Robust Message Serialization in Apache Kafka Using Apache Avro, Part 1

July 19, 2018 (http://blog.cloudera.com/blog/2018/07/robust-message-serialization-in-apache-kafka-using-apache-avro-part-1/) | By Andras Beni (http://blog.cloudera.com/?guest-author=Andras Beni) | 4 Comments (http://blog.cloudera.com/blog/2018/07/robust-message-serialization-in-apache-kafka-using-apache-avro-part-1/#comments)
Categories: Avro (http://blog.cloudera.com/blog/category/avro/) CDH (http://blog.cloudera.com/blog/category/cdh/) How-to (http://blog.cloudera.com/blog/category/how-to/) Kafka (http://blog.cloudera.com/blog/category/kafka/)

In Apache Kafka, Java applications called producers write structured messages to a Kafka cluster (made up of brokers). Similarly, Java applications called consumers read these messages from the same cluster. In some organizations, there are different groups in charge of writing and managing the producers and consumers. In such cases, one major pain point can be in the coordination of the agreed upon message format between producers and consumers.

This example demonstrates how to use Apache Avro (http://blog.cloudera.com/blog/2009/11/avro-a-new-format-for-data-interchange/) to serialize records that are produced to Apache Kafka while allowing evolution of schemas and nonsynchronous update of producer and consumer applications.

Serialization and Deserialization

A Kafka record (formerly called message) consists of a key, a value and headers. Kafka is not aware of the structure of data in records' key and value. It handles them as byte arrays. But systems that read records from Kafka do care about data in those records. So you need to produce data in a readable format. The data format you use should

- Be compact
- Be fast to encode and decode
- Allow evolution
- Allow upstream systems (those that write to a Kafka cluster) and downstream systems (those that read from the same Kafka cluster) to upgrade to newer schemas at different times

JSON, for example, is self explanatory but is not a compact data format and is slow to parse. Avro is a fast serialization framework that creates relatively compact output. But to read Avro records, you require the schema that the data was serialized with.

One option is to store and transfer the schema with the record itself. This is fine in a file where you store the schema once and use it for a high number of records. Storing the schema in each and every Kafka record, however, adds significant overhead in terms of storage space and network utilization. An other option is to have an agreed-upon set of identifier-schema mappings and refer to schemas by their identifiers in the record.

From Object to Kafka Record and Back

Producer applications do not need to convert data directly to byte arrays. KafkaProducer is a generic class that needs its user to specify key and value types. Then, producers accept instances of `ProducerRecord` that have the same type parameters. Conversion from the object to byte array is done by a Serializer. Kafka provides some primitive serializers: for example, `IntegerSerializer`, `ByteArraySerializer`, `StringSerializer`. On consumer side, similar Deserializers convert byte arrays to an object the application can deal with.

So it makes sense to hook in at Serializer and Deserializer level and allow developers of producer and consumer applications to use the convenient interface provided by Kafka. Although latest versions of Kafka allow

```
public class VersionedSchema {
  private final int id;
  private final String name;
  private final int version;
  private final Schema schema;

  public VersionedSchema(int id, String name, int version, Schema schema) {
    this.id = id;
    this.name = name;
    this.version = version;
    this.schema = schema;
  }
```

```
public interface SchemaProvider extends AutoCloseable {
  public VersionedSchema get(int id);
  public VersionedSchema get(String schemaName, int schemaVersion);
  public VersionedSchema getMetadata(Schema schema);
}
```

```
private VersionedSchema getSchema(T data, String topic) {
  return schemaProvider.getMetadata( data.getSchema());
}
```

`SchemaProvider` objects can look up the instances of `VersionedSchema`.

```
private void writeSchemaId(ByteArrayOutputStream stream, int id) throws IOException {
    try (DataOutputStream os = new DataOutputStream(stream)) {
    os.writeInt(id);
  }
}
```

Serializing Generic Data

When serializing a record, we first need to figure out which Schema to use. Each record has a `getSchema`

```
private void writeSerializedAvro(ByteArrayOutputStream stream, T data, Schema schema) throws IOException {
  BinaryEncoder encoder = EncoderFactory.get().binaryEncoder(stream, null);
  DatumWriter<T> datumWriter = new GenericDatumWriter<>(schema);
  datumWriter.write(data, encoder);
  encoder.flush();
}
```

```
```

```
public class KafkaAvroSerializer<T extends GenericContainer> implements Serializer<T> {

  private SchemaProvider schemaProvider;

  @Override
  public void configure(Map<String, ?> configs, boolean isKey) {
    schemaProvider = SchemaUtils.getSchemaProvider(configs);
  }

  @Override
  public byte[] serialize(String topic, T data) {
    try (ByteArrayOutputStream stream = new ByteArrayOutputStream()) {
      VersionedSchema schema = getSchema(data, topic);

      writeSchemaId(stream, schema.getId());
      writeSerializedAvro(stream, data, schema.getSchema());
      return stream.toByteArray();
    } catch (IOException e) {
      throw new RuntimeException("Could not serialize data", e);
    }
  }
```

```
@Override
public void configure(Map<String, ?> configs, boolean isKey) {
  this.schemaProvider = SchemaUtils.getSchemaProvider(configs);
  this.readerSchemasByName = SchemaUtils.getVersionedSchemas(configs, schemaProvider);
}
```

```
@Override
public GenericData.Record deserialize(String topic, byte[] data) {
  try (ByteArrayInputStream stream = new ByteArrayInputStream(data)) {

    int schemaId = readSchemaId(stream);
    VersionedSchema writerSchema = schemaProvider.get(schemaId);

    VersionedSchema readerSchema =
        readerSchemasByName.get(writerSchema.getName());
    GenericData.Record avroRecord = readAvroRecord(stream,
        writerSchema.getSchema(), readerSchema.getSchema());
    return avroRecord;
  } catch (IOException e) {
    throw new RuntimeException(e);
  }
}

private int readSchemaId(InputStream stream ) throws IOException {
  try(DataInputStream is = new DataInputStream(stream)) {
    return is.readInt();
  }
}

private GenericData.Record readAvroRecord(InputStream stream, Schema writerSchema, Schema readerSchema) throws IOException {
  DatumReader<Object> datumReader = new GenericDatumReader<>(writerSchema,
      readerSchema);
  BinaryDecoder decoder = DecoderFactory.get().binaryDecoder(stream, null);
  GenericData.Record record = new GenericData.Record(readerSchema);
  datumReader.read(record, decoder);
  return record;
}
```

