KIP-32 - Add timestamps to Kafka message

Status

Current state: Accepted

Discussion thread: *here*

JIRA: *KAFKA-2511*

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

Motivation

This KIP tries to address the following issues in Kafka.

1. Log retention might not be honored: Log retention is currently at the log segment level, and is driven off the last modification time of a log segment. This approach does not quite work when a replica reassignment happens because the newly created log segment will effectively have its modification time reset to now.
2. Log rolling might break for a newly created replica as well because of the same reason as (1).
3. Some use cases such as streaming processing needs a timestamp in messages.

To solve the above issues, we propose to add a timestamp to Kafka messages.

This KIP is a distilled/improved version of an earlier discussion that we started.

This KIP will likely be implemented with KIP-31, if possible, to avoid changing wire protocol twice.

This KIP is closely related to KIP-33, which is about building time index as well as use cases on top of the time index using the timestamp added in this KIP.

Public Interfaces

This KIP has the following public interface changes:

1. Add a new timestamp field to the message format.
2. Use the fourth least significant bit to indicate the timestamp type. (0 for CreateTime, 1 for LogAppendTime)
3. Add the following two configurations to the broker

    a. **message.timestamp.type** - This topic level configuration defines the type of timestamp in the messages of a topic. The valid values are *CreateTime* or *LogAppendTime*.
    b. **max.message.time.difference.ms** - This configuration only works when message.timestamp.type=CreateTime. The broker will only accept messages whose timestamp differs no more than max.message.time.difference.ms from the broker local time.
4. Add a timestamp field to ProducerRecord and ConsumerRecord. A producer will be able to set a timestamp for a ProducerRecord. A consumer will see the message timestamp when it sees the messages.
5. Add ProduceRequest/ProduceResponse V2 which uses the new message format.
6. Add a timestamp in ProduceResponse V2 for each partition. The timestamp will either be LogAppendTime if the topic is configured to use it or it will be NoTimestamp if create time is used.
7. Add FetchRequest/FetchResponse V2 which uses the new message format.
8. Add a timestamp variable to RecordMetadata. The timestamp is the timestamp of messages appended to partition log.

For more detail information of the above changes, please refer to the Proposed Changes section.

Proposed Changes

There are three options proposed before this proposal. The details of option 1, option 2 and Option 3 are in the Rejected Alternatives section.

The key decision we made in this KIP is **whether use LogAppendTime(Broker Time) or CreateTime(Application Time)**

The good things about LogAppendTime are:

1. Broker is more robust.
2. Monotonically increasing.
3. Deterministic behavior for log rolling and retention.
4. If CreateTime is required, it can always be put into the message payload.

The good things about CreateTime are:

1. More intuitive to users.
2. User may want to have log retention based on when the message is created instead of when the message enters the pipeline.
3. Immutable after entering the pipeline.

Because both LogAppendTime and CreateTime have their own use cases, the proposed change provides users with the flexibility to choose which one they want to use.

For more detail discussion please refer to the mail thread as well as the Rejected Alternatives section.

This KIP is closely related to KIP-33. Some of the contents listed in this KIP are actually part of KIP-33. They are put here because they are important for the design decision. More specifically, KIP-33 will implement the following changes:

1. Build a time index for each log segment using the timestamps in messages.
2. Enforce log retention and log rolling use time based index.

Add a Timestamp field to the message format with maximum allowed time difference configuration on broker.

The proposed change will implement the following behaviors.

