Delftvisor: A network hypervisor for Openflow 1.3

Network virtualization for Openflow 1.3

Harmjan Treep

Master of Science Thesis

Delft University of Technology

Network Architectures and Services

Faculty of Electrical Engineering, Mathematics and Computer Science
Network Architectures and Services Group

Delftvisor: A network hypervisor for Openflow 1.3

Network virtualization for Openflow 1.3

Harmjan Treep
4011724

Committee members:
Supervisor: Dr. ir. Fernando Kuipers
Mentor: Dr. ir. Niels van Adrichem
Member: Dr. ir. Johan Pouwelse

June 26, 2017

Abstract

Openflow is the emerging standard for Software Defined Networking, it allows users, among other things, to push forwarding rules to switches. This allows them to determine how the switches handle arriving packets, in other words can they run applications on the network. Such applications can perform diverse tasks beyond just routing the packets; for example making Quality of Service guarantees, monitoring and even firewalling parts of the network.

Most companies don't actually own datacenters and servers anymore; they rent virtual machines from other companies. Such companies may be interested in accessing the Openflow capabilities of switches in the datacenter, but they shouldn't be allowed to push a rule that drops all traffic from a competitor. A Hypervisor can help out in these instances, allowing multiple tenants to use a physical network, while guaranteeing that they can not influence each other.

This thesis presents a network hypervisor for Openflow 1.3, allowing multiple tenants to use Openflow 1.3 features without being able to influence each other's traffic and allowing the network operator to hide network implementation details. A proof of concept implementation called Delftvisor was produced to test the ideas presented in this thesis.

Acknowledgements

This thesis was a long time in the making, I got my thesis entrance permit on September 3th, 2015. Since then I have been working with varying intensity and effectiveness on this thesis, and on June 26th, 2017 I am finally graduating. I would like to thank Fernando and Niels for the support, encouragement and coffee during this process.

Table of Contents

Chapter 1

Introduction

Most modern computer networks work by forwarding packets along a path of devices between hosts. A forwarding device has an internal table telling it how to forward traffic. A forwarding device can be seen as a crossroads and an entry in such a table can be seen as a road sign, describing where to send traffic heading to a specific destination.

Manually configuring these tables is tedious, especially keeping the configuration up-to-date. The topology of the network can change, links between devices can fail and new links can be established. The logic configuring these devices is embedded into the network devices and protocols such as Open Shortest Path First (OSPF) [12] are used to find routes. A single device can only directly look at the state of its own links, protocols like OSPF are used to spread the state information to all devices in the network. The current systems work but make it hard to manage the network [7].

Software Defined Networking pulls apart the control plane and the data plane of the network. An open interface[1] is defined that allows an application to configure the aforementioned road signs in the switch. The packets can also be edited if the switch supports it. This allows an application to configure a network device to be a router, switch or even a middlebox such as a firewall or traffic shaper. The application has direct access to the status of all links and ports and can make routing decisions using all available information. Custom policies can also be enforced, for example given the traffic of a video conferencing application preferential treatment.

Openflow is currently the dominant SDN standard. Openflow 1.0 [14] was released in 2009 and defined an interface to configure a flowtable entry with a priority, match and instructions field. When a packet arrives at the switch, the switch will look in the flowtable and find the rule with the highest priority that matches the packet, and then execute the instructions associated with that rule. This model deemed insufficient for certain applications, take for example a layer 2 learning switch. Such an application needs to look at the destination mac address on each packet to figure out how to forward it and at the source mac address to see if this is a new mac address it can learn. A packet can only match one rule, such an application

[1]This interface is also called the southbound interface

would need to install in the order of $learnedmacs^2$ rules. This problem is also referred to as the flowtable explosion problem.

Openflow 1.3 [15] was released in 2015 and fixes the flowtable explosion problem and introduces other mechanisms such as the group and meter tables. The flowtable explosion problem was fixed by allowing multiple flowtables and instead of outputting a packet allowing the switch to forward it to another internal flowtable. For the example layer 2 learning switch this means matching on the destination mac address in table 1, then forwarding to table 2 where the packet is matched on source mac address. The forwarding actions are executed in table 1 and the learning actions in table 2. Now only in the order of $learnedmacs \cdot 2$ rules need to be installed.

Infrastructure-as-a-service [8] is a model where a provider maintains data centers and leases virtual machines and network infrastructure to customers. This allows a company to have some computers distributed over the globe without having to invest in and maintain data centers.

An infrastructure-as-a-service provider might want to expose the Openflow interface to customers, allowing them to run Openflow applications that for example can perform load balancing or firewalling. The network is shared between different tenants, the same way that a physical computer can be shared between different virtual machines. The provider has to be sure that a misconfiguration by a tenant doesn't influence the traffic of another tenant. An Openflow hypervisor performs that task, it is situated between the tenant controller and the physical network.

The provider doesn't necessarily want to show all the details to the tenant, or the tenant might not care. An Openflow hypervisor can hide some of the details of the physical network, for example making it seem for a tenant that all its virtual machines are connected to one Openflow switch. Another abstraction level would be to show all virtual machines in a data center as to be connected to one switch, while links between data centers are exposed to the tenant.

This thesis presents a solution for virtualizing a SDN network for Openflow 1.3. Every tenant can use Openflow 1.3 features such as multiple flowtables, groups and meters while not interfering with each other. Topology abstraction allows a virtual switch to consist of any combination of physical ports. A proof of concept implementation named Delftvisor was produced, which is the first Openflow hypervisor that can virtualize Openflow 1.3. Delftvisor is about 6000 lines of C++ code. Tenants can write generic Openflow 1.3 applications and safely run them on their view of the network without disturbing other tenants.

1-1 Thesis structure

The current state of Openflow will be discussed in chapter 2. It contains an overview of how openflow currently functions and processes packets. This chapter concludes with a discussion of 2 existing Openflow hypervisors: Flowvisor and OpenVirteX.

The proposed hypervisor will be discussed in chapter 3. The main problems are:

- How to isolate packets over a link

- How to isolate packets going through the flowtable
- How to isolate groups and meters
- How to expose a different topology to a tenant than the actual nework topology

A prototype called Delftvisor has been produced as part of this thesis using the design from chapter 3. The technical details of Delftvisor are discussed in Figure 5-3.

Delftvisor was verified using a set of experiments and some of its properties were measured. These experiments are discussed in chapter 5.

The thesis concludes in chapter 6.

Chapter 2

Software Defined Networking

Current computer networks use devices where the control plane, making the routing decisions, and the data plane, executing these decisions, are on the same device. An example of a protocol that uses devices with a control and data plane is Open Shortest Path First (OSPF). OSPF works by having each device figure out what other OSPF speakers it is directly attached to and flood that information through the network. Each device can then build a graph of the complete network and plan a route for each packet.

Software Defined Networking (SDN) is a new paradigm where the network control layer is separated from the data layer [13]. This creates a central location with a complete overview of the network that defines the policies in the network, making maintenance and modifying policies easier.

Openflow is called a southbound interface, the interface between the Network Operating System (NOS) and the switches. Actual applications run on top of the NOS and use its interface, called the northbound interface. This can be seen in Figure 2-1, the NOS manages the network and allows different applications to make changes. One application can for example do the routing while another is a firewall.

2-1 Openflow

Openflow is an emerging standard for SDN, the first standard was released in 2009. Since then 5 different specifications haven been released, 1.0 to 1.5; 1.0 and 1.3 are currently the most used versions. 1.0 introduced a pretty bare-bone interface, 1.3 introduced support for a lot of new features such as multiple flowtables, the group table and the meter table.

In Openflow the switch initiates a connection to the controller. The main configuration parameter of a switch is the location of the controller. The controller configures the switch further after the connection is made.

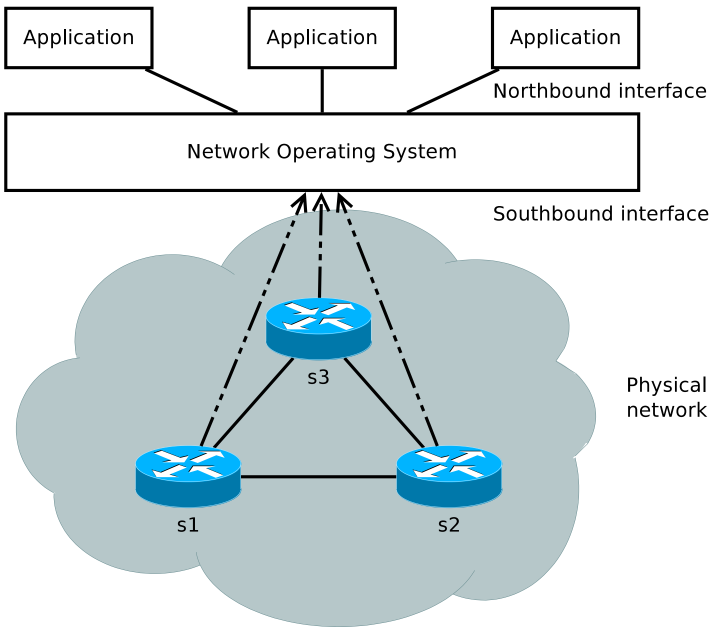


Figure 2-1: Overview of an SDN system with a NOS.

2-1-1 Openflow 1.0 processing

In version 1.0 Openflow defined the messages to manipulate the flowtable of a device. Each entry in the flowtable consists of 4 parts [14]:

- Priority
- Match fields
- Counters
- Actions

A simplified flowchart of Openflow 1.0 processing can be seen in Figure 2-2.

The priority gives precedence to some rules over others, if a packet matches the match fields of more than one rule it matches the rule with highest priority.

The match fields contain values for a subset of all the packet fields with values that must be equal to those fields in a packet for that packet to match this flow entry. The match of a flow entry contain fields such as the Ethernet source and destination addresses but also TCP port. Of these predefined fields a specific value to match on must be provided, except for IP source and destination which can have a subnetmask. The ingress port is a special match field: when a packet enters the switch, the value of the ingress port is added to the packet and a rule can select packets coming only from a specific port.

The counters keep track of how often a rule is matched against and how long this rule has been in the switch. These values can be queried by the controller.

The actions are a list of actions that are executed on the packet sequentially. The possible actions are:

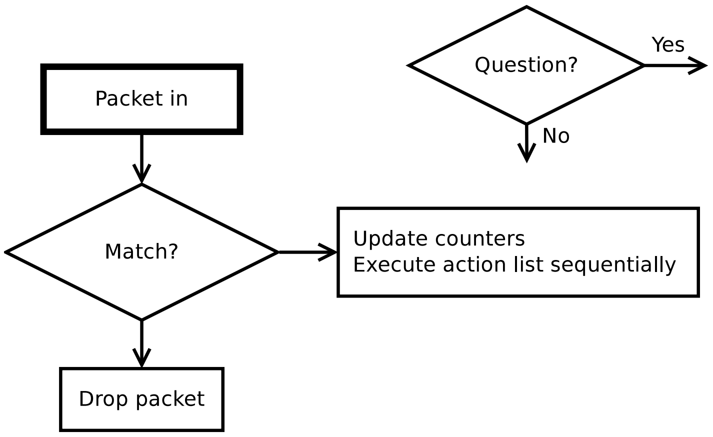


Figure 2-2: A simplified flowchart of Openflow 1.0 matching.

- Forward
- Enqueue
- Modify-Field

If the action list is empty must the packet be dropped.

The Forward action outputs the packet over a specific port on the switch. Special ports exist such as the ALL port that outputs the packet over each port except for the ingress port. The other notable special port is CONTROLLER. This sends the packet to the controller who can inspect it and decide what to do with it, and reactively install a rule in the flowtable to handle all other packets like this one. The controller can also send that packet back to the switch with instructions on how to handle it.

The Modify-Field action can edit a field in the packet. Pre-defined fields, like with the match fields, can be set with this action. One special version removes the Virtual Local Area Network (VLAN) tag from the packet. Trying to set a value in the VLAN tag while there is no tag causes a tag to be added to the packet. No other type of tags are supported in Openflow 1.0.

The Enqueue action sends the packet to a queue attached to a port. This can be used to perform Quality of Service (QOS), preferring some packets over others. What actions the queue's perform depends entirely on the implementation in the switch.

It must be noted that a lot of features in this standard are optional. For example, the Modify and Enqueue actions are optional and some of the match fields are optional. This leads to a lot of implementations being able to say they support Openflow, while it is hard in reality to do something useful with them. This makes it especially difficult to write a generic Openflow solution.

One of the problems with Openflow is that match and modify fields are pre-defined. If you want to write a generic controller that works without dependencies on vendor-specific features you are locked into the list defined in the Openflow 1.0 standard. This doesn't even include IPV6 matching/modifying or using tags other than VLAN tags such as Multi-protocol Label Switch (MPLS). If those become important for Openflow a new standard needs to be released and implemented.

Priority	Match	Actions	Amount
3	eth-src=A, eth-dst=B	Forward(b)	$learnedmacs^2$
2	eth-dst=B	Forward(b), Forward(CONTROLLER)	$learnedmacs$
1	eth-src=A	Forward(ALL)	$learnedmacs$
0	*	Forward(ALL), Forward(CONTROLLER)	1

Table 2-1: Flowtable of a simple Openflow 1.0 Layer 2 learning switch to illustrate the flowtable explosion problem.

Openflow 1.0 flowtable explosion

Another problem with Openflow 1.0 is the flowtable explosion problem. This happens when 2 independent decisions need to be taken over a packet. For example by a Layer 2 learning switch, which learns the mac addresses of the hosts attached to the switch by looking at the source address and routes them based on the destination address. So two decisions need to be made by the switch on each incoming packet:

- Is this a destination mac address I know? If so forward to one port, otherwise forward to ALL port.
- Is this a source mac address I know? If not forward to CONTROLLER.

The full flowtable of this example can be seen in Table 2-1. To make a functioning controller there needs to be a fall-through rule that matches everything and sends the packet to the controller and floods the packet, priority 0 in the example. Then there need to be rules in case a source address is known but the destination is not, and vice versa when a destination is known but the source address is not. Those are the rules with priority 1 and 2 respectively. At the top of the table packets with a known source and destination are dealt with. The total amount of rules for this simple controller is $1 + 2 \cdot learnedmacs + learnedmacs^2$, or simply in the order of $learnedmacs^2$ rules.

2-1-2 Openflow 1.3 processing

Openflow 1.3 improved upon Openflow 1.0 by allowing multiple flowtables and adding other features such as the group table and the meter table. More fields can be matched on and be modified, for example the IPV6 fields. In Openflow 1.0 only the IP field could be partially matched on via a subnetmask, Openflow 1.3 has more fields that support partial matching such as the VLAN-id field.

Every rule in a flowtable has the following attributes [15]:

- Priority
- Match fields
- Counters

- Instructions

Processing works comparable to Openflow 1.0: a packet matches one rule in a flowtable; the rule with the highest priority where the packet fields match, but now a packet can be sent to another flowtable or to a group. Every packet has an action set associated with it in which actions can be added/overwritten/removed by instructions, only at the end are these actions executed. Figure 2-3 contains an overview of processing in Openflow 1.3.

Instructions

Openflow 1.3 makes a distinction between actions and instructions. Entries in the flowtable now contain instructions with what to do with the packet. Instructions edit the actions to perform on the packet or the way the packet is processed. Actions directly edit the packet or cause it to get forwarded to a port or group. The available instructions are:

- Meter
- Apply-Action
- Clear-Action
- Write-Action
- Write-Metadata
- Goto-Table

The Meter instruction directs a packet to the meter table. A meter entry is meant to be able to perform different actions on a packet depending on the speed they arrive at, for example to cap bittorrent traffic at 2MB/s. In Openflow 1.3 meters are mainly used to drop packets if a flow goes over a certain bitrate. If a meter decides to drop a packet it stops processing right at that instruction as can be seen in Figure 2-3.

The Apply-Action instruction has a list of actions and applies all of them to the packet in order. If an output or group action is encountered, a copy of the packet will be made and the copy sent to the port or group. An action list can contain multiple output and/or group instructions and all of them are executed. The Apply-Action instruction can be used to edit a packet in-between flowtables if it is used alongside a Goto-Table instruction.

A packet can go through multiple flowtables, each rule it matches must be able to perform actions on the packet. Rules in later flowtables might want to overwrite actions done in earlier flowtables. This is done via the action set. The action set is a set of actions associated with a packet, it can only contain one type of each action. If in table 0 a packet gets Output(1) added to the action set, and in table 2 it gets the action Output(2) added, the final action set will only contain Output(2). In Figure 2-3 this is called merging the actions sets.

The Clear-Action and Write-Action instructions edit the action set associated with a packet. The Clear-Action instruction clears the action set and the Write-Action instruction adds to the action set, possibly overwriting actions as discussed above.

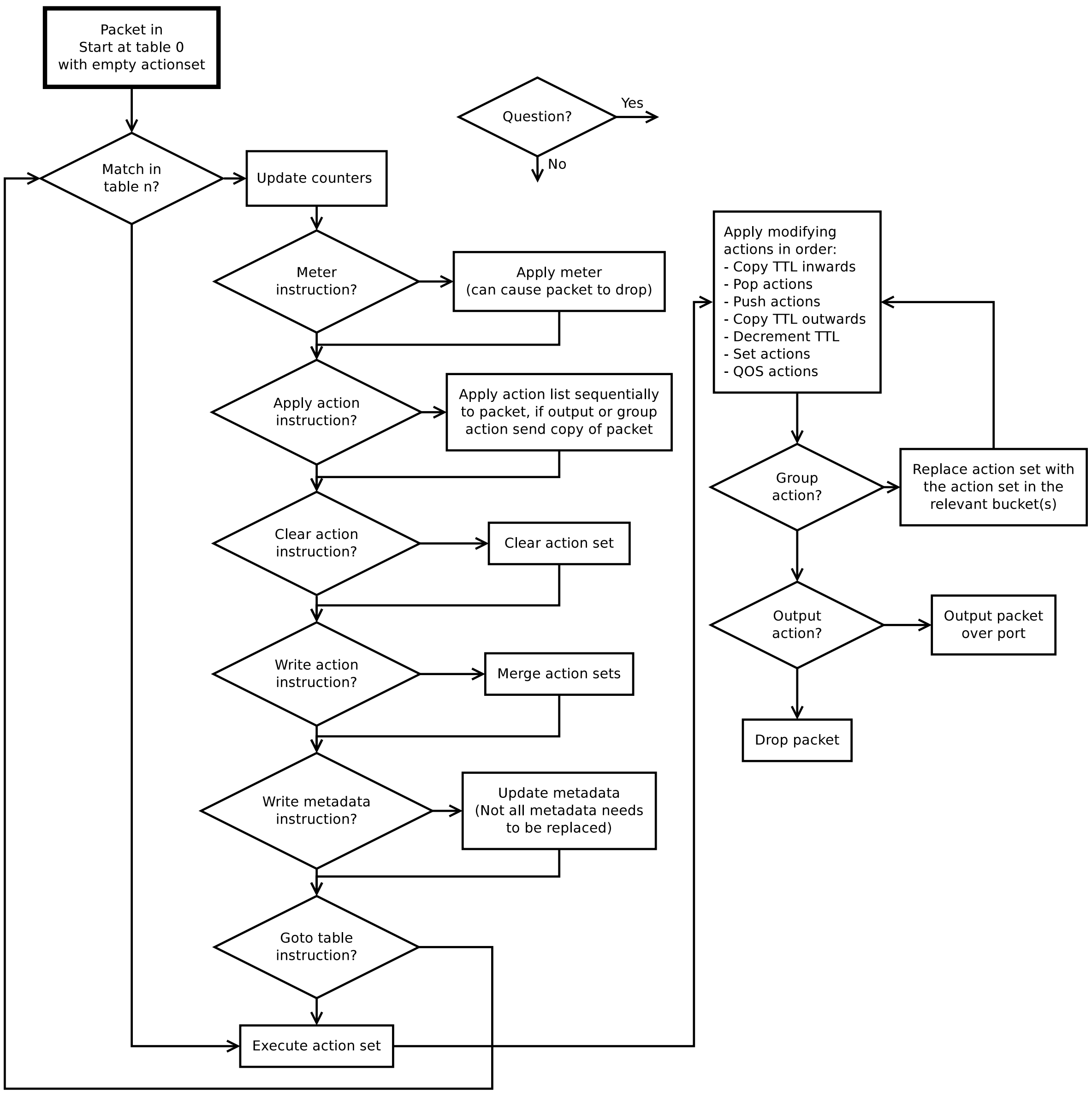


Figure 2-3: A flowchart of Openflow 1.3 procesing.

Another field that gets added to the packet is the metadata field. The associated metadata are 64 bits that can be written to via the Write-Metadata instruction and that can be matched on in the match fields. This data is lost when the packet is outputted or sent to a group. The content of the metadata field is sent to the controller if the packet is output to the controller.

The Goto-Table instruction sends the packet to another flowtable. A packet can only be sent to a flowtable with a higher number than the current one, so no internal routing loops can be created with this instruction. If an instruction set does not include a Goto-Table instruction does processing end and is the current action set executed.

Groups

Groups are another important new feature. A group is identified via a number and contains a list of buckets, each bucket in turn contains an action set. There are four types of groups:

- Indirect
- All
- Select
- Fast-failover

In an All group all the buckets are executed. The packet is copied for each bucket after which the action set in its respective bucket is executed. A packet can however not be sent back over the port it came from via such a group, even if it gets sent to that port in a group later in processing. This can be helpful when creating groups to flood packets along a spanning tree. A packet can be sent out over its ingress port if the special port IN_PORT is used in a bucket.

An indirect group is an all group with only one bucket. This can be useful as a collection point of packets. If for example a lot of different packets need to be sent to a specific host, but that host can move in the network, can an indirect group be useful. If all rules send the packet to the group instead of outputting themselves does the group only need to be updated when the host moves instead of all rules forwarding to the group.

A select group sends the packet to one of the buckets depending on a distribution, these groups can be used to send 75% of traffic over one port and 25% of traffic over another.

Fast-Failover is a special group that sends a packet to the first bucket that is marked live. The purpose is to have backup routes installed when failure is detected locally on the switch. The network can then quickly switch over to these backup routes without intervention from the controller. The mechanism with which to detect failure must be implemented as a hardware feature of the switch.

Note that group buckets contain action sets and not instruction sets. This means that a packet can be sent to other groups, but not back to a flowtable. An internal routing loop can be created this way; it is optional for switches to detect and prevent this.

Flowtable 0

Priority	Match	Instructions	Amount
1	eth-dst=B	apply-action(Output(b)), goto-table(1)	*learnedmacs*
0	*	apply-action(Output(ALL)), goto-table(1)	1

Flowtable 1

Priority	Match	Instructions	Amount
1	eth-src=A	$\emptyset$	*learnedmacs*
0	*	write-action(Output(CONTROLLER))	1

Table 2-2: Flowtables of a simple Openflow 1.3 Layer 2 learning switch to demonstrate the difference with openflow 1.0.

Openflow 1.3 layer 2 router

In Openflow 1.0 the flowtable explosion problem was shown with the example flowtable from Table 2-1. The example redone for Openflow 1.3 looks like Table 2-2. The total number of rules needed is $2 + 2 \cdot learnedmacs$. Note that in flowtable 1 the Apply-Action instruction is used instead of Write-Action. If both tables used Write-Action, then the Output action added in flowtable 0 would be overwritten by the Write-Action(Output(CONTROLLER)) in flowtable 1 priority 0.

2-1-3 The Openflow protocol

The Openflow protocol defines the messages that are sent between the switch and the controller. In version 1.0 of Openflow there are 22 different type of messages, in Openflow 1.3 there are 30. The message types of Openflow 1.0 and Openflow 1.3 can be seen in Table 2-3. This section briefly discusses some of the details of the protocol[1].

During the initial handshake devices negotiate an Openflow version to use. The switch and the controller both sent the versions they support, the highest version both support is used. After that exchange is the connection ready. At this time the controller knows nothing about the switch except the Openflow versions it supports, not its datapath id nor the ports that are attached to the switch. The controller must request all information it wants from the switch. Both parties in this connection can send messages with type EchoRequest to each other and expect a message with type EchoResponse. Both parties can check with Echo messages if the application at the other end is still responding even if the TCP connection still functions.

The message types to request switch information are FeatureRequest and FeatureResponse, where the controller requests the information and the switch responds with the information. In the same vein are the message types GetConfigRequest, GetConfigResponse and SetConfig which allow a controller to request and set the config flags which deal with what happens to fragmented packets. Openflow 1.3 added the GetAsyncRequest, GetAsyncResponse and SetAsync which allows a controller to configure what type of Openflow packets it receives it has not requested. Most Openflow packets are generated by a request from the controller but

[1] A comprehensive overview of all Openflow message types and their fields is, at the time of writing, available at `http://flowgrammable.org/sdn/openflow/message-layer/`

Message type id	**Openflow 1.0**	**Openflow 1.3**
0	Hello	Hello
1	Error	Error
2	EchoRequest	EchoRequest
3	EchoResponse	EchoResponse
4	Vendor	Experimenter
5	FeatureRequest	FeatureRequest
6	FeatureResponse	FeatureResponse
7	GetConfigRequest	GetConfigRequest
8	GetConfigResponse	GetConfigResponse
9	SetConfig	SetConfig
10	PacketIn	PacketIn
11	FlowRemoved	FlowRemoved
12	PortStatus	PortStatus
13	PacketOut	PacketOut
14	FlowMod	FlowMod
15	PortMod	GroupMod
16	StatsRequest	PortMod
17	StatsResponse	TableMod
18	BarrierRequest	MultipartRequest
19	BarrierResponse	MultipartResponse
20	QueueGetConfigRequest	BarrierRequest
21	QueueGetConfigResponse	BarrierResponse
22		QueueGetConfigRequest
23		QueueGetConfigResponse
24		RoleRequest
25		RoleResponse
26		GetAsyncRequest
27		GetAsyncResponse
28		SetAsync
29		MeterMod

Table 2-3: An overview of the message types in Openflow 1.0 and Openflow 1.3, the MultipartRequest and MultipartResponse message types have 15 subtypes.

the switch can also send asynchronous packets such as PacketIn. Via the async message types can the controller control what type of messages it receives. QueueGetConfigRequest and QueueGetConfigResponse are there so the controller can get information about the queues available for a port on the switch.

Openflow 1.3 has the MultipartRequest and MultipartResponse message types. They replace the StatsRequest and StatsResponse message types in Openflow 1.0. Via the multipart messages can a controller ask for information about the flowtable, the ports, the groups, etc. There are 15 subtypes for MultipartRequest and MultipartResponse messages: Description, Flow, Aggregate, Table, PortStats, Queue, Group, GroupDescription, GroupFeatures, Meter, MeterConfig, MeterFeatures, TableFeatures, PortDescription, Experimenter. This means there actually 58 different types of messages in Openflow 1.3, 28 base messages without the multipart messages which each become 15 other message types.

In Openflow 1.0 the ports available on the switch were present in the FeatureResponse message. In Openflow 1.3 these need to be requested separately via the MultipartRequest::-PortDescription message type. Most controllers do not consider the connection fully established until a MultipartResponse::PortDescription message has been received.

The messages with which the controller can modify the way packets are dealt with end in mod. FlowMod allows a controller to add/update/delete entries in the flowtables of the switch. GroupMod and MeterMod do the same for groups and meters respectively. PortMod allows a controller to change the state of a port. TableMod can change how a table handles packets that do not match any rule in that table.

The RoleRequest and RoleResponse message types deal with the role mechanism in Openflow 1.3. Using the role mechanism can a switch connect to multiple controllers, one of those controllers is the master while the others are slaves. Only the master can modify the way in which the switch forwards packets, the slaves can do monitoring and can prepare for the case the current master fails and quickly take over for the current master.

2-1-4 Topology discovery

Controllers need to know the topology of the network they are managing to make routing decisions. A controller needs to know where there are links between switches and when those links appear/disappear, on top of that can a controller be general purpose and usable on all networks if the expected topology isn't hard-coded in it. Their is no built-in Openflow protocol feature to detect the topology. Most frameworks include a module to detect the topology of the network, giving the controller access to a full overview of the network.

Most controllers detect links between switches by sending out packets marked with the switch and port they are sent from. Each switch in turn has a flow rule installed that detects these packets and sends them to the controller. The controller can then figure out from what port the packet was sent and on what port the packet arrives, and infer that a link must exist between those ports. Most controllers use Link Layer Discovery Protocol (LLDP) packets for that purpose, a protocol designed to allow a switch to advertise itself and its capabilities to neighbours. An example of this is shown in Figure 2-4.

