Allan Vidal

libfluid: a lightweight OpenFlow framework

Sorocaba, SP

8 de abril de 2015

Allan Vidal

libfluid: a lightweight OpenFlow framework

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação (PPGCCS) da Universidade Federal de São Carlos como parte dos requisitos exigidos para a obtenção do título de Mestre em Ciência da Computação. Área de concentração: Arquiteturas Distribuídas de Software.

Universidade Federal de São Carlos – UFSCar

Centro de Ciências em Gestão e Tecnologia – CCGT

Programa de Pós-Graduação em Ciência da Computação – PPGCCS

Orientador: Fábio Luciano Verdi

Co-orientador: Christian Rodolfo Esteve Rothenberg

Sorocaba, SP

8 de abril de 2015

Vidal, Allan.
V649l libfluid : a lightweight OpenFlow framework. / Allan Vidal. – – 2015.
108 f. : 30 cm.

Dissertação (mestrado)-Universidade Federal de São Carlos, *Campus* Sorocaba, Sorocaba, 2015
Orientador: Fábio Luciano Verdi
Banca examinadora: Carlos Alberto Kamienski, Gustavo Maciel Dias Vieira
Bibliografia

1. Rede de computador - protocolo. 2. Engenharia de software. I. Título. II. Sorocaba-Universidade Federal de São Carlos.

CDD 004.6

Ficha catalográfica elaborada pela Biblioteca do *Campus* de Sorocaba.

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências em Gestão e Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado do candidato Allan Vidal, realizada em 08/04/2015:

Prof. Dr. Fabio Luciano Verdi
UFSCar

Prof. Dr. Carlos Alberto Kamienski
UFABC

Prof. Dr. Gustavo Maciel Dias Vieira
UFSCar

To all my family and friends.

Acknowledgements

I'd like to thank,

my parents, Lenita and Reinaldo Vidal, for their hard work, their full support in my life and throughout my studies,

my coworker, Eder Fernandes, for the partnership in this work and for patiently guiding me into the SDN world,

my coworker, Marcos Salvador, for supporting and pushing forward awesome work, even when it sounded crazy and impossible,

my advisors, Fabio Verdi and Christian Rothenberg, for guiding me into the world of networking, giving me new ideas when I was short of them and for their hard work reviewing and advising my work,

the people at CPqD, for providing a pleasant work environment and all the conditions for part of this work to get done,

my girlfriend, Adriana Amaral, for the partnership in life during this work (including a review of this dissertation), and for always understanding why I was away from her while working on this,

my host parents as an exchange student in the US, Don and Gina Arnold, for taking me in their home a few years ago, making the task of writing this dissertation in English much easier,

the members of the examination board for this dissertation, Gustavo Vieira and Carlos Kamienski, for their valuable feedback,

all the awesome people who build the knowledge and the software that makes this work possible,

all my teachers, who from early on taught me lots of the little and big things that go into this work.

"[...] I still fervently believe that the only way to make software secure, reliable, and fast is to make it small. Fight Features."

– Andrew S. Tanenbaum

"The nice thing about standards is that you have so many to choose from."

– Andrew S. Tanenbaum

"The best code is no code at all."

– Jeff Atwood

Abstract

Software-defined networking (SDN) introduces a network control paradigm that is centered in controller software that communicates with networking devices via standardized protocols in order to configure their forwarding behavior. Current SDN control protocol implementations (such as OpenFlow) are usually built for one controller or networking device platform, and restrict choices regarding programming languages, protocol versions and feature. A single software architecture that enables controllers and networking devices to use the OpenFlow protocol (for existing and future protocol versions) can benefit network application developers and manufacturers, reducing development effort. Towards this goal, we present libfluid: a lightweight (simple and minimalistic) framework for adding OpenFlow support wherever it is needed. We built a single code base for implementing protocol support in a portable, fast and easy to use manner, a challenge that involved technology choices, architectural decisions and the definition of a minimal API. The implementation was shown to work in all proposed scenarios and contributes to the state-of-the-art with a few novel paradigms for OpenFlow frameworks.

Keywords: Computer networks. Software-defined networks. OpenFlow protocol.

Resumo

Redes-definidas por software (SDN) introduzem um paradigma de controle de redes que é centralizado em um software controlador, que se comunica com dispositivos de rede através de protocolos padronizados para configurar suas políticas de encaminhamento. Implementações existentes de protocolos SDN (como OpenFlow) são geralmente construídas para uma plataforma de controlador ou dispositivo de rede e restringem escolhas como linguagem de programação, versões do protocolo a serem usadas e características suportadas. Uma arquitetura de software que permita controladores e dispositivos de rede usarem o protocolo OpenFlow (em versões existentes e futuras) pode beneficiar desenvolvedores de aplicações de redes e fabricantes, reduzindo o esforço de de desenvolvimento. Para este fim, apresentamos libfluid: um arcabouço leve (simples e minimalista) para adicionar suporte a OpenFlow onde ele for necessário. Construímos uma única base de código para implementar suporte ao protocolo de maneira portável, rápida e fácil de usar, um desafio que envolve escolhas de tecnologia, decisões arquiteturais e a definição de uma API mínimalística. A implementação foi testada com sucesso em todos os cenários propostos e contribui com o estado da arte através de alguns novos paradigmas para arcabouços OpenFlow.

Palavras-chave: Redes de computadores. Redes definidas por software. Protocolo OpenFlow.

List of Figures

List of Tables

List of abbreviations and acronyms

API	Application Programming Interface
HTTP	Hypertext Transfer Protocol
IO or I/O	Input/Output
IP	Internet Protocol
MAC	Media Access Control
MPLS	Multiprotocol Label Switching
NAT	Network Address Translation
NETCONF	Network Configuration Protocol
OS	Operating System
QoS	Quality of service
SDN	Software-defined Networking
SNMP	Simple Network Management Protocol
TCP	Transmission Control Protocol
TLS	Transport Layer Security (protocols)
VLAN	Virtual Local Area Network
VPN	Virtual Private Network

Contents

1 Introduction

Software-defined networking (SDN) emerged as a proposed solution to several problems common to computer networks, such as the increasingly hard management in the face of ever-growing demands (FEAMSTER; REXFORD; ZEGURA, 2013).

Early ideas proposed ways of segmenting physical networking equipment in logical components, such as the use of VLANs and the deployment of VPNs. However, these solutions still required manual configuration, and making them respond to network changes was not an easy task, since network devices did not expose an easy and standardized interface for monitoring state and modifying configuration programmatically when needed. Furthermore, the lack of a standard for managing and configuring devices brings several challenges when different hardware platforms (from different vendors) are put together.

In the SDN approach, most of the complexity is placed in an external, remote software that controls the hardware. 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 simultaneously with more demanding applications and increasingly complex and larger networks (FEAMSTER; REXFORD; ZEGURA, 2013). Most of this research breaks free from current architectures in use, and their central tenet consists in providing a clearer separation of the management/control and data planes.

Network devices can be divided in three layers (or planes), each representing an activity performed by the device (PEPELNJAK, 2013): they expose a **management plane** to users, which provides command line interfaces and remote access protocols in order to configure the **control plane**, which interacts with other devices in order to enable network functions. Finally, the **data plane** is composed by the parts of the device that effectively process and forward traffic based on configuration and instructions specified by the other planes.

Figure 1 illustrates how the separation between the management/control and data planes happens: administrative functions (belonging to the management plane) and common network logic implemented in network devices (such as routing, QoS and firewalls, which belong to the control plane) are removed from specific devices and placed in the hands of software developers. Network administrators (which can be the developers themselves) then interact with the network through abstractions rather than directly dealing with devices and their traditional administration interfaces (typically, command-line or graphical user interfaces provided by vendors).

The introduction of SDN requires two pieces of software: the **controller** and the

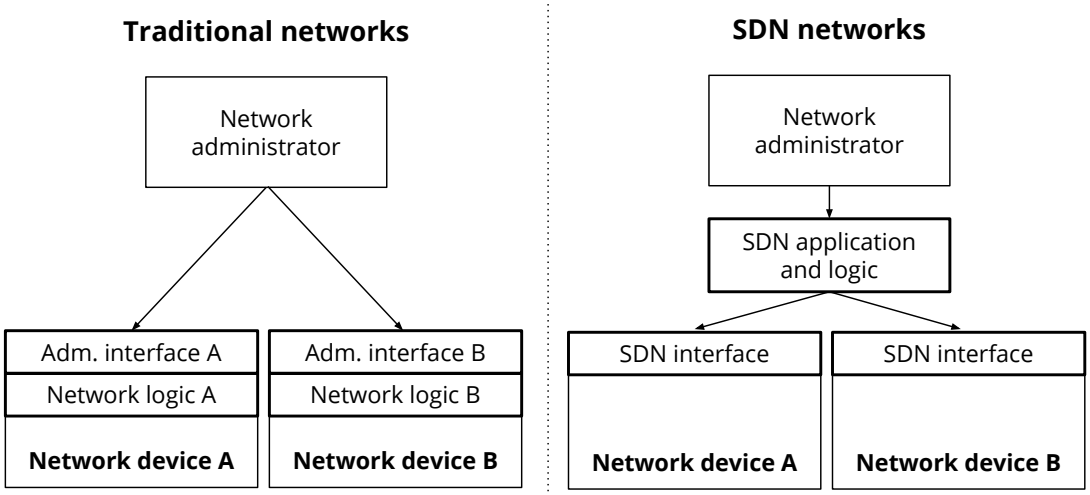


Figure 1 – Overview of traditional and SDN networking approaches.

network device protocol agent. The controller runs in a remote computer and is responsible for instructing network devices on how to make forwarding decisions; in order to communicate with devices, it uses an SDN protocol. The network device protocol agent runs in network devices and is responsible for interpreting the SDN protocol messages and converting the requests into rules and configurations which are then applied to the device.

Typically, an SDN controller provides **southbound interfaces** (the protocols that control the network devices) and a **northbound interface** (the API exposed to developers).

The northbound interface consists of all the programming interfaces that may be useful for developers who are building functionality on top of controllers such as network services and applications (switching, deep packet inspection, firewalls, QoS, routing, etc). We will refer to this custom-developed software as **applications**. There is no standard northbound interface, since this is what differentiates controllers and provide unique features and ways for applications to be developed.

The southbound interface of a controller is responsible for interacting with network devices in order to program behavior into the network. A controller may implement more than one southbound API, so that it can control devices that use different SDN protocols. It is even possible to emulate SDN using traditional configuration interfaces to network devices or popular networking protocols such as SNMP or NETCONF (KREUTZ et al., 2015). Differently from northbound interfaces, southbound interfaces require a standardized specification so that different controllers and network devices can interoperate.

The main purpose of the core of a controller implementation is to provide the bridge between its northbound interface and one or more southbound interfaces. The controller itself may provide additional services and features to applications in order to ease their development and orchestrate the environment.

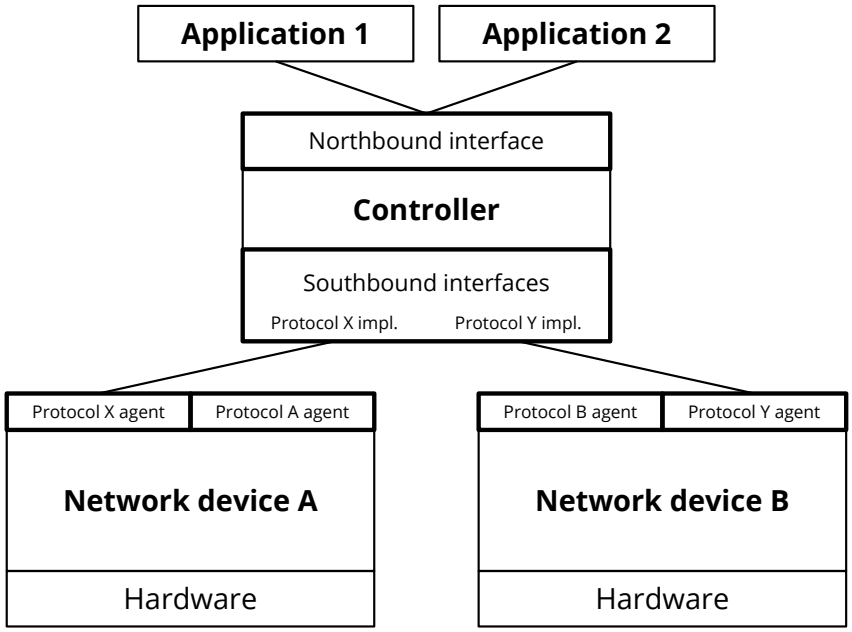


Figure 2 – Northbound and southbound interfaces in an SDN controller.

At the other end of the SDN model, network devices also need to implement the SDN protocol in order to interact with the controller (informing it about network events and receiving instructions). A network device may also support more than one SDN protocol, and support for each of these protocols will be provided by different protocol agents.

More software is needed in the network device for making the conversion between the SDN protocol abstractions and the hardware-specific commands. All of the SDN software stack in a network device, from the protocol agents to the software that effectively implements SDN in the hardware, is typically provided by equipment manufacturers. There are however initiatives like the Open Compute Project - Networking (Open Compute Project, 2014) aiming to develop fully open network devices in which a user would be able to customize the SDN software stack.

Figure 2 illustrates how controllers and network devices expose their interfaces in order to interact with each other, the applications and the underlying hardware in the SDN paradigm. Each network device can expose one or more protocols for controllers to interact with them. As an example in Figure 2, Network device A supports protocols A and X. The controller interacts with this device via the protocols it can use, in this case protocol X.

Both the southbound interfaces of a controller and the protocol agents of a network device share similar functionality. They need to provide two important services:

- Connectivity: manage the establishment of a control channel which is used for communication between the network device and the controller;

- Messaging: provide message handling tools, converting to and from the high-level description used by software and the low-level (binary) protocol message format (wire-format).

One of the most popular approaches to implementing SDN is the OpenFlow protocol (Open Networking Foundation, 2014b) (KREUTZ et al., 2015), which defines the southbound API that controllers can use to communicate with network devices (switches). We detail OpenFlow in Chapter 2.

1.1 Motivation

There is a multitude of controllers and other software implementing the OpenFlow protocol, many targeting different use cases: academic research, large worldwide networks, proof of concept architectures, etc. Network switches from several manufacturers started featuring OpenFlow support in different versions and using different implementations of the protocol agent. It is not an understatement to say that OpenFlow is what has brought SDN into the spotlight of the networking industry.

All this popularity helped the OpenFlow ecosystem gain enormous traction and variety, from academic research networks (BERMAN et al., 2014) (SALLENT et al., 2012) to very large-scale deployments (JAIN et al., 2013). Developing for SDN became almost synonymous of OpenFlow, even though SDN is a much more encompassing concept, while OpenFlow is simply an enabler of SDN.

Therefore, OpenFlow is positioned as an important cornerstone of recent development in SDN. Building software and hardware that supports the OpenFlow protocol is important for those wishing to compete in the industry.

In this dissertation, we analyse existing software implementing the OpenFlow protocol (both as southbound APIs for controllers and as protocol agents for switches) and highligt some issues present in them, such as: little code reuse, inflexible implementations, too much functionality embedded in the core and limited portability.

Based on these issues, we proceed to devise and build a lightweight, configurable, portable and fast OpenFlow protocol implementation that improves on the state-of-the-art. We also give insights into the challenges and trade-offs involved in the implementation of SDN protocols, and more specifically, OpenFlow.

1.2 Contributions

Our key contributions in this work are:

1. **Highlighting the issues that are common to OpenFlow protocol implementations** (and in SDN in general) and establishing a few requirements for improving the state-of-the-art in protocol implementations;

2. **Defining a software architecture for a lightweight OpenFlow implementation** that uses as guidance the acquired set of requirements while introducing novel paradigms for OpenFlow frameworks;

3. **The implementation of the software architecture** in a way that fully satisfies the established requirements and integrates with the existing SDN ecosystem.

Lightweight and portable implementation

We define a **lightweight implementation** of the OpenFlow protocol as one which provides just the most basic tools for protocol support (either for controllers or switch agents), without including additional features such as application management or messaging libraries. We are not interested in defining processor and memory usage when declaring an implementation lightweight, since this is highly dependent on the workload.

A **portable implementation** is one which works on different computer platforms (CPU architectures and operating systems). We also extend the definition of portability to say that a portable implementation should work in different programming languages by the construction of bindings for these languages.

1.3 Text structure

In this Introduction we outlined the fundamental concepts upon which we build our dissertation.

In Chapter 2 we detail background work and related technologies, going deeper into how they are designed and how they relate to our work.

In Chapter 3 we discuss issues in current (and related) work and a few general requirements to solve these issues. Then we propose a software architecture to address these requirements. We further discuss the inner working of this architecture and how it can be implemented.

In Chapter 4 we showcase the implementation of our architecture, detailing how it is built in software and how the components interact with each other.

In Chapter 5 we describe evaluation metrics and approaches in qualitative (how does the implementation fulfill the requirements?) and quantitative terms (how does the implementation perform in benchmarking tests?). We wrap up the chapter with a brief comparison with the related work, highlighting similarities and differences.

In Chapter 6 we conclude the dissertation by analysing its impact and results and presenting possible future work and improvements.

2 Background and related work

In this chapter we will discuss background work (OpenFlow and SDN) in Section 2.1). In order to contextualize our proposal in Chapter 3, we present the state-of-the-art in network device protocol agents, frameworks and libraries that provide OpenFlow functionality to controllers and switches in Sections 2.2, 2.3.

Before we go into details about background and related work, there is an important distinction to be made regarding libraries and frameworks. We will build upon Martin Fowler's definitions (FOWLER, 2004) and slightly adapt it to the realm of OpenFlow software:

- **Library**: a set of functions (organized in classes or not) that can be called by other software in order to do some work and promptly return control.
- **Framework**: a set of (usually abstract) classes with well-defined points of interaction which can be extended by applications in order to provide the desired functionality; the framework code will typically run by itself and be responsible for calling application code when needed.

When applying this to the existing realm of OpenFlow software, we came up with four categories:

- **Controller**: while most controllers are typically implemented as frameworks (per Fowler's definition), they usually provide abstractions to applications at a higher-level then a framework would do (such as application lifecycle management, global view of the network, management interfaces and routing algorithms). Controllers provide the control and management planes in the OpenFlow network model, without being concerned with implementing network device protocol agents or other reusable pieces of code.
- **Switch agent**: the software that goes into switches for implementing the OpenFlow protocol (once again, they typically work as frameworks). This software layer is responsible for sending and receiving protocol messages to/from the controller. When a message requires some action by the hardware, the action is delegated to another layer of software that is responsible for triggering hardware specific features that implement it. Switch agents only provide the switch agent side of the OpenFlow network model, without being concerned with implementing controllers or other reusable pieces of code. Previously in this text, we used the term network device

protocol agents; hereafter, we will use the term switch agent to refer to this type of software.

- **Framework**: a network-oriented framework (as is the case with OpenFlow frameworks), usually only provides fundamental IO/event handling and messaging abstractions to applications, which are called in response to network events. **Controllers are usually built on top of these frameworks**. They may also be used for implementing switch agents. The terms "driver" or "protocol driver" may also be used to refer to frameworks.

- **Messaging library**: software implementing abstractions for manipulating protocol messages (usually object-oriented). They are responsible for converting programming-language constructs representing OpenFlow messages (such as objects) into the network representation of these messages. They can also perform the opposite task: converting the network representation of messages into representations usable by programmers. They typically work with one or a handful of protocol versions. In this text, we may also refer to them as **message building/parsing libraries**.

2.1 OpenFlow

Introducing new features to networks had become a challenge due to the difficulty in testing new ideas in real-world scenarios. A Stanford networking study group released a whitepaper in 2008 detailing their proposal for a solution to the lack of programmability in networks: the OpenFlow network model and its associated protocol (MCKEOWN et al., 2008).

OpenFlow was initially conceived as way for researchers to run their experiments on campus networks, using existing commercial switches adapted to support the OpenFlow protocol. The protocol allows access to some fundamental features of switches, without the need for full disclosure of their inner working.

OpenFlow takes advantage of the fact that most Ethernet switches 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 these 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.

The difference between OpenFlow and previous ideas towards programmable networks is that long tested technologies (e.g.: TCP/IP, Ethernet, VLANs) are embraced

rather then replaced, without excluding the possibility for the future innovation promised by less pragmatic clean-slate solutions (FEAMSTER; REXFORD; ZEGURA, 2013).

As an example, we will explain how packet[1] switching happens in a traditional scenario (using non-OpenFlow switches), and then compare it to what would happen in an OpenFlow environment. We will highlight how software interacts in this workflow, since it will be useful when explaining our work further ahead in this text.

Consider an Ethernet learning switch, a network device that has some degree of distributed knowledge about networking topology inferred from network traffic. When a switch first receives a packet from a connected host (a network connected device), it inspects the packet and creates an association so that traffic is forwarded to the correct destination, avoiding unnecessary flooding to all switch ports in order for packets to reach their destinations (as a hub would do). For the purposes of this example, consider a host X, connected to port A of switch S, attempting to establish bidirectional communication with host Y, connected to port B of switch S.

In a **traditional, non-OpenFlow scenario**, upon receiving an initial network packet from host X, switch S notices the source MAC address of the packet can be used to make an association-pair: (`source MAC address`, `switch port`), meaning that packets having `source MAC address` as destination can be forwarded to the same `switch port` in order to reach their correct destination.

