Master Thesis

Part I: Traffic as a Service

Part II: Debugging OpenFlow Networks

Jochen Mattes

Saturday 10th November, 2012

Advisors:
Dr. Ali Al-Shabibi*,
Nicolas Bastin[1]*
Dr. Xenofontas Dimitropoulos†,
Vasileios Kotronis†
Dr. Bernhard Ager[2]†

Supervisors:
Prof. Dr. Guru Parulkar*,
Prof. Dr. Bernhard Plattner†

†Communication Systems Group
Department Information Technology and Electrical Engineering
Swiss Federal Institute of Technology Zurich (ETH Zürich)

Computer Enineering and
Networks Laboratory

Departement Informationstechnologie
und Elektrotechnik

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

*ON.Lab
245 Lytton Avenue, Suite 175, Palo Alto, CA 94301

ON.LAB

[1] Nicolas Bastin left the project.
[2] Dr. Bernhard Ager was not an official advisor, yet this thesis greatly benefited from his support.

Abstract

Recent research efforts in the field of Software-Defined-Networks (SDN) led to development of the OpenFlow Protocol [1], which decouples the network data plane from the control plane, pushing the latter to an external controller. A hypervisor called FlowVisor [2] has been developed, which intercepts the control channel between controller and the switches. It allows multiple controllers to share the network resources in a specified way.

The header values of a packet can be viewed as a point in the header space which is spanned by the header field dimensions [3]. FlowVisor allows the network administrator to define volumes in this space and associate these with a controller. A controller is allowed to handle packets which fall into one of the volumes associated with it. Yet it is not guaranteed that all such packets are handled by the corresponding controller, as the volumes of different controllers can overlap and FlowVisor forwards related messages only to one of these controllers.

Related to this, the Master Thesis at hand covers two linked yet separate research topics:

(i) The support point of the first part is the necessary-but-not-sufficient nature of FlowVisor, as described in the forgoing paragraph. The transition of the paradigm towards a necessary-and-sufficient nature is proposed and evaluated. As a result researchers may use production traffic in real-time for their work without interfering with the functionality of the network. An implementation called **V** is given and found to prove the claims made.

(ii) The modifications proposed in **V** are results of the modification proposals made in the second part. If the necessary-but-not-sufficient approach is kept, conflicts between controllers can arise. How to use information that can be obtained about the state of the network to find such conflicts is discussed in this part. As a side-product a network debugger has been developed that allows researchers to simulate the path a given packet would take through the network at any given time.

Preface

Acknowledgment

The second part of the present work has been advised and supervised by Dr. Ali Al-Shabibi and Prof. Dr. Guru Parulkar. It was conducted in the then newly founded ON.LABs in Palo Alto, California, USA. After returning to the Communication Systems Group in Zurich, Switzerland the topic "Debugging of OpenFlow Networks" seemed to offer no challenges that would take up the remaining 2.5 months, so I changed to a topic that seemed more promising and more challenging. All conflicts between controllers in OpenFlow networks seemed to stem from misconfiguration or incorrect implementation. Instead of further working on a tool to support network administrators and developers in finding bugs and configuration errors (the development prototype was already working at that time) I wondered whether it was possible to generally preclude collisions between separate controllers. This would not completely foreclose the necessity for a debugging tool, as it would still support developers of the individual stages to search for bugs in their implementation, but it would reduce the possible conflicts that can occur. I experienced great support in doing so from Dr. Xenofontas Dimitropoulos, Vasileios Kotronis and Dr. Bernhard Agner. I want to express my deepest gratitude towards them and thank them for their great support and very constructive criticism.

Jochen Mattes - Wednesday 1st August, 2012 - Zurich, Switzerland

Contents

1 Introduction **5**
- 1.1 OpenFlow . . . 5
- 1.2 FlowVisor . . . 6
- 1.3 Glossary . . . 7
- 1.4 Mathematical Glossary . . . 8
- 1.5 OpenFlow Control Messages . . . 9

2 Production Traffic as a Service **11**
- 2.1 Motivation . . . 11
- 2.2 Related Work . . . 14
  - 2.2.1 Mirror VNETs . . . 14
  - 2.2.2 FlowVisor . . . 15
  - 2.2.3 Splendid . . . 16
- 2.3 Concept . . . 16
  - 2.3.1 Controller Classes . . . 16
  - 2.3.2 Further Controller Classes . . . 18
  - 2.3.3 Hypervisor . . . 19
  - 2.3.4 Dynamic Slices - Reservations . . . 22
- 2.4 Architecture . . . 29
  - 2.4.1 Topology Discovery . . . 29
  - 2.4.2 North Bound Interface . . . 31
  - 2.4.3 Slicer . . . 31
  - 2.4.4 Merger . . . 32
  - 2.4.5 Classifier . . . 38
  - 2.4.6 South Bound Interface . . . 39
- 2.5 Implementation . . . 40
  - 2.5.1 External Sources Used . . . 40
  - 2.5.2 Starting Sequence . . . 42
- 2.6 Proof of principle . . . 43
  - 2.6.1 Validating the Implementation . . . 43

2.6.2 Validation of the Algorithm 44
2.6.3 Validation Topology I 47
2.6.4 Validation Topology II 48
2.7 Scaling Considerations 49
2.7.1 Database Operations 50
2.7.2 North Bound Interface 51
2.7.3 Slicer 51
2.7.4 Merger 52
2.7.5 Classifier 54
2.7.6 South Bound Interface 55
2.8 Conclusions 56

3 Debugging OpenFlow Networks **58**
3.1 Motivation 58
3.2 Related Work 60
3.3 Data Sources 60
3.3.1 OpenFlow 60
3.3.2 FlowVisor 61
3.3.3 sFlow 61
3.3.4 SNMP 62
3.4 Concept 62
3.4.1 Controller-Specific Network Views 64
3.4.2 Network-Neutral Controller Probing 66
3.4.3 Comparing the Controller-Specific Views 68
3.4.4 Small Flows 69
3.5 Implementation 72

4 Conclusions **73**

5 Appendix **77**
5.1 Task Description 77
5.2 Declaration of Originality 78

Chapter 1

Introduction

1.1 OpenFlow

OpenFlow is a realization of a Software Defined Networking protocol (SDN) that separates the control and data plane of network switches. The advantages of moving the control plane to an external controller is, that this controller can be implemented in software. Hence allowing greater flexibility, evolvability, extensibility, etc. in controller development. The data plane remains on the switch, which allows fast, line-rate packet-processing. The controller can modify the behavior of the data plane by modifying the switch's flow table, which consists of the following six parts [4]:

(i) the **match** field which describes the flowspace for which the entry is valid,

(ii) the **priority** field that determines the ordering/prioritization of the flow table entries,

(iii) the **counter** fields that keep record of the number of packets and bytes the flow table entry applied to in the past,

(iv) the **instructions** which define the actions that need to be applied onto packets that match the flow table entry,

(v) the **timeout** fields which defines how long the entry is to remain in the table and

(vi) a **cookie** chosen by the controller.

Packet-Processing When a new packet arrives at a physical port of an OpenFlow-enabled switch the packet's header fields are matched against the internal flow table of the switch.

(i) If a matching entry is found, the associated actions are applied.

(ii) If no matching entry is found, the packet is either dropped or forwarded to the controller. Which of these actions is executed depends on the configuration[1].

If the packet is forwarded to the controller, it is encapsulated into an OFPT_PACKET_IN message that gives the controller context information such as the in-port through which the packet entered the switch. Upon reception of an OFPT_PACKET_IN message by the controller, the packet can be processed in software, with all advantages and disadvantages associated with this practices. The controller would normally install a new flow table entry in the switch's controller and/or send the packet back to the switch together with an associated action to be applied to it.

Proactive Rule Installation A controller can at any time (independent from receiving an OFPT_PACKET_IN message), install forwarding rules into the flow table of a switch and thereby modify the behavior of the data plane. Further the controller can request information from a switch by sending OFPT_STATS_REQUEST[2] messages that, depending on the message type, return information about configuration settings or rules that are installed in the switch's flow table. This source of information can be leveraged to develop dynamic controllers that implement load-dependent policies (e.g. load balancing applications).

1.2 FlowVisor

FlowVisor is a piece of software that resides between controllers (North Bound Interface) and networks switches (South Bound Interface) and intercepts the control channel messages exchanged between them over OpenFlow. It can be seen as a special kind of controller. It offers the normal controller interface for the network switches, while it emulates a

[1] This describes OpenFlow 1.0 behavior. OpenFlow 1.x allow for greater flexibility.
[2] This describes OpenFlow 1.0 behavior. OpenFlow 1.x uses other control messages.

switch interface on its northern side towards the controllers. This transparent architecture allows it to share the resources of a switch among multiple controllers in a safe way without introducing the need to modify controllers or switches.

FlowVisor interprets the header values of a packet as a point in the header space which is spanned by the header field dimensions [3]. It allows the network administrator to define volumes in this space and associate a controller with them. The controller is then allowed to handle packets whose header fields describe a point in a volume associated with it. FlowVisor guarantees that the controller does not handle the traffic outside its slice[3] by limiting the validity of its rules to the volumes that are associated with the controller.

However the controller is not guaranteed to handle all packets that fall within its slice. This is because FlowVisor allows the volumes of different controllers to overlap. As a result control messages arriving on its South Bound Interface related to a packet whose point in the header space is associated with multiple controllers will only be forwarded to one controller. If multiple controllers are associated with the header-space-point in question, their volume-dependent priority determines which controller receives the OFPT_PACKET_IN message.

1.3 Glossary

This section describes the key terms used in the thesis at hand. It is based on the glossary of the OpenFlow Specifications 1.3 [5].

- **Byte:** an 8-bit octet.
- **Packet:** an Ethernet frame, including header and payload.
- **Physical Port:** where packets enter and exit the OpenFlow pipeline.
- **Flow Table:** a switch's internal database table that defines packet patterns and associated actions. Refer to [4, 6, 7, 5].
- **Flow Entry:** an element of a flow table used to match and process packets. It contains a set of match fields for matching packets, a

[3] The controller's slice is the set of volumes that have been associated with it by the network administrator.

priority for matching precedence, a set of counters to track packets, a set of instructions to apply and a controller-chosen cookie.

- **Match Field:** a field against which a packet is matched, including packet headers, the ingress port, and in OpenFlow 1.x the meta-data value. A match field may be wildcarded (match any value) and in some cases bitmasked.
- **Slice:** Collection of header-space volumes that are associated with a specific controller. In the context of FlowVisor the slice describes the sphere of influence of a controller. In the context of **V** it describes which part of the incoming traffic is being seen by a specific controller.
- **V** : Implementation of the first algorithm proposed in this thesis.
- **Controller Rule:** a logical flow entry as defined by the controller.
- **Switch Rule:** a physical flow entry in the switch's flow table.
- **Sliced Rule:** a rule that has been processed by the Slicer unit of **V**. It describes an intermediate, internal state between controller and switch rule.
- **Sacramento:** Implementation of the second algorithm proposed in this thesis.

1.4 Mathematical Glossary

- ϕ_i**:** controller rule.
- χ_i**:** sliced rule.
- ψ_i**:** switch rule.
- ν_i**:** number of controller rules installed by controller i.
- k_{fs_i}**:** number of switch rules whose flowspace overlaps with fs_i.
- p_i**:** a network packet.
- q_i**:** a physical port.
- $\mathcal{H}$**:** the header space spanned by the 12 header fields of a network packet against which OpenFlow 1.0 matches.

1.5 OpenFlow Control Messages

In the OpenFlow Specifications 1.0 [4], the following control messages are defined[4] :

- **OFPT_HELLO:** a message to initiate the connection between switch and controller.
- **OFPT_ERROR:** message sent by the switch to inform the controller about an error.
- **OFPT_ECHO_REQUEST:** message used to implement a keep-alive of the connection[5].
- **OFPT_ECHO_REPLY:** response to an OFPT_ECHO_REQUEST message.
- **OFPT_VENDOR:** a vendor dependent message that can be used to prototype new functions into OpenFlow.
- **OFPT_FEATURES_REQUEST:** a message generated by the controller to request the features of a datapath.
- **OFPT_FEATURES_REPLY:** reply to the OFPT_FEATURES _REQUEST which contains a list of the switch's features.
- **OFPT_GET_CONFIG_REQUEST:** a message sent by the controller to query the current configuration of a datapath.
- **OFPT_GET_CONFIG_REPLY:** a message generated by the datapath in response of an OFPT_GET_CONFIG_REQUEST query containing the value of the queried configuration parameter.
- **OFPT_SET_CONFIG:** a message sent by the controller to modify the settings of a datapath.
- **OFPT_PACKET_IN:** a message generated by the datapath on forwarding a packet to the controller.
- **OFPT_FLOW_REMOVED:** a message sent by the switch to inform the controller about a message that timed out.

4 Be aware that OpenFlow 1.x define a different set of control messages, that allow greater flexibility.

5 For a good discussion on Keep-Alive messages refer to [8].

- **OFPT_PORT_STATUS:** a message generated by the switch to inform the controller about a change in state of a physical port.
- **OFPT_PACKET_OUT:** a message that allows the controller to inject a packet into the data plane of a switch.
- **OFPT_FLOW_MOD**: a message generated by the controller to modify the flow tables of a switch.
- **OFPT_PORT_MOD:** a message that allows the controller to modify the behavior of a physical port of a switch.
- **OFPT_STATS_REQUEST:** a message generated by the controller to query the contents of one of the datapaths flow tables.
- **OFPT_STATS_REPLY**: a message that is generated in response to an OFPT_STATS_REQUEST message containing the requested information about the flows in the switch's flow table.
- **OFPT_BARRIER_REQUEST:** a message generated by the controller to ensure ordering of a sequence of control messages. This is necessary because the switch is allowed to reorder control messages. All messages received by the controller before reception of this message need to be processed before the OFPT_BARRIER_REPLY is sent.
- **OFPT_BARRIER_REPLY:** the switch's reply generated in response to an OFPT_BARRIER_REQUEST.
- **OFPT_QUEUE_GET_CONFIG_REQUEST:** a message that is generated by the controller to query the switch's configuration.
- **OFPT_QUEUE_GET_CONFIG_REPLY:** a message that is generated by the switch after receiving the controller-generated OFPT_QUEUE_GET_CONFIG_REQUEST.

Chapter 2

Production Traffic as a Service

2.1 Motivation

Flow-Rule Priorities If controller slices overlap, OFPT_PACKET_IN messages from the overlapping flowspace are delivered to only one controller. However multiple controllers are allowed to install rules valid for that flowspace. Since a controller can install a rule at any time, the rule active on the switch does not necessarily originate from the controller that would have received the respective OFPT_PACKET_IN messages. If this controller installs an overlapping rule, it chooses a priority that is not being coordinated with the first controller. Hence the latter rule will have a higher, equal or lower priority than the the already installed one[1].

The OpenFlow Specifications [4] specify:

> « Packets are matched against flow entries based on prioritization. An entry that specifies an exact match (i.e., it has no wildcards) is always the highest priority. All wildcard entries have a priority associated with them. Higher priority entries must match before lower priority ones. If multiple entries have the same priority, the switch is free to choose any ordering. Higher numbers have higher priorities. »

[1] The following passage describes the FlowVisor 0.83 behavior. The FlowVisor community is aware of the issue, but at the time of writing (September 8th, 2012) the according ticket is still open: `https://openflow.stanford.edu/bugs/browse/FLOWVISOR-72`.

Always only one rule being chosen, hence the highest priority rule starves lower priority rules. In the case of rules with equal priorities, the network's behavior becomes indeterministic. Because the priorities of controller rules are not being coordinated, a controller that installs rules in a proactive manner with higher priorities has a better chance of the network reflecting its policies than a controller which installs rules only as reaction to an OFPT_PACKET_IN message with lower priorities. Traffic Sharing – if traffic is seen as a network resource – is therefore neither fair nor deterministic.

Constraints on Controller Capabilities In FlowVisor a controller has an associated slice, which is a collection of volumes in the header space for which the controller is responsible. The slice can be switch dependent, but it is being defined by the network administrator at configuration time and stays constant during runtime. This goes against the nature of OpenFlow because it offers a set of features that allow the controller to modify the header fields of packets, thereby pushing the packet to a different point in the header space. If FlowVisor would allow a controller to modify a packet in such a way, that the resulting packet-header would not lie in its slice, the packet would leak, into an uncontrolled part of the header space or potentially even into the slice of another controller. This has been described by Kazemian et al. [3].

