Infoblox
CONTROL YOUR NETWORK

Warp OpenFlow Controller and Driver

Work in progress | 1/2014

Introduction

We have developed Warp, a new Java-based OpenFlow controller using Apache Avro protocol serialization library (modified for our purposes) and a set of Avro schema files for the different versions of the OpenFlow protocol. Our work is made available as open source at www.Flowforwarding.org, where we provide code for the following:

- Warp Openflow driver Java library, which can be used to serialize and deserialize OpenFlow messages.
- Warp OpenFlow controller with Java API
- REST Service for Warp OpenFlow controller to enable more general use.

This document deals with the motivation, approach and architecture of Warp. More information about Warp is available at https://github.com/FlowForwarding/warp.

Motivation and Approach

The main tasks of the Warp project were to develop a) a library that prepares or parses OpenFlow messages and serializes or deserializes the bit streams and b) a scalable controller with programming API and REST API. Thus Warp can be used either as a standalone controller or as a library used by other applications. Our goal for developing Warp was to keep up with the rapidly evolving OpenFlow specifications with easy but well-controlled and scalable implementations.

A natural way to address the problem of evolving standards is to describe the protocol in a language-neutral way with schema written in JSON, XML or other similar format. We considered three popular serialization solutions: Google Protocol Buffers[1], Apache Avro[2] and Apache Thrift [3]. While all of them are free and open-source, none of them fully met all our requirements listed below:

a) 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. In contrast, Avro stores all metadata inside a special object, which is dynamically created at run-time. Avro encodes data just like Thrift and Protobuf, but does not use any additional data such as tags, making Avro a natural choice.
b) 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 described in the Avro schema.
c) An easy and well-controlled schema evolution: At first glance, the Avro approach to schema evolution appears to be the most complicated. On the other hand, Avro schema objects are elegantly self-describing and dynamic. Unlike 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 provides both a serialization format for persistent data, and a wire format for communication between Hadoop nodes, and from client programs to the Hadoop services. In 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.

We started with Apache Avro and built upon it by updating Avro syntax to make it more suitable for describing the OpenFlow protocol. Details of the modifications are provided in Appendix A. The Warp OpenFlow driver is built on this modified Avro library and provides a API to create or parse OpenFlow messages. Use of Avro to describe the protocol has made it easy to define, understand and modify the protocol, making it easy to implement future versions of OpenFlow. In addition, Avro has the ability to parse and update protocol description files in runtime, without needing code generation.

The second design goal for Warp was to develop a scalable OpenFlow controller. To address this goal, we based Warp on AKKA [4]. AKKA implements a simple and powerful message-driven actors model for low-latency multitask applications and is designed for high performance and scalability. We have used AKKA to implement a message-driven multitasking controller, which supports multiple connections with OpenFlow switches.

Architecture and Implementation

Figure 1 shows all the components of Warp enclosed in the dotted red line box. The broken purple line box encloses the Warp OpenFlow driver and the broken green line box encloses the Warp Openflow controller. Other controllers (as shown by grey colored third party controller) can use the Warp OpenFlow driver separately. The Rest Service provides a Restful interface to send messages to OpenFlow switches (shown in green) that are connected to the controller. SDN applications can interface to the Warp controller using the controller's Java API. Warp binaries consists of the following:

- warp.jar + .avpr files ----- complete Warp
- warpOFdriver.jar + .avpr files ------ Warp OpenFlow driver for use with third party controllers

