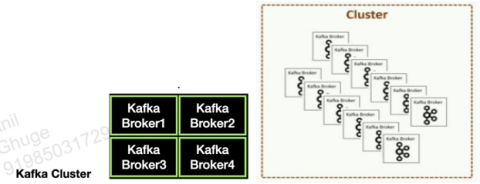
Kafka Cluster Architecture

Kafka Cluster

Kafka brokers are often configured to form a Cluster. A Cluster is nothing but a group of brokers that work together to share the workload, and that's how Apache Kafka becomes a distributed and scalable system.

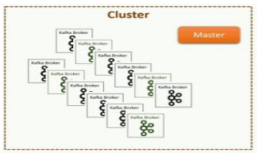


We can start with a single broker in our development environment and deploy a cluster of three or five brokers in our production environment.

Later, as the workload grows, we can increase the number of brokers in the cluster. It can grow up to hundreds of brokers in a single cluster. However, the idea of clustering brings out two critical questions.

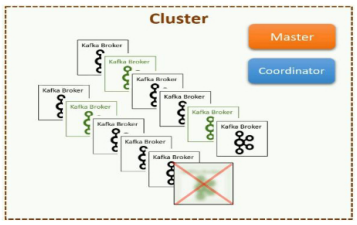
1. **Who manages cluster membership?**
2. **Who performs the routine administrative tasks in the cluster?**

In a typical distributed system, there is a master node that maintains a list of active cluster members. The master always knows the state of the other members.



Suppose a broker is active and it is taking care of some responsibilities in the cluster. Suddenly the broker leaves the cluster or dies for some reason. Now at this stage, who will perform those responsibilities? We need someone to reassign that work to an active active broker to ensure that the cluster continues to function.

Apache Kafka



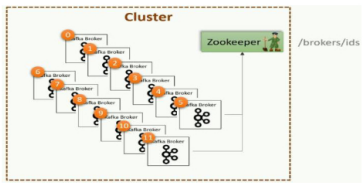
Zookeeper

Kafka Architecture is a master-less cluster Architecture. It does not follow a master-slave architecture. It uses Apache Zookeeper to maintain the list of active brokers.

Every Kafka broker has a unique_id that we define in the broker configuration file. We also specify the Zookeeper connection details in the broker configuration file.

When the broker starts, it connects to the Zookeeper and creates an ephemeral node (using broker_id) to represent an active broker session.

The ephemeral node remains intact as long as the broker session with the Zookeeper is active. When a broker loses connectivity to the Zookeeper for some reason, the Zookeeper automatically removes that ephemeral node. So, the list of active brokers in the cluster is maintained as the list of ephemeral nodes under the /brokers/ids path in the Zookeeper.



To see the brokers list, start a zookeeper shell and to see what we have in the zookeeper database type "ls /" as shown below

- To see the brokers type "ls /brokers"

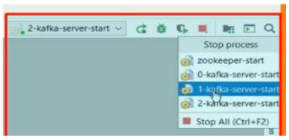
```
ls /brokers
[ids, seqid, topics]
```

- To see the broker IDs type "ls /brokers/ids"

```
ls /brokers/ids
[0, 1, 2]
```

We can see the 3 brokers.

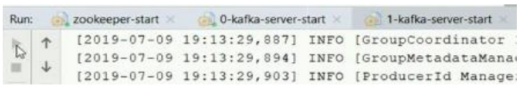
- Lets stop the brokerId-1 and recheck the brokers/ids




```
ls /brokers/ids
[0, 2]
```

Now brokerId-1 gone from the zookeeper.

- Again lets start the broker-1 again and see the brokers/ids




```
ls /brokers/ids
[0, 1, 2]
```

Again the broker-1 registered with the zookeeper. This is how the kafka will maintain the list of brokers in a cluster.

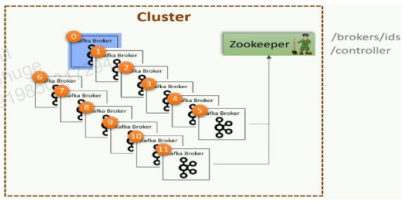
List of active brokers in the cluster is maintained as the list of ephemeral nodes, under the /brokers/ids path in the zookeeper.

