KIP-101 - Alter Replication Protocol to use Leader Epoch rather than High Watermark for Truncation

This KIP describes a change to the Kafka Replication Protocol originally proposed by Jun Rao

Contents

Status

Current state: Accepted

Discussion thread: here

JIRA:

https://issues.apache.org/jira/browse/KAFKA-1211

Relates to:

- https://issues.apache.org/jira/browse/KAFKA-3919
- https://issues.apache.org/jira/browse/KAFKA-1120

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

Definition of Terms

- **Leader Epoch:** A 32 bit, monotonically increasing number representing a continuous period of leadership for a single partition. This is marked on all messages.
- **Leader Epoch Start Offset:** The first offset in a new Leader Epoch.
- **Leader Epoch Sequence File:** A journal of changes of Leadership Epoch mapping the Leader Epoch to a starting offset for that epoch (Leader Epoch Start Offset).
- **Leader Epoch Request:** A request made by a follower to the leader to retrieve the appropriate Leader Epoch Start Offset. This is the Leader Epoch Start Offset of the subsequent epoch, or the Log End Offset if there is no subsequent epoch. The follower uses this offset to truncate its log.

Motivation

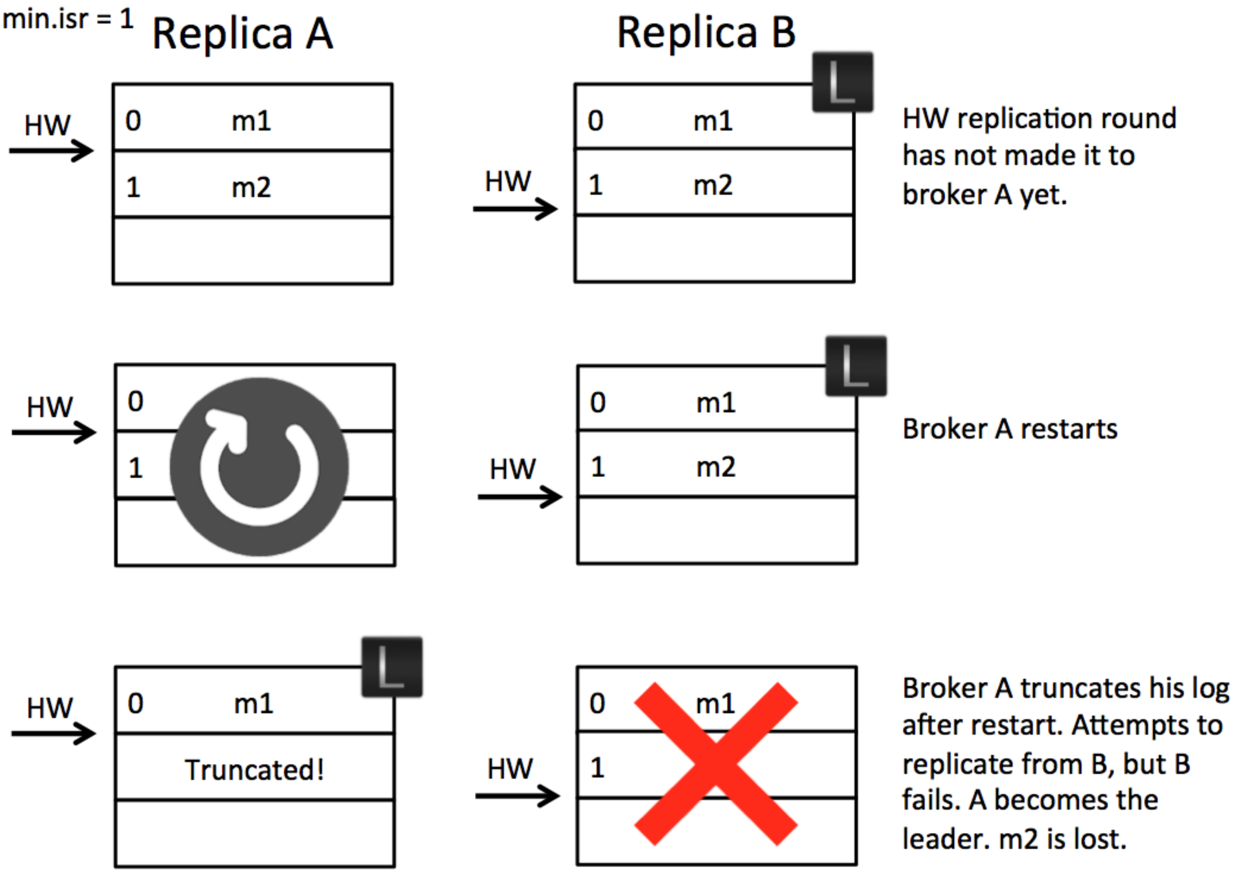
There are a few known areas where replication in Kafka can either create an unexpected lineage in the log, or worse cause runtime outages due to replica divergence. This KIP proposes a change to the replication protocol to ensure such cases cannot occur. First let us describe two use cases where the current replication protocol can lose data or diverge:

Scenario 1: High Watermark Truncation followed by Immediate Leader Election

The replication protocol in Kafka has two phases. Initially the the follower fetches messages. So it might fetch message m2. On the next round of RPC it will confirm receipt of message m2 and, assuming other replicas have confirmed successfully, the leader will progress the High Watermark. This is then passed back to the followers in the responses to their fetch requests. So the leader controls the progression rate of the High Watermark, which is propagated back to followers in subsequent rounds of RPC.

The replication protocol also includes a phase where, on initialisation of a follower, the follower will truncate its log to the High Watermark it has recorded, then fetch messages from the leader. The problem is that, should that follower become leader before it has caught up, some messages may be lost due to the truncation.

Let's take an example. Imagine we have two brokers A & B. B is the leader initially as in the below figure. (A) fetches message m2 from the leader (B). So the follower (A) has message m2, but has not yet got confirmation from the leader (B) that m2 has been committed (the second round of replication, which lets (A) move forward its high watermark past m2, has yet to happen). At this point the follower (A) restarts. It truncates its log to the high watermark and issues a fetch request to the leader (B). (B) then fails and A becomes the new leader. Message m2 has been lost permanently (regardless of whether B comes back or not).


min.isr = 1
Replica A
Replica B
L
HW
0 m1
1 m2
HW
0 m1
1 m2
HW replication round has not made it to broker A yet.
HW
0
1
L
HW
0 m1
1 m2
Broker A restarts
L
HW
0 m1
Truncated!
HW
0 m1
1
Broker A truncates his log after restart. Attempts to replicate from B, but B fails. A becomes the leader. m2 is lost.

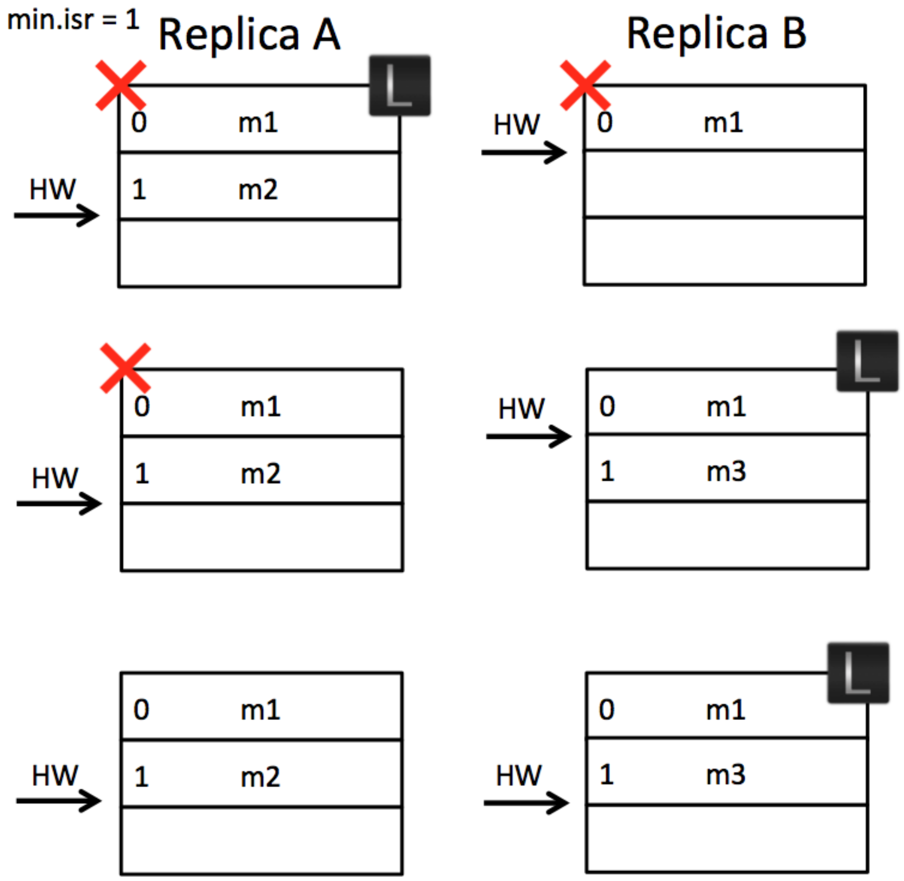
So the essence of this problem is the follower takes an extra round of RPC to update its high watermark. This gap leaves the possibility for a fast leader change to result in data loss as a committed message can be truncated by the follower. There are a couple of simple solutions to this. One is to wait for the followers to move their High Watermark before updating it on the leader. This is not ideal as it adds an extra round of RPC latency to the protocol. Another would be to not truncate on the follower until the fetch request returns from the leader. This should work, but it doesn't address the second problem discussed below.

