KIP-625: Richer encodings for integral-typed protocol fields

Status

Current state: *Under Discussion*

Discussion thread: *here [Change the link from the KIP proposal email archive to your own email thread]*

JIRA: *KAFKA-9927*

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

Motivation

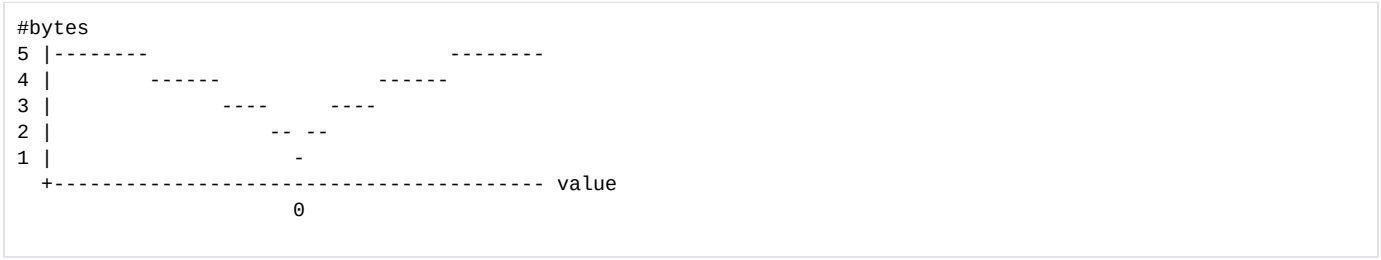
The Kafka protocol already supports variable length encodings for integers. Specifically, KIP-482 added support for using an unsigned variable length integer encoding for the length of variable length data (strings, arrays, bytes) and for integer quantities in tagged fields. However it is currently not possible to use a variable length encoding for regular fields with an integral (short, integer or long) type.

Varints encode two's complement signed ints or longs using a variable number of bytes such that "smaller numbers" require fewer bytes. The trade-off is that larger ints can require up to 5 or 9 bytes (as oposed to the 4 or 8 bytes that a fixed encoding would require).

For 32-bit integers encoded using a "signed" variable length encoding, the histogram of int value to number of encoded bytes looks like this:

- ints in [-2147483648,-134217727] require 5 bytes
- ints in [-134217728,-1048575] require 4 bytes
- ints in [-1048576,-8191] require 3 bytes
- ints in [-8192,-63] require 2 bytes
- ints in [-64,63] require 1 byte
- ints in [64,8191] require 2 bytes
- ints in [8192,1048575] require 3 bytes
- ints in [1048576,134217727] require 4 bytes
- ints in [134217728,2147483647) require 5 bytes

This can be represented as a histogram (with a non-linear x axis):



For 32-bit integers encoded using a "unsigned" variable length encoding, the histogram of int value to number of encoded bytes looks like this:

- ints in [0,127] require 1 byte
- ints in [128,16383] require 2 bytes
- ints in [16384,2097151] require 3 bytes
- ints in [2097152,268435455] require 4 bytes
- ints in [268435456,4294967296) require 5 bytes

Or, as a histogram:

```
#bytes
5 |                          ----------------
4 |             ------------
3 |     --------
2 | ----
1 |-
  +---------------------------------------- value
  0
```

Note

It is important to understand that the use of term "signed" or "unsigned" here does not refer to the signed-ness of the Java `int` or `long` which is serialized (Java `int` and `long` are always signed), but rather to how the encoding is more efficient for a range of numbers which are, roughly, symmetric about zero, or a range whose lower bound is zero.

Numerous existing PRCs use Java `shorts`, `ints` or `longs` for quantities which have a histogram with a predictable shape

- Broker ids are usually numbered sequentially from 0, 1 or perhaps 1000, and are typically less than 16383.
- "Replica ids" and "leader ids" are just broker ids.
- Partition ids are numbered sequentially from 0, and would typically be less than 16383
- Error codes are numbered sequentially from -1.

Using variable length encodings for these quantities in Kafka protocol messages would make those message smaller. For some RPCs the messages could be substantially smaller.

An example: MetadataResponse

Taking `MetadataResponseData` as an example, and looking just at the deeply nested `MetadataResponsePartition` the current schema is:

```
{ "name": "ErrorCode", "type": "int16", "versions": "0+",
  "about": "The partition error, or 0 if there was no error." },
{ "name": "PartitionIndex", "type": "int32", "versions": "0+",
  "about": "The partition index." },
{ "name": "LeaderId", "type": "int32", "versions": "0+", "entityType": "brokerId",
  "about": "The ID of the leader broker." },
{ "name": "LeaderEpoch", "type": "int32", "versions": "7+", "default": "-1", "ignorable": true,
  "about": "The leader epoch of this partition." },
{ "name": "ReplicaNodes", "type": "[]int32", "versions": "0+", "entityType": "brokerId",
  "about": "The set of all nodes that host this partition." },
{ "name": "IsrNodes", "type": "[]int32", "versions": "0+",
  "about": "The set of nodes that are in sync with the leader for this partition." },
{ "name": "OfflineReplicas", "type": "[]int32", "versions": "5+", "ignorable": true,
  "about": "The set of offline replicas of this partition." }
```

