KIP-928: Making Kafka resilient to log directories becoming full

This page is meant as a template for writing a KIP. To create a KIP choose Tools->Copy on this page and modify with your content and replace the heading with the next KIP number and a description of your issue. Replace anything in italics with your own description.

Status

Current state: *"Under Discussion"*

Discussion thread: here

JIRA: *here*

Sample implementation: *here*

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

JBOD is coming to KRaft in KIP-858: Handle JBOD broker disk failure in KRaft**!** It has been a feature in Kafka since KIP-112: Handle disk failure for JBOD. The evolution of how disk failures are handled in KIP-858 is more about the notification and persistence mechanisms rather than the sequence of steps undertaken by the controller and broker to prevent future interactions with the affected disk. **What we propose in this KIP is to treat a subset of a disk failure - a disk becoming full - in such a way to allow space to be cleared up using Kafka functionality such as modifying retention periods or deleting problematic topics.** The examples in this KIP use a Zookeeper-backed Kafka cluster, but we believe the functionality will be easily implementable once KIP-858 is code-complete.

When a broker's log directory becomes full Kafka is terminated forcefully (source: Exit.halt(1) eventually calls Runtime::halt which does not cause shutdown hooks to be started). When a broker stops data cannot be produced and consumed, and administrative operations cannot be carried out on the cluster via the broker. Assuming partitions are evenly distributed across the cluster and have balanced traffic such a saturation can occur across multiple brokers simultaneously. Expanding the log directory and restarting the broker can bring it back up. Such an expansion, however, is not always feasible in cloud deployments, for example with a standard block size of 4KB an AWS EBS volume can support only up to 16TB (source). There are other ways to alleviate the problem as well (manually delete data outside of Kafka; try to split a log directory across multiple disks; attach additional disks) but all of them require intervention outside of Kafka. Tiered Storage (KIP-405: Kafka Tiered Storage) is expected to alleviate some of these problems, but it still requires expertise to ensure that the rate of archival from local to remote storage is higher or equal to the rate at which data is written to local storage. As such, from the point of view of a Kafka customer it is important to provide a solution to free up disk space without requiring external intervention.