The control software embedded in the hardware of switch S stores this information in a special forwarding memory. When making the decision about to which port a packet should be output, the switch hardware consults the forwarding memory. If the destination address is present in this forwarding memory, the switch reads the port that should be used as an output. If the destination is not present in the forwarding memory, the switch will output a copy of the packet to all ports (a flood). For the purposes of this example, assume the latter case is what happens.

Host Y, upon receiving the packet that was flooded, will eventually respond with another packet. When the response packet reaches switch S, it can then be used to learn the information on how to reach host Y just as it was done with host X. From now on, the switch will not need to analyse the packets' source address and source port when forwarding packets between hosts X and Y. It can simply use the forwarding memory, which is a hardware-optimized structure that allows traffic to be forwarded with high throughput and low latency.

In an **SDN/OpenFlow scenario**, the process is very similar to the one outlined before, but the intelligence which was embedded in hardware is replaced by a controller. Upon receiving the request packet from host X, which does not match any entry in its flow

[1] In order to make the text easier to follow, we may sacrifice perfect terminology throughout this dissertation and use the term "packet" to refer to Ethernet frames, IP datagrams and TCP segments.

table (a structure similar to the forwarding memory illustrated earlier in this example), the switch forwards the packet (encapsulated in an OpenFlow ***packet-in* message**) to its controller via a dedicated control interface[2]. The controller receives the *packet-in* message and, assuming it is implemented as to act as a traditional Ethernet switch, learns that the MAC address of host X can be reached by forwarding packets to the port A, since this is the port from which the packet came from. The controller then sends an OpenFlow ***flow-mod* message** to the switch, which will instruct the switch to store the forwarding rule in its forwarding memory (the flow table). Finally, the controller also instructs the switch to output the packet to all ports (flooding), since it still does not know the packet's correct destination.

Host Y, upon receiving the original packet, will eventually send another packet in response. When this response reaches the switch, it can then be used to learn the forwarding rule on how to reach host Y just as it was done with host X. From now on, the switch will have both entries in its flow table (for hosts X and Y). A flow table is typically a hardware-optimized structure; in many current OpenFlow switches, it is the same component used for non-OpenFlow forwarding.

As highlighted in this example, the most defining part of an OpenFlow switch is its flow table[3] , which contains a set of flow rules (which will be called **flows** from now on) and is controlled by an external agent: the controller. A flow is a rule that will be applied when deciding how to forward a given packet. Packets are matched against flows (each flow has a unique matching description), and an action is taken based on this flow's list of actions. Figure 3 illustrates a flow table being used in an OpenFlow datapath: a packet being sent from host X to host Y is matched against the datapath's flow table, and the appropriate action is taken (output the packet port B).

What makes an OpenFlow flow table different from the forwarding memory in traditional Ethernet switches is that it is programmable. OpenFlow switches are not limited to Ethernet-only (L2) fields and actions: they can also forward traffic based on higher-level network protocols, such as IP, MPLS, TCP and perform actions such as modifying packet headers or dropping packets.

---

[2] This is a simplification which is not valid for all OpenFlow versions, but we will assume that in order to keep the explanation simple

[3] This name is used in the OpenFlow whitepaper (MCKEOWN et al., 2008) to denote the set of forwarding rules a switch uses. Its precise definition depends on the hardware and software implementation of a switch.

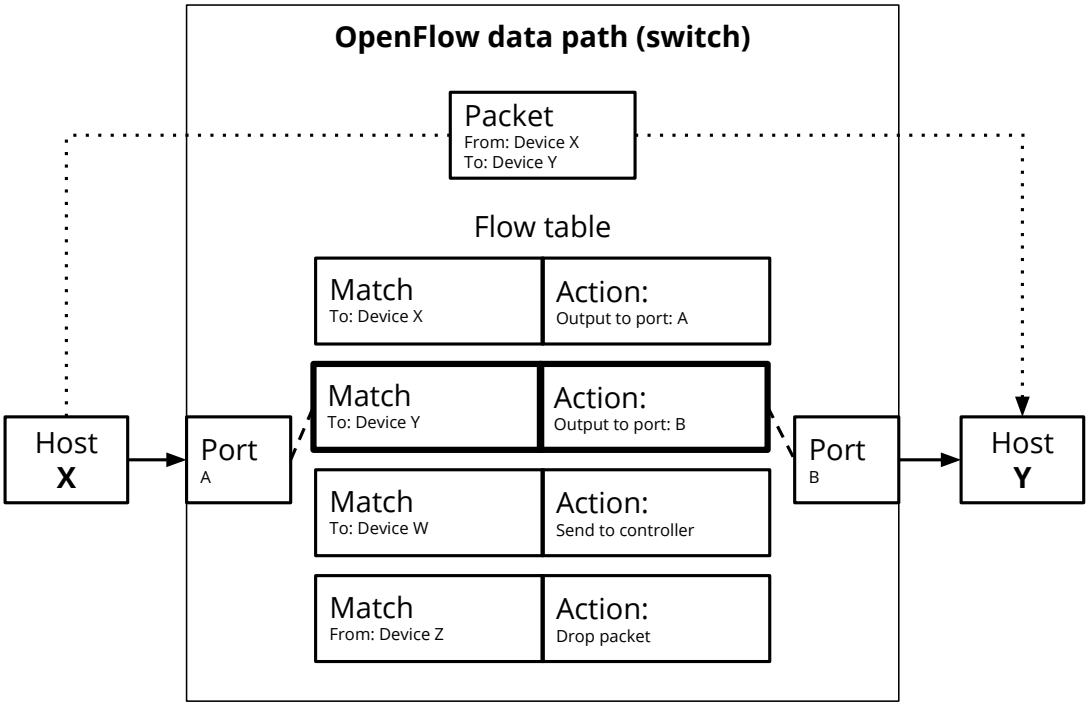


Figure 3 – Flow tables and their role in an OpenFlow datapath.

Control and data path/plane

There are some very important terms that will be used throughout this text:

A **data path** is a structure that contains a flow table and performs the actions defined by this flow table. A switch (or any other network device) is said to support OpenFlow when it implements an OpenFlow data path. A data path is also known as the implementation of the data plane when describing SDN in general. The term **OpenFlow switch** will be used to refer to an **OpenFlow data path** in this text.

The **control path** is the software that instructs a data path on how to behave via control messages and by filling its flow table. It is typically implemented by a **controller**, which is how we will refer to the control path in this text. The control path is also known as the control plane when describing SDN in general (and it may also encompass the management plane).

A **control channel** is used to connect the **control plane** to the **data plane**. In OpenFlow, this channel is usually represented by a TCP connection.

This example also highlights the importance of the control software in the OpenFlow paradigm: it must react quickly, in order to avoid overhead when compared to a local, hardware-based implementation. However, OpenFlow controllers are not responsible for forwarding traffic: they should only instruct switches on how to do that. The optimal way of programming the hardware is before a packet even reaches the switch; in this case, the control application is said to be **proactive**. If a control application only configures the switch to react to an event after it occurs, it is said to be **reactive**.

Most importantly, while behavior cannot be easily changed in hardware, it is very

easy to change it in the control software. It becomes easy to implement new features and networking approaches. It also enables advanced applications to be built, such as firewalls and routers, without the need for explicit hardware support.

Nowadays, different hardware vendors (such as HP, Juniper, NEC, Pica8 and many others) implement OpenFlow support in their traditional networking equipment. Software-based switches supporting OpenFlow are also heavily used in virtualized environments and as development and prototyping tools (PFAFF et al., 2009). There is also a wealth of controllers, from open source projects to commercial solutions (KREUTZ et al., 2015). The OpenFlow specification is currently managed and maintained by the Open Networking Foundation (Open Networking Foundation, 2015), which was created to oversee the promotion and adoption of SDN via OpenFlow and other open standards.

2.2 OpenFlow controllers

The OpenFlow specification does not dictate how controllers should be built: it only specifies the protocol structures (wire-format) and a few of the behaviors expected from switches and controllers. It is up to control software developers to choose the best design.

The most widespread model is that of a network operating system, in which a central controller runs several applications at the same time in a managed environment, each acting according to their purpose. For example, a controller can run routing and firewall applications at the same time, and each would react to events and act on the network elements. It is up to the controller to orchestrate these applications, providing resources when needed and avoiding conflicts.

There are other variations to this approach, such as distributed controllers (KOPONEN et al., 2010) which reduce the risk and costs of centralizing all network control in a single point. Others abstract the use of OpenFlow entirely and provide the means for deploying multiple southbound protocols (OpenDaylight project, 2015).

In this section we will detail two controllers that have influenced our work due to their implementations of the OpenFlow protocol.

2.2.1 NOX

The NOX controller was the original OpenFlow controller, initially published in 2008. At the time, it provided a vision not only for OpenFlow, but also for SDN in general. The authors proposed an approach to networking that resembles that of operating systems:

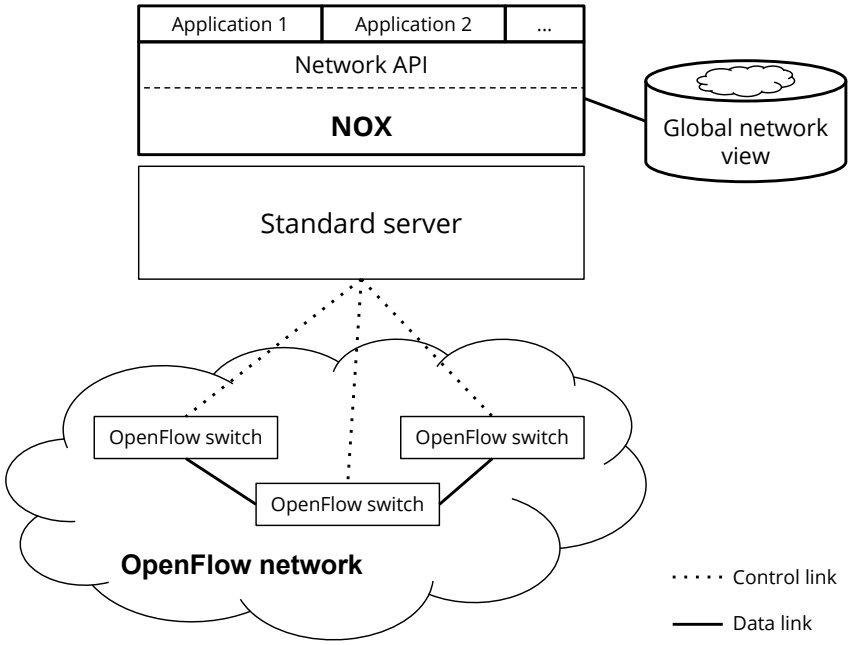


Figure 4 – The NOX network model.

a layer of software that observes and controls hardware, providing an API for applications that run on top of it and use the hardware as means of achieving their purposes.

NOX tries to abstract switches in the network by providing a common API, chosing the OpenFlow switch model to do so. It also features a global view of the network, which is provided to applications in order to aid state monitoring and decision making. By doing this, applications do not have to deal with specific protocols or network devices individually. Figure 4 illustrates this paradigm, which has been highly influential to many SDN projects.

In practice, however, the ideas behind NOX were never fully implemented to the point that applications were truly abstracted from the underlying network. Nowadays, the project seems to be idle, with occasional bug fixes, and without support for newer versions of the OpenFlow protocol. However, its most important contribution is still observable in the design of most current OpenFlow controllers: the use of applications running on top of an API which tries to abstract the network as much as possible.

NOX provided some of the first implementations of OpenFlow libraries for building/parsing messages, built in C/C++ and Python (libopenflow). An evolution of NOX supporting multiple threads (NOX-MT) added multi-threading support to NOX in order to increase its performance and some of its contributions inspired our work as we will detail later (TOOTOONCHIAN et al., 2012). POX, a Python-based controller for quick development was also created based on NOX code.

2.2.2 Beacon

The Beacon project (ERICKSON, 2013) is a Java-based controller that aims to provide a fast and cross-platform basis for applications. It has been in development since 2010 and has been used in experiments and research since then. The goals of the Beacon project can be shortly described as:

1. Developer productivity: Beacon looks for ways to maximize developer productivity via its API and deployment utilities;

2. Runtime modularity: applications running on the controller should be easily controllable and changeable (an application can be replaced by another without affecting running switches);

3. Performance: a multi-threaded architecture that is designed as the central point of the architecture, allowing linear performance scaling.

Java is chosen as the programming language for implementing the controller, since it helps in providing the goals mentioned above: (1) developer productivity is enhanced since it is a high-level programming language, featuring fast compile times and automatic memory management; (2) there are many ways to implement runtime modularity in Java. The Beacon author chose to use the OSGi framework; (3) There are plenty of libraries and ways of optimizing a Java application to a point where it can achieve near-native performance. The results obtained with the Beacon implementation prove this: Beacon was a top-performer in all benchmarks, sometimes by a wide margin.

Beacon's core is composed by a number of IO loops (event loops) running in threads. These threads are responsible for running code that reads from and writes to sockets and dispatches events to application handlers. It is possible to use different event handling approaches with Beacon (such as a synchronized queue, or a run-to-completion approach). When evaluating our work in Chapter 5 we will present some of these approaches and discuss their impact on performance.

2.3 OpenFlow switch agents, frameworks and messaging libraries

In this section we will present software libraries that relate more closely to the objectives of our work: they provide controller frameworks, switch agents and message building/parsing libraries for the OpenFlow protocol.

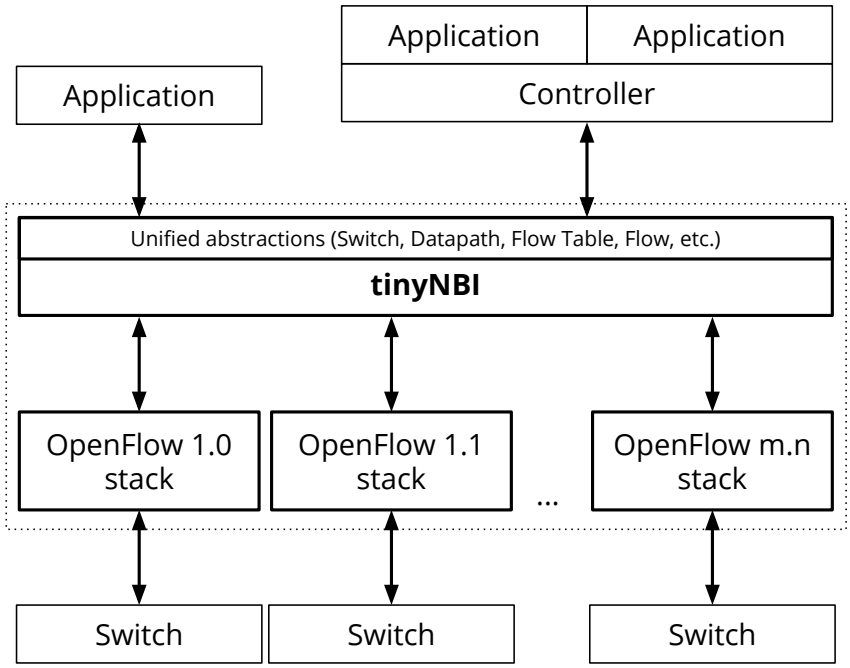


Figure 5 – tinyNBI abstracts differences in OpenFlow protocol versions.

2.3.1 tinyNBI

tinyNBI (CASEY; SUTTON; SPRINTSON, 2014) was designed to overcome the underlying complexity created by the lack of a formal OpenFlow specification. The authors mention an often-discussed issue in the OpenFlow community: there are several versions of the OpenFlow protocol, most of them in use, with a complex model of dependencies between features and a lack of a formal definition for the protocol structures. Adding to this, there is the fact that switches are not mandated to implement all protocol features, so an application developer has to check for all possible variations of protocol versions and device support for these versions and features.

In order to solve these issues, the authors proposed a programming interface that exposes clearly defined SDN abstractions, and deals with the details of implementing them on top of the several existing OpenFlow versions. Applications declare which features they need, and tinyNBI takes care of making the necessary adaptations to application requests. The application is informed in case a feature cannot be supported due to protocol version or hardware constraints. Figure 5 illustrates the tinyNBI approach.

Some examples of abstractions tinyNBI provides are common to all OpenFlow version: Switch, Connection, Datapath, Flow, Match, Meter, Instruction, etc. These abstractions are available with slightly different variations in every OpenFlow version.

An API is developed on top of these abstractions, providing application developers with a high-level view of the protocol, regardless of version or switch support. Event handling can be synchronous, which allows for an application to drive the controller

(asking for events), or asynchronous, which allows the controller to drive an application (informing about events).

2.3.2 Trema

Trema (Trema project, 2014a) is a lightweight framework providing a complete OpenFlow stack: a controller framework, a switch emulator and applications. It was designed to provide an easy way for control applications to be written in Ruby and C, using a common code base for both languages. It supports OpenFlow 1.0 in its main release, and OpenFlow 1.3 support is being added in a new version (Edge) still in development (Trema project, 2014b). Applications are expected to implement several callbacks for common OpenFlow events (switches joining or leaving the network, *packet-in* messages, errors, etc.).

In its current design, it is single-threaded, meaning that applications can only make use of one thread when handling events (the Edge version adds some multithreading capabilities). The bindings for Ruby enables developers to easily build controllers and applications using a high-level, dynamic language. Trema also provides a network and host emulator that can be used to test controllers and applications in a simulated environment.

2.3.3 Indigo

Indigo (Project Floodlight, 2014) is a switch agent framework which aims to enable OpenFlow support on physical and software switches. It provides OpenFlow support upon a hardware abstraction layer (HAL) for switches. By doing that, adding OpenFlow support to a switch is a matter of implementing the OpenFlow features in the hardware using the HAL, since Indigo takes care of establishing the OpenFlow control channel and calling the appropriate HAL functions in order to implement OpenFlow in the hardware.

2.3.4 ROFL

The Revised OpenFlow Libraries set (ROFL, also known as ROFL-core) was developed as a general-purpose set of C/C++ libraries for adding OpenFlow support in software such as control applications, controllers frameworks and switches. ROFL is composed by three sub-libraries:

- ROFL-common: provides an implementation of basic OpenFlow primitives such as control channel connections (in both the client and server sides), event loops and message handling utilities for parsing and building OpenFlow messages. It provides a C++ API and supports OpenFlow versions 1.0, 1.2, 1.3 and 1.4.

- ROFL-pipeline: a platform-independent pipeline that provides OpenFlow message processing and data modeling functionality. It is intended to be used in an OpenFlow switch to take care of the OpenFlow state in the hardware (tracking flow tables, for example).

- ROFL-hal: a hardware abstraction layer for OpenFlow, deploying ROFL-pipeline to integrate it with the hardware. It is intended to provide a unified hardware interface for ROFL-pipeline.

The last two sub-libraries, ROFL-pipeline and ROFL-hal are not of interest to us. They implement features that are mostly hardware-oriented.

ROFL-common is the sub-library that is of interest to us. ROFL-common is built as a set of independent pieces for several purposes (socket/IO manipulation, message and packet building/parsing, an event loop implementation). These pieces can be combined together to form a complete application, either as a controller or as a switch agent.

The term "library" is used for describing ROFL-common. However, following the definitions we proposed at the beginning of this chapter, it would be categorized as both a library and a framework, since it includes several pieces of software that may be useful to OpenFlow controllers and switch agents, and some of these pieces act as a framework when put together.

2.3.5 OpenFlowJ, libopenflow, loxigen and others

Messaging libraries are one of the most modular pieces in controllers and switch agents, being constantly reused in several projects. Some examples are:

1. OpenFlowJ, which started as a Java implementation of the OpenFlow protocol, and is used in several projects, such as FlowVisor, OpenDayLight and Beacon.

2. libopenflow was the library used in the NOX controller (as a C/C++ and Python library) and later improved in the POX controller (as a Python library).

3. Loxigen, which aims to provide automatically generated messaging bindings for several programming languages. It is a more ambitious project than other messaging libraries because it attempts to automatically build a fully functional messaging library for different protocol versions based on the parsing of the OpenFlow specification (the C header defined in it). It is developed by Big Switch Networks, and is used in the Indigo switch agent and in the Floodlight controller.

These libraries are not directly related to the objectives of our work. We will describe an implementation of a messaging library that complements our work in Chapter 4 and exemplify how these libraries can be used in conjunction with our work in Chapter 5.

3 The libfluid framework

This chapter is divided into three main sections. Section 3.1 outlines issues with the related work mentioned in Chapter 2. Section 3.2 provides a list of requirements that solves the issues identified in the previous section. The remainder of the chapter (section 3.3) details a software architecture aimed at implementing the requirements, along with explanations on how this is done.

By the end of the chapter, we will have detailed the core of our proposal: libfluid, a lightweight OpenFlow framework.

The origin of libfluid

The trigger for the development of libfluid was the Open Networking Foundation (ONF) "OpenFlow Driver Competition"[a]. The goal of this competition was to foster the development of open source libraries for OpenFlow, making it easier for third-parties to build software that uses the protocol.

The term "OpenFlow driver" is related to the term "device driver" used in operating systems: a software layer that lies between the applications and the device, abstracting the device and providing a programming interface to the operating system and its applications (CORBET; RUBINI; KROAH-HARTMAN, 2005). In the case of OpenFlow, a driver would be the software that allows a computer program (and thus the computer) to control OpenFlow devices. Additionally, it could also be used in a network switch, enabling its hardware to use the OpenFlow protocol. In this text, we simply call the "driver" a framework.

Our team, then at CPqD, took part in this competition and we eventually won it. Our submission was regarded as the most compliant with the competition requirements.

