Inbuilt Consumer Offset Management

This page is now obsolete

Index

[0] Introduction

This wiki page describes the design of the inbuilt offset management feature. The relevant Jira is KAFKA-1000.

There are two phases or alternatives to implement the solution:

1. Have the consumers create an embedded producer and send offsets as produce messages.
2. Let the consumers use the offset commit request API (See KAFKA-657) to commit their offsets. We want the brokers to log the offset messages to disk and use the replication feature to attain durability and availability. So, effectively the logic for offset commit handling at broker end would be similar as that of produce request handling. Refactoring at the broker end would allow us to reuse the existing code at the broker side.

The current implementation (in KAFKA-1000-v1.patch) implements #1. The ideal end state is to implement #2, but the patch implements a functional inbuilt offset management feature that we can evaluate without the significant refactoring required for #2.

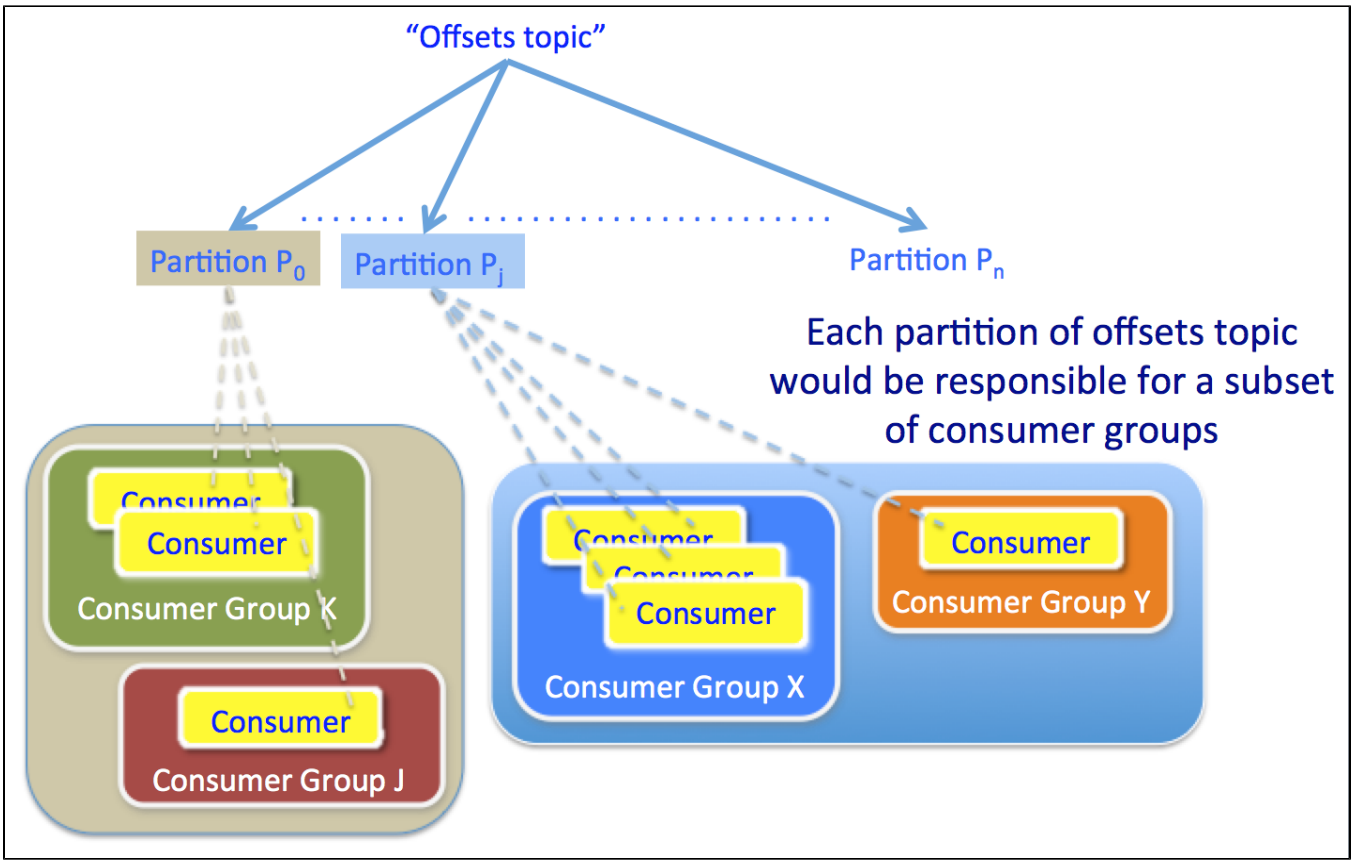
[1] Offset Management

(a) Offsets topic

All the consumer offset commit requests are sent as produce requests to a special topic named "__*offsets*". I will refer to this topic as the "offsets topic" here on. An offset commit message would contain following fields:

