# Chapter 13: Replication and Acknowledgements

Fundamentally, Apache Kafka is a distributed log-centric data store. Data is written across multiple nodes in a cluster and may be subject to a range of contingencies — disk failures, intermittent timeouts, process crashes, and network partitions. How Kafka behaves in the face of a contingency and the effect this has on the published data should be of material concern to the designer of an event-driven system.

This chapter explores one of the more nuanced features of Kafka — its replication protocol.

## Replication basics

The deliberate decisions made during the design of Kafka ensure that data written to the cluster will be both *durable* and *available* — meaning that it will survive failures of broker nodes and will be accessible to clients. The replication protocol is the specific mechanism by which this is achieved.

As it was stated in Chapter 3: Architecture and Core Concepts, the fundamental unit of streaming in Kafka is a partition. For all intents and purposes, a partition is a replicated log. The basic premise of a replicated log is straightforward: data is written to multiple replicas so that the *failure of one replica does not entail the loss of data*. Furthermore, replicas must *agree* among themselves with respect to the contents of the replicated log — reflecting on their local facsimile of the log. It would be unacceptable for two (or more replicas) to differ in their contents in some conflicting manner, as this would lead to data corruption. Broadly speaking, this notional agreement among the replicas is referred to as *distributed consensus*, and is one of the basic challenges faced by the designers of distributed systems.

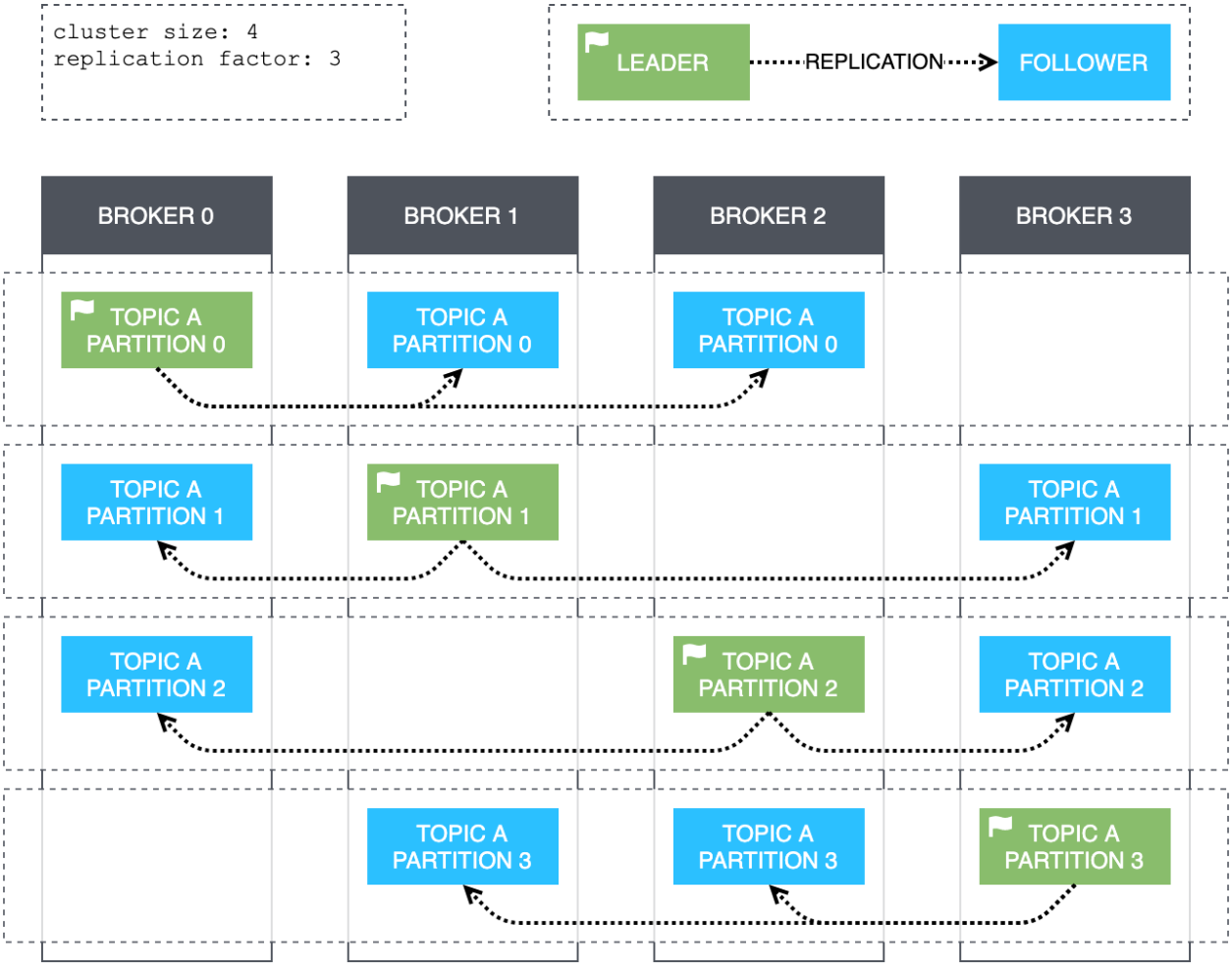
There are several approaches to implementing a distributed log; the one taken by Kafka follows a leader-follower model — a single leader is assigned by the cluster coordinator to take absolute mastership of the partition, with zero or more followers that tail the data written by the leader in near real-time, progressively building their own identical copies of the log. Putting it another way, replication in Kafka is *asynchronous*: replicas lag behind the leader, converging on its state when the traffic flow from the leader quiesces. Consensus is formed by ensuring that only one party administers changes to the log; all other parties implicitly agree by unconditionally replicating all changes from the leader, achieving *sequential consistency*. Under this consistency model, replicas can only vary in a contiguous segment comprising the last few records in the log — they cannot have gaps, nor can two replicas house different records at the same position in the log. This model vastly simplifies the consensus protocol, but some level of agreement is required nonetheless, because

sequential consistency is insufficient on its own. The protocol must ensure that records are durably persisted, which implies that certain aspects of the protocol must be synchronous.

