Introduction to Kafka

# Kafka scaling and resiliency

## Scaling with Kafka

* Horizontally scalable
  + It can process millions of messages per second.
  + It can handle terabytes of data.
* Scalability features
  + Messages can be spread across multiple topics and partitions. These can be stored and processed in parallel.
  + Kafka brokers can be wired together to create clusters. Brokers in a cluster share the workload of managing and serving messages.
  + Kafka supports multiple concurrent producers and consumers to enable parallel publishing and consumption of data.
  + It also has asynchronous publishing and batching options to reduce latency, as well as network traffic.
  + Consumer groups are another capability that allows consumers to share workloads and scale.

## Resiliency with Kafka

* Storage failures
  + Replication & Mirroring: Kafka supports duplication of data storage using replication and mirroring. This allows it to recover from node or data center failures.
* Broker failures
  + Controller election & Topic leader election: Kafka brokers work together to choose controller and topic leaders among them. When one of them goes down, other brokers quickly work together to choose an alternate broker to execute these functions.
  + Offset tracking: Features like offset tracking and partitioned reassignments help consumers to overcome failures and reprocess data without missing any messages.

Kafka Scaling Concepts

# Clusters and controllers

Central to Kafka's scalability and resilience is its ability to create and manage clusters of brokers.

## Kafka Cluster

A Kafka cluster is a group of Kafka brokers working together to execute Kafka functions. While a single Kafka broker provides all the essential functions, it can only scale and provide resiliency when it can share the workload with other brokers.

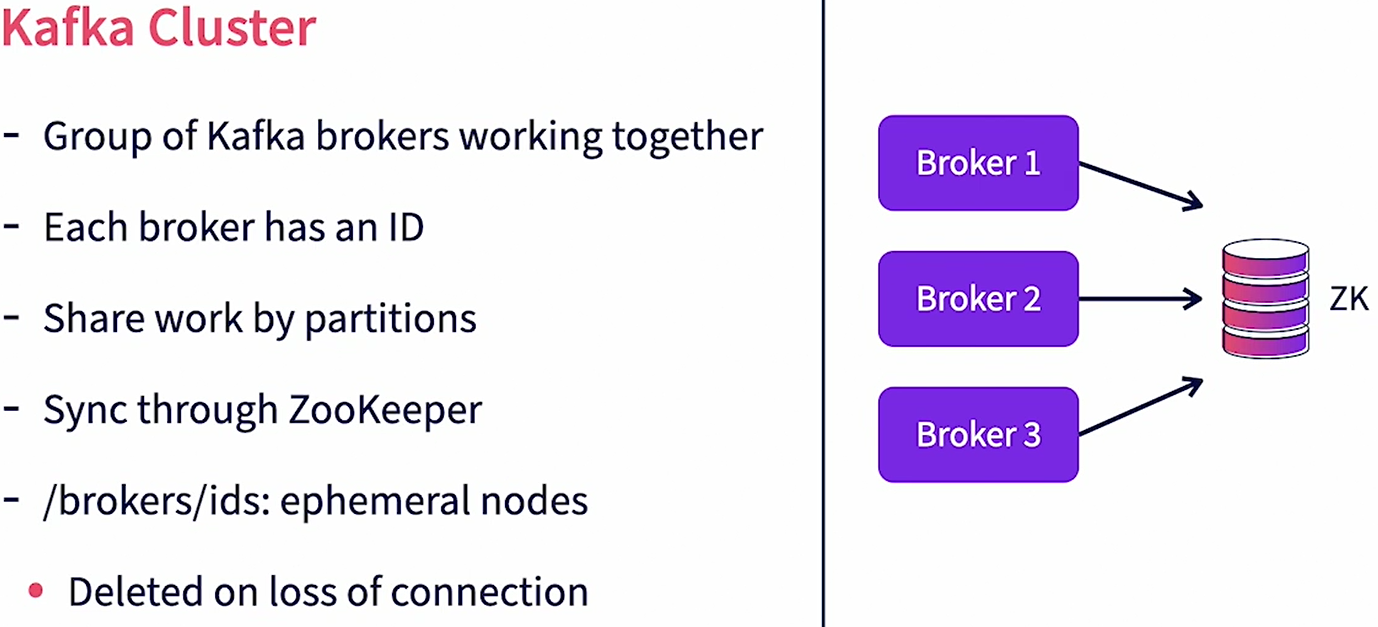
Each broker in a cluster has a unique broker ID. It can either be set manually in broker configuration or Kafka can determine it automatically.

The unit of work sharing among Kafka brokers is a topic partition. Brokers are assigned as partition leaders to manage topic data. They’re also assigned to topic replicas to maintain replicated copies of the data. Kafka can thus scale by adding more brokers to manage additional topics, partitions, and data loads.

