
Ryu による OpenFlow プログラミング

リリース 0.1

Ryu プロジェクト

2014 年 02 月 12 日

目次

はじめに	1
第 1 章 スイッチングハブ	3
1.1 スイッチングハブ	3
1.2 OpenFlow によるスイッチングハブ	3
1.3 Ryu によるスイッチングハブの実装	6
1.4 Ryu アプリケーションの実行	17
1.5 まとめ	23
第 2 章 トラフィックモニター	25
2.1 ネットワークの定期健診	25
2.2 トラフィックモニターの実装	25
2.3 トラフィックモニターの実行	32
2.4 まとめ	34
第 3 章 REST 連携	35
3.1 REST API の組み込み	35
3.2 REST API 付きスイッチングハブの実装	35
3.3 SimpleSwitchRest13 クラスの実装	38
3.4 SimpleSwitchController クラスの実装	39
3.5 REST API 搭載スイッチングハブの実行	41
3.6 まとめ	43
第 4 章 リンク・アグリゲーション	45
4.1 リンク・アグリゲーション	45
4.2 Ryu アプリケーションの実行	46
4.3 Ryu によるリンク・アグリゲーション機能の実装	59
4.4 まとめ	69
第 5 章 スパニングツリー	71
5.1 スパニングツリー	71
5.2 Ryu アプリケーションの実行	73
5.3 OpenFlow によるスパニングツリー	86
5.4 Ryu によるスパニングツリーの実装	87
5.5 まとめ	98

第 6 章 OpenFlow プロトコル	99
6.1 マッチ	99
6.2 インストラクション	100
6.3 アクション	101
第 7 章 ofproto ライブラリ	103
7.1 概要	103
7.2 モジュール構成	103
7.3 基本的な使い方	104
第 8 章 パケットライブラリ	107
8.1 基本的な使い方	107
8.2 アプリケーション例	110
第 9 章 OF-Config ライブラリ	115
9.1 OF-Config プロトコル	115
9.2 ライブラリ構成	115
9.3 使用例	116
第 10 章 ファイアウォール	119
10.1 シングルテナントでの動作例	119
10.2 マルチテナントでの動作例	129
10.3 REST API 一覧	134
第 11 章 ルータ	137
11.1 シングルテナントでの動作例	137
11.2 マルチテナントでの動作例	149
11.3 REST API 一覧	163
第 12 章 OpenFlow スイッチテストツール	167
12.1 テストツールの概要	167
12.2 使用方法	168
12.3 テストツール使用例	171
第 13 章 アーキテクチャ	179
13.1 アプリケーションプログラミングモデル	179
第 14 章 コントリビューション	181
14.1 開発体制	181
14.2 開発環境	181
14.3 パッチを送る	182
第 15 章 導入事例	185
15.1 Stratosphere SDN Platform (ストラトスフィア)	185
15.2 SmartSDN Controller (NTT コムウェア)	186

はじめに

第1章～第5章では、スイッチングハブを例題に、トラヒックモニターやリンクアグリゲーションの機能拡張を通して、Ryu を使ったプログラミング手法をご紹介します。

第6章～第8章では、Ryu を使ったプログラミングで必要となる、OpenFlow プロトコルやパケットライブラリのご紹介をします。

第9章～第11章では、Ryu にサンプルアプリケーションとして同梱されている、ファイアウォールやテストツールの利用方法をご紹介します。

第12章～第14章では、Ryu のアーキテクチャや導入事例についてご紹介します。

第1章

スイッチングハブ

本章では、簡単なスイッチングハブの実装を題材として、Ryuによるアプリケーションの実装方法を解説していきます。

1.1 スイッチングハブ

世の中には様々な機能を持つスイッチングハブがありますが、ここでは次のような単純な機能を持ったスイッチングハブの実装を見てみます。

- ・ポートに接続されているホストの MAC アドレスを学習し、MAC アドレステーブルに保持する
- ・学習済みのホスト宛のパケットを受信したら、ホストの接続されているポートに転送する
- ・未知のホスト宛のパケットを受信したら、フラッディングする

このようなスイッチを Ryu を使って実現してみましょう。

1.2 OpenFlow によるスイッチングハブ

OpenFlow スイッチは、Ryu の様な OpenFlow コントローラからの指示を受けて、次のようなことができます。

- ・受信したパケットのアドレスを書き換えたり、指定のポートから転送
- ・受信したパケットをコントローラへ転送 (Packet-In)
- ・コントローラから転送されたパケットを指定のポートから転送 (Packet-Out)

これらの機能を組み合わせ、スイッチングハブを実現することができます。

まずは、Packet-In の機能を利用した MAC アドレスの学習です。コントローラは、Packet-In の機能を利用し、スイッチからパケットを受け取る事が出来ます。受け取ったパケットを解析し、ホストの MAC アドレスや接続されているポートの情報を学習することができます。

学習の後は受信したパケットの転送です。パケットの宛先 MAC アドレスが学習済みのホストのものか検索します。検索結果によって次の処理を実行します。

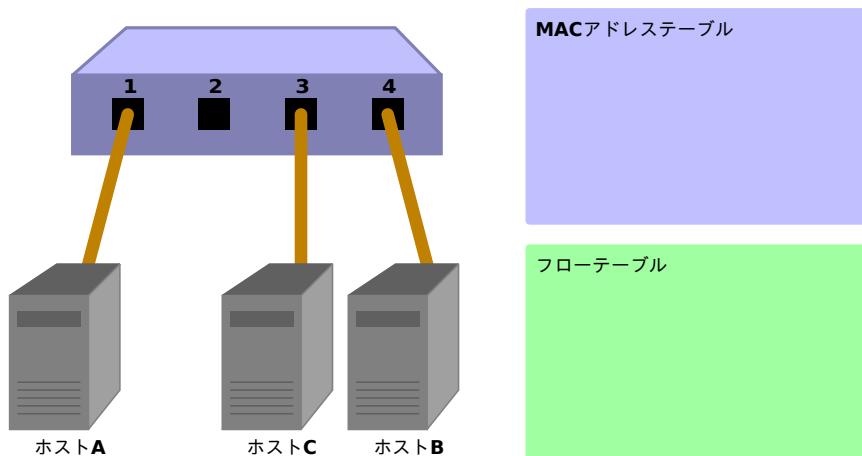
- 学習済みのホストの場合...Packet-Out の機能で、接続先のポートからパケットを転送
- 未知のホストの場合...Packet-Out の機能でパケットをフラッディング

これらの動作を順を追って図とともに説明します。

1. 初期状態

フローテーブルが空の初期状態です。

ポート 1 にホスト A、ポート 4 にホスト B、ポート 3 にホスト C が接続されているものとします。



2. ホスト A → ホスト B

ホスト A からホスト B へのパケットが送信されると、Packet-In メッセージが送られ、ホスト A の MAC アドレスがポート 1 に学習されます。ホスト B のポートはまだ分かっていないため、パケットはフラッディングされ、パケットはホスト B とホスト C で受信されます。



Packet-In:

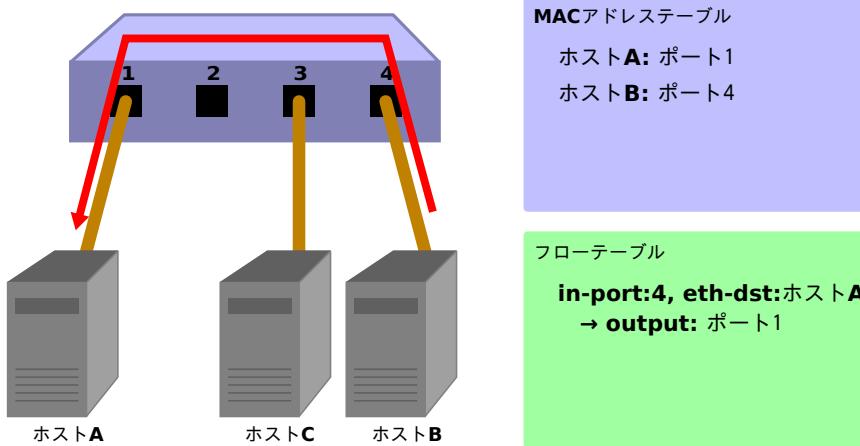
```
in-port: 1
eth-dst: ホスト B
eth-src: ホスト A
```

Packet-Out:

```
action: OUTPUT: フラッディング
```

3. ホスト B → ホスト A

ホスト B からホスト A にパケットが返されると、フローテーブルにエントリを追加し、またパケットはポート 1 に転送されます。そのため、このパケットはホスト C では受信されません。



Packet-In:

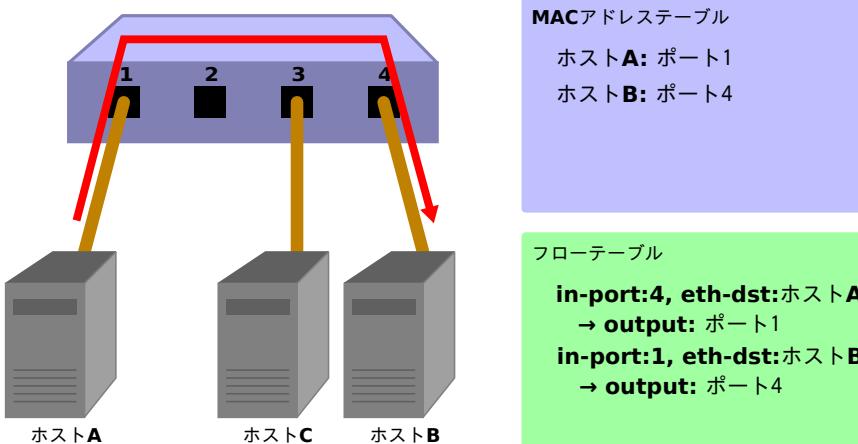
```
in-port: 4
eth-dst: ホスト A
eth-src: ホスト B
```

Packet-Out:

```
action: OUTPUT: ポート 1
```

4. ホスト A → ホスト B

再度、ホスト A からホスト B へのパケットが送信されると、フローテーブルにエントリを追加し、またパケットはポート 4 に転送されます。



Packet-In:

```
in-port: 1
eth-dst: ホスト B
eth-src: ホスト A
```

Packet-Out:

```
action: OUTPUT:ポート 4
```

次に、実際に Ryu を使って実装されたスイッチングハブのソースコードを見ていきます。

1.3 Ryuによるスイッチングハブの実装

スイッチングハブのソースコードは、Ryu のソースツリーにあります。

```
ryu/app/simple_switch_13.py
```

OpenFlow のバージョンに応じて、他にも simple_switch.py(OpenFlow 1.0)、simple_switch_12.py(OpenFlow 1.2) がありますが、ここでは OpenFlow 1.3 に対応した実装を見ていきます。

短いソースコードなので、全体をここに掲載します。

```
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)
```

それでは、それぞれの実装内容について見ていきます。

1.3.1 クラスの定義と初期化

Ryu アプリケーションとして実装するため、ryu.base.app_manager.RyuApp を継承します。また、OpenFlow 1.3 を使用するため、OFP_VERSIONS に OpenFlow 1.3 のバージョンを指定しています。

また、MAC アドレステーブル mac_to_port を定義しています。

OpenFlow プロトコルでは、OpenFlow スイッチとコントローラが通信を行うために必要となるハンドシェイクなどのいくつかの手順が決められていますが、Ryu のフレームワークが処理してくれるため、Ryu アプリケーションでは意識する必要はありません。

```
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 イベントハンドラ

Ryu では、OpenFlow メッセージを受信するとメッセージに対応したイベントが発生します。Ryu アプリケーションは、受け取りたいメッセージに対応したイベントハンドラを実装します。

イベントハンドラは、引数にイベントオブジェクトを持つ関数を定義し、`ryu.controller.handler.set_ev_cls` デコレータで修飾します。

`set_ev_cls` は、引数に受け取るメッセージに対応したイベントクラスと OpenFlow スイッチのステートを指定します。

イベントクラス名は、`ryu.controller.ofp_event.EventOFP+<OpenFlow メッセージ名>`となっています。例えば、Packet-In メッセージの場合は、`EventOFPPacketIn` になります。詳しくは、Ryu のドキュメント *Ryu application API* を参照してください。ステートには、以下のいずれか、またはリストを指定します。

定義	説明
<code>ryu.controller.handler.HANDSHAKE_DISPATCHER</code>	HELLO メッセージの交換
<code>ryu.controller.handler.CONFIG_DISPATCHER</code>	SwitchFeatures メッセージの受信待ち
<code>ryu.controller.handler.MAIN_DISPATCHER</code>	通常状態
<code>ryu.controller.handler.DEAD_DISPATCHER</code>	コネクションの切断

Table-miss フローエントリの追加

OpenFlow スイッチとのハンドシェイク完了後に Table-miss フローエントリをフローテーブルに追加し、Packet-In メッセージを受信する準備を行います。

具体的には、Switch Features(Features Reply) メッセージを受け取り、そこで Table-miss フローエントリの追加を行います。

```
@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` には、イベントに対応する OpenFlow メッセージクラスのインスタンスが格納されています。この場合は、`ryu.ofproto.ofproto_v1_3_parser.OFPSwitchFeatures` になります。

`msg.datapath` には、このメッセージを発行した OpenFlow スイッチに対する ryu.controller.controller.Datapath クラスのインスタンスが格納されています。

Datapath クラスは、OpenFlow スイッチとの実際の通信処理や受信メッセージに対応したイベントの発行などの重要な処理を行っています。

Ryu アプリケーションで利用する主な属性は以下のものです。

属性名	説明
id	接続している OpenFlow スイッチの ID(データパス ID) です。
ofproto	使用している OpenFlow バージョンに対応した ofproto モジュールを示します。現時点では、以下のいずれかになります。 ryu.ofproto.ofproto_v1_0 ryu.ofproto.ofproto_v1_2 ryu.ofproto.ofproto_v1_3 ryu.ofproto.ofproto_v1_4
ofproto_parser	ofproto と同様に、ofproto_parser モジュールを示します。現時点では、以下のいずれかになります。 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 アプリケーションで利用する Datapath クラスの主なメソッドは以下のものです。

`send_msg(msg)`

OpenFlow メッセージを送信します。msg は、送信 OpenFlow メッセージに対応した `ryu.ofproto.ofproto_parser.MsgBase` のサブクラスです。

スイッチングハブでは、受信した Switch Features メッセージ自体は特に使いません。Table-miss フローエントリを追加するタイミングを得るためのイベントとして扱っています。

```
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 フローエントリは、優先度が最低 (0) で、すべてのパケットにマッチするエントリです。このエントリのインストラクションにコントローラポートへの出力アクションを指定することで、受信パケットが、すべての通常のフローエントリにマッチしなかった場合、Packet-In を発行するようになります。

ノート: 2014 年 1 月現在の Open vSwitch は、OpenFlow 1.3 への対応が不完全であり、OpenFlow 1.3 以前と同様にデフォルトで Packet-In が発行されます。また、Table-miss フローエントリにも現時点では未対応で、通常のフローエントリとして扱われます。

すべてのパケットにマッチさせるため、空のマッチを生成します。マッチは `OFPMatch` クラスで表されます。

次に、コントローラポートへ転送するための OUTPUT アクションクラス (`OFPActionOutput`) のインスタンスを生成します。出力先にコントローラ、パケット全体をコントローラに送信するために `max_len` には `OFCML_NO_BUFFER` を指定しています。

ノート: コントローラにはパケットの先頭部分 (Ethernet ヘッダー分) だけを送信させ、残りはスイッチにバッファーされた方が効率の点では望ましいのですが、Open vSwitch のバグ (2014 年 1 月現在) を回避するために、ここではパケット全体を送信させます。

最後に、優先度に 0(最低) を指定して `add_flow()` メソッドを実行して Flow Mod メッセージを送信します。`add_flow()` メソッドの内容については後述します。

Packet-in メッセージ

未知の宛先の受信パケットを受け付けるため、Packet-In イベントのハンドラを作成します。

```
@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` クラスのよく使われる属性には以下のようなものがあります。

