KIP-951: Leader discovery optimisations for the client

Status

Current state: *"Accepted"*

Discussion thread: *here , and here*

JIRA: ⚠ **Unable to render Jira issues macro, execution error.**

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

Motivation

When the leader of a partition changes, the end-to-end latency of produce or fetch client request can increase significantly as a result of the work required to discover the new leader.

The client’s process to recover from NOT_LEADER_OR_FOLLOWER or FENCED_LEADER_EPOCH errors looks like this:

1. Asynchronously refresh the metadata for all topics using the Metadata RPC.
2. Retry the Produce or Fetch RPC after the RETRY_BACKOFF_MS_CONFIG.
3. If the RPC failed with NOT_LEADER_OR_FOLLOWER or FENCED_LEADER_EPOCH, retry again from step 1.

This approach has a few downsides:

- A single Metadata RPC can be slow (order of few 100 milliseconds in extreme cases).
    - Digging deeper into this we found that head of the line blocking from slow produce requests preventing responses from being sent is the main source of these slow metadata RPC's.
- It might take multiple attempts to refresh the metadata if the latest metadata hasn’t yet propagated throughout the cluster.
- The produce and fetch requests have their own delayed retries on failed attempts (default is 100ms in Java client), to avoid overloading the Kafka brokers with failing RPCs.

Our goal is to minimize the time taken to discover a new leader, and use that information as soon as possible on the client. This can be beneficial in a number of situations such as cluster rolls, reassignments, or preferred leader election.

~~***Note***, performance tests are being done to show the improvements with the proposed changes. Results to follow.~~ Added to section Benchmark results.

Proposed Changes

This KIP proposes to make new leader information available in the ProduceResponse and FetchResponse when a newer leader is known by the receiving broker. This should help us eliminate the metadata refresh from the critical path of the Produce and Fetch requests, which can be very latency-sensitive. This is an optimization mentioned in KIP-595, which uses a similar technique but only for the metadata log and the Fetch RPC.

Following sections will consider the situations in which leadership changes. This could happen due to a controlled shutdown, replica reassignment, or preferred leader election.

Broker

When the leader of a partition changes from the old leader to a new leader, the old leader can inform the client of the new leader’s LeaderEpoch & LeaderId via ProduceResponse and FetchResponse if it has that information. Notably, this information is sent together with the existing error codes of NOT_LEADER_OR_FOLLOWER and FENCED_LEADER_EPOCH. The new leader information is obtained from either the replica partition state (if the receiving broker continues to be a replica of the partition) or from the broker’s metadata cache (if the receiving broker is not a replica for the partition because of reassignment). These new leader fields will be optional (tagged), if the old leader does not have this information for any reason it does not need to populate them and the client would do a full metadata refresh.

Client

The client will only accept the new leader information(LeaderId & LeaderEpoch) only if it advances its view of the new leader(i.e. new-leader’s Epoch should be greater than what client knows already) and use it in subsequent retries of the Produce & Fetch requests. On the client, it can happen that the subsequent metadata refreshes return stale leader information, if the latest metadata isn’t yet fully propagated to the entire cluster. The client will make sure that new leader information isn’t overridden by the stale leader’s information(again comparing LeaderEpochs), which is the existing behaviour of Kafka Java client.

Even though client will receive the new leader information in the ProduceResponse & FetchResponse when leader changes, but same as the existing behaviour of the Kafka Java client, it will request expedited metadata-refresh done asynchronously. Since leadership change will likely affect many partitions, so future requests to such partitions will benefit from the upto date leadership information, and reduce requests going to old leaders.

For Produce, if new leader info is available in the response along with errors(NOT_LEADER_OR_FOLLOWER or FENCED_LEADER_EPOCH) that advances client's view of the leadership, client would no longer back off up to RETRY_BACKOFF_MS_CONFIG before retrying the failed batch. This immediate retry is appealing as the client is going to retry on a different broker and it is likely to succeed because it is retrying on a newer leader. On the other hand, subsequent retries to the same new LeaderEpoch should still continue to be subject to clients' backoff strategy.

In the ProduceResponse & FetchResponse, new leader’s connection parameters (host & port) will also be supplied along with LeaderId & LeaderEpoch, upon returning errors NOT_LEADER_OR_FOLLOWER and FENCED_LEADER_EPOCH . This will be useful in situations when new leader's connection parameters are not part of client's metadata cache or stale. This can happen in a scenario where an existing broker restarts, or a new broker is added to the cluster. For instance when existing broker restarts, any subsequent metadata refresh on the client, will remove the connection parameters from its cache while broker is shutdown. After broker is restarted, and this broker becomes the new leader, client will require connection parameters along with LeaderId and LeaderEpoch to connect to this new leader in a subsequent retry of the failed produce or fetch request. To this effect, host & port is returned for all such brokers. Additionally rack is returned to be consistent with the broker information returned in MetadataResponse. This will simply override the previously stored node information in the metadata cached locally.

