libfluid

a lightweight OpenFlow framework

MSc. Defense

Allan Vidal [1]

Advisor: Prof. Dr. Fabio Verdi [1]

Co-advisor: Prof. Dr. Christian Rothenberg [2]

[1] Departamento de Computação
Universidade Federal de São Carlos (UFSCar)

[2] Faculdade de Engenharia Elétrica e Computação (FEEC)
Universidade Estadual de Campinas (UNICAMP)

April 8th, 2015

What is SDN?

Software-defined networking (SDN) is an approach aiming to improve network programmability, removing logic and functionality from closed hardware and putting them in the hands of developers.

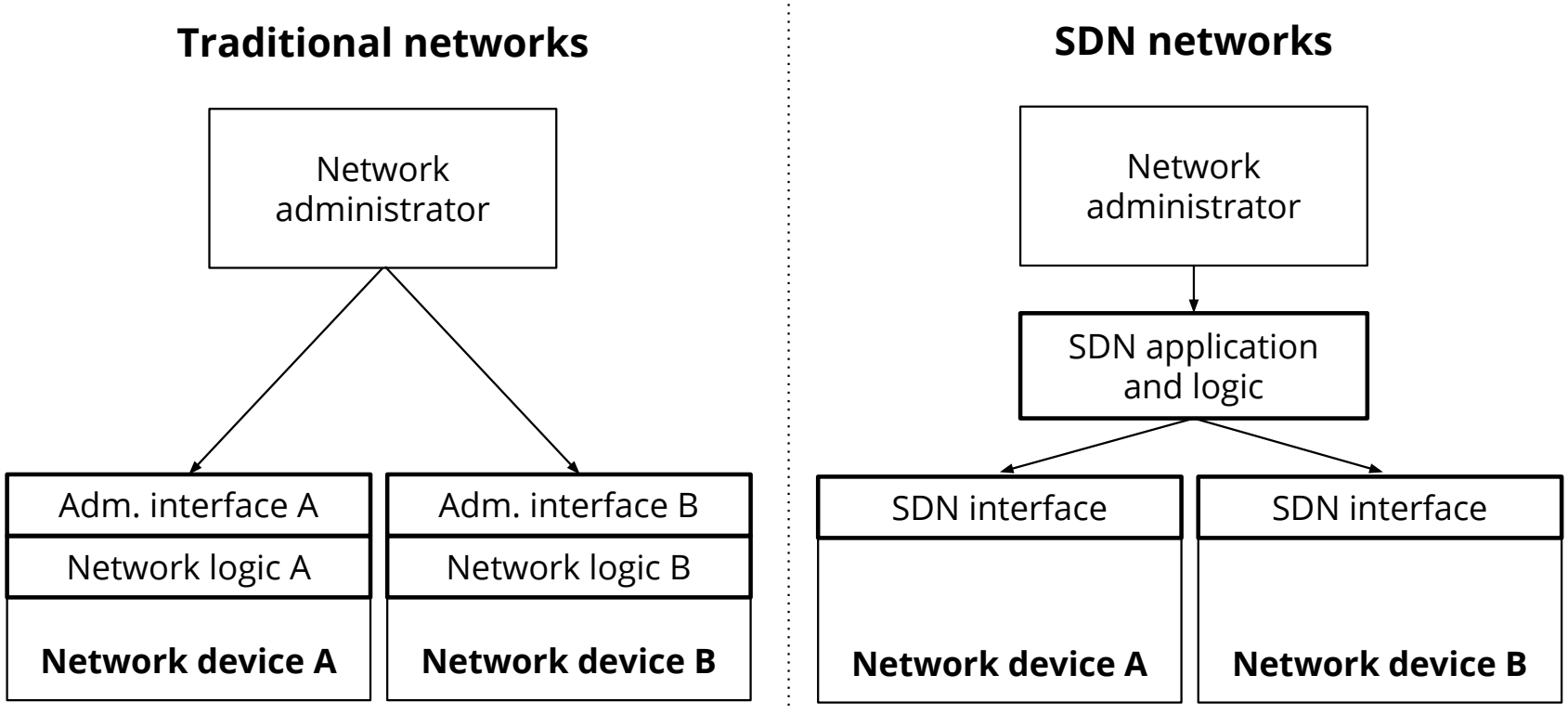
Why SDN?

It emerged as a way to control networks using software, as a solution to several management problems common to networks worldwide:

- Automatic, scalable network configurations
- Global view of the network
- Custom software on commodity hardware

Introduction

Comparison of traditional and SDN networking approaches.



Introduction
The software in software-defined network.

SDN introduces two pieces of software:

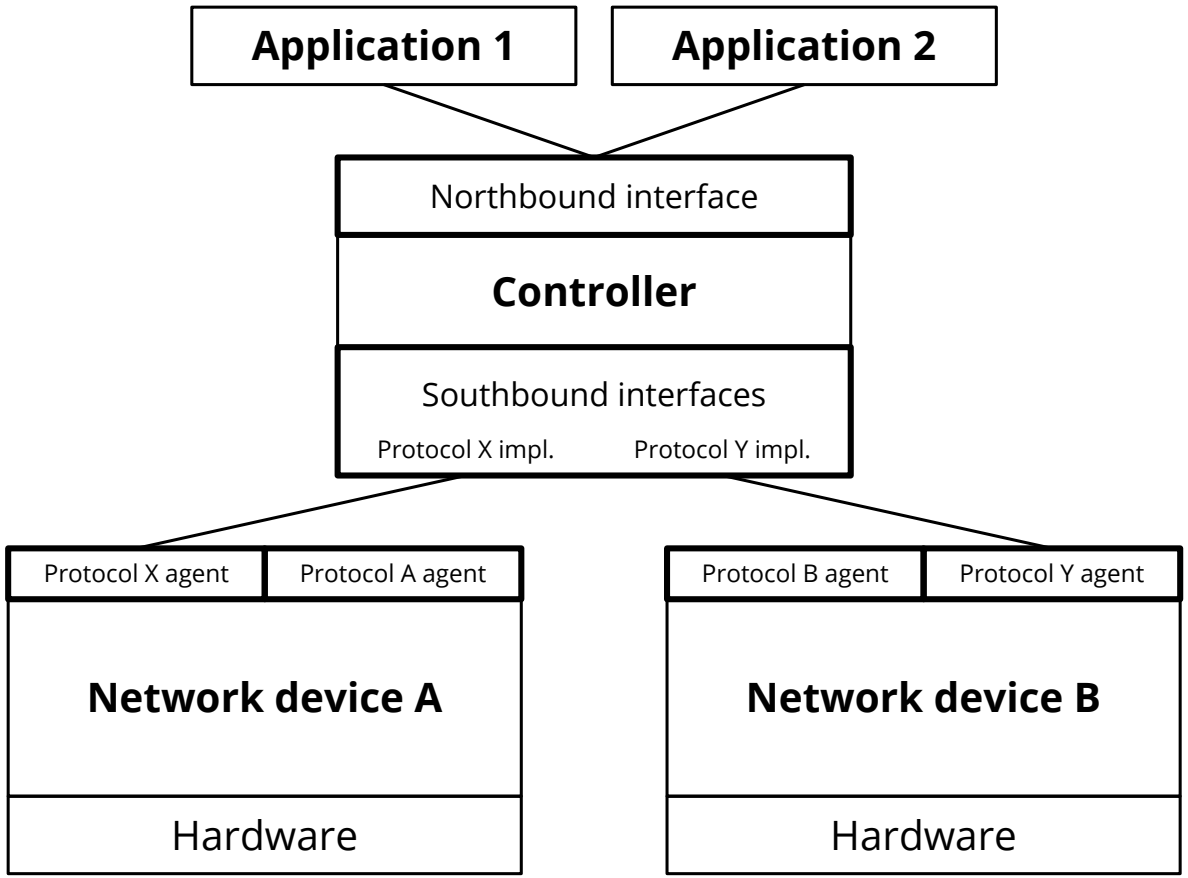
Controller

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.