Assume for example that the network administrator defined a network-wide slice $\{\text{VLAN_id} : u\}$ for a specific controller. This controller would not be aware of the limitation of its realm. Hence installation of a rule that modified the VLAN id of certain packets and set it to $v \neq u$ would seem to be valid. Packets onto which this rule applied would leak the slice and would not be observed by the controller on the next switch. If the network administrator associated another controller with the flowspace $\{\text{VLAN_id} : v\}$, the packet would be handled by that controller. If the network administrator did not associate any controller with the flowspace, the packet would be dropped because no controller was able to install a corresponding rule.

If a controller is aware of the network topology it would observe the leakage of its own packets into slices of other controllers as well as the leakage of foreign packets into its own. It would observe packets being discarded / generated on the network link. From the controllers point-of-view the network would be ill-behaving.

FlowVisor solves this by rejecting rewrite actions, that would push packets out of the controller's slice. This changes the behavior of the network as observed by the controller and limits the controller's capabilities. It breaks the aspiration of providing a multi-controller framework in which *all* controllers that work correctly in single-controller networks can be run without modifications.

Traffic as a Service Evaluating a hypothesis in the field of Computer Networks frequently requires analysis of production traffic. Today this can be done by replaying/reproducing packet traces/statistics that were collected beforehand. The traces would normally be obtained my sampling and aggregation of traffic on the switches and the resulting statistics would be pushed to a collector [9, 10]. This procedure does not involve communication with the measurement application. Hence parts of the traffic that would be important might be missed, whereas unnecessarily unnecessarily collected traffic might cause an overhead.

Conclusion FlowVisor's impact on the network's behavior makes it hard for controller-developers to reason on the network's behavior in the context of shared flowspaces. This is especially true because the controller is not aware of the limitations of its realm[2]. Thus I propose a different approach, called "**V**" that *simulates a virtual network for each controller*. These controller-specific virtual networks are fully isolated from each other and cannot interact in any way. The virtual networks can be simulated efficiently, as the controllers and switches only communicate via control messages. It is sufficient to generate the expected control messages to trick the controller into assuming a different network state. In this approach the controller's slice determines which part of the *incoming traffic* is being observed by the controller, removing the requirement for packets to stay within a slice.

The approach also allows the definition of so called development controllers that obtain the same network view as they would have in a single-controller-network. However they are not allowed to deliver packets to end-hosts. This is exclusively done by primary controllers, whose slices cannot overlap. Hence development controllers can be used as data sources for the evaluation of research hypotheses, while the end to end behavior of the network is not changed by their presence.

[2] Refer to [11] for a short description on how a controller can probe its realm.

2.2 Related Work

2.2.1 Mirror VNETs

Wundsam et al. provide a context in which the same network can be run with different configuration settings at the same time. They propose mirroring VLANs by duplicating packets with a specific VLAN id to a different virtual network. Packets belonging to the "cloned" virtual network are dropped at the border switches before they leave the network. This in [12] Wundersam et al. state:

> « We propose **Mirror VNets**, which replicate networks and traffic in a safe fashion. Thus, the new Mirror VNet and the production VNet can operate in parallel and the user traffic is duplicated either completely or in part to both networks. »

This approach is different from the approach that I propose, as in my concept a packet is only replicated if multiple controllers specify differing paths for it. Mirror VNets (one per controller) generate a traffic overhead of 100% per additional VLAN. The overhead that is introduced by my approach, reaches 100% if and only if the additional controller and the original controller specify different paths for every single flow in the network. Reducing the traffic overhead is covered in their paper, but the proposed methods have clear drawbacks:

(i) *Stripping the payload off the packets* leads to OFPT_PACKET_IN messages with incorrect payload. Controllers that use the payload to determine what to do with the packet, will not function correctly with this approach[3].

(ii) By *collecting packet statistics and generating corresponding traffic in the mirrored network,* the advantage of working with real traffic is lost.

(iii) By *mirroring only on sampled packets,* small flows might be missed. This is especially problematic if the configuration-in-use observes problems with establishing certain TCP connections, because these connections-trials do not generate a lot of traffic and sampling from them is therefore unlikely.

[3] For example, controllers that aim to detect file sharing traffic sometimes rely on payload inspection.

Wundsam et al's approach allows to determine online which controller is allowed to deliver the packets to the end-host. This is currently not supported by my approach, yet by applying the work of Reitblatt et al. [13] this should be possible.

2.2.2 FlowVisor

FlowVisor [2] as described in the Introduction (Section 1.2) is similar to the presented approach in many ways. In fact my implementation is based on the openly available code of FlowVisor. Just as **V**, FlowVisor is a hypervisor that provides safe resource sharing of the switches among controllers:

> « We demonstrate that FlowVisor slices our own production network, with legacy protocols running in their own protected slice, alongside experiments created by researchers. »

Yet in this context "resource sharing" indicates physical resources as the network topology, the switch's flow tables etc. In FlowVisor's approach traffic is not interpreted as a resource that needs to be shared among controllers. In the concept of FlowVisor, a controller that sees[4] a packet is responsible for its delivery. This implies that a researcher needs to generate his/her own traffic if the production network shall stay unaffected by the testing of his/her controller. With the concept proposed in this Master Thesis the researcher can think of the production traffic as a freely available resource, which he/she can use for his/her purposes without any obligation of delivering the packet to the correct destination. The distinction between primary and development controllers is not existent in FlowVisor, in which all controllers are automatically primary controllers and need to deliver the packets they observe. Seen that way **V** is a generalization of FlowVisor.

[4] A controller sees a packet if the packet is either being handled by a flow rule installed by the controller, or the packet is being forwarded to the controller via an OFTP_PACKET_IN message. In the latter case it observes the packet directly in the earlier case it observes it indirectly via an increase in the packet counter associated with the given rule.

2.2.3 Splendid

In the paper "Splendid isolation: a slice abstraction for software-defined networks" [14] which, at the time of writing, was about to be published Gutz et al. describe a different approach towards slice abstraction. They advocate the compilation of the controller's slice at compilation time.

> « Unlike a hypervisor, which must intercept and analyze every event and control message at run-time, the compiler only needs to be executed once – before the program is deployed in the network – which streamlines the control plane and reduces latency. Finally, obtaining isolation through language abstractions provides opportunities for obtaining assurance using formal verification tools. »

This approach seems very promising, and could arguably replace a static hypervisor, if compilation of the controllers is done in a coordinated and safe way. A related implementation of slice compilation at controller compilation time can be found in Frenetic [15].

2.3 Concept

2.3.1 Controller Classes

V's main modification to FlowVisor's approach is the introduction of different controller classes. While in FlowVisor all controllers are treated the same way, **V** accounts for the special needs of different use-cases and provides suitable controller classes. Each controller truly sees its own virtualized network. Primary controllers have the right to deliver the packets of the production traffic to end hosts. Development controllers on the other side do not have this privilege.

Primary Controllers

V's treatment of primary controllers is very similar to the way FlowVisor treats it's controllers. While receiving the OFPT_PACKET_IN messages triggered by packets that fall into their slice[5], a controller of this class is allowed to install rules on the switch that handle the traffic of the associated slice. FlowVisor as well as **V** limit the validity scope of such rules to the intersection of the rule's flowspace and the controller's slice. While FlowVisor allows the slices of its controllers to overlap, **V** only accepts non-overlapping slices for primary controllers[6]. This is done to ensure that for each packet entering the network there is at most one path defined that delivers the packet to an end-host. It is the primary controller's responsibility to ensure correct packet handling of *all* packets in it's slice.

Development Controllers

The slices of development controllers can overlap any way the researchers choose and are unrestrained by the slices of primary controllers. A controller of this class receives the same network feedback (discussed later on in this section) as it would, was it the only controller responsible for the specific flowspace. This view of the network is controller-specific, because multiple primary and development controllers might be active in the flowspace.

Packets seen by development controllers are already been taken care of by a primary controller[7]. This liberates them of the obligation to deliver packets to the correct destination or execute any other desired action correctly. Essentially they receive the production-traffic-as-a-service and are able to push packets they observe through the network in any desired way, without breaking the network's functionality [8].

[5] FlowVisor only sends the packet to the controller with the highest priority should the packet fall into the slices of multiple controllers.

[6] Actually the controller slices can overlap in the same way as they do in FlowVisor, however the priority of the flowspaces is taken into account also when determining the validity space of a rule that is being installed on the switch.

[7] The network administrator can choose not to define a primary controller for a specific flowspace, but then the respective packets would not be delivered to end hosts.

[8] The online modification of the controller's class (development -> primary or primary -> development) is part of the future work.

2.3.2 Further Controller Classes

Further controller classes can be defined with a different set of features. They are listed under "Further Controller Classes", because they were not implemented in the development prototype. Nevertheless they can be very useful.

Measurement Controllers

The measurement controller class is designed to allow convenient use of measurement controllers, which aim to leverage the OpenFlow packet and byte counters associated with every flow table entry to measure some part of the primary traffic. Primary traffic in this context is traffic which is being sent through the network by a primary controller. A measurement controller does not aim to install any forwarding rules, but it rather aims to measure the traffic of the primary controllers. Therefore controllers of this class are freed from the responsibility to install forwarding rules.

Still they obtain a copy of all OFPT_PACKET_IN messages which fall in their slice and are forwarded to a primary controller[9].

By installing a rule, the measurement controller asks **V** to collect the packet and byte counters for primary traffic in the specified flowspace. In essence a development controller thereby defines which entries it expects in the OFPT_STATS_REPLY. The actions associated are ignored.

This class is especially helpful for controllers that want to measure traffic aggregates in the network. It allows simple implementation of measurement controllers as proposed in my semester thesis [11] or in the work of Jose et al. [16] – developed at around the same time. The controllers do not need to be aware of the forwarding policies in the network and can focus on measuring traffic of other controllers.

[9] A measurement controller probably needs to be associated with one or more primary controllers, because the forwarding of OFPT_PACKET_IN messages relies on a dynamic controller-specific method and specifying a static slice for a development controller conflicts with the dynamic nature of the approach.

Debug Controller Class

The debug-controller class is meant for controllers that check the implementation of my proposal. They receive all OFPT_PACKET_IN messages that are generated by the network (directed towards primary and development controllers) and the unsliced network statistics. Rules that they install are handled in the same way as the ones of measurement-controllers.

2.3.3 Hypervisor

V sits between the controllers and switches of an OpenFlow network and acts as hypervisor which ensures that:

(i) packets are delivered to end-hosts through the path installed by the primary controller in whose slice the packet falls,

(ii) sufficient information is gathered from the network to allow simulation of individual virtual network view to every registered controller and

(iii) every controller observers the network view as it would in a single-controller network.

As man-in-the-middle, **V** receives the control messages from both the controllers and switches and can either *(i)* drop this control message, *(ii)* modify and forward it, *(iii)* forward it without modification. Further it injects a newly generated control message to either sides at any given time.

Assuming that the OpenFlow communication channel is the only way through which controllers and switches exchange information[10], the controller's internal picture of the network state can be influenced, by sending control messages that present the desired state. It is therefore **V**'s task to calculate the control messages that a controller would observe in a single-controller network from the information gathered in the physical network.

[10] It is fairly possible that a controller obtains further information of the network's state, e.g. through sFlow samples or SNMP.

A full list of control messages as defined by the OpenFlow 1.0 Standard [4] is given in Section 1.5. For most control messages **V** relies on FlowVisor's standard behavior. This translates to forwarding messages generated by switches to the "correct" owner of the message. Ownership herein is defined by the controller's slice. Messages originating from a controller are modified by FlowVisor so that the controller does not exceed its authority. To support the development-controller class, **V**'s handling of OFPT_PACKET_IN-, OFPT_FLOW_MOD- and OFPT_STATS_REPLY-messages is different from FlowVisor's.

OFPT_PACKET_IN

This message is sent to the controller, if the switch receives a packet, that: *(i)* does not match against any entry in its flow table or *(ii)* the matching entry specifically requires the packets forwarding to the controller. Standard FlowVisor behavior specifies that a OFPT_PACKET_IN message only be forwarded to the controller whose priority for the specific flowspace is highest. In **V**'s implementation the OFPT_PACKET_IN message is forwarded to all controllers in whose slice the packet falls. This can be at most one primary controller (as the slices of primary controllers cannot overlap) and an infinite number of development controllers. Therefore OFPT_PACKET_IN messages received by **V** might be duplicated and forwarded to multiple controllers.

OFPT_FLOW_MOD

With an OFPT_FLOW_MOD message a controller specifies modifications of a switch's flow table. Such a message may: *(i)* add an entry to the flow table, *(ii)* modify one or more existing entries or *(iii)* delete one or more entries from it. As the controller slices in FlowVisor can overlap, multiple controllers have the privilege to install rules that would apply to the same packet. Which rule is being executed depends on multiple factors, that are partially indeterministic[11]. This introduction of uncertainty into a otherwise straight forward concept, breaks my perception of aesthetics. In contrast **V** translates the OFPT_FLOW_MOD messages it receives from

[11] The rule's priority chosen by the individual controllers as well as the switch's implementation can determine which rule is being executed. If multiple rules with equal priority in the flow table would apply to a given packet, the switch is free to choose either.

the controllers (hereafter controller rules) via a straight-forward process into a set of switch rules that are then being installed on the switches. This is illustrated in Figure 2.1. Translation is done so that, for every possible packet, at most one valid rule is installed on the switch. Be aware that rules with different priorities can overlap. However the controller rules have been translated into a set of switch rules which ensure deterministic behavior of the network. In consequence switch rules whose flowspace overlaps have different priorities.

OFPT_STATS_REPLY

An OFPT_STATS_REPLY message is generated by the switch in response to an OFPT_STATS_REQUEST message and contains information about the switch and its state. This message comes in the following types:

- **OFPST_DESC** contains information about the switch, such as its serial number or manufacturer description.
- **OFPST_FLOW** contains information about a specific entry of the switch's flow table. This includes the duration for which the rule has been active when the OFPT_STATS_REPLY message was generated, as well as byte and packet counters.
- **OFPST_AGGREGATE** contains aggregated information about flow entries in the switch's flow table.
- **OFPST_TABLE** contains the flow entry information of all entries in a specific flow table of the queried switch.
- **OFPST_PORT** contains information about a physical port of a switch.
- **OFPST_QUEUE** contains information about a specific queue.

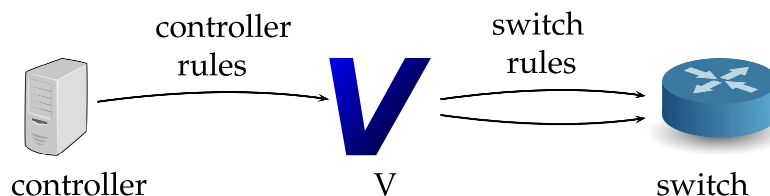

Figure 2.1: Controller and Switch Rules.

As the switch rules in **V** can represent a combination of rules that might stem from different controllers, these messages (except for OFPST_DESC) have to be modified before they are forwarded to the controller that requested the statistics. This is done, so that the response contains all flows that the controller installed but no flows that foreign controllers installed. The packet and byte counter values are set to the values they would carry in a single-controller network with equal traffic.

2.3.4 Dynamic Slices - Reservations

In FlowVisor every controller is associated with a slice. This slice specifies for which part of the traffic a controller is responsible, thereby determining: *(i)* which OFPT_PACKET_IN the controller receives and *(ii)* how the flowspace of OFPT_FLOW_MOD messages need to modified before the rule is installed on the switch[12]. The slices in FlowVisor are defined at configuration time and stay constant unless changed by the network administrator. This implies a limitation on the capabilities of the controller, as the packets need to stay within the given slice as they progress through the network. Actions that rewrite header fields and push the packet out of the controller's slice are thus being rejected by FlowVisor.

V overcomes this limitation by introducing a dynamic datapath-dependent slicing[13]. At configuration time the network administrator defines which part of the **incoming** traffic shall be handled by which controller. This statically defined slice is **valid for border ports only**. Border ports be defined as physical ports that connect the administration domain to other networks. For internal ports the slices are dynamically generated by **V**. To guarantee a correctly functioning network **V** needs to ensure that all legs of the path[14] that a packet transverses, have been installed by the same controller[15]. **V** ensures this by maintaining reservations, as explained below.

