KIP-279: Fix log divergence between leader and follower after fast leader fail over

Status

Current state: *Accepted*

Discussion thread: here

JIRA: **KAFKA-6361** - Getting issue details... STATUS

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

Motivation

We have observed the edge case in the replication failover logic which can cause divergence between logs. This occurs with clean leader election and in spite of the improved truncation logic from KIP-101. There are also known unclean leader election edge cases that KIP-101 did not address. This KIP proposes a change to the Kafka Replication Protocol, a follow up to KIP-101, that will fix edge cases that lead to log divergence for both clean and unclean leader election configs.

We first give two examples when logs can diverge.

Scenario 1 (described in KAFKA-6361): Leader change due to preferred leader election followed by immediate leader change.

Suppose we have brokers A and B. Initially A is the leader in epoch 1. It appends two batches: one in the range (0, 10) and the other in the range (11, 20). The first one successfully replicates to B, but the second one does not. In other words, the logs on the brokers look like this:

Step 1

```
Broker A:
0: offsets [0, 10], leader epoch: 1
1: offsets [11, 20], leader epoch: 1

Broker B:
0: offsets [0, 10], leader epoch: 1
```

Broker B is then elected with epoch 2 due to preferred leader election. So both brokers A and B are still in ISR. Broker B appends a new batch with offsets (11, n) to its local log. We now have this:

Step 2

```
Broker A:
0: offsets [0, 10], leader epoch: 1
1: offsets [11, 20], leader epoch: 1

Broker B:
0: offsets [0, 10], leader epoch: 1
1: offsets: [11, n], leader epoch: 2
```

Normally we expect broker A to truncate to offset 11 on becoming the follower, but before it is able to do so, broker B has Zookeeper session expiration and broker A again becomes leader, now with epoch 3. It then appends a new entry in the range (21, 30). The updated logs look like this:

Step 3

```
Broker A:
0: offsets [0, 10], leader epoch: 1
1: offsets [11, 20], leader epoch: 1
2: offsets: [21, 30], leader epoch: 3

Broker B:
0: offsets [0, 10], leader epoch: 1
1: offsets: [11, n], leader epoch: 2
```

Now what happens next depends on the last offset of the batch appended in epoch 2. On becoming follower, broker B will send an OffsetForLeaderEpoch request to broker A with epoch 2. Broker A will respond with offset 21 (the start offset of the first leader epoch larger than 2, since broker A does not know about epoch 2). There are three cases:

1) n < 20: In this case, broker B will not do any truncation. It will begin fetching from offset n, which will ultimately cause an out of order offset error because broker A will return the full batch beginning from offset 11 which broker B will be unable to append.

2) n == 20: Again broker B does not truncate. It will fetch from offset 21 and everything will appear fine though the logs have actually diverged.

3) n > 20: Broker B will attempt to truncate to offset 21. Since this is in the middle of the batch, it will truncate all the way to offset 10. It can begin fetching from offset 11 and everything is fine.

The problem is that broker B received the offset for the epoch it does not know about, and acts based on comparing it with offsets from a different epoch. What we want to do is truncate the follower's log to the largest common log prefix, where both offsets and epochs match.

Scenario 2: Fast leader fail over with unclean leader election.

Lets consider a scenario for unclean leader election described in Appendix (a) of KIP-101. Consider two brokers A,B, a single topic, a single partition, reps=2, min.isr=1.

Unclean Leader Election 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.

Public Interfaces

We propose to add leader epoch for every topic partition in OffsetForLeaderEpochResponse.

Offset For Leader Epoch Response V1

```
OffsetForLeaderEpochResponse (Version: 0) => [topics]
topics => topic [partitions]
topic => string
partitions => error_id, partition_id, leader_epoch, end_offset
    error_id => INT16
    partition_id => INT32
    end_offset => INT64
    leader_epoch => INT32              <<------ NEW
```

Proposed Changes

We propose to add leader epoch to the OffsetForLeaderEpoch response and slightly modify the replication protocol as follows. When the follower sends an OffsetForLeaderEpoch request, the leader responds with a pair (largest epoch less than or equal to the requested epoch, the end offset of this epoch). If the follower does not know about the leader epoch it receives in the response, it sends another OffsetForLeaderEpoch request with the next leader epoch (going backwards) it knows about. And so forth, until the follower receives a response with the leader epoch it knows about. The follower first truncates all offsets with epochs larger than the leader epoch returned from the leader, which becomes its new Log End Offset (LEO), and then truncates to leader's offset if it's smaller than its LEO. Basically, the idea is to converge to the largest common epoch, and then to the largest common offset in that epoch.