Key	Consumer group, topic, partition
Payload	Offset, metadata, timestamp

The offset commit messages are partitioned based on the consumer group in the key. This would result in all the messages of a given consumer group ending to a single broker and thus facilitates offset fetch requests without having to scatter-gather from several brokers.



One downside to this is that a particular broker would be a bottleneck if any consumer groups it is responsible for consumes a large number of partitions which would increase the number of offset commits to be handled. However, the volume of the offsets topic is expected to be orders of magnitude less than the other (data) topics so this should not be an issue.

(b) Offset managers

Offset manager is responsible for storing, fetching and maintaining consumer offsets. Every live broker has one instance of an offset manager. There are two concrete implementations of the offset manager trait:

- ZookeeperOffsetManager corresponds to the existing approach, which issues calls to zookeeper for offset storage and retrieval.
- DefaultOffsetManager provides the inbuilt offset management of consumer offsets.

A new property named "*offset.storage*" in config/server.properties is used to choose amongst the two offset manager implementations. For DefaultOffsetManager, there are two additional properties: "*offsets.topic.replication.factor*" and "*offsets.topic.num.partitions*", both having default value as "1" and are used for automatic creation of "offsets topic". Ideally, the replication factor of the "offsets topic" should be higher (say "3" or "4") than any normal Kafka topic in order to obtain higher availability of offsets information.

How are offsets stored in DefaultOffsetManager ?

Apart from having the offset stored in the logs on disk, DefaultOffsetManager maintains a table of the consumer offsets in order to serve the offset fetch requests quickly. A broker would contain entries in its offset table for only those partitions of "offsets topic" for which it is currently the leader. (In other words, replicas for these partitions will only have the offset information in their logs and no corresponding entries in their offset tables).

The segment size for logs of the offsets topic is kept low as compared to normal topics ("offsets.topic.segment.bytes" is set to 10 MB by default). Due to this, the LogCleaner picks up unclean segments of "offsets topic" sooner and as a result of frequent cleanup, the effective log size of "offsets topic" is low. This helps in reducing down the time required to read the logs and load them into the offsets table.

(c) Offsets manager interface

Below is a summary of the interface that offset manager trait offers and the implementation by the two concrete offset managers:

Method	DefaultOffsetManager	ZookeeperOffsetManager
startup	Initializes its internal vars (zkClient and log manager)	Initializes its zkClient object
getOffset	Read from the offset table	Read from zookeeper
putOffset	Add offset to the offset table	Does a zookeeper write

triggerLoadOffsets	When there is a leadership change and the existing broker becomes the leader for some partition of offsets topic, the offset manager needs to load the logs from disk for the given partition and populate the offset entries in its offsets table. This method schedules async threads to do the same.	Does nothing
cleanup	When there is a leadership change and the existing broker transitions from being leader to follower for some partition, it may have previous entries in the offsets table which no longer would be needed and must be cleaned up to cut down the size of the offsets table. ***To be implemented***	***To be implemented*** *KAFKA-559 stuff would go here.*
shutdown	Stops the scheduler.	Does nothing

