The libfluid OpenFlow Driver Implementation

Allan Vidal[1], **Christian Esteve Rothenberg** [2], **Fábio Luciano Verdi**[1]

[1] Programa de Pós-graduação em Ciência da Computação, UFSCar, Sorocaba

[2]Faculdade de Engenharia Elétrica e Computação - FEEC, Unicamp

alnvdl@gmail.com, chesteve@dca.fee.unicamp.br, verdi@ufscar.br

Abstract. *Implementations of network control protocols (such as OpenFlow) are usually divided in two parts: connectivity layer and message layer. The former manages the establishment of a control channel, while the latter provides message handling tools. Current implementations tie them together and are too specific regarding programming language, platform and protocol versions. Providing a clear, separate implementation of the connectivity layer based on formal software engineering techniques can benefit controller, switch and application developers by making development easier. Towards these goals we present the design and implementation of* `libfluid`*, the OpenFlow driver winner of the ONF competition. The tool aims at becoming the default open-source artifact to build OpenFlow switches and controllers by virtue of the design and implementation choices that have led to a developer-friendly and high-performance protocol library, as we will show in our early results evaluation.*

1. Introduction

Software-defined networking (SDN) emerged as a way to control networks programmatically, as a solution to several management problems common to networks worldwide [Feamster et al. 2013]. The OpenFlow protocol [McKeown et al. 2008] was proposed as a solution to the lack of programmability in networks. Introducing new features to networks had become a challenge due to the difficulty in testing new ideas in real-world scenarios.

OpenFlow takes advantage of the fact that most Ethernet networking equipment implement features such as NAT, QoS, firewalls, etc. as programmable structures, but do not expose them to an external interface. A protocol that exposes these interfaces to the hardware could benefit researchers interested in testing new ideas. The OpenFlow protocol provides an abstraction to this hardware features, and exposes them to applications running in external agents. These external agents would be responsible for instructing the hardware on how to behave, thus introducing programmability into the network, at device level.

In this new approach, most of the complexity lies in the external agent and the hardware becomes a slave to external software. This was not a completely new idea by itself; it comes from a long tradition of networking research, which started to gain a clearer focus on central control in the early 2000s, based on the needs of network administrators and traffic engineers who were dealing with increasingly complex and larger networks. Most of this research breaks free from current architectures in use, but they innovate in providing a clearer separation of the control and data forwarding planes.

The introduction of SDN and OpenFlow thus created two new pieces in a network: the switch protocol agent and the controller. Typically, an SDN controller abstraction provides a southbound interface (the protocols that effectively control the network) and a northbound interface (the API exposed to applications).

The southbound part of the paradigm can be understood as being divided in two parts: connectivity layer and message layer. The former manages the establishment of a control channel, while the latter provides message handling tools. Current implementations tie them together and are too specific regarding programming language, platform and protocol versions. Existing OpenFlow protocol implementations are a recent remarkable example of said limitations in practice.

Our work aims to create a low-overhead, portable and configurable SDN control connectivity layer that serves different use cases with a very basic set of tools (using the OpenFlow protocol as a mean).

2. Identified issues

In this session we will present two categories of issues: those that can affect SDN protocols in general (we will use as examples implementations of the OpenFlow protocol), and those which are specific to OpenFlow. We will present an overview of the problem and how current implementations are lacking regarding them.

2.1. SDN protocol implementations issues

Issue 1: a unified connectivity layer for both controllers and equipment agents. The connectivity layer on network control protocols can sometimes be understood as a server-client architecture: a control software (server) listens to connections from equipment (client) asking for a control service; this is the OpenFlow model. Alternatively, a control software (client) actively connects to equipment accepting control orientation (server); this happens in some scenarios in which a controller connects through an available interface in the equipment, such as a Telnet session with configuration automated by the control software.

For OpenFlow, this means that, given proper abstractions, a single solution can be built which will manage both sides of this architecture. Current OpenFlow protocol implementations target only one side of this issues. We want to create simple abstractions so that code can be reused for both purposes.

Issue 2: minimal overhead on applications and no performance compromise. The protocol implementation should not be the core of an SDN controller: it is the applications and network abstractions that are its main purpose. Current protocol implementations (in both controllers and switches) vary a lot regarding this issue: some are very fast and lean, others provide higher-level features at the expense of performance. The introduction of OpenFlow in equipments can also cause an impact on performance due to the limited CPUs present in most current networking hardware. Therefore, any improvements in a protocol agent will be beneficial.

