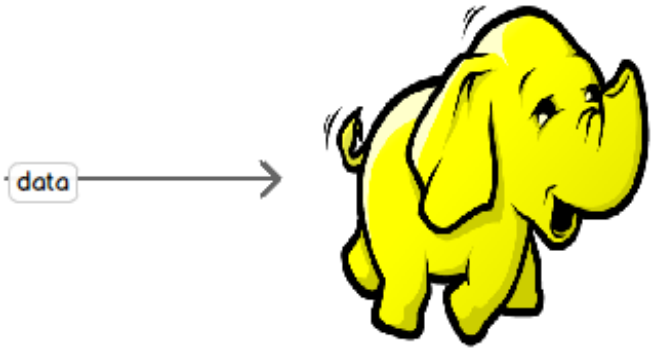
Kafka in a Nutshell

sookocheff.com/post/kafka/kafka-in-a-nutshell

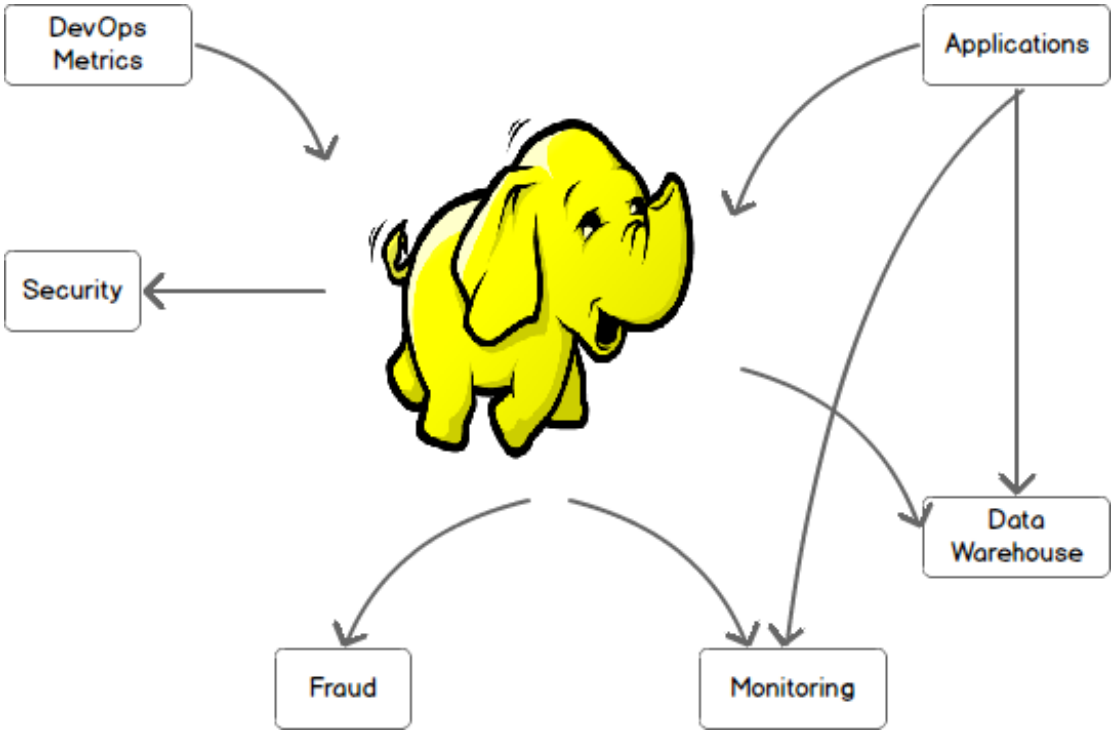
Kafka is a messaging system. That's it. So why all the hype? In reality messaging is a hugely important piece of infrastructure for moving data between systems. To see why, let's look at a data pipeline without a messaging system.

This system starts with Hadoop for storage and data processing. Hadoop isn't very useful without data so the first stage in using Hadoop is getting data in.

So far, not a big deal. Unfortunately, in the real world data exists on many systems in parallel, all of which need to interact with Hadoop and with each other. The situation quickly becomes more complex, ending with a system where multiple data systems are talking to one another over many channels. Each of these channels requires their own custom protocols and communication methods and moving data between these systems becomes a full-time job for a team of developers.