[12] The rule might overlap with different flowspaces in the controller's slice. Sometimes the resulting validity space cannot be expressed in a single OFPT_FLOW_MOD message and multiple messages need to be generated.

[13] FlowVisor also supports datapath dependent slices, yet they are static in nature as they are determined by the network administrator at configuration time and not by the controllers at runtime.

[14] It might better be called "the list of applied actions", as it also includes packet header rewrites, but the path seems to be a clearer expression.

[15] There could be cases in which one controller might want to hand over responsibility of a packet to another controller. Yet it is outside the scope of this thesis.

Assume that a controller installs a rule ϕ_i:

$$\phi_i = (\text{fs}_i^\phi, \text{act}_i^\phi(\cdot)) \tag{2.1}$$

that is valid for the flowspace $\text{fs}_i^\phi \in \mathcal{H}$, where $\mathcal{H}$ is the header space spanned by the header dimensions, and action function $\text{act}_i^\phi(\cdot)$ that is defined for every packet $p_j \in \text{fs}_j^\phi$.

Upon reception of such a controller rule **V** calculates[16]:

$$\widehat{\text{act}}_i^\phi(\text{fs}_i^\phi)), \tag{2.2}$$

that does not take a packet as input value, but a flowspace instead. The mapping function $\widehat{\text{act}}_i^\phi(\cdot)$ is chosen so that[17]:

$$\forall p_i \in \text{fs}_i^\phi : \widehat{\text{act}}_i^\phi(p_i) = \text{act}_i^\phi(p_i). \tag{2.3}$$

By doing so, **V** evaluates how the switch maps the ingress flowspace to an egress flowspace and how it "forwards" it to the next hop[18]. **V** uses this to install reservations for the egress flowspace on the next hop, thereby building dynamic slices. The next hop in this context refers to the switch's physical input port that is connected to the port through which the flowspace is "forwarded."

The dynamic reservations determine which part of the traffic a controller is allowed to handle on internal ports. If the controller installs a rule that is valid for an internal port without having a reservation of at least part of the requested flowspace, no rule is pushed to the switch. It is only when an according reservation is made, that the rule is being activated[19]. This activation leads to a reservation on the next hop and possibly to subsequent activations.

16 Be aware that this is an engineering solution and would lead to some dispute with the maths department (if presented).

17 This works because the exact header of a specific packet forms a degenerated flowspace, i.e. a point in the 12-dimensional header space.

18 For a more profound treatment of transfer functions for flowspaces refer to [3].

19 This implies that physical paths are always established from ingress to egress, even if the controller installed the rules in reverse order.

As **V** allows multiple controllers to share the same flowspace in a dynamic way, it needs to ensure that the traffic belonging to different controllers is not merged at any time. Was a switch to forward overlapping flows from different in-ports belonging to different controllers to the same destination port, the next-hop-switch would not be able to tell these flows apart, because the discriminating factor in-port would have been lost. This Y-Merge is illustrated in Figure 2.2. Be aware that no Merge is performed, if the in-ports of the flows are equal.

In consequence reservations made by controllers must not overlap, as they would otherwise perform a Y-Merge. This can be ensured by checking for colliding reservations before a new reservation is made. If such a conflict is found, **V** checks whether the reservation has been made by a different controller and whether the in-ports for which the rules that trigger the reservations differ. If controller and port are different, a Y-Merge would occur.

This can be overcome by finding a flowspace-volume of equal size that is currently unused and reserving this flowspace instead. On the sending switch the header field values are modified so that the packets fall into the substitution-flowspace. Before the packet leave the network the packet header is set to the original value. This can be done on the border switch just before the packets leave the network. The flows are then still being merged, but the merge occurs on the border of the network domain.

Depending on the OpenFlow version, temporary mapping of the packets into a different flowspace-volume can be realized in different ways.

OpenFlow 1.0

OpenFlow 1.0 [4] only supports packet header rewrites and does not allow for the addition of extra IEEE 802.1Q headers. By setting a specific header field to a defined value, the previously defined information is lost. That is a reversible process if **V** can deduce the overridden information, because the original values can be set again on the receiving switch (before the packet is forwarded by the receiving switch). Thankfully in most practical cases **V** can obtain such information.

In OpenFlow 1.0, specification of a rule's flowspace is done by fixing one or more dimensions to a specific value, the only exception being the IP source and IP destination dimensions which support CIDR wildcarding[20]. The set of fixed dimensions in the flowspace fs_i^{χ} of the rule χ_i are potential rewrite candidates, because they provide such information. **V** varies the values in these dimension to find a flowspace-volume for which no Y-Merge conflict occurs. By only varying these dimensions and not fixing any further or relaxing the restrictions, **V** ensures that the specified volume of the flowspace is mapped to another volume of equal size. It would be desirable to use as few header field rewrites as possible[21].

If such a volume cannot be found or the rule does not fix any dimensions to a specific value, **V** falls back to discovering the traffic characteristics and pushing packets to a volume for which the controllers might have specified actions, but which is not being used, because the traffic source does not generate such packets. One example for this case is a situation in which the controller installed a rule which is valid for all VLAN ids, but the network only uses a subset of these. This procedure is discussed further in Section 2.3.4.

If such a volume has been found, **V** installs a reservation for the original volume on the receiving switch and indicates which header fields have been rewritten. It also installs a "block reservation" for the remapped volume. It does so, because this volume is now being supplied with traffic and its usage by other flows would merge data streams that need to be kept apart.

Suppose a controller made a reservation that had to be mapped to a different flowspace-volume. If this controllers now installs a rule for that flowspace, the resulting flowspace of the sliced rule is calculated in two steps. First the intersection of the controller rule's flowspace with the reservation rule's is calculated. Then the mapping function which was used when finding a substitution flowspace is applied to this result. The action list of the sliced rule is generated by pushing the rewrite rules, which set the original values of the header fields, to the left of the controller rule's action list[22].

[20] Special caution needs do payed to the Ethernet type field, as it influences the switches interpretation of the packet headers.

[21] Implementing such an algorithm is left as an exercise to the gentle reader.

[22] If the controller rule specifies a rewrite on a header field, the according remapping-rewrite action can be omitted.

OpenFlow 1.0: Exploiting Traffic-Source-Characteristics

If installation of a rule would lead to a Y-Merge conflict and no available volume in the flowspace can be found, the characteristics of the traffic source can be exploited to find a space which is currently not being used.

Let Φ be the set of switch rules that are installed on a specific switch[23]. The set $\Omega_i \subset \Phi$ of switch rules which overlap with the new rule χ_i and forward traffic to the same physical port, but belong to a different controllers, contains all potential conflicting rules.

If the set is non-empty, packets belonging to χ_i need to be tunneled to the receiving switch to avoid Y-Merges. This can be done by mapping the VLAN ids[24] that are being used by the traffic source feeding χ_i to VLAN ids, that currently are not being used by any traffic sources related to flows in Ω_i[25].

I propose a dynamic tunneling method, that chooses a VLAN id at random, establishes the tunnel from the sending to the receiving switch and can change the used VLAN id, if as traffic source starts using this VLAN id.

Tunnel-Init Rule Upon reception of a controller rule χ_i, that fulfills the foregoing conditions, **V** installs a rule (tunnel-init rule) on the sending switch that is valid for the flowspace of χ_i and forwards the packets to **V**. That way **V** can determine which VLAN ids are used by χ_i and set up a tunnel for each of these VLAN ids.

$$\begin{aligned} &\text{tunnel-init (sending switch)} : \{ \\ &\quad \text{fs}^{26} \quad : \text{fs}_k^{\omega} \cap \{\text{VLAN_id} : v\} \\ &\quad \text{act}^{27} \quad : \text{“forward to V”} \\ &\quad \text{prio}^{28} \quad : p_{init}\} \end{aligned} \tag{2.4}$$

[23] To ease notation, the switch specific index is omitted.
[24] The VLAN id is just one of 12 dimensions that can be used to tunnel traffic.
[25] Note that the rules in Ω_i, for which the set of used VLAN ids is unknown, cannot overlap. If they would, the following algorithm would have been executed at installation time of the rule that causes the overlap. Thereby the used VLAN ids would have been determined.

Tunnel-Conflict Rule Triggered by a packet with VLAN id v_i^{orig} originating from the tunnel-init rule, **V** chooses a random VLAN id v_i^{tunnel} and establishes a tunnel. The first step in doing so, is setting up a tunnel-conflict rule that forwards packets to **V**, should an already existing rule use this VLAN id. If such a packet arrives at **V**, the tunnel is moved to a different VLAN.

$$\begin{aligned} &\text{tunnel-conflict}_k \text{ (sending switch) : } \{ \\ &\quad \text{fs} \quad : \text{fs}_k^{\omega} \cap \{\text{VLAN_id} : v_{tunnel}\} \\ &\quad \text{act} \quad : \text{“forward to V”} \\ &\quad \text{prio} \quad : p_{conflict} > p_{init}\} \end{aligned} \tag{2.5}$$

Tunnel-End Rule A tunnel-end rule is installed on the receiving switch, that forwards the traffic of the corresponding flowspace to **V**. The VLAN ids of OFPT_PACKET_IN messages generated that way have to be "unmapped" by **V** before they are delivered to the controller. The rules that are installed by the controller in return have to be modified so that they are only valid for the chosen VLAN ids and they have to be expanded when new VLAN tunnels are installed for χ_i.

$$\begin{aligned} &\text{tunnel-end}_k \text{ (receiving switch) : } \{ \\ &\quad \text{fs} \quad : \text{fs}_k^{\omega} \cap \{\text{VLAN_id} : v_{tunnel}\} \\ &\quad \text{act} \quad : \text{“forward to V”} \\ &\quad \text{prio} \quad : p_{conflict} > p_{init}\} \end{aligned} \tag{2.6}$$

Tunnel-Start Rule The rule (tunnel-start) that is installed on the sending switch has to match against the flow space of ψ and the original VLAN id v_{orig}, it modifies the VLAN id to v_{tunnel} and forwards the packet to the receiving switch.

$$\begin{aligned} &\text{tunnel-start}_k \text{ (sending switch)} \{ \\ &\quad \text{fs} \quad : \text{fs}_i^{\chi} \cap \{\text{VLAN_id} : v_{orig}\} \\ &\quad \text{act} \quad : \text{“set VLAN id: } v_{tunnel}\text{”, “deliver”}\} \end{aligned} \tag{2.7}$$

[26] Flowspace of the newly resulting rule.
[27] Action List of the resulting rule.
[28] Priority of the resulting rule.

Tunnel-Established Message As there is no guarantee that the randomly chosen VLAN id v_{tunnel} is not already in use, packets might apply to the tunnel-end rule, that are not caused by the tunnel-start rule. Forwarding these packets to the controller which installed the rule χ_i, would be incorrect. Therefore **V** injects a packet (tunnel-established) with a specific payload into the tunnel after installing the tunnel-start rule. OFPT_PACKET_IN messages that are triggered by the tunnel-end rule prior to observation of the tunnel-established message, are dropped.[29] OFPT_PACKET_IN messages arriving after that packet are delivered to the controller.

Relocation of the Tunnel Upon observation of a tunnel-conflict packet, the tunnel is moved to another VLAN id. After choosing a new random VLAN id v', a corresponding tunnel-conflict rule is installed on the sending switch. On the receiving switch a tunnel-end rule is installed for the new VLAN id v' and a tunnel-established message is injected. A tunnel-mod rule is installed on the receiving switch, which temporarily drops packets belonging to the old tunnel. Thereafter existing controller rules that treat packets originating from the old tunnel, are modified so that they now apply to VLAN v'. The tunnel-mod as well as the old tunnel-conflict rules can be removed after the update procedure is terminated and the tunnel-established message is received.

OpenFlow 1.1, 1.2 and 1.3

OpenFlow 1.1-1.3 [6, 7, 5] allow the controller to add extra VLAN headers (VLAN encapsulation through tag pushing). This feature might be used to perform more efficient tunneling. Yet the problem of scanning the traffic for unused VLAN ids prevails. The addition of VLAN headers does not replace the remapping to unused space described in Section 2.3.4, because if the sending switch adds a VLAN header, the flow table matching algorithm on the receiving switch does not match against any of the inner VLAN id headers. Said in other words, this feature does not introduce another independent dimension that can be used to discriminate flows, because the information about the original VLAN id is being hidden. As this might potentially be changed in an upcoming version of OpenFlow, evaluation has to be postponed.

[29] One might be able to deliver these messages with some extra care. How this is to be realized is part of future work.

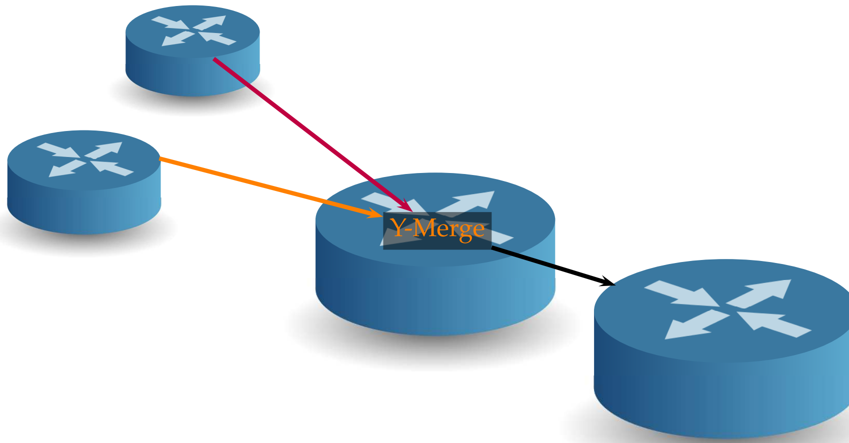

Figure 2.2: Two controllers install rules on a switch for an overlapping flowspace and different inports. A Y-Merge occurs because the receiving switch is not able to tell which packet belonged to which controller.

2.4 Architecture

To implement the described concept I used the architecture of FlowVisor as a basis. While using large parts of its code, the internal structure had to be changed completely. This is mainly due to the fact that the controller's slice is constant in FlowVisor, but dynamic and datapath dependent in **V**. The resulting architecture is depicted in Figure 2.3.

2.4.1 Topology Discovery

It is critical to know the correct topology of the network, because translation from controller rules to switch rules depends on whether the physical port associated with a rule, is an internal or external port. Moreover generating the correct reservations requires the translation of "forward to out-port x" to "forward to in-port y of datapath z", which again requires knowledge of the network's topology.

This can be realized, as is done as an optional feature in FlowVisor, by sending LLDP [17] messages through all physical ports (OpenFlow message OFPT_PACKET_OUT) of all switches that register at **V**. On reception of a corresponding OFPT_PACKET_IN, the sending datapath id and the sending physical port can be extracted from the message's payload.

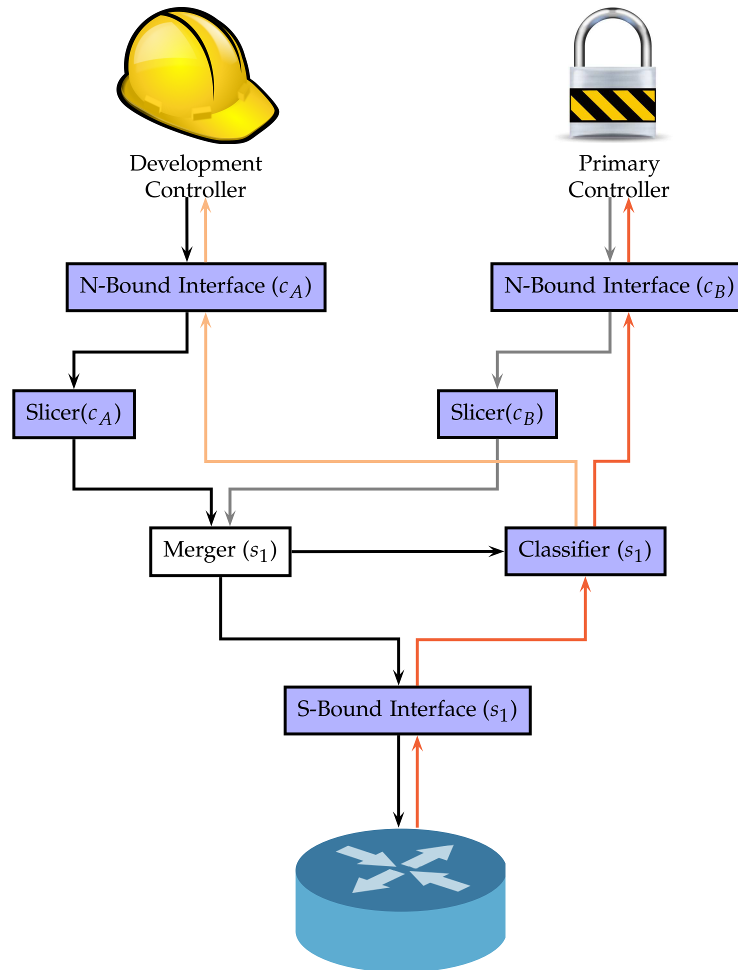

Figure 2.3: Architecture of **V**. The individual controllers are connected via a North Bound Interface to the Slicer, which limits the scope of the rules to at most the controller's slice (on border ports) or the controller's reservations (on non-border ports). This can result in multiple rules, which are called "sliced rules". These are forwarded to the Merger unit which ensures that the sliced rules of the individual controllers are merged in a specific way and informs the Classifier about the reservations made. The Classifier forwards OFPT_PACKET_IN messages received from the switches to controllers which own according slices/reservations, possibly duplicating the message if multiple controllers are responsible for it.