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.

Introduction
The software in software-defined network: interfaces.



Introduction
OpenFlow

- OpenFlow is one of the most popular approaches to SDN nowadays;
- The OpenFlow specification defines the protocol that switches and controllers use to interact with each other;
- We will detail OpenFlow throughout this presentation, since it is central to our work in many aspects.

Introduction
Motivation

- There is a multitude of software implementing the OpenFlow protocol.
- This popularity, along with a strong industry demand positions OpenFlow as an important cornerstone in the area of SDN.
- We want to analyse implementations of the OpenFlow protocol and propose improvements to the state-of-the-art.

Introduction
Objectives

With our work, we aim to:

1. Highlight the issues that are common to OpenFlow protocol implementations
2. Define a software architecture for a lightweight OpenFlow implementation
3. Implement and evaluate the proposed software architecture

Introduction
Methodology

1. We start with an overview of work in the field of OpenFlow controllers, frameworks and libraries;
2. Based on observations about the related work, we will identify strengths and issues;
3. With the issues listed and defined, we outline a set of general requirements that can solve them;
4. We use that issue list as a guideline for the implementation and evaluation of our work.

Background and related work
Types of related work

Before diving into related work, we will present an overview of the OpenFlow specification (and protocol), which serves as background for the related work and ours.

Based on two definitions (**framework** and **library**) and observations, we divided the related work in four categories:

- Controllers
- Switch agents
- Frameworks
- Messaging libraries

Background and related work
But first, what is a framework?

There is not a single definition, but most agree that **a framework is a skeleton for software**, with the following defining characteristics:

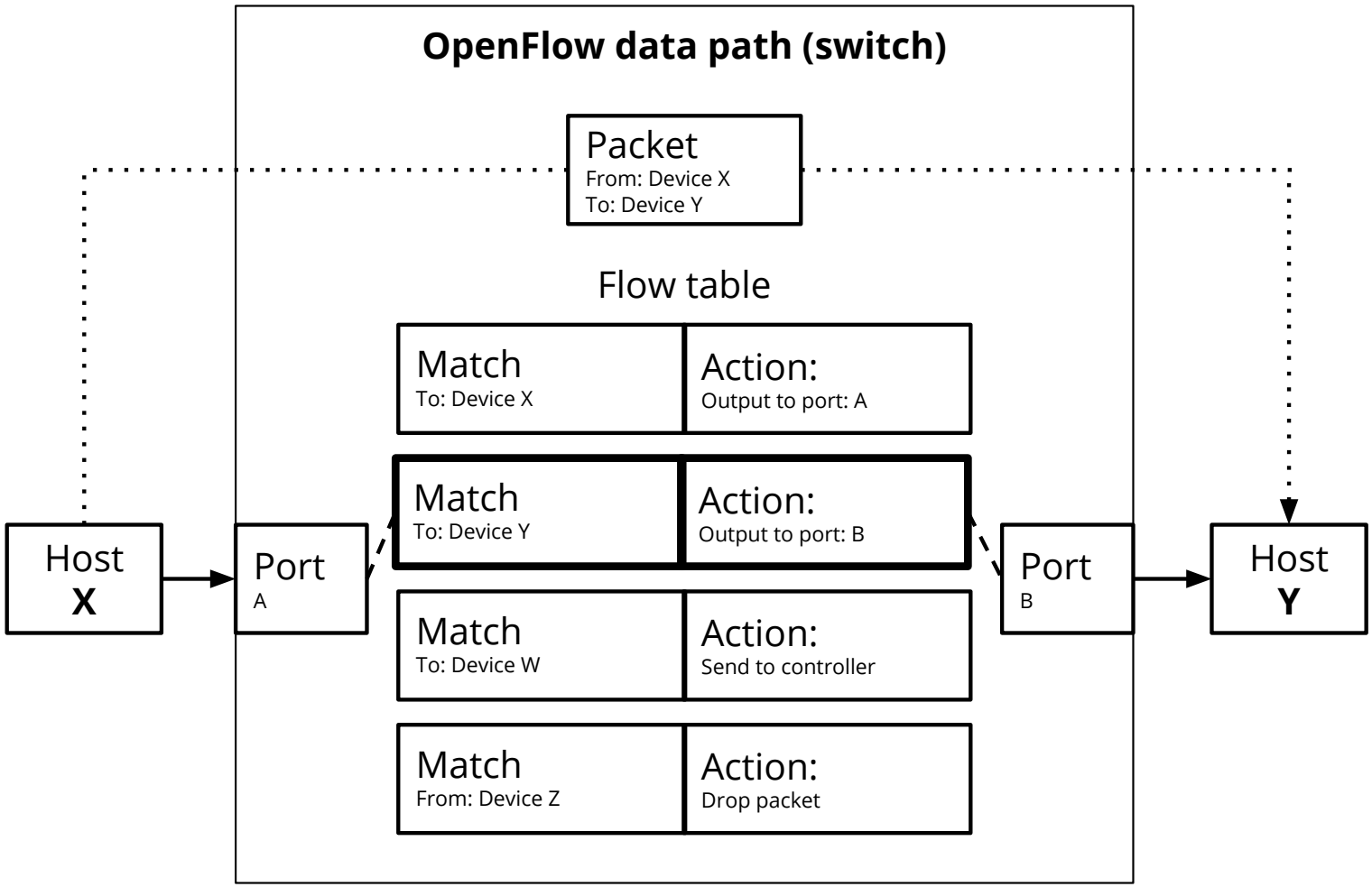
- Inversion of control
- Default behavior
- Extensibility by the user

Background and related work
OpenFlow

- Introducing and testing new features in existing networks had become a challenge;
- To solve this issue, a Stanford networking group published a whitepaper on the OpenFlow network model and its associated protocol, initially as an academic/research initiative;
- OpenFlow is slightly different from previous, similar alternatives in that it seeks to embrace existing technologies, rather than replacing them.

Background and related work

Flow tables and their role in an OpenFlow datapath.