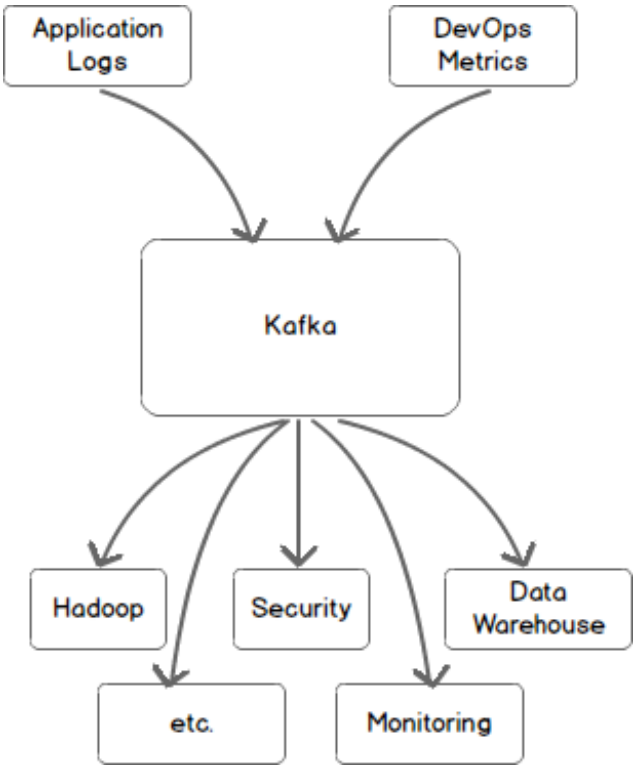


Bringing Data in to Hadoop



Moving Data Between Systems

Let's look at this picture again, using Kafka as a central messaging bus. All incoming data is first placed in Kafka and all outgoing data is read from Kafka. Kafka centralizes communication between producers of data and consumers of that data.



Moving Data Between Systems

What is Kafka?

Kafka is a distributed messaging system providing fast, highly scalable and redundant messaging through a pub-sub model. Kafka's distributed design gives it several advantages. First, Kafka allows a large number of permanent or ad-hoc consumers. Second, Kafka is highly available and resilient to node failures and supports automatic recovery. In real world data systems, these characteristics make Kafka an ideal fit for communication and integration between components of large scale data systems.

Kafka Terminology

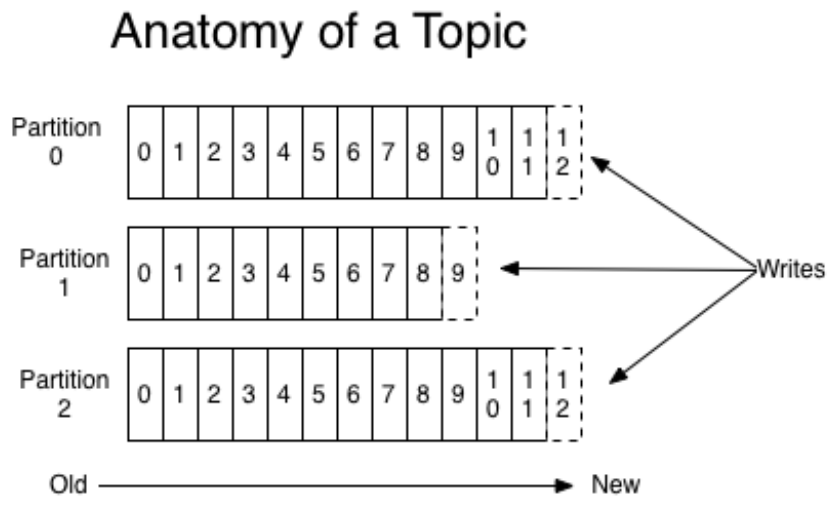
The basic architecture of Kafka is organized around a few key terms: topics, producers, consumers, and brokers.

All Kafka messages are organized into *topics*. If you wish to send a message you send it to a specific topic and if you wish to read a message you read it from a specific topic. A *consumer* pulls messages off of a Kafka topic while *producers* push messages into a Kafka topic. Lastly, Kafka, as a distributed system, runs in a cluster. Each node in the cluster is called a Kafka *broker*.

Anatomy of a Kafka Topic

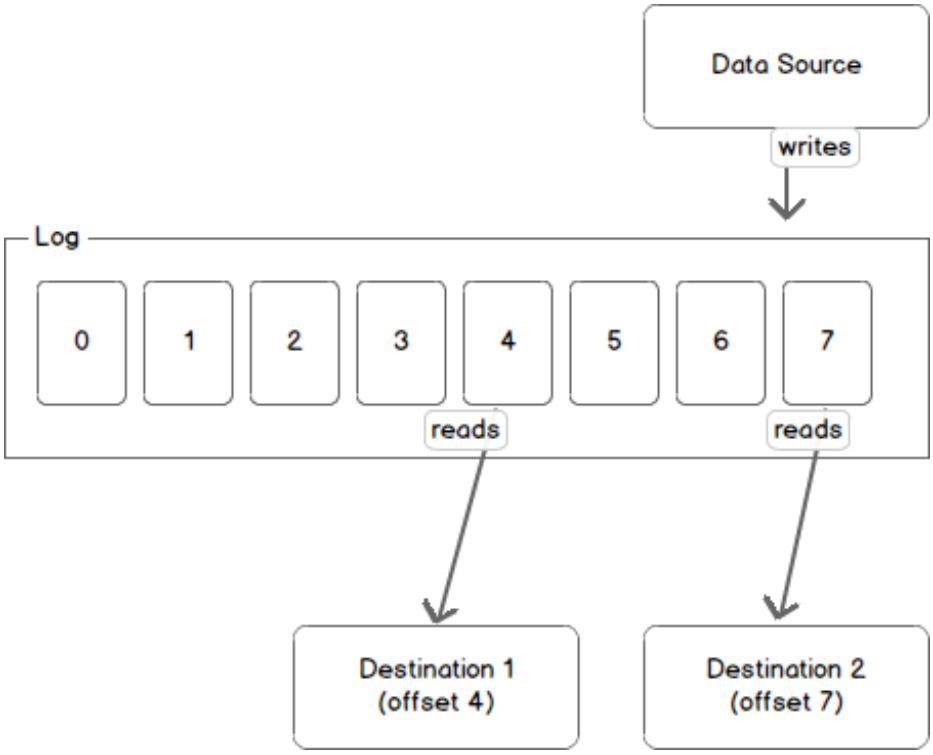
Kafka topics are divided into a number of *partitions*. Partitions allow you to parallelize a topic by splitting the data in a particular topic across multiple brokers — each partition can be placed on a separate machine to allow for multiple consumers to read from a topic in parallel. Consumers can also be parallelized so that multiple consumers can read from multiple partitions in a topic allowing for very high message processing throughput.

