

Carlos Fernandez  
Jose L. Muñoz

UPC Telematics Department

---

# Software Defined Networking (SDN) with OpenFlow 1.3, Open vSwitch and Ryu



# Contents

<b>1</b>	<b>OpenFlow</b>	<b>11</b>
1.1	Introduction to Software-Defined Networking and OpenFlow . . . . .	11
1.1.1	Motivation . . . . .	11
1.1.2	SDN architecture . . . . .	12
1.1.3	OpenFlow . . . . .	14
1.1.4	Benefits of OpenFlow-Based Software-Defined Networks . . . . .	15
1.2	OpenFlow Switches . . . . .	16
1.3	OpenFlow Ports . . . . .	17
1.3.1	Standard Ports . . . . .	17
1.4	OpenFlow Tables . . . . .	18
1.4.1	Pipeline Processing . . . . .	19
1.4.2	Flow Table . . . . .	20
1.4.3	Matching . . . . .	20
1.4.4	Instructions . . . . .	22
1.4.5	Actions . . . . .	23
1.4.6	Table-miss entry . . . . .	25
1.5	Group and Meter tables . . . . .	26
1.5.1	Group Table . . . . .	26
1.5.2	Meter Table . . . . .	27
1.6	OpenFlow Switch Protocol . . . . .	27
1.6.1	Controller-to-Switch messages . . . . .	28

1.6.2	Asynchronous messages . . . . .	28
1.6.3	Symmetric messages . . . . .	29
1.6.4	OpenFlow Channel Connections . . . . .	29
1.6.5	Flow Table Modification Messages . . . . .	30
1.6.6	OpenFlow header . . . . .	31
<b>2</b>	<b>Open vSwitch</b>	<b>33</b>
2.1	Introduction . . . . .	33
2.1.1	Open vSwitch architecture . . . . .	33
2.2	The Kernel Module . . . . .	34
2.2.1	Datapath flows . . . . .	35
2.3	Components and tools of Open vSwitch . . . . .	36
2.3.1	ovs-vswitchd . . . . .	37
2.3.2	ovsdb-server . . . . .	37
2.3.3	ovs-dpctl . . . . .	39
2.3.4	ovs-vsctl . . . . .	40
2.3.5	ovs-appctl . . . . .	40
2.3.6	ovs-ofctl . . . . .	41
2.3.7	ovs-pki . . . . .	41
2.3.8	Further information . . . . .	41
2.4	Practical Examples . . . . .	41
2.4.1	Install . . . . .	41
2.4.2	Basic Commands . . . . .	42
2.4.3	Basic Openflow . . . . .	42
2.5	Implementing a MPLS network with OVS and OpenFlow . . . . .	45
2.5.1	Basics of MPLS . . . . .	45
2.5.2	Setting up the environment . . . . .	46
2.5.3	Testing the network . . . . .	49
2.5.4	Discussion on alternative implementations . . . . .	51

2.6	Final code . . . . .	57
<b>3</b>	<b>Ryu</b>	<b>59</b>
3.1	Introduction . . . . .	59
3.2	Ryu application programming model . . . . .	59
3.2.1	Events and Event Classes . . . . .	59
3.2.2	Event handlers . . . . .	60
3.2.3	The Datapath class . . . . .	60
3.3	OpenFlow protocol implementation . . . . .	61
3.3.1	Controller-to-Switch Messages . . . . .	61
3.3.2	Asynchronous Messages . . . . .	64
3.3.3	Symmetric Messages . . . . .	64
3.3.4	Flow Match Structure . . . . .	65
3.3.5	Flow Instruction Structures . . . . .	66
3.3.6	Action Structures . . . . .	66
3.4	REST API . . . . .	67
3.4.1	Introduction to REST . . . . .	67
3.4.2	Buildng REST APIs with Ryu . . . . .	67
3.4.3	Linking REST Controllers with Ryu applications . . . . .	68
3.5	Components . . . . .	68
3.6	Analysis of a Switch implemented with Ryu . . . . .	69
3.6.1	Class Definition and Initialization . . . . .	70
3.6.2	Event Handler . . . . .	70
3.6.3	Adding Processing of Flow Entry . . . . .	73
3.6.4	Testing the application . . . . .	74
3.6.5	Discussion on alternative implementations . . . . .	78
3.6.6	Source code . . . . .	83
<b>4</b>	<b>MPLS Software-Defined Network</b>	<b>87</b>
4.1	Introduction . . . . .	87

4.2	First Approach: Simple MPLS network . . . . .	87
4.2.1	Mapping the datapaths . . . . .	88
4.2.2	Traffic differentiation . . . . .	88
4.2.3	Testing the application . . . . .	90
4.2.4	Conclusions . . . . .	92
4.3	Second approach: Building the MPLS network from an IP network . . . . .	92
4.3.1	Topology . . . . .	92
4.3.2	REST router . . . . .	93
4.3.3	IP application walkthrough . . . . .	95
4.3.4	Modifications to create a simple MPLS application . . . . .	111
4.3.5	Writing data through the REST API . . . . .	112
4.3.6	Storing labels and hosts . . . . .	115
4.3.7	Handling the packets that enter the network . . . . .	116
4.3.8	Handling the packets that enter the LSR . . . . .	119
4.3.9	Handling the packets that leave the network . . . . .	120
4.3.10	Methods to generate and set the MPLS flows . . . . .	122
4.3.11	Testing the MPLS application . . . . .	124
4.4	Third approach: Next steps . . . . .	127
4.4.1	Simplification . . . . .	127
4.4.2	Generalization . . . . .	128
4.4.3	Robustness and bug fixing . . . . .	128
4.5	Source code . . . . .	128
4.5.1	First approach . . . . .	128
4.5.2	Second approach . . . . .	134

# List of Figures

1.1	SDN architecture	13
1.2	Main components of an OpenFlow Switch	16
1.3	Packet through the processing pipeline	19
1.4	Flowchart detailing packet flow through an OpenFlow switch	21
2.1	OVS architecture	34
2.2	Open vSwitch packet forwarding	35
2.3	Core Table relationship	39
2.4	Topology	46
2.5	Capture from interface s1-eth1	49
2.6	Capture from interface s1-eth2	50
2.7	Capture from interface s2-eth3	50
2.8	Capture from interface s3-eth1	50
2.9	Capture from s1-eth2 and s3-eth2	52
2.10	Capture from s3-eth1	53
2.11	Capture from s1-eth1	54
2.12	Capture from s3-eth1	54
2.13	Capture from s2-eth2	55
2.14	Capture from s1-eth2 in kernel mode	55
2.15	Capture from s2-eth3	56
3.1	Wireshark trace	76
3.2	TCP dump	77

3.3	The wireshark trace shows the message exchange between the controller and the switch . . . . .	78
3.4	Wireshark trace after removing the table-miss entry . . . . .	79
3.5	Wireshark trace without removing the table-miss entry . . . . .	80
3.6	Wireshark trace after the match modification . . . . .	81
3.7	Result of removing packet_out after flow_mod . . . . .	82
4.1	Capture from interface s1-eth1 . . . . .	91
4.2	Capture from interface s1-eth2 . . . . .	91
4.3	Capture from interface s2-eth3 . . . . .	91
4.4	Capture from interface s3-eth1 . . . . .	92
4.5	Topology used for the second approach . . . . .	93
4.6	Topology after the configuration script . . . . .	99
4.7	Capture from interface s1-eth1 . . . . .	107
4.8	Capture from interface s1-eth1 . . . . .	108
4.9	Capture from interface s4-eth1 . . . . .	108
4.10	Capture from interface s4-eth2 . . . . .	108
4.11	Capture from interface s2-eth1 . . . . .	109
4.12	Complete network setup after the REST configuration . . . . .	125
4.13	Capture from interface s2-eth1 . . . . .	125
4.14	Capture from interface s4-eth2 . . . . .	126
4.15	Capture from interface s4-eth3 . . . . .	126
4.16	Capture from interface s3-eth1 . . . . .	126
4.17	Capture from interface s4-eth2 . . . . .	127



# List of Tables

1.1	Match Fields . . . . .	22
1.2	Push/pop tag actions . . . . .	24
1.3	Change TTL actions . . . . .	24
3.1	Controller-to-Switch Messages . . . . .	61
3.2	Asynchronous Messages . . . . .	64
3.3	Symmetric Messages . . . . .	64
3.4	OFPMatch arguments . . . . .	65
3.5	Instruction Classes . . . . .	66
3.6	Action Classes . . . . .	66
3.7	Components of Ryu . . . . .	68



# Chapter 1

## OpenFlow

### 1.1 Introduction to Software-Defined Networking and OpenFlow

The main principle behind Software-Defined Networking (SDN) is the physical separation of the network control plane from the forwarding plane, where a single control plane controls several devices.

As explained in the White Paper published by the Open Networking Foundation [6], Software-Defined Networking is an emerging architecture that is dynamic, manageable, cost-effective, and adaptable, making it ideal for the high-bandwidth, dynamic nature of today's applications. This architecture decouples the network control and forwarding functions enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services. The SDN architecture is:

- **Directly programmable:** Network control is directly programmable because it is decoupled from forwarding functions.
- **Agile:** Abstracting control from forwarding lets administrators dynamically adjust network-wide traffic flow to meet changing needs.
- **Centrally managed:** Network intelligence is (logically) centralized in software-based SDN controllers that maintain a global view of the network, which appears to applications and policy engines as a single, logical switch.
- **Programmatically configured:** SDN lets network managers configure, manage, secure, and optimize network resources very quickly via dynamic, automated SDN programs, which they can write themselves because the programs do not depend on proprietary software.
- **Open standards-based and vendor-neutral:** When implemented through open standards, SDN simplifies network design and operation because instructions are provided by SDN controllers instead of multiple, vendor-specific devices and protocols.

#### 1.1.1 Motivation

SDN addresses the fact that the static architecture of conventional networks is ill-suited to the dynamic computing and storage needs of today's data centers, campuses, and carrier environments. The key computing trends driving the need for a new network paradigm include:

- **Changing traffic patterns:** Applications that commonly access geographically distributed databases and servers through public and private clouds require extremely flexible traffic management and access to bandwidth on demand.
- **The “consumerization of IT”:** The Bring Your Own Device (BYOD) trend requires networks that are both flexible and secure.
- **The rise of cloud services:** Users expect on-demand access to applications, infrastructure, and other IT resources.
- **“Big data” means more bandwidth:** Handling today’s mega datasets requires massive parallel processing that is fueling a constant demand for additional capacity and any-to-any connectivity.

In trying to meet the networking requirements posed by evolving computing trends, network designers find themselves constrained by the limitations of current networks:

- **Complexity that leads to stasis:** Adding or moving devices and implementing network-wide policies are complex, time-consuming, and primarily manual endeavors that risk service disruption, discouraging network changes.
- **Inability to scale:** The time-honored approach of link oversubscription to provision scalability is not effective with the dynamic traffic patterns in virtualized networks—a problem that is even more pronounced in service provider networks with large-scale parallel processing algorithms and associated datasets across an entire computing pool.
- **Vendor dependence:** Lengthy vendor equipment product cycles and a lack of standard, open interfaces limit the ability of network operators to tailor the network to their individual environments.

### 1.1.2 SDN architecture

The architecture of a Software-Defined network can be divided into three planes [7]: Data Plane, Control Plane and Application Plane. Figure 1.1 shows the different elements of a Software-Defined Network, distributed through this three layers:

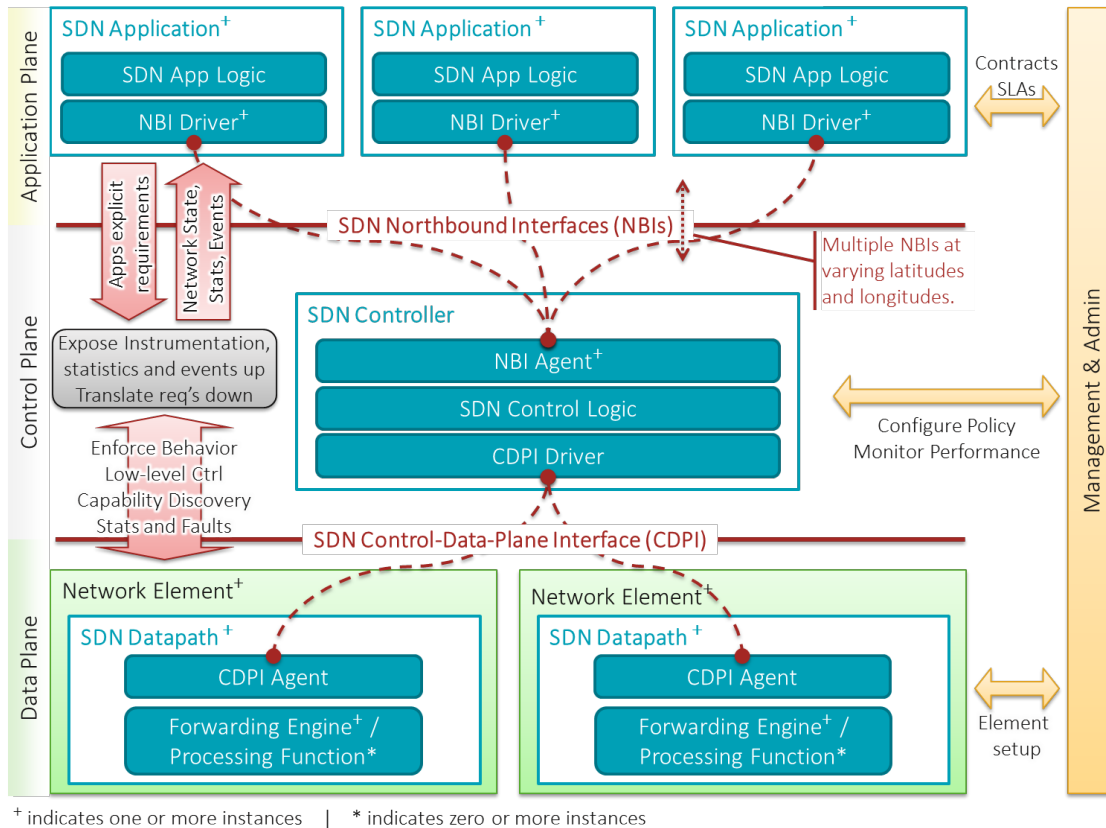


Figure 1.1: SDN architecture

- SDN Application:** SDN Applications are programs that explicitly, directly, and programmatically communicate their network requirements and desired network behavior to the SDN Controller via a northbound interface (NBI). In addition they may consume an abstracted view of the network for their internal decision making purposes. An SDN Application consists of one SDN Application Logic and one or more NBI Drivers. SDN Applications may themselves expose another layer of abstracted network control, thus offering one or more higher-level NBIs through respective NBI agents.
- SDN Controller:** The SDN Controller is a logically centralized entity in charge of (i) translating the requirements from the SDN Application layer down to the SDN Datapaths and (ii) providing the SDN Applications with an abstract view of the network (which may include statistics and events). An SDN Controller consists of one or more NBI Agents, the SDN Control Logic, and the Control to Data-Plane Interface (CDPI) driver. Definition as a logically centralized entity neither prescribes nor precludes implementation details such as the federation of multiple controllers, the hierarchical connection of controllers, communication interfaces between controllers, nor virtualization or slicing of network resources. In this project, the SDN Controller used is Ryu (see chapter 3 for further information).
- SDN Datapath:** The SDN Datapath is a logical network device that exposes visibility and uncontended control over its advertised forwarding and data processing capabilities. The logical representation may encompass all or a subset of the physical substrate resources. An SDN Datapath comprises a CDPI agent and a set of one or more traffic forwarding engines and zero or more traffic processing functions. These engines and functions may include simple forwarding between the datapath's external interfaces or internal traffic processing or termination functions. One or more SDN Datapaths may be contained in a single (physical) network element—an integrated physical combination of communications resources, managed as a unit. An SDN Datapath may also be defined

across multiple physical network elements. This logical definition neither prescribes nor precludes implementation details such as the logical to physical mapping, management of shared physical resources, virtualization or slicing of the SDN Datapath, interoperability with non-SDN networking, nor the data processing functionality, which can include L4-7 functions. In this project, datapaths are implemented using Open vSwitch (see chapter 2) and the network simulator Mininet.

- **SDN Control to Data-Plane Interface (CDPI):** The SDN CDPI is the interface defined between an SDN Controller and an SDN Datapath, which provides at least (i) programmatic control of all forwarding operations, (ii) capabilities advertisement, (iii) statistics reporting, and (iv) event notification. One value of SDN lies in the expectation that the CDPI is implemented in an open, vendor-neutral and interoperable way. This interface is implemented using the OpenFlow protocol, which allows the communication between the Controller and the Datapaths, and specifies how both should handle the information they share.
- **SDN Northbound Interfaces (NBI):** SDN NBIs are interfaces between SDN Applications and SDN Controllers and typically provide abstract network views and enable direct expression of network behavior and requirements. This may occur at any level of abstraction (latitude) and across different sets of functionality (longitude). One value of SDN lies in the expectation that these interfaces are implemented in an open, vendor-neutral and interoperable way. In the case of Ryu, the communication between the Applications and the Controller is implemented using a REST API.

### 1.1.3 OpenFlow

OpenFlow is the first standard communications interface defined between the control and forwarding layers of an SDN architecture. OpenFlow allows direct access to and manipulation of the forwarding plane of network devices such as switches and routers, both physical and virtual (hypervisor-based).

OpenFlow can be compared to the instruction set of a CPU. The protocol specifies basic primitives that can be used by an external software application to program the forwarding plane of network devices, just like the instruction set of a CPU would program a computer system.

The OpenFlow protocol is implemented on both sides of the interface between network infrastructure devices and the SDN control software. OpenFlow uses the concept of **flows** to identify network traffic based on pre-defined **match rules** that can be statically or dynamically programmed by the SDN control software. It also allows IT to define how traffic should flow through network devices based on parameters such as usage patterns, applications, and cloud resources. Since OpenFlow allows the network to be programmed on a per-flow basis, an OpenFlow-based SDN architecture provides extremely granular control, enabling the network to respond to real-time changes at the application, user, and session levels. Current IP-based routing does not provide this level of control, as all flows between two endpoints must follow the same path through the network, regardless of their different requirements.

The OpenFlow protocol is a key enabler for software-defined networks and currently is the only standardized SDN protocol that allows direct manipulation of the forwarding plane of network devices. While initially applied to Ethernet-based networks, OpenFlow switching can extend to a much broader set of use cases. OpenFlow-based SDNs can be deployed on existing networks, both physical and virtual. Network devices can support OpenFlow-based forwarding as well as traditional forwarding, which makes it very easy for enterprises and carriers to progressively introduce OpenFlow-based SDN technologies, even in multi-vendor network environments.

The experiments and information exposed in this document refer to the version 1.3 of this protocol (also known as 0x04 version).

### 1.1.4 Benefits of OpenFlow-Based Software-Defined Networks

OpenFlow-based SDN technologies enable IT to address the high- bandwidth, dynamic nature of today's applications, adapt the network to ever-changing business needs, and significantly reduce operations and management complexity. The benefits that enterprises and carriers can achieve through an OpenFlow-based SDN architecture include:

- **Centralized control of multi-vendor environments:** SDN control software can control any OpenFlow-enabled network device from any vendor, including switches, routers, and virtual switches. Rather than having to manage groups of devices from individual vendors, IT can use SDN-based orchestration and management tools to quickly deploy, configure, and update devices across the entire network.
- **Reduced complexity through automation:** OpenFlow-based SDN offers a flexible network automation and management framework, which makes it possible to develop tools that automate many management tasks that are done manually today. These automation tools will reduce operational overhead, decrease network instability introduced by operator error, and support emerging IT-as-a-Service and self-service provisioning models. In addition, with SDN, cloud-based applications can be managed through intelligent orchestration and provisioning systems, further reducing operational overhead while increasing business agility.
- **Higher rate of innovation:** SDN adoption accelerates business innovation by allowing IT network operators to literally program—and reprogram—the network in real time to meet specific business needs and user requirements as they arise. By virtualizing the network infrastructure and abstracting it from individual network services, for example, SDN and OpenFlow give IT—and potentially even users—the ability to tailor the behavior of the network and introduce new services and network capabilities in a matter of hours.
- **Increased network reliability and security:** SDN makes it possible for IT to define high-level configuration and policy statements, which are then translated down to the infrastructure via OpenFlow. An OpenFlow-based SDN architecture eliminates the need to individually configure network devices each time an end point, service, or application is added or moved, or a policy changes, which reduces the likelihood of network failures due to configuration or policy inconsistencies. Because SDN controllers provide complete visibility and control over the network, they can ensure that access control, traffic engineering, quality of service, security, and other policies are enforced consistently across the wired and wireless network infrastructures, including branch offices, campuses, and data centers. Enterprises and carriers benefit from reduced operational expenses, more dynamic configuration capabilities, fewer errors, and consistent configuration and policy enforcement.
- **More granular network control:** OpenFlow's flow-based control model allows IT to apply policies at a very granular level, including the session, user, device, and application levels, in a highly abstracted, automated fashion. This control enables cloud operators to support multi-tenancy while maintaining traffic isolation, security, and elastic resource management when customers share the same infrastructure.

By decoupling the network control and data planes, OpenFlow-based SDN architecture abstracts the underlying infrastructure from the applications that use it, allowing the network to become as programmable and manageable at scale as the computer infrastructure that it increasingly resembles. An SDN approach fosters network virtualization, enabling IT staff to manage their servers, applications, storage, and networks with a common approach and tool set. Whether in a carrier environment or enterprise data center and campus, SDN adoption can improve network manageability, scalability, and agility.

The future of networking will rely more and more on software, which will accelerate the pace of innovation for networks as it has in the computing and storage domains. SDN promises to transform today's static networks into flexible, programmable platforms with the intelligence to allocate resources dynamically, the scale to support enormous data centers and the virtualization needed to support dynamic, highly automated, and secure cloud environments. With its many advantages and astonishing industry momentum, SDN is on the way to becoming the new norm for networks.

## 1.2 OpenFlow Switches

The following sections are based on OpenFlow Switch Specification version 1.3.5 [8].

An OpenFlow Switch consists of one or more flow tables and a group table, which perform packet lookups and forwarding, and an OpenFlow channel to an external controller. The controller manages the switch via the OpenFlow protocol. Using this protocol, the controller can add, update, and delete flow entries, both reactively (in response to packets) and proactively. The term 'Datapath' is used to refer to OpenFlow Switches as they only implement packet forwarding in comparison to OpenFlow controllers, which implement the intelligence (control path).

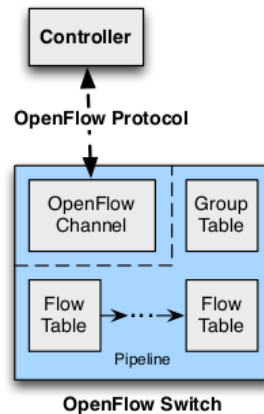


Figure 1.2: Main components of an OpenFlow Switch

Each flow table in the switch contains a set of flow entries; each flow entry consists of match fields, counters, and a set of instructions to apply to matching packets.

Matching starts at the first flow table and may continue to additional flow tables. Flow entries match packets in priority order, with the first matching entry in each table being used. If a matching entry is found, the instructions associated with the specific flow entry are executed. If no match is found in a flow table, the outcome depends on switch configuration: the packet may be forwarded to the controller over the OpenFlow channel, dropped, or may continue to the next flow table.

Instructions associated with each flow entry describe packet forwarding, packet modification, group table processing, and pipeline processing. Pipeline processing instructions allow packets to be sent to subsequent tables for further processing and allow information, in the form of metadata, to be communicated between tables. Table pipeline processing stops when the instruction set associated with a matching flow entry does not specify a next table; at this point the packet is usually modified and forwarded.

Flow entries may forward to a port. This is usually a physical port, but it may also be a virtual port defined by the switch or a reserved virtual port defined by the OpenFlow specification. Reserved virtual ports may specify generic forwarding actions such as sending to the controller, flooding, or forwarding using non-OpenFlow methods, such as “normal” switch processing, while switch-defined virtual ports may specify link aggregation groups, tunnels or loopback interfaces.

Flow entries may also point to a group, which specifies additional processing. Groups represent sets of actions for flooding, as well as more complex forwarding semantics (e.g. multipath, fast reroute, and link aggregation). As a general layer of indirection, groups also enable multiple flows to forward to a single identifier (e.g. IP forwarding to a



common next hop). This abstraction allows common output actions across flows to be changed efficiently.

The group table contains group entries; each group entry contains a list of action buckets with specific semantics dependent on group type. The actions in one or more action buckets are applied to packets sent to the group. Switch designers are free to implement the internals in any way convenient, provided that correct match and instruction semantics are preserved. For example, while a flow may use an all group to forward to multiple ports, a switch designer may choose to implement this as a single bitmask within the hardware forwarding table. Another example is matching; the pipeline exposed by an OpenFlow switch may be physically implemented with a different number of hardware tables.

## 1.3 OpenFlow Ports

OpenFlow ports are the network interfaces for passing packets between OpenFlow processing and the rest of the network. OpenFlow switches connect logically to each other via their OpenFlow ports, a packet can be forwarded from one OpenFlow switch to another OpenFlow switch only via an output OpenFlow port on the first switch and an ingress OpenFlow port on the second switch.

An OpenFlow switch makes a number of OpenFlow ports available for OpenFlow processing. The set of OpenFlow ports may not be identical to the set of network interfaces provided by the switch hardware, some network interfaces may be disabled for OpenFlow, and the OpenFlow switch may define additional OpenFlow ports.

OpenFlow packets are received on an ingress port and processed by the OpenFlow pipeline (see Section 1.4.1), which may forward them to an output port. The packet ingress port is a property of the packet throughout the OpenFlow pipeline and represents the OpenFlow port on which the packet was received into the OpenFlow switch. The ingress port can be used when matching packets. The OpenFlow pipeline can decide to send the packet on an output port using the output action (see Section 1.4.5), which defines how the packet goes back to the network.

### 1.3.1 Standard Ports

The OpenFlow standard ports are defined as physical ports, logical ports, and the LOCAL reserved port if supported (excluding other reserved ports). Standard ports can be used as ingress and output ports, they can be used in groups, they have port counters and they have state and configuration.

#### Physical Ports

The OpenFlow physical ports are switch defined ports that correspond to a hardware interface of the switch. For example, on an Ethernet switch, physical ports map one-to-one to the Ethernet interfaces.

In some deployments, the OpenFlow switch may be virtualised as it's the case of Open vSwitch. In those cases, an OpenFlow physical port may represent a virtual interface of such virtual switch.

#### Logical Ports

The OpenFlow logical ports are switch defined ports that don't correspond directly to a hardware interface of the switch. Logical ports are higher level abstractions that may be defined in the switch using non-OpenFlow methods (e.g. link aggregation groups, tunnels, loopback interfaces).

Logical ports may include packet encapsulation and may map to various physical ports. The processing done by the logical port is implementation dependent and must be transparent to OpenFlow processing, and those ports must interact with OpenFlow processing like OpenFlow physical ports.

**Reserved Ports** are some logical ports defined by the OpenFlow Switch Specification. They specify generic forwarding actions such as sending to the controller, flooding, or forwarding using non-OpenFlow methods, such as “normal” switch processing.

A switch is required to support the following reserved ports:

- **ALL:** Represents all ports the switch can use for forwarding a specific packet. Can be used only as an output port. In that case a copy of the packet is sent to all standard ports, excluding the packet ingress port and ports that are configured to not forward.
- **CONTROLLER:** Represents the control channel with the OpenFlow controllers. Can be used as an ingress port or as an output port. When used as an output port, encapsulates the packet in a packet-in message and sends it using the OpenFlow switch protocol. When used as an ingress port, this identifies a packet originating from the controller.
- **TABLE:** Represents the start of the OpenFlow pipeline. This port is only valid in an output action in the list of actions of a packet-out message and submits the packet to the first flow table so that the packet can be processed through the regular OpenFlow pipeline.
- **IN PORT:** Represents the packet ingress port. Can be used only as an output port, sends the packet out through its ingress port.
- **ANY:** Special value used in some OpenFlow requests when no port is specified (i.e. port is wildcarded). Some OpenFlow requests contain a reference to a specific port that the request only applies to. Using ANY as the port number in these requests allows that request instance to apply to any and all ports. Can neither be used as an ingress port nor as an output port.

The following reserved ports are optional, but supported by Open vSwitch:

- **LOCAL:** Represents the switch’s local networking stack and its management stack. Can be used as an ingress port or as an output port. The local port enables remote entities to interact with the switch and its network services via the OpenFlow network, rather than via a separate control network. With an appropriate set of flow entries, it can be used to implement an in-band controller connection.
- **NORMAL:** Represents forwarding using the traditional non-OpenFlow pipeline of the switch. Can be used only as an output port and processes the packet using the normal pipeline. In general will bridge or route the packet, however the actual result is implementation dependent.
- **FLOOD:** Represents flooding using the traditional non-OpenFlow pipeline of the switch. Can be used only as an output port, actual result is implementation dependent. In general will send the packet out all standard ports, but not to the ingress port, nor ports that are blocked. The switch may also use the packet VLAN ID or other criteria to select which ports to use for flooding.

## 1.4 OpenFlow Tables

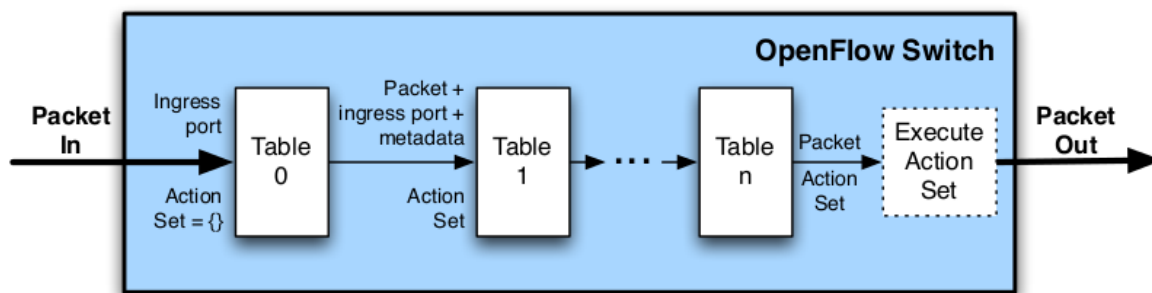
This section describes the components of flow tables and group tables, along with the mechanics of matching and action handling.

## 1.4.1 Pipeline Processing

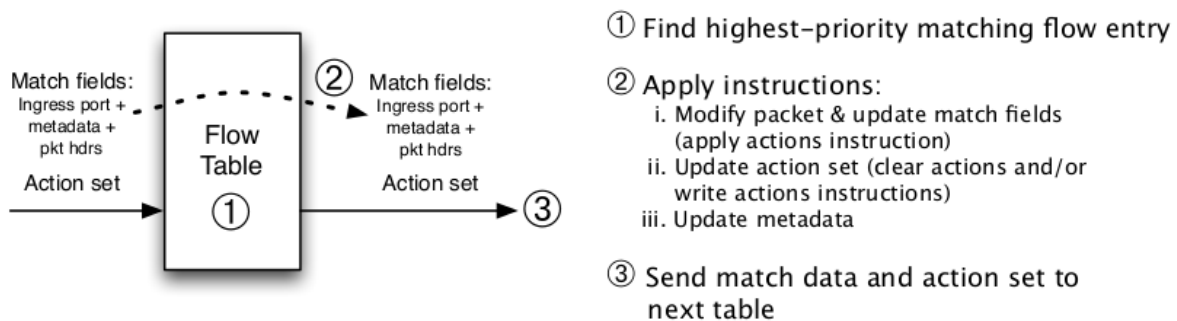
OpenFlow-compliant switches come in two types:

- *OpenFlow-only* switches support only OpenFlow operation, in those switches all packets are processed by the OpenFlow pipeline, and can not be processed otherwise.
- *OpenFlow-hybrid* switches support both OpenFlow operation and normal Ethernet switching operation, i.e. traditional L2 Ethernet switching, VLAN isolation, L3 routing (IPv4 routing, IPv6 routing...), ACL and QoS processing. Those switches should provide a classification mechanism outside of OpenFlow that routes traffic to either the OpenFlow pipeline or the normal pipeline. For example, a switch may use the VLAN tag or input port of the packet to decide whether to process the packet using one pipeline or the other, or it may direct all packets to the OpenFlow pipeline. An *OpenFlow-hybrid* switch may also allow a packet to go from the OpenFlow pipeline to the normal pipeline through the NORMAL and FLOOD reserved ports (see Section 1.3).

An OpenFlow switch is required to have at least one flow table, but it can optionally have more flow tables which is a nice and common feature of switches. Thus, the OpenFlow pipeline contains one or more flow tables. The OpenFlow pipeline processing defines how packets interact with those flow tables (see Figure 1.3).



(a) Packets are matched against multiple tables in the pipeline



(b) Per-table packet processing

Figure 1.3: Packet through the processing pipeline

As shown, all the packets start their processing at Table 0. At this table, the ingress port and the packet headers are used for matching. Tables contain multiple flow entries and each flow has a certain priority within its table. The flow with the highest priority among the possible matches the one selected.

Each flow entry contains a set of **instructions** that are executed when a packet matches the entry. These instructions can be used for different purposes. The main ones are to change the pipeline processing or to apply **actions** to the packet (like send to a port, decrease a TTL, etc.). In particular, we have instructions to:

- **Forward the pipeline processing to another table.** However, before executing the Goto-Table instruction, a table can write a metadata field of 64 bits to make it available to the next table. Therefore, in an intermediate table, in general, a flow entry is matched using the ingress port, the packet headers and metadata that can be written and passed from one table to another.

Metadata can be used for example if in a table we make a classification between two types of packets. These two types of packets can have two different metadata values. Then, we can use the metadata field to pass the type of packet (i.e. the classification made by the previous table) to the next table.

- **Modify the “action set”.** The action set is carried between flow tables. Each flow can add actions to the action set (or clear the action set). When the instruction set of a flow entry does not contain a Goto-Table instruction, pipeline processing stops and the actions in the action set of the packet are executed. It is important to notice that actions of the action set are executed in a certain order (see Section 1.4.5), that in general, is different from the order in which the actions were written in the action set.
- **Apply an “action list”.** The actions of a list of actions are executed in the order specified by the list, and are applied immediately to the packet. After the execution of the list of actions, pipeline execution continues on the modified packet. The action set of the packet is unchanged by the execution of the list of actions.

The previous concepts are further explained in the following sections.

## 1.4.2 Flow Table

A flow table consists of flow entries. Each flow table entry contains:

- **Match fields** to match against packets. These consist of the ingress port and packet headers, and optionally metadata specified by a previous table.
- **Priority** to indicate the matching precedence of the flow entry (higher numbers mean higher priority).
- **Counters** to update for matching packets.
- **Instructions** to modify the action set or pipeline processing.
- **Timeouts** to indicate the maximum amount of time or idle time before flow is expired by the switch. Each flow entry is configured with an `idle_timeout` and a `hard_timeout`. The `hard_timeout` causes the flow entry to be removed after the given number of seconds, regardless of how many packets it has matched. The `idle_timeout` causes the flow entry to be removed when it has matched no packets in the given number of seconds.
- **Cookie:** opaque data value chosen by the controller. May be used by the controller to filter flow statistics, flow modification and flow deletion, not used when processing packets. For example, the controller could choose to modify or delete all flows matching a certain cookie.

## 1.4.3 Matching

On receipt of a packet, an OpenFlow Switch performs the functions shown in Figure 1.4. The switch starts by performing a table lookup in the first flow table, and based on pipeline processing, may perform table lookups in other flow tables.

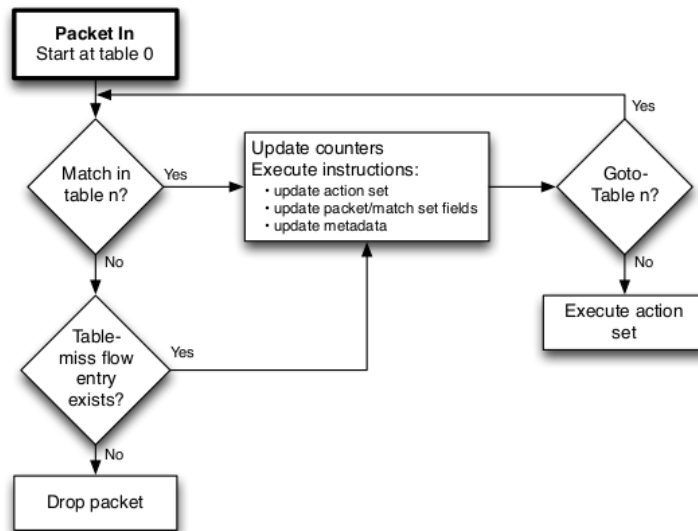


Figure 1.4: Flowchart detailing packet flow through an OpenFlow switch

Packet header fields are extracted from the packet, and packet pipeline fields are retrieved. Packet header fields used for table lookups depend on the packet type, and typically include various protocol header fields, such as Ethernet source address or IPv4 destination address. In addition to packet headers, matches can also be performed against the ingress port, the metadata field and other pipeline fields.

A packet matches a flow entry if all the match fields of the flow entry are matching the corresponding header fields and pipeline fields from the packet. If a match field is omitted in the flow entry (i.e. value ANY), it matches all possible values in the header field or pipeline field of the packet. If the match field is present and does not include a mask, the match field is matching the corresponding header field or pipeline field from the packet if it has the same value. If the switch supports arbitrary bitmasks on specific match fields, these masks can more precisely specify matches, the match field is matching if it has the same value for the bits which are set in the mask.

The packet is matched against flow entries in the flow table and only the highest priority flow entry that matches the packet is selected. The counters associated with the selected flow entry are updated and the instruction set included in the selected flow entry is executed.

Table 1.1 shows the list of match fields supported by OpenFlow 1.3

Table 1.1: Match Fields

Match Field	Description
OXM_OF_ETH_DST	Ethernet destination MAC address.
OXM_OF_ETH_SRC	Ethernet source MAC address.
OXM_OF_ETH_SRC	Ethernet type of the OpenFlow packet payload, after VLAN tags.
OXM_OF_VLAN_VID	VLAN-ID from 802.1Q header. The CFI bit indicates the presence of a valid VLAN-ID, see below.
OXM_OF_VLAN_PCP	VLAN-PCP from 802.1Q header.
OXM_OF_IP_DSCP	Diff Serv Code Point (DSCP). Part of the IPv4 ToS field or the IPv6 Traffic Class field.
OXM_OF_IP_ECN	ECN bits of the IP header. Part of the IPv4 ToS field or the IPv6 Traffic Class field.
OXM_OF_IP_PROTO	IPv4 or IPv6 protocol number.
OXM_OF_IPV4_SRC	IPv4 source address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV4_DST	IPv4 destination address. Can use subnet mask or arbitrary bitmask
OXM_OF_TCP_SRC	TCP source port
OXM_OF_TCP_DST	TCP destination port
OXM_OF_UDP_SRC	UDP source port
OXM_OF_UDP_DST	UDP destination port
OXM_OF_SCTP_SRC	SCTP source port
OXM_OF_SCTP_DST	SCTP destination port
OXM_OF_ICMPV4_TYPE	ICMP type
OXM_OF_ICMPV4_CODE	ICMP code
OXM_OF_ARP_OP	ARP opcode
OXM_OF_ARP_SPA	Source IPv4 address in the ARP payload. Can use subnet mask or arbitrary bitmask
OXM_OF_ARP_TPA	Target IPv4 address in the ARP payload. Can use subnet mask or arbitrary bitmask
OXM_OF_ARP_SHA	Source Ethernet address in the ARP payload.
OXM_OF_ARP_THA	Target Ethernet address in the ARP payload.
OXM_OF_IPV6_SRC	IPv6 source address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV6_DST	IPv6 destination address. Can use subnet mask or arbitrary bitmask
OXM_OF_IPV6_FLABEL	IPv6 flow label.
OXM_OF_ICMPV6_TYPE	ICMPv6 type
OXM_OF_ICMPV6_CODE	ICMPv6 code
OXM_OF_IPV6_ND_TARGET	The target address in an IPv6 Neighbor Discovery message.
OXM_OF_IPV6_ND_SLL	The source link-layer address option in an IPv6 Neighbor Discovery message
OXM_OF_IPV6_ND_TLL	The target link-layer address option in an IPv6 Neighbor Discovery message.
OXM_OF_MPLS_LABEL	The LABEL in the first MPLS shim header.
OXM_OF_MPLS_TC	The TC in the first MPLS shim header.
OXM_OF_MPLS_BOS	The BoS bit (Bottom of Stack bit) in the first MPLS shim header.
OXM_OF_PBB_ISID	The I-SID in the first PBB service instance tag.
OXM_OF_IPV6_EXTHDR	IPv6 Extension Header pseudo-field.

#### 1.4.4 Instructions

Each flow entry contains a set of instructions that are executed when a packet matches the entry. These instructions result in changes to the packet, action set and/or pipeline processing. Supported instructions include:

- **Meter *meter\_id***: Direct packet to the specified meter (See 1.5.2). As the result of the metering, the packet may be dropped (depending on meter configuration and state).

- **Apply-Actions action(s):** Applies the specific action(s) immediately, without any change to the Action Set (see 1.4.5). This instruction may be used to modify the packet between two tables or to execute multiple actions of the same type. The actions are specified as an action list (see 1.4.5).
- **Clear-Actions:** Clears all the actions in the action set immediately.
- **Write-Actions action(s):** Merges the specified action(s) into the current action set. If an action of the given type exists in the current set, overwrite it, otherwise add it.
- **Write-Metadata metadata / mask:** Writes the masked metadata value into the metadata field. The mask specifies which bits of the metadata register should be modified (i.e.  $\text{new metadata} = \text{old metadata} \& \text{mask} | \text{value} \& \text{mask}$ ).
- **Goto-Table next-table-id:** Indicates the next table in the processing pipeline. The table-id must be greater than the current table-id. The flows of last table of the pipeline can not include this instruction.

In practice, the only constraints are that the Meter instruction is executed before the Apply-Actions instruction, that the Clear-Actions instruction is executed before the Write-Actions instruction, and that Goto-Table is executed last.

An OpenFlow switch rejects a flow entry if it is unable to execute the instructions or part of the instructions associated with the flow entry. In this case, the switch returns the error message associated with the issue.

## 1.4.5 Actions

Actions describe packet forwarding, packet modification and group table processing. There are several actions that every OpenFlow Switch is required to support:

- **Output port\_no:** The Output action forwards a packet to a specified OpenFlow port. OpenFlow switches must support forwarding to physical ports, switch-defined logical ports and the required reserved ports.
- **Group group\_id:** Process the packet through the specified group. The exact interpretation depends on group type.
- **Drop:** There is no explicit action to represent drops. Instead, packets whose action sets have no output action and no group action should be dropped. This result could come from empty instruction sets or empty action buckets in the processing pipeline, or after executing a Clear-Actions instruction.

There are also some optional actions that may be supported by the switch. The controller can query the switch about which of the optional actions are supported. The optional actions are:

- **Set-Queue queue\_id:** The set-queue action sets the queue id for a packet. When the packet is forwarded to a port using the output action, the queue id determines which queue attached to this port is used for scheduling and forwarding the packet. Forwarding behavior is dictated by the configuration of the queue and is used to provide basic Quality-of-Service (QoS) support.
- **Push-Tag/Pop-Tag ethertype:** Switches may support the ability to push/pop tags as shown in Table 1.2. To aid integration with existing networks, supporting the ability to push/pop VLAN tags is strongly suggested.
- **Set-Field field\_type value:** The various Set-Field actions are identified by their field type and modify the values of respective header fields in the packet. While not strictly required, the support of rewriting various header fields using Set-Field actions greatly increases the usefulness of an OpenFlow implementation. To aid integration with existing networks, supporting VLAN modification actions is strongly suggested.

- **Change-TTL ttl:** The various Change-TTL actions modify the values of the IPv4 TTL, IPv6 Hop Limit or MPLS TTL in the packet. While not strictly required, the actions shown in Table 1.3 greatly increase the usefulness of an OpenFlow implementation for implementing routing functions.

Table 1.2: Push/pop tag actions

Action	Associated data	Description
Push VLAN header	Ethertype	Push a new VLAN header onto the packet. The Ethertype is used as the Ethertype for the tag. Only Ethertype 0x8100 and 0x88a8 should be used.
Pop VLAN header	-	Pop the outer-most VLAN header from the packet.
Push MPLS header	Ethertype	Push a new MPLS shim header onto the packet. The Ethertype is used as the Ethertype for the tag. Only Ethertype 0x8847 and 0x8848 should be used.
Pop MPLS header	Ethertype	Pop the outer-most MPLS tag or shim header from the packet. The Ethertype is used as the Ethertype for the resulting packet (Ethertype for the MPLS payload).
Push PBB header	Ethertype	Push a new PBB service instance header (I-TAG TCI) onto the packet. The Ethertype is used as the Ethertype for the tag. Only Ethertype 0x88E7 should be used.
Pop PBB header	-	Pop the outer-most PBB service instance header (I-TAG TCI) from the packet.

Table 1.3: Change TTL actions

Action	Associated data	Description
Set MPLS TTL	8 bits: New MPLS TTL	Replace the existing MPLS TTL. Only applies to packets with an existing MPLS shim header
Decrement MPLS TTL	-	Decrement the MPLS TTL. Only applies to packets with an existing MPLS shim header.
Set IP TTL	8 bits: New IP TTL	Replace the existing IPv4 TTL or IPv6 Hop Limit and update the IP checksum. Only applies to IPv4 and IPv6 packets.
Decrement IP TTL	-	Decrement the IPv4 TTL or IPv6 Hop Limit field and update the IP checksum. Only applies to IPv4 and IPv6 packets.
Copy TTL outwards	-	Copy the TTL from next-to-outermost to outermost header with TTL. Copy can be IP-to-IP, MPLS-to-MPLS, or IP-to-MPLS.
Copy TTL inwards	-	Copy the TTL from outermost to next-to-outermost header with TTL. Copy can be IP-to-IP, MPLS-to-MPLS, or MPLS-to-IP.

## Action Set

An action set is associated with each packet. This set is empty by default. A flow entry can modify the action set using a *Write-Action* instruction or a *Clear-Action* instruction associated with a particular match. The action set is carried between flow tables. When the instruction set of a flow entry does not contain a *Goto-Table* instruction, pipeline processing stops and the actions in the action set of the packet are executed.

An action set contains a maximum of one action of each type. The set-field actions are identified by their field types,



therefore the action set contains a maximum of one set-field action for each field type (i.e. multiple fields can be set). The experimenter actions are identified by their experimenter-id and experimenter-type, therefore the action set may contain a maximum of one experimenter action for each combination of experimenter-id and experimenter-type. When an action of a specific type is added in the action set, if an action of the same type exists, it is overwritten by the later action. If multiple actions of the same type are required, e.g. pushing multiple MPLS labels or popping multiple MPLS labels, the Apply-Actions instruction should be used.

The actions in an action set are applied in the order specified below, regardless of the order that they were added to the set. If an action set contains a group action, the actions in the appropriate action bucket(s) of the group are also applied in the order specified below. The switch may support arbitrary action execution order through the list of actions of the Apply-Actions instruction.

1. **copy TTL inwards:** apply copy TTL inward actions to the packets
2. **pop:** apply all tag pop actions to the packet
3. **push-MPLS:** apply MPLS tag push action to the packet
4. **push-PBB:** apply PBB tag push action to the packet
5. **push-VLAN:** apply VLAN tag push action to the packet
6. **copy TTL outwards:** apply copy TTL outwards action to the packet
7. **decrement TTL:** apply decrement TTL action to the packet
8. **set:** apply all set-field actions to the packet
9. **qos:** apply all QoS actions, such as set queue to the packet
10. **group:** if a group action is specified, apply the actions of the relevant group bucket(s) in the order specified by this list.
11. **output:** if no group action is specified, forward the packet on the port specified by the output action

## Action List

The *Apply-Actions* instruction and the *Packet-out* message include a list of actions. The actions of a list of actions are executed in the order specified by the list, and are applied immediately to the packet. The execution of a list of actions starts with the first action in the list and each action is executed on the packet in sequence. The effect of those actions is cumulative, if the list of actions contains two Push VLAN actions, two VLAN headers are added to the packet. If the list of actions contains an output action, a copy of the packet is forwarded in its current state to the desired port. If the output action references a non-existent port, the copy of the packet is dropped. If the list of actions contains a group action, a copy of the packet in its current state is processed by the relevant group buckets.

After the execution of the list of actions in an Apply-Actions instruction, pipeline execution continues on the modified packet. The action set of the packet is unchanged by the execution of the list of actions.

### 1.4.6 Table-miss entry

Every flow table must support a table-miss flow entry to process table misses. The table-miss flow entry specifies how to process packets unmatched by other flow entries in the flow table, and may, for example, send packets to the

controller, drop packets or direct packets to a subsequent table. The table-miss flow entry is identified by its match and its priority, it wildcards all match fields (all fields omitted) and has the lowest priority (0).

If the table-miss flow entry does not exist, by default packets unmatched by flow entries are dropped (discarded). A switch configuration, for example using the OpenFlow Configuration Protocol, may override this default and specify another behaviour.

## 1.5 Group and Meter tables

### 1.5.1 Group Table

A group table consists of group entries. The ability for a flow entry to point to a group enables OpenFlow to represent additional methods of forwarding. Each group entry is identified by its group identifier and contains:

- **group identifier:** a 32 bit unsigned integer uniquely identifying the group on the OpenFlow switch.
- **group type:** to determine group semantics.
- **counters:** updated when packets are processed by a group.
- **action buckets:** an ordered list of action buckets, where each action bucket contains a set of actions to execute and associated parameters. The actions in a bucket are always applied as an action set.

A bucket typically contains actions that modify the packet and an output action that forwards it to a port. A bucket may also include a group action which invokes another group if the switch supports group chaining, in this case packet processing continues in the group invoked. A bucket with no actions is valid, a bucket with no output or group action effectively drops the clone of the packet associated with that bucket.