### How do Kafka brokers in a cluster discover about each other?

They do it through ZooKeeper. Each broker needs a ZooKeeper cluster to work with. When a broker starts, it registers itself under the **/brokers/IDs** node in ZooKeeper. It also discovers other brokers, topics and partition assignments through ZooKeeper.

The ZooKeeper node is an ephemeral node, so when a broker goes down, it is immediately deleted.



### Who manages all brokers in a Kafka Cluster?

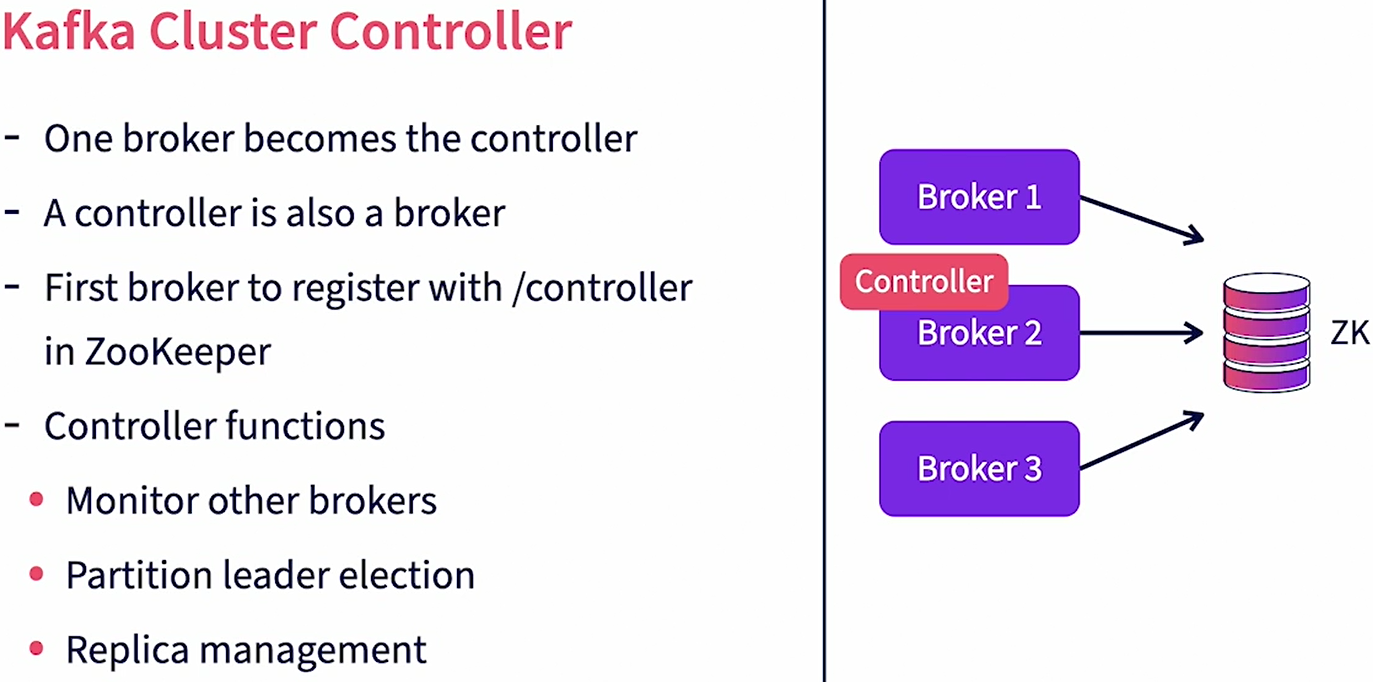
The cluster controller manages all brokers in a Kafka Cluster. A cluster controller is just a broker in the same cluster that assumes additional responsibilities of the controller. A controller is just another broker too, so in addition to controller responsibilities, it will also execute the responsibilities of a regular broker.

### How is a controller assigned in a cluster?

When a Kafka broker starts up, it tries to register itself as the controller by creating an entry under the **/controller** node in ZooKeeper. If the entry has not been created previously, it means that no other controller exists. So this broker becomes the controller. If the entry already exists, it means that a controller is already active. This broker then steps back and starts working with the existing controller.