1. Allow user to stamp the message when produce
2. When a leader broker receives a message
    a. If **message.timestamp.type=LogAppendTime**, the server will override the timestamp with its current local time and append the message to the log.
        i. If the message is a compressed message. the timestamp in the wrapper message will be updated to current server time. Broker will set the timestamp type bit in wrapper messages to 1. Broker will ignore the inner message timestamp. We do this instead of writing current server time to each message is to avoid recompression penalty when people are using LogAppendTime.
        ii. If the message is a non-compressed message, the timestamp in the message will be overwritten to current server time.
    b. If **message.timestamp.type=CreateTime**
        i. If the time difference is within a configurable threshold **max.message.time.difference.ms**, the server will accept it and append it to the log. For compressed message, server will update the timestamp in compressed message to the largest timestamp of the inner messages.
        ii. If the time difference is beyond the configured threshold **max.message.time.difference.ms**, the server will reject the entire batch with TimestampExceededThresholdException.
3. When a follower broker receives a message
    a. If the message is a compressed message, the timestamp in the compressed message will be used to build the time index. i.e. the timestamp in a compressed messages is always the largest timestamp of all its inner messages.
    b. If the message is a non-compressed message, the timestamp of this message will be used to build time index.
4. When a consumer receives a message
    a. If a message is a compressed message
        i. If the wrapper message timestamp attribute bit is 0 (CreateTime), the inner messages' timestamp will be used.
        ii. If the wrapper message timestamp attribute bit is 1, the timestamp of the wrapper message will be used as the timestamp of inner messages.
    b. If a message is a non-compressed message, the timestamp of the message will be used.
5. **message.timestamp.type** and **max.message.time.difference.ms** will be a per topic configuration.
6. In ProduceResponse V2, a timestamp will be returned for each partition.
    a. If the topic uses LogAppendTime, the timestamp returned would be the LogAppendTime for the message set.

    b. If the topic uses Create Time, the timestamp returned would be NoTimestamp.
    c. When producer invokes the callback for each message, it uses the timestamp returned in the produce response if it is not NoTimestamp. Otherwise, it uses the message timestamp which is tracked by the producer.
    d. The producer will be able to tell whether the timestamp is LogAppendTime or CreateTime in this case.
7. The time index will be built so it has the following guarantees. (Please notice that the time index will be implemented in KIP-33 instead of this KIP. The behavior discussion is put here because it is closely related to the design of this KIP)
    a. If user search by a timestamp:
        i. all the messages after the searched timestamp will be consumed.
        ii. user might see earlier messages.
    b. The log retention will take a look at the last time index entry in the time index file. Because the last entry will be the latest timestamp in the entire log segment. If that entry expires, the log segment will be deleted.
    c. The log rolling will depend on the largest timestamp of all messages ever seen. If no message is appended in log.roll.ms after largest appended timestamp, a new log segment will be rolled out.
8. The downside of this proposal are:
    a. The timestamp might not be monotonically increasing if message.timestamp.type=CreateTime.
    b. The log retention might become non-deterministic. i.e. When a message will be deleted now depends on the timestamp of the other messages in the same log segment. And those timestamps are provided by user within a range depending on what the time difference threshold configuration is.
9. Although the proposal has some downsides, it gives user the flexibility to use the timestamp.
    a. If **message.timestamp.type=CreateTime**
        i. When time difference threshold is set to Long.MaxValue. The timestamp in the message is equivalent to CreateTime.
        ii. When time difference threshold is between 0 and Long.MaxValue, it ensures the messages will always have a timestamp within a certain range.
    b. If **message.timestamp.type=LogAppendTime**, the timestamps will be log append time.

The following changes will be made to implement the above proposal.

Wire protocol change - add a Time field to the message format

```
MessageAndOffset => Offset MessageSize Message
  Offset => int64
  MessageSize => int32

  Message => Crc MagicByte Attributes Timestamp KeyLength Key ValueLength Value
    Crc => int32
    MagicByte => int8  <---------------------- Bump up magic byte to 1
    Attributes => int8 <---------------------- The 4th LSB will be used to indicate timestamp type.
    Timestamp => int64 <---------------------- NEW
    KeyLength => int32
    Key => bytes
    ValueLength => int32
    Value => bytes
```

Note: Because this KIP will be implemented in the same patch of KIP-31. The magic byte 1 also means the relative offset is used.

Add a time field to both ProducerRecord and ConsumerRecord

- If user specify the timestamp for a ProducerRecord, the ProducerRecord will be sent with this timestamp.
- If user does not specify the timestamp for a ProducerRecord, the producer stamp the ProducerRecord with current time.
- ConsumerRecord will have the timestamp of the message that were stored on broker.

Broker configuration change

add the following two configurations to broker:

- **message.timestamp.type**
- **max.message.time.difference.ms**

The following default value will be used:

message.timestamp.type=CreateTime

max.message.time.difference.ms=Long.MaxValue

I.e. by default the server will not override any user timestamp.

Add a timestamp field to RecordMetadata

- The timestamp in record metadata will be LogAppendTime if it is returned from broker, or it will be the timestamp set by user in ProducerRecord.
- When producer invokes the callback for a message, the timestamp will be available through RecordMetadata.