Background and related work
Categorized listing of related work

Controllers

NOX and **Beacon**

Switch agents and controller frameworks

- **tinyNBI**
- **Trema**
- **Indigo**
- **ROFL**

Messaging libraries

OpenFlowJ, **libopenflow**, **loxigen** and others

Proposal
Issues vs. Requirements

Issue	Extracted requirement
There is little reuse between switch agents and controller frameworks	Unified protocol implementation for controllers and switches
Applications are constrained	More flexibility for applications
There is no lightweight and portable OpenFlow implementation	A lightweight and portable implementation
Implementations mix protocol support and message handling	Independence from messaging libraries and protocol versions
There is no way to build standalone applications	Enable standalone applications
Protocol implementation behavior is not configurable	Configurable protocol options

Proposal
libfluid

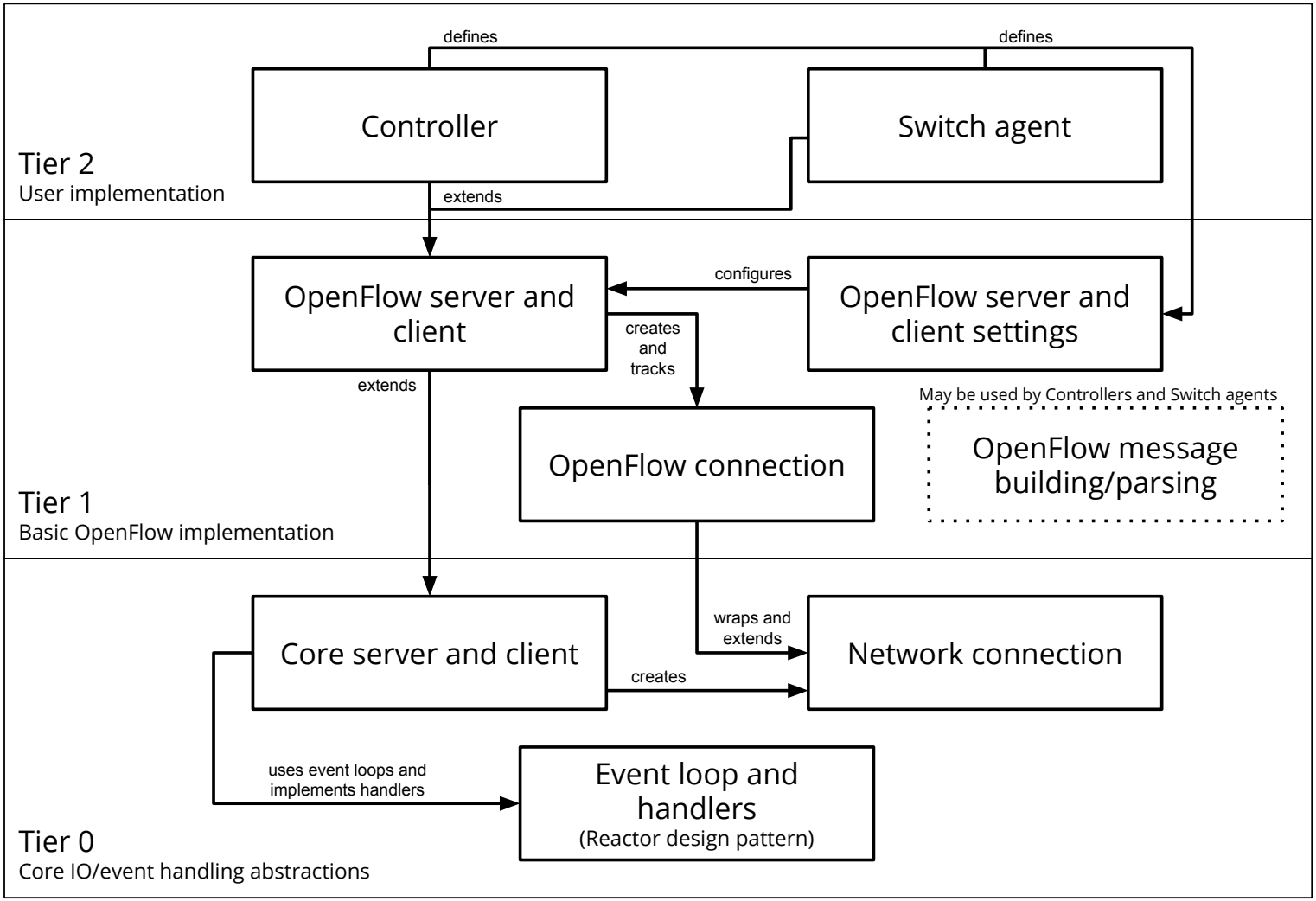
- In order to address the previously outlined requirements, we present **libfluid**.
- Originally started as a project to be submitted to the Open Networking Foundation "OpenFlow Driver Competition".
- It won the competition and is available as an open source project.

Proposal
Architecture

- First, we envisioned the **system**, libfluid, as a framework for the OpenFlow protocol.
- Then we broke down the system into **blocks** that group artifacts with similar or related purposes.
- These blocks were then broken down into **classes** and then into **data and routines**. We will not got into this level of detail in this presentation.

Proposal

Conceptual view of the libfluid architecture showing architectural blocks.



Implementation
Overview

- libfluid is implemented as a C/C++ library, with around 2.2k lines of code.
- This dissertation details **libfluid_base**, which implements the framework.
- We use libevent as the foundation. It is a library/API for monitoring events (IO, signals, timeouts) and dispatching them to handlers via an event loop.

Implementation
A typical stub for a controller written in C++ using libfluid

```
#include <fluid/OFServer.hh>

class Controller : public OFServer {
public:
    Controller(const char* address = "0.0.0.0",
               const int port = 6653) :
        OFServer(address, port, 4, false, OFServerSettings().
                                        supported_version(1).
                                        supported_version(4)) {
        // Controller initialization code
    }

    virtual void message_callback(OFConnection* ofconn,
                                  uint8_t type, void* data, size_t len) {
        // Message handling code
    }

    virtual void connection_callback(OFConnection* ofconn,
                                     OFConnection::Event type) {
        // Connection event handling code
    }
}

Controller c;
c.start();
// Wait for user interruption
c.stop();
```

Evaluation

- Evaluation is based on the previously listed requirements
- Several applications/extensions are built on top of libfluid in order to evaluate it:
  1. Flexible controller
  2. Event handling
  3. Switch agent
  4. Portability
  5. Standalone application
- Results are evaluated in qualitative (does libfluid fulfill the requirement?) and/or quantitative (how does libfluid perform in benchmarking tests?) terms.
- Some tools and metrics were used to conduct the evaluation

Evaluation
Tools and metrics

Tools:

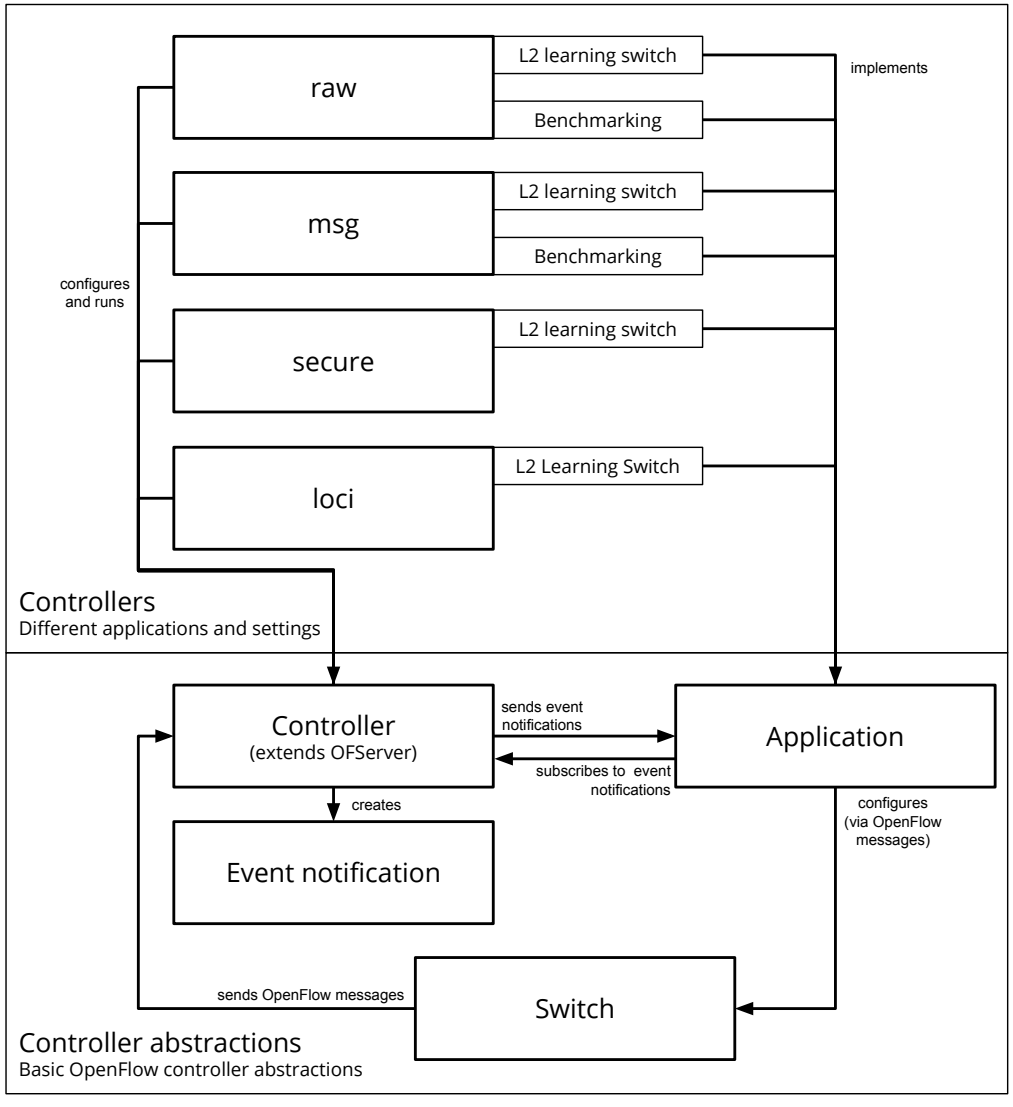
- `cbench`
- Mininet
- Valgrind (Not used directly in the evaluation, but rather as a development aid.)

Metrics (used with cbench):

- Throughput (kflows/ms)
- Latency (μs)
- Fairness (-)