Each message within a partition has an identifier called its *offset*. The offset the ordering of messages as an immutable sequence. Kafka maintains this message ordering for you. Consumers can read messages starting from a specific offset and are allowed to read from any offset point they choose, allowing consumers to join the cluster at any point in time they see fit. Given these constraints, each specific message in a Kafka cluster can be uniquely identified by a tuple consisting of the message's topic, partition, and offset within the partition.



Log Anatomy

Another way to view a partition is as a log. A data source writes messages to the log and one or more consumers reads from the log at the point in time they choose. In the diagram below a data source is writing to the log and consumers A and B are reading from the log at different offsets.

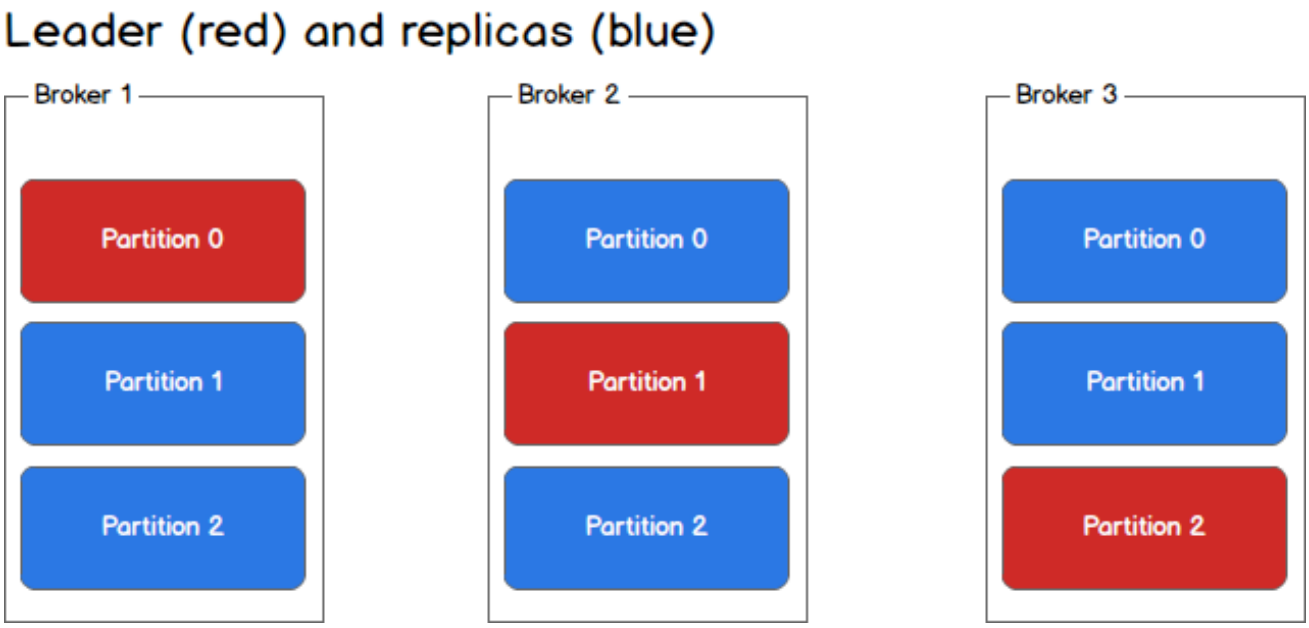


Data Log

Kafka retains messages for a configurable period of time and it is up to the consumers to adjust their behaviour accordingly. For instance, if Kafka is configured to keep messages for a day and a consumer is down for a period of longer than a day, the consumer will lose messages. However, if the consumer is down for an hour it can begin to read messages again starting from its last known offset. From the point of view of Kafka, it keeps no state on what the consumers are reading from a topic.