A switch is not required to support all group types, just those marked “Required” below. The controller can also query the switch about which of the “Optional” group type it supports:

- *Required:* **all:** Execute all buckets in the group. This group is used for multicast or broadcast forwarding. The packet is effectively cloned for each bucket; one packet is processed for each bucket of the group. If a bucket directs a packet explicitly out the ingress port, this packet clone is dropped. If the controller writer wants to forward out the ingress port, the group should include an extra bucket which includes an output action to the `OFPP_IN_PORT` reserved port.
- *Optional:* **select:** Execute one bucket in the group. Packets are processed by a single bucket in the group, based on a switch-computed selection algorithm (e.g. hash on some user-configured tuple or simple round robin). All configuration and state for the selection algorithm is external to OpenFlow. The selection algorithm should implement equal load sharing and can optionally be based on bucket weights. When a port specified in a bucket in a select group goes down, the switch may restrict bucket selection to the remaining set (those with forwarding actions to live ports) instead of dropping packets destined to that port. This behavior may reduce the disruption of a downed link or switch.
- *Required:* **indirect:** Execute the one defined bucket in this group. This group supports only a single bucket. Allows multiple flow entries or groups to point to a common group identifier, supporting faster, more efficient convergence. An indirect group is typically referenced by multiple flow entries, thereby allowing each of these entities to have a centralized action that can be easily updated (e.g. next hops for IP forwarding). This group type is effectively identical to an all group with one bucket.

- **Optional: fast failover:** Execute the first live bucket. Each action bucket is associated with a specific port and/or group that controls its liveness. The buckets are evaluated in the order defined by the group, and the first bucket which is associated with a live port/group is selected. This group type enables the switch to change forwarding without requiring a round trip to the controller. If no buckets are live, packets are dropped.

## 1.5.2 Meter Table

A meter table consists of meter entries, defining per-flow meters. Per-flow meters enable OpenFlow to implement various simple QoS operations, such as rate-limiting, and can be combined with per-port queues to implement complex QoS frameworks, such as DiffServ.

A meter measures the rate of packets assigned to it and enables controlling the rate of those packets. Meters are attached directly to flow entries. Any flow entry can specify a meter in its instruction set: the meter measures and controls the rate of the aggregate of all flow entries to which it is attached. Multiple meters can be used in the same table, but in an exclusive way (disjoint set of flow entries). Multiple meters can be used on the same set of packets by using them in successive flow tables.

Each meter entry is identified by its meter identifier and contains:

- **meter identifier:** a 32 bit unsigned integer uniquely identifying the meter
- **meter bands:** an unordered list of meter bands, where each meter band specifies the rate of the band and the way to process the packet
- **counters:** updated when packets are processed by a meter

## 1.6 OpenFlow Switch Protocol

The OpenFlow channel is the interface that connects each OpenFlow Logical Switch to an OpenFlow controller. Through this interface, the controller configures and manages the forwarding plane of a switch, receives events from the switch, and sends packets out the switch. The Control Channel of a switch may support a single OpenFlow channel with a single controller, or multiple OpenFlow channels enabling multiple controllers to share management of the switch.

Between the datapath and the OpenFlow channel, the interface is implementation-specific, however all OpenFlow channel messages must be formatted according to the **OpenFlow Switch Protocol**. The OpenFlow channel is usually encrypted using TLS, but may be run directly over TCP.

The OpenFlow switch protocol supports three message types, controller-to-switch, asynchronous, and symmetric, each with multiple sub-types.

- **Controller-to-switch messages** are initiated by the controller and used to directly manage or inspect the state of the switch.
- **Asynchronous messages** are initiated by the switch and used to update the controller about network events and changes to the switch state.
- **Symmetric messages** are initiated by either the switch or the controller and sent without solicitation.

## 1.6.1 Controller-to-Switch messages

Controller/switch messages are initiated by the controller and may or may not require a response from the switch.

- **Features:** The controller may request the identity and the basic capabilities of a switch by sending a features request; the switch must respond with a features reply that specifies the identity and basic capabilities of the switch. This is commonly performed upon establishment of the OpenFlow channel.
- **Configuration:** The controller is able to set and query configuration parameters in the switch. The switch only responds to a query from the controller.
- **Modify-State:** Modify-State messages are sent by the controller to manage state on the switches. Their primary purpose is to add, delete and modify flow/group entries in the OpenFlow tables and to set switch port properties.
- **Read-State:** Read-State messages are used by the controller to collect various information from the switch, such as current configuration, statistics and capabilities.
- **Packet-out:** These are used by the controller to send packets out of a specified port on the switch, and to forward packets received via Packet-in messages. Packet-out messages must contain a full packet or a buffer ID referencing a packet stored in the switch. The message must also contain a list of actions to be applied in the order they are specified; an empty list of actions drops the packet.
- **Barrier:** Barrier request/reply messages are used by the controller to ensure message dependencies have been met or to receive notifications for completed operations.
- **Role-Request:** Role-Request messages are used by the controller to set the role of its OpenFlow channel, or query that role. This is mostly useful when the switch connects to multiple controllers.
- **Asynchronous-Configuration:** The Asynchronous-Configuration messages are used by the controller to set a filter on the asynchronous messages that it wants to receive on its OpenFlow channel, or to query that filter. This is mostly useful when the switch connects to multiple controllers and commonly performed upon establishment of the OpenFlow channel.

## 1.6.2 Asynchronous messages

Asynchronous messages are sent without a controller soliciting them from a switch. Switches send asynchronous messages to controllers to denote a packet arrival, switch state change, or error. The four main asynchronous message types are described below.

- **Packet-in:** Transfer the control of a packet to the controller. For all packets forwarded to the CONTROLLER reserved port using a flow entry or the table-miss flow entry, a packet-in event is always sent to controllers. Other processing, such as TTL checking, may also generate packet-in events to send packets to the controller.

Packet-in events can be configured to buffer packets. For packet-in generated by an output action in a flow entry or group bucket, the packet buffering can be specified individually in the output action itself, for other packet-in it can be configured in the switch configuration. If the packet-in event is configured to buffer packets and the switch has sufficient memory to buffer them, the packet-in event contains only some fraction of the packet header and a buffer ID to be used by a controller when it is ready for the switch to forward the packet. Switches that do not support internal buffering, are configured to not buffer packets for the packet-in event, or have run out of internal buffering, must send the full packet to controllers as part of the event.

- **Flow-Removed:** Inform the controller about the removal of a flow entry from a flow table. Flow-Removed messages are only sent for flow entries with the `OFPPF_SEND_FLOW_REM` flag set. They are generated as the result of a controller flow delete request or the switch flow expiry process when one of the flow timeouts is exceeded.
- **Port-status:** Inform the controller of a change on a port. The switch is expected to send port-status messages to controllers as port configuration or port state changes. These events include change in port configuration events, for example if it was brought down directly by a user, and port state change events, for example if the link went down.
- **Error:** The switch is able to notify controllers of problems using error messages.

### 1.6.3 Symmetric messages

Symmetric messages are sent without solicitation, in either direction.

- **Hello:** Hello messages are exchanged between the switch and controller upon connection startup.
- **Echo:** Echo request/reply messages can be sent from either the switch or the controller, and must return an echo reply. They are mainly used to verify the liveness of a controller-switch connection, and may as well be used to measure its latency or bandwidth.
- **Experimenter:** Experimenter messages provide a standard way for OpenFlow switches to offer additional functionality within the OpenFlow message type space. This is a staging area for features meant for future OpenFlow revisions.

### 1.6.4 OpenFlow Channel Connections

The switch must be able to establish communication with a controller at a user-configurable (but otherwise fixed) IP address, using a user-specified port. If the switch knows the IP address of the controller, the switch initiates a standard TLS or TCP connection to the controller. Traffic to and from the OpenFlow channel is not run through the OpenFlow pipeline. Therefore, the switch must identify incoming traffic as local before checking it against the flow tables.

When an OpenFlow connection is first established, each side of the connection must immediately send an `OFPT_HELLO` message with the version field set to the highest OpenFlow protocol version supported by the sender. Upon receipt of this message, the recipient may calculate the OpenFlow protocol version to be used as the smaller of the version number that it sent and the one that it received. If the negotiated version is supported by the recipient, then the connection proceeds. Otherwise, the recipient must reply with an `OFPT_ERROR` message with a type field of `OFPET_HELLO_FAILED`, a code field of `OFPHFC_COMPATIBLE`, and optionally an ASCII string explaining the situation in data, and then terminate the connection.

After the version has been negotiated, the controller issues a `OFPT_FEATURES_REQUEST` message which is answered with a `OFPT_FEATURES_REPLY` by the switch. This reply message contains the number of tables and buffers (packets that can be buffered at once) supported by the switch and its datapath ID. Additionally, can contain a list of the capabilities of the switch, which can be:

- Flow statistics.
- Table statistics.
- Port statistics.
- Group statistics.

- Can reassemble IP fragments.
- Queue statistics.
- Switch will block looping ports.

This features message exchange is what is called OpenFlow **Handshake**. After the controller has received the message `OFPT_FEATURES_REPLY`, the controller usually queries the switch about its configuration parameters.

The controller is able to set and query configuration parameters in the switch with the `OFPT_SET_CONFIG` and `OFPT_GET_CONFIG_REQUEST` messages, respectively. The switch responds to a configuration request with an `OFPT_GET_CONFIG_REPLY` message (but it does not reply to a request to set the configuration).

Configuration messages carry the information inside a set of configuration flags. These flags indicate whether IP fragments should be treated normally, dropped, or reassembled. “Normal” handling of fragments means that an attempt should be made to pass the fragments through the OpenFlow tables.

## 1.6.5 Flow Table Modification Messages

Flow table modification messages can have the following types:

- **Add:** Add a new flow (`OFPPC_ADD`). If the `OFPPF_CHECK_OVERLAP` flag is set, the switch must first check for any overlapping entries in the requested table. Two flow entries overlap if a single packet may match both, and both entries have the same priority. If an overlap conflict exists between an existing flow entry and the add request, the switch must refuse the addition and respond with an *ofp\_error\_msg* with `OFPET_FLOW_MOD_FAILED` type and `OFPPMFC_OVERLAP` code.

For non-overlapping add requests, or those with no overlap checking, the switch must insert the flow entry in the requested table. If a flow entry with identical match fields and priority already resides in the requested table, then that entry, including its duration, must be cleared from the table, and the new flow entry added. If the `OFPPF_RESET_COUNTS` flag is set, the flow entry counters must be cleared, otherwise they should be copied from the replaced flow entry. No flow-removed message is generated for the flow entry eliminated as part of an add request; if the controller wants a flow-removed message it should explicitly send a delete request for the old flow entry prior to adding the new one.

- **Modify:** For modify requests (`OFPPC_MODIFY` or `OFPPC_MODIFY_STRICT`), if a matching entry exists in the table, the instructions field of this entry is updated with the value from the request, whereas its cookie, idle\_timeout, hard\_timeout, flags, counters and duration fields are left unchanged.

If the `OFPPF_RESET_COUNTS` flag is set, the flow entry counters must be cleared. For modify requests, if no flow entry currently residing in the requested table matches the request, no error is recorded, and no flow table modification occurs.

- **Delete:** For delete requests (`OFPPC_DELETE` or `OFPPC_DELETE_STRICT`), if a matching entry exists in the table, it must be deleted, and if the entry has the `OFPPF_SEND_FLOW_REM` flag set, it should generate a flow removed message. For delete requests, if no flow entry currently residing in the requested table matches the request, no error is recorded, and no flow table modification occurs.

Modify and delete flow mod commands have non-strict versions and strict versions:

- In the strict versions, the set of match fields, all match fields, including their masks, and the priority, are strictly matched against the entry, and only an identical flow entry is modified or removed.

For example, if a message to remove entries is sent that has no match fields included, the `OFPPC_DELETE` command would delete all flow entries from the tables, while the `OFPPC_DELETE_STRICT` command would

only delete a flow entry that applies to all packets at the specified priority. For non-strict modify and delete commands, all flow entries that match the flow mod description are modified or removed.

- In the non-strict versions, a match will occur when a flow entry exactly matches or is more specific than the description in the *flow\_mod* command; in the *flow\_mod* the missing match fields are wildcarded, field masks are active, and other flow mod fields such as priority are ignored.

For example, if a `OFPPC_DELETE` command says to delete all flow entries with a destination port of 80, then a flow entry that wildcards all match fields will not be deleted. However, a `OFPPC_DELETE` command that wildcards all match fields will delete an entry that matches all port 80 traffic.

Modify and delete commands can also be filtered by cookie value, if the `cookie_mask` field contains a value other than 0.

## 1.6.6 OpenFlow header

The OpenFlow protocol is implemented using OpenFlow messages transmitted over the OpenFlow channel. Each message type is described by a specific structure, which starts with the common *OpenFlow header*. Each structure defines the order in which information is included in the message and may contain other structures, values, enumerations or bitmasks.

Each OpenFlow message begins with the OpenFlow header:

```
/* Header on all OpenFlow packets. */
struct ofp_header {
    uint8_t version; /* OFP_VERSION. */
    uint8_t type; /* One of the OFPT_ constants. */
    uint16_t length; /* Length including this ofp_header. */
    uint32_t xid; /* Transaction id associated with this packet.
                 Replies use the same id as was in the request
                 to facilitate pairing. */
};
OFP_ASSERT(sizeof(struct ofp_header) == 8);
```

The `version` field specifies the OpenFlow switch protocol version being used. The most significant bit in the version field is reserved and must be set to 0. The 7 lower bits indicate the revision number of the protocol. The version of the protocol described in this chapter is 1.3, and its *ofp\_version* is 0x04

The `length` field indicates the total length of the message, so no additional framing is used to distinguish one frame from the next.

The `type` field specifies the type of message. It can have the following values:

```
enum ofp_type {
    /* Immutable messages. */
    OFPT_HELLO = 0, /* Symmetric message */
    OFPT_ERROR = 1, /* Symmetric message */
    OFPT_ECHO_REQUEST = 2, /* Symmetric message */
    OFPT_ECHO_REPLY = 3, /* Symmetric message */
    OFPT_EXPERIMENTER = 4, /* Symmetric message */
};
```

```
/* Switch configuration messages. */
OFPT_FEATURES_REQUEST = 5, /* Controller/switch message */
OFPT_FEATURES_REPLY = 6, /* Controller/switch message */
OFPT_GET_CONFIG_REQUEST = 7, /* Controller/switch message */
OFPT_GET_CONFIG_REPLY = 8, /* Controller/switch message */
OFPT_SET_CONFIG = 9, /* Controller/switch message */

/* Asynchronous messages. */
OFPT_PACKET_IN = 10, /* Async message */
OFPT_FLOW_REMOVED = 11, /* Async message */
OFPT_PORT_STATUS = 12, /* Async message */

/* Controller command messages. */
OFPT_PACKET_OUT = 13, /* Controller/switch message */
OFPT_FLOW_MOD = 14, /* Controller/switch message */
OFPT_GROUP_MOD = 15, /* Controller/switch message */
OFPT_PORT_MOD = 16, /* Controller/switch message */
OFPT_TABLE_MOD = 17, /* Controller/switch message */

/* Multipart messages. */
OFPT_MULTIPART_REQUEST = 18, /* Controller/switch message */
OFPT_MULTIPART_REPLY = 19, /* Controller/switch message */

/* Barrier messages. */
OFPT_BARRIER_REQUEST = 20, /* Controller/switch message */
OFPT_BARRIER_REPLY = 21, /* Controller/switch message */

/* Queue Configuration messages. */
OFPT_QUEUE_GET_CONFIG_REQUEST = 22, /* Controller/switch message */
OFPT_QUEUE_GET_CONFIG_REPLY = 23, /* Controller/switch message */
/* Controller role change request messages. */

OFPT_ROLE_REQUEST = 24, /* Controller/switch message */
OFPT_ROLE_REPLY = 25, /* Controller/switch message */
/* Asynchronous message configuration. */

OFPT_GET_ASYNC_REQUEST = 26, /* Controller/switch message */
OFPT_GET_ASYNC_REPLY = 27, /* Controller/switch message */
OFPT_SET_ASYNC = 28, /* Controller/switch message */

/* Meters and rate limiters configuration messages. */
OFPT_METER_MOD = 29, /* Controller/switch message */
};
```



# Chapter 2

## Open vSwitch

### 2.1 Introduction

Open vSwitch (OVS) is a multilayer software switch licensed under the open source Apache 2 license. The goal of its developers is to implement a production quality switch platform that supports standard management interfaces and opens the forwarding functions to programmatic extension and control.

Open vSwitch is well suited to function as a virtual switch in VM environments. In addition to exposing standard control and visibility interfaces to the virtual networking layer, it was designed to support distribution across multiple physical servers. Open vSwitch supports multiple Linux-based virtualization technologies including Xen/XenServer, KVM, and VirtualBox.

The bulk of the code is written in platform-independent C and is easily ported to other environments. The current release of Open vSwitch, as stated in their GitHub repository [3], supports the following features:

- Standard 802.1Q VLAN model with trunk and access ports
- NIC bonding with or without LACP on upstream switch
- NetFlow, sFlow(R), and mirroring for increased visibility
- QoS (Quality of Service) configuration, plus policing
- Geneve, GRE, GRE over IPSEC, VXLAN, and LISP tunneling
- 802.1ag connectivity fault management
- OpenFlow 1.3 plus numerous extensions
- Transactional configuration database with C and Python bindings
- High-performance forwarding using a Linux kernel module

#### 2.1.1 Open vSwitch architecture

There are three main components in Open vSwitch, according to the documentation provided by its developers [9]: the database server (ovsdb-server), the daemon (ovs-vswitchd), and the kernel module.

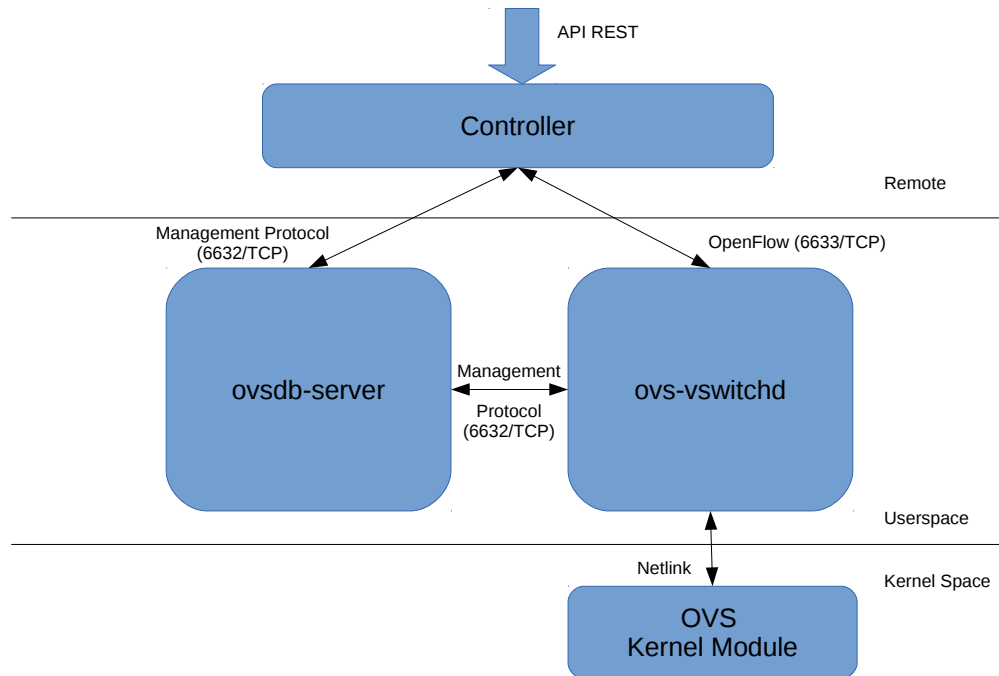


Figure 2.1: OVS architecture

- The controller represented in the Figure 2.1 can be an OpenFlow Controller such as Ryu, or an OVS database manager. All of the components of Open vSwitch are configurable remotely.
- ovsdb-server is the component that has the configuration database, it holds information that will survive a reboot, for instance, configurations for bridges and interfaces.
- ovs-vswitchd is the core part of Open vSwitch, it does all the handling of flow setups.
- The objective of the kernel module is to improve the performance of OVS.

## 2.2 The Kernel Module

The OVS Kernel Module (see [2]) has a simple design, aiming to be portable to several systems. It allows flexible userspace control over flow-level packet processing. Flows allow to implement a plain Ethernet switch, network device bonding, VLAN processing, network access control, flow-based network control and many other programmable applications.

The kernel module can be used to implement multiple "datapaths". A datapath is like a physical switch (bridge). Each datapath can have multiple "vports" (analogous to ports within a switch/bridge). Each datapath also has associated with it a "flow table" that userspace populates with "flows".

A flow has matching fields, for example the fields of the packet being processed or the port in which the packet was received. Each flow has also a set of instructions to do when the flow is hit. These instructions may say that a set of actions has to be executed. Common actions are dropping the packet or forwarding the packet to another vport. Later we provide some examples about this.

When a packet arrives on a vport, the kernel module processes it by extracting the matching fields and by performing a look up in the flow table. If there is a matching flow, it executes the associated instructions and actions. If there is no match, it queues the packet to userspace for processing. As part of its processing, the userspace part of the openvswitch

(ovs-vswitchd) will likely set up a flow to handle further packets of the same type entirely in-kernel.

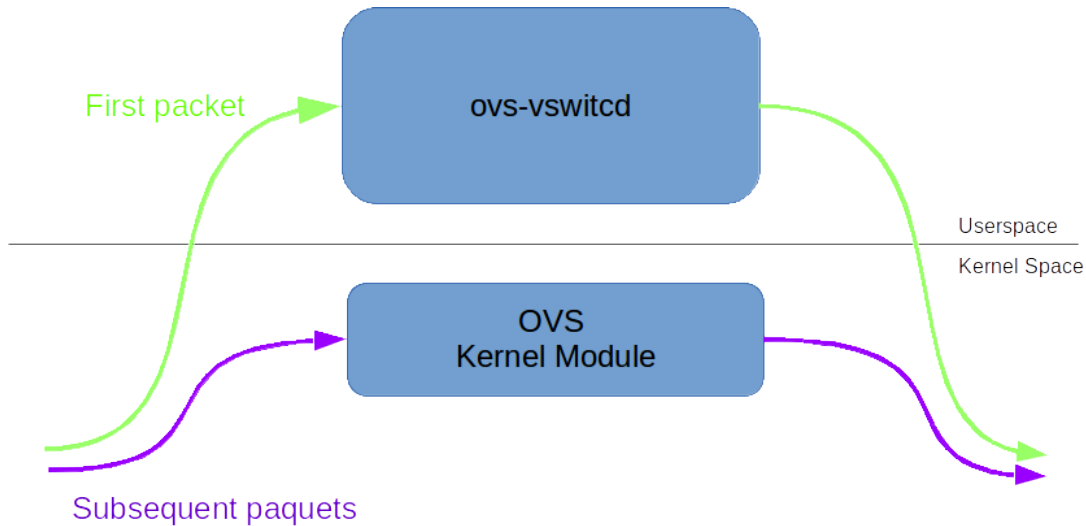


Figure 2.2: Open vSwitch packet forwarding

As it can be seen in Figure 2.2, the decision about how to process a packet is made in userspace, so the first packet of a new flow goes to ovs-vswitchd, while the following packets will hit a cached entry in the kernel.

### 2.2.1 Datapath flows

Open vSwitch uses different kinds of flows for different purposes. OpenFlow flows are the most important kind of flow. OpenFlow controllers use these flows to define a switch's policy. OpenFlow flows support wildcards, priorities, and multiple tables. When in-band control is in use, Open vSwitch sets up a few "hidden" flows, with priority higher than a controller or the user can configure, that are not visible via OpenFlow.

The Open vSwitch software switch implementation uses a second kind of flow internally. These flows, called "datapath" or "kernel" flows, do not support priorities and comprise only a single table, which makes them suitable for caching. OpenFlow flows and datapath flows also support different actions and number ports differently. Datapath flows are an implementation detail that is subject to change in future versions of Open vSwitch. Even with the current version of Open vSwitch, hardware switch implementations do not necessarily use this architecture.

The following field assignments describe how the current datapath flow matches a packet. If any of these assignments is omitted from the flow syntax, the field is treated as a wildcard: Thus, if all of them are omitted, the resulting flow matches all packets. The string \* or ANY may be specified a value to explicitly mark any of these fields as a wildcard.

- `in_port=port_no`: Matches physical port `port_no`. Switch ports are numbered as displayed by `dpctl show`.
- `dl_vlan=vlan`: Matches IEEE 802.1q virtual LAN tag `vlan`. Specify `0xffff` as `vlan` to match packets that are not tagged with a virtual LAN; otherwise, specify a number between 0 and 4095, inclusive, as the 12-bit VLAN ID to match.
- `dl_src=mac`: Matches Ethernet source address `mac`, which should be specified as 6 pairs of hexadecimal digits delimited by colons, e.g. `00:0A:E4:25:6B:B0`.

- `dl_dst=mac`: Matches Ethernet destination address `mac`.
- `dl_type=ethertype`: Matches Ethernet protocol type `ethertype`, which should be specified as a integer between 0 and 65535, inclusive, either in decimal or as a hexadecimal number prefixed by 0x, e.g. 0x0806 to match ARP packets.
- `nw_src=ip[/netmask]`: Matches IPv4 source address `ip`, which should be specified as an IP address or host name, e.g. 192.168.1.1 or www.example.com. The optional netmask allows matching only on an IPv4 address prefix. It may be specified as a dotted quad (e.g. 192.168.1.0/255.255.255.0) or as a count of bits (e.g. 192.168.1.0/24).
- `nw_dst=ip[/netmask]`: Matches IPv4 destination address `ip`.
- `nw_proto=proto`: Matches IP protocol type `proto`, which should be specified as a decimal number between 0 and 255, inclusive, e.g. 6 to match TCP packets.
- `nw_tos=tos/dscp`: Matches ToS/DSCP (only 6-bits, not modify reserved 2-bits for future use) field of IPv4 header `tos/dscp`, which should be specified as a decimal number between 0 and 255, inclusive.
- `tp_src=port`: Matches UDP or TCP source port `port`, which should be specified as a decimal number between 0 and 65535, inclusive, e.g. 80 to match packets originating from a HTTP server.
- `tp_dst=port`: Matches UDP or TCP destination port `port`.
- `icmp_type=type`: Matches ICMP message with `type`, which should be specified as a decimal number between 0 and 255, inclusive.
- `icmp_code=code`: Matches ICMP messages with `code`.

The actions that the kernel module can perform when a match occurs are explained in section 2.3.3.

It is worth to mention that users and controllers directly control only the OpenFlow flow table. Open vSwitch manages the datapath flow table itself, so users should not normally be concerned with it.

## 2.3 Components and tools of Open vSwitch

The OVS distribution comes with two main components:

- `ovs-vswitchd`, a daemon that implements the switch, along with a companion Linux kernel module for flow-based switching.
- `ovsdb-server`, a lightweight database server that `ovs-vswitchd` queries to obtain its configuration.

OVS also provides some tools:

- `ovs-dpctl`, a tool for show and configure datapath flows.
- `ovs-vsctl`, a high-level utility for querying and updating the configuration of `ovs-vswitchd`. `ovs-dpctl` controls the Fast Path and `ovs-vsctl` configures the Slow Path. In general, you'll probably want to use `ovs-vsctl` to handle configuration. `ovs-dpctl` is mostly useful for debugging purposes. Since the datapath is essentially a cache for the traffic that is occurring on the network, you can use the "`ovs-dpctl dump-flows`" command to see what traffic is being processed and what actions are occurring currently.

- ovs-appctl, a utility that sends commands to running OVS daemons.
- ovs-ofctl, a utility for querying and controlling OpenFlow module of OVS switches.
- ovs-pki, a utility for creating and managing the public-key infrastructure for OpenFlow switches.
- ovs-testcontroller, a simple OpenFlow controller that may be useful for testing (though not for production).
- ovsdb-tool, a command-line tool for managing OVS database files.
- ovsdb-client, a command-line client for interacting with a running ovsdb-server process.

### 2.3.1 ovs-vswitchd

ovs-vswitchd is a daemon that manages and controls any number of OVS switches on the local machine. It is the core component in the system:

- Communicates with SDN controllers using OpenFlow.
- Communicates with ovsdb-server using OVSDB protocol.
- Communicates its kernel module over netlink.
- Communicates with the hosting system through netdev interface.

Packet classifier supports efficient flow lookup with wildcards and "explodes" these (possibly) wildcard rules for fast processing by the datapath. It retrieves its configuration from database at startup. It sets up Open vSwitch datapaths and then operates switching across each bridge described in its configuration files. As the database changes, ovs-vswitchd automatically updates its configuration to match. Also, this daemon checks datapath flow counters to handle flow expiration and stats requests.

Only a single instance of ovs-vswitchd is intended to run at a time. A single ovs-vswitchd can manage any number of switch instances, up to the maximum number of supported Open vSwitch datapaths. ovs-vswitchd does all the necessary management of Open vSwitch datapaths itself. Thus, external tools, such ovs-dpctl, are not needed for managing datapaths. In fact, configuring datapath flows with ovs-dpctl when ovs-vswitchd is running can interfere with its operation. Still, ovs-dpctl may still be useful for diagnostics (use only show commands).

### 2.3.2 ovsdb-server

The ovsdb-server program provides RPC (Remote Procedure Call) interfaces to one or more Open vSwitch databases (OVSDBs). It supports JSON-RPC client connections over TCP/IP or Unix domain sockets. These databases hold switch level configuration such as bridge, interface and tunnel definitions, OVSDB managers and OpenFlow controller addresses.

This configuration is stored on disk and survives a reboot. The implementation of this database is log-based, this means that it does not only stores the states of the database but also stores a change log where all the changes are saved.

The server interacts with OVSDB managers and ovs-vswitchd using the OVSDB protocol, which is intended to allow programmatic access to the Open vSwitch database.

## Database schema

A database with this schema holds the configuration for one Open vSwitch daemon. The top-level configuration for the daemon is the Open\_vSwitch table, which must have exactly one record. Records in other tables are significant only when they can be reached directly or indirectly from the Open\_vSwitch table. Records that are not reachable from the Open\_vSwitch table are automatically deleted from the database, except for records in a few distinguished "root set" tables.

## Table summary

The following list summarizes the purpose of each of the tables in the Open\_vSwitch database

- Open\_vSwitch: Open vSwitch configuration.
- Bridge: Bridge configuration.
- Port: Port configuration.
- Interface: One physical network device in a Port.
- Flow\_Table: OpenFlow table configuration
- QoS: Quality of Service configuration
- Queue: QoS output queue.
- Mirror: Port mirroring.
- Controller: OpenFlow controller configuration.
- Manager: OVSDDB management connection.
- NetFlow: NetFlow configuration.
- SSL: SSL configuration.
- sFlow: sFlow configuration.
- IPFIX: IPFIX configuration.
- Flow\_Sample\_Collector\_Set: Flow\_Sample\_Collector\_Set configuration.

However, we don't usually interact with all of this tables, but with a reduced number of them, which we'll call Core Tables. In Figure 2.3 we can see the relationship between these core tables.

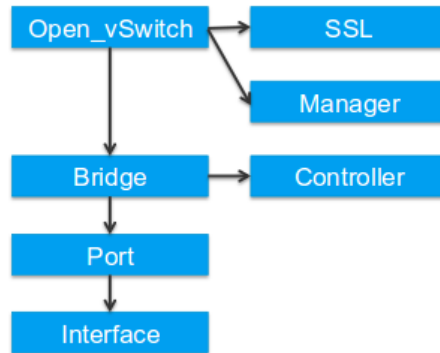


Figure 2.3: Core Table relationship

### ovsdb-tool

The ovsdb-tool program is a command-line tool for managing Open vSwitch database files. It does not interact directly with running Open vSwitch database servers.

For instance, we can use this program to see the change log of the database with:

```
ovsdb-tool show-log [-mmm] <file>
```

### ovsdb-client

The ovsdb-client program is a command-line client for interacting with a running ovsdb-server process. Each command connects to an OVSDDB server, which is `unix:/var/run/openvswitch/db.sock` by default.

### 2.3.3 ovs-dpctl

The ovs-dpctl program can create, modify, and delete Open vSwitch datapaths. A single machine may host any number of datapaths. This program speaks directly to the kernel module.

A newly created datapath is associated with only one network device, a virtual network device sometimes called the datapath's "local port". A newly created datapath is not, however, associated with any of the host's other network devices.

If `ovs-vswitchd` is in use, `ovs-vsctl` must be used instead of `ovs-dpctl`.

The ovs-dpctl tool can write datapath flows to the kernel module, using the fields described in section 2.2.1 to build the desired match. For this purpose, the program has the `add-flow` and `add-flows` commands, which have a field `actions=target[,target...]`

This field specifies a comma-separated list of actions to take on a packet when the flow entry matches. The target may be a decimal port number designating the physical port on which to output the packet, or one of the following keywords:

Keyword	Action
output:port	Outputs the packet on the port specified by port.
enqueue:port:q-id	Enqueue the packet to the queue specified by q-id on the port specified by port.
normal	Subjects the packet to the device's normal L2/L3 processing.
flood	Outputs the packet on all switch physical ports other than the port on which it was received.
all	Outputs the packet on all switch physical ports other than the port on which it was received.
controller:max_len	Sends the packet to the OpenFlow controller as a "packet in" message. If max_len specifies the maximum number of bytes that should be sent.
local	Outputs the packet on the "local port", which corresponds to the ofn network device.
mod_vlan_vid:vlan_vid	Modifies the VLAN id on a packet. The VLAN tag is added or modified as necessary to match the value specified.
mod_vlan_pcp:vlan_pcp	Modifies the VLAN priority on a packet. The VLAN tag is added or modified as necessary to match the value specified.
mod_dl_dst:dst_mac	Modifies the destination mac address on a packet, e.g., actions=mod_dl_dst:12:34:56:78:9a:bc
mod_dl_src:src_mac	Modifies the source mac address on a packet, e.g., actions=mod_dl_src:12:34:56:78:9a:bc
mod_nw_tos:tos/dscp	Modifies the ToS/DSCP (only 6-bits, not modify reserved 2-bits for future use) field of IPv4 header on a packet.
strip_vlan	Strips the VLAN tag from a packet if it is present.

### 2.3.4 ovs-vsctl

The ovs-vsctl program configures ovs-vswitchd by providing a high-level interface to its configuration database.

ovs-vsctl connects to an ovsdb-server process that maintains an Open vSwitch configuration database. Using this connection, it queries and possibly applies changes to the database, depending on the supplied commands. Then, if it applied any changes, by default it waits until ovs-vswitchd has finished reconfiguring itself before it exits. ovs-vsctl can perform any number of commands in a single run, implemented as a single atomic transaction against the database.

As an example of this high level interface, here is a list of some of the commands that can be used within this program:

- ovs-vsctl add-br <bridge>
- ovs-vsctl list-br
- ovs-vsctl add-port <bridge><port>
- ovs-vsctl list-ports <bridge>
- ovs-vsctl get-manager <bridge>
- ovs-vsctl get-controller <bridge>
- ovs-vsctl list <table>

### 2.3.5 ovs-appctl

Open vSwitch daemons accept certain commands at runtime to control their behavior and query their settings. Every daemon accepts a common set of commands and some daemons may support additional commands. In particular, ovs-vswitchd accepts a number of additional commands documented in its man page.



The `ovs-appctl` program provides a simple way to invoke these commands. The command to be sent is specified on `ovs-appctl`'s command line as non-option arguments. `ovs-appctl` sends the command and prints the daemon's response on standard output.

By default, `ovs-appctl` communicates with `ovs-vswitchd`, but this can be changed using the `target` parameter (`-t <target>`). All daemons support the following commands:

- `help` – Lists the commands supported by the target.
- `version` – Displays the version and compilation date of the target.
- `vlog/list` – List the known logging modules and their current levels.
- `vlog/set [spec]` – Sets logging levels.

These features make `ovs-appctl` an excellent tool for debugging and logging.

### 2.3.6 `ovs-ofctl`

The `ovs-ofctl` program is a command line tool for monitoring and administering OpenFlow switches. It can also show the current state of an OpenFlow switch, including features, configuration, and table entries. It should work with any OpenFlow switch, not just Open vSwitch. It's the tool used to add, delete and dump flows. An example of its usage can be found on the scripts of section 2.5.

### 2.3.7 `ovs-pki`

The `ovs-pki` program sets up and manages a public key infrastructure for use with OpenFlow. It is intended to be a simple interface for organizations that do not have an established public key infrastructure. Other PKI tools can substitute for or supplement the use of `ovs-pki`.

### 2.3.8 Further information

For any further information on any of the previous programs, please refer to their respective man pages, which can be found in <http://openvswitch.org/support/dist-docs/>

## 2.4 Practical Examples

### 2.4.1 Install

To install OVS in Ubuntu:

```
$ sudo apt-get install openvswitch-datapath-source \
openvswitch-common openvswitch-switch
```

## 2.4.2 Basic Commands

### Create a Bridge

To create a bridge called `br0` just type:

```
$ sudo ovs-vsctl add-br br0
```

You can avoid errors if the bridge already exists with the option “may exist”:

```
$ sudo ovs-vsctl --may-exist add-br br0
```

### Add Ports

To add ports to the bridge (again you can use the may exist option):

```
$ sudo ovs-vsctl --may-exist add-port br0 eth1
```

### Delete Bridge

To delete an existing bridge we just use the `del-br` option (you can use it with the `if exists` option):

```
$ sudo ovs-vsctl --if-exists del-br br0
```

To delete an existing bridge we just use the `del-br` option:

```
$ sudo ovs-vsctl del-br br0
```

An easy way to solve any problem if something goes wrong while configuring Open vSwitch is to stop `openvswitch` service, then delete the file `/etc/openvswitch/conf.db` and finally start the service.

### Show Config

To show the configuration of existing OVS:

```
$ sudo ovs-vsctl show
```

## 2.4.3 Basic Openflow

### Add Flows

A flow entry primarily contains a set of `field=value` entries and action entry. The `field=value` entries are used to identify the incoming packet and the actions tells the bridge with what to do with the matching traffic.

Table 0 is where packets enter the switch. This table can be used to discard packets that for one reason or another are invalid. For example, packets with a multicast MAC source address are not valid, so we can add a flow to drop them at ingress to the switch with the following flow:

```
$ sudo ovs-ofctl add-flow br0 \  
"table=0, dl_src=01:00:00:00:00:00/01:00:00:00:00:00, actions=drop"
```

Another example to drop all ICMP traffic (nw\_proto=1) from 192.168.7.189 to 192.168.1.100 to the switch (LOCAL port):

```
$ sudo ovs-ofctl add-flow br0 \  
"in_port=LOCAL,table=0,idle_timeout=60,ip,hard_timeout=60, nw_proto=1, \  
nw_dst=192.168.1.100, nw_src=192.168.7.189,actions=drop"
```

## Test

OVS has a few specialized testing tools. The most powerful of these tools is "ofproto/trace". Given a switch and the specification of a flow, "ofproto/trace" shows, step-by-step, how such a flow would be treated as it goes through the switch. For example:

```
$ ovs-appctl ofproto/trace br0 in_port=1,dl_dst=01:80:c2:00:00:05 -generate \  
Flow: metadata=0,in_port=1,vlan_tci=0x0000,dl_src=00:00:00:00:00:00, \  
      dl_dst=01:80:c2:00:00:05,dl_type=0x0000 \  
Rule: table=0 cookie=0 dl_dst=01:80:c2:00:00:00/ff:ff:ff:ff:ff:ff \  
OpenFlow actions=drop \  
Final flow: unchanged \  
Datapath actions: drop
```

## Multiple Tables

To go to another table we have several ways. The action goto\_table:<num> sends the openflow pipeline processing to table num. On the other hand, the action resubmit([port],[table]) makes re-searching the OpenFlow flow table with the in\_port field replaced by port. Or research the table whose number is specified by table. Examples:

```
$ sudo ovs-ofctl add-flow br0 \  
"table=1, priority=99, in_port=1, actions=goto_table:2" \  
$ sudo ovs-ofctl add-flow br0 \  
"table=1, priority=99, in_port=1, actions=resubmit(,2)"
```

## Metadata

Metadata can be passed between tables and it can be used as a match field in the form metadata=value[/mask]. For example:

```
$ sudo ovs-ofctl add-flow br0 "table=1, metadata=0xffffffff00000000, actions=goto_table:2"
```

To write metadata you can use the action write\_metadata:

```
$ sudo ovs-ofctl add-flow br0 "in_port=1 actions=write_metadata:0xcafecafe00000000,resubmit(,1)"
$ sudo ovs-ofctl add-flow br0 <match conditions>,actions=write_metadata:3,goto_table:7"
```

## Show OF Configuration

The flows assigned to the bridge can be listed with the following command:

```
$ sudo ovs-ofctl dump-flows br0 \
cookie=0x0, duration=14.604s, table=0, n_packets=61, n_bytes=7418, idle_timeout=10,
hard_timeout=30,tcp, vlan_tci=0x0000, dl_src=78:2b:cb:4b:db:c5, dl_dst=00:21:9b:8e:36:62,
nw_src=192.168.7.189, nw_dst=192.168.1.150, nw_tos=0, tp_src=22, tp_dst=60221, actions=output:1
```

The above flow should be self-explanatory. If the traffic comes in from src mac address 78:2b:cb:4b:db:c5, destination mac address 00:21:9b:8e:36:62, traffic is tcp traffic, src ip=192.168.7.189, dest ip=192.168.1.150, TCP source port 22, tcp destination port 60221 forward the packet to port 1 (actions:1).

You can also view the OF flows of a particular table:

```
$ sudo ovs-ofctl dump-flows br0 table=10
```

On the other hand, the OF ports configured in the bridge can be seen with the following command:

```
$ sudo ovs-ofctl show br0
OFPT_FEATURES_REPLY (xid=0x1): ver:0x1,
dpid:0000782bcb4bdbc5
n_tables:255, n_buffers:256
features: capabilities:0xc7, actions:0xffff
 1(em1): addr:78:2b:cb:4b:db:c5
   config:      0
   state:       0
   current:     1GB-FD COPPER AUTO_NEG
   advertised:
10MB-HD 10MB-FD 100MB-HD 100MB-FD 1GB-HD 1GB-FD COPPER AUTO_NEG AUTO_PAUSE
supported:  10MB-HD 10MB-FD
100MB-HD 100MB-FD 1GB-HD 1GB-FD COPPER AUTO_NEG
 2(tap0): addr:a6:30:4d:0f:40:49
   config:      0
   state:       LINK_DOWN
   current:     10MB-FD COPPER
LOCAL(ovs): addr:78:2b:cb:4b:db:c5
   config:      0
   state:       0
OFPT_GET_CONFIG_REPLY (xid=0x3): frags=normal
miss_send_len=0
```

## Controller Configuration

In an OVS switch you can set a controller with the following command:

```
$ sudo ovs-vsctl set-controller br-int tcp:192.168.1.208:6633
```

On the other hand, we can also set the controller failure settings. When a controller is configured, it is, ordinarily, responsible for setting up all flows on the switch. Thus, if the connection to the controller fails, no new network

connections can be set up. If the connection to the controller stays down long enough, no packets can pass through the switch at all.

If the value is standalone, or if neither of these settings is set, ovs-vswitchd will take over responsibility for setting up flows when no message has been received from the controller for three times the inactivity probe interval. In this mode, ovs-vswitchd causes the datapath to act like an ordinary MAC-learning switch. ovs-vswitchd will continue to retry connecting to the controller in the background and, when the connection succeeds, it discontinues its standalone behavior. If this option is set to secure, ovs-vswitchd will not set up flows on its own when the controller connection fails. Example:

```
$ sudo ovs-vsctl set-fail-mode br0 secure
```

## 2.5 Implementing a MPLS network with OVS and OpenFlow

The following pages document the experiments performed with Open vSwitch and OpenFlow 1.3 in order to recreate simple MPLS network functionalities. This experiments aim to give a better understanding of Open vSwitch behavior and some of its tools.

### 2.5.1 Basics of MPLS

Multiprotocol Label Switching (MPLS) is a mechanism in high-performance telecommunications networks that directs data from one network node to the next based on short path labels rather than long network addresses, avoiding complex lookups in a routing table. The labels identify virtual links (paths) between distant nodes rather than endpoints. MPLS can encapsulate packets of various network protocols.

MPLS works by prefixing packets with an MPLS header, containing one or more labels. This is called a label stack. Each label stack entry contains four fields:

- A 20-bit label value. A label with the value of 1 represents the router alert label.
- A 3-bit Traffic Class field for QoS (quality of service) priority (experimental) and ECN (Explicit Congestion Notification)..
- A 1-bit bottom of stack flag. If this is set, it signifies that the current label is the last in the stack.
- An 8-bit TTL (time to live) field.

These MPLS-labeled packets are switched after a label lookup/switch instead of a lookup into the IP table.

The presence of such a label, however, has to be indicated to the router/switch. In the case of Ethernet frames this is done through the use of EtherType values 0x8847 and 0x8848, for unicast and multicast connections respectively.

#### MPLS concepts

**Label switch router:** A MPLS router that performs routing based only on the label is called a label switch router (LSR) or transit router. This is a type of router located in the middle of a MPLS network. It is responsible for switching the labels used to route packets. When an LSR receives a packet, it uses the label included in the packet

header as an index to determine the next hop on the label-switched path (LSP) and a corresponding label for the packet from a lookup table. The old label is then removed from the header and replaced with the new label before the packet is routed forward.

**Label edge router:** A label edge router (LER, also known as edge LSR) is a router that operates at the edge of an MPLS network and acts as the entry and exit points for the network. LERs respectively, push an MPLS label onto an incoming packet and pop it off the outgoing packet. Alternatively, under penultimate hop popping this function may instead be performed by the LSR directly connected to the LER.

When forwarding IP datagrams into the MPLS domain, a LER uses routing information to determine appropriate labels to be affixed, labels the packet accordingly, and then forwards the labelled packets into the MPLS domain. Likewise, upon receiving a labelled packet which is destined to exit the MPLS domain, the LER strips off the label and forwards the resulting IP packet using normal IP forwarding rules.

**Label Distribution Protocol:** Labels are distributed between LERs and LSRs using the Label Distribution Protocol (LDP). LSRs in an MPLS network regularly exchange label and reachability information with each other using standardized procedures in order to build a complete picture of the network they can then use to forward packets.

**Label-Switched Paths:** A label-switched path (LSP) is a path through an MPLS network, set up by a signaling protocol such as LDP.

## 2.5.2 Setting up the environment

This experiment is based on the tutorial on MPLS with OpenFlow proposed by the *Universita degli Studi Roma Tre*. This tutorial can be found in the following link:

[http://tocai.dia.uniroma3.it/compunet-wiki/index.php/MPLS\\_with\\_OpenFlow:\\_howto](http://tocai.dia.uniroma3.it/compunet-wiki/index.php/MPLS_with_OpenFlow:_howto)

We are using Open vSwitch version 2.0.2 and mininet 2.2.0 For this experiment we will use a simple topology, illustrated in figure 2.4. Host 2 will not be used in this experiment. The numbers in blue correspond to the label that the packets carry with them in each link.

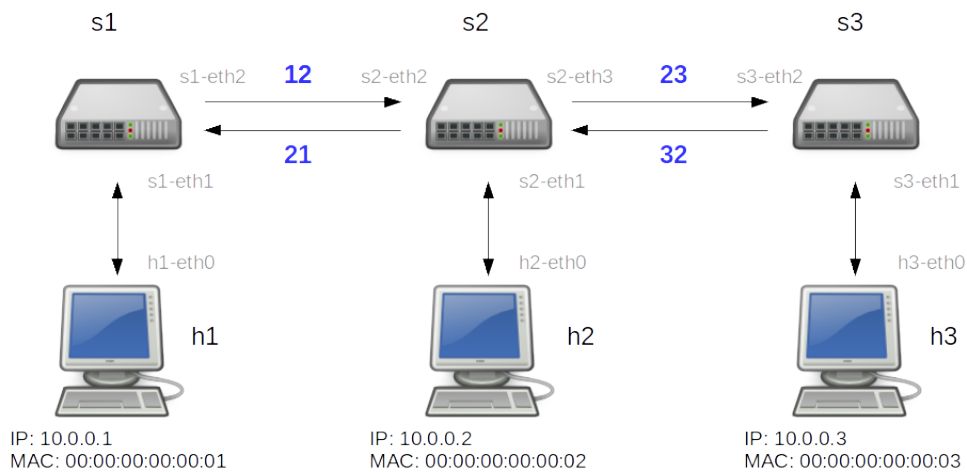


Figure 2.4: Topology

As you can see, when packets travel from h1 to h3:

- The switch s1 pushes the label 12 and forwards the packet to switch s2.
- The switch s2 swaps the label 12 for the label 23 and forwards the packet to switch s3
- The switch s3 pops the label 23 and forwards the packet to host h3.