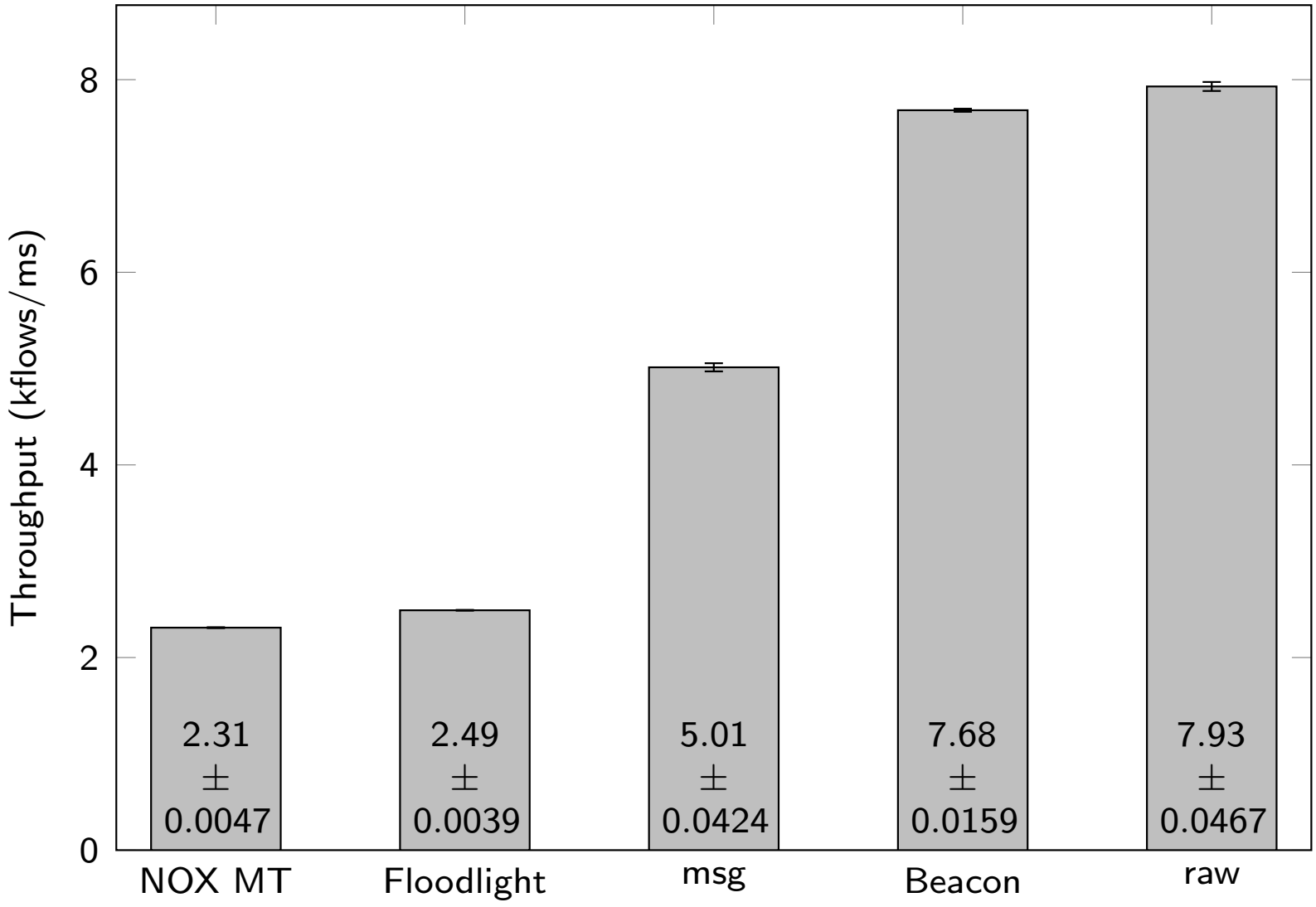
Evaluation - Flexible controller

The flexible controller architecture



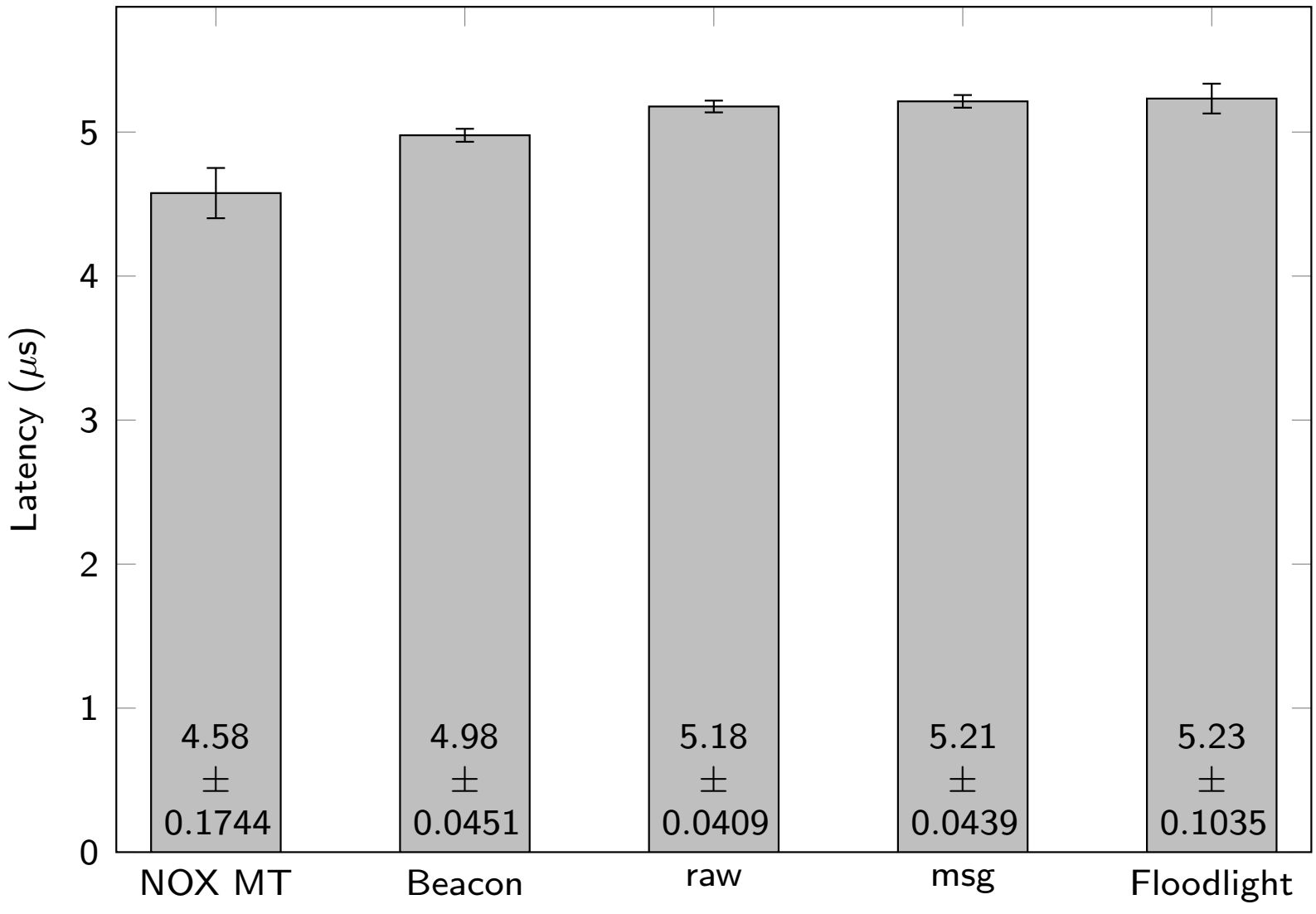
Evaluation - Flexible controller benchmarks

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



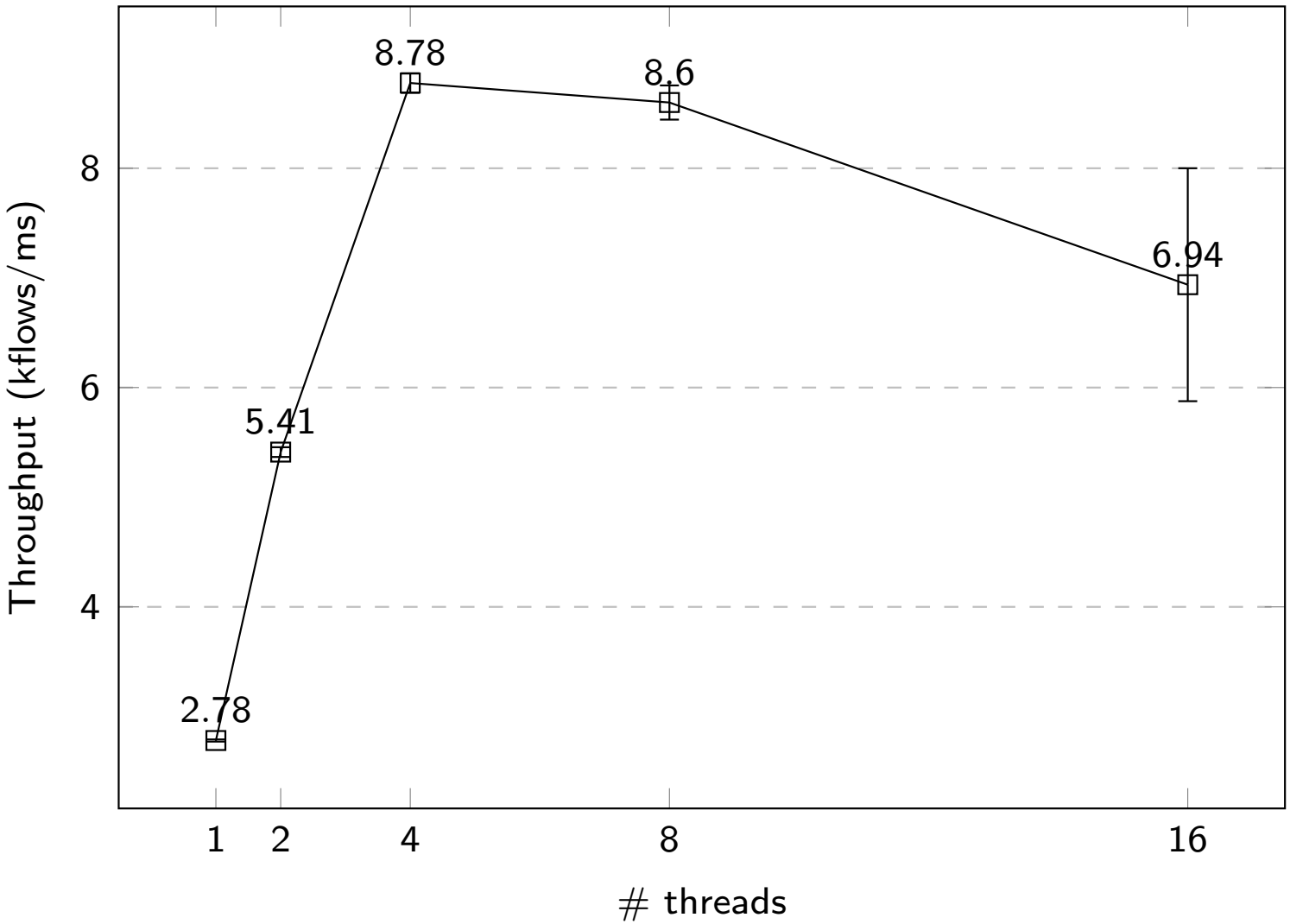
Evaluation - Flexible controller benchmarks

