Using OpenFlow 1.3

# RYU

## SDN Framework

RYU project team

Contents

# 前言

本书是给那些使用 Ryu 作为开发的框架，而目的是为了实现软件定义网络（Software Defined Networking，SDN ）而写的一本参考书。

那么为什么要选择 Ryu 作为开发平台呢？ 我们衷心的希望您可以在

本书中找到解答。

建议您依照本书的章节，依序的阅读第 1 章至第 5 章。首先第 1 章是 Simple Switch 的实作，接着后面的章节是流量监控（Traffic Monitor ）以及网络聚合（Link Aggregation ）。透过实际的例子，我们将介绍 Ryu 的程序是如何运作的。

第 6 章到第 9 章，我们将详细的说明 OpenFlow 通讯协议（OpenFlow Protocol ）以及封包函数的使用方法。接着第 10 章到第 12 章，我们会讨论如何使用 Ryu 内建的防火墙（Firewall ）和测试工具（test tool ）来开发应用程序。第 13 到 15 章将介绍 Ryu 的架构（architecture ）及实际应用案例。

最后，我们非常感谢您对于 Ryu 项目的青睐及支持，特别对于 Ryu 的使用者而言。希望在不久的将来能有机会在 Mailing List 上得到来自于您的宝贵意见，让我们一起加入 Ryu 项目并进行开发吧。

繁体中文版本是由台湾信息工业策进会 - SDN 团队协助整理及翻译，除了不吝批评与指教之外也欢迎更多朋友加入，让中文读者可以更快更实时的了解 Ryu 这个优秀的开放原始码的框架。

CHAPTER 1

# 交换器（Switching Hub ）

本章将会用简单的 Switching hub 安装做为题材，说明 Ryu 如何安装一个应用程序。

## 1.1 Switching Hub

在交换器中有许许多多的功能。在这边我们将看到拥有下列简单功能的交换器。

- 学习连接到端口的 host 之 MAC 地址，并记录在 MAC 地址表当中。
- 对于已经记录下来的 MAC 地址，若是收到送往该 MAC 地址的封包，则转送该封包到相对应的端口。
- 对于未指定目标地址的封包，则执行 Flooding。 让我们使

用 Ryu 来实现这样一个交换器吧。

## 1.2 OpenFlow 实作的交换器

OpenFlow 交换器会接受来自于 controller 的指令并达到以下功能。

- 对于接收到的封包进行修改或针对指定的端口进行转送。
- 对于接收到的封包进行转送到 Controller 的动作（Packet-In ）。
- 对于接收到来自 Controller 的封包转送到指定的端口（Packet-Out ）。上述的功能所组合

起来的就是一台交换器的实现。

首先，利用 Packet-In 的功能来达到 MAC 地址的学习。Controller 使用 Packet-In 接收来自交换器的封包之后进行分析，得到端口相关数据以及所连接的 host 之 MAC 地址。

在学习之后，对所收到的封包进行转送。将封包的目的地址，在已经学习的 host 数据中进行检 索，根据检索的结果会进行下列处理。

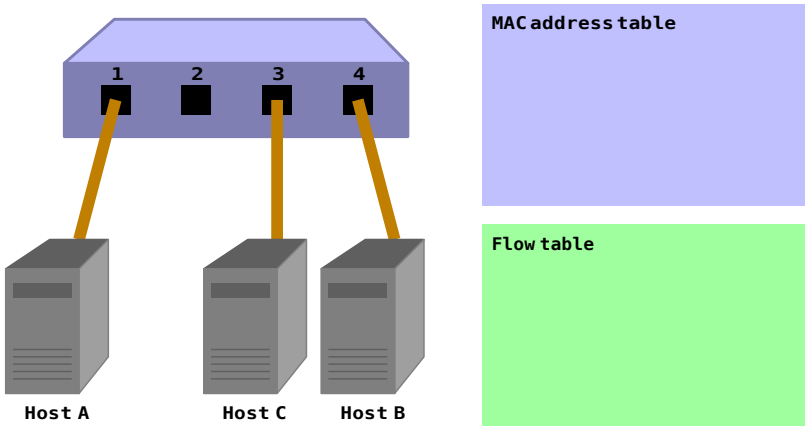
- 如果是已经存在记录中的 host：使用 Packet-Out 功能转送至先前所对应的端口
- 如果是尚未存在记录中的 host：使用 Packet-Out 功能来达到 Flooding 下面将一步一步的

说明并附上图片以帮助理解。

1. 初始状态

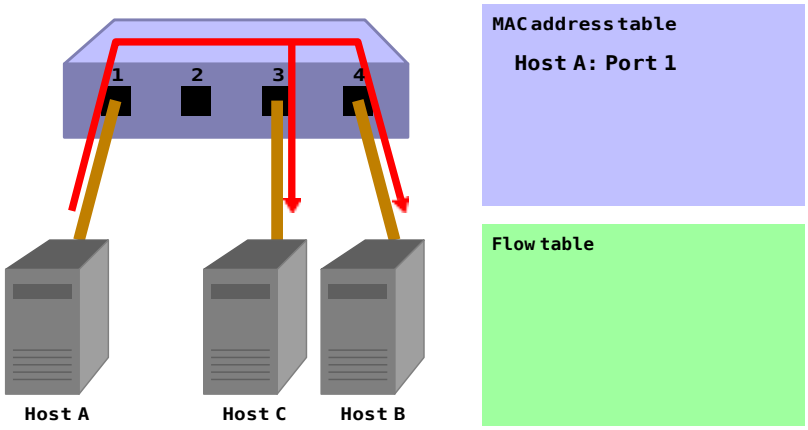
Flow table为空白的状况。

将 host A 接到端口 1，host B 接到端口 4，host C 接到端口 3。




2. host A → host B

当 host A 向 host B 发送封包。这时后会触发 Packet-In 讯息。host A 的 MAC 地址会被端口 1 给记录下来。由于 host B 的 MAC 地址尚未被学习，因此会进行 Flooding 并将封包往 host B 和 host C 发送。




Packet-In:

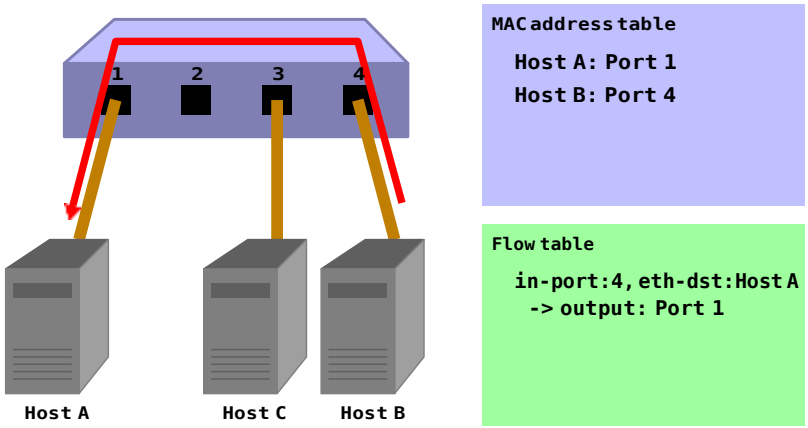
```
in-port: 1
eth-dst: host B
eth-src: host A
```

Packet-Out:

```
action: OUTPUT:Flooding
```

3. host B → host A

封包从 host B 向 host A 返回时，在 Flow table 中新增一笔 Flow Entry，并将封包转送到端口 1。因此该封包并不会被 host C 收到。




Packet-In:

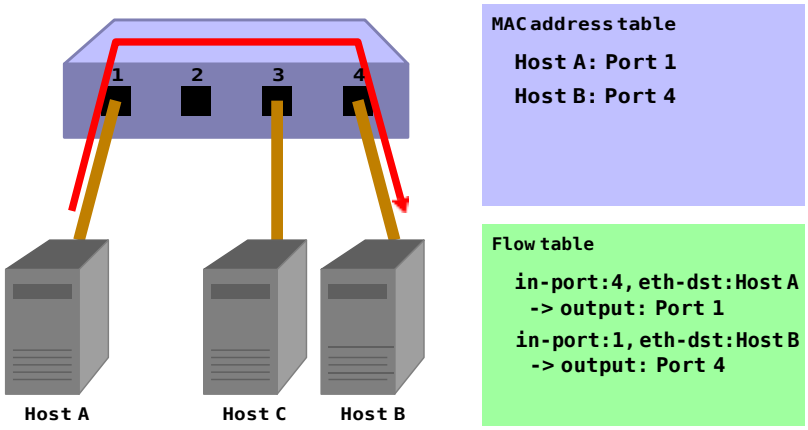
```
in-port: 4
eth-dst: host A
eth-src: host B
```

Packet-Out:

```
action: OUTPUT: port 1
```

4. host A → host B

再一次，host A 向 host B 发送封包，在 Flow table 中新增一个 Flow Entry 接着转送封包到端口 4。




Packet-In:

```
in-port: 1
eth-dst: host B
eth-src: host A
```

Packet-Out:

```
action: OUTPUT: port 4
```

接下来，让我们实际来看一下在 Ryu 当中实作交换器的原始码。

## 1.3 在 Ryu 上实作交换器

Ryu 的原始码之中有提供交换器的程序原始码。

ryu/app/simple_switch_13.py

OpenFlow 其他的版本也有相对应的原始码，例如 simple_switch.py（OpenFlow 1.0）和 simple_switch_12.py（OpenFlow 1.2）。我们现在要来看的则是 OpenFlow 1.3 的版本。

由于原始码不多，因此我们把全部都拿来检视。

```
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.
        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):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

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

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

定义交换机，继承app_manager.RyuApp

实现继承，并初始化父类的方法

mac地址存放在字典中

```
        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)
            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)
```

if目的地址在mac的列表中，输出端口就是xx；如果不在就泛洪

添加流表项

那么我们开始看一下其中的内容吧。

https://www.sdnlab.com/1785.html，讲了ryu中各个目录的意思，包括base里面是什么

### 1.3.1 类别的定义和初始化

为了要实作 Ryu 应用程序，因此继承了 ryu.base.app_manager.RyuApp。接着为了使用 OpenFlow 1.3 ，将 OFP_VERSIONS 指定为 OpenFlow 1.3。

然后，MAC 地址表的 mac_to_port 也已经被定义。

OpenFlow 通讯协议中有些程序像是握手协议（handshake ），是定义好让 OpenFlow 交换器和 Controller 之间进行通讯时使用。但这些细节，对于一个 Ryu 应用程序来说是不用担心或需要特别处理的。

```
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 = {}

    # ...
```

### 1.3.2 事件管理（Event handler）

对于 Ryu 来说，接受到任何一个 OpenFlow 讯息即会产生一个相对应的事件。而 Ryu 应用程序则是必须实作事件管理以处理相对应发生的事件。

事件管理（Event Handler）是一个拥有事件对象（Event Object）做为参数，并且使用``ryu.controller.handler.set_ev_cls``修饰（Decorator）的函数。

set_ev_cls 则指定事件类别得以接受讯息和交换器状态作为参数。

事件类别名称的规则为 `ryu.controller.ofp_event.EventOFP` + <OpenFlow 讯息名称>，例如：在 Packet-In 讯息的状态下的事件名称为 `EventOFPPacketIn` 。详细的内容请参考 Ryu 的文件 API 参考数据。对于状态来说，请指定下面列表的其中一项。

| 名称 | 说明 |
|---|---|
| ryu.controller.handler.HANDSHAKE_DISPATCHER | 交换 HELLO 讯息 |
| ryu.controller.handler.CONFIG_DISPATCHER | 接收 SwitchFeatures 讯息 |
| ryu.controller.handler.MAIN_DISPATCHER | 一般状态 |
| ryu.controller.handler.DEAD_DISPATCHER | 联机中断 |

#### 新增 Table-miss Flow Entry

OpenFlow 交换器的握手协议完成之后，新增 Table-miss Flow Entry 到 Flow table 中为接收 Packet-In 讯息做准备。

具体来说，接收到 Switch features（Features reply）讯息后就会新增 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

    # ...
```

`ev.msg` 是用来储存对应事件的 OpenFlow 讯息类别实体。在这个例子中则是 `ryu.ofproto.ofproto_v1_3_parser.OFPSwitchFeatures` 。

`msg.datapath` 这个讯息是用来储存 OpenFlow 交换器的 `ryu.controller.controller.Datapath` 类别所对应的实体。

Datapath 类别是用来处理 OpenFlow 交换器重要的讯息，例如执行与交换器的通讯和触发接收讯息相关的事件。

Ryu 应用程序所使用的主要属性如下：

| 名称 | 说明 |
| --- | --- |
| id | 连接 OpenFlow 交换器的 ID（datapath ID）。 |
| ofproto | 表示使用的 OpenFlow 版本所对应的 ofproto module。目前的状况会是下述的其中之一。<br>ryu.ofproto.ofproto_v1_0<br>ryu.ofproto.ofproto_v1_2<br>ryu.ofproto.ofproto_v1_3<br>ryu.ofproto.ofproto_v1_4 |
| ofproto_parser | 和 ofproto 一样，表示 ofproto_parser module。目前的状况会是下述的其中之一。<br>ryu.ofproto.ofproto_v1_0_parser<br>ryu.ofproto.ofproto_v1_2_parser<br>ryu.ofproto.ofproto_v1_3_parser<br>ryu.ofproto.ofproto_v1_4_parser |

Ryu 应用程序中 Datapath 类别的主要方法如下：

send_msg(msg)

发送 OpenFlow 讯息。msg 是发送 OpenFlow 讯息 ryu.ofproto.ofproto_parser.MsgBase 类别的子类别。

交换器本身不仅仅使用 Switch features 讯息，还使用事件处理以取得新增 Table-miss Flow Entry 的时间点。

```
def switch_features_handler(self, ev):

    # ...

    # 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.
    match = parser.OFPMatch()
    actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                      ofproto.OFPCML_NO_BUFFER)]
    self.add_flow(datapath, 0, match, actions)
```

Table-miss Flow Entry 的优先权为 0 即最低的优先权，而且此 Entry 可以 match 所有的封包。这个 Entry 的 Instruction 通常指定为 output action，并且输出的端口将指向 Controller。因此当封包没有 match 任何一个普通 Flow Entry 时，则触发 Packet-In。

备注：目前（2014 年 1 月），市面上的 Open vSwitch 对于 OpenFlow 1.3 的支持并不完整，而且对于 OpenFlow 1.3 以前的版本 Packet-In 是个基本的功能。包括 Table-miss Flow Entry 也尚未被支持，仅仅是使用一般的 Flow Entry 取代。

空的 match 将被产生为了 match 所有的封包。match 表示于 OFPMatch 类别中。

接下来，为了转送到 Controller 端口，OUTPUT action 类别（OFPActionOutput）的实例将会被产生。Controller 会被指定为封包的目的地，OFPCML_NO_BUFFER 会被设定为 max_len 以便接下来的封包传送。

备注：送往 Controller 的封包可以仅只传送 header 部分（Ethernet header），剩下的则存在缓冲区间中以增加效率。但目前（2014 年 1 月）Open vSwitch 存在臭虫的关系，会将所有的封包都传送，并不会只传送 header。

最后将优先权设定为 0（最低优先权），然后执行 add_flow() 方法以发送 Flow Mod 讯息。add_flow() 方法的内容将会在稍后进行说明。

### Packet-in 讯息

为了接收处理未知目的地的封包，需要 Packet-In 事件管理。

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    msg = ev.msg
    datapath = msg. datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    # ...
```

OFPPacketIn 类别经常使用的属性如下列所示。

| 名称 | 说明 |
|---|---|
| match | ryu.ofproto.ofproto_v1_3_parser.OFPMatch 类别的实体，用来储存接收封包的 Meta 信息。 |
| data | 接收封包本身的 binary 数据 |
| total_len | 接收封包的数据长度 |
| buffer_id | 接收封包的内容若是存在 OpenFlow 交换器上时所指定的 ID 如果在没有 buffer 的状况下，则设定 ryu.ofproto.ofproto_v1_3.OFP_NO_BUFFER |

### 更新 MAC 地址表

```
def _packet_in_handler(self, ev):

    # ...

    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

    # ...
```

向mac表中添加默认值，默认为空

将入端口写入mac表中

从 OFPPacketIn 类别的 match 得到接收端口（in_port）的信息。目的 MAC 地址和来源 MAC 位 址使用 Ryu 的封包函式库，从接收到封包的 Ethernet header 取得。

藉由得知目的 MAC 地址和来源 Mac 地址，更新 MAC 地址表。

为了可以对应连接到多个 OpenFlow 交换器，MAC 地址表和每一个交换器之间的识别，就使用 datapath ID 来进行确认。

### 判断转送封包的端口

目的 MAC 地址若存在于 MAC 地址表，则判断该端口的号码为输出。反之若不存在于 MAC 地址表则 OUTPUT action 类别的实体并生成 flooding（OFPP_FLOOD）给目的端口使用。

```
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)
        self.add_flow(datapath, 1, match, actions)

    # ...
```

若是找到了目的 MAC 地址，则在交换器的 Flow table 中新增。

Table-miss Flow Entry 包含 match 和 action，并透过 add_flow() 来新增。

不同于平常的 Table-miss Flow Entry，这次将加上设定 match 条件。本次交换器实作中，接收埠（in_port）和目的 MAC 地址（eth_dst）已指定。例如，接收到来自端口 1 的封包就传送到 host B。

在这边指定 Flow Entry 优先权为 1，而优先权的值越大，表示有更高的优先权。因此，这边新增的 Flow Entry 将会先于 Table-miss Flow Entry 而被执行。

上述的内容包含 action 整理如下，这些 Entry 会被新增至 Flow Entry：

> 端口 1 接收到的封包，若是要转送至 host B（目的 MAC 地址 B) 的封包则转送至端口 4。

提示: 在 OpenFlow 中，有个逻辑端口叫做 NORMAL 并在规范中被列为选项（也就是说可以不进行实作）。当被指定的端口为 NORMAL 时，传统 L2/L3 的功能将会被启用来处理封包。意思是当把所有输 出的埠均设定为 NORMAL 时，交换器将会视作一个普通的交换器而存在。跟一般交换器的差别在于我们 是使用 OpenFlow 来达到这样的功能。

### 新增 Flow Entry 的处理

Packet-In handler的处理尚未说明，在这之前我们先来看一看新增Flow Entry的方法。

```
def add_flow(self, datapath, priority, match, actions):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

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

    # ...
```

对于 Flow Entry 来说，设定 match 条件以分辨目标封包、设定 instruction 以处理封包以及 Entry 的优先权和有效时间。

对于交换器的的实作，Apply Actions 是用来设定那些必须立即执行的 action 所使用。

最后透过 Flow Mod 讯息将 Flow Entry 新增到 Flow table 中。

```
def add_flow(self, datapath, port, dst, actions):

    # ...

    mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                            match=match, instructions=inst)
    datapath.send_msg(mod)
```

Flow Mod 讯息的类别为 `OFPFlowMod`。使用 OFPFlowMod 所产生的实体透过 Datapath.send_msg() 方法来发送讯息给 OpenFlow 交换器。

OFPFlowMod 类别的建构子参数相当的多，但是在大多数的情况下都有其默认值，原始码中括 号内的部分即是默认值。

datapath

OpenFlow 交换器以及 Flow table 的操作都是透过 Datapath 类别的实体来进行。在一般的情况下，会由事件传递给事件管理的讯息中取得，例如：Packet-In 讯息。

cookie (0)

Controller 所设定储存的数据，在 Entry 的更新或者删除的时所需要使用的数据都会放在这边，当做过滤器使用，而且不可以作为封包处理的参数。

cookie_mask (0)

Entry 的更新或删除时，若是该值为非零，则做为指定 Entry 的 cookie 使用。

table_id (0)

指定 Flow Entry 的 Table ID。

command (ofproto_v1_3.OFPFC_ADD)

指定要执行何项操作。

| 名称 | 说明 |
|---|---|
| OFPFC_ADD | Flow Entry 新增 |
| OFPFC_MODIFY | Flow Entry 更新 |
| OFPFC_MODIFY_STRICT | 严格的 Flow Entry 更新 |
| OFPFC_DELETE | Flow Entry 删除 |
| OFPFC_DELETE_STRICT | 严格的 Flow Entry 删除 |

idle_timeout (0)

Flow Entry 的有效期限，以秒为单位。Flow Entry 如果未被参照而且超过了指定的时间之后，该 Flow Entry 将会被删除。如果 Flow Entry 有被参照，则超过时间之后会重新归零计算。

在 Flow Entry 被删除之后就会发出 Flow Removed 讯息通知 Controller。

hard_timeout (0)

Flow Entry 的有效期限，以秒为单位。跟 idle_timeout 不同的地方是，hard_timeout 在超过时限后并不会重新归零计算。也就是说跟 Flow Entry 与有没有被参照无关，只要超过指定的时间就会被删除。

跟 idle_timeout 一样，当 Flow Entry 被删除时，Flow Removed 讯息将会被发送来通知 Controller。

priority (0)

Flow Entry 的优先权。数值越大表示权限越高。

buffer_id (ofproto_v1_3.OFP_NO_BUFFER)

指定 OpenFlow 交换器上用来储存封包的缓冲区 ID。缓冲区 ID 会放在通知 Controller 的 Packet-In 讯息中，并且和接下来的 OFPP_TABLE 所指定的输出端口和 Flow Mod 讯息处理时可以被参照。当发送的命令讯息为 OFPFC_DELETE 或 OFPFC_DELETE_STRICT 时，会忽略本数值。

如果不指定缓冲区 ID 的时候，必须使用 `OFP_NO_BUFFER` 作为其设定值。

out_port (0)

OFPFC_DELETE 和 OFPFC_DELETE_STRICT 命令用来指定输出埠的参数。命令为 OFPFC_ADD、OFPFC_MODIFY、OFPFC_MODIFY_STRICT 时则可以忽略。

若要让本参数无效则指定输出埠为 `OFPP_ANY` 。

out_group (0)

跟 out_port 一样，作为一个输出埠，但是转到特定的 group。

若要使其无效，则指定为 `OFPG_ANY` 。

flags (0)

下列的 flags 可以被组合使用。

| 名称 | 说明 |
|---|---|
| OFPFF_SEND_FLOW_REM | Flow Entry 被移除的时候，对 Controller 发送 Removed 讯息。 |
| OFPFF_CHECK_OVERLAP | 使用 OFPFC_ADD 时，检查是否有重复的 Flow Entry 存在。若是有则触发 Flow Mod 失败，并返回错误讯息。 |
| OFPFF_RESET_COUNTS | 重设该 Flow Entry 的 packet counter 和 byte counter。 |
| OFPFF_NO_PKT_COUNTS | 关闭该 Flow Entry 的 packet counter 功能。 |
| OFPFF_NO_BYT_COUNTS | 关闭该 Flow Entry 的 byte counter 功能。 |

match (None)

设定 match 。

instructions ([])

设定 instruction 。

### 转送封包

回到 Packet-In handler 并说明最后的流程。

在 MAC 地址表中找寻目的 MAC 地址，若是有找到则发送 Packet-Out 讯息，并且转送封包。

```
def _packet_in_handler(self, ev):

    # ...

    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)
```

Packet-Out 讯息相对应的类别是 `OFPPacketOut`。

OFPPacketOut 建构子的参数如下所示。

datapath

指定 OpenFlow 交换器对应的 Datapath 类别实体。

buffer_id

指定 OpenFlow 交换器上封包对应的缓冲区。如果不想使用缓冲区，则指定为 `OFP_NO_BUFFER`。

in_port

指定接收封包的端口号。如果不想使用的话就指定为 `OFPP_CONTROLLER` 。

actions

指定 actions list。

data

设定封包的 binary data。主要用在 buffer_id 为 `OFP_NO_BUFFER` 的情况。如果使用了 OpenFlow 交换器的缓冲区则可以省略。

交换器的实作时，在 Packet-In 讯息中指定 buffer_id。若是 Packet-In 讯息中 buffer_id 被设定为无效时。Packet-In 的封包必须指定 data 以便传送。

交换器的原始码说明就到这边。接下来我们将执行交换器以确认相关的动作。

## 1.4 执行 Ryu 应用程序

为了执行交换器，OpenFlow 交换器采用 Open vSwitch，执行环境则是在 mininet 上。

由于 Ryu 的 OpenFlow Tutorial VM 映像档已经准备好了，有了该 VM 映像档会让准备工作较为简单。

VM 映像档

http://sourceforge.net/projects/ryu/files/vmimages/OpenFlowTutorial/

OpenFlow_Tutorial_Ryu3.2.ova（约 1.4GB ）

相关文件（Wiki 网页）

https://github.com/osrg/ryu/wiki/OpenFlow_Tutorial

文件中的描述和 VM 映像档有可能使用较旧版本的 Open vSwitch 和 Ryu，请特别注意。

若是不想使用 VM 映像档，当然可以自行打造环境。若是您决定自行搭建环境，请参考以下软体的版本。

Mininet VM 2.0.0 http://mininet.org/download/

Open vSwitch 1.11.0 http://openvswitch.org/download/

Ryu 3.2 https://github.com/osrg/ryu/

```
$ sudo pip install ryu
```

但是在这边我们使用 Ryu 的 OpenFlow Tutorial VM 映像档。

### 1.4.1 执行 Mininet

因为所需要的终端机 xterm 是从 mininet 中启动，因此 X windows 的环境是必要的。

为了使用 OpenFlow Tutorial 的 VM，请把 ssh 的 X11 Forwarding 功能打开并进行登入。

```
$ ssh -X ryu@<VM 的 IP>
```

用户名称为 ryu、密码也是 ryu。

登入以后使用 mn 指令启动 Mininet 环境。 要建构的是 host 3 台，交换器 1 台的简单环境。 mn 命令的参数如下：

| 名称 | 数值 | 说明 |
|---|---|---|
| topo | single,3 | 交换器 1 台、host 3 台的拓璞 |
| mac | 无 | 自动设定 host 的 MAC 地址 |
| switch | ovsk | 使用 Open vSwitch |
| controller | remote | 指定外部的 OpenFlow Controller |
| x | 无 | 启动 xterm |

执行的方法如下：

```
$ sudo mn --topo single,3 --mac --switch ovsk --controller remote -x
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1)
*** Configuring hosts
h1 h2 h3
*** Running terms on localhost:10.0
*** Starting controller
*** Starting 1 switches
s1
*** Starting CLI:
mininet>
```

执行之后，会在 x windows 出现 5 个 xterm 窗口。分别对应到 host 1 ~ 3 ，交换器和 Controller 。

在交换器的 xterm 窗口中设定 OpenFlow 的版本，窗口的标题为「switch: s1 (root)」。 首先查看 Open vSwitch 的状态。

switch: s1:

```
root@ryu - vm:~# ovs - vsctl show
fdec0957 -12b6 -4417 -9d02 -847654 e9cc1f
```

```
Bridge "s1"
    Controller " ptcp :6634"
    Controller "tcp:127.0.0.1:6633"
    fail_mode: secure
    Port " s1 - eth3 "
        Interface "s1-eth3"
    Port " s1 - eth2 "
        Interface "s1-eth2"
    Port " s1 - eth1 "
        Interface "s1-eth1"
    Port "s1"
        Interface " s1 "
            type: internal
ovs_version: "1.11.0"
root@ryu-vm:~# ovs-dpctl show
system@ovs-system:
        lookups: hit:14 missed:14 lost:0
        flows: 0
        port 0: ovs-system (internal)
        port 1: s1 (internal)
        port 2: s1-eth1
        port 3: s1-eth2
        port 4: s1-eth3
root@ryu-vm:~#
```

交换器（网桥）s1 被建立，并且增加 3 个端口分别联机到 3 个 host。

接下来设定 OpenFlow 的版本为 1.3。

switch: s1:

```
root@ryu-vm:~# ovs-vsctl set Bridge s1 protocols=OpenFlow13
root@ryu-vm:~#
```

检查空白的 Flow table。

switch: s1:

```
root@ryu-vm:~# ovs-ofctl -O OpenFlow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
root@ryu-vm:~#
```

ovs-ofctl 命令选项是用来指定 OpenFlow 版本。默认值是 OpenFlow10。

### 1.4.2 执行交换器

准备工作到此已经结束，接下来开始执行 Ryu 应用程序。 在

窗口标题为「controller: c0 (root)」的 xterm 执行下列指令。

controller: c0:

```
root@ryu-vm:~# ryu-manager --verbose ryu.app.simple_switch_13
loading app ryu.app.simple_switch_13
loading app ryu. controller. ofp_handler
instantiating app ryu.app.simple_switch_13
instantiating app ryu.controller.ofp_handler
BRICK SimpleSwitch13
  CONSUMES EventOFPSwitchFeatures
  CONSUMES  EventOFPPacketIn
BRICK ofp_event
```

```
  PROVIDES EventOFPSwitchFeatures TO {'SimpleSwitch13': set(['config'])}
  PROVIDES EventOFPPacketIn TO {' SimpleSwitch13 ': set([' main '])}
  CONSUMES  EventOFPErrorMsg
  CONSUMES EventOFPHello
  CONSUMES  EventOFPEchoRequest
  CONSUMES EventOFPPortDescStatsReply
  CONSUMES  EventOFPSwitchFeatures
connected socket:<eventlet.greenio.GreenSocket object at 0x2e2c050> address
:('127.0.0.1', 53937)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0x2e2a550>
move onto config mode
EVENT ofp_event->SimpleSwitch13 EventOFPSwitchFeatures
switch features ev version: 0x4 msg_type 0x6 xid 0xff9ad15b
OFPSwitchFeatures( auxiliary_id=0,capabilities=71,datapath_id=1,n_buffers
=256,n_tables=254)
```

正在进行 OVS 的连接动作，这需要花一点时间。

```
connected socket:<....
hello ev ...
...
move onto main mode
```

完成之后就会显示上述的讯息。

现在OVS已经连接，handshake已经执行完毕，Table-miss Flow Entry已经加入，正处于等待Packet-In 的状态。

确认Table-miss Flow Entry已经被加入。

switch: s1:

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=105.975s, table=0, n_packets=0, n_bytes=0, priority=0 actions
=CONTROLLER:65535
root@ryu-vm:~#
```

优先权为 0，没有 matcth，action 为 CONTROLLER，重送的资料大小为 65535 (0xffff = OFPCML_NO_BUFFER )。

### 1.4.3 确认操作

从 host 1 向 host 2 发送 ping。

1. ARP request

   此时 host 1 并不知道 host 2 的 MAC 地址，原则上 ICMP echo request 之前的 ARP request 是用广播的方式发送。这样的广播方式会让 host 2 和 host 3 都同样接受到讯息。

2. ARP reply

   host 2 使用 ARP reply 回复 host 1 要求。

3. ICMP echo request

   现在 host 1 知道了 host 2 的 MAC 地址，因此发送 echo request 给 host 2。

4. ICMP echo reply

host 2 此时也知道了 host 1 的 MAC 地址，因此发送 echo reply 给 host 1。

原则上动作流程应该如同描述一般。

在 ping 命令执行之前，为了确认每一台 host 都可以收到，执行 tcpdump 以确认封包有确实被接收。

host: h1:

```
root@ryu-vm:~# tcpdump -en -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
```

host: h2:

```
root@ryu-vm:~# tcpdump -en -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
```

host: h3:

```
root@ryu-vm:~# tcpdump -en -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
```

然后，在终端机执行 mn 命令，并从 host 1 发送 ping 到 host 2。

```
mininet> h1 ping -c1 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_req=1 ttl=64 time=97.5 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/ avg/ max/ mdev = 97.594/97.594/97.594/0.000 ms
mininet>
```

ICMP echo reply 正常的被回复。

继续下去之前先确认 Flow table。

switch: s1:

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=417.838s, table=0, n_packets=3, n_bytes=182, priority=0
actions=CONTROLLER:65535
 cookie=0x0, duration=48.444s, 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=48.402s, table=0, n_packets=1, n_bytes=42, priority=1,in_port
=1,dl_dst=00:00:00:00:00:02 actions=output:2
root@ryu-vm:~#
```

Table-miss Flow Entry 以外，另外加入两个优先权为 1 的 Flow Entry。

1. 接收埠（in_port）:2, 目的 MAC 地址（dl_dst）:host 1 → actions:host 1 转送
2. 接收埠（in_port）:1, 目的 MAC 地址（dl_dst）:host 2 → actions:host 2 转送

(1) 的 Flow Entry 会被 match 2 次（n_packets）、(2) 的 Flow Entry 则被 match 1 次。因为 (1) 用来让 host 2 向 host 1 传送封包用，ARP reply 和 ICMP echo reply 都会发生 match。(2) 是用来从 host 1 向 host 2 发送讯息，由于 ARP request 是采用广播的方式，原则上透过 ICMP echo request 完成。

然后检查一下 simple_switch_13 log 的输出。

controller: c0:

```
EVENT ofp_event->SimpleSwitch13 EventOFPPacketIn
packet in 1 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
EVENT ofp_event->SimpleSwitch13 EventOFPPacketIn
packet in 1 00:00:00:00:00:02 00:00:00:00:00:01 2
EVENT ofp_event->SimpleSwitch13 EventOFPPacketIn
packet in 1 00:00:00:00:00:01 00:00:00:00:00:02 1
```

第一个 Packet-In 是由 host 1 发送的 ARP request ，因为透过广播的方式所以没有 Flow Entry 存在，故发送 Packet-Out。

第二个是从 host 2 回复的 ARP reply，目的 MAC 地址为 host 1 因此前述的 Flow Entry (1) 被新增。

第三个是从 host 1 向 host 2 发送的 ICMP echo request，因此新增 Flow Entry (2)。

host 2 向 host 1 回复的 ICMP echo reply 则会和 Flow Entry (1) 发生 match，故直接转送封包至 host 1 而不需要发送 Packet-In。

最后让我们看看每一个 host 上的 tcpdump 所呈现的结果。

host: h1:

```
root@ryu-vm:~# tcpdump -en -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
20:38:04.625473 00:00:00:00:00:01 > ff:ff:ff:ff:ff:ff, ethertype ARP (0x0806),
length 42: Request who-has 10.0.0.2 tell 10.0.0.1, length 28
20:38:04.678698 00:00:00:00:00:02 > 00:00:00:00:00:01, ethertype ARP (0x0806),
length 42: Reply 10.0.0.2 is-at 00:00:00:00:00:02, length 28
20:38:04.678731 00:00:00:00:00:01 > 00:00:00:00:00:02, ethertype IPv4 (0x0800),
length 98: 10.0.0.1 > 10.0.0.2: ICMP echo request, id 3940, seq 1, length 64
20:38:04.722973 00:00:00:00:00:02 > 00:00:00:00:00:01, ethertype IPv4 (0x0800),
length 98: 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 3940, seq 1, length 64
```

host 1 首先发送广播 ARP request 封包，接着接收到 host 2 送来的 ARP reply 回复。接着 host 1 发送 ICMP echo request，host 2 则回复 ICMP echo reply 。

host: h2:

```
root@ryu-vm:~# tcpdump -en -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
20:38:04.637987 00:00:00:00:00:01 > ff:ff:ff:ff:ff:ff, ethertype ARP (0x0806),
length 42: Request who-has 10.0.0.2 tell 10.0.0.1, length 28
20:38:04.638059 00:00:00:00:00:02 > 00:00:00:00:00:01, ethertype ARP (0x0806),
length 42: Reply 10.0.0.2 is-at 00:00:00:00:00:02, length 28
20:38:04.722601 00:00:00:00:00:01 > 00:00:00:00:00:02, ethertype IPv4 (0x0800),
length 98: 10.0.0.1 > 10.0.0.2: ICMP echo request, id 3940, seq 1, length 64
20:38:04.722747 00:00:00:00:00:02 > 00:00:00:00:00:01, ethertype IPv4 (0x0800),
length 98: 10.0.0.2 > 10.0.0.1: ICMP echo reply, id 3940, seq 1, length 64
```

对于 host 2 则是接收 host 1 发送的 ARP request 封包，接着对 host 1 发送 ARP reply 回复。然后接收到 host 1 来的 ICMP echo request ，回复 host 1 echo reply。

host: h3:

```
root@ryu-vm:~# tcpdump -en -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
```

```
20:38:04.637954 00:00:00:00:00:01 > ff:ff:ff:ff:ff:ff, ethertype ARP (0x0806),
length 42: Request who-has 10.0.0.2 tell 10.0.0.1, length 28
```

对 host 3 而言，仅有一开始接收到 host 1 的广播 ARP request ，未做其他动作。

## 1.5 本章总结

本章以简单的交换器安装为题，透过 Ryu 应用程序的基本安装步骤和 OpenFlow 交换器的简单操作方法进行说明。

CHAPTER 2

# 流量监控（Traffic Monitor）

本章针对「交换器（Switching Hub）」提到的OpenFlow交换器加入流量监控的功能。

## 2.1 定期检查网络状态

网络已经成为许多服务或业务的基础建设，所以维护一个稳定的网络环境是必要的。但是网络问题总是不断地发生。

网络发生异常的时候，必须快速的找到原因，并且尽速恢复原状。这不需要多说，正在阅读本书的人都知道，找出网络的错误、发现真正的原因需要清楚地知道网络的状态。例如：假设网路中特定的端口正处于高流量的状态，不论是因为他是一个不正常的状态或是任何原因导 致，变成一个由于没有持续监控所发生的问题。

因此，为了网络的安全以及业务的正常运作，持续注意网络的健康状况是最基本的工作。当然， 网络流量的监视并不能够保证不会发生任何问题。本章将说明如何使用 OpenFlow 来取得相关的统计信息。

## 2.2 安装 Traffic Monitor

尝试改写代码看能不能加入错包率，丢包率以及可用带宽的监控

接着说明如何在「交换器（Switching Hub）」中提到的交换器中加入流量监控的功能。

```
from operator import attrgetter

from ryu.app import simple_switch_13
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER, DEAD_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.lib import hub


class  SimpleMonitor(simple_switch_13.SimpleSwitch13):

    def __init__(self, *args, **kwargs):
        super(SimpleMonitor, self).__init__(*args, **kwargs)
        self.datapaths = {}
        self.monitor_thread = hub.spawn(self._monitor)
    @set_ev_cls(ofp_event.EventOFPStateChange,
                [MAIN_DISPATCHER, DEAD_DISPATCHER])
    def _state_change_handler(self, ev):
        datapath = ev.datapath
        if ev.state == MAIN_DISPATCHER:
```

继承simpleswitch13，实现二层交换机的功能

开启一个线程，这个线程用来执行monitor方法，这个方法用来周期性的发送请求到交换机。而主线程继续执行其他的方法，如果不开启这个线程，那么monitor方法和其他方法就会冲突

其实对于程序而言，用handler，也就是需要监听的方法，是控制器对交换机的监听的方法需要用

交换机下线

交换机上线

```
            if not datapath.id in self.datapaths:
                self.logger.debug('register datapath: %016x', datapath.id)
                self.datapaths[datapath.id] = datapath
        elif ev.state == DEAD_DISPATCHER:
            if datapath.id in self.datapaths:
                self.logger.debug('unregister datapath: %016x', datapath.id)
                del self.datapaths[datapath.id]

    def _monitor(self):
        while True:
            for dp in self.datapaths.values():
                self._request_stats(dp)
            hub.sleep(10)

    def _request_stats(self, datapath):
        self.logger.debug('send stats request: %016x', datapath.id)
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        req = parser.OFPFlowStatsRequest(datapath)
        datapath.send_msg(req)

        req = parser.OFPPortStatsRequest(datapath, 0, ofproto.OFPP_ANY)
        datapath.send_msg(req)

    @set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
    def _flow_stats_reply_handler(self, ev):
        body = ev.msg.body

        self.logger.info('datapath         '
                         'in-port  eth-dst           '
                         'out-port packets  bytes')
        self.logger.info('---------------- '
                         '-------- ----------------- '
                         '-------- -------- --------')
        for stat in sorted([flow for flow in body if flow.priority == 1],
                           key=lambda flow: (flow.match['in_port'],
                                             flow.match['eth_dst'])):
            self.logger.info('%016x %8x %17s %8x %8d %8d',
                             ev.msg.datapath.id,
                             stat.match['in_port'], stat.match['eth_dst'],
                             stat.instructions[0].actions[0].port,
                             stat.packet_count, stat.byte_count)

    @set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
    def _port_stats_reply_handler(self, ev):
        body = ev.msg.body

        self.logger.info('datapath         port     '
                         'rx-pkts  rx-bytes rx-error '
                         'tx-pkts  tx-bytes tx-error')
        self.logger.info('---------------- -------- '
                         '-------- -------- -------- '
                         '-------- -------- --------')
        for stat in sorted(body, key=attrgetter('port_no')):
            self.logger.info('%016x %8x %8d %8d %8d %8d %8d %8d',
                             ev.msg.datapath.id, stat.port_no,
                             stat.rx_packets, stat.rx_bytes, stat.rx_errors,
                             stat.tx_packets, stat.tx_bytes, stat.tx_errors)
```

删除字典的一条记录

这个方法是对交换机进行请求，包括请求流表和端口信息。

版本信息

这个方法是对交换机返回的流表信息进行解析，控制器发送请求会送流表信息后，交换机就会给出流表信息，在这儿就需要对流表信息进行解析，也就是对信息输出显示出来

事实上，流量监控功能已经被实作在 SimpleMonitor 类别中并继承自 SimpleSwitch13 ，所以这边已经没有转送相关的处理功能了。

### 2.2.1 固定周期处理

透过 Switching Hub 的平行处理，建立一个线程并定期的向交换器发出要求以取得统计的资料。

```
from operator import attrgetter

from ryu.app import simple_switch_13
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER, DEAD_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.lib import hub

class SimpleMonitor(simple_switch_13.SimpleSwitch13):

    def __init__(self, *args, **kwargs):
        super(SimpleMonitor, self).__init__(*args, **kwargs)
        self.datapaths = {}
        self.monitor_thread = hub.spawn(self._monitor)

# ...
```

在 ryu.lib.hub 中实作了一些 eventlet wrapper 和基本的类别。这里我们使用 hub.spawn() 建立线程。但实际上是使用 eventlet 的 green 线程。

```
# ...

