KIP-112: Handle disk failure for JBOD

- Status
- Motivation
- Proposed change
  - How to handle log directory failure
- Public interface
  - Zookeeper
  - Protocol
  - Scripts
  - Metrics
- Changes in Operational Procedures
- Compatibility, Deprecation, and Migration Plan
- Test Plan
- Rejected Alternatives
- Potential Future Improvement

Status

Current state: *Completed*

Discussion thread: *here*

JIRA: KAFKA-4763 - Getting issue details... STATUS

Released: <Kafka Version>

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

Motivation

The most expensive part of a Kafka cluster is probably its storage system. At LinkedIn we use RAID-10 for storage and set Kafka's replication factor = 2. This setup requires 4X space to store data and tolerates up to 1 broker failure. We are at risk of data loss with just 1 broker failure, which is not acceptable for e.g. financial data. On the other hand, it is prohibitively expensive to set replication factor = 3 with RAID-10 because it will increase our existing hardware cost and operational cost by 50%.

The solution is to use JBOD and set replication factor = 3 or higher. It is based on the idea that Kafka already has replication across brokers and it is unnecessary to use RAID-10 for replication. Let's say we set the replication factor = 4 with JBOD. This setup requires 4X space to store data and tolerate up to 3 broker failures in order not to lose any data. In comparison to our existing setup, this allows us to obtain 3X broker failure tolerance without increasing storage hardware cost.

We have evaluated the possibility of using other RAID setup in LinkedIn. But none of them addresses our problem as JBOD does. RAID-0 stops working entirely with just one disk failure. RAID-5 or RAID-6 has sizable performance loss as compared to RAID-0 and probably JBOD as well, due to their use of block-level striping with distributed parity.

Unfortunately, JBOD is not recommended for Kafka because some important features are missing. For example, Kafka lacks good support for tools as well as load balancing across disks when multiple disks are used. Here is a list of problems that need to be addressed for JBOD to be useful:

1) Broker will shutdown if any disk fails. This means a single disk failure can bring down the entire broker. Instead, broker should still serve those replicas on the good disks as long as there is good disk available.

2) Kafka doesn't provide the necessary tools for users to manage JBOD. For example, Kafka doesn't provide script to re-assign replicas between disks of the same broker. These tools are needed before we can use JBOD with Kafka.

3) JBOD doesn't by itself balance load across disks as RAID-10 does. This will be a new problem for us to solve in order for JBOD setup to work well. We should have a better solution than round-robin which we are using to select disk to place a new replica. And we should probably figure out how to re-assign replicas across disks of the same broker if we notice load imbalance across disks of a broker.

For ease of discussion, we have separated the design of JBOD support into two different KIPs. This KIP address the first problem. See KIP - Support replicas movement between log directories for our proposal to address the second problem.

Since Kafka configuration and implementation does not expose "disk", we will use log directory and disk interchangably in the rest of the KIP.

Goals

The goal of this KIP is to allow broker to serve replicas on good log directories even if some log directories have failed. This addresses the first problem raised in the motivation section. See KIP - Support replicas movement between log directories to read our proposal of how to address the second problem.

Proposed change

How to handle log directory failure

Problem statement:

Currently LeaderAndIsrRequest is used for two purpose: 1) create a new replica on a broker and 2) switch a replica between leader/follower of the partition. If a broker starts with some replicas unavailable because they are on a bad log directory, it will re-create those replicas on a good log directory when it receives LeaderAndIsrRequest from the controller. This is wrong. To avoid this, controller needs to know whether the replica has been created on the broker and explicitly specify whether broker should create replica in the LeaderAndIsrRequest.

Further, currently a replica is offline if and only if the broker hosting the replica is offline. This logic is no longer applicable with JBOD. Broker needs to explicitly tell controller the list of online replicas on its good log directories so that controller can elect leader from only online replicas. And admin tools should be developed to allow user to query which replicas are online or offline.

Solution:

The idea is for controller to explicitly tell broker whether to create replica or not by specifying a newly-added boolean field "isNewReplica" in the LeaderAndIsrRequest. This field will be true only when a replica is transitioning from the NewReplica state to Online state. Each broker can tell controller whether a replica is online by specifying the error_code per partition in the LeaderAndIsrResponse, which in turn allows the controller to derive the offline replicas per partition and elect leader appropriately.