This means that our implementation will have to be as lean as possible, reducing memory and CPU usage.

Issue 3: requirement assumptions of controller applications. Requirements vary wildly in network control applications. The implementation of a controller of networking equipment must allow the user to define which parameters are most important to the applications it will run on the network. It must be possible to build abstractions that reduce development time, or perform well regarding throughput, latency or fairness. It is impossible to provide all these guarantees at the same time, but the architecture must be able to provide them all to the developer with minimal impact.

Current OpenFlow controllers provide different abstractions, but most of them restrict developers to the same pattern of an event handling API. Answering the requirements of an application means optimizing the application for a given controller, since there is not a platform flexible enough for all tasks.

2.2. OpenFlow-specific issues

Issue 4: a single, portable and lightweight implementation. Current OpenFlow protocol implementations are restricted to one or just a handful of programming languages and architectures. While this is fine for general-purpose controllers, we want to provide a lower-level substrate for implementing the protocol while being agnostic to programming language and architectural choices.

Issue 5: protocol version agnosticism. Current implementations of the OpenFlow protocol tie version handling and negotiation in their core. This means that, for every new version, support has to be implemented in both the connectivity layers and the messaging layers, even though there are ways to leave most (if not all) of this task to the message layer implementation.

Issue 6: make it possible to build stand-alone applications. Sometimes, due to hardware constraints or requirements, it might be needed to develop a single, stand-alone application that performs a few functions without the overhead of a full controller. This paves the way for building embeddable applications in limited hardware or simply ad-hoc applications, which can be simpler to deploy and reason about. This might even be suitable in a teaching environment, in which students need to dive straight into the protocol, without having to learn controller-specific abstractions. Most controllers are oriented towards running several applications at the same time, and there is no bare-bones solution that allows programmers to build single-purpose, stand-alone applications. Examples of these applications can be: learning switches, simple routers implementing a custom engine or firewall applications.

3. The libfluid Architecture

libfluid was built in response to the Open Networking Foundation competition [Foundation 2014] and has been chosen as the winner entry (9 were submitted, worldwide), being awarded a cash prize and highlighted as an important implementation of an OpenFlow driver. This section describes the proposed architecture and the rationale behind the design and implementation choices to meet the identified issues in related work while addressing the requirements of the ONF competition.

3.1. Design principles and implementation choices

For each of the issues mentioned above, this is how we plan to tackle them, providing better approaches than those available in existing work.

Issue 1: a unified connectivity layer for both controllers and equipment agents. We will focus on the OpenFlow model and see how it can be implemented so that a single code base can serve for both servers (controllers) and clients (switches), tackling this issue from a formal software-engineering point of view: using inheritance in object orientation, so that common properties and functions are abstracted and can be reused for both servers and clients. How to make this is part of our research.

Issue 2: minimal overhead on applications and no performance compromise. Solving this issue means choosing a side: higher-level abstractions with more features at the cost of performance, or lower-level abstractions with less guarantees, leaving more work for the developer, but also leaving more room for optimization. We have chosen the latter, since we are building a platform, and not a full-fledged solution (such as a controller). This means that our implementation will have to be as lean as possible, reducing memory and CPU usage.

Issue 3: requirement assumptions of controller applications. In our architecture, we plan to leave the task of providing abstraction layers to the user. This means that building upon a single architecture, different requisites for latency, throughput or fairness can be achieved. The Maestro controller outlines different approaches to this problem, and we want to further investigate how their results can be applied to our work [Cai et al. 2011]. We will also explore how different requirements for applications can be implemented using abstractions such as message queues and event/message prioritization in order to validate our architecture.

Issue 4: a single, portable and lightweight implementation. Our solution should be written using low-level tools (i.e.: using C/C++ and dealing with OS abstractions as closely as possible and reasonable). This will help it works on different platforms and reduce the learning curve, since few higher level abstractions will have to be learned and maintained.

Issue 5: protocol version agnosticism. We have devised a solution for this problem using unchanging OpenFlow features to implement version negotiation, meaning that the only requirement for implementing new versions become an implementation of the messaging library. That is: if version X.Y of OpenFlow protocol specification is released, implementing it will be a matter of dealing with raw protocol messages. The underlying control channel will be agnostic to this, since it supports all OpenFlow versions, as long as basic concepts are kept (message header structure). Solving this issue requires us to consider the fundamental guarantees of the OpenFlow specification and implement them in a flexible manner.