record.

```java
@Override
public void configure(Map<String, ?> configs, boolean isKey) {
  String className = configs.get(isKey ? KEY_RECORD_CLASSNAME : VALUE_RECORD_CLASSNAME).toString();
  try (SchemaProvider schemaProvider = SchemaUtils.getSchemaProvider(configs)) {
    Class<?> recordClass = Class.forName(className);
    Schema writerSchema = new
        SpecificData(recordClass.getClassLoader()).getSchema(recordClass);
    this.writerSchemaId = schemaProvider.getMetadata(writerSchema).getId();
  } catch (Exception e) {
    throw new RuntimeException(e);
  }
}
```

```java
@Override
public T deserialize(String topic, byte[] data) {
  try (ByteArrayInputStream stream = new ByteArrayInputStream(data)) {
    int schemaId = readSchemaId(stream);
    VersionedSchema writerSchema = schemaProvider.get(schemaId);
    return readAvroRecord(stream, writerSchema.getSchema(), readerSchema);
  } catch (IOException e) {
    throw new RuntimeException(e);
  }
}

private T readAvroRecord(InputStream stream, Schema writerSchema, Schema readerSchema) throws IOException {
  DatumReader<T> datumReader = new SpecificDatumReader<>(writerSchema, readerSchema);
  BinaryDecoder decoder = DecoderFactory.get().binaryDecoder(stream, null);
  return datumReader.read(null, decoder);
}
```

Thus we do not need the logic to determine schema from topic and data. We use the schema available in the record class to write records.

Similarly, for deserialization, the reader schema can be found out from the class itself. Deserialization logic becomes simpler, because reader schema is fixed at configuration time and does not need to be looked up by schema name.

Additional Reading

For more information on schema compatibility, consult the Avro specification for Schema Resolution (https://avro.apache.org/docs/1.8.1/spec.html#Schema+Resolution).

For more information on canonical forms, consult the Avro specification for Parsing Canonical Form for Schemas (https://avro.apache.org/docs/1.8.1/spec.html#Parsing+Canonical+Form+for+Schemas).

Next Time...

Part 2 (http://blog.cloudera.com/blog/2018/07/robust-message-serialization-in-apache-kafka-using-apache-avro-part-2/) will show an implementation of a system to store the Avro schema definitions.

streaming (http://blog.cloudera.com/blog/tag/streaming/) use case (http://blog.cloudera.com/blog/tag/use-case/)

4 responses on "Robust Message Serialization in Apache Kafka Using Apache Avro, Part 1"

Qgeff
July 24, 2018 at 7:26 pm
(http://blog.cloudera.com/blog/2018/07/robust-message-serialization-in-apache-kafka-using-apache-avro-part-1/#comment-87960)

Hi, good article but why don't you provide example with the schema-registry open-source solution instead of recreate your own?

Reply ↓

Andras Beni
August 14, 2018 at 12:37 am
(http://blog.cloudera.com/blog/2018/07/robust-message-serialization-in-apache-kafka-using-apache-avro-part-1/#comment-88032)

Hi Qgeff,

Thanks.
Schema Registry is an external component independent of Apache Kafka that runs its own dependent services, but indeed it solves the same problem . You can use it with the Cloudera platform, but Cloudera does not support it. Rather, these posts are intended to provide/illustrate a more lightweight solution (also using Avro) that is sufficient for most use cases.

Reply ↓

Ricardo Gaspar
August 5, 2018 at 9:57 am
(http://blog.cloudera.com/blog/2018/07/robust-message-serialization-in-apache-kafka-using-apache-avro-part-1/#comment-88005)

Hi, nice article.
Could you provide a git repository to have a better look at the code?

Reply ↓

Andras Beni
August 14, 2018 at 12:42 am
(http://blog.cloudera.com/blog/2018/07/robust-message-serialization-in-apache-kafka-using-apache-avro-part-1/#comment-88033)

Hi Ricardo,

thanks.
Part 3 contains a link to the kafka-examples repository in general, but code shown in these posts is available in SchemaProvider subdirectory:
https://github.com/cloudera/kafka-examples/tree/master/SchemaProvider
(https://github.com/cloudera/kafka-examples/tree/master/SchemaProvider)

Reply ↓

‹ Fine Grained Access Control in Cloudera Manager (http://blog.cloudera.com/blog/2018/07/fine-grained-access-control-in-cloudera-manager/)

Introducing Cloudera Altus SDX (Beta) › (http://blog.cloudera.com/blog/2018/07/introducing-cloudera-altus-sdx-beta/)