



Using OpenFlow 1.3

RYU SDN Framework

RYU project team

前言	1
1 交換器 (Switching Hub)	3
1.1 Switching Hub	3
1.2 OpenFlow 實作的交換器	3
1.3 在 Ryu 上實作交換器	5
1.4 執行 Ryu 應用程式	14
1.5 本章總結	19
2 流量監控 (Traffic Monitor)	21
2.1 定期檢查網路狀態	21
2.2 安裝 Traffic Monitor	21
2.3 執行 Traffic Monitor	27
2.4 本章總結	29
3 REST API	31
3.1 整合 REST API	31
3.2 安裝包含 REST API 的 Switching Hub	31
3.3 安裝 SimpleSwitchRest13 class	33
3.4 安裝 SimpleSwitchController Class	34
3.5 執行包含 REST API 的 Switching Hub	36
3.6 本章總結	38
4 網路聚合 (Link Aggregation)	39
4.1 網路聚合 (Link Aggregation)	39
4.2 執行 Ryu 應用程式	39
4.3 實作 Ryu 的網路聚合功能	50
4.4 本章總結	59
5 生成樹 (Spanning Tree)	61
5.1 Spanning Tree	61
5.2 執行 Ryu 應用程式	63
5.3 使用 OpenFlow 完成生成樹	73
5.4 使用 Ryu 實作生成樹	74
5.5 本章總結	83
6 OpenFlow 通訊協定	85
6.1 Match	85

6.2	Instruction	86
6.3	Action	87
7	ofproto 函式庫	89
7.1	簡單說明	89
7.2	相關模組	89
7.3	基本使用方法	90
8	封包函式庫	93
8.1	基本使用方法	93
8.2	應用程式範例	95
9	OF-Config 函式庫	99
9.1	OF-Config 通訊協定	99
9.2	函式庫架構	99
9.3	使用範例	100
10	防火牆 (Firewall)	101
10.1	Single tenant 操作範例	101
10.2	Multi tenant 操作範例	110
10.3	REST API 列表	114
11	路由器 (Router)	117
11.1	Single Tenant 的操作範例	117
11.2	Multi-tenant 的操作範例	127
11.3	REST API 列表	139
12	OpenFlow 交換器測試工具	141
12.1	測試工具概要	141
12.2	使用方法	143
12.3	測試工具使用範例	146
13	組織架構	155
13.1	應用程式開發模型 (Application programming model)	155
14	協助專案開發	157
14.1	參與專案	157
14.2	開發環境	157
14.3	送交更新的程式碼	158
15	應用案例	159
15.1	Stratosphere SDN Platform (Stratosphere)	159
15.2	SmartSDN Controller (NTT COMWARE)	159

前言

本書是給那些使用 Ryu 作為開發的框架，而目的是為了實現軟體定義網路（Software Defined Networking，SDN）而寫的一本參考書。

那麼為什麼要選擇 Ryu 作為開發平台呢？

我們衷心的希望您可以在本書中找到解答。

建議您依照本書的章節，依序的閱讀第 1 章至第 5 章。首先第 1 章是 Simple Switch 的實作，接著後面的章節是流量監控（Traffic Monitor）以及網路聚合（Link Aggregation）。透過實際的例子，我們將介紹 Ryu 的程式是如何運作的。

第 6 章到第 9 章，我們將詳細的說明 OpenFlow 通訊協定（OpenFlow Protocol）以及封包函數的使用方法。接著第 10 章到第 12 章，我們會討論如何使用 Ryu 內建的防火牆（Firewall）和測試工具（test tool）來開發應用程式。第 13 到 15 章將介紹 Ryu 的架構（architecture）及實際應用案例。

最後，我們非常感謝您對於 Ryu 專案的青睞及支持，特別對於 Ryu 的使用者而言。希望在不久的將來能有機會在 Mailing List 上得到來自于您的寶貴意見，讓我們一起加入 Ryu 專案並進行開發吧。

繁體中文版本是由台灣資訊工業策進會－ SDN 團隊協助整理及翻譯，除了不吝批評與指教之外也歡迎更多朋友加入，讓中文讀者可以更快更即時的了解 Ryu 這個優秀的開放原始碼的框架。

Contents

交換器 (Switching Hub)

本章將會用簡單的 Switching hub 安裝做為題材，說明 Ryu 如何安裝一個應用程式。

1.1 Switching Hub

在交換器中有許多的功能。在這邊我們將看到擁有下列簡單功能的交換器。

- 學習連接到連接埠的 host 之 MAC 位址，並記錄在 MAC 位址表當中。
- 對於已經記錄下來的 MAC 位址，若是收到送往該 MAC 位址的封包，則轉送該封包到相對應的連接埠。
- 對於未指定目標位址的封包，則執行 Flooding。

讓我們使用 Ryu 來實現這樣一個交換器吧。

1.2 OpenFlow 實作的交換器

OpenFlow 交換器會接受來自于 controller 的指令並達到以下功能。

- 對於接收到的封包進行修改或針對指定的連接埠進行轉送。
- 對於接收到的封包進行轉送到 Controller 的動作 (Packet-In)。
- 對於接收到來自 Controller 的封包轉送到指定的連接埠 (Packet-Out)。

上述的功能所組合起來的就是一台交換器的實現。

首先，利用 Packet-In 的功能來達到 MAC 位址的學習。Controller 使用 Packet-In 接收來自交換器的封包之後進行分析，得到連接埠相關資料以及所連接的 host 之 MAC 位址。

在學習之後，對所收到的封包進行轉送。將封包的目的位址，在已經學習的 host 資料中進行檢索，根據檢索的結果會進行下列處理。

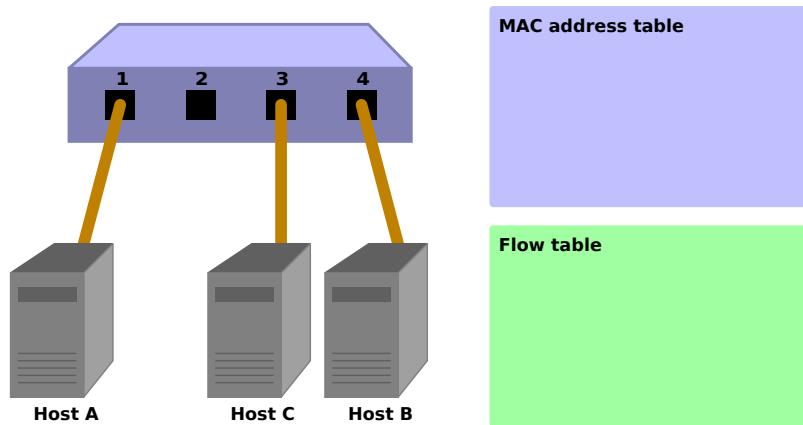
- 如果是已經存在記錄中的 host：使用 Packet-Out 功能轉送至先前所對應的連接埠
- 如果是尚未存在記錄中的 host：使用 Packet-Out 功能來達到 Flooding

下面將一步一步的說明並附上圖片以幫助理解。

1. 初始狀態

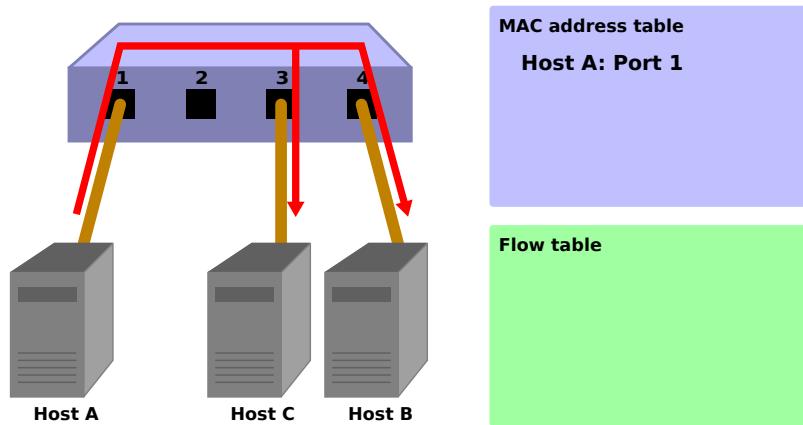
Flow table 為空白的狀況。

將 host A 接到連接埠 1，host B 接到連接埠 4，host C 接到連接埠 3。



2. host A → host B

當 host A 向 host B 發送封包。這時後會觸發 Packet-In 訊息。host A 的 MAC 位址會被連接埠 1 細記錄下來。由於 host B 的 MAC 位址尚未被學習，因此會進行 Flooding 並將封包往 host B 和 host C 發送。



Packet-In:

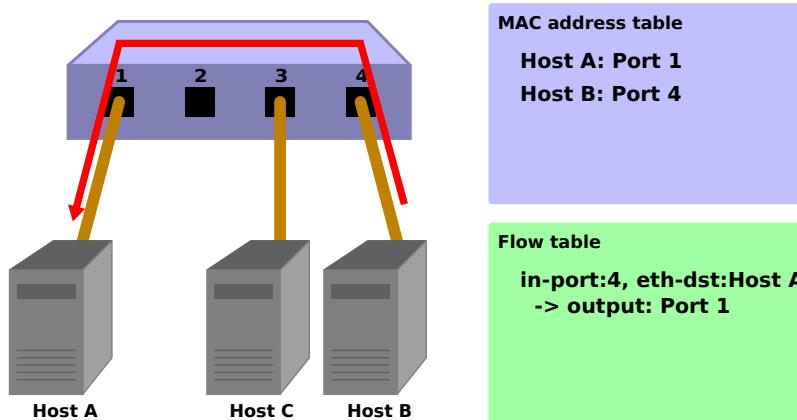
```
in-port: 1
eth-dst: host B
eth-src: host A
```

Packet-Out:

```
action: OUTPUT: Flooding
```

3. host B → host A

封包從 host B 向 host A 返回時，在 Flow table 中新增一筆 Flow Entry，並將封包轉送到連接埠 1。因此該封包並不會被 host C 收到。



Packet-In:

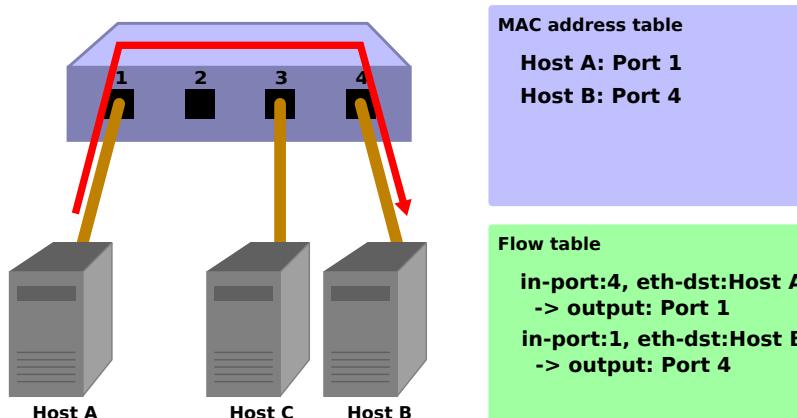
```
in-port: 4
eth-dst: host A
eth-src: host B
```

Packet-Out:

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

4. host A → host B

再一次，host A 向 host B 發送封包，在 Flow table 中新增一個 Flow Entry 接著轉送封包到連接埠 4。



Packet-In:

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

Packet-Out:

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

接下來，讓我們實際來看一下在 Ryu 當中實作交換器的原始碼。

1.3 在 Ryu 上實作交換器

Ryu 的原始碼之中有提供交換器的程式原始碼。

ryu/app/simple_switch_13.py

OpenFlow 其他的版本也有相對應的原始碼，例如 simple_switch.py (OpenFlow 1.0) 和 simple_switch_12.py (OpenFlow 1.2)。我們現在要來看的則是 OpenFlow 1.3 的版本。

由於原始碼不多，因此我們把全部都拿來檢視。

```
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.OFPI_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']

        pkt = packet.Packet(msg.data)
        eth = pkt.get_protocols(ether.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)

```

那麼我們開始看一下其中的內容吧。

1.3.1 類別的定義和初始化

為了要實作 Ryu 應用程式，因此繼承了 `ryu.base.app_manager.RyuApp`。接著為了使用 OpenFlow 1.3，將 `OFP_VERSIONS` 指定為 OpenFlow 1.3。

然後，MAC 位址表的 `mac_to_port` 也已經被定義。

OpenFlow 通訊協定中有些程序像是握手協定（handshake），是定義好讓 OpenFlow 交換器和 Controller 之間進行通訊時使用。但這些細節，對於一個 Ryu 應用程式來說是不用擔心或需要特別處理的。

```

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)

對於 Ryu 來說，接收到任何一個 OpenFlow 訊息即會產生一個相對應的事件。而 Ryu 應用程式則是必須實作事件管理以處理相對應發生的事件。

事件管理 (Event Handler) 是一個擁有事件物件 (Event Object) 做為參數，並且使用 ``ryu.controller.handler.set_ev_cls`` 修飾 (Decorator) 的函數。

set_ev_cls 則指定事件類別得以接受訊息和交換器狀態作為參數。

事件類別名稱的規則為 ryu.controller.ofp_event.EventOFP + <OpenFlow 訊息名稱>，例如：在 Packet-In 訊息的狀態下的事件名稱為 EventOFPPacketIn。詳細的內容請參考 Ryu 的文件 [API 參考資料](#)。對於狀態來說，請指定下面列表的其中一項。

名稱	說明
ryu.controller.handler.HANDSHAKE_DISPATCHER	交換 HELLO 訊息
ryu.controller.handler.CONFIG_DISPATCHER	接收 SwitchFeatures 訊息
ryu.controller.handler.MAIN_DISPATCHER	一般狀態
ryu.controller.handler.DEAD_DISPATCHER	連線中斷

新增 Table-miss Flow Entry

OpenFlow 交換器的握手協議完成之後，新增 Table-miss Flow Entry 到 Flow table 中為接收 Packet-In 訊息做準備。

具體來說，接收到 Switch features (Features reply) 訊息後就會新增 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 是用來儲存對應事件的 OpenFlow 訊息類別實體。在這個例子中則是 ryu.ofproto.ofproto_v1_3_parser.OFPSwitchFeatures。

msg.datapath 這個訊息是用來儲存 OpenFlow 交換器的 ryu.controller.controller.Datapath 類別所對應的實體。

Datapath 類別是用來處理 OpenFlow 交換器重要的訊息，例如執行與交換器的通訊和觸發接收訊息相關的事件。

Ryu 應用程式所使用的主要屬性如下：

名稱	說明
id ofproto	連接 OpenFlow 交換器的 ID (datapath ID)。 表示使用的 OpenFlow 版本所對應的 ofproto module。目前的狀況會是下述的其中之一。 ryu.ofproto.ofproto_v1_0 ryu.ofproto.ofproto_v1_2 ryu.ofproto.ofproto_v1_3 ryu.ofproto.ofproto_v1_4
ofproto_parser	和 ofproto 一樣，表示 ofproto_parser module。目前的狀況會是下述的其中之一。 ryu.ofproto.ofproto_v1_0_parser ryu.ofproto.ofproto_v1_2_parser ryu.ofproto.ofproto_v1_3_parser ryu.ofproto.ofproto_v1_4_parser

Ryu 應用程式中 Datapath 類別的主要方法如下：

```
send_msg(msg)
```

發送 OpenFlow 訊息。msg 是發送 OpenFlow 訊息
ryu.ofproto.ofproto_parser.MsgBase 類別的子類別。

交換器本身不僅僅使用 Switch features 訊息，還使用事件處理以取得新增 Table-miss Flow Entry 的時間點。

```
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 的優先權為 0 即最低的優先權，而且此 Entry 可以 match 所有的封包。這個 Entry 的 Instruction 通常指定為 output action，並且輸出的連接埠將指向 Controller。因此當封包沒有 match 任何一個普通 Flow Entry 時，則觸發 Packet-In。

註解：目前（2014 年 1 月），市面上的 Open vSwitch 對於 OpenFlow 1.3 的支援並不完整，而且對於 OpenFlow 1.3 以前的版本 Packet-In 是個基本的功能。包括 Table-miss Flow Entry 也尚未被支援，僅僅是使用一般的 Flow Entry 取代。

空的 match 將被產生為了 match 所有的封包。match 表示於 OFPMatch 類別中。

接下來，為了轉送到 Controller 連接埠，OUTPUT action 類別（OFPActionOutput）的實例將會被產生。Controller 會被指定為封包的目的地，OFPCML_NO_BUFFER 會被設定為 max_len 以便接下來的封包傳送。

註解：送往 Controller 的封包可以僅只傳送 header 部分（Ethernet header），剩下的則存在緩衝區間中以增加效率。但目前（2014 年 1 月）Open vSwitch 存在臭蟲的關係，會將所有的封包都傳送，並不會只傳送 header。

最後將優先權設定為 0（最低優先權），然後執行 add_flow() 方法以發送 Flow Mod 訊息。
add_flow() 方法的內容將會在稍後進行說明。

Packet-in 訊息

為了接收處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 類別經常使用的屬性如下列所示。

名稱	說明
match	ryu.ofproto.ofproto_v1_3_parser.OFPMatch 類別的實體，用來儲存接收封包的 Meta 資訊。
data	接收封包本身的 binary 資料
total_len	接收封包的資料長度
buffer_id	接收封包的內容若是存在 OpenFlow 交換器上時所指定的 ID 如果在沒有 buffer 的狀況下，則設定 ryu.ofproto.ofproto_v1_3.OFP_NO_BUFFER

更新 MAC 位址表

```
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

    # ...
```

從 OFPPacketIn 類別的 match 得到接收埠 (in_port) 的資訊。目的 MAC 位址和來源 MAC 位址使用 Ryu 的封包函式庫，從接收到封包的 Ethernet header 取得。

藉由得知目的 MAC 位址和來源 Mac 位址，更新 MAC 位址表。

為了可以對應連接到多個 OpenFlow 交換器，MAC 位址表和每一個交換器之間的識別，就使用 datapath ID 來進行確認。

判斷轉送封包的連接埠

目的 MAC 位址若存在于 MAC 位址表，則判斷該連接埠的號碼為輸出。反之若不存在于 MAC 位址表則 OUTPUT action 類別的實體並生成 flooding (OFPP_FLOOD) 紿目的連接埠使用。

```
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.OFPPMatch(in_port=in_port, eth_dst=dst)
self.add_flow(datapath, 1, match, actions)

# ...

```

若是找到了目的 MAC 位址，則在交換器的 Flow table 中新增。

Table-miss Flow Entry 包含 match 和 action，並透過 add_flow() 來新增。

不同於平常的 Table-miss Flow Entry，這次將加上設定 match 條件。本次交換器實作中，接收埠 (in_port) 和目的 MAC 位址 (eth_dst) 已指定。例如，接收到來自連接埠 1 的封包就傳送到 host B。

在這邊指定 Flow Entry 優先權為 1，而優先權的值越大，表示有更高的優先權。因此，這邊新增的 Flow Entry 將會先於 Table-miss Flow Entry 而被執行。

上述的內容包含 action 整理如下，這些 Entry 會被新增至 Flow Entry：

連接埠 1 接收到的封包，若是要轉送至 host B (目的 MAC 位址 B) 的封包則轉送至連接埠 4。

提示: 在 OpenFlow 中，有個邏輯連接埠叫做 NORMAL 並在規範中被列為選項（也就是說可以不進行實作）。當被指定的連接埠為 NORMAL 時，傳統 L2/L3 的功能將會被啟用來處理封包。意思是當把所有輸出的埠均設定為 NORMAL 時，交換器將會視作一個普通的交換器而存在。跟一般交換器的差別在於我們是使用 OpenFlow 來達到這樣的功能。

新增 Flow Entry 的處理

Packet-In handler 的處理尚未說明，在這之前我們先來看一看新增 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)]

```

對於 Flow Entry 來說，設定 match 條件以分辨目標封包、設定 instruction 以處理封包以及 Entry 的優先權和有效時間。

對於交換器的的實作，Apply Actions 是用來設定那些必須立即執行的 action 所使用。

最後透過 Flow Mod 訊息將 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 訊息的類別為 OFPFlowMod。使用 OFPFlowMod 所產生的實體透過 Datapath.send_msg() 方法來發送訊息給 OpenFlow 交換器。

OFPFlowMod 類別的建構子參數相當的多，但是在大多數的情況下都有其預設值，原始碼中括號內的部分即是預設值。

datapath

OpenFlow 交換器以及 Flow table 的操作都是透過 Datapath 類別的實體來進行。在一般的情況下，會由事件傳遞給事件管理的訊息中取得，例如：Packet-In 訊息。

cookie (0)

Controller 所設定儲存的資料，在 Entry 的更新或者刪除的時所需要使用的資料都會放在這邊，當做過濾器使用，而且不可以作為封包處理的參數。

cookie_mask (0)

Entry 的更新或刪除時，若是該值為非零，則做為指定 Entry 的 cookie 使用。

table_id (0)

指定 Flow Entry 的 Table ID 。

command (ofproto_v1_3.OFPFC_ADD)

指定要執行何項操作。

名稱	說明
OFPFC_ADD	Flow Entry 新增
OFPFC MODIFY	Flow Entry 更新
OFPFC MODIFY_STRICT	嚴格的 Flow Entry 更新
OFPFC_DELETE	Flow Entry 刪除
OFPFC_DELETE_STRICT	嚴格的 Flow Entry 刪除

idle_timeout (0)

Flow Entry 的有效期限，以秒為單位。Flow Entry 如果未被參照而且超過了指定的時間之後，該 Flow Entry 將會被刪除。如果 Flow Entry 有被參照，則超過時間之後會重新歸零計算。

在 Flow Entry 被刪除之後就會發出 Flow Removed 訊息通知 Controller 。

hard_timeout (0)

Flow Entry 的有效期限，以秒為單位。跟 idle_timeout 不同的地方是，hard_timeout 在超過時限後並不會重新歸零計算。也就是說跟 Flow Entry 與有沒有被參照無關，只要超過指定的時間就會被刪除。

跟 idle_timeout 一樣，當 Flow Entry 被刪除時，Flow Removed 訊息將會被發送來通知 Controller 。

priority (0)

Flow Entry 的優先權。數值越大表示權限越高。

buffer_id (ofproto_v1_3.OFP_NO_BUFFER)

指定 OpenFlow 交換器上用來儲存封包的緩衝區 ID。緩衝區 ID 會放在通知 Controller 的 Packet-In 訊息中，並且和接下來的 OFPP_TABLE 所指定的輸出埠和 Flow Mod 訊息處理時可以被參照。當發送的命令訊息為 OFPFC_DELETE 或 OFPFC_DELETE_STRICT 時，會忽略本數值。

如果不指定緩衝區 ID 的時候，必須使用 OFP_NO_BUFFER 作為其設定值。

out_port (0)

OFPFC_DELETE 和 OFPFC_DELETE_STRICT 命令用來指定輸出埠的參數。命令為 OFPFC_ADD 、 OFPFC MODIFY 、 OFPFC MODIFY_STRICT 時則可以忽略。

若要讓本參數無效則指定輸出埠為 OFPP_ANY。

`out_group (0)`

跟 `out_port` 一樣，作為一個輸出埠，但是轉到特定的 group。

若要使其無效，則指定為 OFPG_ANY。

`flags (0)`

下列的 `flags` 可以被組合使用。

名稱	說明
<code>OFPFF_SEND_FLOW_Rem</code>	Flow Entry 被移除的時候，對 Controller 發送 Removed 訊息。
<code>OFPFF_CHECK_OVERLAP</code>	使用 <code>OFPFC_ADD</code> 時，檢查是否有重複的 Flow Entry 存在。 若是有則觸發 Flow Mod 失敗，並返回錯誤訊息。
<code>OFPFF_RESET_COUNTS</code>	重設該 Flow Entry 的 packet counter 和 byte counter。
<code>OFPFF_NO_PKT_COUNTS</code>	關閉該 Flow Entry 的 packet counter 功能。
<code>OFPFF_NO_BYT_COUNTS</code>	關閉該 Flow Entry 的 byte counter 功能。

`match (None)`

設定 `match`。

`instructions ([])`

設定 `instruction`。

轉送封包

回到 `Packet-In handler` 並說明最後的流程。

在 MAC 位址表中找尋目的 MAC 位址，若是有找到則發送 `Packet-Out` 訊息，並且轉送封包。

```
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` 訊息相對應的類別是 `OFPPacketOut`。

`OFPPacketOut` 建構子的參數如下所示。

`datapath`

指定 OpenFlow 交換器對應的 Datapath 類別實體。

`buffer_id`

指定 OpenFlow 交換器上封包對應的緩衝區。如果不使用緩衝區，則指定為 `OFP_NO_BUFFER`。

`in_port`

指定接收封包的連接埠號。如果不使用的話就指定為 `OFPP_CONTROLLER`。

actions

指定 actions list。

data

設定封包的 binary data。主要用在 buffer_id 為 OFP_NO_BUFFER 的情況。如果使用了 OpenFlow 交換器的緩衝區則可以省略。

交換器的實作時，在 Packet-In 訊息中指定 buffer_id。若是 Packet-In 訊息中 buffer_id 被設定為無效時。Packet-In 的封包必須指定 data 以便傳送。

交換器的原始碼說明就到這邊。接下來我們將執行交換器以確認相關的動作。

1.4 執行 Ryu 應用程式

為了執行交換器，OpenFlow 交換器採用 Open vSwitch，執行環境則是在 mininet 上。

由於 Ryu 的 OpenFlow Tutorial VM 映像檔已經準備好了，有了該 VM 映像檔會讓準備工作較為簡單。

VM 映像檔

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

OpenFlow_Tutorial_Ryu3.2.ova (約 1.4GB)

