KIP-482: The Kafka Protocol should Support Optional Tagged Fields

Status

Current state: Accepted

Discussion thread: here

JIRA: **KAFKA-8885 - Getting issue details...** STATUS

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

Motivation

The Kafka RPC protocol has its own serialization format for binary data. This format is also used for messages on disk, and for metadata such as consumer offsets. In this strongly typed format, each message has a predefined schema which both senders and receivers must understand in order for communication to take place.

In order to support evolving the protocol over time, messages have a 15-bit version number. Each distinct version of a message has a distinct schema. So, for example, the schema for version 3 of FetchRequest may contain a different set of fields than the schema for version 2.

While this versioning scheme allows us to change the message schemas over time, there are many scenarios that it doesn't support well. One scenario that isn't well-supported is when we have data that should be sent in some contexts, but not others. For example, when a MetadataRequest is made with IncludeClusterAuthorizedOperations set to true, we need to include the authorized operations in the response. However, even when IncludeClusterAuthorizedOperations is set to false, we still must waste bandwidth sending a set of blank authorized operations fields in the response. The problem is that the field that is semantically optional in the message, but that is can't be expressed in the type system for the Kafka RPC protocol.

Another scenario that we don't support is attaching an extra field to a message in a manner that is orthogonal to the normal versioning scheme. For example, we might want to attach a trace ID, a "forwarded-by" field, or a "user-agent" field. It wouldn't make sense to add all these fields to the message schema on the off chance that someone might use them. In order to support these scenarios, we would like to add the concept of "tagged fields" to the Kafka protocol.

Finally, sometimes, we want to add extra information to a message without requiring a version bump for everyone who has to read that message. This is particularly important for metadata like consumer offsets.

Proposed Changes

Tagged Fields

We propose to add tagged fields to the Kafka serialization format. Each tagged field will be identified by its 31-bit numeric tag.

Tagged fields are always optional. When they are not present in a message, they do not take up any space.

A new tagged field can be added to an existing protocol version without bumping the protocol version. If the receiver does not expect a particular tagged field, it will simply skip over the field without deserializing it.

More Efficient Serialization for Variable-Length Objects

Kafka RPC supports variable length strings, byte buffers, and arrays. In each of these cases, the object is serialized as a fixed-length size, followed by the data.

Since these objects are usually small, the size field almost always ends up taking up much more space than is needed. For example, most arrays do not have more than 100 elements. However, every array is currently prefixed by a four-byte length that could theoretically denote a size up to 2 billion.

Rather than using a fixed-length size, we should use a variable-length integer that can take between 1 and 5 bytes, depending on the length. In the common case when the array is small, using variable-length sizes will let us save three bytes per array, three bytes per byte buffer, and one byte per string.

Flexible Versions

It would be tedious to update the JSON message specifications to add tagged fields to each structure. Similarly, we don't want to manually annotate each string, buffer, or array that should now be serialized in a more efficient way. Instead, we should simply have the concept of "flexible versions." Any version of a message that is a "flexible version" has the changes described above.

In order to have flexible version support across all requests and responses, we will bump the version of all requests and responses. The new versions will be flexible. (This version bump may be implemented earlier for some message types than others, depending on implementation considerations.)

RequestHeader Version 1

Requests within a "flexible version" will have a new version of the request header. The new RequestHeader version will be version 1, superseding version 0. In this new version, the RequestHeader's ClientId string will be a COMPACT_STRING rather than STRING. Additionally, the header will contain space for tagged fields at the end. Supporting tagged fields in the request header will give us a natural place to put additional information that is common to all requests.

ResponseHeader Version 1

Responses within a "flexible version" will have a new version of the response header. The new ResponseHeader version will be version 1, superseding version 0. In this new version, the header will contain space for tagged fields at the end. Supporting tagged fields in the response header will give us a natural place to put additional information that is common to all responses.

Public Interfaces

JSON Schemas

flexibleVersions

The flexible versions will be described by a new top-level field in each request and response. The format will be the same as that of existing version fields. The flexible versions must be specified in each JSON file.

Note that adding support for tagged versions to an RPC requires bumping the protocol version number.

Specifying Tagged Fields

Tagged fields can appear at the top level of a message, or inside any structure.

Each optional field has a 31-bit tag number. This number must be unique within the context it appears in. For example, we could use the tag number "1" both at the top level and within a particular substructure without creating ambiguity, since the contexts are separate. Tagged fields can have any type.

Here is an example of a message spec which has tagged fields at both the top level and the array level:

```
{
  "apiKey": 9000,
  "type": "response",
  "name": "FooResponse",
  "validVersions": "0-9",
  "flexibleVersions": "9+",
  "fields": [
    {
      { "name": "UserAgent", "type": "string", "tag": 0, "taggedVersions": "9+",
        "about": "The user-agent that sent this request." },
      { "name": "Foos", "type": "[]Foo", "versions": "0+",
      "about": "Each foo.", "fields": [
        { "name": "Bar", "type": "string", "tag": 0, "taggedVersions": "9+",
          "default": "hello world", "about": "The bar associated with this foo, if any." },
        { "name": "Baz", "type": "int16", "versions": "0+",
          "about": "The baz associated with this foo." },
  ...
  ]
}
```

Type Classes

Type Class Name	Field Class Name	Description
CompactArrayOf	CompactArray	Represents an array whose length is expressed as a variable-length integer rather than a fixed 4-byte length.
COMPACT_STRING	CompactString	Represents a string whose length is expressed as a variable-length integer rather than a fixed 2-byte length.
COMPACT_NULLABLE_STRING	CompactNullableString	Represents a nullable string whose length is expressed as a variable-length integer rather than a fixed 2-byte length.
COMPACT_BYTES	CompactBytes	Represents a byte buffer whose length is expressed as a variable-length integer rather than a fixed 4-byte length.
COMPACT_NULLABLE_BYTES	CompactNullableBytes	Represents a nullable byte buffer whose length is expressed as a variable-length integer rather than a fixed 4-byte length.
TaggedFields	TaggedFieldsSection	Represents a section containing optional tagged fields.