Controller

Kafka is a master-less cluster, and the list of active brokers is maintained in the Zookeeper. However, we still need someone to perform the routine administrative activities such as monitoring the list of active brokers and reassigning the work when an active broker leaves the cluster.

All those activities are performed by a Controller in the Kafka cluster. The controller is not a master.

It is simply a broker that is elected as a controller/leader to take up some extra responsibilities.



That means, the controller also acts as a regular broker. So, if we have a single node cluster, it serves as a controller as well as a broker. ***However, there is only one controller in a Kafka cluster*** at any point in

time.

The controller is responsible for monitoring the list of active brokers in the Zookeeper. When the controller notices that a broker left the cluster, it knows that it is time to reassign some work to the other brokers.

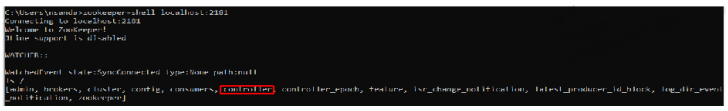
The controller election is straightforward. The first broker that starts in the cluster becomes the controller by creating an ephemeral (/controller) node in Zookeeper.

When other brokers start, they also try to create this node, but they receive an exception as “node already exists” which means that the controller is already elected.

In that case, they start watching the /controller node in the Zookeeper to disappear. When the current controller dies, the ephemeral node disappears. Then every broker again tries to create the /controller node in the Zookeeper, but only one succeeds, and others get an exception once again.

This process ensures that there is always a controller in the cluster and there exists only one controller.

Lets see this /controller node in zookeeper by typing the ls /controller in zookeeper shell as shown below:

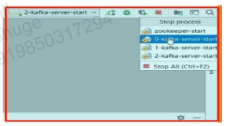


This will shows the controller node as already created. That means out of all 3 brokers one of the broker is elected as a controller. To see which broker elected as a controller we have to query the controller node for the its information as “get /controller”

```
get /controller
{"version":2,"brokerid":0,"timestamp":"1683873264914","kraftControllerEpoch":-1}
```

From the above screen shot it is showing the brokerid:0 is elected as the controller as we started the it first to registered with the zookeeper.

Now lets down the broker-0 and will query the controller node to see which remianing broker will be elected as a controller.

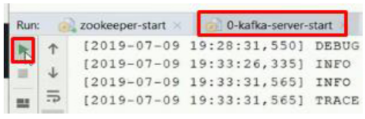


Lets query it again:

```
get /controller
{"version":2,"brokerid":2,"timestamp":"1683875138441","kraftControllerEpoch":-1}
```

Now beoker-2 was elected as a controller.

Lets bring the broker-0 back up:



Again lets re-check the controller node

```
get /controller
{"version":2,"brokerid":2,"timestamp":"1683875138441","kraftControllerEpoch":-1}
```

Though the broker-0 will come back now, still broker-2 was continuing as the master. Because the controller ship of the broker-0 has gone when it broken down, so now even if it back again, then it doenst become a controller. Because in the absence of the broker-0 , broker-2 was already elected as controller.

Apache Kafka

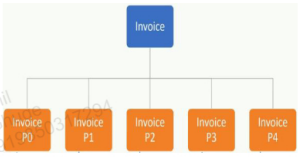
Partition Allocation and Fault Tolerance

So far we have discussed about the

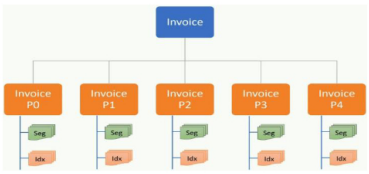
1. Partitions - Log files

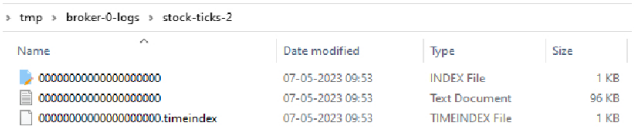
2. Cluster Formation

If we look at the Kafka topic organization, it is broken into independent partitions.

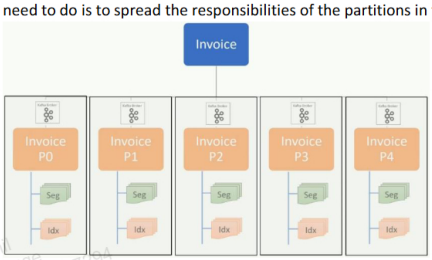


each partition is self contained. This means, all the information about the partition such as segment files, indexes are stored inside the same directory.