相關文件 (Wiki 網頁)

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

文件中的描述和 VM 映像檔有可能使用較舊版本的 Open vSwitch 和 Ryu，請特別注意。

若是不想使用 VM 映像檔，當然可以自行打造環境。若是您決定自行搭建環境，請參考以下軟體的版本。

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
```

但是在這邊我們使用 Ryu 的 OpenFlow Tutorial VM 映像檔。

1.4.1 執行 Mininet

因為所需要的終端機 xterm 是從 mininet 中啟動，因此 X windows 的環境是必要的。

為了使用 OpenFlow Tutorial 的 VM，請把 ssh 的 X11 Forwarding 功能打開並進行登入。

```
$ ssh -X ryu@<VM 的 IP>
```

使用者名稱為 ryu、密碼也是 ryu。

登入以後使用 mn 指令啟動 Mininet 環境。

要建構的是 host 3 台，交換器 1 台的簡單環境。

mn 命令的參數如下：

名稱	數值	說明
topo	single,3	交換器 1 台、host 3 台的拓墣
mac	無	自動設定 host 的 MAC 位址
switch	ovsk	使用 Open vSwitch
controller	remote	指定外部的 OpenFlow Controller
x	無	啟動 xterm

執行的方法如下：

```
$ 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>
```

執行之後，會在 x windows 出現 5 個 xterm 視窗。分別對應到 host 1 ~ 3 ，交換器和 Controller 。

在交換器的 xterm 視窗中設定 OpenFlow 的版本，視窗的標題為「switch: s1 (root)」。

首先查看 Open vSwitch 的狀態。

switch: s1:

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

交換器（橋接器）s1 被建立，並且增加 3 個連接埠分別連線到 3 個 host。

接下來設定 OpenFlow 的版本為 1.3。

switch: s1:

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

檢查空白的 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 命令選項是用來指定 OpenFlow 版本。預設值是 OpenFlow10。

1.4.2 執行交換器

準備工作到此已經結束，接下來開始執行 Ryu 應用程式。

在視窗標題為「controller: c0 (root)」的 xterm 執行下列指令。

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)
move onto main mode
```

正在進行 OVS 的連接動作，這需要花一點時間。

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

完成之後就會顯示上述的訊息。

現在 OVS 已經連接，handshake 已經執行完畢，Table-miss Flow Entry 已經加入，正處於等待 Packet-In 的狀態。

確認 Table-miss Flow Entry 已經被加入。

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:~#
```

優先權為 0，沒有 match，action 為 CONTROLLER，重送的資料大小為 65535 (0xffff = OF-PCML_NO_BUFFER)。

1.4.3 確認操作

從 host 1 向 host 2 發送 ping。

1. ARP request

此時 host 1 並不知道 host 2 的 MAC 位址，原則上 ICMP echo request 之前的 ARP request 是用廣播的方式發送。這樣的廣播方式會讓 host 2 和 host 3 都同樣接受到訊息。

2. ARP reply

host 2 使用 ARP reply 回覆 host 1 要求。

3. ICMP echo request

現在 host 1 知道了 host 2 的 MAC 位址，因此發送 echo request 給 host 2。

4. ICMP echo reply

host 2 此時也知道了 host 1 的 MAC 位址，因此發送 echo reply 給 host 1。

原則上動作流程應該如同描述一般。

在 ping 命令執行之前，為了確認每一台 host 都可以收到，執行 tcpdump 以確認封包有確實被接收。

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
```

然後，在終端機執行 mn 命令，並從 host 1 發送 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 正常的被回覆。

繼續下去之前先確認 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，另外加入兩個優先權為 1 的 Flow Entry。

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

(1) 的 Flow Entry 會被 match 2 次 (n_packets)、(2) 的 Flow Entry 則被 match 1 次。因為 (1) 用來讓 host 2 向 host 1 傳送封包用，ARP reply 和 ICMP echo reply 都會發生 match。(2) 是用來從 host 1 向 host 2 發送訊息，由於 ARP request 是採用廣播的方式，原則上透過 ICMP echo request 完成。

然後檢查一下 simple_switch_13 log 的輸出。

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
```

第一個 Packet-In 是由 host 1 發送的 ARP request，因為透過廣播的方式所以沒有 Flow Entry 存在，故發送 Packet-Out。

第二個是從 host 2 回覆的 ARP reply，目的 MAC 位址為 host 1 因此前述的 Flow Entry (1) 被新增。

第三個是從 host 1 向 host 2 發送的 ICMP echo request，因此新增 Flow Entry (2)。

host 2 向 host 1 回覆的 ICMP echo reply 則會和 Flow Entry (1) 發生 match，故直接轉送封包至 host 1 而不需要發送 Packet-In。

最後讓我們看看每一個 host 上的 tcpdump 所呈現的結果。

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 首先發送廣播 ARP request 封包，接著接收到 host 2 送來的 ARP reply 回覆。接著 host 1 發送 ICMP echo request，host 2 則回覆 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
```

對於 host 2 則是接收 host 1 發送的 ARP request 封包，接著對 host 1 發送 ARP reply 回覆。然後接收到 host 1 來的 ICMP echo request，回覆 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
```

對 host 3 而言，僅有一開始接收到 host 1 的廣播 ARP request，未做其他動作。

1.5 本章總結

本章以簡單的交換器安裝為題，透過 Ryu 應用程式的基本安裝步驟和 OpenFlow 交換器的簡單操作方法進行說明。

流量監控 (Traffic Monitor)

本章針對「交換器（Switching Hub）」提到的 OpenFlow 交換器加入流量監控的功能。

2.1 定期檢查網路狀態

網路已經成為許多服務或業務的基礎建設，所以維護一個穩定的網路環境是必要的。但是網路問題總是不斷地發生。

網路發生異常的時候，必須快速的找到原因，並且儘速恢復原狀。這不需要多說，正在閱讀本書的人都知道，找出網路的錯誤、發現真正的原因需要清楚地知道網路的狀態。例如：假設網路中特定的連接埠正處於高流量的狀態，不論是因為他是一個不正常的狀態或是任何原因導致，變成一個由於沒有持續監控所發生的問題。

因此，為了網路的安全以及業務的正常運作，持續注意網路的健康狀況是最基本的工作。當然，網路流量的監視並不能夠保證不會發生任何問題。本章將說明如何使用 OpenFlow 來取得相關的統計資訊。

2.2 安裝 Traffic Monitor

接著說明如何在「交換器（Switching Hub）」中提到的交換器中加入流量監控的功能。

```
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.EventOFPPStateChange,
               [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)

    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.OFPPPortStatsRequest(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',
                            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)

```

事實上，流量監控功能已經被實作在 SimpleMonitor 類別中並繼承自 SimpleSwitch13，所以這邊已經沒有轉送相關的處理功能了。

2.2.1 固定週期處理

透過 Switching Hub 的平行處理，建立一個執行緒並定期的向交換器發出要求以取得統計的資料。

```
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 中實作了一些 eventlet wrapper 和基本的類別。這裡我們使用 hub.spawn() 建立執行緒。但實際上是使用 eventlet 的 green 執行緒。

```
# ...

@set_ev_cls(ofp_event.EventOFPSwitchFeatures,
            [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)

# ...
```

在執行緒中 _monitor() 方法確保了執行緒可以在每 10 秒的間隔中，不斷地向註冊的交換器發送要求以取得統計資訊。

為了確認連線中的交換器都可以被持續監控，EventOFPSwitchFeatures 就可以用來監測交換器的連線中斷。這個事件偵測是 Ryu 框架所提供的功能，會被觸發在 Datapath 的狀態改變時。

當 Datapath 的狀態變成 MAIN_DISPATCHER 時，代表交換器已經註冊並正處於被監視的狀態。而狀態變成 DEAD_DISPATCHER 時代表已經從註冊狀態解除。

```
# ...

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()` 以驅動 `OFPFlowStatsRequest` 和 `OFPPortStatsRequest` 對交換器發出訊息。

`OFPFlowStatsRequest` 主要用來對交換器的 Flow Entry 取得統計的資料。對於交換器發出的要求可以使用 table ID、output port、cookie 值和 match 條件來限縮範圍，但是這邊的例子是取得所有的 Flow Entry。

`OFPPortStatsRequest` 是用來取得關於交換器的連接埠相關資訊以及統計訊息。使用的時候可以指定連接埠號，這邊使用 `OFPP_ANY` 目的是要取得所有的連接埠統計資料。

2.2.2 FlowStats

為了接收來自交換器的回應，建立一個 event handler 來接受從交換器發送的 `FlowStatsReply` 訊息。

```

# ...

@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` 類別的屬性 `body` 是 `OFPFlowStats` 的列表，當中儲存了每一個 Flow Entry 的統計資訊，並做為 `FlowStatsRequest` 的回應。

權限為零的 Table-miss Flow 除外的全部 Flow Entry 將會被選擇，通過並符合該 Flow Entry 的封包數和位元數統計資料將會被回傳，並以接收埠號和目的 MAC 位址的方式排序。

為了持續的收集以及分析，僅有一部份的資料會被輸出到 log。因此若要進行分析，連結到外部的程式是必須的。在這樣的情況下 `OPFFlowStatsReply` 的內容可以被轉換成為 JSON 的格式進行輸出。

可以設定如下格式

```

import json

# ...

```

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

上述的設定將會產生結果如下

```
{
    "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,
                        "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
                    }
                ]
            }
        ]
    }
}
```

```

        "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

為了接收交換器所回覆的訊息，PortStatsReply 訊息接收的事件接收處理必須要被實作。

```

# ...

@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)

```

OFPPortStatsReply 類別的屬性 body 會列出在 OFPPortStats 中的資料列表。

OFPPortStats 連接埠號儲存接收端的封包數量、位元數量、丟棄封包數量、錯誤數量、frame 錯誤數量、overrun 數量、CRC 錯誤數量、collection 數量等等的統計資訊。

依據連接埠號的排序列出接收的封包數量、接收位元數量、接收錯誤數量、發送封包數量、發送位元數、發送錯誤數量。

2.3 執行 Traffic Monitor

接下來實際的執行流量監控。

首先，跟「[交換器（Switching Hub）](#)」一樣的執行 Mininet。這邊別忘了交換器的 OpenFlow 版本設定為 OpenFlow13。

下一步，執行流量監控程式。

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 EventOFPPacketIn
    CONSUMES EventOFPSwitchFeatures
    BRICK ofp_event
        PROVIDES EventOFPStateChange TO {'SimpleMonitor': set(['main', 'dead'])}
        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	
<hr/>						
EVENT ofp_event->SimpleMonitor EventOFPPortStatsReply						
datapath	port	rx-pkts	rx-bytes	rx-error	tx-pkts	tx-bytes
0000000000000001	1	0	0	0	0	0
0000000000000001	2	0	0	0	0	0
0000000000000001	3	0	0	0	0	0
0000000000000001	ffffffe	0	0	0	0	0

在「交換器（Switching Hub）」中，我們使用 ryu-manager 指令來設定 SimpleSwitch13 模組名稱（ryu.app.simple_switch_13）。至於這邊則自定 SimpleMonitor 的檔案名稱（./simple_monitor.py）。

在這個時候 Flow Entry 是空白的（Table-miss Flow Entry 沒有被顯示出來），每個連接埠的計數器也都為零。

從 host 1 向 host 2 執行 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:~#
```

封包的轉送、Flow Entry 的註冊狀況，統計資料開始有了變化。

controller: c0:

datapath			in-port	eth-dst	out-port			packets	bytes
datapath	port	rx-pkts	rx-bytes	rx-error	tx-pkts	tx-bytes	tx-error		
<hr/>									
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	ffffffe	0	0	0	1	42	0		

Flow Entry 的統計資訊中，在接收埠 1 的 Flow match 流量訊息中，1 個封包，42 個位元組的資訊被記錄下來。接收埠 2 則是 2 個封包，140 個位元組。

連接埠的統計資訊，連接埠 1 的封包接收（rx-pkts）數量為 3，接收位元組（rx-bytes）數量為 102 bytes，連接埠 2 也是 3 個封包，182 位元組。

Flow Entry 的統計資訊和連接埠的統計資訊是不可以混為一談的，這是因為 Flow Entry 的統計資訊是記錄 match 的 Entry 所轉送封包的統計資料。也就是說被 Table-miss 觸發的 Packet-In 以及 Packet-Out 轉送封包都不能算在統計資料內。

在這個案例中，host 1 最初的廣播訊息是 ARP request，而 host 2 回覆了 host 1 的 ARP 訊息，host 1 對 host 2 發送了 echo request 總共 3 個封包，這些都是透過觸發 Packet-Out 所傳送的。因此連接埠的流量會遠大於 Flow Entry 的流量。

2.4 本章總結

本章藉由取得統計資料做為題目，嘗試說明下列事項。

- 產生 Ryu 應用程式執行緒的方法
- 取得 Datapath 狀態的改變
- 取得 FlowStats 和 PortStats 資訊的方法

REST API

本章說明如何新增 REST 於「交換器（Switching Hub）」提到的 switching hub 中。

3.1 整合 REST API

Ryu 本身就有提供 WSGI 對應的 Web 伺服器。透過這個機制建立相關的 REST API 可以與其他系統或瀏覽器整合。

註解：WSGI 是 Python 提供用來連結網頁應用程式和網頁伺服器的框架。

3.2 安裝包含 REST API 的 Switching Hub

接下來讓我們實際加入兩個先前在「交換器（Switching Hub）」說明過的 API。

1. MAC 位址表取得 API

取得 Switching hub 中儲存的 MAC 位址表內容。成對的 MAC 位址和連接埠號將以 JSON 的資料形態回傳。

2. MAC 位址表註冊 API

MAC 位址和連接埠號成對的新增進 MAC 位址表，同時加到交換器的 Flow Entry 中。

接下來我們來看看程式碼。

```
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.simple_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.simple_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.simple_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 是用來定義兩個類別。

前者是控制器類別 SimpleSwitchController，其中定義收到 HTTP request 時所需要回應的相對方法。

後者是 ``SimpleSwitchRest13`` 的類別，用來擴充「交換器（Switching Hub）」讓它得以更新 MAC 位址表。

由於在 SimpleSwitchRest13 中已經有加入 Flow Entry 的功能，因此 FeaturesReply 方法被覆寫 (overridden) 並保留 datapath 物件。

3.3 安裝 SimpleSwitchRest13 class

```

class SimpleSwitchRest13(simple_switch_13.SimpleSwitch13):

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

```

類別變數 _CONTEXT 是用來製定 Ryu 中所支援的 WSGI 網頁伺服器所對應的類別。因此我們可以透過 wsgi Key 來取得 WSGI 網頁伺服器的實體。

```

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

```

建構子需要取得 WSGIApplication 的實體用來註冊 Controller 類別。稍後會有更詳細的說明。註冊則可以使用 register 方法。在呼叫 register 方法的時候，dictionary object 會在名為 simple_switch_api_app 的 Key 中被傳遞。因此 Controller 的建構子就可以存取到 simple_switch_api_app 的實體。

```

@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, {})
...

```

父類別 `switch_features_handler` 已經被覆寫 (overridden)。這個方法會在 `SwitchFeatures` 事件發生時被觸發，從事件物件 `ev` 取得 `datapath` 物件後存放至 ``switches`` 變數中。此時 MAC 位址的初始值將會設定為空白字典 (empty dictionary) 形態。

```
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
...
```

本方法用來註冊 MAC 位址和連接埠號至指定的交換器。當 REST API 的 PUT 方法被觸發時，本方法就會被執行。

參數 `entry` 則是用來儲存已經註冊的 MAC 位址和連結埠的資訊。

參照 MAC 位址表的 `self.mac_to_port` 資訊，被註冊到交換器的 Flow Entry 將被搜尋。

例如：一個成對的 MAC 位址和連接埠號將被登錄在 MAC 位址表中。

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

而且成對的 MAC 位址和連接埠號將被當作參數 `entry`。

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

最後被加入交換器當中的 Flow Entry 將會如下所示。

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

Flow Entry 的加入是透過父類別的 `add_flow` 方法達成。最後經由參數 `entry` 傳遞的訊息將會被儲存在 MAC 位址表。

3.4 安裝 SimpleSwitchController Class

接下來是控制器類別(controller class)中 REST API 的 HTTP request。類別名稱是 ``SimpleSwitchController``。

```

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

```

從建構子 (constructor) 中取得 SimpleSwitchRest13 的實體。

```

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

    simple_switch = self.simple_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)
...

```

這部分是用來實作 REST API 的 URL，還有其相對定的處理動作。為了結合 URL 和其對應的方法，route 這個裝飾器將在 Ryu 中被使用。

被裝飾器 (Decorator) 處理的內容說明如下。

- 第一個參數
任意的名稱
- 第二個參數
指定 URL。使得 URL 為 `http://<伺服器 IP>:8080/simpleswitch/mactable/<datapath ID>`
- 第三參數
指定 HTTP 相對應的方法。指定 GET 相對應的方法。
- 第四參數
指定 URL 的形式。URL/`/simpleswitch/mactable/{dpid}` 中 {dpid} 的部分必須與 ryu/lib/dpid.py 中 DPID_PATTERN 16 個 16 進位的數字定義相吻合。

當 REST API 被第二參數所指定的 URL 呼叫時，相對的 HTTP GET list_mac_table 方法就會被觸發。該方法將會取得 {dpid} 中儲存的 data path ID 以得到 MAC 位址並轉換成 JSON 的格式進行回傳。

如果連結到 Ryu 的未知交換器 data path ID 被指定時，Ryu 會返回編碼為 404 的回應。

```

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

    simple_switch = self.simple_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)
...
```

然後是註冊 MAC 位址表的 REST API。

URL 跟取得 MAC 位址表時的 API 相同，但是 HTTP 在 PUT 的情況下會呼叫 put_mac_table 方法。這個方法的內部會呼叫 switching hub 實體的 set_mac_to_port 方法。

當 put_mac_table 方法產生的例外的時候，回應碼 500 將會被回傳。同樣的，list_mac_table 方法在 Ryu 所連接的交換器使用未知的 data path ID 的話，會回傳回應碼 404。

3.5 執行包含 REST API 的 Switching Hub

接著讓我們執行已經加入 REST API 的 switching hub 吧。

首先執行「交換器（Switching Hub）」和 Mininet。這邊不要忘了設定交換器的 OpenFlow 版本為 OpenFlow13。接著啟動已經加入 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)
move onto main mode
```

上述啟動時的訊息中有一行「(31135) wsgi starting up on <http://0.0.0.0:8080/>」，這是表示網頁伺服器和埠號 8080 已經被啟動。

接下來是在 mininet 的 shell 上從 h1 對 h2 執行 ping 的動作。

```
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
```

這時後會發生三次往 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
```

再來執行 REST API 以便在 switching hub 中取得 MAC 位址表。這次我們使用 curl 指令來驅動 REST API 。

```
ryu@ryu-vm:~$ curl -X GET http://127.0.0.1:8080/simpleswitch/mactable
/0000000000000000
{"00:00:00:00:00:02": 2, "00:00:00:00:00:01": 1}
```

你會發現 h1 和 h2 的 MAC 位址表已經學習並更新完畢。

這次 h1 和 h2 的 MAC 位址表提前在執行 ping 之前被設定好。暫時停止 switching hub 和 mininet 的執行。然後再次啟動 mininet 並在 OpenFlow 版本設定為 OpenFlow13 之後接著啟動。

```
...
(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
```

接著在每個 host 上呼叫 MAC 位址表更新的 REST API 。REST API 呼叫的形式是 {`mac` : ``MAC 位址``, `port` : 連接的連接埠號}

```
ryu@ryu-vm:~$ curl -X PUT -d '{"mac" : "00:00:00:00:00:01", "port" : 1}' http
://127.0.0.1:8080/simpleswitch/mactable/0000000000000000
{"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/0000000000000000
{"00:00:00:00:00:02": 2, "00:00:00:00:00:01": 1}
```

執行上述的指令，h1 和 h2 對應的 Flow Entry 會被加入交換器中。

然後從 h1 對 h2 執行 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
```

這時候交換器中已經存在著 Flow Entry。Packet-In 只會發生在當 h1 到 h2 的 ARP 出現且沒有接連發生的封包交換時。

3.6 本章總結

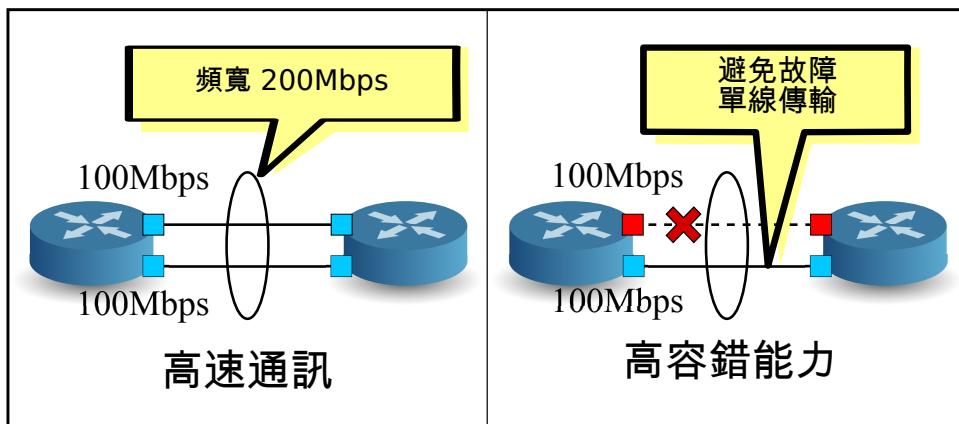
本章使用 MAC 位址表的處理作為題材，來說明如何新增 REST API。至於其他的練習應用，如果可以做個從網頁直接加入 Flow Entry 的 REST API 將會是一個很好的想法。

網路聚合 (Link Aggregation)

本章會說明，Ryu 使用的網路聚合功能的實作方法。

4.1 網路聚合 (Link Aggregation)

網路聚合 (Link Aggregation) 是由 IEEE802.1AX-2008 所制定的，多條實體線路合併為一條邏輯線路。透過本功能可以讓網路中特定的裝置間通訊速度提升、同時確保備援能力、提升容錯的功能。



在使用網路聚合功能之前，個別的網路裝置上界面歸屬於特定群組的關係都必須先設定完成。

起始網路聚合功能的方法是將個別的網路裝置設置完成，此為靜態方法。另外也可以使用 LACP (Link Aggregation Control Protocol) 通訊協定，此為動態方法。

採用動態方法的時候，每一個網路裝置所相對應的界面會定期的進行 LACP data unit 交換以確認彼此之間的通訊狀況。當 LACP data unit 的交換無法完成，代表網路已經出現故障，使用該網路的裝置出現通訊中斷，此時封包的傳送僅能使用殘存的界面和線路完成。

這樣的做法有個優點，即當有個轉送裝置存在於網路之中，例如 media converter，當該裝置的另外一端也斷線時可以被偵測到。本章將會說明使用 LACP 進行動態網路聚合的設置。

4.2 執行 Ryu 應用程式

原始碼的解說將放到後面，首先是執行 Ryu 的網路聚合應用程式。

simple_switch_lacp.py 為 OpenFlow 1.0 專用的應用程式並存在于 Ryu 的原始碼中。在這邊我們要建立新的 OpenFlow 1.3 版本，即 simple_switch_lacp_13.py。此應用程式可以為「交換器（Switching Hub）」中的交換器新增網路聚合功能。

原始碼名稱：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.OFFInstructionActions(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=oproto.OFPFC_DELETE,
        out_port=oproto.OFPP_ANY,
        out_group=oproto.OFGG_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(ether.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, {})

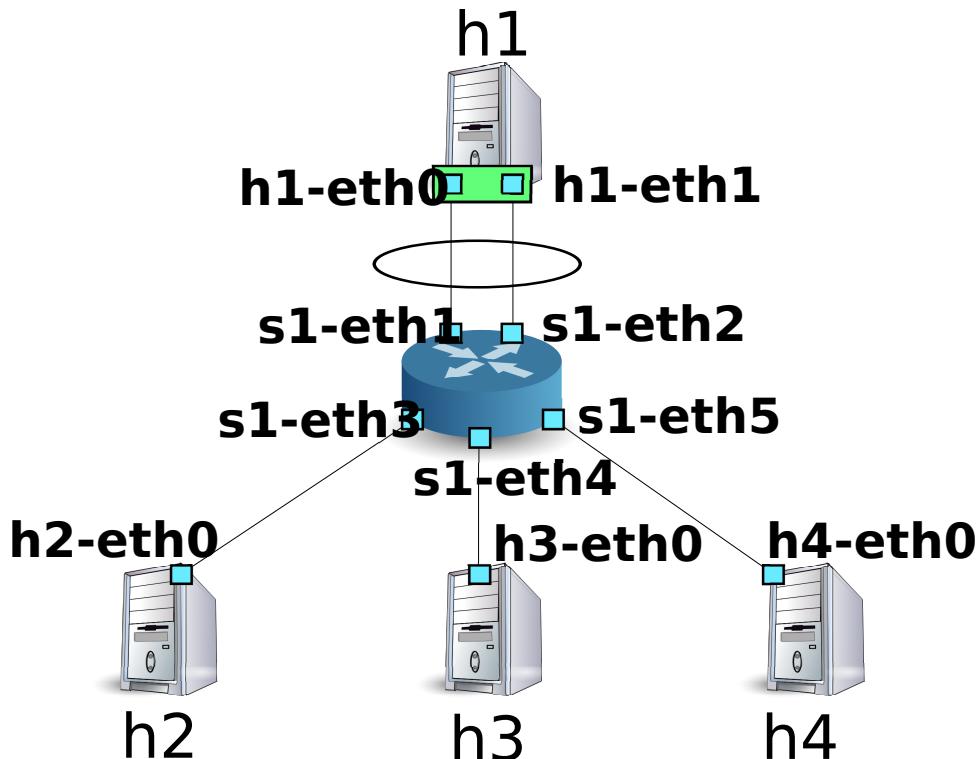
```

4.2.1 建置實驗環境

讓我們來看一下 OpenFlow 交換器和 Linux host 之間的網路聚合。

為了使用 VM 映像檔，詳細的環境設定和登入方法等請參考「[交換器（Switching Hub）](#)」。

首先使用 Mininet 製作出如下一般的網路拓墣。



使用 script 來呼叫 Mininet 的 API 進而完成網路拓墣的建構。

原始碼名稱：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()

```

執行該 script 之後會形成 host 1 和交換器 s1 之間有兩條連線的拓墣結構。這時後可以使用 net 命令來進行確認。

```

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 的網路聚合設定

在這之前在 host h1 的 Linux 作業系統中必須先行設定。

請輸入本節的命令在 host h1 的 xterm 終端機之中。

首先載入 drive module 以完成網路聚合。在 Linux 之中，網路聚合功能是由 bonding drive 所處理。預先建立 drive 的設定檔 /etc/modprobe.d/bonding.conf 以完成該功能。

檔案名稱: /etc/modprobe.d/bonding.conf

```

alias bond0 bonding
options bonding mode=4

```

Node: h1:

```

root@ryu-vm:~# modprobe bonding

```

mode = 4 是 LACP 中代表使用動態網路聚合 (dynamic link aggregation)，由於是預設值的關係這邊可以省略。而 LACP data unit 的交換間隔為 SLOW (30 秒)，並且排序的方式是使用目的 MAC 位址來進行。

接著建立一個名為 bond0 的邏輯界面，然後設定 bond0 的 MAC 位址。

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 的實體網路界面加到已經建立好的邏輯界面群組中。此時先將實體界面設定為 down，然後亂數決定該實體界面成為比較簡單的 MAC 位址之後更新它。

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
```

指定邏輯界面的 IP 位址，這邊指定為 10.0.0.1。由於 h1-eth0 的 IP 位址已經被自動指定所以我們刪除它。

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
```

最後設定邏輯界面為 UP。

Node: h1:

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

接著確認每一個界面的狀態。

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)
```

邏輯界面 bond0 為 MASTER，實體界面 h1-eth0 和 h1-eth1 為 SLAVE。而且你可以看到 bond0、h1-eth0 和 h1-eth1 的 MAC 位址全部都是相同的。

確認 bonding driver 的狀態。

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

802.3ad 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
```

確認 LACP data unit 的交換間隔 (LACP rate: slow) 和排序邏輯的設定 (Transmit Hash Policy: layer2 (0))。並且確認實體界面 h1-eth0 和 h1-eth1 的 MAC 位址。

以上為 host h1 的事前準備。

4.2.3 設定 OpenFlow 版本

交換器 s1 的 OpenFlow 版本設定為 1.3。請在交換器 s1 的 xterm 終端機上輸入下面的指令。

Node: s1:

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

4.2.4 執行交換器

準備完成，接著開始執行 Ryu 應用程式。

在視窗標題為「Node: c0 (root)」的 xterm 終端機中執行下列的命令。

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 會每 30 秒發送一次 LACP data unit。啟動之後，交換器馬上就會收到 host h1 發送的 LACP data unit，並輸出記錄在 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
...
```

下面說明 log 的內容。

- LACP received.