属性名	説明
<code>match</code>	<code>ryu.ofproto.ofproto_v1_3_parser.OFPMatch</code> クラスのインスタンスで、受信パケットのメタ情報が設定されています。
<code>data</code>	受信パケット自体を示すバイナリデータです。
<code>total_len</code>	受信パケットのデータ長です。
<code>buffer_id</code>	受信パケットが OpenFlow スイッチ上でバッファされている場合、その ID が示されます。バッファされていない場合は、 <code>ryu.ofproto.ofproto_v1_3.OFP_NO_BUFFER</code> がセットされます。

MAC アドレステーブルの更新

```
def _packet_in_handler(self, ev):
    # ...

    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

# ...
```

OFPPacketIn クラスの match から、受信ポート (in_port) を取得します。宛先 MAC アドレスと送信元 MAC アドレスは、Ryu のパケットライブラリを使って、受信パケットの Ethernet ヘッダから取得しています。

取得した送信元 MAC アドレスと受信ポート番号で、MAC アドレステーブルを更新します。

複数の OpenFlow スイッチとの接続に対応するため、MAC アドレステーブルは OpenFlow スイッチ毎に管理するようになっています。OpenFlow スイッチの識別にはデータパス ID を用いています。

転送先ポートの判定

宛先 MAC アドレスが、MAC アドレステーブルに存在する場合は対応するポート番号を、見つからなかった場合はフラッディング (OFPP_FLOOD) を出力ポートに指定した OUTPUT アクションクラスのインスタンスを生成します。

```
def _packet_in_handler(self, ev):
    # ...

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

    actions = [parser.OFPActionOutput(out_port)]

    # install a flow to avoid packet_in next time
    if out_port != ofproto.OFPP_FLOOD:
        match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
        self.add_flow(datapath, 1, match, actions)

    # ...
```

宛先 MAC アドレスが見つかった場合は、OpenFlow スイッチのフローテーブルにエントリを追加します。

Table-miss フローエントリの追加と同様に、マッチとアクションを指定して add_flow() を実行し、フローエントリを追加します。

Table-miss フローエントリとは違って、今回はマッチに条件を設定します。今回のスイッチングハブの実装で

は、受信ポート (in_port) と宛先 MAC アドレス (eth_dst) を指定しています。例えば、「ポート 1 で受信したホスト B 宛」のパケットが対象となります。

今回のフローエントリでは、優先度に 1 を指定しています。値が大きいほど優先度が高くなるので、ここで追加するフローエントリは、Table-miss フローエントリより先に評価されるようになります。

前述のアクションを含めてまとめると、以下のようなエントリをフローテーブルに追加します。

ポート 1 で受信した、ホスト B 宛 (宛先 MAC アドレスが B) のパケットを、ポート 4 に転送する

ヒント: OpenFlow では、NORMAL ポートという論理的な出力ポートがオプションで規定されており、出力ポートに NORMAL を指定すると、スイッチの L2/L3 機能を使ってパケットを処理するようになります。つまり、すべてのパケットを NORMAL ポートに出力するように指示するだけで、スイッチングハブとして動作するようにできますが、ここでは各々の処理を OpenFlow を使って実現するものとします。

フローエントリの追加処理

Packet-In ハンドラの処理がまだ終わっていませんが、ここで一旦フローエントリを追加するメソッドの方を見ていきます。

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

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

フローエントリには、対象となるパケットの条件を示すマッチと、そのパケットに対する操作を示すインストラクション、エントリの優先度、有効時間などを設定します。

スイッチングハブの実装では、インストラクションに Apply Actions を使用して、指定したアクションを直ちに適用するように設定しています。

最後に、Flow Mod メッセージを発行してフローテーブルにエントリを追加します。

```
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 メッセージに対応するクラスは OFPFlowMod クラスです。OFPFlowMod クラスのインスタンスを生成して、Datapath.send_msg() メソッドで OpenFlow スイッチにメッセージを送信します。

OFPFlowMod クラスのコンストラクタには多くの引数がありますが、多くのものは大抵の場合、デフォルト値のままで済みます。かっこ内はデフォルト値です。

datapath

フローテーブルを操作する対象となる OpenFlow スイッチに対応する Datapath クラスのインスタンスです。通常は、Packet-In メッセージなどのハンドラに渡されるイベントから取得したものを指定します。

cookie (0)

コントローラが指定する任意の値で、エントリの更新または削除を行う際のフィルタ条件として使用できます。パケットの処理では使用されません。

cookie_mask (0)

エントリの更新または削除の場合に、0 以外の値を指定すると、エントリの cookie 値による操作対象エントリのフィルタとして使用されます。

table_id (0)

操作対象のフローテーブルのテーブル ID を指定します。

command (ofproto_v1_3.OFPFC_ADD)

どのような操作を行うかを指定します。

値	説明
OFPFC_ADD	新しいフローエントリを追加します
OFPFC MODIFY	フローエントリを更新します
OFPFC MODIFY_STRICT	厳格に一致するフローエントリを更新します
OFPFC_DELETE	フローエントリを削除します
OFPFC_DELETE_STRICT	厳格に一致するフローエントリを削除します

idle_timeout (0)

このエントリの有効期限を秒単位で指定します。エントリが参照されずに idle_timeout で指定した時間を過ぎた場合、そのエントリは削除されます。エントリが参照されると経過時間はリセットされます。

エントリが削除されると Flow Removed メッセージがコントローラに通知されます。

hard_timeout (0)

このエントリの有効期限を秒単位で指定します。idle_timeout と違って、hard_timeout では、エントリが参照されても経過時間はリセットされません。つまり、エントリの参照の有無に関わらず、指定された時間が経過するとエントリが削除されます。

idle_timeout と同様に、エントリが削除されると Flow Removed メッセージが通知されます。

priority (0)

このエントリの優先度を指定します。値が大きいほど、優先度も高くなります。

buffer_id (ofproto_v1_3.OFP_NO_BUFFER)

OpenFlow スイッチ上でバッファされたパケットのバッファ ID を指定します。バッファ ID は Packet-In メッセージで通知されたものであり、指定すると OFPP_TABLE を出力ポートに指定した Packet-Out メッセージと Flow Mod メッセージの 2 つのメッセージを送ったのと同じように処理されます。command が OFPFC_DELETE または OFPFC_DELETE_STRICT の場合は無視されます。

バッファ ID を指定しない場合は、OFP_NO_BUFFER をセットします。

out_port (0)

OFPFC_DELETE または OFPFC_DELETE_STRICT の場合に、対象となるエントリを出力ポートでフィルタします。OFPFC_ADD、OFPFC MODIFY、OFPFC MODIFY_STRICT の場合は無視されます。

出力ポートでのフィルタを無効にするには、OFPP_ANY を指定します。

out_group (0)

out_port と同様に、出力グループでフィルタします。

無効にするには、OFPG_ANY を指定します。

flags (0)

以下のフラグの組み合わせを指定することができます。

値	説明
OFPFF_SEND_FLOW_Rem	このエントリが削除された時に、コントローラに FlowRemoved メッセージを発行します。
OFPFF_CHECK_OVERLAP	OFPFC_ADD の場合に、重複するエントリのチェックを行います。重複するエントリがあった場合には Flow Mod は失敗し、エラーが返されます。
OFPFF_RESET_COUNTS	該当エントリのパケットカウンタとバイトカウンタをリセットします。
OFPFF_NO_PKT_COUNTS	このエントリのパケットカウンタを無効にします。
OFPFF_NO_BYT_COUNTS	このエントリのバイトカウンタを無効にします。

match (None)

マッチを指定します。

instructions ([])

インストラクションのリストを指定します。

パケットの転送

Packet-In ハンドラに戻り、最後の処理の説明です。

宛先 MAC アドレスが MAC アドレステーブルから見つかったかどうかに関わらず、最終的には Packet-Out メッセージを発行して、受信パケットを転送します。

```
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 メッセージに対応するクラスは OFPPacketOut クラスです。

OFPPacketOut のコンストラクタの引数は以下のようになっています。

datapath

OpenFlow スイッチに対する Datapath クラスのインスタンスを指定します。

buffer_id

OpenFlow スイッチ上でバッファされたパケットのバッファ ID を指定します。バッファを使用しない場合は、OFP_NO_BUFFER を指定します。

in_port

パケットを受信したポートを指定します。受信パケットでない場合は、OFPP_CONTROLLER を指定します。

actions

アクションのリストを指定します。

data

パケットのバイナリデータを指定します。buffer_id に OFP_NO_BUFFER が指定された場合に使用されます。OpenFlow スイッチのバッファを利用する場合は省略します。

スイッチングハブの実装では、buffer_id に Packet-In メッセージの buffer_id を指定しています。Packet-In メッセージの buffer_id が無効だった場合は、Packet-In の受信パケットを data に指定して、パケットを送信しています。

これで、スイッチングハブのソースコードの説明は終わりです。次は、このスイッチングハブを実行して、実際の動作を確認します。

1.4 Ryu アプリケーションの実行

スイッチングハブの実行のため、OpenFlow スイッチには Open vSwitch、実行環境として mininet を使います。

Ryu 用の OpenFlow Tutorial VM イメージが用意されているので、この VM イメージを利用すると実験環境を簡単に準備することができます。

VM イメージ

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

OpenFlow_Tutorial_Ryu3.2.ova (約 1.4GB)

関連ドキュメント (Wiki ページ)

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

ドキュメントにある VM イメージは、Open vSwitch と Ryu のバージョンが古いためご注意ください。

この VM イメージを使わず、自分で環境を構築することも当然できます。VM イメージで使用している各ソフトウェアのバージョンは以下の通りですので、自分で構築する場合は参考にしてください。

Mininet VM バージョン 2.0.0 <http://mininet.org/download/>

Open vSwitch バージョン 1.11.0 <http://openvswitch.org/download/>

Ryu バージョン 3.2 <https://github.com/osrg/ryu/>

```
$ sudo pip install ryu
```

ここでは、Ryu 用 OpenFlow Tutorial の VM イメージを利用します。

1.4.1 Mininet の実行

mininet から xterm を起動するため、X が使える環境が必要です。

ここでは、OpenFlow Tutorial の VM を利用しているため、デスクトップ PC から ssh で X11 Forwarding を有効にしてログインします。

```
$ ssh -X ryu@<VM のアドレス>
```

ユーザー名は ryu、パスワードも ryu です。

ログインできたら、mn コマンドにより Mininet 環境を起動します。

構築する環境は、ホスト 3 台、スイッチ 1 台のシンプルな構成です。

mn コマンドのパラメータは、以下のようになります。

パラメータ	値	説明
topo	single,3	スイッチが1台、ホストが3台のトポロジ
mac	なし	自動的にホストのMACアドレスをセットする
switch	ovsk	Open vSwitchを使用する
controller	remote	OpenFlowコントローラは外部のものを利用する
x	なし	xtermを起動する

実行例は以下のようになります。

```
$ 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>
```

実行するとデスクトップPC上でxtermが5つ起動します。それぞれ、ホスト1~3、スイッチ、コントローラに対応します。

スイッチのxtermからコマンドを実行して、使用するOpenFlowのバージョンをセットします。ウインドウタイトルが「switch: s1 (root)」となっているものがスイッチ用のxtermです。

まずはOpen vSwitchの状態を見てみます。

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:~#
```

スイッチ(ブリッジ)s1 ができていて、ホストに対応するポートが 3 つ追加されています。

次に OpenFlow のバージョンとして 1.3 を設定します。

switch: s1:

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

空のフローテーブルを確認してみます。

switch: s1:

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

ovs-ofctl コマンドには、オプションで使用する OpenFlow のバージョンを指定する必要があります。デフォルトは *OpenFlow10* です。

1.4.2 スイッチングハブの実行

準備が整ったので、Ryu アプリケーションを実行します。

ウインドウタイトルが「controller: c0 (root)」となっている xterm から次のコマンドを実行します。

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 EventOPPErrorMsg
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=0)
move onto main mode
```

OVSとの接続に時間がかかる場合がありますが、少し待つと上のように

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

と表示されます。

これで、OVSと接続し、ハンドシェイクが行われ、Table-miss フローエントリが追加され、Packet-In を待っている状態になっています。

Table-miss フローエントリが追加されていることを確認します。

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:~#
```

優先度が 0 で、マッチがなく、アクションに CONTROLLER、送信データサイズ 65535(0xffff = OF-PCML_NO_BUFFER) が指定されています。

1.4.3 動作の確認

ホスト 1 からホスト 2 へ ping を実行します。

1. ARP request

この時点では、ホスト 1 はホスト 2 の MAC アドレスを知らないので、ICMP echorequest に先んじて ARP request をブロードキャストするはずです。このブロードキャストパケットはホスト 2 とホスト 3 で受信されます。

2. ARP reply

ホスト 2 が ARP に応答して、ホスト 1 に ARP reply を返します。

3. ICMP echo request

これでホスト 1 はホスト 2 の MAC アドレスを知ることができたので、echo request をホスト 2 に送信します。

4. ICMP echo reply

ホスト 2 はホスト 1 の MAC アドレスを既に知っているので、echo reply をホスト 1 に返します。

このような通信が行われるはずです。

ping コマンドを実行する前に、各ホストでどのようなパケットを受信したかを確認できるように tcpdump コマンドを実行しておきます。

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
```

それでは、最初に mn コマンドを実行したコンソールで、次のコマンドを実行してホスト 1 からホスト 2 へ ping を発行します。

```
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 は正常に返ってきました。

まずはフローテーブルを確認してみましょう。

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=00:00:00:00:00:00 actions=OUTPUT:1
  cookie=0x0, duration=48.402s, table=0, n_packets=1, n_bytes=42, priority=1,in_port=1,dl_dst=00:00:00:00:00:00 actions=OUTPUT:2
root@ryu-vm:~#
```

Table-miss フローエントリ以外に、優先度が 1 のフローエントリが 2 つ登録されています。

1. 受信ポート (in_port):2, 宛先 MAC アドレス (dl_dst):ホスト 1 → 動作 (actions):ポート 1 に転送
2. 受信ポート (in_port):1, 宛先 MAC アドレス (dl_dst):ホスト 2 → 動作 (actions):ポート 2 に転送

(1) のエントリは 2 回参照され (n_packets)、(2) のエントリは 1 回参照されています。(1) はホスト 2 からホスト 1 宛の通信なので、ARP reply と ICMP echo reply の 2 つがマッチしたものでしょう。(2) はホスト 1 からホスト 2 宛の通信で、ARP request はプロードキャストされるので、これは ICMP echo request によるもののはずです。

それでは、simple_switch_13 のログ出力を見てみます。

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
```

1 つ目の Packet-In は、ホスト 1 が発行した ARP request で、プロードキャストなのでフローエントリは登録されず、Packet-Out のみが発行されます。

2 つ目は、ホスト 2 から返された ARP reply で、宛先 MAC アドレスがホスト 1 となっているので前述のフローエントリ (1) が登録されます。

3 つ目は、ホスト 1 からホスト 2 へ送信された ICMP echo request で、フローエントリ (2) が登録されます。

ホスト 2 からホスト 1 に返された ICMP echo reply は、登録済みのフローエントリ (1) にマッチするため、Packet-In は発行されずにホスト 1 へ転送されます。

最後に各ホストで実行した tcpdump の出力を見てみます。

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 10.0.0.1? [ethertype ARP?]
20:38:04.678698 00:00:00:00:00:02 > 00:00:00:00:00:01, ethertype ARP (0x0806), length 42: Reply 10.0.0.1 to 00:00:00:00:00:01 [ethertype ARP?]
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 [tos 0x0]
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 [tos 0x0]
```

ホスト 1 では、最初に ARP request がブロードキャストされていて、続いてホスト 2 から返された ARP reply を受信しています。次にホスト 1 が発行した ICMP echo request、ホスト 2 から返された ICMP echo reply が受信されています。

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 10.0.0.1? [ethertype ARP?]
20:38:04.638059 00:00:00:00:00:02 > 00:00:00:00:00:01, ethertype ARP (0x0806), length 42: Reply 10.0.0.1 [ethertype ARP?]
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 [ethertype ICMP?]
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 [ethertype ICMP?]
```

ホスト 2 では、ホスト 1 が発行した ARP request を受信し、ホスト 1 に ARP reply を返しています。続いて、ホスト 1 からの ICMP echo request を受信し、ホスト 1 に echo reply を返しています。

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 10.0.0.1? [ethertype ARP?]
```

ホスト 3 では、最初にホスト 1 がブロードキャストした ARP request のみを受信しています。

1.5 まとめ

本章では、簡単なスイッチングハブの実装を題材に、Ryu アプリケーションの実装の基本的な手順と、OpenFlow による OpenFlow スイッチの簡単な制御方法について説明しました。