@set_ev_cls(ofp_event.EventOFPStateChange,
            [MAIN_DISPATCHER, DEAD_DISPATCHER])
def _state_change_handler(self, ev):
    datapath = ev.datapath
    if ev.state == MAIN_DISPATCHER:
        if not datapath.id in self.datapaths:
            self.logger.debug('register datapath: %016x', datapath.id)
            self.datapaths[datapath.id] = datapath
    elif ev.state == DEAD_DISPATCHER:
        if datapath.id in self.datapaths:
            self.logger.debug('unregister datapath: %016x', datapath.id)
            del self.datapaths[datapath.id]

def _monitor(self):
    while True:
        for dp in self.datapaths.values():
            self._request_stats(dp)
        hub.sleep(10)

# ...
```

周期性的向上线的交换机发送请求信息以取得统计信息

在线程中 _monitor() 方法确保了线程可以在每 10 秒的间隔中，不断地向注册的交换器 发送要求以取得统计信息。

为了确认联机中的交换器都可以被持续监控，EventOFPStateChange 就可以用来监测交换器的联机中断。这个事件侦测是 Ryu 框架所提供的功能，会被触发在 Datapath 的状态改变时。

当 Datapath 的状态变成 MAIN_DISPATCHER 时，代表交换器已经注册并正处于被监视的状态。而状态变成 DEAD_DISPATCHER 时代表已经从注册状态解除。

```
# ...

def _request_stats(self, datapath):
    self.logger.debug('send stats request: %016x', datapath.id)
    ofproto = datapath.ofproto
```

```
    parser = datapath.ofproto_parser

    req = parser.OFPFlowStatsRequest(datapath)
    datapath.send_msg(req)

    req = parser.OFPPortStatsRequest(datapath, 0, ofproto.OFPP_ANY)
    datapath.send_msg(req)

# ...
```

定期呼叫 `_request_stats()` 以驱动 `OFPFlowStatsRequest` 和 `OFPPortStatsRequest` 对交换器发出讯息。

`OFPFlowStatsRequest` 主要用来对交换器的 Flow Entry 取得统计的资料。对于交换器发出的要求可以使用 table ID、output port、cookie 值和 match 条件来限缩范围，但是这边的例子是取得所有的 Flow Entry。

`OFPPortStatsRequest` 是用来取得关于交换器的端口相关信息以及统计讯息。使用的时候可以指定端口号，这边使用 `OFPP_ANY` 目的是要取得所有的端口统计数据。

### 2.2.2 FlowStats

为了接收来自交换器的响应，建立一个 event handler 来接受从交换器发送的 FlowStatsReply 讯息。

```
# ...

@set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
def _flow_stats_reply_handler(self, ev):
    body = ev.msg.body

    self.logger.info('datapath         '
                     'in-port  eth-dst           '
                     'out-port packets  bytes')
    self.logger.info('---------------- '
                     '-------- ----------------- '
                     '-------- -------- --------')
    for stat in sorted([flow for flow in body if flow.priority == 1],
                       key=lambda flow: (flow.match['in_port'],
                                         flow.match['eth_dst'])):
        self.logger.info('%016x %8x %17s %8x %8d %8d',
                         ev.msg.datapath.id,
                         stat.match['in_port'], stat.match['eth_dst'],
                         stat.instructions[0].actions[0].port,
                         stat.packet_count, stat.byte_count)
# ...
```

`OPFFlowStatsReply` 类别的属性 `body` 是 `OFPFlowStats` 的列表，当中储存了每一个 Flow Entry 的统计信息，并做为 FlowStatsRequest 的响应。

权限为零的 Table-miss Flow 除外的全部 Flow Entry 将会被选择，通过并符合该 Flow Entry 的封包数和位数统计数据将会被回传，并以接收端口号和目的 MAC 地址的方式排序。

为了持续的收集以及分析，仅有一部份的数据会被输出到 log。因此若要进行分析，连结到外部的程序是必须的。在这样的情况下 `OFPFlowStatsReply` 的内容可以被转换成为 JSON 的格式进行输出。

可以设定如下格式

```
import json

# ...

self.logger.info('%s', json.dumps(ev.msg.to_jsondict(), ensure_ascii=True,
                                  indent=3, sort_keys=True))
```

上述的设定将会产生结果如下


```
{
   "OFPFlowStatsReply":
      { "body": [
         {
            "OFPFlowStats":
               { "byte_count":
               0,
               "cookie": 0,
               "duration_nsec": 680000000,
               "duration_sec": 4,
               "flags": 0,
               "hard_timeout": 0,
               "idle_timeout": 0,
               "instructions": [
                  {
                     "OFPInstructionActions":
                        { "actions": [
                           {
                              "OFPActionOutput":
                                 { "len": 16,
                                 "max_len": 65535,
                                 "port": 4294967293,
                                 "type": 0
                              }
                           }
                        ],
                        "len": 24,
                        "type": 4
                     }
                  }
               ],
               "length": 80,
               "match": {
                  "OFPMatch": {
                     " length ":  4,
                     "oxm_fields": [],
                     "type": 1
                  }
               },
               "packet_count": 0,
               "priority": 0,
               "table_id": 0
            }
         },
         {
            " OFPFlowStats ":
               { "byte_count":
               42,
               "cookie": 0,
               "duration_nsec": 72000000,
               "duration_sec": 57,
               "flags": 0,
               "hard_timeout": 0,
               "idle_timeout": 0,
```

```
                    "OFPInstructionActions":
                       { "actions": [
                          {
                             "OFPActionOutput":
                                { "len": 16,
                                "max_len": 65509,
                                "port": 1,
                                "type": 0
                             }
                          }
                       ],
                       "len": 24,
                       "type": 4
                    }
                 }
              ],
              "length": 96,
              "match": {
                 "OFPMatch": {
                    "length": 22,
                    "oxm_fields": [
                       {
                          "OXMTlv": {
                             "field": "in_port",
                             "mask": null,
                             "value": 2
                          }
                       },
                       {
                          "OXMTlv": {
                             "field": "eth_dst",
                             "mask": null,
                             "value": "00:00:00:00:00:01"
                          }
                       }
                    ],
                    "type": 1
                 }
              },
              "packet_count": 1,
              "priority": 1,
              "table_id": 0
           }
        }
     ],
     "flags": 0,
     "type": 1
  }
}
```


### 2.2.3 PortStats

为了接收交换器所回复的讯息，PortStatsReply 讯息接收的事件接收处理必须要被实作。

```
# ...

@set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
def _port_stats_reply_handler(self, ev):
    body = ev.msg.body

    self.logger.info('datapath         port     '
```

```
                    'rx-pkts  rx-bytes rx-error '
                    'tx-pkts  tx-bytes tx-error')
    self.logger.info('---------------- -------- '
                    '-------- -------- -------- '
                    '-------- -------- --------')
    for stat in sorted(body, key=attrgetter('port_no')):
        self.logger.info('%016x %8x %8d %8d %8d %8d %8d %8d',
                         ev.msg.datapath.id , stat.port_no ,
                         stat.rx_packets, stat.rx_bytes, stat.rx_errors,
                         stat.tx_packets, stat.tx_bytes, stat.tx_errors)
```

OPFPortStatsReply类别的属性body会列出在OFPPortStats中的数据列表。

OFPPortStats 端口号储存接收端的封包数量、位数量、丢弃封包数量、错误数量、frame 错误数量、overrrun 数量、CRC 错误数量、collection 数量等等的统计信息。

依据端口号的排序列出接收的封包数量、接收位数量、接收错误数量、发送封包数量、发送位数、发送错误数量。

## 2.3 执行 Traffic Monitor

接下来实际的执行流量监控。

首先，跟「交换器（SwitchingHub）」一样的执行Mininet。这边别忘了交换器的OpenFlow版本设定为 OpenFlow13。

下一步，执行流量监控程序。

controller: c0:

```
ryu@ryu-vm:~# ryu-manager --verbose ./simple_monitor.py
loading app ./simple_monitor.py
loading app ryu.controller.ofp_handler
instantiating app ./simple_monitor.py
instantiating app ryu.controller.ofp_handler
BRICK SimpleMonitor
  CONSUMES EventOFPStateChange
  CONSUMES EventOFPFlowStatsReply
  CONSUMES EventOFPPortStatsReply
  CONSUMES EventOFPPacketIn
  CONSUMES EventOFPSwitchFeatures
BRICK ofp_event
  PROVIDES EventOFPStateChange TO {'SimpleMonitor': set(['main', 'dead'])}
  PROVIDES EventOFPFlowStatsReply TO {'SimpleMonitor': set(['main'])}
  PROVIDES EventOFPPortStatsReply TO {'SimpleMonitor': set(['main'])}
  PROVIDES EventOFPPacketIn TO {'SimpleMonitor': set(['main'])}
  PROVIDES EventOFPSwitchFeatures TO {'SimpleMonitor': set(['config'])}
  CONSUMES EventOFPErrorMsg
  CONSUMES EventOFPPortDescStatsReply
  CONSUMES EventOFPHello
  CONSUMES EventOFPEchoRequest
  CONSUMES EventOFPSwitchFeatures
connected socket:<eventlet.greenio.GreenSocket object at 0x343fb10> address
:('127.0.0.1', 55598)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0x343fed0>
move onto config mode
EVENT ofp_event->SimpleMonitor EventOFPSwitchFeatures
switch features ev version: 0x4 msg_type 0x6 xid 0x7dd2dc58
OFPSwitchFeatures( auxiliary_id=0,capabilities=71,datapath_id=1,n_buffers=256,
n_tables=254)
move onto main mode
```

```
EVENT ofp_event->SimpleMonitor EventOFPStateChange
register datapath: 0000000000000001
send stats request: 0000000000000001
EVENT ofp_event->SimpleMonitor EventOFPFlowStatsReply
datapath         in-port  eth-dst           out-port packets  bytes
---------------- -------- ----------------- -------- -------- --------
EVENT ofp_event->SimpleMonitor EventOFPPortStatsReply
datapath         port     rx-pkts  rx-bytes rx-error tx-pkts  tx-bytes tx-error
---------------- -------- -------- -------- -------- -------- -------- --------
0000000000000001        1        0        0        0        0        0        0
0000000000000001        2        0        0        0        0        0        0
0000000000000001        3        0        0        0        0        0        0
0000000000000001  fffffffe        0        0        0        0        0        0
```

在「交换器（SwitchingHub ）」中，我们使用ryu-manager指令来设定SimpleSwitch13模块名称（ryu.app.simple_switch_13 ）。至于这边则自定 SimpleMonitor 的文件名（ ./simple_monitor.py ）。

在这个时候 Flow Entry 是空白的（Table-miss Flow Entry 没有被显示出来），每个端口的计数 器也都为零。

从 host 1 向 host 2 执行 ping 的指令。

host: h1:

```
root@ryu-vm:~# ping -c1 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_req=1 ttl=64 time=94.4 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/ avg/ max/ mdev = 94.489/94.489/94.489/0.000 ms
root@ryu -vm:~#
```

封包的转送、Flow Entry 的注册状况，统计资料开始有了变化。

controller: c0:

```
datapath         in-port  eth-dst           out-port packets  bytes
---------------- -------- ----------------- -------- -------- --------
0000000000000001        1 00:00:00:00:00:02        2        1       42
0000000000000001        2 00:00:00:00:00:01        1        2      140
datapath         port     rx-pkts  rx-bytes rx-error tx-pkts  tx-bytes tx-error
---------------- -------- -------- -------- -------- -------- -------- --------
0000000000000001        1        3      182        0        3      182        0
0000000000000001        2        3      182        0        3      182        0
0000000000000001        3        0        0        0        1       42        0
0000000000000001  fffffffe        0        0        0        1       42        0
```

Flow Entry 的统计信息中，在接收端口 1 的 Flow match 流量讯息中，1 个封包，42 个字节的信息被记录下来。接收埠 2 则是 2 个封包，140 个字节。

端口的统计信息，端口 1 的封包接收（rx-pkts ）数量为 3，接收字节（rx-bytes ）数量 为 102 bytes，端口 2 也是 3 个封包，182 字节。

Flow Entry 的统计信息和端口的统计信息是不可以混为一谈的，这是因为 Flow Entry 的统计信息是记录 match 的 Entry 所转送封包的统计资料。也就是说被 Table-miss 触发的 Packet-In 以及 Packet-Out 转送封包都不能算在统计资料内。

在这个案例中，host 1 最初的广播消息是 ARP request，而 host 2 回复了 host 1 的 ARP 讯息，host 1 对 host 2 发送了 echo request 总共 3 个封包，这些都是透过触发 Packet-Out 所传送的。因此端口的流量会远大于Flow Entry的流量。

## 2.4 本章总结

本章藉由取得统计资料做为题目，尝试说明下列事项。

- 产生 Ryu 应用程序线程的方法
- 取得 Datapath 状态的改变
- 取得 FlowStats 和 PortStats 信息的方法

CHAPTER 3

# REST API

本章说明如何新增 REST 于「交换器（Switching Hub）」提到的 switching hub 中。

## 3.1 整合 REST API

Ryu 本身就有提供 WSGI 对应的 Web 服务器。透过这个机制建立相关的 REST API 可以与其他系统或浏览器整合。

备注: WSGI 是 Python 提供用来链接网页应用程序和网页服务器的框架。

## 3.2 安装包含 REST API 的 Switching Hub

接下来让我们实际加入两个先前在「交换器（Switching Hub）」说明过的 API。

1. MAC 地址表取得 API

   取得 Switching hub 中储存的 MAC 地址表内容。成对的 MAC 地址和端口号将以 JSON 的数据形态回传。

2. MAC 地址表注册 API

   MAC 地址和端口号成对的新增进 MAC 地址表，同时加到交换器的 Flow Entry 中。

接下来我们来看看程序代码。

```
import json
import logging

from ryu.app import simple_switch_13
from webob import Response
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.app.wsgi import ControllerBase, WSGIApplication, route
from ryu.lib import dpid as dpid_lib

simple_switch_instance_name = 'simple_switch_api_app'
url = '/simpleswitch/mactable/{dpid}'

class  SimpleSwitchRest13(simple_switch_13.SimpleSwitch13):
```

```
    _CONTEXTS = { 'wsgi': WSGIApplication }

    def __init__(self, *args, **kwargs):
        super(SimpleSwitchRest13, self).__init__(*args, **kwargs)
        self.switches = {}
        wsgi = kwargs['wsgi']
        wsgi.register(SimpleSwitchController, {simple_switch_instance_name : self})

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures , CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        super(SimpleSwitchRest13 , self).switch_features_handler(ev)
        datapath = ev.msg.datapath
        self. switches[ datapath. id ] = datapath
        self.mac_to_port.setdefault(datapath.id, {})

    def set_mac_to_port(self, dpid, entry):
        mac_table = self.mac_to_port.setdefault(dpid, {})
        datapath = self.switches.get(dpid)

        entry_port = entry['port']
        entry_mac = entry['mac']

        if datapath is not None:
            parser = datapath.ofproto_parser
            if entry_port not in mac_table.values():

                for mac, port in mac_table.items():

                    # from known device to new device
                    actions = [parser.OFPActionOutput(entry_port)]
                    match = parser.OFPMatch(in_port=port, eth_dst=entry_mac)
                    self.add_flow(datapath, 1, match, actions)

                    # from new device to known device
                    actions = [parser.OFPActionOutput(port)]
                    match = parser.OFPMatch(in_port=entry_port, eth_dst=mac)
                    self.add_flow(datapath, 1, match, actions)

                mac_table.update({entry_mac : entry_port})
        return mac_table

class SimpleSwitchController (ControllerBase):

    def __init__(self, req, link, data, **config):
        super(SimpleSwitchController, self).__init__(req, link, data, **config)
        self.simpl_switch_spp = data[simple_switch_instance_name]

    @route('simpleswitch', url, methods=['GET'], requirements={'dpid': dpid_lib.
DPID_PATTERN })
    def list_mac_table(self, req, **kwargs):

        simple_switch = self.simpl_switch_spp
        dpid = dpid_lib.str_to_dpid(kwargs['dpid'])

        if dpid not in simple_switch.mac_to_port:
            return Response(status=404)

        mac_table = simple_switch.mac_to_port.get(dpid, {})
        body = json.dumps(mac_table)
        return Response(content_type='application/json', body=body)

    @route('simpleswitch', url, methods=['PUT'], requirements={'dpid': dpid_lib.
DPID_PATTERN })
```

```
    def put_mac_table(self, req, **kwargs):

        simple_switch = self.simpl_switch_spp
        dpid = dpid_lib.str_to_dpid(kwargs['dpid'])
        new_entry = eval(req.body)

        if dpid not in simple_switch.mac_to_port:
            return Response(status=404)

        try:
            mac_table = simple_switch.set_mac_to_port(dpid, new_entry)
            body = json.dumps(mac_table)
            return Response(content_type='application/json', body=body)
        except Exception as e:
            return Response(status=500)
```

simple_switch_rest_13.py 是用来定义两个类别。

前者是控制器类别 `SimpleSwitchController` ，其中定义收到 HTTP request 时所需要响应的相对方法。

后者是``SimpleSwitchRest13`` 的类别，用来扩充「交换器（Switching Hub）」让它得以更新 MAC 地址表.

由于在 `SimpleSwitchRest13` 中已经有加入 Flow Entry 的功能，因此 FeaturesReply 方法被覆写（overridden）并保留 datapath 物件。

## 3.3 安装 SimpleSwitchRest13 class

```
class SimpleSwitchRest13(simple_switch_13.SimpleSwitch13):

    _CONTEXTS = { 'wsgi': WSGIApplication }
...
```

类别变量 `_CONTEXT` 是用来制定Ryu中所支持的WSGI网页服务器所对应的类别。因此我们可以透过 `wsgi` Key来取得WSGI网页服务器的实体。

```
def __init__(self, *args, **kwargs):
    super(SimpleSwitchRest13, self).__init__(*args, **kwargs)
    self.switches = {}
    wsgi = kwargs['wsgi']
    wsgi.register(SimpleSwitchController, {simple_switch_instance_name : self})
...
```

建构子需要取得 `WSGIApplication` 的实体用来注册 Controller 类别。稍后会有更详细的说明。注册则可以使用 `register` 方法。在呼叫 `register` 方法的时候，dictionary object 会在名为 `simple_switch_api_app` 的 Key 中被传递。因此 Controller 的建构子就可以存取到 `simple_switch_api_app` 的实体。

```
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
    super(SimpleSwitchRest13, self).switch_features_handler(ev)
    datapath = ev.msg.datapath
    self.switches[datapath.id] = datapath
    self.mac_to_port.setdefault(datapath.id, {})
...
```

父类别 switch_features_handler 已经被覆写（overridden ）。这个方法会在 SwitchFeatures 事件发生时被触发，从事件对象 ev 取得 datapath 对象后存放至 ``switches`` 变量中。此时 MAC 地址的初始值将会设定为空白字典（empty dictionary ）形态。

```
def set_mac_to_port(self, dpid, entry):
    mac_table = self.mac_to_port.setdefault(dpid, {})
    datapath = self.switches.get(dpid)

    entry_port = entry['port']
    entry_mac = entry['mac']

    if datapath is not None:
        parser = datapath.ofproto_parser
        if entry_port not in mac_table.values():

            for mac, port in mac_table.items():

                # from known device to new device
                actions = [parser.OFPActionOutput(entry_port)]
                match = parser.OFPMatch(in_port=port, eth_dst=entry_mac)
                self.add_flow(datapath, 1, match, actions)

                # from new device to known device
                actions = [parser.OFPActionOutput(port)]
                match = parser.OFPMatch(in_port=entry_port, eth_dst=mac)
                self.add_flow(datapath, 1, match, actions)

            mac_table.update({entry_mac : entry_port})
    return mac_table
...
```

本方法用来注册 MAC 地址和端口号至指定的交换器。当 REST API 的 PUT 方法被触发时，本 方法就会被执行。

参数 entry 则是用来储存已经注册的 MAC 地址和链接端口的信息。

参照 MAC 地址表的 self.mac_to_port 信息，被注册到交换器的 Flow Entry 将被搜寻。

例如：一个成对的 MAC 地址和端口号将被登录在 MAC 地址表中。

- 00:00:00:00:00:01, 1

而且成对的 MAC 地址和端口号将被当作参数 entry 。

- 00:00:00:00:00:02, 2

最后被加入交换器当中的 Flow Entry 将会如下所示。

- match 条件：in_port = 1, dst_mac = 00:00:00:00:00:02 action：output=2
- match 条件：in_port = 2, dst_mac = 00:00:00:00:00:01 action：output=1

Flow Entry 的加入是透过父类别的 add_flow 方法达成。最后经由参数 entry 传递的讯息将会被储存在 MAC 地址表.

## 3.4 安装 SimpleSwitchController Class

接下来是控制器类别（controller class ）中 REST API 的 HTTP request 。类别名称是 ``SimpleSwitchController`` 。

```
class SimpleSwitchController (ControllerBase):
    def __init__(self, req, link, data, **config):
        super(SimpleSwitchController, self).__init__(req, link, data, **config)
        self.simpl_switch_spp = data[simple_switch_instance_name]
...
```

从建构子（constructor ）中取得 SimpleSwitchRest13 的实体。

```
@route('simpleswitch', url, methods=['GET'], requirements={'dpid': dpid_lib.
DPID_PATTERN })
def list_mac_table(self, req, **kwargs):

    simple_switch = self.simpl_switch_spp
    dpid = dpid_lib.str_to_dpid(kwargs['dpid'])

    if dpid not in simple_switch.mac_to_port:
        return Response(status=404)

    mac_table = simple_switch.mac_to_port.get(dpid, {})
    body = json.dumps(mac_table)
    return Response(content_type='application/json', body=body)
...
```

这部分是用来实作REST API的URL，还有其相对定的处理动作。为了结合URL和其对应的方法，route 这个装饰器将在 Ryu 中被使用。

被装饰器（Decorator ）处理的内容说明如下。

- 第一个参数

  任意的名称

- 第二个参数

  指定 URL。使得 URL 为 http://< 服务器 IP>:8080/simpleswitch/mactable/<datapath ID>

- 第三参数

  指定 HTTP 相对应的方法。指定 GET 相对应的方法。

- 第四参数

  指定URL的形式。URL(/simpleswitch/mactable/{dpid}) 中 {dpid} 的部分必须与 ryu/lib/dpid.py 中 DPID_PATTERN 16 个 16 进味的数字定义相吻合。

当 REST API 被第二参数所指定的 URL 呼叫时，相对的 HTTP GET list_mac_table 方法就会被触发。该方法将会取得 {dpid} 中储存的 data path ID 以得到 MAC 地址并转换成 JSON 的格式进行回传。

如果连结到 Ryu 的未知交换器 data path ID 被指定时，Ryu 会返回编码为 404 的回应。

```
@route('simpleswitch', url, methods=['PUT'], requirements={'dpid': dpid_lib.
DPID_PATTERN })
def put_mac_table(self, req, **kwargs):

    simple_switch = self.simpl_switch_spp
    dpid = dpid_lib.str_to_dpid(kwargs['dpid'])
    new_entry = eval(req.body)

    if dpid not in simple_switch.mac_to_port:
        return Response(status=404)
```

```
    try:
        mac_table = simple_switch.set_mac_to_port(dpid, new_entry)
        body = json.dumps(mac_table)
        return Response(content_type='application/json', body=body)
    except Exception as e:
        return Response(status=500)
...
```

然后是注册 MAC 地址表的 REST API。

URL 跟取得 MAC 地址表时的 API 相同，但是 HTTP 在 PUT 的情况下会呼叫 `put_mac_table` 方法。这个方法的内部会呼叫 switching hub 实体的 `set_mac_to_port` 方法。

当 `put_mac_table` 方法产生的例外的时候，回应码 500 将会被回传。同样的，`list_mac_table` 方法在 Ryu 所连接的交换器使用未知的 data path ID 的话，会回传响应码 404。

## 3.5 执行包含 REST API 的 Switching Hub

接着让我们执行已经加入 REST API 的 switching hub 吧。

首先执行「交换器（Switching Hub）」和 Mininet。这边不要忘了设定交换器的 OpenFlow 版本为 OpenFlow13。接着启动已经加入 REST API 的 switching hub。

```
ryu@ryu-vm:~/ryu/ryu/app$ cd ~/ryu/ryu/app
ryu@ryu-vm:~/ryu/ryu/app$ sudo ovs-vsctl set Bridge s1 protocols=OpenFlow13
ryu@ryu-vm:~/ryu/ryu/app$ ryu-manager --verbose ./simple_switch_rest_13.py
loading app ./simple_switch_rest_13.py
loading app ryu.controller.ofp_handler
creating context wsgi
instantiating app ryu.controller.ofp_handler
instantiating app ./simple_switch_rest_13.py
BRICK SimpleSwitchRest13
  CONSUMES EventOFPPacketIn
  CONSUMES EventOFPSwitchFeatures
BRICK ofp_event
  PROVIDES EventOFPPacketIn TO {' SimpleSwitchRest13 ': set([' main '])}
  PROVIDES EventOFPSwitchFeatures TO {'SimpleSwitchRest13': set(['config'])}
  CONSUMES EventOFPErrorMsg
  CONSUMES EventOFPPortDescStatsReply
  CONSUMES EventOFPEchoRequest
  CONSUMES EventOFPSwitchFeatures
  CONSUMES EventOFPHello
(31135) wsgi starting up on http://0.0.0.0:8080/
connected socket:<eventlet.greenio.GreenSocket object at 0x318c6d0> address
:('127.0.0.1', 48914)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0x318cc10>
move onto config mode
EVENT ofp_event->SimpleSwitchRest13 EventOFPSwitchFeatures
switch features ev version: 0x4 msg_type 0x6 xid 0x78dd7a72
OFPSwitchFeatures( auxiliary_id=0,capabilities=71,datapath_id=1,n_buffers
=256,n_tables=254)
```

上述启动时的讯息中有一行「(31135) wsgi starting up on http://0.0.0.0:8080/」，这是表示 网页服务器和埠号 8080 已经被启动。

接下来是在 mininet 的 shell 上从 h1 对 h2 执行 ping 的动作。

```
mininet> h1 ping -c 1 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
```

```
64 bytes from 10.0.0.2: icmp_req=1 ttl=64 time=84.1 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 84.171/84.171/84.171/0.000 ms
```

这时后会发生三次往 Ryu 方向的 Packet-In 。

```
EVENT ofp_event->SimpleSwitchRest13 EventOFPPacketIn
packet in 1 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
EVENT ofp_event->SimpleSwitchRest13 EventOFPPacketIn
packet in 1 00:00:00:00:00:02 00:00:00:00:00:01 2
EVENT ofp_event->SimpleSwitchRest13 EventOFPPacketIn
packet in 1 00:00:00:00:00:01 00:00:00:00:00:02 1
```

再来执行 REST API 以便在 switching hub 中取得 MAC 地址表。这次我们使用 curl 指令来驱动 REST API。

```
ryu@ryu-vm:~$ curl -X GET http://127.0.0.1:8080/simpleswitch/mactable
/0000000000000001
{"00:00:00:00:00:02": 2, "00:00:00:00:00:01": 1}
```

你会发现 h1 和 h2 的 MAC 地址表已经学习并更新完毕。

这次 h1 和 h2 的 MAC 地址表提前在执行 ping 之前被设定好。暂时停止 switching hub 和 mininet 的执行。然后再次启动 mininet 并在 OpenFlow 版本设定为 OpenFlow13 之后接着启动。

```
...
(26759) wsgi starting up on http://0.0.0.0:8080/
connected socket:<eventlet.greenio.GreenSocket object at 0x2afe6d0> address
:('127.0.0.1', 48818)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0x2afec10>
move onto config mode
EVENT ofp_event->SimpleSwitchRest13 EventOFPSwitchFeatures
switch features ev version: 0x4 msg_type 0x6 xid 0x96681337
OFPSwitchFeatures( auxiliary_id=0,capabilities=71,datapath_id=1,n_buffers
=256,n_tables=254) switch_features_handler inside sub class
move onto main mode
```

接着在每个 host 上呼叫 MAC 地址表更新的 REST API。REST API 呼叫的形式是{``mac'': ``MAC 地址'', ``port'' : 连接的端口号}

```
ryu@ryu-vm:~$ curl -X PUT -d '{"mac" : "00:00:00:00:00:01", "port" : 1}' http
://127.0.0.1:8080/ simpleswitch/ mactable /0000000000000001
{"00:00:00:00:00:01": 1}
ryu@ryu-vm:~$ curl -X PUT -d '{"mac" : "00:00:00:00:00:02", "port" : 2}' http
://127.0.0.1:8080/ simpleswitch/ mactable /0000000000000001
{"00:00:00:00:00:02": 2, "00:00:00:00:00:01": 1}
```

执行上述的指令，h1 和 h2 对应的 Flow Entry 会被加入交换器中。

然后从 h1 对 h2 执行 ping 指令。

```
mininet> h1 ping -c 1 h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_req=1 ttl=64 time=4.62 ms

--- 10.0.0.2 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 4.623/4.623/4.623/0.000 ms
```

```
...
move onto main mode
(28293) accepted ('127.0.0.1', 44453)
127.0.0.1 - - [19/Nov/2013 19:59:45] "PUT /simpleswitch/mactable/0000000000000001
HTTP/1.1" 200 124 0.002734
EVENT ofp_event->SimpleSwitchRest13 EventOFPPacketIn
packet in 1 00:00:00:00:00:01 ff:ff:ff:ff:ff:ff 1
```

这时候交换器中已经存在着 Flow Entry。Packet-In 只会发生在当 h1 到 h2 的 ARP 出现且没有接连发生的分组交换时。

## 3.6 本章总结

本章使用 MAC 地址表的处理作为题材，来说明如何新增 REST API。至于其他的练习应用，如 果可以做个从网页直接加入 Flow Entry 的 REST API 将会是一个很好的想法。

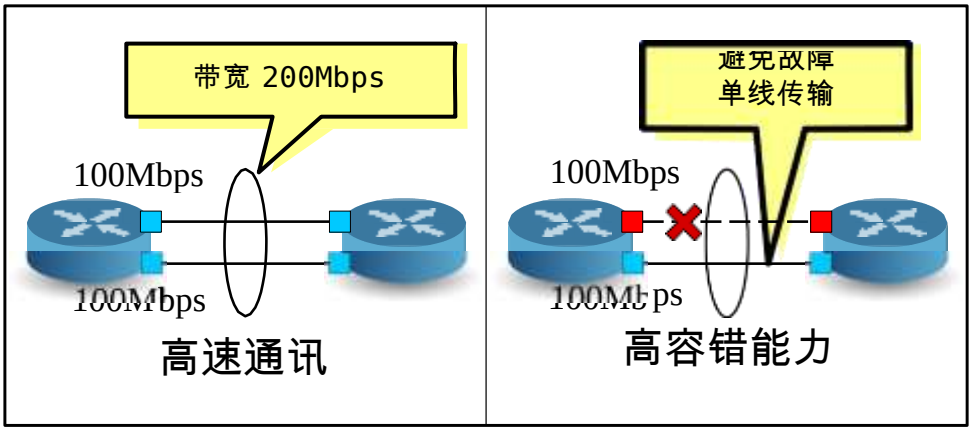
CHAPTER 4

# 网络聚合（Link Aggregation ）

本章会说明，Ryu 使用的网络聚合功能的实作方法。

## 4.1 网络聚合（Link Aggregation ）

网络聚合（Link Aggregation ）是由 IEEE802.1AX-2008 所制定的，多条实体线路合并为一条逻辑线路。透过本功能可以让网络中特定的装置间通讯速度提升、同时确保备援能力、提升容错的功能。




在使用网络聚合功能之前，个别的网络装置上界面归属于特定群组的关系都必须先设定完成。

起始网络聚合功能的方法是将个别的网络装置设置完成，此为静态方法。另外也可以使用 LACP（Link Aggregation Control Protocol ）通讯协议，此为动态方法。

采用动态方法的时候，每一个网络装置所相对应的界面会定期的进行 LACP data unit 交换以确认彼此之间的通讯状况。当 LACP data unit 的交换无法完成，代表网络已经出现故障，使用该网络的装置出现通讯中断，此时封包的传送仅能使用残存的界面和线路完成。

这样的做法有个优点，即当有个转送装置存在于网络之中，例如 meida converter，当该装置的 另外一端也断线时可以被侦测到。本章将会说明使用 LACP 进行动态网络聚合的设置。

## 4.2 执行 Ryu 应用程序

原始码的解说将放到后面，首先是执行 Ryu 的网络聚合应用程序。

simple_switch_lacp.py 为 OpenFlow 1.0 专用的应用程序并存在于 Ryu 的原始码中。在这边我们要建立新的 OpenFlow 1.3 版本，即 simple_switch_lacp_13.py。此应用程序可以为「交换器（Switching Hub）」中的交换器新增网络聚合功能。

原始码名称：simple_switch_lacp_13.py

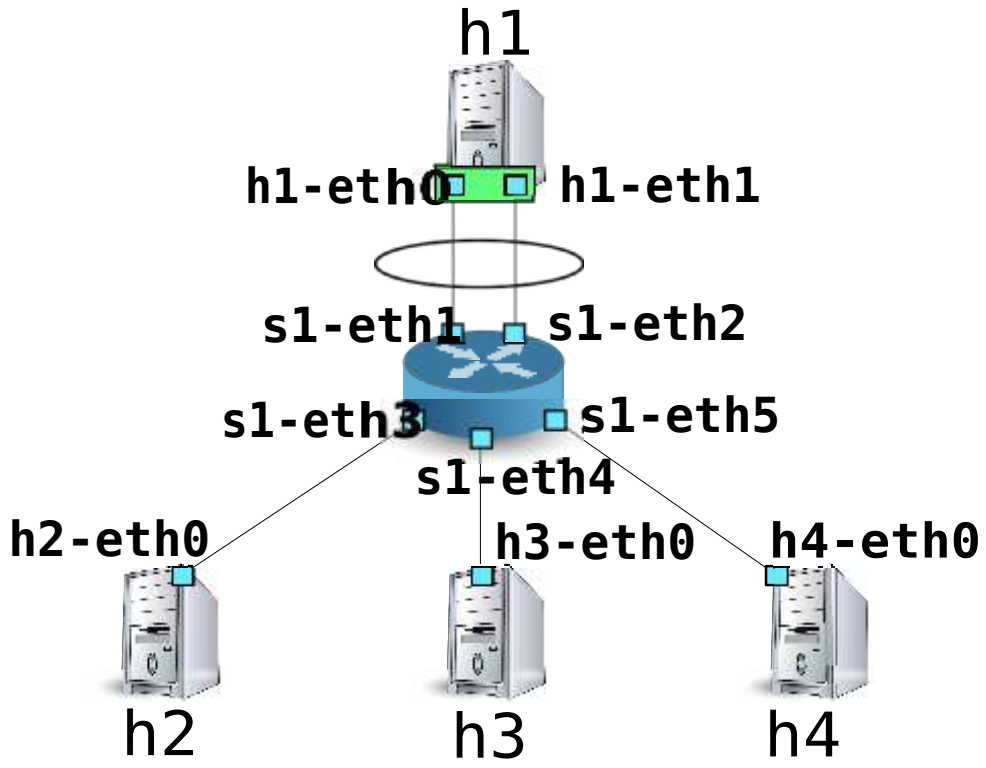
```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER
from ryu.controller.handler import MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib import lacplib
from ryu.lib.dpid import str_to_dpid
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet


class  SimpleSwitchLacp13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    _CONTEXTS = {'lacplib': lacplib.LacpLib}

    def __init__(self, *args, **kwargs):
        super(SimpleSwitchLacp13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        self._lacp = kwargs['lacplib']
        self._lacp.add(
            dpid=str_to_dpid('0000000000000001'), ports=[1, 2])

    @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.
        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):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

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

        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                match=match, instructions=inst)
        datapath.send_msg(mod)

    def del_flow(self, datapath, match):
        ofproto = datapath. ofproto
        parser = datapath.ofproto_parser

        mod = parser.OFPFlowMod(datapath=datapath,
```

```
                                command=ofproto.OFPFC_DELETE,
                                out_port=ofproto.OFPP_ANY,
                                out_group=ofproto.OFPG_ANY,
                                match=match)
        datapath.send_msg(mod)

    @set_ev_cls(lacplib.EventPacketIn,  MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        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)
            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)

    @set_ev_cls(lacplib.EventSlaveStateChanged,  MAIN_DISPATCHER)
    def _slave_state_changed_handler(self, ev):
        datapath = ev.datapath
        dpid = datapath. id
        port_no = ev. port
        enabled = ev.enabled
        self.logger.info("slave state changed port: %d enabled: %s",
                         port_no, enabled)
        if dpid in self.mac_to_port:
            for mac in self.mac_to_port[dpid]:
                match = datapath.ofproto_parser.OFPMatch(eth_dst=mac)
                self.del_flow(datapath, match)
            del self. mac_to_port[ dpid]
        self.mac_to_port.setdefault(dpid, {})
```

### 4.2.1 建置实验环境

让我们来看一下 OpenFlow 交换器和 Linux host 之间的网络聚合。

为了使用 VM 映像档，详细的环境设定和登入方法等请参考「交换器（Switching Hub）」。首先使用 Mininet 制作出如下一般的网络拓璞。




使用 script 来呼叫 Mininet 的 API 进而完成网络拓璞的建构。

原始码名称：`link_aggregation.py`

```
#!/usr/bin/env python

from mininet. cli import CLI
from mininet.link import Link
from mininet.net import Mininet
from mininet.node import RemoteController
from mininet.term import makeTerm

if '__main__' == __name__:
    net = Mininet(controller=RemoteController)

    c0 = net.addController('c0')

    s1 = net.addSwitch('s1')

    h1 = net.addHost('h1')
    h2 = net.addHost('h2', mac='00:00:00:00:00:22')
    h3 = net.addHost('h3', mac='00:00:00:00:00:23')
    h4 = net.addHost('h4', mac='00:00:00:00:00:24')

    Link(s1, h1)
    Link(s1, h1)
    Link(s1, h2)
    Link(s1, h3)
    Link(s1, h4)
```

```
    net. build ()
    c0 . start ()
    s1.start([c0])

    net.terms.append(makeTerm(c0))
    net.terms.append(makeTerm(s1))
    net.terms.append(makeTerm(h1))
    net.terms.append(makeTerm(h2))
    net.terms.append(makeTerm(h3))
    net.terms.append(makeTerm(h4))

    CLI( net)

    net.stop()
```

执行该 script 之后会形成 host 1 和交换器 s1 之间有两条联机的拓璞结构。这时后可以使用 net 命令来进行确认。

```
ryu@ryu-vm:~$ sudo ./link_aggregation.py
Unable to contact the remote controller at 127.0.0.1:6633
mininet>net
c0
s1 lo:   s1-eth1:h1-eth0 s1-eth2:h1-eth1 s1-eth3:h2-eth0 s1-eth4:h3-eth0 s1-eth5:h4-
eth0
h1 h1-eth0:s1-eth1 h1-eth1:s1-eth2
h2 h2-eth0:s1-eth3
h3 h3-eth0:s1-eth4
h4 h4-eth0:s1-eth5
mininet>
```

### 4.2.2 host h1 的网络聚合设定

在这之前在 host h1 的 Linux操作系统中必须先行设定。

请输入本节的命令在 host h1 的 xterm 终端机之中。

首先加载 drive module 以完成网络聚合。在 Linux 之中，网络聚合功能是由 bonding drive 所处理。预先建立 drive 的配置文件 /etc/modprobe.d/bonding.conf 以完成该功能。

文件名: `/etc/modprobe.d/bonding.conf`

```
alias bond0 bonding
options bonding mode=4
```

Node: h1:

```
root@ryu-vm:~# modprobe bonding
```

mode = 4 是 LACP 中代表使用动态网络聚合（dynamic link aggregation ），由于是默认值的关系这边可以省略。而 LACP data unit 的交换间隔为 SLOW（30 秒），并且排序的方式是使用目的 MAC 地址来进行。

接着建立一个名为 bond0 的逻辑界面，然后设定 bond0 的 MAC 地址。

Node: h1:

```
root@ryu-vm:~# ip link add bond0 type bond
root@ryu-vm:~# ip link set bond0 address 02:01:02:03:04:08
```

把 h1-eth0 和 h1-eth1 的实体网络界面加到已经建立好的逻辑界面群组中。此时先将实体界面设定为 down，然后随机数决定该实体界面成为比较简单的 MAC 地址之后更新它。

Node: h1:

```
root@ryu-vm:~# ip link set h1-eth0 down
root@ryu-vm:~# ip link set h1-eth0 address 00:00:00:00:00:11
root@ryu-vm:~# ip link set h1-eth0 master bond0
root@ryu-vm:~# ip link set h1-eth1 down
root@ryu-vm:~# ip link set h1-eth1 address 00:00:00:00:00:12
root@ryu-vm:~# ip link set h1-eth1 master bond0
```

指定逻辑界面的 IP 地址，这边指定为 10.0.0.1。由于 h1-eth0 的 IP 地址已经被自动指定所以我们删除它。

Node: h1:

```
root@ryu-vm:~# ip addr add 10.0.0.1/8 dev bond0
root@ryu-vm:~# ip addr del 10.0.0.1/8 dev h1-eth0
```

最后设定逻辑界面为 UP。

Node: h1:

```
root@ryu-vm:~# ip link set bond0 up
```

接着确认每一个界面的状态。

Node: h1:

```
root@ryu-vm:~# ifconfig
bond0     Link encap:Ethernet  HWaddr 02:01:02:03:04:08
          inet addr:10.0.0.1  Bcast:0.0.0.0  Mask:255.0.0.0
          UP BROADCAST RUNNING MASTER MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:1240 (1.2 KB)

h1-eth0   Link encap:Ethernet  HWaddr 02:01:02:03:04:08
          UP BROADCAST RUNNING SLAVE MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:5 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:620 (620.0 B)

h1-eth1   Link encap:Ethernet  HWaddr 02:01:02:03:04:08
          UP BROADCAST RUNNING SLAVE MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:5 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:620 (620.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

逻辑界面bond0为MASTER，实体界面h1-eth0和h1-eth1为SLAVE。而且你可以看到bond0、h1-eth0 和 h1-eth1 的 MAC 地址全部都是相同的。

确认 bonding driver 的状态。

Node: h1:

```
root@ryu-vm:~# cat /proc/net/bonding/bond0
Ethernet Channel Bonding Driver: v3.7.1 (April 27, 2011)

Bonding Mode: IEEE 802.3ad Dynamic link aggregation
Transmit Hash Policy: layer2 (0)
MII Status: up
MII Polling Interval (ms): 100
Up Delay (ms): 0
Down Delay (ms): 0

802.3 ad info
LACP rate: slow
Min links: 0
Aggregator selection policy (ad_select): stable
Active Aggregator Info:
        Aggregator ID: 1
        Number of ports: 1
        Actor Key: 33
        Partner Key: 1
        Partner Mac Address: 00:00:00:00:00:00

Slave Interface: h1-eth0
MII Status: up
Speed: 10000 Mbps
Duplex: full
Link Failure Count: 0
Permanent HW addr: 00:00:00:00:00:11
Aggregator ID: 1
Slave queue ID: 0

Slave Interface: h1-eth1
MII Status: up
Speed: 10000 Mbps
Duplex: full
Link Failure Count: 0
Permanent HW addr: 00:00:00:00:00:12
Aggregator ID: 2
Slave queue ID: 0
```

确认LACP data unit的交换间隔（LACP rate: slow）和排序逻辑的设定（Transmit Hash Policy: layer2 (0)）。并且确认实体界面 h1-eth0 和 h1-eth1 的 MAC 地址。

以上为 host h1 的事前准备。

### 4.2.3 设定 OpenFlow 版本

交换器 s1 的 OpenFlow 版本设定为 1.3。请在交换器 s1 的 xterm 终端机上输入下面的指令。

Node: s1:

```
root@ryu-vm:~# ovs-vsctl set Bridge s1 protocols=OpenFlow13
```

### 4.2.4 执行交换器

准备完成，接着开始执行Ryu应用程序。

在窗口标题为「Node: c0 (root)」的 xterm 终端机中执行下列的命令。

Node: c0:

```
ryu@ryu-vm:~$ ryu-manager ./simple_switch_lacp_13.py
loading app ./simple_switch_lacp_13.py
loading app ryu.controller.ofp_handler
creating context lacplib
instantiating app ./simple_switch_lacp_13.py
instantiating app ryu.controller.ofp_handler
...
```

host h1 会每 30 秒发送一次 LACP data unit。启动之后，交换器马上就会收到 host h1 发送的 LACP data unit，并输出记录在 log 之中。

Node: c0:

```
...
[LACP][INFO] SW=0000000000000001 PORT=1 LACP received.
[LACP][INFO] SW=0000000000000001 PORT=1 the slave i/f has just been up.
[LACP][INFO] SW=0000000000000001 PORT=1 the timeout time has changed.
[LACP][INFO] SW=0000000000000001 PORT=1 LACP sent.
slave state changed port: 1 enabled: True
[LACP][INFO] SW=0000000000000001 PORT=2 LACP received.
[LACP][INFO] SW=0000000000000001 PORT=2 the slave i/f has just been up.
[LACP][INFO] SW=0000000000000001 PORT=2 the timeout time has changed.
[LACP][INFO] SW=0000000000000001 PORT=2 LACP sent.
slave state changed port: 2 enabled: True
...
```

下面说明 log 的内容。

- LACP received.