 接受到 LACP data unit。

- the slave i/f has just been up.

 原本無效的連接埠轉換成有效狀態。

- the timeout time has changed.

 LACP data unit 的逾時時間變更（本例子中原始狀態為 0 秒，變更為 LONG_TIMEOUT_TIME 的 90 秒）

- LACP sent.

 回覆用的 LACP data unit 傳送完畢

- slave state changed ...

 應用程式接收到 LACP 函式庫中 EventSlaveStateChanged 事件訊息（事件的詳細內容稍後說明）。

交換器對於每一次從 host h1 收到的 LACP data unit 傳送回覆用的 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.
...
```

確認 Flow Entry。

Node: s1:

```
root@ryu-vm:~# ovs-ofctl -0 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
```

在交換器中

- 接收到從 h1 的 h1-eth1 (接收埠為 s1-eth2、目的 MAC 位址為 00:00:00:00:00:12) 傳送 LACP data unit (ether type 0x8809) 時就發送 Packet-In 訊息。
- 接收到從 h1 的 h1-eth0 (接收埠為 s1-eth1、目的 MAC 位址為 00:00:00:00:00:11) 傳送 LACP data unit (ether type 0x8809) 時就發送 Packet-In 訊息。
- Table-miss Flow Entry 跟「[交換器 \(Switching Hub\)](#)」相同。

以上 3 個 Flow Entry 被新增到交換器之中。

4.2.5 確認網路聚合功能

改善傳送速度

首先是確認網路聚合的傳送速度。讓我們來看看使用不同的線路來進行通訊的狀況。

首先，從 host h2 向 host h1 發送 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
...
```

當 ping 訊息持續發送的同時，確認交換器 s1 的 Flow Entry。

Node: s1:

```
root@ryu-vm:~# ovs-ofctl -0 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
```

在確認時會新增下面兩個 Flow Entry。第四和第五 Flow Entry 會有較小的 duration 的值。

分別是

- 若是收到來自第 3 連接埠 (s1-eth3，也就是連接到 h2 的界面) 向 h1 的 bond0 發送的封包就轉送到第 1 連接埠 (s1-eth1)。
- 若是收到來自第 1 連接埠 (s1-eth1) 向第 2 連接埠發送的封包，就從第 3 連接埠 (s1-eth3) 轉送出去。

接著你可以看到 h2 和 h1 之間的通訊是使用 s1-eth1 來完成。

接著從 host h3 向 host h1 發送 ping 訊息。

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 持續發送時，確認交換器 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
```

在剛才確認的時候，2 個 Flow Entry 已經被新增。duration 值較小的 Flow Entry 為第 5 和第 6 個 Entry。

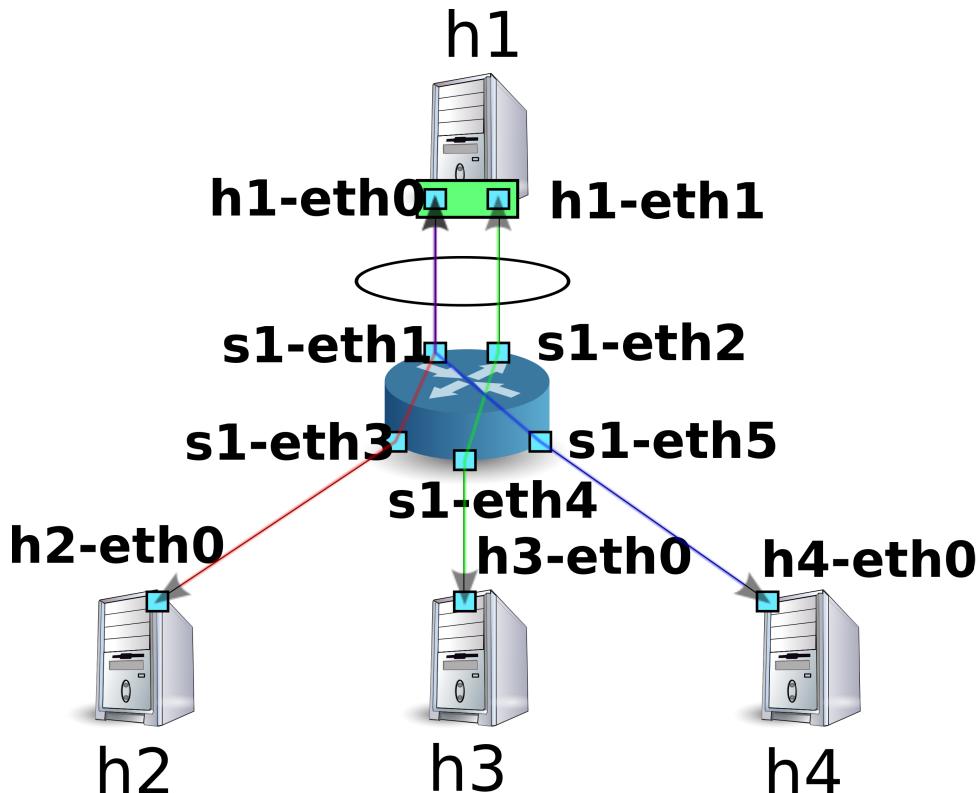
分別是

- 收到來自第 2 連接埠 (s1-eth2) 向 h3 發送的封包時就從第 4 連接埠 (s1-eth4) 轉送出去
- 收到來自第 4 連接埠 (s1-eth4，也就是連接到 h3 的界面) 向 h1 的 bond0 發送的封包時就從第 2 連接埠 (s1-eth2) 轉送出去

這樣可以看出 h3 和 h1 之間的通訊是使用 s1-eth2 完成。

當然 host 4 向 host h1 發送的 ping 指令就可以正常動作。到此為止同樣新的 Flow Entry 會被新增，h4 和 h1 之間的通訊就透過 s1-eth1 來傳送。

目的 host	使用連接埠
h2	1
h3	2
h4	1



經過以上的動作，可以確認多條線路的通訊狀態。

提升容錯能力

接下來，我們要提升網路聚合的容錯能力。現在的狀況是 h2、h4 和 h1 之間的通訊是使用 s1-eth2 連接埠，h3 和 h1 的通訊是使用 s1-eth1 連接埠。

在這邊我們把 s1-eth1 從所屬的 s1-eth0 網路聚合群組中分離出來。

Node: h1:

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

當 h1-eth0 停用時，host h3 向 host h1 的 ping 是無法連通的。在停止通訊的狀態下經過了 90 秒之後，下面的 log 上會出現 Controller 的動作訊息。

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=00000000000000000000000000000001 PORT=2 LACP received.  
[LACP] [INFO] SW=00000000000000000000000000000001 PORT=2 LACP sent.  
[LACP] [INFO] SW=00000000000000000000000000000001 PORT=2 LACP received.  
[LACP] [INFO] SW=00000000000000000000000000000001 PORT=2 LACP sent.  
[LACP] [INFO] SW=00000000000000000000000000000001 PORT=1 LACP exchange timeout has occurred.  
slave state changed port: 1 enabled: False  
...
```

「LACP exchange timeout has occurred.」表示停止通訊已經逾時，曾經學習過的 MAC 位址和轉送封包用的 Flow Entry 將全部被刪除，回到如同剛開機時的初始狀態。

當新的通訊發生時，新的 MAC 位址將會被學習，已存在的線路連結 Flow Entry 將會再次被新增。

host h3 和 host h1 之間的新 Flow Entry 會被新增。

Node: s1:

```
root@ryu-vm:~# ovs-ofctl -0 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 恢復正常。

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  
...
```

經過以上的說明，當一部份的線路發生故障的時候，我們可以確認到其他的線路會自動的恢復以進行通訊。

4.3 實作 Ryu 的網路聚合功能

讓我們來看看如何利用 OpenFlow 實作網路聚合功能。

使用 LACP 的 link aggregation 的行為會像是「當 LACP data unit 傳遞正常時，則表示所使用的實體界面為有效狀態」「反之，當 LACP data unit 傳遞不正常時，則表示所使用的實體界面為停用或無效狀態」實體界面無效的意思是，該界面並無相關聯的 Flow Entry 存在。因此透過下面的步驟：

- 實作接收並回覆 LACP data unit
- 在一個固定的時間區間中沒有收到 LACP data unit 時，則刪除該實體界面所使用到及相關聯的 Flow Entry
- 已經無效的實體界面若是收到了 LACP data unit 封包，則設定該界面為有效。
- LACP data unit 以外的封包則比照「[交換器（Switching Hub）](#)」進行學習及轉送。

經過以上的處理及安裝後，網路聚合基本的動作已經完成。LACP 相關的部分和不相關的部分已經可以被明顯的區隔出來，對於不相關的部分請參考「[交換器（Switching Hub）](#)」以進行交換器的擴充實作。

在接收到 LACP data unit 時給予回覆這件事情無法單純使用 Flow Entry 來實現。因此 Packet-In 訊息被用來支援處理 OpenFlow Controller 方面的溝通。

註解：用來交換 LACP data unit 的實體界面被分為兩種角色，主動（ACTIVE）與被動（PASSIVE）形態。主動：在固定的時間會發送 LACP data unit，以確認線路的狀態。被動：在收到主動狀態發送過來的 LACP data unit 時予以回覆，以確認線路的狀態。

在 Ryu 中的網路聚合應用程式是使用被動模式（PASSIVE mode）來實作。

若是在特定的時間內沒有收到 LACP data unit，則該實體界面視為無效。會這樣是因為先前 LACP data unit 促使 Packet-In 並進行 Flow Entry 新增時設定了 idle_timeout。當時間超過該設定值時，發送 FlowRemoved 訊息通知 Controller。Controller 則讓該界面無效。

已經處於無效狀態的界面收到 LACP data unit 的時候的處理為：接收到 LACP data unit 的時候 Packet-In 訊息的封包會做為該界面的有效或無效的判斷標準，並進行必要的變更。

若實體界面為無效時，OpenFlow Controller 最簡單的處理為「刪除該界面所使用到的 Flow Entry」，但是這樣的條件其實是不充分的。

例如：有一個邏輯界面是由三個實體界面群組化之後所產生，其排序為「依照使用界面的數量排序 MAC 位址」。

界面 1	界面 2	界面 3
剩餘 MAC 位址:0	剩餘 MAC 位址:1	剩餘 MAC 位址:2

然後每一個實體界面所使用的 Flow Entry 分別新增如下。

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 為停用的狀態下，「在有效的界面中以剩餘 MAC 位址數量為準」作為排序的標準，排序結果如下。

界面 1	界面 2	界面 3
無效	剩餘 MAC 位址:0	剩餘 MAC 位址: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，沒有僅使用單一界面 1 的 Flow Entry，在 Flow Entry 修改的同時，也會對界面 2 和界面 3 進行修改。這在實體界面變更為無效的時候需要做，變更為有效的時候同樣也需要。

因此當實體界面在無效與有效之間改變時，該實體界面所屬的邏輯界面所引用到的 Flow Entry 也會一併被進行刪除。

註解：排序的邏輯方法並沒有在規格書中定義，由各家廠商自行實作。Ryu 的網路聚合應用程式是自行處理排序的部分，同時也使用相對應的機器所提供的路徑排序。

接下來實作下面的功能。

LACP 函式庫

- 當接收到 LACP data unit 訊息後，製作回覆訊息並傳送
- 當 LACP data unit 的接收中斷後，相對應的實體界面則變更為無效，並通知交換器
- 當再次接收到 LACP data unit 時，相對應的實體界面則變更為有效，並通知交換器

交換器

- 接收到 LACP 函式庫的通知之後，進行初始化以及刪除必要的 Flow Entry
- 接收到 LACP data unit 以外的封包時，像平常一樣學習並進行封包轉送

LACP 函式庫和其所使用的交換器原始碼，都在 Ryu 的原始碼中提供。

ryu/lib/lacplib.py

ryu/app/simple_switch_lacp.py

註解：因為 simple_switch_lacp.py 為 OpenFlow 1.0 所專用的應用程式，本章所詳細說明的是「執行 Ryu 應用程式」中 simple_switch_lacp_13.py 所對應的 OpenFlow 1.3 版本應用程式。

4.3.1 實作 LACP 函式庫

在接下來的章節中，我們來看一下前述的 LACP 函式庫功能如何實作。說明中所引用的原始碼為部分截取，若要了解全體的樣貌，請參照實際的原始碼。

製作邏輯界面

若要使用網路聚合功能，必須要事先設定好哪一個網路是歸屬於哪一個群組。LACP 函式庫使用下列的方法來進行設定。

```
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)

```

參數的內容說明如下。

dpid

指定 OpenFlow 交換器的 data path ID。

ports

指定要被群組的連接埠編號。

透過呼叫這個方法，LACP 函式庫可以指定特定 data path ID 所屬 OpenFlow 交換器連接埠成為特定的群組。若是要組成多個群組只要重複的呼叫 add() 方法即可。當邏輯界面被指定為特定的 MAC 位址時，OpenFlow 交換器所管理相同 MAC 位址的 LOCAL 連接埠會自動的被使用。

小技巧: 有些 OpenFlow 交換器之中，交換器本身就有提供網路聚合的功能（Open vSwitch...等）在這邊我們並不使用交換器本身的功能，而是使用 OpenFlow Controller 來實現網路聚合的功能。

處理 Packet-In

在「[交換器（Switching Hub）](#)」中，目的 MAC 位址在尚未被學習的狀況下，會將接收到的封包進行 Flooding。由於 LACP data unit 僅會在相鄰的網路間進行交換，若是將該封包轉送至其他的網路時網路聚合將會出現異常。所以 Controller 就進行這樣的處理「若是接收到 Packet-In 來自 LACP data unit 的封包就阻止，LACP data unit 以外的封包就交給交換器繼續後續的動作」，在這樣的操作下 LACP data unit 就會不會出現在交換器上。

```

@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（Event handler）跟在「[交換器（Switching Hub）](#)」是相同的。會根據所收到的訊息是否為 LACP data unit 做為接下來處理的準則。

若是包含 LACP data unit 的狀況下，就執行 LACP 函式庫中 LACP data unit 的處理機制。在不包含 LACP data unit 的狀況下，則呼叫 send_event_to_observers() 方法。這是定義在 ryu.base.app_manager.RyuApp 類別中，為了發送事件訊息的方法。

在「[交換器（Switching Hub）](#)」當中，我們提到了可以被使用者自行定義由 Ryu 所提供的 OpenFlow 訊息接收事件。上述的原始碼當中有個事件叫做 EventPacketIn 即是由 LACP 函式庫提供來讓使用者自行定義。

```

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

```

使用者定義的事件是繼承自 `ryu.controller.event.EventBase` 類別來達成。該類別對於資料的封裝並沒有限制。在 ``EventPacketIn`` 類別中，若是收到了 `Packet-In` 訊息就會用 `ryu.ofproto.OFPPacketIn` 實體來像往常一樣處理。

使用者定義的事件接收方法將在之後提到。

連接埠有效/無效狀態的處理

LACP 函式庫的 LACP data unit 處理如下：

1. 若接收到 LACP data unit 時，連接埠處於無效狀態，這時候會進行狀態的變更，並發送事件通知。
2. 若停止通訊的逾時設定值已經被改變時，收到 LACP data unit 封包後就發送 `Packet-In` 的 Flow Entry 將會再次被新增。
3. 接收 LACP data unit 後的回應、回覆及傳送。

第二點的處理會在稍後的「新增 Flow Entry - 收到 LACP data unit 時發送 Packet-In 」描述，第三點的處理會在稍後的「傳送/接收 LACP data unit 的處理 」說明。接下來說明第一點的處理流程。

```
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()` 方法是用來取得指定的交換器和指定的連接埠狀態為有效或無效。
`_set_slave_enabled()` 方法是用來設定指定的交換器和指定的連接埠狀態為有效或無效。

在上述的原始碼中，當無效狀態的連接埠接收到 LACP data unit 時，連接埠的狀態會進行改變並且呼叫 `EventSlaveStateChanged` 發送訊息，意即表示連接埠狀態已經改變。

```
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，當無效發生時 `EventSlaveStateChanged` 事件也會被觸發並發送。當連接埠無效時所需執行的動作在「接收 `FlowRemoved` 訊息時的處理 」中實作。

`EventSlaveStateChanged` 類別包含以下的資訊

- 連接埠的發生狀態變更時所在的 OpenFlow 交換器
- 有效/無效的狀態改變時的連接埠編號
- 改變後的狀態

新增 Flow Entry - 收到 LACP data unit 時發送 Packet-In

LACP data unit 的傳送間隔定義為 FAST (每隔 1 秒) 和 SLOW (每隔 30 秒) 2 種。在網路聚合的規格中，傳送間隔的 3 倍時間若是無任何通訊發生時，則該界面將自該群組移除，不再使用于封包的傳送。

LACP 函式庫中透過設定 Flow Entry 來達到監視通訊的狀態，即當接受到 LACP data unit 時觸發 Packet-In 並進行 Flow Entry 設定，即為 3 倍傳送間隔 (SHORT_TIMEOUT_TIME 3 秒、LONG_TIMEOUT_TIME 90 秒) 的 idle_timeout。

當傳送間隔的設定值改變時，就必須要重新設定 idle_timeout 的時間，因此實作 LACP 函式庫如下。

```
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()` 方法可以用來取得指定的交換器之指定連接埠的 `idle_timeout` 數值。
`_set_slave_timeout()` 方法可以用來設定指定的交換器之指定連接埠的 `idle_timeout` 數值。

由於初始狀態和移除自網路聚合群組的情況下 `idle_timeout` 為 0，因此接收到新的 LACP data unit 時，傳送間隔將會根據設定被新增到 Flow Entry 中。

依據所使用的 OpenFlow 版本不同 OFPFlowMod 類別建構子的參數也不盡相同，因此必須取得所對應版本 Flow Entry 的方法。以下是採用 OpenFlow 1.2 之後所使用的 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.OFPIInstructionActions(
        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)
```

上述的原始碼為「接收到來自連接的界面所發送的 LACP data unit 時發送 Packet-In」所設定的 Flow Entry，用於在最高的優先權情況下監視停止通訊的狀態。

傳送/接收 LACP data unit 的處理

接收到 LACP data unit 時，進行「連接埠有效/無效狀態的處理」或「新增 Flow Entry - 收到 LACP data unit 時發送 Packet-In」處理以及回覆用的 LACP data unit 產生及發送。

```
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)
```

上述的原始碼中，`_create_response()` 方法是用來產生回覆用的封包。其中使用 `_create_lacp()` 方法則是用來產生回覆用的 LACP data unit。已經完成的回覆用封包則根據所接收的 LACP data unit 發送 Packet-Out。

LACP data unit 中發送端 (Actor) 的資訊和接收端 (Partner) 的資訊已經被設定完成。從接收到的 LACP data unit 可以得到發送端的資訊，因此在製作回覆用的封包時，可以設定在接收端。

```
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 訊息時的處理

在指定的傳送間隔時間中沒有進行 LACP data unit 的收送時，OpenFlow 交換器會發送 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
    send a 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))
```

當接收到 FlowRemoved 方法後，就會使用 `_set_slave_enabled()` 方法設定連接埠的狀態為無效，使用 `_set_slave_timeout()` 方法設定 idle_timeout 值為 0，使用 `send_event_to_observers()` 方法發送 `EventSlaveStateChanged` 訊息。

4.3.2 應用程式的實作

現在說明「執行 Ryu 應用程式」中所提到的 OpenFlow 1.3 對應的網路聚合應用程式 (`simple_switch_lacp_13.py`) 和「交換器 (Switching Hub)」中的交換器差異點。

設定「_CONTEXTS」

繼承自 `ryu.base.app_manager.RyuApp` 的 Ryu 應用程式若要啟動其他的應用程式的話，就必須在「`_CONTEXTS`」目錄中設定。在啟動其他的應用程式時，會使用另外的執行緒。在這邊我們設定 LACP 函式庫中的 `Lacplib` 類別名稱為「`lacplib`」並放在「`_CONTEXTS`」當中。

```
from ryu.lib import lacplib

# ...

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

# ...
```

被設定在「`_CONTEXTS`」中的應用程式可以從 `__init__()` 方法中的 `kwargs` 取得實體。

```
# ...

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

# ...
```

函式庫的初始化設定

初始化設定在「_CONTEXTS」中的 LACP 函式庫。透過執行 LACP 函式庫提供的 add() 方法來完成初始設定。設定的內容如下：

名稱	參數值	說明
dpid ports	str_to_dpid('0000000000000001') [1, 2]	data path ID 群組化的連接埠列表

根據以上的設定，data path ID 為「0000000000000001」的 OpenFlow 交換器的連接埠 1 和連接埠 2 整合為一個網路聚合群組。

```
# ...