This environment will be set up using only mininet and Open vSwitch. Please note that the label stack of the packets is just one label deep, this is because currently Open vSwitch only supports one label. The whole experiment will be performed in userspace, as the OVS kernel module does not support MPLS yet. For this reason we use the option `datapath=user`

In order to create the topology, we execute the following command:

```
$ sudo mn -topo linear,3 -mac -switch ovsk,datapath=user
```

Using the following script, we can write the OpenFlow Rules to the switches, so the network behaves as an MPLS network. Each instruction is previously commented for a better understanding of the script.

```
#!/bin/bash

# Configure switch to use OpenFlow 1.3
echo "Setting Switches to work with OpenFlow 1.3"

for i in s1 s2 s3; do
  sudo ovs-vsctl set bridge $i protocols=OpenFlow13
done

# Clear flow tables
echo "Clearing Flow tables.."

for i in s1 s2 s3; do
  sudo ovs-ofctl -O OpenFlow13 del-flows $i
done

##### Rules at S1 #####
echo "Setting up rules at s1"

## TABLE 0 ##
# If ethertype is IPv4 (800h) or MPLS unicast (8847h), go to table 1
sudo ovs-ofctl -O OpenFlow13 add-flow s1
"table=0,in_port=1,eth_type=0x0800,actions=goto_table:1"
sudo ovs-ofctl -O OpenFlow13 add-flow s1
"table=0,in_port=2,eth_type=0x8847,actions=goto_table:1"

## TABLE 1 ##
# If it came in through port 1 & it's an IPv4 packet:
# push label 12 and send it through port 2
sudo ovs-ofctl -O OpenFlow13 add-flow s1
"table=1,in_port=1,eth_type=0x0800,
actions=push_mpls:0x8847,set_field:12->mpls_label, output:2"
```

```

# If it came in through port 2 & it's an MPLS unicast & bottom of stack:
# pop label and send it through port 1
sudo ovs-ofctl -O OpenFlow13 add-flow s1
    "table=1,in_port=2,eth_type=0x8847,mpls_bos=1,
    actions=pop_mpls:0x0800,output:1"

##### Rules at s2 #####
echo "Setting up rules at s2"

# Pop old label push new label
sudo ovs-ofctl -O OpenFlow13 add-flow s2
    "table=0,in_port=2,eth_type=0x8847,actions=\
    pop_mpls:0x800, push_mpls:0x8847,set_field:23->mpls_label,output:3"
sudo ovs-ofctl -O OpenFlow13 add-flow s2
    "table=0,in_port=3,eth_type=0x8847,actions=\
    pop_mpls:0x800, push_mpls:0x8847,set_field:21->mpls_label,output=2"

##### Rules at s3 #####
echo "Setting up rules at s3"

## TABLE 0 ##
# If ethertype is IPv4 (800h) or MPLS unicast (8847h), go to table 1
sudo ovs-ofctl -O OpenFlow13 add-flow s3
    "table=0,in_port=1,eth_type=0x0800,actions=goto_table:1"
sudo ovs-ofctl -O OpenFlow13 add-flow s3
    "table=0,in_port=2,eth_type=0x8847,actions=goto_table:1"

## TABLE 1 ##
# If it came in through port 1 & it's an IPv4 packet:
# push label 32 and send it through port 2
sudo ovs-ofctl -O OpenFlow13 add-flow s3
    "table=1,in_port=1,eth_type=0x0800,actions=push_mpls:0x8847,
    set_field:32->mpls_label,output:2"

# If it came in through port 2 & it's an MPLS unicast & bottom of stack:
# pop label and send it through port 1
sudo ovs-ofctl -O OpenFlow13 add-flow s3
    "table=1,in_port=2,eth_type=0x8847,mpls_bos=1,
    actions=pop_mpls:0x0800,output:1"

##### Rules for ARP traffic #####
echo "Setting up rules for ARP traffic"
sudo ovs-ofctl -O OpenFlow13 add-flow s1
    "table=0,in_port=1,eth_type=0x806,actions=output:2"
sudo ovs-ofctl -O OpenFlow13 add-flow s1
    "table=0,in_port=2,eth_type=0x806,actions=output:1"

```



```

sudo ovs-ofctl -O OpenFlow13 add-flow s2
"table=0,in_port=2,eth_type=0x806,actions=output:3"
sudo ovs-ofctl -O OpenFlow13 add-flow s2
"table=0,in_port=3,eth_type=0x806,actions=output:2"
sudo ovs-ofctl -O OpenFlow13 add-flow s3
"table=0,in_port=1,eth_type=0x806,actions=output:2"
sudo ovs-ofctl -O OpenFlow13 add-flow s3
"table=0,in_port=2,eth_type=0x806,actions=output:1"

```

The script writes rules for ARP, IPV4 and MPLS unicast ethernet packets. Every other packet will be dropped. Two flow tables have to be used in hosts h1 and h3, as proposed in the tutorial. This implementation is discussed in section 2.5.4

As you can see this is an hibryd application, as it forwards IPV4 traffic through MPLS techniques but it also forwards ARP traffic in a classic way. Also, there's only one IP network, which is not the real scenario for MPLS networks, which were proposed to enable faster packet forwarding between different IP networks.

However, this experiment is a good starting point in order to understand the mechanics of the MPLS protocol and its relation with OpenFlow and Open vSwitch

### 2.5.3 Testing the network

With this setup, we expect the host h1 to reach the host h3 and viceversa. Host 2 cannot reach nor be reached by any other host. When a reachability test is performed with the command *pingall* on the Mininet Command Line, we obtain the following result:

```

mininet> pingall
*** Ping: testing ping reachability
h1 -> X h3
h2 -> X X
h3 -> h1 X
*** Results: 66% dropped (2/6 received)

```

As expected, only h1 and h3 see each other.

Now that the reachability has been checked, we will listen to some of the interfaces to capture the traffic and confirm that the labels are being pushed, swapped, and popped correctly. You can see where each interface is located inside the topology in figure 2.4

The following captures correspond to an ICMP echo request from h1 to h3 and the associated reply

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.0.1	10.0.0.3	ICMP	98	Echo (ping) request id=0x1087, seq=1/256, ttl=64 (reply in 2)
2	0.010074000	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping) reply id=0x1087, seq=1/256, ttl=64 (request in 1)
3	5.010172000	00:00:00_00:00:01	00:00:00_00:00:03	ARP	42	Who has 10.0.0.3? Tell 10.0.0.1
4	5.028555000	00:00:00_00:00:03	00:00:00_00:00:01	ARP	60	Who has 10.0.0.1? Tell 10.0.0.3
5	5.028556000	00:00:00_00:00:01	00:00:00_00:00:03	ARP	42	10.0.0.1 is at 00:00:00:00:00:01
6	5.049286000	00:00:00_00:00:03	00:00:00_00:00:01	ARP	60	10.0.0.3 is at 00:00:00:00:00:03

> Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0  
 > Ethernet II, Src: 00:00:00\_00:00:01 (00:00:00:00:00:01), Dst: 00:00:00\_00:00:03 (00:00:00:00:00:03)  
 > Internet Protocol Version 4, Src: 10.0.0.1 (10.0.0.1), Dst: 10.0.0.3 (10.0.0.3)  
 > Internet Control Message Protocol

Figure 2.5: Capture from interface s1-eth1

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.0.1	10.0.0.3	ICMP	102	Echo (ping) request id=0x0eb5, seq=1/256, ttl=64 (reply in 2)
2	0.003347000	10.0.0.3	10.0.0.1	ICMP	102	Echo (ping) reply id=0x0eb5, seq=1/256, ttl=64 (request in 1)
3	5.004302000	00:00:00_00:00:01	00:00:00_00:00:03	ARP	60	Who has 10.0.0.3? Tell 10.0.0.1
4	5.005442000	00:00:00_00:00:03	00:00:00_00:00:01	ARP	60	Who has 10.0.0.1? Tell 10.0.0.3
5	5.007808000	00:00:00_00:00:01	00:00:00_00:00:03	ARP	60	10.0.0.1 is at 00:00:00:00:00:01
6	5.009045000	00:00:00_00:00:03	00:00:00_00:00:01	ARP	60	10.0.0.3 is at 00:00:00:00:00:03

```

Frame 1: 102 bytes on wire (816 bits), 102 bytes captured (816 bits) on interface 0
Ethernet II, Src: 00:00:00_00:00:01 (00:00:00:00:00:01), Dst: 00:00:00_00:00:03 (00:00:00:00:00:03)
MultiProtocol Label Switching Header, Label: 12 (Reserved - Unknown), Exp: 0, S: 1, TTL: 64
0000 0000 0000 0000 1100 .... = MPLS Label: Unknown (12)
.... = MPLS Experimental Bits: 0
.... = MPLS Bottom Of Label Stack: 1
.... 0100 0000 = MPLS TTL: 64
Internet Protocol Version 4, Src: 10.0.0.1 (10.0.0.1), Dst: 10.0.0.3 (10.0.0.3)
Internet Control Message Protocol

```

Figure 2.6: Capture from interface s1-eth2

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.0.1	10.0.0.3	ICMP	102	Echo (ping) request id=0x0daa, seq=1/256, ttl=64
2	0.005904000	10.0.0.3	10.0.0.1	ICMP	102	Echo (ping) reply id=0x0daa, seq=1/256, ttl=64 (request in 1)
3	5.007084000	00:00:00_00:00:03	00:00:00_00:00:01	ARP	60	Who has 10.0.0.1? Tell 10.0.0.3
4	5.010770000	00:00:00_00:00:01	00:00:00_00:00:03	ARP	60	Who has 10.0.0.3? Tell 10.0.0.1
5	5.020535000	00:00:00_00:00:03	00:00:00_00:00:01	ARP	60	10.0.0.3 is at 00:00:00:00:00:03
6	5.023338000	00:00:00_00:00:01	00:00:00_00:00:03	ARP	60	10.0.0.1 is at 00:00:00:00:00:01

```

Frame 1: 102 bytes on wire (816 bits), 102 bytes captured (816 bits) on interface 0
Ethernet II, Src: 00:00:00_00:00:01 (00:00:00:00:00:01), Dst: 00:00:00_00:00:03 (00:00:00:00:00:03)
MultiProtocol Label Switching Header, Label: 23, Exp: 0, S: 1, TTL: 64
0000 0000 0000 0001 0111 .... = MPLS Label: 23
.... = MPLS Experimental Bits: 0
.... = MPLS Bottom Of Label Stack: 1
.... 0100 0000 = MPLS TTL: 64
Internet Protocol Version 4, Src: 10.0.0.1 (10.0.0.1), Dst: 10.0.0.3 (10.0.0.3)
Internet Control Message Protocol

```

Figure 2.7: Capture from interface s2-eth3

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.0.1	10.0.0.3	ICMP	98	Echo (ping) request id=0x109a, seq=1/256, ttl=64
2	0.000143000	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping) reply id=0x109a, seq=1/256, ttl=64 (request in 1)
3	5.005366000	00:00:00_00:00:01	00:00:00_00:00:03	ARP	60	Who has 10.0.0.3? Tell 10.0.0.1
4	5.005655000	00:00:00_00:00:03	00:00:00_00:00:01	ARP	42	10.0.0.3 is at 00:00:00:00:00:03

```

Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
Ethernet II, Src: 00:00:00_00:00:01 (00:00:00:00:00:01), Dst: 00:00:00_00:00:03 (00:00:00:00:00:03)
Internet Protocol Version 4, Src: 10.0.0.1 (10.0.0.1), Dst: 10.0.0.3 (10.0.0.3)
Internet Control Message Protocol

```

Figure 2.8: Capture from interface s3-eth1

This captures show that the experiment has performed as expected:

- In figure 2.5 the ICMP request enters the switch s1 without any label.
- In figure 2.6 the packet leaves the switch s1 with the label 12 inserted between layers 2 and 3. Push performed correctly
- In figure 2.7 the packet leaves switch s2 with the label 23. Swap performed correctly
- In figure 2.8 the packet leaves the switch s3 without any label. Pop performed correctly.

- The captures also show that the reply packet follows a similar process with the labels 32 and 21 (Not shown in the figures).

This experiment shows the potential of OpenFlow and Open vSwitch for creating MPLS networks.

## 2.5.4 Discussion on alternative implementations

This section discusses alternative implementations of this experiment, some of them are bad implementations that do not perform as expected. Some of them behave slightly different from the original experiment.

### Simplification

#### Reducing the number of tables

The experiment proposed in section 2.5.2 writes two flow tables in the switches s1 and s3. But the flows on the first table only forward to the next stage in the pipeline with the same match that is used in the second table. This is why we wondered if this two tables could be reduced to one.

For this purpose we remove the first two rules for s1 and s2, and also set the other rules to be in table 0. We dump the flows on both switches to be sure that the flows have been added properly.

```
mininet@mininet-vm:~/experiments$ sudo ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
  cookie=0x0, duration=6.278s, table=0, n_packets=0, n_bytes=0,
mpls,in_port=2,mpls_bos=1 actions=pop_mpls:0x0800,output:1
  cookie=0x0, duration=5.768s, table=0, n_packets=0, n_bytes=0, arp,in_port=2
actions=output:1
  cookie=0x0, duration=5.827s, table=0, n_packets=0, n_bytes=0, arp,in_port=1
actions=output:2
  cookie=0x0, duration=6.336s, table=0, n_packets=0, n_bytes=0, ip,in_port=1
actions=push_mpls:0x8847,set_field:12->mpls_label,output:2
mininet@mininet-vm:~/experiments$ sudo ovs-ofctl -O openflow13 dump-flows s3
OFPST_FLOW reply (OF1.3) (xid=0x2):
  cookie=0x0, duration=8.773s, table=0, n_packets=0, n_bytes=0,
mpls,in_port=2,mpls_bos=1 actions=pop_mpls:0x0800,output:1
  cookie=0x0, duration=8.413s, table=0, n_packets=0, n_bytes=0, arp,in_port=2
actions=output:1
  cookie=0x0, duration=8.472s, table=0, n_packets=0, n_bytes=0, arp,in_port=1
actions=output:2
  cookie=0x0, duration=8.84s, table=0, n_packets=0, n_bytes=0, ip,in_port=1
actions=push_mpls:0x8847,set_field:32->mpls_label,output:2
```

The rules have been added properly to the switches. Sending an ICMP request message from h1 to h3 also performs well.

```
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data:
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=17.0 ms
```

```

--- 10.0.0.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 17.041/17.041/17.041/0.000 ms

```

Capture from both s1-eth2 and s3-eth2 shows that the packets have the right label, as shown in figure 2.9

1	0.00000000	00:00:00_00:00:01	Broadcast	ARP	60 Who has 10.0.0.3? Tell 10.0.0.1
2	0.00162300	00:00:00_00:00:03	00:00:00_00:00:01	ARP	60 10.0.0.3 is at 00:00:00:00:00:03
3	0.00925900	10.0.0.1	10.0.0.3	ICMP	102 Echo (ping) request id=0x16c7, seq=1/256, ttl=64 (reply in 4)
4	0.01199600	10.0.0.3	10.0.0.1	ICMP	102 Echo (ping) reply id=0x16c7, seq=1/256, ttl=64 (request in 3)
5	-0.00110400	00:00:00_00:00:01	Broadcast	ARP	60 Who has 10.0.0.3? Tell 10.0.0.1
6	0.00410000	00:00:00_00:00:03	00:00:00_00:00:01	ARP	60 10.0.0.3 is at 00:00:00:00:00:03
7	0.00879000	10.0.0.1	10.0.0.3	ICMP	102 Echo (ping) request id=0x16c7, seq=1/256, ttl=64 (reply in 8)
8	0.01420500	10.0.0.3	10.0.0.1	ICMP	102 Echo (ping) reply id=0x16c7, seq=1/256, ttl=64 (request in 7)
9	5.02394500	00:00:00_00:00:03	00:00:00_00:00:01	ARP	60 Who has 10.0.0.1? Tell 10.0.0.3
10	5.02594000	00:00:00_00:00:01	00:00:00_00:00:03	ARP	60 10.0.0.1 is at 00:00:00:00:00:01
11	5.02262200	00:00:00_00:00:03	00:00:00_00:00:01	ARP	60 Who has 10.0.0.1? Tell 10.0.0.3
12	5.02688300	00:00:00_00:00:01	00:00:00_00:00:03	ARP	60 10.0.0.1 is at 00:00:00:00:00:01

```

▶ Frame 3: 102 bytes on wire (816 bits), 102 bytes captured (816 bits) on interface 1
▶ Ethernet II, Src: 00:00:00_00:00:01 (00:00:00:00:00:01), Dst: 00:00:00_00:00:03 (00:00:00:00:00:03)
▼ MultiProtocol Label Switching Header, Label: 23, Exp: 0, S: 1, TTL: 64
  0000 0000 0000 0001 0111 .... .. = MPLS Label: 23
  .... .. = MPLS Experimental Bits: 0
  .... .. = MPLS Bottom Of Label Stack: 1
  .... .. 0100 0000 = MPLS TTL: 64
▶ Internet Protocol Version 4, Src: 10.0.0.1 (10.0.0.1), Dst: 10.0.0.3 (10.0.0.3)
▶ Internet Control Message Protocol

```

Figure 2.9: Captuture from s1-eth2 and s3-eth2

## Sending ARP traffic through MPLS

We can make a more general rule so both IPV4 and ARP packets are sent through MPLS. This should reduce the number of flows that is written in every switch.

## Marking the packets as IPV4

One possible way of approaching this problem could be by only removing the match of IPV4 from the rules in s1 and s3 so they just care about the input port:

```

sudo ovs-ofctl -O OpenFlow13 add-flow s3
"table=0,in_port=1,actions=push_mpls:0x8847,
set_field:32->mpls_label,output:2"

```

But this is a bad implementation as the ARP message will never reach its destination.

When popping a label, the switch must rewrite the ethertype from MPLS unicast to whatever the packet is. If we have only modified the previous rule, the rule for popping is:

```

sudo ovs-ofctl -O OpenFlow13 add-flow s1
"table=0,in_port=2,eth_type=0x8847,mpls_bos=1,
actions=pop_mpls:0x0800,output:1"

```

Which means that every packet with ethertype MPLS unicast will be forwarded as an IPV4 ethernet packet and not an ARP packet. The packet reaches its destiny but it's not recognized as an ARP request. This effect can be clearly seen

at figure 2.10, which is a capture from s3-eth1 when the ARP request comes from h1. We have no way to know if the MPLS packet is an IPV4 or ARP packet so there's no way that we can write the right ethertype when popping without additional information.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	00:00:00_00:00:01	Broadcast	IPv4	60	Bogus IP header length (0, must be at least 20)
2	0.998375000	00:00:00_00:00:01	Broadcast	IPv4	60	Bogus IP header length (0, must be at least 20)
3	1.997733000	00:00:00_00:00:01	Broadcast	IPv4	60	Bogus IP header length (0, must be at least 20)

---

> Frame 1: 60 bytes on wire (480 bits), 60 bytes captured (480 bits) on interface 0  
 > Ethernet II, Src: 00:00:00\_00:00:01 (00:00:00:00:00:01), Dst: Broadcast (ff:ff:ff:ff:ff:ff)  
 > Internet Protocol Version 4

Figure 2.10: Capture from s3-eth1

## Checking the labels

Alternatively, instead of making a wildcard for every incoming packet, we will push a different label for ARP traffic and for IP traffic, so the LER can know which kind of ethertype has to write to the packet when popping the label. For this experiment we will not swap the labels of ARP packets. We use the label 111 for h1 ARP packets and label 333 for h3.

This will not reduce the number of flows in each switch, but it should send the ARP traffic through the MPLS system. The resulting rules are:

S1:

```
# If it came in through port 1 & it's an IPv4 packet:
# push label 12 and send it through port 2
"table=0,in_port=1,eth_type=0x800,actions=push_mpls:0x8847,
set_field:12->mpls_label, output:2"
# If it came in through port 1 & it's an ARP packet:
# push label 01 and send it through port 2
"table=0,in_port=1,eth_type=0x806,actions=push_mpls:0x8847,
set_field:111->mpls_label,output:2"
# If it came in through port 2 & it's an MPLS unicast & label 03:
# pop label, set ethertype as ARP and send it through port 1
"table=0,in_port=2,eth_type=0x8847,mpls_bos=1,mpls_label=333,
actions=pop_mpls:0x0806, output:1"
# If it came in through port 2 & it's an MPLS unicast & label 21:
# pop label, set ethertype as ARP and send it through port 1
"table=0,in_port=2,eth_type=0x8847,mpls_bos=1,mpls_label=21,
actions=pop_mpls:0x0800, output:1"
```

S2:

```
# IPV4: Pop old label push new label
"table=0,in_port=2,eth_type=0x8847,mpls_label=12,actions=\
  pop_mpls:0x800, push_mpls:0x8847,set_field:23->mpls_label,output:3"
"table=0,in_port=3,eth_type=0x8847,mpls_label=32,actions=\
  pop_mpls:0x800, push_mpls:0x8847,set_field:21->mpls_label,output:2"
# ARP: Do not swap labels
"table=0,in_port=2,eth_type=0x8847,mpls_label=111,actions=output:3"
```

```
"table=0,in_port=3,eth_type=0x8847,mpls_label=333,actions=output:2"
```

S3:

```
# If it came in through port 1 & it's an IPv4 packet:
# push label 32 and send it through port 2
"table=0,in_port=1,eth_type=0x800,actions=push_mpls:0x8847,
set_field:32->mpls_label, output:2"
# If it came in through port 1 & it's an ARP packet:
# push label 03 and send it through port 2
"table=0,in_port=1,eth_type=0x806,actions=push_mpls:0x8847,
set_field:333->mpls_label,output:2"
# If it came in through port 2 & it's an MPLS unicast & label 01:
# pop label, set ethertype as ARP and send it through port 1
"table=0,in_port=2,eth_type=0x8847,mpls_bos=1,mpls_label=111,
actions=pop_mpls:0x0806,output:1"
# If it came in through port 2 & it's an MPLS unicast & label 23:
# pop label, set ethertype as ARP and send it through port 1
"table=0,in_port=2,eth_type=0x8847,mpls_bos=1,mpls_label=23,
actions=pop_mpls:0x0800,output:1"
```

After applying these rules, there is reachability between h1 and h3. As shown in the figures below, the process is completely transparent to the hosts:

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	00:00:00_00:00:01	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.1
2	0.028600000	00:00:00_00:00:03	00:00:00_00:00:01	ARP	60	10.0.0.3 is at 00:00:00:00:00:03
3	0.028864000	10.0.0.1	10.0.0.3	ICMP	98	Echo (ping) request id=0x3193, seq=1/256, ttl=64 (reply in 4)
4	0.048070000	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping) reply id=0x3193, seq=1/256, ttl=64 (request in 3)
5	5.081504000	00:00:00_00:00:03	00:00:00_00:00:01	ARP	60	Who has 10.0.0.1? Tell 10.0.0.3
6	5.081948000	00:00:00_00:00:01	00:00:00_00:00:03	ARP	42	10.0.0.1 is at 00:00:00:00:00:01

Figure 2.11: Capture from s1-eth1

2	2.582586000	00:00:00_00:00:01	Broadcast	ARP	60	Who has 10.0.0.3? Tell 10.0.0.1
3	2.582765000	00:00:00_00:00:03	00:00:00_00:00:01	ARP	42	10.0.0.3 is at 00:00:00:00:00:03
4	2.590917000	10.0.0.1	10.0.0.3	ICMP	98	Echo (ping) request id=0x3425, seq=1/256, ttl=64 (reply in 5)
5	2.591069000	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping) reply id=0x3425, seq=1/256, ttl=64 (request in 4)
6	7.592040000	00:00:00_00:00:03	00:00:00_00:00:01	ARP	42	Who has 10.0.0.1? Tell 10.0.0.3
7	7.606881000	00:00:00_00:00:01	00:00:00_00:00:03	ARP	60	10.0.0.1 is at 00:00:00:00:00:01

Figure 2.12: Capture from s3-eth1

If we capture the packets on the intermediate interfaces we can see that labels have been pushed correctly, although wireshark sees the ARP labeled packets as malformed packets. For instance, figure 2.13 shows the capture from the interface s2-eth2

1	0.00000000	EquipTra_00:00:00	AvlabTec_00:06:04	LLC	64 [Malformed Packet]
2	0.01008000	NetSys_00:00:00	AvlabTec_00:06:04	LLC	64 [Malformed Packet]
3	0.01458600	10.0.0.1	10.0.0.3	ICMP	102 Echo (ping) request id=0x36c6, seq=1/256, ttl=64 (reply in 4)
4	0.02003900	10.0.0.3	10.0.0.1	ICMP	102 Echo (ping) reply id=0x36c6, seq=1/256, ttl=64 (request in 3)
5	5.03118000	EquipTra_00:00:00	AvlabTec_00:06:04	LLC	64 [Malformed Packet]
6	5.03424800	NetSys_00:00:00	AvlabTec_00:06:04	LLC	64 [Malformed Packet]

> Frame 1: 64 bytes on wire (512 bits), 64 bytes captured (512 bits) on interface 0  
 > Ethernet II, Src: 00:00:00\_00:00:01 (00:00:00:00:00:01), Dst: Broadcast (ff:ff:ff:ff:ff:ff)  
 > MultiProtocol Label Switching Header, Label: 111, Exp: 0, S: 1, TTL: 64  
 > IEEE 802.3 Ethernet  
 > Logical-Link Control  
 > [Malformed Packet: LLC]

Figure 2.13: Capture from s2-eth2

However, ARP traffic is not meant to go through an MPLS network, and should be handled by the LERs if necessary, as the MPLS network connects different IP networks, and has no internal IP addresses. This goal of this modification is to show that we can use different labels to differentiate the traffic.

## Kernel mode

When using Open vSwitch in kernel mode, it is impossible to forward a packet after pushing an MPLS header on top of it. If we try to execute this, the first switch seems to drop the packet.

Using the same rules as in section 2.5.4, the following rule seems to fail. Only ARP traffic is forwarded by the first switch, as can be seen in figure 2.14.

```
sudo ovs-ofctl -O OpenFlow13 add-flow s1
"table=0,in_port=1,eth_type=0x800,actions=push_mpls:0x8847,
set_field:12->mpls_label, output:2"
```

No.	Time	Source	Destination	Protocol	Length	Info
1	0.00000000	00:00:00_00:00:01	Broadcast	ARP	42	Who has 10.0.0.3? Tell 10.0.0.1
2	0.00524300	00:00:00_00:00:03	00:00:00_00:00:01	ARP	42	10.0.0.3 is at 00:00:00:00:00:03

Figure 2.14: Capture from s1-eth2 in kernel mode

We wonder if this is due to pushing an MPLS label and forwarding the packet in the same rules, so we try switching the previous rule for the following two in switches s1 and s3:

```
sudo ovs-ofctl -O OpenFlow13 add-flow s1
"table=0,in_port=1,eth_type=0x0800, actions=push_mpls:0x8847,
set_field:12->mpls_label,goto_table:1"
sudo ovs-ofctl -O OpenFlow13 add-flow s1
"table=1,in_port=1,eth_type=0x08847,mpls_label=12,actions=output:2"
```

This rules will push the label into IPV4 packets and forward them to the next table, where they're forwarded to an upputput port. After confirming that this rules work fine with userspace datapaths, we try to run the network in kernel mode, but still obtain the same results as before. Although the mininet distribution (Which is the OS running in the virtual machine where this experiments are performed) does not support kernel mode yet (kernel 3.13), the 3.19 Linux Kernel has finally added MPLS support for Open vSwitch [10]. Also, the current version of OVS is not compatible with this new kernel [5].

## Trying to push/pop more than one label

The ovs-ofctl manpage states that the actions of pushing and popping mpls labels have some limitations:

- **push\_mpls:ethertype** Processing of actions will stop if push\_mpls follows another push\_mpls unless there is a pop\_mpls in between.
- **pop\_mpls:ethertype** The implementation restricts *ethertype* to a non-MPLS Ethertype and thus pop\_mpls should only be applied to packets with an MPLS label stack depth of one. A further limitation is that processing of actions will stop if pop\_mpls follows another pop\_mpls unless there is a push\_mpls in between. *ethertype* can't be 0x8847 nor 0x8848.

When we try to pop a label which is not at bottom of stack and set the ethertype to 0x8847 when popping:

```
sudo ovs-ofctl -O OpenFlow13 add-flow s1
"table=0,in_port=2,eth_type=0x8847,mpls_bos=0,
mpls_label=21,actions=pop_mpls:0x8847, goto_table:1"
```

we receive the following error

```
OFPT_ERROR (OF1.3) (xid=0x2): OFPBAC_BAD_ARGUMENT
OFPT_FLOW_MOD (OF1.3) (xid=0x2):
(***truncated to 64 bytes from 104***)
00000000  04 0e 00 68 00 00 00 02-00 00 00 00 00 00 00 00 |...h.....|
00000010  00 00 00 00 00 00 00 00-00 00 00 00 00 00 80 00 |.....|
00000020  ff ff ff ff ff ff ff ff-ff ff ff ff 00 00 00 00 |.....|
00000030  00 01 00 1f 80 00 00 04-00 00 00 02 80 00 0a 02 |.....|
```

However, ovs-ofctl lets me push more than one label to the same packet, as shown in the figure 2.15, although the last switch cannot pop them properly later:

```
# In switch s1:
sudo ovs-ofctl -O OpenFlow13 add-flow s1
"table=0,in_port=1,eth_type=0x800,actions=push_mpls:0x8847,
set_field:12->mpls_label,output:2"
# In switch s2:
sudo ovs-ofctl -O OpenFlow13 add-flow s2
"table=0,in_port=2,eth_type=0x8847,mpls_label=12,actions=
push_mpls:0x8847,set_field:23->mpls_label,output:3"
```

A new label has been pushed in each switch.

1	0.000000000	10.0.0.1	10.0.0.3	ICMP	106 Echo (ping) request id=0x3a9e, seq=1/256, ttl=64
2	5.004341000	EquipTra_00:00:00	AvlabTec_00:06:04	LLC	64 [Malformed Packet]
3	5.019237000	Netsys_00:00:00	AvlabTec_00:06:04	LLC	64 [Malformed Packet]

```
> Frame 1: 106 bytes on wire (848 bits), 106 bytes captured (848 bits) on interface 0
> Ethernet II, Src: 00:00:00:00:00:01 (00:00:00:00:00:01), Dst: 00:00:00:00:00:03 (00:00:00:00:00:03)
> MultiProtocol Label Switching Header, Label: 23, Exp: 0, S: 0, TTL: 64
> MultiProtocol Label Switching Header, Label: 12 (Reserved - Unknown), Exp: 0, S: 1, TTL: 64
> Internet Protocol Version 4, Src: 10.0.0.1 (10.0.0.1), Dst: 10.0.0.3 (10.0.0.3)
> Internet Control Message Protocol
```

Figure 2.15: Capture from s2-eth3



OVS will add support for pushing/popping more than one label in version 2.4 [1].

## 2.6 Final code

Here we provide the final code after the simplifications and modifications. With this configuration, we send all the traffic using MPLS labels, and have reduced the number of tables.

```
#!/bin/bash

# Configure switch to use OpenFlow 1.3
echo "Setting Switches to work with OpenFlow 1.3"

for i in s1 s2 s3; do
    sudo ovs-vsctl set bridge $i protocols=OpenFlow13
done

# Clear flow tables
echo "Clearing Flow tables.."

for i in s1 s2 s3; do
    sudo ovs-ofctl -O OpenFlow13 del-flows $i
done

# Note: Port 1 is connected to a host in each switch
##### Rules at S1 #####
echo "Setting up rules at s1"

# If it came in through port 1 & it's an IPv4 packet:
# push label 12 and send it through port 2
sudo ovs-ofctl -O OpenFlow13 add-flow s1
    "table=0,in_port=1,eth_type=0x800,actions=push_mpls:0x8847,
    set_field:12->mpls_label,output:2"
# If it came in through port 1 & it's an ARP packet:
# push label 01 and send it through port 2
sudo ovs-ofctl -O OpenFlow13 add-flow s1
    "table=0,in_port=1,eth_type=0x806,actions=push_mpls:0x8847,
    set_field:111->mpls_label,
output:2"
# If it came in through port 2 & it's an MPLS unicast & label 333:
# pop label, set ethertype as ARP and send it through port 1
sudo ovs-ofctl -O OpenFlow13 add-flow s1
    "table=0,in_port=2,eth_type=0x8847,mpls_bos=1,mpls_label=333,
    actions=pop_mpls:0x0806,output:1"
# If it came in through port 2 & it's an MPLS unicast & label 21:
# pop label, set ethertype as ARP and send it through port 1
sudo ovs-ofctl -O OpenFlow13 add-flow s1
    "table=0,in_port=2,eth_type=0x8847,mpls_bos=1,mpls_label=21,
```

```
actions=pop_mpls:0x0800,output:1"

##### Rules at s2 #####
echo "Setting up rules at s2"

# IPv4: Pop old label push new label
sudo ovs-ofctl -O OpenFlow13 add-flow s2
"table=0,in_port=2,eth_type=0x8847,mpls_label=12,actions=\
    pop_mpls:0x800, push_mpls:0x8847,set_field:23->mpls_label,output:3"
sudo ovs-ofctl -O OpenFlow13 add-flow s2
"table=0,in_port=3,eth_type=0x8847,mpls_label=32,actions=\
    pop_mpls:0x800, push_mpls:0x8847,set_field:21->mpls_label,output:2"
# ARP: Do not swap labels
sudo ovs-ofctl -O OpenFlow13 add-flow s2
"table=0,in_port=2,eth_type=0x8847,mpls_label=111,actions=output:3"
sudo ovs-ofctl -O OpenFlow13 add-flow s2
"table=0,in_port=3,eth_type=0x8847,mpls_label=333,actions=output:2"

##### Rules at s3 #####
echo "Setting up rules at s3"

# If it came in through port 1 & it's an IPv4 packet:
# push label 32 and send it through port 2
sudo ovs-ofctl -O OpenFlow13 add-flow s3
"table=0,in_port=1,eth_type=0x800,actions=push_mpls:0x8847,
    set_field:32->mpls_label,output:2"
# If it came in through port 1 & it's an ARP packet:
# push label 03 and send it through port 2
sudo ovs-ofctl -O OpenFlow13 add-flow s3
"table=0,in_port=1,eth_type=0x806,actions=push_mpls:0x8847,
    set_field:333->mpls_label,output:2"
# If it came in through port 2 & it's an MPLS unicast & label 111:
# pop label, set ethertype as ARP and send it through port 1
sudo ovs-ofctl -O OpenFlow13 add-flow s3
"table=0,in_port=2,eth_type=0x8847,mpls_bos=1,mpls_label=111,
    actions=pop_mpls:0x0806,output:1"
# If it came in through port 2 & it's an MPLS unicast & label 23:
# pop label, set ethertype as ARP and send it through port 1
sudo ovs-ofctl -O OpenFlow13 add-flow s3
"table=0,in_port=2,eth_type=0x8847,mpls_bos=1,mpls_label=23,
    actions=pop_mpls:0x0800,output:1"
```

# Chapter 3

## Ryu

### 3.1 Introduction

Ryu is a component-based framework for Software-Defined Networking applications. It provides software components with well defined API that make it easy to create network management and control applications. Ryu supports various protocols for managing network devices, such as OpenFlow, Netconf, OF-config, etc.

Ryu supports OpenFlow 1.0, 1.2, 1.3, 1.4. It's fully developed in Python and all of the code is freely available under the Apache 2.0 license. It is the tool chosen to develop the Control Plane in the experiments exposed in this document. All of the scenarios analyzed rely on OpenFlow 1.3, Open vSwitch, Ryu and Mininet.

The resources used to document Ryu are [4] and [11]

### 3.2 Ryu application programming model

A Ryu application is a python module which defines a subclass of a class called *RyuApp*, contained in the *ryu.base.app\_manager* module. Only a single instance of a given Ryu application is supported by the manager. However, Ryu can run several applications at the same time.

#### 3.2.1 Events and Event Classes

Ryu uses an event-driven programming paradigm in which the flow of the program is determined by events. Ryu applications listen to events, events are implemented by different classes that by convention, use the prefix "Event" in their name. Events can be generated either by the Ryu core or also by Ryu applications. Applications generate their own events through methods provided by the class *RyuApp*. For example, the method *send\_event* can be used to generate an event.

On the other hand, a Ryu application can register its interest for a specific type of event by providing a handler method. In Ryu, this is programmed using a python structure called "decorator". The decorator defined in Ryu to listen to events is called *set\_ev\_cls* and it is contained in the *ryu.controller.handler* module (an example of how to handle events is shown in Section 3.6.2).

For Openflow, a module called *ryu.controller.ofp\_event* exports event classes which describe receptions of OpenFlow messages from connected datapaths (switches).

By convention, openflow messages are named as *ryu.controller.ofp\_event.EventOFPxxxx* where *xxxx* is the name of the corresponding OpenFlow message. For example, *EventOFPPacketIn* is the Event Class that corresponds to packet-in messages. The OpenFlow controller (which is part of Ryu) automatically decodes OpenFlow messages received from switches and sends these events to Ryu applications which expressed an interest using the *set\_ev\_cls* decorator.

OpenFlow event classes have at least the following attributes:

- **msg:** An object which describes the corresponding OpenFlow message.
- **msg.datapath:** A *ryu.controller.controller.Datapath* instance which describes an OpenFlow switch from which we received this OpenFlow message.

The msg object has some more additional values which are extracted from the original OpenFlow message (in Section 3.3 we provide further information about this).

### 3.2.2 Event handlers

As we said previously, the decorator *ryu.controller.handler.set\_ev\_cls(ev\_cls, dispatchers=None)* is used to declare an event handler, meaning that the decorated method will become an event handler. Where:

- The parameter *ev\_cls* is an event class whose instances this RyuApp wants to receive.
- The parameter *dispatchers* specifies one of the negotiation phases between the switch and the controller (or a list of them) for which events should be generated for this handler. The negotiation phases are:
  - **ryu.controller.handler.HANDSHAKE\_DISPATCHER:** Sending and waiting for hello message.
  - **ryu.controller.handler.CONFIG\_DISPATCHER:** Version negotiated and sent features-request message.
  - **ryu.controller.handler.MAIN\_DISPATCHER:** Switch-features message received and sent set-config message.
  - **ryu.controller.handler.DEAD\_DISPATCHER:** Disconnect from the peer. Or disconnecting due to some unrecoverable errors.

### 3.2.3 The Datapath class

The class *Datapath* describes an OpenFlow Switch connected to the controller. An instance has the following attributes/methods:

- **id:** 64-bit OpenFlow Datapath ID. This attribute is only available for *ryu.controller.handler.MAIN\_DISPATCHER* phase.
- **ofproto:** A module which exports OpenFlow definitions, mainly constants appeared in the specification, for the negotiated OpenFlow version.
- **ofproto\_parser:** A module which exports OpenFlow wire message encoder and decoder for the negotiated OpenFlow version.

- **ofproto\_parser.OFPxxxx(datapath, ....):** A callable to prepare an OpenFlow message for the given switch where xxxxx is a name of the message. For example OFPFlowMod for flow-mod message. Arguments depend on the message. The message is sent when Datapath.send\_msg() is called.
- **set\_xid(self, msg):** Generate an OpenFlow XID and put it in msg.xid.
- **send\_msg(self, msg):** Queue an OpenFlow message to send to the corresponding switch. If msg.xid is None, set\_xid is automatically called on the message before queueing.
- **send\_barrier():** Queue an OpenFlow barrier message to send to the switch.

### 3.3 OpenFlow protocol implementation

In accordance to what is exposed in chapter 1, this section aims to explain the implementation of OpenFlow 1.3 in Ryu. Ryu has two modules to handle and generate OpenFlow 1.3 messages:

- **ofproto\_v1\_3:** Contains OpenFlow definitions, mainly constants appeared in the specification of OpenFlow 1.3.
- **ofproto\_v1\_3\_parser:** This module is the implementation of OpenFlow 1.3 and contains wire message encoder and decoder for this version.

#### 3.3.1 Controller-to-Switch Messages

The *ofproto\_v1\_3\_parser* module contains the classes to implement the Controller-to-Switch Messages. Such classes have been organized in Table 3.1 with a brief description. Some of the classes listed below are Event Classes which Ryu can be told to listen to. The remaining ones are events that Ryu can generate. Further information about Controller-to-Switch Messages can be found in section 1.6.1.

Table 3.1: Controller-to-Switch Messages

Class	Type	Description
OFPFeaturesRequest	Handshake	Features request message. The controller sends a feature request to the switch upon session establishment.
OFPSwitchFeatures	Handshake	Features reply message. The switch responds with a features reply message to a features request.
OFPSetConfig	Switch configuration	The controller sends a set config request message to set configuration parameters.
OFPGetConfigRequest	Switch configuration	Get config request message. The controller sends a get config request to query configuration parameters in the switch.
OFPGetConfigReply	Switch configuration	Get config reply message. The switch responds to a configuration request with a get config reply message.
OFPTableMod	Flow Table Configuration	Flow table configuration message. The controller sends this message to configure table state.
OFPFlowMod	Modify State	Modify Flow entry message. The controller sends this message to modify the flow table

OFPPGroupMod	Modify State	Modify group entry message. The controller sends this message to modify the group table
OFPPortMod	Modify State	Port modification message. The controller sends this message to modify the behavior of the port.
OFPMeterMod	Modify State	Meter modification message. The controller sends this message to modify the meter.
OFPPDescStatsRequest	Multipart	Description statistics request message. The controller uses this message to query description of the switch.
OFPPDescStatsReply	Multipart	Description statistics reply message. The switch responds with this message to a description statistics request.
OFPPFlowStatsRequest	Multipart	Individual flow statistics request message. The controller uses this message to query individual flow statistics.
OFPPFlowStatsReply	Multipart	Individual flow statistics reply message. The switch responds with this message to an individual flow statistics request.
OFPPAggregateStatsRequest	Multipart	Aggregate flow statistics request message. The controller uses this message to query aggregate flow statistics.
OFPPAggregateStatsReply	Multipart	Aggregate flow statistics reply message. The switch responds with this message to an aggregate flow statistics request.
OFPPTableStatsRequest	Multipart	Table statistics request message. The controller uses this message to query flow table statistics.
OFPPTableStatsReply	Multipart	Table statistics reply message. The switch responds with this message to a table statistics request.
OFPPPortStatsRequest	Multipart	Port statistics request message. The controller uses this message to query information about ports statistics.
OFPPPortStatsReply	Multipart	Port statistics reply message. The switch responds with this message to a port statistics request.
OFPPPortDescStatsRequest	Multipart	Port description request message. The controller uses this message to query description of all the ports.
OFPPPortDescStatsReply	Multipart	Port description reply message. The switch responds with this message to a port description request.
OFPPQueueStatsRequest	Multipart	Queue statistics request message. The controller uses this message to query queue statistics.
OFPPQueueStatsReply	Multipart	Queue statistics reply message. The switch responds with this message to an aggregate flow statistics request.
OFPPGroupStatsRequest	Multipart	Group statistics request message. The controller uses this message to query statistics of one or more groups.

OFPGroupStatsReply	Multipart	Group statistics reply message. The switch responds with this message to a group statistics request.
OFPGroupDescStatsRequest	Multipart	Group description request message. The controller uses this message to list the set of groups on a switch.
OFPGroupDescStatsReply	Multipart	Group description reply message. The switch responds with this message to a group description request.
OFPGroupFeaturesStatsRequest	Multipart	Group features request message. The controller uses this message to list the capabilities of groups on a switch.
OFPGroupFeaturesStatsReply	Multipart	Group features reply message. The switch responds with this message to a group features request.
OFPMeterStatsRequest	Multipart	Meter statistics request message. The controller uses this message to query statistics for one or more meters.
OFPMeterStatsReply	Multipart	Meter statistics reply message. The switch responds with this message to a meter statistics request.
OFPMeterConfigStatsRequest	Multipart	Meter configuration statistics request message. The controller uses this message to query configuration for one or more meters.
OFPMeterConfigStatsReply	Multipart	Meter configuration statistics reply message. The switch responds with this message to a meter configuration statistics request.
OFPMeterFeaturesStatsRequest	Multipart	Meter features statistics request message. The controller uses this message to query the set of features of the metering subsystem.
OFPMeterFeaturesStatsReply	Multipart	Meter features statistics reply message. The switch responds with this message to a meter features statistics request.
OFPTableFeaturesStatsRequest	Multipart	Table features statistics request message. The controller uses this message to query table features.
OFPTableFeaturesStatsReply	Multipart	Table features statistics reply message. The switch responds with this message to a table features statistics request.
OFPQueueGetConfigRequest	Queue Configuration	Queue configuration request message.
OFPQueueGetConfigRepl	Queue Configuration	Queue configuration reply message. The switch responds with this message to a queue configuration request.
OFPPacketOut	Packet-Out	Packet-Out message. The controller uses this message to send a packet out through the switch.
OFPBarrierRequest	Barrier	Barrier request message. The controller sends this message to ensure message dependencies have been met or receive notifications for completed operations.
OFPBarrierReply	Barrier	Barrier reply message. The switch responds with this message to a barrier request.

OFPPRoleRequest	Role Request	Role request message. The controller uses this message to change its role.
OFPPRoleReply	Role Request	Role reply message. The switch responds with this message to a role request.
OFPPSetAsync	Asynchronous Configuration	Set asynchronous configuration message. The controller sends this message to set the asynchronous messages that it wants to receive on a given OpenFlow channel.
OFPPGetAsyncRequest	Asynchronous Configuration	Get asynchronous configuration request message. The controller uses this message to query the asynchronous message.
OFPPGetAsyncReply	Asynchronous Configuration	Get asynchronous configuration reply message. The switch responds with this message to a get asynchronous configuration request.

### 3.3.2 Asynchronous Messages

The *ofproto\_v1\_3\_parser* module also contains the classes to implement Asynchronous Messages. Such classes have been organized in Table 3.2 with a brief description. All the classes listed below correspond to messages that the switch can generate without the controller soliciting them. Further information about Asynchronous Messages can be found in section 1.6.2.

Table 3.2: Asynchronous Messages

Class	Description
OFPPacketIn	Packet-In message. The switch sends the packet that it has received to the controller, using this message.
OFPPFlowRemoved	Flow removed message. When flow entries time out or are deleted, the switch notifies controller with this message.
OFPPortStatus	Port status message. The switch notifies the controller of a change in the ports.
OFPErrormsg	Error message. The switch notifies the controller of a problem using this message.

### 3.3.3 Symmetric Messages

The *ofproto\_v1\_3\_parser* module also contains the classes to implement Symmetric Messages. Such classes have been organized in Table 3.3 with a brief description. All the classes listed below correspond to messages that can be sent in either direction without solicitation. Further information about Asynchronous Messages can be found in section 1.6.2.

Table 3.3: Symmetric Messages

Class	Description
OFPHello	Hello message. When connection is started, the hello message is exchanged between a switch and a controller.
OFPHelloElemVersionBitmap	Version bitmap Hello Element
OFPEchoRequest	Echo request message.
OFPEchoReply	Echo Reply.
OFPExperimenter	Experimenter extension message.



### 3.3.4 Flow Match Structure

Ryu uses a class contained in the *ofproto\_v1\_3\_parser* module to generate match structures that then can be added to a FlowMod message(*OFPFLOWMod*), for example. This class is called *OFPMATCH* and can take the arguments listed in Table 3.4. Further information about OpenFlow Match Fields and the matching process can be found in section 1.4.3.

Table 3.4: OFPMATCH arguments

	<b>Value</b>	<b>Description</b>
in_port	Integer 32bit	Switch input port
in_phy_port	Integer 32bit	Switch physical input port
metadata	Integer 64bit	Metadata passed between tables
eth_dst	MAC address	Ethernet destination address
eth_src	MAC address	Ethernet source address
eth_type	Integer 16bit	Ethernet frame type
vlan_vid	Integer 16bit	VLAN id
vlan_pcp	Integer 8bit	VLAN priority
ip_dscp	Integer 8bit	IP DSCP (6 bits in ToS field)
ip_ecn	Integer 8bit	IP ECN (2 bits in ToS field)
ip_proto	Integer 8bit	IP protocol
ipv4_src	IPv4 address	IPv4 source address
ipv4_dst	IPv4 address	IPv4 destination address
tcp_src	Integer 16bit	TCP source port
tcp_dst	Integer 16bit	TCP destination port
udp_src	Integer 16bit	UDP source port
udp_dst	Integer 16bit	UDP destination port
sctp_src	Integer 16bit	SCTP source port
sctp_dst	Integer 16bit	SCTP destination port
icmpv4_type	Integer 8bit	ICMP type
icmpv4_code	Integer 8bit	ICMP code
arp_op	Integer 16bit	ARP opcode
arp_spa	IPv4 address	ARP source IPv4 address
arp_tpa	IPv4 address	ARP target IPv4 address
arp_sha	MAC address	ARP source hardware address
arp_tha	MAC address	ARP target hardware address
ipv6_src	IPv6 address	IPv6 source address
ipv6_dst	IPv6 address	IPv6 destination address
ipv6_flabel	Integer 32bit	IPv6 Flow Label
icmpv6_type	Integer 8bit	ICMPv6 type
icmpv6_code	Integer 8bit	ICMPv6 code
ipv6_nd_target	IPv6 address	Target address for ND
ipv6_nd_sll	MAC address	Source link-layer for ND
ipv6_nd_tll	MAC address	Target link-layer for ND
mpls_label	Integer 32bit	MPLS label
mpls_tc	Integer 8bit	MPLS TC
mpls_bos	Integer 8bit	MPLS BoS bit
pbb_isid	Integer 24bit	PBB I-SID
tunnel_id	Integer 64bit	Logical Port Metadata
ipv6_exthdr	Integer 16bit	IPv6 Extension Header pseudo-field

### 3.3.5 Flow Instruction Structures

Ryu also uses specific classes to build OpenFlow Instructions. The classes listed in the table 3.5 are used for this purpose. Further information about OpenFlow Instructions can be found in section 1.4.4.

Table 3.5: Instruction Classes

Class	Description
OFPIInstructionGotoTable	Goto table instruction. This instruction indicates the next table in the processing pipeline.
OFPIInstructionWriteMetadata	Write metadata instruction. This instruction writes the masked metadata value into the metadata field.
OFPIInstructionActions	Actions instruction. This instruction writes/applies/clears the actions.
OFPIInstructionMeter	Meter instruction. This instruction applies the meter.

### 3.3.6 Action Structures

Ryu also provides classes to define the actions that can be written into a flow. This classes are listed in the table 3.6. Further information about OpenFlow Actions can be found in section 1.4.5.

Table 3.6: Action Classes

OFPAActionOutput	Output action. This action indicates output a packet to the switch port.
OFPAActionGroup	Group action. This action indicates the group used to process the packet.
OFPAActionSetQueue	Set queue action. This action sets the queue id that will be used to map a flow to an already-configured queue on a port.
OFPAActionSetMplsTtl	Set MPLS TTL action. This action sets the MPLS TTL.
OFPAActionDecMplsTtl	Decrement MPLS TTL action. This action decrements the MPLS TTL.
OFPAActionSetNwTtl	Set IP TTL action. This action sets the IP TTL.
OFPAActionDecNwTtl	Decrement IP TTL action. This action decrements the IP TTL.
OFPAActionCopyTtlOut	Copy TTL Out action. This action copies the TTL from the next-to-outermost header with TTL to the outermost header with TTL.
OFPAActionCopyTtlIn	Copy TTL In action. This action copies the TTL from the outermost header with TTL to the next-to-outermost header with TTL.
OFPAActionPushVlan	Push VLAN action. This action pushes a new VLAN tag to the packet.
OFPAActionPushMpls	Push MPLS action. This action pushes a new MPLS header to the packet.
OFPAActionPopVlan	Pop VLAN action. This action pops the outermost VLAN tag from the packet.
OFPAActionPopMpls	Pop MPLS action. This action pops the MPLS header from the packet.
OFPAActionSetField	Set field action. This action modifies a header field in the packet. The set of keywords available for this is same as OFPMatch.
OFPAActionExperimenter	Experimenter action. This action is an extensible action for the experimenter.

## 3.4 REST API

### 3.4.1 Introduction to REST

REST (REpresentational State Transfer) is an architectural style, and an approach to communications that is often used in the development of Web services. REST, which typically runs over HTTP, is often used in mobile applications, social networking Web sites, mashup tools, and automated business processes.

The REST style emphasizes that interactions between clients and services is enhanced by having a limited number of operations (verbs). Flexibility is provided by assigning resources (nouns) their own unique Universal Resource Identifiers (URIs). Because each verb has a specific meaning (GET, POST, PUT and DELETE), REST avoids ambiguity.