Add ProduceRequest/ProduceResponse V2 and FetchRequest/FetchResponse V2

The fields of ProduceRequest and FetchResponse V2 will be the same as V1. The difference is the format of the messages.

The fields of ProduceResponse V2 will be changed to following:

ProduceResponse V2

```
ProduceResponse => [ResponseStatus] ThrottleTime
  ResponseStatus => TopicName [Partition ErrorCode Offset timestamp]
    TopicName => string
    Partition => int32
    ErrorCode => int16
    Offset => int64
    timestamp => int64 <------------------- NEW

  ThrottleTime => int 64
```

The fields of FetchRequest V2 will be the same as V1.

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

Add a timestamp variable to RecordMetadata

We add a timestamp variable to RecordMetadata. The timestamp will be the timestamp of the message appended to the server.

Compatibility, Deprecation, and Migration Plan

NOTE: This part is drafted based on the assumption that KIP-31 and KIP-32 will be implemented in one patch.

The proposed protocol is not backward compatible. The migration plan are as below:

Phase 1 (MessageAndOffset V0 on disk):

1. Set message.format.version=0.9.0 on brokers. (Broker will write MessageAndOffset V0 to disk)
2. Create internal ApiVersion 0.9.0-1 which uses ProducerRequest V2 and FetchRequest V2.
3. Configure the broker to use ApiVersion 0.9.0 (ProduceRequest V1 and FetchRequest V1).
4. Do a rolling upgrade of the brokers to let the broker pick up the new code supporting ApiVersion 0.9.0-1.
5. Bump up ApiVersion of broker to 0.9.0-1 to let the broker use FetchRequest V2 for replication.
6. Upgraded brokers support both ProducerRequest V2 and FetchRequest V2 which uses magic byte 1 for MessageAndOffset.
    a. When broker sees a producer request V1 (MessageAndOffset = V0), it will decompress the message, assign offsets using absolute offsets and re-compress the message.
    b. When broker sees a producer request V2 (MessageAndOffset = V1), it will decompress the message, assign offsets using absolute offsets, ignore the time field and do re-compression. i.e. down-convert the message format to MessageAndOffset V0.
    c. When broker sees a fetch request V1 (Supporting MessageAndOffset = V0), because the data format on disk is MessageAndOffset V0, it will use the zero-copy transfer to reply with fetch response V1 with MessageAndOffset V0.
    d. When broker sees a fetch request V2 (Supporting MessageAndOffset = V0, V1), because the data format on disk is MessageAndOffset V0, it will use zero-copy transfer to reply with fetch response V2 with MessageAndOffset V0.
7. Upgrade consumer to send FetchRequest V2.
8. Upgrade producer to send ProducerRequest V2.

Phase 2 (MessageAndOffset V1 on disk):

1. After most of the consumers are upgraded, Bump up message.format.version=0.10.0 and rolling bounce the brokers.
2. Upgraded brokers do the followings:
    a. When broker sees a producer request V1 (MessageAndOffset = V0), it will decompress the message, assign offsets using relative offsets, fill in the time field with -1 if **message.timestamp.type**=CreateTime or current server time if **message.timestamp.type**=LogAppendTime and re-compress the message. i.e. up-convert the message format to MessageAndOffset V1.
    b. When broker sees a producer request V2 (MessageAndOffset = V1), it will decompress the message, assign offsets using relative offsets, check and maybe overwrite the time field, and NOT do re-compression.
    c. When broker sees a fetch request V1 (Supporting MessageAndOffset = V0), because the data format on disk is MessageAndOffset V1, it will NOT use the zero-copy transfer. Instead the broker will read the message from disk, down-convert them to V0 and reply using fetch response V1 with MessageAndOffset V0.
    d. When broker sees a fetch request V2 (Supporting MessageAndOffset = V0, V1), because the data format on disk is MessageAndOffset V1, it will use zero-copy transfer to reply with fetch response V2 with MessageAndOffset V1.

For producer, there will be no impact.

In phase 1, there will be no impact for consumers.

In phase 2, there will be some performance penalty for consumers that only supports MessageAndOffset V0, because there is no zero-copy transfer.