Public Interfaces

FetchResponse Message

FetchResponse already contains CurrentLeader introduced in KIP-595, which will be used to propagate LeaderId & LeaderEpoch. Additionally NodeEndpoints is introduced to propagate host, port & rack for the current-leaders enumerated across all PartitionDatas. The version has been bumped to 16. Allthough note that CurrentLeader was introduced in version 12, it will be supported in Java client from 16. Note that new leader information fields, CurrentLeader & NodeEndpoints are tagged as minor optimisation to save few bytes on the network. Since these fields are optional version 16 onwards. Broker would only set these fields in case of specific errors(NOT_LEADER_OR_FOLLOWER or FENCED_LEADER_EPOCH), and only if it has this information available.

```
{
  "apiKey": 1,
  "type": "response",
  "name": "FetchResponse",
  "validVersions": "0-16",
  "flexibleVersions": "12+",
  "fields": [
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "1+", "ignorable": true,
      "about": "The duration in milliseconds for which the request was throttled due to a quota violation, or
zero if the request did not violate any quota." },
    { "name": "ErrorCode", "type": "int16", "versions": "7+", "ignorable": true,
      "about": "The top level response error code." },
    { "name": "SessionId", "type": "int32", "versions": "7+", "default": "0", "ignorable": false,
      "about": "The fetch session ID, or 0 if this is not part of a fetch session." },
    { "name": "Responses", "type": "[]FetchableTopicResponse", "versions": "0+",
      "about": "The response topics.", "fields": [
      { "name": "Topic", "type": "string", "versions": "0-12", "ignorable": true, "entityType": "topicName",
        "about": "The topic name." },
      { "name": "TopicId", "type": "uuid", "versions": "13+", "ignorable": true, "about": "The unique topic
ID"},
      { "name": "Partitions", "type": "[]PartitionData", "versions": "0+",
        "about": "The topic partitions.", "fields": [
        { "name": "PartitionIndex", "type": "int32", "versions": "0+",
          "about": "The partition index." },
        { "name": "ErrorCode", "type": "int16", "versions": "0+",
          "about": "The error code, or 0 if there was no fetch error." },
        { "name": "HighWatermark", "type": "int64", "versions": "0+",
          "about": "The current high water mark." },
        { "name": "LastStableOffset", "type": "int64", "versions": "4+", "default": "-1", "ignorable": true,
          "about": "The last stable offset (or LSO) of the partition. This is the last offset such that the
state of all transactional records prior to this offset have been decided (ABORTED or COMMITTED)" },
        { "name": "LogStartOffset", "type": "int64", "versions": "5+", "default": "-1", "ignorable": true,
          "about": "The current log start offset." },
        { "name": "DivergingEpoch", "type": "EpochEndOffset", "versions": "12+", "taggedVersions": "12+",
"tag": 0,
          "about": "In case divergence is detected based on the `LastFetchedEpoch` and `FetchOffset` in the
request, this field indicates the largest epoch and its end offset such that subsequent records are known to
diverge",
          "fields": [
            { "name": "Epoch", "type": "int32", "versions": "12+", "default": "-1" },
            { "name": "EndOffset", "type": "int64", "versions": "12+", "default": "-1" }
        ]},
        { "name": "CurrentLeader", "type": "LeaderIdAndEpoch",
          "versions": "12+", "taggedVersions": "12+", "tag": 1, "fields": [
          { "name": "LeaderId", "type": "int32", "versions": "12+", "default": "-1", "entityType": "brokerId",
            "about": "The ID of the current leader or -1 if the leader is unknown."},
          { "name": "LeaderEpoch", "type": "int32", "versions": "12+", "default": "-1",
            "about": "The latest known leader epoch"}
        ]},
        { "name": "SnapshotId", "type": "SnapshotId",
          "versions": "12+", "taggedVersions": "12+", "tag": 2,
          "about": "In the case of fetching an offset less than the LogStartOffset, this is the end offset and
epoch that should be used in the FetchSnapshot request.",
          "fields": [
            { "name": "EndOffset", "type": "int64", "versions": "0+", "default": "-1" },
            { "name": "Epoch", "type": "int32", "versions": "0+", "default": "-1" }
        ]},
        { "name": "AbortedTransactions", "type": "[]AbortedTransaction", "versions": "4+", "nullableVersions":
"4+", "ignorable": true,
          "about": "The aborted transactions.",  "fields": [
          { "name": "ProducerId", "type": "int64", "versions": "4+", "entityType": "producerId",
            "about": "The producer id associated with the aborted transaction." },
          { "name": "FirstOffset", "type": "int64", "versions": "4+",
            "about": "The first offset in the aborted transaction." }
        ]},
        { "name": "PreferredReadReplica", "type": "int32", "versions": "11+", "default": "-1", "ignorable":
false, "entityType": "brokerId",
          "about": "The preferred read replica for the consumer to use on its next fetch request"},
        { "name": "Records", "type": "records", "versions": "0+", "nullableVersions": "0+", "about": "The
record data."}
      ]}
    ]},
    // ---------- Start new field ----------
    { "name": "NodeEndpoints", "type": "[]NodeEndpoint", "versions": "16+", "taggedVersions": "16+", "tag": 0,
      "about": "Endpoints for all current-leaders enumerated in PartitionData, with errors
NOT_LEADER_OR_FOLLOWER & FENCED_LEADER_EPOCH.", "fields": [
      { "name": "NodeId", "type": "int32", "versions": "16+",
        "mapKey": true, "entityType": "brokerId", "about": "The ID of the associated node."},
      { "name": "Host", "type": "string", "versions": "16+",
        "about": "The node's hostname." },
      { "name": "Port", "type": "int32", "versions": "16+",
        "about": "The node's port." },
      { "name": "Rack", "type": "string", "versions": "16+", "nullableVersions": "16+", "default": "null",
        "about": "The rack of the node, or null if it has not been assigned to a rack." }
    ]}
    // ---------- End new field ----------
  ]
}
```