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



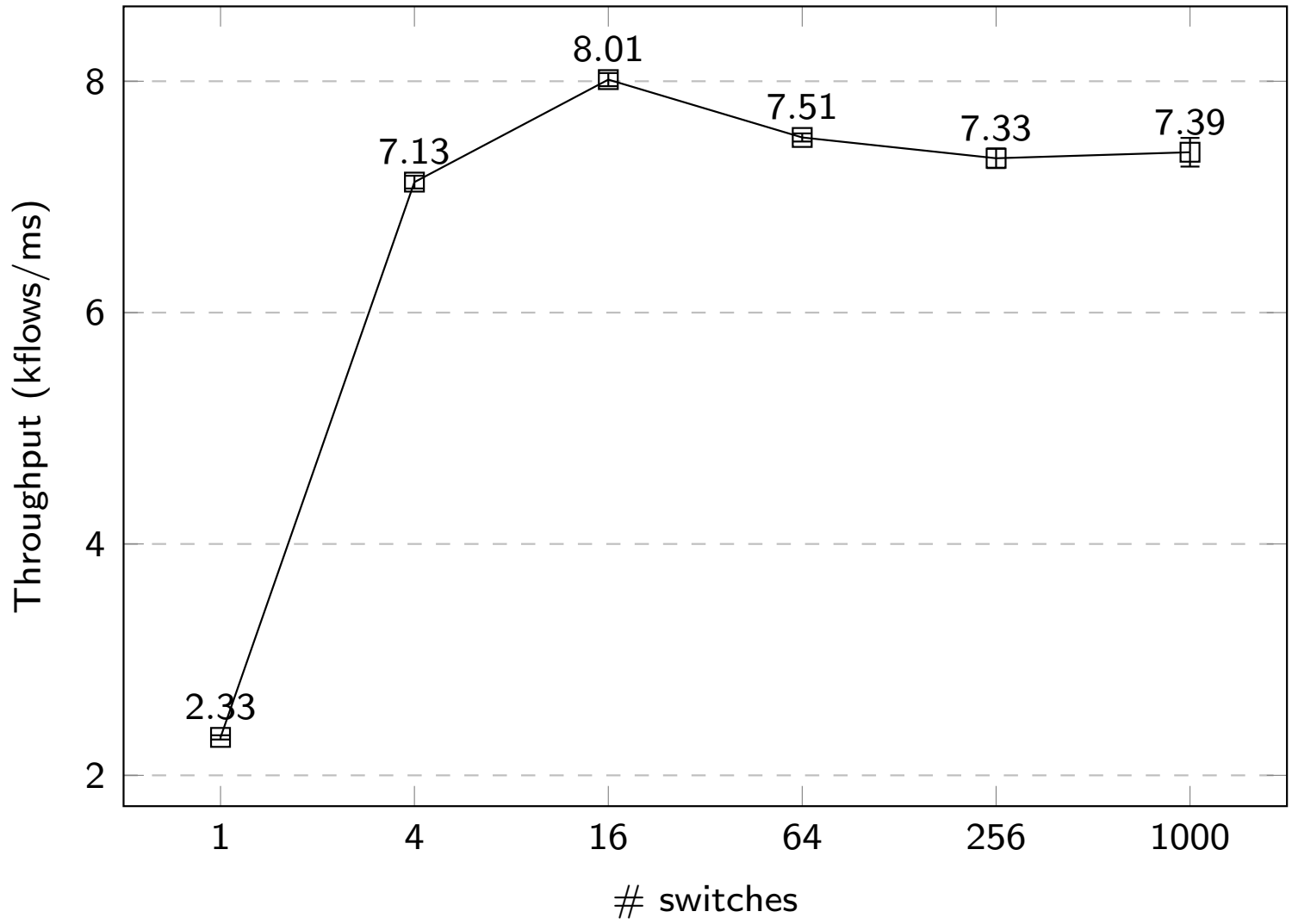
Evaluation - Flexible controller benchmarks