At the beginning of phase 2, there will be some time the log segment contains both MessageAndOffset V0 and V1. The broker will always do down conversion for FetchRequest V1 and zero-copy transfer for FetchRequest V2.

** We introduce internal ApiVersion here to help the user who are running on trunk to upgrade in the future. Otherwise the interim ApiVersion between two official releases will require users to downgrade ApiVersion then upgrade.

To canary a broker

After phase 1, it is possible for user to canary a broker in phase 2 and roll back if something goes wrong. The procedure is:

1. Set message.format.version=0.10.0 on one of the brokers (broker B).
2. Broker B will start to act like what described in phase 2.
    a. It will send FetchRequest V2 to other brokers for replication.
    b. When it sees FetchRequest V2 from other brokers and clietns, it will use zero copy
    c. When it sees FetchRequest V1 from other brokers and clients, it will NOT use zero copy but do down-conversion
3. If something goes wrong, we can do the following to rollback:
    a. shutdown broker B
    b. nuke the data of the topics it was serving as leader before shutdown
    c. set message.format.version=0.9.0
    d. restart the broker to let the broker replicate from leaders. At this point the data on disk will be in MessageAndOffset V0.

In step 2, it is recommended to put only small amount of leaders on the broker, because at that point the broker needs to do down conversion for all the fetch requests.

Rejected Alternatives

After extended discussion over option 1, option 2 and option 3. It turns out to be very difficult to meet all the following requirements at the same time:

1. Have only one timestamp concept in Kafka.
2. Enforce the time based log retention / log rolling for different use cases. (Some use case needs to based on LogAppendTime while others prefer CreateTime)
3. Protect the broker from misbehaving users (e.g. appending wrong create time to messages)

The final proposal adds a configuration to the broker to allow users to decide which timestamp they want to use according to their use case.

Option 1 - Add both CreateTime and LogAppendTime to message format

Wire protocol change

```
MessageAndOffset => MessageSize Offset Message
  MessageSize => int32
  Offset => int64

  Message => Crc MagicByte Attributes Timestamp KeyLength Key ValueLength Value
    Crc => int32
    MagicByte => int8
    Attributes => int8
    CreateTime => int64    <---------------------- NEW
    LogAppendTime => int64 <---------------------- NEW
    KeyLength => int32
    Key => bytes
    ValueLength => int32
    Value => bytes
```

Add CreateTime and LogAppendTime field to message

- **CreateTime**
    - CreateTime will be set by the producer and will not be changed afterward.
    - CreateTime accuracy is millisecond.
    - For compressed message, the CreateTime of the wrapper message will be the CreateTime of the first compressed message.
- **LogAppendTime**
    - The LogAppendTime will be assigned by broker upon receiving the message. If the message is coming from mirror maker, the original CreateTime will be maintained but the LogAppendTime will be changed by the target broker.
    - The LogAppendTime will be used to build the log index.
    - The LogAppendTime accuracy is millisecond
    - The LogAppendTime of the outer message of compressed messages will be the latest LogAppendTime of all its inner messages.
        - If the compressed message is not compacted, the relative offsets of inner messages will be contiguous and share the same LogAppendTime.

        - If the compressed message is compacted, the relative offsets of inner messages may not be contiguous. Its LogAppendTime will be the LogAppendTime of the last inner message.
    - The followers will not reassign LogAppendTime but simply update an in memory lastAppendedLogAppendTime and append the message to the log.
    - To handle leader migration where new leader has slower clock than old leader, all the leader should append max (lastAppendedTimestamp, currentTimeMillis) as the timestamp

A corner case for LogAppendTime(LAT) - Leader migration:

Suppose we have broker0 and broker1. Broker0 is the current leader of a partition and broker1 is a follower. Consider the following scenario:

1. message m0 is produced to broker 0 at time T0 (LAT = T0, T0 is the clock on broker 0).
2. broker1 as a follower replicated m0 and appended to its own log without changing the LogAppendTime of m0.
3. broker0 went down and broker 1 become the new leader.
4. message m1 is produced to broker 1 at time T1 (LAT = T1, T1 is the clock on broker 1).

In step 4, it is possible that T1 < T0 because of the time difference on two brokers. If we naively take T1 as the timestamp of m1, then the timestamp will be out of order. i.e. a message with earlier timestamp will show later in the log. To avoid this problem, at step 4, broker 1 can take max(T1, T0) as the timestamp for m1. So the timestamp in the log will not go backward. Broker 1 will be using T0 as timestamp until its own clock advances after T0.