Partitions and Brokers

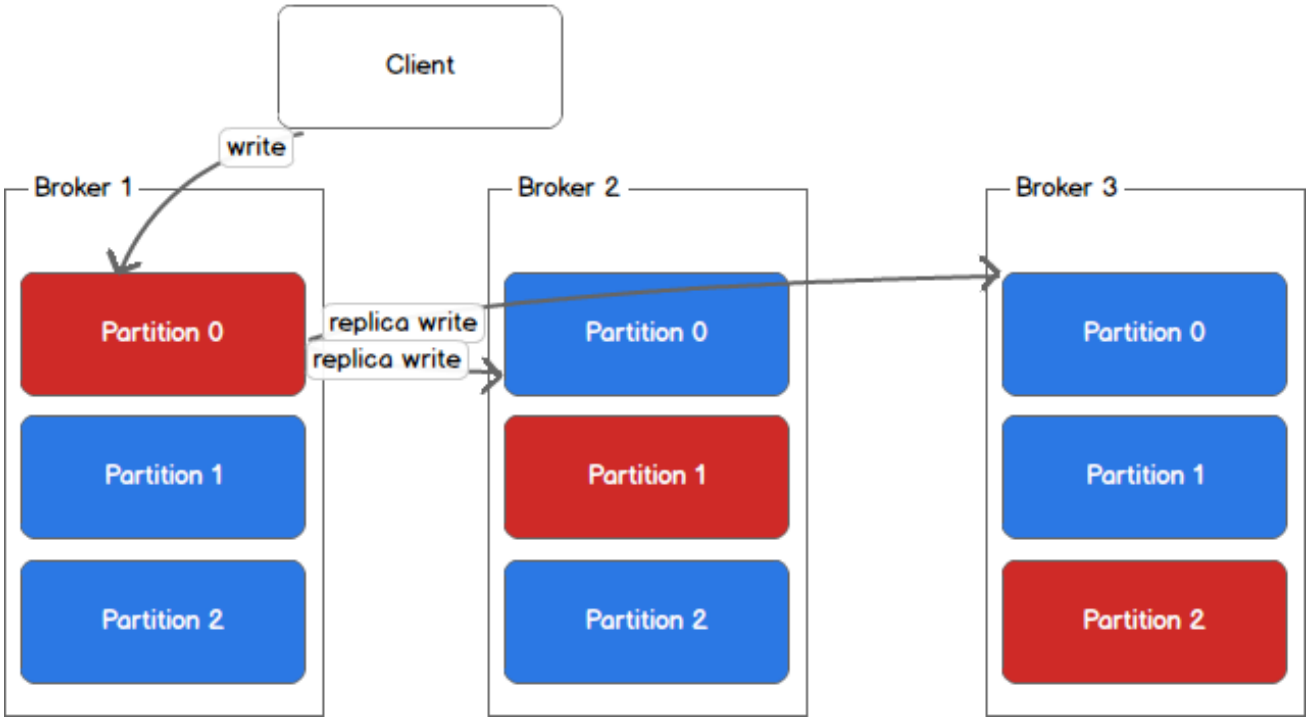
Each broker holds a number of partitions and each of these partitions can be either a leader or a replica for a topic. All writes and reads to a topic go through the leader and the leader coordinates updating replicas with new data. If a leader fails, a replica takes over as the new leader.