This introduces the trade-off how often you want to send out packets over each link. Sending packets more often means you can detect link changes faster at the cost of control channel

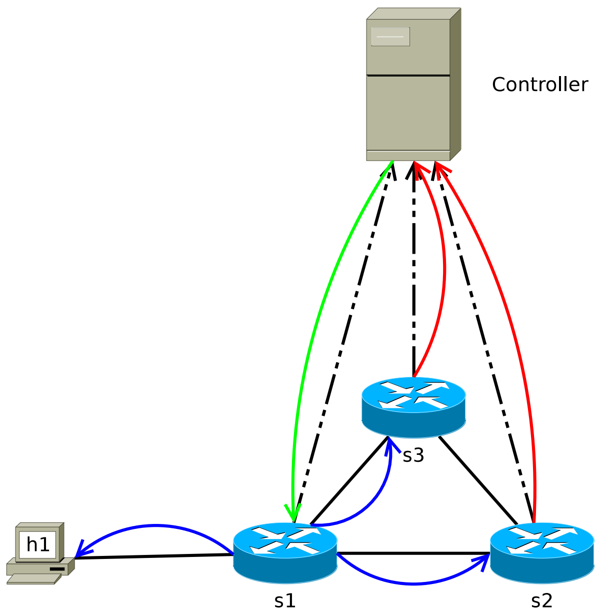


Figure 2-4: An example of topology discovery. The controller sends three packets to switch s1 along the green arrow. The switch outputs those packets over each port along the blue arrows. Every openflow switch has a rule installed sending topology discovery packets back to the controller along the red arrows and hosts ignore them. The controller can now infer the links between s1 and s2 and between s1 and s3.

and data plane overhead. Most controllers prioritise detecting link failure over detecting new links, meaning that these topology discovery packets are sent more often over already discovery links.

2-1-5 The Ryu controller framework

The controller framework used throughout this thesis is Ryu [4]. Ryu is a python framework for which modules can be written that function as a controller. Ryu sits in the place of the network operating system in Figure 2-1. It maintains the connections and parses all the packets.

Ryu is event based, meaning events are generated by Ryu itself and also by Ryu modules. Such events are for example EventSwitchEnter and EventSwitchLeave generated by the topology module. EventSwitchEnter happens when a switch connects to Ryu and Ryu has requested information from that switch. EventSwitchLeave happens when that switch connection is terminated. Other events are when specific Openflow messages are received, such as for example EventOFPPacketIn which occurs when a PacketIn message is received.

Ryu has some built-in modules, the most commonly used module is the topology module. It performs the function discussed in subsection 2-1-4 and emits, among others, the events EventLinkAdd and EventLinkDelete. Ryu can run without this module.

Modules in Ryu register callbacks for events with some logic to deal with those events. Most applications work just by responding to network events by installing rules to deal with them.

2-2 Existing Hypervisor systems

SDN introduces a flexibility in configuring the network. This means that a network is easy to reconfigure for experiments or for different purposes. A network might have different stakeholders: in a university network people need to be able to use the internet, but the networking department might want to run experiments too. In a data center with multiple customers some tenants might want to run a load balancer, while others want to give their teleconferencing application preferential treatment. Both can run on an Openflow switch at the same time via a NOS, as can be seen in Figure 2-1. In the shared data center example the NOS must ensure that a tenant does not accidentally push a rule that drops the traffic from another tenant.

The NOS provides a new API to the applications. Another solution would be to use an Openflow Hypervisor. This can be seen in Figure 2-5. The switches connect to the Hypervisor, which in turn connects to the Openflow controller/NOS with Openflow for each virtual switch. In this model every tenant can run their own flavor of controller/NOS. This type of Hypervisor is what is produced for this thesis and Hypervisors of this type will be discussed below.

2-2-1 Notation

This section will introduce some terms and notation that will be used throughout the rest of the thesis.

A Hypervisor allows multiple tenants to use Openflow on the same network. A tenant is a stakeholder in the network that wants to run a controller on its slice of the network. Every tenant has a slice, which is a part of the network isolated by the Hypervisor. Packets that belong to a tenant's traffic belong in that tenant's slice.

Every network consists of Openflow switches. The switches that are managed by the hypervisor are called the physical switches throughout this thesis. Every switch consists of ports, the ports in the physical network are the physical ports. Physical switches will be denoted as the blue switches in figures, as can be seen in Figure 2-6.

The hypervisor exposes a virtual view of the network to tenants controller, the switches the tenants see are called the virtual switches which have virtual ports. Virtual switches are denoted with grey switches in figures, as can be seen in Figure 2-6. A virtual port usually maps directly to a specific physical port, but this does not have to be the case for a virtual link.

There are two ways a hypervisor can virtualize a link; by exposing a shared virtual link or by exposing a true virtual link. A shared virtual link maps to a specific physical link in the network. That link can be shared by different tenants but all traffic that is sent over that shared virtual link will travel over that specific physical link. A true virtual link can be between any two virtual switches in the network, even if there is no direct physical link between them. If the topology changes can traffic over a true virtual link be rerouted, which is not the case for a shared physical link. The two virtual ports at either end of the virtual link are also truly virtual in that they do not correspond to a specific physical port in the network. The reason for this distinction is that true virtual links are difficult to implement

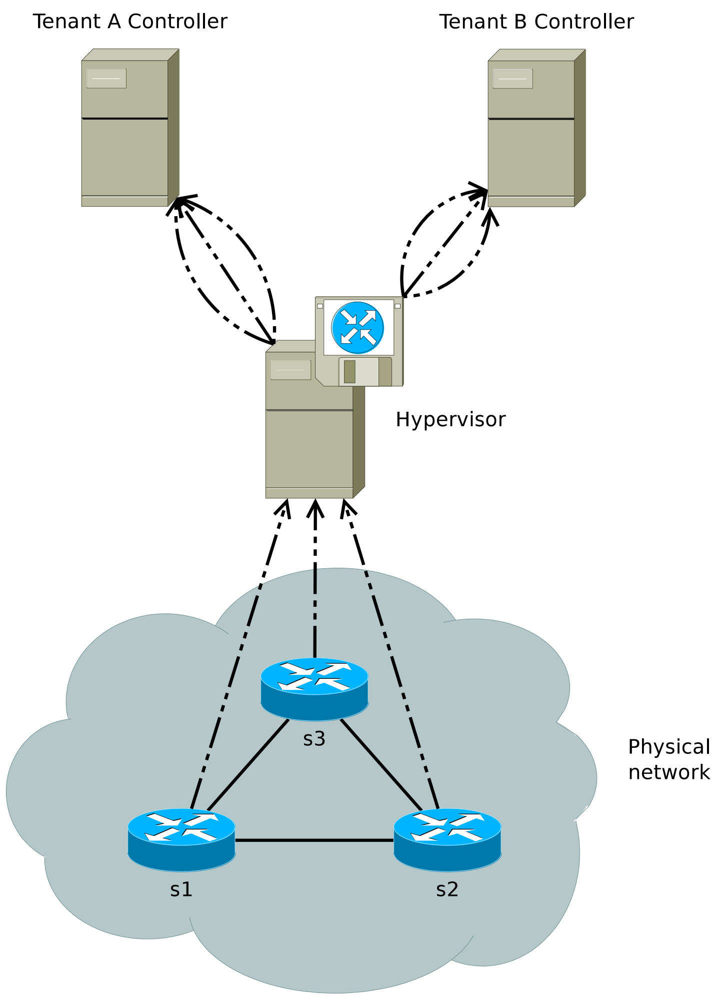


Figure 2-5: Placement of the Hypervisor.

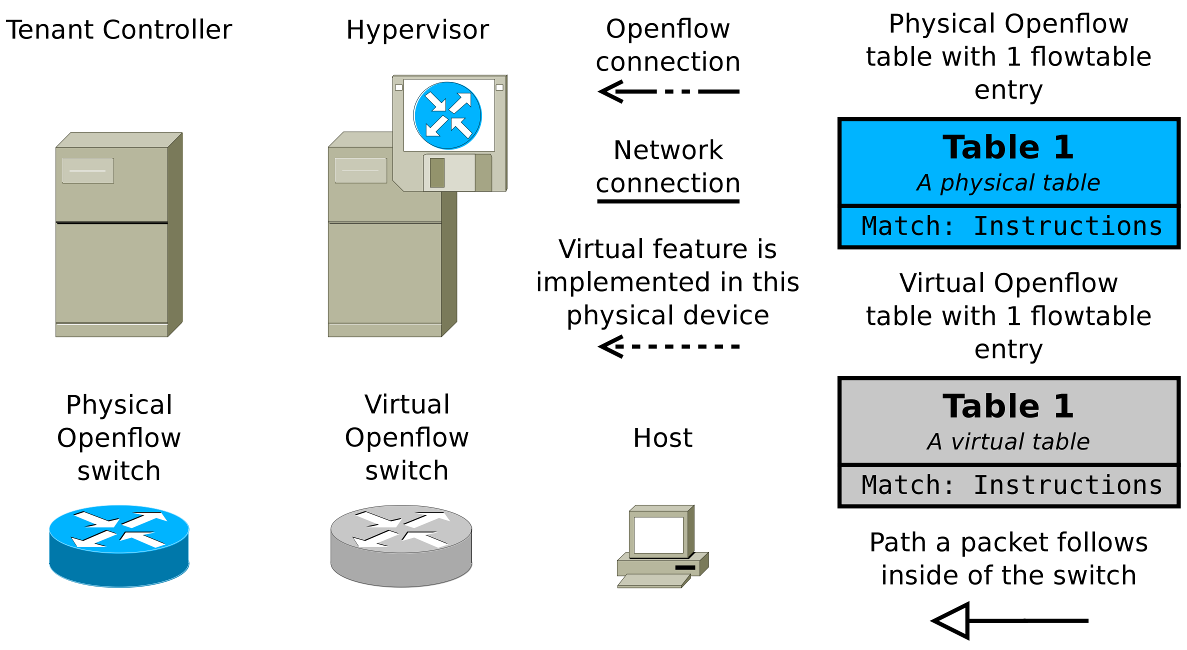


Figure 2-6: Notation used throughout this thesis.

and a lot of hypervisors, including Delftvisor, only support shared physical links and not true virtual links.

Figures in the rest of the thesis will often not show the hypervisor and tenant controllers for the sake of brevity. In all networks are all the physical switches connected to the hypervisor which in turns connects to the tenant controller for each virtual switch in that tenants slice. The mapping between physical ports and virtual ports should be obvious from the example that is demonstrated, to understand how the physical network is virtualized is it often useful to look at what ports the hosts are connected. Hosts are drawn twice, once in the physical network and once in the virtual network. The physical ports connected to a host must correspond to the virtual port in the virtual network. In some cases for clarity are the virtual ports using the notation (physical_dpid,physical_port_number), so (2,3) corresponds to the physical switch with datapath id 2 port number 3.

The design of a hypervisor covers how to handle the packets internally. To show how packets are internally handled is the notation from Figure 2-6 used. A flowtable consists of multiple flow rules which are shown in the format Match: Instructions in each flow table. The highest flow rule in the table is shown at the top, the priority is not explicitly shown but higher rules have a higher priority. If a flow rule has a goto-table instruction or group action in it will there be an arrow going from that flow rule to the target object. Note that groups are not explicitly shown in Figure 2-6 but they look exactly like the flowtables except that instead of Table it says Group and instead of flow rules they have buckets with action sets.

2-2-2 Flowvisor

FlowVisor allows virtualizing an Openflow 1.0 network into multiple independent slices [17]. FlowVisor focuses on isolating the following features:

- Bandwidth
- Topology
- Flowspace
- Flow Entries
- Switch CPU

Bandwidth isolation in FlowVisor is done by editing the VLAN priority bits and using the Quality of Service (QoS) capabilities of the native switch. Whenever a flow rule pushed by a tenant contains a Forward action does FlowVisor add a Set-VLAN-PCP action to map it to one of the eight traffic classes available (every slice has its own traffic class). The meaning of each of those traffic classes needs to be configured on the switch outside of Openflow. The VLAN PCP field is not available to tenants, since the hypervisor uses it.

Topology isolation in FlowVisor is limited to the available topologies. Ports can be removed from switches and switches can be removed from slices, but virtual switches that exist across multiple physical switches are not possible.

Every slice in FlowVisor has a flowspace assigned to it with the type of traffic it handles. An example flowspace can be all TCP port 80 traffic or all traffic in 10.0.0.0/8. A flowspace is a subset of all the bits Openflow can match on.

If a slice in FlowVisor has the flowspace of TCP port 80, that match will be added to all rules it pushes. If the tenant controller pushes a rule that matches everything will the rule in the switch match TCP port 80. If the tenant controller pushes a rule that matches all traffic to the broadcast Ethernet address will FlowVisor add the match for TCP port 80 to the rule before forwarding it to the physical switch. If the match pushed by the tenant controller clashes with the flowspace will FlowVisor reject the rule.

A possible cross-slice attack that still remains is trying to exhaust physical switch resources. This can be prevented by limiting the amount of resources a slice can use; the number of rules a device can have on a switch, the amount of control messages a slice is allowed to send to the switch and the kind of messages to make sure that switch CPU is not blocked by processing those messages.

FlowVisor was deployed in practice on the physical university network of the authors. It allowed them to run several experiments in parallel on a production network.

2-2-3 OpenVirteX

FlowVisor forces slices to use non-overlapping pre-defined flowspaces. OpenVirteX [5] tries to solve this by allocating a virtual flowspace. Another problem OpenVirteX solves is using

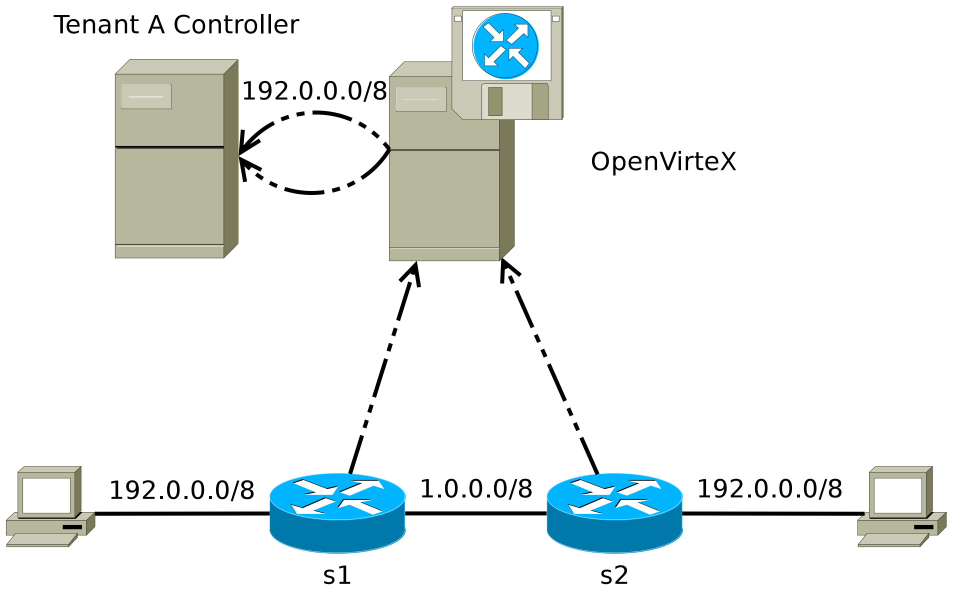


Figure 2-7: OpenVirteX flowspace isolation. IP and Ethernet addresses are rewritten at the edge and internally use a different value.

a virtual topology that allows virtual switches to consist over multiple physical switches and for virtual links to exist that do not map to a single physical link.

Every slice still has a pre-defined flowspace in the IP or Ethernet header space. OpenVirteX at the edge of the network rewrites the addresses by modifying the Openflow rules pushed to handle them. For example if a slice uses the virtual flowspace 192.0.0.0/8 and pushes a rule to an edge switch to forward 192.0.0.1 destination packets to port 2 will that be rewritten to a rule that matches 192.0.0.1 but also rewrites the destination address to 1.0.0.1 if 1.0.0.0/8 is the virtual flowspace for this slice. This can be seen in Figure 2-7. OpenVirteX cannot accommodate a tenant that wants to use the full flowspace, there is still the same amount of available flowspace as with FlowVisor but OpenVirteX allows tenants to use overlapping flowspaces.

OpenVirteX allows for a virtual switch to consist of multiple physical switches, it calls such switches big switches. OpenVirteX does this by forcing rules sent to that switch to have a match on in port [3]. The in port match is rewritten and the rule is sent to the switch that has that virtual port.

If the packet is forwarded to a virtual port that is not on that physical switch, a route is created between the two physical switches. The packet is identified by the same match field as it matched on when it entered the virtual switch. This can route it through physical switches that are part of a different slice, since the packet its internal representation is unique does this not interfere with those slices.

A virtual link is a link that does not correspond to a physical link in the network. It might be between two virtual switches that physically do not even share a link, as seen in Figure 2-8. OpenVirteX even supports precomputed backup routes. If any of the physical devices the current link relies on goes down can OpenVirteX re-route the packet over different hardware, in Figure 2-8 the virtual link can be re-established over the switch at the top.

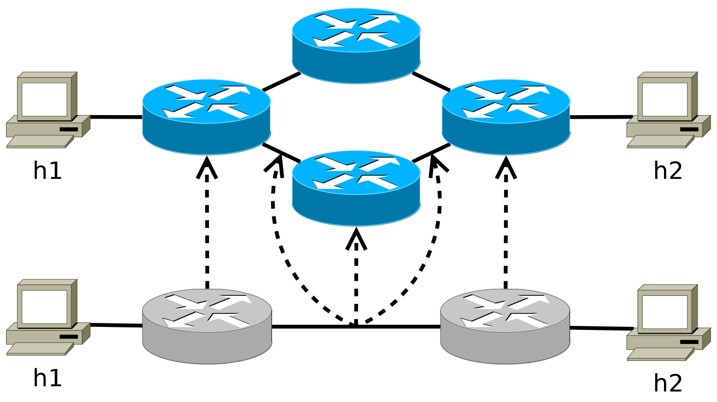


Figure 2-8: Example of an virtual link. The virtual switches are both implemented on the physical switches connected to the host, packets over the link between them are routed over two physical links and a physical switch.

A design limitation of OpenVirteX is that physical switches cannot be part of multiple virtual switches. If a virtual switch consist of multiple physical switches can a tenant not push rules that do not match on a port [3].

Chapter 3

Proposal and Design

This chapter explains the design of Delftvisor. First the requirements of the project will be briefly discussed after which the design will be discussed.

Previously produced Openflow Hypervisors only expose Openflow 1.0 to tenants. The goal of Delftvisor was to create a prototype Hypervisor that can virtualise Openflow 1.3 while exposing as much as possible Openflow 1.3 capabilities to the tenants. Delftvisor should be generic so as much pure Openflow 1.3 features as possible should be used. Delftvisor should support topology abstraction and allow virtual switches that consist of ports on multiple physical switches.

There are Network Operating Systems that can guarantee some kind of isolation between Openflow 1.3 applications, but no Hypervisor that sits between the switch and the controller and speaks Openflow 1.3 both ways. Delftvisor should fill this gap.

3-1 Openflow Hypervisor overview

Delftvisor sits on the connection between the controller and the switch, in the same manner as FlowVisor and OpenVirteX. It needs to speak Openflow 1.3 both ways and allow switches to connect to it and be able to connect to controllers.

The Hypervisor needs to know to what ports hosts are connected and in what slice they are, it needs to know when traffic is at the edge of the network so it can virtualize/devirtualize correctly. Delftvisor does this by running topology discovery. Via the discovered connectivity graph does Delftvisor know when packets stay inside the Hypervisor managed network and when they leave the Hypervisor managed network.

Delftvisor should allow for virtual switches that consist of ports on multiple physical switches like OpenVirteX. This also means that Delftvisor must be able to route traffic towards a goal port. Delftvisor routes by minimizing the amount of hops taken by a packet. After topology changes Delftvisor runs the Floyd-Warshall algorithm on the discovered graph and changes the forwarding rules in the flowtables accordingly.

In Delftvisor, a port on a physical switch can be in one of three states; host connection, link or unused. Ports start in the unused state, via topology discovery can all ports change to the link state. If a virtual switch boots do ports that the virtual switch depends on change to the host state, except if they were already in the link state. If a port is used by multiple virtual switches does it go to the link state, even if a link has not been detected yet over that port.

3-2 Traffic isolation

The physical network is shared between different tenants. This means that the Hypervisor needs to be able to identify the slice a packet that is in transit on the wire belongs to. Since virtual switches might exist on multiple physical switches does the Hypervisor also need to identify switch internal traffic. The information that needs to be encoded is the slice the packet is in, and in case this packet is en route the switch/port this packet is heading to. This section discusses several concepts to achieve this and chooses one.

3-2-1 Link layer/network layer rewriting

As discussed in the previous chapter, OpenVirteX identifies packets in the network by rewriting packets from each slice to a different flowspace. The Hypervisor can detect/configure which hosts are connected to which port on the Hypervisor managed network. In the first flowtable the Hypervisor rewrites the flowspace of the packet with Apply-Action before forwarding to the tenant tables. If a packet arrives over a port that is connected to another Hypervisor managed switch the Hypervisor can forward the packet based on the flowspace the packet is in.

A special slice must be reserved for packets in transit internally in a virtual switch. If a virtual switch consists of multiple physical switches can a packet that arrived on a physical switch be forwarded to a port not on that switch.

This approach limits each slice to a flowspace that all their traffic must be within, so this does not expose the full Openflow capabilities to a tenant.

3-2-2 MPLS/PBB Tagging

Openflow 1.3 allows for more flexible tagging, beside Virtual Local Area Network (VLAN) tags it also allows for MultiProtocol Label Switching (MPLS) and Provider Backbone Bridges (PBB) tags. Openflow 1.3 also supports pushing multiple tags on a packet, as much as the switch hardware allows.

By pushing a tag on a packet before sending it out and removing it before sending it to the tenant flowtables can the Hypervisor expose the full flowspace to a tenant while not removing any Openflow capabilities from tenants. The match fields always work on the outermost tag, so a tenant can use its own VLAN/MPLS/PBB tags without interfering with the Hypervisor.

A PBB tag is also called mac-in-mac, because it is adding an entire new Ethernet link layer to the packet. The entire source/mac addresses are available and are maskable, which means that you can do a partial match on the field. This is important because more information

Destination MAC						Source MAC						VLAN Tag				EtherType		Payload
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19+

Figure 3-1: Distribution of bytes in VLAN Tagged Ethernet frame.

needs to be encoded in the tag, such as target switch and slice id. If the field is maskable, the Hypervisor can create one rule that matches its own switch id. If the field is not maskable, the switch must duplicate that rule for each existing slice, which basically comes down to the flowtable explosion problem. PBB tags would have been the best choice, unfortunately they are not widely supported.

MPLS tagging is widely supported, unlike PBB tagging. MPLS unfortunately is not maskable and the tag replaces the ethertype field of the packet. This means to not put any limits on the tenant controller must the Hypervisor restore the correct ethertype. To support a basic home network, IP (0x0800) and ARP (0x0806) must be supported, and a tenant might want to use VLAN (0x8100, 0x9100) or another tag type. All rules that remove or add the tag must be duplicated for each supported ethertype. This makes MPLS less useful for the Hypervisor.

3-2-3 VLAN Tagging

A VLAN tag adds four bytes between the source mac address and the EtherType, what can be seen in Figure 3-1. Two bytes of those are for the VLAN ethertype, 3 bits for the Priority Code Point (PCP), 1 bit for the Drop Eligible Indicater (DEI) and 12 bits for VLAN Identifier (VID). Openflow 1.3 only exposes an interface to match on and set the PCP and VID fields, of which only the VID field supports masking. This gives a user 15 bits to store information of which 12 are maskable. The PCP field is a separate match field from the VID, meaning that a rule can match only on VID and not on PCP effectively masking the entire PCP field.

There are three different states a packet can be in that a physical switch must be able to recognize:

- Travelling over a shared link in the Hypervisor managed network
- Travelling to a specific switch to be outputted there over a port
- Topology discovery packet from the Hypervisor

In the case that the packet is travelling over a shared link, only the slice this packet is in is relevant. At the receiving switch can then be determined what virtual switch should handle this packet. If a packet is travelling to a specific switch and port is the physical switch and physical port needed. If that port is connected to a host can the packet just be outputted without any problem by popping the VLAN tag off. If the port is connected to a shared link must the slice the packet is in also be available. Topology discovery packets just need to get recognized at all so they can get send to Delftvisor.

To be able to recognize these cases are the available VLAN bits divided over the switch, port and slice identifier. Topology discovery uses a reserved slice and in the case a packet is travelling over a shared link is the port set to a special reserved value. This means that one slice identifier and one port identifier is reserved.

The available bits in the VLAN tag were distributed as follows; 7 bits for the switch identifier, 4 bits for the port identifier and 4 bits for the slice identifier. Topology discovery uses its own slice and the flowtable target uses up a port identifier. This leaves $2^7 = 128$ possible switch identifiers, $2^4 - 1 = 15$ possible ports per switch and $2^4 - 1 = 15$ possible slices. The distribution of these bits can be changed in Delftvisor by changing the constants in tag.hpp and recompiling.

Another approach would be to give each port a globally unique identifier and merge the switch and port fields into one port field. The advantage of the current approach is that with masking can one rule forward all traffic heading to a switch instead of having a separate rule for each port in the network.

3-2-4 Double VLAN Tagging

The original concept was to use two VLAN tags. The inner tag would contain the port and slice fields and if the destination switch was more than a hop away would an extra VLAN tag be pushed that just contained the switch identifier. That tag would be removed one hop away from the destination switch.

There are two different tags, one bit is reserved so a physical switch can recognize which type it is dealing with. The switch tag would have 1 bit reserved for tag type and 14 bits available for the switch identifier. The port tag would also have 1 bit reserved for tag type, and 7 bits each for port and slice. A slice and a port identifier are reserved to handle topology discovery and shared link packets respectively. This concept would give $2^{14} = 16384$ possible switch identifiers, $2^7 - 1 = 127$ possible ports per switch and $2^7 - 1 = 127$ possible slices.

This concept was implemented but OpenVSwitch [16] at the time did not support pushing multiple VLAN tags and actually fails silently, meaning that Openflow processing continues without pushing the extra VLAN tag. OpenVSwitch was the main switch to test the Hypervisor with, so the concept was changed to single VLAN tag as described in the previous section.

3-3 Switch feature isolation

With VLAN tags are the packets on the wire isolated, the switches must also be shared among multiple virtual switches. As much Openflow features as possible, such as multiple flowtables, groups and meters, should be exposed to the tenants.

3-3-1 Hypervisor reserved tables

The traffic on the wire is isolated with VLAN tags. These tags must be invisible to the tenants, with the Apply-Action instruction can packets be edited in-between flowtables. The first flowtables are reserved for the Hypervisor. In these tables packets are routed internally, topology discovery is handled here and packets are identified based on slice and in port and consequently forwarded to the correct virtual switches flowtables.

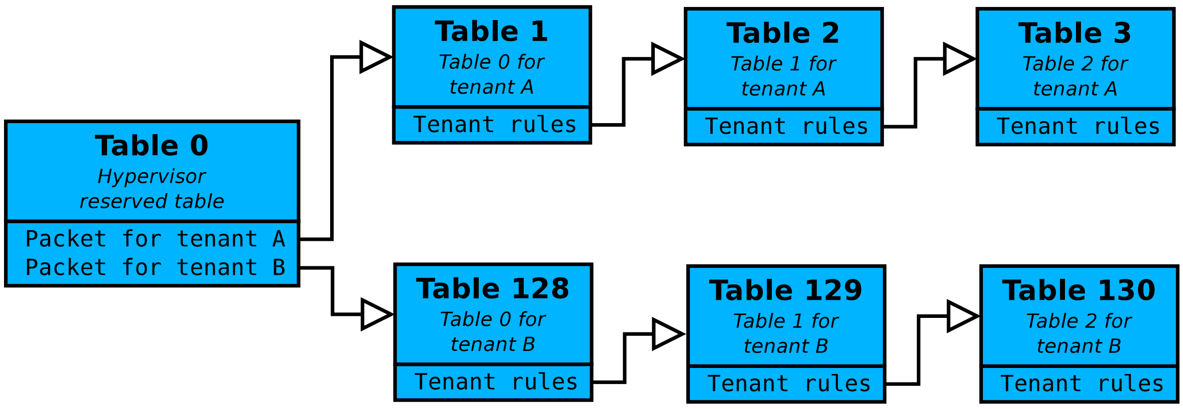


Figure 3-2: A setup where two tenants have a slice of dedicated flowtables. The Hypervisor recognizes which virtual switch the packet is meant for and forwards the packet to the correct flowtables.

Delftvisor reserves two flowtables for the Hypervisor. The first flowtable mainly matches on the in port, if the packet arrived on a host port the packet is forwarded to the correct tenant flowtables. If the packet came over a port that has a shared link the packet is forwarded to the second reserved flowtable. The second flowtable mainly matches on VLAN tag and forwards the packet appropriately.