第2章

トラフィックモニター

本章では、「[スイッチングハブ](#)」で説明したスイッチングハブに、OpenFlow スイッチの統計情報をモニターする機能を追加します。

2.1 ネットワークの定期健診

ネットワークは既に多くのサービスや業務のインフラとなっているため、正常で安定した稼働が維持されることが求められます。とは言え、いつも何かしらの問題が発生するものです。

ネットワークに異常が発生した場合、迅速に原因を特定し、復旧させなければなりません。本書をお読みの方には言うまでもないことですが、異常を検出し、原因を特定するためには、日頃からネットワークの状態を把握しておく必要があります。例えば、あるネットワーク機器のポートのトラフィック量が非常に高い値を示していたとして、それが異常な状態なのか、いつもそうなのか、あるいはいつからそうになったのかということは、継続してそのポートのトラフィック量を測っていなければ判断することができません。

というわけで、ネットワークの健康状態を常に監視しつづけるということは、そのネットワークを使うサービスや業務の継続的な安定運用のためにも必須となります。もちろん、トラフィック情報の監視さえしていれば万全などということはありませんが、本章では OpenFlow によるスイッチの統計情報の取得方法について説明します。

2.2 トラフィックモニターの実装

早速ですが、「[スイッチングハブ](#)」で説明したスイッチングハブにトラフィックモニター機能を追加したソースコードです。

```
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.EventOFPSStateChange,
                [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.OFPPortStatsRequest(datapath, 0, ofproto.OFPP_ANY)
        datapath.send_msg(req)

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

        self.logger.info('datapath         '
                         'in-port  eth-dst           '
                         'out-port packets  bytes')
        self.logger.info('-----  '
                         '-----  -----  '
                         '-----  -----  ')
        for stat in sorted([flow for flow in body if flow.priority == 1],
                           key=lambda stat:
                           (stat.datapath.id, stat.port)):
            self.logger.info('%016x %02x %17s %06d %8d %8d',
                            stat.datapath.id, stat.port,
                            stat.eth_src, stat.eth_dst,
                            stat.out_port, stat.packets, stat.bytes)
```

```

        key=lambda flow: (flow.match['in_port'],
                           flow.match['eth_dst'])):
    self.logger.info('%016x %8x %17s %8x %8d %8d',
                     ev.msg.datapath.id,
                     stat.match['in_port'], stat.match['eth_dst'],
                     stat.instructions[0].actions[0].port,
                     stat.packet_count, stat.byte_count)

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

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

```

SimpleSwitch13 を継承した SimpleMonitor クラスに、トラフィックモニター機能を実装していますので、ここにはパケット転送に関する処理は出てきません。

2.2.1 定周期処理

スイッチングハブの処理と並行して、定期的に統計情報取得のリクエストを OpenFlow スイッチへ発行するために、スレッドを生成します。

```

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)
# ...

```

ryu.lib.hub には、いくつかの eventlet のラッパー や 基本的な クラス の 実装 が あります。ここではスレッドを生成する hub.spawn() を 使用 します。実際に生成されるスレッドは eventlet の グリーンスレッド です。

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

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

スレッド 関数 _monitor() では、登録されたスイッチに対する 統計情報取得リクエスト の 発行 を 10 秒 間隔 で 無限に 繰り返 します。

接続中のスイッチを 監視 対象 と する ため、スイッチの 接続 および 切断 の 検出 に EventOFPStateChange イベント を 利用 しています。この イベント は Ryu フレームワーク が 発行 する もの で、Datapath の ステート が 変わった とき に 発行 されます。

ここでは、Datapath の ステート が MAIN_DISPATCHER になつた 時 に そのスイッチを 監視 対象 に 登録 、 DEAD_DISPATCHER になつた 時 に 登録 の 削除 を 行つ て い ます。

```
# ...
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)
# ...
```

定期的に呼び出される _request_stats() では、スイッチに OFPFlowStatsRequest と OFPPortStatsRequest を 発行 して い ます。

OFPFlowStatsRequest は、 フローエントリ に関する 統計情報 を スイッチ に 要求 し ます。テーブル ID、出

力ポート、cookie 値、マッチの条件などで要求対象のフローエントリを絞ることができますが、ここではすべてのフローエントリを対象としています。

OFPPortStatsRequest は、ポートに関する統計情報をスイッチに要求します。取得したいポートの番号を指定することができます。ここでは OFPP_ANY を指定し、すべてのポートの統計情報を要求しています。

2.2.2 FlowStats

スイッチからの応答を受け取るため、FlowStatsReply メッセージを受信するイベントハンドラを作成します。

```
# ...
@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 クラスの属性 body は、OFPFlowStats のリストで、FlowStatsRequest の対象となった各フローエントリの統計情報が格納されています。

プライオリティが 0 の Table-miss フローを除いて、全てのフローエントリを選択しています。受信ポートと宛先 MAC アドレスでソートして、それぞれのフローエントリにマッチしたパケット数とバイト数を出力しています。

なお、ここでは一部の数値をログに出しているだけですが、継続的に情報を収集、分析するには、外部プログラムとの連携が必要になるでしょう。そのような場合、OPFFlowStatsReply の内容を JSON フォーマットに変換することができます。

例えば次のように書くことができます。

```
import json
# ...
```

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

この場合、以下のように出力されます。

```
{
    "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": [
                {
                    "OFPACTION_OUTPUT": {
                        "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

スイッチからの応答を受け取るため、PortStatsReply メッセージを受信するイベントハンドラを作成します。

```
# ...
@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 クラスの属性 body は、OFPPortStats のリストになっています。

OFPPortStats には、ポート番号、送受信それぞれのパケット数、バイト数、ドロップ数、エラー数、フレームエラー数、オーバーラン数、CRC エラー数、コリジョン数などの統計情報が格納されます。

ここでは、ポート番号でソートし、受信パケット数、受信バイト数、受信エラー数、送信パケット数、送信バイト数、送信エラー数を出力しています。

2.3 トラフィックモニターの実行

それでは、実際にこのトラフィックモニターを実行してみます。

まず、「[スイッチングハブ](#)」と同様に Mininet を実行します。ここで、スイッチの OpenFlow バージョンに OpenFlow13 を設定することを忘れないでください。

次にいよいよトラフィックモニターの実行です。

controller: c0:

```

ryu@ryu-vm:~# ryu-manager --verbose ./simple_monitor.py
loading app ./simple_monitor.py
loading app ryu.controller.ofp_handler
instantiating app ./simple_monitor.py
instantiating app ryu.controller.ofp_handler
BRICK SimpleMonitor
    CONSUMES EventOFPStateChange
    CONSUMES EventOFPFlowStatsReply
    CONSUMES EventOFPPortStatsReply
    CONSUMES EventOFPPacketIn
    CONSUMES EventOFPSwitchFeatures
BRICK ofp_event
    PROVIDES EventOFPStateChange TO {'SimpleMonitor': set(['main', 'dead'])}
    PROVIDES EventOFPFlowStatsReply TO {'SimpleMonitor': set(['main'])}
    PROVIDES EventOFPPortStatsReply TO {'SimpleMonitor': set(['main'])}
    PROVIDES EventOFPPacketIn TO {'SimpleMonitor': set(['main'])}
    PROVIDES EventOFPSwitchFeatures TO {'SimpleMonitor': set(['config'])}
    CONSUMES EventOFPErrorMsg
    CONSUMES EventOFPPortDescStatsReply
    CONSUMES EventOFPHello
    CONSUMES EventOFPEchoRequest
    CONSUMES EventOFPSwitchFeatures
connected socket:<eventlet.greenio.GreenSocket object at 0x343fb10> address:('127.0.0.1', 55598)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0x343fed0>
move onto config mode
EVENT ofp_event->SimpleMonitor EventOFPSwitchFeatures
switch features ev version: 0x4 msg_type 0x6 xid 0x7dd2dc58 OFPSwitchFeatures(auxiliary_id=0, capabilities=0)
move onto main mode
EVENT ofp_event->SimpleMonitor EventOFPStateChange
register datapath: 0000000000000001
send stats request: 0000000000000001
EVENT ofp_event->SimpleMonitor EventOFPFlowStatsReply
datapath      in-port  eth-dst          out-port packets  bytes
-----  -----
EVENT ofp_event->SimpleMonitor EventOFPPortStatsReply
datapath      port      rx-pkts  rx-bytes rx-error tx-pkts  tx-bytes tx-error
-----  -----
0000000000000001      1        0        0        0        0        0        0
0000000000000001      2        0        0        0        0        0        0
0000000000000001      3        0        0        0        0        0        0
0000000000000001 fffffffe      0        0        0        0        0        0

```

「スイッチングハブ」では、ryu-manager コマンドに SimpleSwitch13 のモジュール名 (ryu.app.simple_switch_13) を指定しましたが、ここでは、SimpleMonitor のファイル名 (./simple_monitor.py) を指定しています。

この時点では、フローエントリが無く (Table-miss フローエントリは表示していません)、各ポートのカウントもすべて 0 です。

ホスト 1 からホスト 2 へ ping を実行してみましょう。

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:~#
```

パケットの転送や、フローエントリが登録され、統計情報が変化します。

controller: c0:

datapath	in-port	eth-dst	out-port	packets	bytes	
00000000000000000001	1	00:00:00:00:00:02	2	1	42	
00000000000000000001	2	00:00:00:00:00:01	1	2	140	
datapath	port	rx-pkts	rx-bytes	rx-error	tx-pkts	tx-bytes
00000000000000000001	1	3	182	0	3	182
00000000000000000001	2	3	182	0	3	182
00000000000000000001	3	0	0	0	1	42
00000000000000000001	ffffffe	0	0	0	1	42

フローエントリの統計情報では、受信ポート 1 のフローにマッチしたトラフィックは、1 パケット、42 バイトと記録されています。受信ポート 2 では、2 パケット、140 バイトとなっています。

ポートの統計情報では、ポート 1 の受信パケット数 (rx-pkts) は 3、受信バイト数 (rx-bytes) は 182 バイト、ポート 2 も 3 パケット、182 バイトとなっています。

フローエントリの統計情報とポートの統計情報で数字が合っていませんが、これはフローエントリの統計情報は、そのエントリにマッチし転送されたパケットの情報だからです。つまり、Table-miss により Packet-In を発行し、Packet-Out で転送されたパケットは、この統計の対象になっていないためです。

このケースでは、ホスト 1 が最初にブロードキャストした ARP リクエスト、ホスト 2 がホスト 1 に返した ARP リプライ、ホスト 1 がホスト 2 へ発行した echo request の 3 パケットが、Packet-Out によって転送されています。そのため、ポートの統計量は、フローエントリの統計量よりも多くなっています。

2.4 まとめ

本章では、統計情報の取得機能を題材として、以下の項目について説明しました。

- Ryu アプリケーションでのスレッドの生成方法
- Datapath の状態遷移の捕捉
- FlowStats および PortStats の取得方法

第3章

REST 連携

本章では、「スイッチングハブ」で説明したスイッチングハブに、REST 連携の機能を追加します。

3.1 REST API の組み込み

Ryu には WSGI に対応した Web サーバの機能があります。この機能を利用することで、他のシステムやプラウザなどとの連携をする際に役に立つ、REST API を作成することができます。

ノート: WSGI とは、Pythonにおいて、Web アプリケーションと Web サーバをつなぐための統一されたフレームワークのことを指します。

3.2 REST API 付きスイッチングハブの実装

「スイッチングハブ」で説明したスイッチングハブに、次の二つの REST API を追加してみましょう。

1. MAC アドレステーブル取得 API

スイッチングハブが保持している MAC アドレステーブルの内容を返却します。MAC アドレスおよびポート番号の組を JSON 形式で返却します。

2. MAC アドレステーブル登録 API

MAC アドレスとポート番号の組を MAC アドレステーブルに登録し、スイッチへフローエントリの追加を行います。

それではソースコードを見てみましょう。

```
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 = '/v1/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 では、二つのクラスを定義しています。

一つ目は、HTTP リクエストを受ける URL とそれに対応するメソッドを定義するコントローラクラス SimpleSwitchController です。

二つ目は「スイッチングハブ」を拡張し、MAC アドレステーブルの更新を行えるようにしたクラス SimpleSwitchRest13 です。

SimpleSwitchRest13 では、スイッチにフローエントリを追加するため、FeaturesReply メソッドをオーバライドし、datapath オブジェクトを保持しています。

3.3 SimpleSwitchRest13 クラスの実装

```
class SimpleSwitchRest13(simple_switch_13.SimpleSwitch13):  
  
    _CONTEXTS = { 'wsgi': WSGIApplication }  
    ...
```

クラス変数 _CONTEXTS で、Ryu の WSGI 対応 Web サーバのクラスを指定しています。これにより、wsgi というキーで、WSGI の Web サーバインスタンスが取得できます。

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

コンストラクタでは、後述するコントローラクラスを登録するために、WSGIApplication のインスタンスを取得しています。登録には、register メソッドを使用します。register メソッド実行の際、コントローラのコンストラクタで SimpleSwitchRest13 クラスのインスタンスにアクセスできるように、simple_switch_api_app というキー名でディクショナリオブジェクトを渡しています。

```
@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, {})  
    ...
```

親クラスの switch_features_handler をオーバーライドしています。このメソッドでは、SwitchFeatures イベントが発生したタイミングで、イベントオブジェクト ev に格納された datapath オブジェクトを取得し、インスタンス変数 switches に保持しています。また、このタイミングで、MAC アдресテーブルに初期値として空のディクショナリをセットしています。

```
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
...
```

指定のスイッチに MAC アドレスとポートを登録するメソッドです。REST API が PUT メソッドで呼ばれると実行されます。

引数 `entry` には、登録をしたい MAC アドレスと接続ポートのペアが格納されています。

MAC アドレステーブル `self.mac_to_port` の情報を参照しながら、スイッチに登録するフローエントリを求めていきます。

例えば、MAC アドレステーブルに、次の MAC アドレスと接続ポートのペアが登録されていて、

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

引数 `entry` で渡された MAC アドレスとポートのペアが、

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

の場合、スイッチに登録する必要のあるフローエントリは次の通りです。

- マッチング条件 : `in_port = 1, dst_mac = 00:00:00:00:00:02` アクション : `output=2`
- マッチング条件 : `in_port = 2, dst_mac = 00:00:00:00:00:01` アクション : `output=1`

フローエントリの登録は親クラスの `add_flow` メソッドを利用しています。最後に、引数 `entry` で渡された情報を MAC アドレステーブルに格納しています。

3.4 SimpleSwitchController クラスの実装

次は REST API への HTTP リクエストを受け付けるコントローラクラスです。クラス名は `SimpleSwitchController` です。

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

コンストラクタで、`SimpleSwitchRest13` クラスのインスタンスを取得します。

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

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

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

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

...
```

REST API の URL とそれに対応する処理を実装する部分です。このメソッドと URL との対応づけに Ryu で定義された `route` デコレータを用いています。

デコレータで指定する内容は、次の通りです。

- 第1引数

任意の名前

- 第2引数

URL を指定します。URL が `http://<サーバIP>:8080/simpleswitch/mactable/<データパスID>` となるようにします。

- 第3引数

HTTP メソッドを指定します。GET メソッドを指定しています。

- 第4引数

指定箇所の形式を指定します。URL(/simpleswitch/mactable/{dpid}) の {dpid} の部分が、`ryu/lib/dpid.py` の `DPID_PATTERN` で定義された 16 衔の 16 進数値の表現に合致することを条件としています。

第2引数で指定した URL で REST API が呼ばれ、その時の HTTP メソッドが GET の場合に、`list_mac_table` メソッドが呼ばれます。このメソッドは、{dpid}の部分で指定されたデータパス ID に該当する MAC アドレステーブルを取得し、JSON 形式に変換し呼び出し元に返却しています。

なお、Ryu に接続していない未知のスイッチのデータパス ID を指定するとレスポンスコード 404 を返します。

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

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

```

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

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