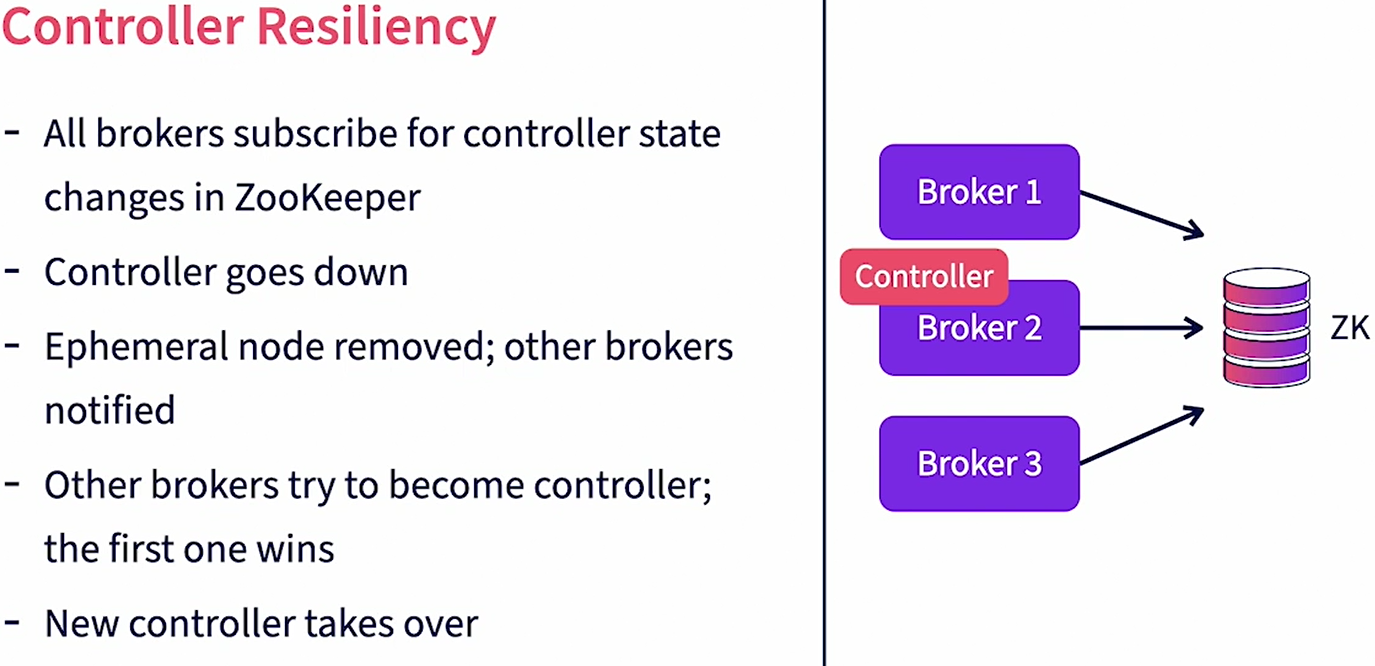
### What are the key controller functions?

The controller monitors all other brokers whom it discovers through ZooKeeper. If another broker goes down, the controller reassigns the leader partitions and replicas owned by this broker to other active brokers. When topics are created, the controller distributes the partitions and its replicas to brokers in the cluster. It ensures even distribution of load across the cluster. It also manages replicas. When a broker, which is a partition leader, goes down, it promotes one of the replicas on another broker as the leader partition.



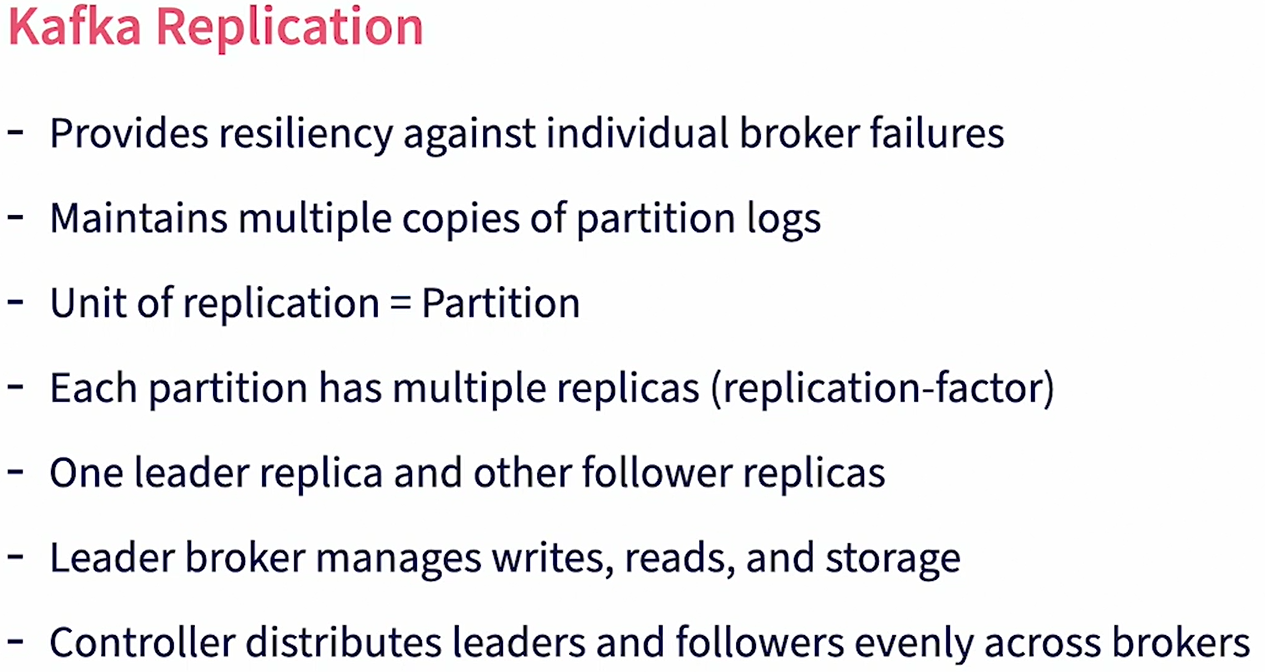
### What if the controller itself goes down? How does Kafka choose a new controller?