The framework/driver that is the core of our proposal is part of what was submitted to the competition. The issues we list in this chapter and the proposed solutions are the result of our experiences and observations of related work in the field of SDN and OpenFlow that led to the development of libfluid.

[a] <https://www.opennetworking.org/competition>

3.1 Issues in current work

In this section we present issues in existing SDN frameworks, controllers and switch agents; some of these issues are relevant not only for OpenFlow, but also for SDN in general (i.e.: they may be relevant for SDN protocols other than OpenFlow).

This list of issues will foster the elaboration of a requirements list for the core of our proposal.

3.1.1 Issue #1: Little reuse between switch agents and controller frameworks

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 network model.

Since client and server OpenFlow implementations share some common responsibilities (namely, the connectivity and messaging services described in Chapter 1), it is possible to reuse code more effectively.

Current OpenFlow protocol implementations (with the exception of ROFL) target only one side of this client-server architecture.

3.1.2 Issue #2: Protocol implementations are inflexible

Current implementations of OpenFlow frameworks, controllers and switch agents offer default event handling approaches and other safe-defaults that cannot be easily changed, such as data structures used inside the core of the framework.

These safe-defaults may make it hard to code in a way that reduces development time, or that optimizes implementations for throughput, latency or fairness. It is impossible to provide all these guarantees at the same time, but the architecture of a protocol framework should enable the developer to choose what is important by tweaking parameters or adding incremental layers of software only when necessary.

A related aspect (to be further detailed in Issue #4) is the bundling of a messaging library in the core of existing implementations, meaning that developers will be forced to use a default library, potentially incurring unnecessary overhead.

3.1.3 Issue #3: No lightweight and portable OpenFlow implementation

Most portable implementations of OpenFlow, such as Beacon and Maestro (CAI ALAN L. COX, 2011), are written in Java (and they base their portability claim on this fact). However, they are not lightweight, providing many features that go beyond the scope of an OpenFlow framework and requiring the setup of an environment for the controller. Lightweight implementations (such as Trema, ROFL-common and Indigo) are restricted to one or two of programming languages and architectures.

Developers could benefit from a protocol implementation that works across programming languages and computer platforms. This means that there could be less restrictions and compromises when choosing technologies for applications.

3.1.4 Issue #4: Protocol implementation core and message handling are mixed

Current implementations of the OpenFlow protocol use libraries that provide protocol message building/parsing in their core. This means that, for every new version of the protocol, the core framework has to be adapted even though there are ways to leave most (if not all) of this task to a dedicated message building/parsing implementation. In other words, the cores of existing OpenFlow controllers, switch agents and frameworks are bigger than they should be.

3.1.5 Issue #5: No clear path for building standalone applications

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. ROFL-common and Trema allow for this, but not in an easy or straightforward manner (and they do not advertise it as an use case).

Researchers seeking to build network control applications to test new ideas, or students wishing to dive straight into the protocol need first to learn how to compile, run and develop for existing controllers. The application will always need the chosen controller in order to run, incurring unnecessary overhead in simple use cases.

Standalone applications may also prove to be interesting when hardware is limited or requirements are stringent. Developing a single, standalone application that performs a few functions without the overhead of a full controller should allow for extreme tweaking of performance metrics and fine-grained control over memory and CPU usage.

3.1.6 Issue #6: Protocol implementation behavior is not configurable

OpenFlow controllers, switch agents and frameworks do not provide easy ways for fine-tuning protocol implementation details. As an example, it is not always trivial to specify the protocol versions the controller, switch agent or framework will use if it supports more than one, and some behaviors cannot be overridden by the user (such as liveness checks and handshake behavior). While this is fine for most use cases, sometimes users run into incompatibilities between controllers and switches[1], and there is no simple way to override the faulty behavior other than directly changing the code responsible for this behavior.

It would be desirable to have an easy way to enable, disable and customize protocol features to allow users to command the behavior of the implementation, working around

[1] As an example, a controller may expect a switch to send a keep-alive message every 15 seconds, but the switch only sends one every 30 seconds, causing the connection to be closed by the controller.

incompatibilities or fine-tuning settings that can improve performance.

3.2 Requirements

These requirements are directly derived from the issues identified in the previous section. This is not an extremely detailed requirement listing, but rather a general guideline for the design of our proposal.

3.2.1 Req. #1: Unified protocol implementation for controllers and switches

In the context of SDN, a unified protocol implementation is one which can be used to add protocol support in all parts of the network in which it is needed. Given proper software abstractions, a single solution can be built which works for devices of all types, be they in control or being controlled. In order to achieve this, it is necessary to extract common functionality, so that code can be reused for different purposes.

We focus on the OpenFlow network model and see how a unified protocol layer can be implemented so that a single code base can serve for both servers (controllers) and clients (switches). This issue should tackled from a formal software-engineering point of view: decomposing the architecture in multiple levels and then implementing it using modularization techniques and extracting reusable pieces when possible.
This requirement addresses Issue #1: Little reuse between switch agents and controller frameworks.

3.2.2 Req. #2: More flexibility in the core of the protocol implementation

The task of providing advanced OpenFlow/SDN functionality will be left to the user. This means that using a single implementation, different requirements for latency, throughput or fairness can be achieved by customizing a common core. We will also explore how different performance or architectural requirements can be addressed by tweaking configuration parameters (such as threading configuration and memory allocation) and choosing different data structures (that are implemented at a higher-level and not as part of the framework core).

By avoiding default and unchangeable features (such as event handling methods and message building/parsing libraries) our implementation should be adaptable for different purposes (such as specific tools or SDN controller plugins), since we will only be providing a lean core that can be customized for different purposes.
This requirement addresses Issue #2: Protocol implementations are inflexible.

3.2.3 Req. #3: A lightweight and portable implementation

In order to keep the implementation lightweight, the public API should be as small as possible, helping keep our implementation lightweight (as per our definition) and minimalistic. By doing this, we will also make it easier to create bindings for other programming languages, since there are less public-facing entities to port and adapt.

The implementation should be written using technologies widely available in different computer platforms (we chose using C/C++ and restricting ourselves to POSIX as much as possible). The implementation must also use proper methods for dealing with endianness and avoid using compiler-specific features. These restrictions are known to make code easier to port to other computer platforms.
This requirement addresses Issue #3: No lightweight and portable OpenFlow implementation.

3.2.4 Req. #4: Independence from messaging libraries and protocol versions

We have chosen a viable method for dealing with the issue of protocol implementations working with only a handful of protocol versions. The OpenFlow protocol specification allows us to make a few basic assumptions that can be used to implement version independence – that is, there are some protocol characteristics that are guaranteed to never change.

Therefore, if version X.Y of OpenFlow protocol specification is released in the future, the underlying implementation should not have to change, since it supports the core features of every OpenFlow version (even future ones), as long as basic protocol characteristics (message header structure) are unchanged, as guaranteed by the OpenFlow specification.

By using these unchanging features to implement version negotiation, the only requirement for implementing support for new protocol versions becomes the messaging library. This library has to be built or adapted so that it can work with the constructs of the new protocol version and advanced features can be used.
This requirement addresses Issue #4: Protocol implementation core and message handling are mixed.

3.2.5 Req. #5: Enable standalone applications

It must be possible to compile the code as a shared or static library[2], so that it can be embedded in applications in order to provide OpenFlow protocol support. This

[2] The implementation is still a framework, but one which is packaged in a software library.

also 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.

Enabling the implementation to be used in standalone applications paves the way for building embeddable applications in limited hardware or simply ad-hoc applications, which can be simpler to deploy and reason about. Examples of these applications can be network traffic simulation, firewalls and router applications deploying custom routing engines.
This requirement addresses Issue #5: No clear path for building standalone applications.

3.2.6 Req. #6: Configurable protocol options

There must be a way to configure and choose protocol implementation details, such as handshake behavior, memory management policies and the use of potentially troublesome features.

This configuration method must also be easy to extend by adding new configuration parameters. These parameters will be detailed when we discuss the architecture further in this text, since some of them are very specific to implementation details.
This requirement addresses Issue #6: Protocol implementation behavior is not configurable.

3.3 Software architecture

In this text (and during the development of libfluid), we adopted a leveled design breakdown as proposed by Steve McConnell (MCCONNELL, 2004) in his book, Code Complete:

1. At first, the system is pictured as a whole;
2. The system is broken down into subsystems or packages;
3. Each subsystem is divided into classes;
4. Each class is divided into data and routines;
5. Finally, each routine is detailed.

For the first level, we have libfluid as an OpenFlow framework; an overview of it is present in Subsection 3.3.1. We chose to develop a framework because it was the simplest way to have a lean implementation that works for both controllers and switch agents.

The second level is shown in Subsection 3.3.2: we chose to rename subsystems to **blocks**. McConnell states that in small systems, design can jump from the first to the

third level. In the case of libfluid, we think that it is beneficial to show the blocks and outline their interactions in broad terms in order to aid comprehension.

An overview of the third level is also shown in Subsection 3.3.2, as **modules** which compose the blocks. Modules are roughly equivalent to classes, but we chose to call them differently because they may also feature additional parts.

Finally, the implementation details of the third, fourth and fifth levels are shown in Chapter 4.

3.3.1 Overview

In order to have a unified architecture for both controllers and switch agents, we need to extract common functionality available in both of them and expose it in a reusable form.

We chose to apply the client-server paradigm to our frameworks's architecture, with servers being OpenFlow controllers (which provide a control service to switches), and switches acting as clients (connecting to controllers and asking for the control service). This is how OpenFlow controllers and switches have always been designed, but this terminology is not the usual one. In our text, we use both interchangeably: OpenFlow controllers are OpenFlow servers, and OpenFlow switches are OpenFlow clients.

In common, both clients and servers need to manage the lifecycle of one or more connections and read and write data to the network using these connections. This lead us to a three-tier architecture[3] for separating responsibilities:

0. Fundamental IO and event handling code that can be reused for both controllers and switch agents, providing only the fundamental pieces for OpenFlow support.

1. Customizable OpenFlow clients and servers built using the abstractions defined in tier 0. Adds support for OpenFlow features.

2. User-built code, building fully-functional software on top of libfluid.

This conceptual view of the architecture is illustrated by the diagram in Figure 6, which contains several blocks outlining functional groupings in the framework.

In Figure 7 we further break down the **blocks** of Figure 6, exposing the **modules** in each block. A module is a set of one or more classes, associated functions and constants, providing some specific functionality in the framework. In practice, these modules are

[3] The traditional definition of a three-tier architecture is that divided in data, logic and presentation tiers. This is not the meaning we intended here, even though we used the same term: ours is simply an architecture divided in 3 parts.

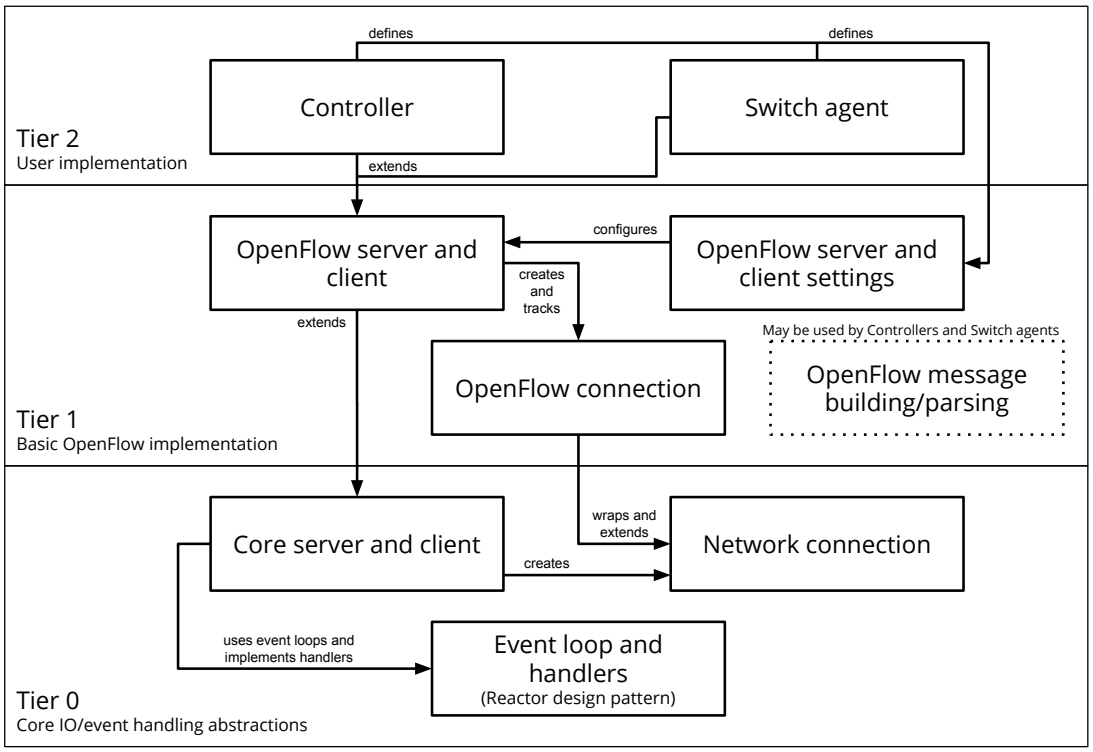


Figure 6 – Conceptual view of the libfluid architecture showing architectural **blocks**.

typically mapped to a class, following the relationship patterns outlined in Figure 7. The *is-a* relationship is represented by inheritance, and the *uses-a* or *has-a* relationships are represented by association (and its specializations: aggregation and composition).

However, we will not get into this level of detail now. In Chapter 4 we show the implementation of this architecture, correlating it with the the modules and relationships illustrated in Figure 7.

3.3.2 Blocks and modules

Each part of this subsection represents a **block** in the conceptual view illustrated in Figure 6. Inside each part, we further explain the role of each block in the architecture and break them down in **modules**. The attributes and behaviors of each module are also detailed in each part.

3.3.2.1 Event loop and handlers

There is no formal definition of what is an event loop. It is typically understood to be a programming construct that waits for events to happen and then dispatches these events for processing. The application defines which events are to be dispatched, and then defines the behavior that happens in response to these events.

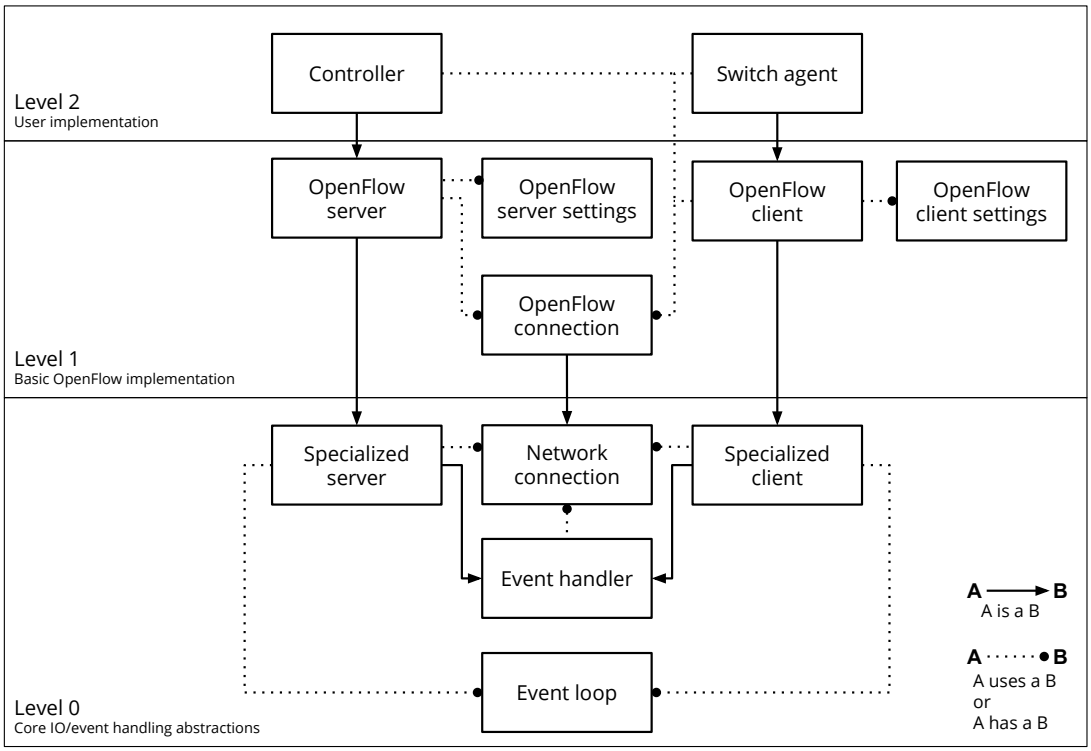


Figure 7 – Module relationship diagram of the libfluid architecture.

As an example, consider a web server. It has to listen to HTTP requests from users. Upon the establishment of a TCP connection (via a socket), it needs to read the HTTP request data and interpret it. Based on the request, it needs to dispatch it to the correct (web) application for a response to be served. In this scenario, an event loop is part of the core web server code: it detects that data is ready to be read from the TCP connection, then it reads the data, and notifies the application responsible for that request. The application processes the request and gives back a response, which is queued for sending in the event loop when the socket is ready to have data written to it. There are several minor variations possible in this design (application event handling itself can be queued, for instance), but this is the overall model of an event loop.

A network-oriented application typically connects to several peers, receiving information from them in a non-deterministic fashion (it is impossible to know exactly when or how much data will be transmitted). Compounding this issue, IO is usually a blocking operation in most operating systems.

One way of dealing with these issues is to use threads dedicated for every connection: the threads block waiting for data to be received or transmitted, and follow the normal application workflow (event handling) whenever IO is completed. There are drawbacks in this approach: threading may lead to poor performance (due to synchronization and context switching) and concurrency control can become complex. A high number of threads is undesirable, since it can worsen these issues and end up creating a bottleneck.

An alternative to threads is using a single-threaded event loop that uses asynchronous or non-blocking operating system primitives to manage IO events. These primitives are known as OS event demultiplexers: in the case of network application they receive a list of sockets and block until an IO event happens (e.g.: data received). In UNIX and UNIX-like systems there are several implementations of event demultiplexers with different fine-tunings of behavior: `select`, `poll`, `epoll`, `kqueue`, etc. In Windows systems, which uses asynchronous IO (a callback is called when an event happens, much like an UNIX signal), blocking event demultiplexing can be emulated with the function `WaitForMultipleObjects`.

A common way of implementing an event loop to deal with IO in an object-oriented design is by using the Reactor pattern (SCHMIDT, 1995). It is a pattern commonly identified in network-oriented applications, and it describes a way to implement the relationship among:

- Event sources (such as sockets in network programming);
- The events, that can be IO-related (such as data ready to be read or written) or management-related, such as a connection establishment or disconnection events;
- Data handling functions (the event handlers).

In the Reactor design pattern, a Reactor component makes the call to the OS event demultiplexer based on a list of handles (connections, files, etc.) and then dispatches the events to an abstract event handler. The application implements a concrete event handler based on the abstract event handler API, providing the actual functionality.

An application that implements the Reactor pattern benefits from improvements to its modularity (IO code is separated from the event handling), reusability (IO-related code can be reused in other applications) and portability (IO functions and semantics, which almost always have to be adapted for different operating systems, can be easily changed). However, care must be taken in order not to perform blocking operations during event handling, since this may impact other connections allocated to the same event loop. Debugging also becomes more challenging, since flow of control is constantly changing among functions that perform IO, OS event demultiplexing and event handling.

Frameworks typically implement patterns (JOHNSON, 1997), and the core pattern of libfluid is the Reactor pattern. This pattern is used in the design of the event loop which is implemented by the `Event loop` module. The `Event loop` module is responsible for dispatching events to handlers, represented by the `Event handler` module. It is possible to run more than one event loop simultaneously, each in its own thread.

3.3.2.2 Network connection

A network connection is a connection established for sending and receiving data via a network. In traditional OS terminology, it is roughly the same as a socket. In libfluid, a network connection wraps a socket, providing functionality for reading and writing data from/to the network. It also wraps a socket life cycle management (open/close/error).

In the libfluid architecture, when an `Event loop` notices that it is possible to read data from the network, code present in the `Network connection` module performs the actual reading of the data from a buffer.

The `Network connection` module is not intended to be reusable in non-OpenFlow scenarios: when reading messages, it is necessary to know their length, so basic knowledge of the protocol OpenFlow protocol header is necessary (Open Networking Foundation, 2014b).

3.3.2.3 Core server and client

The core server and client block represents modules providing a minimal set of client/server functionalities for other modules that build on top of them. These functionalities are:

- Listening for network connections (server);
- Establishing a network connection (client);
- Starting event loops to handle network connection events (both client and server);
- Implementing code for handling events and forwarding them to upper layers (both client and server).

In the libfluid architecture, the `Specialized server` and `Specialized client` modules implement these functionalities. They are said to be specialized because they use the core modules (tier 0) of the libfluid architecture, assuming a default set of behaviors and interfaces is available.

A `Specialized server` is responsible for starting any number of event loops, using threads to provide parallelism (an event loop blocks the calling thread waiting for events). It also listens to incoming network connections, assigning them to an event loop.

A `Specialized client` is responsible for attempting a connection to a remote server and starting and stopping the event loop.