In Kafka's parlance, both the leader and the follower roles are collectively referred to as *replicas*. The number of replicas is configured at the topic level, and is known as the topic's *replication factor*. During the exercises in Chapter 5: Getting Started, we created topics with a replication factor of one. There was no other choice then, as our test cluster comprised a single broker node. In practice, production clusters will comprise multiple nodes — three is often the minimum, although larger clusters are common.

The minimum permitted replication factor is one — offering no redundancy. Increasing the replication factor to two provides for a single follower replica, but will prevent further modifications to the data if one of the replicas fails. This configuration provides durability, but as for availability — this is only provided in the read aspect, as writes are not highly available. A replication factor of three provides both read and write availability, providing certain other conditions are met. Naturally, the replication factor cannot exceed the size of the cluster.

A naive replication model with a replication factor of three is depicted below. This is a simplification of what actually happens in Kafka, but it is nonetheless useful in visualising the relationship between leader and follower replicas. It will be followed shortly with a more complete model.




**Topic replication — simplified model**

Broker nodes are largely identical in every way; each node competes for the mastership of partition data on equal footing with its peers. Given the symmetric nature of the cluster, Kafka designates a single node – the *cluster controller* – for managing the partition assignments within the cluster. Partition leadership is apportioned approximately evenly among the brokers in a cluster when a topic is first created. Kafka allows the cluster to scale to a greater number of topics and partitions by adding more broker nodes; however, changes to the cluster size require explicit rebalancing of replicas on the operator's behalf. More on that later.

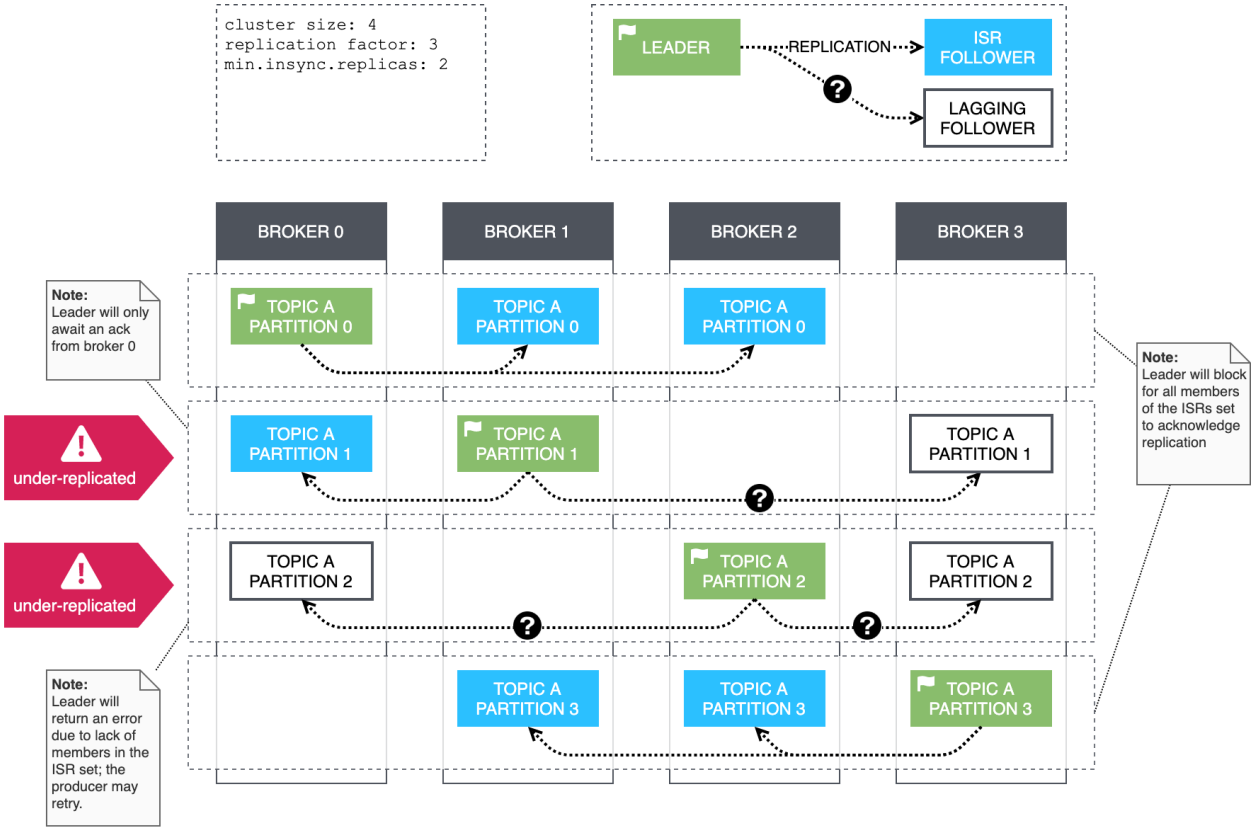
The main challenge with naively replicating data from a leader to a follower in the manner above, is that in order to guarantee durability, the leader must wait for the all replicas to acknowledge the write — before reporting to the producer that the write has been replicated to the degree implied by the replication factor. This makes the replication protocol sensitive to slow replicas, as the acknowledgement times are dependent on the slowest replicas. A single replica that is experiencing a period of degraded performance will affect all durable writes to the partition in question. Also, waiting for all replicas is not strictly necessary to achieving consensus in a replicated log.