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



Evaluation - Flexible controller benchmarks

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



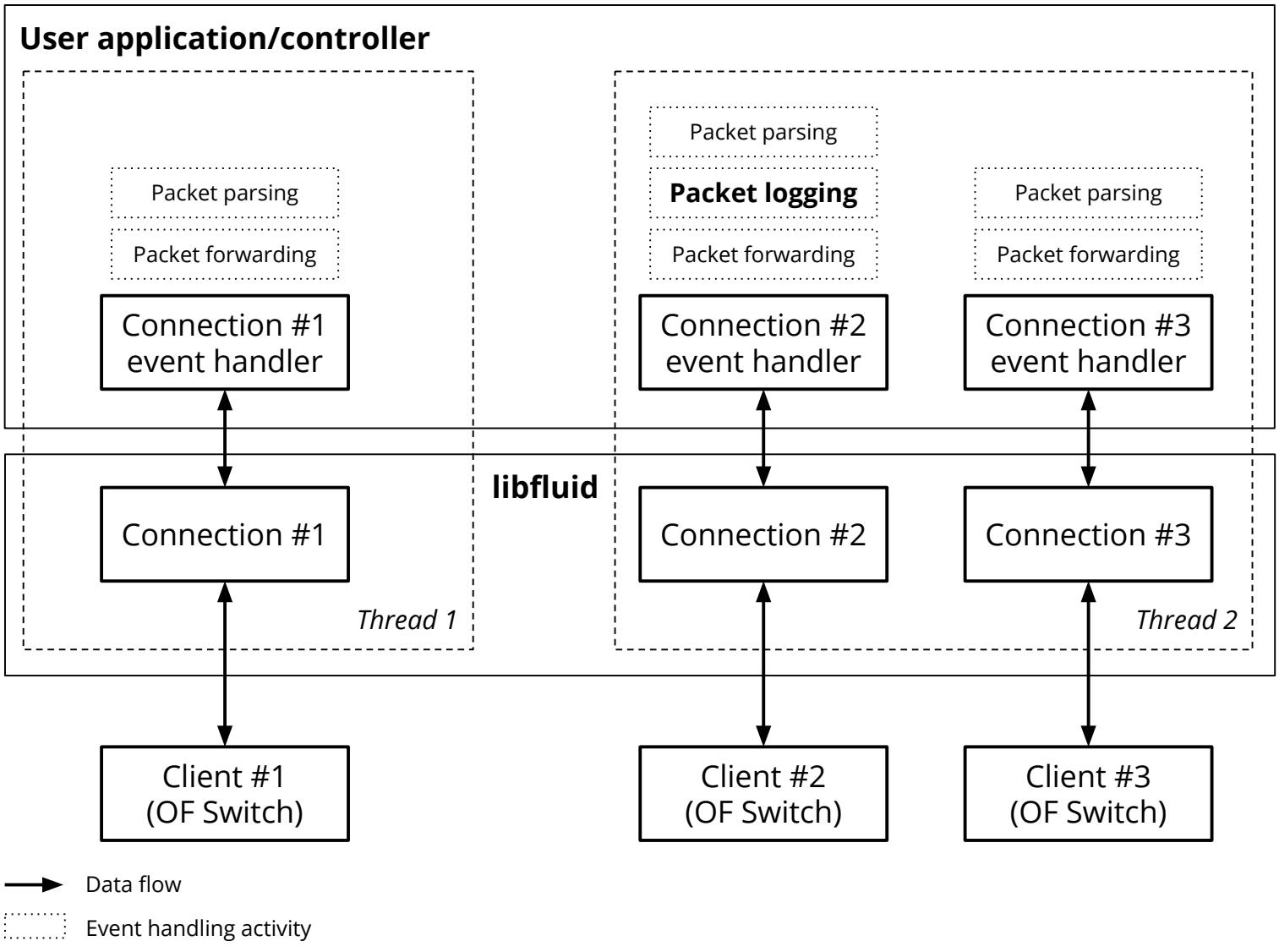
Evaluation - Flexible controller

Requirements vs. Applications

This application shows that libfluid implements the following requirements:

- Unified protocol implementation for controllers and switches
- More flexibility for applications
- Independence from messaging libraries and protocol versions
- Configurable protocol options

Evaluation - Event handling
The problem



Evaluation - Event handling
The problem (cont.)

We built an application that logs all incoming packets of one switch to a PCAP file before forwarding. For other switches, the applications instructs them to forward traffic normally.

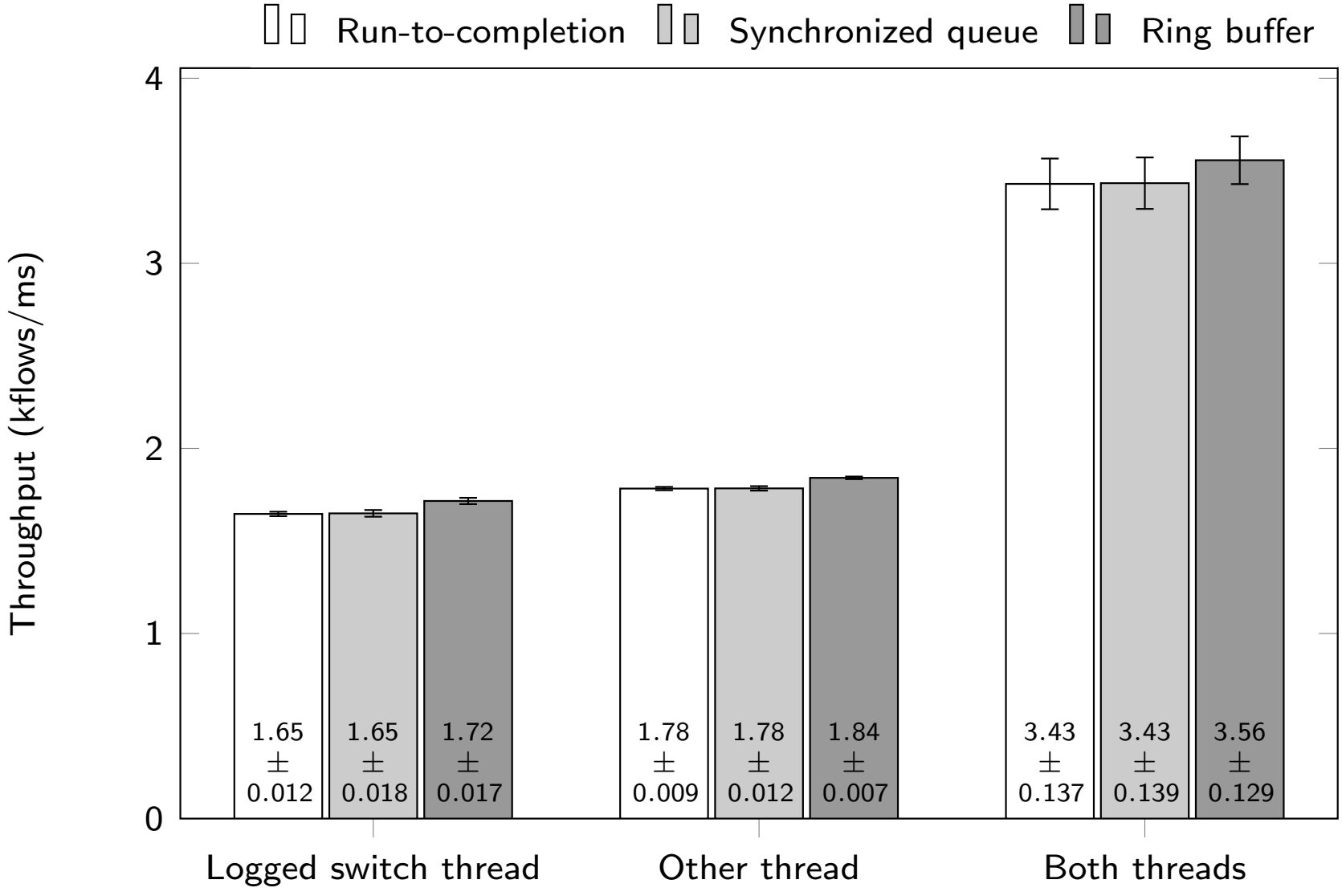
The following event handling methods are implemented by the application:

- Run-to-completion
- Synchronized queue
- Ring buffer

We benchmarked the event handling methods with different threading (1, 2, 4 and 8 threads) and network setups (8, 16, 32, 64 and 128 switches).

Evaluation - Event handling benchmarks

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.