All brokers in a cluster subscribe to state changes in ZooKeeper for the **/controller** node. When the controller goes down, its heartbeat to ZooKeeper is lost. So ZooKeeper removes the **/controller** nodes and notifies other brokers of the change. Immediately, all other brokers try to become the controller. The first broker that successfully manages to write to the **/controller** node now becomes the new controller. It then reads all the clusters, brokers, and topics’ information from ZooKeeper and uses it to start managing the cluster. Clusters help Kafka provide scalability and resiliency.

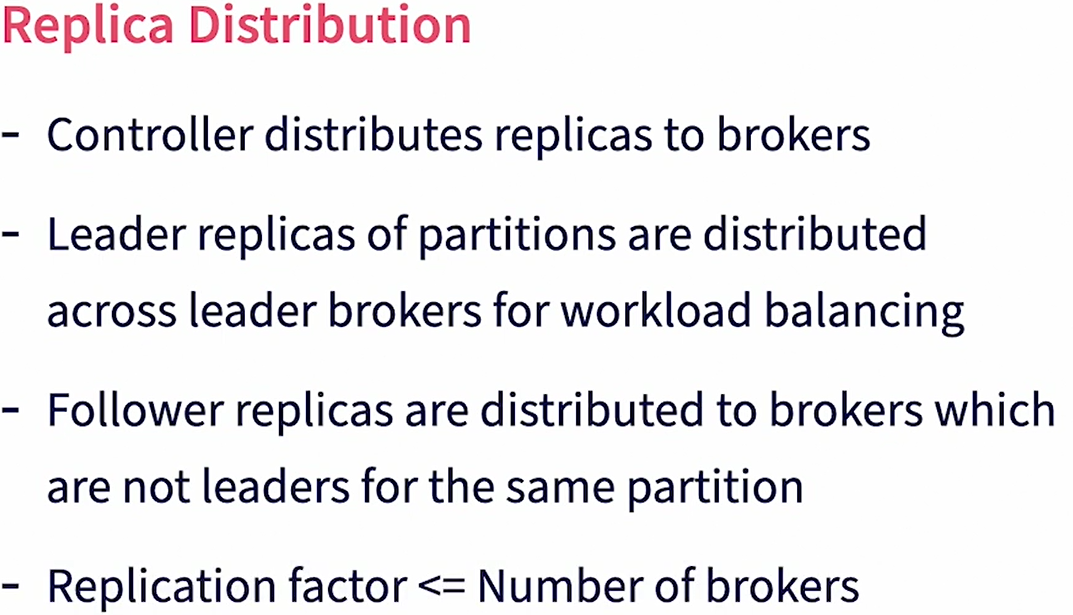


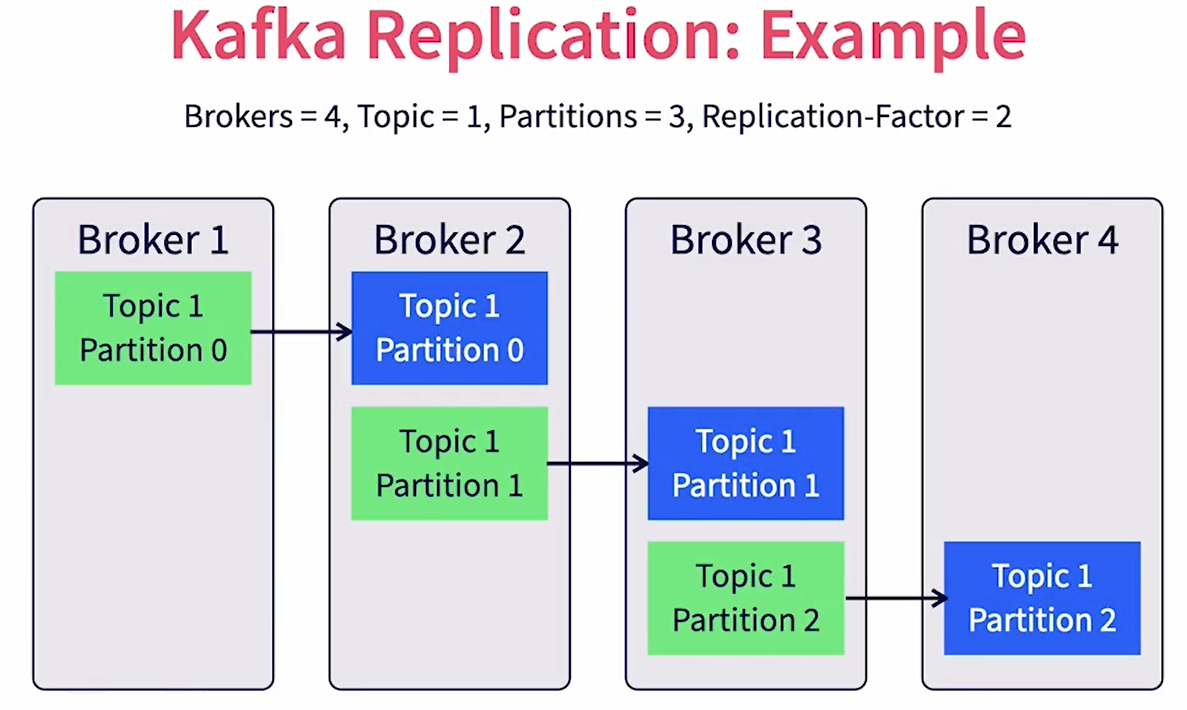
# Replication

Replication is a feature in Kafka that provides resiliency against individual broker failures. Replication works by maintaining multiple copies of individual partition logs across different brokers. The unit of replication is a single topic partition. All messages in a partition are replicated. When a partition is created we can specify the number of replicas needed for that partition. This is set using the parameter **replication-factor.** A partition would always have one leader replica and zero or more follower replicas. Leader replica is the primary copy where all reads and writes happen. Follower replicas are the backups but the primary copy is replicated. The replication factor counts the total copies available including the leader replica. The leader