Scenario 2: Replica Divergence on Restart after Multiple Hard Failures

Imagine again we have two brokers, but this time we have a power outage which affects both of them. It is acceptable, in such a case, that we could lose data if n replicas are lost (Kafka guarantees durability for n-1 replicas). Unfortunately there is an opportunity for the logs, on different machines, to diverge and even, in the worst case, for replication to become stuck.

The underlying issue is that messages are flushed to disk asynchronously. This means, after a crash, machines can be an arbitrary number of messages behind one another. When they come back up, any one might become leader. If the leader happens to be the machine with the fewest messages in its log we'll lose data. Whilst that is within the durability contract of the system, the issue is that the replicas can diverge, with different message lineage in different replicas.

As we support compressed message sets this can, at worst, lead to an inability for replicas to make progress. This happens when the offset for a compressed message set in one replica points to the midpoint of a compressed message set in another.



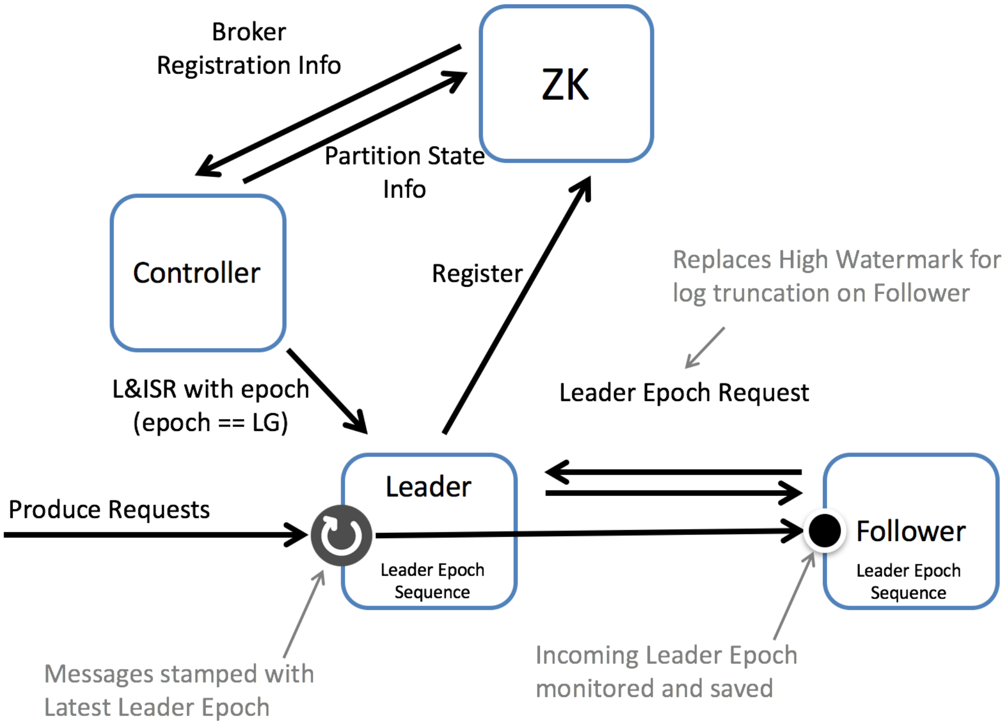
Disk images after crash

Broker B restarts first, becomes leader and accepts message m3

Broker A restarts. Truncates to its HWM but doesn’t realize it has a different message at offset 1. The logs diverge.