2.4.2 North Bound Interface

The North Bound Interface opens up a TCP connection to the controller. It translates received OpenFlow messages into internal objects and vice versa. One thread per controller is used so that multiple requests, sent by different controllers can be processed in parallel (subject to the Operating System's multithread handling).

2.4.3 Slicer

The Slicer receives the controller rule and translates it into a set of sliced rules. Depending on the state of the physical port for which the rule is valid (connected to an internal datapath / connected to an external datapath) the rule is treated differently. A rule can either be valid for a specific port or it can be valid for *all* ports of a switch. In the latter case one rule that handles the internal ports (assuming that a datapath always has more internal ports than external ports) and one rule for every external port of the datapath needs to be installed. Even though this seems to be a huge overhead, today's switch implementations (e.g. NEC IP8800/S3640-24T2XW) do not fully support wildcarded in-ports. They explode such rules internally by expanding a rule for every physical port of the switch. If the expansion (installing one rule per port) is done in **V** and not in the switch, the control traffic increases, but the number of flow table entries stays the same.

External Port On reception of a controller rule which is valid for an external physical port, the Slicer calculates the intersection of the rule's flowspace with the controller's slice, thereby obtaining a new set of rules (hereafter called sliced rules), just as FlowVisor would do. This ensures that the controller does not treat traffic which might belong to other controllers. The slice of a controller is statically defined by the network administrator and in the case of **V** defines which part of the incoming traffic a controller is allowed to handle.

Internal Port Handling of controller rules which are valid for internal physical ports is very similar. The only difference is, that instead of the controller's static slice, dynamic reservations are used to calculate the set of sliced rules.

2.4.4 Merger

The Merger combines the sliced rules of the individual controllers in a way that ensures isolation of the controllers. For a specific packet p_i (a point in the 12-dimensional flowspace) at port q_i on datapath d_i, a set of controllers C_{p_i} are privileged to specify an action. Calculation of this set C_{p_i} depends on whether q_i is an internal or an external port.

If q_i is an external port, the set of controllers C_{p_i} is equal to the set of controllers in whose slice p_i is contained.

$$\text{if } q_i \text{ external: } C_{p_i} = \{c_j | c_j \in \mathcal{C}, p_i \in \text{slice}(c_j)\} \tag{2.8}$$

where $\mathcal{C}$ is the set of all controllers and $\text{slice}(c_j)$ represents the slice (set of flowspaces) that are associated with controller c_j.

If q_i is an internal port (connected to another OpenFlow switch in the administration domain) the set of controllers which are allowed to specify an action is determined by the set of reservations: a controller c_j is only allowed to specify actions for a packet that was in its slice when it entered the network and was forwarded by it to port q_i of datapath d_i.

$$\text{if } q_i \text{ internal: } C_{p_i} = \{c_j | c_j \in \mathcal{C}, p_i \in \text{reservation}(c_j, d_i, q_i)\} \tag{2.9}$$

where $\text{reservation}(c_j, d_i, q_i)$ is the set of reservations made by controller c_j for port q_i on datapath d_i.

Calculation of the Action List To guarantee that the requests of all controllers are fulfilled, the Merger needs to ensure that the switch executes all actions defined by the controllers C_{p_i} for packet p_i. The corresponding action list is calculated by merging the action lists specified by the controllers C_{p_i}. If the action list of a controller does not modify a packet header, the elements of $\text{action}(p_i, c_j)$ are pushed to the left of $\text{action}(p_i)$ – the overall action set of p_i – if they do not already exist in the list. If header fields are modified, the elements are appended to the action list $\text{action}(p_i)$.

This concept does have one limitation: modification is done in sequence and therefore the second rewrite operates on an already modified packet. Hence **V** currently does not support header rewrites on different header fields by different controllers. To overcome this limitation the packet would need to be duplicated before it is modified. The OpenFlow Specifications 1.2 and 1.3 [7, 5] seem to allow this by introducing loop-back devices[30].

Unresponsive Controllers Be aware that if a controller did not explicitly specify an action by installing a rule whose flowspace contains p_i it assumes the switch to execute the default action for a packet miss in the flow table. What that is, depends on the switch's configuration and can either mean that the packet is to be dropped or to be forwarded to the controller[31] [32].

Generalization The forgoing explanation of the Merger's functionality relied on one packet, meaning one point in the flowspace. Yet a controller sends an OFPT_FLOW_MOD message it specifies a list of actions valid for a whole flowspace[33]. As the header space is discrete the flowspace can be interpreted as a set of points. An algorithm that realizes the goals given, therefore installs a set of rules (switch rules) on the switch which ensure that for every point in the flowspace the forgoing developed action list is being executed. With the OpenFlow Specifications 1.0 [4] this is a painful task, yet the Specifications 1.1, 1.2 and 1.3 possibly allow a very simple implementation.

[30] One will have to wait for a first implementation to further validate the performance issues connected to such an approach.

[31] This assumes that the controller in question will eventually install a rule to treat the packet. If it does not, **V** and the controller would be flooded with OFPT_PACKET_IN messages. **V** is therefore falling back on FlowVisor default behavior for the case that a controller does not respond on OFPT_PACKET_IN messages, by assuming a temporary drop action for the packets.

[32] This describes OpenFlow 1.0 behavior. Later versions allow for greater flexibility.

[33] I am calling it flowspace to be consistent with the terminology used in other papers, yet I'd prefer calling it a flowvolume, because minimal and maximal values are defined for every dimension.

OpenFlow 1.0

In OpenFlow 1.0 [4] the controllers can only access one logical flow table of the switch, which is composed of different smaller flow tables. Yet no access to these flow tables is given. As a result, the different flow tables defined by the controllers need to be projected to a single one. Therefore overlapping rules with differing action lists, installed by different controllers need to be handled specially. By imposing an order on the OFPT_FLOW_MOD messages[34] the problem reduces into developing a streaming algorithm. Upon reception of an OFPT_FLOW_MOD message it modifies the current set of switch rules, so that after the transition period the new set of switch rules satisfy the requirements described in Section 2.4.4; assuming that the set of switch rules before the transition did so.

To realize this the Merger maps the sliced rules $\Psi = \{\chi_i\}$ to a set of switch rules $\Phi = \{\psi_i\}$.

$$\text{Merger} : \Psi \rightarrow \Phi \tag{2.10}$$

Assume that Φ only contains rules that were generated by this algorithm. At the startup of a switch, no rules are installed and the empty set $\Phi = \emptyset$ clearly satisfies this condition.

Installation of rule $\chi_i = (\text{fs}_i^\chi, \text{prio}_i^\chi, \text{act}_i^\chi)$ by controller contr_i^χ with flowspace fs_i^χ, priority prio_i^χ and action list act_i^χ can only lead to conflicts with rules whose flowspace overlap with fs_i^χ. A rule does not apply to packets outside its flowspace and thus rules whose flowspace do not overlap with the one of χ_i are not influenced by the installation of the rule.

The new rule might therefore interfere with any rules in Ω:

$$\Omega = \{\omega_k | \omega_k \cap \chi_i \neq , \omega_k \in \Phi\}. \tag{2.11}$$

[34] If **V** is running on a multi-interface, multi-core computer, two OFPT_FLOW_MOD messages from different controllers can be accepted at the same time. Yet the Merger unit is switch specific and can only treat one message per switch at the time.

Sorting this set by descending priority and controller id[35] using the sorting functions $\sigma_k(\Omega)$ yields:

$$\Theta = (\sigma_1(\Omega), \sigma_2(\Omega), \ldots \sigma_n(\Omega)). \tag{2.12}$$

This list is split into two parts:

(i) rules Θ^h with higher or equal priority and

(ii) rules Θ^l with lower priority than prio(χ_i).

These lists have to be treated differently, because higher priority rules can starve the newly installed rule, whereas rules with lower priority could be starved by it.

(i) Higher or Equal Priority Rules Θ^h A higher priority rule that was installed by a different controller and overlaps with the newly installed rule, handles packets, that would be handled by χ_i in a single controller network. Therefore **V** has to install a rule that executes both actions for the overlapping flowspace.

For equal-priority rules the OpenFlow Specifications leave the selection of the rule-to-be-used to the switch. Because only one rule is executed, equal priority rules for overlapping flowspaces necessarily starve each other and the same algorithm has to be used as is for higher priority rules.

The proposed algorithm goes through the elements $\theta_j \in \Theta^h$ in reverse order and handles the following cases:

(i.i) $\theta_j \in \Psi$ is a controller rule, installed by the same controller as χ_i. Starvation of χ_i by θ_j is expected by the controller and no modification is necessary.

[35] The controller that installed χ_i comes last.

(i.ii) $\theta_j \in \Psi$ is a controller rule, installed by a different controller as χ_i. In this case, starvation is not expected by the controller and **V** needs to install a *intersection-rule* ω_k that is valid for the overlapping flowspace and executes both action sets. The priority has to be chosen to be higher than the priority of θ_j, but lower than the one the next higher-priority rule with whose flowspace it overlaps. The possible priority range can therefore be determined on the fly.

$$\begin{cases} \text{fs}_k^\omega & = & \text{fs}_j^\theta \cap \text{fs}_i^\chi \\ \text{act}_k^\omega & = & \text{act}_j^\theta \cup \text{act}_i^\chi \\ \text{prio}_j^\theta & < & \text{prio}_k^\omega < \text{prio}_l^\theta \\ \min\limits_{l>j} & & \left(\text{fs}_k^\omega \cap \text{fs}_l^\theta \neq \emptyset\right) \end{cases} \tag{2.13}$$

(i.iii) $\theta_j \notin \Psi$ is a intersection-rule, unrelated to $contr_i^\chi$. The intersection-rule θ_j originates from multiple rules that were installed by other controllers and would therefore illegitimately starve rule χ_i. Thus the conclusions of (i.ii) apply.

(i.iv) $\theta_j \notin \Psi$ is a intersection-rule, related to $contr_i^\chi$. Suppose that θ_j originates from the intersection of the rule $\hat{\chi}_k$ with a set of other rules, where $\hat{\chi}_k$ was installed by the same controller as χ_i. The newly installed rule is legitimately starved if the priority of $\hat{\chi}_k$ is larger than or equal to the one of χ_i. If that is the case no additional action has to be taken.

If the priority of $\hat{\chi}_k$ is lower another intersection-rule has to be installed, because otherwise the rule would be starved by a lower priority rule of the own controller. This rule has to be valid for the overlapping flowspace. The action set is generated by replacing the actions of $\hat{\chi}_k$ with the actions of χ_i. The priority limitations are the same as in (i.ii).

$$\begin{cases} \text{fs}_k^\omega & = & \text{fs}_j^\theta \cap \text{fs}_i^\chi \\ \text{act}_k^\omega & = & (\text{act}_j^\theta \setminus \text{act}_k^{\hat{\chi}}) \cup \text{act}_i^\chi \\ \text{prio}_j^\theta & < & \text{prio}_k^\omega < \text{prio}_l^\theta \\ \min\limits_{l>j} & & \left(\text{fs}_k^\omega \cap \text{fs}_l^\theta \neq \emptyset\right) \end{cases} \tag{2.14}$$

(ii) Lower Priority Rules Θ^l Installation of rule χ_i can lead to illegal starvation of lower priority rules that were installed by other controllers. To guarantee execution of all actions, **V** iterates through the elements of $\theta_j \in \Theta^l$ and possibly encounters the following cases:

(ii.i) $\theta_j \in \Psi$ is a controller rule, installed by the same controller as χ_i. In this case, starvation of the rule is expected by the controller and nothing has to be done.

(ii.ii) $\theta_j \in \Psi$ is a controller rule, installed by a different controller as χ_i. Starvation of the lower priority rule of the other controller is not legitimate. Therefore **V** needs to install a rule for the flowspace $\mathrm{fs}_i^\chi \cap \mathrm{fs}_j^\theta$ with the action set $\mathrm{act}_i^\chi \cup \mathrm{act}_j^\theta$. The priority has to be chosen so that it is higher than the priority of χ_i, but lower than any higher-priority rule with whose flowspace it would overlap.

$$\begin{cases} \mathrm{fs}_k^\omega & = & \mathrm{fs}_j^\theta \cap \mathrm{fs}_i^\chi \\ \mathrm{act}_k^\omega & = & (\mathrm{act}_j^\theta \cup \mathrm{act}_i^\chi) \\ \mathrm{prio}_j^\theta & < & \mathrm{prio}_k^\omega < \mathrm{prio}_l^\theta \\ \min\limits_l & & \left(\mathrm{fs}_l^\theta \cap \mathrm{fs}_k^\omega \neq \emptyset\right) \end{cases} \tag{2.15}$$

(ii.iii) $\theta_j \notin \Psi$ is a intersection-rule, which does not contain actions defined by $contr_i^\chi$. There is no difference regarding the treatment of this case and (ii.ii). Therefore the same actions apply.

(ii.iv) $\theta_j \notin \Psi$ is a intersection-rule, which does contain actions defined by $contr_i^\chi$. In this case it is legal to starve the own rule, but not the foreign rules. Therefore FlowVisor installs a rule for the overlapping flowspace $\mathrm{fs}_i^\chi \cap \mathrm{fs}_j^\theta$ with a modified action set. The action set that the specific controller defined previously is replace by the action set of the new rule.

$$\begin{cases} \mathrm{fs}_k^\omega & = & \mathrm{fs}_j^\theta \cap \mathrm{fs}_i^\chi) \\ \mathrm{act}_k^\omega & = & (\mathrm{act}_j^\theta \setminus \mathrm{act}_k^{\hat{\chi}}) \cup \mathrm{act}_i^\chi \\ \mathrm{prio}_j^\theta & < & \mathrm{prio}_k^\omega < \mathrm{prio}_l^\theta \\ \min\limits_l & & \left(\mathrm{fs}_l^\theta \cap \mathrm{fs}_k^\omega \neq \emptyset\right) \end{cases} \tag{2.16}$$

OpenFlow 1.1, 1.2 and 1.3

The OpenFlow Specifications 1.1 introduce the concept of pipelining, thereby granting access to the various flow tables of a switch that were logically merged in OpenFlow 1.0. It allows controllers to install rules into specific flow tables and to specify the action "go through flow table x" in the action list of every flow table entry. This provides a convenient solution of the problem the Merger unit is facing. The extra dimension that is introduced by adding multiple controllers to OpenFlow 1.0 – which was arguably designed for one controller only – can than be accounted for with the flow table dimension.

Herein the available flow tables are split among the controllers that share a specific flow space on a switch. This reduced number of flow tables is reported to the controllers[36]. The pipelines that the controllers install are joined by **V** into a single pipeline. A packet that arrives at the switch would therefore pass through all controller-defined-pipelines. Hence all desired actions would be executed.

2.4.5 Classifier

The Classifier of **V** works in a very similar way as the Classifier of FlowVisor does. It takes an internal object representing either an OFPT_PACKET_IN or an OFPT_STATS_REPLY message as input value, potentially modifies it and forwards it to the "desired" controller. The only different between the Classifier of **V** and FlowVisor is, that in **V** the database that determines what happens with the object changes at runtime. For FlowVisor that is not the case, because the slices for both internal and external ports are defined by the network administrator at configuration time.

