Contributing to

OpenFlow 1.1 Support in Open vSwitch

A report
submitted in partial fulfillment
of the requirements for the degree

of

Bachelor of Computing and Mathematical Sciences

at

The University of Waikato

by

Joe Stringer

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

© 2012 Joe Stringer

Abstract

Open vSwitch is a production-quality software switch. It provides high-performance packet-forwarding in virtualised environments. One of the many protocols it supports is OpenFlow, which allows dynamic configuration of network behaviour. Newer versions of OpenFlow have recently been released, adding support for important protocols such as MPLS and IPv6. However, the support for OpenFlow in Open vSwitch is not in sync with the current standards.

At the end of 2011, a call for assistance was posted on the Open vSwitch mailing-list, requesting that the OpenFlow community contribute towards support for newer versions of the protocol. This report describes the process of responding to this call, contributing to an open source project, and working towards providing network researchers with a more flexible platform for their experiments.

Acknowledgements

I would like to take the time to thank the following people for their guidance and support during this project: Richard Nelson, for his supervision of this project; Josh Bailey and the folks at Google, for introducing me to software-defined networking; Ben Pfaff, for his continued patience with my enquiries and patch submissions; Justin Pettit, Jesse Gross, Isaku Yamahata, and the rest of the Open vSwitch community, for their help and assistance with reviewing and discussing implementation details; Brendan Jones and Shane Alcock, for their input into this report; And all of the great bunch at WAND, for providing many memorable times.

Contents

1 Introduction **1**
1.1 Motivation 2
1.2 Goals 2
1.3 Structure of this Document 3

2 Background **4**
2.1 Software-Defined Networks 4
2.2 OpenFlow 5
2.2.1 Prior OpenFlow Implementations 5
2.3 Open vSwitch 6
2.3.1 Development 7
2.4 OpenFlow Testing Framework 7

3 Development Process **9**
3.1 Learning 9
3.1.1 Identifying work 10
3.1.2 Learning the Architecture 11
3.2 Communication 12
3.2.1 Distributed development 12
3.2.2 Ongoing Development 13
3.3 Implementation 14
3.3.1 Code Quality Tools 14
3.4 Submission 15
3.4.1 Preparing to submit 15
3.4.2 Licensing 16
3.4.3 Code Review 17

4 Existing Architecture **19**
4.1 OpenFlow Architecture 19

4.1.1 Flow Modification 19
4.1.2 Instructions 20
4.1.3 Experimenter Extensions 21
4.1.4 Extensible Matches 21
4.2 Open vSwitch Architecture 22
4.2.1 OpenFlow Provider 23
4.2.2 Testing 23
4.3 Summary 25

5 Implementation 26
5.1 Arbitrary Ethernet Address Masking 26
5.2 Metadata 28
5.2.1 Matching on Metadata 29
5.2.2 Writing Metadata 29
5.3 SCTP Support 31
5.3.1 End-to-End Testing 33
5.4 Evaluation 35
5.4.1 Code Review 35
5.4.2 Testing features 36

6 Conclusion 38
6.1 Impact 38
6.2 Future Work 39

References 40

List of Abbreviations

API Application Programming Interface

CPqD Centro de Pesquisa e Desenvolvimento

FIB Forwarding Information Base

HBO Host Byte Order

IETF Internet Engineering Task Force

IRC Internet Relay Chat

KVM Kernel-based Virtual Machine

lksctp Linux Kernel SCTP

LSR Label-Switched Router

LXC Linux Containers

NBO Network Byte Order

NXM Nicira Extensible Match

OXM OpenFlow Extensible Match

RFC Request For Comments

SCTP Stream Control Transmission Protocol

SDN Software-Defined Networks

VM Virtual Machine

List of Figures

2.1 OFTest: Black-box testing 8

4.1 OpenFlow 1.0 Pipeline 20
4.2 OpenFlow 1.1+ Pipeline 20
4.3 Open vSwitch Components 22
4.4 OpenFlow Provider 23

5.1 Ethernet address masking 27
5.2 SCTP test setup 33

List of Tables

5.1 Development breakdown 35

Chapter 1

Introduction

Computer networks have become critically important to our lives—the internet pervades the lives of billions worldwide. For network researchers, this provides a mixed blessing: On one hand, innovation and experimentation on networks is more relevant than ever; on the other hand, the scale of installed network hardware sets limitations on the impact that researchers can make in the field.

Traditionally, network hardware has been implemented as monolithic boxes with limited configurability. The "Clean Slate" group (Stanford University, 2012) have spent the past five years working towards providing a more open, research-friendly platform for managing network behaviour—OpenFlow (McKeown et al., 2008). This movement has attracted global interest, most notably with Google announcing its use of OpenFlow to increase network utilisation and reduce operating costs (Hölzle, 2012).

Open vSwitch (Pfaff et al., 2009) sits at the meeting point of this open networking movement and the recent trend towards increased computer virtualisation. As a software switch that has been designed from the ground up for performance in data center environments, it too has garnered much interest; during the course of this project, the company behind Open vSwitch, Nicira, was acquired by VMware for US$1.26 billion (Herrod, 2012).

The availability of these technologies and the open background for them provides a compelling field for research and development. Applications which have previously only been possible through the use of proprietary network hardware are becoming increasingly accessible. One such application is that of a Label-Switched Router (LSR)—a particular type of network switch that is used in the core of the internet to simplify traffic management.

1.1 Motivation

The motivation for this project arose from the proof-of-concept OpenFlow-based LSR platform developed by Kempf et al. (2011). The use of custom hardware and unstandardised extensions to OpenFlow in that work presents a barrier for research. The custom extensions used by Kempf et al. have since been included in the official OpenFlow 1.1 standard (OpenFlow.org, 2011), and more recently, Google has presented 'Project W'—a project to make the open source LSR work more accessible.

One of the required components for Project W is Open vSwitch. Late in 2011, a call for assistance was posted on the `openvswitch-dev` mailing-list, seeking support from the community to write code to support OpenFlow 1.1 and above (Pfaff, 2011). *Contributing to OpenFlow 1.1 Support in Open vSwitch* is a response to this call for assistance, as a step towards providing a more accessible open source LSR. The contributions of this project will also act to benefit the networking research community by improving the foremost implementation of OpenFlow.

1.2 Goals

Prior work in this area has required specific hardware and software configurations, which limits the accessibility of the work. To provide lasting value for the research community, this project should focus on contributing work into the upstream codebase rather than providing another proof of concept.

Three OpenFlow 1.1 features were selected for this project to focus on:

- Arbitrary ethernet address masking
- Attaching metadata to a flow
- Adding support for Stream Control Transmission Protocol (SCTP)

For each of these features, it should be verified that the implementation is accurate to the OpenFlow specification. At the minimum, unit tests should be developed to prove that the resulting switch behaviour is as expected. Any additional testing—for example, using the OpenFlow testing framework (Ericsson Research, 2011a)—provides additional assurance of the protocol compliance.

1.3 Structure of this Document

- Chapter 2 introduces the concept of Software-Defined Networks (SDN), and provides background on OpenFlow and Open vSwitch.
- Chapter 3 describes the development cycle of committing to an open source project, with particular reference to Open vSwitch.
- Chapter 4 briefly explains aspects of OpenFlow which are relevant to this project and describes the architecture of Open vSwitch.
- Chapter 5 discusses the implementation of the OpenFlow 1.1 features specifically targeted in this project and evaluates the success of this project.
- Chapter 6 discusses the contributions of this project and possible future additions.

Chapter 2

Background

This chapter explores the work that this project is based upon; What SDN is and how OpenFlow is related. Previous implementations of OpenFlow are explored, which provides some context for the work on Open vSwitch. Finally, this chapter discusses the plans for developing and testing the functionality proposed in this report.

2.1 Software-Defined Networks

The core concept of SDN is quite simple: to be able to configure network behaviour in software. As network hardware stands today, each *switch* or *router* (hereafter *forwarding element*) holds its own configuration and view of global network state. Distributed protocols exist to share this state between forwarding elements, but these operate separately on each forwarding element. It is common for forwarding elements to hold state which is inconsistent with that of its neighbours. Under the SDN paradigm, this state can be held separately from the device. A common case is for a single entity to hold a consistent view of the network and distribute this state to individual forwarding elements. SDN makes a clear abstraction between the layer that makes decisions about how packets should be forwarded (the *controller*) and the layer that forwards packets (the *datapath*).

The *controller* keeps track of connectivity information. It builds a map of the network by gathering information about neighbouring devices. Based on this map, it constructs a set of optimal routes for forwarding traffic. The controller then assembles flow entries—a tuple of a packet classifier (match) and how to forward the packet (action). These flow entries are then written to the

Forwarding Information Base (FIB)—a lookup table used by the datapath.