[2] Offset Commit Implementation

(a) At consumer side

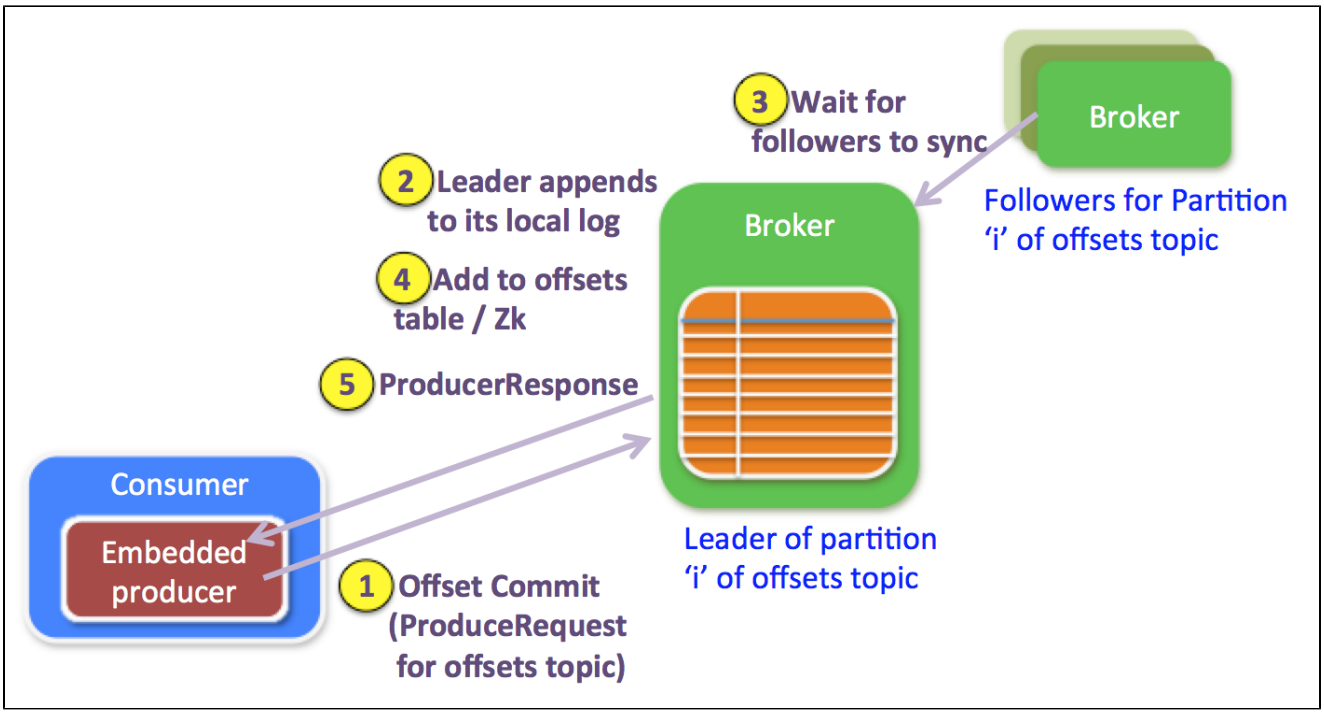
As described in 1(a), an offset commit message would be sent as a keyed produce request. When a consumer starts up, it creates a producer for the “offsets topic”. These are some properties of this “embedded producer”:

producer. type	sync	We could use async, but at least with sync we avoid delayed produce requests (due to batching). Furthermore, we will know immediately whether the offset messages have been received by the broker or not.
request. required. acks	-1	To ensure that all replicas are in sync with the leader and have seen all the offset messages.
serializer. class key. serializer. class	StringE ncoder	Both the key and payload are strings

Note that we do not compress the commit messages using any compression codec as the individual messages are small in size and compressing them would likely be counter-productive. Currently, the key and offset values are passed as plain Strings. We could switch to a more compact binary encoding instead of sending the Long offset and Int partition values as strings. We should also consider incorporating some versioning/format protocol so we can evolve the format if required over time.

(b) At broker-side

Brokers receive the offset commit information in form of a produce request. The normal processing of a producer request is same as it was. Once the data is appended to the local logs of the leader and all the replicas catch up, the leader would then check if this produce request was for the “offsets topic”. If so, then it will request the offset manager to add the offset. (In case of delayed produce requests, the update is done after the delayed produce request is satisfied.) Since we set acks to "-1", only those offsets which were successfully replicated across all brokers in the ISR are committed by the offset manager.



(c) Auto-creation of offsets topic

The offsets topic is auto-created on the first offset commit request. i.e., the very first consumer that consumes from a cluster would trigger its creation.