replica is assigned to a leader broker, the leader broker manages all reads and writes for that partition. It also maintains the local log files. Brokers owning replica copies will subscribe to the leader broker to get new messages and update their copies.



When the topic is created the controller takes care of distributing partitions. It first distributes the leader replicas among the brokers available. If the number of brokers is less than the number of partitions a single broker may get multiple leader replicas for the same topic. Then it distributes the follower replica to brokers such that the replicas are owned by brokers which are not the leader broker. The replication-factor cannot be larger than the total number of brokers available.





Example:

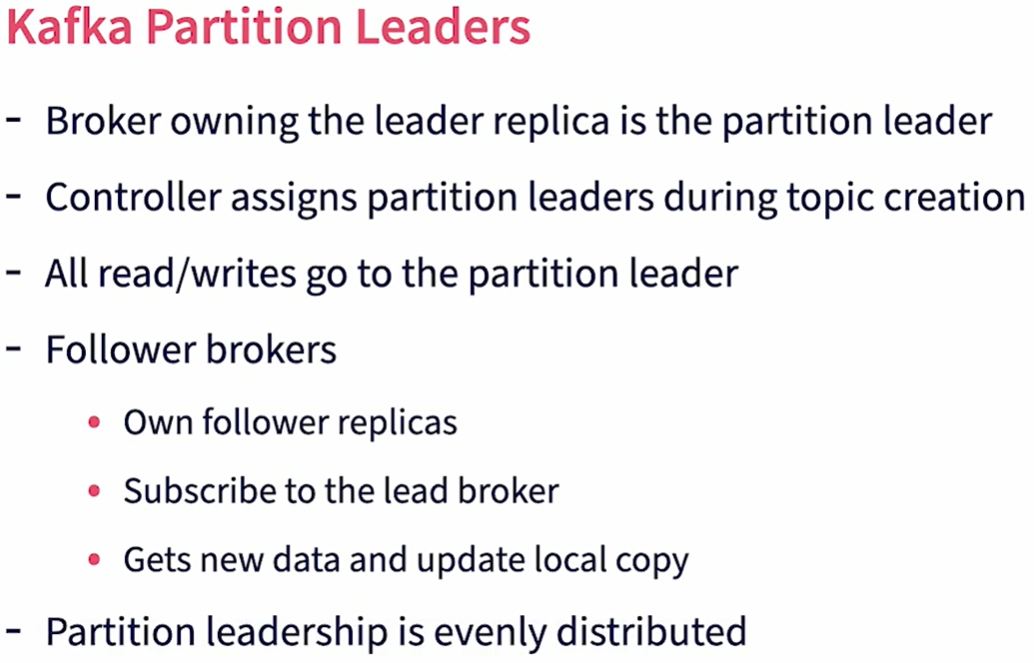
We have a Kafka cluster with four brokers. We have a single topic which is created with three partitions and a replication-factor of two. In this example, leader replicas are shown in green and the follower replicas are shown in blue. For partition zero, broker one is assigned as the leader replica and broker two gets the follower replica. For partition one, broker two gets the leader replica and broker three gets the partition replica, and so on. The controller ensures that the leader replicas are distributed across brokers for workload balancing. Similarly, it ensures that the follower replicas are distributed across brokers to provide maximum resiliency.