Issue 6: make it possible to build stand-alone applications. Solving this issue means allowing the code to be used as part of a program, effectively making our architecture behave like a software library that can be embedded rather than an executable solution. This means that our solution will not have a single entry-point of execution, and different parts of it must be easy to embed and reuse as needed.

3.2. General architecture overview

We arrived at the following practical guidelines for solving the issues regarding both general SDN implementation issues and OpenFlow specific ones:

1. Our work must be implemented as a library that can be bundled into other software, answering issues 1, 4 and 6.
2. We will use multi-threading in order to make use of the parallelism available in modern CPUs, improving performance. By changing the way multi-threading is deployed, we can also work on answering issue 3.
3. An event-loop library will be used to provide an optimized way of dealing with sockets and other operating system abstractions, making the code easier to port to other platforms, making our work on issues 2 and 4 easier.
4. We want to use object-oriented abstractions in order to allow for easier reasoning for most developers, helping us tackle issues 1 and 3.
5. By completely separating protocol and connectivity layers, issue 5 can be solved.

Figure 1 shows a general outline of how these decisions can be implemented on a real solution.

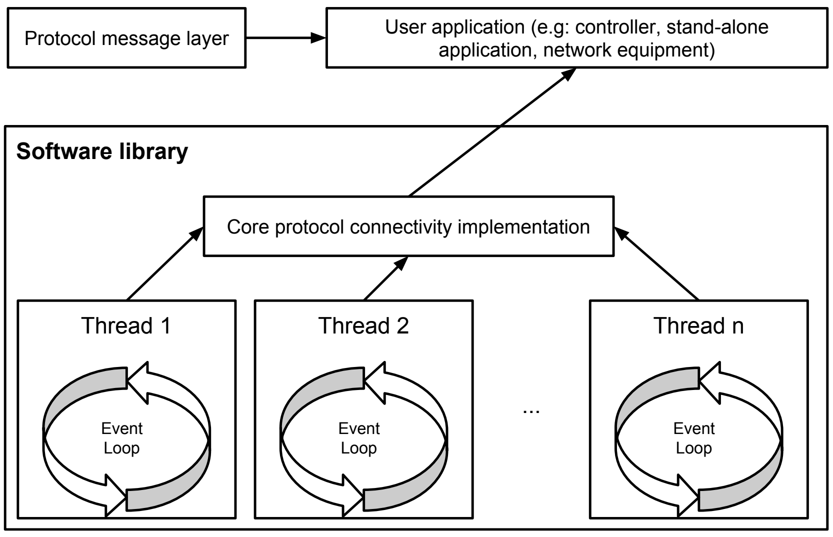
Protocol message layer
User application (e.g: controller, stand-alone application, network equipment)
Software library
Core protocol connectivity implementation
Thread 1
Thread 2
Thread n
Event Loop
Event Loop
...
Event Loop

Figure 1. Arcthiecture overview

3.3. OpenFlow specific design choices

The problem of introducing OpenFlow support into different pieces of software and hardware can be roughly divided in two parts. Figure 1 illustrates how these pieces fit together.

1. The OpenFlow control channel requires the establishment of a network session (in the traditional sense): a transport layer session (TCP/UDP, optionally SSL-protected). This abstraction usually is implemented in the controller itself, and sometimes directly exposed to the application. It involves the connection establishment and protocol handshake, and most importantly, an event abstraction layer that can benefit several types of applications.

2. After this session is established, OpenFlow messages need to be exchanged, and building and parsing them is necessary for the use of the protocol. Different implementations of this are provided in a modular and easily exportable manner in libraries such as OpenFlowJ (written in Java and used in Beacon, OpenDaylight and Floodlight) and libopenflow (used by NOX and POX).

Our architecture manages the connectivity layer necessary for implementing the OpenFlow protocol and can be associated with any OpenFlow message parsing/building library. Our focus is thus on the first part of the problem. Even though our work is oriented towards the OpenFlow protocol, its results can be applied to any protocol that exposes network programmability through a control channel.

Additionally, this connectivity layer will be flexible enough to allow the implementation of different platforms: controllers running several applications, switches connecting to a controller or simply a network application that uses the OpenFlow protocol to perform its functions.