Solution

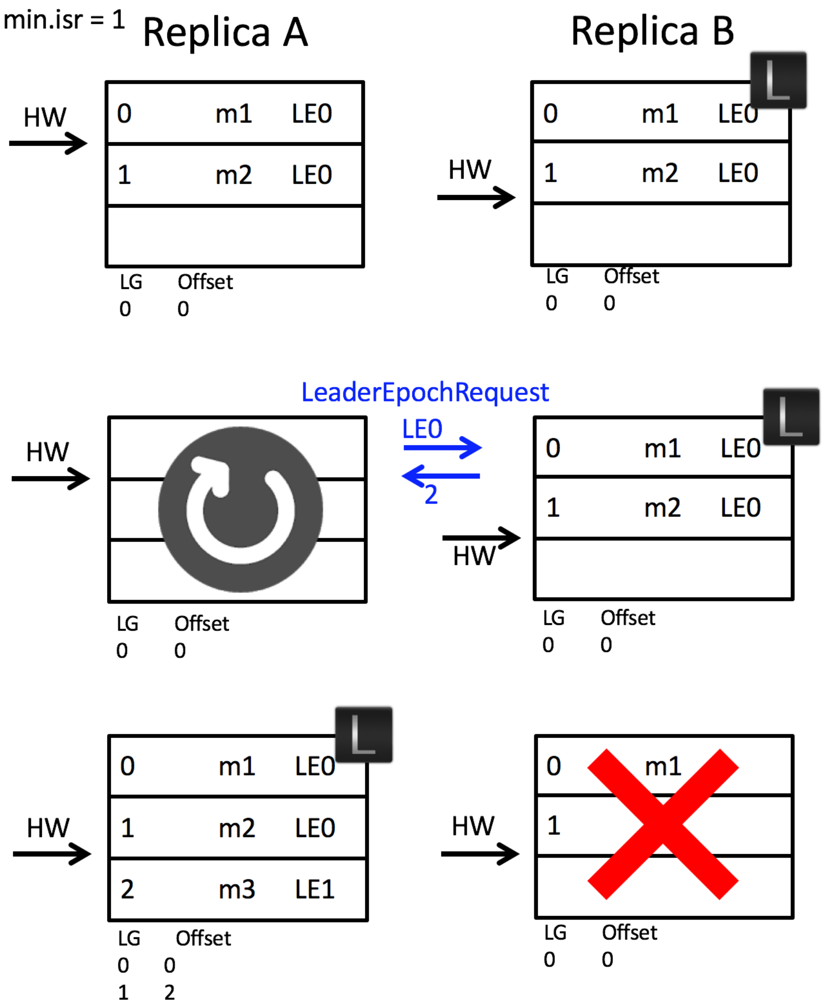
We can solve both of these issues by introducing the concept of a Leader Epoch. This allocates an identifier to a period of leadership, which is then added to each message by the leader. Each replica keeps a vector of [LeaderEpoch => StartOffset] to mark when leaders changed throughout the lineage of its log. This vector then replaces the high watermark when followers need to truncate data (and will be stored in a file for each replica). So instead of a follower truncating to the High Watermark, the follower gets the appropriate LeaderEpoch from the leader’s vector of past LeaderEpochs and uses this to truncate only messages that do not exist in the leader’s log. So the leader effectively tells the follower what offset it needs to truncate to.



We can walk through the implementation of this by addressing scenario 1:

In this solution the follower makes a request to the leader to determine if it has any divergent epochs in its log. It sends a LeaderEpochRequest to the leader for its current LeaderEpoch. In this case the leader returns the log end offset, although if the follower was lagging by more than one Leader Epoch, the leader would return the first offset in (Follower Leader Epoch + 1). So that's to say the LeaderEpoch response contains the offset where the requested LeaderEpoch ends.

