KIP-82 - Add Record Headers

Status

Current state: *Adopted*

Discussion thread: *here*

JIRA: *KAFKA-4208*

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.

In most message systems (JMS, QPID etc), streaming systems and most transport systems(HTTP, TCP), it is typical to have a concept of headers and payload.

The payload is traditionally for the business object, and headers are traditionally used for transport routing, filtering etc. Headers are most typically key=value pairs.

In its current state Kafka does not support the ability to have headers natively in its message/record format.

Examples where having separate supported custom headers becomes useful (this is not an exhaustive list).

- Automated routing of messages based on header information between clusters
- Enterprise APM tools (e.g. Appdynamics, Dynatrace) need to stitch in 'magic' transaction ids for them to provide end to end transaction flow monitoring.
- Audit metadata to be recorded with the message, e.g. clientId that produced the record, unique message id, originating clusterId the message was first produced into for multi cluster routing.
- Business payload needs to be end to end encrypted and signed without tamper, but eco-system components need access to metadata to achieve tasks.

Kafka currently has Record<K, V> structure which originally could be used to follow this semantic where by K could contain the headers information, and the V could be the payload.

- Since message compaction feature it is no longer possible to add metadata to K, else compaction would treat each message as a different keyed message .
- It is not currently possible to use value part and use some form of a wrapper e.g. Message<H, V>, as for compaction to perform a delete a record is sent with a NULL value, as such for where a delete record is sent using a message wrapper to carry the metadata would not work, as the value technically would no longer be null.

This issue has been flagged by many people over the past period in forums.

Further details and a more detailed case for headers can be seen here : A Case for Kafka Headers

Amendments made during implementation, and on KIP-118 being pulled are highlighted orange, changes reviewed during PR and notification sent to dev mailing lists.

Public Interfaces

This KIP has the following public interface changes:

1. Add a new headers length and value (byte[]) to the core message format.
2. Create a Header Interface and implementing class
    a. Interface

```
public interface Header {

   String key();

   byte[] value();
}
```

    b. Implementation Detail

        i. Add a String key field to Header implementing class
        ii. Add a byte[] value field to Header implementing class
3. Create a Headers Interface and implementing class
    a. Headers will be mutable
        i. For the Producer, after send and post interceptors it will be turned into a read only immutable instance.
        ii. This will be done by the invoking "close()" method, this method is not exposed in the api, but an implementation detail.
    b. Interface

```
public interface Headers extends Iterable<Header> {

   /**
    *  Adds a header (key inside), returning if the operation succeeded.
    *  If headers is in read-only, will always fail the operation with throwing
IllegalStateException.
    */
   Headers add(Header header) throws IllegalStateException;

   /**
    *  Adds a header by key and value, returning if the operation succeeded.
    *  If headers is in read-only, will always fail the operation with throwing
IllegalStateException.
    */
   Headers add(String key, byte[] value) throws IllegalStateException;

   /**
    *  Removes ALL HEADERS for the given key returning if the operation succeeded.
    *  If headers is in read-only, will always fail the operation with throwing
IllegalStateException.
    */
   Headers remove(String key) throws IllegalStateException;

   /**
    *  Returns JUST ONE (the very last) header for the given key, if present.
    */
   Header lastHeader(String key)

   /**
    *  Returns ALL headers for the given key, if present.
    */
   Iterable<Header> headers(String key);

}
```

    c. Implementation Detail

        i. Add accessor methods on the Headers class

            1. Headers add(Header header)
            2. Headers add(String key, byte[] value)
            3. Headers remove(Header header)
            4. Header lastHeader(String key)
            5. Iterable<Header> headers(String key)

            a. Inline with ConsumerRecords interface returning the subset of ConsumerRecord for a given topic.
                i. public Iterable<ConsumerRecord<K, V>> records(String topic)
        6. implement Iterable<Header>
    ii. interceptors and k,v serialisers are expected to add headers during the produce intercept stage.
4. Add a headers field to ProducerRecord and ConsumerRecord.
5. Add constructor(s) of Producer/ConsumerRecord to allow passing in of Iterable<Header>
    a. use case is MirrorMakers able to copy headers.
6. Add accessor methods on the Producer/ConsumerRecord Headers headers()

```
public class ProducerRecord<K, V> {

   ...

   ProducerRecord(K key, V value, Iterable<Header> headers, ...)

   ...

   public Headers headers();

   ...

}
```

```
public class ConsumerRecord<K, V> {

   ...
   ConsumerRecord(K key, V value, Iterable<Header> headers, ...)

   ...

   public Headers headers();

   ...

}
```

7. Changes needed, will piggyback onto V3 of ProduceRequest and V5 of FetchRequest which were introduced in KIP-98
8. The serialisation of the [String, byte[]] header array will on the wire using a strict format
9. Each headers value will be custom serialisable by the interceptors/plugins/serdes that use the header.
10. Expose headers to De/Serializers - extended interface added, for lack of default methods available in java 8

```
public interface ExtendedDeserializer<T> extends Deserializer<T> {
    T deserialize(String topic, Headers headers, byte[] data);
}
```

```
public interface ExtendedSerializer<T> extends Serializer<T> {
    byte[] serialize(String topic, Headers headers, T data);
}
```

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

Proposed Changes

There are four options proposed before this proposal. This details our proposed solution of Option 1 described here. The other options are in the Rejected Alternatives section.

The advantages of this proposal are:

- Adds the ability for producers to set standard header key=value value pairs
- No incompatible client api change (only new methods)
- Allows users to specify the serialisation of the header value per header
- Provides a standardised interface to eco systems of tools that then can grow around the feature

The disadvantage of this proposal is:

- Change to the message object

Key duplication and ordering

- Duplicate headers with the same key must be supported.
- The order of headers must be retained throughout a record's end-to-end lifetime: from producer to consumer.