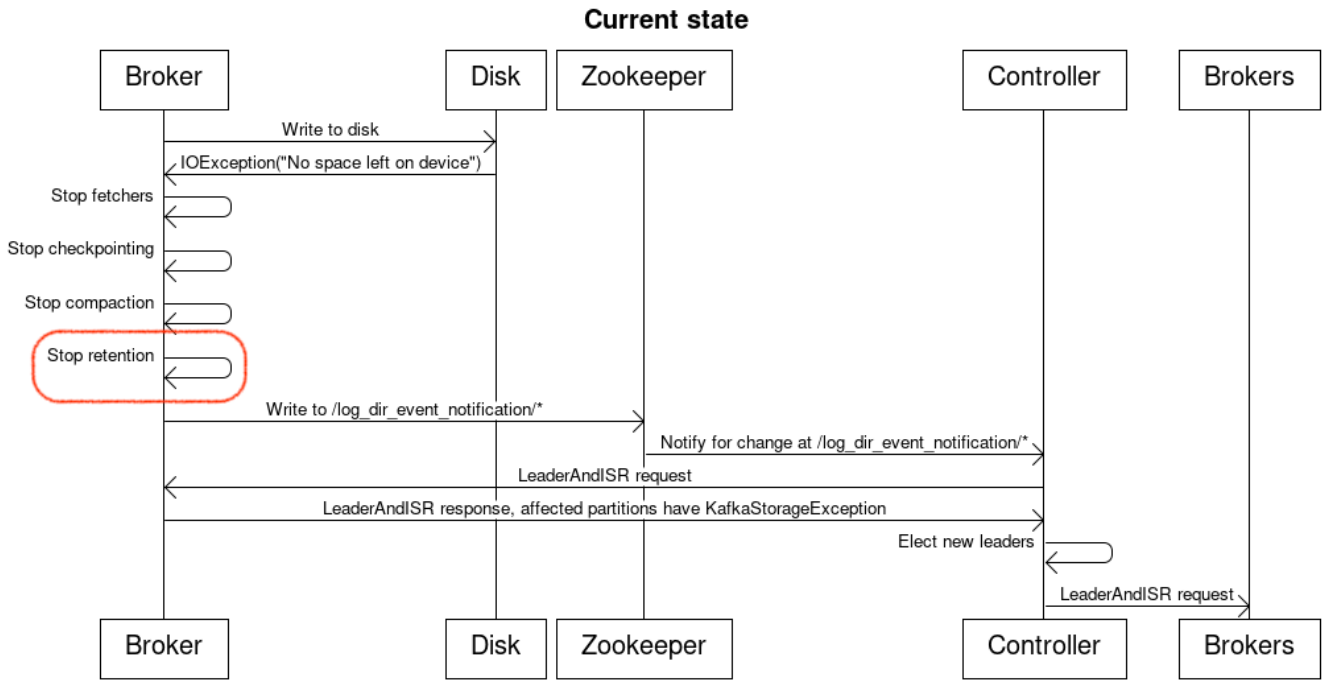
Public Interfaces

Briefly list any new interfaces that will be introduced as part of this proposal or any existing interfaces that will be removed or changed. The purpose of this section is to concisely call out the public contract that will come along with this feature.

No foreseen changes to public-facing interfaces.

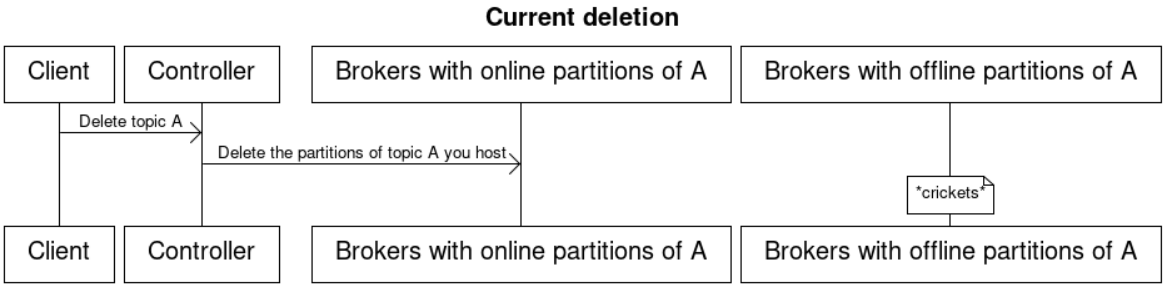
Proposed Changes

Current state



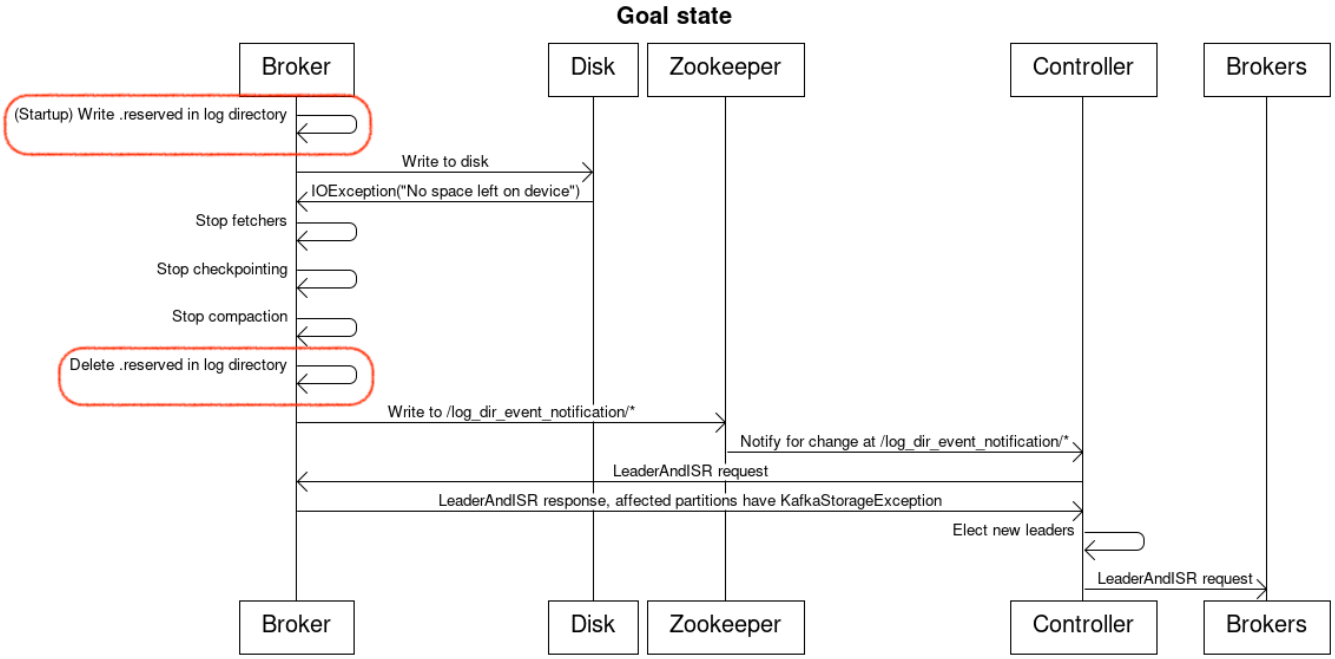
A Kafka broker has a big try-catch statement around all interactions with a log directory. If an IOException is raised the broker will stop all operations on logs located in that log directory, remove all fetchers, stop checkpointing, prevent compaction and retention from taking place and write to a node in Zookeeper.

