Using Apache Avro to implement an OpenFlow protocol library

Dmitry Orekhov (dorekhov@infoblox.com), Sandhya Narayan (snarayan@infoblox.com),
Stuart Bailey (sbailey@infoblox.com), Infoblox Inc., Santa Clara, CA

Introduction

The heart of the OpenFlow switch specification is the set of structures used for OpenFlow Protocol messages. The structures, defines, and enumerations in the C header file openflow.h are used to compose OpenFlow messages, which are binary in nature. Implementations are being developed in different languages and environments. In view of this, it would be useful to strictly control the format and serialization of OF Protocol messages from a single point. Moreover, the OpenFlow standard is evolving fast (OpenFlow 1.0, 1.1, 1.2, 1.3, 1.3.1) so evolution and implementation of new versions has to be easy yet well-controlled.

The main idea is to develop a library that encapsulates the OpenFlow protocol, including well-described messages and serialization, and provides programming API and REST API, so that it is possible to use it either as a stand-alone controller or as a library used by other applications. A natural way to solve this problem is to describe the OpenFlow protocol separately in a language-neutral way with schema written in JSON, XML or other similar format, then parse the schema and serialize the data into a binary stream. Our solution, presented in this paper, follows this approach and makes use of the Apache Avro library.

Why Avro

There are three main serialization solutions that we considered: Google Protocol Buffers[1], Apache Avro[2] and Apache Thrift [3]. All of them are free and open-source; yet none of them fully met all our requirements listed below:

1. Non-encoded serialization. Data has to be put on the wire as-is, representing C-structures from the `openflow.h` header file. Thrift and Protobuf use tags (to store metadata about fields), which are put into the byte stream and on the wire, and this is a corner stone of their serialization/deserialization process. Avro stores all metadata inside a special object, which is dynamically created at run-time. Avro encodes data just like Thrift and Protobuf, but the absence of extra-data such as tags is important and gives Avro an advantage.
2. A strict single-point control of OpenFlow protocol messages format: sizes fields, order in which fields are written on the wire, and the values. All the solutions address this point by using schema, but in Avro it is clear and self-describing: data is just serialized byte by byte in the same order as it is present in the Avro schema.
3. An easy and well-controlled schema evolution. At first glance, the Avro approach to schema evolution appears the most complicated. On the other hand, Avro schema objects are elegantly self-describing and dynamic. Unlinke Protobuf or Thrift, it is not necessary in Avro to compile schemas to generate code that handles the data mapping and serialization. Each schema (or the whole protocol) may be represented as an object in the code and may be parsed in runtime. This makes it easy to quickly make simple changes to the protocol for experimental purposes. Further, schema may be represented as a plain text file or any kind of data stream.

Based on these considerations, we chose Apache Avro as the most suitable serialization library for our purposes. Avro's primary use has been in Hadoop, where it can provide both a serialization format for persistent data, and a wire format for communication between Hadoop nodes, and from client programs to the Hadoop services. Avro schemas are usually written in JSON, but it also supports Avro IDL. Data is usually encoded using a binary format, but there is also a JSON-based data encoder, which is useful for prototyping and debugging.

Implementation of OpenFlow protocol using Avro

Avro defines a number of primitive and complex types, which we have used directly to describe some OpenFlow protocol data structures. For other OpenFlow-specific structures we have composed schemas that use some of Avro's primitive and complex data types, as well as some of our schemas. For example, the Avro type 'fixed' is a byte array of fixed length. We have created different numerical types such as uint_16, uint_32 and uint_64 using the 'fixed' type. The Avro 'record' type has been used to define many of the OpenFlow C-structures. The Avro 'enum' type has been used for some named constants.

Enhancements to Avro
As mentioned earlier, Avro schema design and library lacks some important features needed to fully describe the OpenFlow protocol. To address these deficiencies we needed to make the following enhancements to Avro:

1. Add default values for enums.
2. Add bit manipulation support.
3. Enhance byte array types initialization.
4. Add non-encoder/non-decoder. Avro always encodes data during serialization, but OpenFlow protocol requires data to be sent to the wire as-is, byte by byte. So we introduced the notion of a 'non-encoder' encoder into the Avro library to meet this requirement.

Our enhancements to Avro incorporate these features to make it suitable to implement the OpenFlow protocol.

OpenFlow Java API based on Apache Avro
We have developed a new Java API for OpenFlow using the modified Avro library and a set of Avro schema files for the different versions of the OpenFlow protocol. This is made available for use as following: a) Java API and b) REST Service to enable more general use. Figure 1 shows the construction of the Java API from the Avro API and the .apvr schema files.

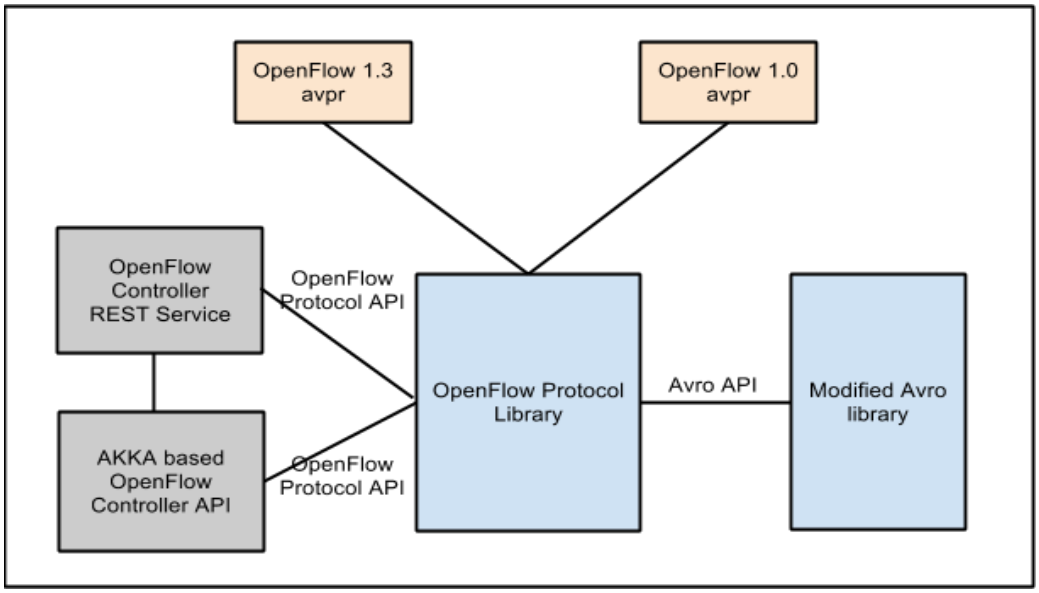

Figure 1

Conclusion
In this paper, we have introduced the use of Apache Avro serialization system in the development of a new Java API for OpenFlow protocols and a REST API for OpenFlow controller. We have described enhancements made to the Avro library to make it suitable for our purposes. Our method for serialization is general and flexible, language-neutral, and uses easy to read schemas written in JSON. It addresses the needs of current and future versions of the OpenFlow protocol and should be applicable to other protocols as well.

References
[1] https://developers.google.com/protocol-buffers
[2] http://avro.apache.org/
[3] http://thrift.apache.org/