To be more general, when a message is appended, broker (whether leader or follower) should remember the timestamp of the last appended message, if a later appended message has a timestamp earlier than the timestamp of last appended message, the timestamp of last appended message will be used.

Change time based log rolling and retention to use LogAppendTime

Time based log rolling and retention currently use the file creation time and last modified time. This does not work for newly created replicas because the time attributes of files is different from the true message append time. The following changes will address this issue.

- The time based log rolling will be based on the timestamp of the first message in the log segment file.
- The time based log retention will be based on the timestamp of the last message in the log segment file.

ConsumerRecord / ProducerRecord format change

- Add a CreateTime field to ProducerRecord. This field can be used by application to set the send time. It will also allow mirror maker to maintain the send time easily.
- Add both CreateTime and LogAppendTime to ConsumerRecord.
    - The CreateTime is useful in use cases such as stream processing
    - The LogAppendTime is useful in use cases such as log rolling and log retention.

This proposed option was rejected because:

1. It introduces 16 bytes overhead to the message.
2. It exposes LogAppendTime to users.

Option 2 - Adding only LogAppendTime to the message

This proposal is pretty much the same as the selected proposal, except it does not include CreateTime in the message format.

```
MessageAndOffset => MessageSize Offset Message
  MessageSize => int32
  Offset => int64

  Message => Crc MagicByte Attributes Timestamp KeyLength Key ValueLength Value
    Crc => int32
    MagicByte => int8
    Attributes => int8
    LogAppendTime => int64 <---------------------- NEW
    KeyLength => int32
    Key => bytes
    ValueLength => int32
    Value => bytes
```

The downside of this proposal are:

1. If the CreateTime is not in the message itself. Application needs to include the timestamp in payload. Instead of asking each application to do this, it is better to include the timestamp in the message.
2. The broker is not able to report the latency metric. While we could let the application to get the End2End latency, we might lose the latency for each hop in the pipeline.

Option 3 - Add CreateTime to the message and use LogAppendTime on brokers

Add CreateTime field to message

- **CreateTime**
    - CreateTime will be set by the producer and will not be changed afterward.
    - CreateTime accuracy is millisecond.
    - For compressed message, the CreateTime of the wrapper message will be the CreateTime of the last compressed message (this is to be consistent with base offset in KIP-31)

Use LogAppendTime for time based usages on the broker

1. Build time index.
2. Honor log rolling time
3. Honor log retention time

Compared with Option 1, while the time based log index has to be based on LogAppendTime, there is some concern about exposing the LogAppendTime (which is a broker internal concept) to user. And there will also be some per message overhead for two timestamps.

So this approach includes only CreateTime in the message format.

Wire protocol change

Change the MessageAndOffset format to:

```
MessageAndOffset => MessageSize Offset Message
  MessageSize => int32
  Offset => int64

  Message => Crc MagicByte Attributes Timestamp KeyLength Key ValueLength Value
    Crc => int32
    MagicByte => int8
    Attributes => int8
    CreateTime => int64 <---------------------- NEW
    KeyLength => int32
    Key => bytes
    ValueLength => int32
    Value => bytes
```

Build time based log index using LogAppendTime

The broker will still build time based index using LogAppendTime, **LogAppendTime will be only in the time index file, but not in message format**. i.e. not exposed to user.

When the broker will append a time index file entry for a message when:

1. The message is the first message of a log segment.
2. The message is the last message of a log segment.
3. The message is the first message received in the minute.

Let replicas to also fetch log index file

Because LogAppendTime is not included in the message format. With current replication design, followers will not be able to get the LogAppendTime from leader. In order to make log retention and log rolling policy work, the LogAppendTime needs to be propagated from leader to followers.

In this option, the LogAppendTime only exist in the time index file, therefore when followers fetch data from the leader, they have to replicate the time index file as well.

There are a few requirements here:

1. Unlike log index file, the time index file should not be rebuilt from local log when it crashes, but should always be fetched from the current leader, just the same as actual data. Otherwise we may have different time index on different replicas.
2. To ensure the log segments are identical on both leader and followers, we should always have a time index entry for the first message in a log segment.
3. In order to make the time based log retention work, we need the timestamp entry for the last message in a log segment.
4. When we truncate the messages in log segment files, we need to truncate entries in the time index files as well.