  接受到 LACP data unit。

- the slave i/f has just been up. 原本无效的端口转换成有效状态。

- the timeout time has changed.

  LACP data unit 的逾时时间变更（本例子中原始状态为 0 秒，变更为 LONG_TIMEOUT_TIME 的 90 秒）

- LACP sent.

  回复用的 LACP data unit 传送完毕

- slave state changed ...

  应用程序接收到 LACP 函式库中 `EventSlaveStateChanged` 事件讯息（事件的详细内容稍后说明）。

交换器对于每一次从 host h1 收到的 LACP data unit 传送回复用的 LACP data unit 。

Node: c0:

```
...
[LACP][INFO] SW=0000000000000001 PORT=1 LACP received.
[LACP][INFO] SW=0000000000000001 PORT=1 LACP sent.
[LACP][INFO] SW=0000000000000001 PORT=2 LACP received.
[LACP][INFO] SW=0000000000000001 PORT=2 LACP sent.
...
```

确认 Flow Entry 。

Node: s1:

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=14.565s, table=0, n_packets=1, n_bytes=124, idle_timeout=90,
send_flow_rem priority=65535,in_port=2,dl_src=00:00:00:00:00:12,dl_type=0x8809
actions= CONTROLLER :65509
 cookie=0x0, duration=14.562s, table=0, n_packets=1, n_bytes=124, idle_timeout=90,
send_flow_rem priority=65535,in_port=1,dl_src=00:00:00:00:00:11,dl_type=0x8809
actions= CONTROLLER :65509
 cookie=0x0, duration=24.821s, table=0, n_packets=2, n_bytes=248, priority=0
actions= CONTROLLER :65535
```

在交换器中

- 接收到从 h1 的 h1-eth1（接收埠为 s1-eth2 、目的 MAC 地址为 00:00:00:00:00:12) 传送 LACP data unit（ethertype 0x8809 ）时就发送 Packet-In 讯息。
- 接收到从 h1 的 h1-eth0（接收埠为 s1-eth1 、目的 MAC 地址为 00:00:00:00:00:11) 传送 LACP data unit（ethertype 0x8809 ）时就发送 Packet-In 讯息。
- Table-miss Flow Entry 跟「交换器（Switching Hub ）」相同。 以

上 3 个 Flow Entry 被新增到交换器之中。

### 4.2.5 确认网络聚合功能

#### 改善传送速度

首先是确认网络聚合的传送速度。让我们来看看使用不同的线路来进行通讯的状况。 首先，

从 host h2 向 host h1 发送 ping 封包。

Node: h2:

```
root@ryu-vm:~# ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_req=1 ttl=64 time=93.0 ms
64 bytes from 10.0.0.1: icmp_req=2 ttl=64 time=0.266 ms
64 bytes from 10.0.0.1: icmp_req=3 ttl=64 time=0.075 ms
64 bytes from 10.0.0.1: icmp_req=4 ttl=64 time=0.065 ms
...
```

当 ping 讯息持续发送的同时，确认交换器 s1 的 Flow Entry 。

Node: s1:

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=22.05s, table=0, n_packets=1, n_bytes=124, idle_timeout=90,
send_flow_rem priority=65535,in_port=2,dl_src=00:00:00:00:00:12,dl_type=0x8809
actions= CONTROLLER :65509
 cookie=0x0, duration=22.046s, table=0, n_packets=1, n_bytes=124, idle_timeout=90,
send_flow_rem priority=65535,in_port=1,dl_src=00:00:00:00:00:11,dl_type=0x8809
actions= CONTROLLER :65509
 cookie=0x0, duration=33.046s, table=0, n_packets=6, n_bytes=472, priority=0
actions= CONTROLLER :65535
 cookie=0x0, duration=3.259s, table=0, n_packets=3, n_bytes=294, priority=1,in_port
=3,dl_dst=02:01:02:03:04:08  actions=output:1
```

```
 cookie=0x0, duration=3.262s, table=0, n_packets=4, n_bytes=392, priority=1,in_port
=1,dl_dst=00:00:00:00:00:22  actions=output:3
```

在确认时会新增下面两个Flow Entry。第四和第五Flow Entry会有较小的duration的值。

分别是

- 若是收到来自第3端口（s1-eth3，也就是连接到h2的界面）向h1的bond0发送的 封包就转送到第 1 端口（s1-eth1 ）。
- 若是收到来自第 1 端口（s1-eth1 ）向第 2 端口发送的封包，就从第 3 端口（s1-eth3 ）转送出去。

接着你可以看到 h2 和 h1 之间的通讯是使用 s1-eth1 来完成。

接着从 host h3 向 host h1 发送 ping 讯息。

Node: h3:

```
root@ryu-vm:~# ping 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_req=1 ttl=64 time=91.2 ms
64 bytes from 10.0.0.1: icmp_req=2 ttl=64 time=0.256 ms
64 bytes from 10.0.0.1: icmp_req=3 ttl=64 time=0.057 ms
64 bytes from 10.0.0.1: icmp_req=4 ttl=64 time=0.073 ms
...
```

在 ping 持续发送时，确认交换器 s1 的 Flow Entry。

Node: s1:

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=99.765s, table=0, n_packets=4, n_bytes=496, idle_timeout=90,
send_flow_rem priority=65535,in_port=2,dl_src=00:00:00:00:00:12,dl_type=0x8809
actions=CONTROLLER:65509
 cookie=0x0, duration=99.761s, table=0, n_packets=4, n_bytes=496, idle_timeout=90,
send_flow_rem priority=65535,in_port=1,dl_src=00:00:00:00:00:11,dl_type=0x8809
actions=CONTROLLER:65509
 cookie=0x0, duration=110.761s, table=0, n_packets=10, n_bytes=696, priority=0
actions=CONTROLLER:65535
 cookie=0x0, duration=80.974s, table=0, n_packets=82, n_bytes=7924, priority=1,
in_port=3,dl_dst=02:01:02:03:04:08  actions=output:1
 cookie=0x0, duration=2.677s, table=0, n_packets=2, n_bytes=196, priority=1,in_port
=2,dl_dst=00:00:00:00:00:23  actions=output:4
 cookie=0x0, duration=2.675s, table=0, n_packets=1, n_bytes=98, priority=1,in_port
=4,dl_dst=02:01:02:03:04:08  actions=output:2
 cookie=0x0, duration=80.977s, table=0, n_packets=83, n_bytes=8022, priority=1,
in_port=1,dl_dst=00:00:00:00:00:22  actions=output:3
```

在刚才确认的时候，2 个 Flow Entry 已经被新增。duration 值较小的 Flow Entry 为第 5 和第 6 个 Entry。

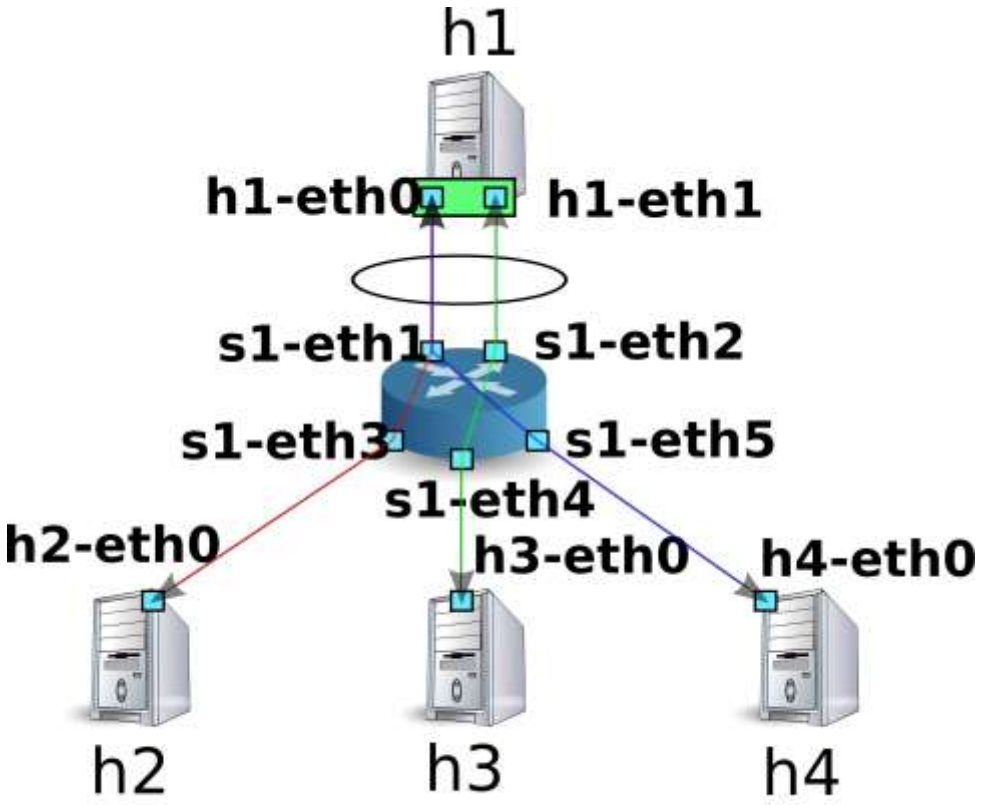
分别是

- 收到来自第2端口（s1-eth2 ）向h3发送的封包时就从第4端口（s1-eth4 ）转送 出去
- 收到来自第4端口（s1-eth4，也就是连接到h3的界面）向h1的bond0发送的封包 时就从第 2 端口（s1-eth2 ）转送出去

这样可以看出 h3 和 h1 之间的通讯是使用 s1-eth2 完成。

当然 host 4 向 host h1 发送的 ping 指令就可以正常动作。到此为止同样新的 Flow Entry 会被新增，h4 和 h1 之间的通讯就透过 s1-eth1 来传送。

| 目的 host | 使用端口 |
|---|---|
| h2 | 1 |
| h3 | 2 |
| h4 | 1 |




经过以上的动作，可以确认多条线路的通讯状态。

### 提升容错能力

接下来，我们要提升网络聚合的容错能力。现在的状况是 h2、h4 和 h1 之间的通讯是使用 s1-eth2 端口，h3 和 h1 的通讯是使用 s1-eth1 端口。

在这边我们把 s1-eth1 从所属的 s1-eth0 网络聚合群组中分离出来。

Node: h1:

```
root@ryu-vm:~# ip link set h1-eth0 nomaster
```

当 h1-eth0 停用时，host h3 向 host h1 的 ping 是无法连通的。在停止通讯的状态下经过了 90 秒之后，下面的 log 上会出现 Controller 的动作讯息。

Node: c0:

```
...
[LACP][INFO] SW=0000000000000001 PORT=1 LACP received.
[LACP][INFO] SW=0000000000000001 PORT=1 LACP sent.
[LACP][INFO] SW=0000000000000001 PORT=2 LACP received.
```

```
[LACP][INFO] SW=0000000000000001 PORT=2 LACP sent.
[LACP][INFO] SW=0000000000000001 PORT=2 LACP received.
[LACP][INFO] SW=0000000000000001 PORT=2 LACP sent.
[LACP][INFO] SW=0000000000000001 PORT=2 LACP received.
[LACP][INFO] SW=0000000000000001 PORT=2 LACP sent.
[LACP][INFO] SW=0000000000000001 PORT=1 LACP exchange timeout has occurred.
slave state changed port: 1 enabled: False
...
```

「LACP exchange timeout has occurred.」表示停止通讯已经逾时，曾经学习过的 MAC 地址和转送封包用的 Flow Entry 将全部被删除，回到如同刚开机时的初始状态。

当新的通讯发生时，新的 MAC 地址将会被学习，已存在的线路连结 Flow Entry 将会再次被新增。

host h3 和 host h1 之间的新 Flow Entry 会被新增。

Node: s1:

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=364.265s, table=0, n_packets=13, n_bytes=1612, idle_timeout
=90, send_flow_rem priority=65535,in_port=2,dl_src=00:00:00:00:00:12,dl_type=0x8809
 actions=CONTROLLER:65509
 cookie=0x0, duration=374.521s, table=0, n_packets=25, n_bytes=1830, priority=0
actions=CONTROLLER:65535
 cookie=0x0, duration=5.738s, table=0, n_packets=5, n_bytes=490, priority=1,in_port
=3,dl_dst=02:01:02:03:04:08 actions=output:2
 cookie=0x0, duration=6.279s, table=0, n_packets=5, n_bytes=490, priority=1,in_port
=2,dl_dst=00:00:00:00:00:23 actions=output:5
 cookie=0x0, duration=6.281s, table=0, n_packets=5, n_bytes=490, priority=1,in_port
=5,dl_dst=02:01:02:03:04:08 actions=output:2
 cookie=0x0, duration=5.506s, table=0, n_packets=5, n_bytes=434, priority=1,in_port
=4,dl_dst=02:01:02:03:04:08 actions=output:2
 cookie=0x0, duration=5.736s, table=0, n_packets=5, n_bytes=490, priority=1,in_port
=2,dl_dst=00:00:00:00:00:21 actions=output:3
 cookie=0x0, duration=6.504s, table=0, n_packets=6, n_bytes=532, priority=1,in_port
=2,dl_dst=00:00:00:00:00:22 actions=output:4
```

host h3 的 ping 恢复正常。

Node: h3:

```
...
64 bytes from 10.0.0.1: icmp_req=144 ttl=64 time=0.193 ms
64 bytes from 10.0.0.1: icmp_req=145 ttl=64 time=0.081 ms
64 bytes from 10.0.0.1: icmp_req=146 ttl=64 time=0.095 ms
64 bytes from 10.0.0.1: icmp_req=237 ttl=64 time=44.1 ms
64 bytes from 10.0.0.1: icmp_req=238 ttl=64 time=2.52 ms
64 bytes from 10.0.0.1: icmp_req=239 ttl=64 time=0.371 ms
64 bytes from 10.0.0.1: icmp_req=240 ttl=64 time=0.103 ms
64 bytes from 10.0.0.1: icmp_req=241 ttl=64 time=0.067 ms
...
```

经过以上的说明，当一部份的线路发生故障的时候，我们可以确认到其他的线路会自动的恢复以进行通讯。

## 4.3 实作 Ryu 的网络聚合功能

让我们来看看如何利用 OpenFlow 实作网络聚合功能。

使用 LCAP 的 link aggregation 的行为会像是「当 LACP data unit 传递正常时，则表示所使用的实体界面为有效状态」「反之，当 LACP data unit 传递不正常时，则表示所使用的实体界面为停用或无效状态」实体界面无效的意思是，该界面并无相关联的 Flow Entry 存在。因此透过下面的步骤：

- 实作接收并回复 LACP data unit
- 在一个固定的时间区间中没有收到 LACP data unit 时，则删除该实体界面所使用到及相关联的 Flow Entry
- 已经无效的实体界面若是收到了 LACP data unit 封包，则设定该界面为有效。
- LACP data unit 以外的封包则比照「交换器（Switching Hub）」进行学习及转送。

经过以上的处理及安装后，网络聚合基本的动作已经完成。LACP 相关的部分和不相关的部分 已经可以被明显的区隔出来，对于不相关的部分请参考「交换器（Switching Hub）」以进行交 换器的扩充实作。

在接收到 LACP data unit 时给予回复这件事情无法单纯使用 Flow Entry 来实现。因此 Packet-In 讯息被用来支持处理 OpenFlow Controller 方面的沟通。

备注: 用来交换 LACP data unit 的实体界面被分为两种角色，主动（ACTIVE）与被动（PASSIVE）形态。主动：在固定的时间会发送 LACP data unit ，以确认线路的状态。被动：在收到主动状态发送过来的 LACP data unit 时予以回复，以确认线路的状态。

在 Ryu 中的网络聚合应用程序是使用被动模式（PASSIVE mode ）来实作。

若是在特定的时间内没有收到 LACP data unit ，则该实体界面视为无效。会这样是因为先前 LACP data unit 促使 Packet-In 并进行 Flow Entry 新增时设定了 idle_timeout。当时间超过该设定值时，发送 FlowRemoved 讯息通知 Controller。Controller 则让该界面无效。

已经处于无效状态的界面收到 LACP data unit 的时候的处理为：接收到 LACP data unit 的时候 Packet-In 讯息的封包会做为该界面的有效或无效的判断标准，并进行必要的变更。

若实体界面为无效时，OpenFlow Controller 最简单的处理为「删除该界面所使用到的 Flow Entry」，但是这样的条件其实是不充分的。

例如：有一个逻辑界面是由三个实体界面群组化之后所产生，其排序为「依照使用界面的数量排序 MAC 地址」。

| 界面 1 | 界面 2 | 界面 3 |
|---|---|---|
| 剩余 MAC 地址:0 | 剩余 MAC 地址:1 | 剩余 MAC 地址:2 |

然后每一个实体界面所使用的 Flow Entry 分别新增如下。

| 界面 1 | 界面 2 | 界面 3 |
|---|---|---|
| 目的: 00:00:00:00:00:00 | 目的: 00:00:00:00:00:01 | 目的: 00:00:00:00:00:02 |
| 目的: 00:00:00:00:00:03 | 目的: 00:00:00:00:00:04 | 目的: 00:00:00:00:00:05 |
| 目的: 00:00:00:00:00:06 | 目的: 00:00:00:00:00:07 | 目的: 00:00:00:00:00:08 |

因此如果在界面 1 为停用的状态下，「在有效的界面中以剩余 MAC 地址数量为准」作为排序的标准，排序结果如下。

| 界面 1 | 界面 2 | 界面 3 |
|---|---|---|
| 无效 | 剩余 MAC 地址:0 | 剩余 MAC 地址:1 |

| 界面 1 | 界面 2 | 界面 3 |
| --- | --- | --- |
| | 目的: 00:00:00:00:00:00 | 目的: 00:00:00:00:00:01 |
| | 目的: 00:00:00:00:00:02 | 目的: 00:00:00:00:00:03 |
| | 目的: 00:00:00:00:00:04 | 目的: 00:00:00:00:00:05 |
| | 目的: 00:00:00:00:00:06 | 目的: 00:00:00:00:00:07 |
| | 目的: 00:00:00:00:00:08 | |

另外，没有仅使用单一界面 1 的 Flow Entry，在 Flow Entry 修改的同时，也会对界面 2 和界面 3 进行修改。这在实体界面变更为无效的时候需要做，变更为有效的时候同样也需要。

因此当实体界面在无效与有效之间改变时，该实体界面所属的逻辑界面所引用到的 Flow Entry 也会一并被进行删除。

备注: 排序的逻辑方法并没有在规格书中定义，由各家厂商自行实作。Ryu 的网络聚合应用程序是自行处理排序的部分，同时也使用相对应的机器所提供的路径排序。

接下来实作下面的功能。

LACP 函式库

- 当接收到 LACP data unit 讯息后，制作回复讯息并传送
- 当 LACP data unit 的接收中断后，相对应的实体界面则变更为无效，并通知交换器
- 当再次接收到 LACP data unit 时，相对应的实体界面则变更为有效，并通知交换器

交换器

- 接收到 LACP 函式库的通知之后，进行初始化以及删除必要的 Flow Entry
- 接收到 LACP data unit 以外的封包时，像平常一样学习并进行封包转送

LACP 函式库和其所使用的交换器原始码，都在 Ryu 的原始码中提供。

ryu/lib/lacplib.py

ryu/app/simple_switch_lacp.py

备注: 因为 simple_switch_lacp.py 为 OpenFlow 1.0 所专用的应用程序，本章所详细说明的是「执行 Ryu 应用程序」中 simple_switch_lacp_13.py 所对应的 OpenFlow 1.3 版本应用程序。

### 4.3.1 实作 LACP 函式库

在接下来的章节中，我们来看一下前述的 LACP 函式库功能如何实作。说明中所引用的原始码 为部分截取，若要了解全体的样貌，请参照实际的原始码。

#### 制作逻辑界面

若要使用网络聚合功能，必须要事先设定好哪一个网络是归属于哪一个群组。LACP 函式库使 用下列的方法来进行设定。

```
def add(self, dpid, ports):
    # ...
    assert isinstance(ports, list)
    assert 2 <= len(ports)
    ifs = {}
    for port in ports:
        ifs[port] = {'enabled': False, 'timeout': 0}
    bond = {}
    bond[ dpid] = ifs
    self._bonds.append(bond)
```

参数的内容说明如下。

dpid

指定 OpenFlow 交换器的 data path ID。

ports

指定要被群组的端口编号。

透过呼叫这个方法，LACP 函式库可以指定特定 data path ID 所属 OpenFlow 交换器端口成为特定的群组。若是要组成多个群组只要重复的呼叫 add() 方法即可。当逻辑界面被指定为特定的 MAC 地址时，OpenFlow 交换器所管理相同 MAC 地址的 LOCAL 端口会自动的被使用。

小诀窍: 有些 OpenFlow 交换器之中，交换器本身就有提供网络聚合的功能（Open vSwitch…等）在这边我们并不使用交换器本身的功能，而是使用 OpenFlow Controller 来实现网络聚合的功能。

### 处理 Packet-In

在「交换器（Switching Hub）」中，目的 MAC 地址在尚未被学习的状况下，会将接收到的封包进行 Flooding。由于 LACP data unit 仅会在相邻的网络间进行交换，若是将该封包转送至其他的网络时网络聚合将会出现异常。所以 Controller 就进行这样的处理「若是接收到 Packet-In 来自 LACP data unit 的封包就阻止，LACP data unit 以外的封包就交给交换器继续后续的动作」，在这样的操作下 LACP data unit 就会不会出现在交换器上。

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def packet_in_handler(self, evt):
    """PacketIn event handler. when the received packet was LACP,
    proceed it. otherwise, send a event."""
    req_pkt = packet.Packet(evt.msg.data)
    if slow.lacp in req_pkt:
        (req_lacp, ) = req_pkt.get_protocols(slow.lacp)
        (req_eth, ) = req_pkt.get_protocols(ethernet.ethernet)
        self._do_lacp(req_lacp, req_eth.src, evt.msg)
    else:
        self.send_event_to_observers(EventPacketIn(evt.msg))
```

事件管理（Eventhandler）跟在「交换器（SwitchingHub）」是相同的。会根据所收到的讯息是否为 LACP data unit 做为接下来处理的准则。

若是包含 LACP data unit 的状况下，就执行 LACP 函式库中 LACP data unit 的处理机制。在不包含 LACP data unit 的状况下，则呼叫 send_event_to_observers() 方法。这是定义在 ryu.base.app_manager.RyuApp 类别中，为了发送事件讯息的方法。

在「交换器（Switching Hub）」当中，我们提到了可以被使用者自行定义由 Ryu 所提供的 OpenFlow 讯息接收事件。上述的原始码当中有个事件叫做 `EventPacketIn` 即是由 LACP 函式库提供来让使用者自行定义。

```
class EventPacketIn( event.EventBase):
    """a PacketIn event class using except LACP."""
    def     __init__(    self    ,    msg):
        """          initialization."""
        super(EventPacketIn, self).__init__()
        self.msg = msg
```

使用者定义的事件是继承自 ryu.controller.event.EventBase 类别来达成。该类别对于资料的封装并没有限制。在 ``EventPacketIn`` 类别中，若是收到了 Packet-In 讯息就会用 ryu.ofproto.OFPPacketIn 实体来像往常一样处理。

使用者定义的事件接收方法将在之后提到。 端口

### 有效/无效状态的处理

LACP 函式库的 LACP data unit 处理如下：

1. 若接收到 LACP data unit 时，端口处于无效状态，这时候会进行状态的变更，并发送事件通知。
2. 若停止通讯的逾时设定值已经被改变时，收到 LACP data unit 封包后就发送 Packet-In 的 Flow Entry 将会再次被新增。
3. 接收 LACP data unit 后的回应、回复及传送。

第二点的处理会在稍后的「新增 Flow Entry - 收到 LACP data unit 时发送 Packet-In」描述，第三点的处理会在稍后的「传送/接收 LACP data unit 的处理」说明。接下来说明第一点的处理流程。

```
def _do_lacp(self, req_lacp, src, msg):
    # ...

    # when LACP arrived at disabled port, update the status of
    # the slave i/f to enabled, and send a event.
    if not self._get_slave_enabled(dpid, port):
        self.logger.info(
            "SW=%s PORT=%d the slave i/f has just been up.",
            dpid_to_str(dpid), port)
        self._set_slave_enabled(dpid, port, True)
        self.send_event_to_observers(
            EventSlaveStateChanged(datapath, port, True))
```

_get_slave_enabled() 方法是用来取得指定的交换器和指定的端口状态为有效或无效。

_set_slave_enabled() 方法是用来设定指定的交换器和指定的端口状态为有效或无效。

在上述的原始码中，当无效状态的端口接收到了 LACP data unit 时，端口的状态会进行改变并且呼叫 `EventSlaveStateChanged` 发送讯息，意即表示端口状态已经改变。

```
class  EventSlaveStateChanged(event.EventBase):
    """a event class that notifies the changes of the statuses of the
    slave i/fs."""
    def __init__( self , datapath , port , enabled):
        """                    initialization."""
        super(EventSlaveStateChanged,  self).__init__()
        self.datapath = datapath
        self. port = port
        self.enabled = enabled
```

除了有效事件之外，当无效发生时 EventSlaveStateChanged 事件也会被触发并发送。当连接埠无效时所需执行的动作在「接收 FlowRemoved 讯息时的处理 」中实作。

EventSlaveStateChanged 类别包含以下的信息

- 端口的发生状态变更时所在的 OpenFlow交换器
- 有效/无效的状态改变时的端口编号
- 改变后的状态

### 新增 Flow Entry - 收到 LACP data unit 时发送 Packet-In

LACP data unit 的传送间隔定义为 FAST（每隔 1 秒）和 SLOW（每隔 30 秒）2 种。在网络聚合的规格中，传送间隔的 3 倍时间若是无任何通讯发生时，则该界面将自该群组移除，不再使用于封包的传送。

LACP 函式库中透过设定 Flow Entry 来达到监视通讯的状态，即当接受到 LACP data unit 时触发 Packet-In 并进行 Flow Entry 设定，即为 3 倍传送间隔（SHORT_TIMEOUT_TIME 3 秒、LONG_TIMEOUT_TIME 90 秒）的 idle_timeout。

当传送间隔的设定值改变时，就必须要重新设定 idle_timeout 的时间，因此实作 LACP 函式库如下。

```
def _do_lacp(self, req_lacp, src, msg):
    # ...

    # set the idle_timeout time using the actor state of the
    # received packet.
    if req_lacp.LACP_STATE_SHORT_TIMEOUT == \
       req_lacp.actor_state_timeout:
        idle_timeout = req_lacp.SHORT_TIMEOUT_TIME
    else:
        idle_timeout = req_lacp.LONG_TIMEOUT_TIME

    # when the timeout time has changed, update the timeout time of
    # the slave i/f and re-enter a flow entry for the packet from
    # the slave i/f with idle_timeout.
    if idle_timeout != self._get_slave_timeout(dpid, port):
        self.logger.info(
            "SW=%s PORT=%d the timeout time has changed.",
            dpid_to_str(dpid), port)
        self._set_slave_timeout(dpid, port, idle_timeout)
        func = self._add_flow.get(ofproto.OFP_VERSION)
        assert func
        func(src, port, idle_timeout, datapath)

    # ...
```

_get_slave_timeout() 方法可以用来取得指定的交换器之指定端口的 idle_timeout 数值。
_set_slave_timeout() 方法可以用来设定指定的交换器之指定端口的 idle_timeout 数值。

由于初始状态和移除自网络聚合群组的情况下 idle_timeout 为 0，因此接收到新的 LACP data unit 时，传送间隔将会根据设定被新增到 Flow Entry 中。

依据所使用的 OpenFlow 版本不同 OFPFlowMod 类别建构子的参数也不尽相同，因此必须取得所对应版本 Flow Entry 的方法。以下是采用 OpenFlow 1.2 之后所使用的 Flow Entry 新增方法。

```
def _add_flow_v1_2(self, src, port, timeout, datapath):
    """enter a flow entry for the packet from the slave i/f
    with idle_timeout. for OpenFlow ver1.2 and ver1.3."""
```

```
    ofproto = datapath. ofproto
    parser = datapath.ofproto_parser

    match = parser.OFPMatch(
        in_port=port, eth_src=src, eth_type=ether.ETH_TYPE_SLOW)
    actions = [parser.OFPActionOutput(
        ofproto.OFPP_CONTROLLER, ofproto.OFPCML_MAX)]
    inst = [parser.OFPInstructionActions(
        ofproto. OFPIT_APPLY_ACTIONS, actions)]
    mod = parser.OFPFlowMod(
        datapath=datapath, command=ofproto.OFPFC_ADD,
        idle_timeout= timeout , priority =65535 ,
        flags=ofproto.OFPFF_SEND_FLOW_REM, match=match,
        instructions= inst)
    datapath.send_msg(mod)
```

上述的原始码为「接收到来自连接的界面所发送的LACP data unit时发送Packet-In」所设定的Flow Entry ，用于在最高的优先权情况下监视停止通讯的状态。

### 传送/接收 LACP data unit 的处理

接收到 LACP data unit 时，进行「端口有效/无效状态的处理 」或「新增 Flow Entry - 收到 LACP data unit 时发送 Packet-In 」处理以及回复用的 LACP data unit 产生及发送。

```
def _do_lacp(self, req_lacp, src, msg):
    # ...

    # create a response packet.
    res_pkt = self._create_response(datapath, port, req_lacp)

    # packet-out the response packet.
    out_port = ofproto.OFPP_IN_PORT
    actions = [parser.OFPActionOutput(out_port)]
    out = datapath.ofproto_parser.OFPPacketOut(
        datapath=datapath, buffer_id=ofproto.OFP_NO_BUFFER,
        data=res_pkt.data, in_port=port, actions=actions)
    datapath.send_msg(out)
```

上述的原始码中，_create_response() 方法是用来产生回复用的封包。其中使用 _create_lacp() 方法则是用来产生回复用的 LACP data unit 。已经完成的回复用封包则根据所接收的 LACP data unit 发送 Packet-Out。

LACP data unit 中发送端（Actor）的信息和接收端（Partner）的信息已经被设定完成。从接收到的 LACP data unit 可以得到发送端的信息，因此在制作回复用的封包时，可以设定在接收端。

```
def _create_lacp(self, datapath, port, req):
    """create a LACP packet."""
    actor_system = datapath.ports[datapath.ofproto.OFPP_LOCAL].hw_addr
    res = slow.lacp(
        # ...
        partner_system_priority=req.actor_system_priority,
        partner_system=  req.  actor_system  ,
        partner_key=  req.  actor_key  ,
        partner_port_priority = req. actor_port_priority ,
        partner_port= req. actor_port ,
        partner_state_activity=req.actor_state_activity,
        partner_state_timeout = req. actor_state_timeout ,
        partner_state_aggregation = req. actor_state_aggregation ,
        partner_state_synchronization = req. actor_state_synchronization ,
        partner_state_collecting = req. actor_state_collecting ,
```

```
        partner_state_distributing=req.actor_state_distributing,
        partner_state_defaulted=req.actor_state_defaulted,
        partner_state_expired = req. actor_state_expired ,
        collector_max_delay =0)
    self.logger.info("SW=%s PORT=%d LACP sent.",
                     dpid_to_str(datapath.id), port)
    self.logger.debug(str(res))
    return res
```

#### 接收 FlowRemoved 讯息时的处理

在指定的传送间隔时间中没有进行 LACP data unit 的收送时，OpenFlow 交换器会发送 FlowRemoved 方法通知 OpenFlow Controller。

```
@set_ev_cls(ofp_event.EventOFPFlowRemoved, MAIN_DISPATCHER)
def flow_removed_handler(self, evt):
    """FlowRemoved event handler. when the removed flow entry was
    for LACP, set the status of the slave i/f to disabled, and
    senda event."""
    msg = evt.msg
    datapath = msg. datapath
    ofproto = datapath.ofproto
    dpid = datapath.id
    match = msg.match
    if ofproto.OFP_VERSION == ofproto_v1_0.OFP_VERSION:
        port = match.in_port
        dl_type = match.dl_type
    else:
        port = match[' in_port ']
        dl_type = match['eth_type']
    if ether.ETH_TYPE_SLOW != dl_type:
        return
    self.logger.info(
        "SW=%s PORT=%d LACP exchange timeout has occurred.",
        dpid_to_str(dpid), port)
    self._set_slave_enabled(dpid, port, False)
    self._set_slave_timeout(dpid, port, 0)
    self.send_event_to_observers(
        EventSlaveStateChanged(datapath, port, False))
```

当接收到 FlowRemoved 方法后，就会使用 _set_slave_enabled() 方法设定端口的状态为无效，使用 _set_slave_timeout() 方法设定 idle_timeout 值为 0，使用 send_event_to_observers() 方法发送 `EventSlaveStateChanged` 讯息。

### 4.3.2 应用程序的实作

现在说明「执行 Ryu 应用程序」中所提到的 OpenFlow 1.3 对应的网络聚合应用程序（simple_switch_lacp_13.py）和「交换器（Switching Hub）」中的交换器差异点。

#### 设定「_CONTEXTS」

继承自 ryu.base.app_manager.RyuApp 的 Ryu 应用程序若要启动其他的应用程序的话，就必须 在「_CONTEXTS」目录中设定。在启动其他的应用程序时，会使用另外的线程。在这边我们 设定 LACP 函式库中的 LacpLib 类别名称为「lacplib」并放在「_CONTEXTS」当中。

```
from ryu.lib import lacplib

# ...

class  SimpleSwitchLacp13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    _CONTEXTS = {'lacplib': lacplib.LacpLib}

    # ...
```

被设定在「_CONTEXTS」中的应用程序可以从 init () 方法中的 kwargs 取得实体。

```
# ...

def __init__(self, *args, **kwargs):
    super(SimpleSwitchLacp13, self).__init__(*args, **kwargs)
    self.mac_to_port = {}
    self._lacp = kwargs['lacplib']

# ...
```

### 函式库的初始化设定

初始化设定在「_CONTEXTS」中的 LACP 函式库。透过执行 LACP 函式库提供的 add() 方法来完成初始设定。设定的内容如下：

| 名称 | 参数值 | 说明 |
|---|---|---|
| dpid | str_to_dpid(`0000000000000001') | data path ID |
| ports | [1, 2] | 群组化的端口列表 |

根据以上的设定，data path ID 为「0000000000000001」的 OpenFlow 交换器的端口 1 和端口 2 整合为一个网络聚合群组。

```
# ...

    self._lacp = kwargs['lacplib']
    self._lacp.add(
        dpid=str_to_dpid('0000000000000001'), ports=[1, 2])

# ...
```

### 使用者自行定义事件的接收方法

在`实作 LACP 函式库`_ 我们已经说明，在 LACP 函式库中处理发送 Packet-In 方法时，LACP data unit 不包含使用者定义的 `EventPacketIn`。使用者定义的事件管理是在 Ryu 当中提供 `ryu.controller.handler.set_ev_cls` 作为装饰子用来装饰事件管理。

```
@set_ev_cls(lacplib.EventPacketIn,  MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    msg = ev.msg
    datapath = msg. datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    in_port = msg.match['in_port']

    # ...
```

接着，当连接埠的状态变更为有效或无效时，会透过 LACP 函式库发送 EventSlaveStateChanged 事件，因此我们必须建立一个事件管理来处理。

```
@set_ev_cls(lacplib.EventSlaveStateChanged, lacplib.LAG_EV_DISPATCHER)
def _slave_state_changed_handler(self, ev):
    datapath = ev.datapath
    dpid = datapath.id
    port_no = ev.port
    enabled = ev.enabled
    self.logger.info("slave state changed port: %d enabled: %s",
                     port_no, enabled)
    if dpid in self.mac_to_port:
        for mac in self.mac_to_port[dpid]:
            match = datapath.ofproto_parser.OFPMatch(eth_dst=mac)
            self.del_flow(datapath, match)
        del self.mac_to_port[dpid]
    self.mac_to_port.setdefault(dpid, {})
```

在本节开始的时候就有提到，端口的有效/无效状态变更时，被逻辑界面所使用的实体界面会因为封包的通过而变更状态。为了这个原因，已经被记录的 Flow Entry 将会被全部删除。

```
def del_flow(self, datapath, match):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    mod = parser.OFPFlowMod(datapath=datapath,
                            command=ofproto.OFPFC_DELETE,
                            match=match)
    datapath.send_msg(mod)
```

透过 OFPFlowMod 进行实体界面 Flow Entry 的删除动作。

如上所述，一个拥有网络聚合功能的交换器可以透过提供网络聚合功能的函式库和使用该函式 库的应用程序来达成。

## 4.4 本章总结

本章使用网络聚合函式库做为题材说明下列的项目。

- 如何使用「_CONTEXTS」函式库
- 使用者自行定义当事件被触发时，事件的处理和方法

CHAPTER 5

# 生成树（Spanning Tree ）

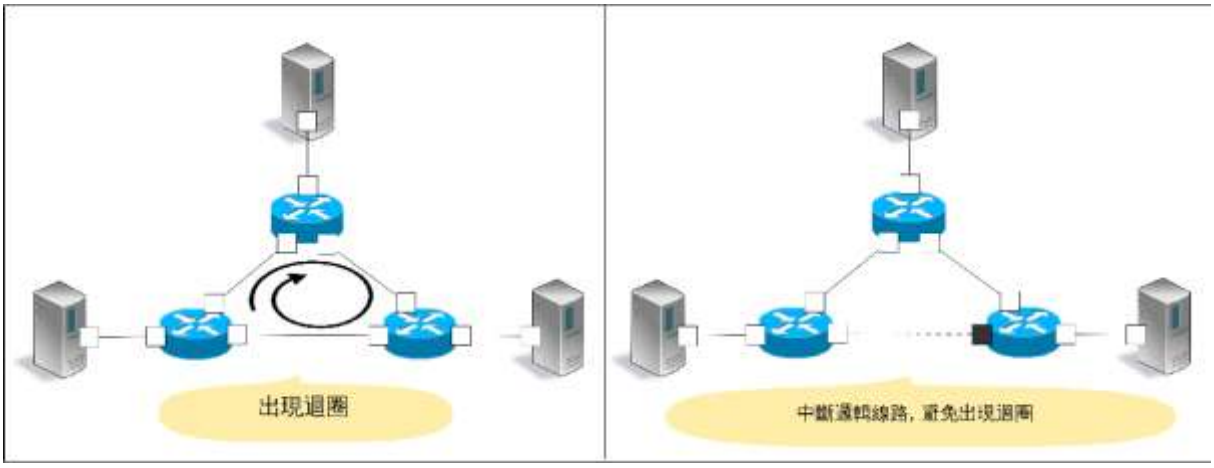
本章将说明与解说Ryu所采用的生成树（SpanningTree ）安装方法。

## 5.1 Spanning Tree

生成树是为了防止在网络的拓璞中出现循环（loop ）进而产生广播风暴（broadcast streams ）的技术。而且，借着应用原本防止循环的这项功能，当网络发生问题的时候，则可以达到确保网络的拓璞被重新计算的目的，如此一来就不会让部分的问题影响整个网络的连通。

生成树有许多的种类，例如：STP、RSTP、PVST+、MSTP... 等不同的种类。本章将说明最基本的 STP。

STP（spanning tree protocol：IEEE 802.1D ）是让网络的拓璞在逻辑上是树状的结构。经由设 定每一个交换器（本章节中有时候会使用网桥称呼）的端口让讯框（frame ）的传送为可 通过与不可通过，来防止循环的产生进而达到阻止网络风暴发生。




STP 会在网桥之间交换 BPDU（Bridge Protocol Data Unit ）封包，分析及比对网桥之间的 端口讯息，决定哪些端口可以传送哪些不行。

具体来说，会以下列的顺序完成。

1．Root 交换器（Root bridge ）选举

网桥之间的 BPDU 封包在交换过后，拥有最小的网桥 ID 即为 Root。接下来的 Root 网桥会发送 original BPDU ，而其他的网桥仅接收及转发 BPDU。

备注: 网桥 ID 的计算方式是组合已经被设定的网桥 priority 和特定端口的 MAC 地址而成。

网桥 ID

| Upper 2byte | Lower 6byte |
|---|---|
| 网桥 priority | MAC 地址 |

2．决定端口的角色

基于每一个端口到Root网桥的距离来决定该端口的角色。

- Root port

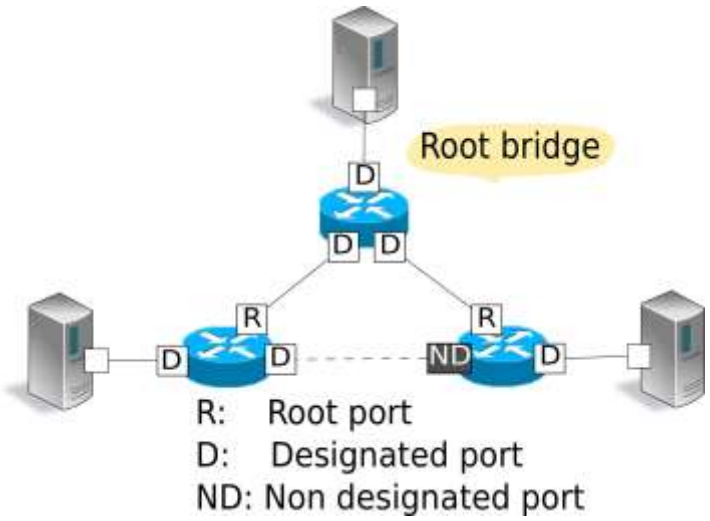
  网桥内连接到Root距离最短的端口。该端口将会接收来自Root 的BPDU。

- Designated port

  该端口从各个联机到 Root 网桥距离最短。主要转送来自 Root 桥 接器的 BPDU 封包，Root 网桥的所有端口均为此种类。

- Non designated port

  除了 Root port、designated port 以外的端口。讯框的传送是被禁止的。




备注: 每一个端口到 Root 网桥的距离是基于收到 BPDU 中的设定值，并加上下列的比较计算出来。

第一优先：比较 root path cost 的值

各网桥在转送 BPDU 的时候，会将封包中 root path cost 值加上设定的 path cost。因此 root path cost 值即是各个埠到 Root 网桥的总和。

第二优先：root path cost 相同的话，则比较网桥 ID

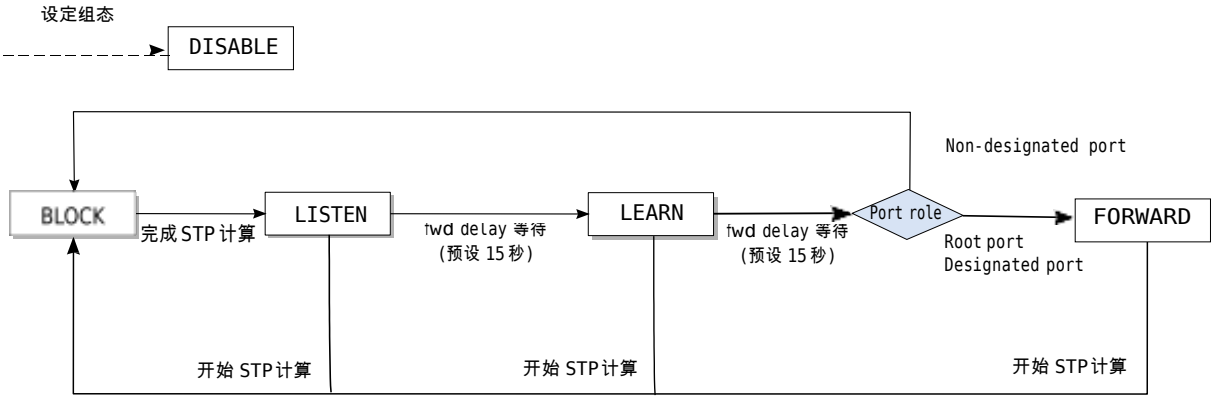
第三优先：若是网桥 ID 相同时（每一个埠都连接到相同的网桥），则比较端口 ID 端口 ID

| upper 2byte | lower 2byte |
|---|---|
| 端口优先权 | 端口编号 |

3．端口的状态变化

端口的角色决定了之后（STP 的处理完成），每一个端口会处于 LISTEN 状态。

之后会如下图进行状态的转换，最后每一个端口的角色会进入 FORWARD 状态或 者 BLOCK 状态。若是设定为无效的端口之后，就会进入 DISABLE 状态，接着不 会进行任何状态的转移。




当这些程序在每一台网桥开始执行之后，进行端口传送封包或抑制封包的决定，如此一来 便可以防止循环在网络拓璞中发生。

另外，断线或 BPDU 封包的最大时限（预设 20 秒）内未收到封包的故障侦测、新的端口加入导致网络拓璞改变。这些变化都会让每一台网桥执行上述 1, 2 和 3 程序以建立新的树状拓璞（STP re-calculation ）。

## 5.2 执行 Ryu 应用程序

执行Ryu生成树应用程序来达到OpenFlow实作的生成树功能。

Ryu 的原始码中 simple_switch_stp.py 是 OpenFlow 1.0 所使用，这边我们要使用新的 OpenFlow 1.3 对应的版本 simple_switch_stp_13.py 。这个应用程序新增加了生成树功能到「交换器（Switching Hub ）」中。

原始码档名：`simple_switch_stp_13.py`

