Achieve high availability for Apache Kafka

Contents

Overview 1

Determine what high availability means for your organization 2

Identify where you can achieve high availability 3

Track replicas that are in sync 6

Use deployment topologies 9

Get started with high availability for Apache Kafka 12

"We wanted to create a solution that would offer flexible, secure financial transfers, improving our customer experience while reducing transaction costs and promoting financial diversity."[1]

Vicente Fernandes
Chief of Division of Server Architecture, Storage and Basic Software, Central Bank of Brazil

Overview

Across industries, high availability is a crucial factor for providing a good customer experience. The challenge for many organizations is achieving robust, highly available infrastructure that does not replicate complexity or drastically increase future costs.

Apache Kafka is a horizontally scalable, highly available, fault tolerant, low-latency, high-throughput event streaming platform. It handles data streams from multiple sources and delivers them wherever data is needed. Organizations use Kafka to move massive amounts of data constantly from points A to Z and anywhere else it is needed.

facebook.com/redhatinc
@RedHat
linkedin.com/company/red-hat

1 Red Hat case study. "Brazilian bank builds instant payment network with Red Hat," 20 January 2021.

Determine what high availability means for your organization

High availability can mean many things to different organizations depending on their size and fault tolerance requirements.

While Kafka can handle millions of data points per second, which makes it well-suited for big data challenges, it is also useful for organizations that do not handle such extreme data scenarios. In many data-processing use cases such as the Internet of Things (IoT) and social media, data is increasing exponentially, and may overwhelm an application you are building.

A good first step to achieving high availability is to begin with an understanding of the components involved. A Kafka cluster can be broken down into two components: a control plane and a data plane, each with its own responsibilities that work together to transfer data where it needs to go.

Control plane responsibilities include:

- Knowing which servers are alive.
- Making appropriate changes when a server is detected as down.
- Storing and exchanging metadata.

Data plane responsibilities include:

- Handling requests to produce and fetch records and other application requests.
- Reacting to metadata changes from the control plane.

Historically, Kafka used an Apache ZooKeeper cluster to provide most of its control plane functionality. ZooKeeper tracks each broker and provides replicated and consistent storage for the cluster metadata. ZooKeeper also elects one Kafka broker to be the controller. The controller has extra, non data plane duties to manage the state of the cluster, such as responding to brokers that crash or restart.

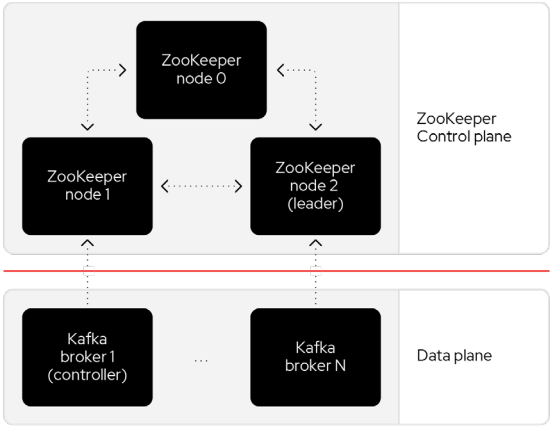

Figure 1: ZooKeeper architecture for Kafka.

The new architecture removes the ZooKeeper dependency and replaces it with a flavor of the Raft consensus protocol, allowing each server in the Kafka cluster to take the role of broker, controller, or both. The controller cluster will perform the same roles as the cluster of ZooKeeper nodes did previously, but the Kafka controller will now be elected from the controllers instead of the brokers.

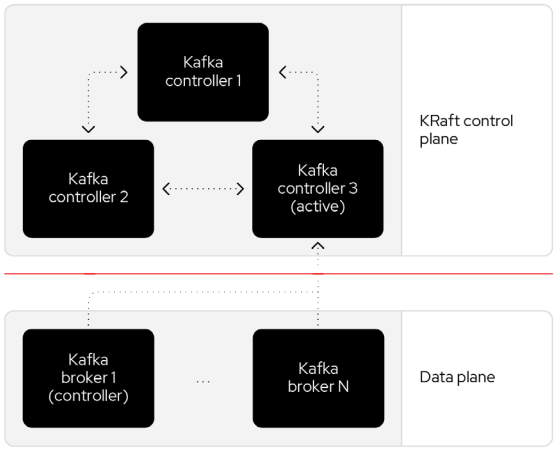

Figure 2: KRaft architecture for Kafka.