This stricture is allowed to distribute the work among the Kafka brokers in a cluster efficiently. All we need to do is to spread the responsibilities of the partitions in the Kafka cluster.

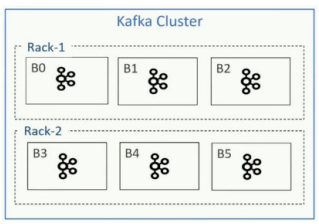


When we create a topic the responsibility to create, store and manage partitions is distributed among the available brokers in the cluster. That means every Kafka broker in the cluster is responsible for managing one or more partitions that are assigned to that broker.

Partition Allocation

Kafka cluster is a group of brokers. These brokers may be running on individual machines. In a large production cluster, we might have organized those machines in multiple racks. The below figure shows a six-node cluster that is designed using two racks. Now we have a simple question.

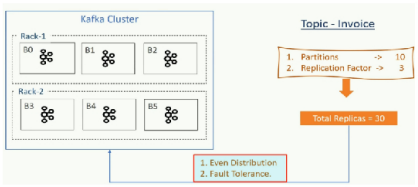
- How are the partitions allocated to brokers?
- Are there any rules?



Let's try to understand the partition assignment with an example. Suppose we have 6 brokers and we placed them into two different racks as shown in the above Figure.

we decide to create a topic with 10 partitions and a replication factor of 3. That means, in Kafka now have 30 replicas to allocate to 6 brokers.

Apache Kafka



Kafka tries to achieve two goals for this partition allocation.

1. Partitions are ***distributed evenly*** (as much as possible) to achieve workload balance.
2. Followers (duplicate copies) are placed on different machines to achieve ***fault tolerance***.

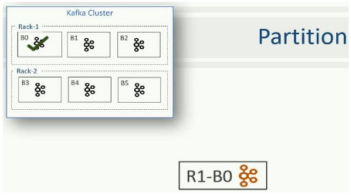
To distribute our (10x3=30) partitions, Kafka applies the following steps.

a) Make an ordered list of available brokers.

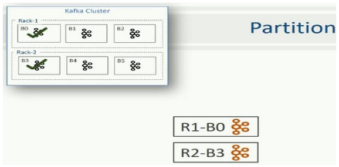
b) Assign Leaders and Followers to the list.

a) Make an ordered list of available brokers.

Kafka starts by making a list of available brokers. It begins with a randomly chosen broker in a rack and alternates with another broker in a different rack to create an ordered list of brokers. Referring to above Figure let's assume Kafka starts with broker B0.

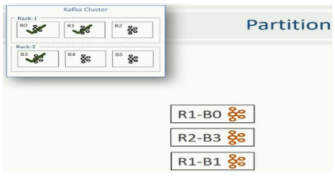


The next broker in the list must be from a different rack. So, Kafka selects B3.

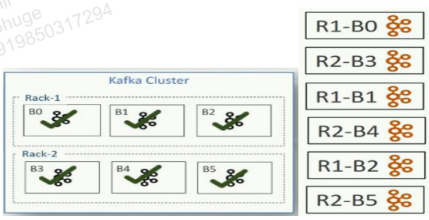


Apache Kafka

The next one again comes from the first rack, and selects the Broker-1



This process goes on as an alternating process for selecting another broker in a different rack. So we take Broker-4, then Borker-2 and finally Broker-5 as shown below:



This method results into an ordered list in which, each broker is followed by a broker from a different rack.

Now we have 30 partitions to allocate to these six brokers. Ideally, Kafka should place five partitions on each broker to achieve the first goal of evenly distributing the partitions.

However, we have another goal to achieve fault tolerance.

What is fault tolerance?