Web service APIs that adhere to the REST architectural constraints are called RESTful APIs. HTTP based RESTful APIs (or REST APIs) are defined with these aspects:

- **Base URI**, such as `http://example.com/resources/`
- **An Internet media type** for the data. This is often JSON but can be any other valid Internet media type (e.g. XML, Atom, microformats, images, etc.).
- **Standard HTTP methods**. Methods like GET, PUT, POST, or DELETE.
- **Hypertext links** to reference state.
- **hypertext links** to reference related resources.

In the case of Ryu, a REST API implements the interface between the Application Plane and the Control Plane. This interface is known in SDN as Northbound Interface (NBI). See section 1.1.2 for more information about the SDN architecture.

### 3.4.2 Building REST APIs with Ryu

Ryu's *ControllerBase* class is the base class used to generate user defined APIs. It will handle the incoming HTTP connections through a **Web Server Gateway Interface** (WSGI). WSGI is a specification for simple and universal interfaces between web servers and web applications or frameworks for Python.

A class that extends from *ControllerBase* will be able to handle HTTP Requests that arrive to the WSGI. The requests will be filtered by their URL and their methods and Ryu will decide which method is called. To indicate which URL and methods will be matched the class' methods must be decorated with the *route* decorator.

The decorator *route* accepts two positional arguments and two key-word arguments:

- **Request name**: It's only an identifier string to name the resource. It doesn't have any further implications.
- **URL**: It contains a string defining the URL to match. It doesn't contain the domain nor the first part of the URI (protocol://ip\_address:port). To define variable URL parts you can use brackets '{' and '}'. Inside this brackets you have to specify a representative name, it doesn't matter which one do you use, you'll use it only to identify the substring contained. You'll be able to access the substring wrapped in braces inside the method by calling `'kwargs['name']`.
- **methods=[]**: It contains an array with all the method(s) that this method will listen to.

- `requirements={}`: It contains a dictionary whose keys are the names defined in the URL (inside the brackets) and the values are patterns. It forces the substring identified by the key to match the pattern contained in the value.

The methods decorated with *route* should take two arguments: a request argument, and a key-word argument. They have to return a Response object from the 'webob' package.

### 3.4.3 Linking REST Controllers with Ryu applications

The rest linkage of a certain Ryu application with a web interface is made outside the *RyuApp* class. As explained before, a controller class must be created to handle requests and responses.

To link a controller with an application a WSGI object is used. This object is created by Ryu and stored in the key-words argument. It is accessible through the key 'wsgi'. This object allows registering controllers for the Application's API.

The registered controller will receive the incoming requests and will have a reference to the instance of the *RyuApp* class. The controller will be able to call the application methods through this reference.

REST API's are not documented in detail in this document, as it's out of the scope of this project. However, they are used in some of the applications discussed in chapter 4

## 3.5 Components

All the components of Ryu are listed in Table 3.7 with a brief description.

Table 3.7: Components of Ryu

Component	Description
<code>ryu.base.app_manager</code>	The central management of Ryu applications. Loads Ryu applications. Provides contexts to Ryu applications. Routes messages among Ryu applications.
<code>ryu.controller.controller</code>	The main component of OpenFlow controller. Handles connections from switches. Generates and routes events to appropriate entities like Ryu applications.
<code>ryu.controller.dpset</code>	Manages switches. Planned to be replaced by <code>ryu/topology</code> .
<code>ryu.controller.ofp_event</code>	OpenFlow event definitions.
<code>ryu.controller.ofp_handler</code>	Basic OpenFlow handling including negotiation.
<code>ryu.ofproto.ofproto_v1_0</code>	OpenFlow 1.0 definitions.
<code>ryu.ofproto.ofproto_v1_0_parser</code>	Decoder/Encoder implementations of OpenFlow 1.0.
<code>ryu.ofproto.ofproto_v1_2</code>	OpenFlow 1.2 definitions.
<code>ryu.ofproto.ofproto_v1_2_parser</code>	Decoder/Encoder implementations of OpenFlow 1.2.
<code>ryu.ofproto.ofproto_v1_3</code>	OpenFlow 1.3 definitions.
<code>ryu.ofproto.ofproto_v1_3_parser</code>	This module implements OpenFlow 1.3.x. This module also implements some of extensions
<code>ryu.ofproto.ofproto_v1_4</code>	OpenFlow 1.4 definitions.
<code>ryu.ofproto.ofproto_v1_4_parser</code>	Decoder/Encoder implementations of OpenFlow 1.4.

ryu.topology	Switch and link discovery module. Planned to replace ryu/controller/dpset.
ryu.lib.packet	Ryu packet library. Decoder/Encoder implementations of popular protocols like TCP/IP.
ryu.lib.ovs	ovsdb interaction library.
ryu.lib.of_config	OF-Config implementation.
ryu.lib.netconf	NETCONF definitions used by ryu/lib/of_config.
ryu.lib.xflow	An implementation of sFlow and NetFlow.
ryu.contrib.ovs	Open vSwitch python binding. Used by ryu.lib.ovs.
ryu.contrib.oslo.config	Oslo configuration library. Used for ryu-manager's command-line options and configuration files.
ryu.contrib.ncclient	Python library for NETCONF client. Used by ryu.lib.of_config.

Besides this components, Ryu als provides some useful applications contained in **ryu.app**

### 3.6 Analysis of a Switch implemented with Ryu

The purpose of this section is to analyze, explain and test a simple learning switch implemented on Ryu, using OpenFlow 1.3. In addition, we will cover some of the important aspects of Ryu applications, so the code can be better understood.

A switching hub (also called switch) has the following functionalities:

- Learns the MAC address of the host connected to a port and retains it in the MAC address table.
- When receiving packets addressed to a host already learned, transfers them to the port connected to the host.
- When receiving packets addressed to an unknown host, performs flooding.

It is possible to achieve a learning switch using OpenFlow, an OpenFlow controller and an OpenFlow switch (We use Open vSwitch).

First of all, the **Packet-In** function is used to learn MAC addresses. The controller can use the Packet-In function to receive packets from the switch. Then, it analyzes the received packets to learn the MAC address of the host and information about the connected port.

After learning, the controller transfers the received packets. The controller investigates whether the **destination MAC address** of the packets belong to a learned host. Depending on the investigation results, the controller performs the following processing:

- If the host is already a **learned host**: Uses the **Packet-Out** function to transfer the packets to **the corresponding port**.
- If the host is unknown host: Use the **Packet-Out** function to perform **flooding**.

This section comments different blocks of code used in this implementation. The entire source code can be found in Section 3.6.6.

### 3.6.1 Class Definition and Initialization

In order to implement the switch intelligence as a Ryu application, *ryu.base.app\_manager.RyuApp* has to be inherited, as explained in section 3.2. Also, to use OpenFlow 1.3, the OpenFlow 1.3 version is specified for *OFP\_VERSIONS*. Also, a table called *mac\_to\_port* is defined, in order to relate MAC addresses and OpenFlow ports (corresponding to switch interfaces as explained in section 1.3.1).

In the OpenFlow protocol, some procedures such as handshake required for communication between the OpenFlow switch and the controller have been defined. However, Ryu's framework takes care of those procedures thus it is not necessary to be aware of those in Ryu applications.

```
class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
```

### 3.6.2 Event Handler

As explained in section 3.2.2, when an OpenFlow message is received, an event corresponding to the message is generated. The Ryu application implements an event handler corresponding to the message desired to be received. The event handler defines a function having the event object for the argument and use the *ryu.controller.handler.set\_ev\_cls* decorator to decorate the handler to listen to a certain event. The *set\_ev\_cls* decorator specifies the event class supporting the received message and the state of the OpenFlow switch for the argument.

The event class name is *ryu.controller.ofp\_event.EventOFP+Message\_name*. For example, in case of a Packet-In message, it becomes *EventOFPPacketIn*. The complete list of OpenFlow messages can be found in section 3.3. The second argument is the so called dispatcher. Dispatchers argument specifies one of the following negotiation phases (or a list of them) for which events should be generated for this handler. The list of dispatchers can be found in section 3.2.2. In the following piece of code, we add a decorator to the *switch\_features\_handler* function to manage the table-miss flow entry:

```
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
    datapath = ev.msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    # NO BUFFER specified to max_len due to an OVS bug
    # The bug has been fixed in OVS v2.1.0.
    match = parser.OFPMatch()
    actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                     ofproto.OFPCML_NO_BUFFER)]
    self.add_flow(datapath, 0, match, actions)
```

### Adding Table-miss Flow Entry

After the handshake with the OpenFlow switch is completed, the **Table-miss flow entry** (See section 1.4.6) is added to the flow table to get ready to receive the Packet-In message. Specifically, upon receiving the Switch Features Reply

message, the Table-miss flow entry is added to the switch using the *add\_flow()* method, which is explained in section 3.6.3.

The instance of the OpenFlow message class corresponding to the event (in this case, *OFPSwitchFeatures*) is stored in *ev.msg*. The instance of the *ryu.controller.controller.Datapath* class corresponding to the OpenFlow switch that issued this message is stored in *msg.datapath*.

The Datapath class is used to perform important processing such as actual communication. As we have already commented, the datapath attribute ofproto indicates the *ofproto module* of the OpenFlow version being used (this module exports OpenFlow definitions, mainly OpenFlow constants). On the other hand, the attribute *ofproto\_parser* indicates the *ofproto\_parser module* of the OpenFlow version being used (this module exports the message encoder and decoder).

It is worth to mention that this switch does not particularly use the received Switch Features message itself. In fact, this message is used as an event to obtain a correct timing to add the Table-miss flow entry. Note also that an empty match is generated to match all packets. The Match is expressed using the *OFPMatch* class.

Next, an instance of the class *OFPACTIONOutput* is generated to configure the Table-miss flow. Instances are created using the following parameters: *OFPACTIONOutput(port, max\_len)*. In this case, the port is the controller port (*OFPP\_CONTROLLER*). The max\_len parameter indicates the maximum amount of data from a packet that should be sent to the controller. In this case, the max\_len is set to *OFPCML\_NO\_BUFFER*, which means that the entire packet should be sent to the controller. This is done because in versions of OVS prior to v2.1.0 there is a bug that may cause that if we specify a lesser number, e.g., 128, OVS will send a packet with an invalid buffer\_id and truncated packet data.

## Packet-in Message: Updating the MAC Address Table

The Packet-In event handler processes the packets that match (only) the Table-miss entry at the switch.

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    # If you hit this you might want to increase
    # the "miss_send_length" of your switch
    if ev.msg.msg_len < ev.msg.total_len:
        self.logger.debug("packet truncated: only %s of %s bytes",
                          ev.msg.msg_len, ev.msg.total_len)

    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']

    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]

    dst = eth.dst
    src = eth.src

    dpid = datapath.id
    self.mac_to_port.setdefault(dpid, {})
```

```

self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)

# learn a mac address to avoid FLOOD next time.
self.mac_to_port[dpid][src] = in_port
# ...

```

This code gets the input port (`in_port`) from the **OFPPacketIn** match. The **destination MAC address** and **sender MAC address** are obtained from the Ethernet header of the received packets using Ryu's packet library.

Based on the acquired sender MAC address and received port number, the MAC address table is updated. To support connection with multiple OpenFlow switches, the MAC address table is designed to be managed for each OpenFlow switch. The datapath ID is used to identify each OpenFlow switch and its corresponding MAC address table.

## Packet-in Message: Judging the Transfer Destination Port

The corresponding port number is used when the destination MAC address exists in the corresponding MAC address table. If not found, the instance of the `OFPACTIONOutput` class specifies flooding (**OFPP\_FLOOD**) as output port.

```

def _packet_in_handler(self, ev):
    # ...
    if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
    else:
        out_port = ofproto.OFPP_FLOOD

    actions = [parser.OFPACTIONOutput(out_port)]

    # install a flow to avoid packet_in next time
    if out_port != ofproto.OFPP_FLOOD:
        match = parser.OFPMATCH(in_port=in_port, eth_dst=dst)
        # verify if we have a valid buffer_id, if yes avoid to send both
        # flow_mod & packet_out
        if msg.buffer_id != ofproto.OFP_NO_BUFFER:
            self.add_flow(datapath, 1, match, actions, msg.buffer_id)
            return
        else:
            self.add_flow(datapath, 1, match, actions)
    # ...

```

If the destination MAC address is found, an entry is added to the flow table of the OpenFlow switch. As with addition of the Table-miss flow entry, a match and an action are specified and **add\_flow()** is executed to add a flow entry.

Unlike the Table-miss flow entry, conditions for match are set. In order to implement the switching hub, the receive port (**in\_port**) and destination MAC address (**eth\_dst**) have been specified.

The priority is specified to 1. The greater the value, the higher the priority, therefore, the flow entry added here will be evaluated before the Table-miss flow entry.





```
datapath.send_msg(mod)
```

Finally, the function adds an entry to the flow table by issuing the Flow Mod message. The class corresponding to the Flow Mod message is the *OFPPFlowMod* class. The instance of the *OFPPFlowMod* class is generated and the message is sent to the OpenFlow switch using the *Datapath.send\_msg()* method.

There are many arguments in the constructor of the *OFPPFlowMod* class. Many of them generally can be left blank so default values are used.

The *buffer\_id* refers to a packet buffered at the switch and sent to the controller by a packet-in message. A flow mod that includes a valid *buffer\_id* is effectively equivalent to sending a two-message sequence of a flow mod and a packet-out to **OFPP\_TABLE**, with the requirement that the switch must fully process the flow mod before the packet out. These semantics apply regardless of the table to which the flow mod refers, or the instructions contained in the flow mod.

### 3.6.4 Testing the application

Now that the source code has been studied and explained, let's test the application.

In order to execute it, we will use a mininet Virtual Machine to virtualize the network. For this experiment we will use a simple topology of 3 hosts connected to an Open vSwitch, controlled by the Ryu Application. we execute the following command in a terminal to create such network:

```
sudo mn --topo single,3 --mac --switch ovsk --controller remote
```

This prompts mininet's Command Line Interface.

Then, we execute wireshark in another terminal, to analyze its behaviour:

```
sudo wireshark &
```

We start capturing packets on the loopback interface, with the **of** filter to see only OpenFlow messages.

We can get some interesting information about the status of the switch using **ovs-vsctl show** and **ovs-dpctl show** commands:

```
sudo ovs-vsctl show

8945dad2-35b4-4549-9122-23d5558992e7
    Bridge "s1"
        Controller "tcp:6634"
        Controller "tcp:127.0.0.1:6633"
        fail_mode: secure
        Port "s1-eth1"
            Interface "s1-eth1"
        Port "s1"
            Interface "s1"
                type: internal
        Port "s1-eth3"
            Interface "s1-eth3"
        Port "s1-eth2"
            Interface "s1-eth2"
```

```
ovs_version: "2.0.2"

sudo ovs-dpctl show
system@ovs-system:
    lookups: hit:11 missed:6 lost:0
    flows: 0
    port 0: ovs-system (internal)
    port 1: s1-eth3
    port 2: s1-eth1
    port 3: s1 (internal)
    port 4: s1-eth2
```

We can see the version of the switch, number of flows and the interface assigned to each port.

In order to force the switch to use OpenFlow 1.3 we must run the following command:

```
sudo ovs-vsctl set Bridge s1 protocols=OpenFlow13
```

This line is really important if we want the application to work with the 1.3 version.  
It is time to start the Ryu application:

```
ryu-manager --verbose ~/apps/switching_hub.py
```

The application starts its execution and returns the following messages

```
loading app /home/mininet/apps/switching_hub.py
loading app ryu.controller.ofp_handler
instantiating app /home/mininet/apps/switching_hub.py of SimpleSwitch13
instantiating app ryu.controller.ofp_handler of OFPHandler
BRICK SimpleSwitch13
    CONSUMES EventOFPPacketIn
    CONSUMES EventOFPSwitchFeatures
BRICK ofp_event
    PROVIDES EventOFPPacketIn TO {'SimpleSwitch13': set(['main'])}
    PROVIDES EventOFPSwitchFeatures TO {'SimpleSwitch13': set(['config'])}
    CONSUMES EventOFPEchoRequest
    CONSUMES EventOFPErrormsg
    CONSUMES EventOFPSwitchFeatures
    CONSUMES EventOFPHello
    CONSUMES EventOFPPortDescStatsReply
connected socket:<eventlet.greenio.base.GreenSocket object at 0xb672348c>
address: ('127.0.0.1', 32891)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0xb672376c>
move onto config mode
EVENT ofp_event->SimpleSwitch13 EventOFPSwitchFeatures
switch features ev version: 0x4 msg_type 0x6 xid 0xcf40c658
    OFPSwitchFeatures(auxiliary_id=0, capabilities=71, datapath_id=1,
        n_buffers=256, n_tables=254)
move onto main mode
```

No.	Time	Source	Destination	Protocol	Length	Info
15966	2294.519736000	127.0.0.1	127.0.0.1	OF 1.3	82	of hello
15968	2294.531052000	127.0.0.1	127.0.0.1	OF 1.3	74	of hello
15970	2294.533301000	127.0.0.1	127.0.0.1	OF 1.3	74	of features_request
15972	2294.534091000	127.0.0.1	127.0.0.1	OF 1.3	98	of features_reply
15973	2294.542181000	127.0.0.1	127.0.0.1	OF 1.3	78	of set_config
15974	2294.543518000	127.0.0.1	127.0.0.1	OF 1.3	82	of port_desc_stats_request
15976	2294.544363000	127.0.0.1	127.0.0.1	OF 1.3	338	of port_desc_stats_reply
15977	2294.545316000	127.0.0.1	127.0.0.1	OF 1.3	146	of flow_add
16011	2299.516068000	127.0.0.1	127.0.0.1	OF 1.3	74	of echo_request
16012	2299.517601000	127.0.0.1	127.0.0.1	OF 1.3	74	of echo_reply
16046	2304.515559000	127.0.0.1	127.0.0.1	OF 1.3	74	of echo_request

Figure 3.1: Wireshark trace

As it can be seen in the figure 3.1, the application has successfully completed the handshake and the exchange of configuration information with the switch. Note that the Table-miss flow entry has already been added during this process, as expected. Further information about the messages exchanged between the SDN Controller and the OpenFlow Switch can be found in section 1.6.

As the switch and the controller exchange echo requests/replies every few seconds, we will filter this messages in order to better analyze the application. For this purpose we use *of and not(of13.echo\_request.type or of13.echo\_reply.type)* expression as a filter.

### Simple ping

As a demonstration of the application let's make one host send an ICMP echo request (ping) to other host. The process should go as follows:

- Host h1 sends a broadcast ARP request to discover the MAC address of h2.
- Host h2 answers the ARP request.
- Host h1 sends ICMP echo request to h2.
- Host h2 sends ICMP echo reply to h1.
- Host h1 sends a unicast ARP request to h2 for ARP cache validation.
- Host h2 answers with an ARP reply to h1.

Before running the experiment let's open an xterm for each host from mininet's CLI:

```
mininet> xterm h1 h2 h3
```

Now we run the following command in each host to analyze the traffic. For instance, in h1:

```
tcpdump -XX -n -i h1-eth0
```

This will show which packets sees each host. Now, from mininet's CLI we send a ping from h1 to h2

```
mininet> h1 ping -c 1 h2
```

```

"Node: h1" (on mininet-vm)
root@mininet-vm:~/apps# tcpdump -XX -n -i h1-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h1-eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
11:53:11.274561 ARP, Request who-has 10.0.0.2 tell 10.0.0.1, length 28
0x0000: ffff ffff ffff 0000 0000 0001 0806 0001 .....
0x0010: 0800 0604 0001 0000 0000 0001 0a00 0001 .....
0x0020: 0000 0000 0000 0a00 0002 .....
11:53:11.238146 ARP, Reply 10.0.0.2 is-at 00:00:00:00:00:02, length 28
0x0000: 0000 0000 0001 0000 0000 0002 0806 0001 .....
0x0010: 0800 0604 0002 0000 0000 0002 0a00 0002 .....
0x0020: 0800 0800 0001 0a00 0001 .....
11:53:11.238286 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 5362, seq 1, length 64
0x0000: 0000 0000 0002 0000 0000 0001 0800 4500 .....E.
0x0010: 0054 4970 4000 4001 dd36 0a00 0001 0a00 .....T.p8.0..6.....
0x0020: 0002 0800 6e51 14f2 0001 1733 2055 4e30 .....n0....3.UN0
0x0030: 0400 0809 0a0b 0c0d 0e0f 1011 1213 1415 .....
0x0040: 1617 1819 1a1b 1c1d 1e1f 2021 2223 2425 .....!##$%
0x0050: 2627 2829 2a2b 2c2d 2e2f 3031 3233 3435 &'()*+,-./012345
0x0060: 3637 67
11:53:11.305691 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 5362, seq 1, length 64
0x0000: 0000 0000 0001 0000 0000 0002 0800 4500 .....E.
0x0010: 0054 4ca7 0000 4001 5a00 0a00 0002 0a00 .....T...e.Z.....
0x0020: 0001 0000 7851 14f2 0001 1733 2055 4e30 .....0....3.UN0
0x0030: 0400 0809 0a0b 0c0d 0e0f 1011 1213 1415 .....
0x0040: 1617 1819 1a1b 1c1d 1e1f 2021 2223 2425 .....!##$%
0x0050: 2627 2829 2a2b 2c2d 2e2f 3031 3233 3435 &'()*+,-./012345
0x0060: 3637 67
11:53:16.311085 ARP, Request who-has 10.0.0.1 tell 10.0.0.2, length 28
0x0000: 0000 0000 0001 0000 0000 0002 0806 0001 .....
0x0010: 0800 0604 0001 0000 0000 0002 0a00 0002 .....
0x0020: 0000 0000 0000 0a00 0001 .....
11:53:16.311409 ARP, Reply 10.0.0.1 is-at 00:00:00:00:00:01, length 28
0x0000: 0000 0000 0002 0000 0000 0001 0806 0001 .....
0x0010: 0800 0604 0002 0000 0000 0001 0a00 0001 .....
0x0020: 0000 0000 0002 0a00 0002 .....

"Node: h2" (on mininet-vm)
root@mininet-vm:~/apps# tcpdump -XX -n -i h2-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h2-eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
11:53:11.284423 ARP, Request who-has 10.0.0.2 tell 10.0.0.1, length 28
0x0000: ffff ffff ffff 0000 0000 0001 0806 0001 .....
0x0010: 0800 0604 0001 0000 0000 0001 0a00 0001 .....
0x0020: 0000 0000 0000 0a00 0002 .....
11:53:11.284603 ARP, Reply 10.0.0.2 is-at 00:00:00:00:00:02, length 28
0x0000: 0000 0000 0001 0000 0000 0002 0806 0001 .....
0x0010: 0800 0604 0002 0000 0000 0002 0a00 0002 .....
0x0020: 0000 0000 0001 0a00 0001 .....
11:53:11.305693 IP 10.0.0.1 > 10.0.0.2: ICMP echo request, id 5362, seq 1, length 64
0x0000: 0000 0000 0002 0000 0000 0001 0800 4500 .....E.
0x0010: 0054 4970 4000 4001 dd36 0a00 0001 0a00 .....T.p8.0..6.....
0x0020: 0002 0800 6e51 14f2 0001 1733 2055 4e30 .....n0....3.UN0
0x0030: 0400 0809 0a0b 0c0d 0e0f 1011 1213 1415 .....
0x0040: 1617 1819 1a1b 1c1d 1e1f 2021 2223 2425 .....!##$%
0x0050: 2627 2829 2a2b 2c2d 2e2f 3031 3233 3435 &'()*+,-./012345
0x0060: 3637 67
11:53:11.305691 IP 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 5362, seq 1, length 64
0x0000: 0000 0000 0001 0000 0000 0002 0800 4500 .....E.
0x0010: 0054 4ca7 0000 4001 5a00 0a00 0002 0a00 .....T...e.Z.....
0x0020: 0001 0000 7851 14f2 0001 1733 2055 4e30 .....0....3.UN0
0x0030: 0400 0809 0a0b 0c0d 0e0f 1011 1213 1415 .....
0x0040: 1617 1819 1a1b 1c1d 1e1f 2021 2223 2425 .....!##$%
0x0050: 2627 2829 2a2b 2c2d 2e2f 3031 3233 3435 &'()*+,-./012345
0x0060: 3637 67
11:53:16.310940 ARP, Request who-has 10.0.0.1 tell 10.0.0.2, length 28
0x0000: 0000 0000 0001 0000 0000 0002 0806 0001 .....
0x0010: 0800 0604 0001 0000 0000 0002 0a00 0002 .....
0x0020: 0000 0000 0000 0a00 0001 .....
11:53:16.311509 ARP, Reply 10.0.0.1 is-at 00:00:00:00:00:01, length 28
0x0000: 0000 0000 0002 0000 0000 0001 0806 0001 .....
0x0010: 0800 0604 0002 0000 0000 0001 0a00 0001 .....
0x0020: 0000 0000 0002 0a00 0002 .....

"Node: h3" (on mininet-vm)
root@mininet-vm:~/apps# tcpdump -XX -n -i h3-eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on h3-eth0, link-type EN10MB (Ethernet), capture size 65535 bytes
11:53:11.284571 ARP, Request who-has 10.0.0.2 tell 10.0.0.1, length 28
0x0000: ffff ffff ffff 0000 0000 0001 0806 0001 .....
0x0010: 0800 0604 0001 0000 0000 0001 0a00 0001 .....
0x0020: 0000 0000 0000 0a00 0002 .....

```

Figure 3.2: TCP dump

As shown in Figure 3.2, the application is transparent to the hosts. Hosts h1 and h2 see the whole process while host h3 only sees the broadcast message. It seems that the application works just like a classic switch. Now let's take a look at the flows that have been added to the switch:

```

sudo ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=456.818s, table=0, n_packets=2, n_bytes=140,
   priority=1,in_port=2, dl_dst=00:00:00:00:00:01 actions=output:1
 cookie=0x0, duration=456.81s, table=0, n_packets=1, n_bytes=42,
   priority=1,in_port=1, dl_dst=00:00:00:00:00:02 actions=output:2
 cookie=0x0, duration=1652.403s, table=0, n_packets=3, n_bytes=182,
   priority=0 actions=CONTROLLER:65535

```

Three flows have been added to the switch: The Table-miss flow entry, with priority 0; and one for each of the hosts with priority 1. The two flows with the higher priority correspond to the packets that come from h1 and target h2 and viceversa.

No.	Time	Source	Destination	Protocol	Length	Info
3598	291.322531000	127.0.0.1	127.0.0.1	OF 1.3	74	of_hello
3600	291.323209000	127.0.0.1	127.0.0.1	OF 1.3	82	of_hello
3602	291.326722000	127.0.0.1	127.0.0.1	OF 1.3	74	of_features_request
3603	291.327136000	127.0.0.1	127.0.0.1	OF 1.3	98	of_features_reply
3604	291.331479000	127.0.0.1	127.0.0.1	OF 1.3	78	of_set_config
3605	291.334514000	127.0.0.1	127.0.0.1	OF 1.3	82	of_port_desc_stats_request
3607	291.335193000	127.0.0.1	127.0.0.1	OF 1.3	338	of_port_desc_stats_reply
3608	291.335675000	127.0.0.1	127.0.0.1	OF 1.3	146	of_flow_add
5811	587.837195000	00:00:00_00:00:01	Broadcast	OF 1.3	150	of_packet_in
5812	587.843687000	00:00:00_00:00:01	Broadcast	OF 1.3	148	of_packet_out
5814	587.845325000	00:00:00_00:00:02	00:00:00_00:00:01	OF 1.3	150	of_packet_in
5815	587.850381000	127.0.0.1	127.0.0.1	OF 1.3	162	of_flow_add
5816	587.851443000	00:00:00_00:00:02	00:00:00_00:00:01	OF 1.3	148	of_packet_out
5818	587.852830000	10.0.0.1	10.0.0.2	OF 1.3	206	of_packet_in
5819	587.858825000	127.0.0.1	127.0.0.1	OF 1.3	162	of_flow_add
5820	587.860171000	10.0.0.1	10.0.0.2	OF 1.3	204	of_packet_out

Figure 3.3: The wireshark trace shows the message exchange between the controller and the switch

In Figure 3.3 we can see the following process:

- Packets 3598 through 3607: OpenFlow 1.3 handshake and configuration messages.
- Packet 3608: Table-miss FLOW\_MOD message. This flow matches every packet and has the lowest priority.
- Packet 5811: The switch received a broadcast packet from h1 (ARP request) that only matches the Table-miss flow. Sends the packet to the controller.
- Packet 5812: The controller returns the package and tells the switch to broadcast it.
- Packet 5814: The switch received a packet from h2 (ARP reply) that only matches the Table-miss flow. The destination of this packet is h1. Sends the packet to the controller.
- Packet 5815: The controller sends a flow\_mod message, telling the switch to write a flow that matches the packets coming from port 2 targeting h1's MAC address. This flow indicates that the action to perform when handling these packets is to send them through port 1.
- Packet 5816: The controller sends the packet back to the switch and tells it to send it through port 1.
- Packet 5818: The switch received a packet from h1 (ICMP echo request) that only matches the Table-miss flow. The destination of this packet is h2. Sends the packet to the controller.
- Packet 5819: The controller sends a flow\_mod message, telling the switch to write a flow that matches the packets coming from port 1 targeting h2's MAC address. This flow indicates that the action to perform when handling these packets is to send them through port 2.
- Packet 5820: The controller sends the packet back to the switch and tells it to send it through port 2.

No more communication is required between the controller and the switch for the rest of the experiment, since the packets left (echo reply and ARP cache validation) match one of the two flows written in the switch flow table.

### 3.6.5 Discussion on alternative implementations

In this section we will try to simplify the code of the switch maintaining its functionalities.

## Removing table-miss flow entry

The OpenFlow 1.3 Switch Specification document states that every flow table must support a table-miss flow entry to process table misses. The table-miss flow entry specifies how to process packets unmatched by other flow entries in the flow table, and may, for example send packets to the controller, drop packets or direct packets to a subsequent table.

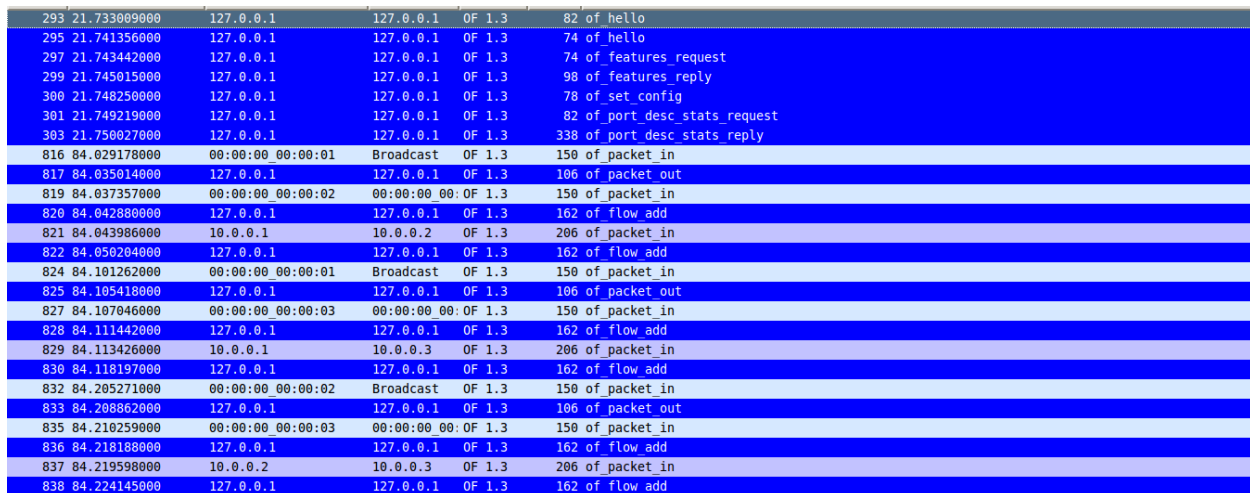
This document also states that if the table-miss flow entry does not exist, by default packets unmatched by flow entries are dropped (discarded). **A switch configuration, for example using the OpenFlow Configuration Protocol, may override this default and specify another behaviour.**

In this experiment we have removed the table-miss flow\_mod message from the code, and launched a pingall instruction on the mininet environment. We can see that the switch asks the controller every time a packet does not match any flow entry. At the end of the experiment, the switch has enough flow entries to deliver every packet to every host correctly.

Flow table:

```
cookie=0x0, duration=6.038s, table=0, n_packets=4, n_bytes=280,
  priority=1, in_port=3, dl_dst=00:00:00:00:00:02 actions=output:2
cookie=0x0, duration=6.213s, table=0, n_packets=4, n_bytes=280,
  priority=1, in_port=2, dl_dst=00:00:00:00:00:01 actions=output:1
cookie=0x0, duration=6.145s, table=0, n_packets=4, n_bytes=280,
  priority=1, in_port=3, dl_dst=00:00:00:00:00:01 actions=output:1
cookie=0x0, duration=6.032s, 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=6.138s, table=0, n_packets=3, n_bytes=238,
  priority=1, in_port=1, dl_dst=00:00:00:00:00:03 actions=output:3
cookie=0x0, duration=6.206s, table=0, n_packets=3, n_bytes=238,
  priority=1, in_port=1, dl_dst=00:00:00:00:00:02 actions=output:2
```

Running a pingall instruction in the mininet command line leaves this trace on wireshark:



293	21.733009000	127.0.0.1	127.0.0.1	OF 1.3	82 of hello
295	21.741356000	127.0.0.1	127.0.0.1	OF 1.3	74 of hello
297	21.743442000	127.0.0.1	127.0.0.1	OF 1.3	74 of features request
299	21.745015000	127.0.0.1	127.0.0.1	OF 1.3	98 of features reply
300	21.748250000	127.0.0.1	127.0.0.1	OF 1.3	78 of set config
301	21.749219000	127.0.0.1	127.0.0.1	OF 1.3	82 of port_desc_stats request
303	21.750027000	127.0.0.1	127.0.0.1	OF 1.3	338 of port_desc_stats reply
816	84.029178000	00:00:00:00:00:01	Broadcast	OF 1.3	150 of packet in
817	84.035014000	127.0.0.1	127.0.0.1	OF 1.3	106 of packet out
819	84.037357000	00:00:00:00:00:02	00:00:00:00:00	OF 1.3	150 of packet in
820	84.042880000	127.0.0.1	127.0.0.1	OF 1.3	162 of flow add
821	84.043986000	10.0.0.1	10.0.0.2	OF 1.3	206 of packet in
822	84.050204000	127.0.0.1	127.0.0.1	OF 1.3	162 of flow add
824	84.101262000	00:00:00:00:00:01	Broadcast	OF 1.3	150 of packet in
825	84.105418000	127.0.0.1	127.0.0.1	OF 1.3	106 of packet out
827	84.107046000	00:00:00:00:00:03	00:00:00:00:00	OF 1.3	150 of packet in
828	84.111442000	127.0.0.1	127.0.0.1	OF 1.3	162 of flow add
829	84.113426000	10.0.0.1	10.0.0.3	OF 1.3	206 of packet in
830	84.118197000	127.0.0.1	127.0.0.1	OF 1.3	162 of flow add
832	84.205271000	00:00:00:00:00:02	Broadcast	OF 1.3	150 of packet in
833	84.208862000	127.0.0.1	127.0.0.1	OF 1.3	106 of packet out
835	84.210259000	00:00:00:00:00:03	00:00:00:00:00	OF 1.3	150 of packet in
836	84.218188000	127.0.0.1	127.0.0.1	OF 1.3	162 of flow add
837	84.219598000	10.0.0.2	10.0.0.3	OF 1.3	206 of packet in
838	84.224145000	127.0.0.1	127.0.0.1	OF 1.3	162 of flow add

Figure 3.4: Wireshark trace after removing the table-miss entry

As you can see, it only sends the packet\_out message when a flood has to be performed. In figure 3.5 we show how the switch performs with the original code.

10611	988.124324000	00:00:00_00:00:01	Broadcast	OF 1.3	150 of packet_in
10612	988.131824000	00:00:00_00:00:01	Broadcast	OF 1.3	148 of packet_out
10614	988.133587000	00:00:00_00:00:02	00:00:00_00:00:02	OF 1.3	150 of packet_in
10615	988.137418000	127.0.0.1	127.0.0.1	OF 1.3	162 of flow add
10616	988.138609000	00:00:00_00:00:02	00:00:00_00:00:02	OF 1.3	148 of packet_out
10618	988.141403000	10.0.0.1	10.0.0.2	OF 1.3	206 of packet_in
10619	988.146581000	127.0.0.1	127.0.0.1	OF 1.3	162 of flow add
10620	988.148442000	10.0.0.1	10.0.0.2	OF 1.3	204 of packet_out
10622	988.178166000	00:00:00_00:00:01	Broadcast	OF 1.3	150 of packet_in
10623	988.181336000	00:00:00_00:00:01	Broadcast	OF 1.3	148 of packet_out
10624	988.186533000	00:00:00_00:00:03	00:00:00_00:00:03	OF 1.3	150 of packet_in
10625	988.190220000	127.0.0.1	127.0.0.1	OF 1.3	162 of flow add
10626	988.191222000	00:00:00_00:00:03	00:00:00_00:00:03	OF 1.3	148 of packet_out
10628	988.192715000	10.0.0.1	10.0.0.3	OF 1.3	206 of packet_in
10629	988.196427000	127.0.0.1	127.0.0.1	OF 1.3	162 of flow add
10630	988.197993000	10.0.0.1	10.0.0.3	OF 1.3	204 of packet_out
10632	988.246223000	00:00:00_00:00:02	Broadcast	OF 1.3	150 of packet_in
10633	988.249670000	00:00:00_00:00:02	Broadcast	OF 1.3	148 of packet_out
10634	988.251178000	00:00:00_00:00:03	00:00:00_00:00:03	OF 1.3	150 of packet_in
10635	988.254069000	127.0.0.1	127.0.0.1	OF 1.3	162 of flow add
10636	988.254956000	00:00:00_00:00:03	00:00:00_00:00:03	OF 1.3	148 of packet_out
10638	988.256469000	10.0.0.2	10.0.0.3	OF 1.3	206 of packet_in
10639	988.259417000	127.0.0.1	127.0.0.1	OF 1.3	162 of flow add
10640	988.261398000	10.0.0.2	10.0.0.3	OF 1.3	204 of packet_out

Figure 3.5: Wireshark trace without removing the table-miss entry

Both traces are the same, but with the above modification we have reduced significantly the amount of packet\_out messages. Theoretically, the Open vSwitch should drop every packet that does not match any of the flows, instead of sending them to the controller. This means that the configuration of the switch has been changed to behave this way.

Also, it seems that the way Open vSwitch is implemented, it buffers the packet (Creates a *buffer\_id*) so when the *FlowMod* message arrives from the controller, it does not need a *Packet Out* message to forward the packet. Also, the application is implemented so if a *buffer\_id* is found, no *Packet Out* message is sent, as it does not contain the packet where the actions have to be applied (*data = None*).

### Matching only the destination

One thing that we observed is that a new flow is created for each pair of hosts: sender and receiver. We wonder if the switch could only store the relation between a MAC address and a port. This should reduce the number of flows significantly.

For this purpose we will modify the match in the Packet-In event handler so the field to match is only the destination address.

```
match = parser.OFPMatch(eth_dst=dst)
```

Then we execute the pingall instruction from mininet command line.

The results are quite different from the expected. There's a destination that can never be added to the flow table, and the switch queries the controller every time it gets a packet addressed to it. The controller does not ever write a flow entry for this destination

As a result, the controller just tells the switch to flood, every time this kind of packets are received. Only two flows are written to the switch flow table.

Flows:



8307	155.725348000	127.0.0.1	127.0.0.1	OF 1.3	146 of flow_add
8465	161.631610000	00:00:00_00:00:01	Broadcast	OF 1.3	150 of packet_in
8466	161.638356000	00:00:00_00:00:01	Broadcast	OF 1.3	148 of packet_out
8468	161.640053000	00:00:00_00:00:02	00:00:00_00:00:01	OF 1.3	150 of packet_in
8469	161.643887000	127.0.0.1	127.0.0.1	OF 1.3	154 of flow_add
8470	161.646636000	00:00:00_00:00:02	00:00:00_00:00:01	OF 1.3	148 of packet_out
8472	161.648172000	10.0.0.1	10.0.0.2	OF 1.3	206 of packet_in
8473	161.654122000	127.0.0.1	127.0.0.1	OF 1.3	154 of flow_add
8474	161.655543000	10.0.0.1	10.0.0.2	OF 1.3	204 of packet_out
8476	161.686530000	00:00:00_00:00:01	Broadcast	OF 1.3	150 of packet_in
8477	161.689812000	00:00:00_00:00:01	Broadcast	OF 1.3	148 of packet_out
8478	161.691916000	10.0.0.1	10.0.0.3	OF 1.3	206 of packet_in
8479	161.695552000	10.0.0.1	10.0.0.3	OF 1.3	204 of packet_out
8481	161.776138000	00:00:00_00:00:02	Broadcast	OF 1.3	150 of packet_in
8482	161.779397000	00:00:00_00:00:02	Broadcast	OF 1.3	148 of packet_out
8484	161.783051000	10.0.0.2	10.0.0.3	OF 1.3	206 of packet_in
8485	161.786796000	10.0.0.2	10.0.0.3	OF 1.3	204 of packet_out
8490	161.871268000	10.0.0.1	10.0.0.3	OF 1.3	206 of packet_in
8491	161.872853000	10.0.0.1	10.0.0.3	OF 1.3	204 of packet_out
8493	161.918762000	10.0.0.2	10.0.0.3	OF 1.3	206 of packet_in
8494	161.922568000	10.0.0.2	10.0.0.3	OF 1.3	204 of packet_out
8575	166.707470000	00:00:00_00:00:01	00:00:00_00:00:03	OF 1.3	150 of packet_in
8580	166.718346000	00:00:00_00:00:01	00:00:00_00:00:03	OF 1.3	148 of packet_out
8583	166.802523000	00:00:00_00:00:02	00:00:00_00:00:03	OF 1.3	150 of packet_in
8584	166.807486000	00:00:00_00:00:02	00:00:00_00:00:03	OF 1.3	148 of packet_out

Figure 3.6: Wireshark trace after the match modification

```

cookie=0x0, duration=42.125s, table=0, n_packets=11, n_bytes=742,
priority=0 actions=CONTROLLER:65535
cookie=0x0, duration=36.196s, table=0, n_packets=6, n_bytes=420,
priority=1, dl_dst=00:00:00:00:00:02 actions=output:2
cookie=0x0, duration=36.206s, table=0, n_packets=7, n_bytes=518,
priority=1, dl_dst=00:00:00:00:00:01 actions=output:1

```

The reason why it's impossible to add a flow for the last host is that every packet that comes from such host already matches a flow, so it's never sent to the controller. As the controller needs to relate an origin MAC with an input port, the flow will never be written to the switch. This is a bad solution.

Another observation is that the original implementation matches the ingress port and the destination MAC, which means that the switch will have the same issues observed in this section when more than one host is reached through the same port. For instance, in a tree topology.

The solution for this problem would be to match the source and the destination MAC, so the controller can write all the rules to the switches and learn all the hosts.

### Removing packet out instruction

We have observed that after every flow\_mod message a packet\_out message is also sent to the switch. We wonder if we can remove this packet\_out messages in a way that does not change the switch behaviour.

For this purpose we change the last lines of the Packet-In event handler so the packet\_out message is only sent when the controller has resolved to send a flood instruction, instead of a flow\_mod message. In the previous implementation a packet\_out message was sent every Packet-In event.

Note that we have made this modification in the controller that adds the table-miss entry at the beginning.

```

# . . .
if out_port != ofproto.OFPP_FLOOD:
    match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
    #print "Match created for " + str(dst)
    # verify if we have a valid buffer_id, if yes avoid to send both
    # flow_mod & packet_out
    if msg.buffer_id != ofproto.OFP_NO_BUFFER:
        self.add_flow(datapath, 1, match, actions, msg.buffer_id)
        #print "Flow added for " + str(dst)
        return
    else:
        self.add_flow(datapath, 1, match, actions)
else:
    data = None
    if msg.buffer_id == ofproto.OFP_NO_BUFFER:
        data = msg.data
    out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                              in_port=in_port, actions=actions, data=data)
    datapath.send_msg(out)

```

The wireshark trace is very interesting, every flow has been added, but the performance has been affected. The results of the pingall test are:

```

*** Ping: testing ping reachability
h1 -> X X
h2 -> h1 X
h3 -> h1 h2
*** Results: 50% dropped (3/6 received)

```

And the trace:

262	25.052924000	00:00:00_00:00:01	Broadcast	OF 1.3	150 of_packet_in
263	25.058020000	00:00:00_00:00:01	Broadcast	OF 1.3	148 of_packet_out
265	25.059770000	00:00:00_00:00:02	00:00:00_00:00:01	OF 1.3	150 of_packet_in
266	25.063064000	127.0.0.1	127.0.0.1	OF 1.3	162 of_flow_add
276	26.051169000	00:00:00_00:00:01	Broadcast	OF 1.3	150 of_packet_in
277	26.053343000	00:00:00_00:00:01	Broadcast	OF 1.3	148 of_packet_out
279	26.056060000	10.0.0.1	10.0.0.2	OF 1.3	206 of_packet_in
280	26.059254000	127.0.0.1	127.0.0.1	OF 1.3	162 of_flow_add
341	35.085313000	00:00:00_00:00:01	Broadcast	OF 1.3	150 of_packet_in
342	35.090809000	00:00:00_00:00:01	Broadcast	OF 1.3	148 of_packet_out
344	35.092376000	00:00:00_00:00:03	00:00:00_00:00:01	OF 1.3	150 of_packet_in
345	35.096738000	127.0.0.1	127.0.0.1	OF 1.3	162 of_flow_add
355	36.083263000	00:00:00_00:00:01	Broadcast	OF 1.3	150 of_packet_in
356	36.086543000	00:00:00_00:00:01	Broadcast	OF 1.3	148 of_packet_out
358	36.088583000	10.0.0.1	10.0.0.3	OF 1.3	206 of_packet_in
359	36.091136000	127.0.0.1	127.0.0.1	OF 1.3	162 of_flow_add
421	45.143965000	00:00:00_00:00:02	Broadcast	OF 1.3	150 of_packet_in
422	45.149942000	00:00:00_00:00:02	Broadcast	OF 1.3	148 of_packet_out
424	45.151967000	00:00:00_00:00:03	00:00:00_00:00:02	OF 1.3	150 of_packet_in
425	45.158597000	127.0.0.1	127.0.0.1	OF 1.3	162 of_flow_add
437	46.143167000	00:00:00_00:00:02	Broadcast	OF 1.3	150 of_packet_in
438	46.145808000	00:00:00_00:00:02	Broadcast	OF 1.3	148 of_packet_out
440	46.148238000	10.0.0.2	10.0.0.3	OF 1.3	206 of_packet_in
441	46.150953000	127.0.0.1	127.0.0.1	OF 1.3	162 of_flow_add

Figure 3.7: Result of removing packet\_out after flow\_mod

No packet\_out message is sent after each flow\_mod message. Six flows have been added, but the reachability test says

that 50% of the packets were dropped.

This means that the packets that didn't match any flow when they were sent have been sent to the controller, then the controller has written the flow, but the switch has not forwarded the packet. So every packet that didn't match a flow and was not a broadcast message has been dropped. It seems that in OpenFlow 1.3 the `packet_out` message is mandatory after the `flow_mod` message for a proper behaviour.

The switch does not forward the packet because the action associated to the table-miss flow entry is to forward the packet to the controller, so it's not buffered at the Switch. This means that a `buffer_id` is not generated for the packet. If a such packet is not returned to the switch with a `Packet Out` message containing the actions to be applied, the switch would not be able to forward it properly.

### 3.6.6 Source code

```
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
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet

class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        # install table-miss flow entry
        #
        # We specify NO BUFFER to max_len of the output action due to
        # OVS bug. At this moment, if we specify a lesser number, e.g.,
        # 128, OVS will send Packet-In with invalid buffer_id and
        # truncated packet data. In that case, we cannot output packets
        # correctly. The bug has been fixed in OVS v2.1.0.
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                         ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)

    def add_flow(self, datapath, priority, match, actions, buffer_id=None):
        ofproto = datapath.ofproto
```

```

parser = datapath.ofproto_parser

inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                     actions)]

if buffer_id:
    mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
                            priority=priority, match=match,
                            instructions=inst)
else:
    mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                            match=match, instructions=inst)

datapath.send_msg(mod)

@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    # If you hit this you might want to increase
    # the "miss_send_length" of your switch
    if ev.msg.msg_len < ev.msg.total_len:
        self.logger.debug("packet truncated: only %s of %s bytes",
                          ev.msg.msg_len, ev.msg.total_len)

    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']

    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]

    dst = eth.dst
    src = eth.src

    dpid = datapath.id
    self.mac_to_port.setdefault(dpid, {})

    self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)

    # learn a mac address to avoid FLOOD next time.
    self.mac_to_port[dpid][src] = in_port

    if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
    else:
        out_port = ofproto.OFPP_FLOOD

    actions = [parser.OFPActionOutput(out_port)]

    # install a flow to avoid packet_in next time
    if out_port != ofproto.OFPP_FLOOD:
        match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
        # verify if we have a valid buffer_id, if yes avoid to send both

```

```
# flow_mod & packet_out
if msg.buffer_id != ofproto.OFP_NO_BUFFER:
    self.add_flow(datapath, 1, match, actions, msg.buffer_id)
    return
else:
    self.add_flow(datapath, 1, match, actions)
data = None
if msg.buffer_id == ofproto.OFP_NO_BUFFER:
    data = msg.data

out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                          in_port=in_port, actions=actions, data=data)
datapath.send_msg(out)
```



## Chapter 4

# MPLS Software-Defined Network

### 4.1 Introduction

This chapter discusses the implementation of a MPLS network using the tools exposed in the previous chapters: Ryu, Open vSwitch, OpenFlow 1.3 and Mininet. The goal of this implementation is to allow the development of a complete SDN solution for MPLS networks. For this purpose, an API Rest must be developed, in order to let the network administrator configure the network at his will.

To know how a MPLS network works, please see section 2.5

### 4.2 First Approach: Simple MPLS network

In this section we will build a Ryu Application that configures a network to work using MPLS labels. There are some constraints:

- Fast forwarding cannot be used as kernel mode does not support MPLS. Current OVS version does not support the Linux Kernel that allows fast forwarding, which is version 3.19
- Only one label can be used, as the current version of OVS (2.3) does not support popping more than one label

There are also several problems to solve in order to properly implement this network.

- ARP, IPV4 and MPLS traffic need separate rules.
- How the controller discovers and adapts to the network.
- Controller criteria for label allocation.
- How the controller distinguishes LER from LSR.

It is clear now that building this MPLS Software-Defined Network goes several steps further than the simple implementation shown in section 2.5, which only used the OVS configuration tools to achieve the desired behavior. We will approach the final solution bit by bit.

In this first approach, we will reproduce the same behavior as in section 2.5, but the flows will be written by the controller.

The features of this first approach are:

- The controller handles the packets in different ways, depending on their ethertypes.
- ARP traffic is forwarded by the network.
- The controller uses ARP traffic to learn where the hosts are, just as the Learning Switch analyzed in section 3.6
- In order to handle the MPLS traffic correctly, the controller identifies if the datapath is LSR or LER by its datapath id, which has been previously configured using a shell script.
- The controller assumes that the IPV4 packets always come from the hosts, therefore, is always handled by the LERs.

Let's see these features in more detail.

## 4.2.1 Mapping the datapaths

The controller must know which routers are LER and LSR as the rules written in each group are completely different.

- LER rules match labels, ports and IP addresses and push or pop the label.
- LSR rules match labels and ports and switch the labels.

As for this first approach we will use the same topology shown in figure 2.4, we will configure the datapath IDs as follows:

```
#Configure switches with a unique ID
echo "Setting switches ID"
sudo ovs-vsctl set bridge s1 other-config:datapath-id=000000000000000001
sudo ovs-vsctl set bridge s2 other-config:datapath-id=000000000000000002
sudo ovs-vsctl set bridge s3 other-config:datapath-id=000000000000000003
```

This way, the controller knows that the IDs 1 and 3 correspond to the LERs and the ID 2 to the LSR. The complete script can be consulted in section 4.5.1

## 4.2.2 Traffic differentiation

Every time a Packet-In event happens, the handler discriminates packets by its ethertype. This means that every time the controller receives a packet, it checks the ethertype and launches a different handler for each type.

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    ...

    # Ethertype 2054=ARP, 2048=IPV4, 34887=MPLS unicast
    ethertype = eth.ethertype
    dpid = datapath.id
    # Set default value for not found keys:
    self.mac_to_port.setdefault(dpid, {})
    self.dst_to_label.setdefault(dpid, {})
```



```

self.logger.info("packet in datapath %s src: %s dst: %s port: %s
    Ethertype=%s", dpid, src, dst, in_port, ethtype)

# If ARP
if ethtype == 2054:
    self.arpHandler(msg)
# If IPV4
elif ethtype == 2048:
    self.ipv4Handler(msg)
#If MPLS unicast
elif ethtype == 34887:
    self.mplsHandler(msg)

```

## ARP Traffic

ARP traffic is handled exactly as in the case of the Switching Hub (Section 3.6). The only difference is that now the matches include also the source mac address and the ethertype. It is really important to include the source address, as if it is not included, several different packets could match a flow and the controller would not be able to map some addresses. This problem is due to the topology used in this approach, where packets with different source addresses may come through the same port.

```

...

match = parser.OFPMatch(in_port=in_port, eth_src=src, eth_dst=dst, eth_type=ethertype)

...