# Partition Leaders

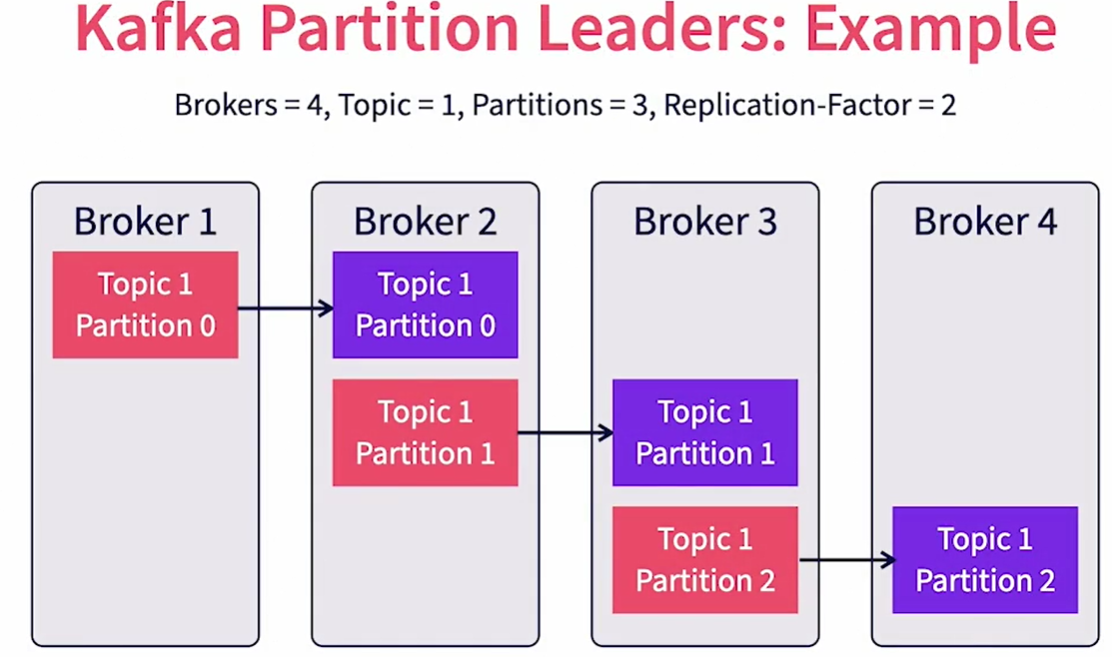
The broker instance that owns the leader replica of a partition is called the partition leader. The controller will assign partition leaders for partitions during topic creation. A Kafka producer, when publishing a message will choose a partition for a message and will send the message to the corresponding partition leader. Similarly, a Kafka consumer will work directly with the partition leader to consume messages in that partition. The partition leader also stored partition data in its log files.

### What about Follower Replicas?

Brokers that own the follower replicas for a partition are called follower brokers. Do note that the same broker can be the partition leader for some partitions and the follower broker for other partitions. Follower brokers subscribe with the leaders for partition data. They keep receiving new messages and will use that to update local copies of their data. Partitioned leadership is evenly distributed by the controller across the brokers in a cluster.