3.3.2.4 OpenFlow server and client

The OpenFlow server and client block represents modules providing minimal OpenFlow support in order to establish and keep an OpenFlow connection according to the protocol specification (Open Networking Foundation, 2014b). Among the functionalities of these modules, we have:

- Performing protocol handshake and version negotiation (both server and client);
- Checking connection status using features defined in the protocol (both server and client);
- Implementing code for handling events and forwarding them to upper layers (both server and client);
- Keeping a list of network connections (both server and client);
- Exposing configurable parameters (both server and client). These parameters are detailed in the next part of this subsection.

The OpenFlow handshake

The OpenFlow handshake is a process that controllers and switches go through when they establish a control channel connection. It is performed in order to negotiate an OpenFlow protocol version to be used in communication (since peers may support more than one version). During the handshake, the switch also notifies the controllers of its features and capabilities.

In libfluid, we are only interested in providing automated version negotiation, and the treatment of switch capabilities is left to the user.

In the libfluid architecture, the `OpenFlow server` and `OpenFlow client` modules implement the previously listed functionalities. These are the first modules that actually implement OpenFlow behavior, and they expose an interface that is similar to the interfaces exposed by the `Specialized server` and `Specialized client` modules (as will be detailed in Chapter 4).

User applications (controllers and switch agents) will implement a *is-a* relationship with the `OpenFlow server` and `OpenFlow client` modules, providing code for handling events in both server and client contexts as necessary.

3.3.2.5 OpenFlow client and server settings

The OpenFlow client and server settings block represents modules providing configuration parameters to the `OpenFlow server` and `OpenFlow client` modules. These

modules provide ways for reading and writing configurations parameters in categories such as:

- Supported versions: a list of protocol versions the framework will accept when negotiating with peers (controllers or switches);
- Handshake behavior: enable or disable automatic handshake by the framework;
- Liveness check behavior: enable or disable automatic liveness checking by the framework, and define the timeout for these checks;
- Message filters: allow framework users to define which messages they are interested in;
- Memory management policies: allow framework users to optimize memory management and avoid unnecessary memory copies;
- Potentially troublesome protocol details: enable or disable features that can cause incompatibility with peers (e.g.: OpenFlow 1.3.1 hello elements).

In the libfluid architecture, the `OpenFlow server settings` and `OpenFlow client settings` modules implement these functionalities. These modules do not feature any actual code to perform the behaviors they define. `OpenFlow server` and `OpenFlow client` receive an object representing the configuration (via dependency injection (FOWLER, 2004)) and then read the configuration parameters in order to define their own behavior.

3.3.2.6 OpenFlow connection

The OpenFlow connection block represents modules providing OpenFlow-specific connection semantics, and is used by the `OpenFlow server` and `OpenFlow client` modules. It wraps a `Network connection` object, representing read and write operations in its behalf. It also provides ways for storing and retrieving the following information:

- Negotiated version (for both servers and clients);
- Connection state (at the OpenFlow level, for both servers and clients);
- Application-defined data related to the connection.

In the libfluid architecture, the `OpenFlow connection` module implements these functionalities. This module is exposed to user applications (controllers and switch agents) as means for them to interact with their respective OpenFlow peers (switches in case of controllers, and controllers in the case of switches).

3.3.2.7 OpenFlow message building/parsing

The libfluid framework does not define a message building/parsing block in its architecture. The function of this library is to provide developers with a high-level representation of OpenFlow messages (usually object-oriented). Messages coming from the network are parsed and converted into this high-level representation. When writing messages, developers can build them using the same high-level abstractions and convert them into binary format, ready to be sent through the OpenFlow control channel to a peer.

Since libfluid aims to provide independence regarding OpenFlow version, it is counterproductive to embed a message building/parsing library into the core of the framework. Doing this would mean that for every new OpenFlow protocol version, this library would have to be first adapted, and only then the framework would be able to use it. Additionally, the user would have the overhead of an additional library, which is not always desired, especially if it is not being used outside the framework's core (as would happen if the user favored another messaging library).

In Figure 6 the OpenFlow message building/parsing block is depicted in a special way to denote this choice. It means that tools for handling protocol messages are not part of the libfluid architecture, even though any implementation of these libraries will fit into the framework.

This is an important design decision that defines much of what is unique to libfluid when compared to other OpenFlow frameworks. A negative aspect of this decision is that we are offloading complexity to the user. On the other hand, users are then free to choose any messaging library, without being restricted to a default that may not address their needs.

In light of this, an implementation of a messaging library called `libfluid_msg` is available, and it can be easily used with the libfluid framework. It was developed alongside the libfluid framework, but it is not implemented by the author of this dissertation. It will be described in further detail in Chapter 4.

3.3.3 Requirements vs. Architecture

Table 1 shows the requirements described in Section 3.2 and how they are implemented in the libfluid architecture. The table describes the architectural blocks (listed in Subsection 3.3.2) which are most related to how these requirements are answered, along with an explanation.

Requirement	Architectural blocks	Explanation
Req. #1: Unified protocol implementation for controllers and switches	OpenFlow server and client	The `OpenFlow server` and `OpenFlow client` modules enable users to build both controllers and switch agents, while sharing common ancestor modules in the architecture.
Req. #2: More flexibility in the core of the protocol implementation	Event loop and handlers, OpenFlow server and client settings, OpenFlow message building/parsing	By building a multi-threaded event-loop design and allowing customizations to default behavior of minor implementation details such as memory allocation, it is possible to tweak libfluid for different use cases. Additionally, not requiring a default message building/parsing library also enables further flexibility when using libfluid.
Req. #3: A lightweight and portable implementation	-	Answering this requirement involves choices made when actually writing the code, so it is not directly influenced by the architecture. However, libfluid's simplified architecture helps when creating bindings for other programming languages, since it features a small number of modules that are exposed to users.
Req. #4: Independence from messaging libraries and protocol versions	OpenFlow server and client, OpenFlow message building/parsing	By implementing only the minimum necessary to enable a functional OpenFlow connection and delegating message building/parsing functions to a third-party module (an external message building/parsing library), libfluid becomes version-independent.
Req. #5: Enable standalone applications	OpenFlow server and client	By extending just the `OpenFlow server` module, it is possible to build fully-functional OpenFlow applications that run on their own, without the need for full controllers.
Req. #6: Configurable protocol options	OpenFlow server and client settings	The settings modules provide an easy way to customize the `OpenFlow server` and `OpenFlow client` modules, allowing users to redefine or override default behaviors.

Table 1 – Requirements mapped to the libfluid architectural blocks.

4 Implementation

In Section 4.1 we will detail each component of the libfluid implementation and some related components that may be used along with it. Then in Section 4.2 we will briefly show how libfluid can be used by controllers, applications and switch agents.

The implementation described in this dissertation is available in the form of the `libfluid_base` library. It is written in C/C++, with approximately 2200 lines of code, and was implemented by the author of this text. We use the name libfluid to refer to `libfluid_base` throughout the work.

The `libfluid_msg` library is an optional companion library that can be used with `libfluid_base` for message building/parsing. It was implemented by Eder Leão Fernandes, and is not the subject of this dissertation (though there is a section dedicated to it in this chapter).

In addition to these libraries, sample applications (detailed in Chapter 5) and additional documentation is provided in the `libfluid` repository, that is a home for both libraries (libfluid project, 2014).

4.1 Components

The following subsections describe different components of the libfluid implementation, which are implemented as classes, functions and libraries in code. These components are mapped to modules in the libfluid architecture (which were introduced in Section 3.3.2).

4.1.1 EventLoop

Implementation to architectural module mapping

`EventLoop` implements the `Event Loop` module of the libfluid architecture.

The actual implementation of the Reactor pattern (as mentioned in Subsection 3.3.2.1) is not done in libfluid. Since this is a widely known and used design pattern, there are several libraries implementing it, such as libevent (MATHEWSON, 2012), libev (LEHMANN, 2015) and libuv (libuv project, 2015). These libraries provide a high degree of portability (to different operating systems and architectures). We chose to use libevent, mostly because it was well-documented and actively maintained.

The `EventLoop` class in libfluid provides a simplified wrapper for the Reactor-based event loop implemented by libevent, with two main methods: `run` and `stop`. Running an event loop is a blocking operation (an operation which suspends the caller until execution is done), and all program execution will happen inside event loops. Several event loops can be run in separate threads, improving resource utilization in modern computers. The use of event loops with threads will be detailed further ahead, when we talk about the `BaseOFServer` class.

The use of the Reactor design pattern for an event loop is by no means a unique contribution of libfluid: several controllers and other OpenFlow libraries make use of it. The main difference here is that libfluid delegates this implementation to a third-party library instead of implementing the event loop algorithm and the Reactor pattern itself. Only the Beacon (ERICKSON, 2013) controller features a similar approach, using the Netty framework.

4.1.2 BaseOFHandler

Implementation to architectural module mapping

`BaseOFHandler` implements the `EventHandler` module of the libfluid architecture.

The `BaseOFHandler` abstract class defines a simple interface that should be implemented by any classes providing basic OpenFlow event handling: connection and message events. It defines two important methods:

- `base_connection_callback`: called when an event happens to a connection;
- `base_message_callback`: called when a message is received.

In libfluid, classes implementing the `BaseOFHandler` interface are expected to provide general callbacks for message and connection events, without making assumptions about protocol-specific (OpenFlow) features. `BaseOFHandler` allows both controllers and switch agents to share a common interface for their basic connection and message event handling.

4.1.3 BaseOFConnection

Implementation to architectural module mapping

`BaseOFConnection` implements the `Network connection` module of the libfluid architecture.

The `BaseOFConnection` class implements the basic mechanisms for connection state tracking and IO (reading/writing). A `BaseOFConnection` instance is constructed as in Listing 1.

```
BaseOFConnection(int id,
                 BaseOFHandler* ofhandler,
                 EventLoop* evloop,
                 int fd,
                 bool secure);
```

Listing 1 – `BaseOFConnection` constructor.

`id` is an externally defined ID for the connection (i.e.: a connection manager defines it upon connection establishment). `ofhandler` is an object that implements the `BaseOFHandler` interface, providing event handling callbacks. `evloop` is the `EventLoop` instance that will be tied to this `BaseOFConnection` instance, being responsible for waiting for events and triggering callbacks. `fd` is a file descriptor, which will be the socket used for the connection. `secure` is a boolean value indicating whether the connection should use transport-layer security.

`BaseOFConnection` is responsible for calling the message callback (implemented by a `BaseOFHandler` instance) whenever a message arrives. It also defines defines three events which are passed to the connection callback of a `BaseOFHandler` instance:

- `EVENT_UP`: fired when the connection is successfully established;
- `EVENT_DOWN`: fired when the connection has been closed by the other end (the peer);
- `EVENT_CLOSED`: fired when the connection has been effectively closed and its resources freed.

There is an important distinction to be made between `EVENT_DOWN` and `EVENT_CLOSED`. `EVENT_DOWN` will only be fired when the other end terminated the connection; after this happens, an `EVENT_CLOSED` is also fired when the connection is done processing its remaining data and freeing its resources. On the other hand, when the user asks for a `BaseOFConnection` to be closed, `EVENT_DOWN` will not be fired, and only `EVENT_CLOSED` will be fired as a result.

A `BaseOFConnection` instance also provides the methods for:

- Sending data to the other end of the connection;
- Scheduling a callback to be invoked in regular intervals (the callback will be invoked as long as the connection is open);

- Closing the connection;
- Freeing buffers allocated by the connection when reading messages.

`BaseOFConnection` does not implement any OpenFlow-specific behavior, just the handling of basic IO and events (the first treatment of events before forwarding to other layers). However, it is able to process OpenFlow message headers (which are guaranteed by the specification to never change), and it needs to do so in order to figure out the length of the messages being read. Such a simplistic design is used because it allows third-parties to easily override the desired OpenFlow-specific support (such as handshake behavior, connection state management) at an upper level, since they are not embedded in the core of the connection abstraction.

A `BaseOFConnection` can be wrapped by a manager object, which will be responsible for providing additional features for network connections. In libfluid, this manager is implemented by the `OFConnection` class, which defines further OpenFlow-specific semantics on top of this connection.

4.1.4 BaseOFServer

Implementation to architectural module mapping

`BaseOFServer` implements the `Specialized server` module of the libfluid architecture.

`BaseOFServer` implements the `BaseOFHandler` interface and acts as a server, listening for incoming connections. It is intended to be used as the basis for OpenFlow controllers, but without implementing any OpenFlow semantics. A `BaseOFServer` instance is constructed as in Listing 2.

```
BaseOFServer(const char* address,
             const int port,
             const int nevloops = 1,
             const bool secure = false);
```

Listing 2 – `BaseOFServer` constructor.

`address` and `port` are respectively: a null-terminated string describing an address and a port number. Both are used for binding a socket which will listen to connections. `nevloops` is the number of event loops that will be run when the server starts: one event loop per thread, and the threads are started and managed automatically by the `BaseOFServer` instance. `secure` is a boolean value indicating whether connections established in this server will be secured by TLS.

In addition to implementing the methods required by `BaseOFHandler`, `BaseOFServer` also provides two other methods:

- `start`: starts the `BaseOFServer` instance, listening at the address and port declared in the constructor and assigning connections to event loops running in threads;
- `stop`: stops listening to new connections and signal the event loops to stop running (causing connections to be closed).

`BaseOFServer` provides a multi-threaded internal design. When a `BaseOFServer` instance is created, one or more `EventLoop` instances can be created. Since `EventLoop` instances block the threads they are running on, several threads are needed in order to provide parallelism. `BaseOFServer` takes care of automatically creating these threads when the `start` method is called. The threads spawned by a `BaseOFServer` instance are managed internally, eliminating the need for manual thread management by users of libfluid (users are still responsible for providing concurrency control).

The connections in a `BaseOFServer` instance are represented by `BaseOFConnection` instances and the `EventLoop` instance running in the first thread is used for listening for new connections. This means that the first thread will have an additional overhead, but it should be insignificant unless connections are constantly being started and closed, which is not normal in OpenFlow environments, that typically feature long-running connections as the norm.

Connections are assigned to an event loop when they are established, and all events happening in this connection will be tracked by the event loop of that thread. This means that several threads will be calling the callbacks upon events (as defined in `BaseOFHandler`).

A `BaseOFServer` instance does not provide any OpenFlow-specific behavior such as connection liveness checking or version negotiation. Connections are simply established at this point in the architecture, and classes wishing to provide a functional OpenFlow controllers must inherit from `BaseOFServer`. Connection tracking management is not provided either: a `BaseOFServer` instance does not keep track of the connections it is currently managing. Implementations building on top of `BaseOFServer` must use the connection event handler to detect and keep information on `BaseOFConnection` instances.

4.1.5 BaseOFClient

Implementation to architectural module mapping

`BaseOFClient` implements the `Specialized client` module of the libfluid architecture.

The current implementation is not yet integrated with the core of the libfluid framework (`libfluid_base`) because we judged it still does not meet the quality we expected in terms of modularity, testing and documentation.

`BaseOFClient` is the other part of the OpenFlow paradigm: like `BaseOFServer` it also implements the `BaseOFHandler` interface, but with the purpose of building switch agents instead of controllers. A `BaseOFClient` instance is constructed as in Listing 3.

```
BaseOFClient(int id,
             char* address,
             int port);
```

Listing 3 – `BaseOFClient` constructor.

`id` is an ID for the connection, akin to the `id` defined in `BaseOFConnection`. `address` and `port` are respectively the address and port of an OpenFlow controller to which this client will connect.

In addition to implementing the methods required by `BaseOFHandler`, `BaseOFClient` also provides two other methods:

- `start`: starts the `BaseOFClient` instance by attempting to connect to a controller listening at the address and port declared in the constructor. Also starts an `EventLoop` which will notify of message and connection events.

- `stop`: closes the connection and stops the `EventLoop` instance.

When attempting to start a connection with a controller, a `BaseOFClient` instance will keep trying every 5 seconds, until it is finally established. If the connection is interrupted, no attempts to reconnect will be made.

A `BaseOFClient` instance does not provide any OpenFlow-specific behavior such as connection liveness checking or version negotiation. A connection to a controller is simply established at this point, and classes wishing to provide a functional OpenFlow switch agent must inherit from `BaseOFClient`.

Interactions between base components

Upon connection establishment, the `BaseOFServer` instance creates a `BaseOFConnection` object which will be responsible for managing the file descriptor associated with the connection (socket).

The `BaseOFConnection` instance is also tied to an `EventLoop` instance (the `BaseOFServer` instances performs the mapping), meaning that this event loop will be responsible for notifying the `BaseOFConnection` instance when events of interest happen (data to be read, socket read for writing, peer closed the connection, etc.).

Code in `BaseOFConnection` will provide the initial treatment for connection events, and forward an interpretation of these events to a `BaseOFHandler` instance.

If a `BaseOFClient` were used, it would act just as a `BaseOFServer` instance, but attempting to establish a connection rather than listening for connections.

4.1.6 OFConnection

Implementation to architectural module mapping

`OFConnection` implements the `OpenFlow connection` module of the libfluid architecture.

`OFConnection` wraps a `BaseOFConnection` instance and enables some OpenFlow-specific attributes and methods, such as:

- **Liveness checking**: a flag that indicates whether the connections is alive or not, according to the OpenFlow echo mechanism. The liveness checking is performed at regular intervals. The `OFServer` class is responsible for automatically sending and receiving OpenFlow echo messages and toggling this flag accordingly.
- **Version negotiation information**: a value indicating which OpenFlow version was negotiated during handshake.
- **Timed callbacks**: registers a function to be called at regular intervals.
- **Connection ID**: a unique connection ID automatically assigned by `OFServer`.
- **Application data**: a pointer to application-specific data, which allows users of libfluid to store any information in `OFConnection` instances. This is useful because it allows persistent data to be stored and be available across callback invocations. A typical use case for this is keeping track of session data (e.g.: a L2 table in a learning switch application) or pointing to a higher-level construct (e.g: a user-defined class representing switch objects which use a connection).
- **Wrappers**: simple wrappers for `BaseOFConnection` methods for closing connections and sending data.

A `OFConnection` instance is defined as in Listing 4.

```
OFConnection(BaseOFConnection* bofconn,
             OFHandler* ofhandler);
```

Listing 4 – `OFConnection` constructor.

`bofconn` is the `BaseOFConnection` instance being wrapped by this `OFConnection`. `ofhandler` is an instance of the `OFHandler` class, which is analogous to the `BaseOFHandler` class, but represents an event handler that is at a higher level in the architecture, implementing OpenFlow-specific behavior, which in our case is an `OFServer` instance (it is not listed in the section for the sake of brevity).

4.1.7 OFServer

Implementation to architectural module mapping

`OFServer` implements the `OpenFlow server` module of the libfluid architecture.

`OFServer` extends the minimal set of functions in `BaseOFServer` and adds important features for OpenFlow support:

- **Connection tracking**: `OFServer` keeps a list of connections (represented by `OFConnection` instances), allowing users to retrieve them by their IDs (and not only when a callback is called).
- **OpenFlow version negotiation**: `OFServer` implements an algorithm for version negotiation that is flexible enough for current and future OpenFlow versions. The algorithm does all that is needed in order to perform the OpenFlow handshake and version negotiation as defined by the protocol specification (sending messages and processing incoming responses).
- **Liveness checking**: checking the liveness of a connection is done by periodically sending OpenFlow echo request messages and waiting for a response from peers (this is a feature defined by the protocol specification). `OFServer` does this by instructing connections managed by it to periodically send an echo request, and then waiting for the response: if the response arrives before another echo request is bound to be made, the connection is considered alive. Otherwise, it is marked as stale and closed.
- **Customizable behavior**: it is possible to disable or tweak most of the behaviors implemented in `OFServer`, including the ones mentioned above. This is done by configuring it via a `OFServerSettings` object.

An `OFServer` instance is constructed as in Listing 5.

```
OFServer(const char* address,
         const int port,
         const int nthreads = 4,
         const bool secure = false,
         const struct OFServerSettings ofsc = OFServerSettings());
```

Listing 5 – `OFServer` constructor.

`address`, `port`, `nevloops` and `secure` are the same as in the `BaseOFServer` class. The only addition is the `ofsc` parameter, which specifies the `OFServerSettings` object used to define the behavior of the `OFServer` instance. If no value is provided, a default set of configuration parameters is used.

`OFServer` allows the user to extend its default behavior via two methods (which come from the `OFHandler` abstract class from which it inherits):

- `message_callback`: called when an OpenFlow message is received;
- `connection_callback`: called when an OpenFlow-relevant event happens to a connection.

By implementing these two methods, all the scope of possible events is covered, since events either relate to the control connection itself, or to OpenFlow messages being received.

4.1.8 OFClient

Implementation to architectural module mapping

`OFClient` implements the `OpenFlow client` module of the libfluid architecture.

The current implementation is not yet integrated with the core of the libfluid framework (`libfluid_base`) because we judged it still does not meet the quality we expected in terms of modularity, testing and documentation.

The `OFClient` class is analogous to the `OFServer` class, and it works by extending the minimal set of functions in `BaseOFClient` and adding important features for OpenFlow support:

- **Connection attempts**: `OFClient` tries to connect to a controller, and keeps trying indefinitely until a connection is established.

- **OpenFlow version negotiation**: `OFClient` also implements the algorithm for version negotiation available in `OFServer`, adapting its behavior for an OpenFlow client connection.
- **Liveness checking**: checking the liveness of a connection is done in the same way as in the `OFServer` class.
- **Customizable behavior**: while it is not currently implemented, it will be possible to configure `OFClient` instance behavior with a `OFClientSettings` object[1].

An `OFClient` instance is constructed as in Listing 6.

```
OFClient(const int id,
         const char* address,
         const int port,
         uint64_t datapath_id,
         const struct OFClientSettings ofcs = OFClientSettings())
```

Listing 6 – `OFClient` constructor.