The *datapath* is the layer that deals with forwarding individual packets. It holds a lookup table for the flow entries, known as the FIB. Each time a packet enters the datapath, it classifies the packet and performs a lookup in the FIB to determine how to forward the packet—for instance, send the packet out a port. This logic is kept simple to minimise the processing time for each packet.

2.2 OpenFlow

OpenFlow is a protocol which follows the SDN paradigm. McKeown et al. describe OpenFlow as an API for the datapath of network hardware, which provides basic building blocks for implementing more complicated functionality in the controller. The canonical use of OpenFlow involves using a simple, commodity switch as the datapath and hosting the controller on a standard PC.

Due to the abstraction between forwarding and control, the network performance of the forwarding element is not degraded by hosting the controller remotely (Bianco et al., 2010). Software applications can be created to generate rules to determine network behaviour, while providing little performance overhead. This suggests that OpenFlow could viably be used in production environments as a replacement for traditional network management.

2.2.1 Prior OpenFlow Implementations

There have been several OpenFlow software switches developed to date. The initial version from Stanford University was developed to provide a reference that other implementations could follow. This switch was updated for each new version of the specification, to understand the new structures and features; however it did not include implementations of all optional features in later versions of the protocol. It was the reference for implementations up to OpenFlow 1.0, but has not been kept in sync with later releases of the specification.

This implementation has been used as a base for several projects: Indigo, a hardware-specific implementation (Big Switch Networks, 2011); an OpenFlow 1.1-only switch (Ericsson Research, 2011b); and OpenFlow 1.2 and 1.3-

only versions (CPqD, 2012). Each of these only targets a single version of the OpenFlow specification, and with the exception of Indigo, have no ability to hook in with hardware. Indigo itself is only targeted for physical switch platforms.

The call for assistance from the Open vSwitch developers outlined plans to support multiple versions of OpenFlow at run-time, unlike previous implementations (Pfaff, 2011). Erlang Solutions have since released an implementation which also does this, however that project focuses on flexibility over performance. As such, the development of further OpenFlow support in Open vSwitch will provide value to the research community above the existing implementations.

2.3 Open vSwitch

Open vSwitch is an open source software switch (datapath) with support for several major protocols including OpenFlow. The codebase is currently over 100,000 lines of C code, contributed by 75 developers. Pfaff et al. (2009) introduced Open vSwitch as an alternative to Linux bridge, for providing high-performance switching between Virtual Machines.

The architecture of Open vSwitch is designed such that hardware vendors can write modules that interface with their own physical switches. This allows Open vSwitch to hardware-accelerate switching functionality, and provide the device with OpenFlow support without requiring the vendor to implement OpenFlow. As of this publication, it is known that at least five commercially available switches already support Open vSwitch in this manner.

Due to the large community backing of Open vSwitch, it can be reasonably expected that the codebase will continue to be maintained well into the future. The review process for submitting code is fairly rigorous, which makes it more difficult to get code accepted upstream; however, this provides a useful guarantee of code quality. As this project seeks to provide lasting benefit to the research community, the assurance regarding code quality makes Open vSwitch a good fit for implementing an open source LSR.

2.3.1 Development

Each version of OpenFlow is incompatible with the others; both the controller and the datapath must communicate using the same version. Previous implementations of OpenFlow 1.1 in the datapath have not been written to speak multiple versions of OpenFlow at run-time, which limits researchers to only using controller software that matches the same version of OpenFlow. While this reduces the development time for new protocol support, it also creates division in the base of available OpenFlow software.

The approach outlined for Open vSwitch from the `openvswitch-dev` mailing-list was to instead support multiple versions of OpenFlow at run-time. The plan for this is to create generic structures that abstract the protocol differences from the core code, then implement the specific protocol handlers separately. For each new version of OpenFlow, a new parser will need to be added and taught how to convert to and from the internal format. The benefit from this approach is that Open vSwitch can be used in a wider range of environments than datapaths that only support a single version.

In addition to this, the plan specifies that newer features should be back-ported to the older protocol, by adding them as experimenter extensions. This approach will allow Open vSwitch to interoperate with a wide range of software and hardware configurations. Section 4.1.3 describes the usage of this OpenFlow feature in this project.

2.4 OpenFlow Testing Framework

Just as there multiple versions of the reference OpenFlow switch have been developed, there have also been multiple versions of a matching test framework, known as OFTest (Big Switch Networks, 2012). This framework is written in Python, and conducts black-box testing of a switch: It surrounds a datapath, controlling all of the inputs and outputs. Figure 2.1 shows this relationship between OFTest and the datapath that it tests. OFTest acts as the controller for the datapath by sending OpenFlow messages to it, and also connects to the ports on that datapath to send packets through it. The behaviour of the switch can then be evaluated against the OpenFlow standard.

Tests are conducted independently by conducting the following:

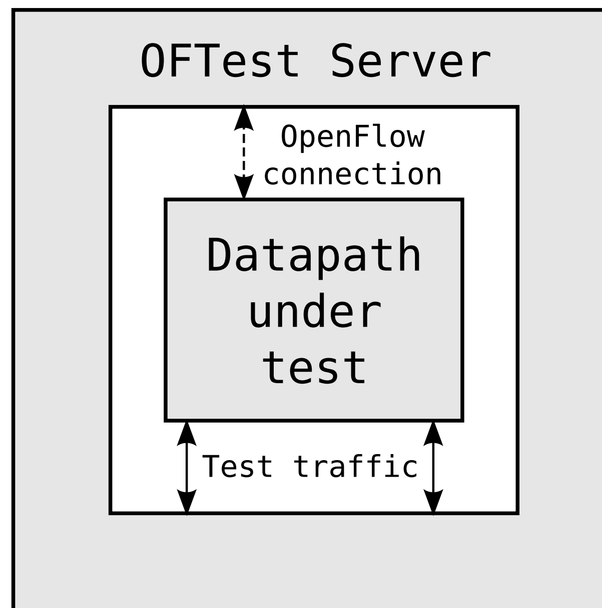

Figure 2.1: OFTest: Black-box testing

- Reset the rules on the test switch
- Install new flow entries into the datapath
- Send packets through the datapath ports
- Observe the behaviour

This provides an external test for protocol compliance—a level of guarantee about interoperability with other OpenFlow implementations. It is aimed at providing OpenFlow developers with a simple ‘plug-and-play’ environment that is easy to set up and use. Section 5.4.2 will discuss the use of OFTest to evaluate this project, and the issues faced in this process.

Chapter 3

Development Process

Contributing code to an open-source codebase requires additional skills on top of the basic skillset of a programmer. While the ability to reason about programs, select appropriate data structures and write quality code are all very useful skills, these need to be supplemented with a social understanding of the project's ecosystem.

This chapter describes the considerations that a potential contributor (hereafter *the contributor*) needs to make when undertaking work on open source software. This will include general commentary on each particular aspect of the development process, followed by examples of how the aspect is managed in Open vSwitch. This will provide insight into the processes involved with a large community consisting of dozens of contributors, and should be widely applicable in any software development community.

The general development process can be broken down into a cycle involving the following activities:

- *Learning* about how to contribute
- *Communicating* plans for contributing
- *Implementing* the functionality
- *Submitting* the work upstream

3.1 Learning

There are a few steps that the contributor must take to learn how to contribute to an open source project. Learning about what modifications are being sought

after, and the desired approach for this implementation is one step. Once it has been established what contributions that the contributor will make, the architecture must be examined and learnt. This provides the basis for the work to be carried out.

3.1.1 Identifying work

This step can be split into two parts. The first of these is negotiating with other developers about what tasks to work on, preventing duplicate efforts; the second is to understand what is expected of the final implementation of the feature. In the case of implementing a standardised protocol, this involves studying the specification to ensure that the work accurately reflects the intended behaviour.

Deciding which feature to work on is the first step in making a contribution. It is recommended that new contributors select something reasonably small at first, so that progress can be made without becoming overwhelmed with the scale of the codebase. With Open vSwitch, the call for assistance provides a useful basis for feature selection. As one becomes more familiar with a codebase, it becomes easier to consider and understand the effects of implementation decisions on other areas of the code.

In any project that has a large number of active developers, there are likely to be several features being worked on at a given time. To avoid duplicate efforts, it is important to make contact with the lead developers and determine whether there is already a developer allocated to the feature. In the case of Open vSwitch, the developer mailing list is the central point for such communication (Open vSwitch, 2012b). The contributor expresses interest in developing a particular feature, and other developers provide feedback about whether the feature is already under development. This may also include some indication as to how they would prefer the work to be carried out.