```
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 import dpid as dpid_lib
from ryu.lib import stplib
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet


class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    _CONTEXTS = {'stplib': stplib.Stp}

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        self.stp = kwargs['stplib']

        # Sample of stplib config.
        #  please refer to stplib.Stp.set_config() for details.
        config = {dpid_lib.str_to_dpid('0000000000000001'):
                      {'bridge': {'priority': 0x8000}},
                  dpid_lib. str_to_dpid(' 0000000000000002 '):
```

```
                    {'bridge': {'priority': 0x9000}},
                dpid_lib.str_to_dpid('0000000000000003'):
                    {'bridge': {'priority': 0xa000}}}
        self.stp.set_config(config)

    @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.
        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):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

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

        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                match=match, instructions=inst)
        datapath.send_msg(mod)

    def delete_flow(self, datapath):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        for dst in self.mac_to_port[datapath.id].keys():
            match = parser.OFPMatch(eth_dst=dst)
            mod = parser.OFPFlowMod(
                datapath, command=ofproto.OFPFC_DELETE,
                out_port=ofproto.OFPP_ANY, out_group=ofproto.OFPG_ANY,
                priority=1, match=match)
            datapath.send_msg(mod)

    @set_ev_cls(stplib.EventPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        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)
            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)

    @set_ev_cls(stplib.EventTopologyChange, MAIN_DISPATCHER)
    def _topology_change_handler(self, ev):
        dp = ev.dp
        dpid_str = dpid_lib.dpid_to_str(dp.id)
        msg = 'Receive topology change event. Flush MAC table.'
        self.logger.debug("[dpid=%s] %s", dpid_str, msg)

        if dp.id in self.mac_to_port:
            self.delete_flow(dp)
            del self.mac_to_port[dp.id]

    @set_ev_cls(stplib.EventPortStateChange, MAIN_DISPATCHER)
    def _port_state_change_handler(self, ev):
        dpid_str = dpid_lib.dpid_to_str(ev.dp.id)
        of_state = {stplib.PORT_STATE_DISABLE: 'DISABLE',
                    stplib.PORT_STATE_BLOCK: 'BLOCK',
                    stplib.PORT_STATE_LISTEN: 'LISTEN',
                    stplib.PORT_STATE_LEARN: 'LEARN',
                    stplib.PORT_STATE_FORWARD: 'FORWARD'}
        self.logger.debug("[dpid=%s][port=%d] state=%s",
                          dpid_str, ev.port_no, of_state[ev.port_state])
```

### 5.2.1 建置实验环境

接下来确认生成树应用程序的执行动作以完成环境建置。

VM 映像档的使用、环境设定和登入方法等请参照「交换器（Switching Hub）」。

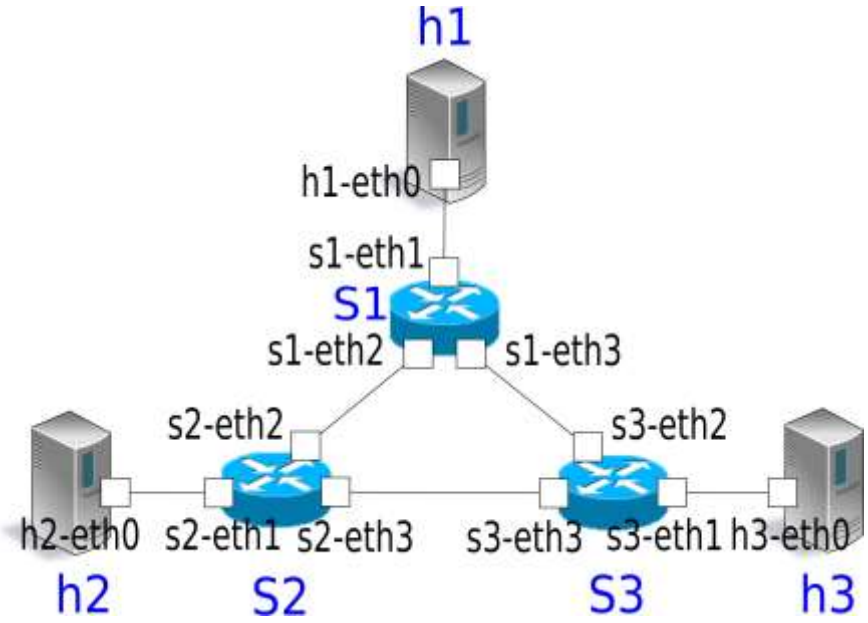
为了使用特殊含有循环的环境，请参考「网络聚合（Link Aggregation）」并使用script进行同样的网络拓璞建置一个 mininet 的环境。

原始码名称：spanning_tree.py

```
#!/usr/bin/env python

from mininet.cli import CLI
```

```
from mininet.link import Link
from mininet.net import Mininet
from mininet.node import RemoteController
from mininet.term import makeTerm

if '__main__' == __name__:
    net = Mininet(controller=RemoteController)

    c0 = net.addController('c0')

    s1 = net.addSwitch('s1')
    s2 = net.addSwitch('s2')
    s3 = net.addSwitch('s3')

    h1 = net.addHost('h1')
    h2 = net.addHost('h2')
    h3 = net.addHost('h3')

    Link(s1, h1)
    Link(s2, h2)
    Link(s3, h3)

    Link(s1, s2)
    Link(s2, s3)
    Link(s3, s1)


    net. build ()
    c0 . start ()
    s1.start([c0])
    s2.start([c0])
    s3.start([c0])

    net.terms.append(makeTerm(c0))
    net.terms.append(makeTerm(s1))
    net.terms.append(makeTerm(s2))
    net.terms.append(makeTerm(s3))
    net.terms.append(makeTerm(h1))
    net.terms.append(makeTerm(h2))
    net.terms.append(makeTerm(h3))

    CLI( net)

    net.stop()
```

在 VM 的环境中执行该程序，交换器 s1 、s2、s3 之间会出现循环。




net 命令执行结果如下。

```
ryu@ryu-vm:~$ sudo ./spanning_tree.py
Unable to contact the remote controller at 127.0.0.1:6633
mininet>net
c0
s1 lo:  s1-eth1:h1-eth0 s1-eth2:s2-eth2 s1-eth3:s3-eth3
s2 lo:  s2-eth1:h2-eth0 s2-eth2:s1-eth2 s2-eth3:s3-eth2
s3 lo:  s3-eth1:h3-eth0 s3-eth2:s2-eth3 s3-eth3:s1-eth3
h1 h1-eth0:s1-eth1
h2 h2-eth0:s2-eth1
h3 h3-eth0:s3-eth1
```

### 5.2.2 设定 OpenFlow 版本

设定 OpenFlow 的版本为 1.3。在 xterm 终端机 s1, s2, s3上使用命令输入。

Node: s1:

```
root@ryu-vm:~# ovs-vsctl set Bridge s1 protocols=OpenFlow13
```

Node: s2:

```
root@ryu-vm:~# ovs-vsctl set Bridge s2 protocols=OpenFlow13
```

Node: s3:

```
root@ryu-vm:~# ovs-vsctl set Bridge s3 protocols=OpenFlow13
```

### 5.2.3 执行 switching hub

准备已经完成，接下来执行 Ryu 应用程序。在窗口标题为「Node: c0 (root)」的 xterm执行下述的命令。

Node: c0:

```
root@ryu-vm:~$ ryu-manager ./simple_switch_stp_13.py
loading app simple_switch_stp_13.py
loading app ryu.controller.ofp_handler
loading app ryu.controller.ofp_handler
instantiating app None of Stp
creating context stplib
instantiating app simple_switch_stp_13.py of SimpleSwitch13
instantiating app ryu.controller.ofp_handler of OFPHandler
```

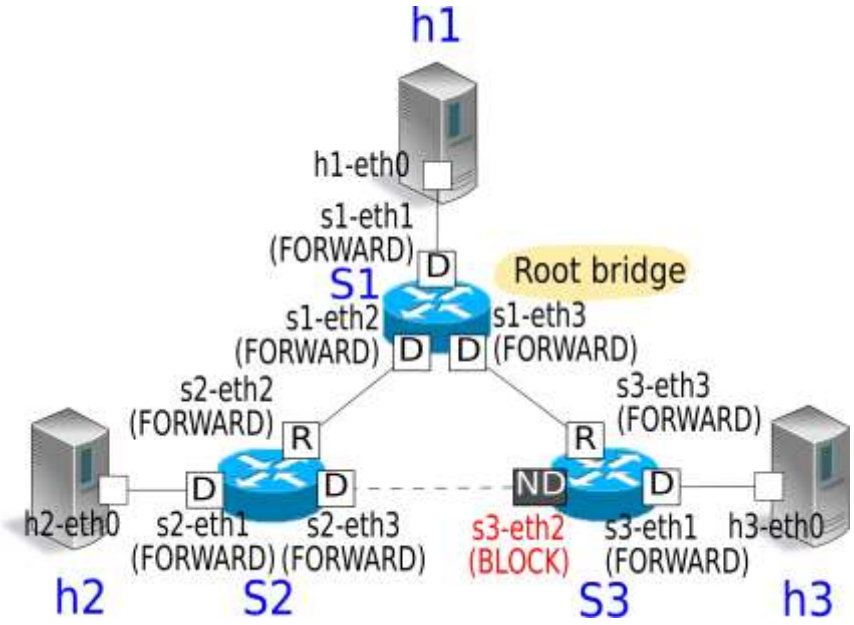
### OpenFlow 交换器启动时的 STP 计算

每一台 OpenFlow 交换器和 Controller 的连接完成后，BPDU 封包的交换就开始了。包括 Root 网桥的选举、端口的角色、端口的状态转移。

```
[STP][INFO] dpid=0000000000000001: Join as stp bridge.
[STP][INFO] dpid=0000000000000001: [port=1] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000001: [port=3] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000002: Join as stp bridge.
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000002: [port=2] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000001: [port=2] Receive superior BPDU.
[STP][INFO] dpid=0000000000000001: [port=1] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000001: [port=3] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000001: Root bridge.
[STP][INFO] dpid=0000000000000001: [port=1] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000001: [port=3] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000002: [port=2] Receive superior BPDU.
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000002: [port=2] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000002: Non root bridge.
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000002: [port=2] ROOT_PORT         / LISTEN
[STP][INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000003: Join as stp bridge.
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000003: [port=2] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000003: [port=3] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000002: [port=3] Receive superior BPDU.
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000002: [port=2] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000002: Non root bridge.
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000002: [port=2] ROOT_PORT         / LISTEN
[STP][INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000001: [port=3] Receive superior BPDU.
[STP][INFO] dpid=0000000000000001: [port=1] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000001: [port=3] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000001: Root bridge.
[STP][INFO] dpid=0000000000000001: [port=1] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000001: [port=3] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000003: [port=2] Receive superior BPDU.
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT      / BLOCK
```

```
[STP][INFO] dpid=0000000000000003: [port=2] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000003: [port=3] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000003: Non root bridge.
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000003: [port=2] ROOT_PORT         / LISTEN
[STP][INFO] dpid=0000000000000003: [port=3] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000003: [port=3] Receive superior BPDU.
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000003: [port=2] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000003: [port=3] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000003: Non root bridge.
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000003: [port=2] NON_DESIGNATED_PORT / LISTEN
[STP][INFO] dpid=0000000000000003: [port=3] ROOT_PORT         / LISTEN
[STP][INFO] dpid=0000000000000001: [port=3] Receive superior BPDU.
[STP][INFO] dpid=0000000000000001: [port=1] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000001: [port=3] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000001: Root bridge.
[STP][INFO] dpid=0000000000000001: [port=1] DESIGNATED_PORT      / LISTEN
[STP][INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT      / LISTEN
[STP][INFO] dpid=0000000000000001: [port=3] DESIGNATED_PORT      / LISTEN
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT      / LEARN
[STP][INFO] dpid=0000000000000002: [port=2] ROOT_PORT            / LEARN
[STP][INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT      / LEARN
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT      / LEARN
[STP][INFO] dpid=0000000000000003: [port=2] NON_DESIGNATED_PORT / LEARN
[STP][INFO] dpid=0000000000000003: [port=3] ROOT_PORT            / LEARN
[STP][INFO] dpid=0000000000000001: [port=1] DESIGNATED_PORT      / LEARN
[STP][INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT      / LEARN
[STP][INFO] dpid=0000000000000001: [port=3] DESIGNATED_PORT      / LEARN
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT      / FORWARD
[STP][INFO] dpid=0000000000000002: [port=2] ROOT_PORT            / FORWARD
[STP][INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT      / FORWARD
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT      / FORWARD
[STP][INFO] dpid=0000000000000003: [port=2] NON_DESIGNATED_PORT / BLOCK
[STP][INFO] dpid=0000000000000003: [port=3] ROOT_PORT            / FORWARD
[STP][INFO] dpid=0000000000000001: [port=1] DESIGNATED_PORT      / FORWARD
[STP][INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT      / FORWARD
[STP][INFO] dpid=0000000000000001: [port=3] DESIGNATED_PORT      / FORWARD
```

以上的结果，最后每一个端口分别为 FORWARD 状态或 BLOCK 状态。




为了确认封包不会产生循环现象，从 host 1 向 host 2 发送 ping 指令。

在 ping 命令执行之前，先执行 tcpdump 命令以确认封包的接收状况。

Node: s1:

```
root@ryu-vm:~# tcpdump -i s1-eth2 arp
```

Node: s2:

```
root@ryu-vm:~# tcpdump -i s2-eth2 arp
```

Node: s3:

```
root@ryu-vm:~# tcpdump -i s3-eth2 arp
```

在使用 script 进行网络拓璞的建构的终端机中，进行接下来的指令，从 host 1 向 host 2 发送 ping 封包。

```
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_req=1 ttl=64 time=84.4 ms
64 bytes from 10.0.0.2: icmp_req=2 ttl=64 time=0.657 ms
64 bytes from 10.0.0.2: icmp_req=3 ttl=64 time=0.074 ms
64 bytes from 10.0.0.2: icmp_req=4 ttl=64 time=0.076 ms
64 bytes from 10.0.0.2: icmp_req=5 ttl=64 time=0.054 ms
64 bytes from 10.0.0.2: icmp_req=6 ttl=64 time=0.053 ms
64 bytes from 10.0.0.2: icmp_req=7 ttl=64 time=0.041 ms
64 bytes from 10.0.0.2: icmp_req=8 ttl=64 time=0.049 ms
64 bytes from 10.0.0.2: icmp_req=9 ttl=64 time=0.074 ms
64 bytes from 10.0.0.2: icmp_req=10 ttl=64 time=0.073 ms
64 bytes from 10.0.0.2: icmp_req=11 ttl=64 time=0.068 ms
^C
--- 10.0.0.2 ping statistics ---
11 packets transmitted, 11 received, 0% packet loss, time 9998ms
rtt min/avg/max/mdev = 0.041/7.784/84.407/24.230 ms
```

从 tcpdump 的结果看来，ARP 并没有出现循环的状态已被确认。

Node: s1:

```
root@ryu-vm:~# tcpdump -i s1-eth2 arp
tcpdump: WARNING: s1-eth2: no IPv4 address assigned
tcpdump: verbose output suppressed , use -v or - vv for full protocol decode
listening on s1-eth2, link-type EN10MB (Ethernet), capture size 65535 bytes
11:30:24.692797 ARP , Request who - has 10.0.0.2 tell 10.0.0.1 , length 28
11:30:24.749153 ARP, Reply 10.0.0.2 is-at 82:c9:d7:e9:b7:52 (oui Unknown), length
28
11:30:29.797665 ARP , Request who - has 10.0.0.1 tell 10.0.0.2 , length 28
11:30:29.798250 ARP, Reply 10.0.0.1 is-at c2:a4:54:83:43:fa (oui Unknown), length
28
```

Node: s2:

```
root@ryu-vm:~# tcpdump -i s2-eth2 arp
tcpdump: WARNING: s2-eth2: no IPv4 address assigned
tcpdump: verbose output suppressed , use -v or - vv for full protocol decode
listening on s2-eth2, link-type EN10MB (Ethernet), capture size 65535 bytes
11:30:24.692824 ARP , Request who - has 10.0.0.2 tell 10.0.0.1 , length 28
11:30:24.749116 ARP, Reply 10.0.0.2 is-at 82:c9:d7:e9:b7:52 (oui Unknown), length
28
11:30:29.797659 ARP , Request who - has 10.0.0.1 tell 10.0.0.2 , length 28
11:30:29.798254 ARP, Reply 10.0.0.1 is-at c2:a4:54:83:43:fa (oui Unknown), length
28
```

Node: s3:

```
root@ryu-vm:~# tcpdump -i s3-eth2 arp
tcpdump: WARNING: s3-eth2: no IPv4 address assigned
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on s3-eth2, link-type EN10MB (Ethernet), capture size 65535 bytes
11:30:24.698477 ARP, Request who-has 10.0.0.2 tell 10.0.0.1, length 28
```

### 网络发现故障时重新计算 STP

接下来，确认断线发生的时候会进行STP的重新计算。在每一个OpenFlow交换器启动之后以及 STP 的计算完成之后，执行下列指令后让线路中断。

Node: s2:

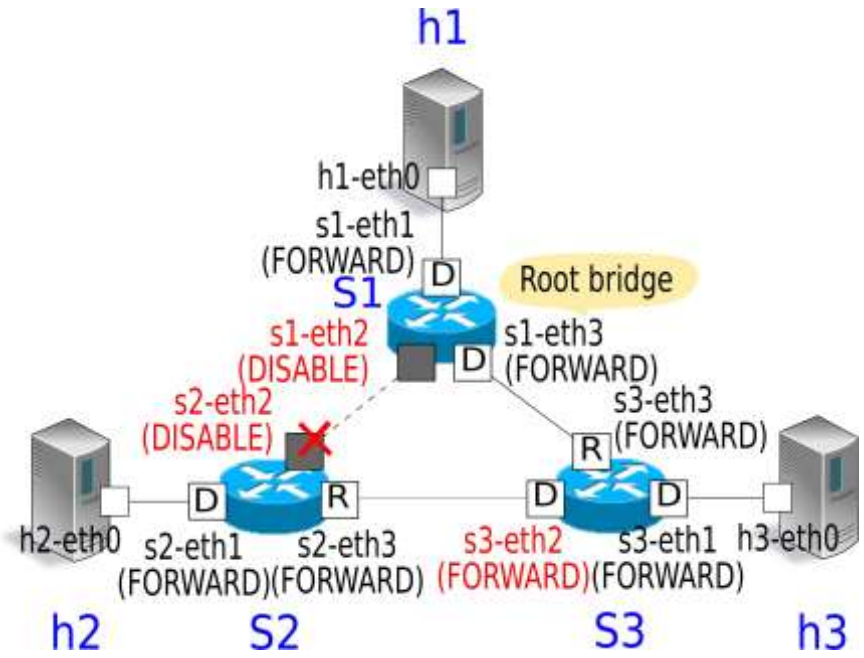
```
root@ryu-vm:~# ifconfig s2-eth2 down
```

断线被侦测到的时候，STP 会被重新计算。

```
[STP][INFO] dpid=0000000000000002: [port=2] Link down.
[STP][INFO] dpid=0000000000000002: [port=2] DESIGNATED_PORT      / DISABLE
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000002: Root bridge.
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT      / LISTEN
[STP][INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT      / LISTEN
[STP][INFO] dpid=0000000000000001: [port=2] Link down.
[STP][INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT      / DISABLE
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT      / LEARN
[STP][INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT      / LEARN
[STP][INFO] dpid=0000000000000003: [port=2] Wait BPDU timer is exceeded.
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000003: [port=2] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000003: [port=3] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000003: Root bridge.
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT      / LISTEN
```

```
[STP][INFO] dpid=0000000000000003: [port=2] DESIGNATED_PORT      / LISTEN
[STP][INFO] dpid=0000000000000003: [port=3] DESIGNATED_PORT      / LISTEN
[STP][INFO] dpid=0000000000000003: [port=3] Receive superior BPDU.
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000003: [port=2] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000003: [port=3] DESIGNATED_PORT      / BLOCK
[ STP][ INFO] dpid =0000000000000003: Non  root  bridge.
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT      / LISTEN
[STP][INFO] dpid=0000000000000003: [port=2] DESIGNATED_PORT      / LISTEN
[STP][INFO] dpid=0000000000000003: [port=3] ROOT_PORT            / LISTEN
[STP][INFO] dpid=0000000000000002: [port=3] Receive superior BPDU.
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT      / BLOCK
[ STP][ INFO] dpid =0000000000000002: Non  root  bridge.
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT      / LISTEN
[ STP][ INFO] dpid =0000000000000002: [ port =3] ROOT_PORT       / LISTEN
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT      / LEARN
[STP][INFO] dpid=0000000000000003: [port=2] DESIGNATED_PORT      / LEARN
[ STP][ INFO] dpid =0000000000000003: [ port =3] ROOT_PORT       / LEARN
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT      / LEARN
[ STP][ INFO] dpid =0000000000000002: [ port =3] ROOT_PORT       / LEARN
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT      / FORWARD
[STP][INFO] dpid=0000000000000003: [port=2] DESIGNATED_PORT      / FORWARD
[ STP][ INFO] dpid =0000000000000003: [ port =3] ROOT_PORT       / FORWARD
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT      / FORWARD
[STP][INFO] dpid=0000000000000002: [port=3] ROOT_PORT            / FORWARD
```

在此之前 s3-eth2 为 BLOCK 状态，但现在端口的状态为 FORWARD，而讯框将可以再次被传送。




### 从线路故障的状态回复时重新计算 STP

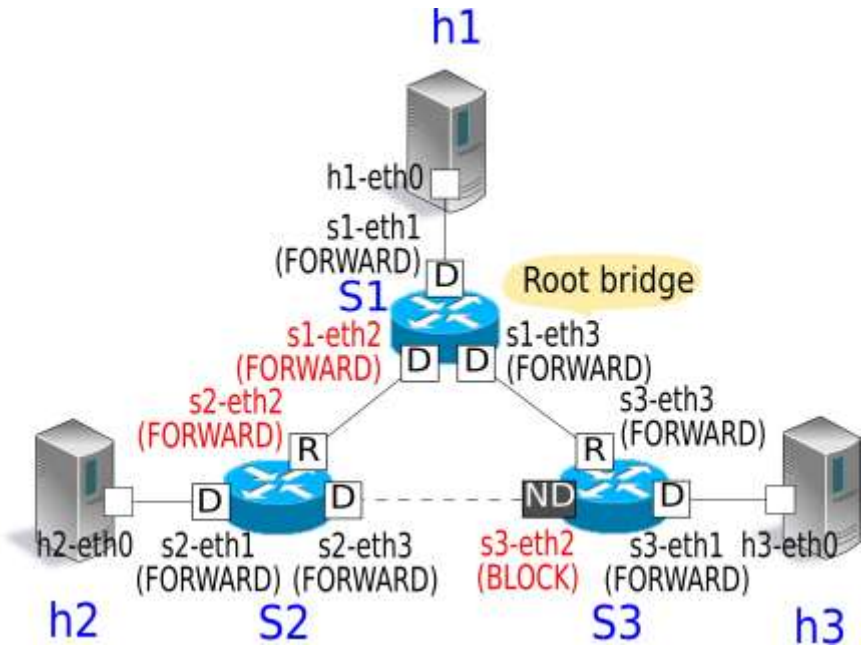
接下来，断线回复的时候 STP 将被重新计算。在断线的状态下执行下列的命令让端口恢复。 Node: s2:

```
root@ryu-vm:~# ifconfig s2-eth2 up
```

联机恢复后被侦测到，STP 就会进行再次的计算。

```
[STP][INFO] dpid=0000000000000002: [port=2] Link down.
[STP][INFO] dpid=0000000000000002: [port=2] DESIGNATED_PORT      / DISABLE
[STP][INFO] dpid=0000000000000002: [port=2] Link up.
[STP][INFO] dpid=0000000000000002: [port=2] DESIGNATED_PORT      / LISTEN
[STP][INFO] dpid=0000000000000001: [port=2] Link up.
[STP][INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT      / LISTEN
[STP][INFO] dpid=0000000000000001: [port=2] Receive superior BPDU.
[STP][INFO] dpid=0000000000000001: [port=1] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000001: [port=3] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000001: Root bridge.
[STP][INFO] dpid=0000000000000001: [port=1] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000001: [port=3] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000002: [port=2] Receive superior BPDU.
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000002: [port=2] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000002: Non root bridge.
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000002: [port=2] ROOT_PORT            / LISTEN
[STP][INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT   / LISTEN
[STP][INFO] dpid=0000000000000003: [port=2] Receive superior BPDU.
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000003: [port=2] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000003: [port=3] DESIGNATED_PORT      / BLOCK
[STP][INFO] dpid=0000000000000003: Non root bridge.
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT      / LISTEN
[STP][INFO] dpid=0000000000000003: [port=2] NON_DESIGNATED_PORT / LISTEN
[STP][INFO] dpid=0000000000000003: [port=3] ROOT_PORT            / LISTEN
[STP][INFO] dpid=0000000000000001: [port=1] DESIGNATED_PORT      / LEARN
[STP][INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT      / LEARN
[STP][INFO] dpid=0000000000000001: [port=3] DESIGNATED_PORT      / LEARN
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT      / LEARN
[STP][INFO] dpid=0000000000000002: [port=2] ROOT_PORT            / LEARN
[STP][INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT      / LEARN
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT      / LEARN
[STP][INFO] dpid=0000000000000003: [port=2] NON_DESIGNATED_PORT / LEARN
[STP][INFO] dpid=0000000000000003: [port=3] ROOT_PORT            / LEARN
[STP][INFO] dpid=0000000000000001: [port=1] DESIGNATED_PORT      / FORWARD
[STP][INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT      / FORWARD
[STP][INFO] dpid=0000000000000001: [port=3] DESIGNATED_PORT      / FORWARD
[STP][INFO] dpid=0000000000000002: [port=1] DESIGNATED_PORT      / FORWARD
[STP][INFO] dpid=0000000000000002: [port=2] ROOT_PORT            / FORWARD
[STP][INFO] dpid=0000000000000002: [port=3] DESIGNATED_PORT      / FORWARD
[STP][INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT      / FORWARD
[STP][INFO] dpid=0000000000000003: [port=2] NON_DESIGNATED_PORT / BLOCK
[STP][INFO] dpid=0000000000000003: [port=3] ROOT_PORT            / FORWARD
```

可以确认目前的状态跟应用程序启动时有相同的树状结构，而讯框可以再次被传送。




## 5.3 使用 OpenFlow 完成生成树

让我们看一下在 Ryu 生成树应用程序中，如何使用 OpenFlow 完成生成树的功能。

OpenFlow 1.3 提供 config 来设定端口的状态。发送 Port Modification 讯息到 OpenFlow 交换器以控制端口对讯框的转送行为。

| 名称 | 说明 |
|---|---|
| OFPPC_PORT_DOWN | 端口无效状态 |
| OFPPC_NO_RECV | 丢弃所有接收到的封包 |
| OFPPC_NO_FWD | 停止转送封包 |
| OFPPC_NO_PACKET_IN | table-miss 发生时，不发送 Packet-In 讯息 |

为了控制端口接收 BPDU 封包和非 BPDU 封包，收到 BPDU 封包就发送 Packet-In 的 Flow Entry 和接收 BPDU 以外的封包就丢弃的 Flow Entry ，分别透过 Flow Mod 讯息新增到 OpenFlow 交换器中。

Controller 对各个 OpenFlow 交换器进行下面 port 设定和 Flow Entry 的管理，以达到控制连接埠状态对于 BPDU 的接收传送和 MAC 地址的学习（BPDU 以外则是封包的接收）和讯框的转送（BPDU 以外则是封包的传送）。

| 名称 | 设定值 | Flow Entry |
|---|---|---|
| DISABLE | NO_RECV / NO_FWD | 无 |
| BLOCK | NO_FWD | BPDU Packet-In / BPDU 以外 drop |
| LISTEN | 无 | BPDU Packet-In / BPDU 以外 drop |
| LEARN | 无 | BPDU Packet-In / BPDU 以外 drop |
| FORWARD | 无 | BPDU Packet-In |

备注: 为了精简化，Ryu 里的生成树函式库并不处理 LEARN 状态的 MAC 地址（接收 BPDU 以外的封包）学习。

为了做这些设定，Controller 产生 BPDU 封包基于 OpenFlow 交换器连接时所收集的端口信息和每一个 OpenFlow 交换器所接收的 BPDU 封包中所设定的 Root 网桥信息，来发送 Packet-Out 讯息达到交换器之间互相交换 BPDU 的效果。

## 5.4 使用 Ryu 实作生成树

接下来，检视一下 Ryu 所用来实作生成树的原始码。生成树的原始码存放在 Ryu 的原始码当中。

ryu/lib/stplib.py

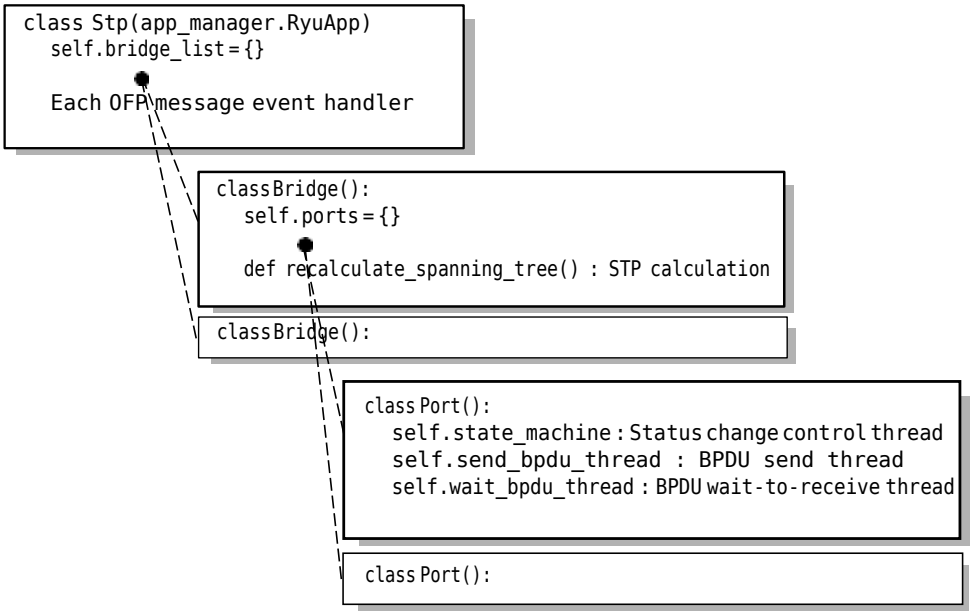
ryu/app/simple_switch_stp.py

stplib.py 是用来提供 BPDU 封包的交换和端口的角色、状态管理的生成树函式库。simple_switch_stp.py 是一个应用程序，用来让交换器的应用程序新增生成树函式库以增加生成树功能使用。

注意: 因为 simple_switch_stp.py 是 OpenFlow 1.0 专用的应用程序，本章为「执行 Ryu 应用程序」因此使用 OpenFlow 1.3 所对应的 simple_switch_stp_13.py 作为详细说明的目标。

### 5.4.1 函式库的安装

#### 函式库概述




STP 函式库（STP 实体）侦测到 OpenFlow 交换器和 Controller 已经连结时，Bridge 类别的实体和 Port类别的实体就会被产生。

当每一个类别的实体产生、启动之后。

- 从 STP 类别实体接收到 OpenFlow 讯息。
- Bridge 类别实体运算 STP（Root 网桥的选择，每一个端口的角色选择）
- Port 类别实体的状态变化，BPDU封包接收及传送 以上动作完成后，即可达成生成树的功能。

#### 设定项目

如果使用 `Stp.set_config()` 方法，则 STP 函式库有提供网桥端口的设定项目。可用的设 定项目如下：

- bridge

| 名称 | 说明 | 默认值 |
|---|---|---|
| priority | 网桥优先权 | 0x8000 |
| sys_ext_id | 设定 VLAN-ID (* 目前的 STP 函式库不支援 VLAN) | 0 |
| max_age | BPDU 封包的传送接收逾时 | 20[sec] |
| hello_time | BPDU 封包的传送接收间隔 | 2 [sec] |
| fwd_delay | 每一个端口停留在 LISTEN 或 LEARN 的时间 | 15[sec] |

- port

| 名称 | 说明 | 默认值 |
|---|---|---|
| priority | 端口优先权 | 0x80 |
| path_cost | Link Cost | 基于连接速度自动设定 |
| enable | 端口的有效/无效 | True |

### 传送 BPDU 封包

传送 BPDU 封包的动作为 BPDU 封包传送执行绪（Port.send_bpdu_thread）所执行，当连接埠的角色为 designated port（DESIGNATED_PORT）时，定期通知 Root 桥接器的 hello time（Port.port_times.hello_time：预设 2 秒）封包就会被产生（Port._generate_config_bpdu()）并进行传送（Port.ofctl.send_packet_out()）。

```
class Port(object):

    def __init__(self, dp, logger, config, send_ev_func, timeout_func,
                 topology_change_func, bridge_id, bridge_times, ofport):
        super(Port, self).__init__()

        # ...

        # BPDU handling threads
        self. send_bpdu_thread = PortThread( self._transmit_bpdu)

    # ...

    def _transmit_bpdu(self):
        while True:
            # Send config BPDU packet if port role is DESIGNATED_PORT.
            if self.role == DESIGNATED_PORT:

                # ...

                bpdu_data  =  self. _generate_config_bpdu  ( flags)
                self.ofctl.send_packet_out(self.ofport.port_no,  bpdu_data)

                # ...

            hub.sleep(self.port_times.hello_time)
```

即将被转送的 BPDU 封包所设定的内容会来自 OpenFlow 交换器与 Controller 连接时所收集到的端口信息（Port.ofport）和曾经接受到的 BPDU 封包中 Root 网桥的信息（Port.port_priority、Port.port_times）。

```
class Port(object):

    def _generate_config_bpdu(self, flags):
        src_mac = self.ofport.hw_addr
```

```
        dst_mac = bpdu.BRIDGE_GROUP_ADDRESS
        length = (bpdu.bpdu._PACK_LEN + bpdu.ConfigurationBPDUs.PACK_LEN
                  + llc.llc._PACK_LEN + llc.ControlFormatU._PACK_LEN)

        e = ethernet.ethernet(dst_mac, src_mac, length)
        l = llc.llc(llc.SAP_BPDU, llc.SAP_BPDU, llc.ControlFormatU())
        b = bpdu.ConfigurationBPDUs(
            flags=              flags              ,
            root_priority=self.port_priority.root_id.priority,
            root_mac_address= self.port_priority.root_id.mac_addr ,
            root_path_cost= self.port_priority.root_path_cost+ self.path_cost ,
            bridge_priority= self. bridge_id. priority ,
            bridge_mac_address = self. bridge_id. mac_addr ,
            port_priority=   self.   port_id.   priority   ,
            port_number=    self.    ofport.    port_no    ,
            message_age=  self.  port_times.  message_age  +1  ,
            max_age=     self.     port_times.     max_age     ,
            hello_time=  self.  port_times.  hello_time   ,
            forward_delay= self.port_times.forward_delay)

        pkt = packet.Packet()
        pkt.add_protocol(e)
        pkt.add_protocol(l)
        pkt.add_protocol(b)
        pkt.serialize()

        return pkt.data
```

### 接收 BPDU 封包

接收BPDU封包是由STP类别的Packet-In事件管理器（Event handler）所发现，经由Bridge类别实体通知给 Port 类别实体。事件管理器的实作请参考「交换器（Switching Hub）」。

接收到 BPDU 封包的端口会对先前接收到的 BPDU 封包以及本次接收到的封包中的网桥 ID 进行比对（`Stp.compare_bpdu_info()`），来决定 STP 是否必须重新计算路径。若是相比之前 的封包之下，本次收到的封包为优先封包（superior BPDU）、（`SUPERIOR`）时，则代表网络的 拓璞已经改变，例如「一个新的 Root 网桥已经被加入网络」，此时则必须开始进行 STP 的重 新计算。

```
class Port(object):

    def rcv_config_bpdu(self, bpdu_pkt):
        # Check received BPDU is superior to currently held BPDU.
        root_id = BridgeId(bpdu_pkt.root_priority,
                           bpdu_pkt.root_system_id_extension ,
                           bpdu_pkt.root_mac_address)
        root_path_cost = bpdu_pkt. root_path_cost
        designated_bridge_id = BridgeId(bpdu_pkt.bridge_priority,
                                        bpdu_pkt.bridge_system_id_extension ,
                                        bpdu_pkt.bridge_mac_address)
        designated_port_id = PortId(bpdu_pkt.port_priority,
                                    bpdu_pkt.port_number)

        msg_priority = Priority(root_id, root_path_cost,
                                designated_bridge_id ,
                                designated_port_id)
        msg_times = Times(bpdu_pkt.message_age,
                          bpdu_pkt. max_age ,
                          bpdu_pkt.hello_time,
                          bpdu_pkt.forward_delay)
```

```
        rcv_info = Stp.compare_bpdu_info(self.designated_priority,
                                         self.designated_times,
                                         msg_priority, msg_times)

        # ...

        return rcv_info, rcv_tc
```

### 故障侦测

直接的故障例如：断线，或者间接的故障例如：在一定时间内没有接收到 Root 网桥所发出的BPDU封包，这时候STP就必须进行重新计算。

断线是由STP类别的PortStatus事件管理器所侦测，并透过Bridge类别的实体进行通知。

BPDU 封包的接收逾时是由 Port 类别的 BPDU 封包接收线程（`Port.wait_bpdu_thread`）所发现。在 max age（预设 20 秒）之间，如果没有接收到 Root 网桥发来的 BPDU 封包，就判断为间接故障，并且对Bridge类别实体发送通知。

BPDU 接收逾时的更新和逾时的侦测分别是 hub 模组（ryu.lib.hub）的 `hub.Event` 和 `hub.Timeout` 。`hub.Event` 经由 `hub.Event.wait()` 进入 wait 状态，透过 `hub.Event.set()` 中断线程。`hub.Timeout` 指定在一定时间内若 `try` 无法结束执行时，则发送 `hub.Timeout` 的例外事件。当 `hub.Event` 进入 wait 状态，并且 `hub.Timeout` 所指定的时间内尚未执行 `hub.Event.set()` 时，则判断为 BPDU 封包的接收逾时，故开始进行 Bridge 类别的 STP 重新计算。

```
class Port(object):

    def __init__(self, dp, logger, config, send_ev_func, timeout_func,
                 topology_change_func, bridge_id, bridge_times, ofport):
        super(Port, self).__init__()
        # ...
        self.wait_bpdu_timeout = timeout_func
        # ...
        self. wait_bpdu_thread = PortThread( self._wait_bpdu_timer)

    # ...

    def _wait_bpdu_timer(self):
        time_exceed = False

        while True:
            self.wait_timer_event = hub.Event()
            message_age = (self.designated_times.message_age
                           if self.designated_times else 0)
            timer = self.port_times.max_age - message_age
            timeout = hub.Timeout(timer)
            try:
                self. wait_timer_event. wait ()
            except hub.Timeout as t:
                if t is not timeout:
                    err_msg = 'Internal error. Not my timeout.'
                    raise       RyuException(      msg=      err_msg)
                self.logger.info('[port=%d] Wait BPDU timer is exceeded.',
                                 self.ofport.port_no, extra=self.dpid_str)
                time_exceed = True
            finally:
                timeout.    cancel    ()
                self.wait_timer_event = None
```

```
            if time_exceed:
                break

        if time_exceed:  # Bridge.recalculate_spanning_tree
            hub.spawn(self.wait_bpdu_timeout)
```

接收的 BPDU 封包比对动作（Stp.compare_bpdu_info()）结束后发现是 SUPERIOR 或 REPEATED 时，就开始接收 Root 桥接器发送过来的封包，并且更新逾时数值（Port._update_wait_bpdu_timer()）。执行 hub.Event 中 Port.wait_timer_event 的 set() 将 Port.wait_timer_event 从 wait 状态解除，即可让 BPDU 封包接收执行绪（Port.wait_bpdu_thread）不要进入 except hub.Timeout 的区间并进行处理，进而重新设定定时器后继续接收下一个 BPDU 封包。

```
class Port(object):

    def rcv_config_bpdu(self, bpdu_pkt):
        # ...

        rcv_info = Stp.compare_bpdu_info(self.designated_priority,
                                         self.designated_times,
                                         msg_priority, msg_times)
        # ...

        if ((rcv_info is SUPERIOR or rcv_info is REPEATED)
                and (self.role is ROOT_PORT
                     or self.role is NON_DESIGNATED_PORT)):
            self._update_wait_bpdu_timer()

        # ...

    def _update_wait_bpdu_timer(self):
        if self.wait_timer_event is not None:
            self.wait_timer_event.set()
            self.wait_timer_event = None
```

### STP 计算

STP 计算（Root 网桥选择、每一个端口的角色选择）是由 Bridge 类别所执行。

STP 重新计算开始的时候代表网路拓璞已经发生变化，此时封包的传递有可能会有出现回圈，因此所有的连接埠会进入 BLOCK（port.down）状态并触发拓璞改变事件（EventTopologyChange）通知上层 APL，初始已经学习完毕的 MAC 地址端口。

然后 Bridge._spanning_tree_algorithm() 开始动作以进行 Root 网桥的选择和端口的 角色设定，所有的端口会从 LISTEN 状态（port.up）开始状态的变化。

```
class Bridge(object):

    def recalculate_spanning_tree(self, init=True):
        """ Re-calculation of spanning tree. """
        # All port down.
        for port in self.ports.values():
            if port.state is not PORT_STATE_DISABLE:
                port.down(PORT_STATE_BLOCK, msg_init=init)

        # Send topology change event.
        if init:
            self.send_event(EventTopologyChange(self.dp))
```

```
        # Update tree roles.
        port_roles ={}
        self.root_priority = Priority(self.bridge_id, 0, None, None)
        self.root_times = self.bridge_times

        if init:
            self.logger.info('Root bridge.', extra=self.dpid_str)
            for port_no in self. ports. keys ():
                port_roles[port_no] = DESIGNATED_PORT
        else:
            ( port_roles ,
             self.root_priority ,
             self.root_times) = self._spanning_tree_algorithm()

        # All port up.
        for port_no, role in port_roles.items():
            if self.ports[port_no].state is not PORT_STATE_DISABLE:
                self.ports[port_no].up(role, self.root_priority,
                                       self. root_times)
```

为了 Root 网桥的选择，像是网桥 ID 之类的信息会拿来跟每一个端口所接收到的 BPDU 封包进行比对（`Bridge._select_root_port`）。

接着会出现这样的结果：选出 Root 端口（本身的网桥信息和从端口所收到的其他 网桥信息比对后结果较差）、其他的网桥开始做为 Root 网桥、或选出 designated ports（`Bridge._select_designated_port`）和选出non-designated ports（Root端口 / designated ports 以外的 non-designated ports 选出）。

反之 Root 网桥如果没有被发现（本身的网桥所带的信息为最优先的情况），则自己就会转 变为 Root 网桥并设定其所有的端口为 designated ports。

```
class Bridge(object):

    def _spanning_tree_algorithm(self):
        """ Update tree roles.
             - Root bridge:
                all port is DESIGNATED_PORT.
             - Non rootbridge:
                select one ROOT_PORT and some DESIGNATED_PORT,
                and the other port is set to NON_DESIGNATED_PORT."""
        port_roles ={}

        root_port = self._select_root_port()

        if root_port is None:
            # My bridge is a root bridge.
            self.logger.info('Root bridge.', extra=self.dpid_str)
            root_priority = self.root_priority
            root_times = self.root_times

            for port_no in self.ports.keys():
                if self.ports[port_no].state is not PORT_STATE_DISABLE:
                    port_roles[port_no] = DESIGNATED_PORT
        else:
            # Other bridge is a root bridge.
            self.logger.info('Non root bridge.', extra=self.dpid_str)
            root_priority = root_port.designated_priority
            root_times = root_port. designated_times

            port_roles[root_port.ofport.port_no] = ROOT_PORT
```

```
            d_ports = self._select_designated_port(root_port)
            for   port_no   in   d_ports:
                port_roles[port_no] = DESIGNATED_PORT

            for port in self.ports.values():
                if port. state is not PORT_STATE_DISABLE:
                    port_roles.setdefault(port.ofport.port_no,
                                          NON_DESIGNATED_PORT)

        return port_roles, root_priority, root_times
```

### 端口的状态转移

连接埠的状态转移是由 Port 类别的状态转移控制执行绪（Port.state_machine）所处理。下一个状态的转移时限是从 Port._get_timer() 取得。当发生逾时之后就会从 Port._get_next_state() 取得接下来将转移的状态并进行移转。而 STP 再次重新计算的时候，端口的状态就会直接使用 Port._change_status() 切换至 BLOCK 的状态，不论先前的状态为何。这样的处理跟故障侦测」一样，是由 hub 模块的 hub.Event 和 hub.Timeout 来达成。

```
class Port(object):

    def _state_machine(self):
        """ Port state machine.
             Change next status when timer is exceeded
             or _change_status() method is called."""