`id`, `address` and `port` are the same as in the `BaseOFClient` class. The only additions are:

- The `datapath_id` field, containing a the id for an OpenFlow data path as mandated by the specification;
- The `ofcs` parameter, which specifies the `OFClientSettings` object used to define the behavior of the `OFClient` instance. If no value is provided, a default set of configuration parameters is used.

The callbacks via which `OFClient` allows the user to customize it are the same as in `OFServer`: `message_callback` and `connection_callback`.

4.1.9 OFServerSettings

Implementation to architectural module mapping

`OFServerSettings` implements the `OpenFlow server settings` module of the libfluid architecture.

The `OFServerSettings` class provides the writing and reading of configuration for a `OFServer` that works via dependency injection (FOWLER, 2004): a `OFServerSettings`

[1] In the example provided in the libfluid repository, `OFClient` configuration is done via a `OFServerSettings` object, since it resembles very closely a future `OFClientSettings` object.

object is built by the user and passed to a `OFServer` instance during initialization, thus defining its behavior. The user writes configuration parameters to a `OFServerSettings` instance, and the `OFServer` instance reads these parameters when deciding how to handle a certain event. These configuration parameters are:

- **Liveness check**: defines whether the default liveness check (via OpenFlow echo messages) should be done by the `OFServer` instance.
- **Echo interval**: the inverval between two OpenFlow echo-request messages used to check connection liveness. It also defines the maximum time an echo-reply message may take to arrive.
- **Handshake**: defines whether the default handshake algorithm should be performed by the `OFServer` instance upon connection establishment.
- **Supported versions**: a set of OpenFlow versions that the `OFServer` instance will accept when performing the handshake with switches.
- **Message dispatching**: defines whether all messages should be dispatched, including those concerning handshaking and liveness checking.
- **Hello elements**: defines whether the OpenFlow hello elements introduced in OpenFlow 1.3.1 should be used when performing handshakes (these elements can potentially break peers with incompatible or older protocol implementations).
- **Memory ownership**: defines whether the user or the `OFServer` instance should be responsible for freeing dynamically allocated memory (buffers) used to store the OpenFlow messages. If the user decides to take ownership of this data, there is no need to copy it for later use, slightly improving performance in some cases.

It is easy to extend this set of configuration parameters in the future, so users can further customize a `OFServer` instance in different ways.

An `OFServerSettings` instance is constructed and initialized as in Listing 7.

```
OFServerSettings()
    .supported_version(0x01)
    .supported_version(0x04)
    .use_hello_elements(false);
```

Listing 7 – `OFServerSettings` constructor.

The object is constructed with default values, and then its methods are called in order to alter these values. Each method call to set a parameter returns a new copy of

the object with the specified parameter set, along with all the ones that were previously set. This is done purely for stylistic reasons (it makes the initialization easier to read in code). In this example, an `OFServerSettings` instance is created, and then support for OpenFlow 1.0 (`0x01`) and 1.3 (`0x04`) is added. Additionally, the use of hello elements is disabled. The resulting object can then be passed to a `OFServer` instance to configure it at runtime.

4.1.10 OFClientSettings

Implementation to architectural module mapping

`OFServerSettings` implements the `OpenFlow client settings` module of the libfluid architecture.

The `OFClientSettings` is analogous to the `OFServerSettings` class. It is currently not implemented (with the `OFServerSettings` being used in its place for now), but it should define all the configuration parameters available in `OFServerSettings` in a way that is suited for an OpenFlow client. It should also define additional parameters, such as enabling or disabling connection attempts and a timeout for these attempts.

The coding style and usage of the `OFClientSettings` class should be very similar to what is present in the `OFServerSettings` class.

4.1.11 TLS

Implementation to architectural module mapping

`TLS` is added as an extra component and is not described in the libfluid architecture.

The OpenFlow protocol specification says that switches and controller should support secure connections. Transport layer security (the concept, not the TLS protocols) is an optional feature in libfluid to enable secure connections, which is not part of its core architecture. It enables `BaseOFConnection` objects to use authentication and encryption for securing the OpenFlow control channel. It is currently enabled only for controllers using libfluid and follows the traditional public-key infrastructure model of the Transport Layer Security (TLS) Protocol v. 1.2 (DIERKS, 2008). It is implemented using the OpenSSL library (OpenSSL Project,), and can be enabled via two functions shown in Listing 8.

The first one, `libfluid_tls_init` enables transport layer security. It requires a certificate to be used by the controller (signed by a certificate authority), a private key which will be used to encrypt data, and a certificate authority's certificate which will be used to define trust in switches connecting to the controller. The `libfluid_tls_clear`

```
void libfluid_tls_init(const char* cert,
                       const char* privkey,
                       const char* trustedcert);
void libfluid_tls_clear();
```

Listing 8 – TLS enablement function.

function is used to free the memory and resources used for implementing the secure channel.

In addition to using these two functions, the user is required to enable the `secure` flag when using any of the `OFServer`, `BaseOFServer` or `BaseOFConnection` classes. Defining the flag at `OFServer` will be enough for enabling transport layer security for all connections to that instance, since the flag's value will be passed to `BaseOFServer` and all of the `BaseOFConnection` instances created by it. It is possible to have both secured and non-secured instances of an `OFServer` running at the same time, listening on different ports.

4.1.12 libfluid_msg

Implementation to architectural module mapping

`libfluid_msg` was not implemented by the author of this dissertation and is not part of the libfluid framework described in this dissertation (which is available in `libfluid_base`).

`libfluid_msg` is implemented as a standalone, independent messaging library. It simplifies message building/parsing in controllers and switch agents by providing classes to build OpenFlow messages with marshalling (pack) and unmarshalling (unpack) methods.

Packing an object results in an OpenFlow message in the network byte order (wire format), ready to be sent through the network. For unpacking, the library parses OpenFlow wire format data (properly adapting byte ordering) and builds a representation of the contents as C++ objects that can be manipulated via methods (mostly getters and setters).

`libfluid_msg` is not a direct part of the architecture proposed in this text. It was developed at the same time as the libfluid core and following similar guidelines for architecture, portability and performance. Its documentation is available together with the libfluid bundle, and it ties in perfectly with the libfluid framework.

`libfluid_msg` is similar to other efforts in the area of messaging libraries for OpenFlow, such as OpenFlowJ (OpenFlow 1.0 in Java) and libopenflow (OpenFlow 1.0 in C++ and Python). Also, like these libraries, `libfluid_msg` follows the manually

coded approach: all structures for manipulating messages are written by a developer. Loxigen (Project Floodlight, 2014) provides an alternative approach: a generator takes as input the OpenFlow specification headers and outputs a set of functions and classes for developers to manipulate binary message data.

The libfluid architecture is flexible enough that it can work with any of these messaging libraries. Some examples illustrating this capability are provided with the code. Chapter 5 shows a sample libfluid-based controller that uses Loxigen-generated code for manipulating messages.

All of the previously existing libraries mentioned above (OpenFlowJ, libopenflow, Loxigen-generated libraries) are used in controllers and switch agents, so this is not our contribution. The controllers and switches make extensive use of these libraries, embedding them into their core frameworks, so that replacing or changing them later can become a daunting task.

The libfluid architecture proposes the separation between the framework components (`BaseOFConnection`, `BaseOFServer`, `OFConnection` and `OFServer` classes) and the messaging library. The framework code has to implement its own set of basic OpenFlow message building/parsing support (and this is done via raw byte manipulation using C structures).

Finding this minimal set of features in an OpenFlow framework is one of the contributions of this work, and it provides us with the ability to easily become independent of OpenFlow version in the framework. Additionally, the developer is then able to choose its own messaging library, without the imposition of a mandatory default.

4.2 Using libfluid

In order to build a **minimally functional controller**, a developer needs to perform the following tasks:

1. Compile and install libfluid;
2. Include libfluid's `OFServer.hh` header file;
3. Create a class definition which inherits from OFServer;
4. Compile the code, linking to libfluid.

This minimal controller is able to listen for connections from switches, follow the OpenFlow handshake procedures, negotiate the protocol version and keep the connection

alive indefinitely (by properly replying to echo requests). It does not perform any other functions. Listing 9 exemplifies such a controller in code.

```
1  #include <fluid/OFServer.hh>
2
3  class Controller : public OFServer {
4  public:
5      Controller() : OFServer("0.0.0.0", 6653) {}
6  };
7
8  Controller c;
9  c.start();
10 // Wait for user interruption
11 c.stop();
```

Listing 9 – A minimal libfluid controller in C++.

Listing 10 shows an example of a **typical controller stub** that allows the user to specify listening address and port, enables support for OpenFlow 1.0 and 1.3 connections, uses 4 threads and secure connections. The two callback methods are provided with empty bodies, but they would normally contain user-defined code for handling events. Functionally, this controller is roughly equivalent to that of Listing 9.

Further customization is possible via different `OFServer` constructor arguments, especially when using an `OFServerSettings` object. A fully functional controller can be built by implementing the two callbacks which are called by `OFServer` upon connection and message events (see Subsection 4.1.7).

In Chapter 5 we will show examples of more functional OpenFlow controllers – which implement these two methods, while also introducing the concept of applications.

The setup required for building an **OpenFlow switch agent with libfluid** is slightly more involved, because in order to have it work minimally, we need to implement (or simulate) a lot of structures available in a switch, such as ports and forwarding tables that are beyond the scope of this work. Despite that, the use of `OFClient` is analogous to that of `OFServer`, with similar callbacks and methods.

For more details on the `OFClient` implementation and the a software-based switch that uses it, see the example implementation provided with the libfluid bundle[2] and the discussion around it in Chapter 5.

[2] <https://github.com/OpenNetworkingFoundation/libfluid/tree/master/examples/switch>

```
1  #include <fluid/OFServer.hh>
2
3  class Controller : public OFServer {
4  public:
5      Controller(const char* address = "0.0.0.0",
6                 const int port = 6653) :
7          OFServer(address, port, 4, false, OFServerSettings().
8                                           supported_version(1).
9                                           supported_version(4)) {
10         // Controller initialization code
11     }
12
13     virtual void message_callback(OFConnection* ofconn,
14                                   uint8_t type, void* data, size_t len) {
15         // Message handling code
16     }
17
18     virtual void connection_callback(OFConnection* ofconn,
19                                      OFConnection::Event type) {
20         // Connection event handling code
21     }
22 }
23
24 Controller c;
25 c.start();
26 // Wait for user interruption
27 c.stop();
```

Listing 10 – A typical stub for a libfluid controller in C++.

5 Evaluation

The evaluation is based on the requirements listed in Chapter 3. For each requirement, we define evaluation approaches that can be qualitative (does libfluid fulfill the requirement?) and/or quantitative (how does libfluid perform in benchmarking tests?). We give an overview of the tools and metrics used for testing and evaluating libfluid in Section 5.1.

Several applications were developed on top of libfluid to enable these evaluation approaches. These applications are detailed in Section 5.2. Table 2 shows the relationship between the requirements, evaluation approaches and applications.

Finally, in Section 5.3 we compare libfluid with the related work in the field of OpenFlow frameworks, controllers and switch agents (which were previously described in Chapter 2).

Requirement	Evaluation approach	Applications
Req. #1: Unified protocol implementation for controllers and switches	Implement both a switch agent and a controller based on the same code base that works with existing switches and controllers (and with each other).	Switch agent Flexible controller
Req. #2: More flexibility in the core of the protocol implementation	Implement different controllers and event handling mechanisms, tweak their settings and analyze how they perform in terms of throughput, latency and fairness under different workloads.	Flexible controller Event handling
Req. #3: A lightweight and portable implementation	Port libfluid to a different CPU platform and to different programming languages. Show that applications still run successfully.	Portability
Req. #4: Independence from messaging libraries and protocol versions	Implement a controller and an application that can simultaneously handle switches using different protocol versions.	Flexible controller
Req. #5: Enable standalone applications	Build an application that uses libfluid directly (without a controller) and analyze the results.	Standalone application
Req. #6: Configurable protocol options	Show different configuration parameters when starting a controller, and discuss how they impact libfluid's behavior.	Flexible controller

Table 2 – Evaluation approaches for the requirements.

5.1 Evaluation tools and metrics

We used some popular tools for evaluating that libfluid works properly and to conduct benchmarks to measure its performance.

Mininet (LANTZ; HELLER; MCKEOWN, 2010) is used to simulate OpenFlow networks with different topologies. It enables from simple connectivity tests (using tools such as ping and iperf) to more advanced network traffic simulation, by emulating switches and hosts. It requires few resources and can run on a standard laptop. It is used to test several libfluid controllers and applications presented in this chapter in a qualitative way (i.e. the application works as expected).

While not discussed in this text, we used tools from the **Valgrind** project (NETHERCOTE; SEWARD, 2007) for checking libfluid for memory leaks and improper memory access during its development.

The `cbench` tool (SHERWOOD, 2013) is used to measure the performance of OpenFlow controllers with two metrics: flow installation throughput and latency. These measurements and the way they are performed are described in Table 3. `cbench` works by simulating the control connections (TCP) of a number of OpenFlow switches connected to a controller.

When conducting a benchmark with `cbench`, the controller is expected to be running a L2 learning switch application. Such an application should respond with *flow-mod* messages to the `cbench`-simulated switch *packet-in* messages, which are meant to instruct the switch to install rules for the traffic flows being detected (or in this case, simulated).

However, the simulated switches only account for *flow-mod* messages and ignore them, never actually learning and thus always sending *packet-in* messages to the controller as if they were always necessary. On the other hand, the controller application being benchmarked may store its local cache of the L2 learning table, thus learning and potentially becoming more efficient as the benchmarks runs and *packet-in* messages are repeated by `cbench` (measuring the efficiency of this cache is also important, as it is part of the application).

For both measurements, `cbench` runs a number of sequential loops of a given time length (both parameters are customizable) and then displays the numbers for each simulated switch in each loop. At the end of the benchmark (when the last loop is finished), `cbench` displays the average and standard deviation for all switches, considering all loops.

Metric	Description	Unit
Throughput	`cbench` simulates the control channel connections of a number of OpenFlow switches filling the TCP buffers with as many OpenFlow *packet-in* messages as possible. Throughput is measured as the number of *flow-mod* messages – matching a previously sent *packet-in* message – that a controller can send in a given period of time.	*kflows per millisecond* The higher the better.
Latency	`cbench` simulates the control channel connections of a number of OpenFlow switches, sending one OpenFlow *packet-in* message and waiting for the controller to respond with an OpenFlow *flow-mod* message. Latency is measured as the time it took from the moment the *packet-in* message was sent until the time the *flow-mod* message arrived in response.	*microseconds* The lower the better.
Fairness	The standard deviation of (flow installation) throughput on a set of switches. A controller could prioritize a small subset of connected switches while others are starved, which is not desired (and thus the importance of this measurement).	*relative to metric* The lower the better.

Table 3 – Metrics for evaluating performance.

5.2 Evaluation applications

5.2.1 Flexible controller

The flexible controller example was built to demonstrate the key points of the libfluid architecture when building a controller. It is actually not a single application, but a common core upon which different controller variations are built. The different controllers are built to show that the requirements proposed in Chapter 3 are in fact implemented by libfluid.

The overall architecture of the flexible controller is shown in Figure 8. There is a common core, illustrated in the *Controller abstractions* portion of the diagram, providing key utilities that are usually present in several existing OpenFlow controllers:

- `Controller`: a class representing the controller, which builds on top of libfluid's `OFServer` class for providing OpenFlow functionality. Different controller variations inherit from this class.

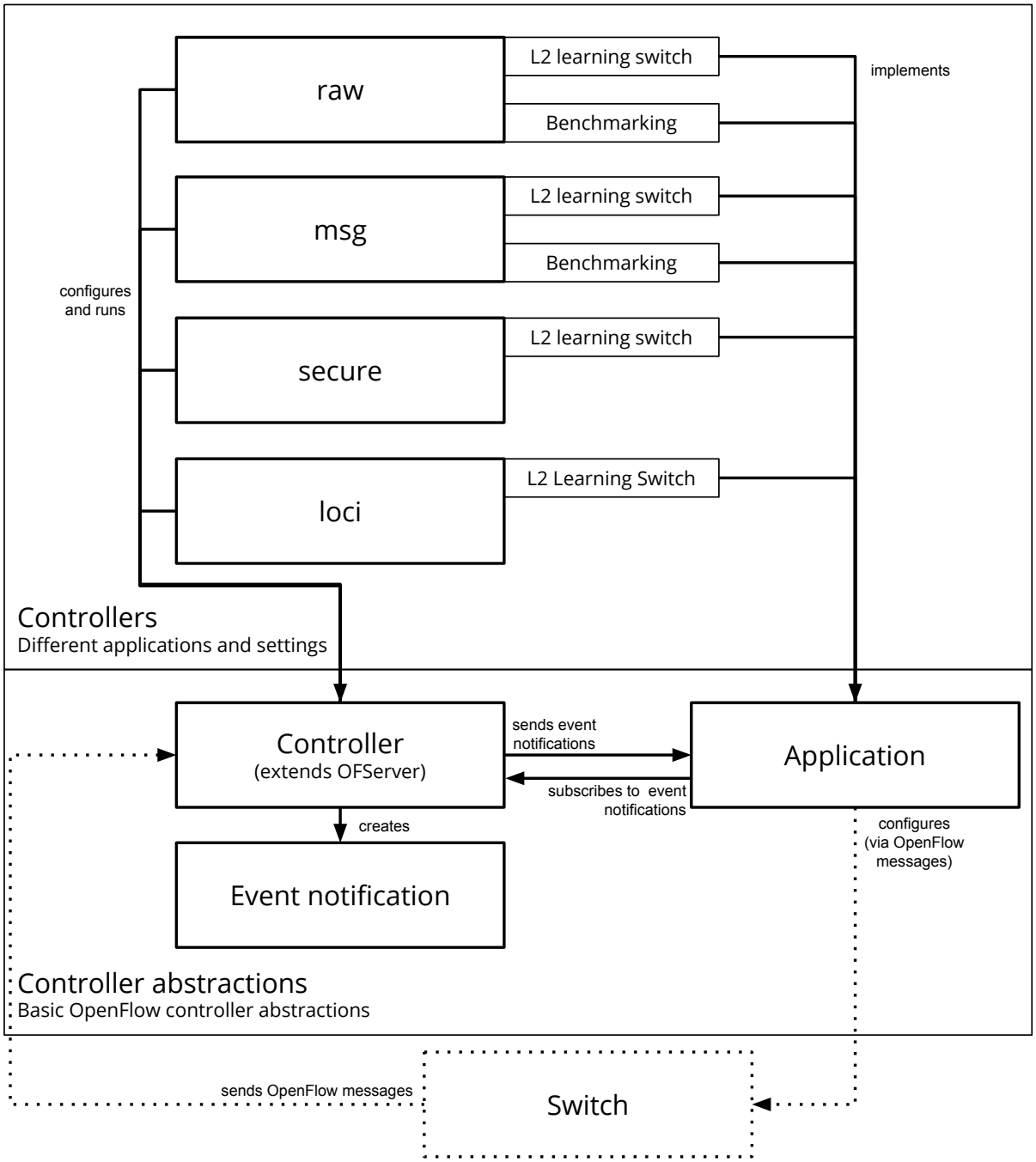


Figure 8 – Conceptual view of the flexible controller.

- `Application`: an application represents a user-defined piece of software that runs on top of the controller. It asks the controller to be notified about events, and then reacts to these events. Applications are specific to a controller, and they register themselves in a `Controller` instance for receiving event notifications of interest to them.

- `Event notification`: an event notification represents a network event (such as a switch connecting or a *packet-in* message from the switch). The `Controller` class create this type of object and then sends them to `Application` instances interested in this type of notification.

On top of these key abstractions, we developed four different controllers, with their own set of applications, as shown in the *Controllers* portion of Figure 8. Further details on these controllers, and why they were built, are shown in Table 4.

There are two types of applications present in the controller implementations:

- `L2 learning switch`: L2 represents the data-link layer in the TCP/IP model. This application implements a L2 learning switch (MAC-address based). It makes an OpenFlow switch mimic the behavior of a traditional switch, enabling it to forward packets in an L2 domain. It exercises important parts of an OpenFlow controller, such as the parsing of *packet-in* messages, packet output, flow installation and switch state tracking. It is also used in benchmarks when comparing with other controllers, since this application is present in almost all OpenFlow controllers.

- `Benchmarking`: A benchmarking application intended to be used with the `cbench` application in order to measure controller throughput and latency when comparing different configurations in libfluid (that is, comparing libfluid with itself). This application simply replies any *packet-in* message with a fixed-content *flow-mod* message, without any processing or learning.

The implementations of these applications are not the same across controller variations (only their names and purposes are the same). For example, the `L2 learning switch` application in the `raw` controller is built using C code for byte manipulation, while the same application in the `msg` controller application uses the object-oriented `libfluid_msg` library for achieving the same purpose.

All controller variations, and all applications in them were successfully tested with hardware and software-based switches and with the Mininet tool.

The `raw` and `loci` controllers show that libfluid can be used with different messaging libraries without the major effort of rearchitecturing the controller.