When a decision has been made with regard to the feature to focus on, there is a process of learning the requisite implementation details of the feature. This may involve searching for literature on the subject—reading papers, specifications and IETF Request For Comments (RFC)s. In the case of implementing OpenFlow features, the primary source of information is the specification distributed by the Open Networking Foundation. Additionally, many of the features make reference to protocols which are defined elsewhere, so it is

important to have an understanding of the operation of these protocols.

3.1.2 Learning the Architecture

Having investigated what modifications should be made to the codebase, the contributor must learn about how to make these changes. Several factors affect the significance of this step. For some idea of the scope, one could consider how many lines of code there are, or the number of files in the codebase. For a more accurate gauge of the effort required for this task, it is worth looking at the quality of documentation—to what degree it exists and how accurate it is.

There is no single accepted format for the documentation of a project. Most codebases have some form of web presence—particularly open-source ones—but even so, the documentation may not be found on the website. In Open vSwitch, the developers have opted to distribute documentation with the codebase, and make these accessible from the main website (Open vSwitch, 2012a). The descriptions of the abstraction layers are found in the `PORTING` file, which is targeted towards developers interested in porting the codebase to additional platforms. This is augmented by the descriptions of design decisions made when developing Open vSwitch, which is detailed in the `DESIGN` file. Both of these files are found in the root directory of the codebase.

Even when there are design documents distributed with the project, they may not provide the level of detail required to implement a feature. The `PORTING` guide gives a useful overview of the Open vSwitch architecture, but it does not mention the specifics of how the existing components parse OpenFlow messages or how these messages are implemented. One example is the `meta-flow` structure which assists with parsing OpenFlow matches at run-time. Such structures can be learnt by searching for components which interact with them and investigating the interactions, and through communication with the upstream developers.

Lastly, the nature of working on a fairly new codebase is that the internal data structures have not been fully stabilised yet. Prior to this project, much of the internal representation of data was based directly on the OpenFlow 1.0 structures. Over the course of this project, the internal structures were adapted to allow for differing representations and additional features. Most notably, the internal representation of OpenFlow actions was introduced as a superset

of the OpenFlow 1.1 actions and instructions so that Open vSwitch could dynamically support multiple versions of OpenFlow at runtime. Such evolution in the codebase will affect the implementation of both new and existing features. As such, the contributor needs to follow developments throughout the entire project codebase to determine the effect that they may have on the proposed code modifications.

3.2 Communication

Learning to communicate appropriately is one of the most important aspects of contributing to an open source codebase. Healthy communication can lead to a streamlined development pipeline and less wasted effort. These provide positive reinforcement and improve the pace of community contributions. This is particularly important with a large project such as Open vSwitch, which receives upwards of twenty posts on the developer mailing-list per day. Any of these could affect the implementation of a feature that the contributor is working on.

3.2.1 Distributed development

With any existing development community, new contributors need a support structure to help them to develop effectively on the codebase. This structure provides information about the architecture, insight into design decisions, and commentary on developments. When working with a distributed development team, the ability to use other contributors as a resource is hindered by the geographic spread of the community.

Typically, open-source codebases with a significant number of contributors have a large variation in development locations. Bird and Nagappan (2012) investigated the geographic distribution of two major open-source communities, Firefox and Eclipse. They found that around a third of major contributions were made by developers on different continents for Firefox versions 1.5 and 2.0. Eclipse was far less distributed, but still involved two continents for a majority of the work.

In Open vSwitch, the majority of commits are contributed by Nicira in California. As the software continues to be developed, major contributions have begun coming in from other locations. For the work described later in this document,

the majority was developed in NZST (UTC+12), while corresponding with California in PDT (UTC-7) and Japan JST (UTC+9).

These timezone differences have a considerable effect on how developers can schedule their workflow. While the direct implementation work can be carried out at any time, discussion with developers on the implementation can only occur during set times of the day. The common case for this project involved using the IRC channel for urgent communication in the morning NZST, or using the mailing list for more detailed discussion on implementation or code quality. To set reasonable expectations for what defines a timely response, it is recommended to be aware of what times of the day are business hours for other developers. Upstream developers may take a day or more to respond to messages that are posted to the mailing-list. As such, it is also recommended for the contributor to have additional tasks to work on while waiting for responses.

3.2.2 Ongoing Development

Every development community will have a commonly agreed means for communication. Common forms are the mailing list, bug tracking systems and IRC channels. For Open vSwitch, the primary communication platform is the mailing lists. There is a list specifically for development discussion, which is separate from the general user discussion. Bug reports are also posted on a Debian Linux bug tracker, but there is no public bug tracker specifically set up for Open vSwitch. To ensure that they reach the attention of the appropriate developer, bug reports are expected to be reported to either the developer list or the Debian tracker.

It is important as a contributor to actively watch the development activity to promote co-operation between developers on related features. Every feature described in this report was affected by other activity on the list. Examples include new abstractions introduced in the core codebase, commentary on related features, and general development guidelines. In particular, the `write-metadata` feature—described in Section 5.2.2—was based upon another patchset from the community.

Active participation on the mailing-list also allows for the contributor to provide assistance to other developers. Some of the work submitted from this project influenced other development. For example, the initial `write-metadata`

patch provided a model for others to implement similar features (Pfaff, 2012a). The discussions following the post provided additional insight into the preferred approach for implementation. This is useful to the contributor to improve the understanding of the codebase.

3.3 Implementation

There are several ways to verify the code that the contributor has created is of acceptable quality to be included upstream. Perhaps the most indicative is to write a comprehensive set of tests to verify that the behaviour is as expected. Often, software projects will have additional criteria to keep the code maintainable and find unexpected bugs. The Open vSwitch codebase includes a guide for submitting patches which clearly outlines the quality criteria for code submissions. The minimum expectations are that the code:

- builds correctly,
- does not break any existing tests,
- changes a single feature,
- and updates the user documentation.

This section will cover the expectations for proving the functionality of the implementation, which will be followed by a discussion of its presentation in Section 3.4.

3.3.1 Code Quality Tools

For code to be accepted upstream, it must first meet particular quality criteria. As code submissions can be sent by a programmer of any skill level, it is useful to have tools to verify the quality of the code. This assists the contributor in improving code programming practice, and assists upstream developers by providing independent verification of code correctness. Two tools are used by Open vSwitch developers for this purpose: Sparse and GNU AutoTest.

Sparse is an open-source static code analyser for C. It is a frontend for popular compilers that provides additional information about the use of memory address space and other resources. In Open vSwitch, this program is used to detect some potential bugs such as mixed byte ordering. When dealing with

network packets, the endianness of data on the wire is often different from that of the host—referred to as Network Byte Order (NBO) and Host Byte Order (HBO). Open vSwitch differentiates these types by defining its own types for NBO values (`ovs_beXX`), and using standard UNIX types for HBO (`uint_XX`); however, the base data type of each of these is the integer. Typical compilers will not detect errors where a developer is directly copying values between NBO and HBO variables without swapping the endianness. Sparse is built to highlight these difficult-to-detect mistakes.

AutoTest is a framework for generating platform-specific test scripts. Developers create AutoTest scripts which define general information about a test, how to run the test, and what the expected output is. The Open vSwitch codebase uses AutoTest as the basis for most of its unit tests. When adding functionality to the code, developers are expected to create additional tests to prove the behaviour of the additions is correct. This also allows developers to verify that there have been no regressions when further changes are made to the codebase.

When a feature has been written and appropriate test cases have been created, the contributor must then prepare the work for review. Firstly, it must be checked that the patch compiles successfully against a fresh copy of the codebase. This can reveal build errors where the contributor has neglected to include all of the relevant modifications in the patch. Following this, the code should be regression-tested. Running `make distcheck` in Open vSwitch will perform these two tasks in a single step.

3.4 Submission

There are several considerations for the submission of a patch beyond the code implementation. Broadly, this section will describe them in terms of modifications required by the contributor, licensing considerations and feedback from upstream maintainers.

3.4.1 Preparing to submit

The Open vSwitch codebase includes a style guide which defines the format which is expected for code before it is to be accepted upstream. In practice, this is unlikely to be an issue if the developer follows the existing style from

code surrounding his modifications. Given a sensible patch, the upstream developers are unlikely to decline to review based on style but it is likely to draw attention away from the functionality.

When the patch modifies or introduces new user-facing features, it is also expected that the user documentation is updated. For each of the userspace utilities provided by Open vSwitch, there are Unix-style manual pages describing the features of the utility and the syntax for using it. A submission that does not include or update the manual pages will not be accepted upstream. It is important that the behaviour described by the manual is accurate to the behaviour of the application.