[3] Offset Fetch Implementation

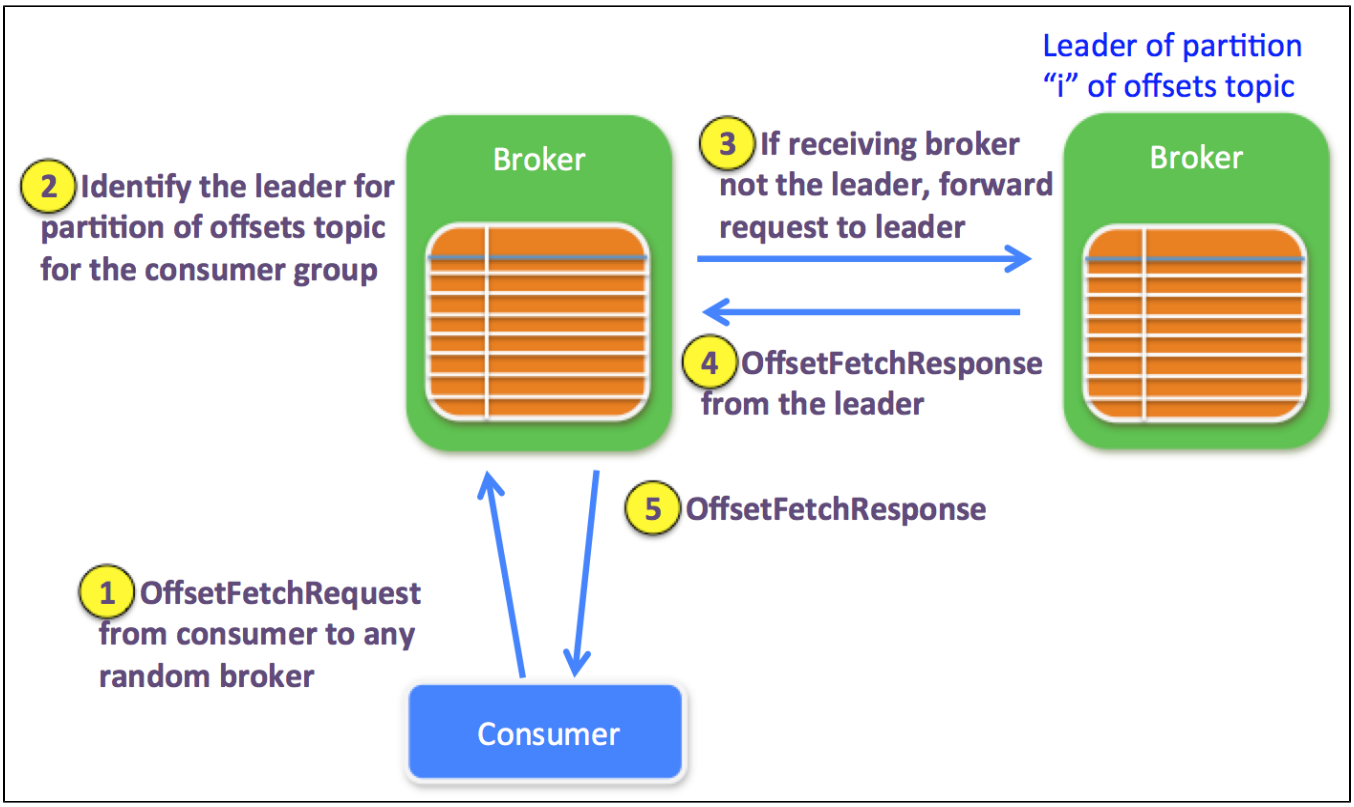
(a) At consumer side

On startup, a consumer would create a channel to any one of the live brokers. Thereafter, consumer would send all its "OffsetFetchRequest" to this randomly picked broker. The channel is closed/recreated on errors.

(b) At broker-side

An offset fetch request would encompass multiple topic-partitions. The receiving broker may or may not have the offset information for the requested topic-partitions. So it will have to talk with the other brokers too. A pool of channels is maintained to forward the requests to the leader brokers and these existing channels are reused for contacting the same leader broker again. Below are the detailed steps that a broker would perform after receiving an offset fetch request:

- The receiving broker would first determine which partition of the “offset topic” is responsible for the request.
- Then it would figure out the leader for the respective partition of “offsets topic” using the leader cache (which is updated after every update metadata request from controller).
- If the receiving broker is the leader, then it would fetch the offset from its offset manager and add it to response.
    1. If the offset is not present, then it would return "UnknownTopicOrPartitionCode".
    2. If the broker is loading the offsets table, then it would return "OffsetLoadingCode" code. The consumer on receiving such code would retry again after some time.
- If the receiving broker is not the leader for a particular topic-partition, then it would forward the OffsetFetchRequest to the current leader of that partition.
- If the "offsets topic" doesn't exist at this point, then it would try to auto-create the "offsets topic" and after a successful auto-creation it would return -1 as the offset.