Partitions and Brokers

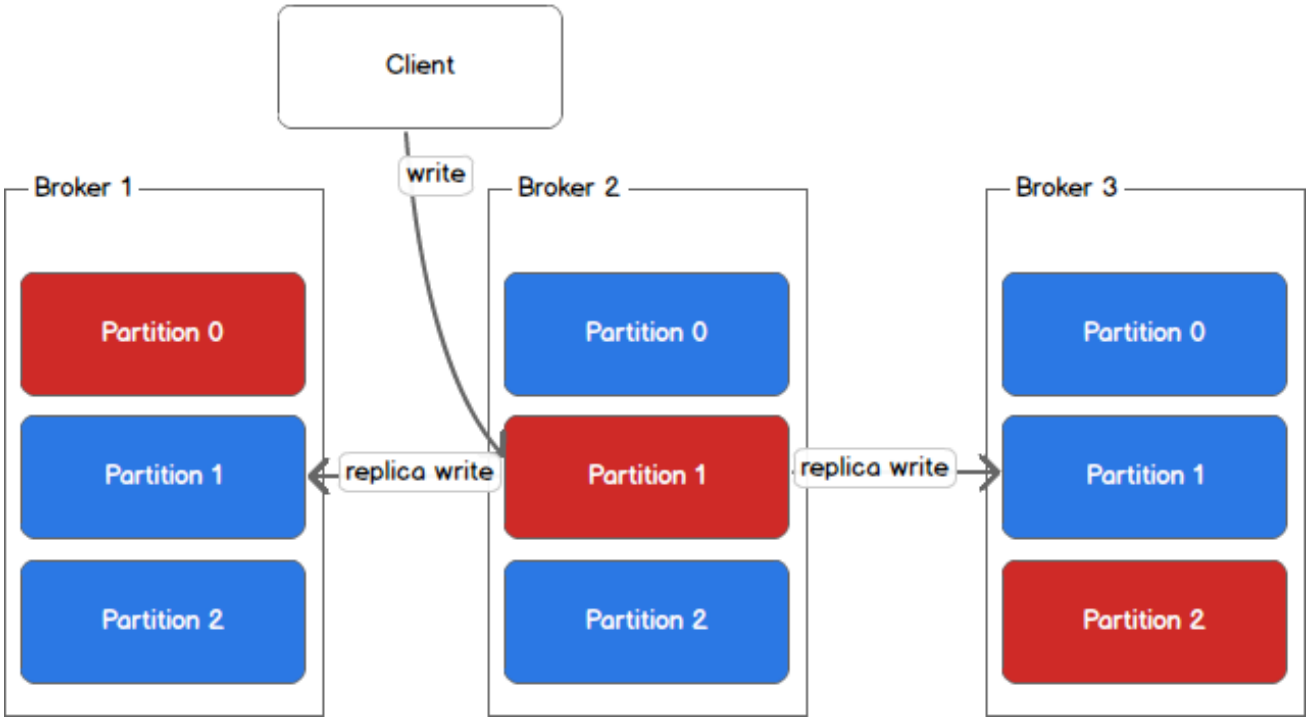
Producers

Producers write to a single leader, this provides a means of load balancing production so that each write can be serviced by a separate broker and machine. In the first image, the producer is writing to partition 0 of the topic and partition 0 replicates that write to the available replicas.



Producer writing to partition.

In the second image, the producer is writing to partition 1 of the topic and partition 1 replicates that write to the available replicas.



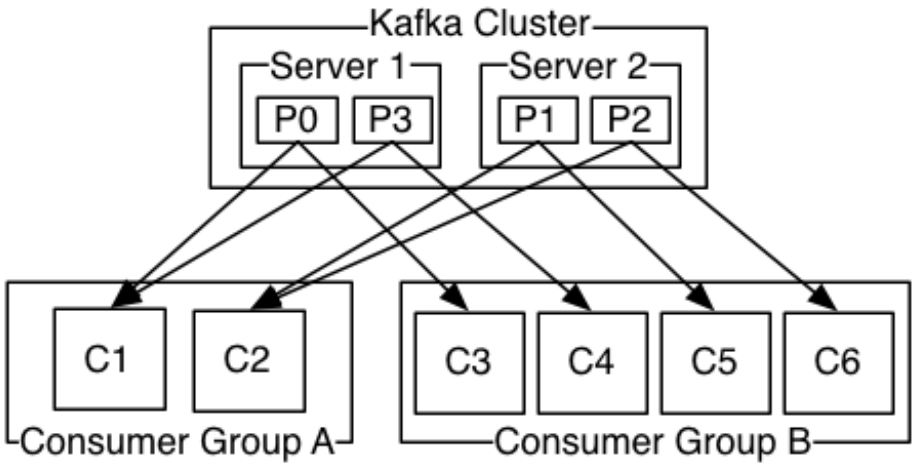
Producer writing to second partition.

Since each machine is responsible for each write, throughput of the system as a whole is increased.

Consumers and Consumer Groups

Consumers read from any single partition, allowing you to scale throughput of message consumption in a similar fashion to message production. Consumers can also be organized into consumer groups for a given topic — each consumer within the group reads from a unique partition and the group as a whole consumes all messages from the entire topic. If you have more consumers than partitions then some consumers will be idle because they have no partitions to read from. If you have more partitions than consumers then consumers will receive messages from multiple partitions. If you have equal numbers of consumers and partitions, each consumer reads messages in order from exactly one partition.

The following picture from the Kafka documentation describes the situation with multiple partitions of a single topic. Server 1 holds partitions 0 and 3 and server 2 holds partitions 1 and 2. We have two consumer groups, A and B. A is made up of two consumers and B is made up of four consumers. Consumer Group A has two consumers of four partitions — each consumer reads from two partitions. Consumer Group B, on the other hand, has the same number of consumers as partitions and each consumer reads from exactly one partition.



