unavailable. If the storage system on a broker is permanently damaged, any unconsumed message is lost forever. In the future, we plan to add built-in replication in Kafka to redundantly store each message on multiple brokers.

Kafka Replication

In Kafka, replication happens at the partition granularity i.e. copies of the partition are maintained at multiple broker instances using the partition's write-ahead log.
Replication factor defines the number of copies of the partition that needs to be kept.
Leader for a partition: For every partition, there is a replica that is designated as the **leader.** The Leader is responsible for sending as well as receiving data for that partition. All the other replicas are called the **in-sync replicas** (or followers) of the partition.

In-sync replicas are the subset of all the replicas for a partition having same messages as the leader.
When a broker goes down if it contains a leader of a partition, one of the in-sync replicas will be selected to become the new leader.
Producers can choose to receive acknowledgements for the data writes to the partition using the "acks" setting.

Acks	Description	Notes
0	don't wait for any acknowledgment	Possible data loss
1	wait for leader's acknowledgement	Parital data loss
all	wait for leader's and all the in-sync replicas' acknowledgement	No data loss

The value of a*cks* varies from application to application. For an application where high durability needs (example in case of transaction data), a*cks* = *all* is recommended whereas in cases where lower latency is more important *acks = 0* is used (example — user's location data).

How does Replication Work?

In Kafka, the following 2 conditions need to be met for a node to be considered alive.

- The node must maintain its session with zookeeper
- If the node is a follower, it must not be "*far behind*" the leader

The answer is fairly simple — leader maintains a list of its followers and tracks their status. If a follower dies or is not able to replicate and falls behind the leader, it gets removed from the in-sync replica list.

- *replica.lag.max.messages*
  _This parameter decides the allowed difference between replica's offset and leader's offset. if the difference becomes more than (replica.lag.max.messages-_1) then that replica is considered to be lagging behind and is removed from the in-sync replica list.
  If the value of replica.lag.max.messages is n, it means that as long as the follower is behind the leader by not more than n-1 messages, it won't be removed from the in-sync replica list.
- _replica.lag.time.max.ms
  _This parameter defines the maximum time interval within which every follower must request the leader for its log. If for some reason a replica is unable to do so, it will be removed from the in-sync replica list.

Only the replicas that are part of in-sync replica list are eligible for becoming the leader and the list of in-sync replicas is persisted to zookeeper whenever any changes are made to it. Also, Kafka's guarantee of no data loss is applicable only if there exists an in-sync replica. In case of no such replica, this guarantee is not applicable.

There are two common strategies for keeping replicas in sync, primary-backup replication and quorum-based replication. In both cases, one replica is designated as the leader and the rest of the replicas are called followers. All write requests go through the leader and the leader propagates the writes to the follower.

In primary-backup replication, the leader waits until the write completes on every replica in the group before acknowledging the client. If one of the replicas is down, the leader drops it from the current group and continues to write to the remaining replicas. A failed replica is allowed to rejoin the group if it comes back and catches up with the leader. With f replicas, primary-backup replication can tolerate f-1 failures.
In the quorum-based approach, the leader waits until a write completes on a majority of the

replicas. The size of the replica group doesn’t change even when some replicas are down. If there are 2f+1 replicas, quorum-based replication can tolerate f replica failures. If the leader fails, it needs at least f+1 replicas to elect a new leader.

Reasons for Replica’s being behind?

A follower replica might be lagging behind the leader for a number of reasons.

1. **Slow Replica**: It is possible for a replica to be unable to cope up with the speed at which leader is getting new messages — the rate at which the leader is getting messages is more than the rate at which the replica is copying messages causing IO bottleneck.
2. **Stuck Replica**: If a replica has stopped requesting the leader for the new messages for reasons like a dead replica or the replica is blocked due to GC (Garbage collector).