Controller	Purpose	Description
`raw`	Build a high-performance, pure libfluid controller	The `raw` controller builds on top of libfluid and implements applications using manual byte manipulation, without the aid of any OpenFlow message parsing/building library. The `L2 learning switch` and `Benchmarking` applications are implemented in this controller.
`msg`	Build a controller integrated with the `libfluid_msg` library	The `msg` controller builds on top of libfluid and implements applications using the `libfluid_msg` library for parsing and building OpenFlow messages, which is designed to be easy to use together with libfluid. The `L2 learning switch` and `Benchmarking` applications are implemented in this controller.
`secure`	Show how secure and non-secure connections can be used simultaneously in libfluid.	The `secure` controller builds on top of libfluid and provides secure connections using the SSL protocol. It shows that it is possible to run controller application instances (`L2 learning switch`) and use them to control the network via normal and secure channels simultaneously. It uses the `raw` controller implementation as its basis, but only enables the `L2 learning switch` application.
`loci`	Show that libfluid can integrate with third-party OpenFlow message building/parsing libraries.	The `loci` controller uses code automatically generated by the Loxigen project. Only the `L2 learning switch` application is implemented in this controller.

Table 4 – The different controllers implemented using libfluid.

The secure connection channel provided by the `secure` controller was successfully tested with the Open vSwitch software (PFAFF et al., 2009).

The configurable options of libfluid were also used when implementing these controllers, as shown in Listing 11. In this example, support for OpenFlow versions 1.0 and 1.3 is declared (versions 1 and 4, respectively). The `keep_data_ownership` flag is set to `false`, indicating that the `OFServer` class should not free the memory allocated for the OpenFlow messages it receives. This allows a `Controller` instance to pass the same chunk of memory to applications, saving an unnecessary copy operation. When the applications are done processing messages, the `Controller` instance then frees the allocated memory. This setting produces small improvements in throughput and latency.

```
OFServerSettings()
    .supported_version(1)
    .supported_version(4)
    .keep_data_ownership(false)
```

Listing 11 – `OFServerSettings` as used in the controllers.

5.2.1.1 Benchmarks

In addition to implementing the four controllers, we also conducted benchmarks using `cbench` to measure throughput and latency as described in Table 3.

Four benchmarks were conducted, with different configurations of libfluid and other OpenFlow controllers. They are not meant to be exhaustive, since there are too many variables involved in each test, but they should show that a libfluid-based controller can perform at least as well as existing controllers (Benchmarks 1 and 2), that it can take advantage of the parallelism available in modern hardware (Benchmark 3) and that it works well in larger networks (Benchmark 4). Benchmark results consist of the average (and standard deviation as error bars) of 16 loops of 10 seconds each.

Benchmarking setup

We used the following hardware and software combination to perform the benchmarks:

- Intel Core i7-2600 CPU (4 cores, 8 threads @ 3.4 GHz);
- 8 GB of RAM (@1333 MHz);
- Running Ubuntu 12.04.2 and its standard software stack.

Recommended benchmarking guidelines were used when available for each controller. TCMalloc was used for better performance in C/C++. OpenJDK 7 64-bit provides the JDK/JRE for the Java controllers.

For more details on the setup, see:
<http://opennetworkingfoundation.github.io/libfluid/md_doc_Benchmarks.html>

A measurement of the throughput of different controllers running a **L2 learning switch** application is shown in Figure 9. The benchmark was conducted with `cbench` simulating **16 switches**, and comparing the **Beacon, NOX MT, Floodlight, `raw`** and **`msg`** controllers when configured to run with **8 threads**.

This benchmark is provided to show that a libfluid-based controller can perform at least as well as existing controllers in terms of throughput. Since learning switch applications perform pretty much the same functions in all controllers (despite implementation variations), this application is a good fit for this benchmark.

The result shows that libfluid in its fastest configuration (the `raw` controller) has a slight advantage in throughput compared to the Beacon controller. It is also very well-positioned when compared to the other controllers. Standard deviation is shown for all controllers as error bars in the graph, but the numbers are very small when compared to the overall throughput (around 0.6% of the throughput in the worst case).

The `msg` controller shows an intermediate performance, mostly because of the overhead incurred by the constant creation and destruction of objects (several for each message) in `libfluid_msg`. Beacon overcomes this issues via the smarter memory allocation mechanisms present in Java. It is possible to improve the `msg` controller throughput by minimizing object creation and destruction by using an object pool.

It is important to note that the throughput numbers for any of these controllers is more than enough, since most switches cannot handle so much flow installation requests anyway (APPELMAN; BOER, 2012).

A measurement of the latency of different controllers running a **L2 learning switch** application is shown in Figure 10. The benchmark was conducted with `cbench` simulating **16 switches**, and comparing the **Beacon, NOX MT, Floodlight, `raw`** and **`msg`** controllers when configured to run with **8 threads**. The purposes of this benchmark

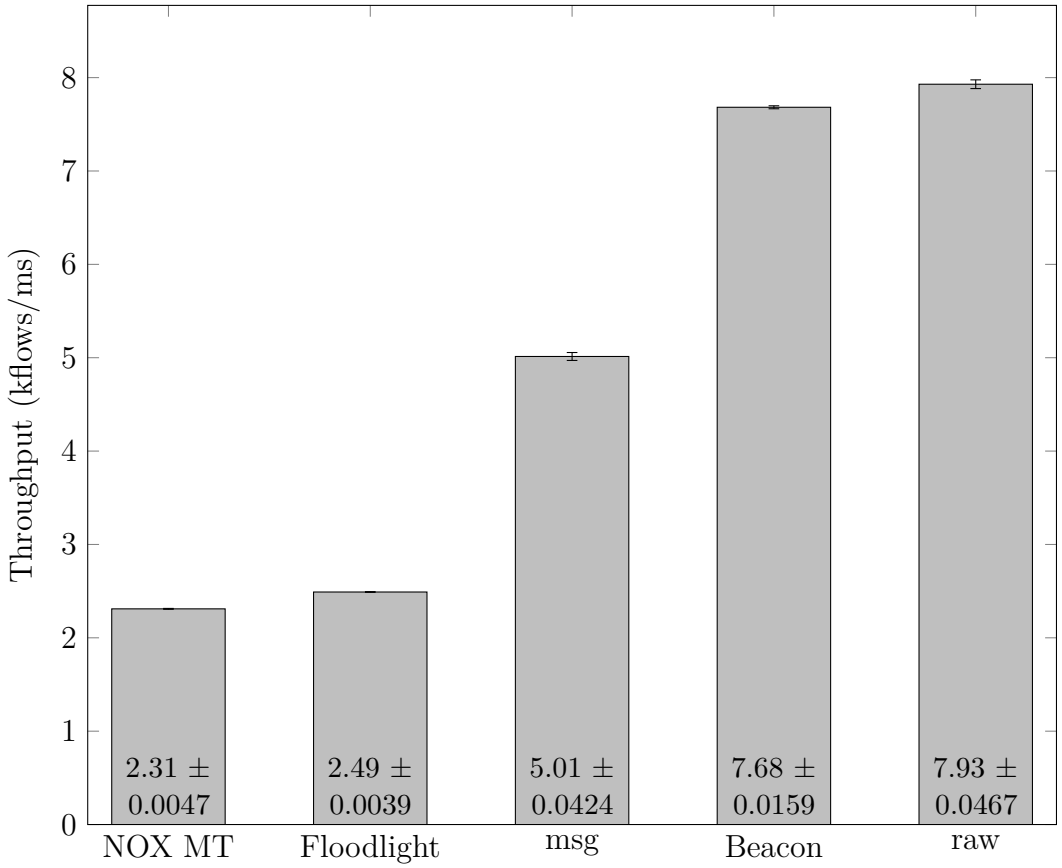


Figure 9 – Comparing throughput for controllers running a L2 learning switch application (higher is better).

are similar to the those of the previous one (throughput), but now for the measurement of latency.

The result shows that libfluid latency performance is up to par with other controllers in an idealized scenario. A latency benchmark in a real network still remains to be done, and is likely to show that the differences in latency shown here are irrelevant when the cost of physical network communication is taken into account (i.e. the figures for all controllers would be similar to each other but much higher, rendering the microseconds differences shown here irrelevant).

Latency and throughput have an inverse relationship: it is possible to improve throughput by using larger buffers for reading and writing, but this increases latency. It is possible to improve latency (decreasing it) by using less buffer space and reading and writing more frequently via system calls (CAI ALAN L. COX, 2011). Finding a balance between both is ideal for most applications, and libfluid does that by having high-throughput without sacrificing too much latency. The benchmark shows that libfluid is more constant regarding latency (along with Beacon), as shown by the smaller error bars in the graph (depicting standard deviation).

A measurement of how throughput varies when threading configuration

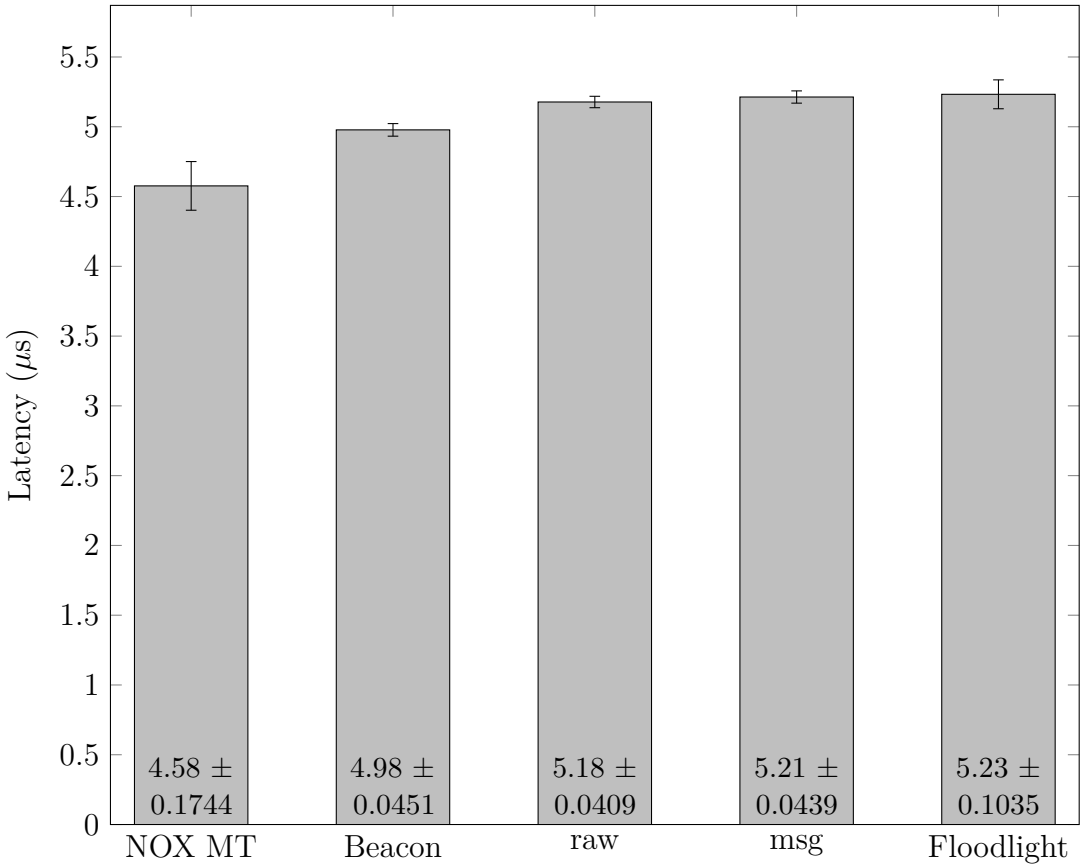
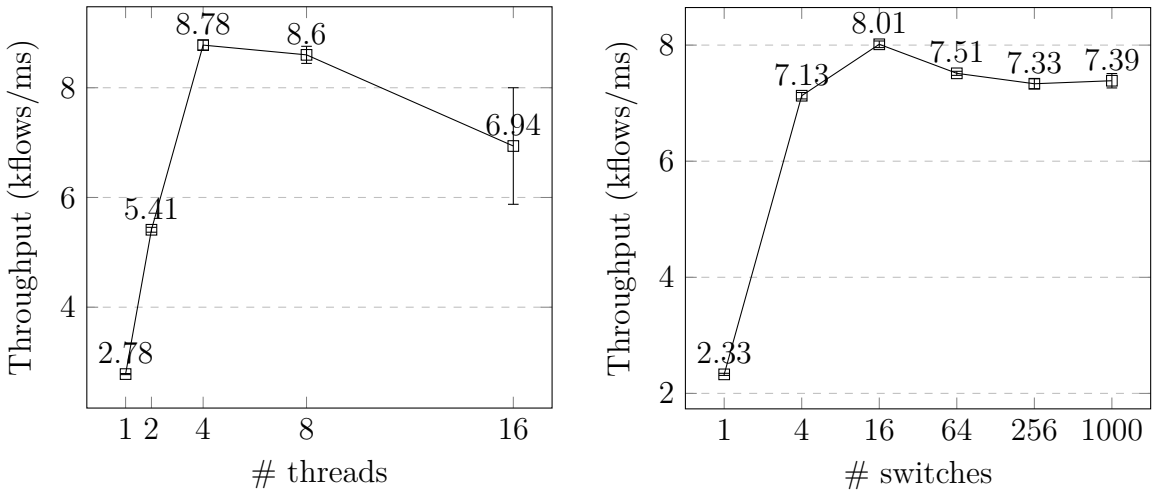


Figure 10 – Comparing latency in controllers running a L2 learning switch application (lower is better).

changes is shown in Figure 11a. This benchmark was conducted with the `Benchmarking` application of the `raw` controller, in a network with **16 switches**. The number of threads in use by the controller is changed in each test. This benchmark is intended to compare libfluid with itself and show how throughput is affected as the number of deployed threads changes.

The result shows that in an ideal scenario in which no thread synchronization is required, libfluid's throughput performance scales almost linearly up to the point where it matches the number of available physical cores in the CPU (4). After reaching this peak, throughput decreases slightly when using all cores (physical + virtual) in the CPU (8); we attribute this result to memory access bottlenecks. When more cores are used (16) than the CPU provides (8), throughput decreases even further (and starts to vary a lot, as shown by the error bars in the graph), to a point where the number of threads is detrimental to performance.

A measurement of the throughput when the network size changes is shown in Figure 11b. This benchmark was conducted using the `raw` controller with the `Benchmarking` application. The number of switches connected to the controller is changed in each test. This benchmark is intended to compare libfluid with itself and show how its



(a) Average throughput of 16 switches as the number of threads changes in `raw` controller running the `Benchmarking` application (higher is better).

(b) Average throughput as the number of connected switches changes in the `raw` controller running the `L2 learning switch` application with 8 threads (higher is better).

Figure 11 – Throughput behavior when varying the number of threads and connected switches.

throughput scales as the number of active control connections increases.

The result shows that libfluid scales gracefully when the number of switches in the network increases. Throughput reaches its peak when there are 2 switch connections per thread, but it does not decrease too much when the number increases even further, to 125 switch connections per thread. Standard deviation is shown for all number of threads as error bars in the graph, but it is not visible due to small values (around 1.7% of the throughput in the worst case).

We attribute these good numbers (even with larger numbers of connections) to libfluid's event loop (actually implemented by libevent) which is designed with the C10K problem in mind (KEGEL, 2011).

5.2.2 Event handling

Starting from the `raw` controller of subsection 5.2.1, we modified it to add packet logging functionality to the `L2 learning switch` application. The purpose of this change is to stress libfluid as much as possible in a kind of scenario in which it might perform poorly: time-consuming operations (such as logging, in this case).

We want to show that by carefully analysing benchmarks with knowledge of the libfluid architecture, it is possible for developers to optimize the usage of libfluid in their code.

For the purpose of stressing the controller (and thus libfluid), we simulated a feature present in traditional Ethernet switches known as port mirroring: switches can be configured to send a copy of network traffic in one switch port to another port. This other port can then be connected to a monitoring interface on a computer, allowing for intrusion detection or troubleshooting (SANDERS, 2011).

In this example, we adopt a slightly different approach to the port mirroring concept: we log all packets that go through a given switch (and not a single port) instead of instructing it on how to forward that traffic. The controller receives the packets, and then logs and forwards them (using a *packet-out* OpenFlow message). We call the switch whose traffic is being logged the **logged switch**.

This means that, for one or more switches in a given set, the learning switch application instead acts as a traffic logging and forwarding application. We did not implement the traffic logging logic for egress traffic, since the benchmarking tool we used (`cbench`) only simulates ingress traffic, which is enough for our purposes. For switches whose traffic is not being logged, no changes are made: the `L2 learning switch` application acts normally, installing rules for traffic forwarding in the switch.

Log data is output to a capture file in the PCAP format (libpcap project, 2015), which can be a costly operation (it may take a while and lock while data is buffered or written to disk).

Long-running operations (such as disk-, network- and CPU-bound operations) should be avoided in asynchronous, non-blocking event loops such as the ones we use in libfluid. This is not a limitation, but rather an architectural trade-off. Sometimes however, such as in the packet logging case (or deep packet inspection applications), it may be unavoidable to have long running operations. We will analyze how to work around the overhead imposed by this sort of operations[1].

The diagram in Figure 12 illustrates a scenario in which three switches are connected to a controller. Due to the distribution of switches among threads, this is a possible scenario. Assuming that the traffic in Switch #2 is being logged, we can see that Thread 2 is potentially under more stress than Thread 1: not only it has to handle events from two switches, it also has to perform the work of saving all packets going through Switch #2. The `Packet logging` activity is specially resource and time consuming, because it may require writes to disk. Other types of tasks that could cause the same problem are CPU bound activities (cryptography, graph algorithms) and other blocking IO operations

[1] As an example, consider the NodeJS platform for building asynchronous applications, which features an event loop as the core of its design. In NodeJS (Joyent, Inc., 2014), synchronous file I/O operations are offloaded to a thread pool instead of being run in the main loop. The result is communicated back via internal threading communication mechanisms. We chose not to use this mechanism in order to avoid making libfluid too complex and harder to port to other architectures and programming languages.

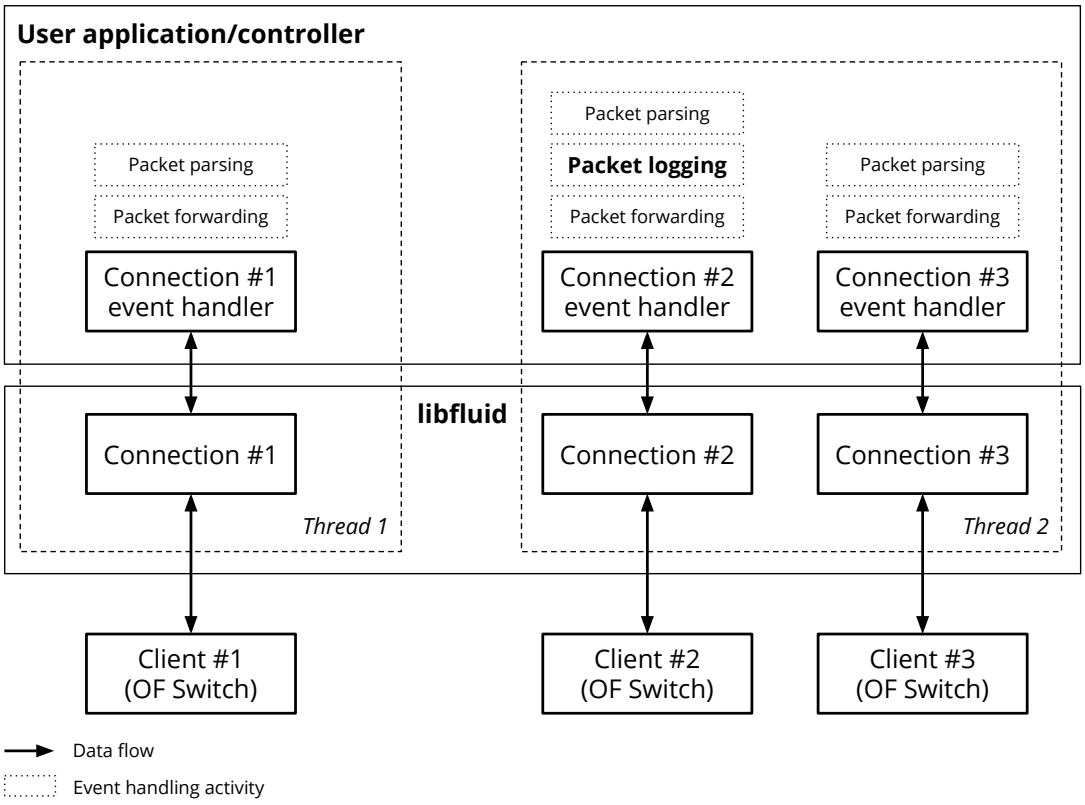


Figure 12 – Event handling unbalance.

(writing to disk or to a socket).

A possible solution to this issue would be offloading long-running or locking IO-bound tasks from the main event loops to other threads, avoiding that event loops become unnecessarily overloaded, impacting the handling of other connections' events. However, this brings the issue of thread synchronization: we need to use a synchronized data structure in order to offload work to the other thread (otherwise, race conditions may happen).

We chose three event handling approaches to the work-offloading task, and we compare how they fare regarding throughput and latency:

- **Run-to-completion**: all the event handling procedures are performed in libfluid's event loop callbacks: the packet is parsed, logged, and then the appropriate forwarding action is taken. This is the easiest approach, and only requires using libfluid as it is (without adding any event queue).
- **Synchronized queue**: the basic event handling procedures are performed in libfluid's event loops: the packet is parsed and forwarded. Logging, however, is delegated to a dedicated thread via a synchronous queue (that is made thread-safe by using mutexes).
- **Ring buffer**: the basic event handling procedures are performed in libfluid's event loops: the packet is parsed and forwarded. Logging, however, is delegated to a

dedicated thread via a special data structure, a wait-free ring buffer (BLECHMANN, 2015).