Note, we use the term 'durable write' to refer to the persistence of those records where the producer has requested the highest level of durability. This is not necessarily the case with all writes; the producer can dictate the level of durability by specifying the number of required acknowledgements.

To lighten the burden of slow replicas, Kafka introduces the concept of *In-Sync Replicas*, abbreviated to *ISR*. This is a dynamically allocated set of replicas that can demonstrably keep pace with the leader. The leader is also included in the ISR. Normally, this implies that a replica is trailing the leader within some bounded window of time. (The time window being the difference between the timestamp of the record at the leader's log end offset and that of the follower.) Slow replicas are automatically removed from the ISR set, and as such, the cardinality of the ISR set may be lower than the replication factor. A partition that has had at least one replica removed from the ISR is said to be *under-replicated*. The responsibility of evaluating the performance of the ISR and maintaining the state of the ISR falls on the partition leader, which must also persist a copy of the ISR to ZooKeeper upon every change. This ensures that, should the leader fail, any of the followers can reconstruct the state of the ISR — enabling them to take over the leadership of the topic.

Instead of requiring the leader to garner acknowledgements from all follower replicas as in the earlier example, a durable write only requires that acknowledgements are received from those replicas in the ISR. The lower bound on the size of the ISR is specified by the `min.insync.replicas` configuration property on the broker. This property is complementary to the `default.replication.factor` property, being the defaults that will be applied to all topics. When creating a topic, its custodian may set an alternate replication factor and override the `min.insync.replicas` property as required. The producer client has no say in the replication factor or the minimum size of the in-sync replica set — it can only stipulate whether a write should be durable or not. Naturally, the consumer client has no say in any matter regarding durability.

When a partition leader is asserting a durability guarantee, it must ensure that all replicas in the ISR have acknowledged the write, before responding to the producer. Because the ISR automatically excludes underperforming replicas, the performance of the replication protocol is minimally impacted by a stalled or lagging replica. To be specific, a deteriorated replica will still affect the replication performance for as long as it is deemed a member of the ISR, but eventually it will be removed from the ISR — at that point and thereafter it will have no impact on performance. Prior to rejoining the ISR, a replica must catch up to the leader. A replication scheme based on ISR has been depicted below.




**Topic replication — model with ISR**

In the diagram above, the topic has been configured with a replication factor of three and a minimum ISR size of 2. For those partitions where there are three replicas in the ISR — a confirmation from both followers is necessary before a durable write can be acknowledged by the leader. For those partitions where the ISR has been reduced to two, the leader will wait for the remaining in-sync replica other than itself, before acknowledging the write. Finally, when the ISR is completely depleted, the leader will communicate an error back to the initiating publisher, which in turn, may reattempt to publish the record.

> Comparing this model to other distributed consensus protocols, such as ZAB, Raft, and Paxos: one no longer requires majority agreement among the cohorts before accepting a log write and thereby deeming it stable. Provided that there is consensus on the membership of the ISR, and all members of the ISR are synchronised with the leader, then the ISR does not need to constitute the majority of the replicas. Conceivably, we could have 100 replicas, with only two in the ISR, and still achieve consensus.
>
> At first glance, it may appear that Kafka's replication protocol has accomplished a feat that eluded distributed systems researches for several decades. This is not the case. The trick is in the consensus on the ISR membership state, which is backed by ZooKeeper. ZooKeeper's ZAB protocol provides the underlying primitives for Kafka to build upon. So although Kafka does not require a majority vote for log replication, it does require a majority vote for making updates to the ISR state. Putting

> it another way, Kafka layers its replication protocol on top of ZAB, forming a hybrid of the two.

The default value of the `min.insync.replicas` property is `1`, which implies that a durable write only extends to the leader replica, which is hardly durable. At minimum, `min.insync.replicas` should be set to `2`, ensuring that a write is reflected on the leader as well as at least one follower. This property can be set for all topics, as well as for individual topics. Instructions for targetting specific topics have been covered in Chapter 9: Broker Configuration.

The maximum tolerable replication lag is configured via the `replica.lag.time.max.ms` broker property, which defaults to `10000` (ten seconds). If a follower hasn't sent any fetch requests or hasn't consumed up to the leader's log end offset within this time frame, it will be summarily dismissed from the ISR. This setting can be applied to all topics, or selectively to individual topics.

Prior to Kafka 0.8.3.0, the maximum tolerable replication lag was configured via the `replica.lag.max.messages` property. The replication protocol used to consider the number of records that a follower was trailing by, to determine whether it should be in the ISR. Replicas could easily be knocked out of the ISR during sudden bursts of traffic, only to rejoin shortly afterwards. Conversely, low-volume topics would take a long time to detect an out-of-sync replica. As part of KIP-16[27], the protocol has since evolved to only consider the time delay between the latest record on the leader and that of each follower. This made it easier to tune the protocol, as it was less susceptible to flutter during traffic bursts. It also made it easier to set meaningful values, as it is more natural to think of lag in terms of time, rather than in terms of arbitrary records.