[36] The number of controllers that share a specific flowspace is dynamic. Therefore the number of available flow tables does change and this change would need to be reported to the controller. Yet the number of flow tables of a physical switch is constant in nature and no change is expected. This is an open point, which either needs some lobbying for the next Specifications or some creative idea once someone implemented any of the current implementation proposals.

OFPT_PACKET_IN

Depending on whether the port from which the OFPT_PACKET_IN messages stems is an internal or external port, the Classifier either uses the dynamically defined reservations or the static controller slices to calculate the set of controllers which can potentially handle the message. Associated with each reservation and slice entry are the set of controller rules which a controller installed for the flowspace-in-question. If a controller already installed a rule that would handle the packet, it does not expect to see the OFPT_PACKET_IN message and therefore these controllers are removed from the set. In consequence a controller obtains a packet if it is allowed to see it and has not already defined an action for it.

If **V** installed a rule that explicitly generates OFPT_PACKET_IN messages because one of the controllers did not define an action for the specific flowspace, the reason-flag in the OFPT_PACKET_IN message needs to be changed.

OFPT_STATS_REPLY

A switch replies with an OFPT_STATS_REPLY message of a controller requested statistical information. The reply does not state which controller sent this request and **V** therefore keeps track of the requests that were sent. This is being done by the Slicer while the information is shared with the Classifier. Upon reception of such an OFPT_STATS_REPLY message, the Classifier checks for open requests and calculates the controller specific statistics which are then being sent to the controller(s).

2.4.6 South Bound Interface

The South Bound interface consists of a Socket Acceptor Thread that listens to a predefined port and generates a thread for every switch that connects to it. After the link has been established, the switch-specific thread creates a new Classifier instance which in turn connects to all North Bound Interfaces of the associated controllers.

2.5 Implementation

Most parts of what is described in the sections above has been implemented and tested using a research prototype of **V**. Be aware that **V** currently is not a ready-to-use piece of software and should *definitely* not be run in an production environment. It requires a special start sequence and does not clean up after itself. Also changes in the configuration have to be made directly in the configuration file and **V** needs to be restarted after every change in the configuration[37]. That said, the implementation runs smoothly and no bugs that would interfere with **V**'s functionality are known.

2.5.1 External Sources Used

V heavily relies on the Java implementation of FlowVisor 0.83 which is publicly available via hg-git mercurial [18] on `https://bitbucket.org/onlab/flowvisor` and comes with the license agreement shown in Figure **??**. Refer to [19] for its documentation.

Even though the source code of FlowVisor was a very good starting point, because a lot of functionality could be reused, the internal structure had to be changed fundamentally. In FlowVisor Slicer and Classifier both operate on their own static copy of the controller's slice. In **V** this slice is being dynamically defined by the Slicer. The Classifier needs access to this information to calculate the set of controllers to which a specific message shall be forwarded. Thus Slicer and Classifier need to share the database[38].

Moreover the Merger unit which ensures that the rules of different controllers do not starve each other is not present in FlowVisor and had to be implemented from scratch. It implements a slight variation[39] of the algorithm described in Section 2.4.4.

---

[37] Featuring online modification of the configuration would require further implementation work without providing further insight or proof.

[38] They could also communicate in order to build up two separate databases with equal contents. This would allow for a decrease in the average time consumed by database blocks, but comes at the cost of increased complexity.

[39] The implementation is able to modify existing flow tables entries, whereas the described algorithm, would always install new rules; even if the flowspace of the new rule and the rule-to-be-starved are equal.

Copyright (c) 2008 The Board of Trustees of The Leland Stanford Junior University
We are making the FlowVisor specification, code, and associated documentation (Software) available for public use and benefit with the expectation that others will use, modify and enhance the Software and contribute those enhancements back to the community. However, since we would like to make the Software available for broadest use, with as few restrictions as possible permission is hereby granted, free of charge, to any person obtaining a copy of this Software to deal in the Software under the copyrights without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions: The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
The name and trademarks of copyright holder(s) may NOT be used in advertising or publicity pertaining to the Software or any derivatives without specific, written prior permission.

Figure 2.4: License Agreement of FlowVisor.

2.5.2 Starting Sequence

V currently is a research prototype and not all eventualities have been accounted for. This imposes a requirement for specific order on some actions. The most important of which is the starting sequence:

1. Power up the network, if not already running. If you are running mininet [20] to simulate your network make sure that the switches are connecting to **V** and not directly to the controllers.
2. During the start-up period the switches exchange a lot of messages, that would only confuse **V**. So wait for 2-3 seconds until the transition period is over and the initial message-exchange phase has been completed[40].
3. Start **all** controllers defined in the configuration. They need to be available when **V** is being started. If any of the controllers is not reachable **V** generates a lot of status-querying messages that consume a lot of resources.
4. Start **V** and be patient. When a switch connects to **V**, topology discovery is started and the connection to the controllers is only made, once the topology has been discovered (safety timeout 5 seconds).
5. Now your all set and ready to go. Have fun!

[40] If **V** receives a packet while the Classifier Thread creates the Slicer and North Bound Interface Threads, it only forwards the message to the controllers whose Threads are already running, but not to all controllers which should receive the packet. In a production implementation the Classifier should not accept any messages during the bootstrap period.

2.6 Proof of principle

The implementation of **V** has been tested with two different topologies, that target different functionalities of **V**. Validation of the implementation has two parts: *(i)* validating, whether the implementation is a correct representation of the algorithm described in this thesis and *(ii)* validating, whether the algorithm is correct and provides the properties it aims to fulfill.

To evaluate this, two different network topologies with different properties were simulated in mininet [20]. In these simulations the switches build up a connection to **V** which in turn connects to the two controllers. One of these controllers is configured to be a primary controller and the other to be a development controller. Their slices cover the full flowspace, which means that both controllers are able to define a path for very packet that enters the network.

2.6.1 Validating the Implementation

To validate whether the implementation correctly represents the algorithm described, the communication between the controllers and **V** as well as the communication between **V** has been captured. This is depicted in Figure 2.5. With full knowledge of the input and output values, as well as the initial internal state, **V** could be treated as black-box whose inner structure was to be discovered[41]. Discovery of the internal structure of the system has one big advantage over manual message-per-message validation: reverse-engineering yields a structure that can be compared to the structure of the algorithm which is to be implemented. This comparison can be done in mathematical terms, whereas message-per-message comparison requires a manual interpretation of the algorithm and manual validation of its implementation. Such validation might therefore be subject to aberration.

[41] A rather complex system of Hidden Markov Chains might be one such approach.

2.6.2 Validation of the Algorithm

The algorithm claims to:

(i) forward exactly one OFPT_PACKET_IN to a specific controller, for each packet that arrives on an *external* port if and only if the packet falls in the slice of the controller and the controller has not installed a controller rule which handles the packet,

(ii) forward exactly one OFPT_PACKET_IN to a specific controller, for each packet that arrives on an *internal* port if and only if the packet has been forwarded by the controller to that port and

(iii) deliver a packet to an end host if and only if its path was installed by a primary controller.

Claim (i)

To fully validate whether *"exactly one OFPT_PACKET_IN is forwarded to a specific controller, for each packet that arrives on an external port, if and only if the packet falls in the slice of the controller and the controllers has not installed a controller rule which handles the packet"*, one would need to capture the stream of OFPT_PACKET_IN messages that are generated by the switches and pair these messages with the OFPT_PACKET_IN messages that are forwarded to the controllers. For every controller one would then need to specify the point in time for which a controller rule handling the specific packet became active[42]. A validation algorithm would need to check whether **V** forwarded the OFPT_PACKET_IN messages before and stopped doing so after that point in time.

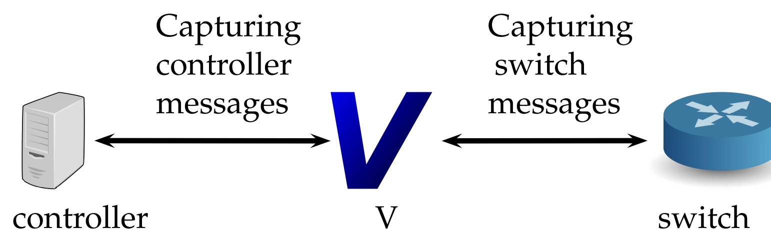

Figure 2.5: Control traffic is captured. **V** is treated as a black-box whose inner structure is to be *recovered* from observed input and output values.

[42] For **V** controller rule becomes valid as soon as it registered the rule in its reservation database.

Out of time considerations I decided only to perform a proof of principle using a simplified scenario. If the rate at which OFPT_PACKET_IN messages are generated is low enough, all queried controllers will have responded by installing an corresponding rule[43] before the next OFPT_PACKET_IN messages is generated. Further the flowspaces of the rules installed by the controllers in this testing scenario fully overlapped as did their slices. Under these conditions, every OFPT_PACKET_IN message is forwarded to every controller exactly once. This reduces validation of the claim to checking whether the number of OFPT_PACKET_IN messages generated by **V** (for external ports) is N times the number of OFPT_PACKET_IN that is received, where N is the number of controllers.

Claim (ii)

Similar to validation of claim (i), one would need a complex validation algorithm to check whether *"exactly one OFPT_PACKET_IN is forwarded to a specific controller, for each packet that arrives on an internal port, if and only if that packet has been forwarded by the controller to that port"*. To do so a stateful validation-system would be needed that identifies the time interval in which a controller is allowed to receive OFPT_PACKET_IN messages from a specific internal port of a datapath. This time interval starts when the controllers installs a rule that generates traffic on the port and ends when the controller installs a rule which treats that traffic.

Under the assumption that the traffic rate is sufficiently low, the controller will install a rule before another OFPT_PACKET_IN is generated and therefore exactly one OFPT_PACKET_IN message is being generated for every internal port that is used during establishment of the path. Thereby validation of this point reduces to comparison of the number of OFPT_PACKET_IN messages originating from internal ports to the number of internal ports that are transversed during establishment of the paths. Ports that are used by u paths need to be counted u times.

[43] This assumes that the controllers do install rules in a reactive manner.

Claim (iii)

Validation of whether *"a packet is delivered if and only if its path was installed by a primary controller"* is not trivial, as it requires information of data plane, which cannot be accessed directly via OpenFlow. Proving this claim would require capturing all packets on all ports as they transverse the network and calculating the path that each packet took through the network. Assuming that a complete list of packets (header fields and payload) could be obtained for every port in the network and the state of the flow tables would be known at every point of time, the associations between the lists could be made and the path that each packet took through the network could be evaluated[44].

Also one could use **Sacramento** – the tool developed in Chapter 3 – to capture the state of the network at every point in time and simulate the path of every packet that entered the network. This would only require capturing of the packets as they enter the network.

Even though this seems to be feasible straight-forward solution, I decided to limit validation to TCP traffic and check

(i) whether the number of packets that entered the network and the number of packets that left the network were equal and

(ii) whether the TCP sequence numbers of packets received by the end hosts were unique.

[44] This is not trivial though. TCP messages can be traced by they sequence number, but it a data source generates identical UDP packets this becomes a tricky task.

2.6.3 Validation Topology I

The first topology is illustrated in Figure 2.6. It consists of five switches and two hosts that are interconnected as illustrated. The primary controller, symbolized by the lock, aims to install the longer path transversing switch 3, while the development controller seeks to install the shorter path that omits switch 3. The dotted lines depict the paths for packets originating from host 1 (bottom right). Be aware that in this scenario, the controllers install rules which forward a packet to the destination depending on the source of the packet. This only works with two hosts in the network, that do not send traffic to themselves. It is a very academic example designed to illustrate nearly all features of **V** with the simplest topology possible. The claims made above were evaluated using the described methods and were found to be true.

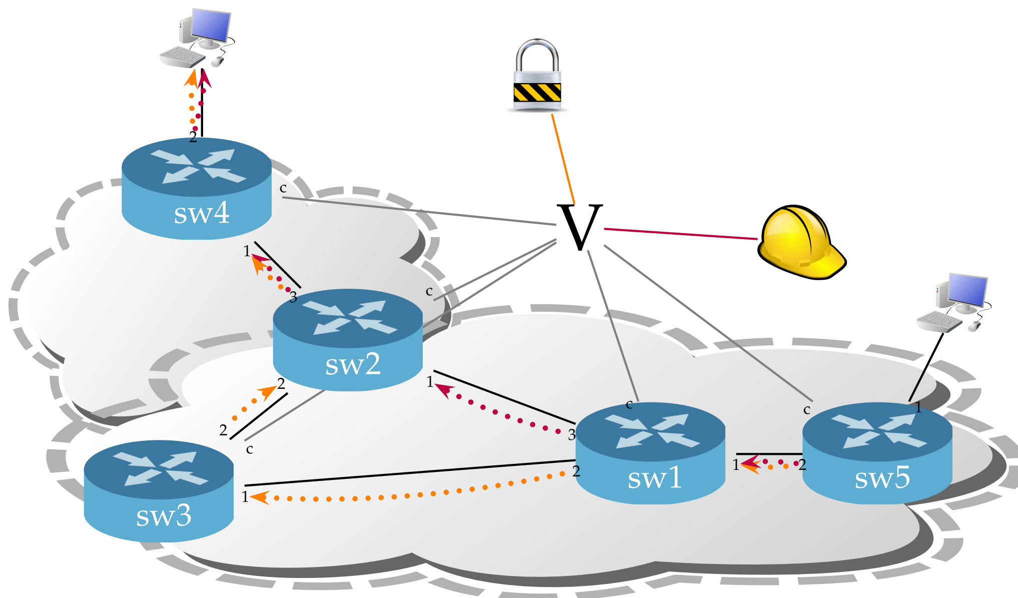

Figure 2.6: Validation Topology 1 – The switches sw1 to sw5 are connected to **V** which in turn is connected to the two controllers. The lock symbolizes a primary controller and the helmet symbolizes a development controller. The dotted orange path depicts what the primary controller aims to install for the packets originating from host 1. The dotted purple path is the respective path that the development controller seeks to install.

2.6.4 Validation Topology II

In Validation Topology 2, illustrated in Figure 2.7, the controllers and hosts are connected in a tree like manner. The primary controller is a naive implementation of a learning switch – the one developed in the OpenFlow Tutorial. Upon reception of an OFPT_PACKET_IN message, the controller installs a rule, which sends messages directed to the Mac Source of the packet to the in-port of the message. It then floods the packet which triggered the OFPT_PACKET_IN message on the remaining physical ports.

Again the given claims evaluated using the described methods and were found to be true.

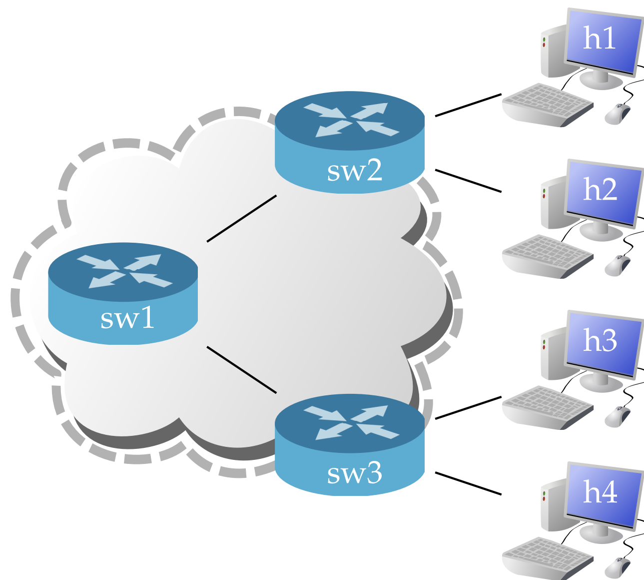

Figure 2.7: Validation Topology 2 – The switches and hosts are connected in a tree like manner. Whereas the primary controller is a naive implementation of a learning switch, the development controller aims to forward all traffic to host h1.

2.7 Scaling Considerations

The implementation strongly supports the functionality of the algorithm. Yet evaluation has only been performed with a small number of switches / controllers and rules. This section evaluates the scaling behavior of the algorithm in the context of time, memory and control traffic.

Time An algorithm that does not scale well in the time dimension will ultimately fail to implement the desired functionality. Hence scaling in time needs to be evaluated.