To replicate the log index entry as well, we can add the log index entry to FetchResponse, so the fetch response will become

FetchResponse format for replication

```
FetchResponse => [TopicName [Partition ErrorCode HighwaterMarkOffset MessageSetSize MessageSet
[TimeIndexEntry]]]
  TopicName => string
  Partition => int32
  ErrorCode => int16
  HighwaterMarkOffset => int64
  MessageSetSize => int32
  TimeIndexEntry => LogAppendTime Offset <------------- new, the time index entry in the message set, one
partition might contain multiple entries. The array will always be empty if the fetch request is not from
followers.
    LogAppendTime => int64
    Offset => int32
```

ConsumerRecord / ProducerRecord format change

- Add a CreateTime field to ProducerRecord. This field can be used by application to set the send time. It will also allow mirror maker to maintain the send time easily.
- Add both CreateTime to ConsumerRecord.
    - The CreateTime is useful in use cases such as stream processing

Option discussions with use cases

For documentation purpose, here are some discussions we had on this KIP.

This section discusses how the three options work with a few use cases. Option 1 and Option 2 are in the rejected option section.

Options comparison

Use cases	Option 1 (Message contains CreateTime + LogAppendTime)	option 2 (Message contains LogAppendTime only)	option 3 (message contains CreateTime only, brokers keep LogAppendTime in log index)	option 4 (Message contains a timestamp that could be overridden by broker depending on configured time difference threshold)	Comparison
Mirror Maker	Broker overrides the LAT and keep the CT as is.	Broker overrides the LAT	Broker keep the CT as. And add index entry with LAT to the log index file.	Mirror maker will keep the consumed messages' timestamp. Those timestamp may or may not be overwritten by target cluster depending on the configuration.	Option 1 provides the most information to user. The only concern is whether we should expose LAT to user. Option 2 loses the CreateTime information. Option 3 have same amount information as option 1 from broker point of view. From user point of view, it does not expose the LAT. Option 4 could be equivalent to either having CreateTime only or having LogAppendTime only depending on configuration.
Log Retention	Broker will use the LAT of the last message in a segment to enforce the policy.	Same as option 1.	Broker will use the LAT of the last entry in the log index file to enforce the retention policy. Because the leader is the source of truth for LAT, followers need to get the LAT from leader when they replicate the messages. That means we need to introduce a new wire protocol to fetch the time based log index file as well. When log recovery happens, the rebuilt time index would have different LAT from the actual arrival time of the messages in the log. And the LAT in the index file will be very close, or even the same.	The broker will take a look at the last Time Index entry of the segment to decide whether to delete the log segment or not.	Option 1 and option 2 can work with existing replication design and solve the log retention issue we have now. Option 3 solves the issue but is a bit involved because it needs to replicate the time index entry as well. Option 4 could be equivalent to either having CreateTime only or having LogAppendTime only depending on configuration.
Log rolling	Broker will use the LAT of the first message in a log segment to enforce the policy.	Same as option 1.	Broker will use the LAT of the first entry in the log index file to enforce the retention policy. Similar to the log retention case, the followers needs to replicate the time index as well. The log recovery happens, the log rolling might not be honored either.	The broker will keep an in memory earliest message timestamp in the active log segment. On broker startup, the broker will need to scan the messages in the active segment to find the earliest timestamp.	Option 1 and option 2 solves the log rolling issue. Option 3 solves the issue but is a bit involved because it needs to replicate the time index entry as well. On broker start up, Option 4 needs to scan the active log segment to find the earliest timestamp in the log segment.
Stream processi	Applications don't need to include the	Applications have to put CreateTime into	Applications don't need to include the CreateTime in the payload but simply use the CreateTime field.	Option 4 could be equivalent to either	The benefit of having a CreateTime with each message rather than put it into payload

ng	CreateTime in the payload but simply use the CreateTime field.	the payload.		having CreateTime only or having LogAppendTime only depending on configuration.	is that application protocol can be simplified. It is convenient for the infrastructure to provide the timestamp so there is no need for each application to worry about the timestamp.
Latency measurement	User can get End2End latency and lag in time.	User can get the lag in time.	User can get End2End latency.	Depending on the max.message.time.difference.ms configuration. User may or may not be able to find out the latency.	Option 1 has most information for user.
Search message by timestamp.	Detail discussion in KIP-33	Detail discussion in KIP-33	Detail discussion in KIP-33	Detail discussion in KIP-33	Detail discussion in KIP-33