Here are a few clarification to make our solution easier to understand:
- Broker assumes a log directory to be good after it starts, and mark log directory as bad once there is IOException when broker attempts to access (i.e. read or write) the log directory.
- Broker will be offline if all log directories are bad.
- Broker will stop serving replicas in any bad log directory. New replicas will only be created on good log directory.
- If LeaderAndIsrResponse shows KafkaStorageException for a given replica, controller will consider that replica to be offline, do leader election if the replica is a leader and broadcast UpdateMetadataRequest.
- Broker will remove offline replica from its replica fetcher threads.
- Even if isNewReplica=false and replica is not found on any log directory, broker will still create replica on a good log directory if there is no bad log directory.

In the following we describe how our solution works under eight different scenarios. Some existing steps (e.g. kafka-topics.sh creates znode) are omitted for simplicity.

1. Topic gets created

- The controller sends LeaderAndIsrRequest with isNewReplica=True to the leader and followers.
- The leader and followers create replicas locally and sends LeaderAndIsrReponse to the controller with error=None.
- After receiving LeaderAndIsrResponse from leader and followers of this partition, the controller considers the replica as successfully created if error=None in LeaderAndIsrResponse. Otherwise, the replica is considered offline.

2. A log directory stops working on a broker during runtime

- The controller watches the path `/log_dir_event_notification` for new znode.
- The broker detects offline log directories during runtime.
- The broker takes actions as if it has received StopReplicaRequest for this replica. More specifically, the replica is no longer considered leader and is removed from any replica fetcher thread. (The clients will receive a `UnknownTopicOrPartitionException` at this point)
- The broker notifies the controller by creating a sequential znode under path `/log_dir_event_notification` with data of the format *{"version" : 1, "broker" : brokerId, "event" : LogDirFailure}*.
- The controller reads the znode to get the brokerId and finds that the event type is LogDirFailure.
- The controller deletes the notification znode
- The controller sends LeaderAndIsrRequest to that broker to query the state of all topic partitions on the broker. The LeaderAndIsrResponse from this broker will specify KafkaStorageException for those partitions that are on the bad log directories.
- The controller updates the information of offline replicas in memory and trigger leader election as appropriate.
- The controller removes offline replicas from ISR in the ZK and sends LeaderAndIsrRequest with updated ISR to be used by partition leaders.
- The controller propagates the information of offline replicas to brokers by sending UpdateMetadataRequest.

3. Broker bootstraps with bad log directories

- Broker collects list of replicas found on the good log directories. If there is no good log directory the broker will exit.
- The controller sends LeaderAndIsrRequest for all partitions that should exist on this broker. If a replica transitioning from NewReplica state to Online state, isNewReplica=True in the LeaderAndIsrRequest. Otherwise, isNewReplica=False.
- The broker will specify error=KafkaStorageException for those partitions that are in the LeaderAndIsrRequest with isNewReplica=False but not found on any good log directory. The broker will create replica on a good log directory if the replica is not found on any good log directory and its isNewReplica=True.
- The controller considers a replica on that broker to be offline if its error!=None in the LeaderAndIsrResponse.
- The controller updates the information of offline replicas in memory and triggers leader election as appropriate.
- The controller removes offline replicas from ISR in the ZK and sends LeaderAndIsrRequest with updated ISR to be used by partition leaders.
- The controller propagates the information of offline replicas to brokers by sending UpdateMetadataRequest.

4. The disk (or log directory) gets fixed

- User can either replace a bad disk with good disk, or remove the bad log directory from broker config.
- User restarts broker with only good log directories. Broker can read all log directories specified in its config.
- The controller sends LeaderAndIsrRequest with *isNewReplica=False* to this broker because all replicas have been created on this broker.
- Broker will create replica if not found on any good log directory because it can access all log directories specified in the config.

5. User queries replicas state

- Controller propagates the information of offline replicas to brokers by sending UpdateMetadataRequest.
- Kafka client can send MetadataRequest to any broker to query offline replicas for the specified partitions. MetadataResponse from broker to client will include offline replicas per partition.
- kafka-topics script will display offline replicas when describing a topic partition. The offline replicas is the union of offline replicas on live brokers and replicas on dead brokers. kafka-topics script obtains offline replicas by sending MetadataRequest to any broker.

6. Controller failover

- Controller sends LeaderAndIsrRequest to all brokers. If a replica transitioning from NewReplica state to Online state, isNewReplica=True in the LeaderAndIsrRequest. Otherwise, isNewReplica=False.
- Broker responds with LeaderAndIsrResponse and specifies error for offline replicas in the response
- Controller derives offline replicas from LeaderAndIsrResponse and make leader election among live replicas.