# Leader election

Only members of the ISR are eligible for leader election. Recall, Kafka's replication protocol is generally asynchronous and a replica in the ISR is not guaranteed to have all records that were written by the outgoing leader, only those records that were confirmed by the leader as having been durably persisted. In other words, the protocol is synchronous only with respect to the durable writes, but not necessarily all writes. It is conceivable then, that some in-sync replicas will have more records than others, while preserving sequential consistency with the leader. When selecting the new leader, Kafka will favour the follower with the highest log end offset, recovering as much of the unacknowledged data as possible.

Kafka's guarantee with respect to durability is predicated on at least one fully-synchronised replica remaining intact. Remember, an in-sync replica will contain all durable writes; the `min.insync.replicas` states the minimum number of replicas that will be fully-synchronised at any given time. When

[27] https://cwiki.apache.org/confluence/display/KAFKA/KIP-16+-+Automated+Replica+Lag+Tuning

suitably sized and appropriately configured, a cluster should tolerate the failure of a bounded number of replicas. But what happens when we've gone over that number?

At this point, Kafka essentially provides two options: either wait until an in-sync replica is restored or perform *unclean leader election*. The latter is enabled by setting `unclean.leader.election.enable` to `true` (it is `false` by default). Unclean leader election allows replicas that were not in the ISR at the time of failure to take over partition leadership, trading consistency for availability.

To maintain consistency in the face of multiple replica failures, one should set the replication factor higher than the minimum recommended value of three, and boost the `min.insync.replicas` value accordingly. This increases the likelihood of a surviving replica being fully-synchronised with the leader. One could further boost `min.insync.replicas` to equate to the replication factor, thereby ensuring that every replica is fit to act as a leader. The downside of this approach is the loss of availability in the write aspect: should a replica fail, consistency will be preserved, but no further writes to the partition will be allowed for having insufficient replicas in the ISR set. It is also less performant, negating the main purpose of an ISR — to reduce the performance impact of slow replicas.

To be clear, the consistency-availability tradeoffs alluded to above are not unique to Kafka's replication protocol, affecting every distributed consensus protocol. Given a constant number of replicas, one can increase the number of replicas that must agree on a write, thereby increasing the likelihood that at least one complete replica survives — but in doing so, impede the system's ability to make progress with a reduced replica set. Conversely, reducing the number of voting replicas will allow the system to tolerate a greater number of replica failures, but increases the likelihood of losing all complete replicas — leaving just the partial ones, or none at all. The only way to solve both problems simultaneously is to increase the replication factor, which increases the cost of the setup and impacts the performance of durable writes by requiring more acknowledgements. Whichever the approach, it has its drawbacks.

## Setting the initial replication factor

The replication factor can be initially assigned when creating a topic using the Kafka Admin API. Alternatively, it can be set via the `--replication-factor` flag in the `kafka-topics.sh` CLI tool. Kafdrop also lets you set the replication factor when creating a topic. If unspecified, the replication factor for a newly created topic is sourced from the `default.replication.factor` broker configuration property, which is `1` by default.

Attempting to create a topic with a replication factor greater than the size of the cluster results in a `org.apache.kafka.common.errors.InvalidReplicationFactorException`.

# Changing the replication factor

Once a replication factor has been set, either the Admin API or the CLI tool can be used to subsequently alter the replication factor. As it happens, this is not as straightforward as setting it initially; changing the replication factor is a semi-manual operation, which entails specifying a new set of replicas for each partition.

We need access to a multi-broker cluster, with at least two brokers, to practice changing the replication factor. If you happen to have a multi-broker cluster at your disposal — perfect. Otherwise, don't fret: you can easily spin one up using the following Docker Compose file. Remember to shut down any existing Kafka, ZooKeeper, and Kafdrop instances before spinning up a Compose stack, as it will conflict on port assignments.

```
version: "3.2"
services:
  zookeeper:
    image: bitnami/zookeeper:3
    ports:
      - 2181:2181
    environment:
      ALLOW_ANONYMOUS_LOGIN: "yes"
  kafka-0:
    image: bitnami/kafka:2
    ports:
      - 9092:9092
    environment:
      KAFKA_CFG_ZOOKEEPER_CONNECT: zookeeper:2181
      ALLOW_PLAINTEXT_LISTENER: "yes"
      KAFKA_LISTENERS: >-
          INTERNAL://:29092,EXTERNAL://:9092
      KAFKA_ADVERTISED_LISTENERS: >-
          INTERNAL://kafka-0:29092,EXTERNAL://localhost:9092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: >-
          INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: "INTERNAL"
    depends_on:
      - zookeeper
  kafka-1:
    image: bitnami/kafka:2
    ports:
      - 9093:9093
    environment:
      KAFKA_CFG_ZOOKEEPER_CONNECT: zookeeper:2181
```

```
      ALLOW_PLAINTEXT_LISTENER: "yes"
      KAFKA_LISTENERS: >-
          INTERNAL://:29092,EXTERNAL://:9093
      KAFKA_ADVERTISED_LISTENERS: >-
          INTERNAL://kafka-1:29092,EXTERNAL://localhost:9093
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: >-
          INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: "INTERNAL"
    depends_on:
      - zookeeper
  kafka-2:
    image: bitnami/kafka:2
    ports:
      - 9094:9094
    environment:
      KAFKA_CFG_ZOOKEEPER_CONNECT: zookeeper:2181
      ALLOW_PLAINTEXT_LISTENER: "yes"
      KAFKA_LISTENERS: >-
          INTERNAL://:29092,EXTERNAL://:9094
      KAFKA_ADVERTISED_LISTENERS: >-
          INTERNAL://kafka-2:29092,EXTERNAL://localhost:9094
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: >-
          INTERNAL:PLAINTEXT,EXTERNAL:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: "INTERNAL"
    depends_on:
      - zookeeper
  kafdrop:
    image: obsidiandynamics/kafdrop:latest
    ports:
      - 9000:9000
    environment:
      KAFKA_BROKERCONNECT: >-
          kafka-0:29092,kafka-1:29092,kafka-2:29092
    depends_on:
      - kafka-0
      - kafka-1
      - kafka-2
```