The current fixed length encoding requires

```
size = 2           // errorCode
     + 4           // partitionIndex
     + 4           // leaderId
     + 4           // leaderEpoch
     + 1           // size of replicaNodes (assuming best case)
     + 4 × replica // replicaNodes
     + 1           // size of isrNodes (assuming best case)
     + 4 × isr     // isrNodes
     + 1           // size of offlineReplicas (assuming best case)
     + 4 × offline // offlineReplicas
     = 17 + 4×(replica + isr + offline) in the best case
```

If the schema for `MetadataResponsePartition` used the unsigned variable length encoding for all fields then in the best case we get the formula:

```
size = 1           // errorCode
     + 1           // partitionIndex
     + 1           // leaderId
     + 1           // leaderEpoch
     + 1           // size of replicaNodes (assuming best case)
     + 1 × replica // replicaNodes
     + 1           // size of isrNodes (assuming best case)
     + 1 × isr     // isrNodes
     + 1           // size of offlineReplicas (assuming best case)
     + 1 × offline // offlineReplicas
     = 7 + R + I + O in the best case
```

More concretely, benchmarking a `MetadataResponse` (just the body, excluding the header) containing a single 100 partition topic replicated across two brokers suggests that:

Encoding	Size/byte	Struct Serialize/µs	Struct Deserialize/µs	Buffer Serialize/µs	Buffer Deserialize/µs
fixed	3216	56,458	14,416	7,215	11,508
variable (best case)	1170	65,226	14,713	7,667	10,155
variable (worst case)	4026	81,328	14,755	21,400	17,681

The worst case would occur if the cluster had brokers with ids greater than 134,217,727, and for topics with more than that many partitions and where the error code was >255.

Since most of the data in a typical `MetadataResponse` is partition data, such a change would make typical responses substantially smaller.

Scope

This KIP proposes a mechanism for allowing RPCs (including new versions of existing RPCs) to use varints.
It does **not** propose any changes to existing RPC messages to make use of the new encoding.
It is envisaged that RPCs will make use of this functionality as those RPCs get changed under other KIPs and guided by benchmarking about the costs and benefits.

Public Interfaces

Field specs in the protocol message JSON format will get support for a new `encoding` property, which will define, for each `version` of the field, how the value should be encoded. This approach makes encoding a first-class concept, separating the logical type of a field from how it is encoded on the wire.

The value of the `encoding` property will be either a JSON object or a JSON string:

- When it is an object each key defines a version range and the corresponding value is named encoding used for the field for those versions.
- When it is a string the value is the named encoding to be used for all versions defined in the FieldSpec's `versions` property.

It will be a generation-time error if:

- `encoding` is present on a field spec with type other than `int16`, `int32` or `int64`.
- the union of the versions defined by `encoding` do not exactly equals the `versions` of the field.
- any pair of version ranges defined by `encoding` have a nonempty intersection.

The names of the supported encodings match the regular expression `(fixed|packed|upacked)(16|32|64)`. "upacked" is short for "unsigned packed". For example:

- `fixed32` is the fixed-size encoding of a 32 bit integer
- `packed32` is variable signed encoding of a 32 bit integer

Any other value for an encoding name will be a generation-time error.

Including the number of bits in the name of the `encoding` (when it's already present in the fields `type`) provides a path to evolving field schemas from 32 to 64 bits. Specifically, a field might originally have been defined

```
{ "name": "tooSmall", "type": "int32", ... }
```

This could be changed (e.g. in version 2 of the message) to:

```
{ "name": "tooSmall", "type": "int64",
  "encoding": { "0-1": "fixed32", "2+", "2+": "fixed64"} }
```

This change would result in type of the `tooSmall` property in the Java representation changing from `int` to `long` (a one-time refactoring). But protocol compatibility would be maintained because:

- In versions 0 and 1 an `int` (using fixed encoding) would be read from the buffer, and promoted to a `long`. When writing, the `long` the value would be range checked prior to downcasting to a `int` and writing using the fixed encoding.
- In versions 2 and above a `long` (using fixed encoding) would be read from the buffer. When writing the `long` value would be written using the fixed encoding.

Using this mechanism:

- fields can evolve between fixed and variable length encodings without any refactoring of the Java code, requiring only a RPC version change.
- fields can evolve from fewer to more numbers of bits between versions requiring only a one-off refactoring. Contrast this to having two distinct fields of different types (and thus different names) existing in different versions of a message.

The default when no `encoding` is present on a field is to use the fixed encoding of the appropriate type.

Example

```
{ "name": "LeaderId",
  "type": "int32",
  "versions": "0+",
  "entityType": "brokerId",
  "about": "The ID of the leader broker.",
  "encoding": {
    "0-9": "fixed32",
    "10+": "unsigned32"
}}
```

Proposed Changes

The message generator will be modified to encode values using the encoding defined for the message's version and relevant `type`.

Compatibility, Deprecation, and Migration Plan

The proposal is backwards compatible: Clients using existing API versions will continue to use fixed-size encoding.

New versions of existing RPC messages will be able to use variable length encoding on a per-field basis.

Rejected Alternatives

- Simply adding support for `varint32` types would in its own allow these types to be used for existing fields.