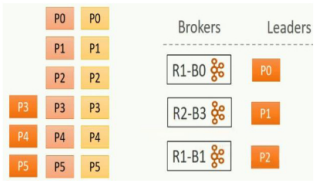
It is as simple as making sure that if one broker fails for some reason, we still have the copy of the on some other broker further making sure that if the entire rack fails we still have a copy on a different rack. We can achieve fault tolerance by placing duplicate copies on different machines.

b) Assign Leaders and Followers to the list

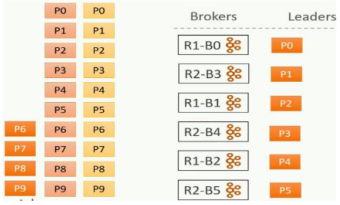
Once we have the ordered list of available brokers, assigning partitions is as simple as assign one to each broker using a round robin method. Kafka starts with the leader partitions and finishes allocating all leaders first. So lets take the leader of the partiton-0 and assign it to the broker-0.

The leader of partition-1 goes to Broker-3.

The leader of partition-2 lands on Broker-1.

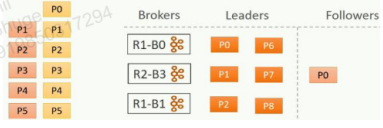


And so on.

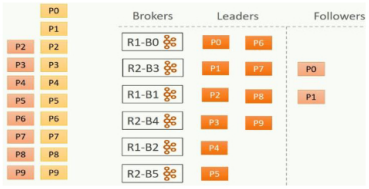


Once the leader partitions are created, it starts creating the first follower. ***The first follower allocation simply happens from the second broker in the list and follows the round robin approach.***

So we skip the first broker in the list and place the first follower of partition-02 in the broker-3.

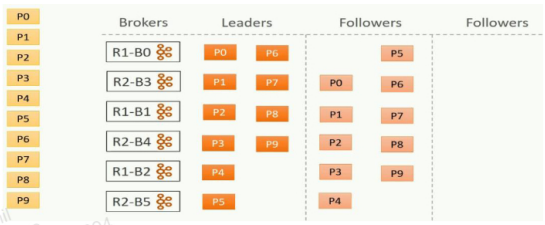


The first follower of the partition-1 goes to broker-1

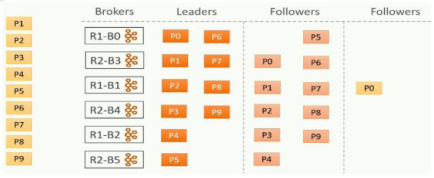


Similarly the first follower of the partiion-2 goes to Broker4 and so on.

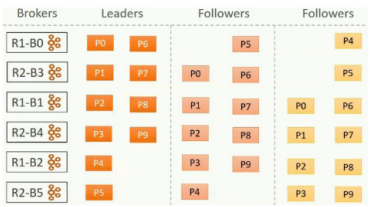
Finally it begins with the list of the second follower and maps them to the same brokers list by jumping one more broker from the previous stack.



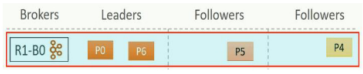
So the second follower of the partion-0 will goes to Broker-1



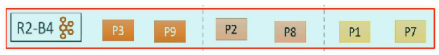
And similarly others are also creates in round robin fashion.



This is what happens when we create a topic. Leaders and followers are the topic are created across the cluster. If we look at the outcome of the above allocation, we couldn't achieved a perfectly even distribution. The broke-0 has got 4 partitions.



And broker-4 has got 6 partitions



However we made an ideal fault tolerance with the litle disparity. Lets check the fault tolerance level. The leader of the partition-4 placed at the Broker-2 that sits in the first rack.

However the first follower of partition-4 goes to a different rack at broker-5

So even if one of the racks is entirely down we still have the atleast one copy of the partition-4 available on the other rack.

Similarly the second follower partion-4 placed at a different broker-0 at rack-1, so if even two brokers are down(R1-B2, R1-B5) we still have the another broker (R1-B0) available to server the partition-4.

We can validated with the other partitions as well. They all are well arranged in a way that at least two copies of partitions placed on two different racks. This arrangement is ideal for fault tolerance.

Partition Leader Vs Partition Follower

We know brokers manages the two types of partitions:

1. Leader Partition
2. Follower Partition

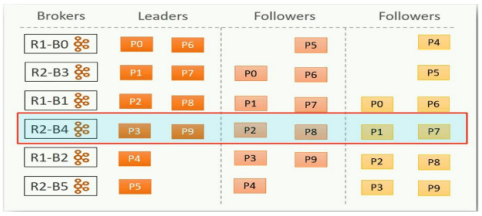
Depending up on the Follower type, a typical Broker performs two kinds of activities:

1. Leader Activity

2. Follower Activity

Let's try to understand it. In the above example, we allocated 30 replicas among six brokers. Now each broker owns multiple replicas.