Control Traffic The control traffic qualifies as limited resource for two reasons: *(i)* processing of a control message either by the controller or the datapath requires resources that are limited (indirect limitation) and *(ii)* **V** is connected to the controllers and datapaths via a socket whose bandwidth is also limited. Sending a sufficiently large number of control messages would therefore either flood the controller/datapath or overload the socket. The internal structure of **V** and the scaling of the control traffic is illustrated in Figure 2.8.

Memory Memory space is a limited resource on *(i)* the controller, *(ii)* **V** and *(iii)* the datapath. As the presence of **V** does not have any influence on the amount of control messages that are being send to the controller, the amount of memory used by the controller does not change[45]. As it is assumed that **V** is running on a modern server architecture, it can also be assumed that the memory used by **V** can be neglected. Studying the memory usage therefore reduces to studying the amount of used flow table entries on the switches.

[45] It might even decrease as the size of the control messages is reduced.

2.7.1 Database Operations

As basis for its internal database, **V** uses the Federated Flow Map [21] of FlowVisor. It contains all the information that **V** handles (in different tables). The database is the only part of **V** that changes at runtime.

This database structure supports various operations such as: *(i)* addition of an entry, *(ii)* search for an exact match, *(iii)* deletion of an entry and *(iv)* search for entries whose flowspace overlap with the query's.

Addition of an Entry Addition of an entry requires updating a constant number of hash tables (indices). In time it therefore scales with $\mathcal{O}(1)$, whereas the size of the database as well its indices grows with the number of entries N. The memory requirements are limited to the storage of the database entry and indices. The memory requirements hence scale with $\mathcal{O}(N)$.

Search for Exact Match Search for an exact match is implemented by evaluating the indices and calculating the intersection of the resulting sets. As discussed in [21] this scales in time with $\mathcal{O}(1)$.

Deletion of an Entry Deletion of an Entry of the tables requires updating the indices. This also scales in time with $\mathcal{O}(1)$ as the indices are implemented as hash tables [22].

Search for Overlapping Entries Querying the database for all flowentries whose flowspace overlap with a given one, involves *(i)* calculating the set-or function of sets returned by the indices and *(ii)* obtaining the resulting entries from the database. As described in [21] this can be done with $\mathcal{O}(k)$ for k being the number of matching entries found.

Nomenclature On the following pages k_i indicates the number of the matching entries for request i. K describes the overall size of the table. The superscript indicates the table which is being used:

- μ: slice table
- ν: reservation table
- ϕ: controller rule table
- χ: sliced rule table
- ψ: switch rule table

Moreover N indicates the number of registered controllers.

For a definition of the Landau symbols used, refer to [23].

2.7.2 North Bound Interface

The Northbound Interface is a stateless unit that translates the controller messages into objects for internal use and vice versa. The unit therefore scales in time with $\Theta(1)$. Its stateless character makes it scale in memory with $\Theta(1)$. There is one internal object generated per control message received. Thus it also scales in control messages with $\Theta(1)$.

2.7.3 Slicer

The Slicer unit takes a controller rule as input and translates it to a set of sliced rules. Depending on whether the port(s) for which the rule is valid, is an internal or an external port, the Slicer uses the reservations or the controller's slice to calculate the sliced rules.

Internal Port The Slicer needs to find the reservations with which the given rule overlaps. As seen above such a database query scales with the number of matching entries k_i^ν, which in turn scale with the table's size K^ν. The size of the reservation table is determined by the number of sliced rules that forward traffic to that specific port, as each of these generates exactly one entry in the reservation table. Thus the Slicer unit scales in time with $\mathcal{O}(k_i^\nu)$, where $k_i \in [0, K^\nu]$ is the number of reservations with whose flowspaces the new rule overlaps.

As the Slicer unit itself does not store any information it scales in memory with $\Theta(1)$.

When processing a controller rule, the Slicer generates a sliced rule for every reservation. In the control traffic dimension it therefore scales with $\mathcal{O}(k_i^\nu)$.

External Port If the port is connected to an external device, the controllers slice is used instead of the reservations. These slices are constant at runtime and the algorithm therefore scales in time with $\Theta(1)$.

Processing a controller rule for an external port generates k_i^μ rules, where k_i^μ is the number of slice entries with which the rule overlaps. In contrast to an internal port, this number stays constant at runtime. Hence control traffic scales with $\mathcal{O}(k_i^\mu)$.

2.7.4 Merger

The Merger has two different tasks, it generates the reservations so that they do not conflict with the reservations of other controllers and merges the rules of the different controllers.

Making Reservations Every sliced rule which is generated by the Slicer triggers generation of a reservation. Checking for a conflict requires checking whether there is a entry in the reservation database which overlaps with the flowspace that needs to be reserved. With the algorithm used by FlowVisor [21] this can be done in $\mathcal{O}(1)$ time[46]. If a conflict occurs another conflict-free volume of equal size in the headerspace $\mathcal{H}$ needs to be found. The prototype implementation assumes that the data source was using a constant VLAN id. It can easily find a volume of equal size by choosing an unused VLAN id. A more general algorithm is outside the scope of this thesis.

For every sliced rule exactly one reservation is generated, therefore the memory requirements scale with $\mathcal{O}(K^\nu)$, K^ν being the number of installed reservations.

[46] The database is queried for rules whose flowspace overlap with the given one. Instead of fetching the resulting entries from the database it is only evaluated whether the result is non-empty.

Merging Merging the sliced rules of different controller requires calculation of the crossproduct of the rules' flowspaces. In OpenFlow 1.0 only one flow table of the datapath can be accessed[47]. Therefore calculation of the crossproduct has to be done in software and cannot be pushed to the switch. Thus the N controller-specific flow tables need to be mapped to a single one-dimensional flow table. The algorithm proposed in my thesis, fetches the switch rules from the database whose flowspace overlap with the flowspace of the controller rule. This can be done in $\mathcal{O}(k_i^\psi)$. Here k_i^ψ is the number of controller rules which match the query. **V** then iterates through these rules and installs at most one new controller rule per entry, before it finally installs the controller rule corresponding to the sliced rule. The overall merge therefore scales with $\mathcal{O}(k_i^\psi)$ in the dimensions of: time, flow tables entry number and control traffic.

Be aware that k_i^ψ itself scales with the number of switch rules that are currently installed on the switch. As they are the result of the crossproduct of controller rules, the number is limited by

$$\prod_{l=1}^{N} K_l^\chi, \tag{2.17}$$

where K_l^χ is the number of controller rules that were installed by controller i. Installation of a rule that is valid for the whole flowspace leads to this worst case scenario. Hence the Merger unit's worst case execution time grows with

$$WCET \in \mathcal{O}\left(\prod_{l=1}^{N} K_l^\chi\right). \tag{2.18}$$

Hence the algorithm does not scale polynomially. However, as described in Section 2.4.4 OpenFlow 1.x [6, 7, 5] might allow implementation for algorithms that scale better.

[47] Depending on the implementation this might be a logical representation of multiple internal flow tables.

2.7.5 Classifier

The Classifier takes an OFPT_PACKET_IN or an OFPT_STATS_REPLY as input value and uses the controller slices or the reservation rules to determine which controller is to receive a modified copy of the message.

OFPT_PACKET_IN

The Classifier needs to check whether a specific controller is allowed to receive the OFPT_PACKET_IN message. Checking whether a table contains an entry whose flowspace matches the packet and was installed by a specific controller, can be done in $\Theta(1)$. Because **V** needs to validate this for all N controllers, it scales in time with $\Theta(N)$.

OFPT_STATS_REPLY

A OFPT_STATS_REPLY is generated by the switch as reply to a corresponding request. The reply needs to be forwarded to all controllers that sent an OFPT_STATS_REQUEST message to the datapath and have not received an OFPT_STATS_REPLY yet. Keeping a simple list of controllers for every datapath, allows a lookup time that scales with $\Theta(1)$.

Yet the controllers need to obtain individualized OFPT_STATS_REPLY messages. This requires re-translation of the switch rules into controller rules. Let w be the number of entries in the OFPT_STATS_REPLY message. For every such entry, **V** searches the corresponding switch rule in its Switch Rule Table. Since an exact match is required the procedure to obtain the switch rule from the database scales with $\mathcal{O}(1)$.

Every switch rule ψ_i is the result of h_i sliced rules. In turn every sliced rule is associated with exactly one controller rule. Hence finding the controller rules that are associated with switch rule psi_i scales with $\mathcal{O}(h_i)$.

It holds $h_i < N$ for N being the number of controllers, because every switch rule is associated with at most one sliced rule of every controller. Herefrom follows the runtime-stable scaling attribute of $\mathcal{O}(N)$.

Once the associated controller rule is found, the entry in the controller-specific OFPT_STATS_REPLY is generated in constant time. Thus the overall procedure scales in time with $\mathcal{O}(wN)$.

2.7.6 South Bound Interface

The South Bound Interface solely translates internal objects into control messages and vice versa. It is fully stateless and therefore scales with $\Theta(1)$ in time, memory and control traffic.

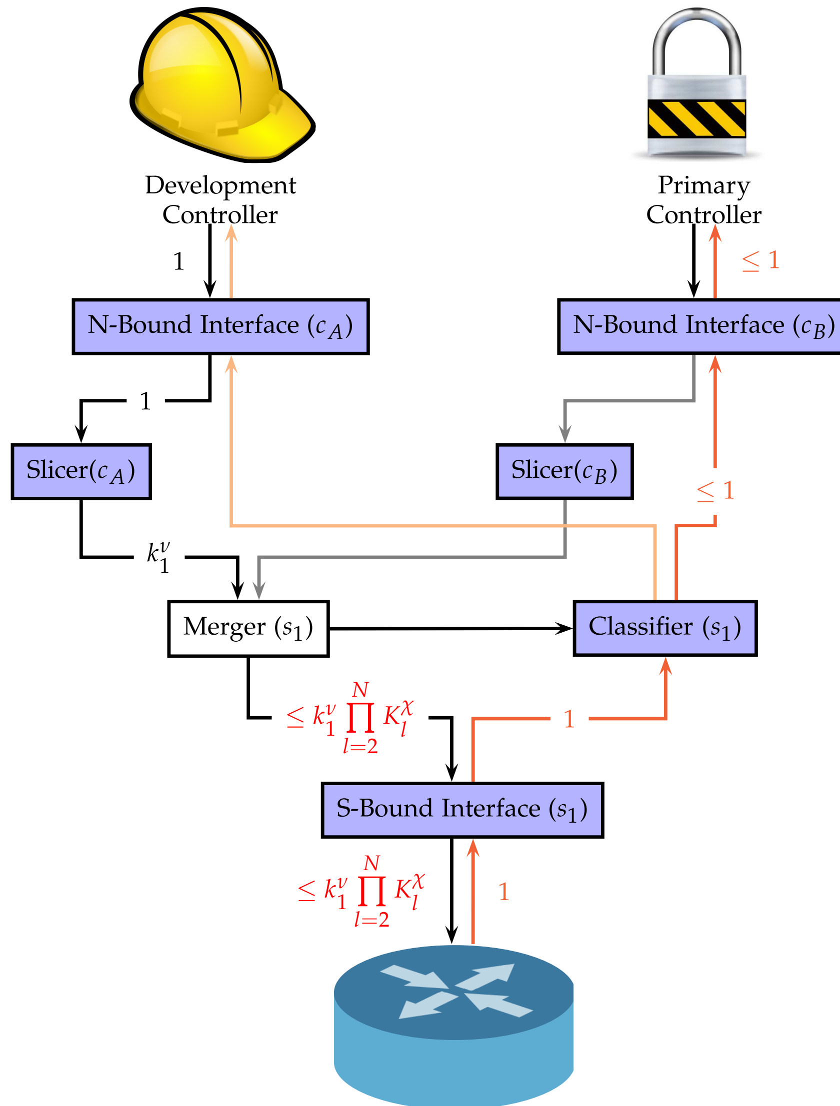

Figure 2.8: Scaling characteristics of **V** in the control traffic dimension. For an OFPT_FLOW_MOD, the Slicer generates k_1^v messages that are sent to the Merger, which in turns generates at most $\prod_{l=2}^{N} K_l^\chi$ messages.

2.8 Conclusions

In the thesis at hand I describe how multiple controllers can share the same flowspace without interfering with each other. From the controller's perspective full isolation is guaranteed. The approach supportes isolated virutalized control planes, which correspond to controller-specific virtualized networks.

This allows researchers to develop controllers, that observe the full network traffic without influencing packet delivery. Therefore I allow development of controllers that do not need to reason about any other controller operating on the same flowspace. Today this is not possible, because topology-aware controllers observe the influence that other controllers have on packets of the overlapping flowspace. The given implementation proves in-principle-feasibility of the approach. However the scalability discussion shows that with today's version 1.0 of the OpenFlow protocol, the approach is not suitable for large or even medium sized networks. Newer versions like 1.1, 1.2, 1.3 and 1.4 promise features that potentially allow for an implementation with better scaling characteristics. Which of these versions (or any future version) will find wide acceptance under switch implementers, is not clear today.

In addition I came to believe that the hypervisor that ensures resource sharing of the network should run a different protocol on north and south bound interfaces. The OpenFlow protocol is arguably not designed to support flowspace sharing among multiple controllers. Yet the north bound protocol of a hypervisor could benefit from features that are not available in the OpenFlow protoco. Frenetic's [15] approach seems very promising. Therein coordination between the controllers is ensured at compile time instead of being artificially enforced afterwards. Implementing the idea of different controller classes on the basis of Frenetic might also solve the scaling problem, because the controllers can be coordinated before problems occur.

If I were to continue this work in an academic setting, I would however evaluate yet another approach. It seems tempting to develop a network programming language, that displays the standard OpenFlow Interface to higher level programming languages, while it uses an extended protocol to communicate with the hypervisor. The network applications would have to be compiled specifically for the hypervisor, but they could reside on a different physical host. This means combining the idea of Frenetic with the architecture of FlowVisor. The controller-specific part of Frenetic could be located on a physical host different from the centralized hypervisor. This would allow a higher controller flexibility. A measurement controller could easily be added to the the network and (depending on the runtime) even reside at a different location.

Chapter 3

Debugging OpenFlow Networks

3.1 Motivation

Foster et al. define a language in their paper “Frenetic: A Network Programming Language” [15] that allows programmers to define multiple modules within their network controller that register for specific event streams. A module that implements an HTTP load balancer for example would want to obtain an event stream in which an event corresponds to the first packet of a new conversation with a specific destination IP address and TCP Port 80. The modules are compiled in a way they do not interfere with each other and are not affected by the asynchronous nature of the OpenFlow control channel. This dramatically reduces the complexity that the developer needs to deal with when writing network control applications.

If resource sharing is realized via a separate application that uses the OpenFlow Protocol on both North and South Bound Interface (such as **V** and FlowVisor), there is no specific way for the controllers to register for a specific event stream. To the hypervisor a controller is a Mealy machine whose internal state, and transition functions are unknown. As the controller might use external data (e.g. the current electricity price in order to do economical routing) only a subset of the input values is known to the hypervisor. To make things even more complex the controller might treat requests in individual threads and therefore the response stream might be ordered differently from the input stream.

Trying to guess the internal structure of the controller (e.g. with Hidden Markov Chains) could therefore only give a mere idea of the controllers inner structure. This lack of information makes it impossible for the hypervisor to ensure that network applications which share a flowspace do not interfere. If the system administrator sets a faulty configuration, or controller / hypervisor / switch implementation is buggy, this interference between controllers can generate undesired effects.

Consider the following academic case of a loop generation. A loop is always "closed" by installation of a flow rule which forwards a packet to a datapath that is has already visited while preserving the relevant packet headers. One might try to find such loops using an analytic method at installation time. The main problem here is the complexity of the network and its dynamic nature, which is introduced by the black-box-nature of the controllers. To check whether installation of a rule would introduce a loop one would have to calculate the path of every possible packet that could be generated by that rule until it leaves the network. In the worst case, this would mean that $> 2^{223}$,[1] paths would have to be evaluated[2]. Such an analysis would consume a vast amount of resources and slowing down the control path of the network is undesirable.

A Debugger that uses the abundance of available data about the state of a functioning network in a opportunistic fashion to validate in near real-time whether the intended network policies are being enforced, would be of great use for developers of all stages. The design and implementation of such a debugger, that helps OpenFlow network administrators to pinpoint problems like routing anomalies is the main goal of this part of the thesis.