Patches must also be signed off by the contributor, to indicate that the code can be submitted to the project without breaching the licensing terms. It is a public record of the contribution. The usual case is where the contributor has written the code, although it may include cases where code from another source has been included under a compatible license—as described in Section 3.4.2.

Lastly, it is expected that any new patches are based on the latest development code branch. It is good practice to fetch the latest version of the development codebase before applying the candidate patch and performing regression testing. This is because the latest code may modify the components that the candidate patch depends on. This is particularly important for projects which have a high volume of code contributions.

3.4.2 Licensing

Open-source applications are defined as such based on the license under which they are distributed. On occasion, it may be sensible to include another project's code in a patch so as to reduce duplicate efforts. This allows the contributor to take advantage of a stable existing codebase rather than re-implementing a new—and potentially buggy—solution. In this case however, it is essential to be aware of the license restrictions on the codebases involved.

In this project, the primary example of needing to include external code is the CRC32c checksum implementation for userspace support of SCTP. To re-implement a checksum algorithm is a risk-prone undertaking and would require a large amount of effort to write correctly. As such, it is sensible to look for existing implementations that could be included in the Open vSwitch source.

Several open-source kernels include copies of the CRC32c algorithm for use by their own SCTP implementations. Each of these has its own license, some of which are compatible with the Open vSwitch license and some which are not. Ultimately, the FreeBSD implementation was chosen, due to its licensing compatibility with the Apache version 2 license that most of Open vSwitch uses.

3.4.3 Code Review

The review process is the last step in the development cycle. For Open vSwitch, this step involves the contributor posting the candidate patches to the `openvswitch-dev` mailing-list, and receiving feedback from other developers. This step provides an additional assurance of code quality for the codebase.

There are some measures of code quality that cannot be measured using static code analysis or written tests. One such measure is of the effect on the architecture of the system. The core developers of a project are able to provide insight into how a particular work interacts with the larger codebase. As maintainers of the code, it is in their interests to ensure that code follows the intended design of the system. Conversely, if they allow code that breaks down the design or behaves unexpectedly, then this will increase the work required in the future to keep the system bug-free.

For the contributor, the most beneficial aspect of this step is an opportunity to discuss implementation with direct reference to code. Although questions can be asked on IRC or on mailing-lists without a patch submission, the lack of context can introduce ambiguities in communication. When the contributor submits a patch, this is a chance for the upstream developers to educate the contributor on how the work should be carried out.

The review process commonly repeats multiple times until the upstream developers are content with the proposed changes. If the contributor is persistent and willing to respond to feedback, then each round of review will bring the code closer to acceptance upstream.

Kernel code

Open vSwitch also includes a kernel module for Linux in its distribution, which has been included in the main distribution since version 3.3 (Calleja, 2012). The review process for this code is more rigorous—the patch must also satisfy

a further 24 patch submission criteria (Riel, 2006), and will be seen by multiple other developers. As such, features that require work in this area will undergo additional scrutiny from the Open vSwitch developers, to reduce the effort required to later get the code included in Linux. The SCTP feature described in Section 5.3 requires work in this area.

Chapter 4

Existing Architecture

Chapter 2 discussed the concepts underlying this project: SDN, OpenFlow, and the approach to extending Open vSwitch. The overview of these concepts is essential for understanding how this project fits into the larger work, but does not provide enough detail to understand the proposed changes. This chapter describes some of the implementation details that this project relies upon; the inner workings of OpenFlow switches in general, and the way that Open vSwitch reflects these. This provides a basis for the modifications detailed in Chapter 5.

4.1 OpenFlow Architecture

Prior descriptions of forwarding elements in this document refer to the forwarding of individual packets. However, to place one flow entry into the FIB for each packet that passes through the datapath would be inefficient. As such, packets are usually classified into *flows* based on attributes such as the source, destination and protocol being used. All packets with the same values in these limited fields is handled by a single rule.

4.1.1 Flow Modification

In the OpenFlow 1.0 specification, the datapath's FIB is presented as a single logical table. The table has a list of *flow entries* describing the flow to *match* on, and a set of *actions* to perform on packets that match. At the bare minimum, an OpenFlow switch must support actions to either forward or drop the packet; there are also provisions for modifying the packet and queueing. When a packet is received, the headers are parsed and used to find

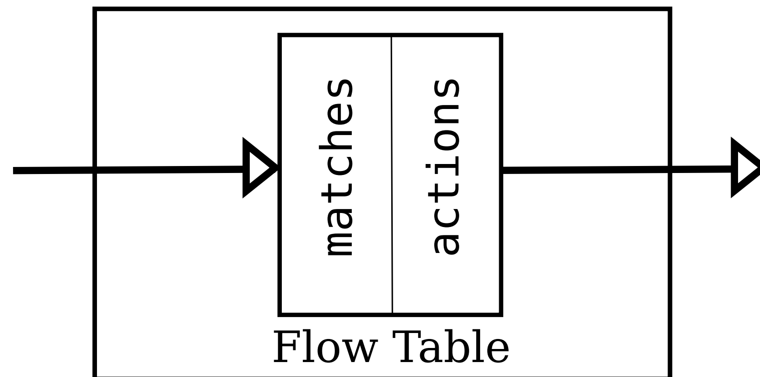

Figure 4.1: OpenFlow 1.0 Pipeline

a matching flow entry. The actions for that flow entry are then immediately applied in order. The default action if there is no flow entry is to forward the packet to the controller so that it can create a flow entry to match the packet. Figure 4.1 contains a simplified representation of the OpenFlow 1.0 pipeline. The arrow represents a packet being received by the datapath. When the packet is received, fields from the packet are looked up in the table to find the associated action. The action is then taken—for instance, forwarding out a particular port.

4.1.2 Instructions

In OpenFlow 1.1, the datapath is expanded to support multiple tables. The pipeline processing begins at the first table, and may finish processing after a single table (as with OpenFlow 1.0), or may be configured to continue processing on another table.

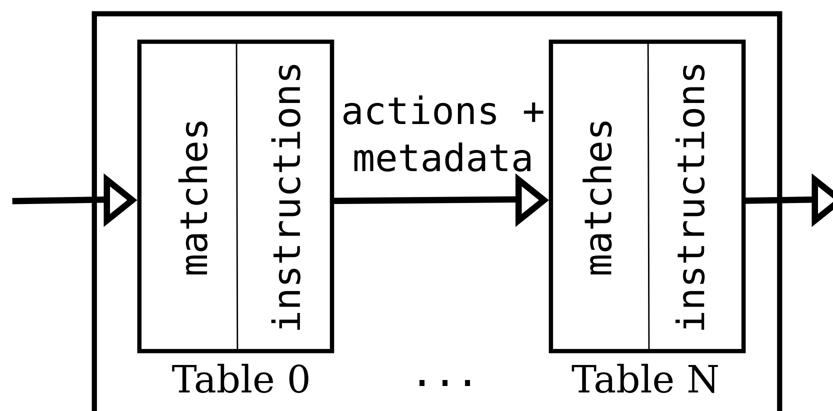

Figure 4.2: OpenFlow 1.1+ Pipeline

Instructions in OpenFlow 1.1 allow configuration of this processing behaviour. They take the place of actions in a flow entry, providing a superset of the functionality. Instructions may have actions attached, in which case they specify how those actions are to be performed; for instance, the *Apply-Actions* instruction performs the attached actions immediately—in the same manner as OpenFlow 1.0. Other instructions provide for attaching actions or data to a packet, and moving the processing immediately to any later table. Figure 4.2 displays the pipeline for OpenFlow 1.1 and above. As with the OpenFlow 1.0 pipeline, the fields from a packet are looked up in a table, starting with the first. The table will provide instructions for further processing. This may forward to a later table, or it may immediately apply a set of actions. If the packet is forwarded to another table, then actions and data can also be attached to the packet as it goes through the pipeline.

4.1.3 Experimenter Extensions

As a protocol developed in the research community, OpenFlow has always had an inherent focus towards assisting experimentation. Early development of the specification was done in the open. By version 0.89, there were provisions specifically for allowing vendor extensions, later known as *Experimenter Extensions*.

The experimenter extensions support consists of standardised structures for adding new matches, actions and instructions (OpenFlow.org, 2011). This allows researchers to develop functionality to be included in subsequent versions of the specification. Vendors and researchers may use this feature to build upon the existing feature-set from the standard protocol. Many of the features from Nicira's extensions to OpenFlow 1.0 have become standardised in later versions of the specifications. One such example is the *extensible matches*, which are used by all of the features described in Chapter 5.