In the case of clean leader election, we still expect only one roundtrip of OffsetForLeaderEpoch request/response. In the corner case described above, the leader will responds with the largest epoch less than the requested epoch, which the follower will know about and do a truncation. For multiple roundtrips of OffsetForLeaderEpoch to happen, there must be at least three fast leader changes without changes in ISR in the first two, which should not happen in the clean leader election case (since we cannot have two prefered leader elections for the same topic partition back to back).

In the case of unclean leader election, one roundtrip of OffsetForLeaderEpoch request/response is still a common case. We may see multiple roundtrips of OffsetForLeaderEpoch request/response in rare cases: 2 roundtrips is required starting at 3 consecutive fast leader failovers (see Scenario 3 later in this section). There is no additional/special handling of unclean leader election config in the proposed approach. The only difference is that leader and follower may look further back than High Watermak in the leader epoch history to find the largest epoch both brokers know about, and then truncating based on the latest offset of that epoch. Theoretically, the solution lets both brokers to compare the complete epoch lineage between them, if needed. In practice, the common case would only require a few roundtrips for the solution to converge, usually exactly one roundtrip.

In more detail, the steps of the protocol are as follows:

1. In the current protocol, when the follower sends OffsetForLeaderEpoch request for the partition to the leader, the request includes the latest Leader Epoch in the follower's Leader Epoch Sequence. This step and steps before are the same as described in KIP-101.
2. The leader responds with the largest epoch less than or equal to the requested epoch (LeaderEpoch) and the end offset of this epoch (LastOffset).
3. If the follower has LeaderEpoch received from the leader in its Leader Epoch Sequence file, then we go to step 4. Otherwise,
    a. The follower truncates to the end offset of the largest epoch less than LeaderEpoch, and
    b. The follower sends OffsetForLeaderEpoch request with the largest epoch less than LeaderEpoch, and
    c. Steps 2 and 3 repeat until the follower receives the epoch it knows about (the epoch is in the follower's Leader Epoch Sequence file).
4. By following steps 2 and 3, the follower truncates all offsets with epochs larger than the epoch received from the leader (LeaderEpoch). In this step, the follower truncates its log to the leader's LastOffset, if leader's LastOffset is smaller than follower's Log End Offset.
5. The follower starts fetching from the leader and the remainder of the protocol remains unchanged.

For backward compatibility, in step 2, if the leader cannot find the LastOffset (e.g., the leader hasn't started tracking Leader Epoch yet, or the leader went back to the point where it hasn't started tracking Leader Epoch), it will respond with the UNKNOWN_OFFSET_FOR_LEADER_EPOCH. The follower will fall back to truncating to its high watermark.

Currently, any error with Offset For Leader Epoch Request will result in falling back to truncating using High Watermark, including a timeout. We will keep this behavior for now.

We will walk through several scenarios and show how the proposed solution solves the problem.

Scenario 1: Leader change due to preferred leader election followed by immediate leader change.

Consider the very first scenario described in this document, where broker A becomes a leader with epoch 3. It then appends a new entry in the range (21, 30). To remind, the logs look like this at this point:

Step 3

```
Broker A:
0: offsets [0, 10], leader epoch: 1
1: offsets [11, 20], leader epoch: 1
2: offsets: [21, 30], leader epoch: 3

Broker B:
0: offsets [0, 10], leader epoch: 1
1: offsets: [11, n], leader epoch: 2
```

On becoming follower, broker B sends OffsetForLeaderEpoch to broker A with leader_epoch 2. Broker A finds largest epoch <= 2 and log end offset in this epoch, and sends response {leader_epoch=1, offset = 21}. Broker B truncates all offsets for epochs > 1, in our example offsets [11, n], its LEO becomes 11. Since 21 > 11, broker B starts fetching from offset 11.

Scenario 2 (scenario 1 from KIP-101)

Here we show that scenarios fixed with KIP-101 will have the same behavior with this approach. Suppose we have brokers A and B. B is the leader. The following is their current state which also includes where their High Watermark is.

```
Broker A:
0: offsets [0, 10], leader epoch: 1
1: offsets [11, 20], leader epoch: 1
HW = 11

Broker B:
0: offsets [0, 10], leader epoch: 1
1: offsets [11, 20], leader epoch: 1
HW = 21
```

Broker A restarts.

Broker A sends OffsetForLeaderEpoch request to broker B with leader_epoch = 1. Broker B responds with {leader_epoch 1, offset 21}. Broker A does not truncate.