4. Implementation Details

The libfluid library as a client-server architecture, in which a controller is a server and a switch is a client. The library is divided in two parts, taking the direction we outlined in Section 3: libfluid_base and libfluid_msg. The first one is that one of most interest to our research, and the second one will be mentioned and used when appropriate.

An overview of the implemented architecture is illustrated in Figure 2, showing the overall architecture of a system that uses libfluid for a controller implementation (YourController). Different clients (OpenFlow switches) will be assigned to multiple threads managed by libfluid_base using a round-robin distribution algorithm (improving this attribution scheme will be part of our future research). Each of these threads will be responsible for reading and writing data to sockets. Whenever data is read, a callback with user-defined behavior is called. Upon this very simple abstraction, all message processing and application workflow has to be implemented. In Figure 2, libfluid_msg is being employed for dealing with protocol specific messaging, and the YourController class implements application workflow.

Another important part in our solution and a core aspect in our research is providing the support for different OpenFlow versions. We decided to leave the responsibility for dealing with each version of the protocol to libfluid_msg. libfluid_base is completely agnostic to OpenFlow versions and message parsing. The user specifies the OpenFlow versions its application or controller will support, and the driver performs the version negotiation, assuming only a very simple OpenFlow header (guaranteed by the specification to never change) to negotiate the version upon connection establishment. The user is then responsible for parsing the messages accordingly (using libfluid_msg or another library).

4.1. Event model and IO

A library is used for providing an event loop that runs in each thread, dealing with socket management and IO (libevent [Mathewson and Provos 2011]). A connection's event callbacks are guaranteed to run in a single thread during its lifecycle. A user specifies how many threads it wants to use when initializing the driver, and these threads will be responsible for listening to connection events. The number of threads will be defined by the user, but to avoid scheduling overhead, it should typically match the number of available CPUs. The first thread also has the task of listening for new connections. The user must implement two methods for handling major types of events: message and connection events. Message events happen whenever a message arrives, and connection