For a Kafka cluster to be highly available, you need to make certain both the data plane and control plane (whichever kind is being used) are highly available.

Identify where you can achieve high availability

To locate where high availability is possible in your organization, consider the control plane and data plane.

To begin, it is important to understand what quorum is and why it matters. In order to maintain cluster integrity and availability, cluster systems use a concept known as quorum to prevent data corruption and loss. A cluster has quorum when more than half of the cluster nodes are online.

Quorum is established using a voting system. When a cluster node does not function as it should or loses communication with the rest of the cluster, the majority working nodes can vote to isolate and, if needed, fence the node for servicing. For example, in a 6-node cluster, quorum is established when at least 4 cluster nodes are functioning. If the majority of nodes go offline or become unavailable, the cluster no longer has quorum.

High availability for a ZooKeeper control plane

ZooKeeper provides quorum within the cluster by solving the distributed consensus problem using the ZooKeeper Atomic Broadcast (ZAB) protocol.

The servers (or nodes) which make up the ZooKeeper cluster (or ensemble) vote to elect a leader. The

server with a majority of votes becomes the leader and can accept writes from clients which change the cluster state. The writes are only successful when they have been replicated to a majority of the servers in the ZooKeeper ensemble.

The need for a majority during both the election and write processes means for another server to become the leader, it must get at least one vote from the server that most recently replicated a successful write done by the last controller. That voter will not vote for the proposed new controller if the proposed controller's view of the cluster state is not up to date. This election process is the basis for ensuring that every node in the ensemble eventually has a consistent copy of the cluster's state.

The availability of a ZooKeeper ensemble can be determined by considering the majorities that can or cannot be formed under various failure scenarios. There are two immediate, topology-independent consequences for needing a majority.

1. **A quorum of nodes must be able to communicate.** Otherwise, the cluster will not function.
2. **Using an odd number of nodes ensures if there is a network partition,** one side of the partition has enough nodes to form a majority.

A highly available ZooKeeper ensemble requires at least three servers. If there are more than three, then an odd number should be used. An ensemble with 2n+1 servers will remain available even while up to n servers are unavailable. While more nodes allow you to tolerate more failures, they also require n+1 acknowledgements (ack) of every write. To summarize this point, your writes are only as fast as the $(n+1)^{th}$ slowest node.

The achievable level of availability can also depend on how independent failures affecting servers are, in practice. For example, in a three-node ensemble, where each node is on a different machine but in the same rack, the protocol will provide resilience against one of the machines failing and node processes crashing, or being restarted. But those nodes rely on the same network and power supply. A deployment like this might be highly available enough for some Kafka use cases, but not others.

Higher levels of availability are achievable by eliminating such single points of failure. These are possible on-premise and in a cloud environment.

- **On-premise:** Use different racks within a datacenter, or for even higher availability use multiple datacenters.
- **In a cloud environment:** Use multiple availability zones (AZs) because a single AZ does not have a service level agreement (SLA) covering availability.

The redundancy inherent in eliminating single points of failure results in a cost or availability trade-off that your organization will need to assess. Using a minimal three-node topology means there is not enough resilience to tolerate the loss (including restart) of any server during a network partition or infrastructure failure. Many use cases that require high availability can accept this limitation, while others cannot.

To tolerate more nodes being offline, it may be acceptable to spread additional servers across the three datacenters, as evenly as possible. For example, here is a comparison of two five-node clusters:

- **Ensemble A:** Consists of two nodes in each AZ1 and AZ2 and a fifth in AZ3.

This will be resilient to the loss of any datacenters or any two nodes. But while either AZ1 or AZ2 are partitioned, the loss of any of the remaining nodes cannot be tolerated.

- **Ensemble B:** Consists of five availability zones, with a node in each. This can tolerate the loss of up to two AZ or nodes, or losing one while retaining the ability to perform nodes restarts.

Result: Ensemble B has higher availability but at a significant extra cost. Note that regions with 5 AZs are not common in public clouds.