7. Partition reassignment
The procedure for partition reassignment will be almost the same as the current procedure. There are two changes:

- Controller will use both broker liveness and LeaderAndIsrResponse to determine a replica is online or not. StopReplicaRequest will fail if replica is offline.
- Controller will follow the same procedure specified in steps of "Topic gets created" when it creates replica on destination brokers, i.e. it will specify proper value for "isNewReplica" in LeaderAndIsrRequest.

8. Topic deleteion

Topic deletion should delete all replicas of all partitions of this topic and update metadata. If a replica is offline, then we handle it in the same way as if its broker is offline.

Public interface

Zookeeper

1) Store data with the following json format in znode `/log_dir_event_notification/log_dir_event_*`

```
{
  "version" : int,
  "broker" : int,
  "event" : int    <-- We currently use 1 to indicate LogDirFailure event.
}
```

Protocol

Add a isNewReplica field to LeaderAndIsrRequestPartitionState which will be used by LeaderAndIsrRequest

```
LeaderAndIsrRequest => controller_id controller_epoch partition_states live_leaders
  controller_id => int32
  controller_epoch => int32
  partition_states => [LeaderAndIsrRequestPartitionState]
  live_leaders => [LeaderAndIsrRequestLiveLeader]

LeaderAndIsrRequestPartitionState => topic partition controller_epoch leader leader_epoch isr zk_version
replicas
  topic => str
  partition => int32
  controller_epoch => int32
  leader => int32
  leader_epoch => int32
  isr => [int32]
  zk_version => int32
  replicas => [int32]
  is_new_replica => boolean <-- NEW
```

Add a offline_replicas field to UpdateMetadataRequestPartitionState which will be used by UpdateMetadataRequest

```
UpdateMetadataRequest => controller_id controller_epoch partition_states live_brokers
  controller_id => int32
  controller_epoch => int32
  partition_states => [UpdateMetadataRequestPartitionState]
  live_brokers => [UpdateMetadataRequestBroker]

UpdateMetadataRequestPartitionState => topic partition controller_epoch leader leader_epoch isr zk_version
replicas offline_replicas
  topic => string
  partition => int32
  controller_epoch => int32
  leader => int32
  leader_epoch => int32
  isr => [int32]
  zk_version => int32
  replicas => [int32]
  offline_replicas => [int32]  <-- NEW. This includes offline replicas due to both broker failure and disk
failure.
```

Add a offline_replicas field to PartitionMetadata which will be used by MetadataResponse

```
MetadataResponse => brokers cluster_id controller_id topic_metadata
  brokers => [MetadatBroker]
  cluster_id => nullable_str
  controller_id => int32
  topic_metadata => TopicMetadata

TopicMetadata => topic_error_code topic is_internal partition_metadata
  topic_error_code => int16
  topic => str
  is_internal => boolean
  partition_metadata => [PartitionMetadata]

PartitionMetadata => partition_error_code partition_id leader replicas isr offline_replicas
  partition_error_code => int16
  partition_id => int32
  leader => int32
  replicas => [int32]
  isr => [int32]
  offline_replicas => [int32]  <-- NEW. This includes offline replicas due to both broker failure and disk
failure.
```

Scripts

1) When describing a topic, *kafka-topics.sh* will show the offline replicas for each partition.

Metrics

Here are the metrics we need to add as part of this proposal

1) kafka.server:name=OfflineReplicasCount,type=ReplicaManager

The number of offline replicas on a live broker. This is equivalent to the number of TopicParition log on the bad log directories of the broker. One gauge per broker.

2) kafka.server:name=OfflineLogDirectoriesCount,type=LogManager

The number of offline log directories on a live broker. One gauge per broker.

Changes in Operational Procedures

In this section we describe the expected changes in operational procedures in order to switch Kafka to run with JBOD instead of RAID. Administrators of Kafka cluster need to be aware of these changes before switching from RAID-10 to JBOD.

1) Need to adjust replication factor and min.insync.replicas

After we switch from RAID-10 to JBOD, the number of disks that can fail will be smaller if replication factor is not changed. Administrator needs to change r eplication factor and min.insync.replicas to balance the cost, availability and performance of Kafka cluster. With proper configuration of these two configs, we can have reduced disk cost or increased tolerance of broker failure and disk failure. Here are a few examples:

- If we switch from RAID-10 to JBOD and keep replication factor to 2, the disk usage of Kafka cluster would be reduced by 50% without reducing the availability against broker failure. But tolerance of disk failure will decrease.
- If we switch from RAID-10 to JBOD and increase replication factor from 2 to 3, the disk usage of Kafka cluster would be reduced by 25%, the number of brokers that can fail without impacting availability can increase from 1 to 2. But tolerance of disk failure will still decrease.
- If we switch from RAID-10 to JBOD and increase replication factor from 2 to 4, the disk usage of Kafka would stay the same, the number of brokers that can fail without impacting availability can increase from 1 to 3, and number of disks that can fail without impacting availability would stay the same.