```

The complete source code can be consulted at section 4.5.1

## IPV4 Traffic

MPLS labels are pushed to IPV4 packets and forwarded following the mapping learnt thanks to the ARP traffic forwarding. The label value is increased each type to warantee that labels are not repeated.

```

def ipv4Handler(self, msg):
    ...

    # If the packet is IPV4, it means that the datapath is a LER
    # IPV4 packets that come trough in_port with this destination
    match = parser.OFPMatch(in_port=in_port, eth_dst=dst, eth_type=ethertype)
    # We relate a label to the destination: We select an unused label
    self.label = self.label + 1
    self.dst_to_label[dpid][dst] = self.label

    # Set the out_port using the relation learnt with the ARP packet
    out_port = self.mac_to_port[dpid][dst]
    # Set the action to be performed by the datapath
    actions = [parser.OFPACTIONPushMpls(ethertype=34887,type_=None, len_=None),
               parser.OFPACTIONSetField(mpls_label=self.label),

```

```

        parser.OFPActionOutput (out_port) ]
# Install a flow
...

```

It is particularly important in this case to write the actions to be performed in the correct order. If the action to forward the packet is written before the push label action, the packet will be forwarded without a label. This is because OpenFlow Switch specification indicates that an Action List must be executed in the order it is written (see 1.4.5).

## MPLS traffic

MPLS unicast packets can arrive to the LERs or to the LSR. The controller must know which kind of datapath is handling the packet, in order to write the correct rule. In this approach, the controller checks the datapath ID to see if the datapath is the LSR.

```

def mplsHandler (self, msg) :
    ...

    # The switch can be a LSR or a LER, but the match is the same
    match = parser.OFPMatch (in_port=in_port, eth_dst=dst, eth_type=eth_type,
                             mpls_label=mpls_proto.label)
    # Set the out_port using the relation learnt with the ARP packet
    out_port = self.mac_to_port [dpid] [dst]
    # we must check the switch ID in order to decide the proper action
    if dpid == 2:
        # The switch is a LSR
        # New label
        self.label = self.label + 1
        # Switch labels
        actions = [parser.OFPActionPopMpls (),
                  parser.OFPActionPushMpls (),
                  parser.OFPActionSetField (mpls_label=self.label),
                  parser.OFPActionOutput (out_port) ]
    else:
        # The switch is a LER
        # Pop that label!
        actions = [parser.OFPActionPopMpls (),
                  parser.OFPActionOutput (out_port) ]

    # Install a flow
    ...

```

### 4.2.3 Testing the application

With the configuration explained in the previous sections and the code provided in section 4.5.1, only hosts h1 and h3 should be able to communicate with each other. We perform a reachability test to see if the application works as expected.

```

*** Ping: testing ping reachability
h1 -> X h3
h2 -> X X
h3 -> h1 X
*** Results: 66% dropped (2/6 received)

```

As we said, only hosts h1 and h3 can reach each other through the network.

Next, we provide the wireshark captures in different interfaces after sending an ICMP message from h1 to h3.

The captures show how the ICMP ethertype goes from IP to MPLS depending on the link it's traveling, and how the labels are being switched.

1	0.00000000	00:00:00_00:00:01	Broadcast	ARP	42 Who has 10.0.0.3? Tell 10.0.0.1
2	0.08595100	00:00:00_00:00:03	00:00:00_00:00:01	ARP	60 10.0.0.3 is at 00:00:00:00:00:03
3	0.08611600	10.0.0.1	10.0.0.3	ICMP	98 Echo (ping) request id=0x2409, seq=1/256, ttl=64 (reply in 4)
4	0.15177300	10.0.0.3	10.0.0.1	ICMP	98 Echo (ping) reply id=0x2409, seq=1/256, ttl=64 (request in 3)
5	5.13223200	00:00:00_00:00:03	00:00:00_00:00:01	ARP	60 Who has 10.0.0.1? Tell 10.0.0.3
6	5.13252300	00:00:00_00:00:01	00:00:00_00:00:03	ARP	42 10.0.0.1 is at 00:00:00:00:00:01

▶ Frame 3: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0  
 ▾ Ethernet II, Src: 00:00:00\_00:00:01 (00:00:00:00:00:01), Dst: 00:00:00\_00:00:03 (00:00:00:00:00:03)  
   ▶ Destination: 00:00:00\_00:00:03 (00:00:00:00:00:03)  
   ▶ Source: 00:00:00\_00:00:01 (00:00:00:00:00:01)  
   Type: IP (0x0800)  
 ▶ Internet Protocol Version 4, Src: 10.0.0.1 (10.0.0.1), Dst: 10.0.0.3 (10.0.0.3)  
 ▶ Internet Control Message Protocol

Figure 4.1: Capture from interface s1-eth1

1	0.00000000	00:00:00_00:00:01	Broadcast	ARP	60 Who has 10.0.0.3? Tell 10.0.0.1
2	0.04958400	00:00:00_00:00:03	00:00:00_00:00:01	ARP	60 10.0.0.3 is at 00:00:00:00:00:03
3	0.09550700	10.0.0.1	10.0.0.3	ICMP	102 Echo (ping) request id=0x1c15, seq=1/256, ttl=64 (reply in 4)
4	0.17281500	10.0.0.3	10.0.0.1	ICMP	102 Echo (ping) reply id=0x1c15, seq=1/256, ttl=64 (request in 3)
5	5.13393200	00:00:00_00:00:03	00:00:00_00:00:01	ARP	60 Who has 10.0.0.1? Tell 10.0.0.3
6	5.14405000	00:00:00_00:00:01	00:00:00_00:00:03	ARP	60 10.0.0.1 is at 00:00:00:00:00:01

▶ Frame 3: 102 bytes on wire (816 bits), 102 bytes captured (816 bits) on interface 0  
 ▾ Ethernet II, Src: 00:00:00\_00:00:01 (00:00:00:00:00:01), Dst: 00:00:00\_00:00:03 (00:00:00:00:00:03)  
   ▶ Destination: 00:00:00\_00:00:03 (00:00:00:00:00:03)  
   ▶ Source: 00:00:00\_00:00:01 (00:00:00:00:00:01)  
   Type: MPLS label switched packet (0x8847)  
 ▶ MultiProtocol Label Switching Header, Label: 21, Exp: 0, S: 1, TTL: 64  
 ▶ Internet Protocol Version 4, Src: 10.0.0.1 (10.0.0.1), Dst: 10.0.0.3 (10.0.0.3)  
 ▶ Internet Control Message Protocol

Figure 4.2: Capture from interface s1-eth2

1	0.00000000	00:00:00_00:00:01	Broadcast	ARP	60 Who has 10.0.0.3? Tell 10.0.0.1
2	0.02588600	00:00:00_00:00:03	00:00:00_00:00:01	ARP	60 10.0.0.3 is at 00:00:00:00:00:03
3	0.06916000	10.0.0.1	10.0.0.3	ICMP	102 Echo (ping) request id=0x1ee5, seq=1/256, ttl=64 (reply in 4)
4	0.08990900	10.0.0.3	10.0.0.1	ICMP	102 Echo (ping) reply id=0x1ee5, seq=1/256, ttl=64 (request in 3)
5	5.09575500	00:00:00_00:00:03	00:00:00_00:00:01	ARP	60 Who has 10.0.0.1? Tell 10.0.0.3
6	5.12662100	00:00:00_00:00:01	00:00:00_00:00:03	ARP	60 10.0.0.1 is at 00:00:00:00:00:01

▶ Frame 3: 102 bytes on wire (816 bits), 102 bytes captured (816 bits) on interface 0  
 ▾ Ethernet II, Src: 00:00:00\_00:00:01 (00:00:00:00:00:01), Dst: 00:00:00\_00:00:03 (00:00:00:00:00:03)  
   ▶ Destination: 00:00:00\_00:00:03 (00:00:00:00:00:03)  
   ▶ Source: 00:00:00\_00:00:01 (00:00:00:00:00:01)  
   Type: MPLS label switched packet (0x8847)  
 ▶ MultiProtocol Label Switching Header, Label: 22, Exp: 0, S: 1, TTL: 64  
 ▶ Internet Protocol Version 4, Src: 10.0.0.1 (10.0.0.1), Dst: 10.0.0.3 (10.0.0.3)  
 ▶ Internet Control Message Protocol

Figure 4.3: Capture from interface s2-eth3

2	12.271088000	00:00:00_00:00:01	Broadcast	ARP	60	Who has 10.0.0.3? Tell 10.0.0.1
3	12.271222000	00:00:00_00:00:03	00:00:00_00:00:01	ARP	42	10.0.0.3 is at 00:00:00:00:00:03
4	12.338905000	10.0.0.1	10.0.0.3	ICMP	98	Echo (ping) request id=0x2170, seq=1/256, ttl=64 (reply in 5)
5	12.339074000	10.0.0.3	10.0.0.1	ICMP	98	Echo (ping) reply id=0x2170, seq=1/256, ttl=64 (request in 4)
6	17.352075000	00:00:00_00:00:03	00:00:00_00:00:01	ARP	42	Who has 10.0.0.1? Tell 10.0.0.3
7	17.416896000	00:00:00_00:00:01	00:00:00_00:00:03	ARP	60	10.0.0.1 is at 00:00:00:00:00:01

```

> Frame 4: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
< Ethernet II, Src: 00:00:00_00:00:01 (00:00:00:00:00:01), Dst: 00:00:00_00:00:03 (00:00:00:00:00:03)
  > Destination: 00:00:00_00:00:03 (00:00:00:00:00:03)
  > Source: 00:00:00_00:00:01 (00:00:00:00:00:01)
    Type: IP (0x0800)
  > Internet Protocol Version 4, Src: 10.0.0.1 (10.0.0.1), Dst: 10.0.0.3 (10.0.0.3)
  > Internet Control Message Protocol

```

Figure 4.4: Capture from interface s3-eth1

## 4.2.4 Conclusions

This first approach is still far from being a real MPLS network, but demonstrates how to integrate MPLS, Ryu and Open vSwitch. We can obtain a very similar behaviour to the experiment in section 2.5 using an SDN controller instead of manually writing the rules on the switches.

However, in order to obtain a more realistic scenario, a second approach to this problem has been proposed.

## 4.3 Second approach: Building the MPLS network from an IP network

This second approach aims to create an MPLS Ryu application using an IP router application as a base.

### 4.3.1 Topology

The topology used in this experiment is a custom topology, built using a python script and the API of Mininet. Figure 4.5 shows a representation of this topology. The python script used can be found in section 4.5.2 and is used for both IP and MPLS application.

In the figure you can observe that MAC and IP addresses have been selected to be easily related to each interface network and host. Observe that the backbone network, colored in gray, does not have any IP configured. This is made on purpose, so the Ryu Application run in the control plane can configure this layer as the Network Administrator wishes. We will use the SDN Northbound Interface (see section 1.1.2), which is implemented in Ryu using REST (see section 3.4.2) to communicate the Ryu Application with the Network Administrator.

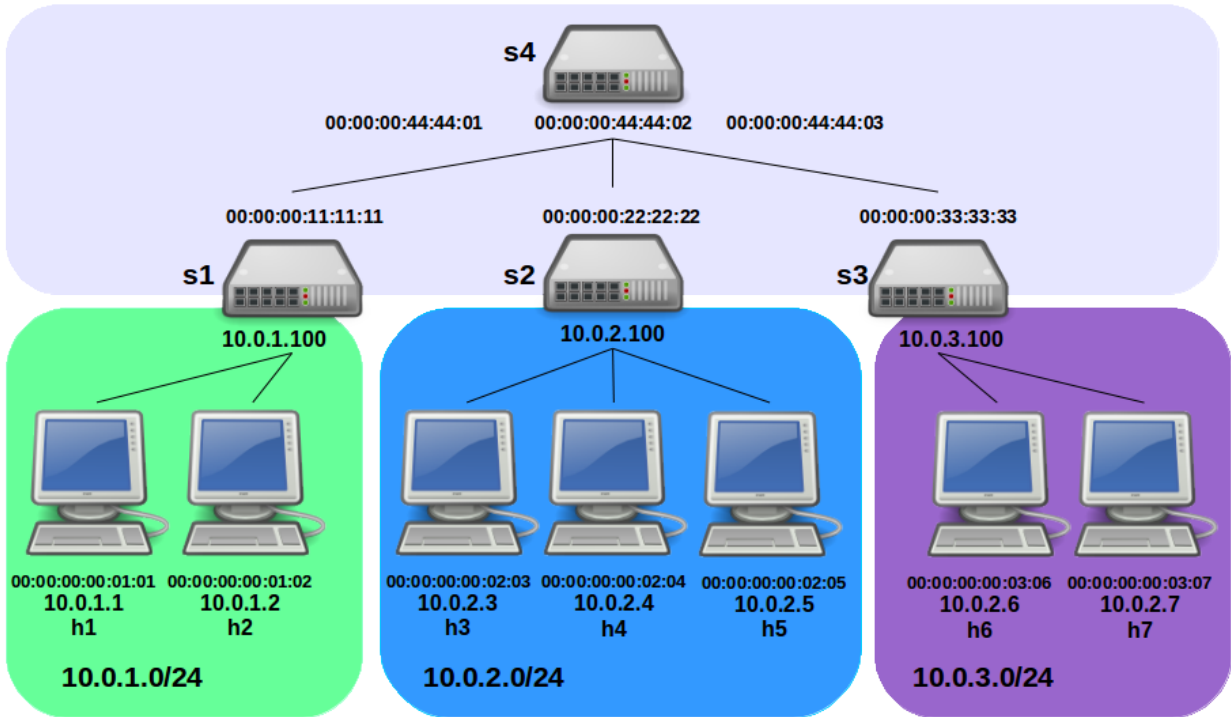


Figure 4.5: Topology used for the second approach

Three IP networks are defined, with seven total hosts. The python script used to generate this topology also configures each host default gateway as the router present in each of this networks. The script also indicates to mininet that OVS switches should be used, and specifies the protocol version to OpenFlow 1.3.

In the following sections we will try to establish the communication between remote hosts using IP routing in the backbone network. After we have understood the basic behaviour of the Ryu Application that implements this, we will modify it in order to make the backbone network forward the packets using MPLS.

### 4.3.2 REST router

The REST router is a Ryu Application that is included in Ryu. This application allows IP routing between different IP networks. This application is also way more complex than the previous applications discussed in this document, so the full code will not be completely analyzed, as it's 1900 lines long and not every piece of code is fundamental to understand and analyze its behaviour.

The complete source code is available online on the Git repository of Ryu:

[https://github.com/osrg/ryu/blob/master/ryu/app/rest\\_router.py](https://github.com/osrg/ryu/blob/master/ryu/app/rest_router.py)

## Classes

As the REST Router application complexity is higher, it takes advantage of Object Oriented Programming to encapsulate the different stakeholders of an openflow controller. The main classes are the following

- **RestRouterAPI:** It's the main class, the one that inherits from *RyuApp*. This class contains four event handlers for the following events: *EventOFPPacketIn*, *EventOFPPFlowStatsReply*, *EventOFPPStatsReply* and *EventDP* which is a non-OpenFlow event contained in the *ryu.controller.dpset* module. This last event refers to the discovery of a datapath by the controller. See more about the OpenFlow messages that can generate events in section 3.3
- **RouterController:** This class inherits from *ryu.controller.ControllerBase* class, and aims to extend its functionality, focused on the scenario of an IP router. Implements the methods to add, delete and access to the routers listed by the Ryu OpenFlow Controller. Also implements the methods for REST commands handling.
- **Router:** Dictionary class. Contains at least one VlanRouter object. An instance of this class corresponds directly to one of the OpenFlow Switches of the network. This class implements several methods for writing, reading or deleting data on the VlanRouters, such as routes or Vlan tags.
- **VlanRouter:** This class implements the intelligence of an Vlan Router for a particular Vlan tag (*vlan\_id*). If it's the case, as it is ours, that no Vlan tags are being used, the *vlan\_id* is zero. This is the most important class from control perspective. It contains information about the router's IP addresses, static and default routes, and implements all the methods for handling that data. Also, implements a *packetin* method that discriminates packets by type (ARP, ICMP or TCP/UDP), and all the methods in order to handle those kind of packets.
- **OfCtl:** This class implements methods to write and delete flows with different OpenFlow versions. Also implements methods to send ARP and ICMP messages.
- **Data structure classes:** Classes that encapsulate IP addresses, routing tables, Ports, etc. These classes are *Route*, *RoutingTable*, *Address*, *AddressData*, *SuspendPacket*, *SuspendPacketList*, *Port* and *PortData*

The module also contains several useful functions such as string to number transformations or mask application to IP addresses.

## REST API

Networks administrators can get, set and delete data from each router using the REST API that is defined in the controller. Administrators can use HTTP commands GET, POST and DELETE to do this.

In order to get the data (addresses and routes) from a particular router:

- No Vlan: GET `/router/{switch_id}`
- Specific Vlan group: GET `/router/{switch_id}/{vlan_id}`

In order to set address or routing data:

- No Vlan: POST `/router/{switch_id}`
- Specific Vlan group: POST `/router/{switch_id}/{vlan_id}`

The POST command parameter is used to introduce the data we want to write in the targeted router:

- Case 1. Set address data: `parameter = {"address": "A.B.C.D/M"}`
- Case 2. Set static route: `parameter = {"destination": "A.B.C.D/M", "gateway": "E.F.G.H"}`
- Case 3. Set default route: `parameter = {"gateway": "E.F.G.H"}`

In order to delete data:

- No Vlan: `DELETE /router/{switch_id}`
- Specific Vlan Group: `DELETE /router/{switch_id}/{vlan_id}`

The DELETE command parameter is used to indicate the data we want to delete in the targeted router:

- Case 1. Delete address data: `parameter = {"address_id": "[int]"} or {"address_id": "all"}`
- Case 2. Delete routing data: `parameter = {"route_id": "[int]"} or {"route_id": "all"}`

More information about how Ryu implements REST API can be found in section 3.4.2

### 4.3.3 IP application walkthrough

In order to better illustrate this application behaviour and how it has been implemented, we will run and test it, showing the results and explaining the blocks of code involved. We first create the topology with mininet. A script has been created for this purpose, the commented code is in Section 4.5.2. We start this topology by running this script:

```
sudo python topology.py
```

Next, we start the ryu application with:

```
ryu-manager ryu.app.rest_router
```

#### Startup

When we start the application, it displays messages through the command interface, explaining that Routers with IDs 0000000000000001, 0000000000000002, 0000000000000003 and 0000000000000004 have joined. Also displays messages that informs that some flows have been installed. Let's take a look at those flows:

```
mininet@mininet-vm:~$ sudo ovs-ofctl -O OpenFlow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=211.990s, table=0, n_packets=0, n_bytes=0, priority=1,
 arp actions=CONTROLLER:65535
 cookie=0x0, duration=211.986s, table=0, n_packets=0, n_bytes=0, priority=1,
 ip actions=drop
```

```
cookie=0x0, duration=211.987s, table=0, n_packets=0, n_bytes=0, priority=0
actions=NORMAL
```

Three flows have been written:

- A flow with priority 1 (low) that matches all ARP packets, where the associated action is to send the packet to the controller.
- A flow with priority 1 (low) that matches all IPv4 packets, where the associated action is to drop the packet.
- A flow with priority 0 (lowest) that matches every packet, where the associated action is to perform normal L2 switching through the non-OpenFlow pipeline. This is the table-miss flow.

The ARP handling flow and table-miss flow are written when a *Router* object is created. It sends the *FlowMod* messages when its constructor method is executed:

```
...

# Set flow: ARP handling (packet in)
priority = get_priority(PRIORITY_ARP_HANDLING)
ofctl.set_packetin_flow(cookie, priority, dl_type=ether.ETH_TYPE_ARP)
self.logger.info('Set ARP handling (packet in) flow [cookie=0x%x]',
                 cookie, extra=self.sw_id)

# Set flow: L2 switching (normal)
priority = get_priority(PRIORITY_NORMAL)
ofctl.set_normal_flow(cookie, priority)
self.logger.info('Set L2 switching (normal) flow [cookie=0x%x]',
                 cookie, extra=self.sw_id)

...
```

The methods *set\_packetin\_flow* and *set\_normal\_flow* contained in the *OfCtl* class use the method *set\_flow* to generate and send the *FlowMod* message that will be sent to the controller. They represent a particular match associated to a particular action:

```
def set_normal_flow(self, cookie, priority):
    out_port = self.dp.ofproto.OFPP_NORMAL
    actions = [self.dp.ofproto_parser.OFPActionOutput(out_port, 0)]
    self.set_flow(cookie, priority, actions=actions)

def set_packetin_flow(self, cookie, priority, dl_type=0, dl_dst=0,
                     dl_vlan=0, dst_ip=0, dst_mask=32, nw_proto=0):
    miss_send_len = UINT16_MAX
    actions = [self.dp.ofproto_parser.OFPActionOutput(
        self.dp.ofproto.OFPP_CONTROLLER, miss_send_len)]
    self.set_flow(cookie, priority, dl_type=dl_type, dl_dst=dl_dst,
                 dl_vlan=dl_vlan, nw_dst=dst_ip, dst_mask=dst_mask,
                 nw_proto=nw_proto, actions=actions)
```

The *dl\_type* field corresponds to the *ethertype* field of the packet, just as *dl\_src* and *dl\_dst* correspond to the source and destination MAC addresses. This nomenclature is the one that was used in previous versions of OpenFlow, which are also supported by this application.



The `set_flow` method is a method designed to allow other methods implement particular flows without additional code. Actions, cookie and priority are passed as a parameter and the match is constructed inside the method with the other values provided, with a limited set of match fields.

At the end, an `OFPFLOWMod` message is generated and sent.

```
def set_flow(self, cookie, priority, dl_type=0, dl_dst=0, dl_vlan=0,
             nw_src=0, src_mask=32, nw_dst=0, dst_mask=32,
             nw_proto=0, idle_timeout=0, actions=None):
    ofp = self.dp.ofproto
    ofp_parser = self.dp.ofproto_parser
    cmd = ofp.OFPFC_ADD

    # Match
    match = ofp_parser.OFPMatch()
    if dl_type:
        match.set_dl_type(dl_type)
    if dl_dst:
        match.set_dl_dst(dl_dst)
    if dl_vlan:
        match.set_vlan_vid(dl_vlan)
    if nw_src:
        match.set_ipv4_src_masked(ipv4_text_to_int(nw_src),
                                  mask_ntob(src_mask))
    if nw_dst:
        match.set_ipv4_dst_masked(ipv4_text_to_int(nw_dst),
                                  mask_ntob(dst_mask))
    if nw_proto:
        if dl_type == ether.ETH_TYPE_IP:
            match.set_ip_proto(nw_proto)
        elif dl_type == ether.ETH_TYPE_ARP:
            match.set_arp_opcode(nw_proto)

    # Instructions
    actions = actions or []
    inst = [ofp_parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS,
                                             actions)]

    m = ofp_parser.OFPFlowMod(self.dp, cookie, 0, 0, cmd, idle_timeout,
                              0, priority, UINT32_MAX, ofp.OFPP_ANY,
                              ofp.OFPG_ANY, 0, match, inst)

    self.dp.send_msg(m)
```

When the `Router` constructor has created this flows, then creates a `VlanRouter` object with the parameter `vlan_id` set to 0. The constructor of this last object sets the flow with the lowest priority, called "Default Route Drop flow"

## Setting up the routers

As seen in section 4.3.2 some information is not provided neither by the controller nor the switches, but the Network Administrator. The network administrator would probably use an application with a graphical interface, that comu-

nicates with the controller through the NBI, but as this is out of the scope of this project, we will simulate it with the messages that such application would use.

For this purpose, we use the program `curl`, which allows us to generate custom HTTP petitions. With this commands we will interact with the REST API of the Ryu application

This part is very straightforward, we write a configuration script simulating the commands an Administration Application would send. See section 4.3.2 to see the commands available.

The script performs the following commands:

```
#!/bin/bash
# Configuration script for the topology under the name topology.py

# Setting router IP addresses using the REST API

##### Addresses for router s1 #####
# Network 1
curl -X POST -d '{"address": "10.0.1.100/24"}'
http://localhost:8080/router/000000000000000001
# Backbone
curl -X POST -d '{"address": "10.10.10.1/24"}'
http://localhost:8080/router/000000000000000001

##### Addresses for router s2 #####
# Network 2
curl -X POST -d '{"address": "10.0.2.100/24"}'
http://localhost:8080/router/000000000000000002
# Backbone
curl -X POST -d '{"address": "10.10.10.2/24"}'
http://localhost:8080/router/000000000000000002

##### Addresses for router s3 #####
# Network 3
curl -X POST -d '{"address": "10.0.3.100/24"}'
http://localhost:8080/router/000000000000000003
# Backbone
curl -X POST -d '{"address": "10.10.10.3/24"}'
http://localhost:8080/router/000000000000000003

##### Addresses for router s4 #####
# Backbone
curl -X POST -d '{"address": "10.10.10.4/24"}'
http://localhost:8080/router/000000000000000004

# Default route for s1: s4
curl -X POST -d '{"gateway": "10.10.10.4"}'
http://localhost:8080/router/000000000000000001

# Default route for s2: s4
curl -X POST -d '{"gateway": "10.10.10.4"}'
http://localhost:8080/router/000000000000000002
```

```

# Default route for s3: s4
curl -X POST -d '{"gateway": "10.10.10.4"}'
http://localhost:8080/router/000000000000000003

# Static routes for s4
curl -X POST -d '{"destination": "10.0.1.0/24", "gateway": "10.10.10.1"}'
http://localhost:8080/router/000000000000000004
curl -X POST -d '{"destination": "10.0.2.0/24", "gateway": "10.10.10.2"}'
http://localhost:8080/router/000000000000000004
curl -X POST -d '{"destination": "10.0.3.0/24", "gateway": "10.10.10.3"}'
http://localhost:8080/router/000000000000000004

```

With this configuration script we achieve the topology displayed in Figure 4.6, where the default route for each Edge Router is *s4* and *s4* has the route of each prefix in its routing table.

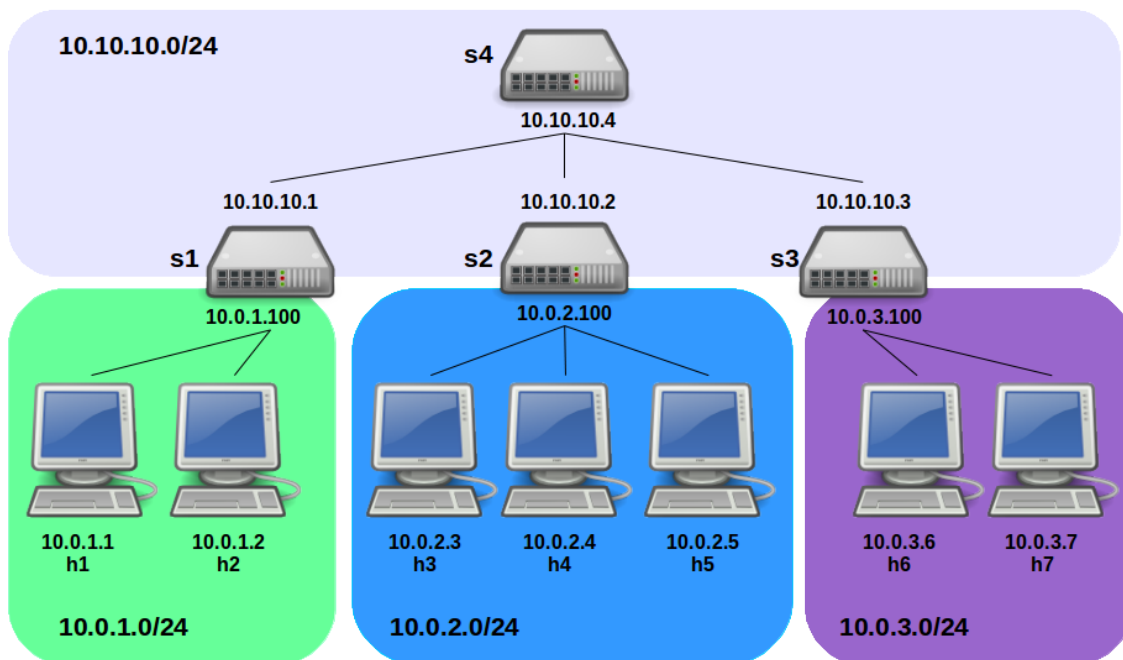


Figure 4.6: Topology after the configuration script

When receiving this REST commands, the *RouterController* passes them to the *VlanRouter* object, through the appropriate *Router*. The *VlanRouter* extracts the data, stores it, and sends the corresponding *FlowMod* messages, depending on which data is received.

We can see that several flows have been added to the Edge Routers. As an example, here are *s1*'s new flows:

```

cookie=0x1, duration=303.304s, table=0, n_packets=0, n_bytes=0,
priority=1037, ip,nw_dst=10.0.1.100 actions=CONTROLLER:65535

```

```

cookie=0x2, duration=303.180s, table=0, n_packets=0, n_bytes=0,
priority=1037, ip, nw_dst=10.10.10.1 actions=CONTROLLER:65535

cookie=0x2, duration=302.846s, table=0, n_packets=0, n_bytes=0,
idle_timeout=1800, priority=35, ip, nw_dst=10.10.10.4
actions=dec_ttl, set_field:00:00:00:11:11:11->eth_src,
set_field:00:00:00:44:44:01->eth_dst, output:3

cookie=0x1, duration=303.302s, table=0, n_packets=0, n_bytes=0,
priority=36, ip, nw_src=10.0.1.0/24, nw_dst=10.0.1.0/24 actions=NORMAL

cookie=0x2, duration=303.179s, table=0, n_packets=0, n_bytes=0,
priority=36, ip, nw_src=10.10.10.0/24, nw_dst=10.10.10.0/24 actions=NORMAL

cookie=0x1, duration=303.306s, table=0, n_packets=0, n_bytes=0,
priority=2, ip, nw_dst=10.0.1.0/24 actions=CONTROLLER:65535

cookie=0x2, duration=303.185s, table=0, n_packets=0, n_bytes=0,
priority=2, ip, nw_dst=10.10.10.0/24 actions=CONTROLLER:65535

```

The new flows are:

- A flow with priority 1037 that matches IPv4 packets which destination is 10.0.1.100 (one of the router's IP addresses). The associated action is to send the packet to the controller.
- A flow with priority 1037 that matches IPv4 packets which destination is 10.10.10.1 (one of the router's IP addresses). The associated action is to send the packet to the controller.
- A flow with priority 35 that matches IPv4 packets with destination 10.10.10.4 (router *s4*). The associated actions are to decrement packet's TTL, to set new values for source and destination MAC addresses and then send the packet through port 3.
- A flow with priority 36 that matches IPv4 packets with source and destination within the network 10.0.1.0/24. The associated action is to apply normal L2 switching.
- A flow with priority 36 that matches IPv4 packets with source and destination within the network 10.10.10.0/24. The associated action is to apply normal L2 switching.
- A flow with priority 2 that matches IPv4 packets with destination 10.0.1.0/24. The associated action is to send the packet to the controller.
- A flow with priority 2 that matches IPv4 packets with destination 10.10.10.0/24. The associated action is to send the packet to the controller.

Some particular flows have also been added to router *s4*:

```

cookie=0x1, duration=306.272s, table=0, n_packets=0, n_bytes=0,
priority=1037, ip, nw_dst=10.10.10.4 actions=CONTROLLER:65535

cookie=0x1, duration=306.175s, table=0, n_packets=0, n_bytes=0,
idle_timeout=1800, priority=35, ip, nw_dst=10.10.10.1
actions=dec_ttl, set_field:00:00:00:44:44:01->eth_src,
set_field:00:00:00:11:11:11->eth_dst, output:1

```

```

    cookie=0x1, duration=306.111s, table=0, n_packets=0, n_bytes=0,
    idle_timeout=1800, priority=35, ip, nw_dst=10.10.10.2
actions=dec_ttl, set_field:00:00:00:44:44:02->eth_src,
set_field:00:00:00:22:22:22->eth_dst, output:2

    cookie=0x1, duration=306.043s, table=0, n_packets=0, n_bytes=0,
    idle_timeout=1800, priority=35, ip, nw_dst=10.10.10.3
actions=dec_ttl, set_field:00:00:00:44:44:03->eth_src,
set_field:00:00:00:33:33:33->eth_dst, output:3

    cookie=0x1, duration=306.269s, table=0, n_packets=0, n_bytes=0,
priority=36, ip, nw_src=10.10.10.0/24, nw_dst=10.10.10.0/24 actions=NORMAL

    cookie=0x1, duration=306.277s, table=0, n_packets=0, n_bytes=0,
priority=2, ip, nw_dst=10.10.10.0/24 actions=CONTROLLER:65535

    cookie=0x10000, duration=305.946s, table=0, n_packets=0, n_bytes=0,
priority=26, ip, nw_dst=10.0.1.0/24
actions=dec_ttl, set_field:00:00:00:44:44:01->eth_src,
set_field:00:00:00:11:11:11->eth_dst, output:1

    cookie=0x20000, duration=305.874s, table=0, n_packets=0, n_bytes=0,
priority=26, ip, nw_dst=10.0.2.0/24
actions=dec_ttl, set_field:00:00:00:44:44:02->eth_src,
set_field:00:00:00:22:22:22->eth_dst, output:2

    cookie=0x30000, duration=305.816s, table=0, n_packets=0, n_bytes=0,
priority=26, ip, nw_dst=10.0.3.0/24
actions=dec_ttl, set_field:00:00:00:44:44:03->eth_src,
set_field:00:00:00:33:33:33->eth_dst, output:3

```

This flows are very similar to the ones added to router *s1*, but also three special flows, corresponding to the static routes have been added:

- A flow with priority 26 (cookie 0x10000) that matches all IPv4 packets wick destination is within the network 10.0.1.0/24. The associated action is to change the values of the source and destination MAC addresses and send the packet through port 1.
- A flow with priority 26 (cookie 0x20000) that matches all IPv4 packets wick destination is within the network 10.0.2.0/24. The associated action is to change the values of the source and destination MAC addresses and send the packet through port 2.
- A flow with priority 26 (cookie 0x30000) that matches all IPv4 packets wick destination is within the network 10.0.3.0/24. The associated action is to change the values of the source and destination MAC addresses and send the packet through port 3.

Now, let's take a look at the code of *VlanRouter*, to understand how the application has processed the REST commands in order to add these flows. A *VlanRouter* object extracts the data from a REST command using the method *set\_data*, *get\_data* or *delete\_data*, depending on the type of command: POST, GET or DELETE. We will focus on the *set\_data* method as it's the one that the application is invoking when receiving the commands launched with our script.

```

def set_data(self, data):
    details = None

    try:
        # Set address data
        if REST_ADDRESS in data:
            address = data[REST_ADDRESS]
            address_id = self._set_address_data(address)
            details = 'Add address [address_id=%d]' % address_id
        # Set routing data
        elif REST_GATEWAY in data:
            gateway = data[REST_GATEWAY]
            if REST_DESTINATION in data:
                destination = data[REST_DESTINATION]
            else:
                destination = DEFAULT_ROUTE
            route_id = self._set_routing_data(destination, gateway)
            details = 'Add route [route_id=%d]' % route_id

    except CommandFailure as err_msg:
        msg = {REST_RESULT: REST_NG, REST_DETAILS: str(err_msg)}
        return self._response(msg)

    if details is not None:
        msg = {REST_RESULT: REST_OK, REST_DETAILS: details}
        return self._response(msg)
    else:
        raise ValueError('Invalid parameter.')

```

The method looks for certain fields, stored in constants beginning with the prefix *REST\_* that correspond to the fields introduced in the parameter of the POST command. Depending on which field it finds, it launches another method to process the data. This method can be *\_set\_address\_data* or *\_set\_routing\_data*.

The method *\_set\_address\_data* is launched when the POST parameter contains an IP address and mask of the router. The method stores the address/mask information in an *AddressData* object and then sets three flows:

- A flow that matches IPv4 packets which destination and mask is the one that has been registered. This flow is created with the method of the OfCtl class *set\_packetin\_flow* which creates a flow where the action is to send the packet to the controller.
- A flow that matches IPv4 packets which destination is within the registered address network. This flow is also created with the *set\_packetin\_flow* method, which makes the associated action: send the packet to the controller.
- A flow that matches IPv4 packets which destination is within the same network as the registered address. This flow is created with the method *set\_routing\_flow* where the output port is passed as an argument. In this case, the output port is set to NORMAL (see section 1.3)

After setting the flows, this method tells the switch to send an ARP request to the default gateway with the method *ofctl.send\_arp\_request*

On the other hand, the method *\_set\_routing\_data* is launched when the POST parameter contains a route, either default or static. The method stores the destination/gateway relation in an *RoutingTable* object and then proceeds to send an

ARP request to the gateway. No flow is written directly in this method. The flow is written when the ARP reply arrives.

## ARP handling

When an ARP packet arrives, an event *EventOFPPacketIn* (see Section 3.3) is generated. The message associated to the event is passed through the objects to the *packet\_in\_handler* which looks at the packet header and discriminates the packets for its Ethertype:

```
def packet_in_handler(self, msg, header_list):
    # Check invalid TTL (for OpenFlow V1.2/1.3)
    ofproto = self.dp.ofproto
    if ofproto.OFP_VERSION == ofproto_v1_2.OFP_VERSION or \
        ofproto.OFP_VERSION == ofproto_v1_3.OFP_VERSION:
        if msg.reason == ofproto.OFPR_INVALID_TTL:
            self._packetin_invalid_ttl(msg, header_list)
            return

    # Analyze event type.
    if ARP in header_list:
        self._packetin_arp(msg, header_list)
        return

    if IPV4 in header_list:
        rt_ports = self.address_data.get_default_gw()
        if header_list[IPV4].dst in rt_ports:
            # Packet to router's port.
            if ICMP in header_list:
                if header_list[ICMP].type == icmp.ICMP_ECHO_REQUEST:
                    self._packetin_icmp_req(msg, header_list)
                    return
            elif TCP in header_list or UDP in header_list:
                self._packetin_tcp_udp(msg, header_list)
                return
        else:
            # Packet to internal host or gateway router.
            self._packetin_to_node(msg, header_list)
            return
```

The *header\_list* argument is extracted from the message in the *packet\_in\_handler* contained in the *Router* object and passed as a parameter to the *packet\_in\_handler* in *VlanRouter*. As can be seen in the code above, if the message is ARP, the method *\_packetin\_arp* is launched. This is a large method so we'll chunk it down for a better analysis:

```
def _packetin_arp(self, msg, header_list):
    src_addr = self.address_data.get_data(ip=header_list[ARP].src_ip)
    if src_addr is None:
        return

    # case: Receive ARP from the gateway
    # Update routing table.
```

```

# case: Receive ARP from an internal host
# Learning host MAC.
gw_flg = self._update_routing_tbl(msg, header_list)
if gw_flg is False:
    self._learning_host_mac(msg, header_list)

...

```

This is the part of the code that writes the rest of the flows to the switches. Specifically the `_update_routing_tbl` method checks if the source IP address correspond to one of the gateways listed in the routing table. If so, it uses the method `set_routing_flow` to write a flow, after updating the MAC address of the gateway in its routing table.

- Ethertype (Match): IPv4
- Destination IP (Match): Source IP of the ARP packet.
- Destination MAC address to be set (Action): Source MAC address of the ARP packet.
- Source MAC address to be set(Action): MAC address from the ingress port of the ARP packet.
- Decrement time to live (Action)
- Output Port (Action): Ingress port of the ARP packet.

This way, the application uses ARP traffic to write the entries of its routing table to the switches, relating them to MAC addresses and ports.

The method `_learning_host_mac` is launched when the source IP address of the ARP packet does not correspond to a gateway. In this case, the application writes a flow to the switch, so it "learns" the host. The flow has the following fields:

- Ethertype (Match): IPv4
- Destination IP (Match): Source IP of the ARP packet.
- Destination MAC address to be set (Action): Source MAC address of the ARP packet.
- Source MAC address to be set(Action): MAC address from the ingress port of the ARP packet.
- Decrement time to live (Action)
- Output Port (Action): Ingress port of the ARP packet.

This is what the Application calls an *Implicit routing flow*.

```

...

# ARP packet handling.
in_port = self.ofctl.get_packetin_inport(msg)
src_ip = header_list[ARP].src_ip
dst_ip = header_list[ARP].dst_ip
srcip = ip_addr_ntoa(src_ip)
dstip = ip_addr_ntoa(dst_ip)
rt_ports = self.address_data.get_default_gw()

```



```

if src_ip == dst_ip:
    # GARP -> packet forward (normal)
    output = self.ofctl.dp.ofproto.OFPP_NORMAL
    self.ofctl.send_packet_out(in_port, output, msg.data)

    self.logger.info('Receive GARP from [%s].', srcip,
                    extra=self.sw_id)
    self.logger.info('Send GARP (normal).', extra=self.sw_id)

    ...

```

If the source and destination IP of the ARP are the same, the packet is identified as a Gratuitous ARP request (GARP). A gratuitous ARP request is an AddressResolutionProtocol request packet where the source and destination IP are both set to the IP of the machine issuing the packet and the destination MAC is the broadcast address ff:ff:ff:ff:ff:ff. A gratuitous ARP reply is a reply to which no request has been made.

In the case of detecting a GARP, the controller sets the output port to NORMAL (see section 1.3) and send the packet back to the switch with a *Packet Out* message.

```

...

elif dst_ip not in rt_ports:
    dst_addr = self.address_data.get_data(ip=dst_ip)
    if (dst_addr is not None and
        src_addr.address_id == dst_addr.address_id):
        # ARP from internal host -> packet forward (normal)
        output = self.ofctl.dp.ofproto.OFPP_NORMAL
        self.ofctl.send_packet_out(in_port, output, msg.data)

        self.logger.info('Receive ARP from an internal host [%s].',
                        srcip, extra=self.sw_id)
        self.logger.info('Send ARP (normal)', extra=self.sw_id)

    ...

```

The variable *rt\_ports* is the list of default gateways plus the networks (the result of applying the mask to the address) to where the router is connected. If the destination IP address is not in this list, the application checks if the address is contained in the dictionary object *address\_data* which is an instance of *AddressData* and contains a list of known addresses with an associated gateway and mask, encapsulated inside *Address* objects.

If the destination address is contained in this dictionary, and both destination and source addresses belong to the same network, the controller sends a *Packet Out* instruction to the switch with output port set to NORMAL. The switch then will perform L2 switching.

```

...

else:
    if header_list[ARP].opcode == arp.ARP_REQUEST:
        # ARP request to router port -> send ARP reply
        src_mac = header_list[ARP].src_mac
        dst_mac = self.port_data[in_port].mac
        arp_target_mac = dst_mac

```

```

output = in_port
in_port = self.ofctl.dp.ofproto.OFPP_CONTROLLER

self.ofctl.send_arp(arp.ARP_REPLY, self.vlan_id,
                   dst_mac, src_mac, dst_ip, src_ip,
                   arp_target_mac, in_port, output)

log_msg = 'Receive ARP request from [%s] to router port [%s].'
self.logger.info(log_msg, srcip, dstip, extra=self.sw_id)
self.logger.info('Send ARP reply to [%s]', srcip,
                 extra=self.sw_id)

elif header_list[ARP].opcode == arp.ARP_REPLY:
    # ARP reply to router port -> suspend packets forward
    log_msg = 'Receive ARP reply from [%s] to router port [%s].'
    self.logger.info(log_msg, srcip, dstip, extra=self.sw_id)

    packet_list = self.packet_buffer.get_data(src_ip)
    if packet_list:
        # stop ARP reply wait thread.
        for suspend_packet in packet_list:
            self.packet_buffer.delete(pkt=suspend_packet)

        # send suspend packet.
        output = self.ofctl.dp.ofproto.OFPP_TABLE
        for suspend_packet in packet_list:
            self.ofctl.send_packet_out(suspend_packet.in_port,
                                       output,
                                       suspend_packet.data)
            self.logger.info('Send suspend packet to [%s].',
                            srcip, extra=self.sw_id)

```

The last case, checks if the ARP is a request or a reply. If the packet is an ARP request, the controller tells the switch to send an ARP reply.

If the packet is an ARP reply, the application checks its buffer to see if there's a packet which destination IP address matches the source IP address of the ARP reply. If there's a match, deletes the packet from the buffer and proceeds to send the packet to the switch where the output port is *OFPP\_TABLE*. This means that the controller is telling the switch to apply the Flow Table again to the packet contained in the *Packet Out* message

Now that we have seen how these new flows have been written to the switches, let's see how the controller handles ICMP messages between hosts.

### Case 1: ICMP messages between internal hosts

For example, let's test what happens if we send an ICMP request from host h1 to host h2. Both hosts are inside the same network, so the router should act as an L2 switch.

When we execute the following command from the Mininet CLI:

```
mininet> h1 ping -c1 h2
```

We get the wireshark capture of Figure 4.7 and the following messages are displayed in the terminal where the Ryu Application is running:

```
[RT][INFO] switch_id=0000000000000001: Set implicit routing flow [cookie=0x1]
[RT][INFO] switch_id=0000000000000001:
    Receive ARP from an internal host [10.0.1.1].
[RT][INFO] switch_id=0000000000000001: Send ARP (normal)
[RT][INFO] switch_id=0000000000000001: Set implicit routing flow [cookie=0x1]
[RT][INFO] switch_id=0000000000000001:
    Receive ARP from an internal host [10.0.1.2].
[RT][INFO] switch_id=0000000000000001: Send ARP (normal)
[RT][INFO] switch_id=0000000000000001: Set implicit routing flow [cookie=0x1]
[RT][INFO] switch_id=0000000000000001:
    Receive ARP from an internal host [10.0.1.2].
[RT][INFO] switch_id=0000000000000001: Send ARP (normal)
[RT][INFO] switch_id=0000000000000001: Set implicit routing flow [cookie=0x1]
[RT][INFO] switch_id=0000000000000001:
    Receive ARP from an internal host [10.0.1.1].
[RT][INFO] switch_id=0000000000000001: Send ARP (normal)
```

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	00:00:00_00:01:01	Broadcast	ARP	42	Who has 10.0.1.2? Tell 10.0.1.1
2	0.028646000	00:00:00_00:01:02	00:00:00_00:01:01	ARP	60	10.0.1.2 is at 00:00:00:00:01:02
3	0.028802000	10.0.1.1	10.0.1.2	ICMP	98	Echo (ping) request id=0x23d6, seq=1/256, ttl=64 (reply in 4)
4	0.029477000	10.0.1.2	10.0.1.1	ICMP	98	Echo (ping) reply id=0x23d6, seq=1/256, ttl=64 (request in 3)
5	5.040133000	00:00:00_00:01:02	00:00:00_00:01:01	ARP	60	Who has 10.0.1.1? Tell 10.0.1.2
6	5.040312000	00:00:00_00:01:01	00:00:00_00:01:02	ARP	42	10.0.1.1 is at 00:00:00:00:01:01

Figure 4.7: Capture from interface s1-eth1

According to these results, the hosts reach each other and the controller has set four new implicit routing flows in router s1. The new flows found on the Flow Table are:

```
# 1
cookie=0x1, duration=875.220s, table=0, n_packets=0, n_bytes=0,
idle_timeout=1800, priority=35, ip,nw_dst=10.0.1.1
actions=dec_ttl,set_field:5a:5e:70:64:60:0a->eth_src,
set_field:00:00:00:00:01:01->eth_dst,output:1

# 2
cookie=0x1, duration=875.227s, table=0, n_packets=0, n_bytes=0,
idle_timeout=1800, priority=35, ip,nw_dst=10.0.1.2
actions=dec_ttl,set_field:e6:6e:36:53:26:0a->eth_src,
set_field:00:00:00:00:01:02->eth_dst,output:2
```

It seems that only two flows have been added to the Flow Table. The second two flows that appear in the log of the Ryu Application are set when the ARP cache validation messages are transmitted to the controller after the ICMP exchange. This two flows are exactly the same as the ones before, so the switch does not write them again. The controller has writes these flows with the `_learning_host_mac` method commented previously. As can be seen above, matches consist on the destination IP address and actions consist on decrementing the TTL, changing the source and destination MAC addresses and sending the packet through the port learnt thanks to the ARP packet.