Consumers and Consumer Groups

Consistency and Availability

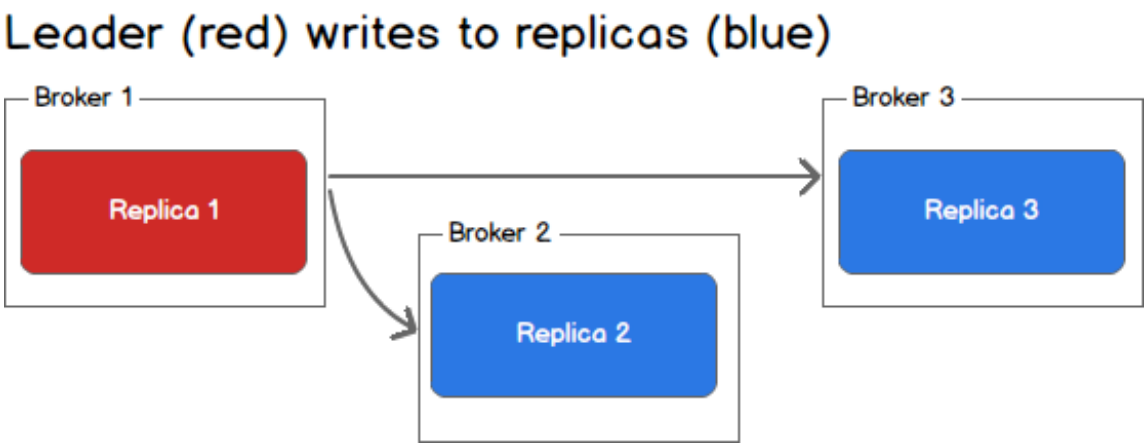
Before beginning the discussion on consistency and availability, keep in mind that these guarantees hold *as long as you are producing to one partition and consuming from one partition.* All guarantees are off if you are reading from the same partition using two consumers or writing to the same partition using two producers.

Kafka makes the following guarantees about data consistency and availability: (1) Messages sent to a topic partition will be appended to the commit log in the order they are sent, (2) a single consumer instance will see messages in the order they appear in the log, (3) a message is 'committed' when all in sync replicas have applied it to their log, and (4) any committed message will not be lost, as long as at least one in sync replica is alive.

The first and second guarantee ensure that message ordering is preserved for each partition. Note that message ordering for the entire topic is not guaranteed. The third and fourth guarantee ensure that committed messages can be retrieved. In Kafka, the partition that is elected the leader is responsible for syncing any messages received to replicas. Once a replica has acknowledged the message, that replica is considered to be in sync. To understand this further, lets take a closer look at what happens during a write.

Handling Writes

When communicating with a Kafka cluster, all messages are sent to the partition's leader. The leader is responsible for writing the message to its own in sync replica and, once that message has been committed, is responsible for propagating the message to additional replicas on different brokers. Each replica acknowledges that they have received the message and can now be called in sync.

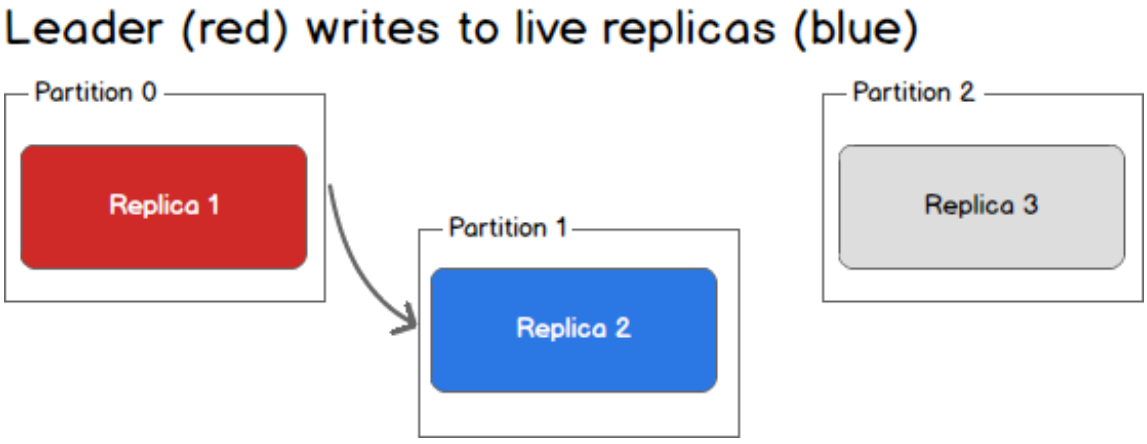


Leader Writes to Replicas

When every broker in the cluster is available, consumers and producers can happily read and write from the leading partition of a topic without issue. Unfortunately, either leaders or replicas may fail and we need to handle each of these situations.