In this case the LeaderEpochResponse returns offset 2. Note this is different to the high watermark which, on the follower, is offset 0. Thus the follower does not truncate any messages and hence message m2 is not lost.



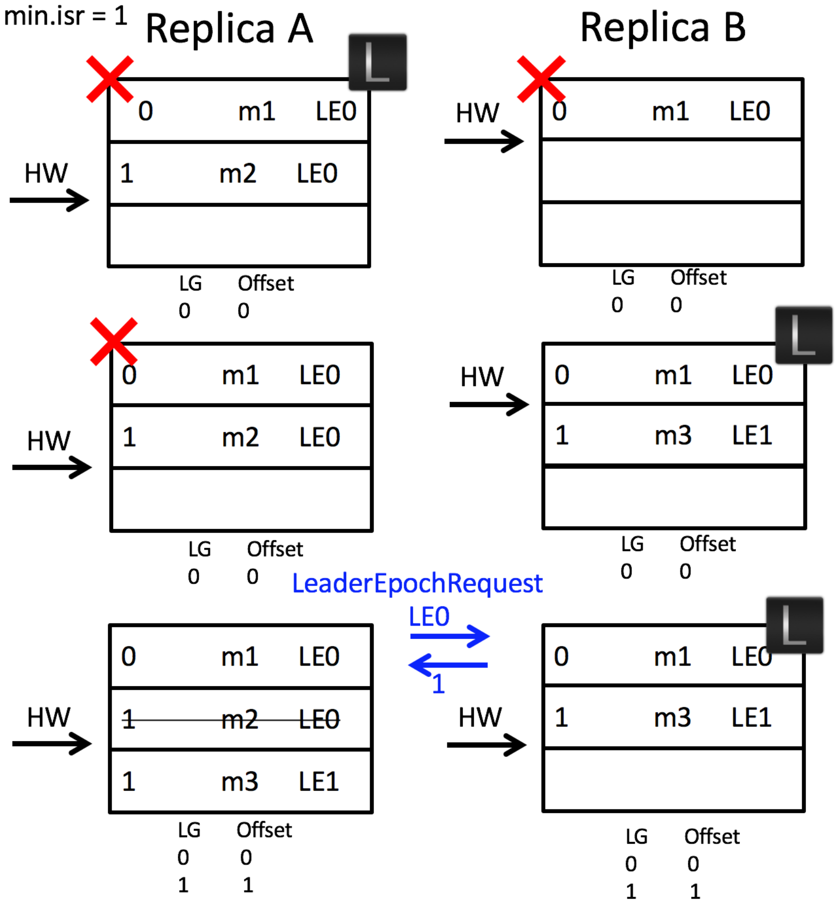
HW replication round has not made it to broker A yet.

Broker A restarts, it sends a LeaderEpoch request to the leader. This returns offset 2 (Log End offset LE0). Broker A thus has nothing to truncate. HW is no longer used for truncation.

Attempts to replicate from B, but B fails. A becomes the leader. m2 is retained. Subsequent messages are written as LE1 as the leader has changed.

This approach also addresses scenario 2:

When the two brokers restart after a crash, broker B becomes leader. It accepts message m3 but with a new Leader Epoch, LE1. Now when broker A starts, and becomes a follower, it sends a LeaderEpoch request to the leader. This returns the first offset of LE1, which is offset 1. The follower knows that m2 is orphaned and truncates it. It then fetches from offset 1 and the logs are consistent.



Disk images after crash

Broker B restarts, becomes leader and accepts message m3

Broker A restarts. Makes a LeaderEpoch request which returns offset 1. A truncates message m2 at offset 1, then fetches m3 from the leader.

NB: It should be noted that this proposal does not protect against log divergence for clusters that use the setting **unclean.leader.election.enable=true**. (See the appendix for more details.)

Proposed Changes

Leader Epoch

A leader has an existing concept of a LeaderEpoch. This is a 32 bit number, managed by the Controller, stored in the Partition State Info in Zookeeper and passed to each new leader as part of the LeaderAndIsrRequest. We propose stamping every message with the LeaderEpoch on the leader that accepts the produce request. This LeaderEpoch number is then propagated through the replication protocol, and used to replace the High Watermark, as a reference point for message truncation. This process takes the following form:

Steps:

1. Evolve the message format so that every message set carries a 4-byte Leader Epoch number.

2. In every log directory, we create a new Leader Epoch Sequence file, where we store the sequence of Leader Epoch and the Start Offset of messages produced in that epoch. This is cached in every replica, in memory as well.

