KIP-320: Allow fetchers to detect and handle log truncation

- Status
- Motivation
- Proposed Changes
  - Leader Fetch Handling
  - Consumer Handling
  - Replica Handling
  - API Changes
  - Protocol Changes
    - Fetch
    - OffsetsForLeaderEpoch
    - Metadata
    - OffsetCommit
    - TxnOffsetCommit
    - OffsetFetch
    - ListOffsets
  - Offset Schema Changes
  - ACL Changes
- Compatibility, Deprecation, and Migration Plan
- Rejected Alternatives

Status

Current state: *Discussion*

Discussion thread: here

JIRA: KAFKA-6880 - Getting issue details... STATUS , KAFKA-7395 - Getting issue details... STATUS , KAFKA-7440 - Getting issue details... STATUS , KAFKA-7747 - Getting issue details... STATUS

Release: The broker side changes which improved fencing were released in 2.1.0. Client-side truncation detection and reset capability was released in 2.3.0

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

Motivation

This KIP aims to solve two related problems in the handling of fetch requests.

Consumer handling of unclean truncation: When unclean leader election is enabled, we may lose committed data. A consumer which is reading from the end of the log will typically see an out of range error, which will cause it to use its `auto.offset.reset` policy. To avoid losing data, users should use the "earliest" option, but that means consuming the log from the beginning.

It is also possible that prior to sending the next fetch, new data is written to the log so that the consumer's fetch offset becomes valid again. In this case, the consumer will just miss whatever data had been written between the truncation point and its fetch offset.

Neither behavior is ideal, but we tend to overlook it because the user has opted into weaker semantics by enabling unclean leader election. Unfortunately in some situations we have to enable unclean leader election in order to recover from serious faults on the brokers. Some users have also opted to keep unclean leader election enabled because they cannot sacrifice availability ever. We would like to offer better client semantics for these situations.

Inadequate Replica Fencing: We encountered a situation in KAFKA-6880 in which a deadlock caused a broker to enter a zombie state in which it was no longer registered in zookeeper. Nevertheless, its fetchers continued attempting to make progress by fetching from leaders. Since it was no longer receiving updates from the controller, the leader metadata became stale. However, the replica was still able to make progress and even rejoin the ISR as the assigned leader periodically became accurate again.

The problem in this situation is that the replica, which is no longer registered in Zookeeper, does not receive any notifications from the controller about leader changes. We depend on these notifications so that the follower can use our log truncation protocol to find the right starting offset for the new leader.

Proposed Changes

The lack of proper fencing has been a major weakness in Kafka, but the solution is straightforward since we already have the leader epoch to track leader changes. We will include the leader epoch in the Fetch request and the leader will reject the request if the epoch does not match its own.

The problem for the consumer is that it does not expect to see truncation. It assumes data below the high watermark is never lost. This assumption is obviously violated in the case of unclean leader election. The solution we propose is to let the consumer behave more like a follower and check for truncation after observing a leader change.

Below we describe in more detail the behavior changes for leaders, followers, and consumers.

Leader Fetch Handling

This KIP adds the leader epoch to the Fetch request. When a broker receives a fetch request, it will compare the requested epoch with its own. The fetch will only be permitted if the requested epoch matches the leader's epoch. If the requested epoch is older than the leader's, we will use a new FENCED_LEADER_EPOCH error code. If the epoch is newer than the leader's (for example in a KIP-232 scenario), we will use a new UNKNOWN_LEADER_EPOCH error code.

For consumers, we do not require strict fencing of fetch requests. We will support a sentinel value for the leader epoch which can be used to bypass the epoch validation. Additionally, the truncation check is optional for consumers. A client can choose to skip it and retain the current semantics.

In addition to fencing the fetch request, we also need stronger fencing for the truncation phase of the transition to becoming a follower. Without it, we cannot be sure that the leader's log does not change between the time that the truncation occurs and the follower begins fetching. To support this, we will add the current leader epoch to the OffsetForLeaderEpoch API. The leader will validate it similarly to fetch requests.