Tagged Fields and Version Compatibility

A tagged field can be retroactively added to an existing message version without breaking compatibility-- provided, of course, that the version in question was a "flexible version." We cannot add any tagged fields to a inflexible version, and we cannot retroactively change an inflexible version to a flexible one.

Tag numbers must never be reused, nor can we alter the format of a tagged field. This includes changing a nullable field to a non-nullable one, or vice versa. When you create the tagged field, you must decide if it will be nullable or not, and stick with that decision forever.

A field can be specified as tagged in some versions and non-tagged in others. The main use-case for this is to gracefully migrate fields which were previously mandatory to tagged fields, where appropriate.

For convenience, if a field is specified as having a tag, we will assume by default that the tag can appear in all flexible versions. Therefore, it isn't usually required to specify "versions" or "taggedVersions." If "taggedVersions" does appear, then it must be a subset of "versions," which must also be specified.

Serialization

Tag Sections

In a flexible version, each structure ends with a tag section. This section stores all of the tagged fields in the structure.

The tag section begins with a number of tagged fields, serialized as a variable-length integer. If this number is 0, there are no tagged fields present. In that case, the tag section takes up only one byte.

If the number of tagged fields is greater than zero, the tagged fields follow. They are serialized in ascending order of their tag. Each tagged field begins with a tag header.

Tag Headers

The tag header contains two unsigned variable-length integers. The first one contains the field's tag. The second one is the length of the field.

The number of tagged fields	Field 1 Tag	Field 1 Length	FIeld 1 Data	Field 2 Tag	Field 2 Length	Tag Data 2	...
UNSIGNED_VARINT	UNSIGNED_VARINT	UNSIGNED_VARINT	<field 1 type>	UNSIGNED_VARINT	UNSIGNED_VARINT	<field 2 type>	...

Compact Arrays

A compact array contains a 32-bit unsigned varint, followed by the array elements.

32-bit length (plus one)	Element 0	Element 1	...
UNSIGNED_VARINT	<array element type>	<array element type>	...

If the length field is 0, the array is null. If the length field is 1, the length is 0. If the length field is 2, the length is 1, etc.

Compact Bytes

A compact bytes field contains a 32-bit unsigned varint, followed by the bytes.

32-bit length (plus one)	Payload
UNSIGNED_VARINT	Bytes

If the length field is 0, the bytes field is null. If the length field is 1, the length is 0. If the length field is 2, the length is 1, etc.

Compact String

A compact string field contains a 32-bit unsigned varint, followed by the string bytes.

32-bit length (plus one)	String
UNSIGNED_VARINT	Bytes

If the length field is 0, the string field is null. If the length field is 1, the length is 0. If the length field is 2, the length is 1, etc.

Unsigned Varints

The UNSIGNED_VARINT type describes an unsigned variable length integer.

To serialize a number as a variable-length integer, you break it up into groups of 7 bits. The lowest 7 bits is written out first, followed by the second-lowest, and so on. Each time a group of 7 bits is written out, the high bit (bit 8) is cleared if this group is the last one, and set if it is not.

So for, example, let's say we were trying to serialize 300, which is 0b100101100 in binary. This would be serialized as the following two-byte sequence:

Continuation bit	B6	B5	B4	B3	B2	B1	B0	Continuation Bit	B13	B12	B11	B10	B9	B8	B7
1	0	1	0	1	1	0	0	0	0	0	0	0	0	1	0

Unlike signed varints, unsigned varints do not use "zig-zag encoding." So they cannot efficiently represent negative numbers. However, an unsigned varint can represent positive numbers in the same or fewer bits than a signed varint.

Requests and Responses

All requests and responses will end with a tagged field buffer. If there are no tagged fields, this will only be a single zero byte.

Compatibility, Deprecation, and Migration Plan

As mentioned earlier, existing request versions will not be changed to support optional fields. However, new versions will have this support going forward.

In general, adding a tagged field is always a compatible operation. However, we do not want to reuse a tag that was previously used for something else. Changing the type or nullability of an existing optional field is also an incompatible change.

Rejected Alternatives

Tagged Field Buffer Serialization Alternatives

- We could serialize optional fields as a tag and a type, rather than a tag and a length. However, this would prevent us from adding new types in the future, since the old deserializers would not understand them.
- We could allow the serialization of arrays of objects. However, this would require a two-pass serialization rather than a single-pass serialization. The first pass would have to cache the lengths of all the optional object arrays. We might support this eventually, but for now, it doesn't seem necessary. We can add it later in a compatible fashion if we decide to.

Make all Fields Tagged

Rather than supporting both mandatory and optional fields, we could make all fields optional. For fields which we always expect to use, however, this would take more space when serialized. There are also situations where it is useful for the recipient to know which features the sender supports, and the mandatory field mechanism handles these situations well.

Use Escape Bytes to Minimize Space Usage

We could use escaping to make the size of a tag buffer zero bytes in some cases. However, this would greatly complicate encoding and decoding the protocol. It is better to make variable length fields more efficient in general, to offset the extra space of tagged field buffers.