The Kafka controller will get notified by Zookeeper that there was a problem with a log directory on a particular broker. The controller will then reach out to the broker to understand the state of partition replicas. The broker responds with which partition replicas are offline due to a log directory going offline. The controller determines the new leaders of said partitions and issues new leader and in-sync replicas requests.



The controller keeps track of which replicas of partitions are offline and does not forward delete topic requests to the brokers hosting said replicas.

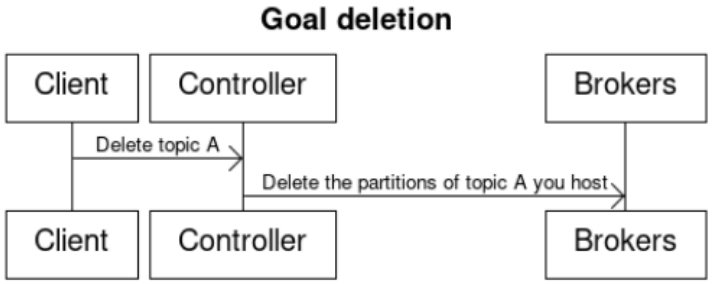
Goal state



A Kafka broker still has a big try-catch statement around all interactions with a log directory. If an IOException due to **No space left on device** is raised (we will check the remaining space at that point in time rather than the exception message) the broker will stop all operations on logs located in that directory, remove all fetchers and stop compaction. **Retention will continue to be respected.** The same node as the current state will be written to in Zookeeper. All other IOExceptions will continue to be treated the same way they are treated now and will result in a log directory going offline.

The Kafka controller will get notified by Zookeeper that there was a problem with a log directory on a particular broker. The controller will then reachout to the broker to understand the state of partition replicas. The broker responds with which partition replicas are offline due to a log directory becoming saturated. The controller determines the new leaders of said partitions and issues new leader and in-sync replicas requests.

In addition to the above, upon Kafka server startup we will write and flush to disk a 40MB (from tests: ~10KB per partition for approximately 4000 partitions) file with random bytes to each log directory which Kafka will delete whenever a broker goes into a saturated state. Since the file will be written before any of the log recovery processes are started if there are any problems Kafka will shut down. This reserved space is a space-of-last-resort for any admin operations requiring disk to run while the broker is in a saturated state. For example, if all segments of a partition are marked for deletion Kafka rolls a new segment before deleting any old ones. If we do not have some space put aside for such operations then we will have to change their ordering.

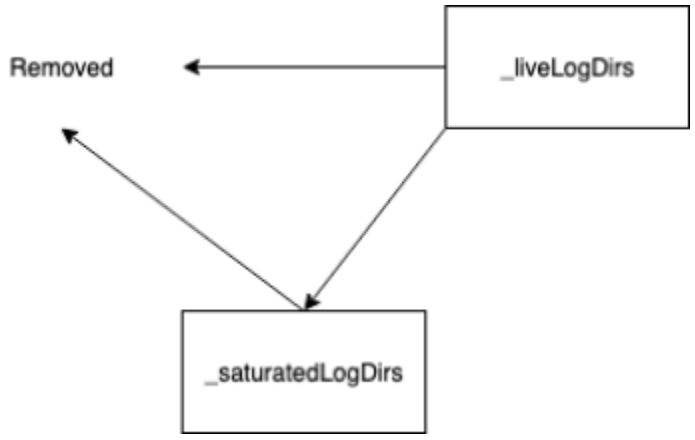


The controller will forward delete topic requests to all brokers hosting replicas of partitions of the topic under deletion. Requests targeting saturated log directories will be respected and will succeed. Requests targeting offline log directories will fail, which is expected.