        # ...

        while True:
            self.logger.info('[port=%d] %s / %s', self.ofport.port_no,
                             role_str[self.role],  state_str[self.state],
                             extra=self.dpid_str)

            self.state_event = hub.Event()
            timer = self._get_timer()
            if timer:
                timeout = hub.Timeout(timer)
                try:
                    self.state_event.wait()
                except hub.Timeout as t:
                    if t is not timeout:
                        err_msg = 'Internal error. Not my timeout.'
                        raise RyuException(msg=err_msg)
                    new_state = self._get_next_state()
                    self._change_status(new_state,  thread_switch=False)
                finally:
                    timeout.cancel()
            else:
                self.state_event.wait()

            self.state_event = None

    def _get_timer(self):
        timer = {PORT_STATE_DISABLE: None,
                 PORT_STATE_BLOCK: None,
                 PORT_STATE_LISTEN:  self.port_times.forward_delay,
                 PORT_STATE_LEARN: self.port_times.forward_delay,
                 PORT_STATE_FORWARD: None}
        return timer[self.state]
```

```
    def _get_next_state(self):
        next_state = {PORT_STATE_DISABLE: None,
                      PORT_STATE_BLOCK : None ,
                      PORT_STATE_LISTEN : PORT_STATE_LEARN ,
                      PORT_STATE_LEARN : (PORT_STATE_FORWARD
                                          if (self.role is ROOT_PORT or
                                              self.role is DESIGNATED_PORT)
                                          else PORT_STATE_BLOCK),
                      PORT_STATE_FORWARD: None}
        return next_state[self.state]
```

### 5.4.2 实作应用程序

本章说明「执行 Ryu 应用程式」中 OpenFlow 1.3 所对应的生成树应用程式（simple_switch_stp_13.py）和「交换器（Switching Hub）」的交换器之间的差异。

#### 设定「_CONTEXTS」

跟「网络聚合（Link Aggregation）」一样用 CONTEXT 登录，藉以应用相同的 STP 函式库。

```
from ryu.lib import stplib

# ...

class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    _CONTEXTS = {'stplib': stplib.Stp}

    # ...
```

#### 设定组态

使用 STP 函式库的 `set_config()` 方法来进行组态设定，下面是简单的例子。

| OpenFlow 交换器 | 名称 | 设定值 |
|---|---|---|
| dpid=0000000000000001 | bridge.priority | 0x8000 |
| dpid=0000000000000002 | bridge.priority | 0x9000 |
| dpid=0000000000000003 | bridge.priority | 0xa000 |

使用这个设定时 dpid=0000000000000001 的 OpenFlow 交换器的网桥 ID 总会是最小值，而 Root 网桥也会选择该交换器。

```
class  SimpleSwitch13(app_manager.RyuApp):

    # ...

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        self.stp = kwargs['stplib']

        # Sample of stplib config.
        #  please refer to stplib.Stp.set_config() for details.
        config = {dpid_lib.str_to_dpid('0000000000000001'):
                     {'bridge': {'priority': 0x8000}},
```

```
                dpid_lib.str_to_dpid('0000000000000002'):
                    {'bridge': {'priority': 0x9000}},
                dpid_lib.str_to_dpid('0000000000000003'):
                    {'bridge': {'priority': 0xa000}}}
        self.stp.set_config(config)
```

### STP 事件处理

跟「网络聚合（Link Aggregation）」一样，准备事件管理器来接收来自 STP 函式库的通知。

使用 STP 函式库中定义的 `stplib.EventPacketIn` 事件来接收 BPDU 以外的封包。因此封包管理及处理动作跟「交换器（Switching Hub）」相同。

```
class SimpleSwitch13(app_manager.RyuApp):

    @set_ev_cls(stplib.EventPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):

        # ...
```

接收网络拓璞的变动事件通知（`stplib.EventTopologyChange`）用以初始化已经学习的 MAC 地址和已经注册的 Flow Entry。

```
class SimpleSwitch13(app_manager.RyuApp):

    def delete_flow( self , datapath):
        ofproto = datapath. ofproto
        parser = datapath.ofproto_parser

        for dst in self.mac_to_port[datapath.id].keys():
            match = parser.OFPMatch(eth_dst=dst)
            mod = parser.OFPFlowMod(
                datapath , command= ofproto. OFPFC_DELETE ,
                out_port=ofproto.OFPP_ANY, out_group=ofproto.OFPG_ANY,
                priority=1, match=match)
            datapath.send_msg(mod)

    # ...

    @set_ev_cls(stplib.EventTopologyChange, MAIN_DISPATCHER)
    def _topology_change_handler(self, ev):
        dp = ev.dp
        dpid_str = dpid_lib.dpid_to_str(dp.id)
        msg = 'Receive topology change event. Flush MAC table.'
        self.logger.debug("[dpid=%s] %s", dpid_str, msg)

        if dp.id in self.mac_to_port:
            self.delete_flow(dp)
            del self.mac_to_port[dp.id]
```

接收到端口的状态通知事件（`stplib.EventPortStateChange`）并且将端口的状态输出。

```
class SimpleSwitch13(app_manager.RyuApp):

    @set_ev_cls(stplib.EventPortStateChange, MAIN_DISPATCHER)
    def _port_state_change_handler(self, ev):
        dpid_str = dpid_lib.dpid_to_str(ev.dp.id)
        of_state = {stplib.PORT_STATE_DISABLE: 'DISABLE',
                    stplib.PORT_STATE_BLOCK: 'BLOCK',
                    stplib.PORT_STATE_LISTEN: 'LISTEN',
```

```
                    stplib.PORT_STATE_LEARN: 'LEARN',
                    stplib.PORT_STATE_FORWARD: 'FORWARD'}
        self.logger.debug("[dpid=%s][port=%d] state=%s",
                          dpid_str, ev.port_no, of_state[ev.port_state])
```

经过以上的处理，透过提供生成树功能的函式库和使用该函式库的应用程序，实作成了拥有生成树功能的交换器应用程序。

## 5.5 本章总结

本章透过生成树函式库的使用，说明下列的目标。

- 使用 hub.Event 实作事件等待处理
- 使用 hub.Timeout 实作逾时管理

CHAPTER 6

# OpenFlow 通讯协议

本章将描述 Match、Instructions 和 Action 在 OpenFlow 协议中的细节。

## 6.1 Match

有许多种类的指定条件可以用在 Match ，随着 OpenFlow 版本的持续更新，数量也在持续增加中。在 OpenFlow 1.0 时仅有 12 种，OpenFlow 1.3 时数量就来到约 40 种。

若想要了解每个指令的细节，请参考 OpenFlow 规格书。本章仅简要列出 OpenFlow 1.3 的 Match 指令。

| Match Field 名称 | 说明 |
|---|---|
| in_port | 接受端口编号 (含逻辑端口) |
| in_phy_port | 接收端口实体编号 |
| metadata | 在 table 间传递使用的 Metadata |
| eth_src | Ethernet MAC 来源地址 |
| eth_dst | Ethernet MAC 目的地址 |
| eth_type | Ethernet 讯框种类 |
| vlan_vid | VLAN ID |
| vlan_pcp | VLAN PCP |
| ip_dscp | IP DSCP |
| ip_ecn | IP ECN |
| ip_proto | IP 协定种类 |
| ipv4_src | IPv4 IP 来源地址 |
| ipv4_dst | IPv4 IP 目的地址 |
| tcp_src | TCP port 来源编号 |
| tcp_dst | TCP port 目的编号 |
| udp_src | UDP port 来源编号 |
| udp_dst | UDP port 目的编号 |
| sctp_src | SCTP port 来源编号 |
| sctp_dst | SCTP port 目的编号 |
| icmpv4_type | ICMP 种类 |
| icmpv4_code | ICMP Code 编码 |
| arp_op | ARP Opcode |
| arp_spa | ARP IP 来源地址 |
| arp_tpa | ARP IP 目的地址 |

下一页继续

表 6.1 – 呈接上一页

| Match Field 名称 | 说明 |
|---|---|
| arp_sha | ARP MAC 来源地址 |
| arp_tha | ARP MAC 目的地址 |
| ipv6_src | IPv6 IP 来源地址 |
| ipv6_dst | IPv6 IP 目的地址 |
| ipv6_flabel | IPv6 Flow label |
| icmpv6_type | ICMPv6 Type |
| icmpv6_code | ICMPv6 Code |
| ipv6_nd_target | IPv6 neighbour discovery 目的地址 |
| ipv6_nd_sll | IPv6 neighbour discovery link-layer 来源地址 |
| ipv6_nd_tll | IPv6 neighbour discovery link-layer 目的地址 |
| mpls_label | MPLS 标签 |
| mpls_tc | MPLS Traffic class(TC) |
| mpls_bos | MPLS Bos bit |
| pbb_isid | 802.1ah PBB I-SID |
| tunnel_id | 逻辑端口的 metadata |
| ipv6_exthdr | IPv6 extension header 的 Pseudo-field |

可以针对 MAC 地址或 IP 地址，透过 Mask 来指定 field。

## 6.2 Instruction

Instruction 是用来定义当封包满足所规范的 Match 条件时，需要执行的动作。下面列出相关的定义。

| Instruction | 说明 |
|---|---|
| Goto Table(必要) | 在 OpenFlow 1.1 或更新的版本中，multiple flow tables 将是必需支援的项目。透过 Goto Table 的指令可以在多个 table 间进行移转，并继续相关的比对及对应的动作。例如：「收到来自 port 1 的封包时，增加 VLAN-ID 200 的 tag ，并移动至 table 2」。而所指定的 table ID 则必须是大于目前的 table ID。 |
| Write Metadata(选项) | 写入 Metadata 以做为下一个 table 所需的参考数据。 |
| Write Actions(必要) | 在目前的 action set 中写入新的 action ，如果有相同的 action 存在时，会进行覆盖。 |
| Apply Actions(选项) | 立刻执行所指定的 action 不对现有的 action set 进行修改。 |
| Clear Actions(选项) | 清空目前存在 action set 中的资料。 |
| Meter(选项) | 指定该封包到所定义的 meter table。 |

以下的类别是对应各个 Instruction 的 Ryu 实作。

- `OFPInstructionGotoTable`
- `OFPInstructionWriteMetadata`
- `OFPInstructionActions`
- `OFPInstructionMeter`

Write/Apply/Clear Actions 已经包含在 OPFInstructionActions 中，可以在安装的时候进行选取。

备注: Write Actions 虽然在规格中被列为必要，但是目前的 Open vSwitch 并不支持该功能。Apply Actions 是目前 Open vSwitch 所提供的功能，所以可以用来替代 Write Actions。Write Actions 预计在 Open vSwitch 2.1.0 中支援。

## 6.3 Action

OFPActionOutput Class 是用来转送指定封包，其中包含 Packet-Out 和 Flow Mod。设定好要传送的最大封包容量（max_len）和要传送的 Controller 目的地做为 Constructor 的参数。对于设定目的地，除了实体端口号之外还有一些其他的值可以进行定义。

| 名称 | 说明 |
|---|---|
| OFPP_IN_PORT | 转送到接收埠 |
| OFPP_TABLE | 转送到最前端的 table |
| OFPP_NORMAL | 使用交换器本身的 L2/L3 功能转送 |
| OFPP_FLOOD | 转送（Flood）到所有 VLAN 的物理端口，除了来源埠跟已闭锁的埠之外 |
| OFPP_ALL | 转送到除了来源埠之外的所有埠 |
| OFPP_CONTROLLER | 转送到 Controller 的 Packet-In 讯息 |
| OFPP_LOCAL | 转送到交换器本身（local port） |
| OFPP_ANY | 使用 Wild card 来指定 Flow Mod（delete）或 Flow Stats Requests 讯息的端口号，主要功能并不是用来转送封包讯息。 |

当指定 max_len 为 0 时，Binary data 将不会被加在 Packet-In 的讯息中。当 `OFPCML_NO_BUFFER` 被指定时，所有的封包将会加入 Packet-In 讯息中而不会暂存在 OpenFlow 交换器。

CHAPTER 7

# ofproto 函式库

本章介绍 Ryu ofproto 函式库。

## 7.1 简单说明

ofproto 函式库是用来产生及解析 OpenFlow 讯息的函式库。

## 7.2 相关模块

每个 OpenFlow（版本 X.Y ）都有相对应的常数模组（ofproto_vX_Y ）和解析模组（ofproto_vX_Y_parser ）每个 OpenFlow 版本的实作基本上是独立的。

| OpenFlow 版本 | 常数模块 | 解析模块 |
|---|---|---|
| 1.0.x | ryu.ofproto.ofproto_v1_0 | ryu.ofproto.ofproto_v1_0_parser |
| 1.2.x | ryu.ofproto.ofproto_v1_2 | ryu.ofproto.ofproto_v1_2_parser |
| 1.3.x | ryu.ofproto.ofproto_v1_3 | ryu.ofproto.ofproto_v1_3_parser |
| 1.4.x | ryu.ofproto.ofproto_v1_4 | ryu.ofproto.ofproto_v1_4_parser |

### 7.2.1 常数模块

常数模块是用来做为通讯协议中的常数设定使用，下面列出几个例子。

| 常数 | 说明 |
|---|---|
| OFP_VERSION | 通讯协议版本编号 |
| OFPP_xxxx | 端口号 |
| OFPCML_NO_BUFFER | 无缓冲区间，直接对全体发送讯息 |
| OFP_NO_BUFFER | 无效的缓冲编号 |

### 7.2.2 解析器模块

解析模块提供各个 OpenFlow 讯息的对应类别，下面列出几个例子。为了更好的说明类别的实 体，接下来的描述将用”讯息对象”取代”讯息类别”。

| 类别（Class ） | 说明 |
|---|---|
| OFPHello | OFPT_HELLO 讯息 |
| OFPPacketOut | OFPT_PACKET_OUT 讯息 |
| OFPFlowMod | OFPT_FLOW_MOD 讯息 |

解析模块对应了 OpenFlow 讯息的 Payload 结构中所需的定义，例如下面的例子。为了更好的 说明类别的实体，今后将用结构对象（structure object ）取代结构类别（structure class ）。

| 类别（Class ） | 结构 |
|---|---|
| OFPMatch | ofp_match |
| OFPInstructionGotoTable | ofp_instruction_goto_table |
| OFPActionOutput | ofp_action_output |

## 7.3 基本使用方法

### 7.3.1 ProtocolDesc 类别

这是为了 OpenFlow 协议所必需存在的类别。使用讯息类别的 __init__作为 datapath 的参数，指定该类别的对象。

```
from ryu.ofproto import ofproto_protocol
from ryu.ofproto import ofproto_v1_3

dp = ofproto_protocol.ProtocolDesc(version=ofproto_v1_3.OFP_VERSION)
```

### 7.3.2 网络地址（Network Address ）

Ryu ofproto 函式库的 API 使用最基本的文字表现网络地址，请看下面的例子。

备注: 但是 OpenFlow 1.0 和这样的标示方法不同。（2014 年 2 月更新）

| 地址（Address ）种类 | python 文字表示 |
|---|---|
| MAC 地址 | `00:03:47:8c:a1:b3' |
| IPv4 地址 | `192.0.2.1' |
| IPv6 地址 | `2001:db8::2' |

### 7.3.3 讯息对象（Message Object ）的产生

每个讯息类别（message class ），结构类别（structure class ）都需要适当的参数以用来产生。参数的名称跟 OpenFlow 协议所定义的域名基本上是一致的。在有冲突的情况下会在最后 加入「_」，例如：「type_ 」就是。

```
from ryu.ofproto import ofproto_protocol
from ryu.ofproto import ofproto_v1_3

dp = ofproto_protocol.ProtocolDesc(version=ofproto_v1_3.OFP_VERSION)
ofp = dp.ofproto
ofpp = dp.ofproto_parser
actions = [parser.OFPActionOutput(port=ofp.OFPP_CONTROLLER,
                                  max_len= ofp. OFPCML_NO_BUFFER)]
inst = [parser.OFPInstructionActions(type_=ofp.OFPIT_APPLY_ACTIONS,
                                     actions= actions)]
fm = ofpp.OFPFlowMod(datapath=dp,
```

```
                priority        =0       ,
                match=ofpp.OFPMatch(in_port=1,
                                    eth_src='00:50:56:c0:00:08'),
                instructions=inst)
```

备注: 常数模块、解析模块最好是在 import 的时候就直接标明。如此一来在 OpenFlow 版本变更的时候，可以将修正的程度将到最低。另外尽量使用 ProtocolDesc 对象的 ofproto 和 ofproto_parser 属性。

### 7.3.4 讯息对象（Message Object ）的解析

讯息对象（message object ）的内容是可以查询的。

例如 OFPPacketIn 对象中 pid 的 match field 用查询 pin.match 即可得到相关的讯息。

OFPMatch 物件中 TLV 的各部分可以使用下列的名称取得相关的资料。

```
print pin.match['in_port']
```

### 7.3.5 JSON

讯息对象（message object ）转换成为 json.dump 的功能是存在的，反之亦然。

备注: 但是目前 OpenFlow 1.0 相关的实作并不完全。（2014 年 2 月更新）

```
import json

print json.dumps(msg.to_jsondict())
```

### 7.3.6 讯息（message ）的解析（parse ）

该功能是为了把讯息的原始数据转换成讯息对象。对于从交换器收到的讯息，框架（Framwork ）会自动地进行处理，Ryu 应用程序（Application ）是不需要特别处理的。

具体来说如下：

1. ryu.ofproto.ofproto_parser.header 用来处理版本相依的解析
2. 上面处理过的结果可以用 ryu.ofproto.ofproto_parser.msg 功能来解析剩余的部分

### 7.3.7 讯息的产生（串行化，Serialize ）

将讯息对象转换并产生对应的讯息 Byte。同样的，来自交换器的讯息将由框架自动处理，Ryu 应用程序无需额外的动作。

具体来说如下：

1. 呼叫讯息对象的串行化方法
2. 从讯息对象中将 buf 的属性读取出来 有些字段，例如“len”即使不指定，在串行化的同时也会自动被计算出来。

CHAPTER 8

# 封包函式库

OpenFlow 中 Packet-In 和 Packet-Out 讯息是用来产生封包，可以在当中的字段放入 Byte 资料并转换为原始封包的方法。Ryu 提供了相当容易使用的封包产生函式库给应用程序使用。

本章将介绍该函式库。

## 8.1 基本使用方法

### 8.1.1 协定标头类别（Protocol Header Class）

Ryu 封包函式库提供许多协议对应的类别，用来解析或包装封包。

下面列出Ryu目前所支持的协议。若需要了解每个协议的细节请参照API参考数据

- arp
- bgp
- bpdu
- dhcp
- ethernet
- icmp
- icmpv6
- igmp
- ipv4
- ipv6
- llc
- lldp
- mpls
- ospf
- pbb
- sctp
- slow

- tcp
- udp
- vlan
- vrrp

每一个协议类别的 init 参数基本上跟 RFC 所提到的名称是一致的。同样的每一个协议实体 命名也相同。但是当 init 名称与 Python 内定的关键词发生冲突时，会在名称的尾端加上底 线「_」。

有些 init 的参数由于有内定的默认值，因此可以忽略。下面的例子中，原本需要被加入的 参数 version = 4 或其他值就可以被忽略。

```
from ryu.lib.ofproto import inet
from ryu.lib.packet import ipv4

pkt_ipv4 = ipv4.ipv4(dst='192.0.2.1',
                     src='192.0.2.2',
                     proto=inet.IPPROTO_UDP)
```

```
print pkt_ipv4.dst
print pkt_ipv4.src
print pkt_ipv4.proto
```

### 8.1.2 网络地址（Network Address）

Ryu 封包函式库的 API 使用最基本的文字作为表现。举例如下：

| 地址种类 | python 文字表示 |
|---|---|
| MAC 地址 | `00:03:47:8c:a1:b3' |
| IPv4 地址 | `192.0.2.1' |
| IPv6 地址 | `2001:db8::2' |

### 8.1.3 封包的解析（Parse）

封包的 Byte String 可以产生相对应的 Python 对象。

具体的事例如下：

1. ryu.lib.packet.packet.Packet 类别的对象产生（指定要解析的 byte string 给 data 作为参数）
2. 使用先前产生的对象中 get_protocol 方法，取得协议中相关属性的对象。

```
pkt = packet.Packet(data=bin_packet)
pkt_ethernet = pkt.get_protocol(ethernet.ethernet)
if not pkt_ethernet:
    # non ethernet
    return
print pkt_ethernet.dst
print pkt_ethernet.src
print pkt_ethernet.ethertype
```

### 8.1.4 封包的产生（串行化，Serialize）

把 Python 对象转换成为相对封包的 byte string。

具体说明如下：

1. 产生 ryu.lib.packet.packet.Packet 类别的对象
2. 产生相对应的协议对象（ethernet, ipv4, ...）
3. 在 1. 所产生的对象中，使用 add_protocol 方法将 2. 所产生的对象依序加入
4. 呼叫 1. 所产生对象中的 serialize 方法将对象转换成 byte string

Checksum 和 payload 的长度不需要特别设定，在串行化的同时会被自动计算出来。详细的各类别细节请参考相关信息。

```
pkt = packet.Packet()
pkt.add_protocol(ethernet.ethernet(ethertype=...,
                                   dst=...,
                                   src=...))
pkt.add_protocol(ipv4.ipv4(dst=...,
                           src=...,
                           proto=...))
pkt.add_protocol(icmp.icmp(type_=...,
                           code=...,
                           csum=...,
                           data=...))
pkt.serialize()
bin_packet = pkt.data
```

另外也提供 API 类似 Scapy，请根据个人喜好选择使用。

```
e = ethernet.ethernet(...)
i = ipv4.ipv4(...)
u = udp.udp(...)
pkt = e/i/u
```

## 8.2 应用程序范例

接下来的例子是使用上述的方法，达成一个可以针对 ping 做出回应的应用程序。

接受 Packet-In 所收到的 ARP REQUEST 和 ICMP ECHO REQUEST 后藉由 Packet-Out 发送回应。IP 地址等 __init__的参数都是使用固定程序代码（hard-code）的方式。

```
# Copyright (C) 2013 Nippon Telegraph and Telephone Corporation.
# Copyright (C) 2013 YAMAMOTO Takashi <yamamoto at valinux co jp>
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#    http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions and
# limitations under the License.
```

```
# a simple ICMP Echo Responder

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 arp
from ryu.lib.packet import ipv4
from ryu.lib.packet import icmp


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

    def __init__(self, *args, **kwargs):
        super(IcmpResponder, self).__init__(*args, **kwargs)
        self.hw_addr = '0a:e4:1c:d1:3e:44'
        self.ip_addr = '192.0.2.9'

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures , CONFIG_DISPATCHER)
    def _switch_features_handler(self, ev):
        msg = ev.msg
        datapath = msg. datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        actions = [parser.OFPActionOutput(port=ofproto.OFPP_CONTROLLER,
                                          max_len= ofproto. OFPCML_NO_BUFFER)]
        inst = [parser.OFPInstructionActions(type_=ofproto.OFPIT_APPLY_ACTIONS,
                                             actions=actions)]
        mod = parser.OFPFlowMod(datapath=datapath,
                                priority   =0   ,
                                match=parser. OFPMatch (),
                                instructions= inst)
        datapath. send_msg(mod)

    @set_ev_cls(ofp_event.EventOFPPacketIn , MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        msg = ev.msg
        datapath = msg. datapath
        port = msg.match['in_port']
        pkt  =  packet.  Packet(  data=  msg.  data)
        self.logger.info("packet-in %s" % (pkt,))
        pkt_ethernet = pkt.get_protocol(ethernet.ethernet)
        if not pkt_ethernet:
            return
        pkt_arp = pkt.get_protocol(arp.arp)
        if pkt_arp:
            self._handle_arp(datapath, port, pkt_ethernet, pkt_arp)
            return
        pkt_ipv4 = pkt.get_protocol(ipv4.ipv4)
        pkt_icmp = pkt.get_protocol(icmp.icmp)
        if pkt_icmp:
            self._handle_icmp(datapath, port, pkt_ethernet, pkt_ipv4, pkt_icmp)
            return

    def _handle_arp(self, datapath, port, pkt_ethernet, pkt_arp):
```

```
        if pkt_arp.opcode != arp.ARP_REQUEST:
            return
        pkt            =            packet.           Packet           ()
        pkt.add_protocol(ethernet.ethernet(ethertype=pkt_ethernet.ethertype,
                                           dst=pkt_ethernet.src,
                                           src=self.hw_addr))
        pkt.add_protocol(arp.arp(opcode=arp.ARP_REPLY,
                                 src_mac=self.hw_addr,
                                 src_ip=self.ip_addr,
                                 dst_mac=pkt_arp.src_mac,
                                 dst_ip=pkt_arp.src_ip))
        self._send_packet(datapath, port, pkt)

    def _handle_icmp(self, datapath, port, pkt_ethernet, pkt_ipv4, pkt_icmp):
        if pkt_icmp.type != icmp.ICMP_ECHO_REQUEST:
            return
        pkt            =            packet.           Packet           ()
        pkt.add_protocol(ethernet.ethernet(ethertype=pkt_ethernet.ethertype,
                                           dst=pkt_ethernet.src,
                                           src=self.hw_addr))
        pkt.add_protocol(ipv4.ipv4(dst=pkt_ipv4.src,
                                   src= self. ip_addr ,
                                   proto=pkt_ipv4.proto))
        pkt.add_protocol(icmp.icmp(type_=icmp.ICMP_ECHO_REPLY,
                                   code=icmp.ICMP_ECHO_REPLY_CODE,
                                   csum=0,
                                   data=pkt_icmp.data))
        self._send_packet(datapath, port, pkt)

    def _send_packet(self, datapath, port, pkt):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        pkt.serialize()
        self.logger.info("packet-out %s" % (pkt,))
        data = pkt.data
        actions = [parser.OFPActionOutput(port=port)]
        out = parser.OFPPacketOut(datapath=datapath,
                                  buffer_id=ofproto.OFP_NO_BUFFER,
                                  in_port=ofproto.OFPP_CONTROLLER,
                                  actions=actions,
                                  data=data)
        datapath.send_msg(out)
```

备注: OpenFlow 1.2 版本之后，因为 match 而带来的 Packet-In 讯息中，将会带有已经被解析过的资讯。但是这些信息的多寡以及详细程度要看每一台交换器的实际处理决定。例如 Open vSwitch 仅放入最低需求的信息，在大多数的情况下 Controller 需要针对资料再进行处理。反之 LINC 则尽可能放入信息。

以下是使用“ping -c 3”时所产生的记录挡（log）。

```
EVENT ofp_event->IcmpResponder EventOFPSwitchFeatures
switch features ev version: 0x4 msg_type 0x6 xid 0xb63c802c
OFPSwitchFeatures( auxiliary_id=0,capabilities=71,datapath_id=11974852296259,
n_buffers=256,n_tables
=254)
move onto main mode
EVENT ofp_event->IcmpResponder EventOFPPacketIn
packet-in ethernet(dst='ff:ff:ff:ff:ff:ff',ethertype=2054,src='0a:e4:1c:d1:3e:43'),
 arp(dst_ip='192.0.2.9',dst_mac='00:00:00:00:00:00',hlen=6,hwtype=1,opcode=1,plen
=4,proto=2048,src_ip='192.0.2.99',src_mac='0a:e4:1c:d1:3e:43'), '\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
packet-out ethernet(dst='0a:e4:1c:d1:3e:43',ethertype=2054,src='0a:e4:1c:d1:3e:44')
,arp(dst_ip='192.0.2.99',dst_mac='0a:e4:1c:d1:3e:43',hlen=6,hwtype=1,opcode=2,plen
=4,proto=2048,src_ip='192.0.2.9',src_mac='0a:e4:1c:d1:3e:44')
```

```
packet-in ethernet(dst='0a:e4:1c:d1:3e:44',ethertype=2048,src='0a:e4:1c:d1:3e:43'),
 ipv4(csum=47390,dst='192.0.2.9',flags=0,header_length=5,identification=32285,
offset=0,option=None,proto=1,src='192.0.2.99',tos=0,total_length=84,ttl=255,version
=4), icmp(code=0,csum=38471,data=echo(data='S,B\x00\x00\x00\x00\x00\x03L)(\x00\x00\
x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()
*+,-./\x00\x00\x00\x00\x00\x00\x00\x00',id=44565,seq=0),type=8)
packet-out ethernet(dst='0a:e4:1c:d1:3e:43',ethertype=2048,src='0a:e4:1c:d1:3e:44')
,ipv4(csum=14140,dst='192.0.2.99',flags=0,header_length=5,identification=0,offset
=0,option=None,proto=1,src='192.0.2.9',tos=0,total_length=84,ttl=255,version=4),
icmp(code=0,csum=40519,data=echo(data='S,B\x00\x00\x00\x00\x00\x03L)(\x00\x00\x00\
x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()
*+,-./\x00\x00\x00\x00\x00\x00\x00\x00',id=44565,seq=0),type=0)
EVENT ofp_event->IcmpResponder EventOFPPacketIn
packet-in ethernet(dst='0a:e4:1c:d1:3e:44',ethertype=2048,src='0a:e4:1c:d1:3e:43'),
 ipv4(csum=47383,dst='192.0.2.9',flags=0,header_length=5,identification=32292,
offset=0,option=None,proto=1,src='192.0.2.99',tos=0,total_length=84,ttl=255,version
=4), icmp(code=0,csum=12667,data=echo(data='T,B\x00\x00\x00\x00\x00Q\x17?(\x00\x00\
x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()
*+,-./\x00\x00\x00\x00\x00\x00\x00\x00',id=44565,seq=1),type=8)
packet-out ethernet(dst='0a:e4:1c:d1:3e:43',ethertype=2048,src='0a:e4:1c:d1:3e:44')
,ipv4(csum=14140,dst='192.0.2.99',flags=0,header_length=5,identification=0,offset
=0,option=None,proto=1,src='192.0.2.9',tos=0,total_length=84,ttl=255,version=4),
icmp(code=0,csum=14715,data=echo(data='T,B\x00\x00\x00\x00\x00Q\x17?(\x00\x00\x00\
x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()
*+,-./\x00\x00\x00\x00\x00\x00\x00\x00',id=44565,seq=1),type=0)
EVENT ofp_event->IcmpResponder EventOFPPacketIn
packet-in ethernet(dst='0a:e4:1c:d1:3e:44',ethertype=2048,src='0a:e4:1c:d1:3e:43'),
 ipv4(csum=47379,dst='192.0.2.9',flags=0,header_length=5,identification=32296,
offset=0,option=None,proto=1,src='192.0.2.99',tos=0,total_length=84,ttl=255,version
=4), icmp(code=0,csum=26863,data=echo(data='U,B\x00\x00\x00\x00\x00!\xa26(\x00\x00\
x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()
*+,-./\x00\x00\x00\x00\x00\x00\x00\x00',id=44565,seq=2),type=8)
packet-out ethernet(dst='0a:e4:1c:d1:3e:43',ethertype=2048,src='0a:e4:1c:d1:3e:44')
,ipv4(csum=14140,dst='192.0.2.99',flags=0,header_length=5,identification=0,offset
=0,option=None,proto=1,src='192.0.2.9',tos=0,total_length=84,ttl=255,version=4),
icmp(code=0,csum=28911,data=echo(data='U,B\x00\x00\x00\x00\x00!\xa26(\x00\x00\x00\
x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()
*+,-./\x00\x00\x00\x00\x00\x00\x00\x00',id=44565,seq=2),type=0)
```

IP fragments 将会是使用者需要解决的课题。由于 OpenFlow 协议本身并没有提供得到 MTU 资讯的方法，目前仅能使用其他方法解决。例如固定程序代码（hard-code）。另外，因为 Ryu 封包函式库会对所有的封包进行解析或串行化，你将会需要使用 API 来处理封包断裂（fragmented）的问题。

CHAPTER 9

# OF-Config 函式库

本章将介绍 Ryu 内建的 OF-Config 客户端函式库。

## 9.1 OF-Config 通讯协议

OF-Config 是用来管理 OpenFlow 交换器的一个通讯协议。OF-Config 通讯协议被定义在 NETCONF（RFC 6241）的标准中，它可以对逻辑交换器的通讯端口（Port）和队列（Queue）进行设定以 及数据撷取。

OF-Config 是被同样制订 OpenFlow 的 ONF（Open Network Foundation）所研拟，请参考下列数据以取得更详尽的信息。

https://www.opennetworking.org/sdn-resources/onf-specifications/openflow-config

Ryu 提供的函式库完全兼容于 OF-Config 1.1.1 版本

备注: 目前 Open vSwitch 并不支持 OF-Config，仅提供 OVSDB 作为替代使用。由于 OF-Config 还算是比较新的规格，因此 Open vSwitch 的 OVSDB 并不实作 OF-Config。

OVSDB 通讯协议虽然公开规范在 RFC 7047 作为标准，但事实上目前仅作为 Open vSwitch 专用的通讯协议。而 OF-Config 相对来说还是相对新的协议，期望在不久的将来会有更多实作它的 OpenFlow 交换器出现。

## 9.2 函式库架构

### 9.2.1 ryu.lib.of_config.capable_switch.OFCapableSwitch Class

本 Class 主要用来处理 NETCONF 会话（Session）。

```
from ryu.lib.of_config.capable_switch import OFCapableSwitch
```

### 9.2.2 ryu.lib.of_config.classes 模块（Module）

本模块用来将协议相关的设定对应至 Python 对象。

备注: 类别名称基本上遵照 OF-Config 1.1.1 中 yang specification 的 Grouping 关键词名称来命名。

例如：OFPortType

```
import ryu.lib.of_config.classes as ofc
```

## 9.3 使用范例

### 9.3.1 交换器的连结

使用 SSH Transport 联机到交换器。回呼（callback）函式 unknown_host_cb 是用来对应未知的 SSH Host Key 时所被执行的函式。下面的范例中我们使用无条件信任对方并继续进行连结。

```
sess =
    OFCapableSwitch( h
    ost='localhost',
    port =1830 ,
    username='linc',
    password='linc',
```

### 9.3.2 GET

使用 NETCONF GET 来取得交换器的状态。下面的范例将会用 /resources/port/resource-id 和 /resources/port/current-rate 表示所有端口的状态。

```
csw = sess.get()
for p in csw.resources.port:
    print p.resource_id, p.current_rate
```

### 9.3.3 GET-CONFIG

下面的范例是使用 NETCONF GET-CONFIG 来取得目前的交换器设定值。

备注: running 用来表示现在储存在 NETCONF 中目前的设定状态。但这跟交换器的实作有关，或者你也可以储存相关设定在 startup（设备启动时）或 candidate（Candidate set）。

其结果会使用 /resources/port/resource-id 和 /resources/port/configuration/admin-state 表示所有端口的状态。

```
csw = sess.get_config('running')
for p in csw.resources.port:
    print p.resource_id, p.configuration.admin_state
```

### 9.3.4 EDIT-CONFIG

这个范例说明如何使用 NETCONF EDIT-CONFIG 来对设定进行变更。首先使用 GET-CONFIG 取得交换器的设定，进行相关的编辑动作，最后使用 EDIT-CONFIG 将变更传送至交换器。

备注: 另外也可以使用 EDIT-CONFIG 直接修改部分的设定，这样做将更为安全。

将全部的端口状态在 /resources/port/configuration/admin-state 中设定为 down。

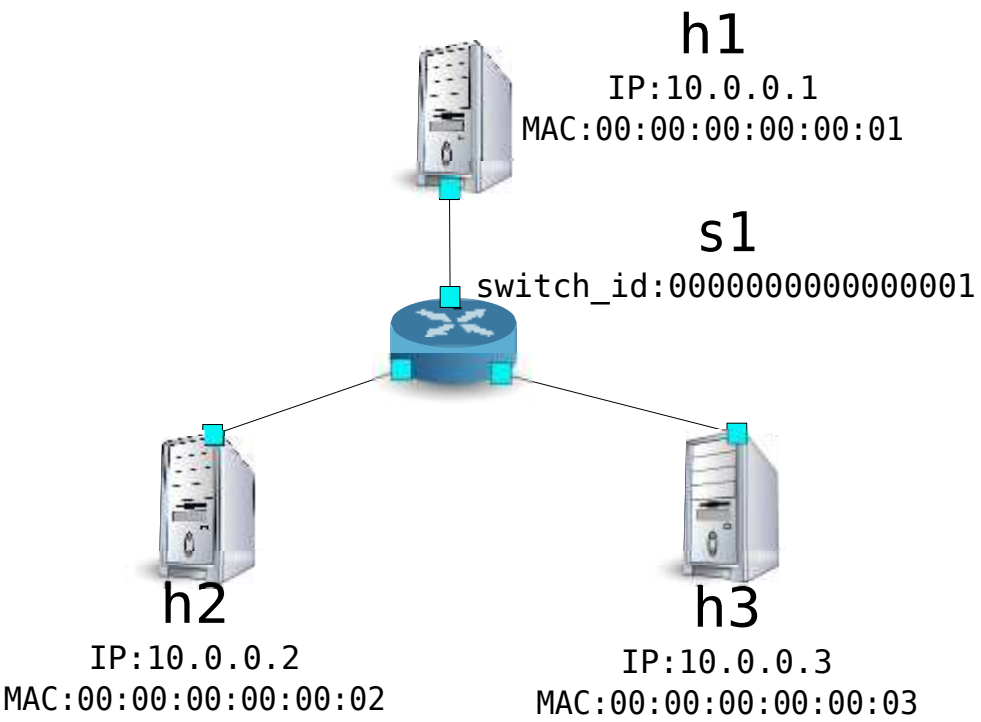
```
csw = sess.get_config('running')
for p in csw.resources.port:
    p.configuration.admin_state = 'down'
sess.edit_config('running', csw)
```

# 防火墙（Firewall ）

本章将说明如何利用 REST 的方式使用防火墙。

## 10.1 Single tenant 操作范例

以下说明如何建立一个如下所示的拓璞，并且对交换器 s1进行路由的增加和删除。




### 10.1.1 环境构筑

首先在 Mininet 上建构环境。所要输入的指令跟「交换器（Switching Hub ）」是一样的。

```
ryu@ryu-vm:~$ sudo mn --topo single,3 --mac --switch ovsk --controller remote -x
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1
```

```
*** Adding links:
(h1, s1) (h2, s1) (h3, s1)
*** Configuring hosts
h1 h2 h3
*** Running terms on localhost:10.0
*** Starting controller
*** Starting 1 switches
s1

*** Starting CLI:
mininet>
```

接着建立一个新的 xterm 用来操作 Controller。

```
mininet> xterm c0
mininet>
```

将 OpenFlow 的版本设定为 1.3。

switch: s1 (root):