Multiple reserved flowtables are used so less rules need to be pushed and maintained in a physical switch. The second reserved flowtable has a rule per physical switch in the network that forwards a packet towards that specific physical switch. With only one reserved flowtable would that rule need to be duplicated for each port with a shared link on the physical switch.

3-3-2 Flowtable isolation

While packets on the wire are part of a slice, packets in the switch are part of a virtual switch. Multiple virtual switches that are in the same slice can have ports on the same physical switch, so identifying packets by slice is not specific enough.

Dedicated flowtables

One concept would be to allocate a dedicated amount of flowtables for each virtual switch. There are 255 possible tables, because the table id field is one byte and one value is reserved. Note that supporting multiple tables is optional, and a switch with one flowtable is a valid Openflow 1.3 switch. Delftvisor chose to not support those switches. A solution would be to give each slice a dedicated amount of flowtables, for example if a switch that supports 255 possible flowtables is shared between two tenants the Hypervisor can reserve 1 flowtable for itself, give 127 flowtables to tenant A and 127 to tenant B. A simplified version of this concept can be seen in Figure 3-2, the Hypervisor reserved table recognizes for what virtual switch a packet is meant and forwards the packet to the correct table.

In this concept, when telling the tenant about the features, the virtual switch has only to report the allocated amount of flowtables to that tenant and when a tenant wants to push a rule to the switch increase the flowtable number with the offset for that client. Every Goto-Table instruction also has to have the table offset added to it.

A problem with this approach is that the tables can quickly run out. There are 15 possible slices in Delftvisor which can all have multiple virtual switch inside a physical switch. If each slice has a virtual switch on a physical switch they would only have $\frac{255}{15} = 17$ flowtables each. A physical switch does not have to support all 255 possible flowtables, it can also support only 20. While 20 is probably enough for a single user, sharing those 20 tables among 15 virtual switches could be problematic.

Metadata tagging

Another method to isolate packets is to add a bit of information to the packet and add a match on that information to each virtual switch rule. The metadata feature of Openflow 1.3 is well suited for this purpose. The Hypervisor can add the virtual switch identifier to the metadata before forwarding to the tenant tables, and each time a tenant pushes a rule add a match on the metadata field for that virtual switch identifier. This allows for each tenant to use all flowtables except the Hypervisor reserved tables.

Isolation is guaranteed because all virtual switch rules match on a metadata virtual switch identifier and the only operation that can change the metadata, the Write-Metadata instruction, is protected by Delftvisor. If a Write-Metadata instruction is present in a rule does Delftvisor make sure it does not match the bits used for matching the Hypervisor reserved bits, if it does is the operation rejected and an error is sent to the tenant controller.

This concept was finally chosen because a bit of the metadata was needed to solve another problem, the Output rewrite problem discussed in the next section. The main argument against this concept was that it would limit the Openflow capabilities of the tenants by limiting what they can do with the metadata field. Since the Hypervisor needs to use part of the metadata field, it makes sense to also solve this problem with the metadata field.

This unfortunately means that tenants cannot use those bits in the metadata field. Delftvisor still allows tenant access to part of the metadata field, the metadata field is 64 bits of which Delftvisor needs 13 bits for the virtual switch identifier and 1 bit for the Output rewriting problem.

3-3-3 Port rewriting

The VLAN tag is removed by the Hypervisor reserved tables, now if a packet is output by a virtual switch rule must a VLAN tag be added if the packet stays within the Hypervisor managed network. If the packet is output to a virtual port that is not on this switch does a VLAN tag also needs to be added. If the packet is output to a host port on this physical switch nothing difficult needs to happen except for rewriting the virtual port number to the physical port number.

In place rewriting

An approach would be to rewrite the instructions in place. If for example an Apply-Action instruction was pushed could the action list be rewritten. Take for example the following action list:

```
[ set−source−ip(192.168.0.1), output(2), output(3) ]
```

This action list can be rewritten by surrounding all output actions that output the packet to shared link or to a non-local port with push-vlan and pop-vlan actions. In this example virtual port 2 stays in the network and virtual port 3 is a local host port:

```
[ set−source−ip(192.168.0.1), push−vlan, set−vid(0xA), set−pcp(0x3),
   output(5), pop−vlan, output(6) ]
```

This works well for action lists since those actions are applied in place, a tenant can even use their own VLAN tags if they want to and the switch supports multiple VLAN tags.

Openflow 1.3 also has action sets that get edited by the Write-Action and Clear-Action instructions. These actions are not directly executed but get attached to the packet and only get executed at the end. Meanwhile a lot can change with these actions.

An intuitive approach would be to add the `push−vlan`, `set−vid`, `set−pcp` actions to each action set with an `output` action. This would limit the tenants their Openflow capabilities since it would stop them from using the VLAN actions in action sets. It would work when merging actions sets with a new output action, the VLAN actions would be overwritten with the new actions. A problem would occur when merging action sets with a `group` action, the `group` action will overwrite the `output` action but not remove the VLAN actions added to the set. As can be seen in Figure 2-3 are those VLAN actions executed before forwarding to the group buckets action set.

A solution to this problem would be to reserve one bit of the metadata that explains if the action set has Hypervisor VLAN actions in it. This means that all rules that have an `output` action in the Write-Action instruction must also get a Write-Metadata instruction that sets that bit. Every rule that adds a `group` action to the action set needs to be duplicated, the first rule matches the case that there are no VLAN actions in the action set and is identical to the normal rule with the virtual identifiers rewritten. The second rule matches the case that there are VLAN actions in the action set and forwards to a special group that is created for each tenant group. The duplication of rules and groups can be seen in Figure 3-3.

Delftvisor routes packets internally in a virtual switch if that virtual switch spans across multiple physical switches. It does this by the topology is has discovered using topology discovery. The topology of the network might change and the Hypervisor needs to be able to deal with that. The only way to do that while rewriting in place is by saving all the rules in the Hypervisor and overwriting the rules in the physical switches when the topology changes.

Summarising: this concept is hard to deal with when the topology changes, reserves all VLAN actions for the Hypervisor and requires duplication of rules and groups.

Indirect group per port

The other generated concept to rewrite virtual ports to physical ports is to create a group of type INDIRECT for each virtual port in a physical switch. The action set contains all actions that need to be applied to a packet before sending it out. All output actions can then be rewritten to group actions in all action sets and action lists.

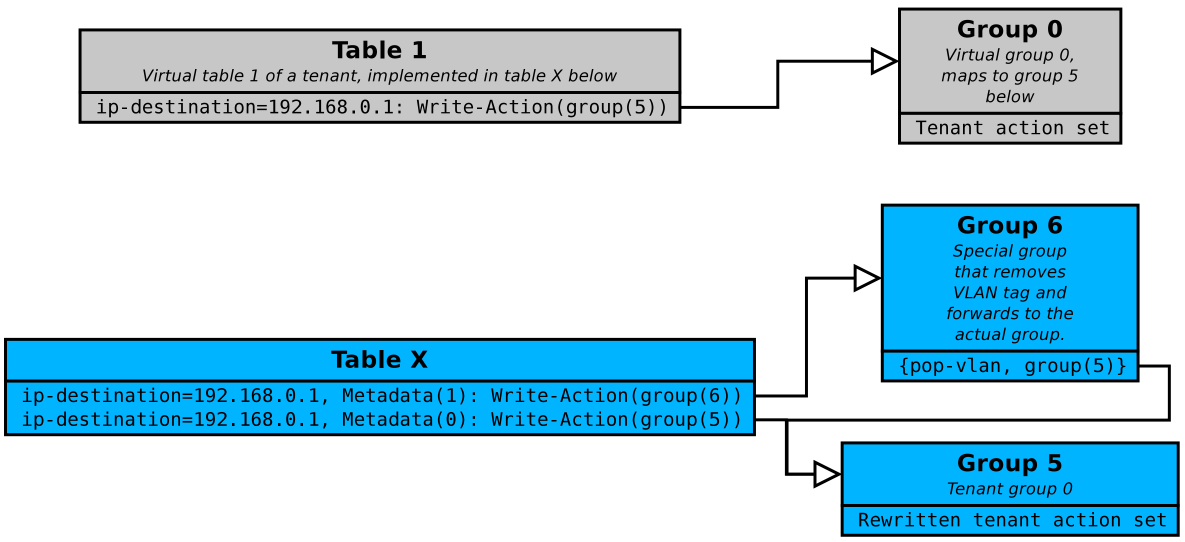


Table 1
Virtual table 1 of a tenant, implemented in table X below
ip-destination=192.168.0.1: Write-Action(group(5))
Group 0
Virtual group 0, maps to group 5 below
Tenant action set
Group 6
Special group that removes VLAN tag and forwards to the actual group.
{pop-vlan, group(5)}
Table X
ip-destination=192.168.0.1, Metadata(1): Write-Action(group(6))
ip-destination=192.168.0.1, Metadata(0): Write-Action(group(5))
Group 5
Tenant group 0
Rewritten tenant action set

Figure 3-3: An example solution to make in place rewriting of ports work. The table and group at the top are a virtual view that the tenant has of the switch, the bottom table and groups are what is pushed to the physical switch.

Since all packets that are forwarded now go to one group, it is easy to change the actions applied to a packet when the topology changes. Delftvisor updates all virtual port groups that need to be updated when the topology changes. An experiment that measures the recovery time is included in section 5-4.

Output rewrite problem

Creating a group for each virtual port and maintaining those groups is pretty straight forward. Every `output` action becomes a `group` action, this creates a problem when merging action sets. If a packet has the following action set attached to it:

```
{group(5)}
```

and it has to merge with the following action set:

```
{output(3)}
```

would that result in:

```
{group(5),output(3)}
```

and following the rules with which Openflow 1.3 processes packets, as seen in Figure 2-3, is only the `group` action executed.

If we redo this example but all `output` actions are rewritten to `group` actions does the virtual `output` action overwrite the `group` action.

To solve this problem Delftvisor reserves one bit in the metadata field that indicates if the action set currently contains a `group` action. Every rule a tenant controller sends to Delftvisor is duplicated, once with the `output` action and once without the `output` action. If an action set in a Write-Action instruction contains a `group` action a Write-Metadata instruction is added that sets the group bit. So if a `group` action is present in the action set will a rewritten `output` action not be added to the action set while if there already is a rewritten `output` action in the action set will it be overwritten by the new `group` action. The end result being that `group` actions have precedence over `output` actions as is defined in the Openflow 1.3 standard.

An `output` action being added to an action set while a `group` action is in there is not a common occurrence. It is important however that controllers can write generic programs, if an Openflow 1.3 program works on physical switches it should work with Delftvisor. This solution requires one bit in the metadata field, the author thinks this is preferable to having wonky action set merging since the Write-Action instruction is mandatory and the Write-Metadata action is optional.

Not all rules need to be duplicated, rules in tenant table 0 cannot already contain a group action so do not need to be duplicated and rules that do not contain an output action do not need to be duplicated. They can not match on the group bit in the metadata. Delftvisor currently does not have this implemented, all the rules currently sent to the virtual switches are rewritten and duplicated.

Special ports

There are still some edge cases that need to be dealt with. The special port ALL and FLOOD must be simulated for each virtual switch, this is done with a group of type ALL that forwards to all virtual switch port groups. There also needs to be a CONTROLLER port simulated, this is done on each physical switch by group 0 that just forwards to the actual CONTROLLER port. The metadata is sent with the packet when a packet is forwarded to the controller, using this information can Delftvisor figure out for which virtual switch this packet was headed.

In port matching

The previous section deals with rewriting the port when outputting the packet, a tenant can push a rule that matches on in-port. This is rewritten by Delftvisor to the actual port this virtual port maps to. If the virtual switch spans over multiple physical switches can the in-port match field only be rewritten for the physical switch that has the physical port that the virtual port rewrites to, so only sent the FlowMod message to that physical switch.

Figure 3-4 contains an example of how ports are rewritten in Delftvisor. It leaves out all the other mechanisms that Delftvisor uses.

3-3-4 Group/Meter isolation

Groups in Openflow are identified by a number. Per physical switch a mapping between the physical numbers and the virtual numbers must be kept. If an Openflow message uses a group number that is not used yet Delftvisor reserves a new id in the physical switch and save the mapping. This mapping is maintained per physical switch, meaning that a virtual group

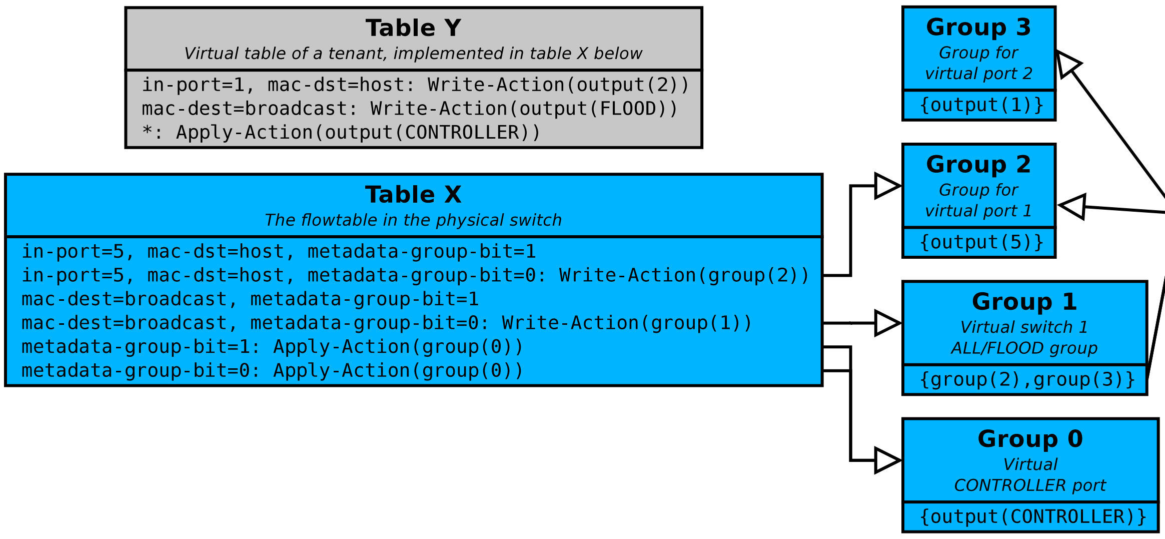


Table Y
Virtual table of a tenant, implemented in table X below
in-port=1, mac-dst=host: Write-Action(output(2))
mac-dest=broadcast: Write-Action(output(FLOOD))
*: Apply-Action(output(CONTROLLER))
Table X
The flowtable in the physical switch
in-port=5, mac-dst=host, metadata-group-bit=1
in-port=5, mac-dst=host, metadata-group-bit=0: Write-Action(group(2))
mac-dest=broadcast, metadata-group-bit=1
mac-dest=broadcast, metadata-group-bit=0: Write-Action(group(1))
metadata-group-bit=1: Apply-Action(group(0))
metadata-group-bit=0: Apply-Action(group(0))
Group 3
Group for virtual port 2
{output(1)}
Group 2
Group for virtual port 1
{output(5)}
Group 1
Virtual switch 1 ALL/FLOOD group
{group(2),group(3)}
Group 0
Virtual CONTROLLER port
{output(CONTROLLER)}

Figure 3-4: An example of how port rewriting works in Delftvisor. Table Y is the view of the switch the tenant has, while Table X contains how the rules are implemented in the physical switch. For every virtual port a group is created that forwards packets to the correct physical ports.

that is installed can rewrite to a different physical group id for each physical switch that the virtual switch depends on.

The actions in the buckets need to be rewritten the same way as the actions in a Write-Action instruction. If the bucket contains a `group` action do not rewrite any `output` action in the action set, only rewrite and add the group action.

The exact same mechanism is used for meters, they are also identified by numbers. Delftvisor has currently not implemented this for meters since OpenVSwitch does not support meters which makes it hard to test.

3-4 Bandwidth isolation

Another important network resource to isolate is bandwidth; every slice should have a guaranteed amount of bandwidth allocated to it. Openflow 1.3 exposes more Quality of Service (QoS) primitives via a standardized interface, mainly meters.

The Hypervisor already has a reserved flowtable that packets first go to to handle removing the VLAN tag. For each slice a meter is created that drops packets after a certain rate. Each rule that forwards packets to the tenant tables also includes the meter instruction forwarding to the meter for that slice. This ensures tenant flowtables only handle as much packets as the rate allows.

This hard limits each slice meaning that if any bandwidth is left after all slices used their part is that bandwidth wasted. In the case of an Infrastructure as a Service provider might that be preferable since customers have to pay for extra bandwidth.

In Delftvisor is traffic that already went through a tenant flowtable not metered again if it needs to be output somewhere else in the network. This can result in a slice using more than

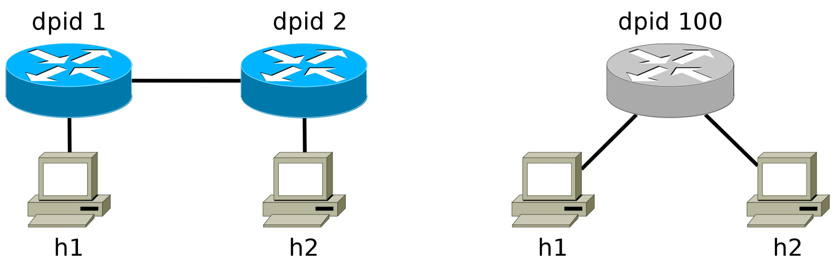


Figure 3-5: An example of topology abstraction. The physical network has two switches, the virtual switch with dpid 100 consist of the ports connected to host h1 and h2, both on different physical switches. Via topology abstraction can some of the details of the network be hidden from a tenant.

the allotted rate over a link. It was deemed that the case that a virtual switch has to route over a link that is also part of the slice is unlikely in a production scenario.

OpenVSwitch unfortunately does not support meters yet, since OpenVSwitch is the main testing platform for this project was bandwidth isolation dropped. The code that setups and configures the meters in the switches was written for Delftvisor, the configuration file for Delftvisor contains an option to re-enable meters. This option is not tested on actual Openflow switches.

A virtual switch can use meters themselves in this solution, a packet in Openflow 1.3 can be metered multiple times. Since meters do not work in OpenVSwitch has the meter rewriting not been implemented in Delftvisor.

3-5 Topology abstraction

With topology abstraction can Delftvisor hide some details of the physical network for the tenants. This allows an Infrastructure as a Service provider to hide some unimportant details so a tenant is only involved in making important routing decisions. The two main ways to abstract topology is by allowing virtual switches to consist of ports on different physical switches and by allowing virtual links, links in the network that are not mapped to a single physical link.

3-5-1 Virtual switches

Virtual switches consist of ports on physical switches. In Delftvisor, these ports do not necessarily have to be on the same physical switch. If a packet has to be forwarded to a port that is not on the current physical switch does Delftvisor route the packet isolated from all virtual switches. An example of such a topology can be seen in Figure 3-5. In this example does a virtual switch consist of two ports that are on different physical switches.

This leads to the question where packets should be handled, should they be handled on the physical switch they arrived at or should one physical switch be chosen to handle all packets and forward all packets to that switch. Delftvisor handles all packets on the switch they arrived at. This is advantageous when for example a packet is sent out again over the port it came in over, this packet now has to travel over less links. The disadvantage is that the flowtable must now be duplicated on each of the physical switches that have ports on them. A special case is a rule that matches on in-port, these rules are only installed on switches that actually have the physical port equivalent of the virtual port.

Another problem is now when a virtual switch boots, when should Delftvisor start the connection to the tenant controller. In the configuration of Delftvisor is the mapping between physical and virtual ports, if all virtual ports are on one physical switch can the virtual switch start when that physical switch connects to Delftvisor. In the case a virtual switch depends on ports on multiple physical switches can the virtual switch not start when all physical switches have connected, Delftvisor must be able to route packets between all depended on physical switches. Otherwise there would be situations where part of a virtual switch could not send packets to ports on another part of the same virtual switch. If every depended on physical switch has a route to each other depended on physical switch can the virtual switch start. The virtual switch must stop again when that is not the case anymore.

Delftvisor detects this by running topology discovery. After the topology has changed is the Floyd-Warshall algorithm [10] run on the discovered topology. Floyd-Warshall computes a shortest path for all possible node pairs in the graph. For each physical switch are forwarding rules in Delftvisor reserved table installed/updated to all other physical switches. The virtual port group are also updated if a packet must be routed differently after the topology has changed. For each virtual switch is checked if all depended on physical switches are mutually reachable. This is done by selecting the first switch in the list of depended on switches and then checking if all other switches have a route to that switch in the result from Floyd-Warshall. If every depended on physical switch can reach the same depended on physical switch must they be able to reach all other depended on physical switches. With this result is decided if the virtual switch should start/stop.

3-5-2 Virtual links

A virtual link is a simulated link in the network, it does not map to a single real physical link. Figure 2-8 is an example of a virtual link as OpenVirteX supports. When the topology changes are the packets in the link routed over a different path by OpenVirteX. Backup routes can even be precomputed and preconfigured by OpenVirteX.

Virtual ports in Delftvisor are already mapped to groups with type INDIRECT. Rewriting these when topology changes is easy and is already done to allow virtual switches that consist of multiple physical switches. It is unfortunately difficult to rewrite matches on in-port, if a virtual link needs to be moved must all rules that match on the current in-port be updated. This means that Delftvisor would have to start keeping track of all the rules pushed to the tables and in the case the in-port needs to be rewritten resend these rules to the physical switches, all while guaranteeing that timeouts set in these rules are not changed. Since the engineering burden of this feature was immense it was decided to not implement this.

3-5-3 Migrating the network

Delftvisor does not have a copy of the tenant flowtable, if an Openflow packet arrives at Delftvisor does it rewrite it and forward it without saving the rule anywhere. This means

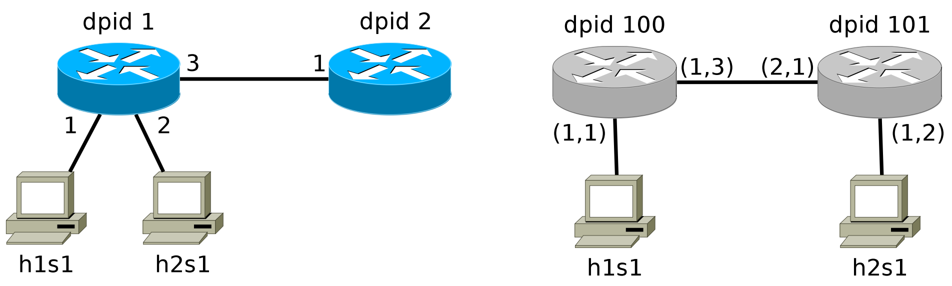


Figure 3-6: A topology with the flood group problem, the ports in the physical network are numbered and the ports in the virtual network consist of (dpid,port_no). A packet that arrives over the port (2,1) in virtual switch dpid 101 and is forwarded to the FLOOD port needs to be send back over the link it came in on which Openflow does not do.

configuration is difficult to change transparently to a tenant. If a virtual port needs to be moved do you have the same problem as with virtual links, all in-port matches need to be changed. The only way to currently accomplish this is by stopping the virtual switch, moving the port and restarting the virtual switch.

3-6 Delftvisor problems

Delftvisor still comes with a few built-in problems. This section discusses those the author is aware off.

3-6-1 FLOOD/ALL ports and groups with type ALL

A special Openflow rule is used for the FLOOD/ALL port and for groups with type ALL. If a bucket from such a group/port outputs over a port that the packet came in on is the packet dropped instead, if the packet needs to be outputted over the port it came in from does the special IN_PORT port need to be used in a bucket. This is problematic because a packet might have arrived over a shared link and needs to be outputted on a virtual port for which the route is through the shared link. Figure 3-6 contains a topology that has this problem.

If the tenant controller of the virtual switches only pushes a rule that floods all traffic over all ports should host h1s1 and h2s1 be able to reach each other, there are no routing loops in this topology. The switch with dpid 2 is only depended upon by virtual switch with dpid 101. Three groups are created to simulate the virtual ports on that virtual switch, a FLOOD/ALL group, a group for port (2,1) and a group for port (1,2).

Almost all routing controllers have this problem since the ARP packet is sent to the broadcast Ethernet address, which in most controllers needs to be flooded using the FLOOD port or a custom ALL group.

3-6-2 Tenant tag overwriting

As discussed previously does OpenVSwitch not support pushing multiple VLAN tags. If it encounters multiple `push−vlan` actions it does not execute the second instruction and continues processing. No error is reported to the controller. If multiple VLAN tags worked in Open-VSwitch could the tenant use VLAN tags themselves, unfortunately this currently means the Hypervisor overwites and removes the added tenant tags. Delftvisor currently cannot detect that this happens.

3-7 Running Delftvisor on Delftvisor

One of the original goals of Delftvisor was to be able to run on top of itself. This in the end is not tested because double VLAN tagging is currently not supported in OpenVSwitch. Delftvisor should be able to run on top of itself at least four times, after which the metadata bits are exhausted. There are 64 available metadata bits, Delftvisor uses 14 of those. 13 for the virtual switch id and 1 to identify if a group action has been added to the action set.

Delftvisor reserves two flowtables for its own use, so if a physical switch has less than 9 flowtables is that the limiting factor before the metadata bits run out. Delftvisor also creates a lot of groups in each physical switch, one for each virtual switch, one for each virtual port and one as the CONTROLLER virtual port. The amount of groups a physical switch can have is limited, the maximum group id is 0xFFFFFF00 or 4294967040 in base 10. Openflow switches do not have to support that amount of groups, so the amount of Delftvisors that can run on top of each other can also be limited by the amount of groups. The same can be set about the amount of flowtable entries, the Content Addressable Memory (CAM) is limited in a switch and Delftvisor can quickly use up a lot of it because it proactively installs flowtable entries.

Chapter 4

Implementation Details

The previous chapter discussed the design decisions taken and why they were taken. This chapter discusses some of the implementation details for Delftvisor. The code is available at https://github.com/TUDelftNAS/Delftvisor/.

Delftvisor's only goal is to demonstrate the ideas from chapter 3, not all features that are needed to make it production software are available. Delftvisor does not support encrypting the Openflow channel or certificates to authenticate switches. Openflow Packets are assumed to be correctly formatted and Delftvisor can be made to crash or even exploited by sending malformed packets.

4-1 Configuration format

Delftvisor takes a json file as input. It is the only configuration that changes behaviour, the rest of the command line arguments can configure the type of log messages that are printed and the number of threads. Running Delftvisor with more than one thread is not supported and will likely crash because of race conditions. During development was multi-threading dropped because the only purpose of Delftvisor is to demonstrate the isolation properties discussed in chapter 3 and multi-threading does not matter for that purpose.

The main thing to configure is the mapping between virtual ports and physical ports. The example below configures the topology as seen in Figure 3-5. Two physical ports on physical switches with dpid 1 and 2 are abstracted into one virtual switch with dpid 100.

```
{
  "switch_endpoint_port" : 6633,
  "use_meters"           : false,
  "slices" : [
    {
      "controller" : {
        "ip"   : "127.0.0.1",
        "port" : 6653
      },
      "max_rate"         : 500,
      "virtual_switches" : [
        {
          "datapath_id" : 100,
          "ports" : [
            {
              "virtual_port"         : 10,
              "physical_datapath_id" : 1,
              "physical_port"        : 1
            },
            {
              "virtual_port"         : 11,
              "physical_datapath_id" : 2,
              "physical_port"        : 1
            }
          ]
        }
      ]
    }
  ]
}
```

The configuration file configures slices, which consist of virtual switches, which consist of ports which refer to specific physical ports.

The `max_rate` property sets that value in the meter limiting this slice and the `use_meters` property tells Delftvisor to try to install meter configuration. As discussed in section 3-4 does OpenVSwitch not support meters so this feature was not tested, setting `use_meters` to true enables again what was produced. All necessary rules are implemented in Delftvisor to enable bandwidth isolation but this was never tested. There are more configuration examples in Appendix A.

