KIP-18 - JBOD Support

- Motivation
- Public Interfaces
- Proposed Changes
    - Implementation Details
        - IO Errors
        - ExceptionHandler
            - Notifying Controller
            - Putting LogDirs Directory Offline
        - Partitions Restart
            - Restarted Partition Leader Selector
        - Open questions
- Compatibility, Deprecation, and Migration Plan
- Rejected Alternatives

Status

Current state: Discarded in favour of KIP-112 and KIP-113

Discussion thread: *here [Change the link from the KIP proposal email archive to your own email thread]*

JIRA: *here [Change the link from KAFKA-1 to your own ticket]*

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

Motivation

For Kafka, which is a distributed self-replicated system, a preferred storage schema is JBOD (Just a bunch of disks). The reasoning is the same as for HDFS DataNode-s, where the main consideration is disk operations performance (comparing to RAID architectures).

With KAFKA-188 was added support of multiple `log.dirs`. The issue is that currently Kafka doesn't tolerate file IO failures – any disk error crashes a broker, which is apparently unreliable in JBOD schema.

The main goal of this change is to introduce more advanced disk management logic, tolerating possible disk errors on application level, yet supporting JBOD architecture.

Public Interfaces

Doesn't change public interfaces but affects many internal server components and brokers behavior upon IO errors.

Proposed Changes

The high-level idea is to catch all IO errors, define affected disks/partitions and “restart” problematic partitions via the mechanism similar to reassign partition.

Currently almost any IO error is wrapped into `KafkaStorageException` which, when caught, results in `Runtime.halt(1)`.

The new work-flow will be the following:

1. On the lowest level catch IO exceptions, wrap them it into specific `KafkaStorageException` and let them “bubble up” where they are handed by a dedicated component

2. Exception handler component on exception does the following:

    2.1. Detect directory that is no longer available and put it to offline - all operations with the respective directory are stopped, new logs are not created there

    2.2. Detect partitions that were lost

    2.3. Notify controller that specific partitions need to be restarted

3. Controller upon receiving notification acts as if broker went shutdown but only for specific partitions – new `LeaderAndIsrRequest` is calculated and propagated to all brokers

Implementation Details

IO Errors

Logically, different IO exceptions may require different actions. It is proposed to create `KafkaStorageException` hierarchy for IO errors. It will help to pattern match on it in a component responsible for error handling and define a context which exceptions need to pass to be handled properly. E.g.:

1) `LogAppendException(log:Log)`

2) `LogReadException(log: Log)`

3) `HighWatermarkIOException(highwatermarkFile: File)`

ExceptionHandler

Currently actual file IO operations are performed by different components on different levels – `Log` writes messages on disk, `LogManager` is responsible for recovery checkpoints, `ReplicaManager` handles high-watermark files. Thus it's hard to localize all exceptions in one place and handle them there. The new approach is: on the lowest level wrap IO exception, enrich it with needed context and rethrow it where it can be handled. For now it is proposed to handle all IO exception on the `ReplicaManager` level.

It is also proposed to add a separate class which will be responsible for IO exceptions handling and encapsulate logic for detecting problematic disks /partitions and firing off “restart” procedure for lost partitions. Since actual exception handling will happen in `ReplicaManager` it is proposed to implement `ExceptionHandler` as its inner class.

Each exception case might be handled differently but typically the workflow will include:

1. From the exception context check whether entire disk (`config.logDirs` entry) went down
2. If entire disk ***d*** is not available go to 2.1, otherwise 3.
   2.1. Remove ***d*** from `config.logDirs` for this broker and put it to offline - ensure data is not written to that disk (e.g. update high-watermark set so they are no checkpointed to the ***d***)
   2.2. Identify (from the `LogManager`) partitions - ***pp***, that were stored in ***d***
   2.3. Offline each partition ***p*** from ***pp*** set:
      2.3.1. Remove ***p*** from `ReplicaManager` partitions pool
      2.3.2. Remove ***p***'s log from `LogManager`, shutdown all housekeeping for ***p***
   2.4. Notify controller that ***pp*** partitions need to be restarted, go to 4
3. From the exception context define log and partition – ***p,*** which is not available and do steps 2.3-2.4 for ***p***
4. Some exception-specific logic that needs to be executed

Notifying Controller

Currently brokers communicate with controller via Zookeeper. It should be refined further in which format ExceptionHandler needs to store partitions that require restart. It might be:

JSON under the new path `/restart_partitions`:

```
{
 “version”: “1”,
 “broker”: “1”,
 “partitions”: [
                         “topic”: “my_topic”,
                         “partitions”: [“1”, “2”, “3”]
                    ]
}
```

The problem is that brokers may update `/restart_partitions` znode simultaneously and controller having restarted partitions should remove respective data from Zookeeper. It should be investigated what's the better solution (maybe a distributed Queue – from ZK recipes).

