PIP-70： Introduce lightweight Broker Entry metadata

- Status: Proposal
- Author: Dawei Zhang/Yunze Xu/Penghui Li/Jia Zhai
- Pull Request:
- Mailing List discussion:
- Release:

1. Motivation

For messages in Pulsar, If we want to add a new property, we always change the `MessageMetadata` in the protocol(PulsarApi.proto), this kind of property could be understood by both the broker side and client-side by deserializing the `MessageMetadata`. But in some different cases,, the property needs to be added from the broker side, Or need to be understood by the broker side in a low-cost way. When the broker side gets the message produced from the client, we could add the property at a new area, which does not combine with `MessageMetadata`, so there is no need to re-calculate the checksum , and no need deserializing the original `MessageMetadata` when gets it out; and when the broker sends the message to the client, we could choose to filter out this part of the property(or not as the client needs). We call this kind of property “Broker Entry Metadata” since it is generated from the Broker side for each entry. In this way, the “Broker Entry Metadata” consumption is independent, and not related to the original `MessageMetadata`.

The benefit for this kind of “Broker Entry Metadata” is that the broker does not need to serialize/deserialize for the protobuf-ed `MessageMetadata` and not need to re-calculate the checksum before send to bookie, this will provide better performance. And also could provide a lot of features that are not supported yet.

Here are some of the use case for Broker entry metadata:
1) Provide ordered messages by time(broker side) sequence to make messages seek by the time more accurate.
Currently, each message has a `publish_time`, it uses client-side time, but for different producers in different clients, the time may not align between clients and cause the message order and the message time (`publish_time`) order may be different. But each topic-partition only has one owner broker, if we append broker side time in the “Broker Entry Metadata”, we could make sure the message order is aligned with broker side time. With this feature, we could handle the message seek by the time more accurately.

2) Provide continuous message sequence-Id for messages in one topic-partition.
MessageId is a combination of ledgerId+entryId+batchIndex; for a partition that contains more than one ledger, the Ids inside is not continuous. By this solution, we could append a sequence-Id at the end of each Message. This will make the message sequence management earlier.

In this proposal, we will take a count in the first feature “provide ordered message by time(broker side) sequence” mentioned above, this will be easier to go through the proposal.

2. Message and “Broker Entry Metadata” structure changes.

As mentioned above, there are 2 main benefits in this proposal:

1. Most of all the change happened on the Broker side.
2. Avoid to serialize/deserialize for the protobuf-ed `MessageMetadata`.
3. Avoid to re-calculate the checksum for `headerAndPayload`.

2.1 Broker Entry metadata structure in Protobuf

Protobuf used a lot in Pulsar, we could use Protobuf to do the Broker metadata serialize/deserialize.
In this example, we will save the broker side timestamp when each message is sent from the broker to BookKeeper. So the definition is very simple.

```protobuf
message BrokerEntryMetadata {
   optional uint64 broker_timestamp = 1;
  }
```

2.2 Message and “Broker Entry Metadata” structure details

Each message is sent from producer client to broker in this frame format:

```
[TOTAL_SIZE] [CMD_SIZE][CMD] [MAGIC_NUMBER] [CHECKSUM] [METADATA_SIZE] [METADATA] [PAYLOAD]
```

The first 3 fields “[TOTAL_SIZE] [CMD_SIZE] [CMD]” will be read in `LengthFieldBasedFrameDecoder` and `PulsarDecoder`, and left the rest part handled in method `org.apache.pulsar.broker.service.Producer.publishMessage`. The left part “[MAGIC_NUMBER] [CHECKSUM] [METADATA_SIZE] [METADATA] [PAYLOAD]” is usually treated as “headersAndPayload” in the code. As described above, we do not want this part to be changed at all, so we could take this part as a whole package.

```
[MAGIC_NUMBER] [CHECKSUM] [METADATA_SIZE] [METADATA] [PAYLOAD] ==> [HEADERS_AND_PAYLOAD]
```

We need to add some fields to make ”Broker Entry Metadata” work well.

```
[BROKER_ENTRY_METADATA_MAGIC_NUMBER] [BROKER_ENTRY_METADATA_SIZE] [BROKER_ENTRY_METADATA]
[HEADERS_AND_PAYLOAD]
```

BROKER_ENTRY_METADATA_MAGIC_NUMBER is used to identify whether this feature is on, and the message is handled by this feature.