We will also change the leader behavior so that it is not permitted to add a replica to the ISR if it is marked as offline by the controller. By currently allowing this, we weaken acks=all semantics since the zombie contributes to the `min.isr` requirement, but is not actually eligible to become leader.

Consumer Handling

The proposal in this KIP is to have the consumer behave more like a follower. The consumer will obtain the current leader epoch using the Metadata API. When fetching from a new leader, the consumer will first check for truncation using the OffsetForLeaderEpoch API. In order to enable this, we need to keep track of the last epoch that was consumed. If we do not have one (e.g. because the user has seeked to a particular offset or because the message format is older), then the consumer will skip this step. To support this tracking, we will extend the OffsetCommit API to include the leader epoch if one is available.

Leader changes are detected either through a metadata refresh or in response to a FENCED_LEADER_EPOCH error. It is also possible that the consumer sees an UNKNOWN_LEADER_EPOCH in a fetch response if its metadata has gotten ahead of the leader.

This change in behavior has implications for the consumer's offset reset policy, which defines what the consumer should do if its fetch offset becomes out of range. With this KIP, the only case in which this is possible other than an out of range seek is if the consumer fetches from an offset earlier than the log start offset. By opting into an offset reset policy, the user allows for automatic adjustments to the fetch position, so we take advantage of this to to reset the offset as precisely as possible when log truncation is detected. In some pathological cases (e.g. multiple consecutive unclean leader elections), we may not be able to find the exact offset, but we should be able to get close by finding the starting offset of the next largest epoch that the leader is aware of. We propose in this KIP to change the behavior for both the "earliest" and "latest" reset modes to do this automatically as long as the message format supports lookup by leader epoch. The consumer will log a message to indicate that the truncation was detected, but will reset the position automatically.

If a user is not using an auto reset option, we will raise a `LogTruncationException` from `poll()` when log truncation is detected. This gives users the ability to reset state if needed or revert changes to downstream systems. The exception will include the partitions that were truncated and the offset of divergence as found above. This gives applications the ability to execute any logic to revert changes if needed and rewind the fetch offset. Users must handle this exception and reset the position of the consumer if no auto reset policy is enabled.

For consumers, we propose some additional extensions:

1. When the consumer needs to reset offsets, it uses the ListOffsets API to query the leader. To avoid querying stale leaders, we will add epoch fencing. Additionally, we will modify this API to return the corresponding epoch for any offsets looked up.
2. We will also provide the leader epoch in the offset commit and fetch APIs. This allows consumer groups to detect truncation across rebalances or restarts. Note that in cases like that found in KIP-232, it is possible for the leader epoch included in the committed offset to be ahead of the metadata that is known to the consumer. Consumers are expected to wait until the metadata has at least reached the epoch of the committed offset before checking for truncation.
3. For users that store offsets in an external system, we will provide APIs which expose the leader epoch of each record and we will provide an alternative `seek` API so that users can initialize the offset and leader epoch.

Replica Handling

This KIP adds replica fencing for both fetching and truncation. When a replica becomes a follower, it will attempt to find the truncation offset using the OffsetsForLeaderEpoch API. The new epoch will be included in this request so that the follower is ensured that it is truncating using the log from the leader in the correct epoch. Without this validation, it is possible to truncate using the log of a leader with stale log state, which can lead to log divergence.

After the follower has truncated its log, it will begin fetching as it does today. It will similarly include the current leader epoch, which will be validated by the leader. If the fetch response contains either the FENCED_REPLICA or UNKNOWN_LEADER_EPOCH error code, the follower will simply retry since both errors may be transient (e.g. if the propagation of the LeaderAndIsr request is delayed).

Note that we have implemented the this fetching model in TLA. So far, we have not found any errors checking this model.