4-2 Full flowtable layout

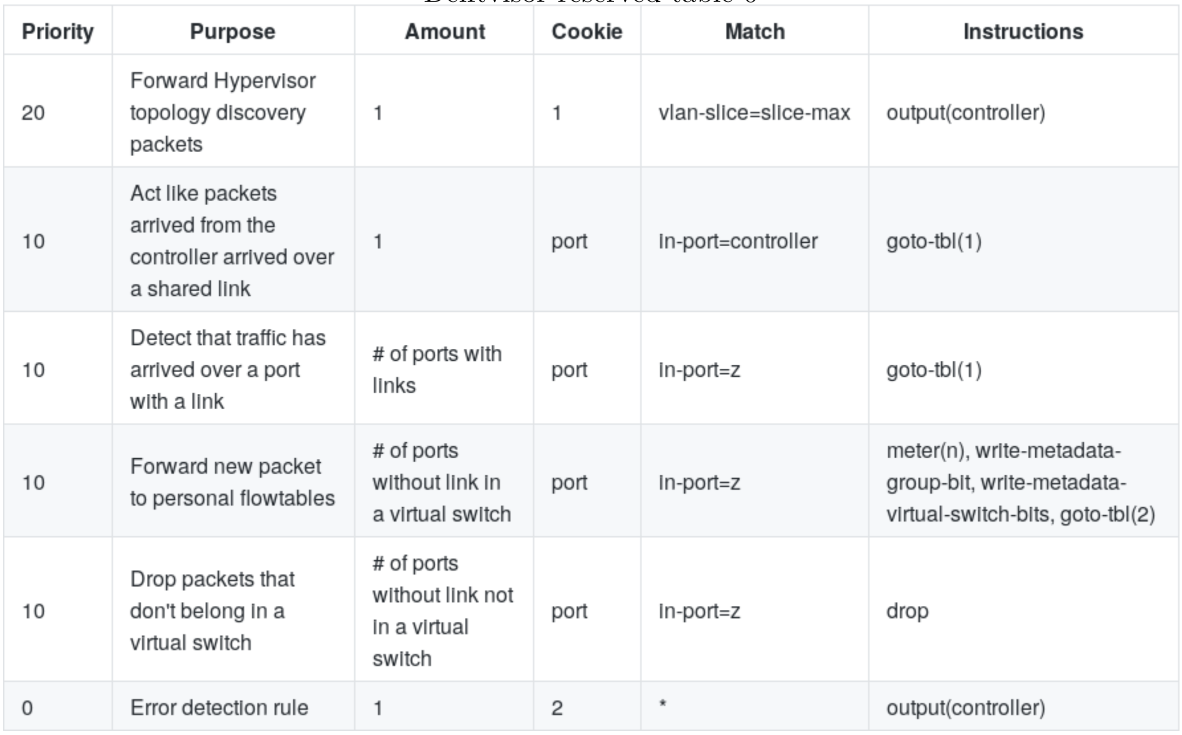
The previous chapter has not presented the full designed flowtable yet. The full flowtable can be seen in Table 4-1. Instead of precisely describing what bits and fields of the vlan field are matched on does it use the convention of using vlan-slice=1, meaning that the slice bits in the vlan field evaluate to identifier 1.

Delftvisor installs rules proactively instead of reactively. This means that the number of rules in the Amount column in Table 4-1 is the amount of rules that are always used by Delftvisor.

4-3 Openflow messages handled

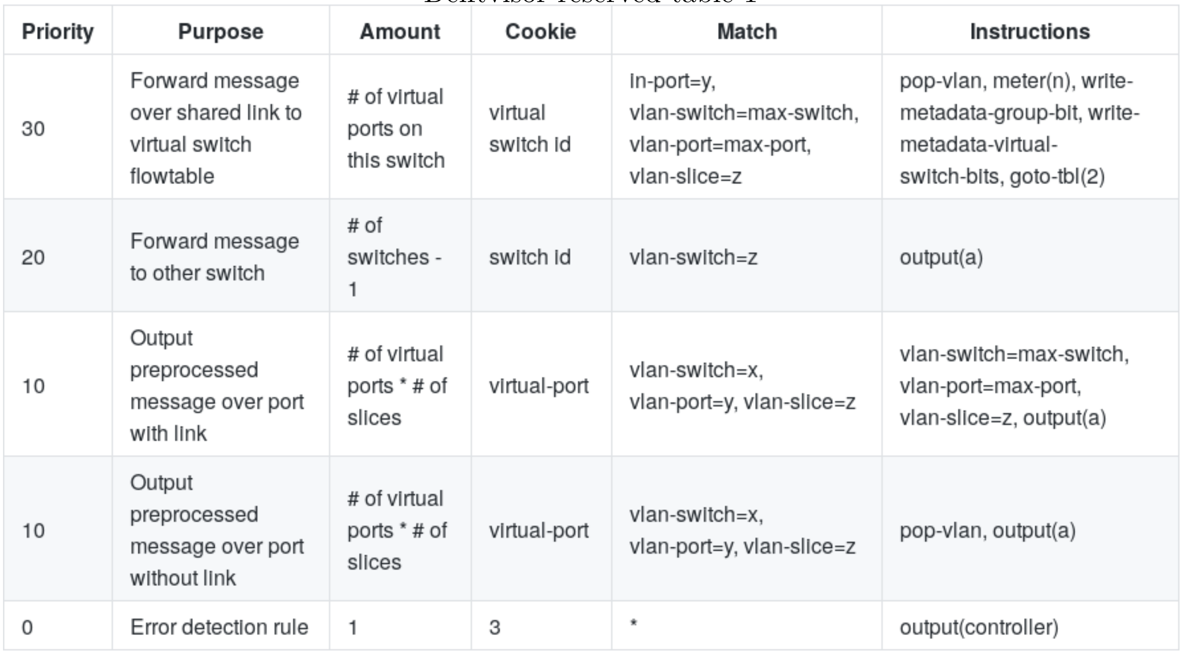
In subsection 2-1-3 are the message types available in Openflow briefly discussed. Openflow 1.3 has 30 base message types, of which the MultipartRequest and MultipartResponse consist of 15 subtypes. This means there are 58 different types of messages to deal with. Delftvisor

Delftvisor reserved table 0



Priority	Purpose	Amount	Cookie	Match	Instructions
20	Forward Hypervisor topology discovery packets	1	1	vlan-slice=slice-max	output(controller)
10	Act like packets arrived from the controller arrived over a shared link	1	port	in-port=controller	goto-tbl(1)
10	Detect that traffic has arrived over a port with a link	# of ports with links	port	in-port=z	goto-tbl(1)
10	Forward new packet to personal flowtables	# of ports without link in a virtual switch	port	in-port=z	meter(n), write-metadata-group-bit, write-metadata-virtual-switch-bits, goto-tbl(2)
10	Drop packets that don't belong in a virtual switch	# of ports without link not in a virtual switch	port	in-port=z	drop
0	Error detection rule	1	2	*	output(controller)

Delftvisor reserved table 1



Priority	Purpose	Amount	Cookie	Match	Instructions
30	Forward message over shared link to virtual switch flowtable	# of virtual ports on this switch	virtual switch id	in-port=y, vlan-switch=max-switch, vlan-port=max-port, vlan-slice=z	pop-vlan, meter(n), write-metadata-group-bit, write-metadata-virtual-switch-bits, goto-tbl(2)
20	Forward message to other switch	# of switches - 1	switch id	vlan-switch=z	output(a)
10	Output preprocessed message over port with link	# of virtual ports * # of slices	virtual-port	vlan-switch=x, vlan-port=y, vlan-slice=z	vlan-switch=max-switch, vlan-port=max-port, vlan-slice=z, output(a)
10	Output preprocessed message over port without link	# of virtual ports * # of slices	virtual-port	vlan-switch=x, vlan-port=y, vlan-slice=z	pop-vlan, output(a)
0	Error detection rule	1	3	*	output(controller)

Table 4-1: Full Delftvisor reserved flowtables

does not handle all of them, for most of them an error is returned. The messages needed to work with the Ryu controller has been focussed on, implementing enough to test the isolation features presented in this thesis.

Most Openflow messages are directional, meaning that they only go from controller to switch or only go from switch to controller. Delftvisor deals with both ends of the protocol, to the physical switches is it a controller and to the tenants is it a switch. This means though that each Openflow 1.3 message could potentially arrive at Delftvisor.

Delftvisor can deal with Openflow connections, so the messages Hello, EchoRequest and EchoResponse work. Delftvisor does currently not pass through Error messages, it just prints that it has received an Error message from either side.

There are several message types that allow the controller to get information about a switch. Delftvisor requests the information from the switch after the connection has established. It sends the FeatureRequest message to the physical switches and saves the FeatureResponse it gets. It does the same with MultipartRequest::MeterFeatures, MultipartRequest::GroupFeatures and MultipartRequest::PortDescription. When a tenant controller sends a FeatureRequest Delftvisor responds with a FeatureResponse with the custom datapath id. The FeatureResponse also indicates that the virtual switch does not support statistics and buffers. The amount of tables the virtual switch advertise to the tenant controller is the minimum amount any of the depended on physical switches have minus 2 for the Delftvisor reserved tables. The MultipartRequest::MeterFeatures, MultipartRequest::GroupFeatures and MultipartRequest::PortDescription from the tenant controller are handled by using the saved data from the physical switches to create the rewritten MultipartResponse messages.

Delftvisor does not support requesting statistics for tenants. This feature could be implemented in the current design but wasn't because of time constraints. This thesis focusses on the isolation between slices and statistics are not important to show that isolation works.

Delftvisor support the FlowMod and GroupMod message types. The messages are cloned for each physical switch the virtual switch depends on, rewritten for that physical switch and sent to them. Delftvisor also support PacketIn, from physical switches, and PacketOut, from tenant controllers. They are rewritten and sent on, the PacketOut message type is currently sent to any of the physical switches the virtual switch depends on. The action list in the PacketOut is rewritten for that physical switch and the message is sent.

Delftvisor only handles Hello, EchoRequest, EchoResponse, FeatureRequest, PacketOut, FlowMod, GroupMod, MultipartRequest::PortDescription and MultipartRequest::GroupFeatures message types from the tenant controller meaningfully. Some network operating systems, such as ONOS (Open Network Operating System), require more messages to work. ONOS for example requires among others the SetConfig message type to be handled and will currently not work with Delftvisor.

4-4 Internal data representation

Delftvisor loads its configuration from a json file, the internal representation looks a bit different. An incomplete class diagram can be seen in Figure 4-1. The configuration format is visible in how the `Hypervisor` class contains slices, which in turn contains virtual switches.

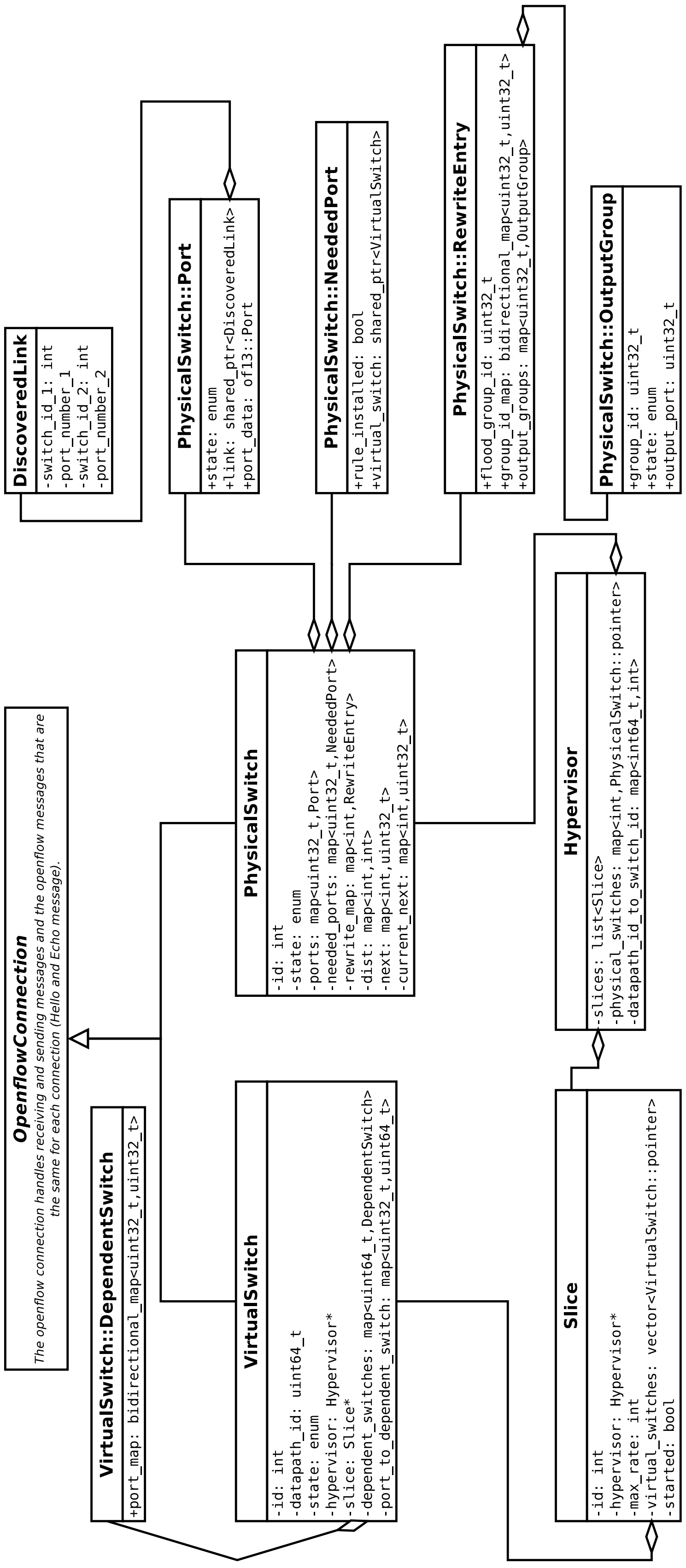

DiscoveredLink
-switch_id_1: int
-port_number_1
-switch_id_2: int
-port_number_2
PhysicalSwitch::Port
+state: enum
+link: shared_ptr<DiscoveredLink>
+port_data: of13::Port
PhysicalSwitch::NeededPort
+rule_installed: bool
+virtual_switch: shared_ptr<VirtualSwitch>
PhysicalSwitch::RewriteEntry
+flood_group_id: uint32_t
+group_id_map: bidirectional_map<uint32_t,uint32_t>
+output_groups: map<uint32_t,OutputGroup>
PhysicalSwitch::OutputGroup
+group_id: uint32_t
+state: enum
+output_port: uint32_t
OpenflowConnection
The openflow connection handles receiving and sending messages and the openflow messages that are the same for each connection (Hello and Echo message).
PhysicalSwitch
-id: int
-state: enum
-ports: map<uint32_t,Port>
-needed_ports: map<uint32_t,NeededPort>
-rewrite_map: map<int,RewriteEntry>
-dist: map<int,int>
-next: map<int,uint32_t>
-current_next: map<int,uint32_t>
Hypervisor
-slices: list<Slice>
-physical_switches: map<int,PhysicalSwitch::pointer>
-datapath_id_to_switch_id: map<int64_t,int>
VirtualSwitch::DependentSwitch
+port_map: bidirectional_map<uint32_t,uint32_t>
VirtualSwitch
-id: int
-datapath_id: uint64_t
-state: enum
-hypervisor: Hypervisor*
-slice: Slice*
-dependent_switches: map<uint64_t,DependentSwitch>
-port_to_dependent_switch: map<uint32_t,uint64_t>
Slice
-id: int
-hypervisor: Hypervisor*
-max_rate: int
-virtual_switches: vector<VirtualSwitch::pointer>
-started: bool

Figure 4-1: An incomplete class diagram of the Hypervisor that shows the data structures and their dependencies.

The `OpenflowConnection` class deals with parsing and handling packets from a socket. The `VirtualSwitch` and `PhysicalSwitch` classes derive from `OpenflowConnection` and each class handles a socket. The `VirtualSwitch` instances are created from configuration, the `Hypervisor` class contains a socket listening on the port that physical switches connect to and creates a `PhysicalSwitch` instance each time a connection is initiated.

During the Openflow handshake it is not yet clear what the datapath id of the switch is and what ports are on the switch. The `PhysicalSwitch` has to send a FeatureRequest message before receiving that information. When the FeatureResponse message arrives at the `PhysicalSwitch` instance it registers back at the `Hypervisor` instance with its datapath id. A `PhysicalSwitch` can therefore be in two states, unregistered and registered.

A `VirtualSwitch` instance can be in three states:

- Down if the physical switches have not connected to Delftvisor yet.
- Trying to connect when the physical switches have registered at Delftvisor but the tenant controller has not responded yet.
- Connected when the tenant controller Openflow connection has succeeded.

Each `PhysicalSwitch` instance periodically sends out topology discovery messages. When a link gets discovered is a `DiscoveredLink` instance create between the two `PhysicalSwitch::Port` instances. Delftvisor only supports bidirectional connections.

The `PhysicalSwitch::RewiteEntry` struct contains the information needed to rewrite messages for one particular virtual switch. The `PhysicalSwitch::OutputGroup` struct contains how one virtual port of a virtual switch is rewritten.

Floyd-Warshall is run by the `Hypervisor` instance, the result is saved in each `PhysicalSwitch` instance in the dist and next maps. Afterward every `VirtualSwitch` instance checks if it can go online. If it can it registers at each depended on `PhysicalSwitch` instance, which stores it in the `PhysicalSwitch::NeededPort` struct. The `PhysicalSwitch` instance stores it in that struct instead of the `PhysicalSwitch::Port` struct, because it is possible that the physical switch has not sent its port information to the `PhysicalSwitch` instance. This means that a virtual switch can start before Delftvisor knows about any of the ports on the virtual switch. This is not a problem, the tenant controller is later updated with PortStatus messages.

If an action has happened that could allow the Delftvisor reserved tables to change or the virtual port groups to change is the function `PhysicalSwitch::update_dynamic_rules()` called. This function looks at the state of all the rules installed in the physical switches. If it detects that a rule should be different from what it currently is does it update/install the rule.

Since the internal structure is so dependent on each other was multi-threading quickly abandoned after development started. Installing mutexes and other locks around this structure was not relevant to demonstrate what Delftvisor needed to demonstrate.

4-5 Technologies used

Delftvisor is implemented in about 6000 lines of C++ and is built with CMake. C++ was used because it is the language that the author is most familiar with.

Delftvisor uses boost asio[1] [1] to program reactively and work with sockets. Boost asio natively supports timers and sockets.

Delftvisor uses libfluid message [2] to parse the Openflow packets and create new Openflow messages. Libfluid message had some bugs that were fixed to get Delftvisor to work, these fixes were propagated back to libfluid.

[1]asio stands for Asynchronous Input/Output

Chapter 5

Experimental Evaluation

This chapter covers the experiments done to evaluate the design after implementation of Delftvisor. The manners in which Delftvisor isolates features in slices are discussed in the previous chapter, this chapter measures some of the properties of the implementation.

5-1 Controllers used

Every experiment needs a controller to perform any kind of routing. This section discusses the three routing controllers used for evaluation. All controllers are written for Ryu [4], a python framework for writing openflow controllers.

5-1-1 Ryu simple switch

This controller comes with the Ryu framework, every new packet is send to the controller which decides what to do with it. The controller saves the source mac and the port the packet came from. If the destination mac is saved in the controller, install a rule forwarding packets with this destination mac to the saved port. The controller then sends a PacketOut message with the instruction to output it on a specific port or with instruction to flood the packet in case it doesn't know how to output it. This controller can be found, at the time of writing, at `https://github.com/osrg/ryu/blob/master/ryu/app/simple_switch_13.py`.

This controller never learns the topology, can't deal with topology changes and routing loops. It uses the FLOOD port instead of the ALL port, so if the Openflow switches run Spanning Tree Protocol (STP) it could work with routing loops.

5-1-2 Flood router

This controller pushes one rule to all switches that register at it, flood all traffic over all ports. This works as long as there is no loop in the topology.

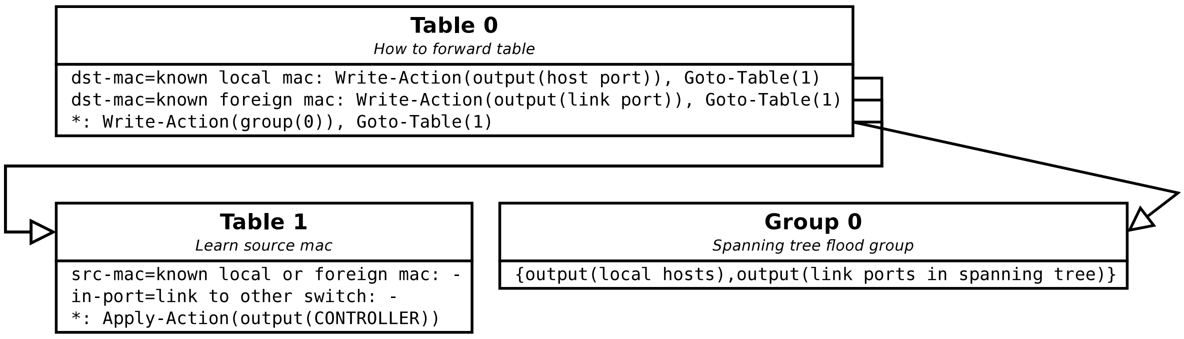


Table 0
How to forward table
dst-mac=known local mac: Write-Action(output(host port)), Goto-Table(1)
dst-mac=known foreign mac: Write-Action(output(link port)), Goto-Table(1)
*: Write-Action(group(0)), Goto-Table(1)
Table 1
Learn source mac
src-mac=known local or foreign mac: -
in-port=link to other switch: -
*: Apply-Action(output(CONTROLLER))
Group 0
Spanning tree flood group
{output(local hosts),output(link ports in spanning tree)}

Figure 5-1: The flowtable of the L2 router. Table 0 looks at the destination mac address and puts in the action set how to forward this packet. Table 1 looks at the source mac address to see if this is a mac address the controller has learned already. Group 0 floods the packet along a spanning tree.

It runs with the Ryu topology discovery mechanism activated so one extra rule is pushed to forward topology discovery packets back to the controller. This was added to help with debugging.

5-1-3 Layer 2 router

This version of the layer 2 router looks a lot like the version discussed in subsection 2-1-2. The layer 2 router learns the topology of the network and creates a spanning tree to flood packets over. It also learns via the source mac addresses of packets on which port in the network a device is connected. From then on it will route packets going to that mac address directly to it without flooding the packet along the minimum spanning tree.

This controller uses an ALL group to flood packets over the minimum spanning tree, it also uses 2 tables; a table to determine how to route the packet and a table to determine if this is a new packet the controller still needs to learn about. Since it uses the group table and multiple flow tables does this controller represent the kind of applications that can be isolated with Delftvisor.

The flowtables can be seen in Figure 5-1. This router can be found, at the time of writing, at `https://github.com/harmjan/l2-router`.

5-2 Isolation testing

Verifying that there is no manner in which a controller can influence another slice is difficult. Delftvisor's reserved flowtables have been manually inspected a few times to see if a packet could be crafted by a host that would escape isolation or a flowrule could be installed to influence another slice. An attack that a host can still perform is crafting topology discovery packets that Delftvisor recognizes in an attempt to destabilize the network. Creating a robust topology discovery mechanism was not the focus of this thesis and therefore only a functioning version has been implemented. If a host crafts a topology discovery packet Delftvisor will immediately accept that there is a link and change the host port to a shared link port, and also change the status of the port that the crafted message pretends it is from. Methods to prevent this attack have been discussed in literature [6] but have not been implemented in Delftvisor since that was not the focus of this thesis.

To demonstrate that Delftvisor can isolate different slices from each other were some experiments designed. All experiments were run in mininet and all possible combination of the controllers in section 5-1 were used as tenant controllers, except if one of the virtual networks contains a loop in which case only the L2 router was used. To verify that the isolation works was the topology percieved by the controller verified with the expected virtual network and via the mininet pingall command was reachability verified. Pingall tries to ping both ways between all host combinations in the network, all hosts in the same virtual network are expected to reach each other while hosts in different networks are expected to not be able to reach each other. The physical and virtual topologies can be seen in Figure 5-2. The output of some of the experiments in Figure 5-2 are available in Appendix A.

The topology in Figure 5-2a tests isolation within a single physical switch. 4 hosts are connected to 1 switch, Delftvisor needs to slice this switch into two virtual networks.

The topology in Figure 5-2b tests if Delftvisor can create a virtual switch consisting of multiple physical switches. In the physical topology 2 switches are available each with a host and a link between them. For the tenant controller there is only 1 virtual switch where both hosts are connected to. In this test Delftvisor discovers the topology and calculates that the virtual switch with dpid 100 can go online and attaches it to the tenant controller. Furthermore does Delftvisor route traffic between the switches when it needs a packet needs to be output on the other switch.

Topology Figure 5-2c exhibits most of Delftvisors capabilities. The physical network consists 4 switches connected in a line, with 2 hosts each. The first slice contains 1 virtual switch with 4 hosts connected to it, each of these hosts is connected to a different physical switch. This virtual switch goes online when Delftvisor figures out that all physical switches are mutually reachable. The second slice contains 2 virtual switches, both with two hosts connected to them. The virtual switch with dpid 200 consists on the two physical switches, the physical switch with dpid 1 and 2 connected to a host on both of them. Idem ditto with the virtual switch with dpid 201, but than on the physical switches with dpid 3 and 4. The link between the virtual switches maps to the physical link between dpid 2 and 3. If in mininet the link between dpid 2 and 3 is disconnected does the virtual switch with dpid 100 go down, while the virtual switch with dpid 200 and 201 stay up. Not all hosts in that slice can at that point reach each other anymore, but the hosts connected to the same virtual switch can.

The topology in Figure 5-2d is named the clique-4 topology, it consists of 4 physical switches that are completely connected meaning that there is a link between each pair of physical switches. Each physical switch furthermore has 2 hosts connected to it. The first slice consists of 1 slice that consists of 1 virtual switch that again is connected to a host on each physical switch. This tests how Delftvisor can hide topology details. Disconnecting links in the network causes Delftvisor to reroute the traffic after it has detected that change. The second slice passes the physical topology directly to the tenant controller, except it only exposes one of the hosts per physical switch. When a link in the network goes down does the tenant controller need to detect this and react to it.

The last of the experimental evaluation topologies is the topology in Figure 5-2e. The physical topology consists of 2 physical switches that have 3 links between them and are also connected to 2 hosts each. There are 2 slices that both are comparable, they both have a comparable topology. Each slice consists of 4 virtual switches in a line, each of the virtual links correspond to a physical link between the two physical switches. This experiment should demonstrate that packets on the link are identified by their slice and packets in the switch are identified by the virtual switch they are in. Each physical switch has 4 virtual switches implemented on it, 2 from each slice. Each link is implemented in 2 slices and Delftvisor is able to isolate these slices from each other.

5-3 Control channel latency

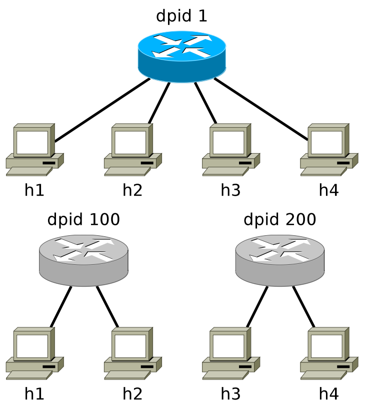
Delftvisor sits in the control channel path, between the controller and the switch. This means that Delftvisor introduces some latency to the packets from the controller to the switch. This latency has been measured in this section.

To measure this delay a controller was developed that sends a PacketOut message down to a connected switch. The action list of the PacketOut message contains an output action to send it back to the controller again. The time between sending and receiving is measured and recorded by the controller. The code is included in section B-1. This experiment was done with a separate computer running OpenVSwitch and a laptop running Delftvisor and the experiment controller. There was only one physical switch in this example.

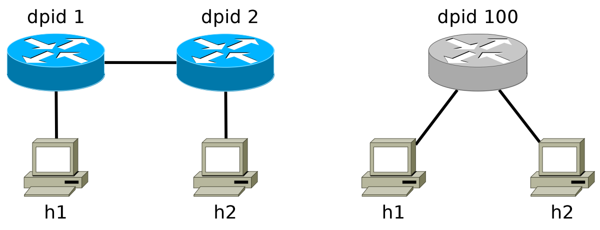
The results can be seen in Figure 5-3 and with spikes above 0.03 seconds filtered in Figure 5-4. Delftvisor clearly adds some delay and periodically has spikes in these delays, during the measurements this only happened 13 times in 1000 samples. Without the spikes are there still two visible classes of delay, two recognizable different groups of samples. The total average delay is still very manageable and should not hamper any experiments.

As discussed in does Delftvisor run in a single thread. This means that Delftvisor can only deal with 1 packet at the time, if multiple packets from multiple connections arrive at Delftvisor at the same time do packets have to wait in a buffer until Delftvisor gets around to handling them. Running in a single thread made the implementation of Delftvisor easier since no data structures need to be locked to prevent race conditions. Delftvisor itself has some periodic tasks, sending Echo messages over Openflow connections and sending topology discovery packets. If a PacketIn from the controller or a PacketOut from the physical switch arrive at Delftvisor while it is executing any of these tasks would the packet need to wait till the tasks are completed.