High availability for a KRaft control plane

Apache Kafka Raft (KRaft) is the consensus protocol introduced to remove Apache Kafka's dependency on ZooKeeper for metadata management. This protocol simplifies Kafka's architecture by consolidating the responsibility for metadata into Kafka itself, rather than splitting it between two different systems: ZooKeeper and Kafka. KRaft mode makes use of a new quorum controller service in Kafka replacing the previous controller and using an event-based variant of the Raft consensus protocol.

While KRaft provides many benefits to Kafka, for high availability the same rules and semantics apply as they do with Zookeeper. KRaft is a different protocol than ZAB, but is still a quorum-based solution of the distributed consensus problem.

Migration from existing ZooKeeper-based Kafka clusters to KRaft-based clusters (once implemented) will be relatively straightforward. You will still need the same number of KRaft controllers as you have ZooKeeper nodes.

Kafka runs on the Java virtual machine (JVM). KRaft supports having the broker and controller within the same JVM. While the ability to colocate broker and controller on a JVM will be valuable for developers running Kafka locally. For production-Kafka use cases, having separate servers for brokers and controllers is the best choice for several reasons. These include:

- When a JVM supporting both a broker and controller crashes or restarts. The results must be handled on both control and data planes at the same time.
- Stop-the-world garbage collection events that would be more frequent and possibly take longer.
- Machine-global resources, such as input/output (IO) bandwidth and page cache are not isolated, so a busy or overloaded broker could affect a controller.

The preferred topology will consist of dedicated controller nodes that are isolated from Kafka clients. Only brokers and dedicated tooling will connect to the controller quorum.

High availability for the data plane

Kafka stores data by topic and each topic is composed of several partitions, which represent a logical slice, or piece, of the records on that topic. Partitions may be replicated across brokers, creating a copy of each record sent to a partition and will be physically stored in a log sitting on multiple different brokers. The copies of the log are called replicas.

The number of copies of a partition is called the replication factor, and should be the same across all the partitions in a topic. It is this replication of record data that means Kafka applications can, when deployed and configured correctly, be resilient to the loss of one or more brokers from the cluster. Losing a broker means the loss of data if the log is not replicated (replication factor of 1).

Track replicas that are in sync

For record data, Kafka uses its own log replication protocol, rather than the quorum-based replication used in the control plane.

One broker is elected as the leader of a partition. The other brokers with replicas are called followers. Because brokers can crash or be restarted, Kafka needs to account for things like followers falling behind the leader or the leader ceasing to be available. To do this, Kafka distinguishes between followers that can keep up as new records are appended, and those that cannot. The replicas that are keeping up are in a subset of all the replicas of the partition, known as the insync replicas (ISR). A follower has in-sync status while it is regularly fetching all the latest records.

When a producer attempts to send some records to the leader to be appended, it should wait for the leader to acknowledge that the records have been appended, and be prepared to resend those messages if an acknowledgement is not received, or does not indicate success.

When a follower fails to make sufficient fetch requests to the leader to keep up, the leader removes it from the ISR—a fact that is persisted in the partition metadata in the control plane. This is controlled using the replica.lag.time.max.ms broker configuration parameter. If the leader should crash or be shut down, the control plane will choose another replica from the ISR to become the new leader.

Producers wanting to append records to the partition's log must talk to the new leader. Followers will refuse to append records from producers, forcing the producer to find the current leader and resend the records.

Combining durability and availability guarantees

Kafka provides another configuration that allows the producer to choose when the broker should send acknowledgements (acks) of produce requests, which corresponds to a commitment of durability.

- **acks=0** means the producer does not want acknowledgements. This is useful for messages that can be lost without affecting business activity. In this case high availability is not a consideration because whether records can be produced or consumed is nonessential.
- **acks=1** means the broker will send acknowledgements to the producer once it has appended the records to the log, regardless if the other replicating brokers have already done so.

  Acknowledged records can still be lost, because of a broker crash or network partition. For example, if the leader crashes immediately after its local append, but before followers have received the appended records, then the new leader would never have received the acknowledged records.
- **acks=all** (or acks=-1) means that the leader sends an acknowledgement only when all the brokers in the ISR have appended the records to their log.

