Kafka API Refactoring

Current Architecture

In 0.8.1.1 with in-built offset management, server side architecture with the life cycle of the produce/fetch request is summarized below (caller --> callee):

```
ProduceRequest --> 1) KafkaApis.appendToLocalLog()         --> ReplicaManager.getPartition()        -->
Partition.appendMessagesToLeader()

                   2) KafkaApis.maybeUnblockDelayedFetch()

                   3) RequestChannel.sendResponse()   OR   ProducerRequestPurgatory.watch()

                   4) ProducerRequestPurgatory.update()

-----------------------------------------------------------------------------------------------

FetchRequest -->   1) KafkaApis.maybeUpdatePartitionHW()   --> ReplicaManager.recordFollowerPosition()   -->
Partition.updateLeaderHWAndMaybeExpandIsr() / maybeIncrementLeaderHW()

                   2) ProducerRequestPurgatory.update()    --> DelayedProduce.isSatisfied()              -->
KafkaApis.maybeUnblockDelayedFetchRequests()

                   3) KafkaApis.readMessageSet             --> ReplicaManager.getReplica()               -->
Log.read()

                   4) RequestChannel.sendResponse()   OR   FetchRequestPurgatory.watch()

-----------------------------------------------------------------------------------------------

ProducerRequestPurgatory:

    // called as step 4) of handling produce request, or step 2) of handling fetch request
    ProducerRequestPurgatory.update()                       --> DelayedProduce.respond()           -->
RequestChannel.sendResponse()

    // any time
    ProducerRequestPurgatory.expire()                       --> DelayedProduce.respond()           -->
RequestChannel.sendResponse()

FetchRequestPurgatory:

    // called as step 2) of handling produce request, or inside DelayedProduce.isSatisfied()
    KafkaApis.maybeUnblockDelayedFetchRequests()            --> FetchRequestPurgatory.update()     -->
DelayedFetch.respond()             --> RequestChannel.sendResponse()

    // any time
    FetchRequestPurgatory.expire()                          --> DelayedFetch.respond()             -->
RequestChannel.sendResponse()
```

The problem

As we can see from above, since delayed produce needs to access KafkaApis.maybeUnblockDelayedFetchRequests(), etc, and delayed fetch needs to fetch the data to form the response. As a result, we ended up keeping the appending and fetching logic inside KafkaAPIs and also keeping purgatories /delayed requests inside Kafka APIs to let them access these functions/variables. The problems for this are:

1) Logic of the append message / read message from Replica Manager leaks into KafkaAPIs, and KafkaAPIs itself becomes very huge containing all purgatory / delayed requests classes.

2) However, logic for satisfying delayed fetch requests are not correct: it needs to be related to HW modifications. Hence it needs to also access Partition, which will lead to more logic leak if we follow current architecture.

3) With inbuild offset management, we have to hack the KafkaAPIs and its corresponding delayed requests as follows:

```
CommitOffsetRequest --> 1) KafkaApis.appendToLocalLog()         --> OffsetManager.
producerRequestFromOffsetCommit()    // returns a new ProducerRequest from the OffsetCommitRequest

                        2) KafkaApis.appendToLocalLog()

                        3) OffsetManager.putOffsets()   // put the offset into cache

                        4) OffsetManager.offsetCommitRequestOpt().get()    // transform back a
OffsetCommitResponse

                        5) RequestChannel.sendResponse()   OR   ProducerRequestPurgatory.watch()

-------------------------------------------------------------------------------------------------

DelayedProduce.respond()  --> 1) if(not timed out) OffsetManager.putOffsets()

                              2) OffsetManager.offsetCommitRequestOpt().get()


                              3) RequestChannel.sendResponse()
```

The architecture diagram is shown below:

blocked URL

The Idea

Is to refactor the Kafka Apis along with Replica Manager and Offset Manager (Coordinator) such that the produce/fetch purgatories are moved to replica manager, and are isolated from requests (i.e. they may be just purgatories for append and fetch operations). By doing so:

1) Kafka API becomes thinner, only handling request-level application logic and talk to Request Channel.