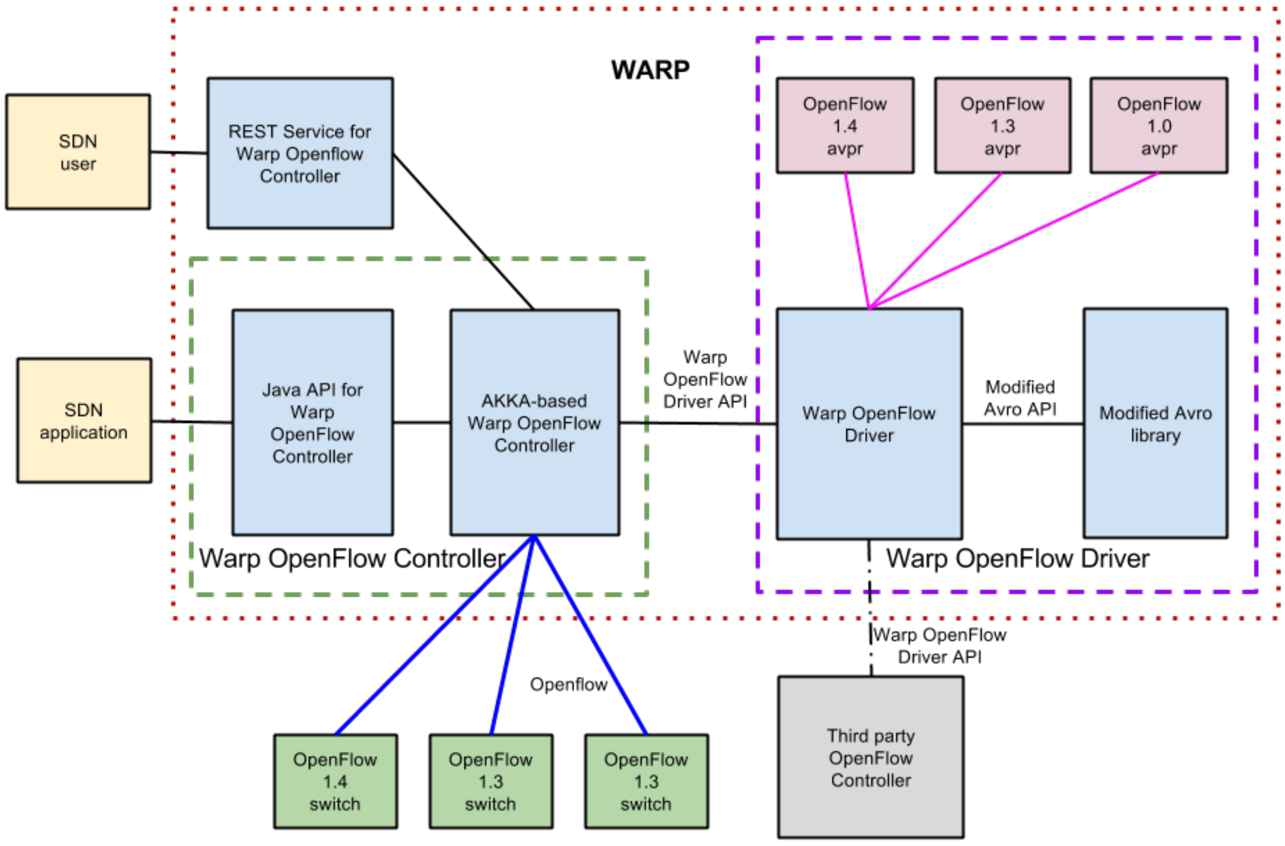

Figure 1

The Warp OpenFlow driver is built from our modified Avro library and the Avro schema files that describe the Openflow protocol. Segments from a sample Avro file are provided in Appendix A as examples to illustrate how the OpenFlow protocol is described in Avro schema. The Avro schema files (of type .avpr) are kept external to the Warp OpenFlow driver jar file, so that they can be modified if needed, without needing to rebuild the jar file. When there is a need to modify the Avro file, say to fix a bug or introduce new features, all that needs to be done is to invoke the "init" method of the message provider class so that the Avro schema file is parsed again and the provider class is regenerated from the modified Avro schema file. (OpenFlow message provider is the class that encapsulates all necessary information about OpenFlow protocol.) Appendix A provides more details of how to use the Warp OpenFlow driver library to generate or parse OpenFlow messages.