```

次は、MAC アドレステーブルを登録する REST API です。

URL は MAC アドレステーブル取得時の API と同じですが、HTTP メソッドが PUT の場合に put_mac_table メソッドが呼ばれます。このメソッドでは、内部でスイッチングハブインスタンスの set_mac_to_port メソッドを呼び出しています。なお、put_mac_table メソッド内で例外が発生した場合、レスポンスコード 500 を返却します。また、list_mac_table メソッドと同様、Ryu に接続していない未知のスイッチのデータパス ID を指定するとレスポンスコード 404 を返します。

3.5 REST API 搭載スイッチングハブの実行

REST API を追加したスイッチングハブを実行してみましょう。

最初に「[スイッチングハブ](#)」と同様に Mininet を実行します。ここでもスイッチの OpenFlow バージョンに OpenFlow13 を設定することを忘れないでください。続いて、REST API を追加したスイッチングハブを起動します。

```

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=0)
move onto main mode
```

起動時のメッセージの中に、「(31135) wsgi starting up on <http://0.0.0.0:8080/>」という行がありますが、これは、Web サーバがポート番号 8080 で起動したことを表しています。

次に mininet のシェル上で、h1 から h2 へ ping を発行します。

```
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
```

この時、Ryu への Packet-In は 3 回発生しています。

```
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
```

ここで、スイッチングハブの MAC テーブルを取得する REST API を実行してみましょう。今回は、REST API の呼び出しに curl コマンドを使用します。

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

h1 と h2 の二つのホストが MAC アドレステーブル上で学習済みであることがわかります。

今度は、h1,h2 の 2 台のホストをあらかじめ MAC アドレステーブルに格納し、ping を実行してみます。いったんスイッチングハブと Mininet を停止します。次に、再度 Mininet を起動し、OpenFlow バージョンを OpenFlow13 に設定後、スイッチングハブを起動します。

```
...
(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=0)
switch_features_handler inside sub class
move onto main mode
```

次に、MAC アドレステーブル更新用の REST API を 1 ホストごとに呼び出します。REST API を呼び出す際

のデータ形式は、{“mac”：“MAC アドレス”, “port”：接続ポート番号}となるようにします。

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

これらのコマンドを実行すると、h1,h2 に対応したフローエントリがスイッチに登録されます。

続いて、h1 から h2 へ ping を実行します。

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

この時、スイッチにはすでにフローエントリが存在するため、Packet-In は h1 から h2 への ARP リクエストの時だけ発生し、それ以降のパケットのやりとりでは発生していません。

3.6 まとめ

本章では、MAC アдресテーブルの参照や更新をする機能を題材として、REST API の追加方法について説明しました。その他の応用として、スイッチに任意のフローエントリを追加できるような REST API を作成し、ブラウザから操作できるようにするのもよいのではないでしょうか。

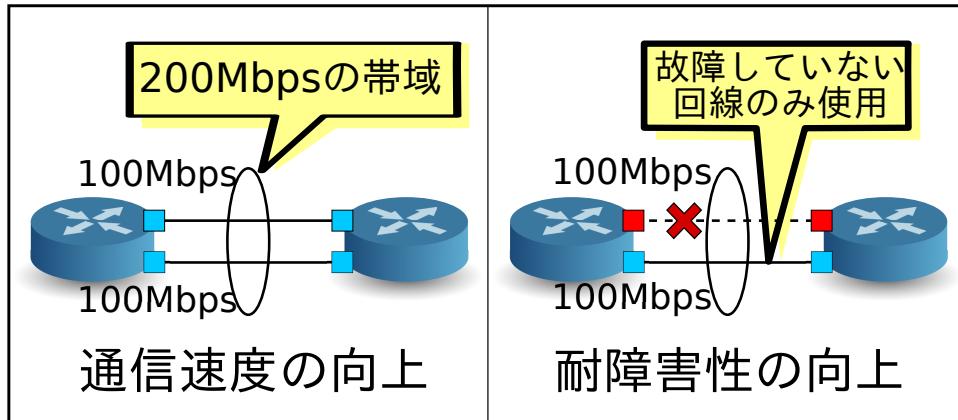
第4章

リンク・アグリゲーション

本章では、Ryu を用いたリンク・アグリゲーション機能の実装方法を解説していきます。

4.1 リンク・アグリゲーション

リンク・アグリゲーションは、IEEE802.1AX-2008 で規定されている、複数の物理的な回線を束ねてひとつの論理的なリンクとして扱う技術です。リンク・アグリゲーション機能により、特定のネットワーク機器間の通信速度を向上させることができ、また同時に、冗長性を確保することで耐障害性を向上させることができます。



リンク・アグリゲーション機能を使用するには、それぞれのネットワーク機器において、どのインターフェースをどのグループとして束ねるのかという設定を事前に行っておく必要があります。

リンク・アグリゲーション機能を開始する方法には、それぞれのネットワーク機器に対し直接指示を行うスタティックな方法と、LACP(Link Aggregation Control Protocol) というプロトコルを使用することによって動的に開始させるダイナミックな方法があります。

ダイナミックな方法を採用した場合、各ネットワーク機器は対向インターフェース同士で LACP データユニットを定期的に交換することにより、疎通不可能になつてないことをお互いに確認し続けます。LACP データユニットの交換が途絶えた場合、故障が発生したものとみなされ、当該ネットワーク機器は使用不可能となり、パケットの送受信は残りのインターフェースによってのみ行われるようになります。この方法には、ネッ

トワーク機器間にメディアコンバータなどの中継装置が存在した場合にも、中継装置の向こう側のリンクダウンを検知することができるというメリットがあります。本章では、LACP を用いたダイナミックなリンク・アグリゲーション機能を取り扱います。

4.2 Ryu アプリケーションの実行

ソースの説明は後回しにして、まずは Ryu のリンク・アグリゲーション・アプリケーションを実行してみます。

Ryu のソースツリーに用意されている simple_switch_lacp.py は OpenFlow 1.0 専用のアプリケーションであるため、ここでは新たに OpenFlow 1.3 に対応した simple_switch_lacp_13.py を作成することとします。このプログラムは、「スイッチングハブ」のスイッチングハブにリンク・アグリゲーション機能を追加したアプリケーションです。

ソース名：simple_switch_lacp_13.py

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

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

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

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

        # install table-miss flow entry
        #
        # We specify NO BUFFER to max_len of the output action due to
        # OVS bug. At this moment, if we specify a lesser number, e.g.,
```

```

# 128, OVS will send Packet-In with invalid buffer_id and
# truncated packet data. In that case, we cannot output packets
# correctly.
match = parser.OFPMatch()
actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                  ofproto.OFPCML_NO_BUFFER) ]
self.add_flow(datapath, 0, match, actions)

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

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

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

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

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

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

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

    dst = eth.dst
    src = eth.src

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

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

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

```

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

actions = [parser.OFPActionOutput(out_port)]

# install a flow to avoid packet_in next time
if out_port != ofproto.OFPP_FLOOD:
    match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
    self.add_flow(datapath, 1, match, actions)

data = None
if msg.buffer_id == ofproto.OFP_NO_BUFFER:
    data = msg.data

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

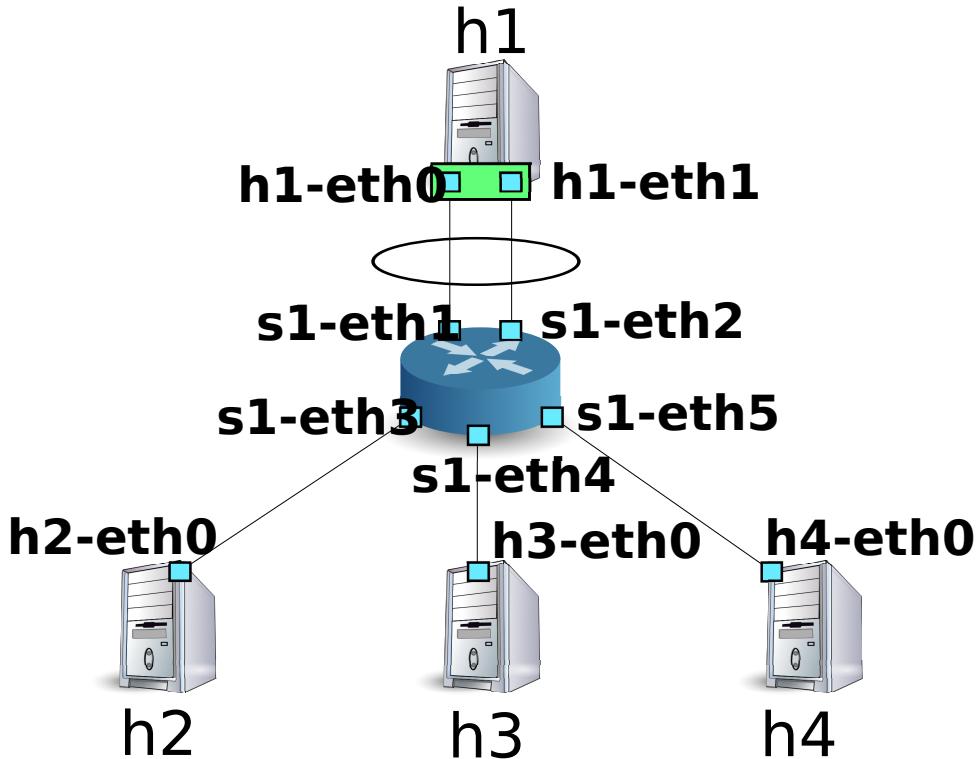
@set_ev_cls(lacplib.EventSlaveStateChanged, MAIN_DISPATCHER)
def _slave_state_changed_handler(self, ev):
    datapath = ev.datapath
    dpid = datapath.id
    port_no = ev.port
    enabled = ev.enabled
    self.logger.info("slave state changed port: %d enabled: %s",
                     port_no, enabled)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

4.2.1 実験環境の構築

OpenFlow スイッチと Linux ホストの間でリンク・アグリゲーションを構成してみましょう。

VM イメージ利用のための環境設定やログイン方法等は「[スイッチングハブ](#)」を参照してください。

最初に Mininet を利用して下図の様なトポロジを作成します。



Mininet の API を呼び出すスクリプトを作成し、必要なトポロジを構築します。

ソース名 : 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()
```

このスクリプトを実行することにより、ホスト h1 とスイッチ s1 の間に 2 本のリンクが存在するトポロジが作成されます。net コマンドで作成されたトポロジを確認することができます。

```
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 ホスト h1 でのリンク・アグリゲーションの設定

ホスト h1 の Linux に必要な事前設定を行いましょう。本節でのコマンド入力は、ホスト h1 の xterm 上で行ってください。

まず、リンク・アグリゲーションを行うためのドライバモジュールをロードします。Linux ではリンク・アグリゲーション機能をボンディングドライバが担当しています。事前にドライバの設定ファイルを /etc/modprobe.d/bonding.conf として作成しておきます。

ファイル名:/etc/modprobe.d/bonding.conf

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

Node: h1:

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

mode=4 は LACP を用いたダイナミックなリンク・アグリゲーションを行うことを表します。デフォルト値で

あるためここでは設定を省略していますが、LACP データユニットの交換間隔は SLOW (30 秒間隔) 振り分けロジックは宛先 MAC アドレスを元に行うように設定されています。

続いて、bond0 という名前の論理インターフェースを新たに作成します。また、bond0 の MAC アドレスとして適当な値を設定します。

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
```

作成した論理インターフェースのグループに、h1-eth0 と h1-eth1 の物理インターフェースを参加させます。このとき、物理インターフェースをダウンさせておく必要があります。また、ランダムに決定された物理インターフェースの MAC アドレスをわかりやすい値に書き換えておきます。

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
```

論理インターフェースに IP アドレスを割り当てます。ここでは 10.0.0.1 を割り当てることにします。また、h1-eth0 に IP アドレスが割り当てられているので、これを削除します。

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
```

最後に、論理インターフェースをアップさせます。

Node: h1:

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

ここで各インターフェースの状態を確認しておきます。

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)
```

論理インターフェース bond0 が MASTER に、物理インターフェース h1-eth0 と h1-eth1 が SLAVE になっていることがわかります。また、bond0、h1-eth0、h1-eth1 の MAC アドレスがすべて同じものになっていることがわかります。

ボンディングドライバの状態も確認しておきます。

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
```

LACP データユニットの交換間隔 (LACP rate: slow) や振り分けロジックの設定 (Transmit Hash Policy: layer2 (0)) が確認できます。また、物理インターフェース h1-eth0 と h1-eth1 の MAC アドレスが確認できます。

以上でホスト h1 への事前設定は終了です。

4.2.3 OpenFlow バージョンの設定

スイッチ s1 の OpenFlow のバージョンを 1.3 に設定します。このコマンド入力は、スイッチ s1 の xterm 上で行ってください。

Node: s1:

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

4.2.4 スイッキングハブの実行

準備が整ったので、冒頭で作成した Ryu アプリケーションを実行します。

ウインドウタイトルが「Node: c0 (root)」となっている xterm から次のコマンドを実行します。

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
...
```

ホスト h1 は 30 秒に 1 回 LACP データユニットを送信しています。起動してからしばらくすると、スイッチはホスト h1 からの LACP データユニットを受信し、動作ログに出力します。

Node: c0:

```
...
[LACP] [INFO] SW=00000000000000001 PORT=1 LACP received.
[LACP] [INFO] SW=00000000000000001 PORT=1 the slave i/f has just been up.
[LACP] [INFO] SW=00000000000000001 PORT=1 the timeout time has changed.
[LACP] [INFO] SW=00000000000000001 PORT=1 LACP sent.
slave state changed port: 1 enabled: True
[LACP] [INFO] SW=00000000000000001 PORT=2 LACP received.
[LACP] [INFO] SW=00000000000000001 PORT=2 the slave i/f has just been up.
[LACP] [INFO] SW=00000000000000001 PORT=2 the timeout time has changed.
[LACP] [INFO] SW=00000000000000001 PORT=2 LACP sent.
slave state changed port: 2 enabled: True
...

```

ログは以下のことを表しています。

- LACP received.

LACP データユニットを受信しました。

- the slave i/f has just been up.

無効状態だったポートが有効状態に変更されました。

- the timeout time has changed.

LACP データユニットの無通信監視時間が変更されました(今回の場合、初期状態の 0 秒から LONG_TIMEOUT_TIME の 90 秒に変更されています)。

- LACP sent.

応答用の LACP データユニットを送信しました。

- slave state changed ...

LACP ライブラリからの EventSlaveStateChanged イベントをアプリケーションが受信しました(イベントの詳細については後述します)。

スイッチは、ホスト h1 から LACP データユニットを受信の都度、応答用 LACP データユニットを送信します。

Node: c0:

```
...
[LACP] [INFO] SW=00000000000000001 PORT=1 LACP received.
[LACP] [INFO] SW=00000000000000001 PORT=1 LACP sent.
[LACP] [INFO] SW=00000000000000001 PORT=2 LACP received.
[LACP] [INFO] SW=00000000000000001 PORT=2 LACP sent.
...

```

フローエントリを確認してみましょう。

Node: s1:

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

スイッチには

- h1 の h1-eth1(入力ポートが s1-eth2 で MAC アドレスが 00:00:00:00:00:12) から LACP データユニット (ethertype が 0x8809) が送られてきたら Packet-In メッセージを送信する
- h1 の h1-eth0(入力ポートが s1-eth1 で MAC アドレスが 00:00:00:00:00:11) から LACP データユニット (ethertype が 0x8809) が送られてきたら Packet-In メッセージを送信する
- 「[スイッチングハブ](#)」と同様の Table-miss フローエントリ

の 3 つのフローエントリが登録されています。

4.2.5 リンク・アグリゲーション機能の確認

通信速度の向上

まずはリンク・アグリゲーションによる通信速度の向上を確認します。通信に応じて複数のリンクを使い分ける様子を見てみましょう。

まず、ホスト h2 からホスト h1 に対し ping を実行します。

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
...
...
```

ping を送信し続けたまま、スイッチ s1 のフローエントリを確認します。

Node: s1:

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=22.05s, table=0, n_packets=1, n_bytes=124, idle_timeout=90, send_flow_rem priority=0
cookie=0x0, duration=22.046s, table=0, n_packets=1, n_bytes=124, idle_timeout=90, send_flow_rem priority=0
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=02:01:02:03:04:05
cookie=0x0, duration=3.262s, table=0, n_packets=4, n_bytes=392, priority=1, in_port=1, dl_dst=00:00:00:00:00:00
```

先ほど確認した時点から、2つのフローエントリが追加されています。durationの値が小さい4番目と5番目のエントリです。

それぞれ、

- 3番ポート(s1-eth3、つまりh2の対向インターフェース)からh1のbond0宛のパケットを受信したら1番ポート(s1-eth1)から出力する
- 1番ポート(s1-eth1)からh2宛のパケットを受信したら3番ポート(s1-eth3)から出力する

というフローエントリです。h2とh1の間の通信にはs1-eth1が使用されていることがわかります。

続いて、ホストh3からホストh1に対しpingを実行します。

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
...
```

pingを送信し続けたまま、スイッチs1のフローエントリを確認します。

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=0
cookie=0x0, duration=99.761s, table=0, n_packets=4, n_bytes=496, idle_timeout=90, send_flow_rem priority=0
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=02:01:00:00:00:00
cookie=0x0, duration=2.677s, table=0, n_packets=2, n_bytes=196, priority=1,in_port=2,dl_dst=00:00:00:00:00:00
cookie=0x0, duration=2.675s, table=0, n_packets=1, n_bytes=98, priority=1,in_port=4,dl_dst=02:01:02:00:00:00
cookie=0x0, duration=80.977s, table=0, n_packets=83, n_bytes=8022, priority=1,in_port=1,dl_dst=00:00:00:00:00:00
```

先ほど確認した時点から、2つのフローエントリが追加されています。durationの値が小さい5番目と6番目のエントリです。

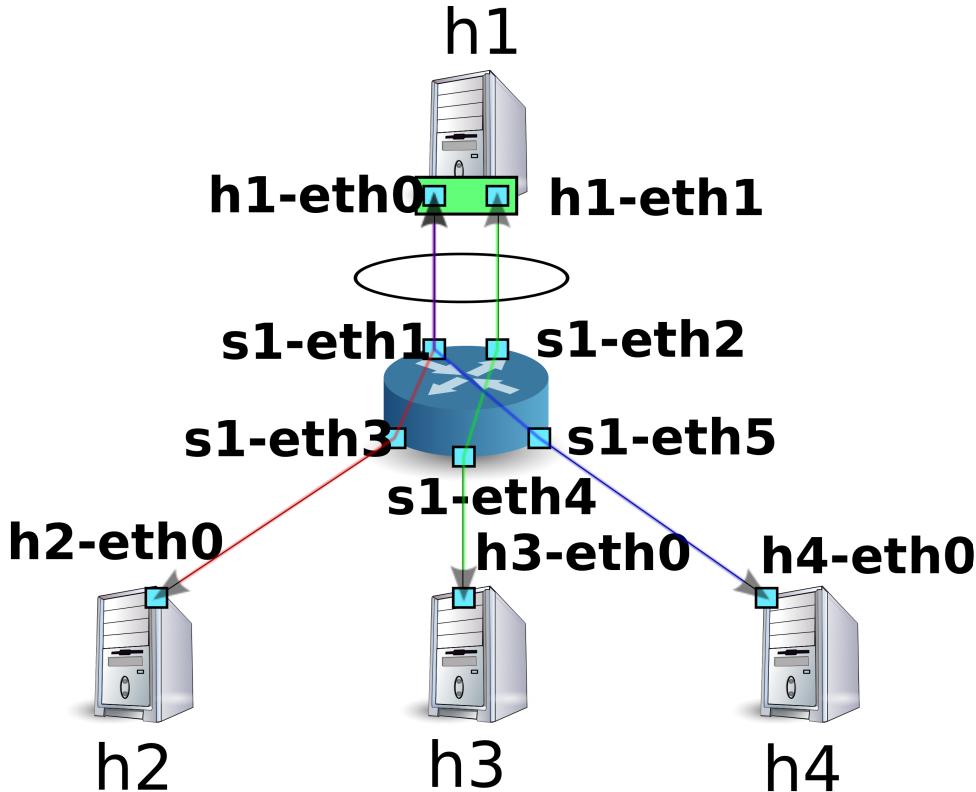
それぞれ、

- 2番ポート(s1-eth2)からh3宛のパケットを受信したら4番ポート(s1-eth4)から出力する
- 4番ポート(s1-eth4、つまりh3の対向インターフェース)からh1のbond0宛のパケットを受信したら2番ポート(s1-eth2)から出力する

というフローエントリです。h3とh1の間の通信にはs1-eth2が使用されていることがわかります。

もちろんホストh4からホストh1に対しても、pingを実行出来ます。これまでと同様に新たなフローエントリが登録され、h4とh1の間の通信にはs1-eth1が使用されます。

宛先ホスト	使用ポート
h2	1
h3	2
h4	1



以上のように、通信に応じて複数リンクを使い分ける様子を確認できました。

耐障害性の向上

次に、リンク・アグリゲーションによる耐障害性の向上を確認します。現在の状況は、h2 と h4 が h1 と通信する際には s1-eth2 を、h3 が h1 と通信する際には s1-eth1 を使用しています。

ここで、s1-eth1 の対向インターフェースである h1-eth0 をリンク・アグリゲーションのグループから離脱させます。

Node: h1:

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

h1-eth0 が停止したことにより、ホスト h3 からホスト h1 への ping が疎通不可能になります。無通信監視時間の 90 秒が経過すると、コントローラの動作ログに次のようなメッセージが出力されます。

Node: c0:

```
...
[LACP] [INFO] SW=00000000000000000000000000000001 PORT=1 LACP received.
[LACP] [INFO] SW=00000000000000000000000000000001 PORT=1 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=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.」は無通信監視時間に達したことを表します。ここでは、学習した MAC アドレスと転送用のフローエントリをすべて削除することで、スイッチを起動直後の状態に戻します。

新たな通信が発生すれば、新たに MAC アドレスを学習し、生きているリンクのみを利用したフローエントリが再び登録されます。

ホスト h3 とホスト h1 の間も新たなフローエントリが登録され、

Node: s1:

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x0, duration=364.265s, table=0, n_packets=13, n_bytes=1612, idle_timeout=90, send_flow_rem p
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=02:01:00:00:00:00
cookie=0x0, duration=6.279s, table=0, n_packets=5, n_bytes=490, priority=1,in_port=2,dl_dst=00:00:00:00:00:00
cookie=0x0, duration=6.281s, table=0, n_packets=5, n_bytes=490, priority=1,in_port=5,dl_dst=02:01:00:00:00:00
cookie=0x0, duration=5.506s, table=0, n_packets=5, n_bytes=434, priority=1,in_port=4,dl_dst=02:01:00:00:00:00
cookie=0x0, duration=5.736s, table=0, n_packets=5, n_bytes=490, priority=1,in_port=2,dl_dst=00:00:00:00:00:00
cookie=0x0, duration=6.504s, table=0, n_packets=6, n_bytes=532, priority=1,in_port=2,dl_dst=00:00:00:00:00:00
```

ホスト h3 で停止していた ping が再開します。

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
...
```

以上のように、一部のリンクに故障が発生した場合でも、他のリンクを用いて自動的に復旧できることが確認できました。

4.3 Ryu によるリンク・アグリゲーション機能の実装

OpenFlow を用いてどのようにリンク・アグリゲーション機能を実現しているかを見ていきます。

LACP を用いたリンク・アグリゲーションでは「LACP データユニットの交換が正常に行われている間は当該物理インターフェースは有効」「LACP データユニットの交換が途絶えたら当該物理インターフェースは無効」という振る舞いをします。物理インターフェースが無効ということは、そのインターフェースを使用するフローエントリが存在しないということでもあります。従って、

- LACP データユニットを受信したら応答を作成して送信する
- LACP データユニットが一定時間受信できなかったら当該物理インターフェースを使用するフローエントリを削除し、以降そのインターフェースを使用するフローエントリを登録しない
- 無効とされた物理インターフェースで LACP データユニットを受信した場合、当該インターフェースを再度有効化する
- LACP データユニット以外のパケットは「[スイッチングハブ](#)」と同様に学習・転送する

という処理を実装すれば、リンク・アグリゲーションの基本的な動作が可能となります。LACP に関わる部分とそうでない部分が明確に分かれているので、LACP に関わる部分を LACP ライブラリとして切り出し、そうでない部分は「[スイッチングハブ](#)」のスイッチングハブを拡張するかたちで実装します。

LACP データユニット受信時の応答作成・送信はフローエントリだけでは実現不可能であるため、Packet-In メッセージを使用して OpenFlow コントローラ側で処理を行います。

ノート: LACP データユニットを交換する物理インターフェースは、その役割によって ACTIVE と PASSIVE に分類されます。ACTIVE は一定時間ごとに LACP データユニットを送信し、疎通を能動的に確認します。PASSIVE は ACTIVE から送信された LACP データユニットを受信した際に応答を返すことにより、疎通を受動的に確認します。

Ryu のリンク・アグリゲーション・アプリケーションは、PASSIVE モードのみ実装しています。

一定時間 LACP データユニットを受信しなかった場合に当該物理インターフェースを無効にする、という処理は、LACP データユニットを Packet-In させるフローエントリに idle_timeout を設定し、時間切れの際に FlowRemoved メッセージを送信させることにより、OpenFlow コントローラで当該インターフェースが無効になった際の対処を行うことができます。

無効となったインターフェースで LACP データユニットの交換が再開された場合の処理は、LACP データユニット受信時の Packet-In メッセージのハンドラで当該インターフェースの有効/無効状態を判別・変更することで実現します。

物理インターフェースが無効となったとき、OpenFlow コントローラの処理としては「当該インターフェースを使用するフローエントリを削除する」だけでよさそうに思えますが、それでは不充分です。

たとえば 3 つの物理インターフェースをグループ化して使用している論理インターフェースがあり、振り分けロジックが「有効なインターフェース数による MAC アドレスの剩余」となっている場合を仮定します。

インターフェース 1	インターフェース 2	インターフェース 3
MAC アドレスの剩余:0	MAC アドレスの剩余:1	MAC アドレスの剩余:2

そして、各物理インターフェースを使用するフローエントリが以下のように3つずつ登録されていたとします。

インターフェース1	インターフェース2	インターフェース3
宛先:00:00:00:00:00:00	宛先:00:00:00:00:00:01	宛先:00:00:00:00:00:02
宛先:00:00:00:00:00:03	宛先:00:00:00:00:00:04	宛先:00:00:00:00:00:05
宛先:00:00:00:00:00:06	宛先:00:00:00:00:00:07	宛先:00:00:00:00:00:08

ここでインターフェース1が無効になった場合、「有効なインターフェース数によるMACアドレスの剩余」という振り分けロジックに従うと、次のように振り分けられなければなりません。

インターフェース1	インターフェース2	インターフェース3
無効	MACアドレスの剩余:0	MACアドレスの剩余:1

インターフェース1	インターフェース2	インターフェース3
	宛先:00:00:00:00:00:00	宛先:00:00:00:00:00:01
	宛先:00:00:00:00:00:02	宛先:00:00:00:00:00:03
	宛先:00:00:00:00:00:04	宛先:00:00:00:00:00:05
	宛先:00:00:00:00:00:06	宛先:00:00:00:00:00:07
	宛先:00:00:00:00:00:08	

インターフェース1を使用していたフローエントリだけではなく、インターフェース2やインターフェース3のフローエントリも書き換える必要があることがわかります。これは物理インターフェースが無効になったときだけでなく、有効になったときも同様です。

従って、ある物理インターフェースの有効/無効状態が変更された場合の処理は、当該物理インターフェースが所属する論理インターフェースに含まれるすべての物理インターフェースを使用するフローエントリを削除する、としています。

ノート：振り分けロジックについては仕様で定められておらず、各機器の実装に委ねられています。Ryuのリンク・アグリゲーション・アプリケーションでは独自の振り分け処理を行わず、対向装置によって振り分けられた経路を使用しています。

ここでは、次のような機能を実装します。

LACPライブラリ

- LACPデータユニットを受信したら応答を作成して送信する
- LACPデータユニットの受信が途絶いたら、対応する物理インターフェースを無効とみなし、スイッチングハブに通知する
- LACPデータユニットの受信が再開されたら、対応する物理インターフェースを有効とみなし、スイッチングハブに通知する

スイッチングハブ

- LACPライブラリからの通知を受け、初期化が必要なフローエントリを削除する
- LACPデータユニット以外のパケットは従来どおり学習・転送する

LACP ライブラリおよびスイッチングハブのソースコードは、Ryu のソースツリーにあります。

`ryu/lib/lacplib.py`

`ryu/app/simple_switch_lacp.py`

ノート: `simple_switch_lacp.py` は OpenFlow 1.0 専用のアプリケーションであるため、本章では「[Ryu アプリケーションの実行](#)」に示した OpenFlow 1.3 に対応した `simple_switch_lacp_13.py` を元にアプリケーションの詳細を説明します。

4.3.1 LACP ライブラリの実装

以降の節で、前述の機能が LACP ライブラリにおいてどのように実装されているかを見ていきます。なお、引用されているソースは抜粋です。全体像については実際のソースをご参照ください。

論理インターフェースの作成

リンク・アグリゲーション機能を使用するには、どのネットワーク機器においてどのインターフェースをどのグループとして束ねるのかという設定を事前に行っておく必要があります。LACP ライブラリでは、以下のメソッドでこの設定を行います。

```
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)
```

引数の内容は以下のとおりです。

`dpid`

OpenFlow スイッチのデータパス ID を指定します。

`ports`

グループ化したいポート番号のリストを指定します。

このメソッドを呼び出すことにより、LACP ライブラリは指定されたデータパス ID の OpenFlow スイッチの指定されたポートをひとつのグループとみなします。複数のグループを作成したい場合は繰り返し `add()` メソッドを呼び出します。なお、論理インターフェースに割り当てられる MAC アドレスは、OpenFlow スイッチの持つ LOCAL ポートと同じものが自動的に使用されます。

ちなみに: OpenFlowスイッチの中には、スイッチ自身の機能としてリンク・アグリゲーション機能を提供しているものもあります(Open vSwitchなど)。ここではそうしたスイッチ独自の機能は使用せず、OpenFlowコントローラによる制御によってリンク・アグリゲーション機能を実現します。

Packet-In処理

「スイッチングハブ」は、宛先のMACアドレスが未学習の場合、受信したパケットをフラッディングします。LACPデータユニットは隣接するネットワーク機器間でのみ交換されるべきもので、他の機器に転送してしまうとリンク・アグリゲーション機能が正しく動作しません。そこで、「Packet-Inで受信したパケットがLACPデータユニットであれば横取りし、LACPデータユニット以外のパケットであればスイッチングハブの動作に委ねる」という処理を行い、スイッチングハブにはLACPデータユニットを見せないようにします。

```
@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))
```

イベントハンドラ自体は「スイッチングハブ」と同様です。受信したメッセージにLACPデータユニットが含まれているかどうかで処理を分岐させています。

LACPデータユニットが含まれていた場合はLACPライブラリのLACPデータユニット受信処理を行います。LACPデータユニットが含まれていなかった場合、send_event_to_observers()というメソッドを呼んでいます。これはryu.base.app_manager.RyuAppクラスで定義されている、イベントを送信するためのメソッドです。

「スイッチングハブ」ではRyuで定義されたOpenFlowメッセージ受信イベントについて触れましたが、ユーザが独自にイベントを定義することもできます。上記ソースで送信しているEventPacketInというイベントは、LACPライブラリ内で作成したユーザ定義イベントです。

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

ユーザ定義イベントは、ryu.controller.event.EventBaseクラスを継承して作成します。イベントクラスに内包するデータに制限はありません。EventPacketInクラスでは、Packet-Inメッセージで受信したryu.ofproto.OFPPacketInインスタンスをそのまま使用しています。

ユーザ定義イベントの受信方法については後述します。

ポートの有効/無効状態変更に伴う処理

LACP ライブラリの LACP データユニット受信処理は、以下の処理からなっています。

1. LACP データユニットを受信したポートが無効状態であれば有効状態に変更し、状態が変更したことをイベントで通知します。
 2. 無通信タイムアウトの待機時間が変更された場合、LACP データユニット受信時に Packet-In を送信するフローエントリを再登録します。
 3. 受信した LACP データユニットに対する応答を作成し、送信します。
2. の処理については後述の「[LACP データユニットを Packet-In させるフローエントリの登録](#)」で、3. の処理については後述の「[LACP データユニットの送受信処理](#)」で、それぞれ説明します。ここでは 1. の処理について説明します。

```
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()` メソッドは、指定したスイッチの指定したポートが有効か否かを取得します。

`_set_slave_enabled()` メソッドは、指定したスイッチの指定したポートの有効/無効状態を設定します。

上記のソースでは、無効状態のポートで LACP データユニットを受信した場合、ポートの状態が変更されたということを示す `EventSlaveStateChanged` というユーザ定義イベントを送信しています。