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

# ...
```

使用者自行定義事件的接收方法

在 `實作 LACP 函式庫` 我們已經說明，在 LACP 函式庫中處理發送 PacketIn 方法時，LACP data unit 不包含使用者定義的 EventPacketIn。使用者定義的事件管理是在 Ryu 當中提供 ryu.controller.handler.set_ev_cls 作為裝飾子用來裝飾事件管理。

```
@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']

    # ...
```

接著，當連接埠的狀態變更為有效或無效時，會透過 LACP 函式庫發送 EventSlaveStateChanged 事件，因此我們必須建立一個事件管理來處理。

```
@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, {})
```

在本節開始的時候就有提到，連接埠的有效/無效狀態變更時，被邏輯界面所使用的實體界面會因為封包的通過而變更狀態。為了這個原因，已經被記錄的 Flow Entry 將會被全部刪除。

```
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)
```

透過 OFPFlowMod 進行實體界面 Flow Entry 的刪除動作。

如上所述，一個擁有網路聚合功能的交換器可以透過提供網路聚合功能的函式庫和使用該函式庫的應用程式來達成。

4.4 本章總結

本章使用網路聚合函式庫做為題材說明下列的項目。

- 如何使用「_CONTEXTS」函式庫
- 使用者自行定義當事件被觸發時，事件的處理和方法

生成樹 (Spanning Tree)

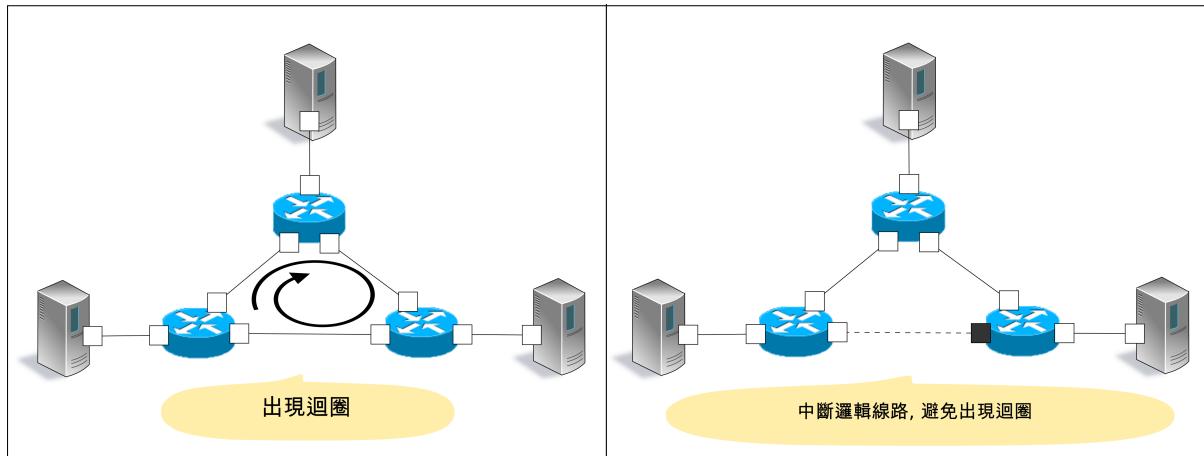
本章將說明與解說 Ryu 所採用的生成樹 (Spanning Tree) 安裝方法。

5.1 Spanning Tree

生成樹是為了防止在網路的拓墣中出現迴圈 (loop) 進而產生廣播風暴 (broadcast streams) 的技術。而且，藉著應用原本防止迴圈的這項功能，當網路發生問題的時候，則可以達到確保網路的拓墣被重新計算的目的，如此一來就不會讓部分的問題影響整個網路的連通。

生成樹有許多的種類，例如：STP、RSTP、PVST+、MSTP... 等不同的種類。本章將說明最基本的 STP。

STP (spanning tree protocol : IEEE 802.1D) 是讓網路的拓墣在邏輯上是樹狀的結構。經由設定每一個交換器 (本章節中有時候會使用橋接器稱呼) 的連接埠讓訊框 (frame) 的傳送為可通過與不可通過，來防止迴圈的產生進而達到阻止網路風暴發生。



STP 會在橋接器之間交換 BPDU (Bridge Protocol Data Unit) 封包，分析及比對橋接器之間的連接埠訊息，決定哪些連接埠可以傳送哪些不行。

具體來說，會以下列的順序完成。

1 · Root 交換器 (Root bridge) 選舉

橋接器之間的 BPDU 封包在交換過後，擁有最小的橋接器 ID 即為 Root 。接下來的 Root 橋接器會發送 original BPDU ，而其他的橋接器僅接收及轉發 BPDU 。

註解： 橋接器 ID 的計算方式是組合已經被設定的橋接器 priority 和特定埠的 MAC 位址而成。

橋接器 ID

Upper 2byte	Lower 6byte
橋接器 priority	MAC 位址

2 · 決定連接埠的角色

基於每一個連接埠到 Root 橋接器的距離來決定該連接埠的角色。

- Root port

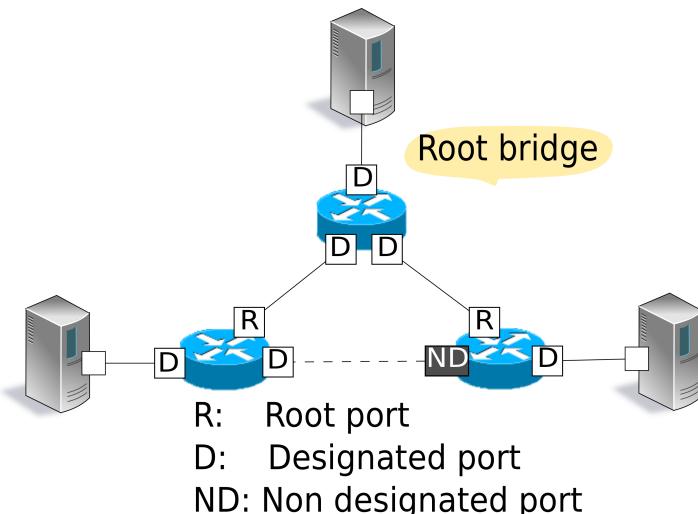
橋接器內連接到 Root 距離最短的連接埠。該連接埠將會接收來自 Root 的 BPDU。

- Designated port

該連接埠從各個連線到 Root 橋接器距離最短。主要轉送來自 Root 橋接器的 BPDU 封包，Root 橋接器的所有連接埠均為此種類。

- Non designated port

除了 Root port、designated port 以外的連接埠。訊框的傳送是被禁止的。



註解：每一個連接埠到 Root 橋接器的距離是基於收到 BPDU 中的設定值，並加上下列的比較計算出來。

第一優先：比較 root path cost 的值

各橋接器在轉送 BPDU 的時候，會將封包中 root path cost 值加上設定的 path cost。
因此 root path cost 值即是各個埠到 Root 橋接器的總和。

第二優先：root path cost 相同的話，則比較橋接器 ID

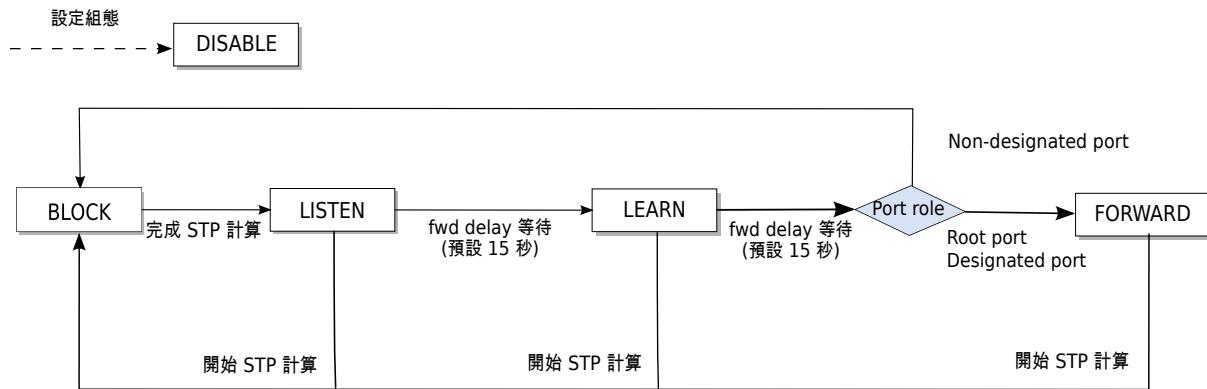
第三優先：若是橋接器 ID 相同時（每一個埠都連接到相同的橋接器），則比較連接埠 ID

連接埠 ID

upper 2byte	lower 2byte
連接埠優先權	連接埠編號

3 · 連接埠的狀態變化

連接埠的角色決定了之後（STP 的處理完成），每一個連接埠會處於 LISTEN 狀態。之後會如下圖進行狀態的轉換，最後每一個連接埠的角色會進入 FORWARD 狀態或者 BLOCK 狀態。若是設定為無效的連接埠之後，就會進入 DISABLE 狀態，接著不會進行任何狀態的轉移。



當這些程序在每一台橋接器開始執行之後，進行連接埠傳送封包或抑制封包的決定，如此一來便可以防止迴圈在網路拓墣中發生。

另外，斷線或 BPDU 封包的最大時限（預設 20 秒）內未收到封包的故障偵測、新的連接埠加入導致網路拓墣改變。這些變化都會讓每一台橋接器執行上述 1, 2 和 3 程序以建立新的樹狀拓墣 (STP re-calculation)。

5.2 執行 Ryu 應用程式

執行 Ryu 生成樹應用程式來達到 OpenFlow 實作的生成樹功能。

Ryu 的原始碼中 `simple_switch_stp.py` 是 OpenFlow 1.0 所使用，這邊我們要使用新的 OpenFlow 1.3 對應的版本 `simple_switch_stp_13.py`。這個應用程式新增加了生成樹功能到「交換器（Switching Hub）」中。

原始碼檔名：`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.OFFInstructionActions(ofproto.OFPI_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(ether.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 建置實驗環境

接下來確認生成樹應用程式的執行動作以完成環境建置。

VM 映像檔的使用、環境設定和登入方法等請參照「[交換器（Switching Hub）](#)」。

為了使用特殊含有迴圈的環境，請參考「[網路聚合（Link Aggregation）](#)」並使用 script 進行同樣的網路拓墣建置一個 mininet 的環境。

原始碼名稱：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])

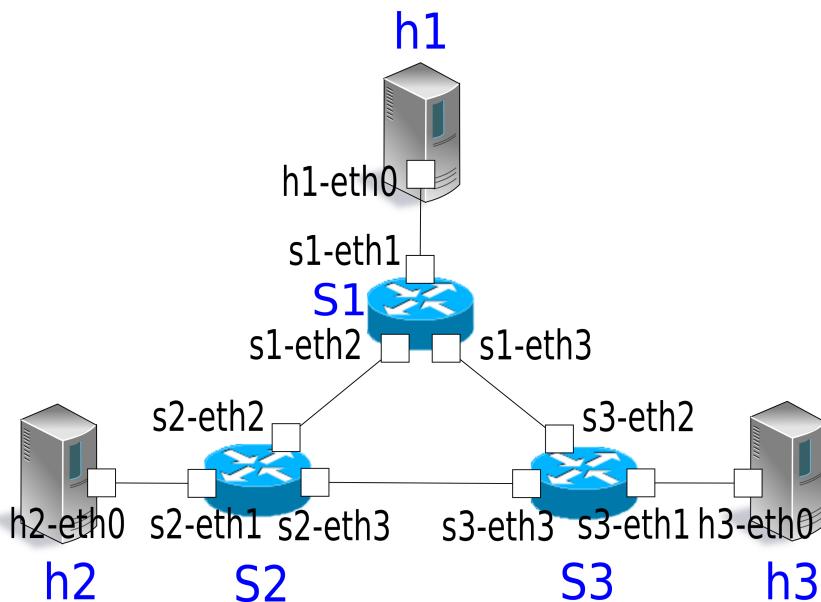
net.terms.append(makeTerm(c0))
net.terms.append(makeTerm(s1))
net.terms.append(makeTerm(s2))
net.terms.append(makeTerm(s3))
net.terms.append(makeTerm(h1))
net.terms.append(makeTerm(h2))
net.terms.append(makeTerm(h3))

CLI(net)

net.stop()

```

在 VM 的環境中執行該程式，交換器 s1 、s2 、s3 之間會出現迴圈。



`net` 命令執行結果如下。

```
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 設定 OpenFlow 版本

設定 OpenFlow 的版本為 1.3。在 `xterm` 終端機 `s1, s2, s3` 上使用命令輸入。

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 執行 switching hub

準備已經完成，接下來執行 Ryu 應用程式。在視窗標題為「Node: c0 (root)」的 `xterm` 執行下述的命令。

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 交換器啟動時的 STP 計算

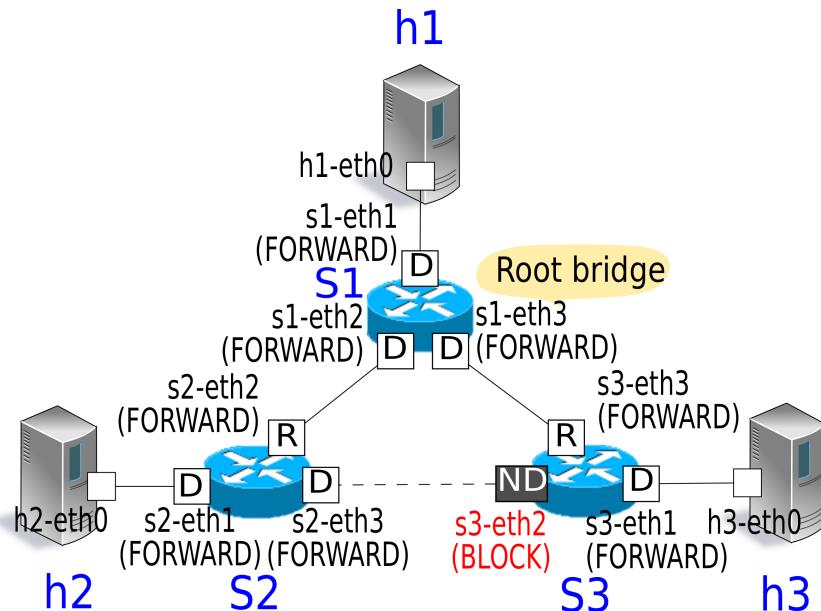
每一台 OpenFlow 交換器和 Controller 的連接完成後，BPDU 封包的交換就開始了。包括 Root 橋接器的選舉、連接埠的角色、連接埠的狀態轉移。

```
[STP] [INFO] dpid=00000000000000001: Join as stp bridge.
[STP] [INFO] dpid=00000000000000001: [port=1] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000001: [port=2] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000001: [port=3] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000002: Join as stp bridge.
[STP] [INFO] dpid=00000000000000002: [port=1] DESIGNATED_PORT      / LISTEN
```

Chapter 5. 生成樹 (Spanning Tree)

```
[STP] [INFO] dpid=00000000000000002: [port=2] ROOT_PORT / LEARN
[STP] [INFO] dpid=00000000000000002: [port=3] DESIGNATED_PORT / LEARN
[STP] [INFO] dpid=00000000000000003: [port=1] DESIGNATED_PORT / LEARN
[STP] [INFO] dpid=00000000000000003: [port=2] NON_DESIGNATED_PORT / LEARN
[STP] [INFO] dpid=00000000000000003: [port=3] ROOT_PORT / LEARN
[STP] [INFO] dpid=00000000000000001: [port=1] DESIGNATED_PORT / LEARN
[STP] [INFO] dpid=00000000000000001: [port=2] DESIGNATED_PORT / LEARN
[STP] [INFO] dpid=00000000000000001: [port=3] DESIGNATED_PORT / LEARN
[STP] [INFO] dpid=00000000000000002: [port=1] DESIGNATED_PORT / FORWARD
[STP] [INFO] dpid=00000000000000002: [port=2] ROOT_PORT / FORWARD
[STP] [INFO] dpid=00000000000000002: [port=3] DESIGNATED_PORT / FORWARD
[STP] [INFO] dpid=00000000000000003: [port=1] DESIGNATED_PORT / FORWARD
[STP] [INFO] dpid=00000000000000003: [port=2] NON_DESIGNATED_PORT / BLOCK
[STP] [INFO] dpid=00000000000000003: [port=3] ROOT_PORT / FORWARD
[STP] [INFO] dpid=00000000000000001: [port=1] DESIGNATED_PORT / FORWARD
[STP] [INFO] dpid=00000000000000001: [port=2] DESIGNATED_PORT / FORWARD
[STP] [INFO] dpid=00000000000000001: [port=3] DESIGNATED_PORT / FORWARD
```

以上的結果，最後每一個連接埠分別為 FORWARD 狀態或 BLOCK 狀態。



為了確認封包不會產生迴圈現象，從 host 1 向 host 2 發送 ping 指令。

在 ping 命令執行之前，先執行 tcpdump 命令以確認封包的接收狀況。

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 進行網路拓墣的建構的終端機中，進行接下來的指令，從 host 1 向 host 2 發送 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
```

從 tcpdump 的結果看來，ARP 並沒有出現迴圈的狀態已被確認。

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
```

網路發現故障時重新計算 STP

接下來，確認斷線發生的時候會進行 STP 的重新計算。在每一個 OpenFlow 交換器啟動之後以及 STP 的計算完成之後，執行下列指令後讓線路中斷。

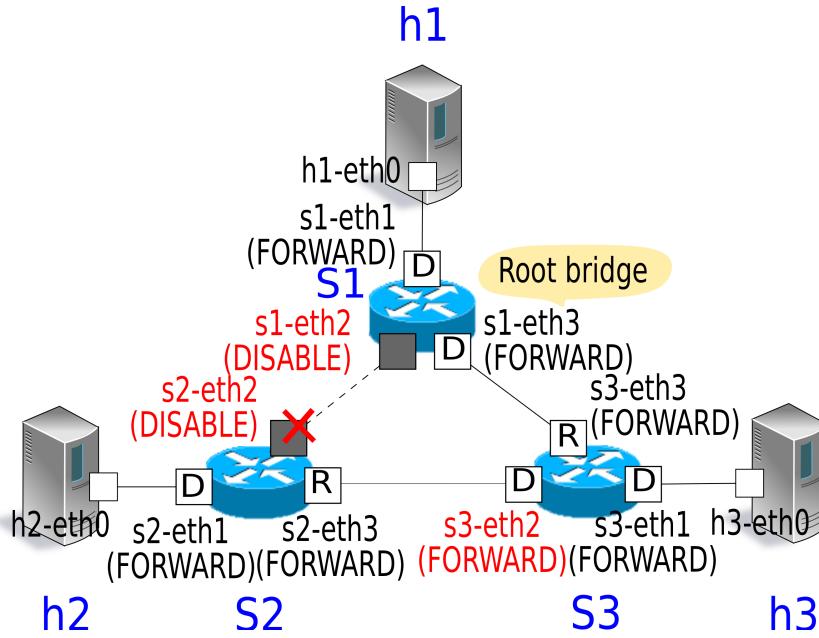
Node: s2:

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

斷線被偵測到的時候，STP 會被重新計算。

```
[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 為 BLOCK 狀態，但現在連接埠的狀態為 FORWARD，而訊框將可以再次被傳送。



從線路故障的狀態回復時重新計算 STP

接下來，斷線回復的時候 STP 將被重新計算。在斷線的狀態下執行下列的命令讓連接埠恢復。

Node: s2:

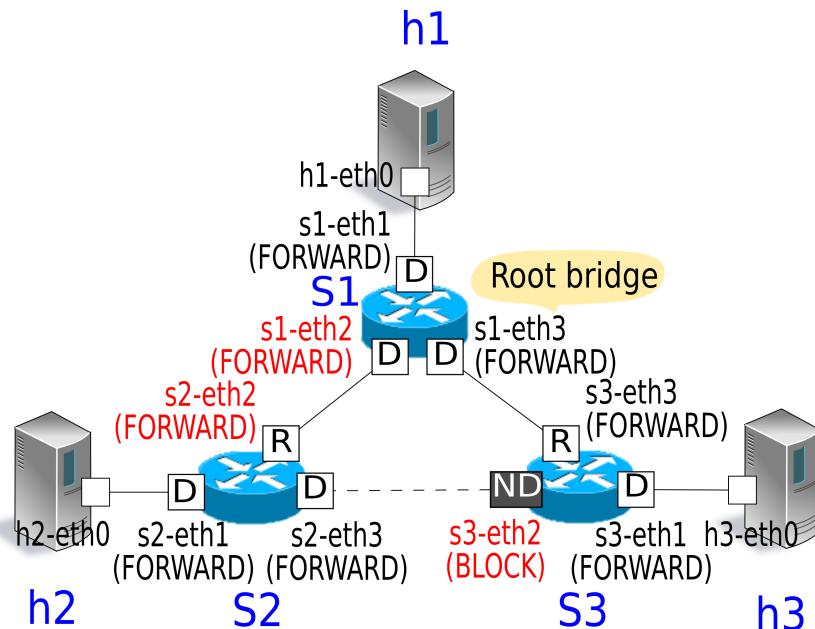
```
root@ryu-vm:~# ifconfig s2-eth2 up
```

連線恢復後被偵測到，STP 就會進行再次的計算。

```
[STP] [INFO] dpid=00000000000000002: [port=2] Link down.
[STP] [INFO] dpid=00000000000000002: [port=2] DESIGNATED_PORT      / DISABLE
[STP] [INFO] dpid=00000000000000002: [port=2] Link up.
[STP] [INFO] dpid=00000000000000002: [port=2] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000001: [port=2] Link up.
[STP] [INFO] dpid=00000000000000001: [port=2] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000001: [port=2] Receive superior BPDU.
[STP] [INFO] dpid=00000000000000001: [port=1] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000001: [port=2] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000001: [port=3] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000001: Root bridge.
[STP] [INFO] dpid=00000000000000001: [port=1] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000001: [port=2] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000001: [port=3] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000002: [port=2] Receive superior BPDU.
[STP] [INFO] dpid=00000000000000002: [port=1] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000002: [port=2] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000002: [port=3] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000002: Non root bridge.
[STP] [INFO] dpid=00000000000000002: [port=1] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000002: [port=2] ROOT_PORT          / LISTEN
[STP] [INFO] dpid=00000000000000002: [port=3] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000003: [port=2] Receive superior BPDU.
[STP] [INFO] dpid=00000000000000003: [port=1] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000003: [port=2] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000003: [port=3] DESIGNATED_PORT      / BLOCK
[STP] [INFO] dpid=00000000000000003: Non root bridge.
[STP] [INFO] dpid=00000000000000003: [port=1] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=00000000000000003: [port=2] NON_DESIGNATED_PORT / LISTEN
[STP] [INFO] dpid=00000000000000003: [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
```

可以確認目前的狀態跟應用程式啟動時有相同的樹狀結構，而訊框可以再次被傳送。



5.3 使用 OpenFlow 完成生成樹

讓我們看一下在 Ryu 生成樹應用程式中，如何使用 OpenFlow 完成生成樹的功能。

OpenFlow 1.3 提供 config 來設定連接埠的狀態。發送 Port Modification 訊息到 OpenFlow 交換器以控制連接埠對訊框的轉送行為。

名稱	說明
OFPPC_PORT_DOWN	連接埠無效狀態
OFPPC_NO_RECV	丟棄所有接收到的封包
OFPPC_NO_FWD	停止轉送封包
OFPPC_NO_PACKET_IN	table-miss 發生時，不發送 Packet-In 訊息

為了控制連接埠接收 BPDU 封包和非 BPDU 封包，收到 BPDU 封包就發送 Packet-In 的 Flow Entry 和接收 BPDU 以外的封包就丟棄的 Flow Entry，分別透過 Flow Mod 訊息新增到 OpenFlow 交換器中。

Controller 對各個 OpenFlow 交換器進行下面 port 設定和 Flow Entry 的管理，以達到控制連接埠狀態對於 BPDU 的接收傳送和 MAC 位址的學習 (BPDU 以外則是封包的接收) 和訊框的轉送 (BPDU 以外則是封包的傳送)。

名稱	設定值	Flow Entry
DISABLE	NO_RECV / NO_FWD	無
BLOCK	NO_FWD	BPDU Packet-In / BPDU 以外 drop
LISTEN	無	BPDU Packet-In / BPDU 以外 drop
LEARN	無	BPDU Packet-In / BPDU 以外 drop
FORWARD	無	BPDU Packet-In

註解：為了精簡化，Ryu 裡的生成樹函式庫並不處理 LEARN 狀態的 MAC 位址（接收 BPDU 以外的封包）學習。

為了做這些設定，Controller 產生 BPDU 封包基於 OpenFlow 交換器連接時所收集的連接埠資訊和每一個 OpenFlow 交換器所接收的 BPDU 封包中所設定的 Root 橋接器資訊，來發送 Packet-Out 訊息達到交換器之間互相交換 BPDU 的效果。

5.4 使用 Ryu 實作生成樹

接下來，檢視一下 Ryu 所用來實作生成樹的原始碼。生成樹的原始碼存放在 Ryu 的原始碼當中。

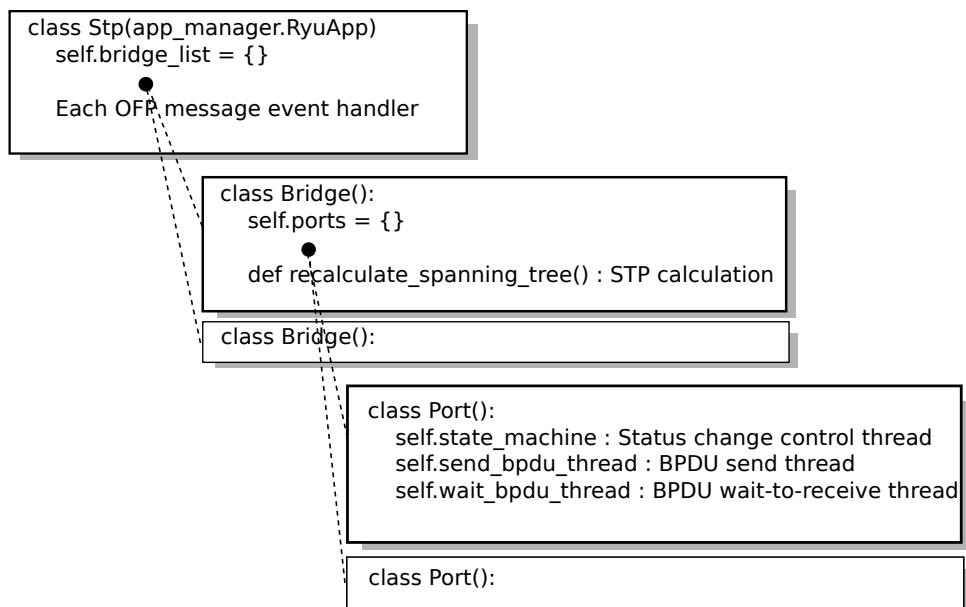
```
ryu/lib/stplib.py  
ryu/app/simple_switch_stp.py
```

stplib.py 是用來提供 BPDU 封包的交換和連接埠的角色、狀態管理的生成樹函式庫。simple_switch_stp.py 是一個應用程式，用來讓交換器的應用程式新增生成樹函式庫以增加生成樹功能使用。