The first two approaches are directly derived from the work on the Beacon controller (ERICKSON, 2013). These are traditional ways of performing processing delegation from an IO-bound thread to a CPU-bound thread. The third one uses a special implementation of a well-known data structure: a wait-free ring buffer.

A wait-free ring buffer is a data structure that can be used to implement a special kind of queue: it allows one thread to add objects to a queue (producer), while another threads reads the objects (consumer) without the use of expensive memory locking. This is made possible by the use of atomic CPU instructions along with the memory organization of a ring buffer acting as a queue.

In the **Ring buffer** and **Synchronized queue** approaches, the event loop thread of libfluid adds packets to a queue (producer), and another thread is used to log packets (consumer). This thread performs only packet logging and is executed as a lower-priority thread (using Linux thread-scheduling primitives).

5.2.2.1 Benchmarks

For benchmarking purposes, we varied the number of threads (1, 2, 4 and 8) and switches connected to the controller (8, 16, 32, 64, 128). In all benchmarks, traffic going through **one** of the switches (for any number of switches) is being logged before being forwarded. The results are formed by the average throughput and latency of 10 loops of traffic bursts lasting 5 seconds each.

Benchmarking setup

We used the following hardware and software combination to perform the benchmarks:

- Amazon EC2 c4.2xlarge VM (8 vCPUs);
- 15 GB of RAM;
- Running Ubuntu 12.04.5 and its standard software stack.

The machine used for these benchmarks was not the same as the one we used in Subsection 5.2.1 because we did not have access to it anymore.

For more details on the setup, the scripts used to run the tests, and detailed results, see: <https://github.com/alnvdl/lf_evhandling>

Tables 5 and 6 show which event handling approach offered maximum throughput (Table 5) and minimum latency (Table 6), even when only by a slight margin. The results

show an overview of the sorts of scenarios (number of threads, switches) in which each approach performs the best for different metrics.

# threads \ # switches	8	16	32	64	128
1	RB	RB	RB	RB	RB
2	RB	RB	RB	RB	RB
4	RTC	SQ	SQ	RTC	SQ
8	SQ	SQ	SQ	SQ	SQ

Table 5 – Best event handling approach for optimizing average throughput.

RB Ring buffer

SQ Synchronized queue

RTC Run-to-completion

# threads \ # switches	8	16	32	64	128
1	RB	RB	RTC	RTC	RTC
2	RTC	RTC	RTC	RTC	SQ
4	RTC	SQ	RB	SQ	SQ
8	RB	RTC	RTC	RB	RTC

Table 6 – Best event handling approach for optimizing average latency.

Table 5 shows that the **Ring buffer** approach offers the best throughput for small numbers of threads (up to 2 threads). As the number of threads increases, the **Synchronized queue** approach offers better throughput. This can be explained in part because, as the number of threads increases, the overhead of mutex-based synchronization used by the **Synchronized queue** approach becomes less important, since only one of these threads is constrained by locking (the one handling events of the logged switch), while the other threads are free to quickly respond to events.

Table 6 shows a less clear picture, but seems to point to a slight advantage of the **Run-to-completion** approach, which can be explained by the smaller number of threads used by it (one less: it does not make use of a packet logging thread, performing logging operation in the event handling thread). Since there are fewer threads competing for resources (adding to context switches costs), latency is slightly improved.

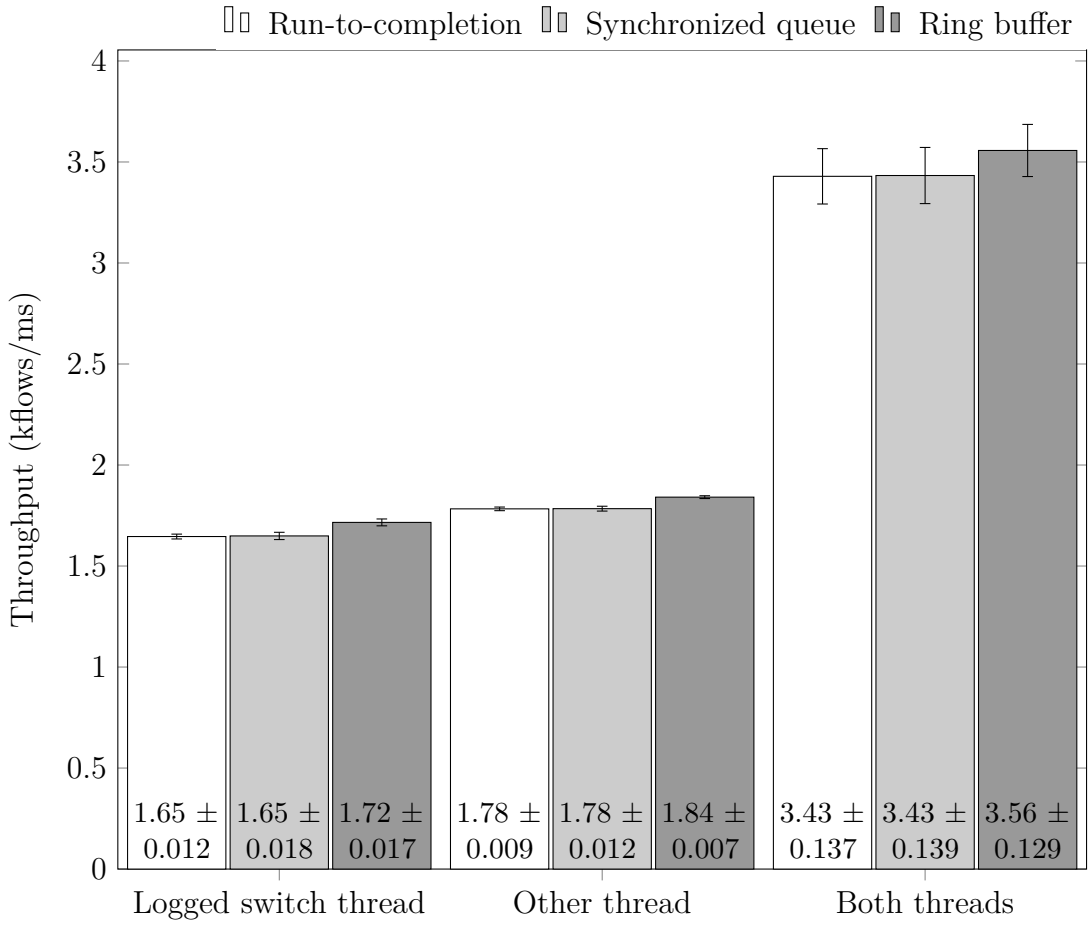


Figure 13 – Throughput with different event handling approaches for a workload of 32 switches distributed in 2 threads, with traffic from one switch being logged to a file.

In addition to these general benchmarks, it is interesting to dive deeper into one of the scenarios present in Table 5 to further understand the sorts of trade-offs a developer would face when using libfluid with different event handling approaches. For that purpose, we chose the throughput measurement with 2 threads and 32 switches because it provides interesting results for analysis. The detailed results from this benchmark are shown in the chart in Figure 13, which is divided in 3 parts:

- The first portion of the chart (*Logged switch thread*) shows the average throughput of the 16 switches that are sharing a event handling thread with the switch that is having its traffic logged (the logged switch). This average is formed by the 15 switches not being logged, and by the logged switch. This thread has a higher workload, since it has to perform the offloading of packets for logging to another thread (being potentially subject to locking) in addition to installing rules for traffic forwarding.

- The second portion of the chart (*Other thread*) shows the average throughput of the 16 switches that are sharing an event handling thread. No switches in this thread

are having their traffic logged, so only the installation of rules for traffic forwarding is performed.

- The third portion of the chart (*Both threads*) shows the average throughput of all 32 switches, regardless of the threads being used for handling their events. Standard deviation is higher because this average takes into account the throughput of switches being handled by the *Logged switch thread* (which is reduced due to the additional workload) and switches being handled by the *Other thread*, thus causing greater variation.

Average throughput when using the **Ring buffer** approach is slightly better than when using the **Run-to-completion** and **Synchronized queue** approaches. This holds true when considering both threads individually. When taking a complete average (32 switches in both threads), the results are inconclusive in terms of average throughput.

In contrast, fairness (which we measure based on standard deviation as described in Table 3) is slightly improved in the scenario shown in Figure 13. This happens because when using the **Ring buffer** approach, there is less throughput difference (in average) between switches in threads with different workloads (logging vs. non-logging).

Thus, considering all metrics (inconclusive throughput, better fairness) in this scenario (2 threads, 32 switches), the **Ring buffer** approach can be considered the best fit. However, as the results in Tables 5 and 6 show, this is not always true, and different event handling approaches yield different results when the scenario changes (number of switches and threads), sometimes resulting in inconclusive numbers. For example: a higher number of threads makes the **Synchronized queue** approach better for throughput.

Therefore, the best way to fine tune libfluid is developing the sort of event handling approaches we outlined here and then conducting benchmarks. Performance alone may not be the only reason for choosing an event handling approach though (especially in edge cases where there are no clear-cut results). It is also important to take into account development effort and code complexity. In this case, if the developer wishes for simpler event handling, it is possible to choose a more traditional approach, such as the **Run-to-completion** approach. The **Synchronized queue** and **Ring buffer** approaches are slightly less straighforward to implement, but they may be a better fit if long-running operations are constantly performed.

These three different approaches to event handling illustrate how developers are able to leverage the threaded architecture of libfluid to fine tune their applications. By changing the way threads are deployed and then using an appropriate data structure, it is possible to favor latency, throughput, fairness or development effort, even in scenarios in which libfluid is put under stress.

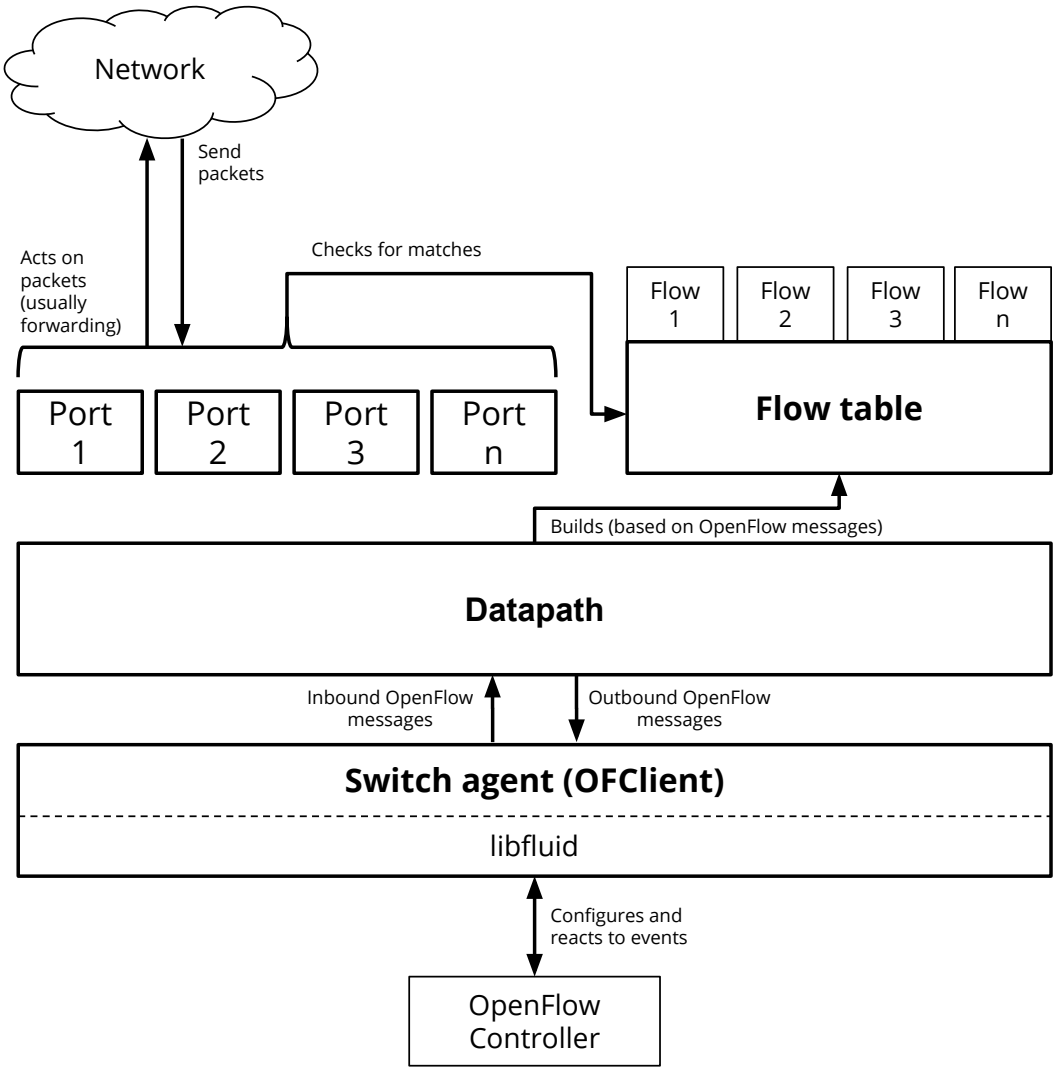


Figure 14 – Building blocks of the libfluid example switch.

5.2.3 Switch agent

In order to demonstrate how to implement a switch agent on top of libfluid, we leveraged the `OFClient` class and implemented a very small software switch, which runs as a program and simulates the behavior of a real switch by forwarding traffic between virtual network interfaces. The switch is implemented to the point where it can act as a learning switch when controlled by the libfluid controller described in Subsection 5.2.1. This example was contributed to us by Eder Leão Fernandes (the developer of `libfluid_msg`).

Figure 14 illustrates the main blocks involved in this sample switch implementation. The switch agent itself is just part of the overall implementation. It is implemented as an `OFClient` instance, building upon libfluid to handle the interaction with the controller via OpenFlow messages.

The datapath structure is the central point in this switch model, performing several tasks:

- It builds the flow table from OpenFlow messages received from the controller (which contains sets of flows: rules for forwarding traffic);
- It creates and manages the lifecycle of Port instances, which are responsible for performing the actual network traffic forwarding;
- It forwards events of interest in the network to the controller (e.g.: inform the controller about packets received from the network that could not be forwarded).

Packets are captured and forwarded by the ports using the libpcap library (libpcap project, 2015). Each port runs in a dedicated thread and consults the flow table when forwarding packets (all of this is inefficient, and it was done only because this was the simplest way to do it for demonstration purposes).

libfluid's footprint in this implementation is very small, providing only the support for sending and receiving OpenFlow messages with a few lines of code. This example is useful to highlight how libfluid can be leveraged not only to build control applications, but also to implement OpenFlow into network switches.

The switch example has been successfully tested with the libfluid controllers of Subsection 5.2.1. It was also embedded in Mininet as a switch that can be used in simulated networks. Further details on how to run it are available in libfluid's website.

5.2.4 Portability

In this text, we refer to portability as not only the ability to run software in different computer platforms (e.g.: different microprocessor architectures and operating systems), but also the ability to use the same software in different programming languages. libfluid achieves both of these goals via architectural decisions and coding practices:

- Platform-specific code encapsulation: code that is specific to platforms is encapsulated or wrapped by libfluid's code.
- No platform-specific types in the public API: the API exposed to users uses only primitive types such as strings, integers and binary values in a way that is portable across different operating systems (using types defined by standard C headers).
- A minimal public API: in addition to only using primitive types, having a small public-facing API helps when building bindings for other programming languages, since there are less classes and methods to port and adapt.
- A third-party, cross-platform event loop: by using libevent, which can be compiled in many operating systems and computer architectures, libfluid is able to be easily

ported. The task of managing sockets, selectors, threads and IO mechanisms in different operating systems would be much harder if we did not reuse an existing work.

5.2.4.1 Cross-platform build

There is a port of libfluid to Android in ARM available in libfluid's website (libfluid project, 2014). It runs the `L2 learning switch` application of the `msg` controller described in Subsection 5.2.1 as an Android application.

libfluid is cross-compiled to Android/ARM without any modifications using the toolchain provided by Android (Android Open Source Project, 2015). In order to build it, we also needed to cross-compile libevent (also without any modifications). In order to call the C/C++ code in Java, it was necessary to create a minimal adapter using Java Native Interface (JNI) (Oracle, 2015).

The application runs successfully on both the Android SDK emulator and in Android (ARM) devices. We tested it by connecting a smartphone (Samsung Galaxy S2 Lite running Android 2.3) to a wireless network, creating a virtual network with a few switches in Mininet running in a laptop that was also connected to the wireless network. The controller running in the smartphone was able to react to network events and make the virtual switches act as L2 switches.

5.2.4.2 Other programming languages

Using frameworks in programming languages other than the one used to write the framework itself is considered a challenging task because most frameworks are closely tied to features specific to a programming language (JOHNSON, 1997). libfluid overcomes this issue by using a popular base language (C/C++) and neutral design decisions, as listed earlier in this subsection.

Bindings for other programming languages are generated using the SWIG tool (BEAZLEY et al., 1996). SWIG reads C/C++ header files with some special directives (`.i` files) and outputs artifacts that enable the use of the C/C++ constructs in a target language (Python, Java, PHP, C# and many others are among the supported languages).

When using SWIG to generate bindings from C/C++ code to other languages, developers must take care with a few constructs and avoid using others. The specific details depend upon the target language, and there is a manual describing the best practices and workarounds (SWIG project, 2015).

We created Java and Python bindings for libfluid. One of our concerns when we designed libfluid's public API was to ease the generation of these bindings, and we did

it by making the API very small (just a handful of classes and methods) and using primitive types when possible. libfluid is also object-oriented, which makes it easier to use in languages such as Python and Java (developers in these languages are used to object orientation constructs).

Code developed using these bindings ends up being very similar to their equivalent C/C++ counterparts. As an example, compare the minimal controller shown in Listing 9 in Chapter 4 to its Python equivalent illustrated in Listing 12.

```
1  from fluid.base import OFServer, OFConnection
2
3  class Controller(OFServer):
4      def __init__(self):
5          OFServer.__init__(self, "0.0.0.0", 6653, 1)
6
7  c = Controller()
8  c.start()
9
10 # Wait for user interruption
11 raw_input()
12
13 c.stop()
```

Listing 12 – A minimal libfluid controller in Python.

The bindings for Java and Python are available in libfluid's repository. Variations of the `L2 learning switch` application were built in each language and successfully tested against physical and virtual switches. The Python bindings for `libfluid_msg` were also used experimentally in the POX controller [2].

These bindings are still considered to be experimental, but they show how libfluid can be easily made available and used in other programming languages.

5.2.5 Standalone application

A standalone application is one which can directly communicate with OpenFlow switches in order to work, without the need to run on top of a third-party infrastructure (such as an SDN controller). Typically, such an application will be a small, single-file executable. Building a standalone executable may be useful for very small and focused SDN applications for testing, debugging, teaching or research purposes.

While deciding which application to build in order to showcase this feature of libfluid, we came across several ideas such as firewalling application and HTTP proxy

[2] <https://mailman.stanford.edu/pipermail/openflow-discuss/2014-April/005315.html>

applications. However, we ended up noticing that one of our previous work in the area of SDN, RouteFlow (NASCIMENTO et al., 2011), was a perfect match for this paradigm.

RouteFlow is a project which aims to provide traditional virtualized IP routing services (such as OSPF and BGP) on top of OpenFlow switches. By providing a way for traditional routing protocols to run on top of OpenFlow switches using existing software (e.g.: Quagga), RouteFlow creates a migration path towards SDN, in which SDN networking devices may interact with legacy (non-SDN) infrastructure.

The RouteFlow architecture is composed by three layers:

- **A virtualized routing environment**: virtual machines (or containers) running routing engines and a RouteFlow daemon called **RFClient**, which notifies a RouteFlow server about changes in Linux routing tables.
- **A routing server**: represented by the **RFServer** module, which gathers routing information from the connected RFClient daemons, processes these routes as needed, translates them into a flow-representation and forwards them to a special application running in an OpenFlow Controller.
- **A controller application**: represented by the **RFProxy** module, which runs on top of existing SDN controllers (there are implementations for OpenFlow controllers such as POX, NOX and Floodlight). Upon receiving a flow change instruction from the RFServer, RFProxy converts it into an OpenFlow message to achieve the desired behavior. RFProxy also listens for special packets on the network (such as routing information packets), and forwards these packets to the virtual environment where they are converted into routing table updates.
  RFProxy was developed to run on top of existing controllers, and assumes a great degree of freedom (i.e. it assumes that no other running application will interfere with or be affected by its behavior).

Because of the its particularities, RFProxy is a good match for building a standalone application. There is an existing implementation in C/C++ for the NOX controller, and we used that as the basis for the libfluid implementation of the RFProxy standalone application.

The NOX controller itself is not seeing much development recently, and so RFProxy for NOX was also abandoned. However, we were able to port most of the code with a little effort, and it worked fine with the rest of the RouteFlow code[3]. We successfully tested this implementation with the default test cases provided by RouteFlow.

One of the advantages of having a C/C++ level controller is the lower latency that is possible to achieve when forwarding packets from the controller application to the

[3] The code is available at: <https://github.com/alnvdl/lf_rfproxy>

virtualized environment[4]. Another advantage is that more powerful optimizations become possible in C/C++, a lower-level programming language (compared to Python or Java).