```
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
```

`EventSlaveStateChanged` イベントは、ポートが有効化したときの他に、ポートが無効化したときにも送信されます。無効化したときの処理は「[FlowRemoved メッセージの受信処理](#)」で実装されています。

`EventSlaveStateChanged` クラスには以下の情報が含まれます。

- ポートの有効/無効状態変更が発生した OpenFlow スイッチ

- 有効/無効状態変更が発生したポート番号
- 変更後の状態

LACP データユニットを Packet-In させるフローエントリの登録

LACP データユニットの交換間隔には、FAST(1秒ごと)とSLOW(30秒ごと)の2種類が定義されています。リンク・アグリゲーションの仕様では、交換間隔の3倍の時間無通信状態が続いた場合、そのインターフェースはリンク・アグリゲーションのグループから除外され、パケットの転送には使用されなくなります。

LACP ライブラリでは、LACP データユニット受信時に Packet-In させるフローエントリに対し、交換間隔の3倍の時間(SHORT_TIMEOUT_TIME は3秒、LONG_TIMEOUT_TIME は90秒)を idle_timeout として設定することにより、無通信の監視を行っています。

交換間隔が変更された場合、idle_timeout の時間も再設定する必要があるため、LACP ライブラリでは以下のような実装をしています。

```
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() メソッドは、指定したスイッチの指定したポートにおける現在の idle_timeout 値を取得します。_set_slave_timeout() メソッドは、指定したスイッチの指定したポートにおける idle_timeout 値を登録します。初期状態およびリンク・アグリゲーション・グループから除外された場合には idle_timeout 値は0に設定されているため、新たに LACP データユニットを受信した場合、交換間隔がどちらであってもフローエントリを登録します。

使用する OpenFlow のバージョンにより OFPFlowMod クラスのコンストラクタの引数が異なるため、バー

ジョンに応じたフローエントリ登録メソッドを取得しています。以下は OpenFlow 1.2 以降で使用するフローエントリ登録メソッドです。

```
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.OFPI_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)
```

上記ソースで、「対向インターフェースから LACP データユニットを受信した場合は Packet-In する」というフローエントリを、無通信監視時間つき最高優先度で設定しています。

LACP データユニットの送受信処理

LACP データユニット受信時、「ポートの有効/無効状態変更に伴う処理」や「LACP データユニットを Packet-In させるフローエントリの登録」を行った後、応答用の LACP データユニットを作成し、送信します。

```
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)
```

上記ソースで呼び出されている_create_response() メソッドは応答用パケット作成処理です。その内で呼び出されている_create_lacp() メソッドで応答用の LACP データユニットを作成しています。作成した応答用パケットは、LACP データユニットを受信したポートから Packet-Out させます。

LACP データユニットには送信側（Actor）の情報と受信側（Partner）の情報を設定します。受信した LACP

データユニットの送信側情報には対向インターフェースの情報が記載されているので、OpenFlowスイッチから応答を返すときにはそれを受信側情報として設定します。

```
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
```

FlowRemovedメッセージの受信処理

指定された時間の間LACPデータユニットの交換が行われなかった場合、OpenFlowスイッチはFlowRemovedメッセージをOpenFlowコントローラに送信します。

```
@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))
```

FlowRemoved メッセージを受信すると、OpenFlow コントローラは_set_slave_enabled() メソッドを使用してポートの無効状態を設定し、_set_slave_timeout() メソッドを使用して idle_timeout 値を 0 に設定し、send_event_to_observers() メソッドを使用して EventSlaveStateChanged イベントを送信します。

4.3.2 アプリケーションの実装

「Ryu アプリケーションの実行」に示した OpenFlow 1.3 対応のリンク・アグリゲーション・アプリケーション (simple_switch_lacp_13.py) と、「スイッチングハブ」のスイッチングハブとの差異を順に説明していきます。

「_CONTEXTS」の設定

ryu.base.app_manager.RyuApp を継承した Ryu アプリケーションは、「_CONTEXTS」ディクショナリに他の Ryu アプリケーションを設定することにより、他のアプリケーションを別スレッドで起動させることができます。ここでは LACP ライブラリの LacpLib クラスを「lacplib」という名前で「_CONTEXTS」に設定しています。

```
from ryu.lib import lacplib

# ...

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

# ...
```

「_CONTEXTS」に設定したアプリケーションは、__init__() メソッドの kwargs からインスタンスを取得することができます。

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

ライブラリの初期設定

「_CONTEXTS」に設定した LACP ライブラリの初期設定を行います。初期設定には LACP ライブラリの提供する add() メソッドを実行します。ここでは以下の値を設定します。

パラメータ	値	説明
dpid	str_to_dpid('0000000000000001')	データパス ID
ports	[1, 2]	グループ化するポートのリスト

この設定により、データパス ID 「0000000000000001」 の OpenFlow スイッチのポート 1 とポート 2 がひとつのリンク・アグリゲーション・グループとして動作します。

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

ユーザ定義イベントの受信方法

LACP ライブラリの実装で説明したとおり、LACP ライブラリは LACP データユニットの含まれない Packet-In メッセージを EventPacketIn というユーザ定義イベントとして送信します。ユーザ定義イベントのイベントハンドラも、Ryu が提供するイベントハンドラと同じように ryu.controller.handler.set_ev_cls デコレータで装飾します。

```
@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']

    # ...
```

また、LACP ライブラリはポートの有効/無効状態が変更されると EventSlaveStateChanged イベントを送信しますので、こちらもイベントハンドラを作成しておきます。

```
@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)
def self.mac_to_port[dpid]
self.mac_to_port.setdefault(dpid, {})
```

本節の冒頭で説明したとおり、ポートの有効/無効状態が変更されると、論理インターフェースを通過するパケットが実際に使用する物理インターフェースが変更になる可能性があります。そのため、登録されているフローエントリを全て削除しています。

```
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)
```

フローエントリの削除は `OFPFlowMod` クラスのインスタンスで行います。

以上のように、リンク・アグリゲーション機能を提供するライブラリと、ライブラリを利用するアプリケーションによって、リンク・アグリゲーション機能を持つスイッチングハブのアプリケーションを実現しています。

4.4 まとめ

本章では、リンク・アグリゲーションライブラリの利用を題材として、以下の項目について説明しました。

- 「`_CONTEXTS`」を用いたライブラリの使用方法
- ユーザ定義イベントの定義方法とイベントトリガーの発生方法

第5章

スパニングツリー

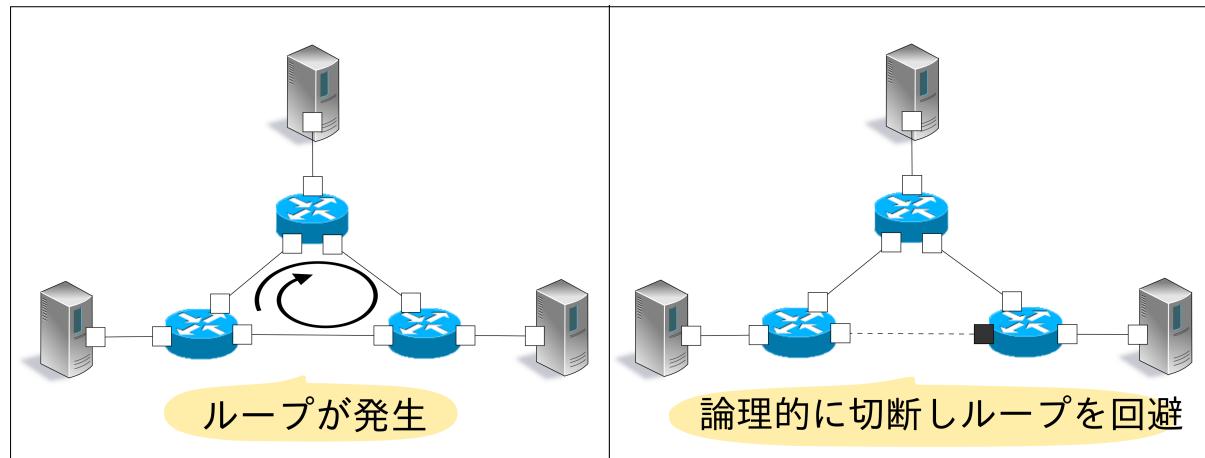
本章では、Ryuを用いたスパニングツリーの実装方法を解説していきます。

5.1 スパニングツリー

スパニングツリーはループ構造を持つネットワークにおけるブロードキャストストームの発生を抑制する機能です。また、ループを防止するという本来の機能を応用して、ネットワーク故障が発生した際に自動的に経路を切り替えるネットワークの冗長性確保の手段としても用いられます。

スパニングツリーにはSTP、RSTP、PVST+、MSTPなど様々な種別がありますが、本章では最も基本的なSTPの実装を見ていきます。

STP(spanning tree protocol : IEEE 802.1D)はネットワークを論理的なツリーとして扱い、各スイッチ(本章ではブリッジと呼ぶことがあります)のポートをフレーム転送可能または不可能な状態に設定することで、ループ構造を持つネットワークでブロードキャストストームの発生を抑制します。



STPではブリッジ間でBPDU(Bridge Protocol Data Unit)パケットを相互に交換し、ブリッジやポートの情報を比較しあうことで、各ポートのフレーム転送可否を決定します。

具体的には、次のような手順により実現されます。

1. ルートブリッジの選出

ブリッジ間の BPDU パケットの交換により、最小のブリッジ ID を持つブリッジがルートブリッジとして選出されます。以降はルートブリッジのみがオリジナルの BPDU パケットを送信し、他のブリッジはルートブリッジから受信した BPDU パケットを転送します。

ノート: ブリッジ ID は、各ブリッジに設定されたブリッジ priority と特定ポートの MAC アドレスの組み合わせで算出されます。

ブリッジ ID	
上位 2byte	下位 6byte
ブリッジ priority	MAC アドレス

2. ポートの役割の決定

各ポートのルートブリッジに至るまでのコストを元に、ポートの役割を決定します。

- ルートポート (Root port)

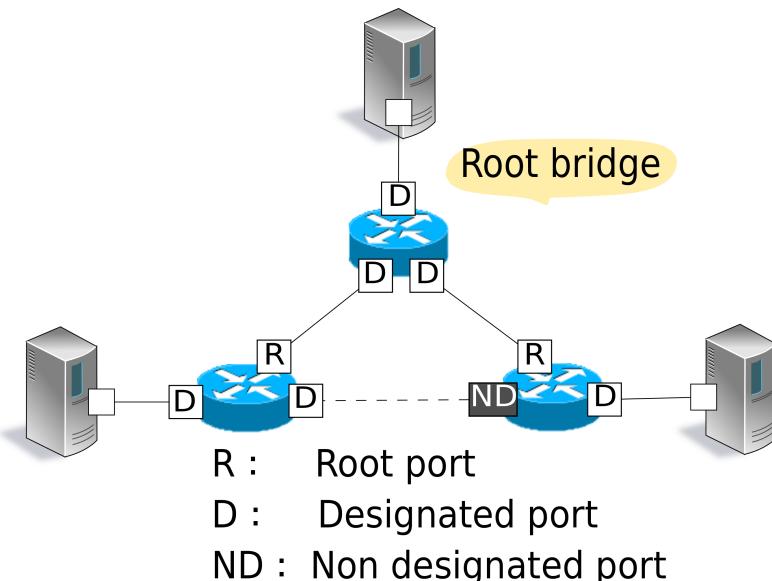
ブリッジ内で最もルートブリッジまでのコストが小さいポート。ルートブリッジからの BPDU パケットを受信するポートになります。

- 指定ポート (Designated port)

各リンクのルートブリッジまでのコストが小さい側のポート。ルートブリッジから受信した BPDU パケットを送信するポートになります。ルートブリッジのポートは全て指定ポートです。

- 非指定ポート (Non designated port)

ルートポート・指定ポート以外のポート。フレーム転送を抑制するポートです。



ノート: ルートブリッジに至るまでのコストは、各ポートが受信した BPDU パケットの設定値から次のように比較されます。

優先 1 : root path cost 値による比較。

各ブリッジは BPDU パケットを転送する際に、出力ポートに設定された path cost 値を BPDU パケットの root path cost 値に加算します。これにより root path cost 値はルートブリッジに到達するまでに経由する各リンクの path cost 値の合計の値となります。

優先 2 : root path cost 値が同じ場合、対向ブリッジのブリッジ ID により比較。

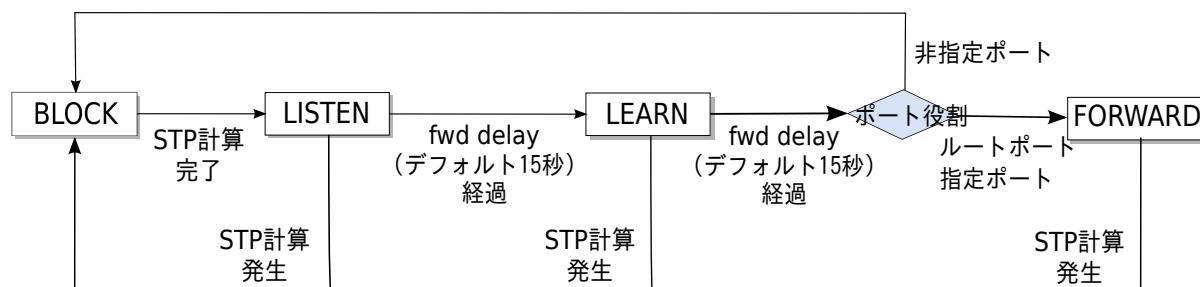
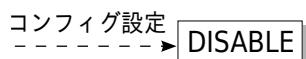
優先 3 : 対向ブリッジのブリッジ ID が同じ場合(各ポートが同一ブリッジに接続しているケース)、対向ポートのポート ID により比較。

ポート ID

上位 2byte	下位 2byte
ポート priority	ポート番号

3. ポートの状態遷移

ポート役割の決定後(STP 計算の完了時)、各ポートは LISTEN 状態になります。その後、以下に示す状態遷移を行い、最終的に各ポートの役割に従って FORWARD 状態または BLOCK 状態に遷移します。コンフィギュレーションで無効ポートと設定されたポートは DISABLE 状態となり、以降、状態遷移は行われません。



これらの処理が各ブリッジで実行されることにより、フレーム転送を行うポートとフレーム転送を抑制するポートが決定され、ネットワーク内のループが解消されます。

また、リンクダウンや BPDU パケットの max age(デフォルト 20 秒)間の未受信による故障検出、あるいはポートの追加等によりネットワーク構造の変更を検出した場合は、各ブリッジで上記の 1. 2. 3. を実行しツリーの再構築が行われます(STP の再計算)。

5.2 Ryu アプリケーションの実行

スパニングツリーの機能を OpenFlow を用いて実現した、Ryu のスパニングツリーアプリケーションを実行してみます。

Ryu のソースツリーに用意されている simple_switch_stp.py は OpenFlow 1.0 専用のアプリケーションであるため、ここでは新たに OpenFlow 1.3 に対応した simple_switch_stp_13.py を作成することとします。このプログラムは、「[スイッチングハブ](#)」にスパニングツリー機能を追加したアプリケーションです。

ソース名：simple_switch_stp_13.py

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib import dpid as dpid_lib
from ryu.lib import stplib
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet

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

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

        # Sample of stplib config.
        # please refer to stplib.Stp.set_config() for details.
        config = {dpid_lib.str_to_dpid('0000000000000001'):
                  {'bridge': {'priority': 0x8000}},
                  dpid_lib.str_to_dpid('0000000000000002'):
                  {'bridge': {'priority': 0x9000}},
                  dpid_lib.str_to_dpid('0000000000000003'):
                  {'bridge': {'priority': 0xa000}}}
        self.stp.set_config(config)