Launch the stack with `docker-compose up`. This will take a few seconds to start, spinning up a single-node ZooKeeper ensemble with three Kafka brokers attached. The brokers will be externally bound to ports `9092`, `9093`, and `9094`; therefore, we must adjust our bootstrap list accordingly. Kafdrop is also bundled — exposed on port `9000`. Three brokers are a slight overkill for this example, but you may find it useful for other exercises.

First, we will create a test topic named `growth-plan`, with two partitions and a replication factor of one. Key in the command below.

```
$KAFKA_HOME/bin/kafka-topics.sh \
    --bootstrap-server localhost:9092,localhost:9093,localhost:9094 \
    --create --topic growth-plan --partitions 2 \
    --replication-factor 1
```

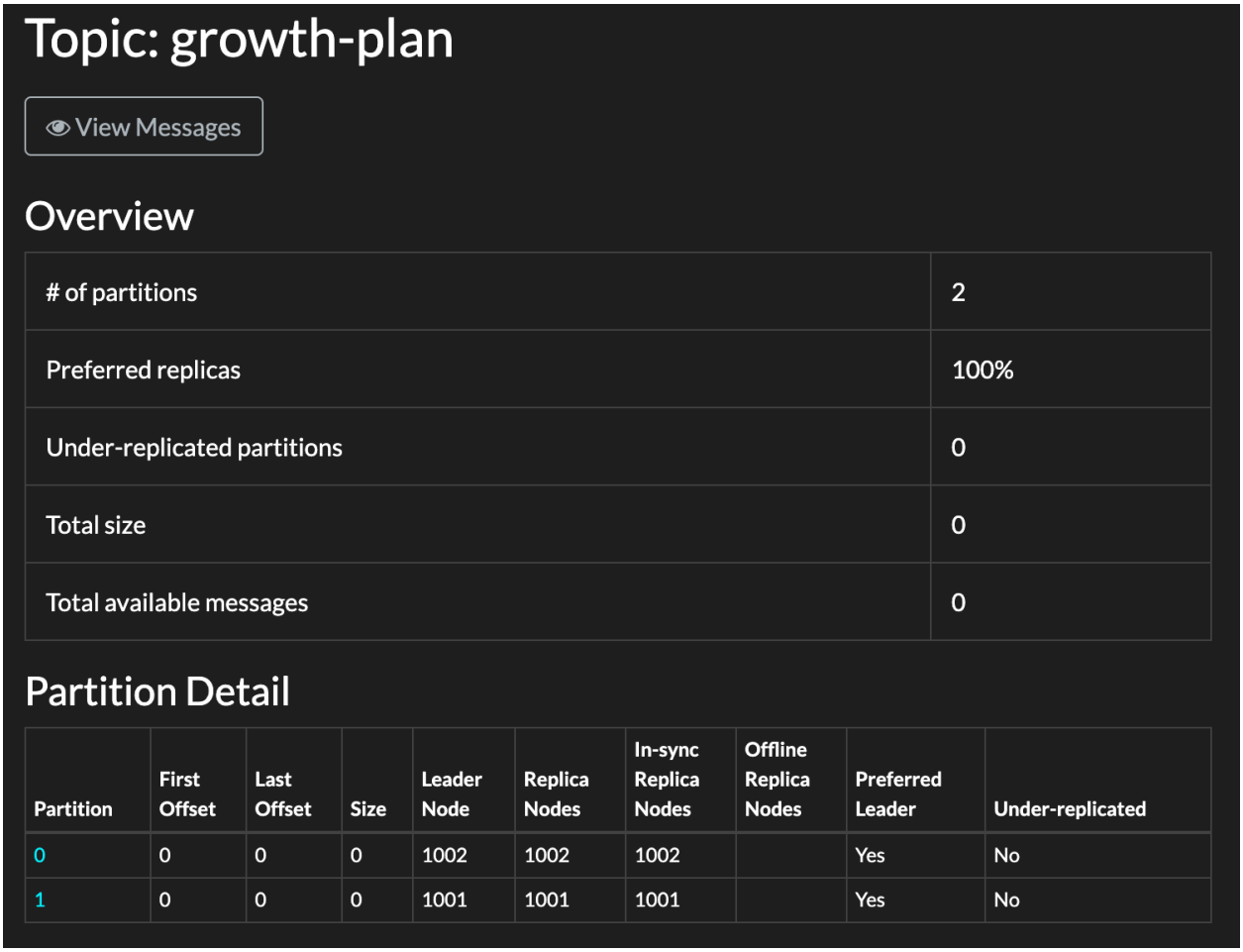
Having created the topic, examine it by running the `kafka-topics.sh` CLI tool:

```
$KAFKA_HOME/bin/kafka-topics.sh \
    --bootstrap-server localhost:9092,localhost:9093,localhost:9094 \
    --describe --topic growth-plan
```