Public Interfaces

API Changes

We will introduce a new exception type, which will be raised from `KafkaConsumer.poll(Duration)` as described above. This exception extends from `OffsetOutOfRangeException` for compatibility. If a user's offset reset policy is set to "none," they will still be able to catch `OffsetOutOfRangeException`. For new usage, users can catch the more specific exception type and use the `seek()` API to resume consumption. This exception is raised by the consumer after using the OffsetForLeaderEpoch API to find the offset of divergence. This offset is included as a field in the exception. Typically users handling this exception will seek to this offset.

```
/**
 * In the even of unclean leader election, the log will be truncated,
 * previously committed data will be lost, and new data will be written
 * over these offsets. When this happens, the consumer will detect the
 * truncation and raise this exception (if no automatic reset policy
 * has been defined) with the first offset to diverge from what the
 * consumer read.
 */
class LogTruncationException extends OffsetOutOfRangeException {
  /**
   * Get the truncation offsets for the partitions which were truncated.
   * This is the first offset which is known to diverge from what the consumer read.
   */
  Map<TopicPartition, OffsetAndMetadata> truncationOffsets();
}
```

We will also add new retriable exceptions for the UNKNOWN_LEADER_EPOCH and FENCED_LEADER_EPOCH error codes:

```
class UnknownLeaderEpochException extends RetriableException {}

class FencedLeaderEpochException extends InvalidMetadataException {}
```

This will be located in the public `errors` package, but the consumer will internally retry when it receives this error.

The leader epoch will be exposed in the `ConsumerRecord` and `OffsetAndMetadata` objects.

```
class ConsumerRecord<K, V> {
  /**
   * Get the leader epoch or empty if it is unknown.
   */
  Optional<Integer> leaderEpoch();
}

class OffsetAndMetadata {
  /**
   * New constructor including optional leader epoch. Old constructors
   * will still be supported and will use Optional.empty() as the default
   * leader epoch.
   */
  OffsetAndMetadata(long offset, String metadata, Optional<Integer> leaderEpoch);

  /**
   * Get the leader epoch of the previously consumed record (if one is known).
   * Log truncation is detected if there exists a leader epoch which is larger
   * than this epoch and begins at an offset earlier than the committed offset.
   */
  Optional<Integer> leaderEpoch();
}
```

We will also have a new API to support seeking to an offset and leader epoch. This is required in order to support storage of offsets in an external store. When the consumer is initialized, the user will call `seek()` with the offset and leader epoch that was stored. The consumer will initialize the position of the consumer using this API.

Like the other `seek()` overload, this method does not make any remote calls. If a leader epoch has been provided, then in the next call to `poll()`, the consumer will use the OffsetForLeaderEpoch API to check for truncation. If the log has been truncated since the time the offsets were stored, the next call to `poll()` will raise a `LogTruncationException` as described above.

```
/**
 * Seek to an offset and initialize the leader epoch (if present).
 */
void seek(TopicPartition partition, OffsetAndMetadata offset);
```

To make the external storage use case simpler, we will provide a helper to ConsumerRecords to get the next offsets. This simplifies commit logic for consumers which only commit offsets after consuming full batches.

```
class ConsumerRecords<K, V> {
  /**
   * Get the next offsets that the consumer will consumer.
   * This can be passed directly to a commitSync(), for example,
   * after the full batch has been consumed. For finer-grained,
   * offset tracking, you should use the offset information from
   * the individual ConsumerRecord instances.
   */
  Map<TopicPartition, OffsetAndMetadata> nextOffsets();
}
```

Finally, we may as well protect the offsets found through the `offsetsForTimes()` API. An unclean leader election may invalidate the results of a lookup by time, so it would be unsafe to use `seek()` without considering the epoch information for the returned offsets. To support this, we will add the leader epoch to the `OffsetAndTimestamp` object which is returned from `offsetsForTimes()`.

```
class OffsetAndTimestamp {
  /**
   * Get the leader epoch corresponding to the offset that was
   * found (if one exists). This can be provided to seek() to
   * ensure that the log hasn't been truncated prior to fetching.
   */
  Optional<Integer> leaderEpoch();
}
```

Protocol Changes

Fetch

We will bump the fetch request version in order to include the current leader epoch. For older versions, we will skip epoch validation as before.