注意：因為 simple_switch_stp.py 是 OpenFlow 1.0 專用的應用程式，本章為「執行 Ryu 應用程式」因此使用 OpenFlow 1.3 所對應的 simple_switch_stp_13.py 作為詳細說明的目標。

5.4.1 函式庫的安裝

函式庫概述



STP 函式庫 (STP 實體) 偵測到 OpenFlow 交換器和 Controller 已經連結時，Bridge 類別的實體和 Port 類別的實體就會被產生。

當每一個類別的實體產生、啟動之後。

- 從 STP 類別實體接收到 OpenFlow 訊息。
- Bridge 類別實體運算 STP (Root 橋接器的選擇，每一個連接埠的角色選擇)
- Port 類別實體的狀態變化，BPDU 封包接收及傳送

以上動作完成後，即可達成生成樹的功能。

設定項目

如果使用 `Stp.set_config()` 方法，則 STP 函式庫有提供橋接器連接埠的設定項目。可用的設定項目如下：

- bridge

名稱	說明	預設值
priority	橋接器優先權	0x8000
sys_ext_id	設定 VLAN-ID (* 目前的 STP 函式庫不支援 VLAN)	0
max_age	BPDU 封包的傳送接收逾時	20[sec]
hello_time	BPDU 封包的傳送接收間隔	2 [sec]
fwd_delay	每一個連接埠停留在 LISTEN 或 LEARN 的時間	15[sec]

- port

名稱	說明	預設值
priority	連接埠優先權	0x80
path_cost	Link Cost	基於連線速度自動設定
enable	連接埠的有效/無效	True

傳送 BPDU 封包

傳送 BPDU 封包的動作為 BPDU 封包傳送執行緒 (Port.send_bpdu_thread) 所執行，當連接埠的角色為 designated port (DESIGNATED_PORT) 時，定期通知 Root 橋接器的 hello time (Port.port_times.hello_time：預設 2 秒) 封包就會被產生 (Port._generate_config_bpdu()) 並進行傳送 (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)
```

即將被轉送的 BPDU 封包所設定的內容會來自 OpenFlow 交換器與 Controller 連接時所收集到的連接埠資訊 (Port.ofport) 和曾經接受到的 BPDU 封包中 Root 橋接器的資訊 (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 類別的 Packet-In 事件管理器（Event handler）所發現，經由 Bridge 類別實體通知給 Port 類別實體。事件管理器的實作請參考「[交換器（Switching Hub）](#)」。

接收到 BPDU 封包的連接埠會對先前接收到的 BPDU 封包以及本次接收到的封包中的橋接器 ID 進行比對（`Stp.compare_bpdu_info()`），來決定 STP 是否必須重新計算路徑。若是相比之前的封包之下，本次收到的封包為優先封包（superior BPDU）、（SUPERIOR）時，則代表網路的拓墣已經改變，例如「一個新的 Root 橋接器已經被加入網路」，此時則必須開始進行 STP 的重新計算。

```

class Port(object):

    def recv_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

```

故障偵測

直接的故障例如：斷線，或者間接的故障例如：在一定時間內沒有接收到 Root 橋接器所發出的 BPDU 封包，這時候 STP 就必須進行重新計算。

斷線是由 STP 類別的 PortStatus 事件管理器所偵測，並透過 Bridge 類別的實體進行通知。

BPDU 封包的接收逾時是由 Port 類別的 BPDU 封包接收執行緒 (Port.wait_bpdu_thread) 所發現。在 max age (預設 20 秒) 之間，如果沒有接收到 Root 橋接器發來的 BPDU 封包，就判斷為間接故障，並且對 Bridge 類別實體發送通知。

BPDU 接收逾時的更新和逾時的偵測分別是 hub 模組 (ryu.lib.hub) 的 hub.Event 和 hub.Timeout。
◦ hub.Event 經由 hub.Event.wait() 進入 wait 狀態，透過 hub.Event.set() 中斷執行緒。
hub.Timeout 指定在一定時間內若 try 無法結束執行時，則發送 hub.Timeout 的例外事件。
當 hub.Event 進入 wait 狀態，並且 hub.Timeout 所指定的時間內尚未執行 hub.Event.set() 時，則判斷為 BPDU 封包的接收逾時，故開始進行 Bridge 類別的 STP 重新計算。

```
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 封包比對動作 (Stp.compare_bpdu_info()) 結束後發現是 SUPERIOR 或 REPEATED 時，就開始接收 Root 橋接器發送過來的封包，並且更新逾時數值 (Port._update_wait_bpdu_timer())。執行 hub.Event 中 Port.wait_timer_event 的 set() 將 Port.wait_timer_event 從 wait 狀態解除，即可讓 BPDU 封包接收執行緒 (Port.wait_bpdu_thread) 不要進入 except hub.Timeout 的區間並進行處理，進而重新設定計時器後繼續接收下一個 BPDU 封包。

```
class Port(object):

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

```

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

    if ((recv_info is SUPERIOR or recv_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 計算

STP 計算（Root 橋接器選擇、每一個連接埠的角色選擇）是由 Bridge 類別所執行。

STP 重新計算開始的時候代表網路拓墣已經發生變化，此時封包的傳遞有可能會有出現迴圈，因此所有的連接埠會進入 BLOCK(port.down) 狀態並觸發拓墣改變事件(EventTopologyChange) 通知上層 API，初始已經學習完畢的 MAC 位址連接埠。

然後 Bridge._spanning_tree_algorithm() 開始動作以進行 Root 橋接器的選擇和連接埠的角色設定，所有的連接埠會從 LISTEN 狀態 (port.up) 開始狀態的變化。

```

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)

```

為了 Root 橋接器的選擇，像是橋接器 ID 之類的資訊會拿來跟每一個連接埠所接收到的 BPDU 封包進行比對 (Bridge._select_root_port)。

接著會出現這樣的結果：選出 Root 連接埠（本身的橋接器資訊和從連接埠所收到的其他橋接器資訊比對後結果較差）、其他的橋接器開始做為 Root 橋接器、或選出 designated ports (Bridge._select_designated_port) 和選出 non-designated ports (Root 連接埠 / designated ports 以外的 non-designated ports 選出)。

反之 Root 橋接器如果沒有被發現（本身的橋接器所帶的資訊為最優先的情況），則自己就會轉變為 Root 橋接器並設定其所有的連接埠為 designated ports。

```
class Bridge(object):

    def _spanning_tree_algorithm(self):
        """ Update tree roles.
            - Root bridge:
                all port is DESIGNATED_PORT.
            - Non root bridge:
                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
```

連接埠的狀態轉移

連接埠的狀態轉移是由 Port 類別的狀態轉移控制執行緒 (Port.state_machine) 所處理。下一個狀態的轉移時限是從 Port._get_timer() 取得。當發生逾時之後就會從 Port._get_next_state() 取得接下來將轉移的狀態並進行移轉。而 STP 再次重新計算的時候，連接埠的狀態就會直接使用 Port._change_status() 切換至 BLOCK 的狀態，不論先前的狀態為何。這樣的處理跟[故障偵測](#)」一樣，是由 hub 模組的 hub.Event 和 hub.Timeout 來達

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_LISTEN,
                      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 實作應用程式

本章說明「執行 Ryu 應用程式」中 OpenFlow 1.3 所對應的生成樹應用程式 (`simple_switch_stp_13.py`) 和「交換器（Switching Hub）」的交換器之間的差異。

設定「_CONTEXTS」

跟「[網路聚合 \(Link Aggregation\)](#)」一樣用 CONTEXT 登錄，藉以應用相同的 STP 函式庫。

```
from ryu.lib import stplib

# ...

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

    # ...
```

設定組態

使用 STP 函式庫的 `set_config()` 方法來進行組態設定，下面是簡單的例子。

OpenFlow 交換器	名稱	設定值
dpid=0000000000000001	bridge.priority	0x8000
dpid=0000000000000002	bridge.priority	0x9000
dpid=0000000000000003	bridge.priority	0xa000

使用這個設定時 `dpid=0000000000000001` 的 OpenFlow 交換器的橋接器 ID 總會是最小值，而 Root 橋接器也會選擇該交換器。

```
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 事件處理

跟「[網路聚合 \(Link Aggregation\)](#)」一樣，準備事件管理器來接收來自 STP 函式庫的通知。

使用 STP 函式庫中定義的 `stplib.EventPacketIn` 事件來接收 BPDU 以外的封包。因此封包管理及處理動作跟「[交換器 \(Switching Hub\)](#)」相同。

```
class SimpleSwitch13(app_manager.RyuApp):

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

        # ...
```

接收網路拓墣的變動事件通知（`stplib.EventTopologyChange`）用以初始化已經學習的 MAC 位址和已經註冊的 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]
```

接收到連接埠的狀態通知事件（`stplib.EventPortStateChange`）並且將連接埠的狀態輸出。

```
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])
```

經過以上的處理，透過提供生成樹功能的函式庫和使用該函式庫的應用程式，實作成了擁有生成樹功能的交換器應用程式。

5.5 本章總結

本章透過生成樹函式庫的使用，說明下列的目標。

- 使用 `hub.Event` 實作事件等待處理
- 使用 `hub.Timeout` 實作逾時管理

OpenFlow 通訊協定

本章將描述 Match、Instructions 和 Action 在 OpenFlow 協定中的細節。

6.1 Match

有許多種類的指定條件可以用在 Match，隨著 OpenFlow 版本的持續更新，數量也在持續增加中。在 OpenFlow 1.0 時僅有 12 種，OpenFlow 1.3 時數量就來到約 40 種。

若想要了解每個指令的細節，請參考 OpenFlow 規格書。本章僅簡要列出 OpenFlow 1.3 的 Match 指令。

Match Field 名稱	說明
in_port	接受埠編號 (含邏輯埠)
in_phy_port	接收埠實體編號
metadata	在 table 間傳遞使用的 Metadata
eth_src	Ethernet MAC 來源位址
eth_dst	Ethernet MAC 目的位址
eth_type	Ethernet 訊框種類
vlan_vid	VLAN ID
vlan_pcp	VLAN PCP
ip_dscp	IP DSCP
ip_ecn	IP ECN
ip_proto	IP 協定種類
ipv4_src	IPv4 IP 來源位址
ipv4_dst	IPv4 IP 目的位址
tcp_src	TCP port 來源編號
tcp_dst	TCP port 目的編號
udp_src	UDP port 來源編號
udp_dst	UDP port 目的編號
sctp_src	SCTP port 來源編號
sctp_dst	SCTP port 目的編號
icmpv4_type	ICMP 種類
icmpv4_code	ICMP Code 編碼
arp_op	ARP Opcode
arp_spa	ARP IP 來源位址
arp_tpa	ARP IP 目的位址
arp_sha	ARP MAC 來源位址

Continued on next page

Table 6.1 – continued from previous page

Match Field 名稱	說明
arp_tha	ARP MAC 目的位址
ipv6_src	IPv6 IP 來源位址
ipv6_dst	IPv6 IP 目的位址
ipv6_flabel	IPv6 Flow label
icmpv6_type	ICMPv6 Type
icmpv6_code	ICMPv6 Code
ipv6_nd_target	IPv6 neighbour discovery 目的位址
ipv6_nd_sll	IPv6 neighbour discovery link-layer 來源位址
ipv6_nd_tll	IPv6 neighbour discovery link-layer 目的位址
mpls_label	MPLS 標籤
mpls_tc	MPLS Traffic class(TC)
mpls_bos	MPLS Bos bit
pbb_isid	802.1ah PBB I-SID
tunnel_id	邏輯埠的 metadata
ipv6_exthdr	IPv6 extension header 的 Pseudo-field

可以針對 MAC 位址或 IP 位址，透過 Mask 來指定 field。

6.2 Instruction

Instruction 是用來定義當封包滿足所規範的 Match 條件時，需要執行的動作。下面列出相關的定義。

Instruction	說明
Goto Table(必要)	在 OpenFlow 1.1 或更新的版本中，multiple flow tables 將是必需支援的項目。透過 Goto Table 的指令可以在多個 table 間進行移轉，並繼續相關的比對及對應的動作。例如：「收到來自 port 1 的封包時，增加 VLAN-ID 200 的 tag，並移動至 table 2」。而所指定的 table ID 則必須是大於目前的 table ID。
Write Metadata(選項)	寫入 Metadata 以做為下一個 table 所需的參考資料。
Write Actions(必要)	在目前的 action set 中寫入新的 action，如果有相同的 action 存在時，會進行覆蓋。
Apply Actions(選項)	立刻執行所指定的 action 不對現有的 action set 進行修改。
Clear Actions(選項)	清空目前存在 action set 中的資料。
Meter(選項)	指定該封包到所定義的 meter table。

以下的類別是對應各個 Instruction 的 Ryu 實作。

- OFPInstructionGotoTable
- OFPInstructionWriteMetadata
- OFPInstructionActions
- OFPInstructionMeter

Write/Apply/Clear Actions 已經包含在 OFPInstructionActions 中，可以在安裝的時候進行選取。

註解： Write Actions 雖然在規格中被列為必要，但是目前的 Open vSwitch 並不支援該功能。Apply Actions 是目前 Open vSwitch 所提供的功能，所以可以用來替代 Write Actions。Write Actions 預計在 Open vSwitch 2.1.0 中支援。

6.3 Action

OFPPActionOutput Class 是用來轉送指定封包，其中包含 Packet-Out 和 Flow Mod。設定好要傳送的最大封包容量 (max_len) 和要傳送的 Controller 目的地做為 Constructor 的參數。對於設定目的地，除了實體連接埠號之外還有一些其他的值可以進行定義。

名稱	說明
OFPP_IN_PORT	轉送到接收埠
OFPP_TABLE	轉送到最前端的 table
OFPP_NORMAL	使用交換器本身的 L2 / L3 功能轉送
OFPP_FLOOD	轉送 (Flood) 到所有 VLAN 的物理連接埠，除了來源埠跟已閉鎖的埠之外
OFPP_ALL	轉送到除了來源埠之外的所有埠
OFPP_CONTROLLER	轉送到 Controller 的 Packet-In 訊息
OFPP_LOCAL	轉送到交換器本身 (local port)
OFPP_ANY	使用 Wild card 來指定 Flow Mod (delete) 或 Flow Stats Requests 訊息的埠號，主要功能並不是用來轉送封包訊息。

當指定 max_len 為 0 時，Binary data 將不會被加在 Packet-In 的訊息中。當 OFPCML_NO_BUFFER 被指定時，所有的封包將會加入 Packet-In 訊息中而不會暫存在 OpenFlow 交換器。

ofproto 函式庫

本章介紹 Ryu ofproto 函式庫。

7.1 簡單說明

ofproto 函式庫是用來產生及解析 OpenFlow 訊息的函式庫。

7.2 相關模組

每個 OpenFlow (版本 X.Y) 都有相對應的常數模組 (ofproto_vX_Y) 和解析模組 (ofproto_vX_Y_parser) 每個 OpenFlow 版本的實作基本上是獨立的。

OpenFlow 版本	常數模組	解析模組
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 常數模組

常數模組是用來做為通訊協定中的常數設定使用，下面列出幾個例子。

常數	說明
OFP_VERSION	通訊協定版本編號
OFPP_xxxx	連接埠號
OFPCML_NO_BUFFER	無緩衝區間，直接對全體發送訊息
OFP_NO_BUFFER	無效的緩衝編號

7.2.2 解析器模組

解析模組提供各個 OpenFlow 訊息的對應類別，下面列出幾個例子。為了更好的說明類別的實體，接下來的描述將用”訊息物件”取代”訊息類別”。

類別 (Class)	說明
OFPHello	OFPT_HELLO 訊息
OFPPacketOut	OFPT_PACKET_OUT 訊息
OFPFlowMod	OFPT_FLOW_MOD 訊息

解析模組對應了 OpenFlow 訊息的 Payload 結構中所需的定義，例如下面的例子。為了更好的說明類別的實體，今後將用結構物件（structure object）取代結構類別（structure class）。

類別 (Class)	結構
OFPMatch	ofp_match
OFPIInstructionGotoTable	ofp_instruction_goto_table
OFPActionOutput	ofp_action_output

7.3 基本使用方法

7.3.1 ProtocolDesc 類別

這是為了 OpenFlow 協定所必需存在的類別。使用訊息類別的 `_init_` 作為 `datapath` 的參數，指定該類別的物件。

```
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 網路位址 (Network Address)

Ryu ofproto 函式庫的 API 使用最基本的文字表現網路位址，請看下面的例子。

註解：但是 OpenFlow 1.0 和這樣的標示方法不同。（2014 年 2 月更新）

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

7.3.3 訊息物件 (Message Object) 的產生

每個訊息類別（message class），結構類別（structure class）都需要適當的參數以用來產生。參數的名稱跟 OpenFlow 協定所定義的欄位名稱基本上是一致的。在有衝突的情況下會在最後加入「_」，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.OFPIINSTRUCTIONACTIONS(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)
```

註解：常數模組、解析模組最好是在 import 的時候就直接標明。如此一來在 OpenFlow 版本變更的時候，可以將修正的程度降到最低。另外儘量使用 ProtocolDesc 物件的 ofproto 和 ofproto_parser 屬性。

7.3.4 訊息物件（Message Object）的解析

訊息物件（message object）的內容是可以查詢的。

例如 OFPPacketIn 物件中 pid 的 match field 用查詢 pin.match 即可得到相關的訊息。

OFPMatch 物件中 TLV 的各部分可以使用下列的名稱取得相關的資料。

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

7.3.5 JSON

訊息物件（message object）轉換成為 json.dump 的功能是存在的，反之亦然。

註解：但是目前 OpenFlow 1.0 相關的實作並不完全。（2014 年 2 月更新）

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

7.3.6 訊息（message）的解析（parse）

該功能是為了把訊息的原始資料轉換成訊息物件。對於從交換器收到的訊息，框架（Framework）會自動地進行處理，Ryu 應用程式（Application）是不需要特別處理的。

具體來說如下：

1. ryu.ofproto.ofproto_parser.header 用來處理版本相依的解析
2. 上面處理過的結果可以用 ryu.ofproto.ofproto_parser.msg 功能來解析剩餘的部分

7.3.7 訊息的產生（序列化，Serialize）

將訊息物件轉換並產生對應的訊息 Byte。同樣的，來自交換器的訊息將由框架自動處理，Ryu 應用程式無需額外的動作。

具體來說如下：

1. 呼叫訊息物件的序列化方法
2. 從訊息物件中將 buf 的屬性讀取出來

有些欄位，例如 “len” 即使不指定，在序列化的同時也會自動被計算出來。

封包函式庫

OpenFlow 中 Packet-In 和 Packet-Out 訊息是用來產生封包，可以在當中的欄位放入 Byte 資料並轉換為原始封包的方法。Ryu 提供了相當容易使用的封包產生函式庫給應用程式使用。

本章將介紹該函式庫。

8.1 基本使用方法

8.1.1 協定標頭類別 (Protocol Header Class)

Ryu 封包函式庫提供許多協定對應的類別，用來解析或包裝封包。

下面列出 Ryu 目前所支援的協定。若需要了解每個協定的細節請參照 [API 參考資料](#)

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

- tcp
- udp
- vlan
- vrrp

每一個協定類別的 `_init_` 參數基本上跟 RFC 所提到的名稱是一致的。同樣的每一個協定實體命名也相同。但是當 `_init_` 名稱與 Python 內定的關鍵字發生衝突時，會在名稱的尾端加上底線「_」。

有些 `_init_` 的參數由於有內定的預設值，因此可以忽略。下面的例子中，原本需要被加入的參數 `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 網路位址 (Network Address)

Ryu 封包函式庫的 API 使用最基本的的文字作為表現。舉例如下：

位址種類	python 文字表示
MAC 位址	`'00:03:47:8c:a1:b3'
IPv4 位址	`'192.0.2.1'
IPv6 位址	`'2001:db8::2'

8.1.3 封包的解析 (Parse)

封包的 Byte String 可以產生相對應的 Python 物件。

具體的事例如下：

1. `ryu.lib.packet.packet.Packet` 類別的物件產生（指定要解析的 byte string 為 `data` 作為參數）
2. 使用先前產生的物件中 `get_protocol` 方法，取得協定中相關屬性的物件。

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

8.1.4 封包的產生 (序列化，Serialize)

把 Python 物件轉換成為相對封包的 byte string。

具體說明如下：

1. 產生 `ryu.lib.packet.packet.Packet` 類別的物件
2. 產生相對應的協定物件 (`ethernet`, `ipv4`, ...)
3. 在 1. 所產生的物件中，使用 `add_protocol` 方法將 2. 所產生的物件依序加入
4. 呼叫 1. 所產生物件中的 `serialize` 方法將物件轉換成 byte string

`Checksum` 和 `payload` 的長度不需要特別設定，在序列化的同時會被自動計算出來。詳細的各類別細節請參考相關資訊。

```
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 類似 Scapy，請根據個人喜好選擇使用。

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

8.2 應用程式範例

接下來的例子是使用上述的方法，達成一個可以針對 ping 做出回應的應用程式。

接受 `Packet-In` 所收到的 ARP REQUEST 和 ICMP ECHO REQUEST 後藉由 `Packet-Out` 發送回應。IP 位址等 `__init__` 的參數都是使用固定程式碼 (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.OFPIInstructionActions(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_ether = pkt.get_protocol(ether.ethernet)
        if not pkt_ether:
            return
        pkt_arp = pkt.get_protocol(arp.arp)
        if pkt_arp:
            self._handle_arp(datapath, port, pkt_ether, 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_ether, pkt_ipv4, pkt_icmp)
            return

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

註解：OpenFlow 1.2 版本之後，因為 match 而帶來的 Packet-In 訊息中，將會帶有已經被解析過的資訊。但是這些資訊的多寡以及詳細程度要看每一台交換器的實際處理決定。例如 Open vSwitch 僅放入最低需求的資訊，在大多數的情況下 Controller 需要針對資料再進行處理。反之 LINC 則盡可能放入資訊。

以下是使用 “ping -c 3” 時所產生的記錄檔 (log)。

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

IP fragments 將會是使用者需要解決的課題。由於 OpenFlow 協定本身並沒有提供得到 MTU 資訊的方法，目前僅能使用其他方法解決。例如固定程式碼 (hard-code)。另外，因為 Ryu 封包函式庫會對所有的封包進行解析或序列化，你將會需要使用 API 來處理封包斷裂 (fragmented) 的問題。

OF-Config 函式庫

本章將介紹 Ryu 內建的 OF-Config 客戶端函式庫。

9.1 OF-Config 通訊協定

OF-Config 是用來管理 OpenFlow 交換器的一個通訊協定。OF-Config 通訊協定被定義在 NETCONF (RFC 6241) 的標準中，它可以對邏輯交換器的通訊埠 (Port) 和佇列 (Queue) 進行設定以及資料擷取。

OF-Config 是被同樣制訂 OpenFlow 的 ONF (Open Network Foundation) 所研擬，請參考下列資料以取得更詳盡的資訊。

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

Ryu 提供的函式庫完全相容于 OF-Config 1.1.1 版本

註解：目前 Open vSwitch 並不支援 OF-Config，僅提供 OVSDB 作為替代使用。由於 OF-Config 還算是比較新的規格，因此 Open vSwitch 的 OVSDB 並不實作 OF-Config。

OVSDB 通訊協定雖然公開規範在 RFC 7047 作為標準，但事實上目前僅作為 Open vSwitch 專用的通訊協定。而 OF-Config 相對來說還是相對新的協定，期望在不久的將來會有更多實作它的 OpenFlow 交換器出現。

9.2 函式庫架構

9.2.1 ryu.lib.of_config.capable_switch.OFCapableSwitch Class

本 Class 主要用來處理 NETCONF 會話 (Session)。

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

9.2.2 ryu.lib.of_config.classes 模組 (Module)

本模組用來將協定相關的設定對應至 Python 物件。

註解：類別名稱基本上遵照 OF-Config 1.1.1 中 yang specification 的 Grouping 關鍵字名稱來命名。

例如：OFPType

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

9.3 使用範例

9.3.1 交換器的連結

使用 SSH Transport 連線到交換器。回呼 (callback) 函式 `unknown_host_cb` 是用來對應未知的 SSH Host Key 時所被執行的函式。下面的範例中我們使用無條件信任對方並繼續進行連結。