4.1.4 Extensible Matches

Extensible matches provide researchers with additional flexibility for modifying flow entries. In OpenFlow 1.0 and 1.1, flow modification messages contain fields for every type of match that the protocol supports. When constructing these messages, the match entries for all of these features must be sent, even if the flow entry only matches on a single field. Extensible matches allow

these messages to only include relevant matches. This feature was introduced in Open vSwitch as Nicira Extensible Match (NXM), and was standardised in OpenFlow 1.2 as OpenFlow Extensible Match (OXM) (Open Networking Foundation, 2012). To support new features on all versions of OpenFlow, these structures will need to be updated.

4.2 Open vSwitch Architecture

The internal structure of Open vSwitch consists of the following components:

- `ovs-vswitchd`: the daemon process that stores state,
- `netdev-provider`: provides network device information,
- `ofproto`: brokers OpenFlow connections,
- `ofproto-provider`: implements OpenFlow functionality.

Figure 4.3 shows the relationship between these components. `ovs-vswitchd` is the main userspace program, which stores and retrieves state from a database. The `netdev-provider` handles interaction with network devices, which is done by interfacing with the running kernel. `ofproto` communicates with the OpenFlow controller, and passes messages down to the `ofproto-provider`. Finally, `ofproto-provider` is the component which understands and implements OpenFlow—parsing the messages and installing rules to forward flows.

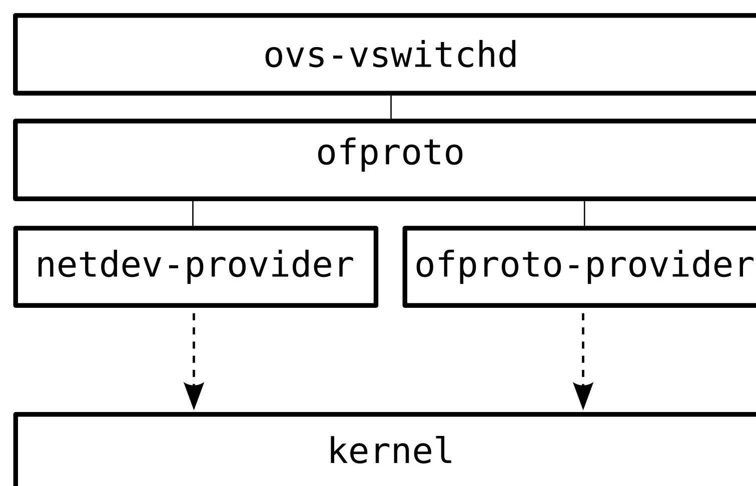

Figure 4.3: Open vSwitch Components

4.2.1 OpenFlow Provider

This project focuses on implementing features in the OpenFlow provider. Figure 4.4 shows the primary functions of this component. The *parser* takes OpenFlow messages from the higher level, and converts them to the internal flow entry format. This flow entry is then written to the *datapath*, which performs the forwarding of packets.

The OpenFlow provider distributed with Open vSwitch contains two datapaths: one in userspace, and one in the Linux kernel. These datapaths make use of a shared OpenFlow implementation for features which they do not directly support. The default behaviour is that Open vSwitch will attempt to use the kernel DataPlane for forwarding, as this is faster—packets do not need to cross the userspace/kernelspace divide. In cases where the kernel implementation does not provide particular OpenFlow functionality, it will revert to using the userspace library.

4.2.2 Testing

Open vSwitch includes an extensive test-suite for over 1000 individual features. The testing of OpenFlow features can be broken up into the two main functions from the OpenFlow provider—parser and DataPlane. The parsing tests just check that the OpenFlow messages can be understood and converted into the

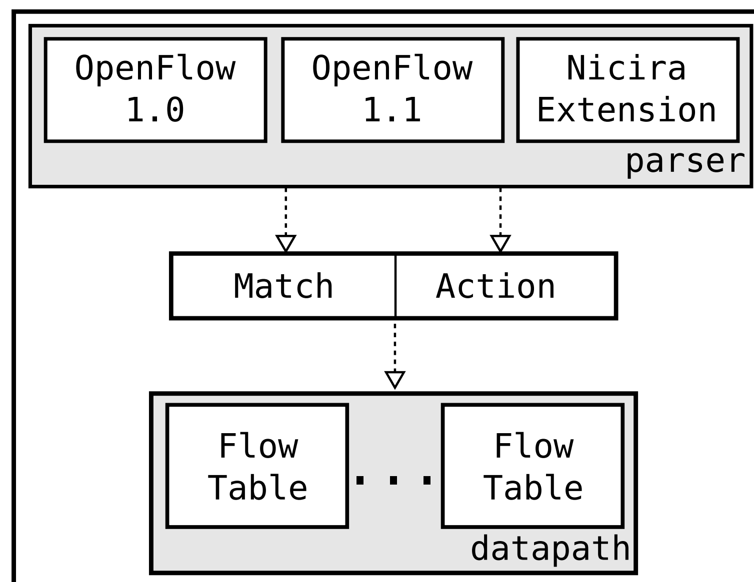

Figure 4.4: OpenFlow Provider

internal format; the DataPlane tests ensure that the functionality of these messages matches the specification. These tests are written with reference to the internal workings of Open vSwitch, known as white-box testing.

Testing the Parser

Open vSwitch includes a commandline tool called `ovs-ofctl` that can parse OpenFlow messages and print text reflecting the purpose of the message. The contributor specifies the hex bytes of the message as input, then `ovs-ofctl` parses the message as though it had been received from the controller.

For a test case, a test's expected output will initially be set to expect an error message for the feature being tested. This shows that Open vSwitch will notify the controller when a message is sent that uses an unsupported feature. When support for the feature is written, the contributor adds the new feature to the print component. Subsequently, the contributor updates the expected output to the new feature to show that the parser recognises the new feature.

Testing Functionality

The functionality of a feature is tested in a different manner: by setting up a copy of Open vSwitch and sending packets through it to determine whether the behaviour is as expected. Unlike the OpenFlow Testing Framework described in Section 2.4, these tests build upon knowledge of the internal structures to test that the behaviour is correct. The test script starts Open vSwitch and carries out the following:

1. Send a *flow modification* message including the new match and an action
2. Send a packet which satisfies the match criteria
3. Verify that the specified action is carried out

Between the parsing tests and the functionality tests, the Open vSwitch test suite covers the interpretation of OpenFlow messages and implementation of their behaviour. The use of this test suite to test the features implemented in this project is discussed further in Chapter 5.

4.3 Summary

This chapter has described the internal structure of how OpenFlow switches—and in particular, Open vSwitch—provides packet-forwarding functionality. The next chapter will build upon this knowledge to describe the work carried out in contribution to OpenFlow 1.1 support in Open vSwitch.

Chapter 5

Implementation

This chapter explores the work carried out for this project: support for arbitrary ethernet address masking, attaching metadata to flows, and implementing support for the Stream Control Transmission Protocol. Each feature is explained separately in the following manner: the feature is described in context of network research—what the feature is and how it can be used. The approach for implementation of the feature is then described with reference to Open vSwitch architecture. Each section will conclude with a description of how the feature was evaluated. Finally, this chapter will evaluate the success of this work based on the original goals.

5.1 Arbitrary Ethernet Address Masking

In IP-based networks, it is common to use an address comparison method known as *address masking*. This allows forwarding elements to use part of an address to direct network traffic. The OpenFlow 1.1 specification adds this feature for use with ethernet addresses (OpenFlow.org, 2011). When the controller adds a flow entry to the Forwarding Information Base (FIB), it specifies an address and a bitmask to match on. Then, when a packet enters the pipeline, the forwarding element checks that the specified mask bits are the same between the flow entry address and the packet's address. Figure 5.1 shows an example of a positive ethernet address match. `0xFF` specifies that the corresponding bits should be identical in the match address and the packet address. `0x00` specifies a wildcard; the corresponding bits need not match.

Traditionally, this functionality has not been provided by ethernet switches. In the interests of providing additional flexibility to researchers, the OpenFlow 1.1

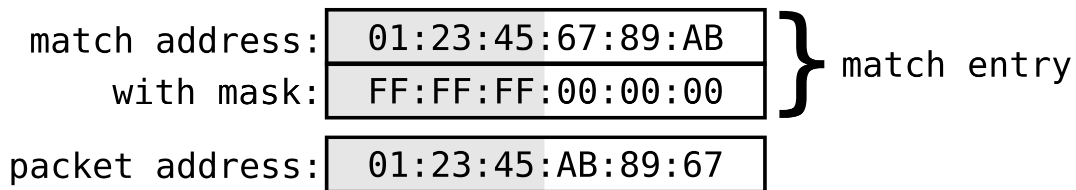

Figure 5.1: Ethernet address masking

standard makes these addresses arbitrarily maskable. Particular sections of the ethernet address have significance, such as identifying the sending device or determining whether the address is globally unique (IEEE Computer Society, 2002). This allows researchers to investigate the behaviour of particular devices in their networks.