1 The OpenFlow allows to define rules that match against $223 + p$ bits. In this context p is the implementation-specific number of bits used to represent the in-port of a packet. Hence the maximum number of rules that could be installed on every switch is 2^{223+p}. If the in-port is not used in the matching process, the maximum number of paths that would need to be evaluated is 2^{223}. In this case, one path for every distinguishable packet would be installed in the network. Usage of the in-port increases complexity, as the ingress switch/port through which the packet entered the network has an effect of the packet's path. If a network has q ingress ports, the number of paths that would need to be evaluated is $q \cdot 2^{223}$. In reality the switches' flow tables are limited in size, hence the 2^{223+p} table entries will hardly be reached and the number of paths to evaluate reduces accordingly.

2 The average case looks way better though.

3.2 Related Work

Besides the work reference in Section 2.2, the recently published paper "VeriFlow: verifying network-wide invariants in real time" [24] is worth being referred to. It takes a very similar approach as proposed by me. The main difference is that it assumes that the controller to installs rules whose validity flowspace is limited. If that assumption is made, the impact which every rule installation has on behavior of the network is limited. Hence analytic methods can be used. Yet installation of a rule, which is valid for the whole header space would make analytic validation very time consuming.

Yet another approach is take by Canini et al. [25]: "NICE applies model checking to explore the state space of the entire system – the controller, the switches, and the hosts." [25]. Unlike the approach-at-hand, NICE is proactive and does not analyze the network state in real-time.

3.3 Data Sources

A debugging tool benefits from each of the information sources that it can leverage. The available data sources that can be used in the context of an OpenFlow network are discussed in this section.

3.3.1 OpenFlow

Upon reception of a packet that does not match against any entry in the switch's flow table, this packet is forwarded to the controller[3]. Ideally the controller installs a new flow entry and/or sends the packet back to the switch together with an associated action.

By monitoring the control channel OFPT_PACKET_IN messages as well as the controller's response can be captured. However associating the OFPT_PACKET_IN events with the installed rules and generated actions is not trivial. There is no specific transaction id defined for events generated by the switches[4] and the rules that a controller installs might not

[3] The application developer can set a differing configuration.

[4] A buffer id is associated with every OFPT_PACKET_IN message, but this id is used to tell the switch to apply a rule to a buffered packet. It is valid for the controller

always match against the the packet that triggered its installation. For example a controller which implements a learning switch and receives a packet with *source* MAC address x_i on port q_i will install a rule that forwards traffic directed to *destination* MAC address x_i through port q_i. In a second step it will request the switch to flood the packet that triggered the controller on the remaining ports.

Besides passively listening to the traffic of the control channel, active injection of messages is possible: e.g. actively polling the packet and byte counter values of the flow table entries.

3.3.2 FlowVisor

FlowVisor injects LLDP packets [17] into the data plane of OpenFlow switches that are connected to FlowVisor and floods these packets on all ports. An OpenFlow switch that receives such a packet forwards it to FlowVisor, which can then tell from the payload how the switches are connected. This is an additional feature, that does needs to be activated. The API of FlowVisor can be used to obtain the topology.

3.3.3 sFlow

If sFlow [9] is enabled on a switch, it samples packets on its network interfaces and forwards the sampled packets to a collector. The collector hence obtains a "good representation"[5] of the current traffic in the network. This can be used to build up statistics and / or to trigger analyses that require specified packet samples. Furthermore sFlow reports interface counters (byte counters) of the interfaces in periodic time intervals. These can be good indicators for the traffic volume that is transversing a specific link[6].

---

to install further rules as a response to the OFPT_PACKET_IN messages.

[5] The quality of the representation depends on the type of traffic observed and the applied sampling strategy.

[6] The quality of the indicators depends on the temporal granularity of the reports.

3.3.4 SNMP

SNMP [26] allows polling of a vast amount of data from the switch. Yet the information most relevant for debugging purposes would be the interface counters which can also be obtained via sFlow. In our experiments SNMP seems to be slow and the ability of polling the interface counters over having them pushed periodically by sFlow does not provide much of an advantage.

3.4 Concept

Assuming that the controllers satisfy fundamental requirements – such as generating control messages at a bounded rate – it is possible to compare the controller's behavior in a multi-controller network with its behavior in a single-controller network. This comparison can be made at runtime with a single instance of the controller application.

To do so I wrote a python script, which simulates a clone of the real network with its switches and links for every individual controller. A small piece of software called **Sacramento**[7] is eavesdropping the communication between the controllers and FlowVisor. It keeps record of the forwarding rules currently active on the switches. This data fully describes how the physical network is supposed[8] to handle incoming traffic. Every such rule is associated with the controller that installed it (this information is being lost once the rule is installed on the switch). Only polling the forwarding table without listening to the conversation between the controllers and switches would therefore not be sufficient.

Furthermore the border switches are configured to use sFlow sampling on their incoming ports so that the collector receives a good picture of the traffic that enters the network. For each of these packets the collector

[7] The Sacramento – San Joaquin River Delta is one of the world's few estuaries ("inverse deltas"). It has a number of springs and one connection towards the open sea. This neatly represents the origin-estimation feature of **Sacramento**. A packet that is being sampled at any point in the network can stem from a number of locations, while its destination is well-defined.

[8] Be aware that implementation-behavior of the network is to be validated. A difference between the simulation and the physical network necessarily originates from a bug in either implementation; or a faulty theory.

simulates what the physical network is expected to do and what each of the N controllers expects to happen in a single-controller network.

While the data obtained from eavesdropping the communication between the controllers and switches is sufficient to simulate what the physical network does[9], more information needs to be gathered to evaluate how the controllers would behave in a single-controller network. If controllers are installing overlapping flow rules in the forwarding tables of the same switch, rules that have higher priority or were installed earlier will "consume" the packet. Rules with lower priority are not being used by packets that matched earlier rules; these rules are being starved.

Therefore OFPT_PACKET_IN messages that would have been generated by the datapaths, was there not interference between the controllers, are being "lost". These OFPT_PACKET_IN messages need to be generated and injected into the control channel to evaluate how the controller would handle such a message. That needs to be done so that neither the controller's nor the network's state is modified[10].

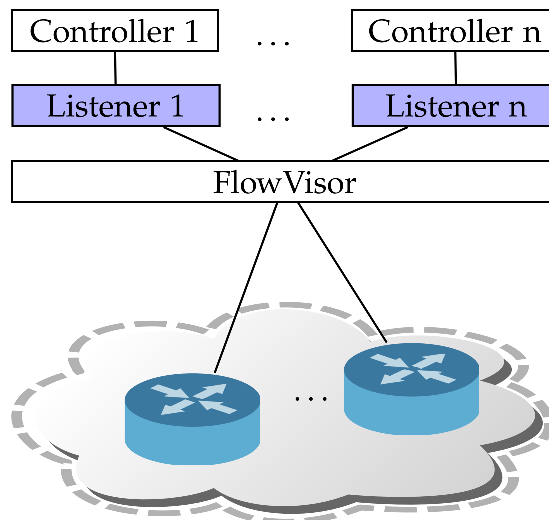

Figure 3.1: Eavesdropping the conversation between the controllers and the switches is done by intercepting the control channel between the controller and FlowVisor.

[9] Assuming that the network's and the simulation's implementation are bug-free.

[10] The controller's state actually can be changed, as far as this change in its state does not have an influence on the network's state.

3.4.1 Controller-Specific Network Views

As illustrated in Figure 3.1 a small piece of software is used that eavesdrops the conversation between the controllers and FlowVisor. It keeps track of the flow rules that are currently active. If the flow modification message, sent by the controller, indicates that the controller does not wish to be informed upon removal of the flow from the flow table of the switch, this parameter is modified and the corresponding flow removal message is filtered out[11] by **V**. In this way information about the rules and their validity period can be obtained. The obtained picture is not perfect, because the time at which the control message is observed is not the time at which the flow rule becomes active/inactive. This point in time can only be estimated.

The rules that are installed on the network's switches represent the physical network view. It is the result of multiple controllers interacting with the network and potentially interfering with each other. The individual controller might however have created another set of rules, had it been operating in a single-controller-network. Determining which set of rules the controllers would install in such a case is not fully straight forward. This information however is needed to pinpoint (unwanted) interference between multiple controllers.

Single-Controller-Volumes In volumes of the headerspace [3] $\mathcal{H}$ in which only *one controller* has writing privileges, the physical network is obviously representing the network view of the controller that is responsible for that flowspace. Building the controller-specific network views for these areas is simple: all installed flow table entries for this region are associated with the according controller and the other $N-1$ controllers perceive this part of the network as rule- and traffic-less.

Multi-Controller-Volumes As depicted by the crosshatched area in Figure 3.2 association with a specific controller the case of overlapping controller slices is more complicated. In the physical network the path of a specific flow might vary drastically from the paths that the individual controllers would have chosen in isolated networks. Even though FlowVisor forwards an OFPT_PACKET_IN to only one of the controllers associated with a point in the headerspace any of the associated controllers

[11] This part has not been implemented in the prototype.

can install rules for it. Whichever controller installs the highest-priority rule, starves the rules that were installed by other controllers. The paths whose establishment would have been triggered by rules that now are being starved, are not being explored in the physical network. These paths can be evaluated by injecting OFPT_PACKET_IN messages into the control channel between FlowVisor and the controller and observing the controller's response.

(i) send an OFPT_TABLE_MOD message and install a new rule,

(ii) send an OFPT_PACKET_OUT message and forward the packet to the next hop or

(iii) ignore the request.

All actions can be interpreted as the controller's intended handling of that specific packet. This allows to draw the path any given packet would take through the specific single-controller-networks.

The soft timeout associated with the rules is interpreted as hard timeout and the controller is queried for the flowspace again after the associated rule timed out. This is done because the simulated path is not being chosen in the real network and no incorrect view of the network is presented to the controller by letting these rules time out. Were the rules actually installed on the network, they might actually be applied, as traffic from other sources, that is not present in the simulation, might match against them. This effect is not expected by the controller and therefore removing them after the soft timeout is presenting a consistent way of the network to the controller[12].

To obtain a reasonable picture of the traffic in the network, the border switches are configured to take sFlow samples on their egress interfaces. These samples trigger the simulation which evaluates their paths through the network. The rules that were generated by the controller in response to the simulation are stored in the database, but not installed on the switches.

12 Letting rules time out, has not been implemented in the prototype.

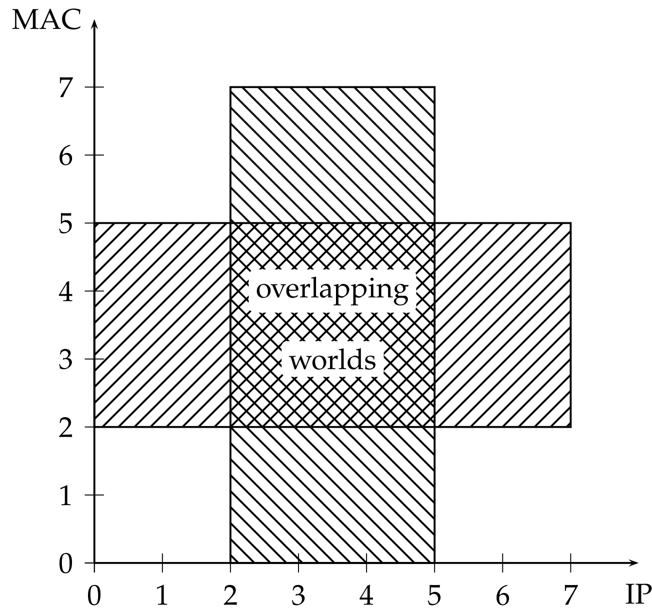

Figure 3.2: Suppose for a specific switch, Controller c_1 has write privileges for the MAC address range 2-5 and Controller c_2 has write privileges for the IP address range 2-5. Then for the shaded areas the picture that they have of the network is identical with what is happening in the real network. Where they overlap, the flow rules installed in the physical network are a mixture of these two worlds. In these flow volumes active querying of the controllers is necessary to investigate the controller's expectation of the network's behavior.

3.4.2 Network-Neutral Controller Probing

Replies to the fake OFPT_PACKET_IN messages have to be dropped as their delivery to the switch would change the state of the network. Yet associating the response with the OFPT_PACKET_IN message that triggered the reply is not trivial, because there are no transaction ids for events originating from the switches. Even though there is a buffer id associated with every packet that is forwarded from the switch to the controller, the programmer might be sloppy and omit the buffer id or fully legitimately install rules, against which the packet does not match.

The following three approaches[13] can be used to ensure that the state of the network is not changed by **Sacramento**.

[13] This list does not claim coverage of all possible approaches.

Stalling Execution

One way of associating OFPT_PACKET_IN replies with the message that triggered them, is stalling forwarding of control messages for the time that the fake message is being processed by the controller. When a fake OFPT_PACKET_IN message is generated, original OFPT_PACKET_IN messages are queued. The fake message is injected into the control channel only after no OFPT_PACKET_IN replies have been observed from the controller for a predefined time. After its injection the controller's response is captured, and associated with the fake message. After another timeout normal processing of legal OFPT_PACKET_IN messages is continued.

Faking Datapaths

If the OpenFlow channel is the only source through with a controller obtains information about the network state, the whole network topology can be virtually duplicated. Every datapath that registers at **Sacramento**, triggers registration of two datapaths at the controller. Of these datapaths one will represent the true datapath with its true datapath id, whereas the second is registered with a datapath id, that is not being used by any datapath in the network[14].

As the state of the duplicated network needs to be the same as the original one, OFPT_PACKET_IN events are duplicated as well. The advantage of this approach is, that there is no need to associate the replies with the OFPT_PACKET_IN events, as control messages destined to fake datapaths can silently be dropped.

Framing OFPT_PACKET_IN messages

For controllers whose output stream represents the ordering of their input stream, i.e. controllers that do not use threads, markers can be used to define the beginning and end of the controllers response to a specific request. The fake OFPT_PACKET_IN-to-be-injected is being framed by

[14] The datapath id is 64 bit long. It is seems valid to assume that in today's network there will be less then 2^{63} network switches associated with one installation of FlowVisor.

two OFPT_PACKET_IN messages with a fake datapath id. As the ordering is preserved, the first occurrence of responses with the fake datapath id marks the beginning to the reply associated with the fake message, while the second occurrence marks the end of the reply.

3.4.3 Comparing the Controller-Specific Views

As illustrated in Figure 3.4 different controllers might want to implement different paths for the same physical packet. The resulting physical path can vary from the paths that the controllers in a single-controller network would have implemented by the: *(i)* datapaths visited / links used in the OpenFlow network, *(ii)* egress datapath / egress port through which the packet leaves the network, *(iii)* delivery (a controller can decide to drop the whole flow) and *(iv)* the header field values when exiting the network. If the packet header fields differ while the packet is in the network but are equal when the packet leaves the network, no external impact is made and hence the case can be ignored.

There seems to be no general way of defining a measure for the level of violation. Even if the physical network is not behaving the way a controller would expect, it is not necessarily a problem, because the network administrator might want different controllers to interact in this specific way.

Therefore a simple policy syntax is defined that allows network administrators to define what variations between the controllers' expectations and the actual network's behavior are tolerable. If the path that is calculated for a specific controller differs from the path in the physical network, the policy chain is evaluated and the associated action is executed.

Integers	n	
Inequality	neq	
ControllerId	$contr$	
DatapathId	$dpid$	
PolicyChain	$chain$	$::= \langle contr, \{pol_1, \ldots, pol_k\}\rangle, k \geq 0$
Policy	pol	$::= \langle pat, cond\rangle$
Patterns	pat	$::= \{h_1 : n_1, \ldots h_k : n_k\}, k \geq 0$
Headers	h	$::= in_port \mid VLAN \mid dl_src \mid dl_dst \mid dl_type \mid$ $nw_src \mid nw_dst \mid nw_proto \mid tp_src \mid tp_dst$
Conditions	$cond$	$::= \langle (attr, loc, val, act)\rangle \mid cond\ AND\ cond$
Location In Path	loc	$::= -pathlength \cdots -1, 1, pathlength \mid any$
Attribute	$attr$	$::= h \mid dpid$
Value	val	$::= n \mid neq$
Action	act	$::= accept \mid info \mid warning \mid error$

Figure 3.3: Policy syntax.

3.4.4 Small Flows

The analysis is triggered by sFlow which samples packets with a predefined sampling rate R. Therewith the minimum size m a flow has to have so that it is sampled with a probability larger than $\hat{p}$ can be calculated.