Apache Kafka



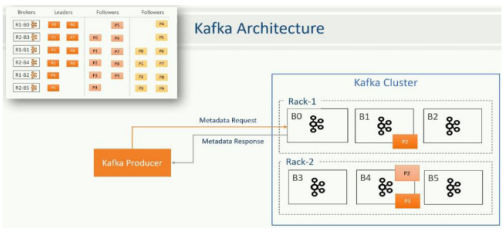
For example, in the above figure, the broker B4 holds six replicas. Two of these replicas are the leader partitions, and the remaining four are the follower partitions. So, the broker acts as a leader for the two leader partitions, and it also works as a follower for the remaining four follower partitions.

Leader

Kafka broker acts as a leader for all the leader partitions that are allocated to the broker. Regarding Kafka, it means two things:

1. The leader is responsible for all the requests from the producers and consumers.
2. The followers do not directly interact with producers and consumers.

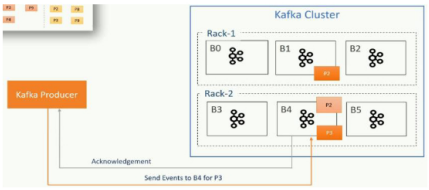
Let's assume that a producer wants to send some messages to a Kafka topic. So, the producer will connect to one of the brokers in the cluster and query for the topic metadata. All Kafka Brokers can answer a metadata request, and hence the producer can connect to any of the broker and query for the metadata.



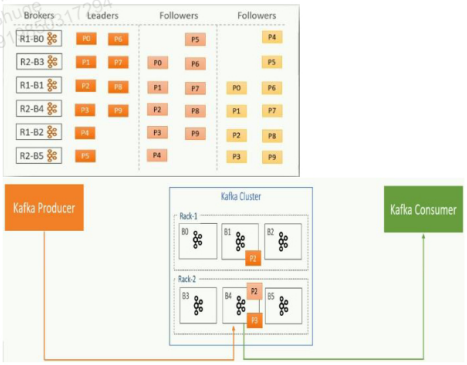
The metadata contains a list of all the leader partitions and their respective host and port information. Now the producer has a list of all leaders. It is the producer that decides on which partition does it want to send the data, and accordingly send the message to the respective broker.

That means, the producer directly transmits the message to a leader. On receiving the message, the leader broker persists the message in the leader partition and sends back an acknowledgment.

Apache Kafka

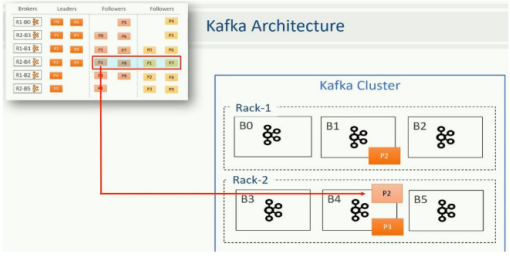


Similarly, when a consumer wants to read messages, it always reads from the leader of the partition. So, in the below figure, whenever a producer wants to send a message to Partion-3(P3) or the Partition-9(P9) partitions, it will reach out the broker B4.



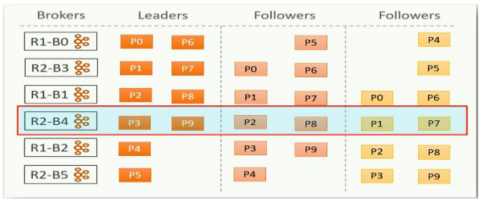
Follower

Kafka broker also acts as a follower for all the follower partitions that are allocated to the broker.

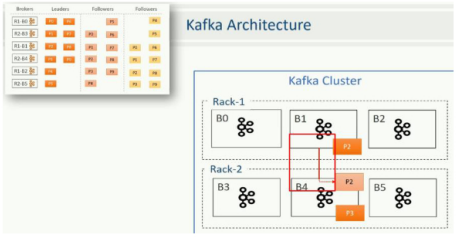


Apache Kafka

In the above Figure, the broker B4 owns four follower partitions(P2, P8,P1,P7) and hence the B4 acts as a follower for these replicas.