Here we can see that as this match only affects to IPv4 packets, matching only the destination does not lead to a problem, as ARP packets are always forwarded to the controller. This would be a good and simple solution to the problem proposed in Section 3.6.5, where we concluded that if only the destination is matched, there is a host that never get its flows written.

As we can see, the controller only has to deal with ARP traffic for an internal ICMP message exchange.

## Case 2: ICMP messages between remote hosts

Now, let's see what happens if we send an ICMP request from host h1 to host h2. In this case, hosts are in different networks, so the packets will go through three different networks before they reach it's destiny.

When introducing the following command in the Mininet CLI: `mininet> h1 ping -c1 h3` We get the wire-shark captures of Figures 4.8, 4.9, 4.10 and 4.11. This captures correspond to the interfaces the ICMP request travels in chronological order. We can observe how the edge routers handle ARP traffic in the capilar networks. There are no ARP messages in the backbone network during this transmission. This is because the ICMP packet matches one of the flows written during the address and route setup through the REST API.

Observe that the MAC addresses are different in each capture, as the rules in each router tell them to change it.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	00:00:00_00:01:01	Broadcast	ARP	42	Who has 10.0.1.100? Tell 10.0.1.1
2	0.009798000	16:50:8b:9e:ab:10	00:00:00_00:01:01	ARP	42	10.0.1.100 is at 16:50:8b:9e:ab:10 (duplicate use of 10.0.1.1 detected!)
3	0.010008000	10.0.1.1	10.0.2.3	ICMP	98	Echo (ping) request id=0x314f, seq=1/256, ttl=64 (reply in 4)
4	0.034186000	10.0.2.3	10.0.1.1	ICMP	98	Echo (ping) reply id=0x314f, seq=1/256, ttl=61 (request in 3)

```

> Frame 3: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
> Ethernet II, Src: 00:00:00_00:01:01 (00:00:00:00:01:01), Dst: 16:50:8b:9e:ab:10 (16:50:8b:9e:ab:10)
> Internet Protocol Version 4, Src: 10.0.1.1 (10.0.1.1), Dst: 10.0.2.3 (10.0.2.3)
> Internet Control Message Protocol

```

Figure 4.8: Capture from interface s1-eth1

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.1.1	10.0.2.3	ICMP	98	Echo (ping) request id=0x3616, seq=1/256, ttl=63 (reply in 2)
2	0.027845000	10.0.2.3	10.0.1.1	ICMP	98	Echo (ping) reply id=0x3616, seq=1/256, ttl=62 (request in 1)

```

> Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
> Ethernet II, Src: 00:00:00_11:11:11 (00:00:00:11:11:11), Dst: 00:00:00_44:44:01 (00:00:00:44:44:01)
> Internet Protocol Version 4, Src: 10.0.1.1 (10.0.1.1), Dst: 10.0.2.3 (10.0.2.3)
> Internet Control Message Protocol

```

Figure 4.9: Capture from interface s4-eth1

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.1.1	10.0.2.3	ICMP	98	Echo (ping) request id=0x3ab8, seq=1/256, ttl=62 (reply in 2)
2	0.023191000	10.0.2.3	10.0.1.1	ICMP	98	Echo (ping) reply id=0x3ab8, seq=1/256, ttl=63 (request in 1)

```

> Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
> Ethernet II, Src: 00:00:00_44:44:02 (00:00:00:44:44:02), Dst: 00:00:00_22:22:22 (00:00:00:22:22:22)
> Internet Protocol Version 4, Src: 10.0.1.1 (10.0.1.1), Dst: 10.0.2.3 (10.0.2.3)
> Internet Control Message Protocol

```

Figure 4.10: Capture from interface s4-eth2

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	ee:0b:dd:29:aa:0a	Broadcast	ARP	42	Who has 10.0.2.3? Tell 10.0.2.100
2	0.000166000	00:00:00_00:02:03	ee:0b:dd:29:aa:0a	ARP	42	10.0.2.3 is at 00:00:00:00:02:03
3	0.015061000	10.0.1.1	10.0.2.3	ICMP	98	Echo (ping) request id=0x3f6a, seq=1/256, ttl=61 (reply in 4)
4	0.015372000	10.0.2.3	10.0.1.1	ICMP	98	Echo (ping) reply id=0x3f6a, seq=1/256, ttl=64 (request in 3)
5	5.021703000	00:00:00_00:02:03	ee:0b:dd:29:aa:0a	ARP	42	Who has 10.0.2.100? Tell 10.0.2.3
6	5.029761000	ee:0b:dd:29:aa:0a	00:00:00_00:02:03	ARP	42	10.0.2.100 is at ee:0b:dd:29:aa:0a (duplicate use of 10.0.2.3 detected!)

> Frame 3: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0  
 > Ethernet II, Src: ee:0b:dd:29:aa:0a (ee:0b:dd:29:aa:0a), Dst: 00:00:00\_00:02:03 (00:00:00:00:02:03)  
 > Internet Protocol Version 4, Src: 10.0.1.1 (10.0.1.1), Dst: 10.0.2.3 (10.0.2.3)  
 > Internet Control Message Protocol

Figure 4.11: Capture from interface s2-eth1

The following messages are displayed in the terminal where the Ryu Application is running:

```
[RT][INFO] switch_id=0000000000000001: Set implicit routing flow [cookie=0x1]
[RT][INFO] switch_id=0000000000000001:
  Receive ARP request from [10.0.1.1] to router port [10.0.1.100].
[RT][INFO] switch_id=0000000000000001: Send ARP reply to [10.0.1.1]
[RT][INFO] switch_id=0000000000000002:
  Receive IP packet from [10.0.1.1] to an internal host [10.0.2.3].
[RT][INFO] switch_id=0000000000000002: Send ARP request (flood)
[RT][INFO] switch_id=0000000000000002: Set implicit routing flow [cookie=0x1]
[RT][INFO] switch_id=0000000000000002:
  Receive ARP reply from [10.0.2.3] to router port [10.0.2.100].
[RT][INFO] switch_id=0000000000000002: Send suspend packet to [10.0.2.3].
[RT][INFO] switch_id=0000000000000002: Set implicit routing flow [cookie=0x1]
[RT][INFO] switch_id=0000000000000002:
  Receive ARP request from [10.0.2.3] to router port [10.0.2.100].
[RT][INFO] switch_id=0000000000000002: Send ARP reply to [10.0.2.3]
```

The first three messages refer to the process explained in the previous section, where the controller learns the host through the ARP handler method. The flow that results from this is the following:

```
cookie=0x1, duration=102.754s, table=0, n_packets=1, n_bytes=98,
idle_timeout=1800, priority=35, ip, nw_dst=10.0.1.1
actions=dec_ttl, set_field:c2:ae:4d:33:79:3a->eth_src,
set_field:00:00:00:00:01:01->eth_dst, output:1}
```

The ICMP request that follows the ARP request, matches one of the low priority flows. The one corresponding to the default route for IP packets:

```
cookie=0x10000, duration=110.731s, table=0, n_packets=1, n_bytes=98,
priority=1, ip actions=dec_ttl, set_field:00:00:00:11:11:11->eth_src,
set_field:00:00:00:44:44:01->eth_dst, output:3
```

Therefore, the MAC addresses are changed and the packet is sent through port 3, which corresponds to the interface s1-eth3 which is connected to the interface s4-eth1.

Once the packet enters router s4, the packet matches one of the static-route flows, therefore the router changes the MAC addresses and forwards the packet to router s2:

```
cookie=0x20000, duration=621.906s, table=0, n_packets=1, n_bytes=98,
priority=26, ip,nw_dst=10.0.2.0/24
actions=dec_ttl,set_field:00:00:00:44:44:02->eth_src,
set_field:00:00:00:22:22:22->eth_dst,output:2
```

The packet then enters router s2, and matches the following flow:

```
cookie=0x1, duration=744.569s, table=0, n_packets=1, n_bytes=98,
priority=2, ip,nw_dst=10.0.2.0/24 actions=CONTROLLER:65535
```

The packet is sent to the controller as it's specified in the flow. The message is passed through the objects in the Ryu Application to the *packet\_in\_handler* method in the *VlanRouter* object associated to the router s2. The Packet-In handler checks that the packet is an IPv4 packet and it's not directed to any of the default gateways. Then, method *\_packetin\_to\_node* is launched to handle this situation.

```
def _packetin_to_node(self, msg, header_list):
    if len(self.packet_buffer) >= MAX_SUSPENDPACKETS:
        self.logger.info('Packet is dropped, MAX_SUSPENDPACKETS exceeded.',
                        extra=self.sw_id)

        return

    # Send ARP request to get node MAC address.
    in_port = self.ofctl.get_packetin_inport(msg)
    src_ip = None
    dst_ip = header_list[IPV4].dst
    srcip = ip_addr_ntoa(header_list[IPV4].src)
    dstip = ip_addr_ntoa(dst_ip)

    address = self.address_data.get_data(ip=dst_ip)
    if address is not None:
        log_msg = 'Receive IP packet from [%s] to an internal host [%s].'
        self.logger.info(log_msg, srcip, dstip, extra=self.sw_id)
        src_ip = address.default_gw
    else:
        route = self.routing_tbl.get_data(dst_ip=dst_ip)
        if route is not None:
            log_msg = 'Receive IP packet from [%s] to [%s].'
            self.logger.info(log_msg, srcip, dstip, extra=self.sw_id)
            gw_address = self.address_data.get_data(ip=route.gateway_ip)
            if gw_address is not None:
                src_ip = gw_address.default_gw
                dst_ip = route.gateway_ip

    if src_ip is not None:
        self.packet_buffer.add(in_port, header_list, msg.data)
        self.send_arp_request(src_ip, dst_ip, in_port=in_port)
        self.logger.info('Send ARP request (flood)', extra=self.sw_id)
```

This method stores the packet in a buffer and sends an ARP request to discover destination host's MAC address. When the reply arrives, the ARP handler will write the flow, look for the packet in the buffer, delete it and send it to the host. This is what is displayed as Send suspend packet to [10.0.2.3] in the Ryu Application's log.

#### 4.3.4 Modifications to create a simple MPLS application

Now that the basics of the IP application have been explained, let's create a MPLS network adding some modifications to the *rest\_router* application.

If we want the backbone network to behave as an MPLS network, we have to take in consideration what this behavior will be.

When the first packet travels from one host to another after the setup the following events should happen:

- The host sends an ARP request to discover the gateway MAC address.
- The gateway forwards the packet to the controller.
- The controller writes a flow to the gateway to learn the host, and tells it to send an ARP reply.
- The host receives the ARP reply and sends an ICMP packet.
- The packet enters the default gateway (LER) of the host.
- The packet matches one of the flows and is sent to the controller.
- The controller chooses a label and stores the relationship between the label, the source IP and the destination IP.
- The controller writes a flow to the LER that matches the source and destination IP address, pushes a MPLS label to the packet and sends it through the port that is connected to the LSR.
- The packet is sent back to the LER, and matches the last written flow.
- The packet enters the LSR, matches a flow and it's sent to the controller.
- The controller examines the label, looks up the destination address that is associated to that label. Chooses a new label, stores the relation between the label and the source and destination addresses. Writes a flow to the LSR matching the ingress port and the label, that pops the MPLS label, pushes a new one, and sends it through the port connected to the LER that belongs to the destination network.
- The packet is sent back to the LSR, and matches the last written flow.
- The packet enters the LER, matches a flow and it's sent to the controller.
- The controller examines the label and looks up the destination address that is associated to that label. Checks if the destination is a known host, if not, buffers the packet and sends an ARP request to learn the host. When (or once) the host is known, the controller writes a flow that matches the label and forwards the packet through the port connected to the destination.

Here is the list of problems that have to be solved:

- Routers have no IP addresses inside the MPLS network, so routing tables are not useful here. The administrator has to be able to pass a relation between the destination network and the output port (corresponding to an interface) to the routers.
- LER and LSR behave differently and the controller should treat them in different ways. The administrator should be able to tell the controller which routers are LER and which routers are LSR
- The controller needs a mapping relating labels to its destination, so it knows what flow to set when a router sends a packet in message with a MPLS packet in it. New classes have to be created to store those values.

- The controller needs a list of the known hosts, to know which port must be used when a packet is leaving the MPLS network.
- New handlers have to be implemented for the different scenarios of a MPLS network: When a packet enters the network, when a packet travels through a LSR and when a packet leaves the network.
- New methods for flow writing have to be implemented, as the ones provided in the OfCtl class don't consider all the fields needed for the flows we will need to write.

In this particular implementation each label will be associated to a virtual circuit between remote hosts.

More than 400 lines of code have been added to the original Ryu Application in order to achieve the desired behavior (Total lines: 2319). **Every modification is precluded by a comment starting with the tag *MPLSmod***. The complete source code can be found in section 4.5.2

### 4.3.5 Writing data through the REST API

The first thing that needs to be done is modifying the REST API of this application so the administrator can properly configure LERs and LSRs

We want to be able to pass the following messages to the controller:

- Network address/Mask and Output port for a given router
- Router type (LER/LSR)

So, for instance, if we want to configure router *s4* as a LSR, and set up the information about the distantion networks and ports, we would send the controller the following messages:

```
curl -X POST -d '{"prefix":"10.0.1.0/24", "port":"1"}'
http://localhost:8080/router/000000000000000004

curl -X POST -d '{"prefix":"10.0.2.0/24", "port":"2"}'
http://localhost:8080/router/000000000000000004

curl -X POST -d '{"prefix":"10.0.3.0/24", "port":"3"}'
http://localhost:8080/router/000000000000000004

curl -X POST -d '{"router":"lsr"}'
http://localhost:8080/router/000000000000000004
```

As we want the controller to be able to store and process this data, we modify the *set\_data* method in the *VlanRouter* class, leaving it as follows:

```
def set_data(self, data):
    details = None

    try:
        # Set address data
        if REST_ADDRESS in data:
            address = data[REST_ADDRESS]
```



```

        address_id = self._set_address_data(address)
        details = 'Add address [address_id=%d]' % address_id
    # Set routing data
    elif REST_GATEWAY in data:
        gateway = data[REST_GATEWAY]
        if REST_DESTINATION in data:
            destination = data[REST_DESTINATION]
        else:
            destination = DEFAULT_ROUTE
        route_id = self._set_routing_data(destination, gateway)
        details = 'Add route [route_id=%d]' % route_id

    # MPLSmod: set prefix-port mapping data
    elif REST_PREFIX in data:
        prefix = data[REST_PREFIX]
        port = data[REST_PORT]
        prefix_id = self._set_prefix_data(prefix, port)
        details = 'Add prefix to port [prefix_id=%d]' % prefix_id

    # MPLSmod: set router type
    elif REST_ROUTER in data:
        router = data[REST_ROUTER]
        self._set_router_type(router)
        details = 'Add router type: %s' % router

    except CommandFailure as err_msg:
        msg = {REST_RESULT: REST_NG, REST_DETAILS: str(err_msg)}
        return self._response(msg)

    if details is not None:
        msg = {REST_RESULT: REST_OK, REST_DETAILS: details}
        return self._response(msg)
    else:
        raise ValueError('Invalid parameter.')
```

The constants REST\_PORT, REST\_PREFIX and REST\_ROUTER correspond to the keywords 'port', 'prefix' and 'router' used in the parameter of the POST command.

If the message is a 'set prefix data' message, `_set_prefix_data` method is launched. If it's a 'set router type' message, `_set_router_type` is launched.

```

#MPLSmod: set router type
def _set_router_type(self, router):
    self.router_type = router

# MPLSmod: set port data method
def _set_prefix_data(self, prefix, port):
    cookie = 0x800
    prefix = self.prefix_data.add(prefix, port)
    # Set flow: IP packets aiming this
    # prefix are sent to the controller
```

```

priority = self._get_priority(PRIORITY_MPLS_PREFIX)
self.ofctl.set_packetin_flow(cookie, priority,
                             dl_type=ether.ETH_TYPE_IP,
                             dst_ip=prefix.address,
                             dst_mask=prefix.netmask)

return prefix.prefix_id

```

As can be seen in the code above the 'set router type' method only sets the attribute *router\_type* to the message passed through the REST API, which can be 'ler' or 'lsr'. When the *VlanRouter* object is created this attribute is set to 'ler' by default.

The method *\_set\_prefix\_data* stores the *prefix* (network/mask) and *port* values in the attribute *prefix\_data* and then writes a flow.

The attribute *prefix\_data* is a *PrefixData* object, which is a class that has been created when modifying the original code. This object stores *Prefix* objects, which encapsulate the prefix/port relation and contain a method to calculate if an IP address corresponds to that prefix:

```

# MPLSmod: class to store the prefix-port info
class PrefixData(dict):
    def __init__(self):
        super(PrefixData, self).__init__()
        self.prefix_id = 1
    # Does not check for overlaps yet

    def add(self, prefix, port):
        err_msg = 'Invalid [%s] value.' % REST_PREFIX
        nw_addr, mask, default_gw = nw_addr_aton(prefix, err_msg=err_msg)
        prefix = Prefix(nw_addr, mask, port, self.prefix_id)
        ip_str = ip_addr_ntoa(nw_addr)
        key = '%s/%d' % (ip_str, mask)
        self[key] = prefix
        self.prefix_id = self.prefix_id + 1
        return prefix

# MPLSmod: class to encapsulate the prefix-port relation
class Prefix(object):

    def __init__(self, address, netmask, port, prefix_id):
        self.prefix_id = prefix_id
        self.address = address
        self.netmask = netmask
        self.port = port

    def compare(self, ip):
        if ipv4_apply_mask(ip, self.netmask) == self.address:
            return True
        else:
            return False

```

The flows written by the method *set\_prefix\_data* in the case of router *s4*, using the REST commands shown previously,

would be the following:

```
cookie=0x800, duration=72.914s, table=0, n_packets=0, n_bytes=0,
priority=1, ip, nw_dst=10.0.1.0/24 actions=CONTROLLER:65535

cookie=0x800, duration=72.862s, table=0, n_packets=0, n_bytes=0,
priority=1, ip, nw_dst=10.0.2.0/24 actions=CONTROLLER:65535

cookie=0x800, duration=72.812s, table=0, n_packets=0, n_bytes=0,
priority=1, ip, nw_dst=10.0.3.0/24 actions=CONTROLLER:65535
```

This flows match the IPv4 packets heading to a certain network. When there's a match, the datapath sends the packets to the controller for further processing.

### 4.3.6 Storing labels and hosts

Information about labels and hosts should be stored in the controller if we want the system to perform as described in section 4.3.4.

#### Labels

In order to store the labels properly, two classes have been defined: *MplsLabel* and *MplsData*. The second class is a dictionary class, which functionality is to store *MplsLabel* objects.

```
# MPLSmod: class mapping IP addresses to labels
class MplsData(dict):

    def __init__(self):
        super(MplsData, self).__init__()

    def add(self, dpid, label_value, dst_ip):
        self[dpid][label_value] = dst_ip

# MPLSmod: class to encapsulate labels
class MplsLabel(object):

    def __init__(self, value=20):
        self.value = value

    def increase(self):
        self.value = self.value + 1
```

The *VlanRouter* class will have two extra attributes: one *MplsLabel* object that will allow to generate Labels, and a *MplsData* object to store them. The *MplsLabel* object will be created (with a default value of 20) in the constructor method while the *MplsData* object will be passed to the constructor as an argument. This last action is really important, as there's a *VlanRouter* object for each *Router* object, and each *Router* object is associated to one of the datapaths of the network. All of them should **share** the same *MplsData* object in order to access the information stored by the other *VlanRouter* objects. For this purpose, the constructor methods of *VlanRouter* and *Router* classes have been modified

to take the *MplsData* object as an argument. The *MplsData* object is created in the constructor of the *RestRouterAPI* class, and passed to each new *Router* object that is created.

## Hosts

Another necessary modification is to make the Ryu Application store relevant information about hosts. It is true that the original application has a method called *\_learning\_host\_mac*, but it only writes a flow in the OpenFlow Switch, and the information about the host is not stored.

To change this, we have created the classes *Host* and *HostDict* so the controller can store the data in a structured manner.

```
# MPLSmod: classes to store host data
class HostDict(dict):

    def __init__(self):
        super(HostDict, self).__init__()

    def add(self, ip, port, mac):
        self[ip] = Host(ip, port, mac)

class Host(object):

    def __init__(self, ip, port, mac):
        self.ip = ip
        self.port = port
        self.mac = mac
```

A *HostDict* object is added as an attribute to the *VlanRouter* constructor. We have also modified the method *\_learning\_host\_mac* so it stores the relevant data (IP, port, MAC) in the *HostDict* object before the flow is written.

### 4.3.7 Handling the packets that enter the network

The question that we want to answer in this section is: What should the controller do when it receives a *Packet-in* message from one of the LERs, containing an IP packet that needs to go through the MPLS network?

We will suppose, in order to illustrate this issue better, that host h1 (network 10.0.1.0/24) sends an ICMP message to host h3 (network 10.0.2.0/24).

After the ARP handling, the ICMP packet enters the router *s1* and matches one of the flows written during the REST setup:

```
cookie=0x800, duration=4.121s, table=0, n_packets=0, n_bytes=0,
priority=1, ip,nw_dst=10.0.2.0/24 actions=CONTROLLER:65535
```

The packet is sent to the controller inside a *Packet-In* message, which is passed until the *packet\_in\_handler* method of the *VlanRouter* object associated to the datapath *s1*. To handle this packets properly, the *packet\_in\_handler* method has been modified as follows:

```

def packet_in_handler(self, msg, header_list):
    # Check invalid TTL (for OpenFlow V1.2/1.3)
    ofproto = self.dp.ofproto
    if ofproto.OFP_VERSION == ofproto_v1_2.OFP_VERSION or \
        ofproto.OFP_VERSION == ofproto_v1_3.OFP_VERSION:
        if msg.reason == ofproto.OFPR_INVALID_TTL:
            self._packetin_invalid_ttl(msg, header_list)
            return

    # Analyze event type.

    if ARP in header_list:
        self._packetin_arp(msg, header_list)
        return
    # MPLSmod: If packetin is an MPLS packet
    if MPLS in header_list:
        if self.router_type == ROUTER_TYPE_LER:
            # LER method
            self.logger.info('MPLS packet entering LER', extra=self.sw_id)
            self._packetin_from_mpls_network(msg, header_list)

        elif self.router_type == ROUTER_TYPE_LSR:
            # LSR method
            self.logger.info('MPLS packet entering LSR', extra=self.sw_id)
            self._packetin_to_lsr(msg, header_list)

    elif IPV4 in header_list:
        self.logger.info('IPV4 packet',
            extra=self.sw_id)
        rt_ports = self.address_data.get_default_gw()
        if header_list[IPV4].dst in rt_ports:
            self.logger.info('Packet in: dst in rt ports',
                extra=self.sw_id)
            # Packet to router's port.
            if ICMP in header_list:
                if header_list[ICMP].type == icmp.ICMP_ECHO_REQUEST:
                    self._packetin_icmp_req(msg, header_list)
                    return
            elif TCP in header_list or UDP in header_list:
                self._packetin_tcp_udp(msg, header_list)
                return
        else:
            # OLDCODE: Packet to internal host or gateway router.
            # self._packetin_to_node(msg, header_list)

            # MPLSmod: Packets to be routed into the MPLS network
            self._packetin_to_mpls_network(msg, header_list)
            return

```

This method has been modified in several ways. If the packet is a MPLS packet then the method checks if the datapath is a LER or a LSR, and then launches the corresponding method.

In the case we are discussing (IPv4 packet entering the MPLS network) the modification that makes the label being pushed to the packet, and the packet being forwarded to the LSR is in the last lines of the method. The `_packetin_to_node` method has been substituted by the `_packetin_to_mpls_network` method, which is in charge of handling the packet.

```
# MPLSmod: method for IP to MPLS handling
def _packetin_to_mpls_network(self, msg, header_list):
    self.logger.info('Launching packetin_to_mpls_network method.',
                     extra=self.sw_id)

    if len(self.packet_buffer) >= MAX_SUSPENDPACKETS:
        self.logger.info('Packet is dropped, MAX_SUSPENDPACKETS exceeded.',
                         extra=self.sw_id)

        return

    in_port = self.ofctl.get_packetin_inport(msg)
    dst_ip = header_list[IPV4].dst
    src_ip = header_list[IPV4].src

    for key in self.prefix_data:
        prefix = self.prefix_data[key]
        if prefix.compare(dst_ip):
            # Write flow & packet out
            priority = self._get_priority(PRIORITY_PUSH_MPLS)
            cookie = 0x810
            out_port = int(prefix.port)
            self.ofctl.set_mpls_flow(cookie, priority, self.mpls_label.value,
                                     in_port, out_port, MPLS_PUSH_LABEL, nw_dst=dst_ip,
                                     nw_src=src_ip)
            self.ofctl.send_mpls_packet_out(in_port,
                                             out_port, msg.data, self.mpls_label.value, MPLS_PUSH_LABEL)
            self.mpls_data.add(self.dpid, self.mpls_label.value, dst_ip)
            self.mpls_label.increase()
            break
    # Else drop
```

The first thing this method does is to extract relevant information from the packet, such as the ingress port, source IP and destination IP. Then, it looks up in its *PrefixData* object the network the destination IP address belongs to. When a match is found, a flow is constructed with the output port found in the *Prefix* object that matched. This flow is constructed and set into the datapath with the method *set\_mpls\_flow* and then, the packet is processed and sent back to the switch with the method *send\_mpls\_packet\_out*. The implementation of these two methods is explained in section 4.3.10. Note that the label value that is passed to this methods is the current value of the *mpls\_label* attribute, which is a *MplsLabel* object.

After the *Packet Out* message has been sent, the controller stores the information related to the label and the datapath that pushes it in the *MplsData* object. After this is done, the value of the *mpls\_label* is increased, so a different label will be pushed into the next packet that comes through this method for this datapath.

The resulting flow is:

```
cookie=0x810, duration=5.229s, table=0, n_packets=0, n_bytes=0,
priority=2, ip, nw_src=10.0.1.1, nw_dst=10.0.2.3
```

```
actions=push_mpls:0x8847,set_field:20->mpls_label,output:3
```

This flow relates a source and a destination IP address to a MPLS label. In this case label 20.

### 4.3.8 Handling the packets that enter the LSR

Once the packet is sent through the right port it enters the LSR and matches the following flow:

```
cookie=0x24, duration=149.726s, table=0, n_packets=0,  
n_bytes=0, priority=1,mpls actions=CONTROLLER:65535
```

This flow has been installed when the *VlanRouter* object has been created. This is a modification we added to the application so every MPLS packet that enters a router is sent to the controller. The method used to install this flow is the *set\_packetin\_flow* with the ethertype set to MPLS. The action associated to this flow is to send the packet to the controller.

The datapath generates a *Packet-In* message and sends it to the controller. The controller passes this message to the modified *packet\_in\_handler* method explained in section 4.3.7.

As this packet is a MPLS packet and the *Packet-In* message was issued by a LSR, the method *\_packetin\_to\_lsr* is launched. Let's analyze this method in detail:

```
# MPLSmod: When packet enters LSR  
def _packetin_to_lsr(self, msg, header_list):  
  
    in_port = self.ofctl.get_packetin_inport(msg)  
    dst_ip = header_list[IPV4].dst  
    src_ip = header_list[IPV4].src  
    label_in = self.ofctl.get_packetin_mplslabel(msg)  
    self.logger.info('packet_in label: %s' % label_in,  
                    extra=self.sw_id)  
  
    address = 0  
    orig_id = self.get_origin_id_lsr(in_port)  
    self.logger.info('Origin ID: %s' % orig_id,  
                    extra=self.sw_id)  
    for label in self.mpls_data[orig_id]:  
        if label == label_in:  
            self.logger.info('Label match: %s' % str(label),  
                            extra=self.sw_id)  
            address = self.mpls_data[orig_id][label]  
            break  
    ...
```

First, the method extracts the relevant data from the packet: ingress port, destination IP, source IP and MPLS label. Using the ingress port as a reference the method *get\_origin\_id* finds the ID of the datapath that pushed the MPLS label. The method then checks in the *MplsData* object if the label of the packet matches any of the labels pushed by the datapath. This lines were wrote down for debugging reasons, but we use them to get the destination IP of the packet (again). We could safely remove them and set `address = dst_ip`

```

...
for key in self.prefix_data:
    prefix = self.prefix_data[key]
    if address == 0:
        self.logger.info('No label match! Dropping packet...',
                          extra=self.sw_id)
        break
    if prefix.compare(address):
        self.logger.info('Label match! Swapping label...',
                          extra=self.sw_id)
        # Create a new label:
        self.mpls_label.increase()
        # Write flow & packet out
        priority = self._get_priority(PRIORITY_SWAP_MPLS)
        cookie = 0x820
        out_port = int(prefix.port)

        self.ofctl.set_mpls_flow(cookie, priority, self.mpls_label.value,
                                  in_port, out_port, MPLS_SWAP_LABEL, nw_dst=dst_ip,
                                  nw_src=src_ip, oldlabel=label_in)
        self.ofctl.send_mpls_packet_out(in_port,
                                         out_port, msg.data, self.mpls_label.value, MPLS_SWAP_LABEL)
        self.mpls_data.add(self.dpid, self.mpls_label, dst_ip)
        break

```

In this piece of code, the application checks if the destination IP address belongs to one of the prefixes stored in the *PrefixData* object. When a match is found, a new label is created, a new flow implementing the label swap is set, and a *Packet Out* message is issued to the datapath. The resulting flow is:

```

cookie=0x820, duration=7.494s, table=0, n_packets=0, n_bytes=0,
priority=2,mpls,in_port=1,mpls_label=20
actions=pop_mpls:0x0800,push_mpls:0x8847,
set_field:21->mpls_label,output:2

```

The switch will swap the label 20 for the label 21 and send the packet through port 2.

### 4.3.9 Handling the packets that leave the network

Once the packet leaves the LSR, it enters another LER, and it's ready to leave the MPLS network. Once inside the LER (*s2* in this case), the packet matches the following flow:

```

cookie=0x24, duration=385.890s, table=0, n_packets=0,
n_bytes=0, priority=1,mpls actions=CONTROLLER:65535

```

Therefore, a *Packet-In* message is generated by the datapath and sent to the controller, with the mpls packet inside. The controller passes this message to the modified *packet\_in\_handler* method explained in section 4.3.7.

As this packet is a MPLS packet and the *Packet-In* message was issued by a LSR, the method *\_packetin\_from\_mpls\_network* is launched. Let's analyze this method in detail:



```

def _packetin_from_mpls_network(self, msg, header_list):
    # Extract data
    in_port = self.ofctl.get_packetin_inport(msg)
    dst_ip = header_list[IPV4].dst
    src_ip = header_list[IPV4].src
    srcip = ip_addr_ntoa(header_list[IPV4].src)
    dstip = ip_addr_ntoa(dst_ip)
    label_in = self.ofctl.get_packetin_mplslabel(msg)

    if dst_ip not in self.hosts:

        # Send ARP to learn the MAC address
        address = self.address_data.get_data(ip=dst_ip)

        if address is not None:
            log_msg = 'Receive IP packet from [%s] to an internal host [%s].'
            self.logger.info(log_msg, srcip, dstip, extra=self.sw_id)
            src_ip = address.default_gw

        if src_ip is not None:
            self.packet_buffer.add(in_port, header_list, msg.data)
            self.send_arp_request(src_ip, dst_ip, in_port=in_port)
            self.logger.info('Send ARP request (flood)', extra=self.sw_id)

    ...

```

This piece of code extracts some relevant data from the packet, such as the ingress port, the destination IP, the source IP and the MPLS label. If the destination IP address is not in the known hosts table, then the MPLS packet is buffered and an ARP packet is generated to discover the host.

```

...

else:
    # Write flow & packet out
    priority = self._get_priority(PRIORITY_POP_MPLS)
    cookie = 0x830
    out_port = self.hosts[dst_ip].port
    dl_src = self.port_data[out_port].mac
    dl_dst = self.hosts[dst_ip].mac
    self.ofctl.set_mpls_flow(cookie, priority, self.mpls_label.value,
                             in_port, out_port, MPLS_POP_LABEL, nw_dst=dst_ip,
                             dst_mac=dl_dst, src_mac=dl_src, oldlabel=label_in)
    self.ofctl.send_mpls_packet_out(in_port, out_port, msg.data,
                                     self.mpls_label.value, MPLS_POP_LABEL, dst_mac=dl_dst,
                                     src_mac=dl_src)

```

If the destination IP is in the known hosts table, the port is extracted from such table to build the action of the flow that will be installed. A flow is set and a *Packet Out* message is issued to the datapath. The flow written to the datapath is:

```
cookie=0x830, duration=271.066s, table=0, n_packets=0, n_bytes=0,
priority=2, mpls, mpls_label=21
actions=pop_mpls:0x0800, set_field:0e:8e:39:5f:7b:d9->eth_src,
set_field:00:00:00:00:02:03->eth_dst, output:1
```

In this particular case, the flows matches MPLS packets with label 21. When there's a match, the datapath pops the MPLS label, changes the source and destination MAC addresses and forwards the packet through port 1.

To understand better how this part works, we'll make a simple walkthrough:

- A MPLS packet with certain label enters the LER.
- The packet is sent to the controller, and passed to the `_packetin_from_mpls_network` method
- The destination is an unknown host so the packet is buffered and an ARP request is generated and sent to the destination network.
- An ARP reply enters the datapath, and it's forwarded to the controller.
- The controller learns the host, removes the MPLS packet from the buffer and sends it again through the pipeline of the OpenFlow switch. The switch sends the packet again to the controller, which passes it again to the `_packetin_from_mpls_network` method
- This time, the destination is a known host, so a `FlowMod` and a `Packet Out` message are generated and sent to the datapath.

#### 4.3.10 Methods to generate and set the MPLS flows

A few new methods of the class `OfCtl` have been shown in the previous section. We will see them in detail in this section. This methods are:

- `set_mpls_flow(self, cookie, priority, label, in_port, out_port, action, dl_vlan=0, nw_src=0, src_mask=32, nw_dst=0, dst_mask=32, src_mac=0, dst_mac=0, idle_timeout=0, oldlabel=0)`
- `send_mpls_packet_out(self, in_port, out_port, data, label, action, dst_mac=0, src_mac=0)`

The method `set_mpls_flow` has been implemented in order to provide a function that handles flow writing in a similar way to the `set_routing_flow` or `set_packetin_flow` methods. The code of this method is:

```
# MPLSmod: method to add mpls flows
def set_mpls_flow(self, cookie, priority, label, in_port, out_port, action,
                  dl_vlan=0, nw_src=0, src_mask=32, nw_dst=0, dst_mask=32,
                  src_mac=0, dst_mac=0, idle_timeout=0, oldlabel=0):

    parser = self.dp.ofproto_parser
    if action == MPLS_PUSH_LABEL:
        dl_type = ether.ETH_TYPE_IP
        actions = [parser.OFPActionPushMpls(ethertype=34887),
                  parser.OFPActionSetField(mpls_label=label),
                  parser.OFPActionOutput(out_port)]
```

```

        self.set_flow(cookie, priority, dl_type=dl_type, dl_vlan=dl_vlan,
                      nw_src=nw_src, src_mask=src_mask,
                      nw_dst=nw_dst, dst_mask=dst_mask,
                      idle_timeout=idle_timeout, actions=actions)

    elif action == MPLS_SWAP_LABEL:
        dl_type = ether.ETH_TYPE_MPLS
        actions = [parser.OFPActionPopMpls(),
                  parser.OFPActionPushMpls(ethertype=34887),
                  parser.OFPActionSetField(mpls_label=label),
                  parser.OFPActionOutput(out_port)]
        match = parser.OFPMatch(in_port=in_port,
                                eth_type=dl_type, mpls_label=oldlabel)
        self.set_my_flow(cookie, priority, match,
                         idle_timeout=idle_timeout, actions=actions)

    elif action == MPLS_POP_LABEL:
        dl_type = ether.ETH_TYPE_MPLS
        actions = [parser.OFPActionPopMpls(),
                  parser.OFPActionSetField(eth_src=src_mac),
                  parser.OFPActionSetField(eth_dst=dst_mac),
                  parser.OFPActionOutput(out_port)]
        match = parser.OFPMatch(eth_type=dl_type, mpls_label=oldlabel)
        self.set_my_flow(cookie, priority, match,
                         idle_timeout=idle_timeout, actions=actions)

```

The method checks if the action to be performed is to push, swap or pop a label, takes the parameters it needs to generate a match and an action and passes them to another custom method called *set\_my\_flow*. The exception to this is the case of the 'push' action, which only generates an action and uses the method *set\_flow* to generate the match and set the flow.

This is the code of the method *set\_my\_flow*:

```

# MPLSmod: custom flow method
def set_my_flow(self, cookie, priority, match, idle_timeout=0, actions=None):
    ofp = self.dp.ofproto
    ofp_parser = self.dp.ofproto_parser
    cmd = ofp.OFPFC_ADD

    inst = [ofp_parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS,
                                             actions)]

    m = ofp_parser.OFPFlowMod(self.dp, cookie, 0, 0, cmd, idle_timeout,
                               0, priority, UINT32_MAX, ofp.OFPP_ANY,
                               ofp.OFPG_ANY, 0, match, inst)

    self.dp.send_msg(m)

```

The method creates an 'Apply-Actions' instruction with the provided actions and generates a *FlowMod* message with such instructions plus the match passed as an argument.

The method *send\_mpls\_packet\_out* creates list of actions to be applied to the packet and sends it back to the datapath in a *Packet Out* message. When the datapath receives such message, it applies the list of actions.

```

def send_mpls_packet_out(self, in_port, out_port, data, label, action,
                        dst_mac=0, src_mac=0):

    parser = self.dp.ofproto_parser
    if action == MPLS_PUSH_LABEL:
        actions = [parser.OFPActionPushMpls(ethertype=34887),
                  parser.OFPActionSetField(mpls_label=label),
                  parser.OFPActionOutput(out_port)]
    elif action == MPLS_SWAP_LABEL:
        actions = [parser.OFPActionPopMpls(),
                  parser.OFPActionPushMpls(ethertype=34887),
                  parser.OFPActionSetField(mpls_label=label),
                  parser.OFPActionOutput(out_port)]
    elif action == MPLS_POP_LABEL:
        actions = [parser.OFPActionPopMpls(),
                  parser.OFPActionSetField(eth_src=src_mac),
                  parser.OFPActionSetField(eth_dst=dst_mac),
                  parser.OFPActionOutput(out_port)]
    self.dp.send_packet_out(buffer_id=UINT32_MAX, in_port=in_port,
                            actions=actions, data=data)

```

Depending on the action to be performed (push,pop or swap) it builds a different Action List. Then, uses the method *send\_packet\_out* to forward the packet and the actions to the datapath.

### 4.3.11 Testing the MPLS application

Now that the main pieces of code have been reviewed, let's test the application and see the results of the wireshark captures to make sure everything works as planned.

To load everything, we first launch the topology using the python script for mininet (complete code in section 4.5.2):

```
$ sudo python topology
```

Then we execute our application with the following command:

```
$ ryu-manager rest_MPLS_router.py
```

The complete source code of the application can be found on section 4.5.2

Once the application is running, and the routers have joined the controller, it's time to launch the script that configures the network through REST API. The code of the script can be found in section 4.5.2 and configures the network as shown in Figure 4.12. We would launch the scriot with something like:

```
$ ./MPLSconfig.sh
```

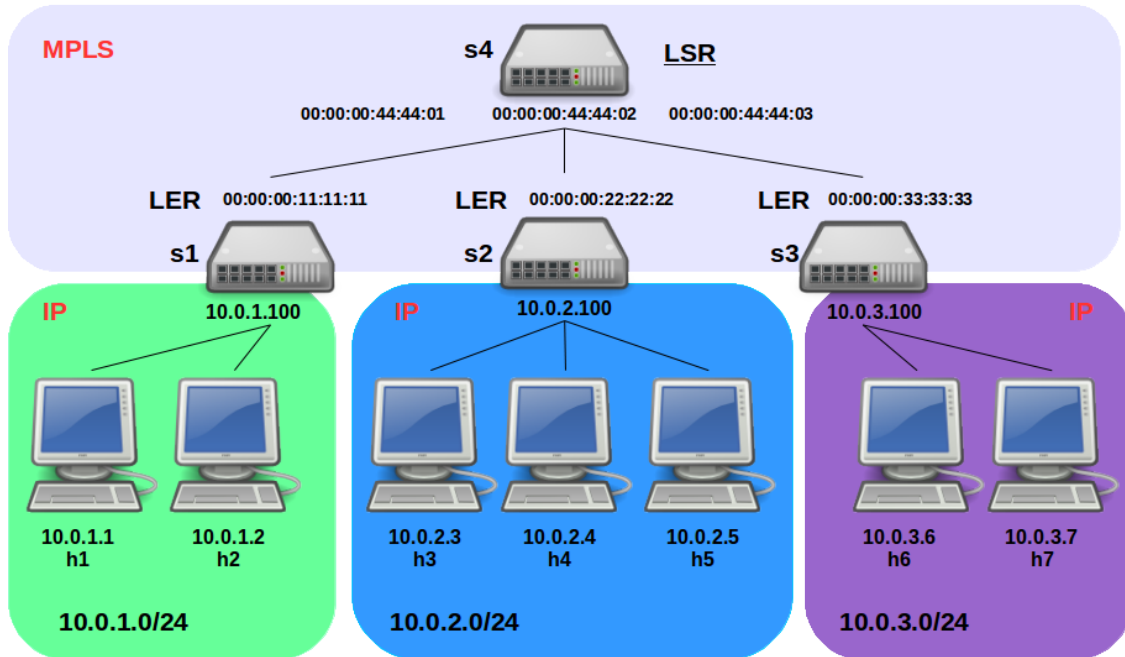


Figure 4.12: Complete network setup after the REST configuration

### Simple ping

If we send an ICMP message from host h3 (network 10.0.2.0/24) to host h6 (network 10.0.3.0/24), we instantly see that there's reachability:

```
mininet> h3 ping -c1 h6
PING 10.0.3.6 (10.0.3.6) 56(84) bytes of data.
64 bytes from 10.0.3.6: icmp_seq=1 ttl=64 time=105 ms

--- 10.0.3.6 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 105.148/105.148/105.148/0.000 ms
```

The captures show that labels have been pushed, swapped and popped, correctly. In figure 4.13, we can see the ICMP request and reply. The label has been correctly popped from the reply, and the packet has been forwarded to the right host.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	00:00:00_00:02:03	Broadcast	ARP	42	Who has 10.0.2.100? Tell 10.0.2.3
2	0.008976000	26:a3:0b:d2:38:87	00:00:00_00:02:03	ARP	42	10.0.2.100 is at 26:a3:0b:d2:38:87 (duplicate use of 10.0.2.3 detected!)
3	0.009165000	10.0.2.3	10.0.3.6	ICMP	98	Echo (ping) request id=0x0ae1, seq=1/256, ttl=64 (reply in 4)
4	0.104838000	10.0.3.6	10.0.2.3	ICMP	98	Echo (ping) reply id=0x0ae1, seq=1/256, ttl=64 (request in 3)

> Frame 3: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0  
 > Ethernet II, Src: 00:00:00\_00:02:03 (00:00:00:00:02:03), Dst: 26:a3:0b:d2:38:87 (26:a3:0b:d2:38:87)  
 > Internet Protocol Version 4, Src: 10.0.2.3 (10.0.2.3), Dst: 10.0.3.6 (10.0.3.6)  
 > Internet Control Message Protocol

Figure 4.13: Capture from interface s2-eth1

In figure 4.14, we can see the ICMP request and reply inside the MPLS network. The request just left the LER and carries the label 20. The reply comes from the LSR and carries the label 22.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.2.3	10.0.3.6	ICMP	102	Echo (ping) request id=0x0b3d, seq=1/256, ttl=64 (reply in 2)
2	0.002233000	10.0.3.6	10.0.2.3	ICMP	102	Echo (ping) reply id=0x0b3d, seq=1/256, ttl=64 (request in 1)

```

Frame 1: 102 bytes on wire (816 bits), 102 bytes captured (816 bits) on interface 0
Ethernet II, Src: 00:00:00_00:02:03 (00:00:00:00:02:03), Dst: 26:a3:0b:d2:38:87 (26:a3:0b:d2:38:87)
  Destination: 26:a3:0b:d2:38:87 (26:a3:0b:d2:38:87)
  Source: 00:00:00_00:02:03 (00:00:00:00:02:03)
  Type: MPLS label switched packet (0x8847)
MultiProtocol Label Switching Header, Label: 20, Exp: 0, S: 1, TTL: 64
Internet Protocol Version 4, Src: 10.0.2.3 (10.0.2.3), Dst: 10.0.3.6 (10.0.3.6)
Internet Control Message Protocol
  
```

Figure 4.14: Capture from interface s4-eth2

In figure 4.15, we can see the ICMP request and reply inside the MPLS network. The request just left the LSR and carries the label 21. The reply comes from the LER and carries the label 20. We can confirm that label swapping has been done correctly.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.0.2.3	10.0.3.6	ICMP	102	Echo (ping) request id=0x0b7f, seq=1/256, ttl=64 (reply in 2)
2	0.001057000	10.0.3.6	10.0.2.3	ICMP	102	Echo (ping) reply id=0x0b7f, seq=1/256, ttl=64 (request in 1)

```

Frame 1: 102 bytes on wire (816 bits), 102 bytes captured (816 bits) on interface 0
Ethernet II, Src: 00:00:00_00:02:03 (00:00:00:00:02:03), Dst: 26:a3:0b:d2:38:87 (26:a3:0b:d2:38:87)
  Destination: 26:a3:0b:d2:38:87 (26:a3:0b:d2:38:87)
  Source: 00:00:00_00:02:03 (00:00:00:00:02:03)
  Type: MPLS label switched packet (0x8847)
MultiProtocol Label Switching Header, Label: 21, Exp: 0, S: 1, TTL: 64
Internet Protocol Version 4, Src: 10.0.2.3 (10.0.2.3), Dst: 10.0.3.6 (10.0.3.6)
Internet Control Message Protocol
  
```

Figure 4.15: Capture from interface s4-eth3

In figure 4.16, we can see the ICMP request and reply inside the 10.0.3.0/24 network. The label has been correctly popped from the request, and the packet has been forwarded to the right host.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	4e:aa:4c:a0:e8:76	Broadcast	ARP	42	Who has 10.0.3.6? Tell 10.0.3.100
2	0.000192000	00:00:00_00:03:06	4e:aa:4c:a0:e8:76	ARP	42	10.0.3.6 is at 00:00:00:00:03:06
3	0.021572000	10.0.2.3	10.0.3.6	ICMP	98	Echo (ping) request id=0x5033, seq=1/256, ttl=64 (reply in 4)
4	0.021733000	10.0.3.6	10.0.2.3	ICMP	98	Echo (ping) reply id=0x5033, seq=1/256, ttl=64 (request in 3)
5	5.034293000	00:00:00_00:03:06	4e:aa:4c:a0:e8:76	ARP	42	Who has 10.0.3.100? Tell 10.0.3.6
6	5.042204000	4e:aa:4c:a0:e8:76	00:00:00_00:03:06	ARP	42	10.0.3.100 is at 4e:aa:4c:a0:e8:76 (duplicate use of 10.0.3.6 detected!)

```

Frame 3: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface 0
Ethernet II, Src: 4e:aa:4c:a0:e8:76 (4e:aa:4c:a0:e8:76), Dst: 00:00:00_00:03:06 (00:00:00:00:03:06)
  Destination: 00:00:00_00:03:06 (00:00:00:00:03:06)
  Source: 4e:aa:4c:a0:e8:76 (4e:aa:4c:a0:e8:76)
  Type: IP (0x8000)
Internet Protocol Version 4, Src: 10.0.2.3 (10.0.2.3), Dst: 10.0.3.6 (10.0.3.6)
Internet Control Message Protocol
  
```

Figure 4.16: Capture from interface s3-eth1

## Reachability

We can also see that all the hosts reach each other. If we perform a reachability test:

```

mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7
h2 -> h1 h3 h4 h5 h6 h7
h3 -> h1 h2 h4 h5 h6 h7
h4 -> h1 h2 h3 h5 h6 h7
h5 -> h1 h2 h3 h4 h6 h7
h6 -> h1 h2 h3 h4 h5 h7
h7 -> h1 h2 h3 h4 h5 h6
*** Results: 0% dropped (42/42 received)

```

If we capture the packets in one of the interfaces of the LSR (for instance, `s4-eth2`) we can see that every packet coming from the LSR has a different label. Also we see that every packet coming from the LER (`s2`) have different labels as well. In figure 4.17 we see such capture, highlighting an ICMP request traveling from host h2 to host h4 with label 33.