3. When a replica becomes a leader, it first adds the new Leader Epoch and the log end offset of the replica to the end of Leader Epoch Sequence file and flushes it *. Each new message set produced to the leader is tagged with the new Leader Epoch.

4. When a replica becomes a follower, it does the following steps:

4.1 Recover all Leader Epochs from the Leader Epoch Sequence file, if needed.

4.2 Send a new LeaderEpoch request for the partition to the leader. The request includes the latest Leader Epoch in the follower's Leader Epoch Sequence.

4.3 The leader responds with the LastOffset for that LeaderEpoch. LastOffset will be the start offset of the first Leader Epoch larger than the Leader Epoch passed in the request or the Log End Offset if the leader's current epoch is equal to the one requested.

4.4 If the follower has any LeaderEpoch with a start offset larger than the LastOffset returned from the leader, it resets its last Leader Epoch Sequence to be the leader’s LastOffset and flushes the Leader Epoch Sequence File.

4.5 The follower truncates its local log to the leader’s LastOffset.

4.6 The follower starts fetching from the leader.

4.7.1 During fetching, if the follower sees the LeaderEpoch of a message set larger than its latest LeaderEpoch, it adds the new LeaderEpoch and the starting offset to its LeaderEpochSequence and flushes the file to disk.

4.7.2 The follower proceeds to append the fetched data to its local log.

For backward compatibility, in step 4.3, if the leader can't find the LastOffset (e.g., the leader hasn't started tracking Leader Epoch yet), the follower will fall back to the current approach by truncating the log to its high watermark.

*Flushing requires an I/O and could delay the process of becoming leaders. We could potentially optimize this by not flushing the LGS file here and moving the flushing to the background thread. Basically, when flushing a log segment, we now require the corresponding LGS file to be flushed first if at least one of the starting offsets in LGS is in this segment. If the broker has a hard crash, we will rebuild the potentially missing part in LGS from the last flushing point.

Additional Concerns

Extending LeaderEpoch to include Returning Leaders

It is possible that a leader might crash and rejoin the cluster without a round of leader election, meaning that the LeaderEpoch would not change. This can happen if a hard shutdown occurs, where all replicas for a partition crash. If the first machine starts back up, and was previously the leader, no election will occur and the LeaderEpoch would not be incremented, but, we require that such a condition trigger a new Leader Epoch.

To protect against this eventuality, if the controller receives a Broker Registration it will increment the Leader Epoch and propagate that value to the broker via the Leader and ISR Request (in the normal way). This mechanism will also be affected by the bug: KAFKA-1120 which is not addressed directly in the initial version of this KIP.

Maintaining the LeaderEpochSequenceFile

The Leader Epoch information will be held in files, per replica, on each broker. The files represent a commit log of LeaderEpoch changes for each replica and are cached in memory on each broker. When segments are deleted, or compacted the appropriate entries are removed from the cache & file. For deleted segments all LeaderEpoch entries for offsets less than equal to the segment log end offset are removed from the LeaderEpochSequenceFile. For compacted topics the process is slightly more involved. The LogCleaner creates a LeaderEpochSequence for every segment it cleans, which represents the earliest offset for each LeaderEpoch. Before the old segment(s) are replaced with the cleaned one(s) the LeaderEpochSequenceFile is regenerated and the cache updated.

Unclean Shutdown

After an unclean shutdown the Leader Epoch Sequence File could be ahead of the log (as the log is flushed asynchronously by default). Thus, if an unclean shutdown is detected on broker start (regardless of whether it's a leader or follower), all entries are removed from the Leader Epoch Sequence File, where the offset is greater than the log end offset.