Example



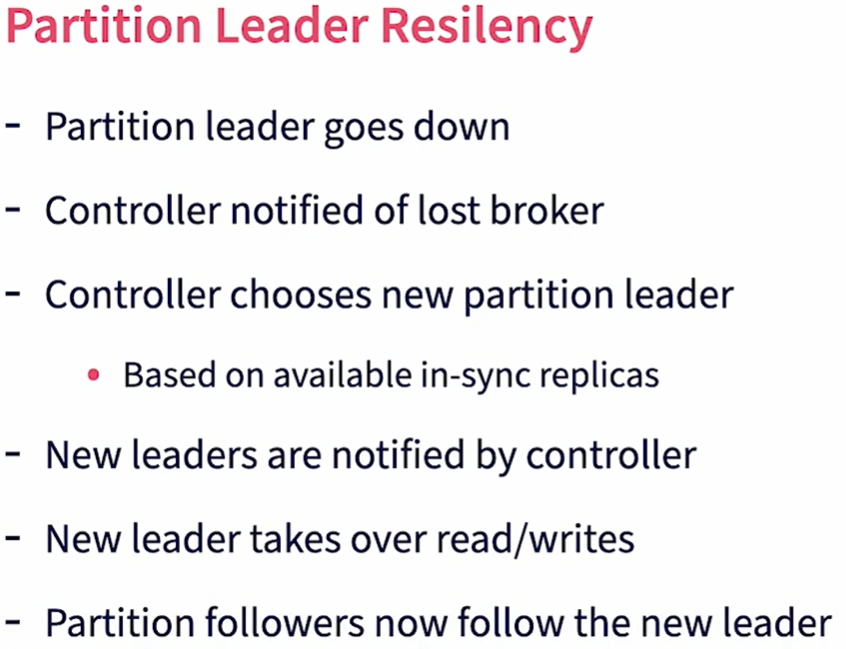
In this case, brokers one, two and three are the lead brokers for partitions zero, one and two respectively. Broker four is not the leader broker for any partition. Brokers two, three and four are the follower brokers for partitions zero, one and two respectively. Given that the **replication-factor** is two, there is only one follower for each partition. Increasing replication-factor will create more followers and hence more redundancy, but at the expense of more work to replicate data.

### How does Kafka take care of ensuring uninterrupted service when a partition leader goes down?

Let’s say that a partition leader for a specific partition goes down. The controller is notified of the lost broker through ZooKeeper, as it keeps track of ZooKeeper changes. The controller will then retrieve the list of all topic partitions for which this broker is the partition leader. Note that this broker can handle multiple topics and their partitions. The controller will then identify the list of in-sync replicas for each partition. An in-sync replica is one that is completely synced up with the partition leader with no messages left to replicate. It will then choose one of the brokers with an in-sync replica as the new partition leader. Again, the controller ensures even distribution of load. The broker chosen as the new partition leader is notified by the controller. This information is also received by the producers and consumers of that partition. The new leader then takes over the read/write operations and the producers and consumers will start talking to this broker going forward. The follower brokers will not start following this leader to keep track of their replicas.

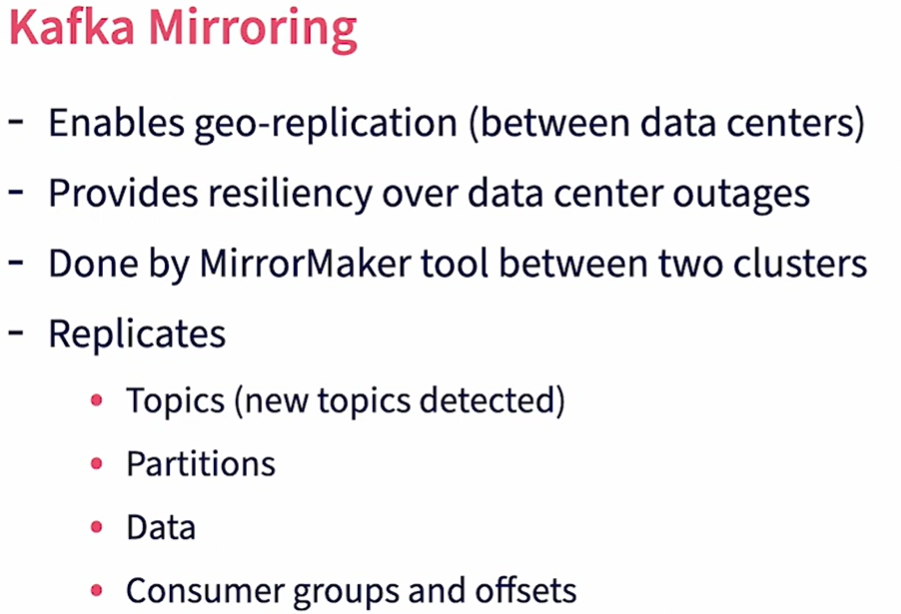
### What happens if a follower broker goes down?

In this case, Kafka will simply mark that replica as out of sync. When the broker comes back up it will catch up with the leader.