```
root@ryu-vm:~# ovs-vsctl set Bridge s1 protocols=OpenFlow13
```

最后在控制 Controller 的 xterm 上启动 rest_firewall。

controller: c0 (root):

```
root@ryu-vm:~# ryu-manager ryu.app.rest_firewall
loading app ryu.app.rest_firewall
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
creating context wsgi
instantiating app ryu.app.rest_firewall of RestFirewallAPI
instantiating app ryu.controller.ofp_handler of OFPHandler
(2210) wsgi starting up on http://0.0.0.0:8080/
```

Ryu 和交换器中间的联机已经完成后，会出现下面的讯息。

controller: c0 (root):

```
[FW][INFO] switch_id=0000000000000001: Join as firewall
```

### 10.1.2 改变初始状态

防火墙启动后，在初始状态下全部的网络都会处于无法联机的状态。接下来我们要下指令使其生效，并开放网络的联机。

备注: 接下来的说明会使用到 REST API，若需要详细的解释请参考本章结尾的「REST API 列表」。

Node: c0 (root):

```
root@ryu-vm:~# curl -X PUT http://localhost:8080/firewall/module/enable
/0000000000000001
  [
    {
      "switch_id": "0000000000000001",
      " command_result ":
        { "result": "
        success",
```

```
      }
    }
  ]

root@ryu-vm:~# curl http://localhost:8080/firewall/module/status
  [
    {
      "   status   ":   "   enable",
      "switch_id": "0000000000000001"
    }
  ]
```

备注: REST命令执行的结果已经被格式为较为容易理解的格式。

确认可以从 h1 向 h2 执行 ping 指令。但是存取的权限规则并没有被设定，所以目前是处于无法连通的状态。

host: h1:

```
root@ryu-vm:~# ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
^C
--- 10.0.0.2 ping statistics ---
20 packets transmitted, 0 received, 100% packet loss, time 19003ms
```

封包被阻挡的过程被写进记录挡（log ）中。

controller: c0 (root):

```
[FW][INFO] dpid=0000000000000001: Blocked packet = ethernet(dst
='00:00:00:00:00:02',ethertype=2048,src='00:00:00:00:00:01'),ipv4(csum=9895,dst
='10.0.0.2',flags=2,header_length=5,identification=0,offset=0,option=None,proto=1,
src='10.0.0.1',tos=0,total_length=84,ttl=64,version=4),icmp(code=0,csum=55644,data
=echo(data='K\x8e\xaeR\x00\x00\x00\x00=\xc6\r\x00\x00\x00\x00\x00\x10\x11\x12\x13\
x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567',id=6952,
seq=1),type=8)
...
```

### 10.1.3 新增规则

增加 h1 和 h2 之间允许 ping 发送的规则。不论是从哪个方向都需要加入。

接下来新增规则，规则的编号会自动编码。

| 来源 | 目的 | 通讯协议 | 联机状态 | 规则 ID |
|---|---|---|---|---|
| 10.0.0.1/32 | 10.0.0.2/32 | ICMP | 通过 | 1 |
| 10.0.0.2/32 | 10.0.0.1/32 | ICMP | 通过 | 2 |

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"nw_src": "10.0.0.1/32", "nw_dst": "10.0.0.2/32",
"nw_proto": "ICMP"}' http://localhost:8080/firewall/rules/0000000000000001
  [
    {
      "switch_id": "0000000000000001",
      "command_result": [
        {
          "result": "success",
          "details": "Rule added. : rule_id=1"
        }
```

```
      ]
    }
  ]

root@ryu-vm:~# curl -X POST -d '{"nw_src": "10.0.0.2/32", "nw_dst": "10.0.0.1/32",
"nw_proto": "ICMP"}' http://localhost:8080/firewall/rules/0000000000000001
  [
    {
      "switch_id": "0000000000000001",
      "command_result": [
        {
          "result": "success",
          "details": "Rule added. : rule_id=2"
        }
      ]
    }
  ]
```

新增加的规则做为 FlowEntry 被注册到交换器中。

switch: s1 (root):

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=823.705s, table=0, n_packets=10, n_bytes=420, priority=65534,
arp actions=NORMAL
 cookie=0x0, duration=542.472s, table=0, n_packets=20, n_bytes=1960, priority=0
actions=CONTROLLER:128
 cookie=0x1, duration=145.05s, table=0, n_packets=0, n_bytes=0, priority=1,icmp,
nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=NORMAL
 cookie=0x2, duration=118.265s, table=0, n_packets=0, n_bytes=0, priority=1,icmp,
nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=NORMAL
```

接着 h2 和 h3 之间，新增加规则允许包含 ping 的所有 ipv4 封包通过。

| 来源 | 目的 | 通讯协议 | 联机状态 | 规则 ID |
|---|---|---|---|---|
| 10.0.0.2/32 | 10.0.0.3/32 | any | 通过 | 3 |
| 10.0.0.3/32 | 10.0.0.2/32 | any | 通过 | 4 |

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"nw_src": "10.0.0.2/32", "nw_dst": "10.0.0.3/32"}'
 http://localhost:8080/firewall/rules/0000000000000001
  [
    {
      "switch_id": "0000000000000001",
      "command_result": [
        {
          "result": "success",
          "details": "Rule added. : rule_id=3"
        }
      ]
    }
  ]

root@ryu-vm:~# curl -X POST -d '{"nw_src": "10.0.0.3/32", "nw_dst": "10.0.0.2/32"}'
 http://localhost:8080/firewall/rules/0000000000000001
  [
    {
      "switch_id": "0000000000000001",
      "command_result": [
        {
          "result": "success",
```

```
          "details": "Rule added. : rule_id=4"
        }
      ]
    }
  ]
```

新增的规则作为 FlowEntry 被注册到交换器当中。

switch: s1 (root):

```
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x3, duration=12.724s, table=0, n_packets=0, n_bytes=0, priority=1,ip,
nw_src=10.0.0.2,nw_dst=10.0.0.3  actions=NORMAL
 cookie=0x4, duration=3.668s, table=0, n_packets=0, n_bytes=0, priority=1,ip,nw_src
=10.0.0.3,nw_dst=10.0.0.2 actions=NORMAL
 cookie=0x0, duration=1040.802s, table=0, n_packets=10, n_bytes=420, priority
=65534,arp actions=NORMAL
 cookie=0x0, duration=759.569s, table=0, n_packets=20, n_bytes=1960, priority=0
actions=CONTROLLER:128
 cookie=0x1, duration=362.147s, table=0, n_packets=0, n_bytes=0, priority=1,icmp,
nw_src=10.0.0.1,nw_dst=10.0.0.2  actions=NORMAL
 cookie=0x2, duration=335.362s, table=0, n_packets=0, n_bytes=0, priority=1,icmp,
nw_src=10.0.0.2,nw_dst=10.0.0.1  actions=NORMAL
```

可以设定规则的优先权。

新增阻断 h2 和 h3 之间的 ping（ICMP）封包规则。优先权的默认值设定为大于 1 的值。

| 优先权 | 来源 | 目的 | 通讯协议 | 联机状态 | 规则 ID |
|---|---|---|---|---|---|
| 10 | 10.0.0.2/32 | 10.0.0.3/32 | ICMP | 中断 | 5 |
| 10 | 10.0.0.3/32 | 10.0.0.2/32 | ICMP | 中断 | 6 |

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d  '{"nw_src": "10.0.0.2/32", "nw_dst": "10.0.0.3/32",
 "nw_proto": "ICMP", "actions": "DENY", "priority": "10"}' http://localhost:8080/
firewall/rules/0000000000000001
  [
    {
      "switch_id": "0000000000000001",
      "command_result": [
        {
          "result": "success",
          "details": "Rule added. : rule_id=5"
        }
      ]
    }
  ]

root@ryu-vm:~# curl -X POST -d  '{"nw_src": "10.0.0.3/32", "nw_dst": "10.0.0.2/32",
 "nw_proto": "ICMP", "actions": "DENY", "priority": "10"}' http://localhost:8080/
firewall/rules/0000000000000001
  [
    {
      "switch_id": "0000000000000001",
      "command_result": [
        {
          "result": "success",
          "details": "Rule added. : rule_id=6"
        }
      ]
    }
  ]
```

新增的规则做为 Flow Entry 注册到交换器当中。

switch: s1 (root):

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x3, duration=242.155s, table=0, n_packets=0, n_bytes=0, priority=1,ip,
nw_src=10.0.0.2,nw_dst=10.0.0.3 actions=NORMAL
 cookie=0x4, duration=233.099s, table=0, n_packets=0, n_bytes=0, priority=1,ip,
nw_src=10.0.0.3,nw_dst=10.0.0.2 actions=NORMAL
 cookie=0x0, duration=1270.233s, table=0, n_packets=10, n_bytes=420, priority
=65534,arp actions=NORMAL
 cookie=0x0, duration=989s, table=0, n_packets=20, n_bytes=1960, priority=0 actions
=CONTROLLER:128
 cookie=0x5, duration=26.984s, table=0, n_packets=0, n_bytes=0, priority=10,icmp,
nw_src=10.0.0.2,nw_dst=10.0.0.3 actions=CONTROLLER:128
 cookie=0x1, duration=591.578s, table=0, n_packets=0, n_bytes=0, priority=1,icmp,
nw_src=10.0.0.1,nw_dst=10.0.0.2 actions=NORMAL
 cookie=0x6, duration=14.523s, table=0, n_packets=0, n_bytes=0, priority=10,icmp,
nw_src=10.0.0.3,nw_dst=10.0.0.2 actions=CONTROLLER:128
 cookie=0x2, duration=564.793s, table=0, n_packets=0, n_bytes=0, priority=1,icmp,
nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=NORMAL
```

### 10.1.4 确认规则

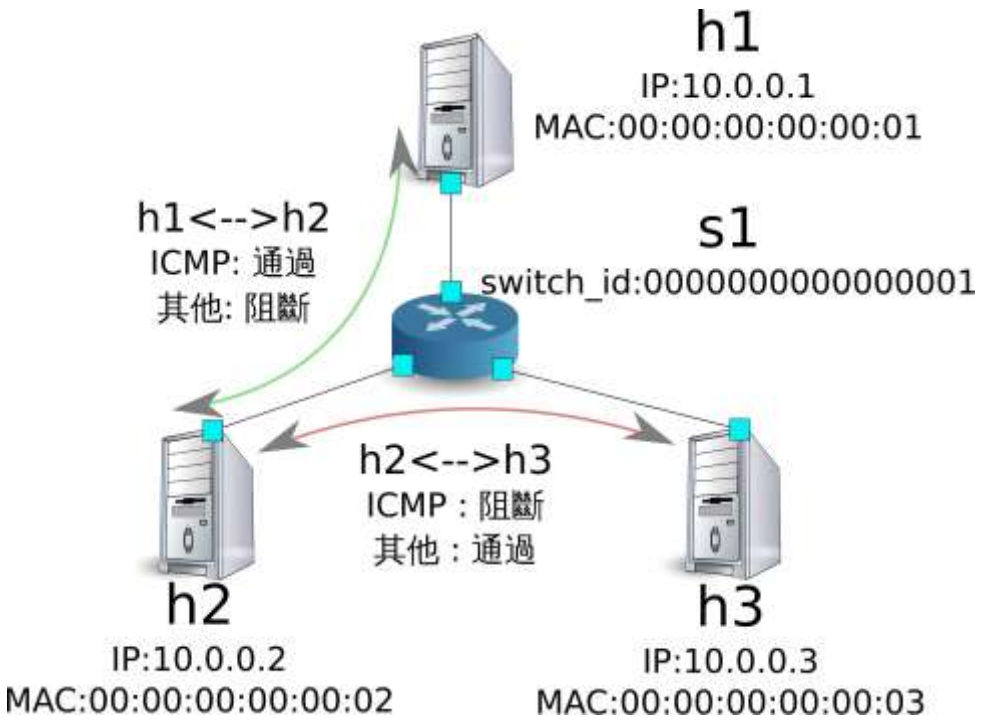
确认已经设定完成的规则。

Node: c0 (root):

```
root@ryu-vm:~# curl http://localhost:8080/firewall/rules/0000000000000001
  [
    {
      "access_control_list": [
        {
          "rules": [
            {
              "  priority  ":   1,
              "dl_type": "IPv4",
              "nw_dst": "10.0.0.3",
              "nw_src": "10.0.0.2",
              " rule_id ":  3,
              "actions": "ALLOW"
            },
            {
              "  priority  ":   1,
              "dl_type": "IPv4",
              "nw_dst": "10.0.0.2",
              "nw_src": "10.0.0.3",
              " rule_id ":  4,
              "actions": "ALLOW"
            },
            {
              "priority": 10,
              "dl_type": "IPv4",
              "nw_proto": "ICMP",
              "nw_dst": "10.0.0.3",
              "nw_src": "10.0.0.2",
              "rule_id": 5,
              "actions": "DENY"
            },
            {
              "priority": 1,
```

```
            "dl_type": "IPv4",
            "nw_proto": "ICMP",
            "nw_dst": "10.0.0.2",
            "nw_src": "10.0.0.1",
            " rule_id ":  1,
            "actions": "ALLOW"
          },
          {
            " priority ": 10 ,
            "dl_type": "IPv4",
            "nw_proto": "ICMP",
            "nw_dst": "10.0.0.2",
            "nw_src": "10.0.0.3",
            "rule_id": 6,
            "actions": "DENY"
          },
          {
            " priority ":  1,
            "dl_type": "IPv4",
            "nw_proto": "ICMP",
            "nw_dst": "10.0.0.1",
            "nw_src": "10.0.0.2",
            " rule_id ":  2,
            "actions": "ALLOW"
          }
        ]
      }
    ],
    "switch_id": "0000000000000001"
  }
]
```

设定完成的规则如下。




从 h1 向 h2 执行 ping 。如果允许的规则有被正确设定的话，ping 就可以正常联机。

host: h1:

```
root@ryu-vm:~# ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_req=1 ttl=64 time=0.419 ms
64 bytes from 10.0.0.2: icmp_req=2 ttl=64 time=0.047 ms
64 bytes from 10.0.0.2: icmp_req=3 ttl=64 time=0.060 ms
64 bytes from 10.0.0.2: icmp_req=4 ttl=64 time=0.033 ms
...
```

从 h1 发送到 h2 非 ping 的封包会被防火墙所阻挡。例如从 h1 发送到 h2 的 wget 指令就会被阻挡下来并记录在记录文件（log ）中。

host: h1:

```
root@ryu-vm:~# wget http://10.0.0.2
--2013-12-16 15:00:38--  http://10.0.0.2/
Connecting to 10.0.0.2:80... ^C
```

controller: c0 (root):

```
[FW][INFO] dpid=0000000000000001: Blocked packet = ethernet(dst
='00:00:00:00:00:02',ethertype=2048,src='00:00:00:00:00:01'),ipv4(csum=4812,dst
='10.0.0.2',flags=2,header_length=5,identification=5102,offset=0,option=None,proto
=6,src='10.0.0.1',tos=0,total_length=60,ttl=64,version=4),tcp(ack=0,bits=2,csum
=45753,dst_port=80,offset=10,option='\x02\x04\x05\xb4\x04\x02\x08\n\x00H:\x99\x00\
x00\x00\x00\x01\x03\x03\t',seq=1021913463,src_port=42664,urgent=0,window_size
=14600)
...
```

h2 和 h3 之间除了 ping 以外的封包则允许被通过。例如从 h2 向 h3 发送 ssh 指令，记录文件
（log ）中并不会出现封包被阻挡的记录（如果 ssh 是发送到 h3 以外的地点，则 ssh 的联机将会失败）。

host: h2:

```
root@ryu-vm:~# ssh 10.0.0.3
ssh: connect to host 10.0.0.3 port 22: Connection refused
```

从 h2 向 h3 发送 ping 指令，封包将会被防火墙所阻挡，并出现在记录文件（log ）中。

host: h2:

```
root@ryu-vm:~# ping 10.0.0.3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
^C
--- 10.0.0.3 ping statistics ---
8 packets transmitted, 0 received, 100% packet loss, time 7055ms
```

controller: c0 (root):

```
[FW][INFO] dpid=0000000000000001: Blocked packet = ethernet(dst
='00:00:00:00:00:03',ethertype=2048,src='00:00:00:00:00:02'),ipv4(csum=9893,dst
='10.0.0.3',flags=2,header_length=5,identification=0,offset=0,option=None,proto=1,
src='10.0.0.2',tos=0,total_length=84,ttl=64,version=4),icmp(code=0,csum=35642,data
=echo(data='\r\x12\xcaR\x00\x00\x00\x00\xab\x8b\t\x00\x00\x00\x00\x00\x10\x11\x12\
x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567',id
=8705,seq=1),type=8)
...
```

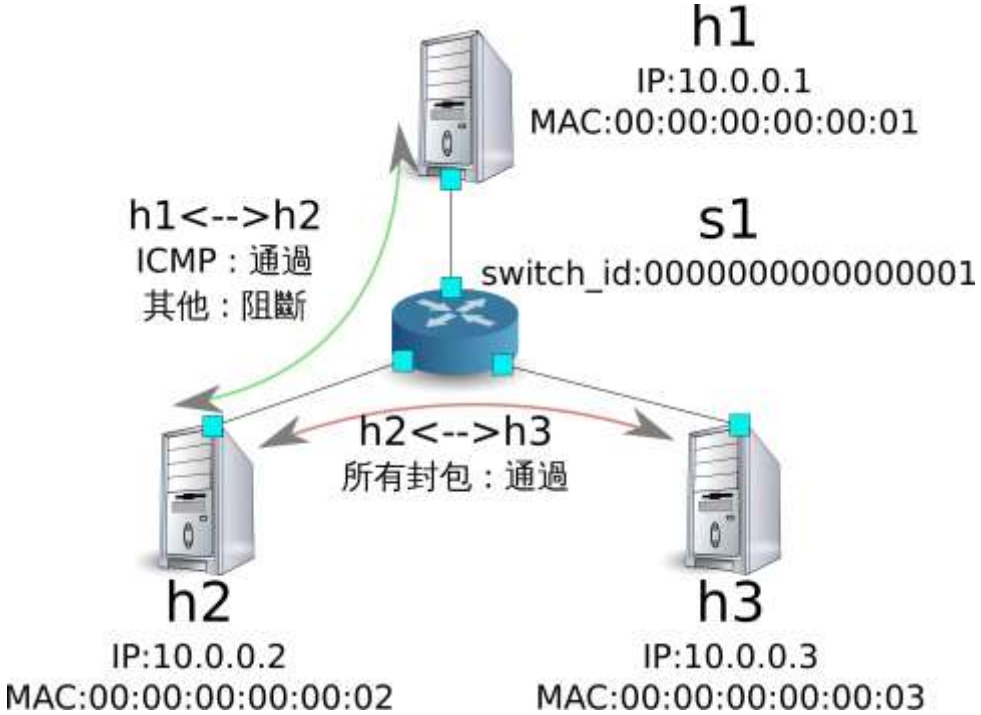
### 10.1.5 删除规则

删除 ``rule_id:5'' 和 ``rule_id:6'' 的规则。

Node: c0 (root):

```
root@ryu-vm:~# curl -X DELETE -d '{"rule_id": "5"}' http://localhost:8080/firewall/
rules /0000000000000001
  [
    {
      "switch_id": "0000000000000001",
      "command_result": [
        {
          "result": "success",
          "details": "Rule deleted. : ruleID=5"
        }
      ]
    }
  ]

root@ryu-vm:~# curl -X DELETE -d '{"rule_id": "6"}' http://localhost:8080/firewall/
rules /0000000000000001
  [
    {
      "switch_id": "0000000000000001",
      "command_result": [
        {
          "result": "success",
          "details": "Rule deleted. : ruleID=6"
        }
      ]
    }
  ]
```

现在的规则如下图所示。



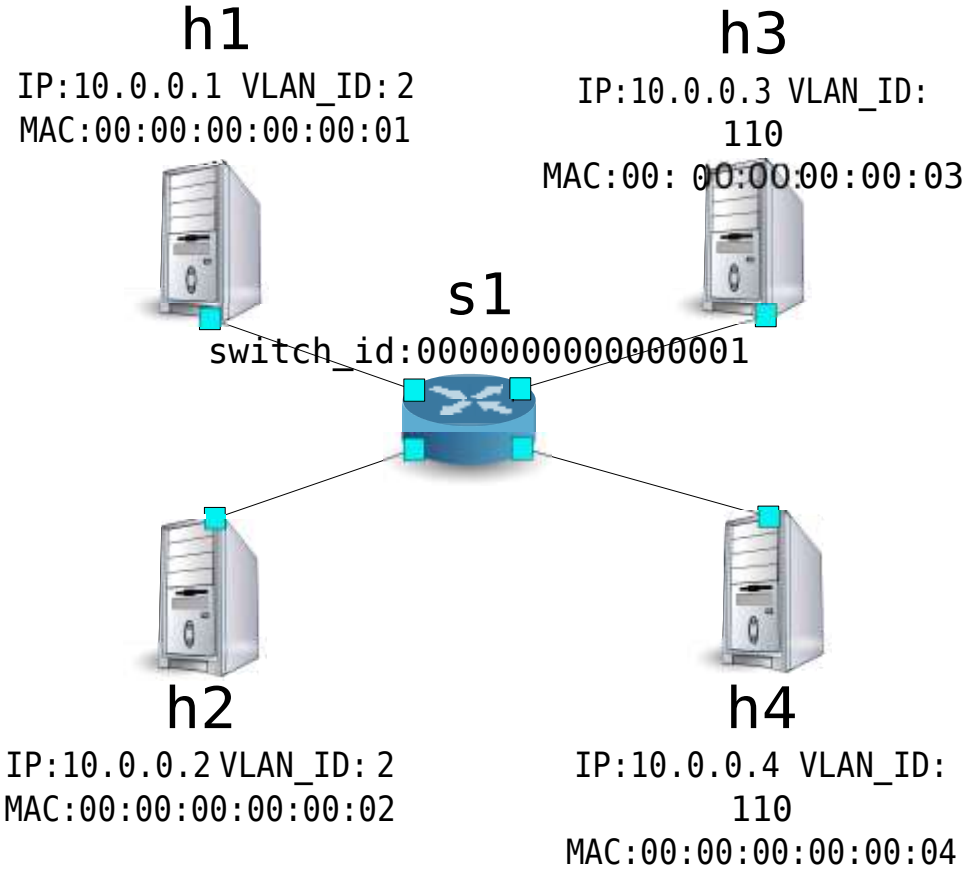

经实际确认。h2 和 h3 之间的 ping（ICMP）阻挡联机的规则删除后，ping 指令现在可以被正常执行并进行通讯。

host: h2:

```
root@ryu-vm:~# ping 10.0.0.3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_req=1 ttl=64 time=0.841 ms
64 bytes from 10.0.0.3: icmp_req=2 ttl=64 time=0.036 ms
64 bytes from 10.0.0.3: icmp_req=3 ttl=64 time=0.026 ms
64 bytes from 10.0.0.3: icmp_req=4 ttl=64 time=0.033 ms
...
```

## 10.2 Multi tenant 操作范例

接下来这个例子将建立拓璞并使用 VLAN 来对 tenants 进行处理，还有像是路由或是地址对于交换器 s1 对的新增或删除，以及每一个端口之间的连通做验证。




### 10.2.1 环境构筑

下面的例子使用 Single-tenant ，在 Mininet 上进行环境的建置，另外开启一个 xterm 做为控制 Controller 的方法，请注意与之前相比这边需要多一台 host。

```
ryu@ryu-vm:~$ sudo mn --topo single,4 --mac --switch ovsk --controller remote -x
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1
```

```
*** Adding links:
(h1, s1) (h2, s1) (h3, s1) (h4, s1)
*** Configuring hosts
h1 h2 h3 h4
*** Running terms on localhost:10.0
*** Starting controller
*** Starting 1 switches
s1

*** Starting CLI:
mininet> xterm c0
mininet>
```

接下来到每一个 host 的界面中设定 VLAN ID。

host: h1:

```
root@ryu-vm:~# ip addr del 10.0.0.1/8 dev h1-eth0
root@ryu-vm:~# ip link add link h1-eth0 name h1-eth0.2 type vlan id 2
root@ryu-vm:~# ip addr add 10.0.0.1/8 dev h1-eth0.2
root@ryu-vm:~# ip link set dev h1-eth0.2 up
```

host: h2:

```
root@ryu-vm:~# ip addr del 10.0.0.2/8 dev h2-eth0
root@ryu-vm:~# ip link add link h2-eth0 name h2-eth0.2 type vlan id 2
root@ryu-vm:~# ip addr add 10.0.0.2/8 dev h2-eth0.2
root@ryu-vm:~# ip link set dev h2-eth0.2 up
```

host: h3:

```
root@ryu-vm:~# ip addr del 10.0.0.3/8 dev h3-eth0
root@ryu-vm:~# ip link add link h3-eth0 name h3-eth0.110 type vlan id 110
root@ryu-vm:~# ip addr add 10.0.0.3/8 dev h3-eth0.110
root@ryu-vm:~# ip link set dev h3-eth0.110 up
```

host: h4:

```
root@ryu-vm:~# ip addr del 10.0.0.4/8 dev h4-eth0
root@ryu-vm:~# ip link add link h4-eth0 name h4-eth0.110 type vlan id 110
root@ryu-vm:~# ip addr add 10.0.0.4/8 dev h4-eth0.110
root@ryu-vm:~# ip link set dev h4-eth0.110 up
```

接着将使用的 OpenFlow 版本设定为 1.3。

switch: s1 (root):

```
root@ryu-vm:~# ovs-vsctl set Bridge s1 protocols=OpenFlow13
```

最后，从 controller 的 xterm 画面中启动 rest_firewall 。

controller: c0 (root):

```
root@ryu-vm:~# ryu-manager ryu.app.rest_firewall
loading app ryu.app.rest_firewall
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
creating context wsgi
instantiating app ryu.app.rest_firewall of RestFirewallAPI
instantiating app ryu.controller.ofp_handler of OFPHandler
(13419) wsgi starting up on http://0.0.0.0:8080/
```

Ryu 和交换器之间的联机已经成功的话，就会出现接下来的讯息。

controller: c0 (root):

```
[FW][INFO] switch_id=0000000000000001: Join as firewall
```

### 10.2.2 变更初始状态

启动防火墙。

Node: c0 (root):

```
root@ryu-vm:~# curl -X PUT http://localhost:8080/firewall/module/enable
/0000000000000001
  [
    {
      "switch_id": "0000000000000001",
      " command_result ":
        { "result": "
        success",
        "details": "firewall running."
      }
    }
  ]

root@ryu-vm:~# curl http://localhost:8080/firewall/module/status
  [
    {
      "   status  ":   "   enable",
      "switch_id": "0000000000000001"
    }
```

### 10.2.3 新增规则

新增允许使用 VLAN_ID = 2 向 10.0.0.0/8 发送 ping 讯息（ICMP 封包）的规则到交换器中，设定双向的规则是必要的。

| 优先权 | VLAN ID | 来源 | 目的 | 通讯协议 | 联机状态 | 规则 ID |
|---|---|---|---|---|---|---|
| 1 | 2 | 10.0.0.0/8 | any | ICMP | 通过 | 1 |
| 1 | 2 | any | 10.0.0.0/8 | ICMP | 通过 | 2 |

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"nw_src": "10.0.0.0/8", "nw_proto": "ICMP"}' http
://localhost:8080/firewall/rules/0000000000000001/2
  [
    {
      "switch_id": "0000000000000001",
      "command_result": [
        {
          "result": "success",
          "vlan_id": 2,
          "details": "Rule added. : rule_id=1"
        }
      ]
    }
  ]

root@ryu-vm:~# curl -X POST -d '{"nw_dst": "10.0.0.0/8", "nw_proto": "ICMP"}' http
://localhost:8080/firewall/rules/0000000000000001/2
```

```
  [
    {
      "switch_id": "0000000000000001",
      "command_result": [
        {
          "result": "success",
          "vlan_id": 2,
          "details": "Rule added. : rule_id=2"
        }
      ]
    }
  ]
```

### 10.2.4 规则确认

确认已经设定的规则。

Node: c0 (root):

```
root@ryu-vm:~# curl http://localhost:8080/firewall/rules/0000000000000001/all
  [
    {
      "access_control_list": [
        {
          "rules": [
            {
              " priority ":  1,
              "dl_type": "IPv4",
              "nw_proto": "ICMP",
              "dl_vlan": 2,
              "nw_src": "10.0.0.0/8",
              "   rule_id   ":    1,
              "actions": "ALLOW"
            },
            {
              " priority ":  1,
              "dl_type": "IPv4",
              " nw_proto ": " ICMP",
              "nw_dst": "10.0.0.0/8",
              "dl_vlan": 2,
              " rule_id ":  2,
              "actions": "ALLOW"
            }
          ],
          "vlan_id": 2
        }
      ],
      "switch_id": "0000000000000001"
    }
  ]
```

让我们确认一下实际状况。在 VLAN_ID = 2 的情况下，从 h1 发送的 ping 在 h2 也同样是 VLAN_ID = 2 的情况下，你会发现他是连通的，因为我们刚才已经把规则加入。

host: h1:

```
root@ryu-vm:~# ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_req=1 ttl=64 time=0.893 ms
64 bytes from 10.0.0.2: icmp_req=2 ttl=64 time=0.098 ms
64 bytes from 10.0.0.2: icmp_req=3 ttl=64 time=0.122 ms
```

```
64 bytes from 10.0.0.2: icmp_req=4 ttl=64 time=0.047 ms
...
```

VLAN_ID = 110 的情况下 h3 和 h4 之间，由于规则没有被加入，所以 ping 封包被阻挡。

host: h3:

```
root@ryu-vm:~# ping 10.0.0.4
PING 10.0.0.4 (10.0.0.4) 56(84) bytes of data.
^C
--- 10.0.0.4 ping statistics ---
6 packets transmitted, 0 received, 100% packet loss, time 4999ms
```

封包被阻断的时候会被记录在记录文件（log ）之中。

controller: c0 (root):

```
[FW][INFO] dpid=0000000000000001: Blocked packet = ethernet(dst
='00:00:00:00:00:04',ethertype=33024,src='00:00:00:00:00:03'), vlan(cfi=0,ethertype
=2048,pcp=0,vid=110), ipv4(csum=9891,dst='10.0.0.4',flags=2,header_length=5,
identification=0,offset=0,option=None,proto=1,src='10.0.0.3',tos=0,total_length=84,
ttl=64,version=4), icmp(code=0,csum=58104,data=echo(data='\xb8\xa9\xaeR\x00\x00\x00
\x00\xce\xe3\x02\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\
x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./01234567',id=7760,seq=4),type=8)
...
```

本章中，透过具体的例子说明学到如何使用防火墙。

## 10.3 REST API 列表

本章说明中所提到的rest_firewall REST API 一览。

### 10.3.1 取得交换器的防火墙状态

| 方法 | GET |
|---|---|
| URL | /firewall/module/status |

### 10.3.2 变更交换器的防火墙状态

| 方法 | PUT |
|---|---|
| URL | /firewall/module/{ op }/{ switch }<br>-- op : [ ``enable" \| ``disable" ]<br>-- switch : [ ``all" \| 交换器 ID ] |
| 备注 | 交换器的初始状态均为 ``disable" |

### 10.3.3 取得全部规则

| 方法 | GET |
|---|---|
| URL | /firewall/rules/{ switch }[/{ vlan }]<br>-- switch : [ ``all" \| 交换器 ID ]<br>-- vlan: [ ``all" \| VLAN ID ] |
| 备注 | VLAN ID 的指定可选择加或不加。 |

### 10.3.4 新增规则

| 方法 | POST |
| --- | --- |
| URL | /firewall/rules/{ switch }[/{ vlan }]<br>-- switch : [ ``all" \| 交换器 ID ]<br>-- vlan : [ ``all" \| VLAN ID ] |
| 资料 | priority :[ 0 - 65535]<br>in_port :[ 0 - 65535 ]<br>dl_src :"<xx:xx:xx:xx:xx:xx>"<br>dl_dst :"<xx:xx:xx:xx:xx:xx>"<br>dl_type :[ ``ARP" \| ``IPv4" ]<br>nw_src :"<xxx.xxx.xxx.xxx/xx>"<br>nw_dst :"<xxx.xxx.xxx.xxx/xx">"<br>nw_proto ":[ ``TCP" \| ``UDP" \| ``ICMP" ]<br>tp_src :[ 0 - 65535 ]<br>tp_dst :[ 0 - 65535 ]<br>actions : [ ``ALLOW" \| ``DENY" ] |
| 备注 | 注册成功的规则会自动产生规则 ID，并注明在响应的讯息中。<br>指定VLAN ID为可附加之选项。 |

### 10.3.5 删除规则

| 方法 | DELETE |
| --- | --- |
| URL | /firewall/rules/{ switch }[/{ vlan }]<br>-- switch : [ ``all" \| 交换器 ID ]<br>-- vlan : [ ``all" \| VLAN ID ] |
| 资料 | rule_id : [ ``all" \| 1 - ... ] |
| 备注 | 指定VLAN ID为可附加之选项。 |

### 10.3.6 取得交换器的记录文件

| 方法 | GET |
| --- | --- |
| URL | /firewall/log/status |

### 10.3.7 变更交换器记录文件的状态

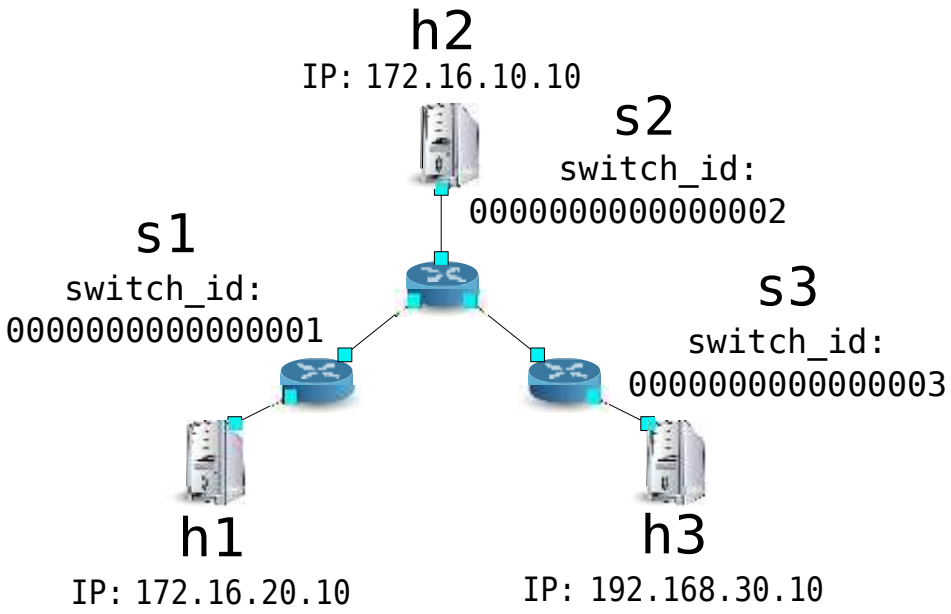
| 方法 | PUT |
| --- | --- |
| URL | /firewall/log/{ op }/{ switch }<br>-- op: [ ``enable" \| ``disable" ]<br>-- switch: [ ``all" \| 交换器 ID ] |
| 备注 | 设定每一个交换器的初始状态为”启用” |

# 路由器（Router）

本章将说明如何使用 REST 来设定一个路由器。

## 11.1 Single Tenant 的操作范例

下面的例子介绍如何建立拓璞，每个交换器（路由器）的地址新增或删除，及确认 host 之间的 联机状况确认。




### 11.1.1 环境建置

首先要在 Mininet 上建置环境。mn 的命令及参数如下。

| 名称 | 设定值 | 说明 |
|---|---|---|
| topo | linear,3 | 3 台交换器直接链接的网络拓璞 |
| mac | 无 | 自动设定各 host 的 MAC 地址 |
| switch | ovsk | 使用 Open vSwitch |
| controller | remote | 使用外部的 Controller 做为 OpenFlow controller |
| x | 无 | 启动 xterm |

执行的动作如下：

```
ryu@ryu-vm:~$ sudo mn --topo linear,3 --mac --switch ovsk --controller remote -x
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1 s2 s3
*** Adding links:
(h1, s1) (h2, s2) (h3, s3) (s1, s2) (s2, s3)
*** Configuring hosts
h1 h2 h3
*** Running terms on localhost:10.0
*** Starting controller
*** Starting 3 switches
s1 s2 s3

*** Starting CLI:
mininet>
```

接着，开启 Controller 所使用的 xterm。

```
mininet> xterm c0
mininet>
```

然后设定每个路由器的 OpenFlow 版本为 1.3。

switch: s1 (root):

```
root@ryu-vm:~# ovs-vsctl set Bridge s1 protocols=OpenFlow13
```

switch: s2 (root):

```
root@ryu-vm:~# ovs-vsctl set Bridge s2 protocols=OpenFlow13
```

switch: s3 (root):

```
root@ryu-vm:~# ovs-vsctl set Bridge s3 protocols=OpenFlow13
```

接着每一个 host 删除原先自动配置的 IP 地址，并设定新的 IP 地址。

host: h1:

```
root@ryu-vm:~# ip addr del 10.0.0.1/8 dev h1-eth0
root@ryu-vm:~# ip addr add 172.16.20.10/24 dev h1-eth0
```

host: h2:

```
root@ryu-vm:~# ip addr del 10.0.0.2/8 dev h2-eth0
root@ryu-vm:~# ip addr add 172.16.10.10/24 dev h2-eth0
```

host: h3:

```
root@ryu - vm:~# ip addr del 10.0.0.3/8 dev h3 - eth0
root@ryu-vm:~# ip addr add 192.168.30.10/24 dev h3-eth0
```

最后在操作 Controller 的 xterm 上启动 rest_router。

controller: c0 (root):

```
root@ryu-vm:~# ryu-manager ryu.app.rest_router
loading app ryu.app.rest_router
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
creating context wsgi
instantiating app ryu.app.rest_router of RestRouterAPI
instantiating app ryu.controller.ofp_handler of OFPHandler
(2212) wsgi starting up on http://0.0.0.0:8080/
```

若 Ryu 和交换器之间的连接成功，接下来的讯息将会被显示。

controller: c0 (root):

```
[RT][INFO] switch_id=0000000000000003: Set SW config for TTL error packet in.
[RT][INFO] switch_id=0000000000000003: Set ARP handling (packet in) flow [cookie=0
x0]
[RT][INFO] switch_id=0000000000000003: Set L2 switching (normal) flow [cookie=0x0]
[RT][INFO] switch_id=0000000000000003: Set default route (drop) flow [cookie=0x0]
[RT][INFO] switch_id=0000000000000003: Start cyclic routing table update.
[RT][INFO] switch_id=0000000000000003: Join as router.
...
```

上述的 log 表示 3 台交换器已经准备完成。

### 11.1.2 设定 IP 地址

设定每一个路由器的 IP 地址。

首先，设定交换器 s1 的 IP 地址为「172.16.20.1/24」和「172.16.30.30/24」。

备注: 接下来的说明中所使用的 REST API 请参考本章结尾的「REST API 列表」以取得更详细的资料。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"address":"172.16.20.1/24"}' http://localhost
:8080/router/0000000000000001
  [
    {
      "switch_id": "0000000000000001",
      "command_result": [
        {
          "result": "success",
          "details": "Add address [address_id=1]"
        }
      ]
    }
  ]

root@ryu-vm:~# curl -X POST -d '{"address": "172.16.30.30/24"}' http://localhost
:8080/router/0000000000000001
  [
    {
      "switch_id": "0000000000000001",
      "command_result": [
        {
          "result": "success",
          "details": "Add address [address_id=2]"
        }
      ]
```

```
    }
  ]
```

备注: REST命令的执行结果已经被整理为较好阅读的格式。

接着，设定交换器 s2 的 IP 位址为「172.16.10.1/24」、「172.16.30.1/24」和「192.168.10.1/24」。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"address":"172.16.10.1/24"}' http://localhost
:8080/router/0000000000000002
  [
    {
      "switch_id": "0000000000000002",
      "command_result": [
        {
          "result": "success",
          "details": "Add address [address_id=1]"
        }
      ]
    }
  ]

root@ryu-vm:~# curl -X POST -d '{"address": "172.16.30.1/24"}' http://localhost
:8080/router/0000000000000002
  [
    {
      "switch_id": "0000000000000002",
      "command_result": [
        {
          "result": "success",
          "details": "Add address [address_id=2]"
        }
      ]
    }
  ]

root@ryu-vm:~# curl -X POST -d '{"address": "192.168.10.1/24"}' http://localhost
:8080/router/0000000000000002
  [
    {
      "switch_id": "0000000000000002",
      "command_result": [
        {
          "result": "success",
          "details": "Add address [address_id=3]"
        }
      ]
    }
  ]
```

接着设定交换器 s3 的 IP 地址为「192.168.30.1/24」和「192.168.10.20/24」。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"address": "192.168.30.1/24"}' http://localhost
:8080/router/0000000000000003
  [
    {
      "switch_id": "0000000000000003",
      "command_result": [
        {
```

```
          "result": "success",
          "details": "Add address [address_id=1]"
        }
      ]
    }
  ]

root@ryu-vm:~# curl -X POST -d '{"address": "192.168.10.20/24"}' http://localhost
:8080/ router /0000000000000003
  [
    {
      "switch_id": "0000000000000003",
      "command_result": [
        {
          "result": "success",
          "details": "Add address [address_id=2]"
        }
      ]
    }
  ]
```

交換器的 IP 地址已经被设定完成，接着对每一个 host 新增预设的网关。

host: h1:

```
root@ryu-vm:~# ip route add default via 172.16.20.1
```

host: h2:

```
root@ryu-vm:~# ip route add default via 172.16.10.1
```

host: h3:

```
root@ryu-vm:~# ip route add default via 192.168.30.1
```

### 11.1.3 设定预设路由

设定每一个路由器的预设路由。 首先，设定

路由器 s1 的路由为路由器 s2 。 Node: c0

(root):

```
root@ryu-vm:~# curl -X POST -d '{"gateway": "172.16.30.1"}' http://localhost:8080/
router /0000000000000001
  [
    {
      "switch_id": "0000000000000001",
      "command_result": [
        {
          "result": "success",
          "details": "Add route [route_id=1]"
        }
      ]
    }
  ]
```

设定路由器 s2 的预设路由为路由器 s1。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"gateway": "172.16.30.30"}' http://localhost:8080/
router/0000000000000002
  [
    {
      "switch_id": "0000000000000002",
      "command_result": [
        {
          "result": "success",
          "details": "Add route [route_id=1]"
        }
      ]
    }
  ]
```

设定路由器 s3 的预设路由为路由器 s2。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"gateway": "192.168.10.1"}' http://localhost:8080/
router/0000000000000003
  [
    {
      "switch_id": "0000000000000003",
      "command_result": [
        {
          "result": "success",
          "details": "Add route [route_id=1]"
        }
      ]
    }
  ]
```