Scenario 3: Fast leader fail over with unclean leader election.

This is the second scenario described in motivation. Here is the reminder:

Unclean Leader Election 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.

At step 5, broker A sends OffsetForLeaderEpoch to broker B with leader_epoch 2. Broker B responds with (leader_epoch 1, offset 1). Broker A sends another OffsetForLeaderEpoch to broker B with leader_epoch 0. Broker B responds with UNKNOWN_OFFSET_FOR_LEADER_EPOCH since it exhausted all epochs (in a more common case, there will be some epoch they both know about). Broker A will truncate to its HW which is offet 0 and starts fetching from offset 0.

Compatibility, Deprecation, and Migration Plan

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.

Impact of topic compaction

The proposed solution requires that we preserve history in LeaderEpochSequence file. Note that this is also required in the current implementation if we want to guarantee no log divergence. The only reason for "losing" entries in LeaderEpoch file is if we actually lose LeaderEpoch file and have to rebuild it from the log. If we delete all offsets for a particular epoch for some topic partition, we may miss some entries in the LeaderEpochSequence file.

We will not do any changes to compaction logic in this KIP, but here is possible fixes to compaction logic:

1. Leave a tombstone in the log if we delete all offsets for some epoch, so that LeaderEpoch file can be rebuilt
2. Do not compact further than persistent HW.

Rejected Alternatives

Alternative 1: Multiple rounds of OffsetForLeaderEpoch between follower and leader until they find the largest epoch both of them know about.

This approach is very similar to the proposed solution, but it does not change a format of OffsetForLeaderEpoch request or response. Instead, if the follower sends OffsetForLeaderEpoch with the epoch now known to the leader, the leader responds with an error message that it does not have any offsets for a requested epoch. This will be a new error code, lets call it UNKNOWN_OFFSET_FOR_LEADER_EPOCH, to distinguish between UNDEFINE D_EPOCH_OFFSET error that results in falling back to truncating using the high watermark. The follower then sends another OffsetForLeaderEpoch request with the next epoch (going backwards), and so on, until the leader receives the OffsetForLeaderEpoch with the epoch it knows about and replies with the end offset for that epoch. At that point, the follower has (largest epoch both leader and follower know about, end offset of this epoch), and the follower truncates to that epoch and offset, same as in the proposed solution.

Here is how this alternative will work for Scenario 1 (Proposed Changes section): On becoming follower, broker B sends OffsetForLeaderEpoch to broker A with leader_epoch 2. Broker A responds with UNKNOWN_OFFSET_FOR_LEADER_EPOCH. Broker B sends another OffsetForLeaderEpoch to broker A with leader_epoch 1. Broker A responds with offset = 21. Broker B truncates all offsets for epochs > 1, in our example offsets [11, n], its LEO becomes 11. Since 21 > 11, broker B starts fetching from offset 11.

This alternative was rejected, because it requires more round trips of OffsetForLeaderEpoch and the proposed solution is cleaner in a way that it removes any ambiguity in the OffsetForLeaderEpoch response by indicating which leader epoch the end offset corresponds to. The proposed solution guarantees one roundtrip for clean leader election case, while the alternative may result in 2 roundtrips like in the scenario described above. The proposed solution does add 32bits to OffsetForLeaderEpoch response per partition to achieve this. Both solutions require a bump in the protocol version and have very similar complexity of the implementation.

Alternative 2: The follower sends all sequences of {leader_epoch, end_offset} between HW and LEO, and the leader responds with the offsets where the logs start diverging.

Modify OffsetForLeaderEpoch request to contain a sequence of (leader_epoch, end_offset) for each topic/partition. The sequence includes epochs between the follower's High Watermark (HW) and log end offset. The leader will look at the sequence starting with the latest epoch, and find the first matching leader_epoch, lets call it leader_epoch_match. The leader will respond with offset = min(end_offset for leader_epoch_match on the leader, end_offset for leader_epoch_match on the follower). The follower truncates based on the leader's offset, as in the current implementation. In the case of unlean leader election, it is not guaranteed that the follower's prefix before high watermak will match leader's prefix. So, it is possible that sequence of (leader_epoch, end_offset), with epochs between follower's high watermark and log end offset, does not contain a leader epoch that leader knows about. We will need to extend this approach so that the leader would then respond with UNKNOWN_OFFSET_FOR_LEADER_EPOCH, so that the follower sends a longer sequence of (leader_epoch, end_offset) with epochs below high watermark. We would need to tune this approach to decide on how long the sequence should be to minimize number of roundtrips vs. length of OffsetForLeaderEpoch.