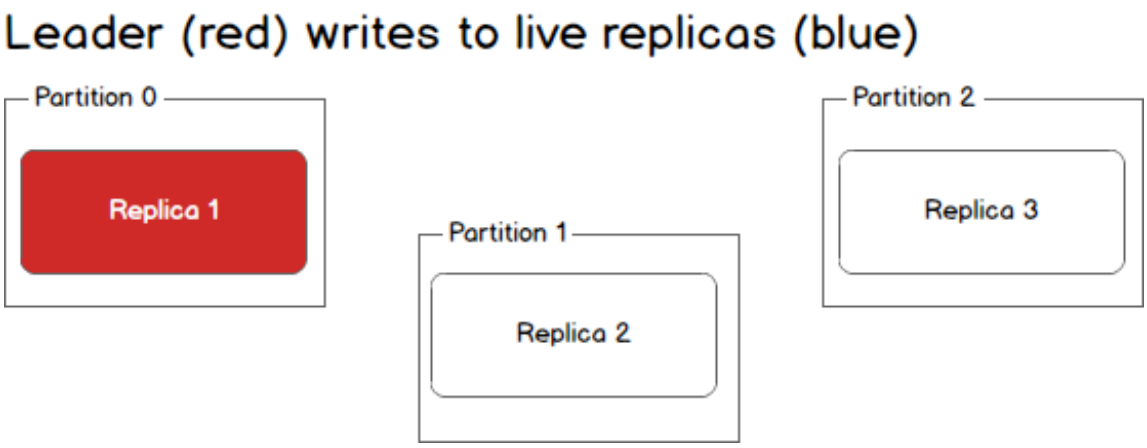
Handling Failure

What happens when a replica fails? Writes will no longer reach the failed replica and it will no longer receive messages, falling further and further out of sync with the leader. In the image below, Replica 3 is no longer receiving messages from the leader.



First Replica Fails

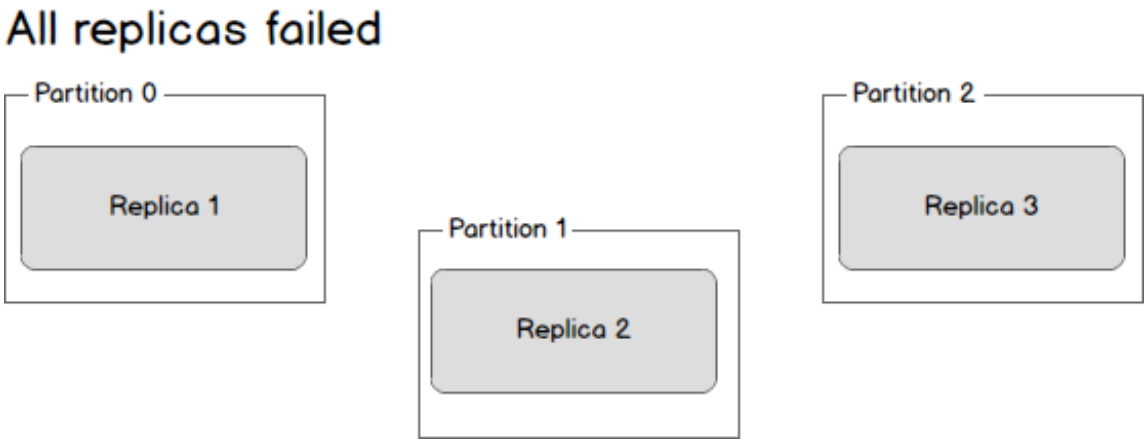
What happens when a second replica fails? The second replica will also no longer receive messages and it too becomes out of sync with the leader.



Second Replica Fails

At this point in time, only the leader is in sync. In Kafka terminology we still have one in sync replica even though that replica happens to be the leader for this partition.

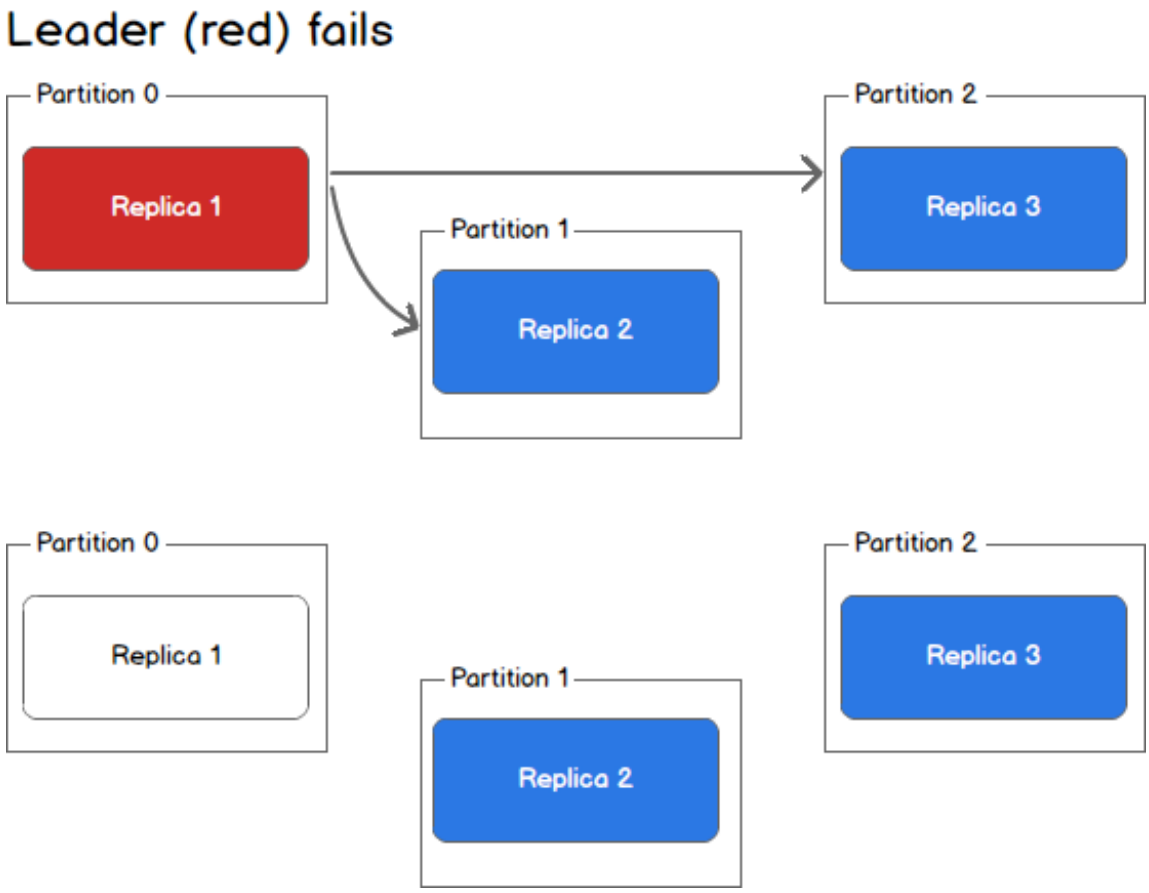
What happens if the leader dies? We are left with three dead replicas.