Putting LogDirs Directory Offline

Logs are stored in the separate directories inside `config.logDirs` entries. Naturally, the entire directory can become unavailable (e.g. disk was removed) in this case particular directory should be removed from the managed pool of `logDirs`. Currently `LogManager` holds this pool of disks, manages recovery checkpoints, retention etc.

When `ExceptionHandler` detects that `logDirs` directory is should be put to offline. It is proposed to expose this functionality in `LogManager`. The workflow is the following:

1. Define logs and partitions which were stored in the unavailable directory

2. Abort and pause all future cleaning for defined partitions

3. Update recovery checkpoints list to remove the respective directory

4. Remove defined logs from the logs pool and update `logDirs` (so that scheduled jobs - `kafka-log-retention`, `kafka-log-flusher` and `kafka-recovery-point-checkpoint` are not executed on logs put to offline)

Note: currently scheduled jobs are not executed in lock and logs pool is not protected by lock, so with these changes data races are possible. It should be considered how changing jobs (executing them in lock) may affect performance.

Partitions Restart

Partitions restart means re-electing leader, in-sync replicas and assigned replicas so that partitions that were lost on some broker due to an IO error were re-replicated on that broker.

On startup the controller (similarly to reassign partitions, preferred replica leader election procedures) registers Zookeeper listener on `/restart_partitions` path. Upon receiving notification controller:

1. Parses zookeeper data to a `Map[TopicAndPartiton, List[BrokerId]]`

2. In lock, for each partition (filtering out partitions that are being deleted, reassigned etc) triggers partition state machine state transition `OnlinePartition` with a special leader selector – `RestartedPartitionsLeaderSelector`

3. Removes data from `/restart_partitions`

4. Releases the lock

Other operations (like propagating `LeaderAndIsr`, `UpdateMetadata` to all brokers etc) will be handled as part of state transition automatically by the partition state machine.

Restarted Partition Leader Selector

In essence, the logic is similar to the case when the entire replica goes down, the only difference is that replica which requested partition restart should remain in assigned replicas list.
ISR, AR and leader are set according to the following rules:

Given the partition ***p*** on replica (which requested partition restart) ***b*** with assigned replicas ***ar***, in-sync replicas ***isr*** and a leading replica ***leader***:

a) if ***b*** was a leading replica for ***p***

new_isr := ***isr*** - ***b***

new_ar := ***ar*** (assigned replicas remain the same)

new_leader := (***new_ar*** which are in ***new_isr)***.head()

b) if ***b*** was a follower replica for ***p***

new_isr := ***isr*** - ***b***

new_leader := ***leader*** (leader remains the same)

new_ar := ***ar*** (assigned replicas remain the same)

All edge cases (like new isr set is empty) are handled similarly to offlinePartionLeaderSelector.

Open questions

1. Disk availability check operation

`ExceptionHandler` needs to check disk availability. What can be a simple, fast operation to do it?

2. Handling `LogReadException`

`Log.read()` differs from `Log.append()` because Kafka avoids unnecessary data copying and actual file IO operations happen in `SocketServer` – data is copied write to the channel via `FileChannel.transferTo()`. This code belongs to network package so there is really no notion of Log, Replica and other things to handle this case properly.

3. `/restart_partitions` format

Brokers may update `/restart_partitions` znode simultaneously and controller having restarted partitions should remove respective data from Zookeeper. It should be investigated what's the better solution (maybe a distributed Queue – from ZK recipes)

4. Operation retries

Does it makes sense to retry operation before firing restart partitions?

Compatibility, Deprecation, and Migration Plan

No public interfaces changes. Users won't have to restart brokers on IO errors (e.g. after disk becomes unavailable).

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.