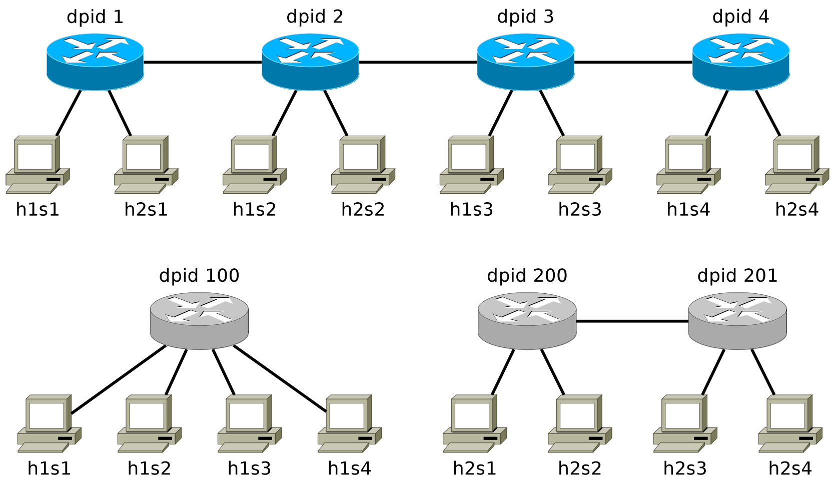
The author has not been able to definitively explain why the spikes are there and why there are two classes of delay. The experiments were run with the controller and Delftvisor on the same computer which means they could interfere with each other. Another explanation could be that the operating system of the controller/Delftvisor computer or of the OpenVSwitch computer needed to perform tasks that delayed handling the packet. Overall is the delay fine for a proof of concept implementation, the focus of the thesis was on isolating traffic into slices and not on guaranteeing a minimal latency.



(a) 1 switch shared between 2 virtual switches

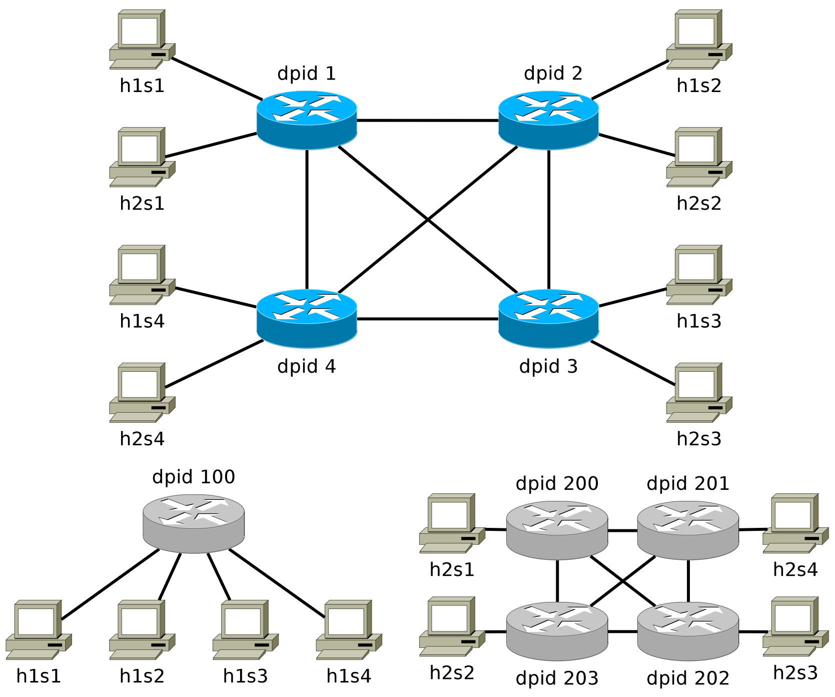


(b) 2 switches shared between 1 virtual switch

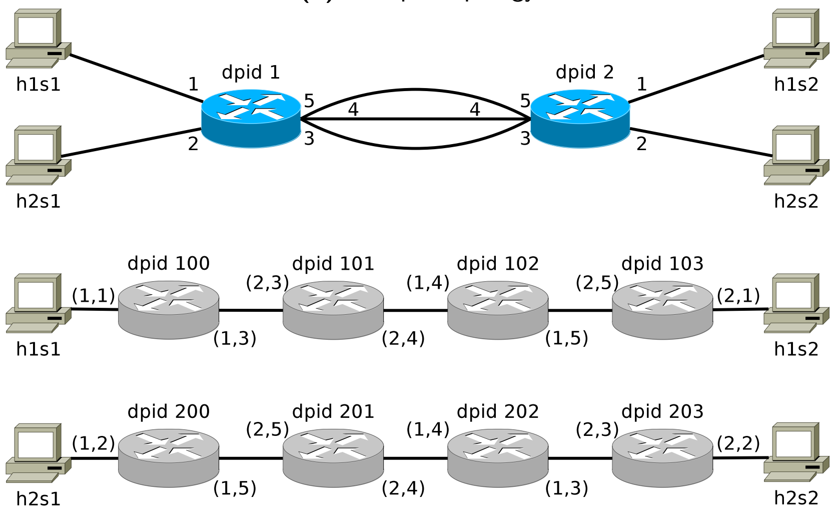


(c) 4 linear switches, the link between the virtual switches dpid 200 and dpid 201 is the physical link between the physical switches dpid 2 and dpid 3.

Figure 5-2: The first 3 test topologies used to test slice isolation



(d) 4 clique topology



(e) 2 switches turned into 8 virtual switches in 2 slices, virtual switches in the same slice can be implemented on the same physical switch. To clarify how this topology works are the virtual ports marked using (physical dpid,port_no).

Figure 5-2: The second 2 topologies used to test slice isolation, continued from the previous page.

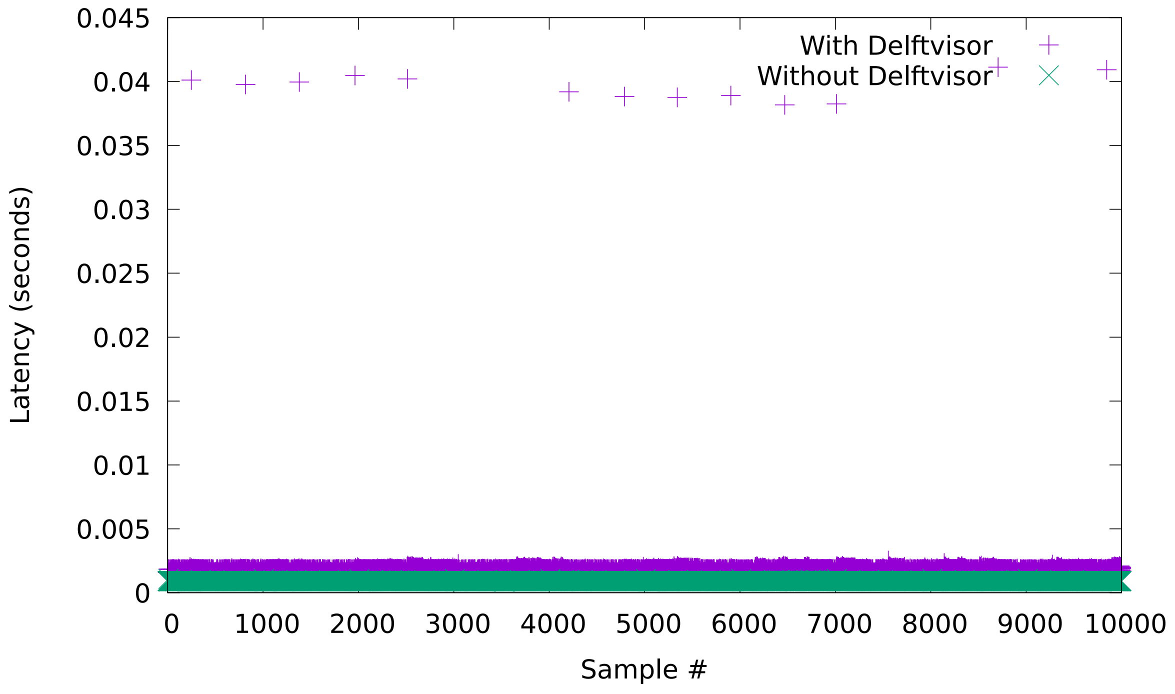


Figure 5-3: The delay of a round-trip from the controller to the switch and back, with and without Delftvisor in the path. Delftvisor on average has a bit more delay and some spikes.

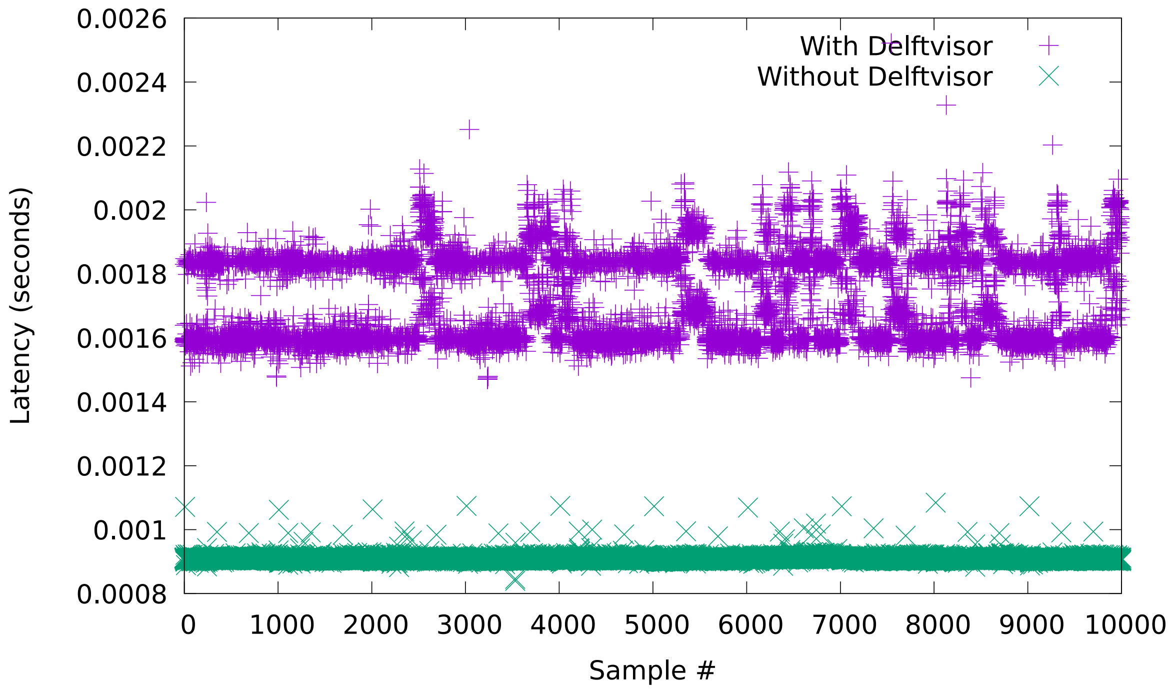


Figure 5-4: The delay of a round-trip from the controller to the switch and back, with and without Delftvisor in the path with the spikes from Figure 5-3 removed.

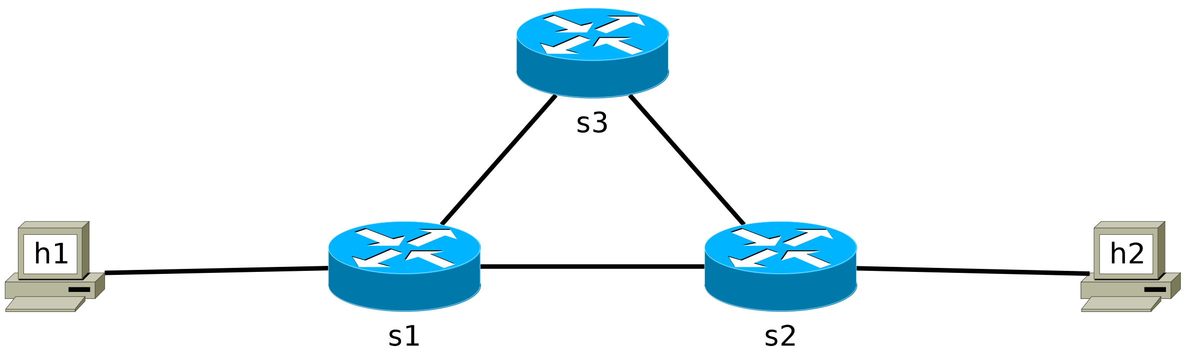


Figure 5-5: The topology used in the recovery experiments. Host h1 pings host h2, during which the link between switch s1 and switch s2 is severed. The controller needs to detect this and re-route the traffic via switch s3. The time it takes until traffic can make its way between host h1 and h2 is measured.

5-4 Recovery latency

Delftvisor can recover from failure in the network and re-route the traffic if possible. It takes some time to detect the problem, find a new route and update the rules in the switches. This section discusses an experiment that quantifies this latency and compares it to a normal controller.

The topology in Figure 5-5 is used in this experiment. Host h1 and h2 can reach each other via the link between switch s1 and s2, when that link goes down the network needs to reconfigure to send the traffic via switch s3. The time between the last packet arriving back at host h1 before the link failure and the earliest packet arriving at h1 after recovery is called the recovery latency.

3 different setups are compared:

- The network is directly attached to the L2 router discussed in subsection 5-1-3.
- The network is attached to Delftvisor which exposes the entire network to the L2 router.
- The network is attached to Delftvisor which exposes a single switch with host h1 and h2 attached to the flood router discussed in subsection 5-1-2.

Link failure is detected via the topology discovery mechanisms discussed in subsection 2-1-4. The last setup is done twice with different topology discovery timeouts.

All experiments in this section are run in mininet [11], the scripts used are available in section B-2. The script starts the network with the topology from Figure 5-5 and starts pinging host h2 from host h1 100 times per second. Another script records the time between ping responses coming back. After 2 seconds it takes down the link between switch s1 and s2, waits 6 seconds for the network to detect the fault and correct the routing. This process is repeated a 1000 times for each setup. The first 5 recovery experiments are plotted in Figure 5-6 to show what the results of an experiment looks like.

Another script identifies the peeks seen in Figure 5-6 and this data is shown in Figure 5-7. Delftvisor is clearly a lot slower than the normal controllers, this is mainly because of the

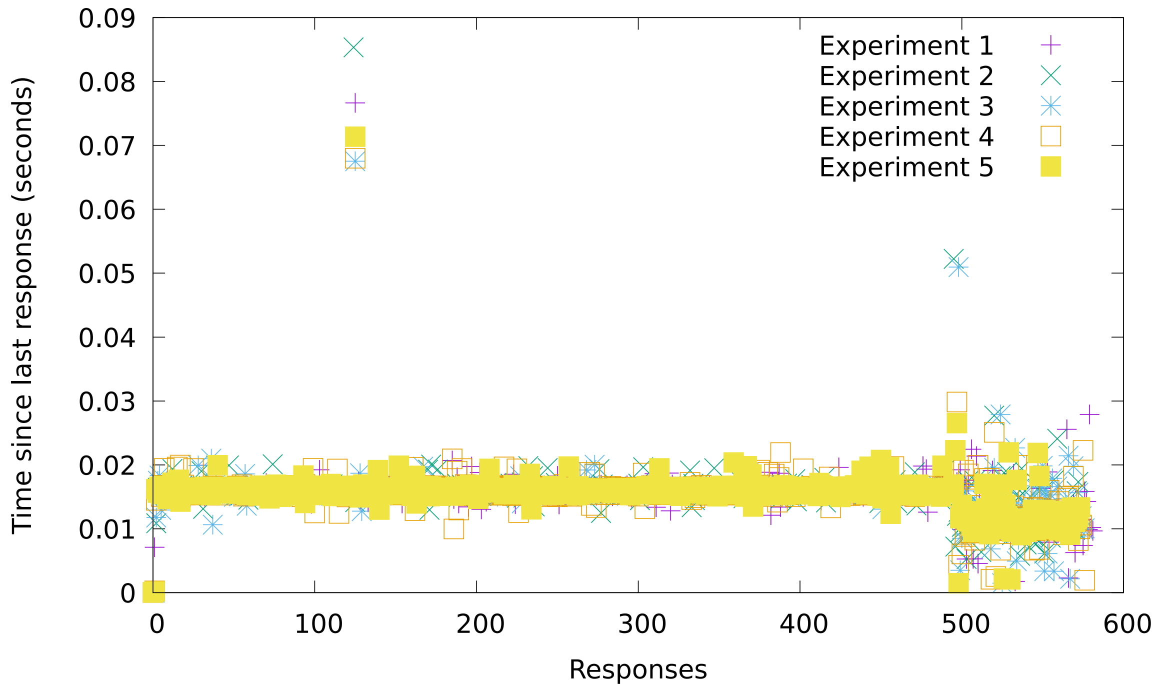


Figure 5-6: All the data from the first 5 recovery experiments plotted. The recovery delay are the spikes in the first 200 samples. The disturbance at the end comes from the link being restored and the network reconfiguring.

link discovery mechanism as the last experiment shows. Delftvisor's system has not been optimized for detection of link failure, picking a frequency to send out LLDP packets is a trade-off between fast failure recognition and control channel utilization. Since this wasn't the focus of this thesis has this part of Deltvisor not been optimized.

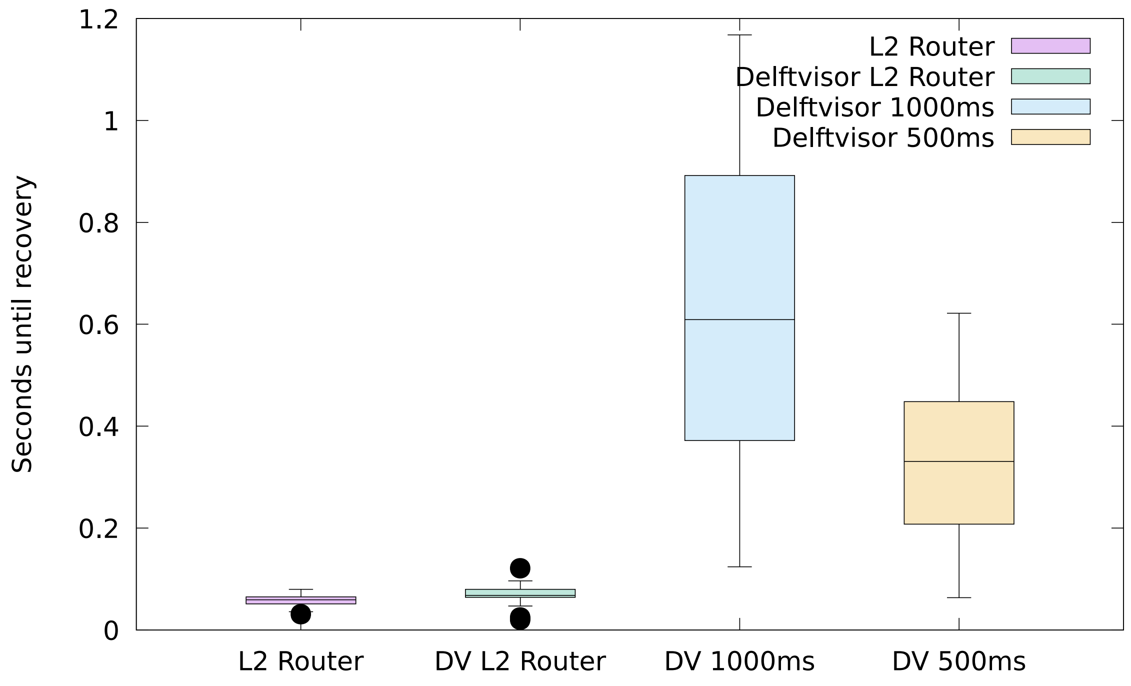


Figure 5-7: A boxplot of the 1000 experiments. What can be seen is that the L2 router can correct quicker than Delftvisor, even in the case where Delftvisor is between the L2 Router and the switch. This is because the topology discovery mechanism for Ryu is better optimized than the Delftvisor's topology discovery. In the second Delftvisor experiment the recovery is greatly improved just by decreasing the topology discovery timeout and sendout.

Chapter 6

Conclusion

This thesis main contribution are methods to embed virtual networks in physical Openflow 1.3 networks. Delftvisor is a proof of concept implementation of these ideas that allows isolating network resources. It allows a physical Openflow 1.3 network to be sliced into different virtual networks, which each consist of virtual switches which in turn consist of virtual ports. Virtual ports map to a specific physical port in the network, although not all virtual ports of a virtual switch need to be implemented on the same physical switch.

Delftvisor is a program that sits on the Openflow connection between the tenant controllers and the physical switches. It accepts physical switch connections and discovers the topology between these physical switches. If a virtual switch can go online because all physical switches it depends on are online and mutually reachable will the connection to the tenant controller be established. From that moment can a tenant controller push Openflow messages to Delftvisor which will emulate their intent on the physical switches while maintaining isolation between slices.

Delftvisor isolates packets on the wire by adding a VLAN tag to each packet inside of the Delftvisor managed network that contains the slice id. Depending on the slice id in the tag can Delftvisor decide at ingress into a physical switch what virtual switch this packet belongs to. It pops the VLAN tag off the packet and forwards it to the tenant tables with the virtual switch id added in the metadata field. In Delftvisor are packets on the wire identified by the slice id and in the physical switch by virtual switch id.

The tenant tables are populated by the flow entries pushed by the tenant controllers. Each match field is expanded to match on the virtual switch id it belongs to prevent matching on packets from other virtual switches.

Groups and Meters in Openflow are identified by numbers, Delftvisor allocates numbers for these as they come along and saves the mapping.

Delftvisor currently introduces some duplication by duplicating the entire virtual switch flowtable on each physical switch it depends on. It also pushes two flow entries for each virtual flow entry to deal with the FLOOD group problem discussed in subsection 3-6-1.

Delftvisor does not save the virtual flowtable but just rewrites and passes on all Openflow messages. This makes it impossible to migrate the network. To support migrating the network or moving virtual ports to different physical ports Delftvisor needs to be able to push all flow entries that are already in the virtual switch to a new switch. Delftvisor can recover and rewrite packets that are internally routed because all packets are output to INDIRECT groups managed by Delftvisor instead of the tenant controllers. Delftvisor does not support virtual links because it cannot update all flow entries that match on in-port for that specific virtual link.

Delftvisor is not production quality software but does show that the ideas presented in chapter 3 work and can slice a physical network into virtual networks. Delftvisor, as presented in this thesis, is a jumping off point to a network hypervisor that can completely virtualize Openflow 1.3 networks.

6-1 Future work

A common problem encountered during the design of Delftvisor was that useful Openflow 1.3 features were optional. Features such as the metadata field or multiple flowtables that are required for the functioning of Delftvisor. This makes it hard to write generic Openflow 1.3 applications, the author thinks that forcing switch vendors to implement more useful features would greatly help the deployment of Openflow 1.3.

Deltvisor currently can recover from link failure if another route is possible. It still needs to detect this via the topology discovery mechanism which imposes enormous delays on the recovery latency. Delftvisor only routes internal traffic, the packets that get lost because of link failure look to the tenant like they got lost somewhere internal in the virtual switch which makes this problem worse. Openflow 1.3 has a group type called FAST-FAILOVER which allows a switch to monitor connectivity of a port via a mechanism as Bidirectional Forwarding Detection (BFD), and when a link goes down have a bucket with different actions that take over routing. Mechanisms to pre-compute backup paths have been researched before for Openflow 1.3 [18]. These mechanisms could be implemented in Delftvisor to improve its recovery latency.

Routing in Delftvisor is currently done by Delftvisor itself based on the least amount of hops to get to another switch. The network administrator of the physical network that Delftvisor manages probably want to have some control over how Delftvisor does this. Instead of extending the configuration format or defining some kind of custom API to allow access to this functionality does the author think that exposing this functionality via a subset of Openflow could work well. For this idea would Delftvisor reserve a special management slice consisting of all ports in the network attached to an internal link. The controller could therefore via normal topology discovery mechanisms figure out the topology of the physical network. The controller can push routing rules by pushing FlowMod messages that would be passed onto the second Delftvisor reserved table. Delftvisor also needs to figure out how to route the packets from the INDIRECT port groups, to do that it could look at the FlowMod messages pushed and interpret what slice/destination packets would match that rule and install the actions in the FlowMod in the INDIRECT group bucket. If an internal route was needed by Delftvisor it could generate a PacketIn to the internal routing controller with a fake packet that contains the information about the route needed.

In the system described above would a network administrator be able to configure traffic from certain slices to use a preferential queue, or add extra metering to certain slices. The interface would not be completely Openflow 1.3 compliant but the advantage would be that a network administrator could configure the network using a technology they are already familiar with.

Delftvisor currently uses VLAN tags which define most of the limitations on amount of slices,physical switches and ports. The current limitations are so confining that Delftvisor cannot be used in most production networks. Provider Backbone Bridges (PBB) tags are exposed via Openflow 1.3, using these tags could the ideas presented in this thesis be implemented for production level requirements.

A network hypervisor is a single point of failure in the system. The problem of how many hypervisor instance are needed and where to place them has been studied before [9]. For Delftvisor to be usable it needs to be able to sync state and allow multiple instances to manage part of the network and communicate with the other instances at the edge of the network.

Appendix A

Experimental output

This appendix contains all input and output for three experiments to show the flows generated by Delftvisor. In this appendix can the internal workings of Delftvisor be inspected without having to compile and install Delftvisor.

The first experiment in section A-1 shows the rules generated in one single switch when it is shared by two tenants without any shared links. This experiment shows how the features of a single switch are split up between two virtual switches without interfering with each other. The second experiment in section A-2 is about two switches that get abstracted into one virtual switch. This shows how traffic meant for a virtual port not on the current switch is routed by Delftvisor. The third experiment in section A-3 shows more complexly how topology can be abstracted. Each physical switch is part of multiple virtual switches and there are links in the physical network that are used to transport Delftvisor internal traffic and are used by a slice in the topology.

For each of these experiments is the input configuration and output given. The mininet output is briefly discussed per output line, explaining why each rule and group was created. Delftvisor also creates a debug file giving insight into its internal state for each connected physical switch, these files are also shown.

A-1 Single switch

This first experiment shows how one single switch with four hosts connected to it can be shared by two tenant. The topology can be seen in Figure A-1, both virtual switch are connected to the flood router discussed in subsection 5-1-2. This experiment was run in mininet started with the following command:

```
mn --controller remote,ip=192.168.56.1 --switch ovsk,protocols=OpenFlow13
    --mac --topo single,4
```

This tells mininet to start 1 switch with 4 hosts connected to it. Those hosts have ascending mac addresses, the first host has mac address 00:00:00:00:00:01, the second host has mac

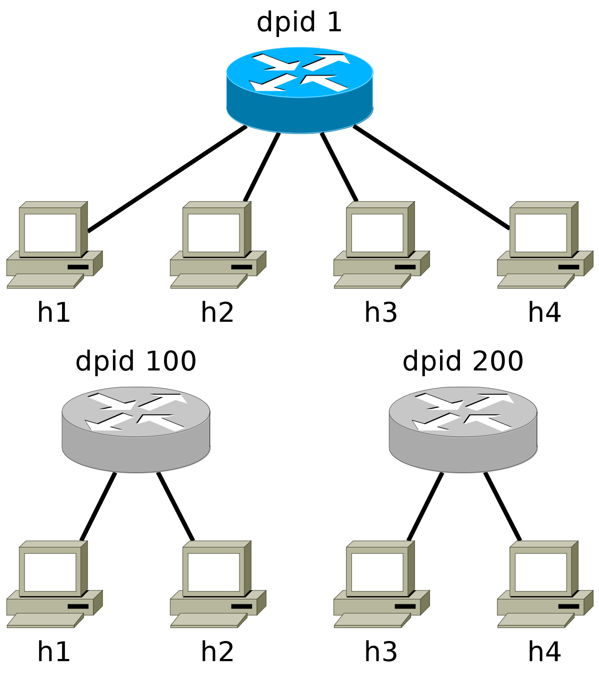


Figure A-1: A single physical switch with dpid 1 that gets virtualised into two virtual switches with dpid 100 and 101.

address 00:00:00:00:00:02, etc. The switch is configured to speak Openflow 1.3 and connect to a controller at 192.168.56.1.

The configuration file passed to Delftvisor is visible in Listing A.1, the output from mininet showing that the hosts not connected to the same virtual switch cannot ping each other and how the rules/groups in the physical switch look is shown in Listing A.2. Every physical switch in Delftvisor writes a debug output file when things change, it shows most of the internal structure. For the single switch in this example that file shown in Listing A.3.

Line 19 until 22 from Listing A.2 shows the output from the mininet pingall command which shows that reachability is as configured.

Line 27 to 45 show the rules installed by Delftvisor in the reserved tables. Line 27 is the topology discovery rule, forwarding packets in the topology discovery slice to the controller. Line 28 to 33 only match on in-port and based on the in-port drop or forward to tenant tables. Since there are no shared links in the topology are those rules not there. Line 34 is the error detection rule of the first table. Line 35 to 45 contains rules proactively pushed by Delftvisor to handle traffic arrived over a link, since there are no shared links in the topology do these rules not matter. None of these rules have been triggered.