Evaluation - Event handling benchmarks
Overall results

- **Run-to-completion**: tends to provide better results for average latency
- **Synchronized queue**: provides better average throughput for larger numbers of threads (≥ 2)
- **Ring buffer**: provides better average throughput for smaller numbers of threads (≤ 2)

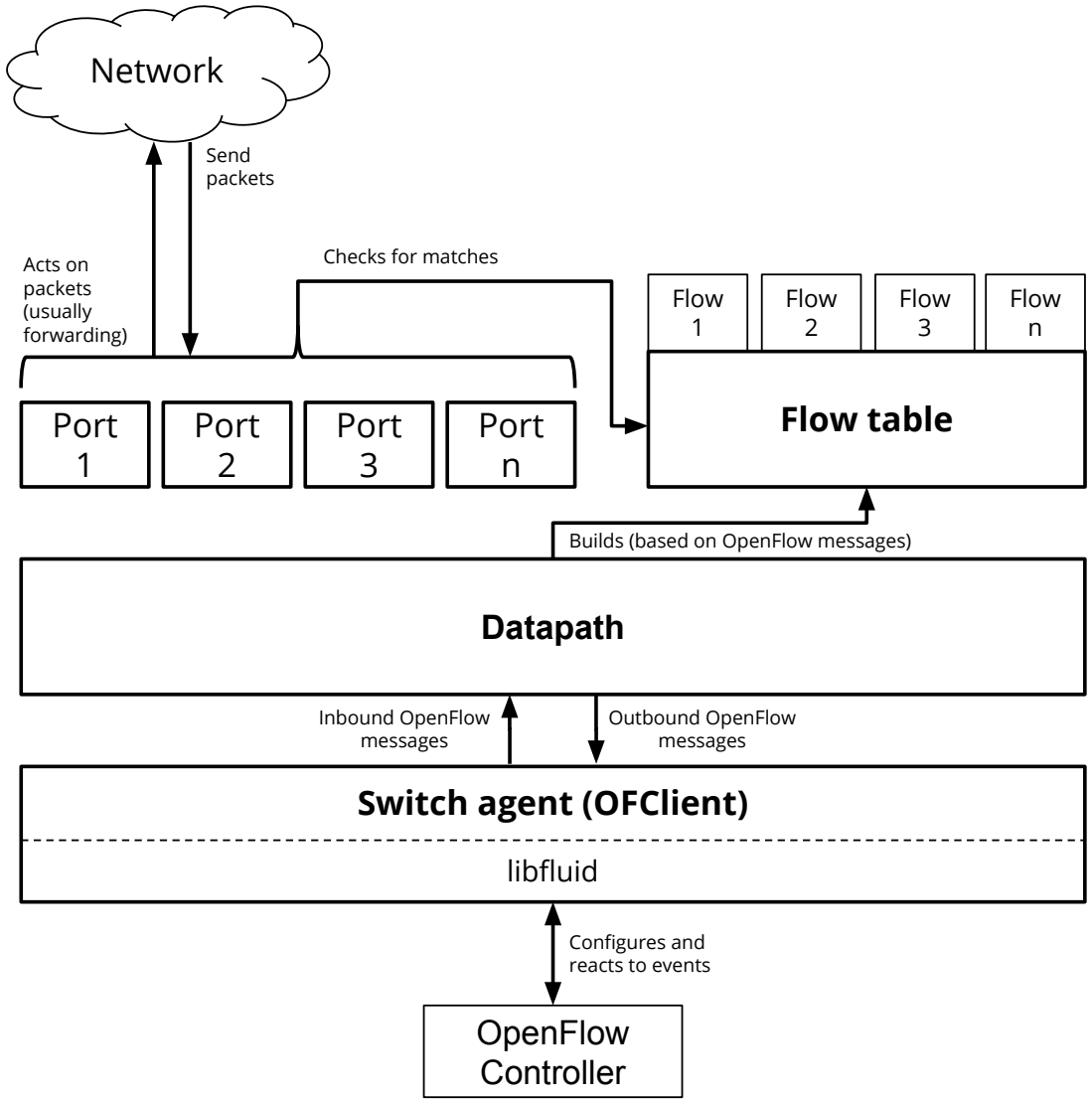
Our conclusion: libfluid allows for the implementation of several event handling methods that can yield different results; always implement and benchmark.

Evaluation - Event handling
Requirements vs. Applications

This application shows that libfluid implements the following requirement:

- More flexibility for applications

Evaluation - Switch agent
The sample switch architecture



Evaluation - Switch agent

Requirements vs. Applications

This application shows that libfluid implements the following requirement:

- Unified protocol implementation for controllers and switches

Evaluation - Portability

The sample switch architecture

Cross-platform build

We built a port of libfluid for the ARM architecture running in Android. No modifications were made to libfluid or its dependencies. The application runs the `L2 learning switch` application of the `msg` controller. It was tested successfully in a smartphone running Android 2.3, with a Mininet instance running accross the network.

Other programming languages

Using SWIG, we built libfluid bindings for Java and Python. We successfully implemented and tested small controllers and applications written with these bindings.

Here we extend the term *portability* to not only refer to porting to other computer platforms, but also to mean the porting of libfluid to other programming languages.

Evaluation - Portability

Requirements vs. Applications

This application shows that libfluid implements the following requirement:

- A lightweight and portable implementation

Evaluation - Standalone application
The sample switch architecture

RouteFlow enables virtual IP routing services on top of a SDN infrastructure. It is composed of three modules:

- RFClient
- RFServer
- RFProxy

We rewrote the RFProxy module using libfluid.

It worked successfully with the rest of the existing RouteFlow modules, and enabled a new way of running RouteFlow (no longer requires an existing, fully-fledged OpenFlow controller).

Evaluation - Standalone application

Requirements vs. Applications

This application shows that libfluid implements the following requirement:

- Enable standalone applications

Evaluation
Comparison to related work

Controllers

NOX and **Beacon**: influenced our work; not directly related to it.

Switch agents and controller frameworks

- **tinyNBI**: different goals, some shared intentions
- **Trema**: similar goals, different implementation
- **Indigo**: narrower scope, specialized, some similar goals
- **ROFL**: very similar goals, different approaches

Messaging libraries

OpenFlowJ, **libopenflow**, **loxigen** and others: inspired us because of their reusability; libfluid can be used with these libraries.

Conclusions

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** and the **best paper award at the Tools Session of SBRC 2014**.

More recently, we have started to receive questions and contributions from the community around the world.

Conclusions
Future work

Mundane tasks: better handling of error conditions, writing unit tests, integrating `OFClient` code into the main implementation of libfluid, etc.

Exploratory lines of work: implement hardware switch agents (NetFPGAs are a good candidate), building plug-ins for the southbound interfaces of controllers such as OpenDayLight.

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.

Bibliography and more information

For the bibliography with the references, see the dissertation:

http://[URLtobeincludedafterpublishing]

For more information on libfluid, including source code and documentation, see the libfluid website:

http://opennetworkingfoundation.github.io/libfluid/