No.	Time	Source	Destination	Protocol	Length	Info
2	0.030294000	10.0.2.3	10.0.1.1	ICMP	102	Echo (ping) reply id=0x5400, seq=1/256, ttl=64 (request in 1)
3	0.147484000	10.0.1.1	10.0.2.4	ICMP	102	Echo (ping) request id=0x54d7, seq=1/256, ttl=64 (reply in 4)
4	0.209107000	10.0.2.4	10.0.1.1	ICMP	102	Echo (ping) reply id=0x54d7, seq=1/256, ttl=64 (request in 3)
5	0.306506000	10.0.1.1	10.0.2.5	ICMP	102	Echo (ping) request id=0x54d8, seq=1/256, ttl=64 (reply in 6)
6	0.384660000	10.0.2.5	10.0.1.1	ICMP	102	Echo (ping) reply id=0x54d8, seq=1/256, ttl=64 (request in 5)
7	0.820776000	10.0.1.2	10.0.2.3	ICMP	102	Echo (ping) request id=0x54dc, seq=1/256, ttl=64 (reply in 8)
8	0.844111000	10.0.2.3	10.0.1.2	ICMP	102	Echo (ping) reply id=0x54dc, seq=1/256, ttl=64 (request in 7)
9	0.916884000	10.0.1.2	10.0.2.4	ICMP	102	Echo (ping) request id=0x54dd, seq=1/256, ttl=64 (reply in 10)
10	0.935505000	10.0.2.4	10.0.1.2	ICMP	102	Echo (ping) reply id=0x54dd, seq=1/256, ttl=64 (request in 9)
11	1.011460000	10.0.1.2	10.0.2.5	ICMP	102	Echo (ping) request id=0x54de, seq=1/256, ttl=64
12	1.031003000	10.0.2.5	10.0.1.2	ICMP	102	Echo (ping) reply id=0x54de, seq=1/256, ttl=64 (request in 11)
13	1.310136000	10.0.2.3	10.0.1.1	ICMP	102	Echo (ping) request id=0x54e1, seq=1/256, ttl=64 (reply in 14)

▶ Frame 9: 102 bytes on wire (816 bits), 102 bytes captured (816 bits) on interface 0  
 ▼ Ethernet II, Src: 00:00:00:00:01:02 (00:00:00:00:01:02), Dst: da:d1:b2:18:b1:ae (da:d1:b2:18:b1:ae)  
 ▶ Destination: da:d1:b2:18:b1:ae (da:d1:b2:18:b1:ae)  
 ▶ Source: 00:00:00:00:01:02 (00:00:00:00:01:02)  
 Type: MPLS Label switched packet (0x8847)  
 ▶ MultiProtocol Label Switching Header, Label: 33, Exp: 0, S: 1, TTL: 64  
 ▶ Internet Protocol Version 4, Src: 10.0.1.2 (10.0.1.2), Dst: 10.0.2.4 (10.0.2.4)  
 ▶ Internet Control Message Protocol

Figure 4.17: Capture from interface `s4-eth2`

## 4.4 Third approach: Next steps

In this section we will discuss how a third approach for building a robust ryu application for MPLS networks should be.

### 4.4.1 Simplification

We will begin with some simplifications that could be done to the application. This application works but some things could be done in a different way:

- When setting the prefix data through the API rest there's no need to write a flow each time. A single low priority flow matching all IP packets with an associated action of sending the packets to the controller will be enough. This flow could be installed when the `VlanRouter` object associated to a certain datapath is created.
- Also, writing flows for IP packets in the LSR does not make a much sense. This could also be simplified.

- The *MplsData* object could be avoided, as all the information needed for packet processing is already in the packet header. This object can be useful however, for a network administrator who wants to extract information about the labels being used in the network through the REST API

## 4.4.2 Generalization

This application is too specific, it could be generalized in some ways:

- Some of the methods have been specified for this topology for simplicity, such methods are *get\_lsr\_id* and *get\_origin\_id\_lsr*. Changing the code contained in these methods will probably be necessary if a different topology is used.
- Although being based on the IP router application, this new application modifies some methods that make the application not compatible with IP routing, even if it can process routing information. The ideal application (What we were originally seeking for) is an application that can configure some routers to work as classic IP routers with routes, and some others like MPLS LER or LSR.
- Testing the application in different scenarios will also improve its adaptability: adding more networks, adding more LSRs, etc.

## 4.4.3 Robustness and bug fixing

There are also a couple of things that could be done to improve the application's robustness.

- The simple application discussed in previous section only supports POST commands for its MPLS capabilities. GET and DELETE commands should also be implemented in order to make a robust application
- In some of the captures a message is displayed in the ARP replies: 'Duplicate use of xxx.xxx.xxx.xxx detected! This is a bug that was included in the original application and should be fixed.

## 4.5 Source code

This section provides the source code of every application analyzed and explained during this chapter.

### 4.5.1 First approach

#### Configuration script

```
#!/bin/bash
# Topology: sudo mn --topo linear,3 --mac --switch ovsk,datapath=user
# --controller remote
# Configure switch to use OpenFlow 1.3
echo "Setting Switches to work with OpenFlow 1.3"

for i in s1 s2 s3; do
```



```

sudo ovs-vsctl set bridge $i protocols=OpenFlow13
done

# Clear flow tables
echo "Clearing Flow tables.."

for i in s1 s2 s3; do
sudo ovs-ofctl -O OpenFlow13 del-flows $i
done

#Configure switches with a unique ID
echo "Setting switches ID"
sudo ovs-vsctl set bridge s1 other-config:datapath-id=0000000000000001
sudo ovs-vsctl set bridge s2 other-config:datapath-id=0000000000000002
sudo ovs-vsctl set bridge s3 other-config:datapath-id=0000000000000003

# Launch the application
echo "Launching the MPLS application..."
ryu-manager --verbose mpls_controller.py

```

## Application

```

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
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import mpls

class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        self.label = 20
        self.dst_to_label = {}

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        # install table-miss flow entry
        #
        # We specify NO BUFFER to max_len of the output action due to

```



```

# If ARP
if ethtype == 2054:
    self.arpHandler(msg)
# If IPV4
elif ethtype == 2048:
    self.ipv4Handler(msg)
#If MPLS unicast
elif ethtype == 34887:
    self.mplsHandler(msg)

def arpHandler(self, msg):
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']
    dpid = datapath.id
    self.logger.info("Launching ARP handler for datapath %s", dpid)
    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]

    dst = eth.dst
    src = eth.src
    ethtype = eth.ethertype
    # learn a mac address to avoid FLOOD next time.
    self.mac_to_port[dpid][src] = in_port

    if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
    else:
        out_port = ofproto.OFPP_FLOOD

    actions = [parser.OFPActionOutput(out_port)]

    # install a flow to avoid packet_in next time
    if out_port != ofproto.OFPP_FLOOD:
        match = parser.OFPMatch(in_port=in_port, eth_src=src, eth_dst=dst,
                                eth_type=ethertype)
        self.logger.info("Flow match: in_port=%s, src=%s, dst=%s, type=ARP",
                        in_port, src, dst)
        self.logger.info("Flow actions: out_port=%s",
                        out_port)
        # verify if we have a valid buffer_id, if yes avoid to send both
        # flow_mod & packet_out
        if msg.buffer_id != ofproto.OFP_NO_BUFFER:
            self.add_flow(datapath, 1, match, actions, msg.buffer_id)
            return
        else:
            self.add_flow(datapath, 1, match, actions)
    data = None
    if msg.buffer_id == ofproto.OFP_NO_BUFFER:
        data = msg.data

```

```
out = parser.OFPPacketOut (datapath=datapath, buffer_id=msg.buffer_id,
                           in_port=in_port, actions=actions, data=data)
datapath.send_msg(out)
```

```
def ipv4Handler(self, msg):
    # Variables needed:
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']
    dpid = datapath.id
    self.logger.info("Launching IPV4 handler for datapath %s", dpid)
    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]
    dst = eth.dst
    #src = eth.src
    ethtype = eth.ethertype
    # If the packet is IPV4, it means that the datapath is a LER
    # IPV4 packets that come through in_port with this destination
    match = parser.OFPMatch(in_port=in_port, eth_dst=dst, eth_type=ethtype)
    # We relate a label to the destination: We select an unused label
    self.label = self.label + 1
    self.dst_to_label[dpid][dst] = self.label

    # Set the out_port using the relation learnt with the ARP packet
    out_port = self.mac_to_port[dpid][dst]
    # Set the action to be performed by the datapath
    actions = [parser.OFPACTIONPushMpls(ethertype=34887,type_=None, len_=None),
               parser.OFPACTIONSetField(mpls_label=self.label),
               parser.OFPACTIONOutput(out_port)]
    self.logger.info("Flow match: in_port=%s, dst=%s, type=IP",
                    in_port, dst)
    self.logger.info("Flow actions: pushMPLS=%s, out_port=%s",
                    self.label, out_port)
    # Install a flow
    # verify if we have a valid buffer_id, if yes avoid to send both
    # flow_mod & packet_out
    if msg.buffer_id != ofproto.OFP_NO_BUFFER:
        self.add_flow(datapath, 1, match, actions, msg.buffer_id)
        return
    else:
        self.add_flow(datapath, 1, match, actions)
    data = None
    if msg.buffer_id == ofproto.OFP_NO_BUFFER:
        data = msg.data
    out = parser.OFPPacketOut (datapath=datapath, buffer_id=msg.buffer_id,
                              in_port=in_port, actions=actions, data=data)
    datapath.send_msg(out)
```

```

def mplsHandler(self,msg):

    # Variables needed:
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']
    dpid = datapath.id
    self.logger.info("Launching MPLS Handler for datapath %s", dpid)
    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]
    mpls_proto = pkt.get_protocol(mpls.mpls)
    dst = eth.dst
    #src = eth.src
    ethtype = eth.ethertype
    # The switch can be a LSR or a LER, but the match is the same
    match = parser.OFPMatch(in_port=in_port, eth_dst=dst, eth_type=ethtype,
        mpls_label=mpls_proto.label)
    self.logger.info("Flow match: in_port=%s, dst=%s, type=IP, label=%s",
        in_port, dst, mpls_proto.label)
    # Set the out_port using the relation learnt with the ARP packet
    out_port = self.mac_to_port[dpid][dst]
    # we must check the switch ID in order to decide the proper action
    if dpid == 2:
        # The switch is a LSR
        # New label
        self.label = self.label + 1
        # Switch labels
        actions = [parser.OFPACTIONPopMpls(),
            parser.OFPACTIONPushMpls(),
            parser.OFPACTIONSetField(mpls_label=self.label),
            parser.OFPACTIONOutput(out_port)]
        self.logger.info("Flow actions: switchMPLS=%s, out_port=%s",
            self.label, out_port)
    else:
        # The switch is a LER
        # Pop that label!
        actions = [parser.OFPACTIONPopMpls(),
            parser.OFPACTIONOutput(out_port)]
        self.logger.info("Flow actions: popMPLS, out_port=%s", out_port)

    # Install a flow
    # verify if we have a valid buffer_id, if yes avoid to send both
    # flow_mod & packet_out
    if msg.buffer_id != ofproto.OFP_NO_BUFFER:
        self.add_flow(datapath, 1, match, actions, msg.buffer_id)
        return
    else:
        self.add_flow(datapath, 1, match, actions)
    data = None
    if msg.buffer_id == ofproto.OFP_NO_BUFFER:

```

```
        data = msg.data
    out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                              in_port=in_port, actions=actions, data=data)
    datapath.send_msg(out)
```

## 4.5.2 Second approach

### Topology script

```
#!/usr/bin/python
# The goal of this script is to define the topology to develop the second
# approach of

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

class Approach2Topo(Topo):

    def __init__(self, cpu=.1, max_queue_size=None, **params):

        # Initialize topo
        Topo.__init__(self, **params)

        # Hosts and switches
        s1 = self.addSwitch('s1')
        s2 = self.addSwitch('s2')
        s3 = self.addSwitch('s3')
        s4 = self.addSwitch('s4')

        h1 = self.addHost('h1')
        h2 = self.addHost('h2')
        h3 = self.addHost('h3')
        h4 = self.addHost('h4')
        h5 = self.addHost('h5')
        h6 = self.addHost('h6')
        h7 = self.addHost('h7')

        # Wire h1 and h2 to s1
        self.addLink(h1, s1)
        self.addLink(h2, s1)

        # Wire h3, h4 and h5 to s2
        self.addLink(h3, s2)
        self.addLink(h4, s2)
```

```

self.addLink(h5, s2)

# Wire h6 and h7 to s3
self.addLink(h6, s3)
self.addLink(h7, s3)

# Wire switches
self.addLink(s4, s1)
self.addLink(s4, s2)
self.addLink(s4, s3)

def setup():
    topo = Approach2Topo()
    # We use Open vSwitch and OpenFlow 1.3
    switch = partial(OVSSwitch, protocols='OpenFlow13', datapath='user')
    #controller
    net = Mininet(topo, controller=RemoteController,
                  switch=switch, cleanup=True)

    # Setting up hosts
    net['h1'].setIP('10.0.1.1/24')
    net['h1'].setMAC('00:00:00:00:01:01')
    net['h1'].cmd('route add default gw 10.0.1.100')

    net['h2'].setIP('10.0.1.2/24')
    net['h2'].setMAC('00:00:00:00:01:02')
    net['h2'].cmd('route add default gw 10.0.1.100')

    net['h3'].setIP('10.0.2.3/24')
    net['h3'].setMAC('00:00:00:00:02:03')
    net['h3'].cmd('route add default gw 10.0.2.100')

    net['h4'].setIP('10.0.2.4/24')
    net['h4'].setMAC('00:00:00:00:02:04')
    net['h4'].cmd('route add default gw 10.0.2.100')

    net['h5'].setIP('10.0.2.5/24')
    net['h5'].setMAC('00:00:00:00:02:05')
    net['h5'].cmd('route add default gw 10.0.2.100')

    net['h6'].setIP('10.0.3.6/24')
    net['h6'].setMAC('00:00:00:00:03:06')
    net['h6'].cmd('route add default gw 10.0.3.100')

    net['h7'].setIP('10.0.3.7/24')
    net['h7'].setMAC('00:00:00:00:03:07')
    net['h7'].cmd('route add default gw 10.0.3.100')

    # Setting up routers
    net['s1'].cmd('ifconfig s1-eth3 hw ether 00:00:00:11:11:11')

```

```

net['s2'].cmd('ifconfig s2-eth4 hw ether 00:00:00:22:22:22')
net['s3'].cmd('ifconfig s3-eth3 hw ether 00:00:00:33:33:33')
net['s4'].cmd('ifconfig s4-eth1 hw ether 00:00:00:44:44:01')
net['s4'].cmd('ifconfig s4-eth2 hw ether 00:00:00:44:44:02')
net['s4'].cmd('ifconfig s4-eth3 hw ether 00:00:00:44:44:03')

net.start()
CLI(net)
net.stop

if __name__ == '__main__':
    # Tell mininet to print useful information
    setLogLevel('info')
    setup()

```

### MPLS REST configuration script

```

#!/bin/bash
# MPLSconfig.sh

# Setting router IP addresses using the REST API

##### Addresses for router s1 #####
# Network 1
curl -X POST -d '{"address":"10.0.1.100/24"}'
http://localhost:8080/router/000000000000000001

##### Addresses for router s2 #####
# Network 2
curl -X POST -d '{"address":"10.0.2.100/24"}'
http://localhost:8080/router/000000000000000002

##### Addresses for router s3 #####
# Network 3
curl -X POST -d '{"address":"10.0.3.100/24"}'
http://localhost:8080/router/000000000000000003

##### Prefix mapping for router s1 #####
curl -X POST -d '{"prefix":"10.0.2.0/24", "port":"3"}'
http://localhost:8080/router/000000000000000001

curl -X POST -d '{"prefix":"10.0.3.0/24", "port":"3"}'
http://localhost:8080/router/000000000000000001

##### Prefix mapping for router s2 #####
curl -X POST -d '{"prefix":"10.0.1.0/24", "port":"4"}'
http://localhost:8080/router/000000000000000002

curl -X POST -d '{"prefix":"10.0.3.0/24", "port":"4"}'
http://localhost:8080/router/000000000000000002

```



```

##### Prefix mapping for router s3 #####
curl -X POST -d '{"prefix":"10.0.1.0/24", "port":"3"}'
http://localhost:8080/router/000000000000000003

curl -X POST -d '{"prefix":"10.0.2.0/24", "port":"3"}'
http://localhost:8080/router/000000000000000003

##### Prefix mapping for router s4 #####
curl -X POST -d '{"prefix":"10.0.1.0/24", "port":"1"}'
http://localhost:8080/router/000000000000000004

curl -X POST -d '{"prefix":"10.0.2.0/24", "port":"2"}'
http://localhost:8080/router/000000000000000004

curl -X POST -d '{"prefix":"10.0.3.0/24", "port":"3"}'
http://localhost:8080/router/000000000000000004

##### Setting router s4 as LSR #####
curl -X POST -d '{"router":"lsr"}'
http://localhost:8080/router/000000000000000004

```

## MPLS Ryu Application Code

```

import logging
import numbers
import socket
import struct

import json
from webob import Response

from ryu.app.wsgi import ControllerBase
from ryu.app.wsgi import WSGIApplication
from ryu.base import app_manager
from ryu.controller import dpset
from ryu.controller import ofp_event
from ryu.controller.handler import set_ev_cls
from ryu.controller.handler import MAIN_DISPATCHER
from ryu.exception import OFPUnknownVersion
from ryu.exception import RyuException
from ryu.lib import dpid as dpid_lib
from ryu.lib import hub
from ryu.lib import mac as mac_lib
from ryu.lib import addrconv
from ryu.lib.packet import arp
from ryu.lib.packet import ethernet
from ryu.lib.packet import icmp
from ryu.lib.packet import ipv4
from ryu.lib.packet import packet

```

```

from ryu.lib.packet import tcp
from ryu.lib.packet import udp
from ryu.lib.packet import vlan
from ryu.ofproto import ether
from ryu.ofproto import inet
from ryu.ofproto import ofproto_v1_0
from ryu.ofproto import ofproto_v1_2
from ryu.ofproto import ofproto_v1_3
# MPLSmod: library for mpls
from ryu.lib.packet import mpls

# =====
#           REST API
# =====
#
# Note: specify switch and vlan group, as follows.
# {switch_id} : 'all' or switchID
# {vlan_id}   : 'all' or vlanID
#
# 1. get address data and routing data.
#
# * get data of no vlan
# GET /router/{switch_id}
#
# * get data of specific vlan group
# GET /router/{switch_id}/{vlan_id}
#
# 2. set address data or routing data.
#
# * set data of no vlan
# POST /router/{switch_id}
#
# * set data of specific vlan group
# POST /router/{switch_id}/{vlan_id}
#
# case1: set address data.
#   parameter = {"address": "A.B.C.D/M"}
# case2-1: set static route.
#   parameter = {"destination": "A.B.C.D/M", "gateway": "E.F.G.H"}
# case2-2: set default route.
#   parameter = {"gateway": "E.F.G.H"}
#
# MPLSmod case3: set mapping for mpls routers:
#   parameter = {"prefix": "A.B.C.D/M", "port": "N"}
#
# 3. delete address data or routing data.
#
# * delete data of no vlan
# DELETE /router/{switch_id}

```

```

#
# * delete data of specific vlan group
# DELETE /router/{switch_id}/{vlan_id}
#
# case1: delete address data.
#   parameter = {"address_id": "<int>"} or {"address_id": "all"}
# case2: delete routing data.
#   parameter = {"route_id": "<int>"} or {"route_id": "all"}
#
#
# 4. MPLSmod: Identify the router as LER or LSR
#
# case1: LER.
#   parameter = {"router": "ler"}
# case2: LSR
#   parameter = {"router": "lsr"}

UINT16_MAX = 0xffff
UINT32_MAX = 0xffffffff
UINT64_MAX = 0xffffffffffffffff

ETHERNET = ethernet.ethernet.__name__
VLAN = vlan.vlan.__name__
IPV4 = ipv4.ipv4.__name__
ARP = arp.arp.__name__
ICMP = icmp.icmp.__name__
TCP = tcp.tcp.__name__
UDP = udp.udp.__name__
# MPLSmod
MPLS = mpls.mpls.__name__

MAX_SUSPENDPACKETS = 50 # Threshold of the packet suspends thread count.

ARP_REPLY_TIMER = 2 # sec
OFP_REPLY_TIMER = 1.0 # sec
CHK_ROUTING_TBL_INTERVAL = 1800 # sec

SWITCHID_PATTERN = dpid_lib.DPID_PATTERN + r'|all'
VLANID_PATTERN = r'[0-9]{1,4}|all'

VLANID_NONE = 0
VLANID_MIN = 2
VLANID_MAX = 4094

COOKIE_DEFAULT_ID = 0
COOKIE_SHIFT_VLANID = 32
COOKIE_SHIFT_ROUTEID = 16

DEFAULT_ROUTE = '0.0.0.0/0'
IDLE_TIMEOUT = 1800 # sec
DEFAULT_TTL = 64

```

```
REST_COMMAND_RESULT = 'command_result'
REST_RESULT = 'result'
REST_DETAILS = 'details'
REST_OK = 'success'
REST_NG = 'failure'
REST_ALL = 'all'
REST_SWITCHID = 'switch_id'
REST_VLANID = 'vlan_id'
REST_NW = 'internal_network'
REST_ADDRESSID = 'address_id'
REST_ADDRESS = 'address'
REST_ROUTEID = 'route_id'
REST_ROUTE = 'route'
REST_DESTINATION = 'destination'
REST_GATEWAY = 'gateway'
# MPLSmod: mpls REST parameters
REST_PREFIX = 'prefix'
REST_PORT = 'port'
REST_ROUTER = 'router'

PRIORITY_VLAN_SHIFT = 1000
PRIORITY_NETMASK_SHIFT = 32

PRIORITY_NORMAL = 0
PRIORITY_ARP_HANDLING = 1
PRIORITY_DEFAULT_ROUTING = 1
PRIORITY_MAC_LEARNING = 2
PRIORITY_STATIC_ROUTING = 2
PRIORITY_IMPLICIT_ROUTING = 3
PRIORITY_L2_SWITCHING = 4
PRIORITY_IP_HANDLING = 5
# MPLSmod: mpls priority values
PRIORITY_MPLS_PREFIX = 1
PRIORITY_PUSH_MPLS = 2
PRIORITY_POP_MPLS = 2
PRIORITY_SWAP_MPLS = 2

# MPLSmod: Tags for actions
MPLS_PUSH_LABEL = 1
MPLS_POP_LABEL = 2
MPLS_SWAP_LABEL = 3

# MPLSmod: Router types
ROUTER_TYPE_LER = 'ler'
ROUTER_TYPE_LSR = 'lsr'

# MPLSmod: Hardcoded LSR Datapath ID
LSR_DPID = '0000000000000004'

PRIORITY_TYPE_ROUTE = 'priority_route'
```

```

def get_priority(priority_type, vid=0, route=None):
    log_msg = None
    priority = priority_type

    if priority_type == PRIORITY_TYPE_ROUTE:
        assert route is not None
        if route.dst_ip:
            priority_type = PRIORITY_STATIC_ROUTING
            priority = priority_type + route.netmask
            log_msg = 'static routing'
        else:
            priority_type = PRIORITY_DEFAULT_ROUTING
            priority = priority_type
            log_msg = 'default routing'

    if vid or priority_type == PRIORITY_IP_HANDLING:
        priority += PRIORITY_VLAN_SHIFT

    if priority_type > PRIORITY_STATIC_ROUTING:
        priority += PRIORITY_NETMASK_SHIFT

    if log_msg is None:
        return priority
    else:
        return priority, log_msg

def get_priority_type(priority, vid):
    if vid:
        priority -= PRIORITY_VLAN_SHIFT
    return priority

class NotFoundError(RyuException):
    message = 'Router SW is not connected. : switch_id=%(switch_id)s'

class CommandFailure(RyuException):
    pass

class RestRouterAPI(app_manager.RyuApp):

    OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION,
                    ofproto_v1_2.OFP_VERSION,
                    ofproto_v1_3.OFP_VERSION]

    _CONTEXTS = {'dpset': dpset.DPSet,
                 'wsgi': WSGIApplication}

    def __init__(self, *args, **kwargs):

```

```

super(RestRouterAPI, self).__init__(*args, **kwargs)

# MPLSmod: Object to allow all routers access mpls data
self.mpls_data = MplsData()
# logger configure
RouterController.set_logger(self.logger)

wsgi = kwargs['wsgi']
self.waiters = {}
self.data = {'waiters': self.waiters}

mapper = wsgi.mapper
wsgi.registry['RouterController'] = self.data
requirements = {'switch_id': SWITCHID_PATTERN,
                'vlan_id': VLANID_PATTERN}

# For no vlan data
path = '/router/{switch_id}'
mapper.connect('router', path, controller=RouterController,
               requirements=requirements,
               action='get_data',
               conditions=dict(method=['GET']))
mapper.connect('router', path, controller=RouterController,
               requirements=requirements,
               action='set_data',
               conditions=dict(method=['POST']))
mapper.connect('router', path, controller=RouterController,
               requirements=requirements,
               action='delete_data',
               conditions=dict(method=['DELETE']))

# For vlan data
path = '/router/{switch_id}/{vlan_id}'
mapper.connect('router', path, controller=RouterController,
               requirements=requirements,
               action='get_vlan_data',
               conditions=dict(method=['GET']))
mapper.connect('router', path, controller=RouterController,
               requirements=requirements,
               action='set_vlan_data',
               conditions=dict(method=['POST']))
mapper.connect('router', path, controller=RouterController,
               requirements=requirements,
               action='delete_vlan_data',
               conditions=dict(method=['DELETE']))

@set_ev_cls(dpset.EventDP, dpset.DPSET_EV_DISPATCHER)
def datapath_handler(self, ev):
    if ev.enter:
        # MPLSmod: added mpls_data parameter
        RouterController.register_router(ev.dp, self.mpls_data)
    else:
        RouterController.unregister_router(ev.dp)

```

```

@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def packet_in_handler(self, ev):
    RouterController.packet_in_handler(ev.msg)

def _stats_reply_handler(self, ev):
    msg = ev.msg
    dp = msg.datapath

    if (dp.id not in self.waiters
        or msg.xid not in self.waiters[dp.id]):
        return
    event, msgs = self.waiters[dp.id][msg.xid]
    msgs.append(msg)

    if ofproto_v1_3.OFP_VERSION == dp.ofproto.OFP_VERSION:
        more = dp.ofproto.OFPMPF_REPLY_MORE
    else:
        more = dp.ofproto.OFPSF_REPLY_MORE
    if msg.flags & more:
        return
    del self.waiters[dp.id][msg.xid]
    event.set()

# for OpenFlow version1.0
@set_ev_cls(ofp_event.EventOFPPFlowStatsReply, MAIN_DISPATCHER)
def stats_reply_handler_v1_0(self, ev):
    self._stats_reply_handler(ev)

# for OpenFlow version1.2/1.3
@set_ev_cls(ofp_event.EventOFPPStatsReply, MAIN_DISPATCHER)
def stats_reply_handler_v1_2(self, ev):
    self._stats_reply_handler(ev)

# TODO: Update routing table when port status is changed.

# REST command template
def rest_command(func):
    def _rest_command(*args, **kwargs):
        try:
            msg = func(*args, **kwargs)
            return Response(content_type='application/json',
                           body=json.dumps(msg))

        except SyntaxError as e:
            status = 400
            details = e.msg

        except (ValueError, NameError) as e:
            status = 400
            details = e.message

```

```

    except NotFoundError as msg:
        status = 404
        details = str(msg)

    msg = {REST_RESULT: REST_NG,
          REST_DETAILS: details}
    return Response(status=status, body=json.dumps(msg))

return _rest_command

```

```

class RouterController(ControllerBase):

    _ROUTER_LIST = {}
    _LOGGER = None

    def __init__(self, req, link, data, **config):
        super(RouterController, self).__init__(req, link, data, **config)
        self.waiters = data['waiters']

    @classmethod
    def set_logger(cls, logger):
        cls._LOGGER = logger
        cls._LOGGER.propagate = False
        hdlr = logging.StreamHandler()
        fmt_str = '[RT][%(levelname)s] switch_id=%(sw_id)s: %(message)s'
        hdlr.setFormatter(logging.Formatter(fmt_str))
        cls._LOGGER.addHandler(hdlr)

    # MPLSmod: adding mpls_data parameter
    @classmethod
    def register_router(cls, dp, mpls_data):
        dpid = {'sw_id': dpid_lib.dpid_to_str(dp.id)}
        try:
            router = Router(dp, cls._LOGGER, mpls_data)
        except OFPUnknownVersion as message:
            cls._LOGGER.error(str(message), extra=dpid)
            return
        cls._ROUTER_LIST.setdefault(dp.id, router)
        cls._LOGGER.info('Join as router.', extra=dpid)

    @classmethod
    def unregister_router(cls, dp):
        if dp.id in cls._ROUTER_LIST:
            cls._ROUTER_LIST[dp.id].delete()
            del cls._ROUTER_LIST[dp.id]

        dpid = {'sw_id': dpid_lib.dpid_to_str(dp.id)}
        cls._LOGGER.info('Leave router.', extra=dpid)

    @classmethod

```



```

def packet_in_handler(cls, msg):
    dp_id = msg.datapath.id
    if dp_id in cls._ROUTER_LIST:
        router = cls._ROUTER_LIST[dp_id]
        router.packet_in_handler(msg)

# GET /router/{switch_id}
@rest_command
def get_data(self, req, switch_id, **kwargs):
    return self._access_router(switch_id, VLANID_NONE,
                               'get_data', req.body)

# GET /router/{switch_id}/{vlan_id}
@rest_command
def get_vlan_data(self, req, switch_id, vlan_id, **kwargs):
    return self._access_router(switch_id, vlan_id,
                               'get_data', req.body)

# POST /router/{switch_id}
@rest_command
def set_data(self, req, switch_id, **kwargs):
    return self._access_router(switch_id, VLANID_NONE,
                               'set_data', req.body)

# POST /router/{switch_id}/{vlan_id}
@rest_command
def set_vlan_data(self, req, switch_id, vlan_id, **kwargs):
    return self._access_router(switch_id, vlan_id,
                               'set_data', req.body)

# DELETE /router/{switch_id}
@rest_command
def delete_data(self, req, switch_id, **kwargs):
    return self._access_router(switch_id, VLANID_NONE,
                               'delete_data', req.body)

# DELETE /router/{switch_id}/{vlan_id}
@rest_command
def delete_vlan_data(self, req, switch_id, vlan_id, **kwargs):
    return self._access_router(switch_id, vlan_id,
                               'delete_data', req.body)

def _access_router(self, switch_id, vlan_id, func, rest_param):
    rest_message = []
    routers = self._get_router(switch_id)
    param = json.loads(rest_param) if rest_param else {}
    for router in routers.values():
        function = getattr(router, func)
        data = function(vlan_id, param, self.waiters)
        rest_message.append(data)

    return rest_message

```

```

def _get_router(self, switch_id):
    routers = {}

    if switch_id == REST_ALL:
        routers = self._ROUTER_LIST
    else:
        sw_id = dpid_lib.str_to_dpid(switch_id)
        if sw_id in self._ROUTER_LIST:
            routers = {sw_id: self._ROUTER_LIST[sw_id]}

    if routers:
        return routers
    else:
        raise NotFoundError(switch_id=switch_id)

class Router(dict):
    def __init__(self, dp, logger, mpls_data):
        super(Router, self).__init__()
        self.dp = dp
        self.dpid_str = dpid_lib.dpid_to_str(dp.id)
        self.sw_id = {'sw_id': self.dpid_str}
        self.logger = logger

        self.port_data = PortData(dp.ports)

        # MPLSmod: object to store all MPLS data (Modified constructor)
        self.mpls_data = mpls_data
        ofctl = OfCtl.factory(dp, logger)
        cookie = COOKIE_DEFAULT_ID

        # Set SW config: TTL error packet in (for OFPv1.2/1.3)
        ofctl.set_sw_config_for_ttl()

        # Set flow: ARP handling (packet in)
        priority = get_priority(PRIORITY_ARP_HANDLING)
        ofctl.set_packetin_flow(cookie, priority, dl_type=ether.ETH_TYPE_ARP)
        self.logger.info('Set ARP handling (packet in) flow [cookie=0x%x]',
                        cookie, extra=self.sw_id)

        # Set flow: L2 switching (normal)
        priority = get_priority(PRIORITY_NORMAL)
        ofctl.set_normal_flow(cookie, priority)
        self.logger.info('Set L2 switching (normal) flow [cookie=0x%x]',
                        cookie, extra=self.sw_id)

        # Set VlanRouter for vid=None.
        vlan_router = VlanRouter(VLANID_NONE, dp, self.port_data, logger,
                                self.mpls_data)
        self[VLANID_NONE] = vlan_router

```

```

# Start cyclic routing table check.
self.thread = hub.spawn(self._cyclic_update_routing_tbl)
self.logger.info('Start cyclic routing table update.',
                 extra=self.sw_id)

def delete(self):
    hub.kill(self.thread)
    self.thread.wait()
    self.logger.info('Stop cyclic routing table update.',
                    extra=self.sw_id)

def _get_vlan_router(self, vlan_id):
    vlan_routers = []

    if vlan_id == REST_ALL:
        vlan_routers = self.values()
    else:
        vlan_id = int(vlan_id)
        if (vlan_id != VLANID_NONE and
            (vlan_id < VLANID_MIN or VLANID_MAX < vlan_id)):
            msg = 'Invalid {vlan_id} value. Set [%d-%d]'
            raise ValueError(msg % (VLANID_MIN, VLANID_MAX))
        elif vlan_id in self:
            vlan_routers = [self[vlan_id]]

    return vlan_routers

def _add_vlan_router(self, vlan_id):
    vlan_id = int(vlan_id)
    if vlan_id not in self:
        vlan_router = VlanRouter(vlan_id, self.dp, self.port_data,
                                self.logger, self.mpls_data)
        self[vlan_id] = vlan_router
    return self[vlan_id]

def _del_vlan_router(self, vlan_id, waiters):
    # Remove unnecessary VlanRouter.
    if vlan_id == VLANID_NONE:
        return

    vlan_router = self[vlan_id]
    if (len(vlan_router.address_data) == 0
        and len(vlan_router.routing_tbl) == 0):
        vlan_router.delete(waiters)
        del self[vlan_id]

def get_data(self, vlan_id, dummy1, dummy2):
    vlan_routers = self._get_vlan_router(vlan_id)
    if vlan_routers:
        msgs = [vlan_router.get_data() for vlan_router in vlan_routers]
    else:
        msgs = [{REST_VLANID: vlan_id}]

```

```

    return {REST_SWITCHID: self.dpid_str,
            REST_NW: msgs}

def set_data(self, vlan_id, param, waiters):
    vlan_routers = self._get_vlan_router(vlan_id)
    if not vlan_routers:
        vlan_routers = [self._add_vlan_router(vlan_id)]

    msgs = []
    for vlan_router in vlan_routers:
        try:
            msg = vlan_router.set_data(param)
            msgs.append(msg)
            if msg[REST_RESULT] == REST_NG:
                # Data setting is failure.
                self._del_vlan_router(vlan_router.vlan_id, waiters)
        except ValueError as err_msg:
            # Data setting is failure.
            self._del_vlan_router(vlan_router.vlan_id, waiters)
            raise err_msg

    return {REST_SWITCHID: self.dpid_str,
            REST_COMMAND_RESULT: msgs}

def delete_data(self, vlan_id, param, waiters):
    msgs = []
    vlan_routers = self._get_vlan_router(vlan_id)
    if vlan_routers:
        for vlan_router in vlan_routers:
            msg = vlan_router.delete_data(param, waiters)
            if msg:
                msgs.append(msg)
                # Check unnecessary VlanRouter.
                self._del_vlan_router(vlan_router.vlan_id, waiters)
    if not msgs:
        msgs = [{REST_RESULT: REST_NG,
                 REST_DETAILS: 'Data is nothing.'}]

    return {REST_SWITCHID: self.dpid_str,
            REST_COMMAND_RESULT: msgs}

def packet_in_handler(self, msg):
    pkt = packet.Packet(msg.data)
    # TODO: Packet library convert to string
    # self.logger.debug('Packet in = %s', str(pkt), self.sw_id)
    header_list = dict((p.protocol_name, p)
                       for p in pkt.protocols if type(p) != str)

    # print("****HEADER LIST**** %s", str(header_list))
    if header_list:
        # Check vlan-tag

```

```

vlan_id = VLANID_NONE
if VLAN in header_list:
    vlan_id = header_list[VLAN].vid

    # Event dispatch
if vlan_id in self:
    self.logger.info('Launching Vlan_router PacketIn handler',
        extra=self.sw_id)
    self[vlan_id].packet_in_handler(msg, header_list)
else:
    self.logger.info('Drop unknown vlan packet. [vlan_id=%d]',
        vlan_id, extra=self.sw_id)

def _cyclic_update_routing_tbl(self):
    while True:
        # send ARP to all gateways.
        for vlan_router in self.values():
            vlan_router.send_arp_all_gw()
            hub.sleep(1)

        hub.sleep(CHK_ROUTING_TBL_INTERVAL)

class VlanRouter(object):
    def __init__(self, vlan_id, dp, port_data, logger, mpls_data):
        super(VlanRouter, self).__init__()
        self.vlan_id = vlan_id
        self.dp = dp
        self.sw_id = {'sw_id': dpid_lib.dpid_to_str(dp.id)}
        self.logger = logger

        self.port_data = port_data
        self.address_data = AddressData()
        self.routing_tbl = RoutingTable()
        self.packet_buffer = SuspendPacketList(self.send_icmp_unreach_error)
        self.ofctl = OfCtl.factory(dp, logger)

        # MPLSmod: attribute relating prefixes to ports
        self.prefix_data = PrefixData()
        # MPLSmod: objects containing mpls labels
        self.mpls_label = MplsLabel()
        self.mpls_data = mpls_data
        self.dpid = dpid_lib.dpid_to_str(dp.id)
        self.mpls_data.setdefault(self.dpid, {})
        # MPLSmod: Router type. LER by default
        self.router_type = ROUTER_TYPE_LER
        #MPLSmod: Object to store info abut hosts, by IP
        self.hosts = HostDict()
        # OLDCODE: Set flow: default route (drop)
        # self._set_defaultroute_drop()

        # Mplsmod:

```

```

# Set flow: MPLS packets are sent to the controller
self.ofctl.set_packetin_flow(0x24, PRIORITY_MPLS_PREFIX,
                             dl_type=ether.ETH_TYPE_MPLS)

def delete(self, waiters):
    # Delete flow.
    msgs = self.ofctl.get_all_flow(waiters)
    for msg in msgs:
        for stats in msg.body:
            vlan_id = VlanRouter._cookie_to_id(REST_VLANID, stats.cookie)
            if vlan_id == self.vlan_id:
                self.ofctl.delete_flow(stats)

    assert len(self.packet_buffer) == 0

@staticmethod
def _cookie_to_id(id_type, cookie):
    if id_type == REST_VLANID:
        rest_id = cookie >> COOKIE_SHIFT_VLANID
    elif id_type == REST_ADDRESSID:
        rest_id = cookie & UINT32_MAX
    else:
        assert id_type == REST_ROUTEID
        rest_id = (cookie & UINT32_MAX) >> COOKIE_SHIFT_ROUTEID

    return rest_id

def _id_to_cookie(self, id_type, rest_id):
    vid = self.vlan_id << COOKIE_SHIFT_VLANID

    if id_type == REST_VLANID:
        cookie = rest_id << COOKIE_SHIFT_VLANID
    elif id_type == REST_ADDRESSID:
        cookie = vid + rest_id
    else:
        assert id_type == REST_ROUTEID
        cookie = vid + (rest_id << COOKIE_SHIFT_ROUTEID)

    return cookie

def _get_priority(self, priority_type, route=None):
    return get_priority(priority_type, vid=self.vlan_id, route=route)

def _response(self, msg):
    if msg and self.vlan_id:
        msg.setdefault(REST_VLANID, self.vlan_id)
    return msg

def get_data(self):
    address_data = self._get_address_data()
    routing_data = self._get_routing_data()
    # MPLSmod: Port data for the MPLS network

```

```

    #prefix_data = self._get_prefix_data()

    data = {}
    if address_data[REST_ADDRESS]:
        data.update(address_data)
    if routing_data[REST_ROUTE]:
        data.update(routing_data)

    # MPLSmod: Update data
    #if prefix_data[REST_PREFIX]:
    #    data.update(prefix_data)

    return self._response(data)

#MPLSmod: get prefix data method
def _get_prefix_data(self):
    # Yet to implement
    pass

def _get_address_data(self):
    address_data = []
    for value in self.address_data.values():
        default_gw = ip_addr_ntoa(value.default_gw)
        address = '%s/%d' % (default_gw, value.netmask)
        data = {REST_ADDRESSID: value.address_id,
                REST_ADDRESS: address}
        address_data.append(data)
    return {REST_ADDRESS: address_data}

def _get_routing_data(self):
    routing_data = []
    for key, value in self.routing_tbl.items():
        if value.gateway_mac is not None:
            gateway = ip_addr_ntoa(value.gateway_ip)
            data = {REST_ROUTEID: value.route_id,
                    REST_DESTINATION: key,
                    REST_GATEWAY: gateway}
            routing_data.append(data)
    return {REST_ROUTE: routing_data}

def set_data(self, data):
    details = None

    try:
        # Set address data
        if REST_ADDRESS in data:
            address = data[REST_ADDRESS]
            address_id = self._set_address_data(address)
            details = 'Add address [address_id=%d]' % address_id
        # Set routing data
        elif REST_GATEWAY in data:
            gateway = data[REST_GATEWAY]

```

```

        if REST_DESTINATION in data:
            destination = data[REST_DESTINATION]
        else:
            destination = DEFAULT_ROUTE
        route_id = self._set_routing_data(destination, gateway)
        details = 'Add route [route_id=%d]' % route_id
        # MPLSmod: set prefix-port mapping data
    elif REST_PREFIX in data:
        prefix = data[REST_PREFIX]
        port = data[REST_PORT]
        prefix_id = self._set_prefix_data(prefix, port)
        details = 'Add prefix to port [prefix_id=%d]' % prefix_id
        # MPLSmod: set router type
    elif REST_ROUTER in data:
        router = data[REST_ROUTER]
        self._set_router_type(router)
        details = 'Add router type: %s' % router

except CommandFailure as err_msg:
    msg = {REST_RESULT: REST_NG, REST_DETAILS: str(err_msg)}
    return self._response(msg)

if details is not None:
    msg = {REST_RESULT: REST_OK, REST_DETAILS: details}
    return self._response(msg)
else:
    raise ValueError('Invalid parameter.')

#MPLSmod: set router type
def _set_router_type(self, router):
    self.router_type = router

# MPLSmod: set port data method
def _set_prefix_data(self, prefix, port):
    cookie = 0x800
    prefix = self.prefix_data.add(prefix, port)
    # Set flow: IP packets aiming this prefix are sent to the controller
    priority = self._get_priority(PRIORITY_MPLS_PREFIX)
    self.ofctl.set_packetin_flow(cookie, priority,
                                dl_type=ether.ETH_TYPE_IP,
                                dst_ip=prefix.address,
                                dst_mask=prefix.netmask)

    return prefix.prefix_id

def _set_address_data(self, address):
    address = self.address_data.add(address)

    cookie = self._id_to_cookie(REST_ADDRESSID, address.address_id)

    # Set flow: host MAC learning (packet in)
    priority = self._get_priority(PRIORITY_MAC_LEARNING)
    self.ofctl.set_packetin_flow(cookie, priority,

```



```

        dl_type=ether.ETH_TYPE_IP,
        dl_vlan=self.vlan_id,
        dst_ip=address.nw_addr,
        dst_mask=address.netmask)
log_msg = 'Set host MAC learning (packet in) flow [cookie=0x%x]'
self.logger.info(log_msg, cookie, extra=self.sw_id)

# set Flow: IP handling(PacketIn)
priority = self._get_priority(PRIORITY_IP_HANDLING)
self.ofctl.set_packetin_flow(cookie, priority,
                             dl_type=ether.ETH_TYPE_IP,
                             dl_vlan=self.vlan_id,
                             dst_ip=address.default_gw)
self.logger.info('Set IP handling (packet in) flow [cookie=0x%x]',
                 cookie, extra=self.sw_id)

# Set flow: L2 switching (normal)
outport = self.ofctl.dp.ofproto.OFPP_NORMAL
priority = self._get_priority(PRIORITY_L2_SWITCHING)
self.ofctl.set_routing_flow(
    cookie, priority, outport, dl_vlan=self.vlan_id,
    nw_src=address.nw_addr, src_mask=address.netmask,
    nw_dst=address.nw_addr, dst_mask=address.netmask)
self.logger.info('Set L2 switching (normal) flow [cookie=0x%x]',
                 cookie, extra=self.sw_id)

# Send GARP
self.send_arp_request(address.default_gw, address.default_gw)

return address.address_id

def _set_routing_data(self, destination, gateway):
    err_msg = 'Invalid [%s] value.' % REST_GATEWAY
    dst_ip = ip_addr_aton(gateway, err_msg=err_msg)
    address = self.address_data.get_data(ip=dst_ip)
    if address is None:
        msg = 'Gateway=%s\'s address is not registered.' % gateway
        raise CommandFailure(msg=msg)
    elif dst_ip == address.default_gw:
        msg = 'Gateway=%s is used as default gateway of address_id=%d' \
            % (gateway, address.address_id)
        raise CommandFailure(msg=msg)
    else:
        src_ip = address.default_gw
        route = self.routing_tbl.add(destination, gateway)
        self._set_route_packetin(route)
        self.send_arp_request(src_ip, dst_ip)
        return route.route_id

def _set_defaultroute_drop(self):
    cookie = self._id_to_cookie(REST_VLANID, self.vlan_id)
    priority = self._get_priority(PRIORITY_DEFAULT_ROUTING)

```

```

outport = None # for drop
self.ofctl.set_routing_flow(cookie, priority, outport,
                             dl_vlan=self.vlan_id)
self.logger.info('Set default route (drop) flow [cookie=0x%x]',
                 cookie, extra=self.sw_id)

def _set_route_packetin(self, route):
    cookie = self._id_to_cookie(REST_ROUTEID, route.route_id)
    priority, log_msg = self._get_priority(PRIORITY_TYPE_ROUTE,
                                           route=route)
    self.ofctl.set_packetin_flow(cookie, priority,
                                  dl_type=ether.ETH_TYPE_IP,
                                  dl_vlan=self.vlan_id,
                                  dst_ip=route.dst_ip,
                                  dst_mask=route.netmask)
    self.logger.info('Set %s (packet in) flow [cookie=0x%x]', log_msg,
                    cookie, extra=self.sw_id)

def delete_data(self, data, waiters):
    if REST_ROUTEID in data:
        route_id = data[REST_ROUTEID]
        msg = self._delete_routing_data(route_id, waiters)
    elif REST_ADDRESSID in data:
        address_id = data[REST_ADDRESSID]
        msg = self._delete_address_data(address_id, waiters)
    else:
        raise ValueError('Invalid parameter.')

    return self._response(msg)

def _delete_address_data(self, address_id, waiters):
    if address_id != REST_ALL:
        try:
            address_id = int(address_id)
        except ValueError as e:
            err_msg = 'Invalid [%s] value. %s'
            raise ValueError(err_msg % (REST_ADDRESSID, e.message))

    skip_ids = self._chk_addr_relation_route(address_id)

    # Get all flow.
    delete_list = []
    msgs = self.ofctl.get_all_flow(waiters)
    max_id = UINT16_MAX
    for msg in msgs:
        for stats in msg.body:
            vlan_id = VlanRouter._cookie_to_id(REST_VLANID, stats.cookie)
            if vlan_id != self.vlan_id:
                continue
            addr_id = VlanRouter._cookie_to_id(REST_ADDRESSID,
                                              stats.cookie)
            if addr_id in skip_ids:

```

```

        continue
    elif address_id == REST_ALL:
        if addr_id <= COOKIE_DEFAULT_ID or max_id < addr_id:
            continue
    elif address_id != addr_id:
        continue
    delete_list.append(stats)

delete_ids = []
for flow_stats in delete_list:
    # Delete flow
    self.ofctl.delete_flow(flow_stats)
    address_id = VlanRouter._cookie_to_id(REST_ADDRESSID,
                                          flow_stats.cookie)

    del_address = self.address_data.get_data(addr_id=address_id)
    if del_address is not None:
        # Clean up suspend packet threads.
        self.packet_buffer.delete(del_addr=del_address)

        # Delete data.
        self.address_data.delete(address_id)
        if address_id not in delete_ids:
            delete_ids.append(address_id)

msg = {}
if delete_ids:
    delete_ids = ','.join(str(addr_id) for addr_id in delete_ids)
    details = 'Delete address [address_id=%s]' % delete_ids
    msg = {REST_RESULT: REST_OK, REST_DETAILS: details}

if skip_ids:
    skip_ids = ','.join(str(addr_id) for addr_id in skip_ids)
    details = 'Skip delete (related route exist) [address_id=%s]' \
             % skip_ids
    if msg:
        msg[REST_DETAILS] += ', %s' % details
    else:
        msg = {REST_RESULT: REST_NG, REST_DETAILS: details}

return msg

def _delete_routing_data(self, route_id, waiters):
    if route_id != REST_ALL:
        try:
            route_id = int(route_id)
        except ValueError as e:
            err_msg = 'Invalid [%s] value. %s'
            raise ValueError(err_msg % (REST_ROUTEID, e.message))

    # Get all flow.
    msg = self.ofctl.get_all_flow(waiters)

```

```

delete_list = []
for msg in msgs:
    for stats in msg.body:
        vlan_id = VlanRouter._cookie_to_id(REST_VLANID, stats.cookie)
        if vlan_id != self.vlan_id:
            continue
        rt_id = VlanRouter._cookie_to_id(REST_ROUTEID, stats.cookie)
        if route_id == REST_ALL:
            if rt_id == COOKIE_DEFAULT_ID:
                continue
            elif route_id != rt_id:
                continue
        delete_list.append(stats)

# Delete flow.
delete_ids = []
for flow_stats in delete_list:
    self.ofctl.delete_flow(flow_stats)
    route_id = VlanRouter._cookie_to_id(REST_ROUTEID,
                                        flow_stats.cookie)

    self.routing_tbl.delete(route_id)
    if route_id not in delete_ids:
        delete_ids.append(route_id)

# case: Default route deleted. -> set flow (drop)
route_type = get_priority_type(flow_stats.priority,
                               vid=self.vlan_id)
if route_type == PRIORITY_DEFAULT_ROUTING:
    self._set_defaultroute_drop()

msg = {}
if delete_ids:
    delete_ids = ','.join(str(route_id) for route_id in delete_ids)
    details = 'Delete route [route_id=%s]' % delete_ids
    msg = {REST_RESULT: REST_OK, REST_DETAILS: details}

return msg

def _chk_addr_relation_route(self, address_id):
    # Check exist of related routing data.
    relate_list = []
    gateways = self.routing_tbl.get_gateways()
    for gateway in gateways:
        address = self.address_data.get_data(ip=gateway)
        if address is not None:
            if (address_id == REST_ALL
                and address.address_id not in relate_list):
                relate_list.append(address.address_id)
            elif address.address_id == address_id:
                relate_list = [address_id]
            break