```
sess = OFCapableSwitch(
    host='localhost',
    port=1830,
    username='linc',
    password='linc',
    unknown_host_cb=lambda host, fingerprint: True)
```

9.3.2 GET

使用 NETCONF GET 來取得交換器的狀態。下面的範例將會用 `/resources/port/resource-id` 和 `/resources/port/current-rate` 表示所有連接埠的狀態。

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

9.3.3 GET-CONFIG

下面的範例是使用 NETCONF GET-CONFIG 來取得目前的交換器設定值。

註解： `running` 用來表示現在儲存在 NETCONF 中目前的設定狀態。但這跟交換器的實作有關，或者你也可以儲存相關設定在 `startup` (設備啟動時) 或 `candidate` (Candidate set)。

其結果會使用 `/resources/port/resource-id` 和 `/resources/port/configuration/admin-state` 表示所有連接埠的狀態。

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

9.3.4 EDIT-CONFIG

這個範例說明如何使用 NETCONF EDIT-CONFIG 來對設定進行變更。首先使用 GET-CONFIG 取得交換器的設定，進行相關的編輯動作，最後使用 EDIT-CONFIG 將變更傳送至交換器。

註解： 另外也可以使用 EDIT-CONFIG 直接修改部分的設定，這樣做將更為安全。

將全部的連接埠狀態在 `/resources/port/configuration/admin-state` 中設定為 `down`。

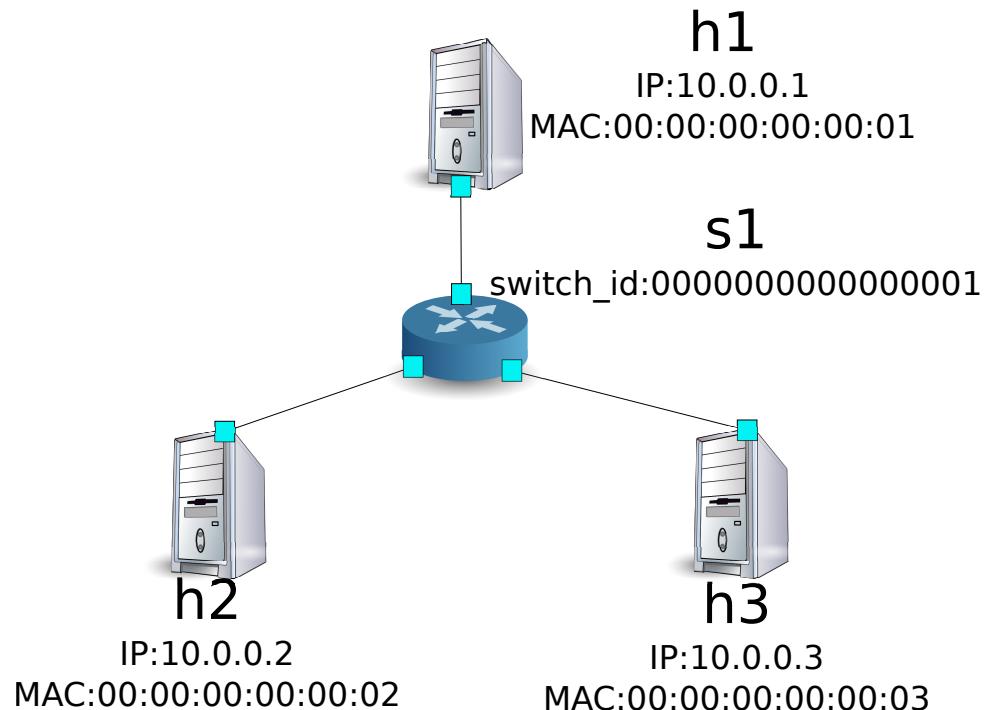
```
csw = sess.get_config('running')
for p in csw.resources.port:
    p.configuration.admin_state = 'down'
sess.edit_config('running', csw)
```

防火牆 (Firewall)

本章將說明如何利用 REST 的方式使用防火牆。

10.1 Single tenant 操作範例

以下說明如何建立一個如下所示的拓撲，並且對交換器 s1 進行路由的增加和刪除。



10.1.1 環境構築

首先在 Mininet 上建構環境。所要輸入的指令跟「交換器 (Switching Hub)」是一樣的。

```

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>
```

接著建立一個新的 xterm 用來操作 Controller。

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

將 OpenFlow 的版本設定為 1.3。

switch: s1 (root):

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

最後在控制 Controller 的 xterm 上啟動 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 和交換器中間的連線已經完成後，會出現下面的訊息。

controller: c0 (root):

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

10.1.2 改變初始狀態

防火牆啟動後，在初始狀態下全部的網路都會處於無法連線的狀態。接下來我們要下指令使其生效，並開放網路的連線。

註解：接下來的說明會使用到 REST API，若需要詳細的解釋請參考本章結尾的「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",  
      "details": "firewall running."
```

```

        }
    ]
}

root@ryu-vm:~# curl http://localhost:8080/firewall/module/status
[
{
    "status": "enable",
    "switch_id": "00000000000000000001"
}
]

```

註解： REST 命令執行的結果已經被格式為較為容易理解的格式。

確認可以從 h1 向 h2 執行 ping 指令。但是存取的權限規則並沒有被設定，所以目前是處於無法連通的狀態。

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

```

封包被阻擋的過程被寫進記錄檔（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 新增規則

增加 h1 和 h2 之間允許 ping 發送的規則。不論是從哪個方向都需要加入。

接下來新增規則，規則的編號會自動編碼。

來源	目的	通訊協定	連線狀態	規則 ID
10.0.0.1/32	10.0.0.2/32	ICMP	通過	1
10.0.0.2/32	10.0.0.1/32	ICMP	通過	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/00000000000000000001
[
{
    "switch_id": "00000000000000000001",
    "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/00000000000000000001
[
{
    "switch_id": "00000000000000000001",
    "command_result": [
        {
            "result": "success",
            "details": "Rule added. : rule_id=2"
        }
    ]
}
]

```

新增加的規則做為 Flow Entry 被註冊到交換器中。

switch: s1 (root):

```

root@ryu-vm:~# ovs-ofctl -0 openflow13 dump-flows s1
OFPST_FLOW reply (0F1.3) (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

```

接著 h2 和 h3 之間，新增加規則允許包含 ping 的所有 ipv4 封包通過。

來源	目的	通訊協定	連線狀態	規則 ID
10.0.0.2/32	10.0.0.3/32	any	通過	3
10.0.0.3/32	10.0.0.2/32	any	通過	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/00000000000000000001
[
{
    "switch_id": "00000000000000000001",
    "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/00000000000000000001
[
{
    "switch_id": "00000000000000000001",
    "command_result": [
        {
            "result": "success",

```

```

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

新增的規則作為 Flow Entry 被註冊到交換器當中。

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

```

可以設定規則的優先權。

新增阻斷 h2 和 h3 之間的 ping (ICMP) 封包規則。優先權的預設值設定為大於 1 的值。

優先權	來源	目的	通訊協定	連線狀態	規則 ID
10	10.0.0.2/32	10.0.0.3/32	ICMP	中斷	5
10	10.0.0.3/32	10.0.0.2/32	ICMP	中斷	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/00000000000000000001
[
  {
    "switch_id": "00000000000000000001",
    "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/00000000000000000001
[
  {
    "switch_id": "00000000000000000001",
    "command_result": [
      {
        "result": "success",
        "details": "Rule added. : rule_id=6"
      }
    ]
  }
]
```

新增的規則做為 Flow Entry 註冊到交換器當中。

switch: s1 (root):

```
root@ryu-vm:~# ovs-ofctl -0 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 確認規則

確認已經設定完成的規則。

Node: c0 (root):

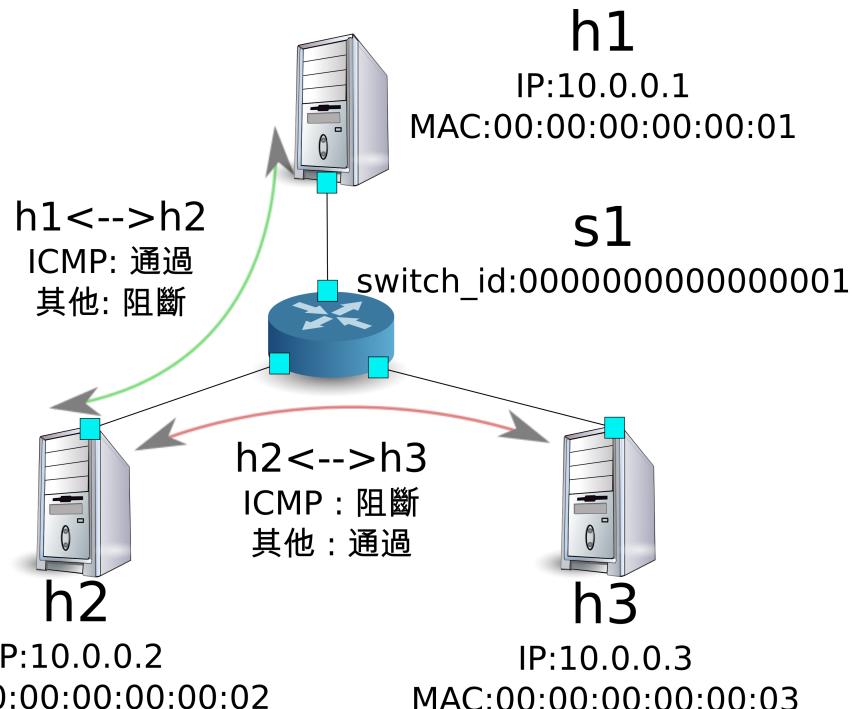
```
root@ryu-vm:~# curl http://localhost:8080/firewall/rules/00000000000000000001
[
  {
    "access_control_list": [
      {
        "rules": [
          {
            "priority": 1,
            "dl_type": "IPv4",
            "nw_dst": "10.0.0.3",
            "nw_src": "10.0.0.2",
            "rule_id": 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"
          }
        ]
      }
    ]
  }
]
```

```

        "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"
}
]

```

設定完成的規則如下。



從 h1 向 h2 執行 ping。如果允許的規則有被正確設定的話，ping 就可以正常連線。

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
...
```

從 h1 發送到 h2 非 ping 的封包會被防火牆所阻擋。例如從 h1 發送到 h2 的 wget 指令就會被阻擋下來並記錄在記錄檔 (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=00000000000000001: 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 之間除了 ping 以外的封包則允許被通過。例如從 h2 向 h3 發送 ssh 指令，記錄檔 (log) 中並不會出現封包被阻擋的記錄（如果 ssh 是發送到 h3 以外的地點，則 ssh 的連線將會失敗）。

host: h2:

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

從 h2 向 h3 發送 ping 指令，封包將會被防火牆所阻擋，並出現在記錄檔 (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=00000000000000001: 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\xab\x8b\t\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 刪除規則

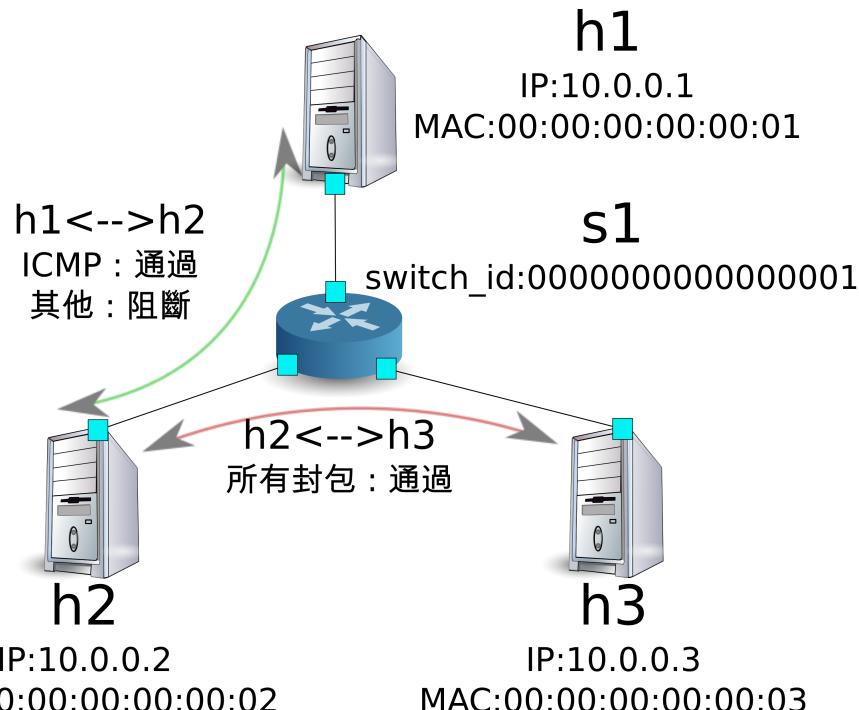
刪除 ``rule_id:5'' 和 ``rule_id:6'' 的規則。

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"
      }
    ]
  }
]
```

現在的規則如下圖所示。



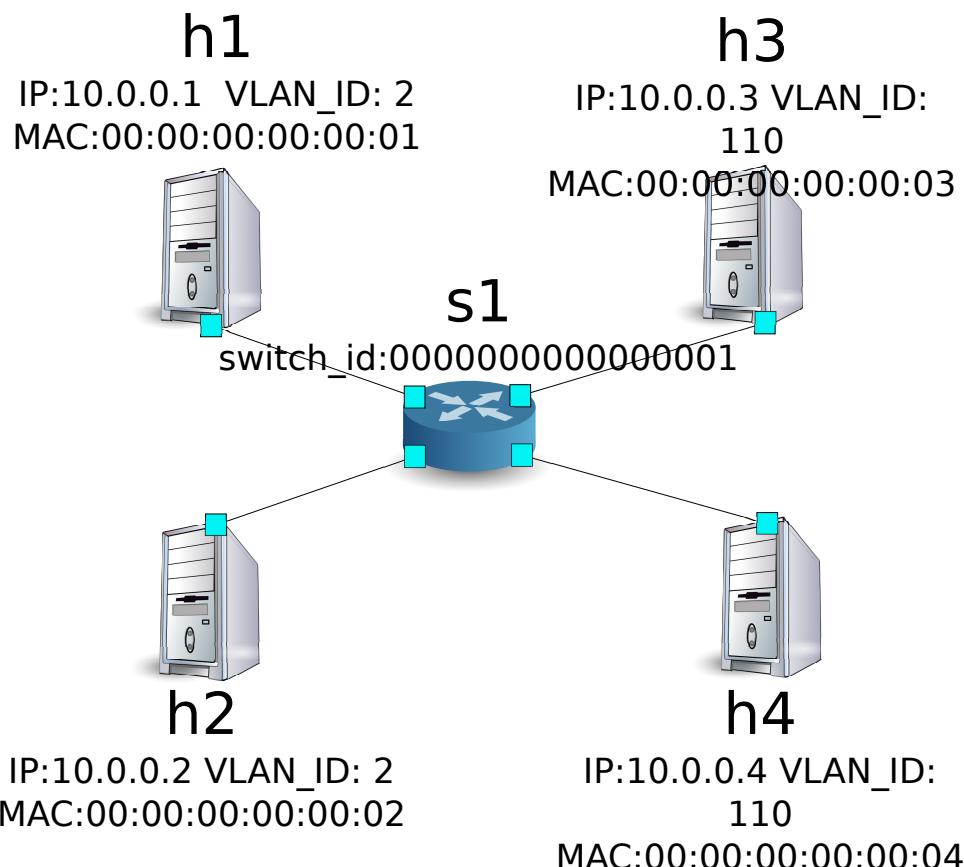
經實際確認。h2 和 h3 之間的 ping (ICMP) 阻擋連線的規則刪除後，ping 指令現在可以被正常執行並進行通訊。

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 操作範例

接下來這個例子將建立拓墣並使用 VLAN 來對 tenants 進行處理，還有像是路由或是位址對於交換器 s1 對的新增或刪除，以及每一個連接埠之間的連通做驗證。



10.2.1 環境構築

下面的例子使用 Single-tenant，在 Mininet 上進行環境的建置，另外開啟一個 xterm 做為控制 Controller 的方法，請注意與之前相比這邊需要多一台 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>
```

接下來到每一個 host 的界面中設定 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
```

接著將使用的 OpenFlow 版本設定為 1.3。

switch: s1 (root):

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

最後，從 controller 的 xterm 畫面中啟動 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 和交換器之間的連線已經成功的話，就會出現接下來的訊息。

controller: c0 (root):

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

10.2.2 變更初始狀態

啟動防火牆。

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 新增規則

新增允許使用 VLAN_ID = 2 向 10.0.0.0/8 發送 ping 訊息 (ICMP 封包) 的規則到交換器中，設定雙向的規則是必要的。

優先權	VLAN ID	來源	目的	通訊協定	連線狀態	規則 ID
1	2	10.0.0.0/8	any	ICMP	通過	1
1	2	any	10.0.0.0/8	ICMP	通過	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": "00000000000000000001",  
    "command_result": [  
      {  
        "result": "success",  
        "vlan_id": 2,  
        "details": "Rule added. : rule_id=2"  
      }  
    ]  
  }  
]
```

10.2.4 規則確認

確認已經設定的規則。

Node: c0 (root):

```
root@ryu-vm:~# curl http://localhost:8080/firewall/rules/00000000000000000001/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": "00000000000000000001"  
  }  
]
```

讓我們確認一下實際狀況。在 VLAN_ID = 2 的情況下，從 h1 發送的 ping 在 h2 也同樣是 VLAN_ID = 2 的情況下，你會發現他是連通的，因為我們剛才已經把規則加入。

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 的情況下 h3 和 h4 之間，由於規則沒有被加入，所以 ping 封包被阻擋。

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
```

封包被阻斷的時候會被記錄在記錄檔 (log) 之中。

controller: c0 {root}:

```
[FW] [INFO] dpid=00000000000000000001: 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\x9a\xaeR\x00\x00\x00
\x00\xce\xe3\x02\x00\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x
\x1b\x1c\x1d\x1e\x1f !"#$%&\'( )*+, -./01234567', id=7760, seq=4), type=8)
...
```

本章中，透過具體的例子說明學到如何使用防火牆。

10.3 REST API 列表

本章說明中所提到的 rest_firewall REST API 一覽。

10.3.1 取得交換器的防火牆狀態

方法	GET
URL	/firewall/module/status

10.3.2 變更交換器的防火牆狀態

方法	PUT
URL	/firewall/module/{ op }/{ switch }
備註	-- op : [``enable'' ``disable''] -- switch : [``all'' 交換器 ID] 交換器的初始狀態均為 ``disable''

10.3.3 取得全部規則

方法	GET
URL	/firewall/rules/{ switch }/{ vlan }
備註	-- switch : [``all'' 交換器 ID] -- vlan: [``all'' VLAN ID] VLAN ID 的指定可選擇加或不加。

10.3.4 新增規則

方法	POST
URL	/firewall/rules/{ switch }[{ vlan }] -- switch : [``all'' 交換器 ID] -- vlan : [``all'' VLAN ID]
資料	priority :[0 - 65535] in_port :[0 - 65535] dl_src :"<xx:xx:xx:xx:xx:xx>" dl_dst :"<xx:xx:xx:xx:xx:xx>" dl_type :[``ARP'' ``IPv4''] nw_src :"<xxx.xxx.xxx.xxx/xx>" nw_dst :"<xxx.xxx.xxx.xxx/xx>" nw_proto "[``TCP'' ``UDP'' ``ICMP''] tp_src :[0 - 65535] tp_dst :[0 - 65535] actions :[``ALLOW'' ``DENY'']
備註	註冊成功的規則會自動產生規則 ID，並註明在回應的訊息中。 指定 VLAN ID 為可附加之選項。

10.3.5 刪除規則

方法	DELETE
URL	/firewall/rules/{ switch }[{ vlan }] -- switch : [``all'' 交換器 ID] -- vlan : [``all'' VLAN ID]
資料	rule_id : [``all'' 1 - ...]
備註	指定 VLAN ID 為可附加之選項。

10.3.6 取得交換器的記錄檔

方法	GET
URL	/firewall/log/status

10.3.7 變更交換器記錄檔的狀態

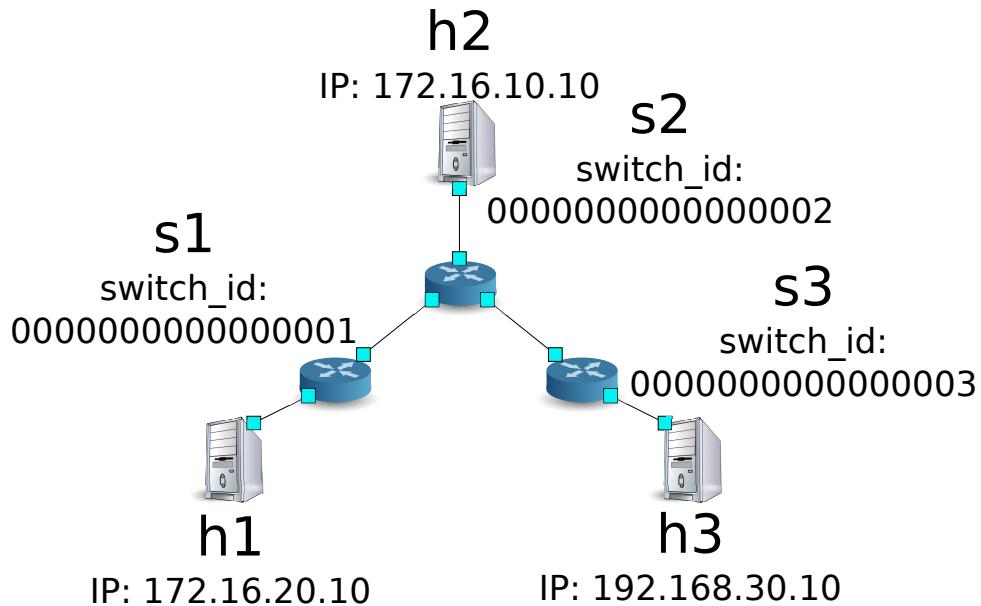
方法	PUT
URL	/firewall/log/{ op }/{ switch } -- op: [``enable'' ``disable''] -- switch: [``all'' 交換器 ID]
備註	設定每一個交換器的初始狀態為”啟用”

路由器 (Router)

本章將說明如何使用 REST 來設定一個路由器。

11.1 Single Tenant 的操作範例

下面的例子介紹如何建立拓墣，每個交換器（路由器）的位址新增或刪除，及確認 host 之間的連線狀況確認。



11.1.1 環境建置

首先要在 Mininet 上建置環境。mn 的命令及參數如下。

名稱	設定值	說明
topo	linear,3	3 台交換器直接連結的網路拓墣
mac	無	自動設定各 host 的 MAC 位址
switch	ovsk	使用 Open vSwitch
controller	remote	使用外部的 Controller 做為 OpenFlow controller
x	無	啟動 xterm

執行的動作如下：

```
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>
```

接著，開啟 Controller 所使用的 xterm。

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

然後設定每個路由器的 OpenFlow 版本為 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
```

接著每一個 host 刪除原先自動配置的 IP 位址，並設定新的 IP 位址。

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
```

最後在操作 Controller 的 xterm 上啟動 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 和交換器之間的連接成功，接下來的訊息將會被顯示。

controller: c0 (root):

```
[RT] [INFO] switch_id=00000000000000000003: Set SW config for TTL error packet in.
[RT] [INFO] switch_id=00000000000000000003: Set ARP handling (packet in) flow [cookie=0x0]
[RT] [INFO] switch_id=00000000000000000003: Set L2 switching (normal) flow [cookie=0x0]
[RT] [INFO] switch_id=00000000000000000003: Set default route (drop) flow [cookie=0x0]
[RT] [INFO] switch_id=00000000000000000003: Start cyclic routing table update.
[RT] [INFO] switch_id=00000000000000000003: Join as router.
...
```

上述的 log 表示 3 台交換器已經準備完成。

11.1.2 設定 IP 位址

設定每一個路由器的 IP 位址。

首先，設定交換器 s1 的 IP 位址為「172.16.20.1/24」和「172.16.30.30/24」。

註解：接下來的說明中所使用的 REST API 請參考本章結尾的「[REST API 列表](#)」以取得更詳細的資料。

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]"
      }
    ]
  }
]
```

```
    }
]
```

註解：REST 命令的執行結果已經被整理為較好閱讀的格式。

接著，設定交換器 s2 的 IP 位址為「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]"
      }
    ]
  }
]
```

接著設定交換器 s3 的 IP 位址為「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=4]"
      }
    ]
  }
]
```

```

        "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 位址已經被設定完成，接著對每一個 host 新增預設的閘道。

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 設定預設路由

設定每一個路由器的預設路由。

首先，設定路由器 s1 的路由為路由器 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]"
        }
    ]
}
]
```

設定路由器 s2 的預設路由為路由器 s1 。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"gateway": "172.16.30.30"}' http://localhost:8080/router/00000000000000000002
[
  {
    "switch_id": "00000000000000000002",
    "command_result": [
      {
        "result": "success",
        "details": "Add route [route_id=1]"
      }
    ]
  }
]
```

設定路由器 s3 的預設路由為路由器 s2。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"gateway": "192.168.10.1"}' http://localhost:8080/router/00000000000000000003
[
  {
    "switch_id": "00000000000000000003",
    "command_result": [
      {
        "result": "success",
        "details": "Add route [route_id=1]"
      }
    ]
  }
]
```

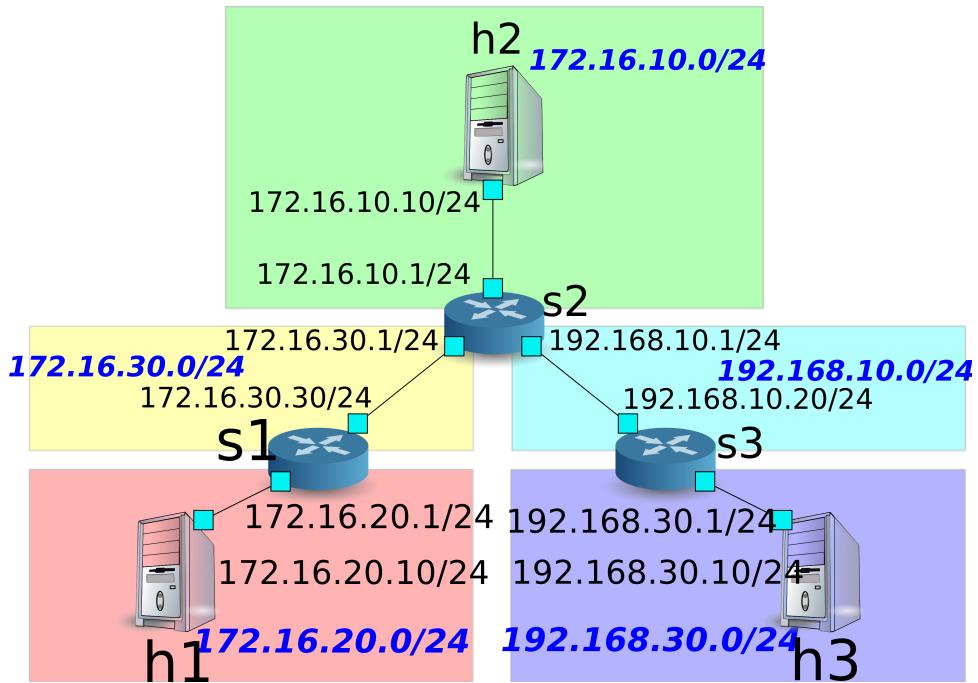
11.1.4 設定靜態路由

為了路由器 s2，設定路由器 s3 的靜態路由為 (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/00000000000000000002
[
  {
    "switch_id": "00000000000000000002",
    "command_result": [
      {
        "result": "success",
        "details": "Add route [route_id=2]"
      }
    ]
  }
]
```

IP 位址及路由的設定狀態如下。



11.1.5 確認設定的內容

確認每一個路由器的內容。

Node: c0 (root):