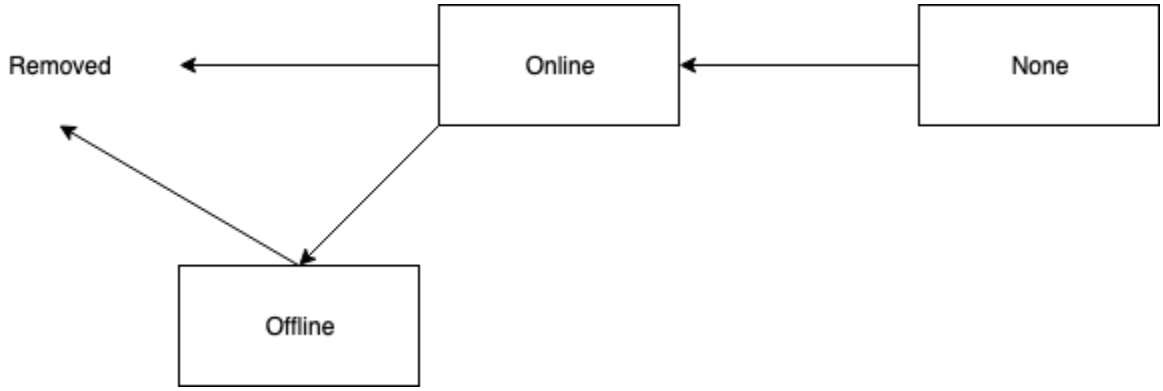
There are three state machines related to this change, two on the broker (log directory and partition state) and one on the controller (partition state) and we will be modifying only the state machines of the broker. **Both the broker and the controller have a state machine for a partition state, but these are two completely separate state machines which just share a name**. **There are more states in a controller's partition state machine and they need to be more in order to choose a leader correctly. A broker's partition state machine is simplified because it only needs to know whether it hosts the partition or not.** Whenever a broker detects a problem with a log directory it carries out contingency measures and writes a node to Zookeeper at location `/log_dir_event_notification` with data of the format `{"version" : 1, "broker" : brokerId, "event" : LogDirFailure}`. The controller watches for changes at that location and when notified sends a LeaderAndISR request to the affected broker. The broker gathers the state of its partition replicas and responds to the controller with which partition replicas should be considered impaired. Leadership election is triggered for every partition, which has an impaired leader replica on said broker.

Current Broker Log Directory State Machine

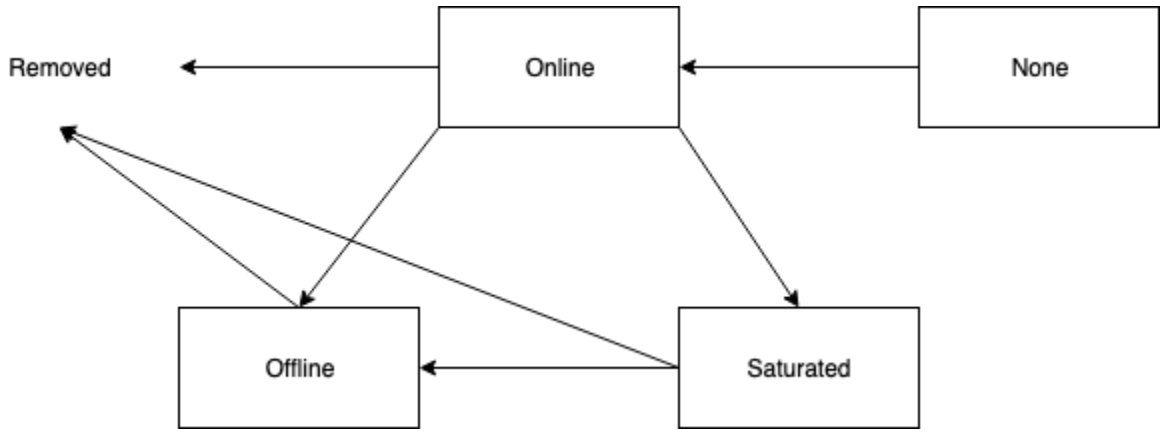
Target Broker Log Directory State Machine



Current Broker Partition State Machine



Target Broker Partition State Machine



We will add a new state to the broker state machines of a log directory (saturated) and a partition replica (saturated). The partition state machine is only known to the broker and it won't be replicated on the controller. We need these additional states in order to restrict which background tasks operate on them. If we do not have a separate state then we have no way to tell Kafka that we would like deletion and retention to continue working on saturated log directories and partitions.