Line 46 to 53 contain the tenant rules. Both virtual switches that depend on this physical switch use the same controller, which pushes two rules to a switch. One to do topology discovery and the other matches everything and floods the packet. These rules are installed for each controller and because of the output rewrite problem discussed in section 3-3-3 are they duplicated again. That is why there are 8 tenant rules installed.

Line 57 to 63 contain all the groups installed in this physical switch. There are 7 groups used, their meaning can be found via the internal structure shown in Listing A.3 or via inspection.

Group 0 is always the CONTROLLER port group. Group 1 is the FLOOD/ALL port for virtual switch 1, group 2 and 3 are virtual port for physical port 1 and 2 respectively. Group 4 is the FLOOD/ALL port for virtual switch 2, group 5 and 6 are virtual port for physical port 3 and 4 respectively.

Listing A.1: Configuration file

```
{
  "switch_endpoint_port" : 6633,
  "use_meters"           : false,
  "slices" : [
    {
      "controller" : {
        "ip"   : "127.0.0.1",
        "port" : 6653
      },
      "max_rate"         : 500,
      "virtual_switches" : [
        {
          "datapath_id" : 100,
          "ports" : [
            {
              "virtual_port"          : 10,
              "physical_datapath_id" : 1,
              "physical_port"        : 1
            },
            {
              "virtual_port"          : 11,
              "physical_datapath_id" : 1,
              "physical_port"        : 2
            }
          ]
        }
      ]
    },
    {
      "controller" : {
        "ip"   : "127.0.0.1",
        "port" : 6654
      },
      "max_rate"         : 500,
      "virtual_switches" : [
        {
          "datapath_id" : 200,
          "ports" : [
            {
              "virtual_port"          : 20,
              "physical_datapath_id" : 1,
              "physical_port"        : 3
            },
            {
              "virtual_port"          : 21,
              "physical_datapath_id" : 1,
                "physical_port"        : 4
              }
            ]
          }
        ]
      }
    ]
}
```

Listing A.2: Mininet output

```
*** Creating network
*** Adding controller
Unable to contact the remote controller at 192.168.56.1:6633
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1) (h4, s1)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 X X
h2 -> h1 X X
h3 -> X X h4
h4 -> X X h3
*** Results: 66% dropped (4/12 received)
mininet> dpctl dump-flows -O OpenFlow13
*** s1
   ------------------------------------------------------------------------

OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x1, duration=45.435s, table=0, n_packets=0, n_bytes=0, priority
     =20,vlan_tci=0xf800/0xf800 actions=write_actions(CONTROLLER:65535)
 cookie=0xfffffffd, duration=45.435s, table=0, n_packets=0, n_bytes=0,
     priority=10,in_port=CONTROLLER actions=goto_table:1
 cookie=0xfffffffe, duration=45.389s, table=0, n_packets=0, n_bytes=0,
     priority=10,in_port=LOCAL actions=drop
 cookie=0x1, duration=45.389s, table=0, n_packets=10, n_bytes=532,
     priority=10,in_port=1 actions=write_metadata:0x2/0x3fff,goto_table:2
 cookie=0x2, duration=45.389s, table=0, n_packets=10, n_bytes=532,
     priority=10,in_port=2 actions=write_metadata:0x2/0x3fff,goto_table:2
 cookie=0x3, duration=45.434s, table=0, n_packets=10, n_bytes=532,
     priority=10,in_port=3 actions=write_metadata:0x4/0x3fff,goto_table:2
 cookie=0x4, duration=45.389s, table=0, n_packets=10, n_bytes=532,
     priority=10,in_port=4 actions=write_metadata:0x4/0x3fff,goto_table:2
 cookie=0x2, duration=45.435s, table=0, n_packets=0, n_bytes=0, priority=0 actions=write_actions(CONTROLLER:65535)
 cookie=0xfffffffe, duration=45.389s, table=1, n_packets=0, n_bytes=0, priority=10,dl_vlan=3840,dl_vlan_pcp=0 actions=write_actions(LOCAL)
 cookie=0xfffffffe, duration=45.389s, table=1, n_packets=0, n_bytes=0, priority=10,dl_vlan=1792,dl_vlan_pcp=1 actions=write_actions(LOCAL)
 cookie=0x1, duration=45.389s, table=1, n_packets=0, n_bytes=0, priority=10,dl_vlan=2176,dl_vlan_pcp=0 actions=write_actions(pop_vlan,output:1)
 cookie=0x1, duration=45.389s, table=1, n_packets=0, n_bytes=0, priority=10,dl_vlan=128,dl_vlan_pcp=1 actions=write_actions(pop_vlan,output:1)
 cookie=0x2, duration=45.389s, table=1, n_packets=0, n_bytes=0, priority=10,dl_vlan=2304,dl_vlan_pcp=0 actions=write_actions(pop_vlan,output:2)
 cookie=0x2, duration=45.389s, table=1, n_packets=0, n_bytes=0, priority=10,dl_vlan=256,dl_vlan_pcp=1 actions=write_actions(pop_vlan,output:2)
 cookie=0x3, duration=45.389s, table=1, n_packets=0, n_bytes=0, priority=10,dl_vlan=2432,dl_vlan_pcp=0 actions=write_actions(pop_vlan,output:3)
 cookie=0x3, duration=45.389s, table=1, n_packets=0, n_bytes=0, priority=10,dl_vlan=384,dl_vlan_pcp=1 actions=write_actions(pop_vlan,output:3)
 cookie=0x4, duration=45.389s, table=1, n_packets=0, n_bytes=0, priority=10,dl_vlan=2560,dl_vlan_pcp=0 actions=write_actions(pop_vlan,output:4)
 cookie=0x4, duration=45.389s, table=1, n_packets=0, n_bytes=0, priority=10,dl_vlan=512,dl_vlan_pcp=1 actions=write_actions(pop_vlan,output:4)
 cookie=0x3, duration=45.435s, table=1, n_packets=0, n_bytes=0, priority=0 actions=write_actions(CONTROLLER:65535)
 cookie=0x0, duration=34.386s, table=2, n_packets=0, n_bytes=0, priority=65535,metadata=0x2/0x3fff,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=group:0
 cookie=0x0, duration=34.386s, table=2, n_packets=0, n_bytes=0, priority=65535,metadata=0x3/0x3fff,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=group:0
 cookie=0x0, duration=32.385s, table=2, n_packets=0, n_bytes=0, priority=65535,metadata=0x4/0x3fff,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=group:0
 cookie=0x0, duration=32.385s, table=2, n_packets=0, n_bytes=0, priority=65535,metadata=0x5/0x3fff,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=group:0
 cookie=0x5, duration=34.386s, table=2, n_packets=20, n_bytes=1064, priority=0,metadata=0x2/0x3fff actions=write_actions(group:1)
 cookie=0x5, duration=34.386s, table=2, n_packets=0, n_bytes=0, priority=0,metadata=0x3/0x3fff actions=drop
 cookie=0x5, duration=32.385s, table=2, n_packets=20, n_bytes=1064, priority=0,metadata=0x4/0x3fff actions=write_actions(group:4)
 cookie=0x5, duration=32.385s, table=2, n_packets=0, n_bytes=0, priority=0,metadata=0x5/0x3fff actions=drop
mininet> dpctl dump-groups -O OpenFlow13
```

```
*** s1 ──────────────────────────────────────────────────────────────────────────

OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
 group_id=2,type=indirect,bucket=actions=output:1
 group_id=3,type=indirect,bucket=actions=output:2
 group_id=5,type=indirect,bucket=actions=output:3
 group_id=6,type=indirect,bucket=actions=output:4
 group_id=0,type=indirect,bucket=actions=CONTROLLER:65535
 group_id=1,type=all,bucket=actions=group:2,bucket=actions=group:3
 group_id=4,type=all,bucket=actions=group:5,bucket=actions=group:6
```

Listing A.3: Debug output for physical switch 1

```
[PhysicalSwitch id=0, dpid=1] = {
  ports = [
    {
      id = 3
      state = host_rule
      needed-ports = { [Virtual switch dpid=200, state=connected] }
    }
    {
      id = 4
      state = host_rule
      needed-ports = { [Virtual switch dpid=200, state=connected] }
    }
    {
      id = 1
      state = host_rule
      needed-ports = { [Virtual switch dpid=100, state=connected] }
    }
    {
      id = 4294967294
      state = drop_rule
      needed-ports = { }
    }
    {
      id = 2
      state = host_rule
      needed-ports = { [Virtual switch dpid=100, state=connected] }
    }
  ]
  rewrite-map = [
    {
      virtual-switch-id = 2
      flood-group-id = 4
      group-id-map = [
      ]
      output-groups = [
        {
          virtual-port-id = 21
          group-id = 6
          output-port = 4
          state = host_rule
        }
        {
          virtual-port-id = 20
          group-id = 5
          output-port = 3
          state = host_rule
        }
      ]
    }
    {
      virtual-switch-id = 1
      flood-group-id = 1
      group-id-map = [
      ]
      output-groups = [
        {
          virtual-port-id = 11
          group-id = 3
          output-port = 2
          state = host_rule
        }
        {
          virtual-port-id = 10
          group-id = 2
          output-port = 1
          state = host_rule
        }
      ]
    }
  ]
}
```

A-2 2 switches

This experiment covers the case where a network consist of two switches, each of which has a host attached to it and a link between the switches. The only virtual switch in this experiment connects both hosts to one switch. Delftvisor recognizes that some packets must be output on the other switch and can route these packets correctly. The topology can be seen in Figure A-2. This experiment was run in mininet started with the following command:

```
mn --controller remote,ip=192.168.56.1 --switch ovsk,protocols=OpenFlow13
   --mac --topo linear,2
```

This does almost the same as in the previous experiment, except that the topology seen in Figure A-2 is started. Both switches speak Openflow 1.3 and connect to the controller at 192.168.56.1.

The input configuration file is shown in Listing A.4, the mininet output in Listing A.5. The Delftvisor debug outputs are shown in Listing A.6 and Listing A.7 for the physical switch with dpid 1 and dpid 2 respectively.

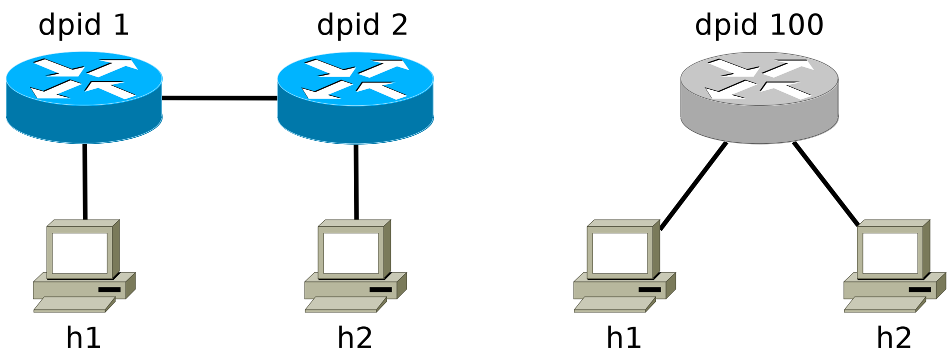


Figure A-2: Two physical switch with dpid 1 and 2 that gets virtualised into one virtual switches with dpid 100.

This mininet output in Listing A.5 will be discussed in the following paragraphs. Lines 17 to 20 of the output show the `pingall` command and that both hosts can reach each other.

Lines 25 to 41 are the rules in the physical switch with dpid 1. Line 25 is the topology discovery rule for Delftvisor. Lines 26 to 29 look at the in port of a packet and decide what to do with the packet. Line 28 covers packets that arrive over the link and forwards those packets to table 1. Line 29 detect packets send from host h1, sets the correct metadata value and forwards the packets to table 2; the tenant tables. Line 30 is the error detection rule for table 0.

Line 31 forwards messages meant for the switch with dpid 2. These rules are installed proactively, so even though this rule will never be used is it installed. Lines 32 to 34 handle outputting messages meant for this switch. Per physical port on the switch there is a rule dealing with the case that that port is a virtual port somewhere in the network. Only port 1, connected to host h1, is actually a virtual port in the network. Rule 34 covers this port and it is the only rule of this type that has actually handled packets. Line 35 is the error detection rule for table 1.

Lines 36 to 41 are tenant rules pushed by the tenant controller. The tenant controller is the Ryu simple switch discussed in subsection 5-1-1. All these rules are duplicated to deal with the output rewrite problem discussed in section 3-3-3. Lines 36 and 37 are the topology discovery rule installed by Ryu. Lines 38 and 39 are a rule that forward the packet to host h1. The controller has also pushed a rule that deals with packets to host h2, those rules are not duplicated on this switch because they match on in-port. Since that port is not available on this switch can the match fields not be rewritten and are those rules not duplicated on switch dpid 1. Lines 57 and 58 deal with those rules in switch dpid 2. Lines 40 and 41 are the catch-all rule of the controller, packets it has not learned about are caught by this rule and sent to the controller.

The rules for switch dpid 2 are very similar. The groups installed in switch dpid 1 can be seen in lines 63 to 67. The group with id 0 is the virtual port that forwards to the controller. The group with id 1 simulates the FLOOD and ALL port for virtual switch with dpid 100. Group 2 is the virtual port attached to host h2, group 3 is the virtual port attached to host h1.

Listing A.4: Configuration file

```
{
  "switch_endpoint_port" : 6633,
  "use_meters"           : false,
  "slices" : [
    {
      "controller" : {
        "ip"   : "127.0.0.1",
        "port" : 6653
      },
      "max_rate"         : 500,
      "virtual_switches" : [
        {
          "datapath_id" : 200,
          "ports" : [
            {
              "virtual_port"          : 30,
              "physical_datapath_id" : 1,
              "physical_port"         : 1
            },
            {
              "virtual_port"          : 31,
              "physical_datapath_id" : 2,
              "physical_port"         : 1
            }
          ]
        }
      ]
    }
  ]
}
```

Listing A.5: Mininet output

```
*** Creating network
*** Adding controller
Unable to contact the remote controller at 192.168.56.1:6633
*** Adding hosts:
h1 h2
*** Adding switches:
s1 s2
*** Adding links:
(h1, s1) (h2, s2) (s2, s1)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 2 switches
s1 s2 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
```

```
*** Results: 0% dropped (2/2 received)
mininet> dpctl dump-flows -O OpenFlow13
*** s1
    ----------------------------------------------------------------------------
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x1, duration=42.795s, table=0, n_packets=85, n_bytes=5780,
    priority=20,vlan_tci=0xf800/0xf800 actions=write_actions(CONTROLLER
    :65535)
 cookie=0xfffffffd, duration=42.795s, table=0, n_packets=0, n_bytes=0,
    priority=10,in_port=CONTROLLER actions=goto_table:1
 cookie=0xfffffffe, duration=42.701s, table=0, n_packets=0, n_bytes=0,
    priority=10,in_port=LOCAL actions=drop
 cookie=0x2, duration=42.745s, table=0, n_packets=5, n_bytes=453,
    priority=10,in_port=2 actions=goto_table:1
 cookie=0x1, duration=42.701s, table=0, n_packets=3, n_bytes=238,
    priority=10,in_port=1 actions=write_metadata:0x2/0x3fff,goto_table:2
 cookie=0x2, duration=42.795s, table=0, n_packets=0, n_bytes=0, priority
    =0 actions=write_actions(CONTROLLER:65535)
 cookie=0x0, duration=6.884s, table=1, n_packets=0, n_bytes=0, priority
    =20,vlan_tci=0x1001/0x107f actions=write_actions(output:2)
 cookie=0xfffffffe, duration=42.701s, table=1, n_packets=0, n_bytes=0,
    priority=10,dl_vlan=3840,dl_vlan_pcp=0 actions=write_actions(LOCAL)
 cookie=0x2, duration=42.701s, table=1, n_packets=0, n_bytes=0, priority
    =10,dl_vlan=2304,dl_vlan_pcp=0 actions=write_actions(set_field:8191->
    vlan_vid,set_field:0->vlan_pcp,output:2)
 cookie=0x1, duration=42.701s, table=1, n_packets=4, n_bytes=346,
    priority=10,dl_vlan=2176,dl_vlan_pcp=0 actions=write_actions(pop_vlan
    ,output:1)
 cookie=0x3, duration=42.795s, table=1, n_packets=1, n_bytes=107,
    priority=0 actions=write_actions(CONTROLLER:65535)
 cookie=0x0, duration=6.884s, table=2, n_packets=0, n_bytes=0, priority
    =65535,metadata=0x2/0x3fff,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc
    actions=group:0
 cookie=0x0, duration=6.884s, table=2, n_packets=0, n_bytes=0, priority
    =65535,metadata=0x3/0x3fff,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc
    actions=group:0
 cookie=0x0, duration=4.043s, table=2, n_packets=1, n_bytes=98, priority
    =1,metadata=0x2/0x3fff,in_port=1,dl_dst=00:00:00:00:00:02 actions=
    group:3
 cookie=0x0, duration=4.041s, table=2, n_packets=0, n_bytes=0, priority
    =1,metadata=0x3/0x3fff,in_port=1,dl_dst=00:00:00:00:00:02 actions=
    group:3
 cookie=0x0, duration=6.884s, table=2, n_packets=2, n_bytes=140, priority
    =0,metadata=0x2/0x3fff actions=group:0
 cookie=0x0, duration=6.884s, table=2, n_packets=0, n_bytes=0, priority
    =0,metadata=0x3/0x3fff actions=group:0
*** s2
    ----------------------------------------------------------------------------
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x1, duration=42.805s, table=0, n_packets=85, n_bytes=5780,
    priority=20,vlan_tci=0xf800/0xf800 actions=write_actions(CONTROLLER
        :65535)
 cookie=0xfffffffd, duration=42.805s, table=0, n_packets=0, n_bytes=0,
        priority=10,in_port=CONTROLLER actions=goto_table:1
 cookie=0xfffffffe, duration=42.713s, table=0, n_packets=0, n_bytes=0,
        priority=10,in_port=LOCAL actions=drop
 cookie=0x2, duration=42.756s, table=0, n_packets=4, n_bytes=389,
        priority=10,in_port=2 actions=goto_table:1
 cookie=0x1, duration=42.713s, table=0, n_packets=3, n_bytes=238,
        priority=10,in_port=1 actions=write_metadata:0x2/0x3fff,goto_table:2
 cookie=0x2, duration=42.805s, table=0, n_packets=0, n_bytes=0, priority
        =0 actions=write_actions(CONTROLLER:65535)
 cookie=0x0, duration=6.896s, table=1, n_packets=0, n_bytes=0, priority
        =20,vlan_tci=0x1000/0x107f actions=write_actions(output:2)
 cookie=0xfffffffe, duration=42.713s, table=1, n_packets=0, n_bytes=0,
        priority=10,dl_vlan=3841,dl_vlan_pcp=0 actions=write_actions(LOCAL)
 cookie=0x2, duration=42.713s, table=1, n_packets=0, n_bytes=0, priority
        =10,dl_vlan=2305,dl_vlan_pcp=0 actions=write_actions(set_field:8191->
        vlan_vid,set_field:0->vlan_pcp,output:2)
 cookie=0x1, duration=42.713s, table=1, n_packets=3, n_bytes=282,
        priority=10,dl_vlan=2177,dl_vlan_pcp=0 actions=write_actions(pop_vlan
        ,output:1)
 cookie=0x3, duration=42.805s, table=1, n_packets=1, n_bytes=107,
        priority=0 actions=write_actions(CONTROLLER:65535)
 cookie=0x0, duration=6.896s, table=2, n_packets=0, n_bytes=0, priority
        =65535,metadata=0x2/0x3fff,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc
        actions=group:0
 cookie=0x0, duration=6.896s, table=2, n_packets=0, n_bytes=0, priority
        =65535,metadata=0x3/0x3fff,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc
        actions=group:0
 cookie=0x0, duration=4.060s, table=2, n_packets=2, n_bytes=196, priority
        =1,metadata=0x2/0x3fff,in_port=1,dl_dst=00:00:00:00:00:01 actions=
        group:2
 cookie=0x0, duration=4.060s, table=2, n_packets=0, n_bytes=0, priority
        =1,metadata=0x3/0x3fff,in_port=1,dl_dst=00:00:00:00:00:01 actions=
        group:2
 cookie=0x0, duration=6.896s, table=2, n_packets=1, n_bytes=42, priority
        =0,metadata=0x2/0x3fff actions=group:0
 cookie=0x0, duration=6.896s, table=2, n_packets=0, n_bytes=0, priority
        =0,metadata=0x3/0x3fff actions=group:0
mininet> dpctl dump-groups -O OpenFlow13
*** s1
    ------------------------------------------------------------------------

OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
 group_id=0,type=indirect,bucket=actions=CONTROLLER:65535
 group_id=3,type=indirect,bucket=actions=push_vlan:0x8100,set_field
        :6273->vlan_vid,set_field:0->vlan_pcp,output:2
 group_id=1,type=all,bucket=actions=group:2,bucket=actions=group:3
 group_id=2,type=indirect,bucket=actions=output:1
*** s2
    ------------------------------------------------------------------------

OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
```

```
group_id=0,type=indirect,bucket=actions=CONTROLLER:65535
group_id=1,type=all,bucket=actions=group:2,bucket=actions=group:3
group_id=3,type=indirect,bucket=actions=output:1
group_id=2,type=indirect,bucket=actions=push_vlan:0x8100,set_field
    :6272->vlan_vid,set_field:0->vlan_pcp,output:2
```

Listing A.6: Debug output for physical switch 1

```
[PhysicalSwitch id=0, dpid=1] = {
  ports = [
    {
      id = 2
      state = link_rule
      needed-ports = { }
    }
    {
      id = 1
      state = host_rule
      needed-ports = { [Virtual switch dpid=200, state=connected] }
    }
    {
      id = 4294967294
      state = drop_rule
      needed-ports = { }
    }
  ]
  rewrite-map = [
    {
      virtual-switch-id = 1
      flood-group-id = 1
      group-id-map = [
      ]
      output-groups = [
        {
          virtual-port-id = 31
          group-id = 3
          output-port = 2
          state = switch_rule
        }
        {
          virtual-port-id = 30
          group-id = 2
          output-port = 1
          state = host_rule
        }
      ]
    }
  ]
}
```

Listing A.7: Debug output for physical switch 2

```
[PhysicalSwitch id=1, dpid=2] = {
  ports = [
    {
      id = 2
      state = link_rule
      needed-ports = { }
    }
    {
      id = 1
      state = host_rule
      needed-ports = { [Virtual switch dpid=200, state=connected] }
    }
    {
      id = 4294967294
      state = drop_rule
      needed-ports = { }
    }
  ]
  rewrite-map = [
    {
      virtual-switch-id = 1
      flood-group-id = 1
      group-id-map = [
      ]
      output-groups = [
        {
          virtual-port-id = 31
          group-id = 3
          output-port = 1
          state = host_rule
        }
        {
          virtual-port-id = 30
          group-id = 2
          output-port = 2
          state = switch_rule
        }
      ]
    }
  ]
}
```

A-3 4 linear switches

This experiment covers 4 switches in a line, with 2 hosts connected to each. The topology and slices can be seen in Figure A-3. The first slice, dpid 100, is connected to the flood router discussed in subsection 5-1-2; the second slice, dpid 200 and 201, to the ryu learning switch discussed in subsection 5-1-1. This appendix contains the configuration file in Listing A.8, the output from mininet in Listing A.9 and the debug output in Listing A.10, Listing A.11, Listing A.12 and Listing A.13. This experiment was run in mininet started with the following command:

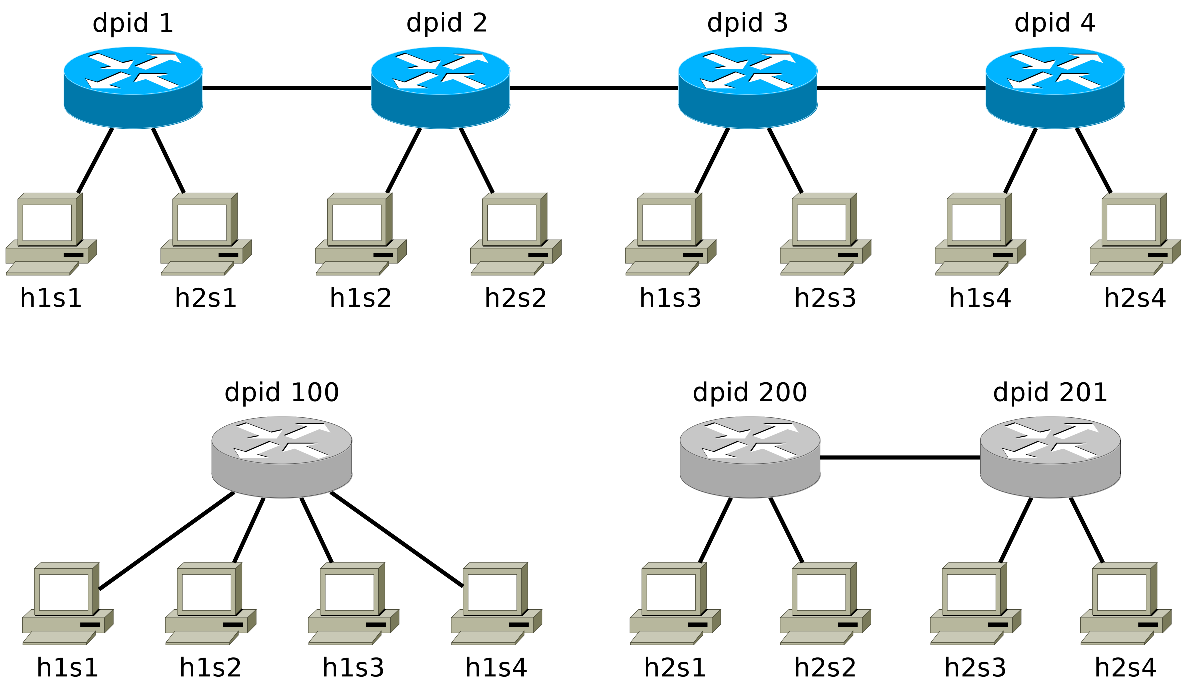


Figure A-3: The topologies of the experiment. The switch with dpid 1, 2, 3 and 4 are the physical switches and the switches with dpid 100, 200 and 201 are the virtual switches. Dpid 100 is in a slice and dpid 200 and 201 are both in a slice.

```
mn --controller remote,ip=192.168.56.1 --switch ovsk,protocols=OpenFlow13
    --mac --topo linear,4,2
```

The rules that these controllers would normally generate if they were connected can be seen in Listing A.14 and Listing A.15 in the form of the mininet output. The topologies used are the mininet topologies single,4 and linear,2,2. This should give the same view of the network as Delftvisor exposes to the tenants. There are small differences though, the mac addresses of the hosts, the dpid of the switches and the port numbers are different. These output files should show how much rules and what type of the rules the controllers would normally generate.

The input configuration file in Listing A.8 just explains the topology in Figure A-3 to the hypervisor. Switch with dpid 100 is in slice 1 and the switches with dpid 200 and dpid 201 are in slice 2. Note that the experiment would be almost the same if all those virtual switches are in the same slice, since they don't have a shared link they both use.

The mininet output in Listing A.9 shows first the output of the pingall command. This pings each host from each other host, the output shows that all hosts that should be able to reach each other can reach each other. After this command has the ryu learning switch learned about all hosts on the network and installed rules to handle that traffic.

Only the output of the switch with dpid 1 will be discussed since all the other switches are similar. This output is visible in Listing A.9 from line 29 to 60. Table 0 are the rules from line 30 to 36. Line 30 contains the topology discovery rule sending topology discovery packets to the hypervisor, which is why 233 packets have been handled by this rule. Lines 31 to 35 handle traffic coming in per port. For example on line 34 handles the traffic coming in over port 1. This port is connected to a host and thus tags the packet with the virtual switch id in metadata and forwards to the tenant tables. Line 36 is the error detection rule, just so unhandled packets are detected. This rule was triggered once, probably during startup when that rule was already pushed but the rules above it were not.

Lines 37 to 48 contain that shared link rules. Lines 37 to 39 detect traffic that is meant for other switches and output it over a link that gets the packet to that switch. These rules haven't been triggered in this example because the switch with dpid 2 sits on the end of a line and a packets for another switch would never be sent to this switch. Lines 40 to 47 deal with how to output a packet sent to this switch. For example lines 44 and 45 output packets to the host on port 1, both for a different slice. Line 48 is again an error detection rule to detect packets falling through. It has been triggered 23 times, which is most likely because of ipv6 packets being send over shared links.