```
root@ryu-vm:~# curl http://localhost:8080/router/00000000000000000001
[
  {
    "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/00000000000000000002
[
  {
    "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"  
    }  
]
```

在這樣的狀態下，執行 ping 來確認相互間的連接狀態。首先執行從 h2 向 h3 執行 ping。確認正

常連通的狀態。

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
...
```

接著，從 h2 向 h1 執行 ping。確認這邊也是正常的連接狀態。

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 刪除靜態路由

刪除路由器 s2 上指向路由器 s3 的靜態路由。

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]"
      }
    ]
  }
]
```

確認路由器 s2 的設定。這邊可以看到原先指向路由器 s3 的靜態路由已經被刪除了。

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": [
          {
            "ip": "172.16.30.30",
            "mask": "255.255.255.0"
          }
        ]
      }
    ]
  }
]
```

```
        "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"
}
```

在這個狀態下，使用 ping 來確認連結狀態。從 h2 向 h3 執行 ping 會發現無法通過連接測試，這是因為我們已經刪除了路由的關係。

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 刪除 IP 位址

刪除已經設定在路由器 s1 上的 IP 位址「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]"
            }
        ]
    }
]
```

確認路由器 s1 的設定狀態。這邊可以看到路由器 s1 中原先被設定的「172.16.20.1/24」已經被刪除。

Node: c0 (root):

```
root@ryu-vm:~# curl http://localhost:8080/router/0000000000000001
[
    {
        "internal_network": [
            {
                "route": [
                    {
                        "id": 1,
                        "network": "172.16.20.0/24",
                        "next_hop": "192.168.10.1"
                    }
                ]
            }
        ]
    }
]
```

```

        "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"
}
]
]
]
]
]
```

在這個狀態下，使用 ping 指令來確認連通的狀況。從 h2 向 h1 執行，這時可以發現由於 h1 的子網路相關設定及路由已經被刪除的關係，是無法連通的。

host: h2:

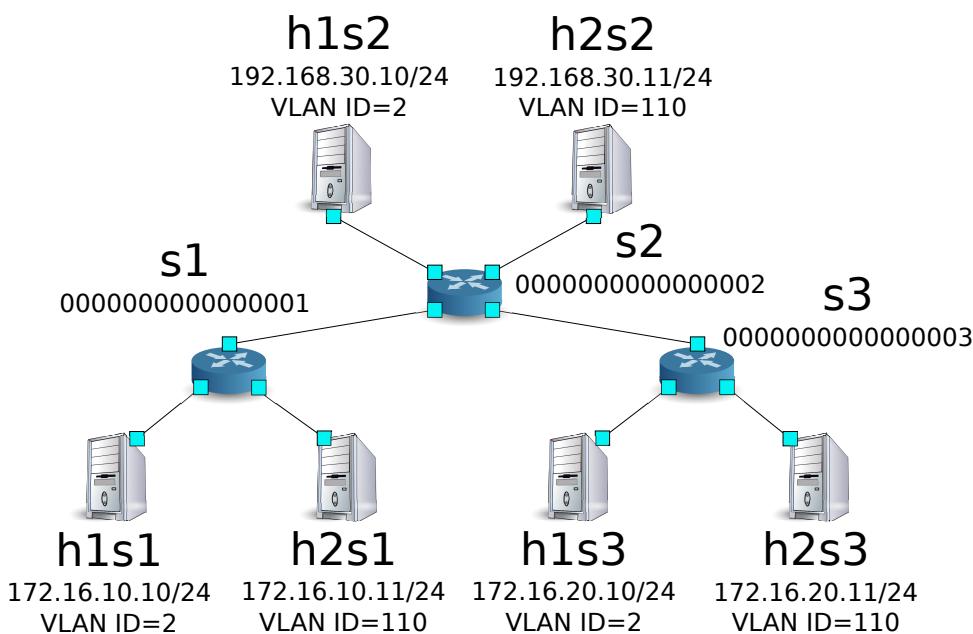
```

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 的操作範例

接下來的例子將建立一個網路拓墣，使用 VLAN 來分割 tenant 的使用。對各個交換器（路由器）的位址或路由進行新增和刪除，並確認每一個 host 之間的連通狀況。



11.2.1 環境建置

首先是在 Mininet 上進行環境的建置。mn 命令的參數如下。

參數	參數值	說明
topo	linear,3,2	3 台交換器直接連結的網路拓墣 (每個交換器連接兩台 host)
mac	無	自動設定每一個 host 的 MAC 位址
switch	ovsk	使用 Open vSwitch
controller	remote	使用外部的 OpenFlow Controller
x	無	啟動 xterm

執行的範例如下。

```
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>
```

接著啟動 Controller 用的 xterm。

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

然後，將每一台路由器所使用的 OpenFlow 版本設定為 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
```

之後設定每一個 host 的 VLAN ID 和 IP 位址。

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
```

最後在連線 Controller 的 xterm 上啟動 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 和路由器之間的聯結完成的話會出現下面的訊息。

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=0x0]
[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.
...
```

上面的記錄表示三台路由器的準備已經完成。

11.2.2 設定 IP 位址

設定每一台路由器的 IP 位址。

首先，設定路由器 s1 的 IP 位址為「172.16.20.1/24」和「10.10.10.1/24」，接著 VLAN ID 的設定也是必要的。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"address": "172.16.10.1/24"}' http://localhost:8080/router/0000000000000001/2
[
  {
    "switch_id": "00000000000000000000000000000001",
    "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": "00000000000000000000000000000001",
    "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": "00000000000000000000000000000001",
    "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": "00000000000000000000000000000001",
    "command_result": [
      {
        "result": "success",
        "vlan_id": 110,
        "details": "Add address [address_id=2]"
      }
    ]
  }
]
```

接下來，設定路由器 s2 的 IP 位址為「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]"
      }
    ]
  }
]
```

然後設定路由器 s3 的 IP 位址為「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 位址已經設定好，接著設定每一個 host 的預設閘道器。

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
```

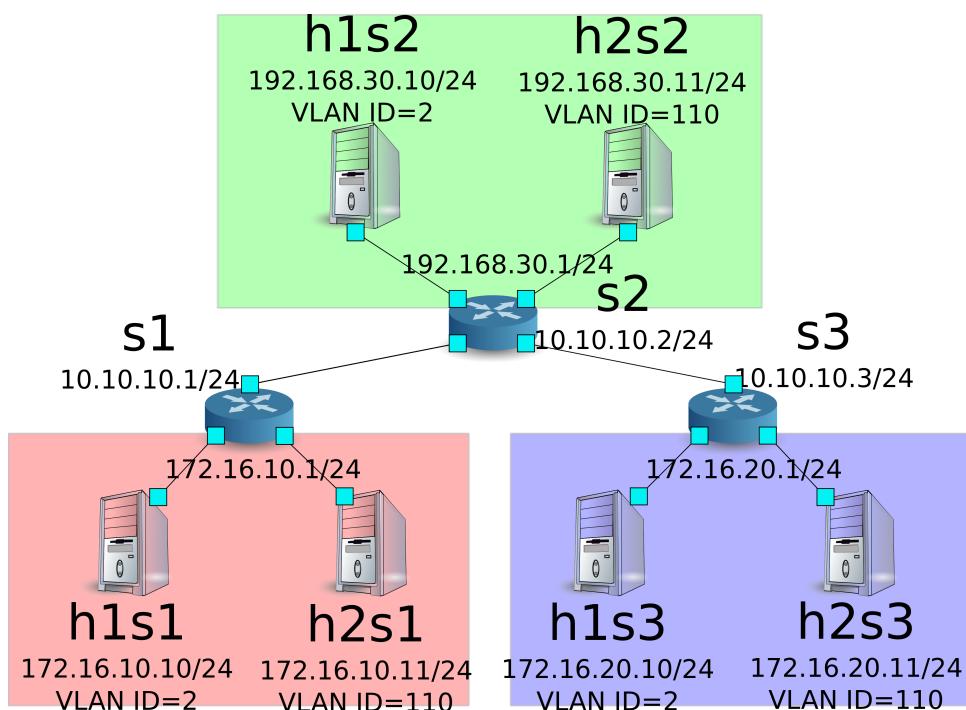
host: h1s3:

```
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 位址被設定如下。



11.2.3 設定預設靜態路由

設定每一台路由器的預設靜態路由。

首先，設定路由器 s1 的預設路由為路由器 s2。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"gateway": "10.10.10.2"}' http://localhost:8080/router/00000000000000001/2
[
  {
    "switch_id": "00000000000000001",
    "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/00000000000000001/110
[
  {
    "switch_id": "00000000000000001",
    "command_result": [
      {
        "result": "success",
        "vlan_id": 110,
        "details": "Add route [route_id=1]"
      }
    ]
  }
]
```

路由器 s2 的預設路由設定為路由器 s1。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"gateway": "10.10.10.1"}' http://localhost:8080/router/00000000000000002/2
[
  {
    "switch_id": "00000000000000002",
    "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/00000000000000002/110
[
  {
    "switch_id": "00000000000000002",
    "command_result": [
      {
        "result": "success",
        "vlan_id": 110,
        "details": "Add route [route_id=1]"
      }
    ]
  }
]
```

```

        ]
    }
]
```

路由器 s3 的預設路由設定為路由器 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]"
      }
    ]
  }
]
```

接著為了路由器 s2，將路由器 s3 的靜態路由指向 host (172.16.20.0/24)，但僅只有在 VLAN ID = 2 的情況下。

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 確認設定的內容

確認每一台路由器的設定內容。

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"}, {"route_id": 2, "destination": "172.16.10.1/24", "gateway": "10.10.10.2"}, {"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"}, {"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": 110, "address": [{"address_id": 2, "address": "10.10.10.1/24"}, {"address_id": 1, "address": "172.16.10.1/24"}]}, {"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"}]}], "switch_id": "0000000000000001"}]}
```

```
        }
    ],
    "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"
]
]
]
]
```

每一台路由器的設定內容將會如下所示。

路由器	VLAN ID	IP 位址	預設路由	靜態路由
s1	2	172.16.10.1/24 , 10.10.10.1/24	10.10.10.2(s2)	
s1	110	172.16.10.1/24 , 10.10.10.1/24	10.10.10.2(s2)	
s2	2	192.168.30.1/24 , 10.10.10.2/24	10.10.10.1(s1)	目的: 172.16.20.0/24 , 閘道: 10.10.10.3(s3)"
s2	110	192.168.30.1/24 , 10.10.10.2/24	10.10.10.1(s1)	
s3	2	172.16.20.1/24 , 10.10.10.3/24	10.10.10.2(s2)	
s3	110	172.16.20.1/24 , 10.10.10.3/24	10.10.10.2(s2)	

從 h1s1 向 h1s3 發送 ping 訊息。因為是處於相同的 `vlan_id = 2` 的相同 host，且已經設置了指向 s3 的靜態路由在 s2 上，因此應該是可以正常連線的。

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

```

從 h2s1 向 h2s3 發送 ping 封包，雖然他們處於相同的 `vlan_id = 110` 的 host，但是路由器 s2 上並沒有設置指向路由器 s3 的靜態路由，因此無法成功連線。

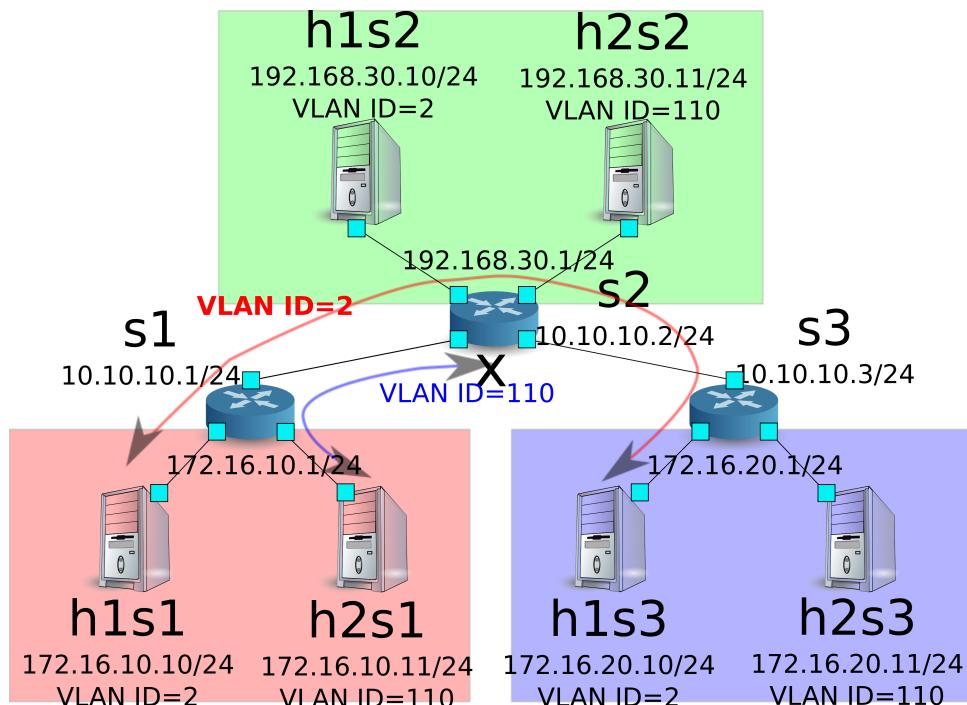
host: h2s1:

```

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，使用一個具體的例子來說明路由器的使用方法。

11.3 REST API 列表

本章所介紹的 `rest_router` 的 REST API 列表。

11.3.1 取得設定內容

方法	GET
URL	<code>/router/{switch}[/{vlan}]</code>
備註	-- switch: [``all`` 交換器 ID] -- vlan: [``all`` VLAN ID] 指定 VLAN ID 為可選項目。

11.3.2 設定位址

方法	POST
URL	<code>/router/{switch}[/{vlan}]</code>
內容 備註	-- switch: [``all`` 交換器 ID] -- vlan: [``all`` VLAN ID] <code>address:<xxx.xxx.xxx.xxx/xx></code> 在設定路由之前要先設定位址 指定 VLAN ID 為可選項目。

11.3.3 設定靜態路由

方法	POST
URL	/router/{switch}[/{vlan}] -- switch: [``all'' 交換器 ID] -- vlan: [``all'' VLAN ID]
內容	destination:"<xxx.xxx.xxx.xxx/xx>" gateway:"<xxx.xxx.xxx.xxx>"
備註	指定 VLAN ID 為可選項目。

11.3.4 設定預設路由

方法	POST
URL	/router/{switch}[/{vlan}] -- switch: [``all'' 交換器 ID] -- vlan: [``all'' VLAN ID]
內容	gateway:"<xxx.xxx.xxx.xxx>"
備註	指定 VLAN ID 為可選項目。

11.3.5 刪除位址

方法	DELETE
URL	/router/{switch}[/{vlan}] -- switch: [``all'' 交換器 ID] -- vlan: [``all'' VLAN ID]
內容	address_id:[1 - ...]
備註	指定 VLAN ID 為可選項目。

11.3.6 刪除路由

方法	DELETE
URL	/router/{switch}[/{vlan}] -- switch: [``all'' 交換器 ID] -- vlan: [``all'' VLAN ID]
內容	route_id:[1 - ...]
備註	指定 VLAN ID 為可選項目。

OpenFlow 交換器測試工具

本章將說明如何檢驗 OpenFlow 交換器對於 OpenFlow 規範的功能支援完整度，及測試工具的使用方法。

12.1 測試工具概要

本工具使用測試樣板檔案，對待測的 OpenFlow 交換器，進行 Flow Entry 和 Meter Entry 的新增以處理封包，並且將 OpenFlow 交換器所處理及轉送封包的結果與測試樣板檔案所描述的“預期處理結果”做比對。亦即檢驗 OpenFlow 交換器對於 OpenFlow 規格功能的支援狀態。

在測試工具中，已經有包含 OpenFlow 1.3 版本中的 FlowMod 訊息、MeterMod 訊息和 GroupMod 訊息的測試動作。

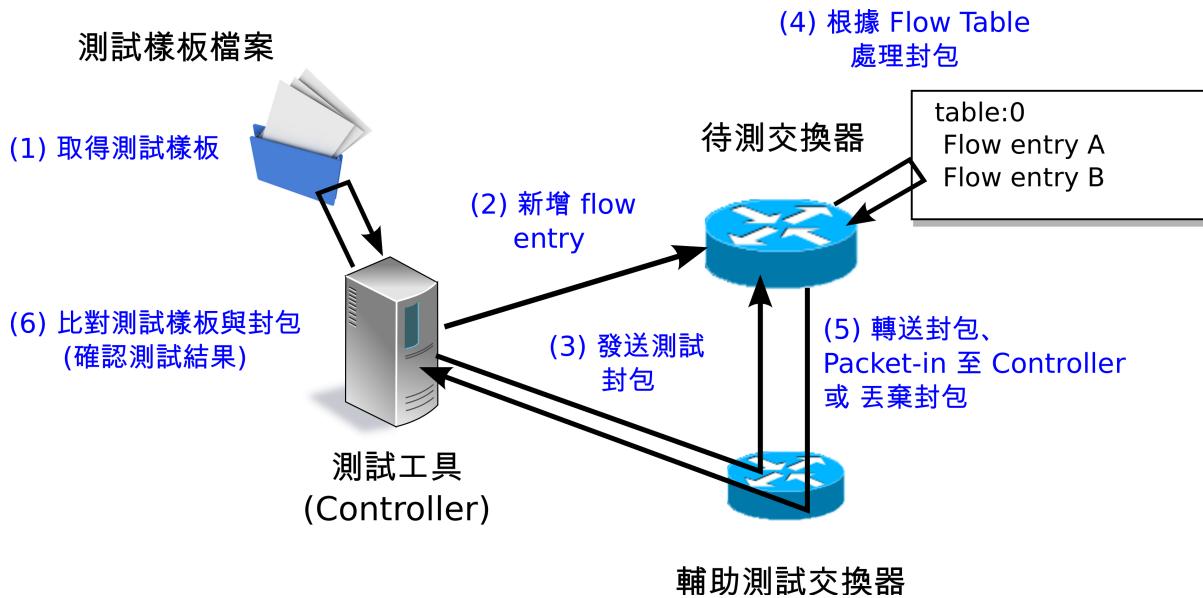
測試訊息種類	相應的參數
OpenFlow1.3 FlowMod 訊息	match (IN_PHY_PORT 除外) actions (SET_QUEUE 除外)
OpenFlow1.3 MeterMod 訊息	全部
OpenFlow1.3 GroupMod 訊息	全部

若要了解關於封包產生及修改的詳細資料，請參考「[封包函式庫](#)」。

12.1.1 操作綱要

測試驗證環境示意圖

測試工具實際執行時如下列示意圖。測試樣板檔案中包含了“待加入的 Flow Entry 和 Meter Entry”、“檢驗封包”和“預期處理結果”。為了執行測試執行所需的環境設定將在後面的執行測試的環境 章節中描述。



輸出測試結果

在指定了測試樣板檔案後，樣板中的測試案例會被依序執行，最後並顯示結果 (OK / ERROR)。在出現 ERROR 的測試結果時，錯誤訊息會同時一併出現在畫面上。最後的測試結果會顯示 OK / ERROR 的數量及錯誤內容。

```
--- 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'           ethernet/vlan/ipv6/icmpv6(type=128)
    match: 29_ICMPV6_TYPE                           ethernet/vlan/ipv6/icmpv6(code=0)-->' 
-->'icmpv6_type=128,actions=output:CONTROLLER' 
    match: 30_ICMPV6_CODE                           ethernet/vlan/ipv6/icmpv6(code=0)-->' 
icmpv6_code=0,actions=output:2'                  ethernet/vlan/ipv6/icmpv6(code=0)-->' 
    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

下面說明如何使用測試工具。

12.2.1 測試範本檔案

你需要依照測試樣板的相關規則來建立一個測試樣板，以完成你想要的測試項目。

測試樣板的附檔名是「.json」，格式如下。

```

[
    "xxxxxxxxxxxx",                                # 測試名稱
    {
        "description": "xxxxxxxxxxxx", # 測試內容的描述
        "prerequisite": [
            {
                "OFPFlowMod": {...} # 所要新增的 flow entry、meter entry、group
entry
            },                               # ( Ryu 的 OFPFlowMod、OFPMeterMod、
OFPGGroupMod 使用 json 的形態描述 )
            {
                "OFPMeterMod": {...} # 要將 flow entry 處理的結果轉送出去的情況下
                # (actions=output)
            },
            {
                "OFPGGroupMod": {...} # 若是封包轉送至 group entry 的情況
            },
            {
                "..." # 請指定輸出埠的編號為「2」或「3」
            }
        ],
        "tests": [
            {
                "# 產生封包
                # 單次產生封包或者一定時間內連續產生封包均可。
                # 封包的產生方法有 (A) (B) 兩種
                # (A) 單次產生封包
                "ingress": [
                    "ethernet(...)", # ( 在 Ryu 封包函式庫的建構子 ( Constructor )
中描述 )
                    "ipv4(...)",
                    "tcp(...)"
                ],
                "# (B) 一段時間內連續產生封包
                "ingress": {
                    "packets": {
                        "data": [
                            "ethernet(...)", # 與 (A) 相同
                            "ipv4(...)",
                            "tcp(...)"
                        ],

```

```

        "pktps": 1000,           # 每秒產生封包的數量 ( packet per
second )
        "duration_time": 30    # 連續產生封包的時間長度，以秒為單位。
    },
}

# 預期處理的結果
# 處理的結果有 (a) (b) (c) (d) 這幾種
# (a) 封包轉送 ( actions=output:X )
"egress": [
    "ethernet(...)",
    "ipv4(...)",
    "tcp(...)"
]
# (b) Packet in ( actions=CONTROLLER )
"PACKET_IN": [
    "ethernet(...)",
    "ipv4(...)",
    "tcp(...)"
]
# (c) table-miss
"table-miss": [
    0
]
# (d) 封包轉送 ( actions=output:X ) 時的流量 ( Throughput ) 測試
"egress": [
    "throughput": [
        {
            "OFPMatch": {           # 為了 Throughput 測試
                ...
            },                   # 新增在輔助交換器中
            "kbps": 1000          # 指定期望的流量以 Kbps 為單位
        },
        {...},
        {...}
    ]
},
{...},
{...}
]
},
{...},           # 測試項目 1
{...},           # 測試項目 2
{...}            # 測試項目 3
]

```

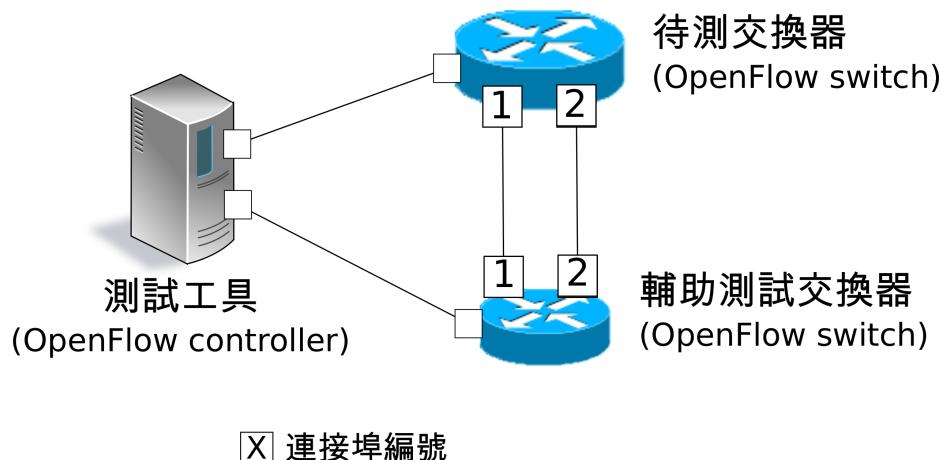
例如，產生封包中「(b) 一段時間內產生封包」和預期處理結果中「(d) 封包轉送 (actions=output:X) 時流量測試」搭配時就可以用來對待測交換器進行流量 (Throughput) 的測試。

註解：作為一個測試樣板在 Ryu 的原始碼中，提供了一些樣板檔案來檢查測試參數是否符合 OpenFlow1.3 FlowMod 中的 match / action 訊息。

ryu/tests/switch/of13

12.2.2 執行測試的環境

接下來說明測試工具執行時所需的環境。



對於做為一個輔助交換器來說，下面的條件是一個 OpenFlow 交換器必須要支援的。

- actions = CONTROLLER Flow Entry 新增
- 流量監控用的 Flow Entry 新增
- 透過 Flow Entry 發送 Packet-In 訊息到 Controller (actions = CONTROLLER)。
- 接受 Packet-Out 訊息並發送封包