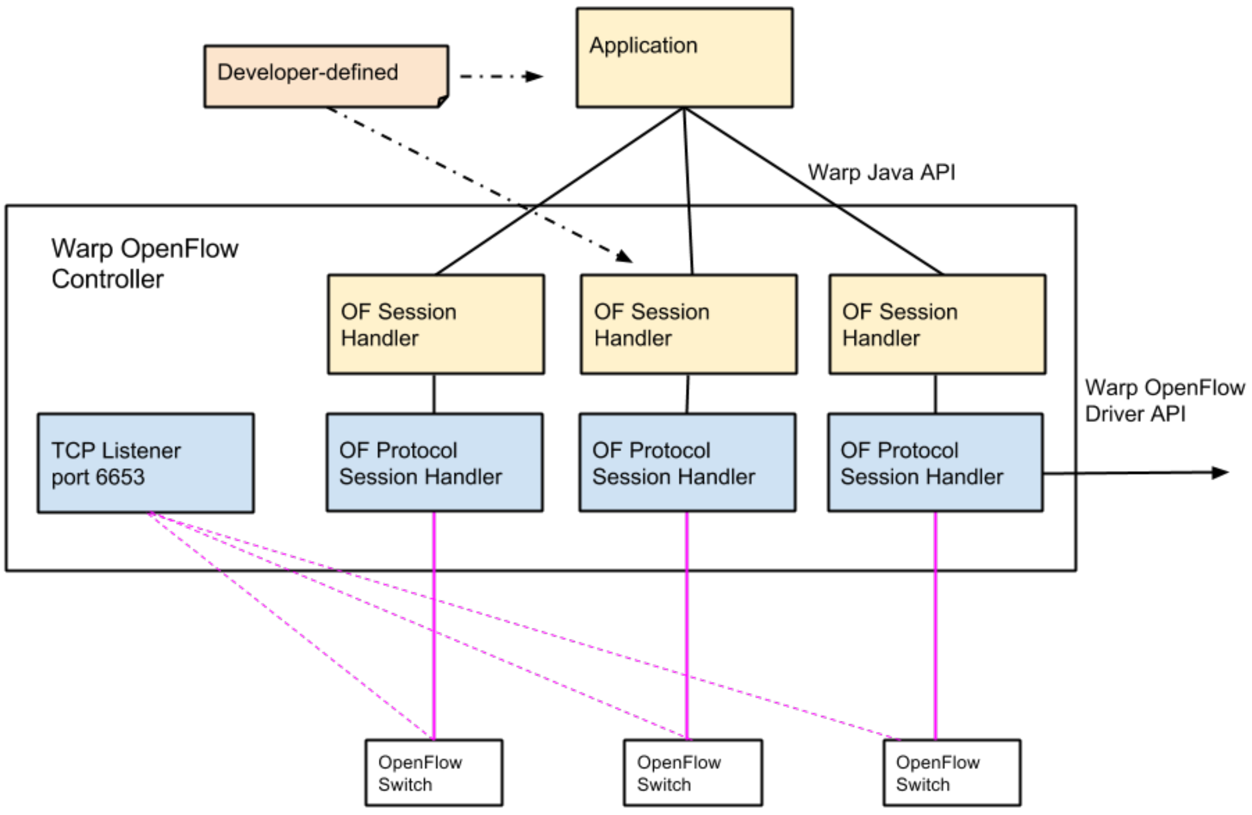

Figure 2

Warp uses the AKKA framework to build a concurrent and scalable controller. AKKA is an asynchronous, non-blocking and highly performant event-driven programming model which provides simple and high-level abstractions for concurrency and parallelism using actors which are lightweight event-driven processes. AKKA is open source and available under the Apache 2 License. AKKA library comes with both Scala and Java API. We have implemented Warp using AKKA's Java API.

The Figure 2 shows the different actors of the Warp controller. The controller itself is the main Supervisor of all other actors. TCP Listener actor listens the port 6653 (configurable). When it gets an incoming connection from an OpenFlow switch, the TCPListener actor establishes a TCP session, launches a OF Session Handler and OF Protocol Session Handler. The OF Protocol Session Handler (one per switch) handles OpenFlow protocol session related behavior such as exchange of "hello" and "echo" messages with the switch. The OF Session

Handler is an application developer defined actor. The developer has to encapsulate all reactions to OF Events in this actor. Sending OpenFlow messages, such as “flow-mod” messages, to the switches is also implemented using events.

Conclusion

We have aimed to achieve a scalable, high performance, modular and extensible architecture. The use of AKKA addresses the requirements of performance and scalability. The use of AVRO addresses the requirements of code clarity and protocol extensibility. Warp OpenFlow protocol driver and Warp OpenFlow controller driver are separate, so it is easy to use the Warp OpenFlow protocol driver alone inside other OpenFlow controllers. Currently the drivers are developed in Java for good portability. In addition, as Avro already has Python and C++ implementations, we plan to port the OpenFlow protocol driver to these languages as well.