Third Replica Fails

Replica one is actually still in sync — it cannot receive any new data but it is in sync with everything that was possible to receive. Replica two is missing some data, and replica three (the first to go down) is missing even more data. Given this state, there are two possible solutions. The first, and simplest, scenario is to wait until the leader is back up before continuing. Once the leader is back up it will begin receiving and writing messages and as the replicas are brought back online they will be made in sync with the leader. The second scenario is to elect the second broker to come back up as the new leader. This broker will be out of sync with the existing leader and all data written between the time where this broker went down and when it was elected the new leader will be lost. As additional brokers come back up, they will see that they have committed messages that do not exist on the new leader and drop those messages. By electing a new leader as soon as possible messages may be dropped but we will minimized downtime as any new machine can be leader.

Taking a step back, we can view a scenario where the leader goes down while in sync replicas still exist.



Leader Fails

In this case, the Kafka controller will detect the loss of the leader and elect a new leader from the pool of in sync replicas. This may take a few seconds and result in `LeaderNotAvailable` errors from the client. However, no data loss will occur as long as producers and consumers handle this possibility and retry appropriately.

Consistency as a Kafka Client

Kafka clients come in two flavours: producer and consumer. Each of these can be configured to different levels of consistency.

For a producer we have three choices. On each message we can (1) wait for all in sync replicas to acknowledge the message, (2) wait for only the leader to acknowledge the message, or (3) do not wait for acknowledgement. Each of these methods have their merits and drawbacks and it is up to the system implementer to decide on the appropriate strategy for their system based on factors like consistency and throughput.

On the consumer side, we can only ever read committed messages (i.e., those that have been written to all in sync replicas). Given that, we have three methods of providing consistency as a consumer: (1) receive each message at most once, (2) receive each message at least once, or (3) receive each message exactly once. Each of these scenarios deserves a discussion of its own.

For at most once message delivery, the consumer reads data from a partition, commits the offset that it has read, and then processes the message. If the consumer crashes between committing the offset and processing the message it will restart from the next offset without ever having processed the message. This would lead to potentially undesirable message loss.

A better alternative is at least once message delivery. For at least once delivery, the consumer reads data from a partition, processes the message, and then commits the offset of the message it has processed. In this case, the consumer could crash between processing the message and committing the offset and when the consumer restarts it will process the message again. This leads to duplicate messages in downstream systems but no data loss.

Exactly once delivery is guaranteed by having the consumer process a message and commit the output of the message along with the offset to a transactional system. If the consumer crashes it can re-read the last transaction committed and resume processing from there. This leads to no data loss and no data duplication. In practice however, exactly once delivery implies significantly decreasing the throughput of the system as each message and offset is committed as a transaction.

In practice most Kafka consumer applications choose at least once delivery because it offers the best trade-off between throughput and correctness. It would be up to downstream systems to handle duplicate messages in their own way.

Conclusion

Kafka is quickly becoming the backbone of many organization's data pipelines — and with good reason. By using Kafka as a message bus we achieve a high level of parallelism and decoupling between data producers and data consumers, making our architecture more flexible and adaptable to change. This article provides a birds eye view of Kafka architecture. From here, consult the Kafka documentation. Enjoy learning Kafka and putting this tool to more use!

kafka

See also

- ← Previous Post

- Next Post →