The output indicates that there are two partitions, with their leadership assigned to two different brokers. As expected, the replication factor is one. (The assignment of replicas is random, so your output may vary from the one below.)

```
Topic: growth-plan  PartitionCount: 2 ReplicationFactor: 1 □
    Configs: segment.bytes=1073741824
  Topic: growth-plan  Partition: 0  Leader: 1002  Replicas: 1002 □
      Isr: 1002
  Topic: growth-plan  Partition: 1  Leader: 1001  Replicas: 1001 □
      Isr: 1001
```

Looking at Kafdrop, we can see a similar picture:




**Kafdrop showing a topic with one replica**

Let's increase the replication factor from one to two. We would like to keep the partitions balanced, letting the brokers alternate in the leader-follower status for each partition. Ideally, we don't want to disrupt any existing assignments, or change the leader status of any replicas. Broker `1002` should be the leader for partition `0`, and follower for partition `1`. Conversely, broker `1001` should lead partition `1` and follow partition `0`.

Having formed a plan in our head, it is time to capture it in a *reassignment file*. Create a file named `alter-replicas.json`, containing the following:

```
{
  "version": 1,
  "partitions": [
    {
      "topic": "growth-plan",
      "partition": 0,
      "replicas": [1002, 1001]
    },
    {
      "topic": "growth-plan",
      "partition": 1,
      "replicas": [1001, 1002]
    }
  ]
}
```

The reassignment file enumerates over all topic-partitions that require alteration, listing the new replicas as an array of numeric broker IDs. By convention, the first element in the array identifies the preferred leader.

Next, apply the reassignment file using the `kafka-reassign-partitions.sh` tool.

```
$KAFKA_HOME/bin/kafka-reassign-partitions.sh \
    --zookeeper localhost:2181 \
    --reassignment-json-file alter-replicas.json --execute
```

The `kafka-reassign-partitions.sh` tool is limited to working with ZooKeeper. It cannot be used without the `--zookeeper` flag.

The application of replica changes will echo the original assignments to the console as a JSON document. This can be saved as a JSON file in case it becomes necessary to revert to the original configuration.

```
Current partition replica assignment

{"version":1,"partitions":[{"topic":"growth-plan","partition":1, ⏋
"replicas":[1001],"log_dirs":["any"]},{"topic":"growth-plan", ⏋
"partition":0,"replicas":[1002],"log_dirs":["any"]}]}

Save this to use as the --reassignment-json-file option during ⏋
rollback
Successfully started reassignment of partitions.
```

Given the lack of any partition data in this simple example, the reassignment should be near-instant. This wouldn't necessarily be the case for a production topic, potentially containing lots of data; the reassignment time would depend on the size of the partitions, the available network bandwidth, and the performance of the brokers. To ascertain whether the replica changes completed successfully, run the `kafka-reassign-partitions.sh` tool with the `--verify` switch:

```
$KAFKA_HOME/bin/kafka-reassign-partitions.sh \
    --zookeeper localhost:2181 \
    --reassignment-json-file alter-replicas.json --verify
```

When all partitions have been successfully reassigned, the output should resemble the following:

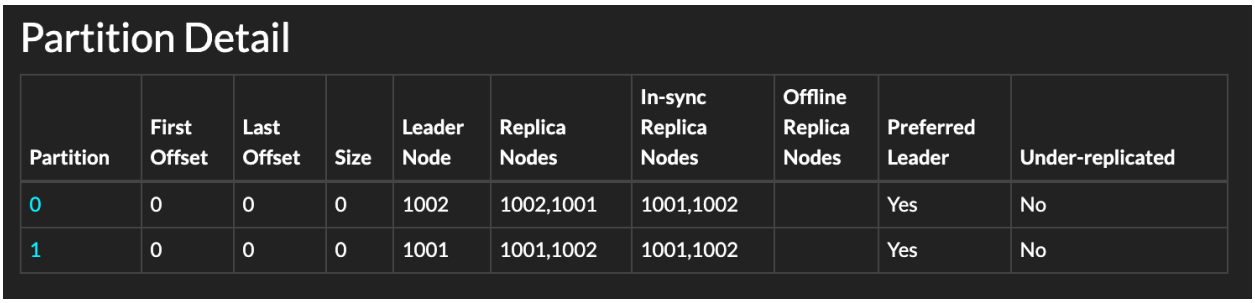
```
Status of partition reassignment:
Reassignment of partition growth-plan-0 completed successfully
Reassignment of partition growth-plan-1 completed successfully
```

Reassignment may involve copying large amounts of data over the network, which may affect the performance of the cluster. To throttle the replication process, run `kafka-reassign-partitions.sh` with the `-throttle` flag, specifying a cap on the desired bandwidth in bytes per second. The throttle will persist even after the initial replication completes, affecting regular replication for the topic in question. To lift the throttle, run the `kafka-reassign-partitions.sh` command with the `-verify` switch. The throttle will be cleared only if all partitions have had their reassignments completed successfully; otherwise, the throttle will remain in force.

Once reassignment completes, run the `kafka-topics.sh` command again. The output should now resemble the following.

```
Topic: growth-plan  PartitionCount: 2 ReplicationFactor: 2 □
    Configs: segment.bytes=1073741824
  Topic: growth-plan  Partition: 0  Leader: 1002 □
      Replicas: 1002,1001 Isr: 1002,1001
  Topic: growth-plan  Partition: 1  Leader: 1001 □
      Replicas: 1001,1002 Isr: 1001,1002
```

Similarly, Kafdrop will also reflect the new replica nodes.