Create a Headers Interface and Implementation to encapsulate headers protocol.

- See above public interfaces section for sample interfaces.

Add a headers field Headers to both ProducerRecord and ConsumerRecord

- Accessor methods of Headers headers() added to interface of the ProducerRecord/ConsumerRecord.
- Add constructor(s) of Producer/ConsumerRecord to allow passing in of an existing/pre-constructed headers via Iterable<Header>
    1. use case is MirrorMakers able to copy headers.

Add new method to make headers accessible during de/serialization

- Add new default method to Serialization/Deserialzation class, with Header parameter
    - Due to KIP-118 not being implemented, a new extended interface ExtendedSerializer ExtendedDeserializer with new extra methods is introduced instead
        - Existing De/Serialization will be wrapped and work as before.

Wire protocol change - add array of headers to end of the message format

The below is for the core Message wire protocol change needed to fit headers into the message.

- A header key can occur multiple times, clients should expect to handle this, this can be represented as a multi map, or an array of values per key in clients.

This is basing off KIP-98 Message protocol proposals.

```
Message =>
        Length => varint
        Attributes => int8
        TimestampDelta => varlong
        OffsetDelta => varint
        KeyLen => varint
        Key => data
        ValueLen => varint
        Value => data
        Headers => [Header] <------------ NEW Added Array of headers

Header =>
                Key => string (utf8) <------------------------------ NEW UTF8 encoded string (uses varint
length)
        Value => bytes  <----------------------------------- NEW header value as data (uses varint length)
```

Compatibility, Deprecation, and Migration Plan

- Current client users should not be affected, this is new api methods being added to client apis
- Message version allows clients expecting older message version, would not be impacted
    - older producers simply would produce a message which doesn't support headers
        - upcasting would result in a message without headers
        - new consumer would simply get a message and as no headers on send would have none.
    - older consumers simply would consume a message they would not be aware of any headers
        - down casting would remove headers from the message as older message format doesn't have these in protocol.
- Message version migration would be handled as like in KIP-32

- Given the mutability of the headers and the fact that we close them at some point, it means that users should not resend the same record if they use headers and interceptors.
    - They should either configure the producer so that automatic retries are attempted (preferred although there are some known limitations regarding records that are timed out before a send isactually attempted)
    - or
    - They need to create a new record while being careful about what headers they include (if they rely fully on interceptors for headers, they could just not include any headers in the new headers).

Out of Scope

Some additional features/benefits were noticed and discussed on the above but are deemed out of scope and should be tackled by further KIPS.

- Core message size reduction
    - remove overhead of 4 bytes for KeyLength when no headers using attributes bit
    - reduce overhead of 4 bytes for KeyLength by using variable length encoded int
    - reduce overhead of 4 bytes for ValueLength by using variable length encoded int
- Broker side interceptors
    - with headers we could start introducing broker side message interceptors to append meta data or handle messages
- Single Record consumer API
    - There is many uses cases where single record consumer/listener api is more user friendly - this is evident by the fact spring kafka have already created a wrapper, it would be good to support this natively.

Rejected Alternatives

Map<Int, byte[]> Headers added to the Producer/ConsumerRecord

The concept is similar to the above proposed but int keys

- Benefits
    - more compact much reduced byte size overhead only 4 bytes.
    - String keys can dwarf the value in byte size.
- Disadvantages
    - String keys are more common in many systems
    - Requires management of the int key space, where as string keys have natural key space management.

Map<String, String> Headers added to the ConsumerRecord

The concept is similar to the above proposed but with a few more disadvantages.

- Benefits
    - Adds the ability for producers to set standard header key=value string value pairs
    - No incompatible client api change (only new methods)
    - Allows users to specify the serialisation of the key=value map (String(&=), JSON, AVRO).
    - Provides a standardised interface to eco systems of tools can grow around the feature
- Disadvantages
    - Change to the message object
    - String key cause a large key, this can cause a message payload thats of 60bytes to be dwarfed by its headers
    - String value again doesn't allow for compact values, and restricts that a value must be a String
    - Not able to use headers on the broker side with custom serialisation

ProducerRecord<K, H, V>, ConsumerRecord<K, H, V>

The proposed change is that headers are Map<int, byte[]> only, this alternative is that headers can be of any type denoted by H

- Benefits
    - Complete customisation of what a header is.
- Disadvatages
    - As generics don't allow for default type, this would cause breaking client interface compatibility if done on Producer/ConsumerRecord.
        - Possible work-around would be to have HeadersProducer/ConsumerRecord<K, H, V> that then Producer/ConsumerRecord extend where H is object, this though becomes ugly fast if kept for a time or would require a deprecation / refactor v2 period.

Common Value Message Wrapper - Message<V>

This builds on the status quo and addresses some core issues, but fails to address some more advanced and future use cases and also has some compatibility issues for upgrade/clients not supporting.

please see: Headers Value Message Wrapper

Status Quo - Keep Custom Value Message Wrapper - Message<H, P>

This concept is the current defacto way many users are having to temporally deal with the situation, but has some core key issues that it does not resolve.

- Benefits
  - This will cause no broker side changes to handle the message
- Disadvantages
  - This would not work with compaction where headers are needed to be sent on delete record which then would not deliver on many of the requirements.
  - Every organization has custom solution
    - No ecosystem of tooling can evolve
    - Cost of Third party vendors integration high, as custom for every organization they integrate for.
  - Not able to make use of headers server side
  - Couple Serialisation and Deserialisation of the value for both the header and payload.