Prior to this project, the Open vSwitch codebase provided limited functionality for masking ethernet addresses. As an OpenFlow 1.0 compliant switch, it allowed exact matching of ethernet source and destinations. Additionally, Nicira have added extensions to match the ethernet multicast bit—a section of the address that specifies whether the destination is a group of devices or just a single device.

The existing support for masking was implemented with a set of statically defined masks that indicate the following mask types:

- Mask all bits—Match a specific ethernet address
- Mask all bits except multicast—Match an ethernet address, but allow it to be directed at one or more destinations
- Mask the multicast bit—Match all traffic directed at multiple destinations
- Mask no bits—Do not match the address

The NXM match message for ethernet source and destination addresses had a field to specify which of these masks should be applied. When the message was parsed, the enumerated mask value would be placed into the flow structure, which would be used by the classifier to dynamically apply the appropriate mask.

Rather than applying pre-determined flow masks, this feature allows for specific masks to be attached to the flow entry. The work carried out in the userspace datapath was primarily in the classifier and flow components. A

new field was added to the flow structure to store the ethernet mask and the classifier was updated to use these new fields. In the OpenFlow parser, the NXM message handler was modified to use the new mask field rather than the previous masks.

Additionally, many of these functions implemented their own bitwise operations to modify and compare multi-byte ethernet values. To reduce duplicate code and increase the readability of functions that use this code, it was deemed worthwhile to refactor it into separate functions.

The testing for this feature followed the description in Section 4.2.2. The `ovs-ofctl` tool was used to parse a flow modification message containing an ethernet address and a previously unsupported mask. For it to print out the address and mask correctly shows that the parser has correctly interpreted the message and placed it into the internal structures. The classifier tests were also updated to use the new mask structures, and confirm that rules which match on arbitrary ethernet masks would take action when the mask matched a packet being received.

The implementation of this feature was considered as an opportunity to explore the codebase and become familiar with the processes as described in Chapter 3. As the first feature written as part of this project, the implementation took longer than originally scheduled. These delays can be categorised into two areas: Inexperience with the large codebase, and lack of familiarity with the code submission process. The knowledge gained in these areas improved the development speed for the subsequent features.

5.2 Metadata

The *metadata* feature in OpenFlow 1.1 allows additional information to be attached to a packet as it passes through the forwarding pipeline. One table could perform classification of a packet based on particular protocol fields and pass this information to later tables. This data can then be used to perform actions later in the pipeline. A similar feature was previously introduced to BGP (Traina et al., 1996). Donnet and Bonaventure found that on average this feature was being used by half of the routes held by the sampled set of routers, and this number was increasing over the period they were monitored. This gives some indication of the value that a similar feature is providing to

network administrators.

5.2.1 Matching on Metadata

The datapath implementation for matching on metadata is similar to that of ethernet matching—Add a field to the flow structure and update the functions which pack and unpack this structure. After the lessons learnt from the ethernet matching work, these areas were well understood. This increased the speed of implementation initially; however, the work required in the parser differed significantly. This was because the ethernet mask implementation had some existing code to extend, whereas the metadata feature did not build upon an existing feature.

The two standardised representations for metadata are the OpenFlow 1.1 match type and the OXM type used in 1.2 and above. The parser for each of these representations was updated to handle the new field. In the case of the extensible match parser, the existing implementation only handled the NXM format and any structures in the OXM format that mirrored the NXM counterpart. Prior to this project, the parser would only handle these messages if there was a representation in both formats. Duplicating the same structures for new features in both NXM and OXM provides no benefit, and is only used for older features for backwards compatibility. As such, it was deemed worthwhile to extend the extensible match support so that new OXM structures could be used with the Nicira extension format.

To test the new field, some tests were written to send metadata match messages to the datapath and check that the parser recognises the messages correctly. This was done for each of the possible representations, including the use of the new OXM metadata structure within a Nicira match message. Unfortunately, while the current version of Open vSwitch implements multiple tables internally, it does not support the OpenFlow 1.1 instruction to direct the pipeline to a later table. As such, the matching functionality of transferring metadata to another table was not unit-tested.

5.2.2 Writing Metadata

The initial approach to writing metadata was performed using a Nicira action extension, `NXAST_REG_LOAD`. This extension allows modification of any field in the flow structure. Each field from the flow structure is represented by

a number, designated by the NXM and OXM implementation. This is used to identify which flow field that `NXAST_REG_LOAD` will change. This action made it possible to independently test metadata matching before developing the write-metadata instruction. Later, the behaviour of the `write-metadata` instruction could be compared against this to determine whether the behaviour was correct.

During the implementation of this feature, a new abstraction for actions was introduced internally. The new format abstracts away the difference between actions and instructions from the core code, leaving it up to the parser to translate these structures into specific OpenFlow versions. This keeps the datapath code simple, as it does not need to handle the specifics of which OpenFlow version is being used. Instead, the conversion between the internal format and particular OpenFlow versions is handled in the parser, as described in Section 4.2.

This approach to implementation has curious effects when combined with the outlined plan to support new features on OpenFlow 1.0 with Nicira extensions. The OpenFlow 1.1 specification defines the order in which instructions should be applied; however, actions do not have the same guarantee. This is further complicated when one considers that the same Nicira extension message could be attached to an OpenFlow 1.1 message in addition to the official instruction.

After a discussion with the Open vSwitch developers, it was determined that the best approach is to enforce specific rules regarding the action's position in a message (Pfaff, 2012b). OpenFlow 1.1 instructions that are implemented as Nicira extension actions must occur at the end of a flow modification message, and must appear in the correct order—as specified for OpenFlow 1.1 instructions in the specification. Furthermore, an OpenFlow 1.1 or higher message cannot contain the metadata action structure. Any flow modification message that does not comply with these restrictions should be responded to with an error message.

Testing Metadata Writing

Testing the `write-metadata` instruction is performed using the OpenFlow *packet-in* message. This message is usually used when a packet enters the datapath and there is no flow entry that matches it. In this case, the OpenFlow switch encapsulates the incoming packet and sends it to the controller with

any additional contextual information that it can gather. The controller may then investigate the packet and create a rule to match it. In Nicira extensions (and later, OpenFlow 1.2), this message also passes back any metadata that is attached to the packet. A flow entry can also explicitly specify that matching packets should be sent to the controller. This particular behaviour is used to test the writing of metadata.

The test procedure is carried out as follows: Firstly, a flow entry is installed on the datapath to match all traffic and apply the following actions:

- Attach some specific metadata value to the flow
- Send the packet back to the controller

Secondly, a packet that matches the flow entry is sent to the datapath. The datapath matches the packet, and performs the actions specified—attaching metadata and sending a `packet-in` message to the controller. The message can then be examined to determine whether the metadata was written as expected.

There was a particular counter-intuitive test case with the expected functionality. There are no instructions in OpenFlow 1.0, so the `write-metadata` instruction is implemented as a vendor extension to the actions. When `ovs-ofctl` is used to test the conversion of this message to a OpenFlow 1.0 action structure, the tool outputs a Nicira extension structure. However, when the same is done for converting to a OpenFlow 1.1 structure, the metadata is not displayed.

This could be mistaken as an example of the OpenFlow 1.1 parser failing to parse instruction messages correctly. The actual case is that in OpenFlow 1.1, this message will only be represented as an instruction. This test case only uses `ovs-ofctl` to convert into action structures, while the actual behaviour in practice is that an OpenFlow 1.1 message will be parsed for both instructions and actions. Hence, the expected output from the test is actually to drop the action.

5.3 SCTP Support

SCTP is a transport protocol similar to TCP or UDP (Stewart, 2007). SCTP is used by the telecommunications industry to interconnect various systems in their networks, in particular newer billing and authorisation systems (Calhoun et al., 2003). As a relatively recent protocol (first introduced 2000), it is of

interest to researchers to experiment with its behaviour and compare it to traditional transport protocols.

The implementation of this feature was split into four areas: introducing a CRC32c checksum implementation for use in the userspace datapath, parsing OpenFlow matches for SCTP, and support for matching SCTP in each of the userspace and kernelspace datapaths. Section 3.4.2 described the considerations for including a CRC32c implementation, and the previous two sections outlined the modifications required to support a new OpenFlow match type. Therefore, this section will focus on the implementation of SCTP in the datapaths.