Followers do not serve producer and consumer requests. Their only job is to copy messages from the leader and stay up-to-date with all the messages.



The aim of the follower is to get elected as a leader when the current leader fails or dies. However, they can't get elected as a leader if they are falling behind the leader and fail to be in sync with the leader by copying all the messages.

This goes on forever as an infinite loop to ensure that the followers are in sync with the leader.

ISR - In Sync Replica

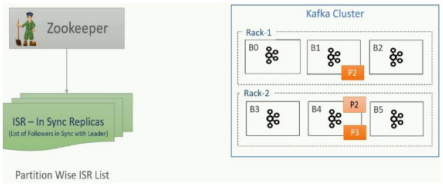
The method of copying messages at the follower appears full proof. However, some followers can still fail to stay in sync for various reasons. Two common causes are network congestion and broker failure.

1. Network congestion can slow down replication, and the followers may start falling behind.

2. When a broker crashes, all replicas on that broker will begin falling back until we restart the broker and they can start replicating again.

Apache Kafka

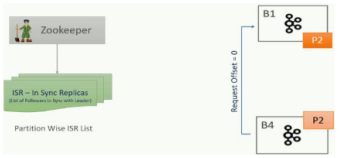
Since the replicas may be falling behind, the leader has one more important job to maintain a list of In-Sync-Replicas (ISR). This list is known as the ISR list of the partition and persisted in the Zookeeper.



All the followers in the ISR list are in-sync with the leader, and they are an excellent candidate to be elected as a new leader when something wrong happens to the current leader.

However, we might wonder, how do a leader would know if the follower is in sync or still lagging? Let's try to understand that.

The follower will connect to the leader and request for the messages. The first request would ask the leader to send messages starting from the offset zero.



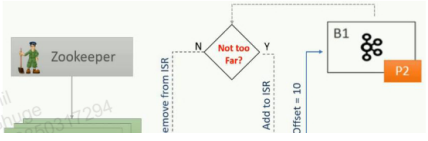
Assume that the leader has got 10 messages.

so it sends all of them to the follower. The follower stores them into the replica and again requests for new messages starting from offset 10.

Apache Kafka

In this case, since the follower asked for offset 10, that means a leader can safely assume that the follower has already persisted all the earlier messages.

So, by looking at the last offset requested by the follower, the leader can tell how far behind is the replica. Now the ISR list is easy to maintain. If the replica is “not too far”, the leader will add the follower to the ISR list, else the follower is removed for the ISR list.



That means, the ISR list is dynamic and the followers keep getting added and removed from the ISR list depending on how far they maintain their In-Sync status. However, there is a catch here. How do we define the "not too far"?

As a matter of fact, the follower will always be a little behind than the leader, and that's obvious because follower needs to ask for the message from the leader, receive the message over the network, store them into the replica and then ask for more.

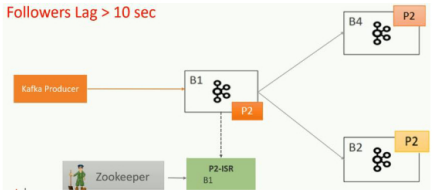
All this activity takes some time. And hence the leader gives them some minimum time as a margin to accomplish this. That is where the notion of "not too far" arrives.

The default value of "not too far" is 10 seconds. However, we can increase or decrease it using Kafka configurations. So, the replica is kept in the ISR list if they are not more than 10 seconds behind the leader. That means, If the replica has requested the most recent message in the last 10 seconds, they deserve to be in the ISR. If not, the leader removes the replica from the ISR list.

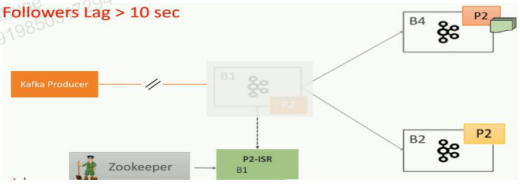
Committed vs Uncommitted

The mechanism to maintain the ISR list is quite fancy. However, we still have a gotcha in this mechanism. Let's try to understand it.

Assume all the followers in the ISR list are 11 seconds behind the leader. That means none of them to qualify for the ISR.