Partition Detail

| Partition | First Offset | Last Offset | Size | Leader Node | Replica Nodes | In-sync Replica Nodes | Offline Replica Nodes | Preferred Leader | Under-replicated |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1002 | 1002,1001 | 1001,1002 | | Yes | No |
| 1 | 0 | 0 | 0 | 1001 | 1001,1002 | 1001,1002 | | Yes | No |

**Kafdrop showing a topic with two replicas**

Preparing reassignment files by hand can be laborious. There are several open-source helper scripts that automate this process. One such script is the `kafka-reassign-tool`, hosted on GitHub at github.com/dimas/kafka-reassign-tool[28].

# Decommissioning broker nodes

The built-in `kafka-reassign-partitions.sh` tool might appear overly low-level and cumbersome for simple use cases such as adjusting the replication factor. However, there is a reason for that: this tool is designed to administer arbitrary changes to the replication topology. One such use case is for rebalancing partitions among broker nodes, levelling the load when new nodes are added to the cluster. Another use case is for moving partitions off a broker node before it can be safely decommissioned

Having selected a broker for decommissioning, create a reassignment file that covers all partitions for which the outgoing broker is a replica. Substitute the outgoing broker ID with one of the remaining brokers, ensuring that the original replication factor is preserved (unless you wish to change the replication factor in the process). Once preparations are complete, run `kafka-reassign-partitions.sh` with the `-execute` switch, then follow up with the `-verify` switch to ensure that the process completes. Having reassigned the partitions, run the `kafka-topics.sh` command with the `-describe` switch without specifying a topic name, to verify that the outgoing broker no longer features in any replicas, across any of the existing topics. Only after the broker has been completely fenced off, can it be permanently removed from the cluster.

[28] https://github.com/dimas/kafka-reassign-tool

# Acknowledgements

The `acks` property stipulates the number of acknowledgements the producer requires the leader to have received before considering a request complete, and before acknowledging the write with the producer. This is fundamental to the durability of records; a misconfigured `acks` property may result in the loss of data while the producer naively assumes that a record has been stably persisted.

Although the property relates to the number of acknowledgements, it accepts an enumerated constant being one of —

- `0`: Don't require an acknowledgement from the leader.
- `1`: Require one acknowledgement from the leader, being the persistence of the record to its local log. This is the default setting when `enable.idempotence` is set to `false`.
- `-1` or `all`: Require the leader to receive acknowledgements from all in-sync replicas. This is the default setting when `enable.idempotence` is set to `true`.

Each of these modes is discussed in detail in the following subsections.

## No acknowledgements

When `acks=0`, any records queued on the outgoing socket are assumed to have been published, with no explicit acknowledgements required from the leader. This option provides the weakest durability guarantee. Not only is it impacted by a trivial loss of a broker, but the availability of the client is also a determinant in the durability of the record. If the client were to fail, any queued records would be lost.

In addition to forfeiting any sort of durability guarantees, the client will not be informed of the offset of the published record. Normally, a client would obtain the offset in one of two different ways: supplying a `org.apache.kafka.clients.producer.Callback` to the `Producer.send()` method, or blocking on the result of a `Future<RecordMetadata>` object that is returned from `send()`. In either case, the resulting `RecordMetadata` would contain the offset of the published record. However, when `acks=0`, `RecordMetadata.hasOffset()` will return `false`, and `RecordMetadata.offset()` will return `-1`, signifying that no offset is available.

The main use case for `acks=0` is that the publisher can afford to lose either a one-off record or small contiguous batches of records with a negligible impact on the correctness of the overall system. Consider a remote temperature sensor that is periodically submitting telemetry readings to a server, which in turn, is publishing these to a Kafka topic. A downstream process will ingest these readings and populate a web-based dashboard in real-time.

In this hypothetical scenario, and provided that historical temperature readings are not significant, we can afford to lose some number of intermediate readings, provided that a reading for every sensor eventually comes through. This is not to say that a higher durability rating would somehow

be inapplicable, only that it may not be strictly necessary. Furthermore, by neglecting acknowledgements, we are also largely sidestepping the matters of error handling. This leads to a simpler, more maintainable application, albeit one that offers virtually no delivery guarantees.

Setting `acks=0` in no way implies that records will not be replicated by the leader to the in-sync replicas. Standard asynchronous replication behaviour applies identically in all cases, irrespective of the number of requested acknowledgements, provided that the record actually arrives at the broker. You can think of the `acks` property as the assurance level of a notional contract between three parties — the client, the partition leader, and the follower replicas. The contract is always present, with each party having the best intentions, but nonetheless, the contract may not be fulfilled if one of the parties were to fail. At its most relaxed level — `acks=0` — the extent of the assurance is constrained to the client. With each increase in the value of `acks`, the producer expands the scope of assurance to include the next party in the list. With `acks=all`, the producer will not be satisfied until all parties have fulfilled their contractual obligations.

## One acknowledgement

When `acks=1`, the client will wait until the partition reader queues the write to its local log. The broker will asynchronously forward the record to all follower replicas and may respond to the publisher before it receives acknowledgements from all in-sync replicas.

On the face of it, this setting may appear to offer a significantly stronger durability guarantee over `acks=0`. If in the `acks=0` case the system was intolerant of the failure of a single party, in the `acks=1` case the failure of the partition leader should be tolerated, provided the client remains online. Should the leader fail, the producer can forward its request to another replica, whichever one takes the place of the leader.