The new schema is given below:

```
FetchRequest => MaxWaitTime ReplicaId MinBytes IsolationLevel FetchSessionId FetchSessionEpoch [Topics]
[RemovedTopics]
  MaxWaitTime => INT32
  ReplicaId => INT32
  MinBytes => INT32
  IsolationLevel => INT8
  FetchSessionId => INT32
  FetchSessionEpoch => INT32
  Topics => TopicName Partitions
          TopicName => STRING
          Partitions => [Partition FetchOffset StartOffset LeaderEpoch MaxBytes]
                  Partition => INT32
                CurrentLeaderEpoch => INT32   // New
                  FetchOffset => INT64
                  StartOffset => INT64
                  MaxBytes => INT32
  RemovedTopics => RemovedTopicName [RemovedPartition]
          RemovedTopicName => STRING
          RemovedPartition => INT32
```

The response schema will not change, but we will have two new error codes as mentioned above:

1. **FENCED_LEADER_EPOCH:** The replica has a lower epoch than the leader. This is retriable.
2. **UNKNOWN_LEADER_EPOCH:** This is a retriable error code which indicates that the epoch in the fetch request was larger than any known by the broker.

Previously, we would return the NOT_LEADER_FOR_PARTITION error code if a follower receives an unexpected Fetch request. In this case, the requested epoch is different from what the follower has, so we can now return one of the error codes above, which contain more specific information.

OffsetsForLeaderEpoch

The changes to the OffsetsForLeaderEpoch request API are similar.

```
OffsetsForLeaderEpochRequest => [Topic]
  Topic => TopicName [Partition]
    TopicName => STRING
    Partition => PartitionId CurrentLeaderEpoch LeaderEpoch
      PartitionId => INT32
      CurrentLeaderEpoch => INT32 // New
      LeaderEpoch => INT32
```

If the current leader epoch does not match that of the leader, then we will send either FENCED_REPLICA or UNKNOWN_LEADER_EPOCH as we do for the Fetch API.

Since this API is no longer exclusively an inter-broker API, we will also add the throttle time to the response schema.

```
OffsetsForLeaderEpochResponse => ThrottleTimeMs [TopicMetadata]
  ThrottleTimeMs => INT32 // New
  TopicMetadata => TopicName PartitionMetadata
    TopicName => STRING
    PartitionMetadata => [ErrorCode PartitionId LeaderEpoch EndOffset]
      ErrorCode => INT16
      PartitionId => INT32
      LeaderEpoch => INT32
      EndOffset => INT64
```

Metadata

The metadata response will be extended to include the leader epoch. This enables stronger fencing for consumers. We can enable similar protection in producers in the future, but that is out of the scope of this KIP.

```
MetadataResponse => ThrottleTimeMs Brokers ClusterId ControllerId [TopicMetadata]
  ThrottleTimeMs => INT32
  Brokers => [MetadataBroker]
  ClusterId => NULLABLE_STRING
  ControllerId => INT32

TopicMetadata => ErrorCode TopicName IsInternal [PartitionMetadata]
  ErrorCode => INT16
  TopicName => STRING
  IsInternal => BOOLEAN

PartitionMetadata => ErrorCode PartitionId Leader LeaderEpoch Replicas ISR OfflineReplicas
  ErrorCode => INT16
  PartitionId => INT32
  Leader => INT32
  LeaderEpoch => INT32 // New
  Replicas => [INT32]
  ISR => [INT32]
  OfflineReplicas => [INT32]
```

There are no changes to the Metadata request schema.

OffsetCommit

The new OffsetCommit request schema is provided below. A field for the leader epoch has been added. Note that this is the epoch of the previously fetched record. The response schema matches the previous version.

```
OffsetCommitRequest => GroupId Generation MemberId [TopicName [Partition Offset LeaderEpoch Metadata]]
  GroupId => STRING
  Generation => INT32
  MemberId => STRING
  RetentionTime => INT64
  TopicName => STRING
  Partition => INT32
  Offset => INT64
  LeaderEpoch => INT32  // New
  Metadata => STRING
```