FetchRequest Message

Version would be bumped to 16 to match response, other than no change.

ProduceResponse Message

For ProduceResponse, similarly will add CurrentLeader & NodeEndpoints to convey new leader info. Similarly these fields are tagged and version is bumped to 10.

```
{
  "apiKey": 0,
  "type": "response",
  "name": "ProduceResponse",
  "validVersions": "0-10",
  "flexibleVersions": "9+",
  "fields": [
    { "name": "Responses", "type": "[]TopicProduceResponse", "versions": "0+",
      "about": "Each produce response", "fields": [
      { "name": "Name", "type": "string", "versions": "0+", "entityType": "topicName", "mapKey": true,
        "about": "The topic name" },
      { "name": "PartitionResponses", "type": "[]PartitionProduceResponse", "versions": "0+",
        "about": "Each partition that we produced to within the topic.", "fields": [
        { "name": "Index", "type": "int32", "versions": "0+",
          "about": "The partition index." },
        { "name": "ErrorCode", "type": "int16", "versions": "0+",
          "about": "The error code, or 0 if there was no error." },
        { "name": "BaseOffset", "type": "int64", "versions": "0+",
          "about": "The base offset." },
        { "name": "LogAppendTimeMs", "type": "int64", "versions": "2+", "default": "-1", "ignorable": true,
          "about": "The timestamp returned by broker after appending the messages. If CreateTime is used for
the topic, the timestamp will be -1.  If LogAppendTime is used for the topic, the timestamp will be the broker
local time when the messages are appended." },
        { "name": "LogStartOffset", "type": "int64", "versions": "5+", "default": "-1", "ignorable": true,
          "about": "The log start offset." },
        { "name": "RecordErrors", "type": "[]BatchIndexAndErrorMessage", "versions": "8+", "ignorable": true,
          "about": "The batch indices of records that caused the batch to be dropped", "fields": [
          { "name": "BatchIndex", "type": "int32", "versions":  "8+",
            "about": "The batch index of the record that cause the batch to be dropped" },
          { "name": "BatchIndexErrorMessage", "type": "string", "default": "null", "versions": "8+",
"nullableVersions": "8+",
            "about": "The error message of the record that caused the batch to be dropped"}
        ]},
        { "name":  "ErrorMessage", "type": "string", "default": "null", "versions": "8+", "nullableVersions":
"8+", "ignorable":  true,
          "about":  "The global error message summarizing the common root cause of the records that caused the
batch to be dropped"},
                // ---------- Start new field ----------
        { "name": "CurrentLeader", "type": "LeaderIdAndEpoch",
          "versions": "10+", "taggedVersions": "10+", "tag": 0, "fields": [
            { "name": "LeaderId", "type": "int32", "versions": "10+", "default": "-1", "entityType": "brokerId",
              "about": "The ID of the current leader or -1 if the leader is unknown."},
            { "name": "LeaderEpoch", "type": "int32", "versions": "10+", "default": "-1",
              "about": "The latest known leader epoch"}
        ]}
                  // ---------- End new field ----------
      ]}
    ]},
    { "name": "ThrottleTimeMs", "type": "int32", "versions": "1+", "ignorable": true, "default": "0",
      "about": "The duration in milliseconds for which the request was throttled due to a quota violation, or
zero if the request did not violate any quota." },
    // ---------- Start new field ----------
    { "name": "NodeEndpoints", "type": "[]NodeEndpoint", "versions": "10+", "taggedVersions": "10+", "tag": 0,
      "about": "Endpoints for all current-leaders enumerated in PartitionProduceResponses, with errors
NOT_LEADER_OR_FOLLOWER & FENCED_LEADER_EPOCH.", "fields": [
      { "name": "NodeId", "type": "int32", "versions": "10+",
        "mapKey": true, "entityType": "brokerId", "about": "The ID of the associated node."},
      { "name": "Host", "type": "string", "versions": "10+",
        "about": "The node's hostname." },
      { "name": "Port", "type": "int32", "versions": "10+",
        "about": "The node's port." },
      { "name": "Rack", "type": "string", "versions": "10+", "nullableVersions": "10+", "default": "null",
        "about": "The rack of the node, or null if it has not been assigned to a rack." }
     // ---------- End new field ----------
    ]}
  ]
}
```