The SCTP support in OpenFlow 1.1 refers to the ability to perform matching and actions using SCTP source and destination port numbers. These port numbers allow hosts to distinguish between multiple connections with the same destination. This functionality mirrors that of several other protocols, including TCP and UDP. Furthermore, the location within a packet for these port fields is also identical. As such, some of the functionality for this feature was implemented in the same format as the code for these other protocols. This was the case for the matching support; the logic consists of looking at a particular byte offset into a packet to find the port, and using the value to compare with the relevant flow entry.

The feature support that deviates the most from these other protocols provides support for the `set-field` action. This action allows particular fields of a packet to be altered by the datapath as it passes through the pipeline. When altering part of a packet in this manner, the checksum for the packet must also be updated—hence the need for an implementation of CRC32c.

Another consideration was revealed during discussion of this feature with upstream developers. In typical ethernet networks, when a switch receives a packet that has an invalid checksum, the expected behaviour is to forward the packet normally. The end host will recognise that the packet is corrupt and deal with it accordingly. However, consider the case where a `set-field` action is used on a corrupt packet. To behave in the expected manner, this error in the checksum must be propagated across the field change.

This is performed in the following manner:

1. Take a copy of the initial checksum from the packet

2. Calculate the correct checksum for the original packet
3. Modify a field in the packet
4. Calculate the new checksum for the modified packet
5. If the first two are not identical, apply the difference to the new checksum
6. Place the new checksum into the packet

In the case where the original checksum is correct, the resulting checksum will be correct for the packet. Therefore, the destination host for the packet will see a valid packet. In the case where the original checksum was incorrect, the destination host will receive a packet with an invalid checksum. If the behaviour was not written in this manner, then it would be possible for an invalid packet to arrive at the destination host with a valid checksum.

5.3.1 End-to-End Testing

A virtualised test environment was constructed to test the implementation of this feature. Figure 5.2 shows the test setup. The test PC is running Debian GNU/Linux 6.05, with the modified version of Open vSwitch connected to two Virtual Machines (VMs). Each of the VMs has Linux Kernel SCTP (lksctp) (IBM Corporation, 2001) installed for testing. One VM acts as a server, listening for SCTP traffic while the other is instructed to send SCTP packets to the other host.

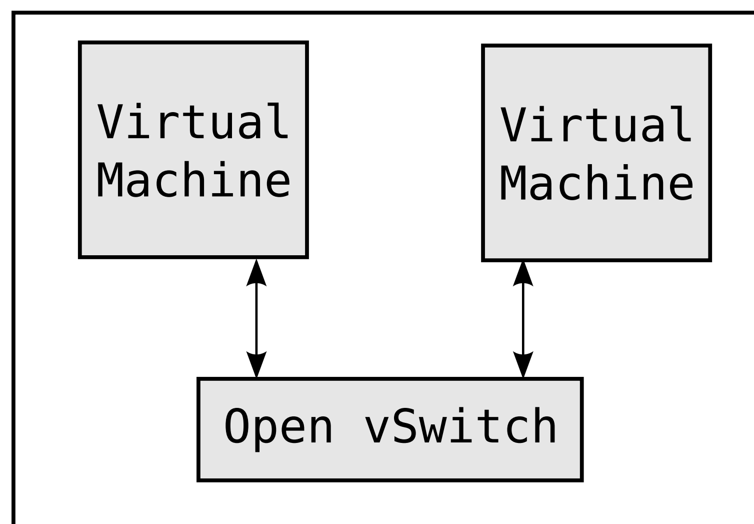

Figure 5.2: SCTP test setup

Initially, Open vSwitch acts as a standard ethernet switch—learning how to forward packets between the hosts, and allowing traffic to freely pass between them. A rule is installed into Open vSwitch to drop all SCTP traffic on the port being used. Subsequently, the SCTP traffic cannot reach the other host. Another application is used to test reachability between the two hosts, and indicates that the hosts can still communicate normally. Finally, the flow entry is deleted from Open vSwitch and the SCTP connection is observed to re-establish.

The first attempt at setting up this test environment was made using Linux Containers (LXC). LXC is a lightweight virtualisation software that provides isolated Linux systems based on the host operating system (Lezcano, 2008). This minimises the set up required to run the containers. Additionally, the virtual environment will have access to the same software that is installed on the host. LXC provides a kernel module that allows the containers to have a separate process and network space. Unfortunately, despite the support for SCTP in the rest of the Linux networking stack, this module does not support SCTP. After a query regarding this behaviour on the `lxc-users` mailing-list, the developers confirmed this lack of support (Lezcano, 2012).

For the second attempt, a more established virtualiser was chosen: QEMU (Bellard, 2003). Libvirt (Red Hat, Inc, 2005) and Kernel-based Virtual Machine (KVM) (Red Hat, Inc, 2007) were also used to speed up the installation and management of the VMs. This process involved installing a copy of Linux into each VM and installing lksctp. With these VMs set up, the test was carried out for the userspace datapath and the kernelspace datapath. This testing provided confirmation that the SCTP port matching was operating correctly.

Before spending more time on implementing and writing automated tests for the remaining functionality in SCTP support, the patchset for this feature was submitted to the `openvswitch-dev` mailing-list. This was planned to give assurance that the approach was correct, particularly with the implementation of the kernel datapath code. This review process was delayed due to the acquisition of Nicira (Herrod, 2012), and was not carried out in time for this feature to be completed to the expected quality. The remaining time of this project was spent working on the OpenFlow Testing Framework and writing this report.

5.4 Evaluation

Each prior section in this chapter has described the uses, implementation and testing of a feature individually. This section describes how well the implementation lines up with the goals set at the outset of this project.

5.4.1 Code Review

The primary goal for this project was for all implementation to be accepted upstream. As of publication, eleven patches have been accepted into the Open vSwitch codebase. Table 5.1 shows a summary of the development over the course of this project. The development time includes time spent learning the tools and processes, but does not include any time spent after the first submission of that patchset. The review time is counted from the first submission of the patch until it is submitted into the upstream repository. With the exception of SCTP and misc patches, the implementation phase was usually carried out after the review process of the previous feature. Time not accounted for in the table includes time spent on testing using the OpenFlow Testing Framework.

The notable delay in the review process for `write-metadata` is due to two factors: the patch was based against another patchset from the community which needed to go through the review process; and the acquisition of Nicira further delayed this process. This delay also affected the work on SCTP support. At the time of writing, the SCTP support is yet to be accepted upstream. The patch has received one round of review to date. However, due to time constraints, this round of feedback could not be integrated into the implementation.

The number of reviews conducted for the first three features appears to increase each time. The first round for each usually solicited a discussion on the

Feature	Patches	Reviews	Development	Review Time
Arbitrary ethernet	2	2	7 weeks	2 weeks
Metadata matching	1	3	3 weeks	1 week
Metadata writing	2	4	5 weeks	14 weeks
SCTP	5	1	9 weeks	-
Misc	5	1ea	-	-

Table 5.1: Development breakdown

implementation approach for the feature or an explanation of the architecture of Open vSwitch. In the case of metadata matching, one of the reviews was a simple request to remove an unused function which was unintentionally included in the patch.

Metadata writing had the most reviews to date, in part due to the discussion on how the feature should be represented, and also due to the decision made to base the patch on an alternative branch of the codebase. In retrospect, this decision slowed development and provided additional workload for the developers involved. The recommendation from this experience is to use alternative branches of the codebase only in cases where branch is likely to be integrated into the main development branch before completion of the feature.

5.4.2 Testing features

Each of the features that was accepted upstream included unit-tests with the code submission. These tests demonstrated the compliance with the OpenFlow protocol. This is reinforced by the reviews conducted by upstream developers, most of whom have had some involvement in the creation of the OpenFlow specifications.

An additional guarantee of code compliance was also investigated—the use of the OpenFlow Testing Framework. However, several issues impeded the use of this software. These consist of problems running the framework, and problems with the representation of OpenFlow messages.

Despite the goal for the testing framework to be easy to set up and use, there were particular hurdles faced in this area. Two versions of the framework were used: One from Ericsson Research, and one from CPqD. The latter of these required an additional library that is not developed primarily for Linux. There was some work involved in patching this codebase to have it build correctly on the test PC. Even when the framework was successfully installed, it exhibited unusual behaviour; the failure of one test would prevent the next test from running correctly. This behaviour was traced to the way in which it reserved addresses on the local host for sending OpenFlow messages.

Furthermore, despite the support of various OpenFlow 1.1 features in Open vSwitch, the current behaviour is to only use these through Nicira extensions. As of publication, the `ofproto` component that brokers connections with

OpenFlow controllers will not negotiate an OpenFlow 1.1 connection. A patchset to add this feature is currently undergoing review on the mailing-list. However, due to time constraints, this patchset could not be used to test the functionality of this project.