[4] On broker failover

When a broker comes up, it would perform all the steps that the existing code does. If the broker is supposed to be the leader and / or follower for some partition of "offsets topic", then below are the actions triggered by the LeaderAndIsrRequest request:

- If it is the leader for some partition(s) of the "offsets topic": The broker would read the logs for the given partition of "offsets topic" and load the offsets into the offset table.
    - Any commit requests for this partition of the "offsets topic" would still update the offsets table. We prevent the stale offsets (from log) from overriding the recent offsets from the commit requests (if any).
    - Offset fetch requests on the keys managed by the partition of "offsets topic" being loaded will not be served until the loading completes. The broker would return a "OffsetLoadingCode" in response.
- If it is the follower for some partition(s) of the "offsets topic" : As for any normal kafka topic, the follower would start fetching data from the leader. As the followers' offset manager (if any) is not responsible for the partitions it follows, no action would be performed by the offset manager. (When cleanup is implemented, this will change.)

[5] Offsets cleanup from offset table, Zk and logs

(a) Offset removal protocol

OffsetCommitRequest (aka produce request for "offsets topic") with "null" payload would indicate that the offset with a given key (group-topic-partition) has to be deleted. Lets say a broker leading some partition X of offsets topic receives such request with null payloads. The normal append-to-logs would be performed and the followers would sync too. While the offset manager commits the offset, it will check if the payload is "Null". If so, then any existing entry with that key would be deleted from Zk or offsets table. If it turns out that all entries of a consumer group are getting cleanup up at this point, remove the znode (/consumers/XXXX) for this consumer group from Zk. This just covers the cleanup of offsets from Zk or offset table. The data would be still in the logs of offsets-topic. The log cleaner would perform cleanup using null payload while running in "dedupe" mode on the logs of offsets topic.

(b) Manual cleanup

User provides a consumer group to be deleted. The tool would first interact with Zk and get following info: i. Does the requested consumer group exists ? if no then abort ii. Does the consumer group has active consumers ? if yes then abort iii. Get all the topics consumed under this consumer group from this path in ZK : /consumers/XXXXX/owners/ iv. Get #partitions for each of these topics. Once it gets all this info, then it would send an OffsetCommitRequest (aka produce request for "offsets topic") with all these group-topic-partition pairs as key and a null payloads. The removal would take place as per "Offset removal protocol" described above. Times based removal would be possible with the offset fetch response returning the timestamp for each offset.

(c) Automatic cleanup

There would a separate thread running periodically inside the OffsetManager. This thread would be responsible for reading logs of all the partitions of the offsets topic that the current brokers leads. For each key (ie. group-topic-partition pair) discovered, it would save the timestamp of the latest entry found in the logs. Then it would send a "self" OffsetCommitRequest (aka produce request for "offsets topic") with Null payload for each key (ie. group-topic-partition pair) which satisfies: (Current time - timestamp of the latest offset entry) > OFFSET_RETENTION_PERIOD

(d) Automatic cleanup

. The concept of "self" OffsetCommitRequest (aka produce request for "offsets topic") would be ugly to code . The data from logs would get cleaned-up eventually when the log cleaner runs. . The approach uses existing notions of offset commits and tombstones in key-based log cleaner. Minimal amount of new code to be written.
. Once all brokers log a given group-topic-partition with "Null" payload, its deleted for sure. Even if the broker dies and comes back before log cleaner has actually cleaned it up, the loading phase would prevent the deleted offsets from being added to offset table.
. One additon in previous point: If the broker logs the commit message with null payload to disk and dies before performing cleanup from Zookeeper (for Zk based offset mgmt), then there would be stale entries in Zk. The loading phase of Zk based offset mgmt must take care of this.