Offset Management

This would be a two phase change:

1. First add a new API to move the details of offset storage to the broker.
2. Improve the way we store the offsets to move off Zookeeper

Phase I: API

The JIRA for this API is here.

OffsetCommit

This api saves out the consumer's position in the stream for one or more partitions. In the scala API this happens when the consumer calls commit() or in the background if "autocommit" is enabled. This is the position the consumer will pick up from if it crashes before its next commit().

Request:

```
OffsetCommitRequest => ConsumerGroup [TopicName [Partition Offset Metadata]]
  ConsumerGroup => string
  TopicName => string
  Partition => int32
  Offset => int64
  Metadata => string
```

These fields should be mostly self explanatory, except for metadata. This is meant as a way to attach arbitrary metadata that should be committed with this offset commit. It could be the name of a file that contains state information for the processor, or a small piece of state. Basically it is just a generic string field that will be passed back to the client when the offset is fetched. It will likely have a tight size limit to avoid server impact.

Response:

```
OffsetCommitResponse => [TopicName [Partition ErrorCode]]]
  ErrorCode => int16
```

OffsetFetch

This api reads back a consumer position previously written using the OffsetCommit api.

Request:

```
OffsetFetchRequest => ConsumerGroup [TopicName [Partition]]
```

Response:

```
OffsetFetchResponse => [TopicName [Partition Offset Metadata ErrorCode]]
```

For phase I the implementation would remain the existing zookeeper structure:

```
/consumers/[group_id]/offsets/[topic]/[broker_id-partition_id] --> offset_counter_value
```

If we started requiring zookeeper 3.4.x we could potentially optimize the implementation to use zk multi support to bundle together the updates (and reads?).

Integration With Clients

These APIs are optional, clients can store offsets another way if they like.

In the scala client we should not try to support "pluggable storage" but only implement support for using this API. To support folks who want to store offsets another way we already give back offsets in the message stream so they can store them in the way that makes sense. This will make more sense then some kind of SPI interface thingy. What is laking to complete this picture is allowing the consumer to initialize to particular known offset, but that can be added as a separate issue. If we had this, then a consumer would just need to turn off autocommit and implement the storage mechanism of their choice without needing to implement a particular interface.

Open Questions

Do we need some kind of optimistic locking?

E.g. the request could potentially include the current offset and would have the semantics "update the offset to x, iff the current offset is y". This would be nice for sanity checking a consumer implementation, but in the scala implementation the mutual exclusion for consumption is handled by zookeeper (and we had done some work to port this over to the broker) so this would just be a nice sanity check.

Phase 2: Backend storage

Zookeeper is not a good way to service a high-write load such as offset updates because zookeeper routes each write though every node and hence has no ability to partition or otherwise scale writes. We have always known this, but chose this implementation as a kind of "marriage of convenience" since we already depended on zk. The problems in this have become more apparent in our usage at LinkedIn with thousands of partitions and hundreds of consumers--even with pretty high commit intervals it is still...exciting.

Offset Storage Requirements

1. **High write load**: it should be okay if consumers want to commit after every message if they need to do that (it will be slower, but the system shouldn't collapse). We need to be able to horizontally scale offset writes.
2. **Durability**: once an offset is written it can't be lost even if a server crashes or disappears.
3. **Consistent reads**: all reads must return the last written value--no "eventual consistency" can be apparent.
4. **Small data size**. A consumer-group/topic/partition/offset tuple should be no more than 64 bytes in memory even with all the inefficiencies of jvm object overhead assuming the strings are interned and a compact lookup structure. So each 1GB of memory could potentially support ~16 million entries.
5. **Transactionality** across updates for multiple topic/partitions. The current implementation doesn't have this property, which is basically a bug. For example if you subscribe to N partitions in one consumer process and we mix those together into one iterator, when you call commit() it should not be possible for some of these offsets to get saved and some not as that leaves you in an inconsistent state. To fix the current implementation we would need to switch to ZK multi-write (only available in 3.4.x).

Implementation Proposal

I propose we make use of replication support and keyed topics and store the offset commits in Kafka as a topic. This would make offset positions consistent, fault tolerant, and partitioned. The key-based cleaner would be used to deduplicate the log and remove older offset updates. The implementation of an offset commit would just be publishing the offset messages to an "offset-commit-log" topic.The topic would be a poor data structure for serving offset fetch requests, so we would keep an in-memory structure that mapped group/topic/partition to the latest offset for fast retrieval. This structure would be loaded in full when the server started up. Actually this implementation isn't very different from what zookeeper itself is, except that it supports partitioning and would scale with the size of the kafka cluster.

A number of questions remain:

1. How do we partition data in this topic?
2. What is the format of these messages?
3. How do we ensure the atomicity of updates?
4. Which brokers can handle an offset update or fetch?

I propose that we partition the commit log topic by consumer group to make to give a total order to commits within a single group (as they would all be in the same partition). Downsides to this partitioning would be that the all traffic from a given group would go through a single server and if some groups committed much more than others load might balance poorly.

It would be possible to either store all offsets sent in a commit request in a single message or to have one offset per message. The problem with having multiple messages is that it breaks our atomicity requirement if the server loses mastership in the middle of such a write. This is a rare case, but should be dealt with. Grouping offsets together guarantees the atomicity of updates but raises the question of what key is being used for deduplicating updates. Since a consumer would generally send a single commit for all its partitions, but the partition assignment could change, it is hard to think of a key that would result in retaining the complete set of offsets for the consumer group.

I propose we use one message per offset and I will outline a scheme for making this fully transactional below.

I would propose that any broker can handle an offset request to make life easy for the client. If the broker happens to be the broker that is the master for the log for the consumer group doing the commit it can just apply the change locally, if not it would invoke a commit request to the appropriate server that is currently the leader for the correct partition. A simple client can just direct their requests anywhere; a client that optimizes a bit can try to hit the right server and save themselves the extra hop.

Some Nuances

All replicas will keep the in-memory lookup structure for offsets. This could just be a simple hashmap. This will contain only committed offsets to ensure we never need to undo updates.

The offset request will never receive a response until the the offset messages are fully committed to the log, and an unsuccessful commit must not result in updates to the hashmap.

Let's deal with some of the nuances of ensuring this kind of atomicity. To do this we will publish messages where the key is a string in the form "groupid-topic-partition". The contents of the message will be the offset, a transaction id, and a transaction size. The transaction id is just a counter maintained by the leader that is incremented for each commit request. The size allows the broker to ensure it received a complete set of messages.

A commit with 100 offsets will lock the hashmap, do 100 updates, and then unlock it to ensure the updates are atomic.

Some logic is required on broker initialization or leader transfer. In both these cases it may be the case that a partial write was accepted in the log. This is fine, but we need to ensure that partial writes do not end up in the hash map and do not lead to key-deduplication deleting the correct value. We can detect partial writes by just looking for the expected number of successive messages (i.e. if the request indicates it contains 5 offsets, but only 4 sequential messages with that transaction id are present then that commit cannot be applied. To prevent the prior record from being cleaned up by key-deduplication we should recommit the correct state at the end of the log when this is detected.