Notification mechanism - Zookeeper/KRaft Controller
No changes will be introduced to Zookeeper. We continue to use it only as a notification mechanism.

Controller
Instead of sending a delete topic request only to replicas we know to be online, we will allow a delete topic request to be sent to all replicas regardless of their state. Previously a controller did not send delete topic requests to brokers because it knew they would fail. In the future, topic deletions for saturated topics will succeed, but topic deletions for the offline scenario will continue to fail.

Limitations

Disk space won't be reclaimable via compaction because compaction will stop working. Compaction won't work because behind the scenes it creates a new file, populates it with the latest records and then deletes the old file. If there is no space left we cannot create a new file.

Disk space won't be reclaimable via DeleteRecords. DeleteRecords just moves the pointer named LogStartOffset which renders earlier records unreadable, but it doesn't go out of its way to delete them. Deletion of older segments still happens via the normal retention procedure.

Space may be reclaimable via moving partitions from affected brokers to other brokers. Behind the scenes moving partitions creates one more follower on a different broker, waits for it to catch up and then deletes one of the original followers. In the case where only some of the brokers in a cluster have experienced a disk full moving partitions will be possible. If all partition replicas are on brokers with filled up disks moving partitions won't be possible as the partition will be offline and no consumption can happen.

(KIP-405: Kafka Tiered Storage specific) Space can be reclaimed for tiered topics by deleting the topic, and depending on the situation by more aggressive retention. Archival in Tiered Storage is carried out by a partition's leader. A non-active segment is not deleted before it is copied to remote storage. If all replicas are on brokers with disks which are full then the partition will be offline and will lack a leader. If there is no leader the partition cannot be archived. If there are non-active segments which aren't archived then retention won't work on them. In such a situation only deleting the topic will free up space. However, as long as there is a leader replica on a broker with empty space on the disk then a more aggressive retention should free up space.

Compatibility, Deprecation, and Migration Plan

- *What impact (if any) will there be on existing users?* Currently users cannot interact with a Kafka broker at all when its log directories become full. In the future, users will be able to reclaim space by deleting topics or modifying retention settings to be more aggressive.
- *If we are changing behavior how will we phase out the older behavior?* We won't phase out the old behaviour as we are building on top of it.
- *If we need special migration tools, describe them here.* N/A
- *When will we remove the existing behavior?* N/A

Test Plan

Describe in few sentences how the KIP will be tested. We are mostly interested in system tests (since unit-tests are specific to implementation details). How will we know that the implementation works as expected? How will we know nothing broke?

We will implement new integration and system tests which artificially constraint the space available to the Kafka log directories in order to gain confidence in the behaviour of the system.

Rejected Alternatives

If there are alternative ways of accomplishing the same thing, what were they? The purpose of this section is to motivate why the design is the way it is and not some other way.

Alternative approaches

- *This is a reactive proposal, a proactive approach can help us not get into this situation in the first place. In our opinion a proactive approach is a reactive approach with a boundary which is moved (i.e. instead of putting a broker in a saturated state when it reaches 100% usage of a log directory and we start getting IOExceptions we will put the broker in a saturated state when it reaches X% and a background thread checks that condition). The problem lies not with where the boundary is, but with defining the behaviour at that boundary.* ***One of the limitations which will be alleviated by a reactive approach is to allow compaction to continue working (since there will be space for new segments) - in our experience, however, compacted topics are rarely the cause of disks becoming full.***

Alternative details to this approach

- *Do not reserve space-of-last-resort, but turn operations which use a create-delete pattern (such as the rolling of a segment) turn into a delete-create pattern. In the case of rolling a segment this means that we will first delete old segments (or a subset of them) and then roll a new one. This is not exactly a rejected alternative - ideally we would like to do this. However, we believe that this should be an incremental change.*