High availability without high durability

It is possible for a topic not to have a high durability guarantee and still require high availability. For those applications, acks=1 should be sufficient. More common is a requirement for a high availability and high durability guarantee. Acks=all has the potential to provide this requirement, but it is not sufficient on its own. There is nothing preventing the ISR from being the set that contains only the leader. In this case, acks=all will be the same as acks=1.

The min.insync.replicas topic configuration (sometimes referred to as min ISR) can put a lower bound on how large the ISR must be for an acknowledgement to be sent to an acks=all producer. Using min.insync.replicas greater than 1 in combination with acks=all allows you to prevent the singleton ISR case and provide a meaningful durability guarantee.

Delaying the sending of the acknowledgement to the producer means the producer might not make progress. For example, where the producer limits the number of inflight requests or buffer sizes. Combining durability and availability guarantees is a tradeoff between the ability of the producer to send records, sometimes known as producer availability and ability of the replicas to store those records with high durability.

What does appended mean?

The meaning of the phrase "appended the records to the log" depends on how the log is configured. By default, records are appended when the memory-mapped write has returned. Once the broker process has passed responsibility to the operating system (OS) kernel to update the bytes on disk. However, this does not mean that the actual bytes on disk have been updated. The data must pass through several layers in the kernel and disk firmware before that happens. If the machine hosting a broker crashes, then those records will not make it to the disk. If the leader sent an acknowledgement to a producer based on false information that the records were safely stored, then it is possible the record could be lost.

Assuming acks=all and min ISR is greater than 1 then Kafka's log replication protocol ensures if this happens to a single broker it will not be a problem. For example, assuming one broker in the ISR crashed this way, when it rejoined the cluster (as a follower) it would start fetching from the last valid record that was on the disk. A cyclic redundancy check (CRC) is used to detect partially written, corrupted records. Its log would be a consistent copy of the logs of other replicas.

This safety property rests on several assumptions, including:

- That crashes will not affect all members of the ISR at the same time.
- That crashes will result in replicas being removed from the ISR.
- That the CRC check is strong enough to detect partially written records.

For some high availability use cases, these are assumptions that cannot be made. Instead, you can configure the broker to flush writes to the log rather than letting the kernel decide when to flush. The topic configurations flush.messages and flush.ms can put a bound on how often to flush—based on the number of messages or elapsed time. With flush.messages=1, the broker will flush after every write to disk, so that the leader only sends acknowledgements when the records have arrived on the storage device. This additional safety causes increased latency and decreased throughput.

It is unimportant to consider that drives can report bytes as being written before they have actually hit the persistent media, but usually if they are cached on the device in volatile memory, the device will be battery-backed to allow writes to complete in the event of power failure.

Together, the replication factor,min.insync.replica, acks and flush can provide configurable high availability and durability for records. But note that the required durability affects high availability from the producers point of view.

Configurations required for high availability, include:

- The topic having a replication factor greater than 1 (3 is typical).
- The topic having a min.insync.replicas greater than 1 (2 is typical).
- The producer using acks=all (the default in Kafka 3, but not earlier versions).
- Being suitable to set flush.messages for the topic.

The replication factor (RF)=3, min ISR=2 configuration is really the starting point for a highly available topic, but it might not be enough for some applications. For example, if a broker crashes then it means that neither of the other brokers with replicas can be restarted without affecting producers with acks=all. Using RF=4 and min ISR=3 would avoid this, but would increase producer latency and lower throughput. Setting RF and min ISR to the same value should be avoided since it means producers with acks=all will be blocked if any replicating brokers go down.

Availability for consumers

Because of the need to replicate records to the ISR, producers can be affected if the size of the ISR is smaller than min.insync.replicas. But what about consumers? A consumer only requires a partition leader to make fetch requests. A consumer that is reading historical data (that is not current through to the end of the log) can make progress if there is a leader, even when a partition is under its min ISR. However, most Kafka consumers also append to a log.

These appends happen to partitions of the __consumer_offsets topic if offsets are being committed to Kafka, or if the consumer is part of a consumer group. It is important to bear this in mind when reasoning about high availability.

Alternatively, consumers can fetch records from any of the insync replicas. Although by default, the insync replicas will also use the leader.