ProduceRequest Message

Version would be bumped to 10 to match response, other than no change.

Benchmark Results

These are the benchmark results for leader discovery optimization. Two sets of tests were performed, a shorter micro-benchmark targeting redirection more directly and a longer running benchmark simulating a roll. The micro benchmark shows an 88% reduction in p99.9 produce latency while the roll simulation shows up to a 5% improvement in p99 E2E with acks=all.

Micro-Benchmark

Performance was tested using the kafka-producer-perf-test.sh script and reassigning leadership of all partitions of a 100 partition topic. We see an end-to-end reduction in the p99.9 produce latency of the overall run of 88%, from 1675ms to 215ms (average of 3 runs). Digging deeper into the baseline, & rejected alternate, metadata RPC becomes slow due to slower produce RPCs ahead of it to a given broker. This results into metadata refresh being slow for baseline & rejected alternate over all, resulting into higher latencies. KIP-951 doesn't rely on the Metadata RPC for the new leader information, hence sees better latencies.

Baseline

Run 1: 40000000 records sent, 99997.750051 records/sec (95.37 MB/sec), 12.56 ms avg latency, 8087.00 ms max latency, 6 ms 50th, 8 ms 95th, 12 ms 99th, **2967 ms 99.9th.**
Run 2: 40000000 records sent, 99998.250031 records/sec (95.37 MB/sec), 15.51 ms avg latency, 11652.00 ms max latency, 6 ms 50th, 8 ms 95th, 13 ms 99th, **859 ms 99.9th.**
Run 3: 40000000 records sent, 99998.000040 records/sec (95.37 MB/sec), 8.63 ms avg latency, 3224.00 ms max latency, 6 ms 50th, 8 ms 95th, 14 ms 99th, **1201 ms 99.9th.**

KIP-951

Run 1: 40000000 records sent, 99998.000040 records/sec (95.37 MB/sec), 8.51 ms avg latency, 2949.00 ms max latency, 6 ms 50th, 8 ms 95th, 15 ms 99th, **346 ms 99.9th.**
Run 2: 40000000 records sent, 99998.000040 records/sec (95.37 MB/sec), 15.11 ms avg latency, 11118.00 ms max latency, 6 ms 50th, 8 ms 95th, 12 ms 99th, **174 ms 99.9th.**
Run 3: 40000000 records sent, 99997.500062 records/sec (95.37 MB/sec), 11.71 ms avg latency, 6933.00 ms max latency, 6 ms 50th, 8 ms 95th, 15 ms 99th, **125 ms 99.9th.**

Rejected Alternative

Run 1: 40000000 records sent, 99997.500062 records/sec (95.37 MB/sec), 9.77 ms avg latency, 6756.00 ms max latency, 6 ms 50th, 8 ms 95th, 14 ms 99th, **1781 ms 99.9th.**
Run 2: 40000000 records sent, 99997.750051 records/sec (95.37 MB/sec), 11.07 ms avg latency, 7409.00 ms max latency, 5 ms 50th, 7 ms 95th, 11 ms 99th, **1934 ms 99.9th.**
Run 3: 40000000 records sent, 99997.750051 records/sec (95.37 MB/sec), 16.26 ms avg latency, 14211.00 ms max latency, 6 ms 50th, 9 ms 95th, 16 ms 99th, **5352 ms 99.9th.**