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

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

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

```

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

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

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

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

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

    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(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 実験環境の構築

スパニングツリーアプリケーションの動作確認を行う実験環境を構築します。

VMイメージ利用のための環境設定やログイン方法等は「[スイッチングハブ](#)」を参照してください。

ループ構造を持つ特殊なトポロジで動作させるため、「リンク・アグリゲーション」と同様にトポロジ構築スクリプトにより mininet 環境を構築します。

ソース名 : 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)

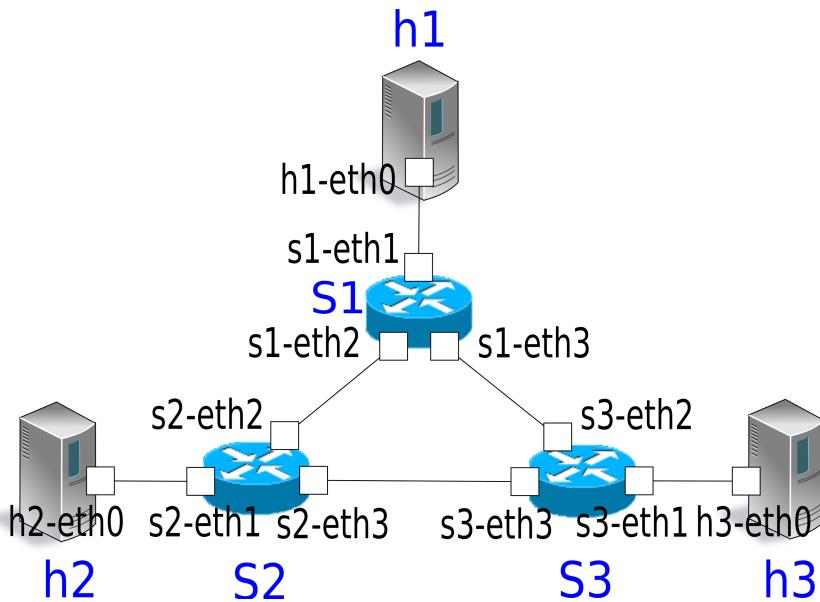
net.build()
c0.start()
s1.start([c0])
s2.start([c0])
s3.start([c0])

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()
```

VM 環境でこのプログラムを実行することにより、スイッチ s1、s2、s3 の間でループが存在するトポロジが作成されます。



net コマンドの実行結果は以下の通りです。

```

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 OpenFlow バージョンの設定

使用する OpenFlow のバージョンを 1.3 に設定します。このコマンド入力は、スイッチ s1、s2、s3 の xterm 上で行ってください。

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 スイッチングハブの実行

準備が整ったので、Ryu アプリケーションを実行します。ウインドウタイトルが「Node: c0 (root)」となっている xterm から次のコマンドを実行します。

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 スイッチ起動時の STP 計算

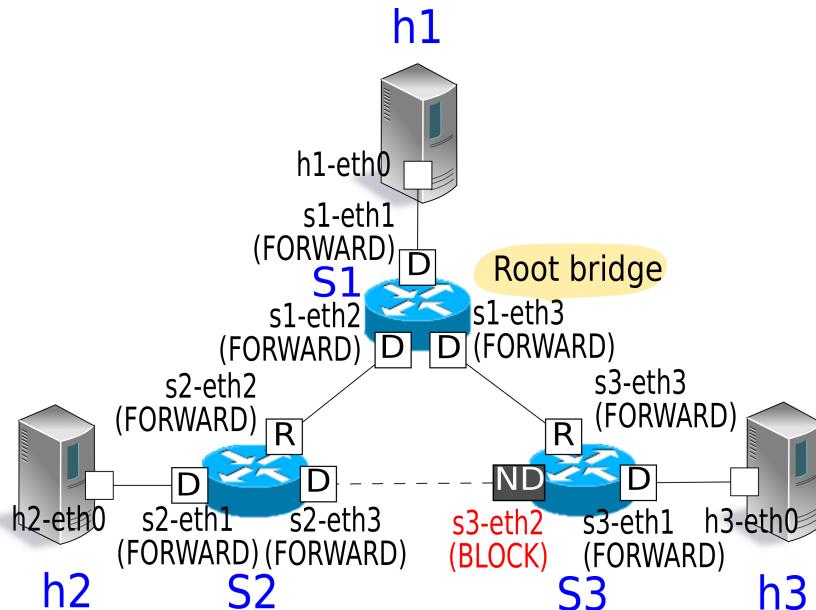
各 OpenFlow スイッチとコントローラの接続が完了すると、BPDU パケットの交換が始まり、ルートブリッジの選出・ポート役割の設定・ポート状態遷移が行われます。

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

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

```
[STP] [INFO] dpid=0000000000000000: [port=3] DESIGNATED_PORT      / FORWARD
[STP] [INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT      / FORWARD
[STP] [INFO] dpid=0000000000000003: [port=2] NON_DESIGNATED_PORT / BLOCK
[STP] [INFO] dpid=0000000000000003: [port=3] ROOT_PORT        / FORWARD
[STP] [INFO] dpid=0000000000000001: [port=1] DESIGNATED_PORT      / FORWARD
[STP] [INFO] dpid=0000000000000001: [port=2] DESIGNATED_PORT      / FORWARD
[STP] [INFO] dpid=0000000000000001: [port=3] DESIGNATED_PORT      / FORWARD
```

この結果、最終的に各ポートは FORWARD 状態または BLOCK 状態となります。



パケットがループしないことを確認するため、ホスト 1 からホスト 2 へ ping を実行します。

ping コマンドを実行する前に、tcpdump コマンドを実行しておきます。

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
```

トポロジ構築スクリプトを実行したコンソールで、次のコマンドを実行してホスト 1 からホスト 2 へ ping を発行します。

```
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
```

tcpdumpの出力結果から、ARPがループしていないことが確認できます。

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
```

故障検出時のSTP再計算

次に、リンクダウンが起こった際のSTP再計算の動作を確認します。各OpenFlowスイッチ起動後のSTP計算が完了した状態で次のコマンドを実行し、ポートをダウンさせます。

Node: s2:

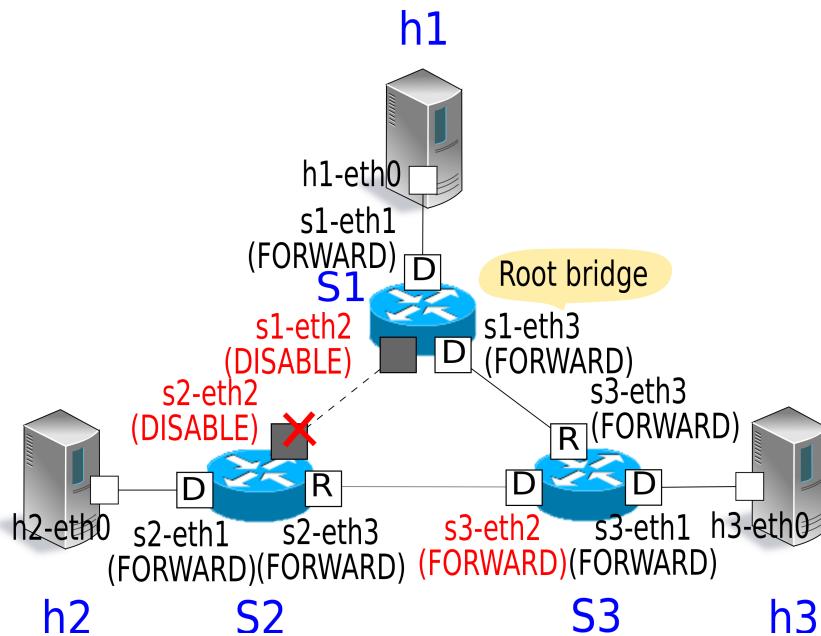
```
root@ryu-vm:~# ifconfig s2-eth2 down
```

リンクダウンが検出され、STP 再計算が実行されます。

```
[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=0000000000000003: [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
```

これまで BLOCK 状態だった s3-eth2 のポートが FORWARD 状態となり、再びフレーム転送可能な状態と

なったことが確認できます。



故障回復時の STP 再計算

続けて、リンクダウンが回復した際の STP 再計算の動作を確認します。リンクダウン中の状態で次のコマンドを実行し、ポートを起動させます。

Node: s2:

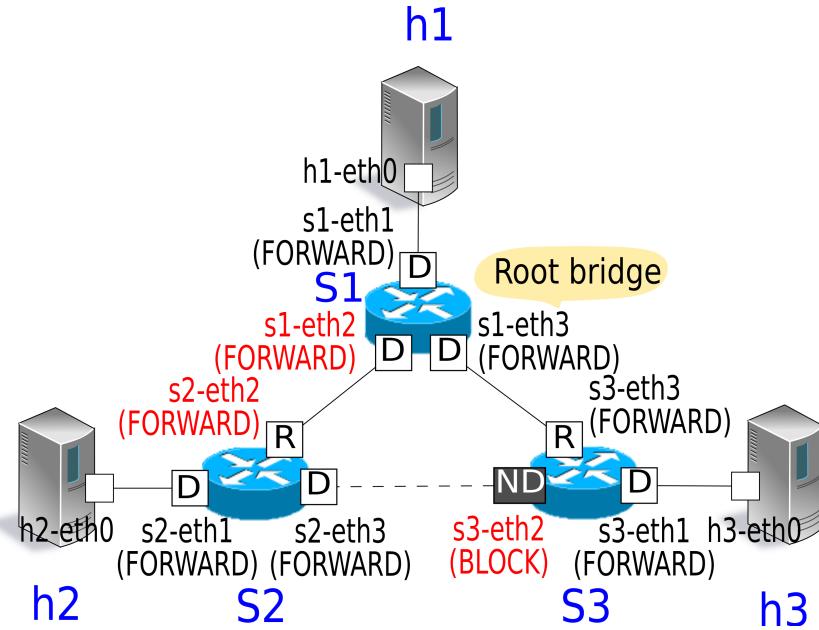
```
root@ryu-vm:~# ifconfig s2-eth2 up
```

リンク復旧が検出され、STP 再計算が実行されます。

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

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

アプリケーション起動時と同様のツリー構成となり、再びフレーム転送可能な状態となったことが確認できます。



5.3 OpenFlowによるスパニングツリー

Ryuのスパニングツリーアプリケーションにおいて、OpenFlowを用いてどのようにスパニングツリーの機能を実現しているかを見ていきます。

OpenFlow 1.3には次のようなポートの動作を設定するコンフィグが用意されています。Port ModificationメッセージをOpenFlowスイッチに発行することで、ポートのフレーム転送有無などの動作を制御することができます。

値	説明
OFPPC_PORT_DOWN	保守者により無効設定された状態です
OFPPC_NO_RECV	当該ポートで受信した全てのパケットを廃棄します
OFPPC_NO_FWD	当該ポートからパケット転送を行いません
OFPPC_NO_PACKET_IN	table-missとなった場合にPacket-Inメッセージを送信しません

また、ポート状態ごとのBPDUパケット受信とBPDU以外のパケット受信を制御するため、BPDUパケットをPacket-InするフローエントリとBPDU以外のパケットをdropするフローエントリをそれぞれFlow ModメッセージによりOpenFlowスイッチに登録します。

コントローラは各OpenFlowスイッチに対して、下記のようにポートコンフィグ設定とフローエントリ設定を行うことで、ポート状態に応じたBPDUパケットの送受信やMACアドレス学習(BPDU以外のパケット受信)、フレーム転送(BPDU以外のパケット送信)の制御を行います。

状態	ポートコンフィグ	フローエントリ
DISABLE	NO_RECV / NO_FWD	設定無し
BLOCK	NO_FWD	BPDU Packet-In / BPDU 以外 drop
LISTEN	設定無し	BPDU Packet-In / BPDU 以外 drop
LEARN	設定無し	BPDU Packet-In / BPDU 以外 drop
FORWARD	設定無し	BPDU Packet-In

ノート: Ryu に実装されているスパニングツリーのライブラリは、簡略化のため LEARN 状態での MAC アドレス学習 (BPDU 以外のパケット受信) を行っていません。

これらの設定に加え、コントローラは OpenFlow スイッチとの接続時に収集したポート情報や各 OpenFlow スイッチが受信した BPDU パケットに設定されたルートブリッジの情報を元に、送信用の BPDU パケットを構築し Packet-Out メッセージを発行することで、OpenFlow スイッチ間の BPDU パケットの交換を実現します。

5.4 Ryu によるスパニングツリーの実装

続いて、Ryu を用いて実装されたスパニングツリーのソースコードを見ていきます。スパニングツリーのソースコードは、Ryu のソースツリーにあります。

ryu/lib/stplib.py

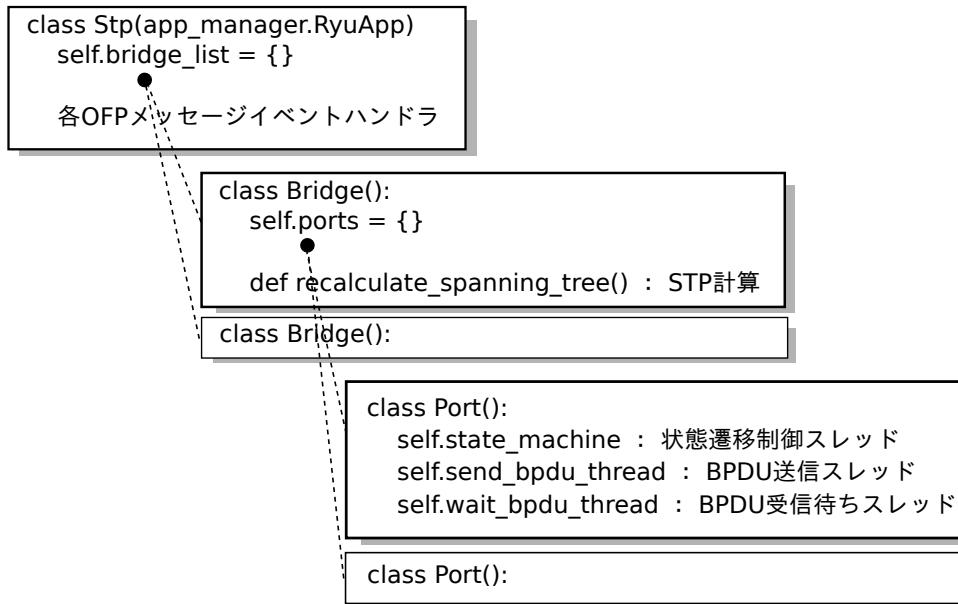
ryu/app/simple_switch_stp.py

stplib.py は BPDU パケットの交換や各ポートの役割・状態の管理などのスパニングツリー機能を提供するライブラリです。simple_switch_stp.py はスパニングツリーライブラリを適用することでスイッチングハブのアプリケーションにスパニングツリー機能を追加したアプリケーションプログラムです。

注意: simple_switch_stp.py は OpenFlow 1.0 専用のアプリケーションであるため、本章では「Ryu アプリケーションの実行」に示した OpenFlow 1.3 に対応した simple_switch_stp_13.py を元にアプリケーションの詳細を説明します。

5.4.1 ライブリの実装

ライブラリ概要



STP ライブリ (Stp クラスインスタンス) が OpenFlow スイッチのコントローラへの接続を検出すると、Bridge クラスインスタンス・Port クラスインスタンスが生成されます。各クラスインスタンスが生成・起動された後は、

- Stp クラスインスタンスからの OpenFlow メッセージ受信通知
- Bridge クラスインスタンスの STP 計算 (ルートプリッジ選択・各ポートの役割選択)
- Port クラスインスタンスのポート状態遷移・BPDU パケット送受信

が連動し、スパニングツリー機能を実現します。

コンフィグ設定項目

STP ライブリは `Stp.set_config()` メソッドによりブリッジ・ポートのコンフィグ設定 IF を提供します。設定可能な項目は以下の通りです。

- bridge

項目	説明	デフォルト値
<code>priority</code>	ブリッジ優先度	0x8000
<code>sys_ext_id</code>	VLAN-ID を設定 (*現状の STP ライブリは VLAN 未対応)	0
<code>max_age</code>	BPDU パケットの受信待ちタイマー値	20[sec]
<code>hello_time</code>	BPDU パケットの送信間隔	2 [sec]
<code>fwd_delay</code>	各ポートが LISTEN 状態および LEARN 状態に留まる時間	15[sec]

- port

項目	説明	デフォルト値
priority	ポート優先度	0x80
path_cost	リンクのコスト値	リンクスピードを元に自動設定
enable	ポートの有効無効設定	True

BPDU パケット送信

BPDU パケット送信は Port クラスの BPDU パケット送信スレッド (`Port.send_bpdu_thread`) で行っています。ポートの役割が指定ポート (`DESIGNATED_PORT`) の場合、ルートブリッジから通知された `hello_time` (`Port.port_times.hello_time`: デフォルト 2 秒) 間隔で BPDU パケット生成 (`Port._generate_config_bpdu()`) および BPDU パケット送信 (`Port.ofctl.send_packet_out()`) を行います。