2) Need to monitor disk failure via OfflineLogDirectoriesCount metric

Administrator will need to detect log directory failure by looking at OfflineLogDirectoriesCount. After log directory failure is detected, administrator needs to fix disks and reboot broker.

3) Need to decide whether to restart broker that had known disk failure before fixing the disk

Although this KIP allows broker to start with bad disks (i.e. log directories), Kafka administrator needs to be aware that problematic disks may be simply slow (e.g. 100X slower) without giving fatal error (e.g. IOException) and Kafka currently does not handle this scenario. Kafka cluster may be stuck in an unhealthy state if disk is slow but not showing fatal error. Since disk with known failure is more likely to have problematic behavior, administrator may choose not to restart broker before fixing its disks to play on the safe side.

In addition, administor needs to be aware that if a bad log directory is removed from broker config, all existing replicas on the bad log directory will be re-created on the good log directories. Thus bad log directories should only be removed from broker config if there is enough space on the good log directories.

Compatibility, Deprecation, and Migration Plan

The KIP changes the inter-broker protocol. Therefore the migration requires two rolling bounce. In the first rolling bounce we will deploy the new code but broker will still communicate using the existing protocol. In the second rolling bounce we will change the config so that broker will start to communicate with each other using the new protocol.

Test Plan

The new features will be tested through unit, integration, and system tests. In the following we explain the system tests only. In addition to the tests described in this KIP, we also have test in KIP-113 to verify that replicas already created on good log directories will not be affected by failure of other log directories.

Note that we validate the following when we say "validate client/cluster state" in the system tests.

- Brokers are all running and show expected error message
- topic description shows expected results for all topics
- A pair of producer and consumer can successfully produce/consume from a topic without message loss or duplication.

1) Log directory failure discovered during bootstrap

- Start 1 zookeeper and 3 brokers. Each broker has 2 log directories.
- Create a topic of 1 partition with 3 replicas
- Start a pair of producer and consumer to produce/consume from the topic
- Kill the leader of the partition
- Change permission of the first log direcotry of the leader to be 000
- Start the previous leader again
- Validated client/cluster state

2) Log directory failure discovered on leader during runtime

- Start 1 zookeeper and 3 brokers. Each broker has 2 log directories.
- Create a topic of 1 partition with 3 replicas
- Start a pair of producer and consumer to produce/consume from the topic
- Change permission of the leader's log direcotry to be 000
- Validated client/cluster state
// Now validate that the previous leader can still serve replicas on the good log directories
- Create another topic of 1 partition with 3 replicas
- Kill the other two brokers
- Start a pair of producer and consumer to produce/consume from the new topic
- Validated client/cluster state

3) Log directory failure discovered on follower during runtime

- Start 1 zookeeper and 3 brokers. Each broker has 2 log directories.
- Create a topic of 1 partition with 3 replicas
- Start a pair of producer and consumer to produce/consume from the topic
- Change permission of the follower's log direcotry to be 000
- Validated client/cluster state
// Now validate that the follower can still serve replicas on the good log directories
- Create another topic of 1 partition with 3 replicas
- Kill the other two brokers
- Start a pair of producer and consumer to produce/consume from the new topic
- Validated client/cluster state

Rejected Alternatives

- Let broker keep track of the replicas that it has created.

The cons of this approach is that each broker, instead of controller, keeps track of the replica placement information. However, this solution will split the task of determining offline replicas among controller and brokers as opposed to the current Kafka design, where the controller determines states of replicas and propagate this information to brokers. We think it is less error-prone to still let controller be the only entity that maintains metadata (e.g. replica state) of Kafka cluster.

- Avoid adding "create" field to LeaderAndIsrRequest.
If we don't add "*create*" field to LeaderAndIsrRequest, then broker will need to keep track of the list of replicas it has created and persists this information in either local disks or zookeeper.

- Add a new field "created" in the existing znode `/broker/topics/[topic]/partitions/[partitionId]/state` instead of creating a new znode
If we don't include list of created replicas in the `LeaderAndIsrRequset`, the leader would need to read this list of created replicas from zookeeper before updating isr in the zookeeper. This is different from the current design where all information except isr are read from LeaderAndIsrRequest from controller. And it creates opportunity for race condition. Thus we propose to add a new znode to keep those information that can only be written by controller.