Suppose that in a time interval T, m out of the total N packets belong to the flow. Sampling with rate R essentially means drawing $n = R \cdot N$ samples without replacement. The probability of k successes is described by the hypergeometric distribution:

$$P(X = k) = \frac{\binom{m}{k}\binom{N-m}{n-k}}{\binom{N}{n}} \tag{3.1}$$

therefore the probability that least one packet of the flow is sampled becomes:

$$\begin{aligned} p &= \sum_{k=1}^{min(m,n)} P(X = k) \\ &= \sum_{k=1}^{min(m,n)} \frac{\binom{m}{k}\binom{N-m}{n-k}}{\binom{N}{n}} \end{aligned} \tag{3.2}$$

The problem with this formula is that the probability of detecting the flow of size m depends on the traffic load. If the flow-to-be-detected would be the only flow in the network, every sample taken would be from this flow.

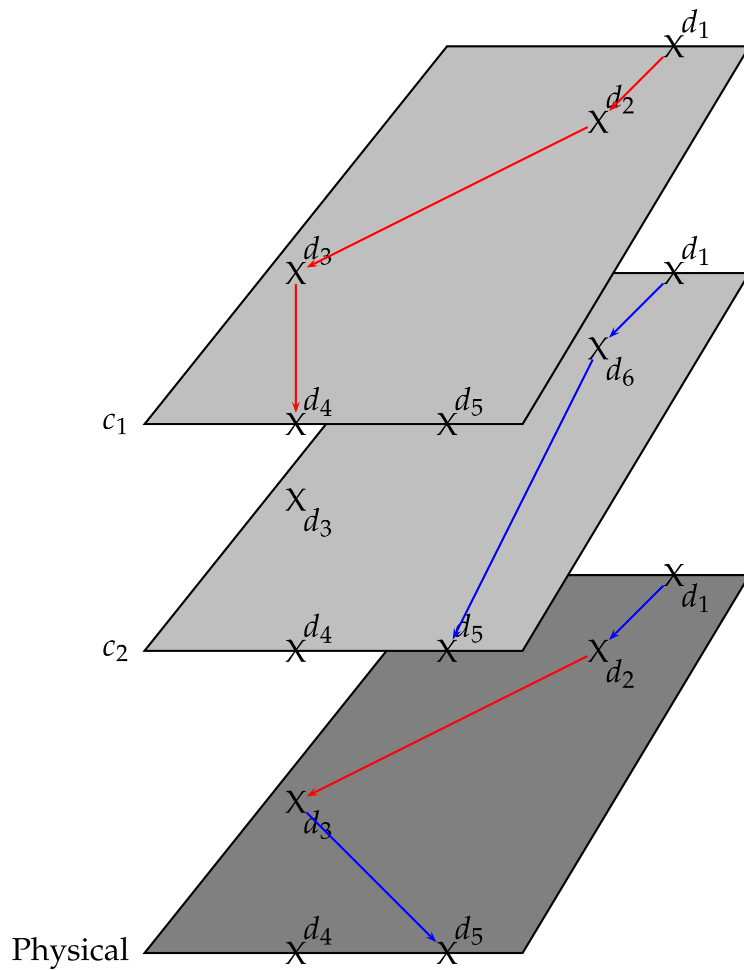

Figure 3.4: Suppose there are two controllers c_1 and c_2 connected to FlowVisor and the packet that arrives at d_1 falls within the slice of both controllers. If the controllers were acting in a isolated network, the packet would take the path that is indicated. The path that the packet actually takes might be one that differs from what the controllers expect.

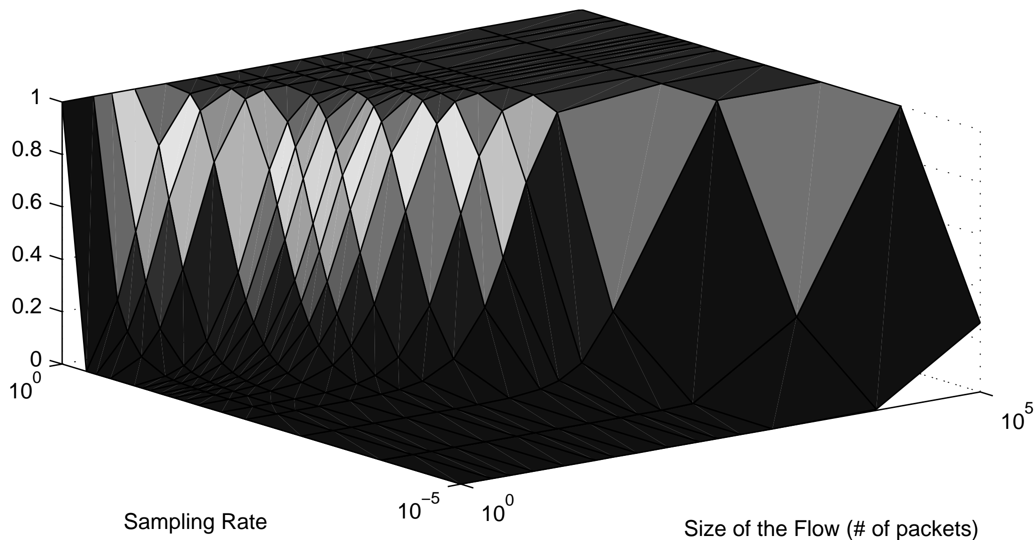

Figure 3.5: Probability of sampling from a flow of size m (in a time interval T) while sampling with rate R therefore taking $R \times T$ samples. Simulated with 1.000.000 packets.

If it is a marginal flow[15] it is harder to detect. Figure 3.5 illustrates the probability of detecting a flow with size m (in a given time interval) and sampling rate R while assuming that the network is processing 1.000.000 packets in the same time. With a low sampling rate it is therefore unlikely to find small flows. This is especially a problem for TCP connections that are not established because there was a problem in the network. They will only generate very few packets but would be interesting to sample from. Problems that could cause this are: *(i)* a controller installs a drop rule *(ii)* no controller responds to the OFPT_PACKET_IN event *(iii)* loops are introduces into the network and *(iv)* the packet is delivered to an incorrect destination.

The main limitation of the forgoing approach is, that the flows have to have a specific size in order to obtain a sample of each flow with a sufficiently high probability. Small flows are not likely to be detected and a different approach has to be used to obtain samples from these small flows. By observing the change in counter values of the flow statistics[16], these small flows can be found.

15 A marginal flow in this context is a flow that, relative to the total number of packets per time unit, generates sufficiently few packets.

16 Here the counter values that are provided by the switch via the OpenFlow protocol are used.

3.5 Implementation

Sacramento was implemented as Python script and tested with mininet [20] networks. Even though not described in the forgoing sections, the implementation allows the user to inject any packet at any point in the simulated network, thereby allowing to evaluate what the network and the individual controllers would do with the packet, without injecting it into the physical network. **Sacramento** is furthermore able to estimate the source of a given packet that was sampled on an internal datapath, by leveraging packet statistics.

A sample output is given in Figure 3.6.

```
IN DPID: 20:1, FLOWSPACE: {'dl_type': '0x800L', 'nw_dst': 167772176L, 'dl_src': '0x1L', 'nw_proto': 1L, 'nw_tos': 0L,
 # 'tp_dst': 0L, 'tp_src': 8L, 'dl_dst': '0x10L', 'nw_src': 167772161L}
 #  - CONTROLLER - 491
 #  - CONTROLLER - 492
 |
 +=[X] OUT DPID: 20:3, FLOWSPACE: {'dl_type': '0x800L', 'nw_dst': 167772176L, 'dl_src': '0x1L', 'nw_proto': 1L,
    # 'nw_tos': 0L, 'tp_dst': 0L, 'tp_src': 8L, 'dl_dst': '0x10L', 'nw_src': 167772161L}
    #  - CONTROLLER - 492
    #  - CONTROLLER - 491
    |
    +=[X] IN DPID: 19:1, FLOWSPACE: {'dl_type': '0x800L', 'nw_dst': 167772176L, 'dl_src': '0x1L', 'nw_proto': 1L,
       #'nw_tos': 0L, 'tp_dst': 0L, 'tp_src': 8L, 'dl_dst': '0x10L', 'nw_src': 167772161L}
       #  - CONTROLLER - 492
       #  - CONTROLLER - 491
       |
       +=[X] OUT DPID: 19:3, FLOWSPACE: {'dl_type': '0x800L', 'nw_dst': 167772176L, 'dl_src': '0x1L', 'nw_proto': 1L,
          # 'nw_tos': 0L, 'tp_dst': 0L, 'tp_src': 8L, 'dl_dst': '0x10L', 'nw_src': 167772161L}
          #  - CONTROLLER - 492
          #  - CONTROLLER - 491
          |
          +=[X] IN DPID: 18:1, FLOWSPACE: {'dl_type': '0x800L', 'nw_dst': 167772176L, 'dl_src': '0x1L', 'nw_proto': 1L,
             # 'nw_tos': 0L, 'tp_dst': 0L, 'tp_src': 8L, 'dl_dst': '0x10L', 'nw_src': 167772161L}
             #  - CONTROLLER - 492
             #  - CONTROLLER - 491
             |
             +=[X] OUT DPID: 18:3, FLOWSPACE: {'dl_type': '0x800L', 'nw_dst': 167772176L, 'dl_src': '0x1L',
                # 'nw_proto': 1L, 'nw_tos': 0L, 'tp_dst': 0L, 'tp_src': 8L, 'dl_dst': '0x10L', 'nw_src': 167772161L}
                #  - CONTROLLER - 492
                #  - CONTROLLER - 491
                |
                +=[X] LEAVES OPENFLOW NETWORK - KNOWN DESTINATION (mac:0:0:0:0:0:16, ip=10.0.0.16)
                   #  - CONTROLLER - 492
                   #  - CONTROLLER - 491
```

Figure 3.6: Sample output of **Sacramento**. The packet ⟨*Ethernet-Type: 0x800, MAC Source: 00::01, MAC destination 00::16, IP Source 10.0.0.0.1 (167772161L), IP Destination 10.0.0.16 (167772176L), Network Protocol: ICMP (id 1), TOS-bits: 0, ICMP-Type: Echo Request (8)*⟩ enters the network through port 1 of datapath 20 and leaves it through port 3 of datapath 18. The destination host (MAC address 00::16, IP address 10.0.0.16) is known to **Sacramento** and connected to the physical destination port. Both controllers (ids 491 and 492) specify the same path.

Chapter 4

Conclusions

If the slices of different controllers overlap, conflicts can occur. Finding these conflicts is possible by sampling from the network traffic while taking special care of small flows and simulating the paths that the individual controllers would implement for the given packet. The network administrator is being informed about variations of the resulting flows. As some of these differences might be tolerated, a syntax is defined that allows the administrator to specify such cases.

Sacramento can also be used to inject a given packet into the simulation and evaluate which path a specific controller would implement for it. The concept has been implemented as Python script and tested. Out of personal experience the ability to analyze the path of a specific packet proves very useful in developing network controllers. The ability to detect conflicts that occur between different controllers however looses it's attractiveness if **V** or any other hypervisor that ensures that such conflicts cannot occur, is used.

Bibliography

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

[2] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Flowvisor: A network virtualization layer," tech. rep., Deutsche Telekom Inc. R&D Lab, Stanford University, Nicira Networks, 2009.

[3] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proceedings of the 9th ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2012.

[4] O. N. Foundation, "Openflow specifications 1.0," tech. rep., Open Networking Foundation, December 2009.

[5] O. N. Foundation, "Openflow specifications 1.3," tech. rep., Open Networking Foundation, June 2012.

[6] O. N. Foundation, "Openflow specifications 1.1," tech. rep., Open Networking Foundation, February 2011.

[7] O. N. Foundation, "Openflow specifications 1.2," tech. rep., Open Networking Foundation, December 2011.

[8] R. Braden, "Requirements for internet hosts - communication layers." RFC 1122 (Standard), pp. 100–101, Oct. 1989. Updated by RFCs 1349, 4379, 5884, 6093, 6298, 6633.

[9] M. Wang, B. Li, and Z. Li, "sflow: towards resource-efficient and agile service federation in service overlay networks," in *Distributed*

Computing Systems, 2004. Proceedings. 24th International Conference on, pp. 628 – 635, 2004.

[10] M. Rohmad, "Enhanced netflow version 9 (e-netflow v9) for network mediation: Structure, experiment and analysis," *Information Technology, 2008, ITSim 2008, International Symposium on*, vol. 3, pp. 1–6, 2008.

[11] J. Mattes, "Traffic measurement on openflow-enabled switches," semester thesis, ETH Zurich, Swiss Federal Institute of Technology, Zurich, Switzerland, Febr. 2012.

[12] A. Wundsam, A. Mehmood, A. Feldmann, and O. Maennel, "Network troubleshooting with mirror vnets," in *Proceedings of IEEE Globecom 2010 Workshop of Network of the Future (FutureNet-III)*, (New York, NY, USA), p. 283–287, IEEE, December 2010.

[13] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '12, (New York, NY, USA), pp. 323–334, ACM, 2012.

[14] S. Gutz, A. Story, C. Schlesinger, and N. Foster, "Splendid isolation: a slice abstraction for software-defined networks," in *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, (New York, NY, USA), pp. 79–84, ACM, 2012.

[15] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: a network programming language," *SIGPLAN Not.*, vol. 46, pp. 279–291, Sept. 2011.

[16] L. Jose, M. Yu, and J. Rexford, "Online measurement of large traffic aggregates on commodity switches," in *Proceedings of the 11th USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and services, Hot-ICE'11*, (Berkeley, CA, USA), p. 13–13, USENIX Association, 2011.

[17] The Institute of Electrical and Electronics Engineers, Inc., 3 Park Avenue, New York, NY 10016-5997, USA, *IEEE Standard for Local and metropolitan area networks - Station and Media Access Control Connectivity Discovery*, June 2005.

[18] M. Mackall *et al.*, "The hg-git mercurial plugin." http://hg-git.github.com/, 2005-2011. Seen 5. Sept. 2012.

[19] A. Al-Shabibi *et al.*, "Flowvisor documentation." http://www.openflow.org/wk/index.php/FlowVisor, 2012. Seen 5. Sept. 2012.

[20] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, (New York, NY, USA), pp. 19:1–19:6, ACM, 2010.

[21] A. Al-Shabibi *et al.*, "Flowvisor documentation, section replacement of configuration backend." https://openflow.stanford.edu/display/DOCS/Replacement+of+configuration+backend, 2012. Seen 5. Sept. 2012.

[22] M. T. Goodrich and R. Tamassia, *Data Structures and Algorithms in Java, chap. 9 ("Maps and Dictionaries")*. John Wiley & Sons, Inc., January 2006. ISBN 0-471-73884-0.

[23] D. E. Knuth, "Big omicron and big omega and big theta," *SIGACT News*, vol. 8, pp. 18–24, Apr. 1976.

[24] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey, "Veriflow: verifying network-wide invariants in real time," in *Proceedings of the first workshop on Hot topics in software defined networks*, HotSDN '12, (New York, NY, USA), pp. 49–54, ACM, 2012.

[25] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A nice way to test openflow applications," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI'12, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2012.

[26] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "Simple Network Management Protocol (SNMP)." RFC 1098, Apr. 1989. Obsoleted by RFC 1157.

Chapter 5

Appendix

5.1 Task Description

OpenFlow enabled switches offer enhanced flexibility for networks over alternative contemporary and legacy control environments. This increased flexibility comes at a cost, however - new control structures challenge network designers and administrators to adapt to new unknowns and trust very immature control software. They would therefore greatly benefit from an independent system for validating that the implemented network is functioning as designed.

Legacy approaches to network policy validation have focused on systems with static flow tables and fixed protocol implementations and are much less useful in OpenFlow environments. The surgical control that an OpenFlow controller can exert over the forwarding of individual packet flows create an environment where previous approaches to policy validation are insufficient.

The goal of this Master Thesis is to use the abundance of available data about the state of a functioning network to validate in near real time whether the intended network policies are being enforced at the proper locations and times. This would allow a user to verify the implementation of controller applications, the controllers and the datapaths. Therefore it would deliver an incredible value to both network operators and software/firmware implementers.

A successful implementation could also integrate with FlowVisor to police intended network policy on a per-slice basis, independent of the upstream controllers used.

5.2 Declaration of Originality

The Declaration of Originality may be found on the following page.