In reality, this stance is ill-advised for several reasons. One is to do with how Kafka writes records to its log. Records are considered written when they are handed over to the underlying file system buffers. However, the broker does not invoke the `fsync` operation for each written record, in other words, it does not wait for the write to be flushed before replying to the client. Therefore, it is possible for the broker to acknowledge the write and fail immediately thereafter, losing the written chunk before it has been replicated onto the followers. The producer will continue on the assumption that the write has been acknowledged.

The second reason is a close relative of the first. Assume for the moment that the write was successfully committed to the leader's disk, but the leader suffered a failure before the record was received by any of the followers. The controller node, having identified the failure, will appoint a new leader from the remaining in-sync replicas. There is no guarantee that the new leader will have received the last few records that were queued for replication by the outgoing leader.

This is not to say that `acks=1` offers no additional durability benefits whatsoever, rather, the distinction between `acks=0` and `acks=1` is only discernible if there is a material difference between

the relative reliability of the client and broker nodes. And sometimes this is indeed the case: brokers may be assembled using higher-grade hardware components with more integrated redundancy, making them less failure-prone. Conversely, clients may be running on ephemeral instances that are susceptible to termination at short notice. But taking this for granted would be credulously presumptuous; it is prudent to treat a broker like any other component, assuming that if something can fail, it probably will, and when we least expect it to.

With the above in mind, set `acks=1` in those scenarios where the client can trivially afford to lose a published record, but still requires the offset of the *possibly* published record — either for its internal housekeeping or for responding to an upstream application.

## All acknowledgements

When `acks=all` or `acks=-1` (the two are synonymous), the client will wait until the partition leader has gathered acknowledgements from the complete set of in-sync replicas. This is the highest guarantee on offer, ensuring that the record is durably persisted for as long as one in-sync replica remains. As it was stated earlier, the number of in-sync replicas may vary from the total number of replicas; the in-sync replicas are represented by a set that may grow and shrink dynamically depending on the observed replication lag between the leader and each follower. Since the latency of a publish operation when `acks=all` is attributable to the slowest replica in the ISR, the in-sync replica set is designed to minimise this latency by temporarily dismissing those replicas that are experiencing starvation due to I/O bottlenecks or other performance-impacting contingencies.

There is a pair of subtle configuration gotchas lurking in Kafka: one being on the client while the other on the broker. The default setting of `acks` is `1` when idempotence is disabled. (Idempotence is discussed in Chapter 10: Client Configuration.) This is unlikely to meet the durability expectations of most applications. While the rationale behind this decision is undocumented, the choice of `acks=1` may be related to the disclosure of `RecordMetadata` to the producer. With `acks=1`, the producer will be informed of the offset of the published record, whereas with `acks=0` the producer remains unaware of this. Kafka's own documentation states: "When used together, `min.insync.replicas` and `acks` allow you to enforce greater durability guarantees. A typical scenario would be to create a topic with a replication factor of `3`, set `min.insync.replicas` to `2`, and produce with `acks` of `all`." In spite of recognising `acks=all` as being necessary to fulfill a 'typical' scenario, the default value of `acks` remains `1`. The second 'gotcha': the default value of `min.insync.replicas` is `1`, again contradicting the 'typical' scenario.

Set `acks=all` in those scenarios where the correctness of the system is predicated on the consensus between the producer and the Kafka cluster as to whether a record has been published. In other words, if the client thinks that the record has been published, then it better be, or the integrity of the system may be compromised. This is often the case for transactional business applications where recorded events correlate to financial activity; the loss of a record may directly or indirectly incur financial losses.

The main drawback of `acks=all` is the increased latency as a result of several blocking network calls — between the producer and the leader, and between the leader and a subset of its followers. This is the price of durability.

## Which setting is right for me?

The explanations above have clarified the behaviour of the producer client and the partition leader with respect to the value of the `acks` property. The reader should now have a better appreciation of the scenarios that may be fitting to the different acknowledgement modes.

If in doubt, set `acks` to `all`. This will ensure that a record is acknowledged by a number of brokers that is, *at minimum*, equal to the value of `min.insync.replicas`. Bear in mind that 'acknowledged' is not the same as 'stably persisted'. Failure of a broker immediately following an acknowledgement may result in the loss of data, therefore it is essential that the value of `min.insync.replicas` accurately reflects your tolerance for data loss. Setting `min.insync.replicas` to `2` on a topic with a replication factor of three, and using `acks=all` on the producer ensures that every published record is acknowledged by the leader and *at least* one other replica. The increased latency might be noticeable, but it may be preferable to data loss. One might even argue that if latency is the overarching concern that dominates technical decision-making, then Kafka is perhaps not the best solution for the problem at hand.

---

This chapter explored the innards of Kafka's internal replication protocol. We learned how Kafka utilises in-sync replicas to reduce the number of acknowledgements required for durable persistence, efficiently achieving distributed consensus.

We looked at how partition availability and data consistency concerns may be impacted by the replication setup, namely the inherent availability-consistency trade-offs present in deciding on the replication factor and the `min.insync.replicas` parameter. This ties into Kafka's leader election — the decisions made when promoting follower replicas in the event of leader failure.

Configuration of the replication topology was then explored. We worked through examples of using the built-in `kafka-reassign-partitions.sh` tool to affect fine-grained control over a partition's replicas, and its potential use cases outside of simple replication factor adjustments. The performance trade-offs of durable persistence were covered, and we delved into the idiosyncrasies of Kafka's default configuration settings, and how they may catch out unsuspecting users.

Finally, durability was explored from the eyes of the producer. We looked at how the `acks` producer setting impacts the durability guarantees of individual records, and its interplay with the replication factor and the `min.insync.replicas` broker-side settings.