Rack-aware replicas

The configuration parameters described so far are not enough to guarantee high availability. This is because there is nothing that forces the replicas to be assigned to brokers in different racks or AZs. Problematic assignments are not possible for clusters of three brokers in different zones. Each replica of a partition with three replicas is necessarily in a different AZs in that case. The problem manifests with larger Kafka clusters using multiple brokers in the same zone.

Kafka has some rack-awareness support that can spread replicas across zones when topics or new partitions of existing topics are created. This best-effort is only honored during creation. A later reassignment of partitions to brokers, which places replicas in the same rack or AZ, will not be rejected by the controller.

Kafka cluster management systems, such as cruise control, make certain the rack-aware placement of replicas is used. Whether such systems are used, it is good practice to monitor for, and schedule alerts for partitions which are not spread over racks or AZs.

Use deployment topologies

The typical practice is to design the system to tolerate the loss of an entire zone. You want your resources spread across zones so the loss of a zone does not cause an outage.

The minimal cluster that supports high availability in the data plane is one with three brokers, each in a different AZ, and where topics have a replication factor of 3 and a minISR of 2. This will permit a single broker to be down without affecting producers with acks=all. Having fewer brokers will sacrifice either availability or durability.

Evenly spread more brokers out across the three AZs if they are needed for the expected traffic volumes. But, make certain that the replicas are in different AZs using Kafka's rack awareness support.

On-premise scenario

If more brokers are needed for the expected traffic volumes, but deployed on-premise with only two datacenters, they could have a similar topology.

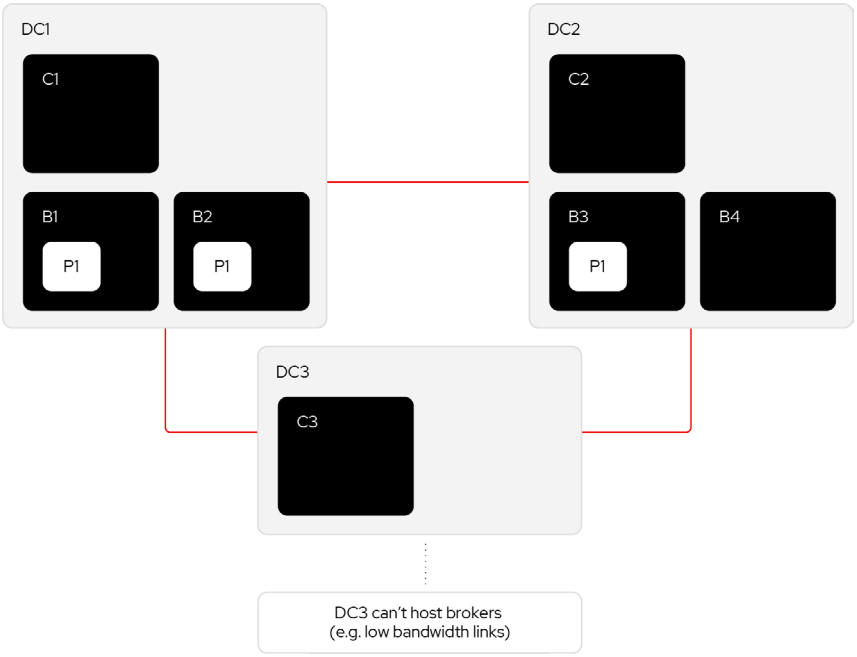

Figure 3: On-premise scenario with only two datacenters.

If we have a replication factor of 3, two of the replicas will be in one DC and the third replica will be in another. The situation is symmetrical, so consider the two replicas are in DC1.