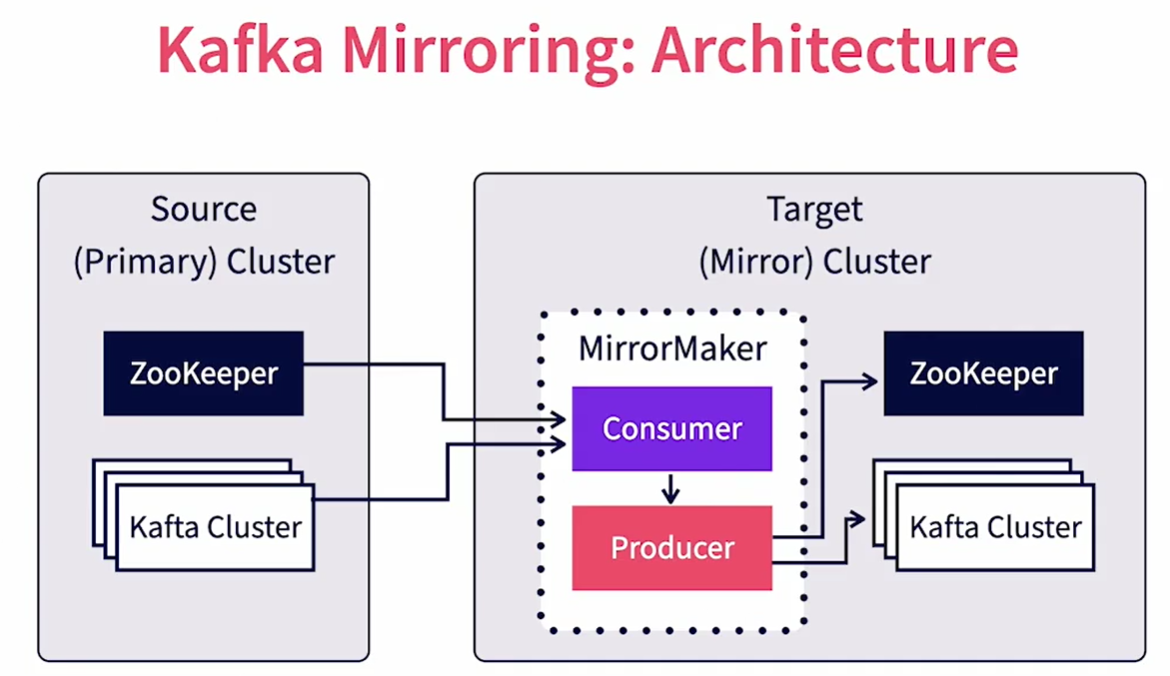


# Mirroring

Mirroring is another option in Kafka that helps with geo-redundancy. Mirroring is an option in Kafka that allows for message redundancy across data centers. While replication works to copy data within a Kafka cluster, mirroring helps to copy data between Kafka clusters. Mirroring provides resilience when the entire primary cluster goes down or becomes unreachable. Mirroring happens between a primary and a backup Kafka cluster and is done using the MirrorMaker tool. Mirroring will replicate all data and metadata in the primary cluster. This includes **topics, topic partitions** and **actual message data.** In addition, information about consumer groups and their offsets are also replicated. So when the primary Kafka cluster goes down, consumers can connect to the backup cluster and continue from where they left off.



### Let’s now review how mirroring works?



We have two Kafka clusters involved. We have the source or primary cluster set up in a primary data center. We also have a target or mirror cluster set up in the target data center. While both clusters can technically be together in the same physical location, even in the same virtual machine, the true benefit of mirroring materializes only when they are both in physically independent locations. A MirrorMaker tool runs in the target data center. The MirrorMaker tool behaves as a consumer of data for the primary cluster and a producer of data to the secondary cluster. Thus the Kafka clusters themselves don't know about mirroring. The primary cluster treats the MirrorMaker as a regular consumer while the target cluster treats the MirrorMaker as a normal producer. MirrorMaker would consume the messages from the source, then turn around and publish the messages to the target. MirrorMaker will also subscribe to the source ZooKeeper to keep track of topic and consumer group changes and faithfully replicates them to the target ZooKeeper. Note that both the producers and consumers should be aware of both the primary and target clusters and be able to switch from one to the other when connections to the primary are lost.

# Security

One of the requirements for enterprise data processing is ensuring security of data both at rest and in motion. Let's review various security options available in Kafka in this video. Kafka provides client authentication using SSL/SASL. This applies to producers, consumers, other brokers and zookeepers. Kafka provides read-write authentication control by individual topics and consumer groups. Data-inflight can be encrypted using SSL. For at-rest encryption, Kafka does not provide an out of the box solution. It is recommended to use encrypted disks for storage. Kafka security features are basic as it is expected to be used as a messaging system within a trusted network where the producers and consumers are also within the same trusted domain. It's not recommended to publicly expose Kafka through the internet without an API layer. Similarly multitenancy is hard to implement in Kafka as SSL certificate key management by individual tenants and users does not scale. It's recommended to use additional custom layers in the producers and consumers to provide fine-grained security and multitenancy.