BROKER_ENTRY_METADATA_SIZE is added for this feature and records the size of the serialized broker metadata content.

BROKER_ENTRY_METADATA is the serialized broker metadata content.

“HEADERS_AND_PAYLOAD” is the original ByteBuf data that contains metadata and payload.

We put the newly added fields before instead of after "HEADERS_AND_PAYLOAD" here, this is because

- Firstly, the “Broker Entry Metadata” is an addition for origin protocol and it can support this new feature without much modification for origin wire protocol, especially it does not need to serialize/deserialize for the protobuf-ed `MessageMetadata` build from the producer.
- Secondly, if we put new fields after "HEADERS_AND_PAYLOAD", we can't know where the offset for newly added fields since we don't know the length of "PAYLOAD", and also, fields after "PAYLOAD" will change the CRC32 checksum which means we need to re-calculate the checksum one more time.

2.3 Message and “Broker Entry Metadata” lifecycle

The message process logic would be like this:

1. Producer client send the original message to the broker, and parsed by “PulsarDecoder”, only “[HEADERS_AND_PAYLOAD]” left;
2. originally, “[HEADERS_AND_PAYLOAD]” will be written into BookKeeper by method `asyncAddEntry` in `ManagedLedger`; but now we first add above “[BROKER_ENTRY_METADATA]” and related parts along with “[HEADERS_AND_PAYLOAD]”, then write the whole ByteBuf into BookKeeper by `asyncAddEntry` ; so the "Broker metadata" is kept together with the original message in BookKeeper.
3. When some features need to use this “Broker Entry Metadata”, read the new BookKeeper “Entry”, and get the Broker Entry Metadata information to serve the new feature.
4. In this “provide ordered message by time(broker side) sequence” feature, when a seek-by-time operation passes into the broker, we readout “broker_timestamp” from “[BROKER_ENTRY_METADATA]”.
5. After brokers read BookKeeper Entry out, and before send to the consumer client, we need to filter out the Broker metadata related part and only return the “[HEADERS_AND_PAYLOAD]” part.

By this way, we don't need to serialize/deserialize for the protobuf-ed `MessageMetadata`.

For more details, in these steps:

a)) produced message handle and store

Producer client-side message send to the broker will be handled by the method

```
publishMessageToTopic(ByteBuf headersAndPayload, long sequenceId, long batchSize, boolean isChunked)
```

in the class `org.apache.pulsar.broker.service.Producer`. And it will finally call into the method `asyncAddEntry` of `ManagedLedger`, it will write the passed in serialized message as an Entry stored in BookKeeper.

```
void asyncAddEntry(ByteBuf headersAndPayload, AddEntryCallback callback, Object ctx)
```

For Broker Message metadata content, we need to put the current broker timestamp in `message BrokerMetadata` and use protobuf to serialize it into `ByteBuf`. Then we put the size and content of serialized `ByteBuf` along with the original `ByteBuf headersAndPayload`.

Then “[HEADERS_AND_PAYLOAD]” turned into “[BROKER_ENTRY_METADATA_MAGIC_NUMBER] [BROKER_ENTRY_METADATA_SIZE] [BROKER_ENTRY_METADATA] [HEADERS_AND_PAYLOAD]” and stored in BookKeeper.

b)) seek by time happens

Originally, when a seek_by_time comes in, it will finally call into class `org.apache.pulsar.broker.service.persistent.PersistentMessageFinder.findMessages` and the original logic is like this

```java
public void findMessages(final long timestamp, AsyncCallbacks.FindEntryCallback callback) {
    …
     cursor.asyncFindNewestMatching(ManagedCursor.FindPositionConstraint.SearchAllAvailableEntries, entry -> {
        MessageImpl msg = null;
        msg = MessageImpl.deserialize(entry.getDataBuffer());  < === deserialize
        return msg.getPublishTime() < timestamp;
     }, this, callback);
    …
```

Find message through the topic-partition, and read out the target Entry from BookKeeper, deserialize Entry into a MessageImpl and read the message publish time that matches the requirements; if not match, iterate to another round of reading another message.