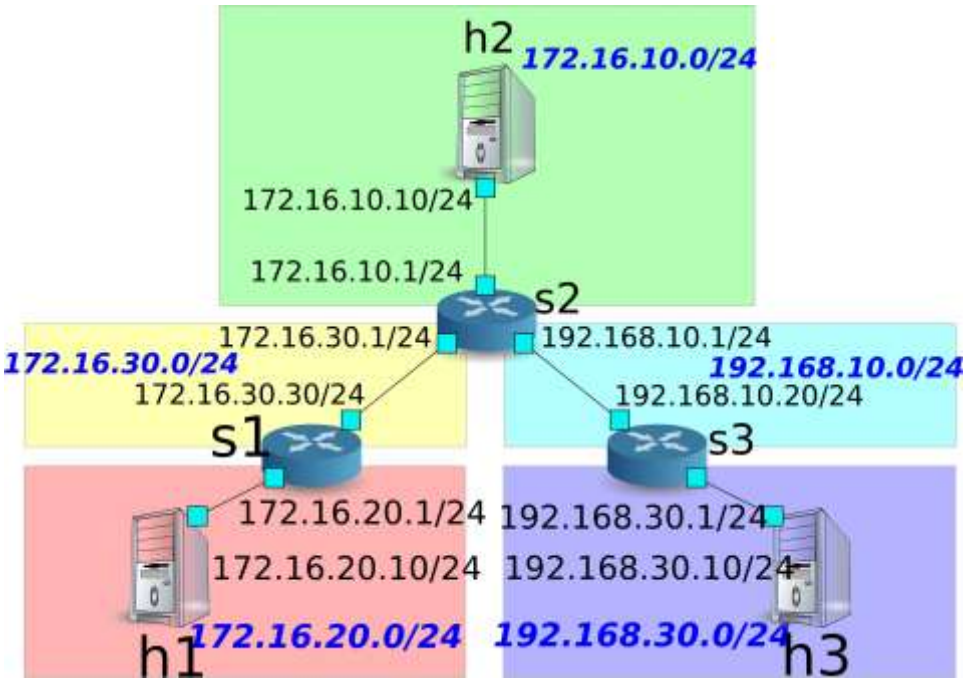
### 11.1.4 设定静态路由

为了路由器 s2，设定路由器 s3 的静态路由为（192.168.30.0/24 ）。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"destination": "192.168.30.0/24", "gateway":
"192.168.10.20"}'  http://localhost:8080/router/0000000000000002
  [
    {
      "switch_id": "0000000000000002",
      "command_result": [
        {
          "result": "success",
          "details": "Add route [route_id=2]"
        }
      ]
    }
  ]
```

IP 地址及路由的设定状态如下。




### 11.1.5 确认设定的内容

确认每一个路由器的内容。

Node: c0 (root):

```
root@ryu-vm:~# curl http://localhost:8080/router/0000000000000001
  [
    {
      "internal_network": [
        {
          "route": [
            {
              "    route_id    ":     1,
              "destination": "0.0.0.0/0",
              "gateway": "172.16.30.1"
            }
          ],
          "address": [
            {
              "address_id": 1,
              "address": "172.16.20.1/24"
            },
            {
              "address_id": 2,
              "address": "172.16.30.30/24"
            }
          ]
        }
      ],
      "switch_id": "0000000000000001"
    }
  ]

root@ryu-vm:~# curl http://localhost:8080/router/0000000000000002
  [
    {
      "internal_network": [
```

```
        {
          "route": [
            {
              "    route_id    ":     1,
              "destination": "0.0.0.0/0",
              "gateway": "172.16.30.30"
            },
            {
              "route_id": 2,
              "destination": "192.168.30.0/24",
              "gateway": "192.168.10.20"
            }
          ],
          "address": [
            {
              "address_id": 2,
              "address": "172.16.30.1/24"
            },
            {
              "address_id": 3,
              "address": "192.168.10.1/24"
            },
            {
              "address_id": 1,
              "address": "172.16.10.1/24"
            }
          ]
        }
      ],
      "switch_id": "0000000000000002"
    }
  ]

root@ryu-vm:~# curl http://localhost:8080/router/0000000000000003
  [
    {
      "internal_network": [
        {
          "route": [
            {
              "    route_id    ":     1,
              "destination": "0.0.0.0/0",
              "gateway": "192.168.10.1"
            }
          ],
          "address": [
            {
              "address_id": 1,
              "address": "192.168.30.1/24"
            },
            {
              "address_id": 2,
              "address": "192.168.10.20/24"
            }
          ]
        }
      ],
      "switch_id": "0000000000000003"
    }
  ]
```


在这样的状态下，执行 ping 来确认相互间的连接状态。首先执行从 h2 向 h3 执行 ping。确认

正常连通的状态。

host: h2:

```
root@ryu-vm:~# ping 192.168.30.10
PING 192.168.30.10 (192.168.30.10) 56(84) bytes of data.
64 bytes from 192.168.30.10: icmp_req=1 ttl=62 time=48.8 ms
64 bytes from 192.168.30.10: icmp_req=2 ttl=62 time=0.402 ms
64 bytes from 192.168.30.10: icmp_req=3 ttl=62 time=0.089 ms
64 bytes from 192.168.30.10: icmp_req=4 ttl=62 time=0.065 ms
...
```

接着，从 h2 向 h1 执行 ping。确认这边也是正常的连接状态。

host: h2:

```
root@ryu-vm:~# ping 172.16.20.10
PING 172.16.20.10 (172.16.20.10) 56(84) bytes of data.
64 bytes from 172.16.20.10: icmp_req=1 ttl=62 time=43.2 ms
64 bytes from 172.16.20.10: icmp_req=2 ttl=62 time=0.306 ms
64 bytes from 172.16.20.10: icmp_req=3 ttl=62 time=0.057 ms
64 bytes from 172.16.20.10: icmp_req=4 ttl=62 time=0.048 ms
...
```

### 11.1.6 删除静态路由

删除路由器 s2 上指向路由器 s3 的静态路由。

Node: c0 (root):

```
root@ryu-vm:~# curl -X DELETE -d '{"route_id": "2"}' http://localhost:8080/router
/0000000000000002
  [
    {
      "switch_id": "0000000000000002",
      "command_result": [
        {
          "result": "success",
          "details": "Delete route [route_id=2]"
        }
      ]
    }
  ]
```

确认路由器 s2 的设定。这边可以看到原先指向路由器 s3 的静态路由已经被删除了。

Node: c0 (root):

```
root@ryu-vm:~# curl http://localhost:8080/router/0000000000000002
  [
    {
      "internal_network": [
        {
          "route": [
            {
              "    route_id    ":     1,
              "destination": "0.0.0.0/0",
              "gateway": "172.16.30.30"
            }
          ],
          "address": [
            {
```

```
            "address_id": 2,
            "address": "172.16.30.1/24"
          },
          {
            "address_id": 3,
            "address": "192.168.10.1/24"
          },
          {
            "address_id": 1,
            "address": "172.16.10.1/24"
          }
        ]
      }
    ],
    "switch_id": "0000000000000002"
  }
]
```

在这个状态下，使用 ping 来确认连结状态。从 h2 向 h3 执行 ping 会发现无法通过连接测试，这是因为我们已经删除了路由的关系。

host: h2:

```
root@ryu-vm:~# ping 192.168.30.10
PING 192.168.30.10 (192.168.30.10) 56(84) bytes of data.
^C
--- 192.168.30.10 ping statistics ---
12 packets transmitted, 0 received, 100% packet loss, time 11088ms
```

### 11.1.7 删除 IP 地址

删除已经设定在路由器 s1 上的 IP 地址「172.16.20.1/24」。

Node: c0 (root):

```
root@ryu-vm:~# curl -X DELETE -d '{"address_id": "1"}' http://localhost:8080/router
/0000000000000001
[
  {
    "switch_id": "0000000000000001",
    "command_result": [
      {
        "result": "success",
        "details": "Delete address [address_id=1]"
      }
    ]
  }
]
```

确认路由器 s1 的设定状态。这边可以看到路由器 s1 中原先被设定的「172.16.20.1/24」已 经被删除。

Node: c0 (root):

```
root@ryu-vm:~# curl http://localhost:8080/router/0000000000000001
[
  {
    "internal_network": [
      {
        "route": [
          {
```

```
            "    route_id    ":     1,
            "destination": "0.0.0.0/0",
            "gateway": "172.16.30.1"
          }
        ],
        "address": [
          {
            "address_id": 2,
            "address": "172.16.30.30/24"
          }
        ]
      }
    ],
    "switch_id": "0000000000000001"
  }
]
```

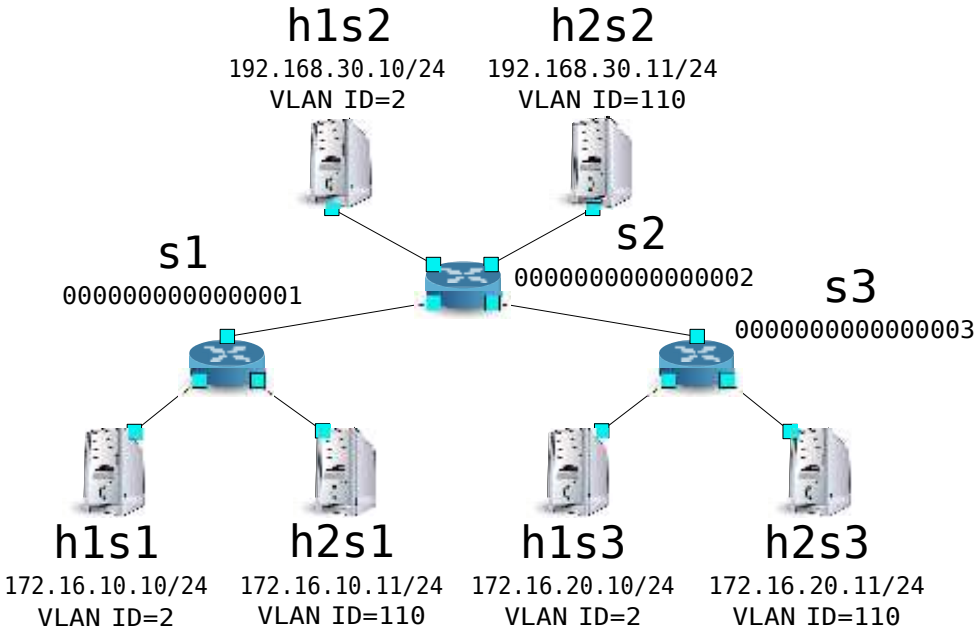
在这个状态下，使用 ping 指令来确认连通的状况。从 h2 向 h1 执行，这时可以发现由于 h1 的子网相关设定及路由已经被删除的关系，是无法连通的。

host: h2:

```
root@ryu-vm:~# ping 172.16.20.10
PING 172.16.20.10 (172.16.20.10) 56(84) bytes of data.
^C
--- 172.16.20.10 ping statistics ---
19 packets transmitted, 0 received, 100% packet loss, time 18004ms
```

## 11.2 Multi-tenant 的操作范例

接下来的例子将建立一个网络拓璞，使用VLAN来分割tenant的使用。对各个交换器（路由器）的地址或路由进行新增和删除，并确认每一个 host之间的连通状况。




### 11.2.1 环境建置

首先是在 Mininet 上进行环境的建置。mn 命令的参数如下。

| 参数 | 参数值 | 说明 |
| --- | --- | --- |
| topo | linear, 3,2 | 3 台交换器直接链接的网络拓璞（每个交换器连接两台 host） |
| mac | 无 | 自动设定每一个 host 的 MAC 地址 |
| switch | ovsk | 使用 OpenvSwitch |
| controller | remote | 使用外部的 OpenFlow Controller |
| x | 无 | 启动 xterm |

执行的范例如下。

```
ryu@ryu-vm:~$ sudo mn --topo linear,3,2 --mac --switch ovsk --controller remote -x
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
h1s1 h1s2 h1s3 h2s1 h2s2 h2s3
*** Adding switches:
s1 s2 s3
*** Adding links:
(h1s1, s1) (h1s2, s2) (h1s3, s3) (h2s1, s1) (h2s2, s2) (h2s3, s3) (s1, s2) (s2, s3)
*** Configuring hosts
h1s1 h1s2 h1s3 h2s1 h2s2 h2s3
*** Running terms on localhost:10.0
*** Starting controller
*** Starting 3 switches
s1 s2 s3
*** Starting CLI:
mininet>
```

接着启动 Controller 用的 xterm。

```
mininet> xterm c0
mininet>
```

然后，将每一台路由器所使用的 OpenFlow 版本设定为 1.3。

switch: s1 (root):

```
root@ryu-vm:~# ovs-vsctl set Bridge s1 protocols=OpenFlow13
```

switch: s2 (root):

```
root@ryu-vm:~# ovs-vsctl set Bridge s2 protocols=OpenFlow13
```

switch: s3 (root):

```
root@ryu-vm:~# ovs-vsctl set Bridge s3 protocols=OpenFlow13
```

之后设定每一个 host 的 VLAN ID 和 IP 地址。

host: h1s1:

```
root@ryu-vm:~# ip addr del 10.0.0.1/8 dev h1s1-eth0
root@ryu-vm:~# ip link add link h1s1-eth0 name h1s1-eth0.2 type vlan id 2
root@ryu-vm:~# ip addr add 172.16.10.10/24 dev h1s1-eth0.2
root@ryu-vm:~# ip link set dev h1s1-eth0.2 up
```

host: h2s1:

```
root@ryu-vm:~# ip addr del 10.0.0.4/8 dev h2s1-eth0
root@ryu-vm:~# ip link add link h2s1-eth0 name h2s1-eth0.110 type vlan id 110
root@ryu-vm:~# ip addr add 172.16.10.11/24 dev h2s1-eth0.110
root@ryu-vm:~# ip link set dev h2s1-eth0.110 up
```

host: h1s2:

```
root@ryu-vm:~# ip addr del 10.0.0.2/8 dev h1s2-eth0
root@ryu-vm:~# ip link add link h1s2-eth0 name h1s2-eth0.2 type vlan id 2
root@ryu-vm:~# ip addr add 192.168.30.10/24 dev h1s2-eth0.2
root@ryu-vm:~# ip link set dev h1s2-eth0.2 up
```

host: h2s2:

```
root@ryu-vm:~# ip addr del 10.0.0.5/8 dev h2s2-eth0
root@ryu-vm:~# ip link add link h2s2-eth0 name h2s2-eth0.110 type vlan id 110
root@ryu-vm:~# ip addr add 192.168.30.11/24 dev h2s2-eth0.110
root@ryu-vm:~# ip link set dev h2s2-eth0.110 up
```

host: h1s3:

```
root@ryu-vm:~# ip addr del 10.0.0.3/8 dev h1s3-eth0
root@ryu-vm:~# ip link add link h1s3-eth0 name h1s3-eth0.2 type vlan id 2
root@ryu-vm:~# ip addr add 172.16.20.10/24 dev h1s3-eth0.2
root@ryu-vm:~# ip link set dev h1s3-eth0.2 up
```

host: h2s3:

```
root@ryu-vm:~# ip addr del 10.0.0.6/8 dev h2s3-eth0
root@ryu-vm:~# ip link add link h2s3-eth0 name h2s3-eth0.110 type vlan id 110
root@ryu-vm:~# ip addr add 172.16.20.11/24 dev h2s3-eth0.110
root@ryu-vm:~# ip link set dev h2s3-eth0.110 up
```

最后在联机 Controller 的 xterm 上启动 rest_router。

controller: c0 (root):

```
root@ryu-vm:~# ryu-manager ryu.app.rest_router
loading app ryu.app.rest_router
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
creating context wsgi
instantiating app ryu.app.rest_router of RestRouterAPI
instantiating app ryu.controller.ofp_handler of OFPHandler
(2447) wsgi starting up on http://0.0.0.0:8080/
```

Ryu　和路由器之间的联结完成的话会出现下面的讯息。

controller: c0 (root):

```
[RT][INFO] switch_id=0000000000000003: Set SW config for TTL error packet in.
[RT][INFO] switch_id=0000000000000003: Set ARP handling (packet in) flow [cookie=0
x0]
[RT][INFO] switch_id=0000000000000003: Set L2 switching (normal) flow [cookie=0x0]
[RT][INFO] switch_id=0000000000000003: Set default route (drop) flow [cookie=0x0]
[ RT][ INFO] switch_id =0000000000000003: Start cyclic routing table update.
[RT][INFO] switch_id=0000000000000003: Join as router.
...
```

上面的记录表示三台路由器的准备已经完成。

### 11.2.2 设定 IP 地址

设定每一台路由器的 IP地址。

首先，设定路由器 s1 的 IP 地址为「172.16.10.1/24」和「10.10.10.1/24」，接着VLAN ID 的设定也是必要的。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"address": "172.16.10.1/24"}' http://localhost
:8080/router/0000000000000001/2
  [
    {
      "switch_id": "0000000000000001",
      "command_result": [
        {
          "result": "success",
          "vlan_id": 2,
          "details": "Add address [address_id=1]"
        }
      ]
    }
  ]

root@ryu-vm:~# curl -X POST -d '{"address": "10.10.10.1/24"}' http://localhost
:8080/router/0000000000000001/2
  [
    {
      "switch_id": "0000000000000001",
      "command_result": [
        {
          "result": "success",
          "vlan_id": 2,
          "details": "Add address [address_id=2]"
        }
      ]
    }
  ]

root@ryu-vm:~# curl -X POST -d '{"address": "172.16.10.1/24"}' http://localhost
:8080/router/0000000000000001/110
  [
    {
      "switch_id": "0000000000000001",
      "command_result": [
        {
          "result": "success",
          "vlan_id": 110,
          "details": "Add address [address_id=1]"
        }
      ]
    }
  ]

root@ryu-vm:~# curl -X POST -d '{"address": "10.10.10.1/24"}' http://localhost
:8080/router/0000000000000001/110
  [
    {
      "switch_id": "0000000000000001",
      "command_result": [
        {
          "result": "success",
          "vlan_id": 110,
```

```
            "details": "Add address [address_id=2]"
          }
        ]
      }
    ]
```

接下来，设定路由器 s2 的 IP 地址为「192.168.30.1/24」和「10.10.10.2/24」。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"address": "192.168.30.1/24"}' http://localhost
:8080/router/0000000000000002/2
  [
    {
      "switch_id": "0000000000000002",
      "command_result": [
        {
          "result": "success",
          "vlan_id": 2,
          "details": "Add address [address_id=1]"
        }
      ]
    }
  ]

root@ryu-vm:~# curl -X POST -d '{"address": "10.10.10.2/24"}' http://localhost
:8080/router/0000000000000002/2
  [
    {
      "switch_id": "0000000000000002",
      "command_result": [
        {
          "result": "success",
          "vlan_id": 2,
          "details": "Add address [address_id=2]"
        }
      ]
    }
  ]

root@ryu-vm:~# curl -X POST -d '{"address": "192.168.30.1/24"}' http://localhost
:8080/router/0000000000000002/110
  [
    {
      "switch_id": "0000000000000002",
      "command_result": [
        {
          "result": "success",
          "vlan_id": 110,
          "details": "Add address [address_id=1]"
        }
      ]
    }
  ]

root@ryu-vm:~# curl -X POST -d '{"address": "10.10.10.2/24"}' http://localhost
:8080/router/0000000000000002/110
  [
    {
      "switch_id": "0000000000000002",
      "command_result": [
        {
          "result": "success",
```

```
          "vlan_id": 110,
          "details": "Add address [address_id=2]"
        }
      ]
    }
  ]
```

然后设定路由器 s3 的 IP 地址为「172.16.20.1/24」和「10.10.10.3/24」。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"address": "172.16.20.1/24"}' http://localhost
:8080/router/0000000000000003/2
  [
    {
      "switch_id": "0000000000000003",
      "command_result": [
        {
          "result": "success",
          "vlan_id": 2,
          "details": "Add address [address_id=1]"
        }
      ]
    }
  ]

root@ryu-vm:~# curl -X POST -d '{"address": "10.10.10.3/24"}' http://localhost
:8080/router/0000000000000003/2
  [
    {
      "switch_id": "0000000000000003",
      "command_result": [
        {
          "result": "success",
          "vlan_id": 2,
          "details": "Add address [address_id=2]"
        }
      ]
    }
  ]

root@ryu-vm:~# curl -X POST -d '{"address": "172.16.20.1/24"}' http://localhost
:8080/router/0000000000000003/110
  [
    {
      "switch_id": "0000000000000003",
      "command_result": [
        {
          "result": "success",
          "vlan_id": 110,
          "details": "Add address [address_id=1]"
        }
      ]
    }
  ]

root@ryu-vm:~# curl -X POST -d '{"address": "10.10.10.3/24"}' http://localhost
:8080/router/0000000000000003/110
  [
    {
      "switch_id": "0000000000000003",
      "command_result": [
        {
```

```
            "result": "success",
            "vlan_id": 110,
            "details": "Add address [address_id=2]"
        }
    ]
  }
]
```

路由器的 IP 地址已经设定好，接着设定每一个 host 的默认网关。

host: h1s1:

```
root@ryu-vm:~# ip route add default via 172.16.10.1
```

host: h2s1:

```
root@ryu-vm:~# ip route add default via 172.16.10.1
```

host: h1s2:

```
root@ryu-vm:~# ip route add default via 192.168.30.1
```

host: h2s2:

```
root@ryu-vm:~# ip route add default via 192.168.30.1
```

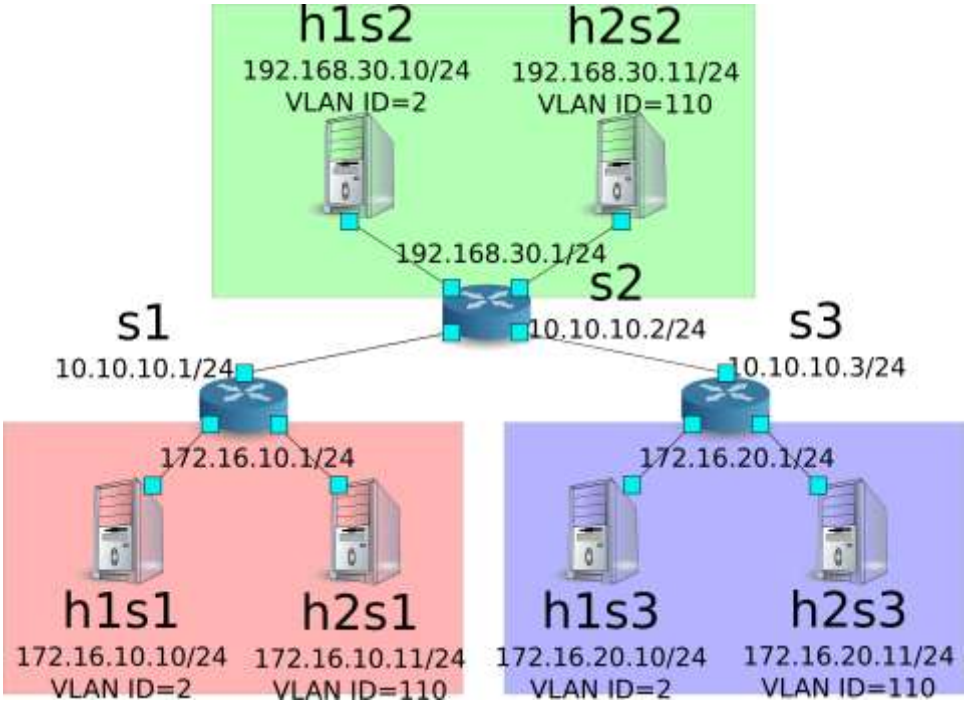
host: h1s3:

```
root@ryu-vm:~# ip route add default via 172.16.20.1
```

host: h2s3:

```
root@ryu-vm:~# ip route add default via 172.16.20.1
```

IP 地址被设定如下。



### 11.2.3 设定默认静态路由

设定每一台路由器的默认静态路由。 首先，设定路由器 s1 的预设路由为路由器 s2。 Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"gateway": "10.10.10.2"}' http://localhost:8080/
router/0000000000000001/2
  [
    {
      "switch_id": "0000000000000001",
      "command_result": [
        {
          "result": "success",
          "vlan_id": 2,
          "details": "Add route [route_id=1]"
        }
      ]
    }
  ]

root@ryu-vm:~# curl -X POST -d '{"gateway": "10.10.10.2"}' http://localhost:8080/
router/0000000000000001/110
  [
    {
      "switch_id": "0000000000000001",
      "command_result": [
        {
          "result": "success",
          "vlan_id": 110,
          "details": "Add route [route_id=1]"
        }
      ]
    }
  ]
```

路由器 s2 的预设路由设定为路由器 s1。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"gateway": "10.10.10.1"}' http://localhost:8080/
router/0000000000000002/2
  [
    {
      "switch_id": "0000000000000002",
      "command_result": [
        {
          "result": "success",
          "vlan_id": 2,
          "details": "Add route [route_id=1]"
        }
      ]
    }
  ]

root@ryu-vm:~# curl -X POST -d '{"gateway": "10.10.10.1"}' http://localhost:8080/
router/0000000000000002/110
  [
    {
      "switch_id": "0000000000000002",
      "command_result": [
```

```
        {
          "result": "success",
          "vlan_id": 110,
          "details": "Add route [route_id=1]"
        }
      ]
    }
  ]
```

路由器 s3 的预设路由设定为路由器 s2。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"gateway": "10.10.10.2"}' http://localhost:8080/
router/0000000000000003/2
  [
    {
      "switch_id": "0000000000000003",
      "command_result": [
        {
          "result": "success",
          "vlan_id": 2,
          "details": "Add route [route_id=1]"
        }
      ]
    }
  ]

root@ryu-vm:~# curl -X POST -d '{"gateway": "10.10.10.2"}' http://localhost:8080/
router/0000000000000003/110
  [
    {
      "switch_id": "0000000000000003",
      "command_result": [
        {
          "result": "success",
          "vlan_id": 110,
          "details": "Add route [route_id=1]"
        }
      ]
    }
  ]
```

接着为了路由器 s2，将路由器 s3 的静态路由指向 host（172.16.20.0/24 ），但仅只有在 VLAN ID = 2 的情况下。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"destination": "172.16.20.0/24", "gateway":
"10.10.10.3"}'  http://localhost:8080/router/0000000000000002/2
  [
    {
      "switch_id": "0000000000000002",
      "command_result": [
        {
          "result": "success",
          "vlan_id": 2,
          "details": "Add route [route_id=2]"
        }
      ]
    }
  ]
```

### 11.2.4 确认设定的内容

确认每一台路由器的设定内容。

Node: c0 (root):

```
root@ryu-vm:~# curl http://localhost:8080/router/all/all
  [
    {
      "internal_network": [
        {},
        {
          "route": [
            {
              "    route_id    ":     1,
              "destination": "0.0.0.0/0",
              "gateway": "10.10.10.2"
            }
          ],
          "vlan_id": 2,
          "address": [
            {
              "   address_id   ":    2,
              "address": "10.10.10.1/24"
            },
            {
              "address_id": 1,
              "address": "172.16.10.1/24"
            }
          ]
        },
        {
          "route": [
            {
              "    route_id    ":     1,
              "destination": "0.0.0.0/0",
              "gateway": "10.10.10.2"
            }
          ],
          "vlan_id": 110,
          "address": [
            {
              "   address_id   ":    2,
              "address": "10.10.10.1/24"
            },
            {
              "address_id": 1,
              "address": "172.16.10.1/24"
            }
          ]
        }
      ],
      "switch_id": "0000000000000001"
    },
    {
      "internal_network": [
        {},
        {
          "route": [
            {
              "route_id": 2,
              "destination": "172.16.20.0/24",
              "gateway": "10.10.10.3"
```

```
            },
            {
              "    route_id    ":     1,
              "destination": "0.0.0.0/0",
              "gateway": "10.10.10.1"
            }
          ],
          "vlan_id": 2,
          "address": [
            {
              "   address_id   ":    2,
              "address": "10.10.10.2/24"
            },
            {
              "address_id": 1,
              "address": "192.168.30.1/24"
            }
          ]
        },
        {
          "route": [
            {
              "    route_id    ":     1,
              "destination": "0.0.0.0/0",
              "gateway": "10.10.10.1"
            }
          ],
          "vlan_id": 110,
          "address": [
            {
              "   address_id   ":    2,
              "address": "10.10.10.2/24"
            },
            {
              "address_id": 1,
              "address": "192.168.30.1/24"
            }
          ]
        }
      ],
      "switch_id": "0000000000000002"
    },
    {
      "internal_network": [
        {},
        {
          "route": [
            {
              "    route_id    ":     1,
              "destination": "0.0.0.0/0",
              "gateway": "10.10.10.2"
            }
          ],
          "vlan_id": 2,
          "address": [
            {
              "address_id": 1,
              "address": "172.16.20.1/24"
            },
            {
              "   address_id   ":    2,
              "address": "10.10.10.3/24"
            }
```

```
        ]
      },
      {
        "route": [
          {
            "    route_id    ":     1,
            "destination": "0.0.0.0/0",
            "gateway": "10.10.10.2"
          }
        ],
        "vlan_id": 110,
        "address": [
          {
            "address_id": 1,
            "address": "172.16.20.1/24"
          },
          {
            "   address_id   ":    2,
            "address": "10.10.10.3/24"
          }
        ]
      }
    ],
    "switch_id": "0000000000000003"
  }
]
```

每一台路由器的设定内容将会如下所示。

| 路由器 | VLAN ID | IP 地址 | 预设路由 | 静态路由 |
|---|---|---|---|---|
| s1 | 2 | 172.16.10.1/24 | 10.10.10.2(s2) | |
| | | , 10.10.10.1/24 | | |
| s1 | 110 | 172.16.10.1/24 | 10.10.10.2(s2) | |
| | | , 10.10.10.1/24 | | |
| s2 | 2 | 192.168.30.1/24 | 10.10.10.1(s1) | 目的: 172.16.20.0/24 |
| | | , 10.10.10.2/24 | | , 閘道: 10.10.10.3(s3)" |
| s2 | 110 | 192.168.30.1/24 | 10.10.10.1(s1) | |
| | | , 10.10.10.2/24 | | |
| s3 | 2 | 172.16.20.1/24 | 10.10.10.2(s2) | |
| | | , 10.10.10.3/24 | | |
| s3 | 110 | 172.16.20.1/24 | 10.10.10.2(s2) | |
| | | , 10.10.10.3/24 | | |

从 h1s1 向 h1s3 发送 ping 讯息。因为是处于相同的 vlan_id = 2 的相同 host ，且已经设置了指向 s3 的静态路由在 s2 上，因此应该是可以正常联机的。

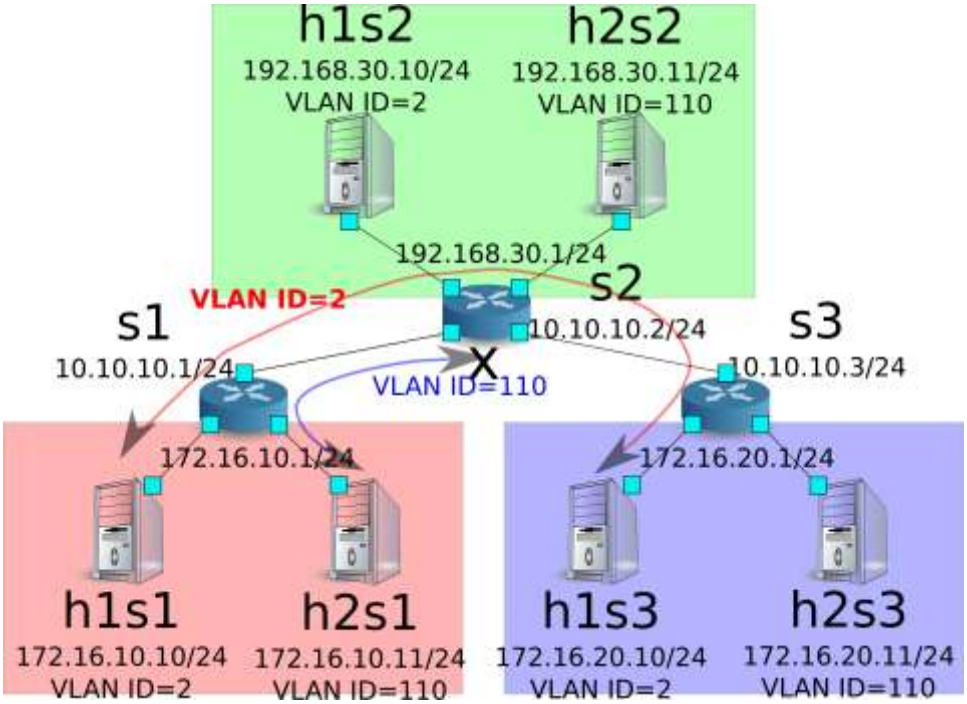
host: h1s1:

```
root@ryu-vm:~# ping 172.16.20.10
PING 172.16.20.10 (172.16.20.10) 56(84) bytes of data.
64 bytes from 172.16.20.10: icmp_req=1 ttl=61 time=45.9 ms
64 bytes from 172.16.20.10: icmp_req=2 ttl=61 time=0.257 ms
64 bytes from 172.16.20.10: icmp_req=3 ttl=61 time=0.059 ms
64 bytes from 172.16.20.10: icmp_req=4 ttl=61 time=0.182 ms
```

从 h2s1 向 h2s3 发送 ping 封包，虽然他们处于相同的 vlani_id = 110 的 host ，但是路由器 s2 上并没有设置指向路由器 s3 的静态路由，因此无法成功联机。

host: h2s1:

```
root@ryu-vm:~# ping 172.16.20.11
PING 172.16.20.11 (172.16.20.11) 56(84) bytes of data.
^C
--- 172.16.20.11 ping statistics ---
8 packets transmitted, 0 received, 100% packet loss, time 7009ms
```




在本章中，使用一个具体的例子来说明路由器的使用方法。

## 11.3 REST API 列表

本章所介绍的 rest_router 的 REST API 列表。

### 11.3.1 取得设定内容

| 方法 | GET |
|---|---|
| URL | /router/{switch}[/{vlan}] |
| | -- switch: [ ``all" \| 交换器 ID ] |
| | -- vlan: [ ``all" \| VLAN ID ] |
| 备注 | 指定 VLAN ID 为可选项目。 |

### 11.3.2 设定地址

| 方法 | POST |
|---|---|
| URL | /router/{switch}[/{vlan}] |
| | -- switch: [ ``all" \| 交换器 ID ] |
| | -- vlan: [ ``all" \| VLAN ID ] |
| 内容 | address:"<xxx.xxx.xxx.xxx/xx>" |
| 备注 | 在设定路由之前要先设定地址 |
| | 指定VLAN ID为可选项目。 |

### 11.3.3 设定静态路由

| 方法 | POST |
|---|---|
| URL | /router/{switch}[/{vlan}] |
| | -- switch: [ ``all" \| 交换器 ID ] |
| | -- vlan: [ ``all" \| VLAN ID ] |
| 内容 | destination:"<xxx.xxx.xxx.xxx/xx>" |
| | gateway:"<xxx.xxx.xxx.xxx>" |
| 备注 | 指定VLAN ID为可选项目。 |

### 11.3.4 设定预设路由

| 方法 | POST |
|---|---|
| URL | /router/{switch}[/{vlan}] |
| | -- switch: [ ``all" \| 交换器 ID ] |
| | -- vlan: [ ``all" \| VLAN ID ] |
| 内容 | gateway:"<xxx.xxx.xxx.xxx>" |
| 备注 | 指定VLAN ID为可选项目。 |

### 11.3.5 删除地址

| 方法 | DELETE |
|---|---|
| URL | /router/{switch}[/{vlan}] |
| | -- switch: [ ``all" \| 交换器 ID ] |
| | -- vlan: [ ``all" \| VLAN ID ] |
| 内容 | address_id:[ 1 - ... ] |
| 备注 | 指定VLAN ID为可选项目。 |

### 11.3.6 删除路由

| 方法 | DELETE |
|---|---|
| URL | /router/{switch}[/{vlan}] |
| | -- switch: [ ``all" \| 交换器 ID ] |
| | -- vlan: [ ``all" \| VLAN ID ] |
| 内容 | route_id:[ 1 - ... ] |
| 备注 | 指定VLAN ID为可选项目。 |

CHAPTER 12

# OpenFlow 交换器测试工具

本章将说明如何检验 OpenFlow 交换器对于 OpenFlow 规范的功能支持完整度，及测试工具的 使用方法。

## 12.1 测试工具概要

本工具使用测试样板档案，对待测的 OpenFlow 交换器，进行 Flow Entry 和 Meter Entry 的新增以处理封包，并且将 OpenFlow 交换器所处理及转送封包的结果与测试样板档案所描述的”预期 处理结果”做比对。亦即检验 OpenFlow 交换器对于 OpenFlow 规格功能的支持状态。

在测试工具中，已经有包含 OpenFlow 1.3 版本中的 FlowMod 讯息、MeterMod 讯息和 GroupMod 讯息的测试动作。

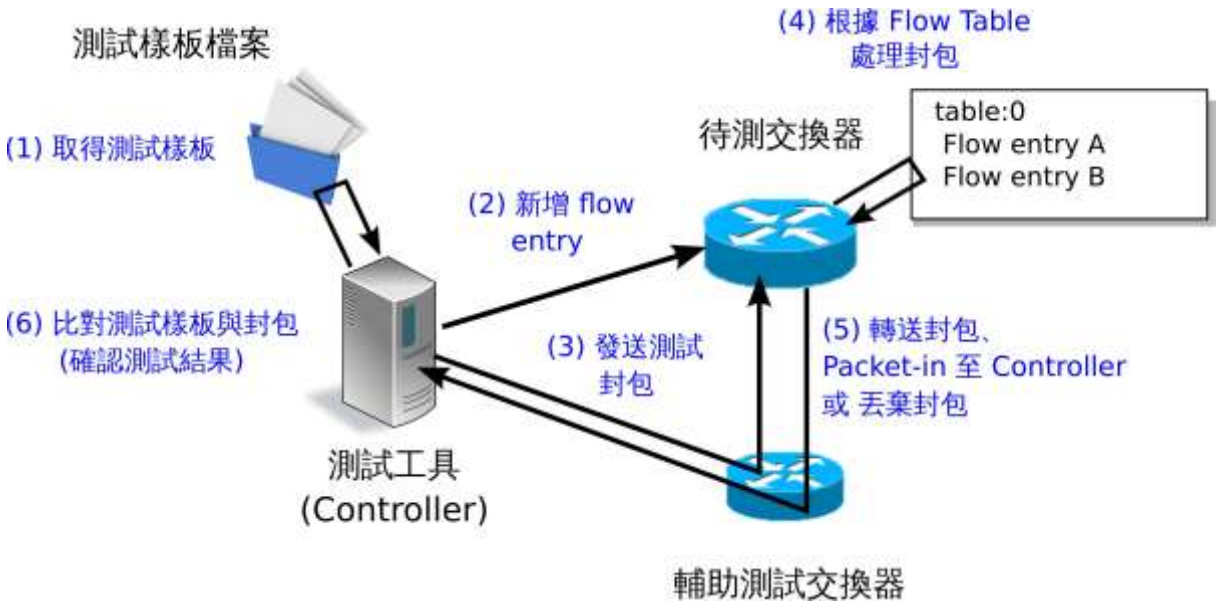
| 测试讯息种类 | 相应的参数 |
|---|---|
| OpenFlow1.3 FlowMod 讯息 | match (IN_PHY_PORT 除外)<br>actions（SET_QUEUE 除外） |
| OpenFlow1.3 MeterMod 讯息 | 全部 |
| OpenFlow1.3 GroupMod 讯息 | 全部 |

若要了解关于封包产生及修改的详细数据，请参考「封包函式库 」。

### 12.1.1 操作纲要

#### 测试验证环境示意图

测试工具实际执行时如下列示意图。测试样板档案中包含了“待加入的 Flow Entry 和 Meter Entry “、”检验封包”和”预期处理结果”。为了执行测试执行所需的环境设定将在后面的执行测试的环境 章节中描述。




### 输出测试结果

在指定了测试样板档案后，样板中的测试案例会被依序执行，最后并显示结果（OK / ERROR）。在出现 ERROR 的测试结果时，错误讯息会同时一并出现在画面上。最后的测试结果会显示 OK / ERROR 的数量及错误内容。

```
--- Test start ---

match: 29_ICMPV6_TYPE
    ethernet/ipv6/icmpv6(type=128)-->'icmpv6_type=128,actions=output:2'
              OK
    ethernet/ipv6/icmpv6(type=128)-->'icmpv6_type=128,actions=output:CONTROLLER'
     OK
    ethernet/ipv6/icmpv6(type=135)-->'icmpv6_type=128,actions=output:2'
              OK
    ethernet/vlan/ipv6/icmpv6(type=128)-->'icmpv6_type=128,actions=output:2'
         ERROR
        Received incorrect packet - in: ethernet( ethertype =34525)
    ethernet/vlan/ipv6/icmpv6(type=128)-->'icmpv6_type=128,actions=output:
CONTROLLER' ERROR
        Received incorrect packet-in: ethernet(ethertype=34525)
match: 30_ICMPV6_CODE
    ethernet/ipv6/icmpv6(code=0)-->'icmpv6_code=0,actions=output:2'
                  OK
    ethernet/ipv6/icmpv6(code=0)-->'icmpv6_code=0,actions=output:CONTROLLER'
         OK
    ethernet/ipv6/icmpv6(code=1)-->'icmpv6_code=0,actions=output:2'
                  OK
    ethernet/vlan/ipv6/icmpv6(code=0)-->'icmpv6_code=0,actions=output:2'
            ERROR
        Received  incorrect  packet - in:  ethernet( ethertype =34525)
    ethernet/vlan/ipv6/icmpv6(code=0)-->'icmpv6_code=0,actions=output:CONTROLLER'
    ERROR
        Received incorrect packet-in: ethernet(ethertype=34525)

---  Test end  ---

--- Test report ---
Received incorrect packet-in(4)
```

```
    match: 29_ICMPV6_TYPE                      ethernet/vlan/ipv6/icmpv6(type=128)
-->'icmpv6_type=128,actions=output:2'
    match: 29_ICMPV6_TYPE                      ethernet/vlan/ipv6/icmpv6(type=128)
-->'icmpv6_type=128,actions=output:CONTROLLER'
    match: 30_ICMPV6_CODE                      ethernet/vlan/ipv6/icmpv6(code=0)-->'
icmpv6_code=0,actions=output:2'
    match: 30_ICMPV6_CODE                      ethernet/vlan/ipv6/icmpv6(code=0)-->'
icmpv6_code=0,actions=output:CONTROLLER'

OK(6) / ERROR(4)
```