Additionally, having a standalone application for RFProxy simplifies the RouteFlow architecture a great deal in some scenarios. Since there is no need for an external controller, the setup becomes much easier (it just involves compiling a small project with a few dependencies).

Finally, it is possible to use switch virtualization features to make a standalone application even more powerful: if just a subset of the ports available in a switch is allocated to a given controller (in this case, the standalone application), it becomes possible to run several applications on different segments of the same switch. An example of software that enables this sort of virtualization is FlowVisor (SHERWOOD et al., 2010). In this scenario, part of an OpenFlow switch could be running a RouteFlow-based router, while other ports run a firewall, with both applications running as standalone, low-overhead implementations based on libfluid, possibly enabling more flexible and cost-effective network setups. In this case, the operating system could become the SDN controller, managing interactions between applications competing for resources (network devices).

5.3 Comparison to related work

In the following Subsections we present a brief comparison with the related work mentioned in Chapter 2, in order to evaluate and position our work by highlighting differences and similarities.

5.3.1 Controllers

NOX: While the NOX design heavily influenced other controllers, it was meant to provide high-level abstractions to programmers such as a global network view and applications management. As for the new development on NOX-MT, its multi-threaded design is similar to that present in libfluid and Beacon, but it is tightly integrated into the controller, making it harder to reuse in switch agents or standalone applications. NOX and NOX-MT also only support OpenFlow 1.0, though there are unofficial forks supporting other versions (CPqD, 2013). The author of these versions described the work of extending NOX to support these additional versions as complex. libfluid differs from NOX (and

[4] We actually noticed this in our tests when comparing with the existing POX-based implementation, but we will not go into further details because it would require an in-depth introduction to RouteFlow that is beyond the scope of this work.

NOX-MT) because it is built to provide only minimal OpenFlow support to controllers, applications and switches, in a way that is easy to support newer OpenFlow versions.

Beacon: The work on Beacon differs from ours mostly in its purpose: Beacon is concerned with implementing a high-performance, modular and fully-featured OpenFlow controller. It is not concerned with implementing an OpenFlow framework (i.e. it does not provide any means for building switch agents, nor for building other controllers on top of it). However, it implements a networking IO pattern that is very interesting and can be reused in any OpenFlow framework: the multithreaded IO/event handling architecture. While this design is not exclusive to Beacon (in fact, it is a simple combination of existing concepts in a very traditional way), the experience with Beacon provides an example of this archictectural model being applied to OpenFlow controllers, and the advantages and possible shortcomings of it. libfluid features a very similar IO/event handling architecture, but it does not intend to go as far as providing a complete and modular OpenFlow controller.

5.3.2 Switch agents, frameworks and messaging libraries

tinyNBI: tinyNBI provides a really interesting approach to an OpenFlow framework, hiding from developers the complexity of the protocol and its several versions and ambiguities, while also allowing for a flexible controller and application architecture. Because of its different purposes, it cannot be directly compared to libfluid, but only cited as an alternative (and perfectly valid) approach to the problem of writing OpenFlow controllers.

With that in mind, and considering tinyNBI only as an OpenFlow framework for controllers, one of its disadvantages is that applications will be highly dependent on the high-level abstractions provided, thus making them harder to port to other environments, since the abstractions are more than just protocol-level features. In addition to that, the path for optimizing applications for performance, tuning behavior or supporting additional features is not clearly defined. tinyNBI does not address the development of switch agents, so we cannot compare it to our work in that aspect.

Trema: While Trema and libfluid share some similar goals, Trema's design is different. It is limited to specific OpenFlow versions for now, since it depends on the implementation of the messaging library. It also adds Ruby support as a first-class citizen, which requires adaptations to the code that make it difficult to add new features. The Edge version of Trema, still in development, seems to add multithreading support, but its design is still not clear. Finally, the task of replicating the network simulation tools available in Trema is a good candidate for future work on libfluid.

Indigo: Indigo and libfluid have a different purposes, which only overlap slightly.

Indigo aims to provide an OpenFlow framework for switches only, including a hardware abstraction layer that is meant to ease the work of hardware developers. libfluid can also work in switches, but it only goes as far as providing OpenFlow support, without getting to the hardware level. Making libfluid be more like Indigo would require additional code, possibly in the form of an additional software library.

ROFL: ROFL shares the most similarities with lifluid. First of all, it is written in C/C++ and has been ported to other architectures, in the form of another related project (xdpd, an extensible datapath framework (Berlin Institute for Software Defined Networks (BISDN) GmbH, 2014)). ROFL enables both controller and networking devices agents to be built under a common code base. It also supports several OpenFlow protocol versions. There are however a few key aspects in which ROFL and libfluid differ.

ROFL-common has been ported to other CPU architectures, but it does not support other programming languages. While it is possible to add that support later, its larger public API means that this might be a considerable challenge.

ROFL-common also mixes the messaging concerns and IO/event handling: code for building/parsing messages is used when performing basic, unchanging protocol primitives such as handshaking and version negotiation. This means that when adding support for new OpenFlow versions, ROFL-common developers have to add support for the newer messages in both the messaging parts of the library and in the core OpenFlow support mechanisms. In libfluid, these concerns are separated: `libfluid_base` handles all the fundamental OpenFlow support in all its versions (existing or future), and any other messaging library can be used for building/parsing messages to/from the wire-format (including parts of ROFL-common itself). This gives a little more flexibility to developers, allowing them to choose their own messaging libraries without the overhead of a mandatory default.

Another key difference is protocol behavior configurability: ROFL-common defines some sane defaults for values in a hard-coded manner. As an example, we have the liveness check interval, which defines the interval between two OpenFlow *echo* messages used to check the status of the other peer. In ROFL-common, this interval is hard-coded in the library. While it is possible to change this, this is not easily exposed to the developer. libfluid allows developers to more easily customize protocol implementation features at runtime via the use of `OFServerSettings`, allowing for more flexibility and easing the development of workarounds for incompatible protocol implementations.

Finally, like most existing OpenFlow frameworks, ROFL-common reimplements all of the core IO/event loop mechanism. This means that the framework developers have to worry about sockets and event demultiplexing at OS level. This is a solved problem, and many third-party libraries (e.g.: libevent, libuv, libev, Java's Netty) implement these functionalities, exposing them in a portable and very easy to user manner, freeing the

OpenFlow framework developer from having to worry about these issues.

OpenFlowJ, libopenflow, loxigen and others: Our goal was to build an independent, minimal IO/event handling framework for OpenFlow. If our work is successful in its goals, any of these libraries can be used in conjunction with our work, but we are not dependent on any of these libraries or similar ones. In a certain way, we are trying to create an OpenFlow IO/event handling framework that is as modular as these messaging libraries are, so that it can be reused in several projects.

In this text, we have shown that libfluid works with both `libfluid_msg` and Loxigen-generated bindings, and nothing prevents it from working with other message building/parsing libraries.

6 Concluding remarks

In this dissertation, we highlighted the issues that are common to OpenFlow protocol implementations, then designed a software architecture for a minimal framework that solves these issues and finally implemented and evaluated the architecture.

In Chapter 3 we outlined the issues in existing solutions, which were very broad in their definitions, but gave us a sense of where to go when defining a set of requirements for libfluid. Using concepts from the field of software engineering, we designed a software architecture and showed how it can be used to fulfill the requirements we identified. Finally, we detailed the code implementation and evaluated it in Chapters 4 and 5.

The evaluation showed us that libfluid is up to par with the state-of-the art regarding features and performance, and that it adds benefits in terms of portability to different computer platforms and programming languages and flexibility for different use cases (controllers, switch agents and standalone applications). We have also showed that libfluid can be integrated with existing SDN applications, enriching the ecosystem.

To our knowledge, we have also introduced a few unique approaches to implementing OpenFlow frameworks, namely: (a) the clear-cut division between the connectivity and messaging responsibilities, making it easier to support new OpenFlow versions; (b) the configurable parameters for fine-tuning the framework; (c) a very small code base that can be easily ported to other programming languages and platforms.

Our key contribution is the definition and implementation of an OpenFlow framework with a very minimalistic API that can be reused for different purposes. With libfluid we won the ONF OpenFlow Driver Competition (Open Networking Foundation, 2014a) and the best paper award at the Tools Session of the 32nd Brazilian Symposium on Computer Networks and Distributed Systems (VIDAL; ROTHENBERG; VERDI, 2014). More recently, we have started to receive questions and contributions from the community around the world (libfluid project, 2015) (libfluid community, 2015).

6.1 Future work

libfluid continues to be developed, and we plan to keep on improving it. Some tasks that remain to be done are more mundane, such as implementing better support for logging and writing unit tests. Others require more effort, such as integrating the `OFClient` code into the main implementation of libfluid (there is a patch from the community implementing most of this integration).

In a more exploratory line of work, we want to see how libfluid can be used to implement hardware switch agents for NetFPGAs. Building plug-ins for the southbound interfaces of controllers such as OpenDayLight (which should support not only OpenFlow, but other protocols as well) is also a possible use case for libfluid. Conducting benchmarks in real-world scenarios, especially in real SDN networks, is another important task that remains to be done.

Finally, we hope to see libfluid used in controllers, standalone applications and switch agents and see how the needs of users will change some of the ideas behind our work.

6.2 Publications and awards

The **main publication** of our work is the following paper in the tools session of a conference:

> VIDAL, A.; ROTHENBERG, C. E.; VERDI, F. L. The libfluid OpenFlow Driver Implementation. In: *Proc. 32nd Brazilian Symposium on Computer Networks and Distributed Systems (SBRC)*. [S.l.: s.n.], 2014. p. 1029–1036.

The paper won the **best paper award** in the tools session.

We also won a 50,000 USD prize for building the **best OpenFlow driver (framework) in a competition promoted by the Open Networking Foundation**, the organization that oversees the development of the OpenFlow protocol:

> Open Networking Foundation. *Open Networking Foundation Announces "OpenFlow Driver" Contest Winner*. 2014. Open Networking Foundation's website. Available at: <https://www.opennetworking.org/news-and-events/press-releases/1431>. Access date: Feb 22nd, 2015.

libfluid was made an **open source project** after winning the competition.

Another publication was made during this Master's program, but it is not directly related to the work on libfluid:

> VIDAL, A. et al. Building upon RouteFlow: a SDN development experience. In: *Proc. 31st Brazilian Symposium on Computer Networks and Distributed Systems (SBRC)*. [S.l.: s.n.], 2013. v. 2013.

Bibliography

Android Open Source Project. *Android NDK*. 2015. Android Developer Documentation. Available at: <https://developer.android.com/tools/sdk/ndk/index.html>. Access date: Feb 22nd, 2015. Cited in page 94.

APPELMAN, M.; BOER, M. de. *Performance Analysis of OpenFlow Hardware*. [S.l.], 2012. Cited in page 82.

BEAZLEY, D. M. et al. SWIG: An easy to use tool for integrating scripting languages with C and C++. In: *Proceedings of the 4th USENIX Tcl/Tk workshop*. [S.l.: s.n.], 1996. p. 129–139. Cited in page 94.

Berlin Institute for Software Defined Networks (BISDN) GmbH. *xdpd*. 2014. xdpd website. Available at: <http://www.xdpd.org/>. Access date: Nov 16th, 2014. Cited in page 99.

BERMAN, M. et al. GENI: a federated testbed for innovative network experiments. *Computer Networks*, Elsevier, v. 61, p. 5–23, 2014. Cited in page 28.

BLECHMANN, T. *Class template spsc_queue*. 2015. Next The Boost C++ Libraries BoostBook Documentation Subset. Available at: <http://www.boost.org/doc/libs/1_55_0/doc/html/boost/lockfree/spsc_queue.html>. Access date: Feb 22nd, 2015. Cited in page 88.

CAI ALAN L. COX, T. S. E. N. Z. *Maestro: Balancing Fairness, Latency and Throughput in the OpenFlow Control Plane - Rice University Technical Report TR11-07*. [S.l.], 2011. 2 citations in pages 44 and 83.

CASEY, C. J.; SUTTON, A.; SPRINTSON, A. tinyNBI: Distilling an API from essential OpenFlow abstractions. *arXiv preprint arXiv:1403.6644*, 2014. Cited in page 39.

CORBET, J.; RUBINI, A.; KROAH-HARTMAN, G. *Linux Device Drivers*. O'Reilly Media, 2005. ISBN 9780596555382. Available at: <https://books.google.com.br/books?id=M7RHMACEkg4C>. Cited in page 43.

CPqD. *nox13oflib*. 2013. nox13oflib GitHub repository. Available at: <https://github.com/CPqD/nox13oflib>. Access date: Nov 16th, 2014. Cited in page 97.

DIERKS, T. *The Transport Layer Security (TLS) Protocol Version 1.2*. 2008. RFC 5246. Cited in page 70.

ERICKSON, D. The Beacon OpenFlow Controller. In: ACM. *HotSDN*. [S.l.], 2013. 3 citations in pages 38, 60, and 88.

FEAMSTER, N.; REXFORD, J.; ZEGURA, E. The road to SDN. *Queue*, ACM, v. 11, n. 12, p. 20, 2013. 2 citations in pages 25 and 33.

FOWLER, M. *Inversion of Control Containers and the Dependency Injection pattern*. 2004. Martin Fowler's website. Available at: <http://martinfowler.com/articles/injection.html>. Access date: Nov 16th, 2014. 3 citations in pages 31, 55, and 68.

JAIN, S. et al. B4: Experience with a globally-deployed software defined WAN. In: ACM. *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*. [S.l.], 2013. p. 3–14. Cited in page 28.

JOHNSON, R. E. Frameworks=(components+ patterns). *Communications of the ACM*, ACM, v. 40, n. 10, p. 39–42, 1997. 2 citations in pages 52 and 94.

Joyent, Inc. *Node.js*. 2014. NodeJS website. Available at: <http://nodejs.org/>. Access date: Nov 16th, 2014. Cited in page 86.

KEGEL, D. *The C10K problem*. 2011. Dan Kegel's Web Hostel. Available at: <http://www.kegel.com/c10k.html>. Access date: Nov 16th, 2014. Cited in page 85.

KOPONEN, T. et al. Onix: A Distributed Control Platform for Large-scale Production Networks. In: *OSDI*. [S.l.: s.n.], 2010. v. 10, p. 1–6. Cited in page 36.

KREUTZ, D. et al. Software-defined networking: A comprehensive survey. *proceedings of the IEEE*, IEEE, v. 103, n. 1, p. 14–76, 2015. 3 citations in pages 26, 28, and 36.

LANTZ, B.; HELLER, B.; MCKEOWN, N. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. New York, NY, USA: ACM, 2010. (Hotnets-IX), p. 19:1–19:6. ISBN 978-1-4503-0409-2. Available at: <http://doi.acm.org/10.1145/1868447.1868466>. Cited in page 76.

LEHMANN, M. *libev*. 2015. libev website. Available at: <http://software.schmorp.de/pkg/libev.html>. Access date: Feb 22nd, 2015. Cited in page 59.

libfluid community. *libfluid pull requests*. 2015. libfluid GitHub repository. Available at: <https://github.com/OpenNetworkingFoundation/libfluid_base/pulls>. Access date: Feb 22nd, 2015. Cited in page 101.

libfluid project. *libfluid*. 2014. libfluid's website. Available at: <http://opennetworkingfoundation.github.io/libfluid/>. Access date: Nov 16th, 2014. 2 citations in pages 59 and 94.

libfluid project. *libfluid mailing list*. 2015. Google Groups. Available at: <https://groups.google.com/forum/#!forum/libfluid>. Access date: Feb 22nd, 2015. Cited in page 101.

libpcap project. *libpcap*. 2015. TCPDUMP/LIBPCAP public repository. Available at: <http://www.tcpdump.org/>. Access date: Feb 22nd, 2015. 2 citations in pages 86 and 93.

libuv project. *libuv*. 2015. libuv repository. Available at: <https://github.com/libuv/libuv>. Access date: Feb 22nd, 2015. Cited in page 59.

MATHEWSON, N. *Fast portable non-blocking network programming with Libevent*. 2012. Libevent's website. Available at: <http://www.wangafu.net/~nickm/libevent-book/>. Access date: Nov 16th, 2014. Cited in page 59.

MCCONNELL, S. *Code Complete, 2nd Edition*. [S.l.]: Wiley India Pvt. Limited, 2004. ISBN 9789350041246. Cited in page 48.

MCKEOWN, N. et al. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, ACM, v. 38, n. 2, p. 69–74, 2008. 2 citations in pages 32 and 34.

NASCIMENTO, M. R. et al. Virtual Routers as a Service: The RouteFlow Approach Leveraging Software-Defined Networks. In: ACM. *Proceedings of the 6th International Conference on Future Internet Technologies*. [S.l.], 2011. p. 34–37. Cited in page 96.

NETHERCOTE, N.; SEWARD, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In: ACM. *ACM Sigplan notices*. [S.l.], 2007. v. 42, n. 6, p. 89–100. Cited in page 76.

Open Compute Project. *Networking*. 2014. Open Compute Project website. Available at: <http://www.opencompute.org/projects/networking/>. Access date: Nov 16th, 2014. Cited in page 27.

Open Networking Foundation. *Open Networking Foundation Announces "OpenFlow Driver" Contest Winner*. 2014. Open Networking Foundation's website. Available at: <https://www.opennetworking.org/news-and-events/press-releases/1431>. Access date: Feb 22nd, 2015. 2 citations in pages 101 and 102.

Open Networking Foundation. *OpenFlow Switch Specification*. 2014. Open Networking Foundation website. Available at: <https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.3.4.pdf>. Access date: Nov 16th, 2014. 3 citations in pages 28, 53, and 54.

Open Networking Foundation. *Open Networking Foundation*. 2015. Open Networking Foundation website. Available at: <https://www.opennetworking.org/>. Access date: Feb 22nd, 2015. Cited in page 36.

OpenDaylight project. *OpenDaylight*. 2015. OpenDaylight website. Available at: <http://www.opendaylight.org/>. Access date: Feb 22nd, 2015. Cited in page 36.

OpenSSL Project. *OpenSSL*. OpenSSL's website. Available at: <https://www.openssl.org/>. Cited in page 70.

Oracle. *Java Native Interface*. 2015. Java SE Documentation. Available at: <http://docs.oracle.com/javase/7/docs/technotes/guides/jni/>. Access date: Feb 22nd, 2015. Cited in page 94.

PEPELNJAK, I. *Management, Control and Data Planes in Network Devices and Systems*. 2013. ipSpace blog. Available at: <http://blog.ipspace.net/2013/08/management-control-and-data-planes-in.html>. Access date: Feb 22nd, 2015. Cited in page 25.

PFAFF, B. et al. Extending networking into the virtualization layer. In: *Hotnets*. [S.l.: s.n.], 2009. 2 citations in pages 36 and 81.

Project Floodlight. *Indigo*. 2014. Project Floodlight website. Available at: <http://www.projectfloodlight.org/indigo/>. Access date: Nov 16th, 2014. 2 citations in pages 40 and 72.

SALLENT, S. et al. FIBRE project: Brazil and Europe unite forces and testbeds for the Internet of the future. In: *Testbeds and Research Infrastructure. Development of Networks and Communities*. [S.l.]: Springer, 2012. p. 372–372. Cited in page 28.

SANDERS, C. *Practical Packet Analysis, 2nd Edition: Using Wireshark to Solve Real-world Network Problems*. No Starch Press, 2011. (No Starch Press Series). ISBN 9781593272661. Available at: <https://books.google.com.br/books?id=Zl6LBAAAQBAJ>. Cited in page 86.

SCHMIDT, D. C. Using design patterns to develop reusable object-oriented communication software. *Communications of the ACM*, ACM, v. 38, n. 10, p. 65–74, 1995. Cited in page 52.

SHERWOOD, R. *Cbench*. 2013. OpenFlowHub. Available at: <http://www.openflowhub.org/display/floodlightcontroller/Cbench+(New)>. Access date: Nov 16th, 2014. Cited in page 76.

SHERWOOD, R. et al. Can the Production Network Be the Testbed? In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2010. (OSDI'10), p. 1–6. Available at: <http://dl.acm.org/citation.cfm?id=1924943.1924969>. Cited in page 97.

SWIG project. *SWIG-2.0 Documentation*. 2015. SWIG website. Available at: <http://www.swig.org/Doc2.0/SWIGDocumentation.html>. Access date: Feb 22nd, 2015. Cited in page 94.

TOOTOONCHIAN, A. et al. On controller performance in software-defined networks. In: *USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*. [S.l.: s.n.], 2012. v. 54. Cited in page 37.

Trema project. *Trema*. 2014. Trema website. Available at: <http://trema.github.io/trema/>. Access date: Nov 16th, 2014. Cited in page 40.

Trema project. *Trema Edge*. 2014. Trema Edge repository. Available at: <https://github.com/trema/trema-edge>. Access date: Feb 22nd, 2015. Cited in page 40.

VIDAL, A.; ROTHENBERG, C. E.; VERDI, F. L. The libfluid OpenFlow Driver Implementation. In: *Proc. 32nd Brazilian Symposium on Computer Networks and Distributed Systems (SBRC)*. [S.l.: s.n.], 2014. p. 1029–1036. 2 citations in pages 101 and 102.

VIDAL, A. et al. Building upon RouteFlow: a SDN development experience. In: *Proc. 31st Brazilian Symposium on Computer Networks and Distributed Systems (SBRC)*. [S.l.: s.n.], 2013. v. 2013. Cited in page 102.