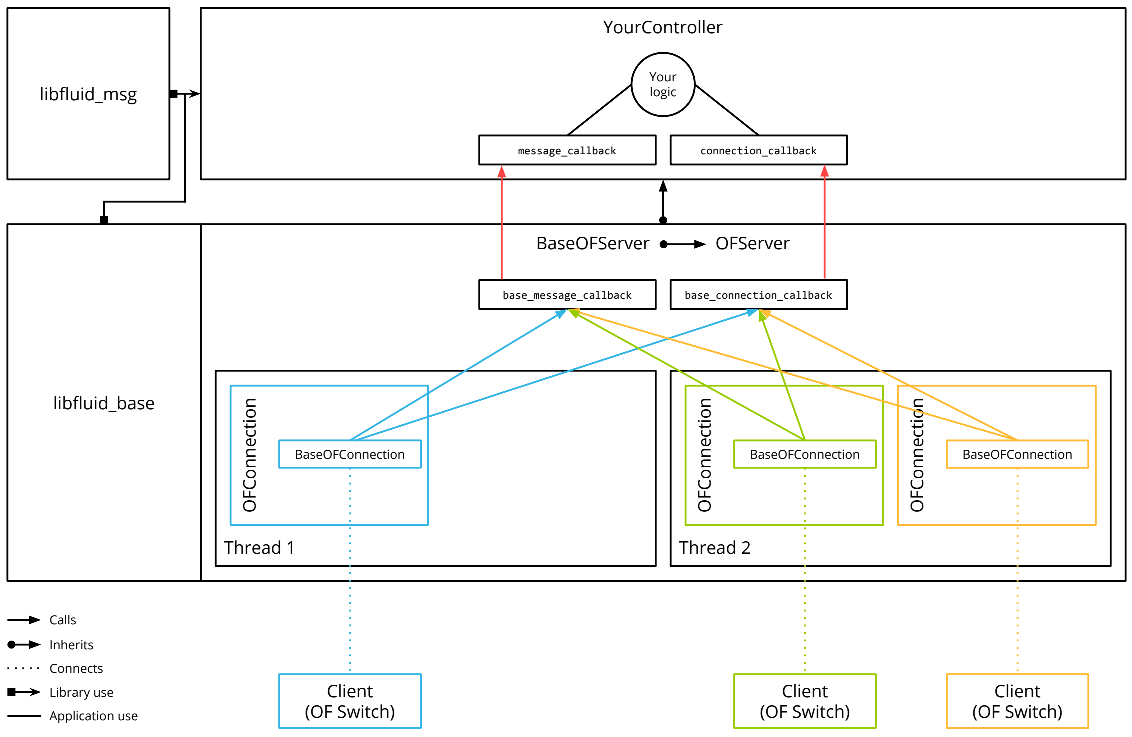

Figure 2. libfluid architecture

events when the connection is interrupted, fails or becomes stale. The user needs to implement the two callback methods and call a start method to have a bare-bones OpenFlow controller running.

The way of deploying threading is a core issue to our research and it impacts on design decisions on several of the issues mentioned in Section 3. Instead of the current model, an event queue could be provided to the user. However, this would impose an impact on the performance, since this structure needs to be synchronized, a result similar to the one achieved in Beacon [Erickson 2013]. Since different users of the driver will have different requirements, we did not want to limit the ways it can be used. A synchronized queue of events can still be implemented on top of this multithreaded architecture, so we delegate this responsibility to application developers. We are performing investigations on how this can be improved by using wait-free synchronization queues for lower priority messages [Kogan and Petrank 2011], while still allowing the user to have a low latency, dedicated event treatment directly in the connections threads. This makes it possible to have different applications dealing with events with different priorities.

4.2. Programming language choice

The submission requirements listed C/C++ as the preferred languages. We chose to follow this recommendation, since the team was used to these languages. We chose C++ due to its higher level abstractions which make it easier (and clearer) to introduce software-engineering concepts, while keeping a low-level implementation. However, we avoided using advanced C++ features (such as templates, smart pointers, etc.) in order to keep the solution fast, small and portable, reducing the number of classes and the overall complexity of the architecture.

5. Demonstration at SBRC 2014

The demonstration will be divided in 5 parts. These parts will be presented directly in the stand to small groups of interested people.

1. We will show libfluid's website, and the resources available.
2. Using the website resources, we will give a quick overview of the architecture, highlighting key points of interest to developers (events, callbacks and message building and parsing). Examples of applications and uses for the driver will be briefly discussed.
3. A quick overview of the installation steps will be shown on a Linux PC (it is a simple and quick installation process).
4. After installing libfluid, we'll show how to write and compile a dummy compiler that doesn't do anything, but keeps connections to OpenFlow switches alive. This example will be coded live, and it's composed of just a few lines of code.
5. Finally, we will talk about a previously written learning switch application and show it running with Mininet. We will quickly talk about how the sample controller and application were built.

Code, documentation, and initial performance benchmarks are available at:
`http://opennetworkingfoundation.github.io/libfluid/`

6. Conclusion

By recognizing the distinct nature of the different parts in an SDN control protocol, we have identified a few issues in available solutions and possible improvements to advance the state of the art in OpenFlow driver implementations. Our solution, `libfluid`, shows promising results and has been selected as the winning choice by the ONF. We intend to improve the tool by applying a research-oriented approach to solving issues in the current implementation (e.g., event model, protocol version agnosticism and a low-overhead implementation) and also to future work (e.g., formalization of the client-server architecture, higher-level abstractions).

References

[Cai et al. 2011] Cai, T. Z., Cox, A. L., and Ng, T. E. (2011). Maestro: balancing fairness, latency and throughput in the OpenFlow control plane. Technical report, Rice University Technical Report TR11-07.

[Erickson 2013] Erickson, D. (2013). The Beacon OpenFlow Controller. In *HotSDN*. ACM.

[Feamster et al. 2013] Feamster, N., Rexford, J., and Zegura, E. (2013). The Road to SDN. *Queue*, 11(12):20.

[Foundation 2014] Foundation, O. N. (2014). Open Networking Foundation. `https://www.opennetworking.org/`. [Access: 15-Feb-2014].

[Kogan and Petrank 2011] Kogan, A. and Petrank, E. (2011). Wait-free queues with multiple enqueuers and dequeuers. *ACM SIGPLAN Notices*, 46(8):223–234.

[Mathewson and Provos 2011] Mathewson, N. and Provos, N. (2011). libevent. `http://libevent.org/`. [Access: 15-Feb-2014].

[McKeown et al. 2008] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008). OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74.