Appendix A

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 structures in openflow.h. The Avro ‘enum’ type has been used for some named constants.

Modifications to Avro

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 schema:

1. Add default values for enums.
2. Add bit manipulation support.
3. Enhance initialization of byte array types.
4. Add non-encoder/non-decoder. Avro 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.

Appendix B provides some examples of how Avro is used for describing the OpenFlow protocol.

Example Avro file

Let’s consider a simple example to generate a Hello message for OpenFlow 1.0 protocol. A sample segment of the OpenFlow 1.0 Avro protocol file is given below. It shows how the “ofp_hello” message is composed of “ofp_hello_header”, which in turn is composed of “version”, “type”, “length” and “xid” fields with all the default values filled in to start with.

It is important to note that Avro generates the message fields that have the exact size as per the Avro schema file, and message fields are encoded exactly in the order specified in the Avro file. In addition, the use of ‘default’ values keeps much of the protocol description in the schema and avoids of putting such values in the Java code.

```
==========================================================================
 {"namespace" :"of",
  "protocol":"ofp10",
  "types":[

  {"name":"uint_8", "type":"fixed", "size":1},
  {"name":"uint_16", "type":"fixed", "size":2},
  {"name":"uint_32", "type":"fixed", "size":4},

  {"name":"ofp_type",
   "type":"enum",
   "items":"uint_8",
   "list":[
    {"name":"OFPT_HELLO", "default":[0]}
   ]
  },

  {"name":"ofp_length",
   "type":"enum",
   "items":"uint_16",
   "list":[
   {"name":"OFPL_HELLO_LEN", "default":[0,8]}
   ]
  },

  {"name":"**ofp_header**",
    "type":"record",
    "fields":[
    {"name":"version", "type":"uint_8"},
    {"name":"type", "type":"ofp_type"},
    {"name":"length", "type":"ofp_length"},
    {"name":"xid", "type":"uint_32"}
    ]
  },

  {"name":"**hello_header**",
    "type":"record",
    "fields":[
    {"name":"version", "type":"uint_8", "default":[1]},
    {"name":"type", "type":"ofp_type", "default":"OFPT_HELLO"},
    {"name":"length", "type":"ofp_length", "default":"OFPL_HELLO_LEN"},
    {"name":"xid", "type":"uint_32", "default":[0,0,0,0]}
    ]
  },

  {"name":"**ofp_hello**",
    "type":"record",
    "fields":[
     {"name":"header", "type":"**hello_header**"}
    ]
  }
]
```
==

Java code examples that uses Warp OpenFlow Driver

Java code to prepare a “Hello” message for OpenFlow version 1.0 ready to be sent on the wire is given below:

==

```
import org.flowforwarding.warp.protocol.ofmessages.IOFMessageProvider;
import org.flowforwarding.warp.protocol.ofmessages.IOFMessageProviderFactory;
import org.flowforwarding.warp.protocol.ofmessages.OFMessageProviderFactoryAvroProtocol;

/* This factory builds Message Provider based on Avro protocol file */
		OFMessageProviderFactory factory = new OFMessageProviderFactoryAvroProtocol();
 /* Build and obtain a MessageProvider instance */
		IOFMessageProvider provider = factory.getMessageProvider(“1.0”);
```

```
/* Now we initialize the Message Provider */
            provider.init();
 /* Build a Byte array containing Hello message */
            ByteString out = ByteString.fromArray(provider.encodeHelloMessage());
=============================================================================
```

Let's assume that we have an incoming Hello message. Java sample to decode the Hello message and create a new Hello message of the same the version is shown below:

```
=============================================================================
            Byte[] inHello; // Incoming Hello message should be here
            /* This factory builds Message Provider based on Avro protocol file */
            IOFMessageProviderFactory factory = new OFMessageProviderFactoryAvroProtocol();
            /* Build and obtain a MessageProvider instance based on incoming Hello message */
            IOFMessageProvider provider = factory.getMessageProvider(inHello);
            /* Now we initialize the Message Provider */
             provider.init();
            /* Build a Byte array containing outgoing Hello message */
            ByteString outHello = ByteString.fromArray(provider.encodeHelloMessage());
==================================================================================
```