註解：Ryu 原始碼當中利用腳本實作了一個在 mininet 上的測試環境，當中是採用 Open vSwitch 做為待測交換器。

`ryu/tests/switch/run_mininet.py`

腳本的使用範例請參照測試工具使用範例。

12.2.3 執行測試工具的方法

測試工具已經被公開在 Ryu 的原始碼當中。

原始碼	說明
<code>ryu/tests/switch/tester.py</code>	測試工具
<code>ryu/tests/switch/of13</code>	測試樣板的一些範例
<code>ryu/tests/switch/run_mininet.py</code>	建立測試環境的腳本

使用下面的指令來執行測試工具。

```
$ ryu-manager [--test-switch-target DPID] [--test-switch-tester DPID]
[--test-switch-dir DIRECTORY] ryu/tests/switch/tester.py
```

選項	說明	預設值
<code>--test-switch-target</code>	待測交換器的 datapath ID	00000000000000000001
<code>--test-switch-tester</code>	輔助交換器的 datapath ID	00000000000000000002
<code>--test-switch-dir</code>	測試樣板的存放路徑	<code>ryu/tests/switch/of13</code>

註解：測試工具是繼承自 `ryu.base.app_manager.RyuApp` 的一個應用程式。跟其他的 Ryu 應用程式一樣使用 `--verbose` 選項顯示除錯的訊息。

測試工具啟動之後，待測交換器和輔助交換器會跟 Controller 進行連接，接著測試動作就會使用指定的測試樣板開始進行測試。

12.3 測試工具使用範例

下面介紹如何使用和測試樣板檔案和原始測試樣板檔案的步驟。

12.3.1 執行測試樣板檔案的步驟

使用 Ryu 的原始碼中測試樣板範本（ryu/tests/switch/of13）來檢查 FlowMod 訊息的 match / action，MeterMod 的訊息和 GroupMod 訊息。

本程序中測試環境和測試環境的產生腳本（ryu/tests/switch/run_mininet.py），也因此測試目標是 Open vSwitch。使用 VM image 來打造測試環境以及登入的方法請參照「[交換器（Switching Hub）](#)」以取得更詳細的資料。

1. 建構測試環境

VM 環境的登入，執行測試環境的建構腳本。

```
ryu@ryu-vm:~$ sudo ryu/ryu/tests/switch/run_mininet.py
```

net 命令的執行結果如下。

```
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. 執行測試工具

為了執行測試工具，打開連線到 Controller 的 xterm。

```
mininet> xterm c0
```

在「Node: c0 (root)」的 xterm 中啟動測試工具。這時候做為測試樣板檔案的位置，請指定測試樣板範例路徑（ryu/tests/switch/of13）。接著，由於 mininet 測試環境中待測交換器和輔助交換器的 datapath ID 均有預設值，因此 --test-switch-target / --test-switch-tester 選項可省略。

Node: c0:

```
root@ryu-vm:~$ ryu-manager --test-switch-dir ryu/ryu/tests/switch/
of13 ryu/ryu/tests/switch/tester.py
```

測試工具執行之後就會出現下列訊息，並等待待測交換器和輔助交換器連結到 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...
```

待測交換器和輔助交換器連接 Controller 完成，測試開始。

```
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 資料夾以下的測試樣板全部執行完畢，測試也隨之結束。

< 參考資料 >

測試樣板範本檔案一覽

提供測試樣板範本檔案包括，對應 match / actions 各種設定的 Flow Entry 新增：match (或不 match) 多數 pattern 的封包改寫、對應滿足一定頻率的後變更優先權的 Meter Entry 新增：Meter Entry 中 match 的封包連續改寫、對應全連接埠的 FLOODING 的 Group Entry 新增：Group Entry 中 match 封包的連續改寫。

```
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
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 原始測試樣板的執行步驟

接著，原始的測試樣板製作並執行測試工具的步驟如下所示。

例如 OpenFlow 交換器若要實作路由器的功能，match / actions 的處理功能是必須的，因此我們製作測試樣板來確認他。

1 · 製作測試樣板檔案

透過路由器的路由表（Routing table）實作封包的轉送功能。下面的 Flow Entry 會確認整個動作是否正確。

match IP 網域 「192.168.30.0/24」	actions 修改發送端 MAC 位址為 「aa:aa:aa:aa:aa:aa」 修改目的端 MAC 位址為 「bb:bb:bb:bb:bb:bb」 降低 TTL 值 封包轉送
match IP 網域 「192.168.30.0/24」 ryu/tests/switch/of13 ryu/tests/switch/run_mininet.py	actions 修改發送端 MAC 位址為 「aa:aa:aa:aa:aa:aa」 測試樣版的一些範例 建立測試環境的腳本

依照這個測試樣板產生測試樣板檔案。

檔案名稱：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": [
        {
          "OFPIInstructionActions": {
            "actions": [
              {
                "OFPActionSetField": {
                  "field": {
                    "OXMTlv": {
                      "field": "eth_src",
                      "value": "aa:aa:aa:aa:aa:aa"
                    }
                  }
                }
              },
              {
                "OFPActionSetField": {
                  "field": {
                    "OXMTlv": {
                      "field": "eth_dst",
                      "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 · 建構測試環境

使用測試環境建置腳本來完成測試環境。詳細的操作細節請參照[執行測試樣板檔案的步驟](#)。

3 · 執行測試工具

使用 Controller 的 xterm 視窗，指定先前做好的測試樣板檔案位置並執行測試工具。可以使用 `--test-switch-dir` 選項來指定樣板檔案的位置。如果想要確認收送封包的內容，可以指定 `--verbose` 選項。

Node: c0:

```
root@ryu-vm:~$ ryu-manager --verbose --test-switch-dir ./sample_test_pattern.json ryu/ryu/tests/switch/tester.py
```

待測交換器和輔助交換器已經和 Controller 連接的情況下，測試即將開始。

「dpid=0000000000000002 : receive_packet...」的訊息在記錄檔中，表示測試樣板檔案的 egress 封包已經設定完成，即將送出預期的封包。然後，我們截取部分測試工具的輸出記錄檔。

```
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 ---
```

下面列出實際的 OpenFlow 交換器所登錄的 Flow Entry。你可以看到測試工具所產生的封包 match 所登錄的 Flow Entry，而且 n_packets 計數器數字被增加。

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->eth_src
, set_field:bb:bb:bb:bb:bb->eth_dst, dec_ttl, output:2
```

12.3.3 錯誤訊息一覽表

下面列出所有測試工具可能會顯示的錯誤訊息。

錯誤訊息	說明
Failed to initialize flow tables: barrier request timeout.	初始待測交換器的 flow entry 失敗 (Barrier Request 作業逾時)
Failed to initialize flow tables: [err_msg]	初始待測交換器的 flow entry 失敗 (接收到 FlowMod 錯誤訊息)
Failed to initialize flow tables of tester_sw: barrier request timeout.	初始輔助交換器的 flow entry 失敗 (Barrier Request 作業逾時)
Failed to initialize flow tables of tester_sw: [err_msg]	初始輔助交換器的 flow entry 失敗 (接收到 FlowMod 錯誤訊息)
Failed to add flows: barrier request timeout.	待測交換器的 flow entry 新增失敗 (Barrier Request 作業逾時)
Failed to add flows: [err_msg]	待測交換器的 flow entry 新增失敗 (接收到 FlowMod 錯誤訊息)
Failed to add flows to tester_sw: barrier request timeout.	輔助交換器的 flow entry 新增失敗 (Barrier Request 作業逾時)
Failed to add flows to tester_sw: [err_msg]	輔助交換器的 flow entry 新增失敗 (接收到 FlowMod 錯誤訊息)
Failed to add meters: barrier request timeout.	待測交換器的 meter entry 新增失敗 (Barrier Request 作業逾時)
Failed to add meters: [err_msg]	待測交換器的 meter entry 新增失敗 (接收到 MeterMod 錯誤訊息)
Failed to add groups: barrier request timeout.	待測交換器的 group entry 新增失敗 (Barrier Request 作業逾時)
Failed to add groups: [err_msg]	待測交換器的 group entry 新增失敗 (接收到 GroupMod 錯誤訊息)
Added incorrect flows: [flows]	待測交換器的 flow entry 新增失敗 (新增的 flow entry 不符合規範)
Failed to add flows: flow stats request timeout.	待測交換器的 flow entry 新增失敗 (FlowStats Request 作業逾時)
Failed to add flows: [err_msg]	待測交換器的 flow entry 新增失敗 (接收到 FlowStats Request 的錯誤訊息)
Added incorrect meters: [meters]	待測交換器的 meter entry 新增錯誤 (新增的 meter entry 不符合規範)
Failed to add meters: meter config stats request timeout.	待測交換器的 meter entry 新增失敗 (MeterConfigStats Request 作業逾時)
Failed to add meters: [err_msg]	待測交換器的 meter entry 新增失敗 (接收到 MeterConfigStats Request 錯誤訊息)
Added incorrect groups: [groups]	待測交換器的 group entry 新增錯誤 (新增的 group entry 不符合規範)
Failed to add groups: group desc stats request timeout.	待測交換器的 group entry 新增失敗 (GroupDescStats Request 作業逾時)
Failed to add groups: [err_msg]	待測交換器的 group entry 新增失敗 (接收到 GroupDescStats Request 錯誤訊息)
Failed to request port stats from target: request timeout.	待測交換器的 PortStats 取得失敗 (PortStats Request 作業逾時)
Failed to request port stats from target: [err_msg]	待測交換器的 PortStats 取得失敗

Continued on next page

Table 12.1 – continued from previous page

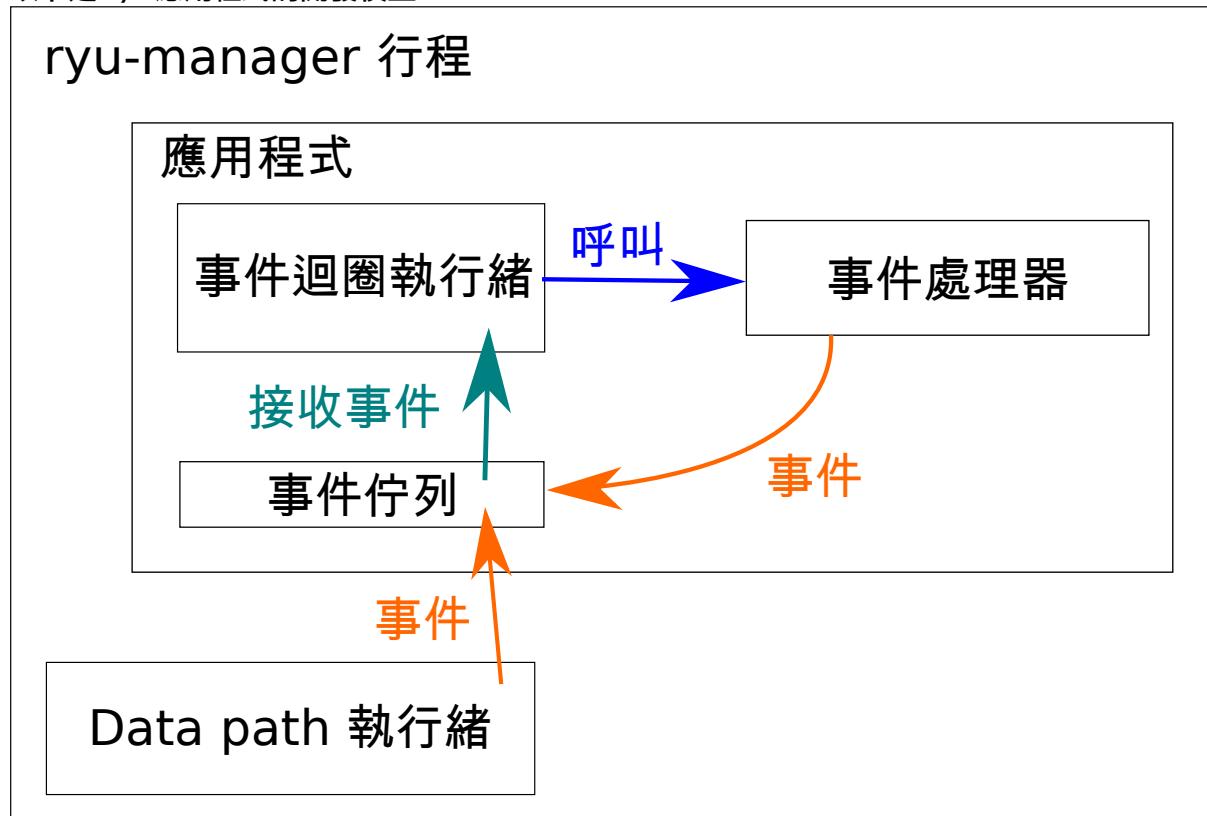
錯誤訊息	說明
Failed to request port stats from tester: request timeout.	(接受到 PortStats Request 的錯誤訊息) 輔助交換器的 PortStats 取得失敗 (PortStats Request 作業逾時)
Failed to request port stats from tester: [err_msg]	輔助交換器的 PortStats 取得失敗 (接受到 PortStats Request 的錯誤訊息)
Received incorrect [packet]	封包接收錯誤 (接受到錯誤的封包)
Receiving timeout: [detail]	封包接收錯誤 (作業逾時)
Faild to send packet: barrier request timeout.	封包傳送失敗 (Barrier Request 作業逾時)
Faild to send packet: [err_msg]	封包傳送失敗 (Packet-Out 的錯誤訊息)
Table-miss error: increment in matched_count.	table-miss 錯誤 (match flow)
Table-miss error: no change in lookup_count.	table-miss 錯誤 (封包不會被 flow table 所處理)
Failed to request table stats: request timeout.	table-miss 失敗 (TableStats Request 作業逾時)
Failed to request table stats: [err_msg]	table-miss 失敗 (接收到 TableStats Request 的錯誤訊息)
Added incorrect flows to tester_sw: [flows]	輔助交換器 flow entry 新增錯誤 (新增的 flow entry 不符合規範)
Failed to add flows to tester_sw: flow stats request timeout.	輔助交換器 flow entry 新增失敗 (FlowStats Request 作業逾時)
Failed to add flows to tester_sw: [err_msg]	輔助交換器 flow entry 新增失敗 (FlowStats Request 的錯誤訊息)
Failed to request flow stats: request timeout.	測試 Throughput 時，輔助交換器 flow entry request 失敗 (FlowStats Request 作業逾時)
Failed to request flow stats: [err_msg]	測試 Throughput 時，輔助交換器 flow entry request 失敗 (FlowStats Request 的錯誤訊息)
Received unexpected throughput: [detail]	測試 Throughput 時，得到非預期的結果
Disconnected from switch	待測交換器或輔助交換器的連結中斷

組織架構

本章介紹 Ryu 的架構。關於每一個 Class 請參照 API 參考資料 來得到更細節的資料。

13.1 應用程式開發模型 (Application programming model)

以下是 Ryu 應用程式的開發模型



13.1.1 應用程式 (Application)

應用程式是繼承 `ryu.base.app_manager.RyuApp` 而來。User logic 被視作是一個應用程式。

13.1.2 事件 (Event)

事件是繼承 `ryu.controller.event.EventBase` 而來，並藉由 `Transmitting` 和 `receiving event` 來相互溝通訊息。

13.1.3 事件佇列 (Event queue)

每個應用程式都有一個自己的佇列用來接受事件訊息。

13.1.4 執行緒 (Thread)

Ryu 採用 `eventlet` 來實現多執行緒。因為執行緒是不可插斷的 (non-preemptive)，因此在使用上要特別注意長時間運行所帶來的風險。

事件迴圈 (Event loop)

當應用程式執行時，將會有一個執行緒自動被產生用來執行該應用程式。該執行緒將會做為事件迴圈的模式來執行。如果在事件佇列中發現有事件存在，該事件迴圈將會讀取該事件並且呼叫相對應的事件處理器來處理它。

額外的執行緒 (Additional thread)

如果需要的話，你可以使用 `hub.spawn` 產生額外的執行緒用來執行特殊的應用程式功能。

eventlet

雖然你可以直接使用 `eventlet` 所提供的所有功能，但不建議你這麼做。請使用 `hub module` 所包裝過的功能取代直接使用 `eventlet`。

13.1.5 事件處理器 (Event handler)

藉由使用 `ryu.controller.handler.set_ev_cls` 裝飾器類別來定義自己的事件管理器。當定義的事件發生時，應用程式中的事件迴圈將會偵測到並呼叫對應的事件管理器。

協助專案開發

開放原始碼最大的優點在於對開發專案的共同參與，本章將介紹如何參與 Ryu 專案的開發。

14.1 參與專案

Mailing list 是 Ryu 專案開發者主要聚集的地方。因此首先就是要加入 Mailing list 來得到最新的消息。

<https://lists.sourceforge.net/lists/listinfo/ryu-devel>

在 Mailing list 中主要使用英文來進行溝通。當你在使用 Ryu 上遇到了困難，包括不知道如何使用或發現了程式上的錯誤。任何時候都歡迎在 Mailing list 上提出你的疑問。千萬不需要對於遇到了麻煩而感到不好意思結果不敢發言。因為使用 Open source 本身並提出疑問對於專案而言就是一種貢獻。

14.2 開發環境

接下來說明 Ryu 專案在開發的環境上需要注意的事情。

14.2.1 Python 版本

目前 Ryu 專案支援 Python 2.6 以上的版本，並請勿使用比 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援。但請小心撰寫程式碼以保持擴充性並隨時注意將來改版的可能。

14.2.2 程式碼撰寫風格（Coding Style）

Ryu 的原始碼是遵守 PEP8 的規範來開發。當你在送交程式碼時，請務必確認程式碼符合 PEP8 的規範。

<http://www.python.org/dev/peps/pep-0008/>

為了讓開發者更好的檢查自己的程式碼是否符合 PEP8 規範，這邊有個檢查器可以幫助開發者，請參考下述連結。

<https://pypi.python.org/pypi/pep8>

14.2.3 測試

Ryu 專案本身提供自動化測試的工具，最容易也是最常使用的是單元測試（Unit test），該功能也同樣包含在 Ryu 專案內。在送出程式碼之前，務必確認通過所有的單元測試，確保您的修改不會對現有的程式碼造成影響。如果是增加的程式碼，請在單元測試及註解的地方詳細說明該功能。

```
$ cd ryu/  
$ ./run_tests.sh
```

14.3 送交更新的程式碼

因為新功能的增加、程式錯誤的修復而需要對原始碼進行修改時，請針對修改的部分製作更新檔並寄送到 Mailing list。我們可以先在 Mailing list 中進行討論是否更新的必要。

註解： Ryu 的原始碼目前存放在 GitHub 上進行託管。但請注意 Pull request 並不是開發程序的必要條件。

在更新檔的格式上，我們期望收到如同更新 Linux Kernel 相同或類似的格式。下面的例子是用來說明送交程式更新檔時所需要符合的格式。請參閱相關的文件以達到該要求。

<http://lxr.linux.no/linux/Documentation/SubmittingPatches>

接下來說明整個流程。

1. 原始碼 Check out

首先是從 Ryu 專案下載原始碼。你可以在 Github 上 fork 一份專案到自己的帳號。下面的例子說明如何複製一份專案到自己的工作環境中。

```
$ git clone https://github.com/osrg/ryu.git  
$ cd ryu/
```

2. 原始碼的修改及增加

對 Ryu 的原始碼進行必要的修改。接著將變更的內容進行 Commit。

```
$ git commit -a ``
```

3. 更新檔的製作

對於變更的地方製作更新檔。不要忘記加入 Signed-off-by: 在即將送出的更新檔中。這個署名是用來辨認你對開放原始碼的貢獻，以及滿足相關的授權所使用。

```
$ git format-patch origin -s
```

4. 送出更新檔

已經完成的更新檔，在確認過沒有問題之後，請寄送到 Mailing list。你可以使用自己熟悉的郵件軟體或寄送方法，或者也可以使用 git 內建的寄送郵件指令。

```
$ git send-email 0001-sample.patch
```

5. 等待回應

等待開發團隊的討論以及對於更新檔的回應。原則上整個流程已經結束，若是你的更新檔有任何問題，你需要針對提問進行說明，必要時再次修改並送交更新檔。

應用案例

本章介紹幾個使用 Ryu 作為產品或服務的案例。

15.1 Stratosphere SDN Platform (Stratosphere)

Stratosphere SDN Platform (以下簡稱 SSP) 是 Stratosphere 公司所開發的軟體。經由 SSP 可以建構 Edge Overlay-model 的虛擬網路，它使用到的 Tunnelling 技術有：VXLAN、STT 和 MPLS。

每一種通道協定可在 VLAN 間相互轉換。因為每一種 Tunnelling 協定的 ID 通常都大於 VLAN 的 12 bits，所以使用 L2 segments 來管理而不直接使用 VLAN。而且，SSP 可以和其他軟體如：OpenStack、CloudStack 或 IaaS 協同運作。

在 1.1.4 的版本中，SSP 使用 OpenFlow 實作它的功能並嵌入 Ryu 做為它的 Controller。其中一個理由是為了支援 OpenFlow 1.1 之後的版本。為了在 SSP 之上支援 MPLS，考慮導入已支援 OpenFlow 1.1 的框架在協定層。

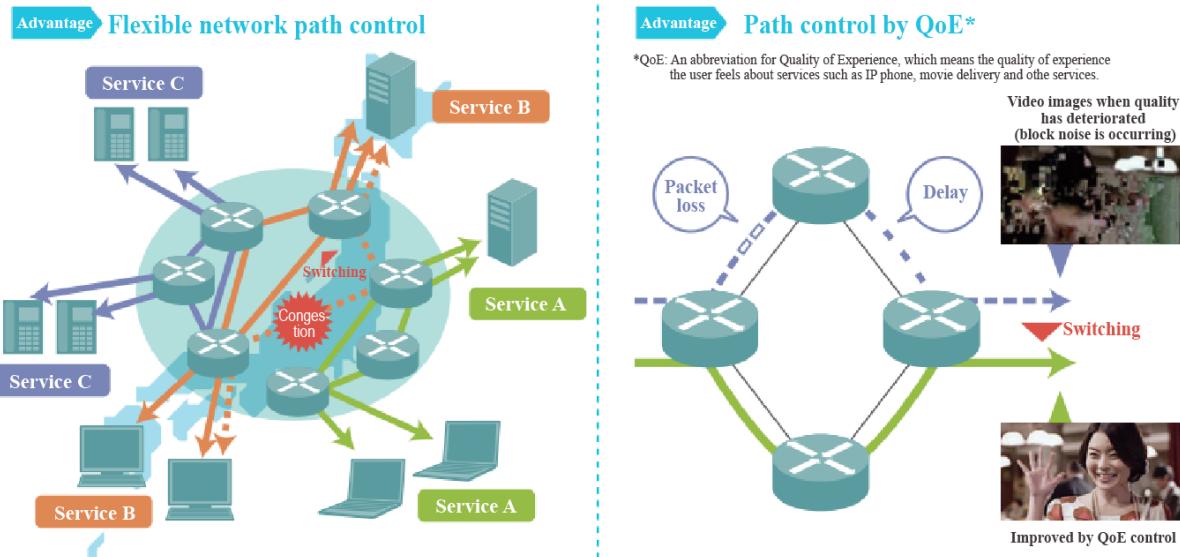
註解：OpenFlow 協定本身支援的程度也是一個非常重要的考慮點。由於規範內有許多列為選項的功能，因此必須注意交換器對於該功能的支援程度是否足夠。

開發語言 Python 也是一個很重要的原因。Stratosphere 所用的開發語言就是 Python，許多 SSP 的元件也都是使用 Python 所撰寫。Python 本身的自我描述能力很高，對於習慣使用的開發者來說，可以大幅提升開發的效率。

軟體是由多個 Ryu 應用程式所組成，並透過 REST API 與其他的 SSP 元件溝通。將軟體透過切割功能的方式分為多個應用程式，最基本的方法就是保持良好的原始碼。

15.2 SmartSDN Controller (NTT COMWARE)

「SmartSDN Controller」是一個提供集中管理功能（網路虛擬化/最佳化）的 SDN Controller。



「SmartSDN Controller」有以下兩個特徵。

1. 擁有彈性的虛擬網路路由管理

在相同的實體網路中建構多個虛擬網路，可以提供一個有彈性的環境以回應使用者對於變動性的要求，設備則可以被有效的利用以降低購買成本。而且每一台裝置、交換器、路由器的設定均集中管理之後，可以清楚的知道整體網路目前的狀態。當網路發生故障或者通訊狀況發生改變時，可以做相對應的變更處置。

藉由注重使用者的客戶體驗品質（「QoE」：Quality of Experience）、網路通訊品質（頻寬、延遲、封包丟失、流量變化），判斷客戶體驗品質後選擇較好的路由以達到維持穩定的服務。

2. 確保網路的高度彈性及可用性

為了在 Controller 發生故障的時候服務可以持續提供，備援的設定是必要的。主動產生封包並在節點間通訊，得以早期發現一般的監控無法及時發現的網路問題，並進行各種檢驗（路由測試、線路測試... 等）。

另外經由網路設計和狀態確認的視覺化圖形界面（GUI），讓即使沒有專業技能的操作人員都得以進行控制，降低網路運用的成本。

因此在「SmartSDN Controller」的開發上所選定的框架必須滿足下列的條件。

- 框架必須盡可能的支援 OpenFlow 規範的定義
- 框架必須隨時更新以追上持續更新的 OpenFlow 版本

在這之中 Ryu 是：

- 全面的支援各種 OpenFlow 版本
- 最快的跟上 OpenFlow 所發佈的新版本。而且社群相當活躍快速回應及解決臭蟲。
- sample code / 文件均相當豐富。

由於以上的特徵，所以是相當適合採用的開發框架。