TxnOffsetCommit

Similarly, we need to add the leader epoch to the TxnOffsetCommit API, which is used by transactional producers.

```
TxnOffsetCommitRequest => TransactionalId GroupId ProducerId ProducerEpoch Topics
  GroupId => STRING
  Generation => INT32
  MemberId => STRING
  RetentionTime => INT64
  Topics => [TopicName [Partition Offset LeaderEpoch Metadata]]
    TopicName => STRING
    Partition => INT32
    Offset => INT64
    LeaderEpoch => INT32  // New
    Metadata => STRING
```

OffsetFetch

The OffsetFetch response schema will be similarly modified. The request schema will remain the same.

```
OffsetFetchResponse => ThrottleTimeMs Topics ErrorCode
  ThrottleTimeMs => INT64
  Topics => [TopicName [Partition Offset LeaderEpoch Metadata ErrorCode]]
          TopicName => STRING
          Partition => INT32
          Offset => INT64
          LeaderEpoch => INT32  // New
          Metadata => STRING
    ErrorCode => INT16
  ErrorCode => INT16
```

ListOffsets

We need two changes to the ListOffsets API. First, we add the current leader epoch to the request schema in order to protect clients from querying stale leaders.

```
ListOffsetRequest => ReplicaId [TopicName [Partition CurrentLeaderEpoch Timestamp]]
 ReplicaId => INT32
 TopicName => STRING
 Partition => INT32
 CurrentLeaderEpoch => INT32 // New
 Timestamp => INT64
```

Second, we add the leader epoch to the response that corresponds with the returned offset. The client can use this to reconcile the log prior to fetching from a new leader.

```
ListOffsetResponse => ThrottleTimeMs Topics
  ThrottleTimeMs => INT64
  Topics => [TopicName [Partition ErrorCode Timestamp Offset LeaderEpoch]]
    TopicName => STRING
    Partition => INT32
    ErrorCode => INT16
    Timestamp => INT64
    Offset => INT64
    LeaderEpoch => INT32  // New
```

As with the Fetch and OffsetForLeaderEpoch APIs, the response will support the FENCED_LEADER_EPOCH and UNKNOWN_LEADER_EPOCH error codes.

Offset Schema Changes

This KIP introduces a new schema version for committed offsets in the internal __consumer_offsets topic. This is not a public API, but we mention it for compatibility implications. The version will be bumped to 3. The new schema is given below and corresponds with the changes to the OffsetCommit APIs:

```
Offset Commit Value Schema (Version: 3) =>
  Offset => INT64
  LeaderEpoch => INT32 // New
  Metadata => STRING
  CommitTimestamp => INT64
```

As with previous changes to this schema, we will not begin using the new schema after an upgrade until the inter-broker version has been updated. This ensures that replicas on older versions do not fail if they need to parse the new schema version.

ACL Changes

Currently the OffsetForLeaderEpoch request is restricted to inter-broker communication. It requires authorization to the Cluster resource. As part of this KIP, we will change this to require Describe access on the Topic resource. For backwards compatibility, we will continue to allow the old authorization.

Compatibility, Deprecation, and Migration Plan

Older versions of the the modified APIs will continue to work as expected. When using older message format versions, which do not support leader epoch in the message format, we will use a sentinel value (-1) in the APIs that expose it.

Rejected Alternatives

- When the consumer cannot find the precise truncation point, another option is to use the timestamp of the last consumed message in order to find the right offset to reset to. One benefit of this is that it works for older message formats. The downsides are 1) it complicates the API and 2) the truncation point determined using timestamp is not as accurate as what can be determined using the leader epoch. Ultimately we decided it was not worthwhile complicating the APIs for the old message format.
- Initially, this proposal suggested that the follower should include the expected next epoch in the fetch request rather than the current epoch. Unfortunately, model checking showed a weakness in this approach. To make the advancement of the high watermark safe, the leader must be be able to guarantee that all followers in the ISR are on the correct epoch. Any disagreement about the current leader epoch and the ISR can lead to the loss of committed data. See https://github.com/hachikuji/kafka-specification/blob/master/Kip320FirstTry.tla for more detail. We considered including both the current epoch and the expected epoch, but the truncation only occurs on leadership changes, so checking on every fetch was not necessary.