Java example to create a "FlowMod" message is given below:

```
==================================================================================
import org.flowforwarding.warp.protocol.ofmessages.OFMessageFlowMod.OFMessageFlowModeRef;

            OFMessageFlowModRef fmRef = provider.buildFlowMod();
            fmRef.addTableId("1");
            fmRef.addPriority("256");
            fmRef.addMatchInPort("128");
            OFStructureInstructionRef instrRef = provider.buildInstructionApplyActions();
            instrRef.addActionOutput("12"); fmRef.addInstuction(instrRef);
            byte [] fmBuffer = provider.encodeFlowMod(fmRef);
==================================================================================
```

Java code examples that use Java Warp Controller API
Start a simple Controller with default configuration (port 6653, Avro protocol file in ./resource directory) as follows:

```
==================================================================================
import org.flowforwarding.warp.controller.Controller;
import org.flowforwarding.warp.controller.Controller.ControllerRef;

ControllerRef cRef = Controller.launch(SimpleHandler.class);
Simple Session Handler class:
public class SimpleHandler extends OFSessionHandler {
  @Override
  protected void switchConfig(SwitchRef swH, OFMessageSwitchConfigRef configH) {
    super.switchConfig(swH, configH);
    System.out.print("[OF-INFO] DPID: " + Long.toHexString(swH.getDpid()) + " Configuration: ");
    if (configH.isFragDrop()) {
      System.out.println("Drop fragments");
    }
/*-------------------------------------------------*/
  }

  @Override
  protected void handshaked(SwitchRef swH) {
    super.handshaked(swH);
    System.out.println("[OF-INFO] HANDSHAKED " + Long.toHexString(swH.getDpid()));
    sendSwitchConfigRequest(swH);
  }

  @Override
  protected void packetIn(SwitchRef swRef, OFMessagePacketInRef packetIn) {
    super.packetIn(swRef, packetIn);
    IOFMessageProvider provider = swRef.getProvider();
    OFMessageFlowModRef flowMod = provider.buildFlowModMsg();
    if (packetIn.existMatchInPort()) {
      flowMod.addMatchInPort(packetIn.getMatchInPort().getMatch());
    } else if (packetIn.existMatchEthDst()) {
      flowMod.addMatchEthDst(packetIn.getMatchEthDst().getMatch());
    } else if (packetIn.existMatchEthSrc()) {
      flowMod.addMatchEthSrc(packetIn.getMatchEthSrc().getMatch());
    }

    OFStructureInstructionRef instruction = provider.buildInstructionApplyActions();
    instruction.addActionOutput("2");
    flowMod.addInstruction("apply_actions", instruction);
    sendFlowModMessage(swRef, flowMod);
  }

  protected void flowMod (SwitchRef swRef, OFMessageFlowModRef fmRef) {
    sendFlowModMessage(swRef, fmRef);
  }
}
==================================================================================
```

Appendix B

Avro for OpenFlow: Some techniques

Emulation of Base class (or Interface)

It's possible to use Union to emulate Base class paradigm. In the example below ofp_instruction is a kind of base class for all OF instruction structures. We can refer to it from Java code. The example also shows how it is possible to create an Avro array containing a variable ofp_instructions (instruction_set).

```
==================================================================================
  {"name":"ofp_instruction",
   "type":"record",
   "fields":[
     {"name":"instruction", "type":["ofp_instruction_goto_table",
                                    "ofp_instruction_write_metadata",
                                    "ofp_instruction_write_actions",
                                    "ofp_instruction_clear_actions",
                                    "ofp_instruction_apply_actions"]}
   ]
  },

  {"name":"instruction_set",
   "type":"record",
   "fields":[
     {"name":"set", "type":{"type":"array", "items":["ofp_instruction", "null"]}}
   ]
  },
  {"name":"ofp_flow_mod",
   "type":"record",
   "fields":[
     {"name":"header", "type":"flow_mod_header"},
     {"name":"base", "type":["flow_mod_body_add","flow_mod_body_delete"]},
     {"name":"match", "type":"ofp_match"},
     {"name":"instructions", "type":"instruction_set"}
   ]
},
```

References

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

Contact

Dr. Sandhya Narayan
Principal Researcher
Office of CTO, Infoblox Inc.
snarayan@infoblox.com