2) Read message and append message logic is moved to Replica Manager, which handles the logic of "committed" appending and fetching "committed data".

3) Offset Manager (Coordinator) only needs to talk to the Replica Manager for handling offset commits, no need to hack Kafka Apis and Delayed Fetch requests.

This refactoring will also benefit the following new consumer / coordinator development.

Purgatories and Delayed Operations

We refactor the purgatories and delayed requests (now should be called operations) as follows. Here all the module names remain the same, with renaming suggestion in the bracket just for clear indication.

Delayed Request (Kafka Operation)

The base kafka operation just contains:

```
keys: Seq[Any] // currently it is called DelayedRequestKey, but we can rename it as we like

timeout: long // timeout value in milliseconds

callback: Callback // this is the callback function triggered upon complete, either due to timeout or operation
finished
```

One note here is that a new callback instance needs to be created for each operation, since it will need to remember the request object that it needs to respond on. The base callback class can be extended, with the basic parameters:

```
// trigger the onComplete function given that whether the operation has succeeded or failed (e.g. timed out).
onComplete(Boolean)
```

Besides these fields, a kafka operation also have the following interface:

```
// check if myself is satisfied
isSatisfied(): Boolean

// operations upon expiring myself
expire() = this.callback.onComplete(false) // can be overridden for recording metrics, etc
```

Delayed Produce Request (Message Append Operation)

Maintains the append metadata, including the replicate condition:

```
appendStatus: Map[TopicAndPartition, AppendStatus] // AppendStatus include starting offset, required offset,
error code, etc

replicateCondition: ReplicateCondition // ReplicationCondition can be just a ack integer, but may be extended
in the future
```

In addition, it implements the interface as:

```
isSatisfied(replicaManager): Boolean = // access the replicaManager to check if each partition's append status
has satisfied the replicate condition
```

Delayed Fetch Request (Message Fetch Operation)

Maintains the fetch metadata, including the fetch min bytes:

```
fetchInfo: Map[TopicAndPartition, LogOffsetMetadata] // LogOffsetMetadata include message offset, segment
starting offset and relative segment position

fetchMinBytes: int
```

And implements the interface as:

```
isSatisfied(replicaManager): Boolean = // access the replicaManager to check if the accumulated bytes exceeds
the minimum bytes, with some corner case special handling.
```

Request Purgatory (Kafka Purgatory)

The base purgatory provides the following APIs:

```
// check if an operation is satisfied already; if not, watch it.
maybeWatch(operation: Operation): Boolean

// return a list of operations that are satisfied given the key
update(Any): List[KafkaOperations]
```

And its expiry reaper will purge the watch list and for each expired operation trigger operation.expire(). This purgatory is generic and can be actually used for different kinds of operations.

Kafka Server Modules

With these purgatories and operations, server side modules can be refactored as follows:

Replica Manager

Replica Manager maintain metadata of partitions, including their local and remote replicas, and talks to Log Manager for log operations such as log truncation, etc. Here is the proposed API:

```
// append messages to leader replicas of the partition, and wait for replicated to other replicas,
// the callback function will be triggered either when timeout or the replicate condition is satisfied
appendMessages(Map[TopicAndPartition, ByteBufferMessageSet], ReplicateCondition /* acks, etc */,  long /*
timeout */, Callback) {

  // 1. Partition.appendToLocalLog()
  // 2. If can respond now, call Callback.onComplete(true)
  // 3. Otherwise create new DelayedAppend(..., Callback)
  // 4. AppendPurgatory.maybeWatch(append)
}

// fetch only committed messages from the leader replica,
// the callback function will be triggered either when timeout or required fetch info is satisfied
fetchMessages(Map[TopicAndPartition, FetchInfo], int /* min bytes*/, long /* timeout */, RespondCallback) {

  // 1. Log.read()
  // 2. If can respond now, call Callback.onComplete(true)
  // 3. Otherwise create new DelayedFetch(..., new FetchCallback() { onComplete(): { fetchMessages;
RespondCallback.onComplete(true); } } )
  // 4. FetchPurgatory.maybeWatch(append)
}

// stop a local replica
stopReplica(TopicAndPartition, Boolean)

// make local replica leader of the partitions
leadPartition(Map[TopicAndPartition, PartitionState])

// make local replica follower of the partitions
followPartition(Map[TopicAndPartition, PartitionState])

// get (or create) partition, get (or create) replica, etc..
getPartition(TopicAndPartition)
getReplica(TopicAndPartition, int)
..
```