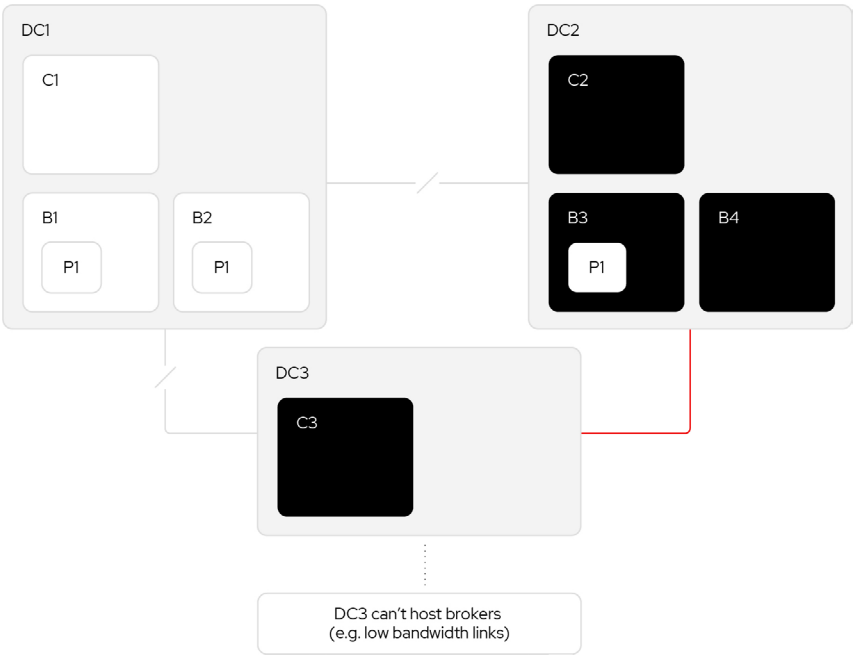

Figure 4: On-premise scenario with a replication factor of 3.

A network partition that separated DC1 from the others would mean the control plane still had a quorum. Partition P1 has a leader, but the two followers are partitioned. They will be unable to fetch from the leader and will drop out of the ISR, leaving only the leader. So producers with acks=all will be blocked because the min ISR will not be met. The result of this architecture is not high availability from the perspective of such producers. Producers with acks=0 or acks=1 will not be affected.

However, this scenario can be made high availability to producers with acks=all by using a replication factor for 4 (and keeping the min ISR of 2). In this case, the other broker in DC2 would also be a follower which meets the min ISR and producers with acks=all will not be blocked.

There are drawbacks to consider that include:

- 1/3 increase in storage costs.
- Increased latency.

But these may be acceptable for on-premise deployments if the alternative requires a whole new datacenter.

Even numbered replication factors

Is an odd number of replicas necessary in all cases? Let us think through what might happen if a partition occurs at an inopportune moment:

1. A record is appended to the leader, B3, at offset 12.
2. B4 replicates it.
3. Once B3 knows B4 has replicated it, B3 cannot send an acknowledgement while B1 or B2 are in the ISR.

Let us consider the possibilities if a partition or crash happens at this point.

- If B1 or B2 crashes, then after replica.lag.time.max.ms milliseconds, they will drop out of the ISR. The leader, B3, will tell the controller so that the failed broker cannot be elected as leader until it is caught up again. If they do not catch up within the producer's request.timeout.ms the producer will receive an acknowledgement with the NOT_ENOUGH_REPLICAS error code.
- If DC1 is partitioned, then after replica.lag.time.max.ms milliseconds both B1 and B2 will drop out of the ISR. Similar to the previous case, the producer will receive an acknowledgement with the NOT_ENOUGH_REPLICAS error code.
- If B3 crashes, the control plane will notice the lack of heart beats. The controller will elect a new leader from the ISR. This action could be completed by any of the other brokers. It does not matter if it is B1 or B2, although they did not replicate the record, because the record was never acknowledged. The producer will resend the produce request to the new leader.
- If B4 crashes, then we have to wait for B1 and B2 to replicate.
- If DC2 is partitioned, then the result is the same as when B3 crashes.

Conclusion

While high-availability requirements vary considerably between industries and use cases, it is important to understand the outcomes you are striving for and the cost you can incur to achieve successful outcomes. The cost can be measured both in terms of infrastructure needs and operational complexity, each measure needs to be considered to achieve high availability.

Ready to get started? Learn more and register for Red Hat training.

North America	Europe, Middle East, and Africa	Asia Pacific	Latin America
1 888 REDHAT1 www.redhat.com	00800 7334 2835 europe@redhat.com	+65 6490 4200 apac@redhat.com	+54 11 4329 7300 info-latam@redhat.com

facebook.com/redhatinc
@RedHat
linkedin.com/company/red-hat

redhat.com
#F31411_0422