Lines 49 to 60 contain the tenant rules. All the tenant rules are duplicated to deal with the action set merge problem discussed in section 3-3-3. Lines 49 and 50 contain the topology discovery rules from the flood controller. Even though it doesn't use the topology information to do routing does the flood controller print the discovered topology for debugging. Lines 51 to 56 contain the routing rules from the ryu learning switch example which know how to forward packets with specific ethernet destination addresses. Lines 57 and 58 are also from the ryu learning switch and send packets with a destination address that is unknown to the controller. Lines 59 and 60 are the rules from the flood router that match everything and forward to the flood group. These rules are also the only rules in this example that actually differ in the actions while originating from the same virtual rule.

Lines 178 to 188 of Listing A.9 describe the groups created in the physical switch. For each virtual port a group is created that deals with outputting the packet. Line 184 contains the group with id 0 that simulates the controller port and send the packet to the hypervisor. Lines 179 and 182 contain the virtual FLOOD groups created for each virtual switch that has ports on this switch. Line 182 has the group with id 1 that forwards to the groups 2, 3, 4 and 5. This is the virtual FLOOD group for virtual switch with dpid 100. No groups are created by the controllers in this example so all other groups are virtual port groups.

Listing A.10 contains the debug output generated by the hypervisor for the physical switch with dpid 1. It shows the internal state of the hypervisor, which is discussed in section 4-4. Lines 2 to 23 cover the ports that the hypervisor knows about on the physical switch. For each port it has a state in which it stores the current rule that has been pushed to the hypervisor. The needed ports are a separate structure that store what ports are needed by which virtual switch. On this switch only port 1 and 2 are needed by virtual switches. Lines 24 to 83 cover the rewrite-map, which is a structure that handles group allocation and virtual ports per virtual switch. In this structure it is easy to see that for virtual switch 1 the flood group has id 1, no group id's have been allocated for it and group 5 is the group for virtual port 13. The state and output port have also been saved for each group, this is so the hypervisor can correctly identify the cases in which it needs to update the group.

Listing A.8: Configuration file

```
{
  "switch_endpoint_port" : 6633,
  "use_meters"           : false,
  "slices" : [
    {
      "controller" : {
        "ip"   : "127.0.0.1",
        "port" : 6653
      },
      "max_rate"         : 500,
      "virtual_switches" : [
        {
          "datapath_id" : 100,
          "ports" : [
            {
              "virtual_port"         : 10,
              "physical_datapath_id" : 1,
              "physical_port"        : 1
            },
            {
              "virtual_port"         : 11,
              "physical_datapath_id" : 2,
              "physical_port"        : 1
            },
            {
              "virtual_port"         : 12,
              "physical_datapath_id" : 3,
              "physical_port"        : 1
            },
            {
              "virtual_port"         : 13,
              "physical_datapath_id" : 4,
              "physical_port"        : 1
            }
          ]
        }
      ]
    },
    {
      "controller" : {
        "ip"   : "127.0.0.1",
        "port" : 6654
      },
      "max_rate"         : 500,
      "virtual_switches" : [
        {
          "datapath_id" : 200,
          "ports" : [
            {
              "virtual_port"         : 20,
              "physical_datapath_id" : 1,
              "physical_port"        : 2
            },
            {
              "virtual_port"         : 21,
              "physical_datapath_id" : 2,
                "physical_port"         : 2
              },
              {
                "virtual_port"          : 22,
                "physical_datapath_id" : 2,
                "physical_port"         : 4
              }
            ]
          },
          {
            "datapath_id" : 201,
            "ports" : [
              {
                "virtual_port"          : 30,
                "physical_datapath_id" : 3,
                "physical_port"         : 2
              },
              {
                "virtual_port"          : 31,
                "physical_datapath_id" : 4,
                "physical_port"         : 2
              },
              {
                "virtual_port"          : 32,
                "physical_datapath_id" : 3,
                "physical_port"         : 3
              }
            ]
          }
        ]
      }
    ]
}
```

Listing A.9: Mininet output

```
*** Creating network
*** Adding controller
*** Adding hosts:
h1s1 h1s2 h1s3 h1s4 h2s1 h2s2 h2s3 h2s4
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(h1s1, s1) (h1s2, s2) (h1s3, s3) (h1s4, s4) (h2s1, s1) (h2s2, s2) (h2s3,
    s3) (h2s4, s4) (s2, s1) (s3, s2) (s4, s3)
*** Configuring hosts
h1s1 h1s2 h1s3 h1s4 h2s1 h2s2 h2s3 h2s4
*** Starting controller
c0
*** Starting 4 switches
s1 s2 s3 s4 ...
*** Starting CLI:
mininet> pingall
```

```
*** Ping: testing ping reachability
h1s1 -> h1s2 h1s3 h1s4 X X X X
h1s2 -> h1s1 h1s3 h1s4 X X X X
h1s3 -> h1s1 h1s2 h1s4 X X X X
h1s4 -> h1s1 h1s2 h1s3 X X X X
h2s1 -> X X X X h2s2 h2s3 h2s4
h2s2 -> X X X X h2s1 h2s3 h2s4
h2s3 -> X X X X h2s1 h2s2 h2s4
h2s4 -> X X X X h2s1 h2s2 h2s3
*** Results: 57% dropped (24/56 received)
mininet> dpctl dump-flows -O OpenFlow13
*** s1
-------------------------------------------------------------------------------
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x1, duration=112.883s, table=0, n_packets=223, n_bytes=15164, priority=20,vlan_tci=0xf800/0xf800 actions=write_actions(CONTROLLER:65535)
 cookie=0xfffffffd, duration=112.883s, table=0, n_packets=0, n_bytes=0, priority=10,in_port=CONTROLLER actions=goto_table:1
 cookie=0xfffffffe, duration=112.807s, table=0, n_packets=0, n_bytes=0, priority=10,in_port=LOCAL actions=drop
 cookie=0x3, duration=112.807s, table=0, n_packets=178, n_bytes=14005, priority=10,in_port=3 actions=goto_table:1
 cookie=0x1, duration=112.807s, table=0, n_packets=30, n_bytes=1776, priority=10,in_port=1 actions=write_metadata:0x2/0x3fff,goto_table:2
 cookie=0x2, duration=112.807s, table=0, n_packets=31, n_bytes=1902, priority=10,in_port=2 actions=write_metadata:0x4/0x3fff,goto_table:2
 cookie=0x2, duration=112.883s, table=0, n_packets=1, n_bytes=78, priority=0 actions=write_actions(CONTROLLER:65535)
 cookie=0x0, duration=112.030s, table=1, n_packets=0, n_bytes=0, priority=20,vlan_tci=0x1003/0x107f actions=write_actions(output:3)
 cookie=0x0, duration=112.030s, table=1, n_packets=0, n_bytes=0, priority=20,vlan_tci=0x1002/0x107f actions=write_actions(output:3)
 cookie=0x0, duration=112.030s, table=1, n_packets=0, n_bytes=0, priority=20,vlan_tci=0x1000/0x107f actions=write_actions(output:3)
 cookie=0xfffffffe, duration=112.807s, table=1, n_packets=0, n_bytes=0, priority=10,dl_vlan=3841,dl_vlan_pcp=0 actions=write_actions(LOCAL)
 cookie=0xfffffffe, duration=112.807s, table=1, n_packets=0, n_bytes=0, priority=10,dl_vlan=1793,dl_vlan_pcp=1 actions=write_actions(LOCAL)
 cookie=0x3, duration=112.807s, table=1, n_packets=0, n_bytes=0, priority=10,dl_vlan=2433,dl_vlan_pcp=0 actions=write_actions(set_field:8191->vlan_vid,set_field:0->vlan_pcp,output:3)
 cookie=0x3, duration=112.807s, table=1, n_packets=0, n_bytes=0, priority=10,dl_vlan=385,dl_vlan_pcp=1 actions=write_actions(set_field:6143->vlan_vid,set_field:1->vlan_pcp,output:3)
 cookie=0x1, duration=112.807s, table=1, n_packets=87, n_bytes=5508, priority=10,dl_vlan=2177,dl_vlan_pcp=0 actions=write_actions(pop_vlan,output:1)
 cookie=0x1, duration=112.807s, table=1, n_packets=0, n_bytes=0, priority=10,dl_vlan=129,dl_vlan_pcp=1 actions=write_actions(pop_vlan,output:1)
 cookie=0x2, duration=112.807s, table=1, n_packets=0, n_bytes=0, priority
```

```
    =10,dl_vlan=2305,dl_vlan_pcp=0 actions=write_actions(pop_vlan,output
    :2)
 cookie=0x2, duration=112.807s, table=1, n_packets=66, n_bytes=5400,
    priority=10,dl_vlan=257,dl_vlan_pcp=1 actions=write_actions(pop_vlan,
    output:2)
 cookie=0x3, duration=112.883s, table=1, n_packets=23, n_bytes=2929,
    priority=0 actions=write_actions(CONTROLLER:65535)
 cookie=0x0, duration=112.025s, table=2, n_packets=0, n_bytes=0, priority
    =65535,metadata=0x2/0x3fff,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc
    actions=group:0
 cookie=0x0, duration=112.025s, table=2, n_packets=0, n_bytes=0, priority
    =65535,metadata=0x3/0x3fff,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc
    actions=group:0
 cookie=0x0, duration=41.362s, table=2, n_packets=2, n_bytes=140,
    priority=1,metadata=0x4/0x3fff,in_port=2,dl_dst=00:00:00:00:00:06
    actions=group:8
 cookie=0x0, duration=41.321s, table=2, n_packets=0, n_bytes=0, priority
    =1,metadata=0x5/0x3fff,in_port=2,dl_dst=00:00:00:00:00:06 actions=
    group:8
 cookie=0x0, duration=41.256s, table=2, n_packets=2, n_bytes=140,
    priority=1,metadata=0x4/0x3fff,in_port=2,dl_dst=00:00:00:00:00:07
    actions=group:9
 cookie=0x0, duration=41.213s, table=2, n_packets=0, n_bytes=0, priority
    =1,metadata=0x5/0x3fff,in_port=2,dl_dst=00:00:00:00:00:07 actions=
    group:9
 cookie=0x0, duration=41.193s, table=2, n_packets=2, n_bytes=140,
    priority=1,metadata=0x4/0x3fff,in_port=2,dl_dst=00:00:00:00:00:08
    actions=group:9
 cookie=0x0, duration=41.149s, table=2, n_packets=0, n_bytes=0, priority
    =1,metadata=0x5/0x3fff,in_port=2,dl_dst=00:00:00:00:00:08 actions=
    group:9
 cookie=0x0, duration=112.030s, table=2, n_packets=23, n_bytes=1314,
    priority=0,metadata=0x4/0x3fff actions=group:0
 cookie=0x0, duration=112.030s, table=2, n_packets=0, n_bytes=0, priority
    =0,metadata=0x5/0x3fff actions=group:0
 cookie=0x5, duration=112.025s, table=2, n_packets=30, n_bytes=1776,
    priority=0,metadata=0x2/0x3fff actions=write_actions(group:1)
 cookie=0x5, duration=112.025s, table=2, n_packets=0, n_bytes=0, priority
    =0,metadata=0x3/0x3fff actions=drop
*** s2
   ------------------------------------------------------------------------

OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x1, duration=112.882s, table=0, n_packets=446, n_bytes=30328,
    priority=20,vlan_tci=0xf800/0xf800 actions=write_actions(CONTROLLER
    :65535)
 cookie=0xfffffffd, duration=112.882s, table=0, n_packets=0, n_bytes=0,
    priority=10,in_port=CONTROLLER actions=goto_table:1
 cookie=0xfffffffe, duration=112.818s, table=0, n_packets=0, n_bytes=0,
    priority=10,in_port=LOCAL actions=drop
 cookie=0x4, duration=112.818s, table=0, n_packets=196, n_bytes=14559,
    priority=10,in_port=4 actions=goto_table:1
 cookie=0x3, duration=112.818s, table=0, n_packets=164, n_bytes=12423,
    priority=10,in_port=3 actions=goto_table:1
cookie=0x1, duration=112.818s, table=0, n_packets=30, n_bytes=1824,
    priority=10,in_port=1 actions=write_metadata:0x2/0x3fff,goto_table:2
cookie=0x2, duration=112.818s, table=0, n_packets=30, n_bytes=1824,
    priority=10,in_port=2 actions=write_metadata:0x4/0x3fff,goto_table:2
cookie=0x2, duration=112.882s, table=0, n_packets=1, n_bytes=78,
    priority=0 actions=write_actions(CONTROLLER:65535)
cookie=0x0, duration=112.041s, table=1, n_packets=51, n_bytes=3778,
    priority=30,in_port=4,dl_vlan=2047,dl_vlan_pcp=1 actions=pop_vlan,
    write_metadata:0x4/0x3fff,goto_table:2
cookie=0x0, duration=112.041s, table=1, n_packets=30, n_bytes=1896,
    priority=20,vlan_tci=0x1002/0x107f actions=write_actions(output:4)
cookie=0x0, duration=112.041s, table=1, n_packets=60, n_bytes=3826,
    priority=20,vlan_tci=0x1001/0x107f actions=write_actions(output:3)
cookie=0x0, duration=112.041s, table=1, n_packets=30, n_bytes=1896,
    priority=20,vlan_tci=0x1000/0x107f actions=write_actions(output:4)
cookie=0xfffffffe, duration=112.818s, table=1, n_packets=0, n_bytes=0,
    priority=10,dl_vlan=3843,dl_vlan_pcp=0 actions=write_actions(LOCAL)
cookie=0xfffffffe, duration=112.818s, table=1, n_packets=0, n_bytes=0,
    priority=10,dl_vlan=1795,dl_vlan_pcp=1 actions=write_actions(LOCAL)
cookie=0x4, duration=112.818s, table=1, n_packets=0, n_bytes=0, priority
    =10,dl_vlan=2563,dl_vlan_pcp=0 actions=write_actions(set_field:8191->
    vlan_vid,set_field:0->vlan_pcp,output:4)
cookie=0x4, duration=112.818s, table=1, n_packets=26, n_bytes=1952,
    priority=10,dl_vlan=515,dl_vlan_pcp=1 actions=write_actions(set_field
    :6143->vlan_vid,set_field:1->vlan_pcp,output:4)
cookie=0x3, duration=112.818s, table=1, n_packets=0, n_bytes=0, priority
    =10,dl_vlan=2435,dl_vlan_pcp=0 actions=write_actions(set_field:8191->
    vlan_vid,set_field:0->vlan_pcp,output:3)
cookie=0x3, duration=112.818s, table=1, n_packets=0, n_bytes=0, priority
    =10,dl_vlan=387,dl_vlan_pcp=1 actions=write_actions(set_field:6143->
    vlan_vid,set_field:1->vlan_pcp,output:3)
cookie=0x1, duration=112.818s, table=1, n_packets=90, n_bytes=5722,
    priority=10,dl_vlan=2179,dl_vlan_pcp=0 actions=write_actions(pop_vlan
    ,output:1)
cookie=0x1, duration=112.818s, table=1, n_packets=0, n_bytes=0, priority
    =10,dl_vlan=131,dl_vlan_pcp=1 actions=write_actions(pop_vlan,output
    :1)
cookie=0x2, duration=112.818s, table=1, n_packets=0, n_bytes=0, priority
    =10,dl_vlan=2307,dl_vlan_pcp=0 actions=write_actions(pop_vlan,output
    :2)
cookie=0x2, duration=112.818s, table=1, n_packets=23, n_bytes=1686,
    priority=10,dl_vlan=259,dl_vlan_pcp=1 actions=write_actions(pop_vlan,
    output:2)
cookie=0x3, duration=112.882s, table=1, n_packets=48, n_bytes=6058,
    priority=0 actions=write_actions(CONTROLLER:65535)
cookie=0x0, duration=112.036s, table=2, n_packets=0, n_bytes=0, priority
    =65535,metadata=0x2/0x3fff,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc
    actions=group:0
cookie=0x0, duration=112.036s, table=2, n_packets=0, n_bytes=0, priority
    =65535,metadata=0x3/0x3fff,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc
    actions=group:0
cookie=0x0, duration=41.401s, table=2, n_packets=3, n_bytes=238,
        priority=1,metadata=0x4/0x3fff,in_port=2,dl_dst=00:00:00:00:00:05
        actions=group:7
  cookie=0x0, duration=41.378s, table=2, n_packets=0, n_bytes=0, priority
        =1,metadata=0x5/0x3fff,in_port=2,dl_dst=00:00:00:00:00:05 actions=
        group:7
  cookie=0x0, duration=41.272s, table=2, n_packets=7, n_bytes=562,
        priority=1,metadata=0x4/0x3fff,in_port=4,dl_dst=00:00:00:00:00:05
        actions=group:7
  cookie=0x0, duration=41.272s, table=2, n_packets=0, n_bytes=0, priority
        =1,metadata=0x5/0x3fff,in_port=4,dl_dst=00:00:00:00:00:05 actions=
        group:7
  cookie=0x0, duration=28.846s, table=2, n_packets=7, n_bytes=562,
        priority=1,metadata=0x4/0x3fff,in_port=4,dl_dst=00:00:00:00:00:06
        actions=group:8
  cookie=0x0, duration=28.805s, table=2, n_packets=0, n_bytes=0, priority
        =1,metadata=0x5/0x3fff,in_port=4,dl_dst=00:00:00:00:00:06 actions=
        group:8
  cookie=0x0, duration=28.799s, table=2, n_packets=2, n_bytes=140,
        priority=1,metadata=0x4/0x3fff,in_port=2,dl_dst=00:00:00:00:00:07
        actions=group:9
  cookie=0x0, duration=28.777s, table=2, n_packets=0, n_bytes=0, priority
        =1,metadata=0x5/0x3fff,in_port=2,dl_dst=00:00:00:00:00:07 actions=
        group:9
  cookie=0x0, duration=28.759s, table=2, n_packets=2, n_bytes=140,
        priority=1,metadata=0x4/0x3fff,in_port=2,dl_dst=00:00:00:00:00:08
        actions=group:9
  cookie=0x0, duration=28.717s, table=2, n_packets=0, n_bytes=0, priority
        =1,metadata=0x5/0x3fff,in_port=2,dl_dst=00:00:00:00:00:08 actions=
        group:9
  cookie=0x0, duration=112.041s, table=2, n_packets=59, n_bytes=3870,
        priority=0,metadata=0x4/0x3fff actions=group:0
  cookie=0x0, duration=112.040s, table=2, n_packets=0, n_bytes=0, priority
        =0,metadata=0x5/0x3fff actions=group:0
  cookie=0x5, duration=112.036s, table=2, n_packets=27, n_bytes=1574,
        priority=0,metadata=0x2/0x3fff actions=write_actions(group:1)
  cookie=0x5, duration=112.036s, table=2, n_packets=0, n_bytes=0, priority
        =0,metadata=0x3/0x3fff actions=drop
*** s3

OFPST_FLOW reply (OF1.3) (xid=0x2):
  cookie=0x1, duration=112.892s, table=0, n_packets=447, n_bytes=30396,
        priority=20,vlan_tci=0xf800/0xf800 actions=write_actions(CONTROLLER
        :65535)
  cookie=0xfffffffd, duration=112.892s, table=0, n_packets=0, n_bytes=0,
        priority=10,in_port=CONTROLLER actions=goto_table:1
  cookie=0xfffffffe, duration=112.828s, table=0, n_packets=0, n_bytes=0,
        priority=10,in_port=LOCAL actions=drop
  cookie=0x4, duration=112.828s, table=0, n_packets=163, n_bytes=12339,
        priority=10,in_port=4 actions=goto_table:1
  cookie=0x2, duration=112.828s, table=0, n_packets=31, n_bytes=1902,
        priority=10,in_port=2 actions=write_metadata:0x6/0x3fff,goto_table:2
  cookie=0x3, duration=112.828s, table=0, n_packets=189, n_bytes=14005,
    priority=10,in_port=3 actions=goto_table:1
 cookie=0x1, duration=112.828s, table=0, n_packets=31, n_bytes=1902,
    priority=10,in_port=1 actions=write_metadata:0x2/0x3fff,goto_table:2
 cookie=0x2, duration=112.892s, table=0, n_packets=0, n_bytes=0, priority
    =0 actions=write_actions(CONTROLLER:65535)
 cookie=0x0, duration=112.369s, table=1, n_packets=51, n_bytes=3842,
    priority=30,in_port=3,dl_vlan=2047,dl_vlan_pcp=1 actions=pop_vlan,
    write_metadata:0x6/0x3fff,goto_table:2
 cookie=0x0, duration=112.051s, table=1, n_packets=31, n_bytes=1976,
    priority=20,vlan_tci=0x1003/0x107f actions=write_actions(output:3)
 cookie=0x0, duration=112.051s, table=1, n_packets=31, n_bytes=1976,
    priority=20,vlan_tci=0x1001/0x107f actions=write_actions(output:3)
 cookie=0x0, duration=112.051s, table=1, n_packets=57, n_bytes=3578,
    priority=20,vlan_tci=0x1000/0x107f actions=write_actions(output:4)
 cookie=0xfffffffe, duration=112.828s, table=1, n_packets=0, n_bytes=0,
    priority=10,dl_vlan=3842,dl_vlan_pcp=0 actions=write_actions(LOCAL)
 cookie=0xfffffffe, duration=112.828s, table=1, n_packets=0, n_bytes=0,
    priority=10,dl_vlan=1794,dl_vlan_pcp=1 actions=write_actions(LOCAL)
 cookie=0x4, duration=112.828s, table=1, n_packets=0, n_bytes=0, priority
    =10,dl_vlan=2562,dl_vlan_pcp=0 actions=write_actions(set_field:8191->
    vlan_vid,set_field:0->vlan_pcp,output:4)
 cookie=0x4, duration=112.828s, table=1, n_packets=0, n_bytes=0, priority
    =10,dl_vlan=514,dl_vlan_pcp=1 actions=write_actions(set_field:6143->
    vlan_vid,set_field:1->vlan_pcp,output:4)
 cookie=0x2, duration=112.828s, table=1, n_packets=0, n_bytes=0, priority
    =10,dl_vlan=2306,dl_vlan_pcp=0 actions=write_actions(pop_vlan,output
    :2)
 cookie=0x2, duration=112.828s, table=1, n_packets=21, n_bytes=1546,
    priority=10,dl_vlan=258,dl_vlan_pcp=1 actions=write_actions(pop_vlan,
    output:2)
 cookie=0x3, duration=112.828s, table=1, n_packets=0, n_bytes=0, priority
    =10,dl_vlan=2434,dl_vlan_pcp=0 actions=write_actions(set_field:8191->
    vlan_vid,set_field:0->vlan_pcp,output:3)
 cookie=0x3, duration=112.828s, table=1, n_packets=25, n_bytes=1858,
    priority=10,dl_vlan=386,dl_vlan_pcp=1 actions=write_actions(set_field
    :6143->vlan_vid,set_field:1->vlan_pcp,output:3)
 cookie=0x1, duration=112.828s, table=1, n_packets=88, n_bytes=5554,
    priority=10,dl_vlan=2178,dl_vlan_pcp=0 actions=write_actions(pop_vlan
    ,output:1)
 cookie=0x1, duration=112.828s, table=1, n_packets=0, n_bytes=0, priority
    =10,dl_vlan=130,dl_vlan_pcp=1 actions=write_actions(pop_vlan,output
    :1)
 cookie=0x3, duration=112.892s, table=1, n_packets=46, n_bytes=5858,
    priority=0 actions=write_actions(CONTROLLER:65535)
 cookie=0x0, duration=112.045s, table=2, n_packets=0, n_bytes=0, priority
    =65535,metadata=0x2/0x3fff,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc
    actions=group:0
 cookie=0x0, duration=112.045s, table=2, n_packets=0, n_bytes=0, priority
    =65535,metadata=0x3/0x3fff,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc
    actions=group:0
 cookie=0x0, duration=41.328s, table=2, n_packets=3, n_bytes=238,
    priority=1,metadata=0x6/0x3fff,in_port=2,dl_dst=00:00:00:00:00:05
    actions=group:4
 cookie=0x0, duration=41.285s, table=2, n_packets=0, n_bytes=0, priority=1,metadata=0x7/0x3fff,in_port=2,dl_dst=00:00:00:00:00:05 actions=group:4
 cookie=0x0, duration=41.230s, table=2, n_packets=5, n_bytes=414, priority=1,metadata=0x6/0x3fff,in_port=3,dl_dst=00:00:00:00:00:07 actions=group:2
 cookie=0x0, duration=41.229s, table=2, n_packets=0, n_bytes=0, priority=1,metadata=0x7/0x3fff,in_port=3,dl_dst=00:00:00:00:00:07 actions=group:2
 cookie=0x0, duration=41.166s, table=2, n_packets=5, n_bytes=414, priority=1,metadata=0x6/0x3fff,in_port=3,dl_dst=00:00:00:00:00:08 actions=group:3
 cookie=0x0, duration=41.165s, table=2, n_packets=0, n_bytes=0, priority=1,metadata=0x7/0x3fff,in_port=3,dl_dst=00:00:00:00:00:08 actions=group:3
 cookie=0x0, duration=28.863s, table=2, n_packets=3, n_bytes=238, priority=1,metadata=0x6/0x3fff,in_port=2,dl_dst=00:00:00:00:00:06 actions=group:4
 cookie=0x0, duration=28.858s, table=2, n_packets=0, n_bytes=0, priority=1,metadata=0x7/0x3fff,in_port=2,dl_dst=00:00:00:00:00:06 actions=group:4
 cookie=0x0, duration=16.369s, table=2, n_packets=2, n_bytes=140, priority=1,metadata=0x6/0x3fff,in_port=2,dl_dst=00:00:00:00:00:08 actions=group:3
 cookie=0x0, duration=16.367s, table=2, n_packets=0, n_bytes=0, priority=1,metadata=0x7/0x3fff,in_port=2,dl_dst=00:00:00:00:00:08 actions=group:3
 cookie=0x0, duration=112.413s, table=2, n_packets=63, n_bytes=4222, priority=0,metadata=0x6/0x3fff actions=group:0
 cookie=0x0, duration=112.413s, table=2, n_packets=0, n_bytes=0, priority=0,metadata=0x7/0x3fff actions=group:0
 cookie=0x5, duration=112.045s, table=2, n_packets=29, n_bytes=1734, priority=0,metadata=0x2/0x3fff actions=write_actions(group:5)
 cookie=0x5, duration=112.045s, table=2, n_packets=0, n_bytes=0, priority=0,metadata=0x3/0x3fff actions=drop
*** s4
```

```
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x1, duration=112.931s, table=0, n_packets=224, n_bytes=15232, priority=20,vlan_tci=0xf800/0xf800 actions=write_actions(CONTROLLER:65535)
 cookie=0xfffffffd, duration=112.931s, table=0, n_packets=0, n_bytes=0, priority=10,in_port=CONTROLLER actions=goto_table:1
 cookie=0xfffffffe, duration=112.843s, table=0, n_packets=0, n_bytes=0, priority=10,in_port=LOCAL actions=drop
 cookie=0x3, duration=112.843s, table=0, n_packets=176, n_bytes=13979, priority=10,in_port=3 actions=goto_table:1
 cookie=0x2, duration=112.843s, table=0, n_packets=29, n_bytes=1734, priority=10,in_port=2 actions=write_metadata:0x6/0x3fff,goto_table:2
 cookie=0x1, duration=112.843s, table=0, n_packets=32, n_bytes=1944, priority=10,in_port=1 actions=write_metadata:0x2/0x3fff,goto_table:2
 cookie=0x2, duration=112.931s, table=0, n_packets=1, n_bytes=78,
     priority=0 actions=write_actions(CONTROLLER:65535)
 cookie=0x0, duration=112.055s, table=1, n_packets=0, n_bytes=0, priority
     =20,vlan_tci=0x1003/0x107f actions=write_actions(output:3)
 cookie=0x0, duration=112.055s, table=1, n_packets=0, n_bytes=0, priority
     =20,vlan_tci=0x1002/0x107f actions=write_actions(output:3)
 cookie=0x0, duration=112.055s, table=1, n_packets=0, n_bytes=0, priority
     =20,vlan_tci=0x1001/0x107f actions=write_actions(output:3)
 cookie=0xfffffffe, duration=112.843s, table=1, n_packets=0, n_bytes=0,
     priority=10,dl_vlan=3840,dl_vlan_pcp=0 actions=write_actions(LOCAL)
 cookie=0xfffffffe, duration=112.843s, table=1, n_packets=0, n_bytes=0,
     priority=10,dl_vlan=1792,dl_vlan_pcp=1 actions=write_actions(LOCAL)
 cookie=0x3, duration=112.843s, table=1, n_packets=0, n_bytes=0, priority
     =10,dl_vlan=2432,dl_vlan_pcp=0 actions=write_actions(set_field:8191->
     vlan_vid,set_field:0->vlan_pcp,output:3)
 cookie=0x3, duration=112.843s, table=1, n_packets=0, n_bytes=0, priority
     =10,dl_vlan=384,dl_vlan_pcp=1 actions=write_actions(set_field:6143->
     vlan_vid,set_field:1->vlan_pcp,output:3)
 cookie=0x2, duration=112.843s, table=1, n_packets=0, n_bytes=0, priority
     =10,dl_vlan=2304,dl_vlan_pcp=0 actions=write_actions(pop_vlan,output
     :2)
 cookie=0x2, duration=112.843s, table=1, n_packets=67, n_bytes=5622,
     priority=10,dl_vlan=256,dl_vlan_pcp=1 actions=write_actions(pop_vlan,
     output:2)
 cookie=0x1, duration=112.843s, table=1, n_packets=86, n_bytes=5428,
     priority=10,dl_vlan=2176,dl_vlan_pcp=0 actions=write_actions(pop_vlan
     ,output:1)
 cookie=0x1, duration=112.843s, table=1, n_packets=0, n_bytes=0, priority
     =10,dl_vlan=128,dl_vlan_pcp=1 actions=write_actions(pop_vlan,output
     :1)
 cookie=0x3, duration=112.931s, table=1, n_packets=23, n_bytes=2929,
     priority=0 actions=write_actions(CONTROLLER:65535)
 cookie=0x0, duration=112.052s, table=2, n_packets=0, n_bytes=0, priority
     =65535,metadata=0x2/0x3fff,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc
     actions=group:0
 cookie=0x0, duration=112.052s, table=2, n_packets=0, n_bytes=0, priority
     =65535,metadata=0x3/0x3fff,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc
     actions=group:0
 cookie=0x0, duration=41.221s, table=2, n_packets=3, n_bytes=238,
     priority=1,metadata=0x6/0x3fff,in_port=2,dl_dst=00:00:00:00:00:05
     actions=group:4
 cookie=0x0, duration=41.221s, table=2, n_packets=0, n_bytes=0, priority
     =1,metadata=0x7/0x3fff,in_port=2,dl_dst=00:00:00:00:00:05 actions=
     group:4
 cookie=0x0, duration=28.775s, table=2, n_packets=3, n_bytes=238,
     priority=1,metadata=0x6/0x3fff,in_port=2,dl_dst=00:00:00:00:00:06
     actions=group:4
 cookie=0x0, duration=28.775s, table=2, n_packets=0, n_bytes=0, priority
     =1,metadata=0x7/0x3fff,in_port=2,dl_dst=00:00:00:00:00:06 actions=
     group:4
 cookie=0x0, duration=16.422s, table=2, n_packets=3, n_bytes=238,
     priority=1,metadata=0x6/0x3fff,in_port=2,dl_dst=00:00:00:00:00:07
     actions=group:2
 cookie=0x0, duration=16.378s, table=2, n_packets=0, n_bytes=0, priority
```

```
    =1,metadata=0x7/0x3fff,in_port=2,dl_dst=00:00:00:00:00:07 actions=
    group:2
 cookie=0x0, duration=112.390s, table=2, n_packets=20, n_bytes=1020,
    priority=0,metadata=0x6/0x3fff actions=group:0
 cookie=0x0, duration=112.390s, table=2, n_packets=0, n_bytes=0, priority
    =0,metadata=0x7/0x3fff actions=group:0
 cookie=0x5, duration=112.052s, table=2, n_packets=30, n_bytes=1776,
    priority=0,metadata=0x2/0x3fff actions=write_actions(group:5)
 cookie=0x5, duration=112.052s, table=2, n_packets=0, n_bytes=0, priority
    =0,metadata=0x3/0x3fff actions=drop
mininet> dpctl dump-groups -O OpenFlow13
*** s1
    ------------------------------------------------------------------------

OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
 group_id=6,type=all,bucket=actions=group:7,bucket=actions=group:8,bucket
    =actions=group:9
 group_id=5,type=indirect,bucket=actions=push_vlan:0x8100,set_field
    :6272->vlan_vid,set_field:0->vlan_pcp,output:3
 group_id=8,type=indirect,bucket=actions=push_vlan:0x8100,set_field
    :4355->vlan_vid,set_field:1->vlan_pcp,output:3
 group_id=1,type=all,bucket=actions=group:2,bucket=actions=group:3,bucket
    =actions=group:4,bucket=actions=group:5
 group_id=2,type=indirect,bucket=actions=output:1
 group_id=0,type=indirect,bucket=actions=CONTROLLER:65535
 group_id=3,type=indirect,bucket=actions=push_vlan:0x8100,set_field
    :6275->vlan_vid,set_field:0->vlan_pcp,output:3
 group_id=4,type=indirect,bucket=actions=push_vlan:0x8100,set_field
    :6274->vlan_vid,set_field:0->vlan_pcp,output:3
 group_id=9,type=indirect,bucket=actions=push_vlan:0x8100,set_field
    :4611->vlan_vid,set_field:1->vlan_pcp,output:3
 group_id=7,type=indirect,bucket=actions=output:2
*** s2
    ------------------------------------------------------------------------

OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
 group_id=3,type=indirect,bucket=actions=output:1
 group_id=5,type=indirect,bucket=actions=push_vlan:0x8100,set_field
    :6272->vlan_vid,set_field:0->vlan_pcp,output:4
 group_id=8,type=indirect,bucket=actions=output:2
 group_id=7,type=indirect,bucket=actions=push_vlan:0x8100,set_field
    :4353->vlan_vid,set_field:1->vlan_pcp,output:3
 group_id=1,type=all,bucket=actions=group:2,bucket=actions=group:3,bucket
    =actions=group:4,bucket=actions=group:5
 group_id=2,type=indirect,bucket=actions=push_vlan:0x8100,set_field
    :6273->vlan_vid,set_field:0->vlan_pcp,output:3
 group_id=4,type=indirect,bucket=actions=push_vlan:0x8100,set_field
    :6274->vlan_vid,set_field:0->vlan_pcp,output:4
 group_id=6,type=all,bucket=actions=group:7,bucket=actions=group:8,bucket
    =actions=group:9
 group_id=0,type=indirect,bucket=actions=CONTROLLER:65535
 group_id=9,type=indirect,bucket=actions=push_vlan:0x8100,set_field
    :6143->vlan_vid,set_field:1->vlan_pcp,output:4
```

```
*** s3 ________________________________________________________________________________
OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
 group_id=5,type=all,bucket=actions=group:6,bucket=actions=group:7,bucket
    =actions=group:8,bucket=actions=group:9
 group_id=0,type=indirect,bucket=actions=CONTROLLER:65535
 group_id=6,type=indirect,bucket=actions=push_vlan:0x8100,set_field
    :6273->vlan_vid,set_field:0->vlan_pcp,output:3
 group_id=1,type=all,bucket=actions=group:2,bucket=actions=group:3,bucket
    =actions=group:4
 group_id=2,type=indirect,bucket=actions=output:2
 group_id=4,type=indirect,bucket=actions=push_vlan:0x8100,set_field
    :6143->vlan_vid,set_field:1->vlan_pcp,output:3
 group_id=7,type=indirect,bucket=actions=push_vlan:0x8100,set_field
    :6275->vlan_vid,set_field:0->vlan_pcp,output:3
 group_id=3,type=indirect,bucket=actions=push_vlan:0x8100,set_field
    :4352->vlan_vid,set_field:1->vlan_pcp,output:4
 group_id=8,type=indirect,bucket=actions=output:1
 group_id=9,type=indirect,bucket=actions=push_vlan:0x8100,set_field
    :6272->vlan_vid,set_field:0->vlan_pcp,output:4
*** s4 ________________________________________________________________________________
OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
 group_id=8,type=indirect,bucket=actions=push_vlan:0x8100,set_field
    :6274->vlan_vid,set_field:0->vlan_pcp,output:3
 group_id=5,type=all,bucket=actions=group:6,bucket=actions=group:7,bucket
    =actions=group:8,bucket=actions=group:9
 group_id=3,type=indirect,bucket=actions=output:2
 group_id=0,type=indirect,bucket=actions=CONTROLLER:65535
 group_id=2,type=indirect,bucket=actions=push_vlan:0x8100,set_field
    :4354->vlan_vid,set_field:1->vlan_pcp,output:3
 group_id=7,type=indirect,bucket=actions=push_vlan:0x8100,set_field
    :6275->vlan_vid,set_field:0->vlan_pcp,output:3
 group_id=1,type=all,bucket=actions=group:2,bucket=actions=group:3,bucket
    =actions=group:4
 group_id=6,type=indirect,bucket=actions=push_vlan:0x8100,set_field
    :6273->vlan_vid,set_field:0->vlan_pcp,output:3
 group_id=4,type=indirect,bucket=actions=push_vlan:0x8100,set_field
    :4482->vlan_vid,set_field:1->vlan_pcp,output:3
 group_id=9,type=indirect,bucket=actions=output:1
```