- Identify replica by 4-tuple (topic, partition, broker, log_directory) in zookeeper and various requests
This would require big change to both wire protocol and znode data format in order to specify log directory for every replica. And it requires Kafka to keep track of log directory of replica and update information in zookeeper every time a replica is moved between log directories on the same broker for load-balance purpose. We would like to avoid the additional code complexity and performance overhead.

- Use RAID-5/6
We have tested RAID 5/6 in the past (and recently) and found it to be lacking. So, as noted, rebuild takes more time than RAID 10 because all the disks need to be accessed to recalculate parity. In addition, there's a significant performance loss just in normal operations. It's been a while since I ran those tests, but it was in the 30-50% range - nothing to shrug off. We didn't even get to failure testing because of that.

we ran the tests with numerous combinations of block sizes and FS parameters. The performance varied, but it was never good enough to warrant more than a superficial look at using RAID 5/6. We also tested both software RAID and hardware RAID.

- Setup one broker per disk and deploy multiple broker on the same machine
one-broker-per-disk would work and should require no major change in Kafka as compared to this KIP. So it would be a good short term solution. But it has a few drawbacks which makes it less desirable in the long term. Assume we have 10 disks on a machine. Here are the problems:

1) It seems if we were to tell kafka user to deploy 50 brokers on a machine of 50 disks. The overhead of managing so many brokers' config would also increase.

Running one broker per disk adds a good bit of administrative overhead and complexity. If you perform a one by one rolling bounce of the cluster, you're talking about a 10x increase in time. That means a cluster that restarts in 30 minutes now takes 5 hours. If you try and optimize this by shutting down all the brokers on one host at a time, you can get close to the original number, but you now have added operational complexity by having to micro-manage the bounce. The broker count increase will percolate down to the rest of the administrative domain as well - maintaining ports for all the instances, monitoring more instances, managing configs, etc.

2) Either when user deploys Kafka on a commercial cloud platform or when user deploys their own cluster, the size or largest disk is usually limited. There will be scenarios where user want to increase broker capacity by having multiple disks per broker. This JBOD KIP makes it feasible without hurting availability due to single disk failure.

3) There is performance concern when you deploy 10 broker vs. 1 broker on one machine. The metadata the cluster, including FetchRequest, ProduceResponse, MetadataRequest and so on will all be 10X more. The packet-per-second will be 10X higher which may limit performance if pps is the performance bottleneck. The number of socket on the machine is 10X higher. And the number of replication thread will be 100X more. The impact will be more significant with increasing number of disks per machine. Thus it will limit Kakfa's scalability in the long term. Our stress test result shows that one-broker-per-disk has 15% lower throughput.

You also have the overhead of running the extra processes - extra heap, task switching, etc. We don't have a problem with page cache really, since the VM subsystem is fairly efficient about how it works. But just because cache works doesn't mean we're not wasting other resources. And that gets pushed downstream to clients as well, because they all have to maintain more network connections and the resources that go along with it.

4) Less efficient way to manage quota. If we deploy 10 brokers on a machine, each broker should receive 1/10 of the original quota to make sure the user doesn't exceed a given byte-rate limit on this machine. It will be harder for user to reach this limit on the machine if e.g. user only sends/receives from one partition on this machine.

5) Rebalance between disks/brokers on the same machine will be less efficient and less flexible. Broker has to read data from another broker on the same machine via socket. It is also harder to do automatic load balance between disks on the same machine in the future.

6) Running more brokers in a cluster also exposes you to more corner cases and race conditions within the Kafka code. Bugs in the brokers, bugs in the co ntrollers, more complexity in balancing load in a cluster (though trying to balance load across disks in a single broker doing JBOD negates that).

Potential Future Improvement

1. Distribute segments of a given replica across multiple log directories on the same broker. It is useful but complicated. It is something that can be done later via a separate KIP.
2. Provide intelligent solution to select log directory to place new replicas and re-assign replicas across log directories to balance the load.
3. Have broker automatically rebalance replicas across its log directories. It is worth exploring separately in a future KIP as there are a few options in the design space.
4. Allow controller/user to specify quota when moving replicas between log directories on the same broker.

5. Let broker notify controller of ISR change and disk state change via RPC instead of using zookeeper
6. Handle various failure scenarios (e.g. slow disk) on a case-by-case basis. For example, we may want to detect slow disk and consider it as offline.
7. Allow admin to mark a directory as bad so that it will not be used.