Clean leader election example, Scenario 1 in Motivation: On becoming follower, broker B sends OffsetForLeaderEpoch to broker A with sequence (leader epoch 1, offset 11), (leader epoch 2, offset n+1). Broker B finds that first matching leader epoch is 1, and responds with min(its own end offset for epoch 1 = 21, followers end offset for epoch 1 = 11) = 11. Broker B truncates to offset = 11, and starts fetching from offset 11.

Unclean leader election example, Scenario 2 in Motivation. In that example, min.isr = 1 and replication factor = 2.

Unclean Leader Election Scenario

1. [LeaderEpoch0] Write a message to A (offset A:0), HW=1, Stop broker A. Bring up broker B which becomes leader
2. [LeaderEpoch1] Write a message to B (offset B:0), HW=1, Stop broker B. Bring up broker A which becomes leader
3. [LeaderEpoch2] Write a message to A (offset A:1), HW=2, Stop broker A. Bring up broker B which becomes leader
4. [LeaderEpoch3] Write a message to B (offset B:1), HW=2
5. Bring up broker A.

Since min.isr is 1, the high watermark advances on leader at every step. At step 5, Broker A sends OffsetForLeaderEpoch request to broker B with (leader_epoch = 1, offset = 1). Broker B responds with UNKNOWN_OFFSET_FOR_LEADER_EPOCH. Suppose, we decide second round to include several more epochs, so broker A sends OffsetForLeaderEpoch request to broker B with (leader_epoch = 1, offset = 1), (leader_epoc = 2, offset = 2). Broker B responds with UNKNOWN_OFFSET_FOR_LEADER_EPOCH since it exhausted all epochs (in a more common case, there will be some epoch they both know about). Broker A will truncate to its HW which is offet 0 and starts fetching from offset 0.

Comparison to the proposed solution. For clean leader election, this approach adds minimum 64bits to OffsetForLeaderEpoch request, and maximum (clean leader election edge case described above) 160bits to OffsetForLeaderEpoch request. We originally considered this approach vs. alternative #1, because this approach guarantees one OffsetForLeaderEpoch roundtrip in case of clean leader election. However, after we investigated the proposed solution, we found that proposed solution also guarantees one OffsetForLeaderEpoch roundtrip with a smaller change to the protocol: we cannot have more than one back-to-back leader change due to prefered leader election such that leader is not pushed out of the ISR, and so the follower will have at most one leader epoch unknown to the new leader, and so the proposed solution will be able to converge to the (largest common epoch, end offset in this epoch) in one roundtrip. This alternative provides a smaller number of OffsetForLeaderEpoch roundtrips in unclean leader election case. However, this advantage comes into play for rare cases, while adding more complexity and larger OffsetForLeaderEpoch request size. Notice that even in unclean leader election edge case above, this alternative requires 2 roundtrips, which is the same for the proposed solution (see Scenario 3 in Proposed Changes section). To get the advantage of less roundtrips, the scenario above shoud add one more "stop broker, bring up another broker, write a message" step.

Alternatives that address clean leader election only.

1. **If we guarantee that HW is always persisted after each leader change, then the follower sends OffsetForLeaderEpoch with the leader_epoch corresponding to current HW and the leader responds with the last offset of that epoch.** If we could guarantee that HW does not move back crossing the leader_epoch boundary (HW is persisted after each leader change), then the highest leader_epoch known to both leader and follower is a leader_epoch that corresponds to HW in the follower. In that case, we only need to change the protocol, where follower sends OffsetForLeaderEpoch with the leader_epoch corresponding to its HW (lets call it leader_epoch_hw), and the leader responds with the last offset of that epoch. The follower truncates all offsets that correspond to epochs > leader_epoch_hw, and then truncates offsets up to offset received from the leader.
2. **Push the old leader out of the ISR following a preferred leader change**. This approach is based on the observation that the reason for the above scenario (and similar scenarios) is that the leader change happened due to prefered leader election that did not cause the former leader to be kicked out of the ISR. We might do this by changing the ISR information in the LeaderAndISR request that is sent during preferred leader election.

Approach 1 is most simple, but it requires the guarantee that HW will not move backwards crossing leader epoch boundary. We will need to make sure that HW is persisted at each leader change. Otherwise, we may lose committed offsets. Flushing HW on every leader change seem to add more overhead than its worth (and if we compare to the proposed approach).

Approach 2 does not require any changes in protocol, but it does not address the unclean leader election.