Public Interfaces

Add API for OffsetForLeaderEpochRequest/Response

Offset For Leader Epoch Request V0

```
OffsetForLeaderEpochRequest (Version: 0) => [topics]

topics => topic [partitions]

topic => string

partitions => partition_id, leader_epoch

    partition_id => INT32

    leader_epoch => INT32
```

Offset For Leader Epoch Response V0

```
OffsetForLeaderEpochResponse (Version: 0) => [topics]

topics => topic [partitions]

topic => string

partitions => error_id, partition_id, end_offset

    error_id => INT16

    partition_id => INT32

    end_offset => INT64
```

Error Codes:

```
-1 -> Unknown Error

0 -> No Error

1 -> Not leader for partition

2 -> No epoch information for partition
```

Request Semantics

1. The offset returned in the response will be the start offset of the first Leader Epoch larger than last_leader_epoch_num or the Log End Offset if the leader's current epoch is equal to the partition_leader_epoch from the request.
2. The response will only include offsets for partition IDs, supplied in the request, which are leaders on the broker the request was sent to.

Add LeaderEpoch field to MessageSet

LeaderEpoch is added to MessageSets used in Fetch Responses returned as part of the internal replication protocol

MessageSet

```
MessageSet => [Offset MessageSize Message]

 Offset => int64

 MessageSize => int32

 leader_epoch => int32 <--------[NEW]
```

We bump up ProduceRequest/FetchRequest (and responses) versions to indicate the broker that this client supports new message format.

Add Leader Epoch Sequence File per Partition

A file will be used, per replica (located inside the log directory), containing the leader epoch and its corresponding start offset. This will be a text file with the schema:

leader-epoch-sequence-file

```
Version int32

[leader_epoch int32 start_offset int64]
```

Compatibility, Deprecation, and Migration Plan

For the message format changes, the upgrade path from KIP-32 can be reused. To upgrade from a previous message format version, users should:

1. Upgrade the brokers once with the inter-broker protocol set to the previous deployed version.
2. Upgrade the brokers again with an updated inter-broker protocol, but leaving the message format unchanged.
3. Upgrade all or most clients.
4. Restart the brokers, with the message format version set to the latest.

The LeaderEpochRequest will only be sent if all brokers support it. Otherwise the existing logic for High Watermark truncation will be used.

Rejected Alternatives

As discussed in the Motivations section, Scenario 1 could be addressed by delaying truncation until fetch requests return from the leader. This could potentially be done by creating a temporary segment file(s) which contains the messages required for the follower to catch up. Then, once caught up, the original segment is truncated to the High Watermark and the temporary segment(s) made active. This was rejected it does not solve Scenario 2.

We considered whether the cleaning of the LeaderEpochSequenceFiles would be simpler if there was one LGSF per segment, so they were effectively immutable once the segment is no longer active. This was rejected as there will be many log segments in the same Leader Epoch, typically meaning we could create a fair number of files we do not need.

Appendix (a): Possibility for Divergent Logs with Leader Epochs & Unclean Leader Election

There is still the possibility for the log to corrupt, even with Leader epochs, if min.isr=1 and unclean.leader.election.enabled=true. Consider two brokers A, B, a single topic, a single partition, reps=2, min.isr=1.
Intuitively the issue can be seen as:
-> The first two writes create a divergent log at offset 0 on completely isolated brokers.
-> The second two writes “cover up” that first divergent write so the LeaderEpoch request doesn’t see it.
Scenario:
1. [LeaderEpoch0] Write a message to A (offset A:0), Stop broker A. Bring up broker B which becomes leader
2. [LeaderEpoch1] Write a message to B (offset B:0), Stop broker B. Bring up broker A which becomes leader
3. [LeaderEpoch2] Write a message to A (offset A:1), Stop broker A. Bring up broker B which becomes leader
4. [LeaderEpoch3] Write a message to B (offset B:1),
5. Bring up broker A. It sends a Epoch Request for Epoch 2 to broker B. B has only epochs 1,3, not 2, so it replies with the first offset of Epoch 3 (which is 1). So offset 0 is divergent.
The underlying problem here is that, whilst B can tell something is wrong, it can't tell where in the log the divergence started.

One solution is to detect the break, by comparing complete epoch lineage between brokers, then truncate either to (a) zero or (b) the point of divergence, then refetch. However compacted topics make both of these options hard as arbitrary epochs & offset information can be 'lost' from the log. This information could be retained and managed in the LeaderEpoch file instead, but the whole solution is becoming quite complex. Hence it seems sensible to forgo this guarantee for the unclean leader election case, or at least push it to a subsequent kip.