By the new design, after reading out the target Entry from BookKeeper, it does not need to do the deserialize, but only need to read out the “broker_timestamp” from the “Broker Entry Metadata”.
As mentioned above, the frame stored in BookKeeper is as: “[BROKER_ENTRY_METADATA_MAGIC_NUMBER] [BROKER_ENTRY_METADATA_SIZE] [BROKER_ENTRY_METADATA] [HEADERS_AND_PAYLOAD]”,

1. Still use `entry.getDataBuffer()` to get the ByteBuf out;
2. Read and check “[BROKER_ENTRY_METADATA_MAGIC_NUMBER]” to see if the “Broker entry metadata” feature is enabled; if not enabled, turn into old behavior;
3. If enabled, read “ [BROKER_ENTRY_METADATA_SIZE] “, then read “[BROKER_ENTRY_METADATA] ”.
4. Use protobuf to deserialize “[BROKER_ENTRY_METADATA]”, and read “broker_timestamp” out.

c)) read an entry from BookKeeper and send it to the consumer client

The main logic happened in method `sendMessages` in class `org.apache.pulsar.broker.service.Consumer`, the change is similar to above “seek by time”, after reading BookKeeper Entry, we need to filter out "BROKER_ENTRY_METADATA" related part, and only return “[HEADERS_AND_PAYLOAD]” to consumer client.

3 Summary of changes that need

Here is a summary of the above changes that related:

1. Add Broker metadata protobuf

```protobuf
message BrokerEntryMetadata {
    optional uint64 broker_timestamp = 1;
}
```

2. change how produced message is saved in bookkeeper:
   `org.apache.bookkeeper.mledger.impl.OpAddEntry`

3. change how message is seek-by-time:
   `org.apache.pulsar.broker.service.persistent.PersistentMessageFinder.findMessages`

4. change how message send back to Consumer
   `org.apache.pulsar.broker.service.Consumer`
5. add interceptor interface for Broker Entry Metadata
```java
BrokerEntryMetadataInterceptor {

    BrokerEntryMetadata.Builder intercept(BrokerEntryMetadata.Builder brokerMetadata);

}
```
6. add an implement of BrokerEntryMetadataInterceptor for adding broker timestamp to entry
```java
public class AppendBrokerTimestampMetadataInterceptor implements BrokerEntryMetadataInterceptor{

    @Override
    public PulsarApi.BrokerEntryMetadata.Builder intercept(PulsarApi.BrokerEntryMetadata.Builder brokerMetadata) {
        return brokerMetadata.setBrokerTimestamp(System.currentTimeMillis());
    }
}
```

other changes:

1. A broker config to enable this Broker entry metadata feature. e.g. “brokerEntryMetadataInterceptors” which is a comma-separated list of BrokerEntryMetadataInterceptors.
2. besides change `org.apache.pulsar.broker.service.persistent.PersistentMessageFinder.findMessages`, search other places, which called `MessageImpl.deserialize(entry.getDataBuffer())` and `msg.getPublishTime()`, such as `PersistentTopic.isOldestMessageExpired()` to read message publish-time from new “Broker entry metadata” instead of deserialize the whole BookKeeper Entry into Message and call `Message.getPublishTime()` if this feature enabled.

4. Compatibility, Deprecation and Migration Plan

For the first feature, since most of the changes are internal changes in brokers, and it is a new feature which doesn’t change pulsar’s wire protocol(between broker and client), and not changed public API. There is no backward compatibility issue. It is a newly added feature. So there is nothing to deprecate or migrate.

In the future, we will consider the compatibility when implementing the secondary feature of “continuous message sequence-Id” since we should send messages with “sequence_id” to consumers. In this case, we will take different actions depending on the data format, the feature enables or not, and the consumer client version. Data is a new format if it contains "[BROKER_ENTRY_METADATA_MAGIC_NUMBER] [BROKER_ENTRY_METADATA_SIZE] [BROKER_ENTRY_METADATA]" and consumer is a new version if it can process message contains "[BROKER_ENTRY_METADATA]".

5. Test Plan

* Unit tests for each individual change: broker/client worked well for produce/consume.
* Integration tests or UT for the end-to-end pipeline: multi producers set the message's time in random time order; seek-by-time could seek to the right message.
* Load testing for ensuring performance: seek should be quicker.