```

```

return relate_list

def packet_in_handler(self, msg, header_list):
    # Check invalid TTL (for OpenFlow V1.2/1.3)
    ofproto = self.dp.ofproto
    if ofproto.OFP_VERSION == ofproto_v1_2.OFP_VERSION or \
        ofproto.OFP_VERSION == ofproto_v1_3.OFP_VERSION:
        if msg.reason == ofproto.OFPR_INVALID_TTL:
            self._packetin_invalid_ttl(msg, header_list)
            return

    # Analyze event type.

    if ARP in header_list:
        self._packetin_arp(msg, header_list)
        return

    # MPLSmod: If packetin is an MPLS packet
    if MPLS in header_list:
        if self.router_type == ROUTER_TYPE_LER:
            # LER method
            self.logger.info('MPLS packet entering LER', extra=self.sw_id)
            self._packetin_from_mpls_network(msg, header_list)

            elif self.router_type == ROUTER_TYPE_LSR:
                # LSR method
                self.logger.info('MPLS packet entering LSR', extra=self.sw_id)
                self._packetin_to_lsr(msg, header_list)

        elif IPV4 in header_list:
            self.logger.info('IPV4 packet',
                extra=self.sw_id)
            rt_ports = self.address_data.get_default_gw()
            if header_list[IPV4].dst in rt_ports:
                self.logger.info('Packet in: dst in rt ports',
                    extra=self.sw_id)
                # Packet to router's port.
                if ICMP in header_list:
                    if header_list[ICMP].type == icmp.ICMP_ECHO_REQUEST:
                        self._packetin_icmp_req(msg, header_list)
                        return
                    elif TCP in header_list or UDP in header_list:
                        self._packetin_tcp_udp(msg, header_list)
                        return
                else:
                    # OLDCODE: Packet to internal host or gateway router.
                    # self._packetin_to_node(msg, header_list)

                    # MPLSmod: Packets to be routed into the MPLS network
                    self._packetin_to_mpls_network(msg, header_list)
                    return

    # MPLSmod: function for IP to MPLS handling

```

```

def _packetin_to_mpls_network(self, msg, header_list):
    self.logger.info('Launching packetin_to_mpls_network method.',
                     extra=self.sw_id)
    if len(self.packet_buffer) >= MAX_SUSPENDPACKETS:
        self.logger.info('Packet is dropped, MAX_SUSPENDPACKETS exceeded.',
                         extra=self.sw_id)
        return

    in_port = self.ofctl.get_packetin_inport(msg)
    dst_ip = header_list[IPV4].dst
    src_ip = header_list[IPV4].src

    for key in self.prefix_data:
        prefix = self.prefix_data[key]
        if prefix.compare(dst_ip):
            # Write flow & packet out
            priority = self._get_priority(PRIORITY_PUSH_MPLS)
            cookie = 0x810
            out_port = int(prefix.port)
            self.ofctl.set_mpls_flow(cookie, priority, self.mpls_label.value,
                                     in_port, out_port, MPLS_PUSH_LABEL, nw_dst=dst_ip,
                                     nw_src=src_ip)
            self.ofctl.send_mpls_packet_out(in_port,
                                             out_port, msg.data, self.mpls_label.value, MPLS_PUSH_LABEL)
            self.mpls_data.add(self.dpid, self.mpls_label.value, dst_ip)
            self.mpls_label.increase()
            break
        # Else drop

# MPLSmod: When packet enters LSR
def _packetin_to_lsr(self, msg, header_list):

    in_port = self.ofctl.get_packetin_inport(msg)
    dst_ip = header_list[IPV4].dst
    src_ip = header_list[IPV4].src
    label_in = self.ofctl.get_packetin_mplslabel(msg)
    self.logger.info('packet_in label: %s' % label_in,
                     extra=self.sw_id)

    address = 0
    orig_id = self.get_origin_id_lsr(in_port)
    self.logger.info('Origin ID: %s' % orig_id,
                     extra=self.sw_id)
    for label in self.mpls_data[orig_id]:
        self.logger.info('Labels pushed by origin: %s' % str(label),
                         extra=self.sw_id)
        if label == label_in:
            self.logger.info('Label match: %s' % str(label),
                             extra=self.sw_id)
            address = self.mpls_data[orig_id][label]
            break

```

```

for key in self.prefix_data:
    prefix = self.prefix_data[key]
    if address == 0:
        self.logger.info('No label match! Dropping packet...',
                          extra=self.sw_id)
        break
    if prefix.compare(address):
        self.logger.info('Label match! Swapping label...',
                          extra=self.sw_id)
        # Create a new label:
        self.mpls_label.increase()
        # Write flow & packet out
        priority = self._get_priority(PRIORITY_SWAP_MPLS)
        cookie = 0x820
        out_port = int(prefix.port)

        self.ofctl.set_mpls_flow(cookie, priority, self.mpls_label.value,
                                in_port, out_port, MPLS_SWAP_LABEL, nw_dst=dst_ip,
                                nw_src=src_ip, oldlabel=label_in)
        self.ofctl.send_mpls_packet_out(in_port,
                                       out_port, msg.data, self.mpls_label.value, MPLS_SWAP_LABEL)
        self.mpls_data.add(self.dpid, self.mpls_label, dst_ip)
        break

# MPLSmod: When packet enters a LER from the LSR
def _packetin_from_mpls_network(self, msg, header_list):
    # Extract data
    in_port = self.ofctl.get_packetin_inport(msg)
    dst_ip = header_list[IPV4].dst
    src_ip = header_list[IPV4].src
    srcip = ip_addr_ntoa(header_list[IPV4].src)
    dstip = ip_addr_ntoa(dst_ip)
    label_in = self.ofctl.get_packetin_mplslabel(msg)

    if dst_ip not in self.hosts:

        # Send ARP to learn the MAC address
        address = self.address_data.get_data(ip=dst_ip)
        if address is not None:
            log_msg = 'Receive IP packet from [%s] to an internal host [%s].'
            self.logger.info(log_msg, srcip, dstip, extra=self.sw_id)
            src_ip = address.default_gw

        if src_ip is not None:
            self.packet_buffer.add(in_port, header_list, msg.data)
            self.send_arp_request(src_ip, dst_ip, in_port=in_port)
            self.logger.info('Send ARP request (flood)', extra=self.sw_id)
    else:
        # Write flow & packet out
        priority = self._get_priority(PRIORITY_POP_MPLS)
        cookie = 0x830
        out_port = self.hosts[dst_ip].port

```

```

dl_src = self.port_data[out_port].mac
dl_dst = self.hosts[dst_ip].mac
self.ofctl.set_mpls_flow(cookie, priority, self.mpls_label.value,
                        in_port, out_port, MPLS_POP_LABEL, nw_dst=dst_ip,
                        dst_mac=dl_dst, src_mac=dl_src, oldlabel=label_in)
self.ofctl.send_mpls_packet_out(in_port, out_port, msg.data,
                                self.mpls_label.value, MPLS_POP_LABEL, dst_mac=dl_dst,
                                src_mac=dl_src)

# MPLSmod: Get origin LSR ID, quick and dirty method
def get_lsr_id(self, ler_id):
    if ler_id is not None:
        return LSR_DPID

# MPLSmod: Get origin ID, quick and dirty method
def get_origin_id_lsr(self, in_port):
    return "0000000000000000" + str(in_port)

def _packetin_arp(self, msg, header_list):
    src_addr = self.address_data.get_data(ip=header_list[ARP].src_ip)
    if src_addr is None:
        return

    # case: Receive ARP from the gateway
    # Update routing table.
    # case: Receive ARP from an internal host
    # Learning host MAC.
    gw_flg = self._update_routing_tbl(msg, header_list)
    if gw_flg is False:
        self._learning_host_mac(msg, header_list)

    # ARP packet handling.
    in_port = self.ofctl.get_packetin_inport(msg)
    src_ip = header_list[ARP].src_ip
    dst_ip = header_list[ARP].dst_ip
    srcip = ip_addr_ntoa(src_ip)
    dstip = ip_addr_ntoa(dst_ip)
    rt_ports = self.address_data.get_default_gw()

    if src_ip == dst_ip:
        # GARP -> packet forward (normal)
        output = self.ofctl.dp.ofproto.OFPP_NORMAL
        self.ofctl.send_packet_out(in_port, output, msg.data)

        self.logger.info('Receive GARP from [%s].', srcip,
                        extra=self.sw_id)
        self.logger.info('Send GARP (normal).', extra=self.sw_id)

    elif dst_ip not in rt_ports:
        dst_addr = self.address_data.get_data(ip=dst_ip)
        if (dst_addr is not None and
            src_addr.address_id == dst_addr.address_id):

```



```

        # ARP from internal host -> packet forward (normal)
        output = self.ofctl.dp.ofproto.OFPP_NORMAL
        self.ofctl.send_packet_out(in_port, output, msg.data)

        self.logger.info('Receive ARP from an internal host [%s].',
                        srcip, extra=self.sw_id)
        self.logger.info('Send ARP (normal)', extra=self.sw_id)
    else:
        if header_list[ARP].opcode == arp.ARP_REQUEST:
            # ARP request to router port -> send ARP reply
            src_mac = header_list[ARP].src_mac
            dst_mac = self.port_data[in_port].mac
            arp_target_mac = dst_mac
            output = in_port
            in_port = self.ofctl.dp.ofproto.OFPP_CONTROLLER

            self.ofctl.send_arp(arp.ARP_REPLY, self.vlan_id,
                               dst_mac, src_mac, dst_ip, src_ip,
                               arp_target_mac, in_port, output)

            log_msg = 'Receive ARP request from [%s] to router port [%s].'
            self.logger.info(log_msg, srcip, dstip, extra=self.sw_id)
            self.logger.info('Send ARP reply to [%s]', srcip,
                             extra=self.sw_id)

        elif header_list[ARP].opcode == arp.ARP_REPLY:
            # ARP reply to router port -> suspend packets forward
            log_msg = 'Receive ARP reply from [%s] to router port [%s].'
            self.logger.info(log_msg, srcip, dstip, extra=self.sw_id)

            packet_list = self.packet_buffer.get_data(src_ip)
            if packet_list:
                # stop ARP reply wait thread.
                for suspend_packet in packet_list:
                    self.packet_buffer.delete(pkt=suspend_packet)

                # send suspend packet.
                output = self.ofctl.dp.ofproto.OFPP_TABLE
                for suspend_packet in packet_list:
                    self.ofctl.send_packet_out(suspend_packet.in_port,
                                                output,
                                                suspend_packet.data)
                    self.logger.info('Send suspend packet to [%s].',
                                     srcip, extra=self.sw_id)

    def _packetin_icmp_req(self, msg, header_list):
        self.logger.info('Launching _packetin_icmp_req method',
                        extra=self.sw_id)
        # Send ICMP echo reply.
        in_port = self.ofctl.get_packetin_inport(msg)
        self.ofctl.send_icmp(in_port, header_list, self.vlan_id,
                             icmp.ICMP_ECHO_REPLY,

```

```

        icmp.ICMP_ECHO_REPLY_CODE,
        icmp_data=header_list[ICMP].data)

srcip = ip_addr_ntoa(header_list[IPV4].src)
dstip = ip_addr_ntoa(header_list[IPV4].dst)
log_msg = 'Receive ICMP echo request from [%s] to router port [%s].'
self.logger.info(log_msg, srcip, dstip, extra=self.sw_id)
self.logger.info('Send ICMP echo reply to [%s].', srcip,
                 extra=self.sw_id)

def _packetin_tcp_udp(self, msg, header_list):
    self.logger.info('Launching _packetin_tcp_udp method',
                    extra=self.sw_id)
    # Send ICMP port unreachable error.
    in_port = self.ofctl.get_packetin_inport(msg)
    self.ofctl.send_icmp(in_port, header_list, self.vlan_id,
                        icmp.ICMP_DEST_UNREACH,
                        icmp.ICMP_PORT_UNREACH_CODE,
                        msg_data=msg.data)

    srcip = ip_addr_ntoa(header_list[IPV4].src)
    dstip = ip_addr_ntoa(header_list[IPV4].dst)
    self.logger.info('Receive TCP/UDP from [%s] to router port [%s].',
                    srcip, dstip, extra=self.sw_id)
    self.logger.info('Send ICMP destination unreachable to [%s].', srcip,
                    extra=self.sw_id)

def _packetin_to_node(self, msg, header_list):
    self.logger.info('Launching _packetin_to_node method',
                    extra=self.sw_id)
    if len(self.packet_buffer) >= MAX_SUSPENDPACKETS:
        self.logger.info('Packet is dropped, MAX_SUSPENDPACKETS exceeded.',
                        extra=self.sw_id)

    return

    # Send ARP request to get node MAC address.
    in_port = self.ofctl.get_packetin_inport(msg)
    src_ip = None
    dst_ip = header_list[IPV4].dst
    srcip = ip_addr_ntoa(header_list[IPV4].src)
    dstip = ip_addr_ntoa(dst_ip)

    address = self.address_data.get_data(ip=dst_ip)
    if address is not None:
        log_msg = 'Receive IP packet from [%s] to an internal host [%s].'
        self.logger.info(log_msg, srcip, dstip, extra=self.sw_id)
        src_ip = address.default_gw
    else:
        route = self.routing_tbl.get_data(dst_ip=dst_ip)
        if route is not None:
            log_msg = 'Receive IP packet from [%s] to [%s].'
            self.logger.info(log_msg, srcip, dstip, extra=self.sw_id)
            gw_address = self.address_data.get_data(ip=route.gateway_ip)

```

```

        if gw_address is not None:
            src_ip = gw_address.default_gw
            dst_ip = route.gateway_ip0

    if src_ip is not None:
        self.packet_buffer.add(in_port, header_list, msg.data)
        self.send_arp_request(src_ip, dst_ip, in_port=in_port)
        self.logger.info('Send ARP request (flood)', extra=self.sw_id)

def _packetin_invalid_ttl(self, msg, header_list):
    # Send ICMP TTL error.
    srcip = ip_addr_ntoa(header_list[IPV4].src)
    self.logger.info('Receive invalid ttl packet from [%s].', srcip,
                    extra=self.sw_id)

    in_port = self.ofctl.get_packetin_inport(msg)
    src_ip = self._get_send_port_ip(header_list)
    if src_ip is not None:
        self.ofctl.send_icmp(in_port, header_list, self.vlan_id,
                             icmp.ICMP_TIME_EXCEEDED,
                             icmp.ICMP_TTL_EXPIRED_CODE,
                             msg_data=msg.data, src_ip=src_ip)
        self.logger.info('Send ICMP time exceeded to [%s].', srcip,
                        extra=self.sw_id)

def send_arp_all_gw(self):
    gateways = self.routing_tbl.get_gateways()
    for gateway in gateways:
        address = self.address_data.get_data(ip=gateway)
        self.send_arp_request(address.default_gw, gateway)

def send_arp_request(self, src_ip, dst_ip, in_port=None):
    # Send ARP request from all ports.
    for send_port in self.port_data.values():
        print "SENDING"
        if in_port is None or in_port != send_port.port_no:
            src_mac = send_port.mac
            dst_mac = mac_lib.BROADCAST_STR
            arp_target_mac = mac_lib.DONTCARE_STR
            inport = self.ofctl.dp.ofproto.OFPP_CONTROLLER
            output = send_port.port_no
            self.ofctl.send_arp(arp.ARP_REQUEST, self.vlan_id,
                               src_mac, dst_mac, src_ip, dst_ip,
                               arp_target_mac, inport, output)

def send_icmp_unreach_error(self, packet_buffer):
    # Send ICMP host unreachable error.
    self.logger.info('ARP reply wait timer was timed out.',
                    extra=self.sw_id)
    src_ip = self._get_send_port_ip(packet_buffer.header_list)
    if src_ip is not None:
        self.ofctl.send_icmp(packet_buffer.in_port,

```

```

        packet_buffer.header_list,
        self.vlan_id,
        icmp.ICMP_DEST_UNREACH,
        icmp.ICMP_HOST_UNREACH_CODE,
        msg_data=packet_buffer.data,
        src_ip=src_ip)

    dstip = ip_addr_ntoa(packet_buffer.dst_ip)
    self.logger.info('Send ICMP destination unreachable to [%s].',
                    dstip, extra=self.sw_id)

def _update_routing_tbl(self, msg, header_list):
    # Set flow: routing to gateway.
    out_port = self.ofctl.get_packetin_inport(msg)
    src_mac = header_list[ARP].src_mac
    dst_mac = self.port_data[out_port].mac
    src_ip = header_list[ARP].src_ip

    gateway_flg = False
    for key, value in self.routing_tbl.items():
        if value.gateway_ip == src_ip:
            gateway_flg = True
            if value.gateway_mac == src_mac:
                continue
            self.routing_tbl[key].gateway_mac = src_mac

    cookie = self._id_to_cookie(REST_ROUTEID, value.route_id)
    priority, log_msg = self._get_priority(PRIORITY_TYPE_ROUTE,
                                           route=value)
    self.ofctl.set_routing_flow(cookie, priority, out_port,
                                dl_vlan=self.vlan_id,
                                src_mac=dst_mac,
                                dst_mac=src_mac,
                                nw_dst=value.dst_ip,
                                dst_mask=value.netmask,
                                dec_ttl=True)
    self.logger.info('Set %s flow [cookie=0x%x]', log_msg, cookie,
                    extra=self.sw_id)

    return gateway_flg

def _learning_host_mac(self, msg, header_list):
    # Set flow: routing to internal Host.
    out_port = self.ofctl.get_packetin_inport(msg)
    src_mac = header_list[ARP].src_mac
    dst_mac = self.port_data[out_port].mac
    src_ip = header_list[ARP].src_ip

    # MPLSmod: store values
    self.hosts.add(src_ip, out_port, src_mac)

    gateways = self.routing_tbl.get_gateways()
    if src_ip not in gateways:

```

```

address = self.address_data.get_data(ip=src_ip)
if address is not None:
    cookie = self._id_to_cookie(REST_ADDRESSID, address.address_id)
    priority = self._get_priority(PRIORITY_IMPLICIT_ROUTING)

    self.ofctl.set_routing_flow(cookie, priority,
                                out_port, dl_vlan=self.vlan_id,
                                src_mac=dst_mac, dst_mac=src_mac,
                                nw_dst=src_ip,
                                idle_timeout=IDLE_TIMEOUT,
                                dec_ttl=True)
    self.logger.info('Set implicit routing flow [cookie=0x%x]',
                    cookie, extra=self.sw_id)

def _get_send_port_ip(self, header_list):
    try:
        src_mac = header_list[ETHERNET].src
        if IPV4 in header_list:
            src_ip = header_list[IPV4].src
        else:
            src_ip = header_list[ARP].src_ip
    except KeyError:
        self.logger.debug('Receive unsupported packet.', extra=self.sw_id)
        return None

address = self.address_data.get_data(ip=src_ip)
if address is not None:
    return address.default_gw
else:
    route = self.routing_tbl.get_data(gw_mac=src_mac)
    if route is not None:
        address = self.address_data.get_data(ip=route.gateway_ip)
        if address is not None:
            return address.default_gw

self.logger.debug('Receive packet from unknown IP[%s].',
                 ip_addr_ntoa(src_ip), extra=self.sw_id)
return None

class PortData(dict):
    def __init__(self, ports):
        super(PortData, self).__init__()
        for port in ports.values():
            data = Port(port.port_no, port.hw_addr)
            self[port.port_no] = data

class Port(object):
    def __init__(self, port_no, hw_addr):
        super(Port, self).__init__()

```

```

        self.port_no = port_no
        self.mac = hw_addr

# MPLSmod: classes to store host data
class HostDict(dict):

    def __init__(self):
        super(HostDict, self).__init__()

    def add(self, ip, port, mac):
        self[ip] = Host(ip, port, mac)

class Host(object):

    def __init__(self, ip, port, mac):
        self.ip = ip
        self.port = port
        self.mac = mac

# MPLSmod: class to store the prefix-port info
class PrefixData(dict):
    def __init__(self):
        super(PrefixData, self).__init__()
        self.prefix_id = 1
    # Does not check for overlaps yet

    def add(self, prefix, port):
        err_msg = 'Invalid [%s] value.' % REST_PREFIX
        nw_addr, mask, default_gw = nw_addr_aton(prefix, err_msg=err_msg)
        prefix = Prefix(nw_addr, mask, port, self.prefix_id)
        ip_str = ip_addr_ntoa(nw_addr)
        key = '%s/%d' % (ip_str, mask)
        self[key] = prefix
        self.prefix_id = self.prefix_id + 1
        return prefix

# MPLSmod: class to encapsulate the prefix-port relation
class Prefix(object):

    def __init__(self, address, netmask, port, prefix_id):
        self.prefix_id = prefix_id
        self.address = address
        self.netmask = netmask
        self.port = port

    def compare(self, ip):
        if ipv4_apply_mask(ip, self.netmask) == self.address:
            return True

```

```

        else:
            return False

# MPLSmod: class mapping IP addresses to labels
class MplsData(dict):

    def __init__(self):
        super(MplsData, self).__init__()

    def add(self, dpid, label_value, dst_ip):
        self[dpid][label_value] = dst_ip

# MPLSmod: class to encapsulate labels
class MplsLabel(object):

    def __init__(self, value=20):
        self.value = value

    def increase(self):
        self.value = self.value + 1

class AddressData(dict):
    def __init__(self):
        super(AddressData, self).__init__()
        self.address_id = 1

    def add(self, address):
        err_msg = 'Invalid [%s] value.' % REST_ADDRESS
        nw_addr, mask, default_gw = nw_addr_aton(address, err_msg=err_msg)

        # Check overlaps
        for other in self.values():
            other_mask = mask_ntob(other.netmask)
            add_mask = mask_ntob(mask, err_msg=err_msg)
            if (other.nw_addr == ipv4_apply_mask(default_gw, other.netmask) or
                nw_addr == ipv4_apply_mask(other.default_gw, mask,
                                           err_msg)):
                msg = 'Address overlaps [address_id=%d]' % other.address_id
                raise CommandFailure(msg=msg)

        address = Address(self.address_id, nw_addr, mask, default_gw)
        ip_str = ip_addr_ntoa(nw_addr)
        key = '%s/%d' % (ip_str, mask)
        self[key] = address

        self.address_id += 1
        self.address_id &= UINT32_MAX
        if self.address_id == COOKIE_DEFAULT_ID:
            self.address_id = 1

    return address

```

```

def delete(self, address_id):
    for key, value in self.items():
        if value.address_id == address_id:
            del self[key]
            return

def get_default_gw(self):
    return [address.default_gw for address in self.values()]

def get_data(self, addr_id=None, ip=None):
    for address in self.values():
        if addr_id is not None:
            if addr_id == address.address_id:
                return address
        else:
            assert ip is not None
            if ipv4_apply_mask(ip, address.netmask) == address.nw_addr:
                return address
    return None

class Address(object):
    def __init__(self, address_id, nw_addr, netmask, default_gw):
        super(Address, self).__init__()
        self.address_id = address_id
        self.nw_addr = nw_addr
        self.netmask = netmask
        self.default_gw = default_gw

    def __contains__(self, ip):
        return bool(ipv4_apply_mask(ip, self.netmask) == self.nw_addr)

class RoutingTable(dict):
    def __init__(self):
        super(RoutingTable, self).__init__()
        self.route_id = 1

    def add(self, dst_nw_addr, gateway_ip):
        err_msg = 'Invalid [%s] value.'

        if dst_nw_addr == DEFAULT_ROUTE:
            dst_ip = 0
            netmask = 0
        else:
            dst_ip, netmask, dummy = nw_addr_aton(
                dst_nw_addr, err_msg=err_msg % REST_DESTINATION)

        gateway_ip = ip_addr_aton(gateway_ip, err_msg=err_msg % REST_GATEWAY)

        # Check overlaps

```



```

overlap_route = None
if dst_nw_addr == DEFAULT_ROUTE:
    if DEFAULT_ROUTE in self:
        overlap_route = self[DEFAULT_ROUTE].route_id
elif dst_nw_addr in self:
    overlap_route = self[dst_nw_addr].route_id

if overlap_route is not None:
    msg = 'Destination overlaps [route_id=%d]' % overlap_route
    raise CommandFailure(msg=msg)

routing_data = Route(self.route_id, dst_ip, netmask, gateway_ip)
ip_str = ip_addr_ntoa(dst_ip)
key = '%s/%d' % (ip_str, netmask)
self[key] = routing_data

self.route_id += 1
self.route_id &= UINT32_MAX
if self.route_id == COOKIE_DEFAULT_ID:
    self.route_id = 1

return routing_data

def delete(self, route_id):
    for key, value in self.items():
        if value.route_id == route_id:
            del self[key]
    return

def get_gateways(self):
    return [routing_data.gateway_ip for routing_data in self.values()]

def get_data(self, gw_mac=None, dst_ip=None):
    if gw_mac is not None:
        for route in self.values():
            if gw_mac == route.gateway_mac:
                return route
        return None

    elif dst_ip is not None:
        get_route = None
        mask = 0
        for route in self.values():
            if ipv4_apply_mask(dst_ip, route.netmask) == route.dst_ip:
                # For longest match
                if mask < route.netmask:
                    get_route = route
                    mask = route.netmask

    if get_route is None:
        get_route = self.get(DEFAULT_ROUTE, None)
    return get_route

```

```
else:
    return None
```

```
class Route(object):
```

```
    def __init__(self, route_id, dst_ip, netmask, gateway_ip):
        super(Route, self).__init__()
        self.route_id = route_id
        self.dst_ip = dst_ip
        self.netmask = netmask
        self.gateway_ip = gateway_ip
        self.gateway_mac = None
```

```
class SuspendPacketList(list):
```

```
    def __init__(self, timeout_function):
        super(SuspendPacketList, self).__init__()
        self.timeout_function = timeout_function

    def add(self, in_port, header_list, data):
        suspend_pkt = SuspendPacket(in_port, header_list, data,
                                     self.wait_arp_reply_timer)
        self.append(suspend_pkt)

    def delete(self, pkt=None, del_addr=None):
        if pkt is not None:
            del_list = [pkt]
        else:
            assert del_addr is not None
            del_list = [pkt for pkt in self if pkt.dst_ip in del_addr]

        for pkt in del_list:
            self.remove(pkt)
            hub.kill(pkt.wait_thread)
            pkt.wait_thread.wait()

    def get_data(self, dst_ip):
        return [pkt for pkt in self if pkt.dst_ip == dst_ip]

    def wait_arp_reply_timer(self, suspend_pkt):
        hub.sleep(ARP_REPLY_TIMER)
        if suspend_pkt in self:
            self.timeout_function(suspend_pkt)
            self.delete(pkt=suspend_pkt)
```

```
class SuspendPacket(object):
```

```
    def __init__(self, in_port, header_list, data, timer):
        super(SuspendPacket, self).__init__()
        self.in_port = in_port
        self.dst_ip = header_list[IPV4].dst
        self.header_list = header_list
```

```

self.data = data
# Start ARP reply wait timer.
self.wait_thread = hub.spawn(timer, self)

```

```

class OfCtl(object):
    _OF_VERSIONS = {}

    @staticmethod
    def register_of_version(version):
        def _register_of_version(cls):
            OfCtl._OF_VERSIONS.setdefault(version, cls)
            return cls
        return _register_of_version

    @staticmethod
    def factory(dp, logger):
        of_version = dp.ofproto.OFP_VERSION
        if of_version in OfCtl._OF_VERSIONS:
            ofctl = OfCtl._OF_VERSIONS[of_version](dp, logger)
        else:
            raise OFPUnknownVersion(version=of_version)

        return ofctl

    def __init__(self, dp, logger):
        super(OfCtl, self).__init__()
        self.dp = dp
        self.sw_id = {'sw_id': dpid_lib.dpid_to_str(dp.id)}
        self.logger = logger

    def set_sw_config_for_ttl(self):
        # OpenFlow v1_2/1_3.
        pass

    def set_flow(self, cookie, priority, dl_type=0, dl_dst=0, dl_vlan=0,
                 nw_src=0, src_mask=32, nw_dst=0, dst_mask=32,
                 nw_proto=0, idle_timeout=0, actions=None):
        # Abstract method
        raise NotImplementedError()

    def send_arp(self, arp_opcode, vlan_id, src_mac, dst_mac,
                 src_ip, dst_ip, arp_target_mac, in_port, output):
        # Generate ARP packet
        if vlan_id != VLANID_NONE:
            ether_proto = ether.ETH_TYPE_8021Q
            pcp = 0
            cfi = 0
            vlan_ether = ether.ETH_TYPE_ARP
            v = vlan.vlan(pcp, cfi, vlan_id, vlan_ether)
        else:
            ether_proto = ether.ETH_TYPE_ARP

```

```

hwtype = 1
arp_proto = ether.ETH_TYPE_IP
hlen = 6
plen = 4

pkt = packet.Packet()
e = ethernet.ethernet(dst_mac, src_mac, ether_proto)
a = arp.arp(hwtype, arp_proto, hlen, plen, arp_opcode,
           src_mac, src_ip, arp_target_mac, dst_ip)
pkt.add_protocol(e)
if vlan_id != VLANID_NONE:
    pkt.add_protocol(v)
pkt.add_protocol(a)
pkt.serialize()

# Send packet out
self.send_packet_out(in_port, output, pkt.data, data_str=str(pkt))

def send_icmp(self, in_port, protocol_list, vlan_id, icmp_type,
             icmp_code, icmp_data=None, msg_data=None, src_ip=None):
    # Generate ICMP reply packet
    csum = 0
    offset = ethernet.ethernet._MIN_LEN

    if vlan_id != VLANID_NONE:
        ether_proto = ether.ETH_TYPE_8021Q
        pcp = 0
        cfi = 0
        vlan_ether = ether.ETH_TYPE_IP
        v = vlan.vlan(pcp, cfi, vlan_id, vlan_ether)
        offset += vlan.vlan._MIN_LEN
    else:
        ether_proto = ether.ETH_TYPE_IP

    eth = protocol_list[ETHERNET]
    e = ethernet.ethernet(eth.src, eth.dst, ether_proto)

    if icmp_data is None and msg_data is not None:
        ip_datagram = msg_data[offset:]
        if icmp_type == icmp.ICMP_DEST_UNREACH:
            icmp_data = icmp.dest_unreach(data_len=len(ip_datagram),
                                           data=ip_datagram)
        elif icmp_type == icmp.ICMP_TIME_EXCEEDED:
            icmp_data = icmp.TimeExceeded(data_len=len(ip_datagram),
                                           data=ip_datagram)

    ic = icmp.icmp(icmp_type, icmp_code, csum, data=icmp_data)

    ip = protocol_list[IPV4]
    if src_ip is None:
        src_ip = ip.dst
    ip_total_length = ip.header_length * 4 + ic._MIN_LEN

```

```

if ic.data is not None:
    ip_total_length += ic.data._MIN_LEN
    if ic.data.data is not None:
        ip_total_length += + len(ic.data.data)
i = ipv4.ipv4(ip.version, ip.header_length, ip.tos,
             ip_total_length, ip.identification, ip.flags,
             ip.offset, DEFAULT_TTL, inet.IPPROTO_ICMP, csum,
             src_ip, ip.src)

pkt = packet.Packet()
pkt.add_protocol(e)
if vlan_id != VLANID_NONE:
    pkt.add_protocol(v)
pkt.add_protocol(i)
pkt.add_protocol(ic)
pkt.serialize()

# Send packet out
self.send_packet_out(in_port, self.dp.ofproto.OFPP_IN_PORT,
                    pkt.data, data_str=str(pkt))

# MPLSmod: push label and send
def send_mpls_packet_out(self, in_port, out_port, data, label, action,
                        dst_mac=0, src_mac=0):

    parser = self.dp.ofproto_parser
    if action == MPLS_PUSH_LABEL:
        actions = [parser.OFPACTIONPushMpls(ethertype=34887),
                  parser.OFPACTIONSetField(mpls_label=label),
                  parser.OFPACTIONOutput(out_port)]
    elif action == MPLS_SWAP_LABEL:
        actions = [parser.OFPACTIONPopMpls(),
                  parser.OFPACTIONPushMpls(ethertype=34887),
                  parser.OFPACTIONSetField(mpls_label=label),
                  parser.OFPACTIONOutput(out_port)]
    elif action == MPLS_POP_LABEL:
        actions = [parser.OFPACTIONPopMpls(),
                  parser.OFPACTIONSetField(eth_src=src_mac),
                  parser.OFPACTIONSetField(eth_dst=dst_mac),
                  parser.OFPACTIONOutput(out_port)]
    self.dp.send_packet_out(buffer_id=UINT32_MAX, in_port=in_port,
                          actions=actions, data=data)

def send_packet_out(self, in_port, output, data, data_str=None):
    actions = [self.dp.ofproto_parser.OFPACTIONOutput(output, 0)]
    self.dp.send_packet_out(buffer_id=UINT32_MAX, in_port=in_port,
                          actions=actions, data=data)

# TODO: Packet library convert to string
# if data_str is None:
#     data_str = str(packet.Packet(data))
# self.logger.debug('Packet out = %s', data_str, extra=self.sw_id)

```

```

def set_normal_flow(self, cookie, priority):
    out_port = self.dp.ofproto.OFPP_NORMAL
    actions = [self.dp.ofproto_parser.OFPActionOutput(out_port, 0)]
    self.set_flow(cookie, priority, actions=actions)

def set_packetin_flow(self, cookie, priority, dl_type=0, dl_dst=0,
                      dl_vlan=0, dst_ip=0, dst_mask=32, nw_proto=0):
    miss_send_len = UINT16_MAX
    actions = [self.dp.ofproto_parser.OFPActionOutput(
        self.dp.ofproto.OFPP_CONTROLLER, miss_send_len)]
    self.set_flow(cookie, priority, dl_type=dl_type, dl_dst=dl_dst,
                  dl_vlan=dl_vlan, nw_dst=dst_ip, dst_mask=dst_mask,
                  nw_proto=nw_proto, actions=actions)

def send_stats_request(self, stats, waiters):
    self.dp.set_xid(stats)
    waiters_per_dp = waiters.setdefault(self.dp.id, {})
    event = hub.Event()
    msgs = []
    waiters_per_dp[stats.xid] = (event, msgs)
    self.dp.send_msg(stats)

    try:
        event.wait(timeout=OFP_REPLY_TIMER)
    except hub.Timeout:
        del waiters_per_dp[stats.xid]

    return msgs

@OfCtl.register_of_version(ofproto_v1_0.OFP_VERSION)
class OfCtl_v1_0(OfCtl):

    def __init__(self, dp, logger):
        super(OfCtl_v1_0, self).__init__(dp, logger)

    def get_packetin_inport(self, msg):
        return msg.in_port

    def get_all_flow(self, waiters):
        ofp = self.dp.ofproto
        ofp_parser = self.dp.ofproto_parser

        match = ofp_parser.OFPMatch(ofp.OFPFW_ALL, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0, 0, 0)
        stats = ofp_parser.OFPFlowStatsRequest(self.dp, 0, match,
                                                0xff, ofp.OFPP_NONE)
        return self.send_stats_request(stats, waiters)

    def set_flow(self, cookie, priority, dl_type=0, dl_dst=0, dl_vlan=0,
                 nw_src=0, src_mask=32, nw_dst=0, dst_mask=32,
                 nw_proto=0, idle_timeout=0, actions=None):

```

```

ofp = self.dp.ofproto
ofp_parser = self.dp.ofproto_parser
cmd = ofp.OFPFC_ADD

# Match
wildcards = ofp.OFPFW_ALL
if dl_type:
    wildcards &= ~ofp.OFPFW_DL_TYPE
if dl_dst:
    wildcards &= ~ofp.OFPFW_DL_DST
if dl_vlan:
    wildcards &= ~ofp.OFPFW_DL_VLAN
if nw_src:
    v = (32 - src_mask) << ofp.OFPFW_NW_SRC_SHIFT | \
        ~ofp.OFPFW_NW_SRC_MASK
    wildcards &= v
    nw_src = ipv4_text_to_int(nw_src)
if nw_dst:
    v = (32 - dst_mask) << ofp.OFPFW_NW_DST_SHIFT | \
        ~ofp.OFPFW_NW_DST_MASK
    wildcards &= v
    nw_dst = ipv4_text_to_int(nw_dst)
if nw_proto:
    wildcards &= ~ofp.OFPFW_NW_PROTO

match = ofp_parser.OFPMatch(wildcards, 0, 0, dl_dst, dl_vlan, 0,
                             dl_type, 0, nw_proto,
                             nw_src, nw_dst, 0, 0)

actions = actions or []

m = ofp_parser.OFPFlowMod(self.dp, match, cookie, cmd,
                          idle_timeout=idle_timeout,
                          priority=priority, actions=actions)

self.dp.send_msg(m)

def set_routing_flow(self, cookie, priority, outport, dl_vlan=0,
                    nw_src=0, src_mask=32, nw_dst=0, dst_mask=32,
                    src_mac=0, dst_mac=0, idle_timeout=0, **dummy):
    ofp_parser = self.dp.ofproto_parser

    dl_type = ether.ETH_TYPE_IP

    # Decrement TTL value is not supported at OpenFlow V1.0
    actions = []
    if src_mac:
        actions.append(ofp_parser.OFPActionSetDlSrc(
            mac_lib.haddr_to_bin(src_mac)))
    if dst_mac:
        actions.append(ofp_parser.OFPActionSetDlDst(
            mac_lib.haddr_to_bin(dst_mac)))
    if outport is not None:

```

```

        actions.append(ofp_parser.OFPActionOutput (outport))

    self.set_flow(cookie, priority, dl_type=dl_type, dl_vlan=dl_vlan,
                  nw_src=nw_src, src_mask=src_mask,
                  nw_dst=nw_dst, dst_mask=dst_mask,
                  idle_timeout=idle_timeout, actions=actions)

def delete_flow(self, flow_stats):
    match = flow_stats.match
    cookie = flow_stats.cookie
    cmd = self.dp.ofproto.OFPFC_DELETE_STRICT
    priority = flow_stats.priority
    actions = []

    flow_mod = self.dp.ofproto_parser.OFPFlowMod(
        self.dp, match, cookie, cmd, priority=priority, actions=actions)
    self.dp.send_msg(flow_mod)
    self.logger.info('Delete flow [cookie=0x%x]', cookie, extra=self.sw_id)

class OfCtl_after_v1_2(OfCtl):

    def __init__(self, dp, logger):
        super(OfCtl_after_v1_2, self).__init__(dp, logger)

    def set_sw_config_for_ttl(self):
        pass

    def get_packetin_inport(self, msg):
        in_port = self.dp.ofproto.OFPP_ANY
        for match_field in msg.match.fields:
            if match_field.header == self.dp.ofproto.OXM_OF_IN_PORT:
                in_port = match_field.value
                break
        return in_port

    def get_all_flow(self, waiters):
        pass

    # MPLSmod: custom flow method
    def set_my_flow(self, cookie, priority, match, idle_timeout=0, actions=None):
        ofp = self.dp.ofproto
        ofp_parser = self.dp.ofproto_parser
        cmd = ofp.OFPFC_ADD

        inst = [ofp_parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS,
                                                actions)]

        m = ofp_parser.OFPFlowMod(self.dp, cookie, 0, 0, cmd, idle_timeout,
                                   0, priority, UINT32_MAX, ofp.OFPP_ANY,
                                   ofp.OFPG_ANY, 0, match, inst)

        self.dp.send_msg(m)

```



```

def set_flow(self, cookie, priority, dl_type=0, dl_dst=0, dl_vlan=0,
            nw_src=0, src_mask=32, nw_dst=0, dst_mask=32,
            nw_proto=0, idle_timeout=0, actions=None):
    ofp = self.dp.ofproto
    ofp_parser = self.dp.ofproto_parser
    cmd = ofp.OFPFC_ADD

    # Match
    match = ofp_parser.OFPMatch()
    if dl_type:
        match.set_dl_type(dl_type)
    if dl_dst:
        match.set_dl_dst(dl_dst)
    if dl_vlan:
        match.set_vlan_vid(dl_vlan)
    if nw_src:
        match.set_ipv4_src_masked(ipv4_text_to_int(nw_src),
                                   mask_ntob(src_mask))
    if nw_dst:
        match.set_ipv4_dst_masked(ipv4_text_to_int(nw_dst),
                                   mask_ntob(dst_mask))
    if nw_proto:
        if dl_type == ether.ETH_TYPE_IP:
            match.set_ip_proto(nw_proto)
        elif dl_type == ether.ETH_TYPE_ARP:
            match.set_arp_opcode(nw_proto)

    # Instructions
    actions = actions or []
    inst = [ofp_parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS,
                                             actions)]

    m = ofp_parser.OFPFlowMod(self.dp, cookie, 0, 0, cmd, idle_timeout,
                              0, priority, UINT32_MAX, ofp.OFPP_ANY,
                              ofp.OFPG_ANY, 0, match, inst)

    self.dp.send_msg(m)

def set_routing_flow(self, cookie, priority, outport, dl_vlan=0,
                    nw_src=0, src_mask=32, nw_dst=0, dst_mask=32,
                    src_mac=0, dst_mac=0, idle_timeout=0, dec_ttl=False):
    ofp = self.dp.ofproto
    ofp_parser = self.dp.ofproto_parser

    dl_type = ether.ETH_TYPE_IP

    actions = []
    if dec_ttl:
        actions.append(ofp_parser.OFPActionDecNwTtl())
    if src_mac:
        actions.append(ofp_parser.OFPActionSetField(eth_src=src_mac))
    if dst_mac:

```

```

        actions.append(ofp_parser.OFPActionSetField(eth_dst=dst_mac))
    if output is not None:
        actions.append(ofp_parser.OFPActionOutput(output, 0))

    self.set_flow(cookie, priority, dl_type=dl_type, dl_vlan=dl_vlan,
                  nw_src=nw_src, src_mask=src_mask,
                  nw_dst=nw_dst, dst_mask=dst_mask,
                  idle_timeout=idle_timeout, actions=actions)

def delete_flow(self, flow_stats):
    ofp = self.dp.ofproto
    ofp_parser = self.dp.ofproto_parser

    cmd = ofp.OFPPFC_DELETE
    cookie = flow_stats.cookie
    cookie_mask = UINT64_MAX
    match = ofp_parser.OFPMatch()
    inst = []

    flow_mod = ofp_parser.OFPFlowMod(self.dp, cookie, cookie_mask, 0, cmd,
                                      0, 0, 0, UINT32_MAX, ofp.OFPP_ANY,
                                      ofp.OFPG_ANY, 0, match, inst)

    self.dp.send_msg(flow_mod)
    self.logger.info('Delete flow [cookie=0x%x]', cookie, extra=self.sw_id)

@OfCtl.register_of_version(ofproto_v1_2.OFP_VERSION)
class OfCtl_v1_2(OfCtl_after_v1_2):

    def __init__(self, dp, logger):
        super(OfCtl_v1_2, self).__init__(dp, logger)

    def set_sw_config_for_ttl(self):
        flags = self.dp.ofproto.OFPC_INVALID_TTL_TO_CONTROLLER
        miss_send_len = UINT16_MAX
        m = self.dp.ofproto_parser.OFPSetConfig(self.dp, flags,
                                                  miss_send_len)

        self.dp.send_msg(m)
        self.logger.info('Set SW config for TTL error packet in.',
                          extra=self.sw_id)

    def get_all_flow(self, waiters):
        ofp = self.dp.ofproto
        ofp_parser = self.dp.ofproto_parser

        match = ofp_parser.OFPMatch()
        stats = ofp_parser.OFPFlowStatsRequest(self.dp, 0, ofp.OFPP_ANY,
                                                ofp.OFPG_ANY, 0, 0, match)

        return self.send_stats_request(stats, waiters)

@OfCtl.register_of_version(ofproto_v1_3.OFP_VERSION)

```

```

class OfCtl_v1_3(OfCtl_after_v1_2):

    def __init__(self, dp, logger):
        super(OfCtl_v1_3, self).__init__(dp, logger)

    def set_sw_config_for_ttl(self):
        packet_in_mask = (1 << self.dp.ofproto.OFPR_ACTION |
                          1 << self.dp.ofproto.OFPR_INVALID_TTL)
        port_status_mask = (1 << self.dp.ofproto.OFPPR_ADD |
                             1 << self.dp.ofproto.OFPPR_DELETE |
                             1 << self.dp.ofproto.OFPPR_MODIFY)
        flow_removed_mask = (1 << self.dp.ofproto.OFPPR_IDLE_TIMEOUT |
                              1 << self.dp.ofproto.OFPPR_HARD_TIMEOUT |
                              1 << self.dp.ofproto.OFPPR_DELETE)
        m = self.dp.ofproto_parser.OFPSetAsync(
            self.dp, [packet_in_mask, 0], [port_status_mask, 0],
            [flow_removed_mask, 0])
        self.dp.send_msg(m)
        self.logger.info('Set SW config for TTL error packet in.',
                          extra=self.sw_id)

    def get_all_flow(self, waiters):
        ofp = self.dp.ofproto
        ofp_parser = self.dp.ofproto_parser

        match = ofp_parser.OFPMatch()
        stats = ofp_parser.OFPFlowStatsRequest(self.dp, 0, 0, ofp.OFPP_ANY,
                                                ofp.OFPG_ANY, 0, 0, match)

        return self.send_stats_request(stats, waiters)

    # MPLSmod: method to add mpls flows
    def set_mpls_flow(self, cookie, priority, label, in_port, out_port, action,
                      dl_vlan=0, nw_src=0, src_mask=32, nw_dst=0, dst_mask=32,
                      src_mac=0, dst_mac=0, idle_timeout=0, oldlabel=0):

        parser = self.dp.ofproto_parser
        if action == MPLS_PUSH_LABEL:
            dl_type = ether.ETH_TYPE_IP
            actions = [parser.OFPActionPushMpls(ethertype=34887),
                       parser.OFPActionSetField(mpls_label=label),
                       parser.OFPActionOutput(out_port)]

            self.set_flow(cookie, priority, dl_type=dl_type, dl_vlan=dl_vlan,
                          nw_src=nw_src, src_mask=src_mask,
                          nw_dst=nw_dst, dst_mask=dst_mask,
                          idle_timeout=idle_timeout, actions=actions)

        elif action == MPLS_SWAP_LABEL:
            dl_type = ether.ETH_TYPE_MPLS
            actions = [parser.OFPActionPopMpls(),
                       parser.OFPActionPushMpls(ethertype=34887),
                       parser.OFPActionSetField(mpls_label=label),

```

```

        parser.OFPActionOutput(out_port)]
    match = parser.OFPMatch(in_port=in_port,
                            eth_type=dl_type, mpls_label=oldlabel)
    self.set_my_flow(cookie, priority, match,
                    idle_timeout=idle_timeout, actions=actions)

    elif action == MPLS_POP_LABEL:
        dl_type = ether.ETH_TYPE_MPLS
        actions = [parser.OFPActionPopMpls(),
                  parser.OFPActionSetField(eth_src=src_mac),
                  parser.OFPActionSetField(eth_dst=dst_mac),
                  parser.OFPActionOutput(out_port)]
        match = parser.OFPMatch(eth_type=dl_type, mpls_label=oldlabel)
        self.set_my_flow(cookie, priority, match,
                        idle_timeout=idle_timeout, actions=actions)

# MPLSmod: get mpls label
def get_packetin_mplslabel(self, msg):
    pkt = packet.Packet(msg.data)
    mpls_proto = pkt.get_protocol(mpls.mpls)
    return mpls_proto.label

def ip_addr_aton(ip_str, err_msg=None):
    try:
        return addrconv.ipv4.bin_to_text(socket.inet_aton(ip_str))
    except (struct.error, socket.error) as e:
        if err_msg is not None:
            e.message = '%s %s' % (err_msg, e.message)
        raise ValueError(e.message)

def ip_addr_ntoa(ip):
    return socket.inet_ntoa(addrconv.ipv4.text_to_bin(ip))

def mask_ntob(mask, err_msg=None):
    try:
        return (UINT32_MAX << (32 - mask)) & UINT32_MAX
    except ValueError:
        msg = 'illegal netmask'
        if err_msg is not None:
            msg = '%s %s' % (err_msg, msg)
        raise ValueError(msg)

def ipv4_apply_mask(address, prefix_len, err_msg=None):
    import itertools

    assert isinstance(address, str)
    address_int = ipv4_text_to_int(address)
    return ipv4_int_to_text(address_int & mask_ntob(prefix_len, err_msg))

```

```

def ipv4_int_to_text(ip_int):
    assert isinstance(ip_int, numbers.Integral)
    return addrconv.ipv4.bin_to_text(struct.pack('!I', ip_int))

def ipv4_text_to_int(ip_text):
    if ip_text == 0:
        return ip_text
    assert isinstance(ip_text, str)
    return struct.unpack('!I', addrconv.ipv4.text_to_bin(ip_text))[0]

def nw_addr_aton(nw_addr, err_msg=None):
    ip_mask = nw_addr.split('/')
    default_route = ip_addr_aton(ip_mask[0], err_msg=err_msg)
    netmask = 32
    if len(ip_mask) == 2:
        try:
            netmask = int(ip_mask[1])
        except ValueError as e:
            if err_msg is not None:
                e.message = '%s %s' % (err_msg, e.message)
            raise ValueError(e.message)
    if netmask < 0:
        msg = 'illegal netmask'
        if err_msg is not None:
            msg = '%s %s' % (err_msg, msg)
        raise ValueError(msg)
    nw_addr = ipv4_apply_mask(default_route, netmask, err_msg)
    return nw_addr, netmask, default_route

```



# Bibliography

- [1] Does open vswitch support mpls? <https://github.com/openvswitch/ovs/blob/master/FAQ.md#q-does-open-vswitch-support-mpls>.
- [2] Open vswitch datapath developer documentation. <https://www.kernel.org/doc/Documentation/networking/openvswitch.txt>.
- [3] Open vswitch github repository. <https://github.com/openvswitch/ovs>.
- [4] Ryu online documentation. <http://ryu.readthedocs.org/en/latest/index.html>.
- [5] What linux kernel versions does each open vswitch release work with? <https://github.com/openvswitch/ovs/blob/master/FAQ.md#q-what-linux-kernel-versions-does-each-open-vswitch-release-work-with>.
- [6] Open Networking Foundation. Software-defined networking: The new norm for networks. onf white paper, 2012.
- [7] Open Networking Foundation. Sdn architecture overview, 2013.
- [8] Open Networking Foundation. Openflow switch specification. version 1.3.5 ( protocol version 0x04 ), 2015.
- [9] Eric Lopez. Openstack: Ovs deep dive. conference at openstack summit, hong kong. <https://www.youtube.com/watch?v=x-F9bDRxjAM>, 2013.
- [10] Sean Michael. Linux 3.19 release adds mpls support to openvswitch. <http://www.linuxplanet.com/news/linux-3.19-release-adds-mpls-support-to-openvswitch.html>.
- [11] Ryu project team. *Ryu SDN Framework - English Edition*. RYU project team, 2014.