Chapter 6

Conclusion

This report has documented the process of learning about a large open source project and contributing features to the codebase. This process involved a cycle of learning about the project and its architecture, discussing how modifications should be made, implementing those modifications, and the steps involved in ultimately getting the code into the upstream codebase.

An exploration of the computer networking concepts was presented, to provide a basis for this project—Software-Defined Networks, OpenFlow, and how Open vSwitch implements these. This was expanded upon with specific reference to the OpenFlow specifications and Open vSwitch design documentation. The use of OpenFlow features such as instructions, experimenter extensions and extensible matches was described and linked to the architecture of Open vSwitch.

These elements—the concepts, process, and existing architecture—provided a basis for the implementation of arbitrary ethernet masking, metadata support, and SCTP support. The details of each of these features as explained and linked to how they were implemented in the Open vSwitch codebase. Finally, the implementation of each feature was assessed programmatically through software tests and independently by upstream developers. This resulted in the acceptance of eleven patches into the mainline codebase.

6.1 Impact

This project resulted in the addition of new OpenFlow features and general improvements to the Open vSwitch codebase. With the completion of the review process for SCTP support, this will also involve the inclusion of code

into the Linux kernel. This work provides additional flexibility to researchers seeking to use a production-quality software switch in their experiments.

In terms of its positioning as part of Project W, this project pushes one of the LSR components closer to compatibility with OpenFlow 1.1. This version of the protocol standardised extensions used by Kempf et al. for the original OpenFlow-based LSR platform. The contributions of this project are step towards providing a more accessible open source LSR.

6.2 Future Work

Implementing features for OpenFlow 1.1 and above in Open vSwitch is an active development area. While the majority of work towards OpenFlow 1.1 support in Open vSwitch has been contributed, there are some remaining features. To provide an interoperable platform with support for a wide range of features, support for the later 1.2 and 1.3 protocols will also need to be developed. These developments would provide the research community with a fully OpenFlow-compliant datapath. Further work will need to be carried out in the broader community to provide support for these versions of OpenFlow in controller software.

References

Bellard, F. (2003). QEMU. Retrieved 20 October, 2012, from http://www.qemu.org/.

Bianco, A., Birke, R., Giraudo, L., Palacin, M. (2010). OpenFlow Switching: Data Plane Performance. In *ICC*, pp. 1–5. IEEE.

Big Switch Networks (2011). Indigo - Open Source OpenFlow Switches. Retrieved 8 October, 2012, from http://www.openflowhub.org/display/Indigo.

Big Switch Networks (2012). OFTest—Validating OpenFlow Switches. Retrieved 8 October, 2012, from http://oftest.openflowhub.org/.

Bird, C., Nagappan, N. (2012). Who? Where? What? Examining distributed development in two large open source projects. In Lanza, M., Penta, M. D., Xi, T. (Eds.), *MSR*, pp. 237–246. IEEE.

Calhoun, P. R., Loughney, J., Arkko, J., Guttman, E., Zorn, G. (2003). Diameter Base Protocol. RFC 3588.

Calleja, D. (2012). Linux 3.3. Retrieved 22 October, 2012, from http://kernelnewbies.org/Linux_3.3.

CPqD (2012). OpenFlow Software Switch. Retrieved 8 October, 2012, from http://github.com/CPqD/.

Donnet, B., Bonaventure, O. (2008). On BGP communities. *SIGCOMM Comput. Commun. Rev.*, *38*(2), 55–59. doi:10.1145/1355734.1355743.

Ericsson Research (2011a). OFTest for OpenFlow 1.1. Retrieved 8 October, 2012, from http://github.com/TrafficLab/oftest11.

Ericsson Research (2011b). OpenFlow 1.1 Software Switch. Retrieved 8 October, 2012, from http://github.com/TrafficLab/of11softswitch.

Erlang Solutions (2012). FlowForwarding/LINC-Switch. Retrieved 8 October, 2012, from http://github.com/FlowForwarding/LINC-Switch.

Herrod, S. (2012). VMware and Nicira—Advancing the Software-Defined Datacenter. Retrieved 23 September, 2012, from http://blogs.vmware.com/console/2012/07/.

Hölzle, U. (2012). OpenFlow @ Google. Open Networking Summit.

IBM Corporation (2001). LKSCTP. Retrieved 20 October, 2012, from http://lksctp.sourceforge.net/.

IEEE Computer Society (2002). *IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture*. New York, NY: IEEE.

Kempf, J., Whyte, S., Ellithorpe, J., Kazemian, P., Haitjema, M., Beheshti, N., Stuart, S., Green, H. (2011). OpenFlow MPLS and the open source label switched router. In *Proceedings of the 23rd International Teletraffic Congress*, ITC '11, pp. 8–14. ITCP.

Lezcano, D. (2008). lxc Linux Containers. Retrieved 20 October, 2012, from http://lxc.sourceforge.net/.

Lezcano, D. (2012). Re: [Lxc-users] Alternative network protocols. Retrieved 20 October, 2012, from http://www.mail-archive.com/lxc-users@lists.sourceforge.net/msg03826.html.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J. (2008). OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, *38*(2), 69–74.

Open Networking Foundation (2012). The OpenFlow Switch Specification (1.2 and above). Retrieved 22 March, 2012, from http://www.opennetworking.org/about/onf-documents.

Open vSwitch (2012a). Open vSwitch Documentation. Retrieved 20 October, 2012, from http://openvswitch.org/support/.

Open vSwitch (2012b). Open vSwitch Mailing Lists. Retrieved 20 October, 2012, from http://openvswitch.org/mlists/.

OpenFlow.org (2011). The OpenFlow Switch Specification (1.0,1.1). Retrieved 22 October, 2012, from http://www.openflow.org/wp/documents/.

Pfaff, B. (2011). Call for assistance: OpenFlow 1.1 and 1.2 support in Open vSwitch. Retrieved 21 March, 2012, from http://www.mail-archive.com/dev@openvswitch.org/msg06532.html.

Pfaff, B. (2012a). [PATCH v2 00/11] instruction apply-actions/goto-table support. Retrieved 20 October, 2012, from http://www.mail-archive.com/dev@openvswitch.org/msg11500.html.

Pfaff, B. (2012b). [PATCH v2] ofp-actions: Implement writing to metadata field. Retrieved 20 October, 2012, from http://www.mail-archive.com/dev@openvswitch.org/msg11290.html.

Pfaff, B., Pettit, J., Koponen, T., Amidon, K., Casado, M., Shenker, S. (2009). Extending networking into the virtualization layer. In *HotNets-VIII*.

Red Hat, Inc (2005). Libvirt: The virtualization API. Retrieved 20 October, 2012, from http://libvirt.org/.

Red Hat, Inc (2007). Kernel Based Virtual Machine. Retrieved 20 October, 2012, from http://www.linux-kvm.org/.

Riel, R. (2006). UpstreamMerge/SubmitChecklist. Retrieved 22 October, 2012, from http://kernelnewbies.org/UpstreamMerge/SubmitChecklist.

Stanford University (2012). Clean Slate Design for the Internet. Retreived 12 October, 2012, from http://cleanslate.stanford.edu/.

Stewart, R. R. (2007). Stream Control Transmission Protocol. RFC 4960.

Traina, P., Chandrasekeran, R., Li, T. (1996). BGP Communities Attribute. RFC 1997.

Glossary

action How the forwarding element should handle particular matches .

AutoTest Allows developers to create platform-independent test cases. Part of GNU Build tools .

BGP Border Gateway Protocol. The de facto standard protocol for sharing routes with other organisations .

black-box Testing the behaviour of a component based on sending particular inputs to the component, and monitoring the outputs .

controller The part of a forwarding element which is responsible for route aggregation and advertisement .

CRC32c The checksum algorithm used in SCTP to detect transmission or storage errors for the data contained in a packet .

datapath The part of a forwarding element which is responsible for the forwarding of packets. Typically used to refer to a software or hardware switch .

flow A class of packets, as determined through a set of common attributes such as the same source or destination address. .

flow entry A combination of a match and action .

forwarding element A router or switch .

Forwarding Information Base A table which stores forwarding entries for fast packet switching .

Host Byte Order The endianness of bytes when the data is stored in memory. Architecture dependent .

Label-Switched Router A type of router that provides label-switching functionality. This functionality is used to simplify traffic management in high-traffic network environments. .

match The classification of a packet based on particular field values .

Network Byte Order The endianness of bytes when the data is written to the network interface. Typically standardised as big-endian—Most significant bytes first .

Sparse The semantic parser, a compiler frontend and static code analyzer for ANSI C programs .

white-box Testing the behaviour of a component with reference to the component's structure, often by using shortcut code that allows direct access to internal functions .