Mirror maker case in detail

The behavior of broker for all the options are the same: **The broker will always override the LogAppendTime(if exists) when message arrives the broker and keep the CreateTime(if exists) untouched.**

The broker does not distinguish mirror maker from other producers. The following example explains what will the timestamp look like when there is mirror maker in the picture.(CT - CreateTime, LAT - LogAppendTime)

1. Application producer produces *message* at **T0.** ([CT = T0, LAT = -1])
2. The broker in cluster1 received *message* at **T1 = T0 + latency1** and appended the message to the log. Latency1 includes linger.ms and some other latency. ([CT = T0, LAT = T1])
3. Mirror maker copies the *message* to broker in cluster2. ([CT = T0, LAT = T1])
4. The broker in cluster2 receives message at **T2 = T1 + latency2** and appended the message to the log. ([CT = T0, LAT = T2])

The CreateTime of a message in source cluster and target cluster will be same. i.e. the timestamp is passed across clusters.

The LogAppendTime of a message in source cluster and target cluster will be different.

Discussion: should we use CreateTime OR LogAppendTime for log retention and time based log rolling?

To discuss the usage of CreateTime and LogAppendTime, it is useful to summarize the latency pattern of the messages flowing through the pipeline. The latency can be summarized to the following pattern:

1. **the messages flow through the pipeline with small latency.**
2. **the messages flow through the pipeline with similar large latency.**
3. **the messages flow through the pipeline with large latency difference.**

Also it would be useful to think about the impact of a completely wrong timestamp set by client. i.e. the robustness of the system.

Log Retention

There are both pros and cons for log retention to be based on CreateTime or LogAppendTime.

- **Pattern 1:**
  Because the latency is small, so CreateTime and LogAppendTime will be close and their won't be much difference.
- **pattern 2:**
  If the log retention is based on the message creation time, it will not be affected by the latency in the data pipeline because the send time will not change.
  If the log retention is based on the LogAppendTime, it will be affected by the latency in the pipeline. Because of the latency difference, some data can be deleted on one cluster, but not on another cluster in the pipeline.
- **Pattern 3:**
  When the messages with significantly different timestamp goes into a cluster at around same time, the retention policy is hard to follow if we use CreateTime. For example, imagine two mirror makers copy data from two source clusters to the same target cluster. If MirrorMaker1 is copying Messages with CreateTime around 1:00 PM today, and MirrorMaker2 is copying messages with CreateTime around 1:00 PM yesterday. Those messages can go to the same log segment in the target cluster. It will be difficult for broker to apply retention policy to the log segment. The broker needs to maintain the knowledge about the latest CreateTime of all messages in a log segment and persist the information somewhere.
- **Robustness:**
  If there is a message with CreateTime set to the future, the log might be kept for very long. Broker needs to sanity check the timestamp when receive the message. It could by tricky to determine which timestamp is not valid

Comparison:

	pattern 1	pattern 2	pattern 3	Robustness
Preference	CT or LAT	CT	LAT	LAT

In reality, we usually don't see all the pipeline has same large latency, so it looks **LogAppendTime is preferable than CreateTime for log retention.**

Time based log rolling

The main purpose of time based log rolling is to avoid the situation where a low volume topic always has only one segment which is also the active segment. From its nature, server side log rolling makes more sense.

- **Pattern 1:**
  Because the latency is small, so CreateTime and LogAppendTime will be close and their won't be much difference.
- **Pattern 2**:
  When the latency is large, it is possible that when a new message is produced to a broker, the CreateTime has already reached the rolling criteria. This might cause a segment with only one message.
- **Pattern 3:**
  Similar to pattern 2, a lagged message might result in a single message log segment.
- **Robustness**:
  Similar to as pattern 2 and pattern 3. Also a CreateTime in the future might break the log rolling as well.

	pattern 1	**pattern 2**	**pattern 3**	**Robustness**
Preference	CT or LAT	LAT	LAT	LAT