## 12.2 使用方法

下面说明如何使用测试工具。

### 12.2.1 测试范本档案

你需要依照测试样板的相关规则来建立一个测试样板，以完成你想要的测试项目。 测试样板的附档名是「.json」，格式如下。

```
[
    "xxxxxxxxxx",                    # 测试名称
    {
        "description": "xxxxxxxxxx", # 测试内容的描述
        "prerequisite": [
            {
                "OFPFlowMod": {...}  # 所要新增的 flow entry、meter entry、group
entry
            },                       # (Ryu 的 OFPFlowMod、OFPMeterMod、
OFPGroupMod 使用 json 的形态描述)
            {                        #
                "OFPMeterMod": {...} # 要将 flow entry 处理的结果转送出去的情况下
            },                       # (actions=output)
            {                        # 请指定输出埠的编号为 '2」
                "OFPGroupMod": {...} # 若是封包转送至 group entry 的情况
            },                       # 请指定输出埠号为「2」或「3」
            {...}                    #
        ],
        "tests": [
            {
                # 产生封包
                # 单次产生封包或者一定时间内连续产生封包均可。
                # 封包的产生方法有 (A) (B) 两种
                #  (A) 单次产生封包
                "ingress": [
                    "ethernet(...)", # (在 Ryu 封包函式库的建构子 (Constructor)
中描述)
                    "ipv4(...)",
                    "tcp(...)"
                ],
                #  (B) 一段时间内连续产生封包
                "ingress": {
                    "packets":{
                        "data":[
                            "ethernet(...)", # 与 (A) 相同
                            "ipv4(...)",
                            "tcp(...)"
                        ],
```

```
                        "pktps": 1000,        # 每秒产生封包的数量 ( packet per
second )
                        "duration_time": 30   # 连续产生封包的时间长度，以秒为单位。
                    }
                },

                # 预期处理的结果
                # 处理的结果有 (a) (b) (c) (d) 这几种
                #  (a) 封包转送 ( actions=output:X )
                "egress": [           # 预期转送封包
                    "ethernet(...)",
                    "ipv4(...)",
                    "tcp(...)"
                ]
                #  (b) Packet in ( actions=CONTROLLER )
                "PACKET_IN": [        # 预期出现的 Packet in 封包
                    "ethernet(...)",
                    "ipv4(...)",
                    "tcp(...)"
                ]
                #  (c) table-miss
                "table-miss": [       # 期望 table-miss 发生时的 table ID
                    0
                ]
                #  (d) 封包转送 ( actions=output:X ) 时的流量 ( Throughput ) 测试
                "egress":[
                    "throughput":[
                        {
                            "OFPMatch":{    # 为了 Throughput 测试
                              ...           # 新增在辅助交换器中
                            },              # flow entry 的 match 条件
                            "kbps":1000     # 指定期望的流量以 Kbps 为单位
                        },
                        {...},
                        {...}
                    ]
                ]
            },
            {...},
            {...}
        ]
    },                                  # 测试项目1
    {...},                              # 测试项目2
    {...}                               # 测试项目3
]
```

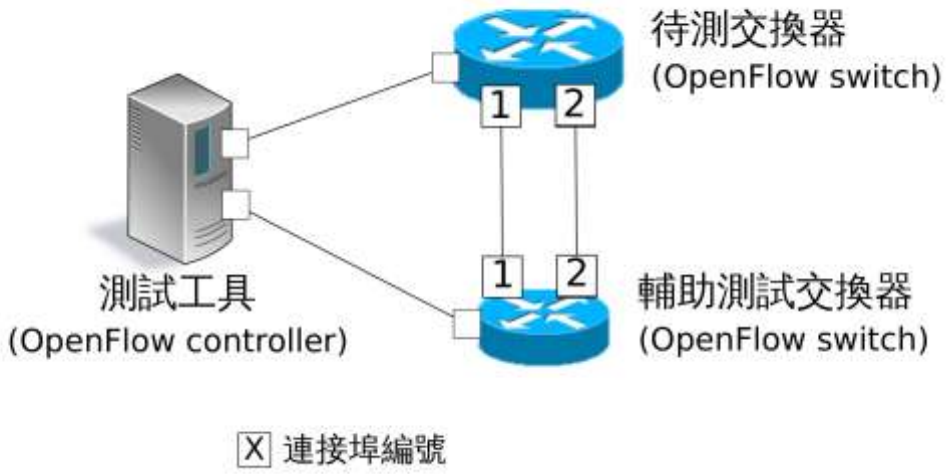
例如，产生封包中「(B) 一段时间内产生封包」和预期处理结果中「(d) 封包转送 ( actions=output:X ) 时流量测试」搭配时就可以用来对待测交换器进行流量 ( Throughput ) 的测试。

备注: 作为一个测试样板在 Ryu 的原始码中，提供了一些样板档案来检查测试参数是否符合 OpenFlow1.3 FlowMod 中的 match / action 讯息。

ryu/tests/switch/of13

### 12.2.2 执行测试的环境

接下来说明测试工具执行时所需的环境。




对于做为一个辅助交换器来说，下面的条件是一个 OpenFlow 交换器必须要支持的。

- actions=CONTROLLER Flow Entry 新增
- 流量监控用的 Flow Entry 新增
- 透过 Flow Entry 发送 Packet-In 讯息到 Controller（actions = CONTROLLER）。
- 接受 Packet-Out 讯息并发送封包

备注: Ryu 原始码当中利用脚本实作了一个在 mininet 上的测试环境，当中是采用 Open vSwtich 做为待测交换器。

ryu/tests/switch/run_mininet.py 脚本

的使用范例请参照测试工具使用范例 。

### 12.2.3 执行测试工具的方法

测试工具已经被公开在 Ryu 的原始码当中。

| 原始码 | 说明 |
|---|---|
| ryu/tests/switch/tester.py | 测试工具 |
| ryu/tests/switch/of13 | 测试样板的一些范例 |
| ryu/tests/switch/run_mininet.py | 建立测试环境的脚本 |

使用下面的指令来执行测试工具。

```
$ ryu-manager [--test-switch-target DPID] [--test-switch-tester DPID]
 [--test-switch-dir DIRECTORY] ryu/tests/switch/tester.py
```

| 选项 | 说明 | 默认值 |
|---|---|---|
| --test-switch-target | 待测交换器的 datapath ID | 0000000000000001 |
| --test-switch-tester | 辅助交换器的 datapath ID | 0000000000000002 |
| --test-switch-dir | 测试样板的存放路径 | ryu/tests/switch/of13 |

备注: 测试工具是继承自 ryu.base.app_manager.RyuApp 的一个应用程序。跟其他的 Ryu 应用程序一样使用 --verbose选项显示除错的讯息。

测试工具启动之后，待测交换器和辅助交换器会跟 Controller 进行连接，接着测试动作就会使用指定的测试样板开始进行测试。

## 12.3 测试工具使用范例

下面介绍如何使用和测试样板档案和原始测试样板档案的步骤。

### 12.3.1 执行测试样板档案的步骤

使用 Ryu 的原始码中测试样板模板（ryu/tests/switch/of13 ）来检查 FlowMod 讯息的 match / action，MeterMod 的讯息和 GroupMod 讯息。

本程序中测试环境和测试环境的产生脚本（ryu/tests/switch/run_mininet.py ），也因此测试目标是 Open vSwitch。使用 VM image 来打造测试环境以及登入的方法请参照「交换器（Switching Hub ）」以取得更详细的资料。

1. 建构测试环境

   VM 环境的登入，执行测试环境的建构脚本。

```
ryu@ryu-vm:~$ sudo ryu/ryu/tests/switch/run_mininet.py
```

   net 命令的执行结果如下。

```
mininet> net
c0
s1 lo:  s1-eth1:s2-eth1 s1-eth2:s2-eth2 s1-eth3:s2-eth3
s2 lo:  s2-eth1:s1-eth1 s2-eth2:s1-eth2 s2-eth3:s1-eth3
```

2. 执行测试工具

   为了执行测试工具，打开联机到 Controller 的 xterm。

```
mininet> xterm c0
```

   在「Node: c0 (root)」的 xterm 中启动测试工具。这时候做为测试样板档案的位置，请指定测试样板范例路径（ryu/tests/switch/of13 ）。接着，由于 mininet 测试环境中待测交换器和辅助交换器的 datapath ID 均有默认值，因此 --test-switch-target / --test-switch-tester 选项可省略。

   Node: c0:

```
root@ryu-vm:~$ ryu-manager --test-switch-dir ryu/ryu/tests/switch/
of13 ryu/ryu/tests/switch/tester.py
```

   测试工具执行之后就会出现下列讯息，并等待待测交换器和辅助交换器连结到 Controller。

```
root@ryu-vm:~$ ryu-manager --test-switch-dir ryu/ryu/tests/switch/
of13/ ryu/ryu/tests/switch/tester.py
loading app ryu/ryu/tests/switch/tester.py
loading app ryu.controller.ofp_handler
instantiating app ryu/ryu/tests/switch/tester.py of OfTester
target_dpid =0000000000000001
tester_dpid =0000000000000002
Test files directory = ryu/ ryu/ tests/ switch/ of13 /
instantiating app ryu.controller.ofp_handler of OFPHandler
```

```
--- Test start ---
waiting for switches connection...
```

待测交换器和辅助交换器连接Contreoller完成，测试开始。

```
root@ryu-vm:~$ ryu-manager --test-switch-dir ryu/ryu/tests/switch/
of13/ ryu/ryu/tests/switch/tester.py
loading app ryu/ryu/tests/switch/tester.py
loading app ryu.controller.ofp_handler
instantiating app ryu/ryu/tests/switch/tester.py of OfTester
target_dpid =0000000000000001
tester_dpid =0000000000000002
Test files directory = ryu/ ryu/ tests/ switch/ of13 /
instantiating app ryu.controller.ofp_handler of OFPHandler
--- Test start ---
waiting for switches connection ...
dpid=0000000000000002 : Join tester SW.
dpid=0000000000000001 : Join target SW.
action: 00_OUTPUT
    ethernet/ipv4/tcp-->'actions=output:2'  OK
    ethernet/ipv6/tcp-->'actions=output:2'  OK
    ethernet/arp-->'actions=output:2'       OK
action: 11_COPY_TTL_OUT
    ethernet/mpls(ttl=64)/ipv4(ttl=32)/tcp-->'eth_type=0x8847,actions
=copy_ttl_out,output:2'         ERROR
        Failed to add flows: OFPErrorMsg[type=0x02, code=0x00]
    ethernet/mpls(ttl=64)/ipv6(hop_limit=32)/tcp-->'eth_type=0x8847,
actions=copy_ttl_out,output:2'   ERROR
        Failed to add flows: OFPErrorMsg[type=0x02, code=0x00]
...
```

ryu/tests/switch/of13 文件夹以下的测试样板全部执行完毕，测试也随之结束。

## <参考数据>

测试样板范本档案一览

提供测试样板模板档案包括，对应match / actions各种设定的Flow Entry新增：match（或不 match ）多数 pattern 的封包改写、对应满足一定频率的后变更优先权的 Meter Entry 新增：Meter Entry 中 match 的封包连续改写、对应全端口的 FLOODING 的 Group Entry 新增：Group Entry 中 match 封包的连续改写。

```
ryu/tests/switch/of13/action:
00_OUTPUT.json               20_POP_MPLS.json
11_COPY_TTL_OUT.json         23_SET_NW_TTL_IPv4.json
12_COPY_TTL_IN.json          23_SET_NW_TTL_IPv6.json
15_SET_MPLS_TTL.json         24_DEC_NW_TTL_IPv4.json
16_DEC_MPLS_TTL.json         24_DEC_NW_TTL_IPv6.json
17_PUSH_VLAN.json            25_SET_FIELD
17_PUSH_VLAN_multiple.json   26_PUSH_PBB.json
18_POP_VLAN.json             26_PUSH_PBB_multiple.json
19_PUSH_MPLS.json            27_POP_PBB.json
19_PUSH_MPLS_multiple.json

ryu/tests/switch/of13/action/25_SET_FIELD:
03_ETH_DST.json         14_TCP_DST_IPv4.json    24_ARP_SHA.json
04_ETH_SRC.json         14_TCP_DST_IPv6.json    25_ARP_THA.json
05_ETH_TYPE.json        15_UDP_SRC_IPv4.json    26_IPV6_SRC.json
06_VLAN_VID.json        15_UDP_SRC_IPv6.json    27_IPV6_DST.json
```

```
07_VLAN_PCP.json          16_UDP_DST_IPv4.json    28_IPV6_FLABEL.json
08_IP_DSCP_IPv4.json      16_UDP_DST_IPv6.json    29_ICMPV6_TYPE.json
08_IP_DSCP_IPv6.json      17_SCTP_SRC_IPv4.json   30_ICMPV6_CODE.json
09_IP_ECN_IPv4.json       17_SCTP_SRC_IPv6.json   31_IPV6_ND_TARGET.json
09_IP_ECN_IPv6.json       18_SCTP_DST_IPv4.json   32_IPV6_ND_SLL.json
10_IP_PROTO_IPv4.json     18_SCTP_DST_IPv6.json   33_IPV6_ND_TLL.json
10_IP_PROTO_IPv6.json     19_ICMPV4_TYPE.json     34_MPLS_LABEL.json
11_IPV4_SRC.json          20_ICMPV4_CODE.json     35_MPLS_TC.json
12_IPV4_DST.json          21_ARP_OP.json          36_MPLS_BOS.json
13_TCP_SRC_IPv4.json      22_ARP_SPA.json         37_PBB_ISID.json
13_TCP_SRC_IPv6.json      23_ARP_TPA.json         38_TUNNEL_ID.json

ryu/tests/switch/of13/group:
00_ALL.json             01_SELECT_IP.json             01_SELECT_Weight_IP.
json
01_SELECT_Ether.json    01_SELECT_Weight_Ether.json

ryu/tests/switch/of13/match:
00_IN_PORT.json           13_TCP_SRC_IPv4.json    25_ARP_THA.json
02_METADATA.json          13_TCP_SRC_IPv6.json    25_ARP_THA_Mask.json
02_METADATA_Mask.json     14_TCP_DST_IPv4.json    26_IPV6_SRC.json
03_ETH_DST.json           14_TCP_DST_IPv6.json    26_IPV6_SRC_Mask.json
03_ETH_DST_Mask.json      15_UDP_SRC_IPv4.json    27_IPV6_DST.json
04_ETH_SRC.json           15_UDP_SRC_IPv6.json    27_IPV6_DST_Mask.json
04_ETH_SRC_Mask.json      16_UDP_DST_IPv4.json    28_IPV6_FLABEL.json
05_ETH_TYPE.json          16_UDP_DST_IPv6.json    29_ICMPV6_TYPE.json
06_VLAN_VID.json          17_SCTP_SRC_IPv4.json   30_ICMPV6_CODE.json
06_VLAN_VID_Mask.json     17_SCTP_SRC_IPv6.json   31_IPV6_ND_TARGET.json
07_VLAN_PCP.json          18_SCTP_DST_IPv4.json   32_IPV6_ND_SLL.json
08_IP_DSCP_IPv4.json      18_SCTP_DST_IPv6.json   33_IPV6_ND_TLL.json
08_IP_DSCP_IPv6.json      19_ICMPV4_TYPE.json     34_MPLS_LABEL.json
09_IP_ECN_IPv4.json       20_ICMPV4_CODE.json     35_MPLS_TC.json
09_IP_ECN_IPv6.json       21_ARP_OP.json          36_MPLS_BOS.json
10_IP_PROTO_IPv4.json     22_ARP_SPA.json         37_PBB_ISID.json
10_IP_PROTO_IPv6.json     22_ARP_SPA_Mask.json    37_PBB_ISID_Mask.json
11_IPV4_SRC.json          23_ARP_TPA.json         38_TUNNEL_ID.json
11_IPV4_SRC_Mask.json     23_ARP_TPA_Mask.json    38_TUNNEL_ID_Mask.json
12_IPV4_DST.json          24_ARP_SHA.json         39_IPV6_EXTHDR.json
12_IPV4_DST_Mask.json     24_ARP_SHA_Mask.json    39_IPV6_EXTHDR_Mask.json

ryu/tests/switch/of13/meter:
01_DROP_00_KBPS_00_1M.json        02_DSCP_REMARK_00_KBPS_00_1M.json
01_DROP_00_KBPS_01_10M.json       02_DSCP_REMARK_00_KBPS_01_10M.json
01_DROP_00_KBPS_02_100M.json      02_DSCP_REMARK_00_KBPS_02_100M.json
01_DROP_01_PKTPS_00_100.json      02_DSCP_REMARK_01_PKTPS_00_100.json
01_DROP_01_PKTPS_01_1000.json     02_DSCP_REMARK_01_PKTPS_01_1000.json
01_DROP_01_PKTPS_02_10000.json    02_DSCP_REMARK_01_PKTPS_02_10000.json
```

### 12.3.2 原始测试样板的执行步骤

接着，原始的测试样板制作并执行测试工具的步骤如下所示。

例如 OpenFlow 交换器若要实作路由器的功能，match / actions的处理功能是必须的，因此我们制作测试样板来确认他。

1．制作测试样板档案

透过路由器的路由表（Routing table）实作封包的转送功能。下面的 Flow Entry 会确认整个动作是否正确。

| match | actions |
| --- | --- |
| IP 网域「192.168.30.0/24」 | 修改发送端 MAC 地址为「aa:aa:aa:aa:aa:aa」 |
| | 修改目的端 MAC 地址为「bb:bb:bb:bb:bb:bb」 |
| | 降低TTL值 |
| | 封包转送 |

| match | actions |
| --- | --- |
| IP 网域「192.168.30.0/24」 | 修改发送端 MAC 地址为「aa:aa:aa:aa:aa:aa」 |
| ryu/tests/switch/of13 | 测试样版的一些范例 |
| ryu/ tests/ switch/ run_mininet.py | 建立测试环境的脚本 |

依照这个测试样板产生测试样板档案。

文件名：sample_test_pattern.json

```
[
   "sample: Router test",
   {
       "description": "static routing table",
       "prerequisite": [
           {
               " OFPFlowMod":
                   { "table_id":
                   0,
                   "match": {
                       "OFPMatch": {
                           "oxm_fields": [
                               {
                                   "OXMTlv": {
                                       "field": "eth_type",
                                       "value": 2048
                                   }
                               },
                               {
                                   "OXMTlv": {
                                       "field": "ipv4_dst",
                                       "mask": 4294967040,
                                       "value": "192.168.30.0"
                                   }
                               }
                           ]
                       }
                   },
                   " instructions":[
                       {
                           "OFPInstructionActions":
                               { "actions":[
                                   {
                                       "OFPActionSetField
                                           ":{ "field":{
                                               "OXMTlv":{
                                                   "    field":"   eth_src",
                                                   "value":"aa:aa:aa:aa:aa:aa"
                                               }
                                           }
                                       }
                                   },
                                   {
                                       "OFPActionSetField
                                           ":{ "field":{
                                               "OXMTlv":{
```

```
                                                    "value":"bb:bb:bb:bb:bb:bb"
                                                }
                                            }
                                        }
                                    },
                                    {
                                        "OFPActionDecNwTtl":{}
                                    },
                                    {
                                        "OFPActionOutput":
                                            { "port":2
                                        }
                                    }
                                ],
                                "type": 4
                            }
                        }
                    ]
                }
            }
        ],
        "tests":[
            {
                "ingress":[
                    "ethernet(dst='22:22:22:22:22:22',src='11:11:11:11:11:11',
ethertype=2048)",
                    "ipv4(tos=32, proto=6, src='192.168.10.10', dst='192.168.30.10',
 ttl=64)",
                    "tcp(dst_port=2222, option='\\x00\\x00\\x00\\x00', src_port
=11111)",
                    "'\\x01\\x02\\x03\\x04\\x05\\x06\\x07\\x08\\t\\n\\x0b\\x0c\\r\\
x0e\\x0f'"
                ],
                "egress":[
                    "ethernet(dst='bb:bb:bb:bb:bb:bb',src='aa:aa:aa:aa:aa:aa',
ethertype=2048)",
                    "ipv4(tos=32, proto=6, src='192.168.10.10', dst='192.168.30.10',
 ttl=63)",
                    "tcp(dst_port=2222, option='\\x00\\x00\\x00\\x00', src_port
=11111)",
                    "'\\x01\\x02\\x03\\x04\\x05\\x06\\x07\\x08\\t\\n\\x0b\\x0c\\r\\
x0e\\x0f'"
                ]
            }
        ]
    }
]
```

2．建构测试环境

使用测试环境建置脚本来完成测试环境。详细的操作细节请参照执行测试样板档案的步骤。

3．执行测试工具

使用 Conrtroller 的 xterm 窗口，指定先前做好的测试样板档案位置并执行测试工具。可以使用 --test-switch-dir 选项来指定样板档案的位置。如果想要确认收送封包的内容，可以指定 --verbose 选项。

Node: c0:

```
root@ryu-vm:~$ ryu-manager --verbose --test-switch-dir ./
sample_test_pattern.json  ryu/ryu/tests/switch/tester.py
```

待测交换器和辅助交换器已经和Controller连接的情况下，测试即将开始。

「dpid=0000000000000002 ： receive_packet…」的讯息在记录文件中，表示测试样 板档案的 egress 封包已经设定完成，即将送出预期的封包。然后，我们截取部分测试工具的输出记录文件。

```
root@ryu-vm:~$ ryu-manager --verbose --test-switch-dir ./
sample_test_pattern.json ryu/ryu/tests/switch/tester.py
loading app ryu/tests/switch/tester.py
loading app ryu.controller.ofp_handler
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ryu/tests/switch/tester.py of OfTester
target_dpid =0000000000000001
tester_dpid =0000000000000002
Test files directory = ./sample_test_pattern.json

--- Test start ---
waiting for switches connection...

dpid=0000000000000002 : Join tester SW.
dpid=0000000000000001 : Join target SW.

sample: Router test

send_packet:[ethernet(dst='22:22:22:22:22:22',ethertype=2048,src
='11:11:11:11:11:11'), ipv4(csum=53560,dst='192.168.30.10',flags=0,
header_length=5,identification=0,offset=0,option=None,proto=6,src
='192.168.10.10',tos=32,total_length=59,ttl=64,version=4), tcp(ack=0,bits
=0,csum=33311,dst_port=2222,offset=6,option='\x00\x00\x00\x00',seq=0,
src_port=11111,urgent=0,window_size=0), '\x01\x02\x03\x04\x05\x06\x07\x08
\t\       n\       x0b\       x0c\       r\       x0e\       x0f       ']
egress:[ethernet(dst='bb:bb:bb:bb:bb:bb',ethertype=2048,src='aa:aa:aa:aa:
aa:aa'), ipv4(csum=53816,dst='192.168.30.10',flags=0,header_length=5,
identification=0,offset=0,option=None,proto=6,src='192.168.10.10',tos=32,
total_length=59,ttl=63,version=4), tcp(ack=0,bits=0,csum=33311,dst_port
=2222,offset=6,option='\x00\x00\x00\x00',seq=0,src_port=11111,urgent=0,
window_size=0), '\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f']
packet_in:[]
dpid=0000000000000002 : receive_packet[ethernet(dst='bb:bb:bb:bb:bb:bb',
ethertype=2048,src='aa:aa:aa:aa:aa:aa'), ipv4(csum=53816,dst
='192.168.30.10',flags=0,header_length=5,identification=0,offset=0,option
=None,proto=6,src='192.168.10.10',tos=32,total_length=59,ttl=63,version
=4), tcp(ack=0,bits=0,csum=33311,dst_port=2222,offset=6,option='\x00\x00\
x00\x00',seq=0,src_port=11111,urgent=0,window_size=0), '\x01\x02\x03\x04\
x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f']
    static routing table                              OK
---   Test end   ---
```

下面列出实际的 OpenFlow 交换器所登录的 Flow Entry。你可以看到测试工具所产生的封包 match 所登录的 Flow Entry，而且 n_packets 计数器数字被增加。

Node: s1:

```
root@ryu-vm:~# ovs-ofctl -O OpenFlow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=56.217s, table=0, n_packets=1, n_bytes=73, priority
=0,ip,nw_dst=192.168.30.0/24  actions=set_field:aa:aa:aa:aa:aa:aa->eth_src
,set_field:bb:bb:bb:bb:bb:bb->eth_dst,dec_ttl,output:2
```

### 12.3.3 错误讯息一览表

下面列出所有测试工具可能会显示的错误讯息。

| 错误讯息 | 说明 |
| --- | --- |
| Failed to initialize flow tables: barrier request timeout. | 初始待测交换器的 flow entry失败 |
| | (BarrierRequest作业逾时) |
| Failed to initialize flow tables: [err_msg] | 初始待测交换器的 flow entry失败 |
| | (接收到 FlowMod 错误讯息) |
| Failed to initialize flow tables of tester_sw: barrier request timeout. | 初始辅助交换器的 flow entry失败 |
| | (BarrierRequest作业逾时) |
| Failed to initialize flow tables of tester_sw: [err_msg] | 初始辅助交换器的 flow entry失败 |
| | (接收到 FlowMod 错误讯息) |
| Failed to add flows: barrier request timeout. | 待测交换器的 flow entry新增失败 |
| | (BarrierRequest作业逾时) |
| Failed to add flows: [err_msg] | 待测交换器的 flow entry新增失败 |
| | (接收到 FlowMod 错误讯息) |
| Failed to add flows to tester_sw: barrier request timeout. | 辅助交换器的 flow entry新增失败 |
| | (BarrierRequest作业逾时) |
| Failed to add flows to tester_sw: [err_msg] | 辅助交换器的 flow entry新增失败 |
| | (接收到 FlowMod 错误讯息) |
| Failed to add meters: barrier request timeout. | 待测交换器的 meter entry新增失败 |
| | (BarrierRequest作业逾时) |
| Failed to add meters: [err_msg] | 待测交换器的 meter entry新增失败 |
| | (接收到 MeterMod错误讯息) |
| Failed to add groups: barrier request timeout. | 待测交换器的 group entry 新增失败 |
| | (BarrierRequest作业逾时) |
| Failed to add groups: [err_msg] | 待测交换器的 group entry 新增失败 |
| | (接受到 GroupMod 错误讯息) |
| Added incorrect flows: [flows] | 待测交换器的 flow entry新增失败 |
| | (新增的 flow entry 不符合规范) |
| Failed to add flows: flow stats request timeout. | 待测交换器的 flow entry新增失败 |
| | (FlowStats Request 作业逾时) |
| Failed to add flows: [err_msg] | 待测交换器的 flow entry新增失败 |
| | (接受到FlowStats Request 的错误讯息) |
| Added incorrect meters: [meters] | 待测交换器的 meter entry新增错误 |
| | (新增的 meter entry 不符合规范) |
| Failed to add meters: meter config stats request timeout. | 待测交换器的 meter entry新增失败 |
| | (MeterConfigStats Request 作业逾时) |
| Failed to add meters: [err_msg] | 待测交换器的 meter entry新增失败 |
| | (接受到MeterConfigStats Request错误讯息) |
| Added incorrect groups: [groups] | 待测交换器的 group entry 新增错误 |
| | (新增的 group entry 不符合规范) |

下一页继续

表 12.1 – 呈接上一页

| 错误讯息 | 说明 |
| --- | --- |
| Failed to add groups: group desc stats request timeout. | 待测交换器的 group entry 新增失败 |
| | (GroupDescStats Request 作业逾时) |
| Failed to add groups: [err_msg] | 待测交换器的 group entry 新增失败 |
| | (接受到 GroupDescStats Request 错误讯息) |
| Failed to request port stats from target: request timeout. | 待测交换器的 PortStats 取得失败 |
| | (PortStats Request 作业逾时) |
| Failed to request port stats from target: [err_msg] | 待测交换器的 PortStats 取得失败 |
| | (接受到 PortStats Request 的错误讯息) |
| Failed to request port stats from tester: request timeout. | 辅助交换器的 PortStats 取得失败 |
| | (PortStats Request 作业逾时) |
| Failed to request port stats from tester: [err_msg] | 辅助交换器的 PortStats 取得失败 |
| | (接受到 PortStats Request 的错误讯息) |
| Received incorrect [packet] | 封包接收错误 |
| | (接受到错误的封包) |
| Receiving timeout: [detail] | 封包接收错误 |
| | (作业逾时) |
| Failed to send packet: barrier request timeout. | 封包传送失败 |
| | (Barrier Request 作业逾时) |
| Failed to send packet: [err_msg] | 封包传送失败 |
| | (Packet-Out 的错误讯息) |
| Table-miss error: increment in matched_count. | table-miss 错误 |
| | (match flow) |
| Table-miss error: no change in lookup_count. | table-miss 错误 |
| | (封包不会被 flow table 所处理) |
| Failed to request table stats: request timeout. | table-miss 失败 |
| | (TableStats Request 作业逾时) |
| Failed to request table stats: [err_msg] | table-miss 失败 |
| | (接收到 TableStats Request 的错误讯息) |
| Added incorrect flows to tester_sw: [flows] | 辅助交换器 flow entry 新增错误 |
| | (新增的 flow entry 不符合规范) |
| Failed to add flows to tester_sw: flow stats request timeout. | 辅助交换器 flow entry 新增失败 |
| | (FlowStats Request 作业逾时) |
| Failed to add flows to tester_sw: [err_msg] | 辅助交换器 flow entry 新增失败 |
| | (FlowStats Request 的错误讯息) |
| Failed to request flow stats: request timeout. | 测试 Throughput 时，辅助交换器 flow entry |
| | request 失败 (FlowStats Request 作业逾时) |
| Failed to request flow stats: [err_msg] | 测试 Throughput 时，辅助交换器 flow entry |

下一页继续

表 12.1 – 呈接上一页

| 错误讯息 | 说明 |
| --- | --- |
| | request 失败 (FlowStats Request 的错误讯息) |
| Received unexpected throughput: [detail] | 测试 Throughput 时，得到非预期的结果 |
| Disconnected from switch | 待测交换器或辅助交换器的连结中断 |

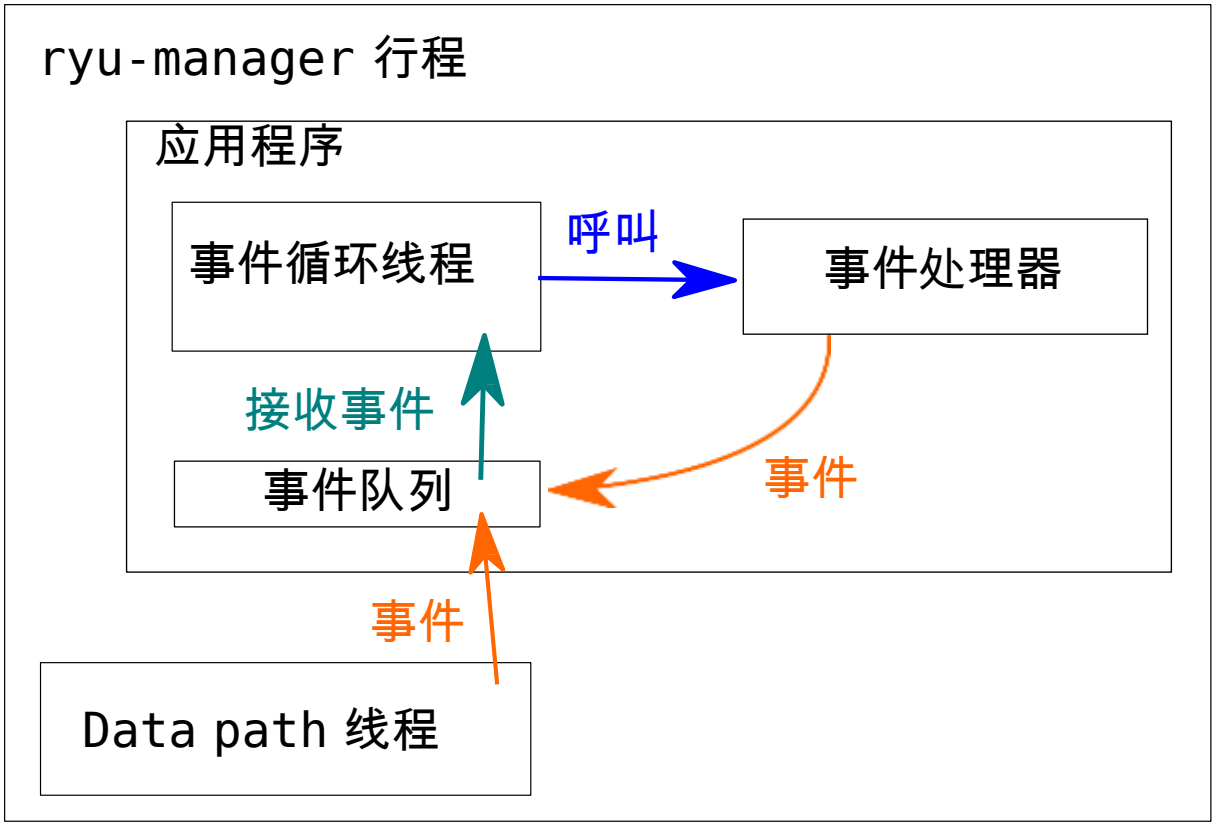
CHAPTER 13

# 组织架构

本章介绍 Ryu 的架构。关于每一个 Class 请参照 API 参考数据 来得到更细节的数据。

## 13.1 应用程序开发模型（Application programming model）

以下是 Ryu 应用程序的开发模型




### 13.1.1 应用程序（Application）

应用程序是继承 `ryu.base.app_manager.RyuApp` 而来。User logic 被视作是一个应用程序。

### 13.1.2 事件（Event）

事件是继承 `ryu.controller.event.EventBase` 而来，并藉由 Transmitting 和 receiving event 来相互沟通讯息。

### 13.1.3 事件队列（Event queue）

每个应用程序都有一个自己的队列用来接受事件讯息。

### 13.1.4 线程（Thread）

Ryu 采用 eventlet 来实现多线程。因为线程是不可插断的（non-preemptive），因此在使用 上要特别注意长时间运行所带来的风险。

#### 事件循环（Event loop）

当应用程序执行时，将会有一个线程自动被产生用来执行该应用程序。该线程将会做为事 件循环的模式来执行。如果在事件队列中发现有事件存在，该事件循环将会读取该事件并且呼叫相对应的事件处理器来处理它。

#### 额外的线程（Additional thread）

如果需要的话，你可以使用 hub.spawn 产生额外的线程用来执行特殊的应用程序功能。

#### eventlet

虽然你可以直接使用 eventlet 所提供的所有功能，但不建议你这么做。请使用 hub module所包装过的功能取代直接使用 eventlet。

### 13.1.5 事件处理器（Event handler）

藉由使用 `ryu.controller.handler.set_ev_cls` 装饰器类别来定义自己的事件管理器。当定 义的事件发生时，应用程序中的事件循环将会侦测到并呼叫对应的事件管理器。

CHAPTER 14

# 协助项目开发

开放原始码最大的优点在于对开发项目的共同参与，本章将介绍如何参与 Ryu 项目的开发。

## 14.1 参与项目

Mailing list 是 Ryu 项目开发者主要聚集的地方。因此首先就是要加入 Mailing list 来得到最新的消息。

https://lists.sourceforge.net/lists/listinfo/ryu-devel

在 Mailing list 中主要使用英文来进行沟通。当你在使用 Ryu 上遇到了困难，包括不知道如何使用或发现了程序上的错误。任何时候都欢迎在 Mailing list 上提出你的疑问。千万不需要对于遇到了麻烦而感到不好意思结果不敢发言。因为使用 Open source 本身并提出疑问对于项目而言就是一种贡献。

## 14.2 开发环境

接下来说明 Ryu 项目在开发的环境上需要注意的事情。

### 14.2.1 Python 版本

目前 Ryu 项目支持 Python 2.6 以上的版本，并请勿使用比 2.7 更早之前的版本。

Python 3.0 以上的版本在目前尚未提供支持。但请小心撰写程序代码以保持扩充性并随时注意将 来改版的可能。

### 14.2.2 程序代码撰写风格（Coding Style ）

Ryu 的原始码是遵守 PEP8 的规范来开发。当你在送交程序代码时，请务必确认程序代码符合 PEP8 的规范。

http://www.python.org/dev/peps/pep-0008/

为了让开发者更好的检查自己的程序代码是否符合 PEP8 规范，这边有个检查器可以帮助开发者，请参考下述连结。

https://pypi.python.org/pypi/pep8

### 14.2.3 测试

Ryu 项目本身提供自动化测试的工具，最容易也是最常使用的是单元测试（Unit test ），该功能也同样包含在Ryu项目内。在送出程序代码之前，务必确认通过所有的单元测试，确保您的修改 不会对现有的程序代码造成影响。如果是增加的程序代码，请在单元测试及批注的地方详细说明该 功能。

```
$ cd ryu/
$ ./run_tests.sh
```

## 14.3 送交更新的程序代码

因为新功能的增加、程序错误的修复而需要对原始码进行修改时，请针对修改的部分制作更新档并寄送到 Mailing list。，我们可以先在 Mailing list 中进行讨论是否更新的必要。

备注: Ryu 的原始码目前存放在 GitHub 上进行托管。但请注意 Pull request 并不是开发程序的必要条件。

在更新档的格式上，我们期望收到如同更新 Linux Kernel 相同或类似的格式。下面的例子是用来说明送交程序更新文件时所需要符合的格式。请参阅相关的文件以达到该要求。

http://lxr.linux.no/linux/Documentation/SubmittingPatches 接下来说明整个流程。

1. 原始码 Checkout

   首先是从 Ryu 项目下载原始码。你可以在 Github 上 fork 一份项目到自己的账号。下面的例子说明如何复制一份项目到自己的工作环境中。

   ```
   $ git clone https://github.com/osrg/ryu.git
   $ cd ryu/
   ```

2. 原始码的修改及增加

   对 Ryu 的原始码进行必要的修改。接着将变更的内容进行 Commit。

   ```
   $ git commit -a``
   ```

3. 更新档的制作

   对于变更的地方制作更新档。不要忘记加入 Signed-off-by: 在即将送出的更新档中。这个署名是用来辨认你对开放原始码的贡献，以及满足相关的授权所使用。

   ```
   $ git format-patch origin -s
   ```

4. 送出更新檔

   已经完成的更新档，在确认过没有问题之后，请寄送到 Mailing list。你可以使用自 己熟悉的邮件软件或寄送方法，或者也可以使用 git 内建的寄送邮件指令。

   ```
   $ git send-email 0001-sample.patch
   ```

5. 等待回应

   等待开发团队的讨论以及对于更新档的响应。原则上整个流程已经结束，若是你的更新档有任何问题，你需要针对提问进行说明，必要时再次修改并送交更新档。

CHAPTER 15

# 应用案例

本章介绍几个使用Ryu作为产品或服务的案例。

## 15.1 Stratosphere SDN Platform (Stratosphere)

Stratosphere SDN Platform（以下简称SSP）是Stratosphere公司所开发的软件。经由SSP可以建构Edge Overlay-model的虚拟网络，它使用到的Tunnelling技术有：VXLAN、STT和MPLS。

每一种通道协议可在VLAN间相互转换。因为每一种Tunnelling协议的ID通常都大于VLAN的12 bits，所以使用L2 segments来管理而不直接使用VLAN。而且，SSP可以和其他软件如：OpenStack、CloudStack 或 IaaS协同运作。

在1.1.4的版本中，SSP使用OpenFlow实作它的功能并嵌入Ryu做为它的Controller。其中一个理由是为了支持OpenFlow 1.1之后的版本。为了在SSP之上支持MPLS，考虑导入已支持OpenFlow 1.1的框架在协议层。

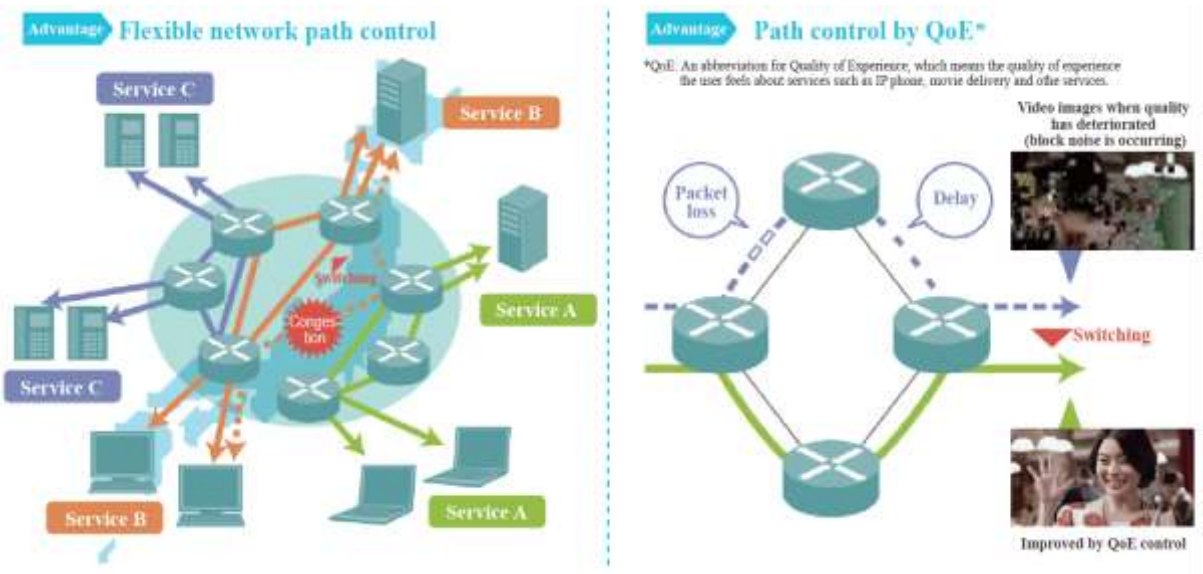
备注: OpenFlow协议本身支持的程度也是一个非常重要的考虑点。由于规范内有许多列为选项的功能，因此必须注意交换器对于该功能的支持程度是否足够。

开发语言 Python 也是一个很重要的原因。Stratosphere 所用的开发语言就是 Python，许多 SSP的组件也都是使用Python所撰写。Python本身的自我描述能力很高，对于习惯使用的开发者来说，可以大幅提升开发的效率。

软件是由多个Ryu应用程序所组成，并透过REST API与其他的SSP组件沟通。将软件透过切割功能的方式分为多个应用程序，最基本的方法就是保持良好的原始码。

## 15.2 SmartSDN Controller（NTT COMWARE）

「SmartSDN Controller」是一个提供集中管理功能（网络虚拟化/优化）的 SDN Controller。




「SmartSDN Controller」有以下两个特征。

1. 拥有弹性的虚拟网络路由管理

   在相同的实体网络中建构多个虚拟网络，可以提供一个有弹性的环境以响应使 用者对于变动性的要求，设备则可以被有效的利用以降低购买成本。而且每一 台装置、交换器、路由器的设定均集中管理之后，可以清楚的知道整体网络目 前的状态。当网络发生故障或者通讯状况发生改变时，可以做相对应的变更处 置。

   藉由注重用户的客户体验质量（「QoE」：Quality of Experience ）、网络通讯 质量（带宽、延迟、封包丢失、流量变化），判断客户体验质量后选择较好的路由以达到维持稳定的服务。

2. 确保网络的高度弹性及可用性

   为了在Controller发生故障的时候服务可以持续提供，备援的设定是必要的。主动产生封包并在节点间通讯，得以早期发现一般的监控无法及时发现的网络问 题，并进行各种检验（路由测试、线路测试... 等）。

   另外经由网络设计和状态确认的可视化图形界面（GUI ），让即使没有专业技能的操作人员都得以进行控制，降低网络运用的成本。

因此在「SmartSDN Controller」的开发上所选定的框架必须满足下列的条件。

- 框架必须尽可能的支持 OpenFlow规范的定义
- 框架必须随时更新以追上持续更新的 OpenFlow版本 在

这之中 Ryu 是：

- 全面的支持各种 OpenFlow版本
- 最快的跟上 OpenFlow所发布的新版本。而且社群相当活跃快速响应及解决臭虫。
- sample code / 文件均相当丰富。 由于以上的特征，

所以是相当适合采用的开发框架。