Listing A.10: Debug output for physical switch 1

```
[PhysicalSwitch id=1, dpid=1] = {
  ports = [
    {
      id = 3
      state = link_rule
      needed-ports = { }
    }
    {
      id = 2
      state = host_rule
      needed-ports = { [Virtual switch dpid=200, state=connected] }
    }
    {
      id = 1
      state = host_rule
      needed-ports = { [Virtual switch dpid=100, state=connected] }
    }
    {
      id = 4294967294
      state = drop_rule
      needed-ports = { }
    }
  ]
  rewrite-map = [
    {
      virtual-switch-id = 2
      flood-group-id = 6
      group-id-map = [
      ]
      output-groups = [
        {
          virtual-port-id = 22
          group-id = 9
          output-port = 3
          state = switch_rule
        }
        {
          virtual-port-id = 21
          group-id = 8
          output-port = 3
          state = switch_rule
        }
        {
          virtual-port-id = 20
          group-id = 7
          output-port = 2
          state = host_rule
        }
      ]
    }
    {
      virtual-switch-id = 1
      flood-group-id = 1
      group-id-map = [
      ]
      output-groups = [
        {
          virtual-port-id = 13
          group-id = 5
          output-port = 3
          state = switch_rule
        }
        {
          virtual-port-id = 12
          group-id = 4
          output-port = 3
          state = switch_rule
        }
        {
          virtual-port-id = 11
          group-id = 3
          output-port = 3
          state = switch_rule
        }
        {
          virtual-port-id = 10
          group-id = 2
          output-port = 1
          state = host_rule
        }
      ]
    }
  ]
}
```

Listing A.11: Debug output for physical switch 2

```
[PhysicalSwitch id=3, dpid=2] = {
  ports = [
    {
      id = 3
      state = link_rule
      needed-ports = { }
    }
    {
      id = 4
      state = link_rule
      needed-ports = { [Virtual switch dpid=200, state=connected] }
    }
    {
      id = 1
      state = host_rule
      needed-ports = { [Virtual switch dpid=100, state=connected] }
    }
    {
      id = 4294967294
      state = drop_rule
      needed-ports = { }
    }
    {
      id = 2
      state = host_rule
      needed-ports = { [Virtual switch dpid=200, state=connected] }
    }
  ]
  rewrite-map = [
    {
      virtual-switch-id = 2
      flood-group-id = 6
      group-id-map = [
      ]
      output-groups = [
        {
          virtual-port-id = 22
          group-id = 9
          output-port = 4
          state = shared_link_rule
        }
        {
          virtual-port-id = 21
          group-id = 8
          output-port = 2
          state = host_rule
        }
        {
          virtual-port-id = 20
          group-id = 7
          output-port = 3
          state = switch_rule
        }
      ]
    }
    {
      virtual-switch-id = 1
      flood-group-id = 1
      group-id-map = [
      ]
      output-groups = [
        {
          virtual-port-id = 13
          group-id = 5
          output-port = 4
          state = switch_rule
        }
        {
          virtual-port-id = 12
          group-id = 4
          output-port = 4
          state = switch_rule
        }
        {
          virtual-port-id = 11
          group-id = 3
          output-port = 1
          state = host_rule
        }
        {
          virtual-port-id = 10
          group-id = 2
          output-port = 3
          state = switch_rule
        }
      ]
    }
  ]
}
```

Listing A.12: Debug output for physical switch 3

```
[PhysicalSwitch id=2, dpid=3] = {
  ports = [
    {
      id = 3
      state = link_rule
      needed-ports = { [Virtual switch dpid=201, state=connected] }
    }
    {
      id = 4
      state = link_rule
      needed-ports = { }
    }
    {
      id = 1
      state = host_rule
      needed-ports = { [Virtual switch dpid=100, state=connected] }
    }
    {
      id = 4294967294
      state = drop_rule
      needed-ports = { }
    }
    {
      id = 2
      state = host_rule
      needed-ports = { [Virtual switch dpid=201, state=connected] }
    }
  ]
  rewrite-map = [
    {
      virtual-switch-id = 1
      flood-group-id = 5
      group-id-map = [
      ]
      output-groups = [
        {
          virtual-port-id = 13
          group-id = 9
          output-port = 4
          state = switch_rule
        }
        {
          virtual-port-id = 12
          group-id = 8
          output-port = 1
          state = host_rule
        }
        {
          virtual-port-id = 11
          group-id = 7
          output-port = 3
          state = switch_rule
        }
        {
          virtual-port-id = 10
          group-id = 6
          output-port = 3
          state = switch_rule
        }
      ]
    }
    {
      virtual-switch-id = 3
      flood-group-id = 1
      group-id-map = [
      ]
      output-groups = [
        {
          virtual-port-id = 32
          group-id = 4
          output-port = 3
          state = shared_link_rule
        }
        {
          virtual-port-id = 31
          group-id = 3
          output-port = 4
          state = switch_rule
        }
        {
          virtual-port-id = 30
          group-id = 2
          output-port = 2
          state = host_rule
        }
      ]
    }
  ]
}
```

Listing A.13: Debug output for physical switch 4

```
[PhysicalSwitch id=0, dpid=4] = {
  ports = [
    {
      id = 3
      state = link_rule
      needed−ports = { }
    }
    {
      id = 2
      state = host_rule
      needed−ports = { [Virtual switch dpid=201, state=connected] }
    }
    {
      id = 1
      state = host_rule
      needed−ports = { [Virtual switch dpid=100, state=connected] }
    }
    {
      id = 4294967294
      state = drop_rule
      needed−ports = { }
    }
  ]
  rewrite−map = [
    {
      virtual−switch−id = 1
      flood−group−id = 5
      group−id−map = [
      ]
      output−groups = [
        {
          virtual−port−id = 13
          group−id = 9
          output−port = 1
          state = host_rule
        }
        {
          virtual−port−id = 12
          group−id = 8
          output−port = 3
          state = switch_rule
        }
        {
          virtual−port−id = 11
          group−id = 7
          output−port = 3
          state = switch_rule
        }
        {
          virtual−port−id = 10
          group−id = 6
          output−port = 3
          state = switch_rule
        }
      ]
    }
    {
      virtual-switch-id = 3
      flood-group-id = 1
      group-id-map = [
      ]
      output-groups = [
        {
          virtual-port-id = 32
          group-id = 4
          output-port = 3
          state = switch_rule
        }
        {
          virtual-port-id = 31
          group-id = 3
          output-port = 2
          state = host_rule
        }
        {
          virtual-port-id = 30
          group-id = 2
          output-port = 3
          state = switch_rule
        }
      ]
    }
  ]
}
```

Listing A.14: Virtual flows in dpid 100 with the flood router controller

```
*** Creating network
*** Adding controller
Unable to contact the remote controller at 192.168.56.1:6633
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1) (h4, s1)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
```

```
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
mininet> dpctl dump-flows -O OpenFlow13
*** s1
    ----------------------------------------------------------------------------------
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=4.607s, table=0, n_packets=0, n_bytes=0, priority
    =65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER
    :65535
 cookie=0x5, duration=4.607s, table=0, n_packets=40, n_bytes=3136,
    priority=0 actions=write_actions(FLOOD)
mininet> dpctl dump-groups -O OpenFlow13
*** s1
    ----------------------------------------------------------------------------------
OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
```

Listing A.15: Virtual flows in dpid 200 and 201 with the ryu learning switch controller

```
*** Creating network
*** Adding controller
Unable to contact the remote controller at 192.168.56.1:6633
*** Adding hosts:
h1s1 h1s2 h2s1 h2s2
*** Adding switches:
s1 s2
*** Adding links:
(h1s1, s1) (h1s2, s2) (h2s1, s1) (h2s2, s2) (s2, s1)
*** Configuring hosts
h1s1 h1s2 h2s1 h2s2
*** Starting controller
c0
*** Starting 2 switches
s1 s2 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1s1 -> h1s2 h2s1 h2s2
h1s2 -> h1s1 h2s1 h2s2
h2s1 -> h1s1 h1s2 h2s2
h2s2 -> h1s1 h1s2 h2s1
*** Results: 0% dropped (12/12 received)
mininet> dpctl dump-flows -O OpenFlow13
*** s1
    ----------------------------------------------------------------------------------
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=7.573s, table=0, n_packets=7, n_bytes=518, priority
    =1,in_port=3,dl_dst=00:00:00:00:00:01 actions=output:1
 cookie=0x0, duration=7.571s, table=0, n_packets=2, n_bytes=140, priority
    =1,in_port=1,dl_dst=00:00:00:00:00:02 actions=output:3
 cookie=0x0, duration=7.563s, table=0, n_packets=3, n_bytes=238, priority
```

```
    =1,in_port=2,dl_dst=00:00:00:00:00:01 actions=output:1
 cookie=0x0, duration=7.561s, table=0, n_packets=2, n_bytes=140, priority
    =1,in_port=1,dl_dst=00:00:00:00:00:03 actions=output:2
 cookie=0x0, duration=7.551s, table=0, n_packets=2, n_bytes=140, priority
    =1,in_port=1,dl_dst=00:00:00:00:00:04 actions=output:3
 cookie=0x0, duration=7.540s, table=0, n_packets=3, n_bytes=238, priority
    =1,in_port=2,dl_dst=00:00:00:00:00:02 actions=output:3
 cookie=0x0, duration=7.536s, table=0, n_packets=6, n_bytes=420, priority
    =1,in_port=3,dl_dst=00:00:00:00:00:03 actions=output:2
 cookie=0x0, duration=7.513s, table=0, n_packets=2, n_bytes=140, priority
    =1,in_port=2,dl_dst=00:00:00:00:00:04 actions=output:3
 cookie=0x0, duration=14.201s, table=0, n_packets=21, n_bytes=1432,
    priority=0 actions=CONTROLLER:65535
*** s2
   ------------------------------------------------------------------------

OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=7.584s, table=0, n_packets=3, n_bytes=238, priority
    =1,in_port=1,dl_dst=00:00:00:00:00:01 actions=output:3
 cookie=0x0, duration=7.577s, table=0, n_packets=6, n_bytes=420, priority
    =1,in_port=3,dl_dst=00:00:00:00:00:02 actions=output:1
 cookie=0x0, duration=7.561s, table=0, n_packets=3, n_bytes=238, priority
    =1,in_port=2,dl_dst=00:00:00:00:00:01 actions=output:3
 cookie=0x0, duration=7.558s, table=0, n_packets=5, n_bytes=378, priority
    =1,in_port=3,dl_dst=00:00:00:00:00:04 actions=output:2
 cookie=0x0, duration=7.546s, table=0, n_packets=2, n_bytes=140, priority
    =1,in_port=1,dl_dst=00:00:00:00:00:03 actions=output:3
 cookie=0x0, duration=7.538s, table=0, n_packets=3, n_bytes=238, priority
    =1,in_port=2,dl_dst=00:00:00:00:00:02 actions=output:1
 cookie=0x0, duration=7.536s, table=0, n_packets=2, n_bytes=140, priority
    =1,in_port=1,dl_dst=00:00:00:00:00:04 actions=output:2
 cookie=0x0, duration=7.524s, table=0, n_packets=3, n_bytes=238, priority
    =1,in_port=2,dl_dst=00:00:00:00:00:03 actions=output:3
 cookie=0x0, duration=14.208s, table=0, n_packets=21, n_bytes=1376,
    priority=0 actions=CONTROLLER:65535
mininet> dpctl dump-groups -O OpenFlow13
*** s1
   ------------------------------------------------------------------------

OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
*** s2
   ------------------------------------------------------------------------

OFPST_GROUP_DESC reply (OF1.3) (xid=0x2):
```

Appendix B

Measurement controller code

This appendix contains the code used for the measurements.

B-1 Control channel latency code

```python
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3 as ofproto
from ryu.ofproto import ofproto_v1_3_parser as parser
from ryu.ofproto import ether

from ryu.topology import event as topo_event

from ryu.lib.packet import ethernet, arp, packet

from time import time
from array import array

class Router(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(Router,self).__init__(self,*args,**kwargs)

        # A map from from switch dpid to (prev_time_send, file_handle)
        self.switches = {}

    @set_ev_cls(topo_event.EventSwitchEnter)
    def add_switch(self,ev):
        print "Found switch", ev.switch.dp.id

        self.switches[ev.switch.dp.id] = (time(), open("switch_" + str(ev
            .switch.dp.id) + ".dat","a"))
        self.send_packet_out(ev.switch.dp)

    @set_ev_cls(topo_event.EventSwitchLeave)
    def remove_switch(self,ev):
        print "Lost switch", ev.switch.dp.id

    def send_packet_out(self,dp):
        # A sample packet from http://ryu.readthedocs.io/en/latest/
            library_packet.html
        e = ethernet.ethernet(dst='ff:ff:ff:ff:ff:ff',
                              src='08:60:6e:7f:74:e7',
                              ethertype=ether.ETH_TYPE_ARP)
        a = arp.arp(hwtype=1, proto=0x0800, hlen=6, plen=4, opcode=2,
                    src_mac='08:60:6e:7f:74:e7', src_ip='192.0.2.1',
                    dst_mac='00:00:00:00:00:00', dst_ip='192.0.2.2')
        p = packet.Packet()
        p.add_protocol(e)
        p.add_protocol(a)
        p.serialize()

        packet_out = parser.OFPPacketOut(
                datapath=dp,
                buffer_id=ofproto.OFP_NO_BUFFER,
                in_port=ofproto.OFPP_CONTROLLER,
                actions=[parser.OFPActionOutput(ofproto.OFPP_CONTROLLER)
                    ],
                data=p.data
        )
        dp.send_msg(packet_out)

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def receive_packet_in(self,ev):
        pkt = packet.Packet(array('B', ev.msg.data))
        if pkt[0].dst!='ff:ff:ff:ff:ff:ff' or pkt[0].src!='08:60:6e:7f
            :74:e7':
            return

        prev_time, file_handle = self.switches[ev.msg.datapath.id]
        cur_time = time()
        file_handle.write(str(cur_time-prev_time)+"\n")
        self.switches[ev.msg.datapath.id] = (cur_time, file_handle)

        self.send_packet_out(ev.msg.datapath)
```

B-2 Recovery latency code

Listing B.1: The recovery testing script. This script sets up the recovery experiment, starts the timing script and takes down the link.

```
"""
A simple topology to test recovery from failure
```

```
"""

from mininet.cli import CLI
from mininet.log import setLogLevel
from mininet.net import Mininet
from mininet.topo import Topo
from mininet.node import RemoteController, OVSSwitch

import time
import random

class RecoveryTopo( Topo ):
  "Recovery topology with 3 switches and 2 hosts"

  def build( self ):
    h1 = self.addHost("h1")
    h2 = self.addHost("h2")

    s1 = self.addSwitch("s1")
    s2 = self.addSwitch("s2")
    s3 = self.addSwitch("s3")

    self.addLink(s1,h1)
    self.addLink(s2,h2)

    self.addLink(s1,s2)
    self.addLink(s1,s3)
    self.addLink(s2,s3)

if __name__ == '__main__':
  net = Mininet(
    topo = RecoveryTopo(),
    controller = lambda name: RemoteController( name, ip='192.168.56.1' )
        ,
    switch = OVSSwitch,
    autoSetMacs = True )

  # Start the network
  net.start()
  print "Waiting for the switches to connect to the controller"
  # Wait for all switches to connect to a controller
  net.waitConnected()
  print "All switches connected"

  # Get references to the host objects
  h1 = net.get("h1")
  h2 = net.get("h2")

  # Perform the experiment 10 times
  for i in range(1,1001):
    print "Running experiment " + str(i)

    # Start the follow ping command
    h1.cmd("ping -i 0,01 -w 9 "+h2.IP()+" | python timing-script.py
       output_"+str(i)+" &")

    # Wait for the routing to be done by the controller
    time.sleep(2)

    # Drop the link
    net.configLinkStatus('s1','s2','down')

    # Wait for the link to drop and it be detected
    time.sleep(6)

    # Add the link back
    net.configLinkStatus('s1','s2','up')

    # Wait for thing to settle again
    time.sleep(2)

    # Sleep for a random amount between 0 and 1 second, the topology
    # discovery mechanism is periodic and this makes the arrival of
    # the link down event more random
    time.sleep(random.uniform(0,1))

  # Stop the network
  net.stop()
```

Listing B.2: The timing script, it measures the time between ping responses arriving and writes it to a file.

```
import sys
import time

if __name__ == '__main__':
  file_handle = open(sys.argv[1]+".dat",'w')
  file_handle2 = open(sys.argv[1]+".log",'w')

  prev_time = time.time()
  while True:
    line=sys.stdin.readline()
    file_handle2.write(line)
    if not line:
      break
    elif "64 bytes" in line:
      cur_time = time.time()
      file_handle.write(str(cur_time-prev_time)+"\n")
      prev_time = cur_time

  file_handle.close()
```

Listing B.3: The script to analyze the output files produced by the timing script. It detects the delay peak generated by the link down event.

```
import sys

```

```
if __name__ == '__main__':
    result_handle = open("result.dat","w")
    for i in range(1,1001):
        experiment_handle = open("./"+sys.argv[1]+"/output_"+str(i)+".dat
            ","r")

        # Skip the first 50 samples
        for i in range(0,50):
            experiment_handle.readline()

        # The longest delay is the result of the lost link
        highest = 0
        for i in range(0,100):
            highest = max(highest,float(experiment_handle.readline()))

        result_handle.write(str(highest)+"\n")
```

Bibliography

[1] Boost asio documentation. http://www.boost.org/doc/libs/1_55_0/doc/html/boost_asio/overview.html. Accessed: 2017-04-28.

[2] libfluid documentation. https://opennetworkingfoundation.github.io/libfluid/. Accessed: 2017-04-28.

[3] Openvirtex documentation. http://ovx.onlab.us/documentation/architecture/operation-and-subsystems/. Accessed: 2017-04-20.

[4] Ryu documentation. https://osrg.github.io/ryu/. Accessed: 2017-06-03.

[5] Ali Al-Shabibi, Marc De Leenheer, Matteo Gerola, Ayaka Koshibe, Guru Parulkar, Elio Salvadori, and Bill Snow. Openvirtex: Make your virtual sdns programmable. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 25–30, New York, NY, USA, 2014. ACM.

[6] Talal Alharbi, Marius Portmann, and Farzaneh Pakzad. The (in) security of topology discovery in software defined networks. In *Local Computer Networks (LCN), 2015 IEEE 40th Conference on*, pages 502–505. IEEE, 2015.

[7] Theophilus Benson, Aditya Akella, and David A Maltz. Unraveling the complexity of network management.

[8] Sushil Bhardwaj, Leena Jain, and Sandeep Jain. Cloud computing: A study of infrastructure as a service (iaas). *International Journal of engineering and information Technology*, 2(1):60–63, 2010.

[9] Andreas Blenk, Arsany Basta, Johannes Zerwas, Martin Reisslein, and Wolfgang Kellerer. Control plane latency with sdn network hypervisors: The cost of virtualization. *IEEE Transactions on Network and Service Management*, 13(3):366–380, 2016.

[10] Robert W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–, June 1962.

[11] Bob Lantz, Brandon Heller, and Nick 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*, page 19. ACM, 2010.

[12] John Moy. Ospf version 2. 1997.

[13] Bruno Astuto A Nunes, Marc Mendonca, Xuan-Nam Nguyen, Katia Obraczka, and Thierry Turletti. A survey of software-defined networking: Past, present, and future of programmable networks. *IEEE Communications Surveys & Tutorials*, 16(3):1617–1634, 2014.

[14] Open Network Foundation. *OpenFlow Switch Specification Version 1.0.0*, December 2009.

[15] Open Network Foundation. *OpenFlow Switch Specification Version 1.3.5*, March 2015.

[16] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design and implementation of open vswitch. In *NSDI*, pages 117–130, 2015.

[17] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Flowvisor: A network virtualization layer. *OpenFlow Switch Consortium, Tech. Rep*, pages 1–13, 2009.

[18] Niels van Adrichem, Farabi Iqbal, and Fernando A Kuipers. Backup rules in software-defined networks. In *Proc. of the IEEE Conference on Network Function Virtualization and Software Defined Networks*, 2016.