Coordinator / Offset Manager

Coordinator's offset manager will talk to the replica manager for appending messages.

```
// trigger the callback only when the offset is committed to replicated logs
putOffsets(Map[TopicAndPartition, OffsetInfo], RespondCallback) {

  // 1. replicaManager.appendMessage(... , new OffsetCommitCallback{ onComplete (): { putToCache;
RespondCallback.onComplete(true); } })
}

// access the cache to get the offsets
getOffsets(Set[TopicAndPartition]) : Map[TopicAndPartition, OffsetInfo]
```

Kafka Apis

Now the Kafka APIs becomes:

```
handleProduce(ProduceRequest) = // call replica-manager's appendMessages with callback sending produce response

handleFetch(FetchRequest) = // call replica-manager's fetchMessages with callback sending fetch response

handleCommitOffset(CommitOffsetRequest) = // call coordinator's offset manager with callback sending commit
offset response

handleFetchOffset(FetchOffsetRequest) = // call coordinator's offset manager to get the offset, and then send
back the response.
```

Request Handling Workflow

With the above refactoring, the request life cycle becomes:

```
ProduceRequest --> KafkaApis.handleProduce()         --> ReplicaManager.appendMessages()              -->
Partition.appendMessagesToLeader()

                                                                                                      -->
AppendPurgatory.maybeWatch()

                                                     --> Callback(): RequestChannel.sendResponse()

-------------------------------------------------------------------------------------------------------

FetchRequest -->   KafkaApis.handleFetch()           --> ReplicaManager.fetchMessages()               -->
ReplicaManager.readMessageSet()        --> Log.read()

                                                                                                      -->
FetchPurgatory.maybeWatch(new OffsetCommitCallback)

                                                                                                      -->
OffsetCommitCallback(): ReplicaManager.readMessageSet()


RespondCallback.onComplete()

                                                     --> RespondCallback: RequestChannel.sendResponse()



-------------------------------------------------------------------------------------------------------

FetchRequest -->   KafkaApis.handleCommitOffset()    --> Coordinator.CommitOffsets()                  -->
ReplicaManager.appendMessages(new OffsetCommitCallback)        --> Partition.appendMessagesToLeader()


--> AppendPurgatory.maybeWatch()

                                                                                                      -->
OffsetCommitCallback(): OffsetManager.putOffsets()


RespondCallback.onComplete()

                                                        RespondCallback(): RequestChannel.sendResponse()

-------------------------------------------------------------------------------------------------------

Partition.maybeIncrementLeaderHW()                   --> ReplicaManager.unblockDelayedFetchRequests() /
unblockDelayedProduceRequests()

Partition.appendMessagesToLeader()                   --> ReplicaManager.unblockDelayedFetchRequests()

Partition.recordFollowerLOE()                        --> ReplicaManager.unblockDelayedProduceRequests()
```

The caveat

As we can see, with fetch request and offset commit request, a nested callback is used (RepondCallback from Kafka APIs for sending the response through channel, and FetchCallback / OffsetCommitCallback for fetching the data for response / putting offset into cache). This nesting is not ideal, but necessary if want to have the strict layered architecture.

The new architectural diagram will be:

blocked URL