Workload Details

All tests are run on 6 m5.xlarge Apache Kafka brokers running with Kraft as the metadata quorum in 3 m5.xlarge instances. The client is a m5.xlarge instance running the kafka-producer-perf-test.sh script with the following parameters. The test lasts for around 6 minutes, during which all partitions of the 100 partition test topic are reassigned by rotating the replica set from the previous leader to the next in-line replica.

```
./bin/kafka-producer-perf-test.sh --producer.config ... --throughput 100000 --record-size 1000  --num-records 40000000 --topic ...  --producer-props acks=all linger.ms=0 batch.size=16384 --print-metrics
```

Roll Simulation

Performance was tested on low partition and high partition workloads, more details on the setup are under Workload Details. We see up to 5% improvement in E2E latencies when run with acks=all and up to 9% improvement in produce latencies when run with acks=1. Our hypothesis for why the improvement when run with acks=1 is higher than acks=all is that metadata convergence delays for partition movement on the server side during software upgrades are higher than the client side redirection in the KIP which impacts ack=all requests more than ack=1. We believe this is also the reason why low partitions workload shows better improvements in ack=1 than high partitions workload. The results are averaged over 2 runs.

Latency improvement of workloads run with acks=all

	Baseline latency (ms) avg (run1, run2)	Optimized latency (ms) avg (run1, run2)	Improvement
High Partitions			

p99 E2E	188 (183, 193)	184 (177, 191)	2.1%
p99 Produce	155.65 (153.1, 158.2)	151.8 (148.3, 155.3)	2.5%
Low Partitions			
p99 E2E	393 (402, 384)	374.5 (349, 400)	4.7%
p99 Produce	390.95 (396.9, 385)	374.35 (348.4, 400.3)	4.2%

Latency improvement of workloads run with acks=1

	Baseline latency (ms) avg (run1, run2)	**Optimized latency (ms) avg (run1, run2)**	**Improvement**
High Partitions			
p99 E2E	106.5 (111, 102)	101 (104, 98)	5.2%
p99 Produce	84.7 (85.8, 83.6)	83.3 (82.5, 84.1)	1.7%
Low Partitions			
p99 E2E	12.5 (13, 12)	12.5 (11, 14)	0%
p99 Produce	3.25 (3.3, 3.2)	2.95 (3, 2.9)	9.2%

Workload Details

All tests are run on 6 m5.xlarge Apache Kafka brokers running with Kraft as the metadata quorum in 3 m5.xlarge instances. The clients are 6 m5.xlarge instances running the OpenMessagingBenchmark. The test is run for 70 minutes, during which the brokers are restarted one by one with a 10 minute interval between restarts.

High partitions workload parameters:

- One 9,000 partition topic
- ~12 MB/s ingress
  - 512b message size
  - 23,400 messages/s distributed over 60 producers
- ~36 MB/s egress
  - 3 subscriptions, 60 consumers per subscription

Low partitions workload parameters:

- One 108 partition topic
- ~120 MB/s ingress
  - 512b message size
  - 234,000 messages/s distributed over 6 producers
- ~360 MB/s egress
  - 3 subscriptions, 6 consumers per subscription

Compatibility, Deprecation, and Migration Plan

This change is backwards compatible. If the current leader information aren't set in the ProduceResponse or FetchResponse, then the client falls back to the original behaviour of waiting for the metadata to be refreshed and consistent through the Metadata RPC call.

Rejected Alternatives

For a produce-request, another idea considered was to fetch new leader on the client using the usual Metadata RPC call, once produce-batch fails with NOT_LEADER_OR_FOLLOWER or FENCED_LEADER_EPOCH. And save time on the client by avoiding the static retry delay(RETRY_BACKOFF_MS_CONFIG) on a failed request, instead retry immediately as soon as possible when the new leader is available for the partition. Consider the total time taken for a produce-path, when leader changes -

1. Total Time for alternative = Produce RPC(client to old leader) + Time taken to refresh metadata to get new leader + Produce RPC(client to new leader)
2. Total Time for the favored proposed changes = Produce RPC(client to old leader, ProduceResponse has new leader) + Produce RPC(client to new leader)

It can be clearly seen alternative has an extra-component, i.e. time taken to refresh metadata to get new leader. This time has a lower bound of 1 single Metadata RPC call, but degrades to many such calls if metadata propagation is slower through the cluster. Due to this, proposed changes, is the preferred alternative.