Now for some reason, the leader crashed, and we need to elect a new leader. Who do we choose? If we elect a new leader among the followers that are not in the ISR, don't we lose those messages that are collected at the leader during the most recent 11 seconds? Yes, we lose them. How do we handle that?

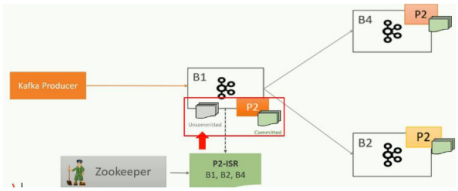


The solution for the above problem we can implement using two concepts:

1. Committed Vs Un Committed

2. Minimum In-Sync Replicas

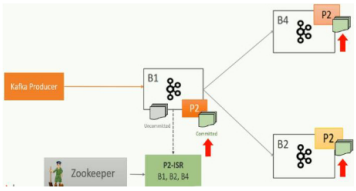
We can configure the leader not to consider a message committed until the message is copied at all the followers in the ISR list.



If we do this , the leader may have some committed as well as some uncommitted messages.

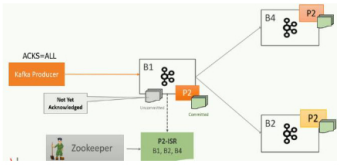
The message is committed when it is safely copied at all replicas in the ISR. Once the message is committed, we can't lose it until we lose all the replicas.

Apache Kafka



However, if we lose the leader, we will also miss the uncommitted messages. We no need to worry here about the un committed messages because the protection against the uncommitted messages can be implemented on the producer side.

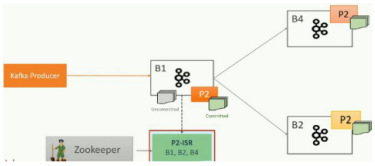
Producers can choose to receive acknowledgments of sent messages only after the message is fully committed. In that case, the producer waits for the acknowledgment for a timeout period and resend the messages in the absence of commit acknowledgment.



So, the uncommitted messages are lost at the failing leader, but the newly elected leader will receive those again from the producer. that's how all the messages can be protected from the loss.

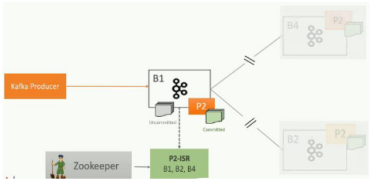
Minimum In-Sync Replicas

The data is considered committed when it is written to all the in-sync replicas. Let's Assume that we start with three replicas and all of them are healthy enough to be in the ISR.

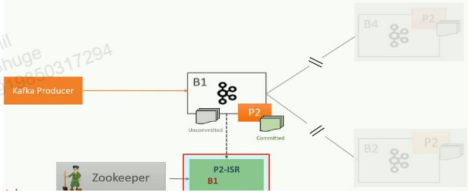


However, after some time, two of them failed

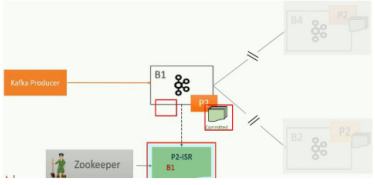
Apache Kafka



and as a result of that, the leader will remove them from the ISR.

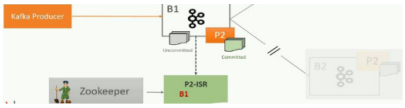


In this case, even though we configured a topic to have three replicas, we are left with a single in-sync replica that is the leader itself. Now the data is considered committed when it is written to all in-sync replicas, even when all means just one replica (the leader itself) now.



It is a risky scenario for data consistency because data could be lost if we lose the leader. Kafka protects this scenario by setting the minimum number of in-sync replicas for a topic.

If we would like to be sure that committed data is written to at least two replicas, we need to set the minimum number of in-sync replicas as two.



Apache Kafka

However, there is a side effect of this setting. If a topic has three replicas and we set minimum in-sync replicas to 2, then we can only write to a partition in the topic if at least two out of the three replicas are in-sync.

When at least two replicas are not in sync, the broker will not accept any messages and respond with "Not Enough Replicas" Exception. In other words, the leader becomes a read-only partition. We can read, but we can't write until we bring another replica online and wait for it to catch up and get in sync.