```
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)
```

送信する BPDU パケットは、OpenFlow スイッチのコントローラ接続時に収集したポート情報 (`Port.ofport`) や受信した BPDU パケットに設定されたルートブリッジ情報 (`Port.port_priority`、`Port.port_times`)などを元に構築されます。

```
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
```

BPDU パケット受信

BPDU パケットの受信は、`Stp` クラスの `Packet-In` イベントハンドラによって検出され、`Bridge` クラスインスタンスを経由して `Port` クラスインスタンスに通知されます。イベントハンドラの実装は「[スイッチングハブ](#)」を参照してください。

BPDU パケットを受信したポートは、以前に受信した BPDU パケットと今回受信した BPDU パケットのブリッジ ID などの比較 (`Stp.compare_bpdu_info()`) を行い、STP 再計算の要否判定を行います。以前に受信した BPDU より優れた BPDU(SUPERIOR) を受信した場合、「新たなるルートブリッジが追加された」などのネットワークトポロジ変更が発生したことを意味するため、STP 再計算の契機となります。

```
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

```

故障検出

リンク断などの直接故障や、一定時間ルートブリッジからの BPDU パケットを受信できない間接故障を検出した場合も、STP 再計算の契機となります。

リンク断は Stp クラスの PortStatus イベントハンドラによって検出し、Bridge クラスインスタンスへ通知されます。

BPDU パケットの受信待ちタイムアウトは Port クラスの BPDU パケット受信待ちスレッド (Port.wait_bpdu_thread) で検出します。max age(デフォルト 20 秒) 間、ルートブリッジからの BPDU パケットを受信できない場合に間接故障と判断し、Bridge クラスインスタンスへ通知されます。

BPDU 受信待ちタイマーの更新とタイムアウトの検出には hub モジュール (ryu.lib.hub) の hub.Event と hub.Timeout を用います。hub.Event は hub.Event.wait() で wait 状態に入り hub.Event.set() が実行されるまでスレッドが中断されます。hub.Timeout は指定されたタイムアウト時間内に try 節の処理が終了しない場合、hub.Timeout 例外を発行します。hub.Event が wait 状態に入り hub.Timeout で指定されたタイムアウト時間内に hub.Event.set() が実行されない場合に、BPDU パケットの受信待ちタイムアウトと判断し Bridge クラスの STP 再計算処理を呼び出します。

```

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)

```

受信した BPDU パケットの比較処理 (Stp.compare_bpdu_info()) により SUPERIOR または REPEATED と判定された場合は、ルートブリッジからの BPDU パケットが受信出来ていることを意味するため、BPDU 受信待ちタイマーの更新 (Port._update_wait_bpdu_timer()) を行います。hub.Event である Port.wait_timer_event の set() 処理により Port.wait_timer_event は wait 状態から解放され、BPDU パケット受信待ちスレッド (Port.wait_bpdu_thread) は except hub.Timeout 節のタイムアウト処理に入ることなくタイマーをキャンセルし、改めてタイマーをセットし直すことでの BPDU パケットの受信待ちを開始します。

```

class Port(object):

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

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

        if ((rcv_info is SUPERIOR or rcv_info is REPEATED)

```

```

        and (self.role is ROOT_PORT
              or self.role is NON_DESIGNATED_PORT)):
    self._update_wait_bpdu_timer()

# ...

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

```

STP 計算

STP 計算(ルートブリッジ選択・各ポートの役割選択)は Bridge クラスで実行します。

STP 計算が実行されるケースではネットワークトポロジの変更が発生しておりパケットがループする可能性があるため、一旦全てのポートを BLOCK 状態に設定(port.down)し、かつトポロジ変更イベント(EventTopologyChange)を上位 API に対して通知することで学習済みの MAC アドレス情報の初期化を促します。

その後、Bridge._spanning_tree_algorithm() でルートブリッジとポートの役割を選択した上で、各ポートを LISTEN 状態で起動(port.up)しポートの状態遷移を開始します。

```

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)
```

ルートブリッジの選出のため、ブリッジ ID などの自身のブリッジ情報と各ポートが受信した BPDU パケットに設定された他ブリッジ情報を比較します (Bridge._select_root_port)。

この結果、ルートポートが見つかった場合 (自身のブリッジ情報よりもポートが受信した他ブリッジ情報が優れていた場合)、他ブリッジがルートブリッジであると判断し指定ポートの選出 (Bridge._select_designated_port) と非指定ポートの選出 (ルートポート / 指定ポート以外のポートを非指定ポートとして選出)を行います。

一方、ルートポートが見つからなかった場合 (自身のブリッジ情報が最も優れていた場合) は自身をルートブリッジと判断し各ポートは全て指定ポートとなります。

```
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

```

ポート状態遷移

ポートの状態遷移処理は、Port クラスの状態遷移制御スレッド (Port.state_machine) で実行しています。次の状態に遷移するまでのタイマーを Port._get_timer() で取得し、タイマー満了後に Port._get_next_state() で次状態を取得し、状態遷移を行います。また、STP 再計算が発生しこれまでのポート状態に関係無く BLOCK 状態に遷移させるケースなど、Port._change_status() が実行された場合にも状態遷移が行われます。これらの処理は「[故障検出](#)」と同様に hub モジュールの hub.Event と hub.Timeout を用いて実現しています。

```

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 アプリケーションの実装

「Ryu アプリケーションの実行」に示した OpenFlow 1.3 対応のスパニングツリーアプリケーション (simple_switch_stp_13.py) と、「スイッチングハブ」のスイッチングハブとの差異を順に説明していきます。

「_CONTEXTS」の設定

「リンク・アグリゲーション」と同様に STP ライブラリを利用するため CONTEXT を登録します。

```
from ryu.lib import stplib

# ...

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

# ...
```

コンフィグ設定

STP ライブラリの `set_config()` メソッドを用いてコンフィグ設定を行います。ここではサンプルとして、以下の値を設定します。

OpenFlow スイッチ	項目	設定値
dpid=0000000000000001	bridge.priority	0x8000
dpid=0000000000000002	bridge.priority	0x9000
dpid=0000000000000003	bridge.priority	0xa000

この設定により dpid=0000000000000001 の OpenFlow スイッチのプリッジ ID が常に最小の値となり、ルートプリッジに選択されることになります。

```
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 イベント処理

「リンク・アグリゲーション」と同様に STP ライブラリから通知されるイベントを受信するイベントハンドラを用意します。

STP ライブラリで定義された `stplib.EventPacketIn` イベントを利用してすることで、BPDU パケットを除いたパケットを受信することができるため、「スイッチングハブ」と同様のパケットハンドリングを行います。

```
class SimpleSwitch13(app_manager.RyuApp):

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

        # ...
```

ネットワークトポジの変更通知イベント (`stplib.EventTopologyChange`) を受け取り、学習した MAC アドレスおよび登録済みのフローエントリを初期化しています。

```
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.OFGPG_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]
```

ポート状態の変更通知イベント(stplib.EventPortStateChange)を受け取り、ポート状態のデバッグログ出力を行っています。

```
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])
```

以上のように、スパニングツリー機能を提供するライブラリと、ライブラリを利用するアプリケーションによって、スパニングツリー機能を持つスイッチングハブのアプリケーションを実現しています。

5.5 まとめ

本章では、スパニングツリーライブラリの利用を題材として、以下の項目について説明しました。

- hub.Event を用いたイベント待ち合わせ処理の実現方法
- hub.Timeout を用いたタイマー制御処理の実現方法

第 6 章

OpenFlow プロトコル

本章では、OpenFlow プロトコルで定義されている、マッチとインストラクションおよびアクションについて説明します。

6.1 マッチ

マッチに指定できる条件には様々なものがあり、OpenFlow のバージョンが上がる度にその種類は増えています。OpenFlow 1.0 では 12 種類でしたが、OpenFlow 1.3 では 40 種類もの条件が定義されています。

個々の詳細については、OpenFlow の仕様書などを参照して頂くとして、ここでは OpenFlow 1.3 の Match フィールドを簡単に紹介します。

Match フィールド名	説明
in_port	受信ポートのポート番号
in_phy_port	受信ポートの物理ポート番号
metadata	テーブル間で情報を受け渡すために用いられるメタデータ
eth_dst	Ethernet の宛先 MAC アドレス
eth_src	Ethernet の送信元 MAC アドレス
eth_type	Ethernet のフレームタイプ
vlan_vid	VLAN ID
vlan_pcp	VLAN PCP
ip_dscp	IP DSCP
ip_ecn	IP ECN
ip_proto	IP のプロトコル種別
ipv4_src	IPv4 の送信元 IP アドレス
ipv4_dst	IPv4 の宛先 IP アドレス
tcp_src	TCP の送信元ポート番号
tcp_dst	TCP の宛先ポート番号
udp_src	UDP の送信元ポート番号
udp_dst	UDP の宛先ポート番号

次のページに続く

表 6.1 – 前のページからの続き

Match フィールド名	説明
sctp_src	SCTP の送信元ポート番号
sctp_dst	SCTP の宛先ポート番号
icmpv4_type	ICMP の Type
icmpv4_code	ICMP の Code
arp_op	ARP のオペコード
arp_spa	ARP の送信元 IP アドレス
arp_tpa	ARP のターゲット IP アドレス
arp_sha	ARP の送信元 MAC アドレス
arp_tha	ARP のターゲット MAC アドレス
ipv6_src	IPv6 の送信元 IP アドレス
ipv6_dst	IPv6 の宛先 IP アドレス
ipv6_flabel	IPv6 のフローラベル
icmpv6_type	ICMPv6 の Type
icmpv6_code	ICMPv6 の Code
ipv6_nd_target	IPv6 ネイバーディスカバリのターゲットアドレス
ipv6_nd_sll	IPv6 ネイバーディスカバリの送信元リンクレイヤーアドレス
ipv6_nd_tll	IPv6 ネイバーディスカバリのターゲットリンクレイヤーアドレス
mpls_label	MPLS のラベル
mpls_tc	MPLS のトラフィッククラス (TC)
mpls_bos	MPLS の BoS ビット
pbb_isid	802.1ah PBB の I-SID
tunnel_id	論理ポートに関するメタデータ
ipv6_exthdr	IPv6 の拡張ヘッダの擬似フィールド

MAC アドレスや IP アドレスなどのフィールドによっては、さらにマスクを指定することができます。

6.2 インストラクション

インストラクションは、マッチに該当するパケットを受信した時の動作を定義するもので、次のタイプが規定されています。

インストラクション	説明
Goto Table (必須)	OpenFlow 1.1 以降では、複数のフローテーブルがサポートされています。Goto Table によって、マッチしたパケットの処理を、指定したフローテーブルに引き継ぐことができます。例えば、「ポート 1 で受信したパケットに VLAN-ID 200 を付加して、テーブル 2 へ飛ぶ」といったフローエントリが設定できます。 指定するテーブル ID は、現在のテーブル ID より大きい値でなければなりません。
Write Metadata (オプション)	以降のテーブルで参照できるメタデータをセットします。
Write Actions (必須)	現在のアクションセットに指定されたアクションを追加します。同じタイプのアクションが既にセットされていた場合には、新しいアクションで置き換えられます。
Apply Actions (オプション)	アクションセットは変更せず、指定されたアクションを直ちに適用します。
Clear Actions (オプション)	現在のアクションセットのすべてのアクションを削除します。
Meter (オプション)	指定したメーターにパケットを適用します。

Ryu では、各インストラクションに対応する次のクラスが実装されています。

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

Write/Apply/Clear Actions は、`OFPInstructionActions` にまとめられていて、インスタンス生成時に選択します。

ノート: Write Actions のサポートは必須とされていますが、現時点の Open vSwitch ではサポートされていません。Apply Actions がサポートされているので、代わりにこちらを使う必要があります。Write Actions は Open vSwitch 2.1.0 からサポートされる予定です。

6.3 アクション

`OFPActionOutput` クラスは、Packet-Out メッセージや Flow Mod メッセージで使用するパケット転送を指定するものです。コンストラクタの引数で転送先と、コントローラへ送信する場合は最大データサイズ (`max_len`) を指定します。転送先には、スイッチの物理的なポート番号の他にいくつかの定義された値が指定できます。

値	説明
OFPP_IN_PORT	受信ポートに転送されます
OFPP_TABLE	先頭のフローテーブルに摘要されます
OFPP_NORMAL	スイッチの L2/L3 機能で転送されます
OFPP_FLOOD	受信ポートやブロックされているポートを除く当該 VLAN 内のすべての物理ポートにフラッディングされます
OFPP_ALL	受信ポートを除くすべての物理ポートに転送されます
OFPP_CONTROLLER	コントローラに Packet-In メッセージとして送られます
OFPP_LOCAL	スイッチのローカルポートを示します
OFPP_ANY	Flow Mod(delete) メッセージや Flow Stats Requests メッセージでポートを選択する際にワイルドカードとして使用するもので、パケット転送では使用されません

max_len に 0 を指定すると、Packet-In メッセージにパケットのバイナリデータは添付されなくなります。 OFPCML_NO_BUFFER を指定すると、OpenFlow スイッチ上でそのパケットをバッファせず、Packet-In メッセージにパケット全体が添付されます。

第 7 章

ofproto ライブライ

本章では Ryu の ofproto ライブライについて紹介します。

7.1 概要

ofproto ライブライは OpenFlow プロトコルのメッセージの作成・解析を行なうためのライブライです。

7.2 モジュール構成

各 OpenFlow バージョン (バージョン X.Y) について、定数モジュール (ofproto_vX_Y) とパーサーモジュール (ofproto_vX_Y_parser) が用意されています。各 OpenFlow バージョンの実装は基本的に独立しています。

OpenFlow バージョン	定数モジュール	パーサーモジュール
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 定数モジュール

定数モジュールにはプロトコル定数の定義があります。例えば以下のようなものです。

定数	説明
OFP_VERSION	プロトコルバージョン番号
OFPP_XXXX	ポート番号
OFPCML_NO_BUFFER	バッファせずに、パケット全体を送信
OFP_NO_BUFFER	無効なバッファ番号

7.2.2 パーサーモジュール

パーサーモジュールには各 OpenFlow メッセージに対応したクラスが定義されています。例えば以下のようないます。これらのクラスとそのインスタンスを、今後メッセージクラス、メッセージオブジェクトと呼びます。

クラス	説明
OFPHello	OFPT_HELLO メッセージ
OFPPacketOut	OFPT_PACKET_OUT メッセージ
OFPFlowMod	OFPT_FLOW_MOD メッセージ

また、パーサーモジュールには OpenFlow メッセージのペイロード中で使われる構造体に対応するクラスも定義されています。例えば以下のようないます。これらのクラスとそのインスタンスを、今後構造体クラス、構造体オブジェクトと呼びます。

クラス	構造体
OFPMatch	ofp_match
OFPInstructionGotoTable	ofp_instruction_goto_table
OFPActionOutput	ofp_action_output

7.3 基本的な使い方

7.3.1 ProtocolDesc クラス

使用する OpenFlow プロトコルを指定するためのクラスです。メッセージクラスの`_init_`の datapath 引数には、このクラス（またはその派生クラスである Datapath クラス）のオブジェクトを指定します。

```
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 ネットワークアドレス

Ryu ofproto ライブライの API では、基本的に文字列表現のネットワークアドレスが使用されます。例えば以下のようないます。

ノート： ただし、OpenFlow 1.0 に関しては異なる表現が使用されています。（2014 年 2 月現在）

アドレス種別	python 文字列の例
MAC アドレス	'00:03:47:8c:a1:b3'
IPv4 アドレス	'192.0.2.1'
IPv6 アドレス	'2001:db8::2'

7.3.3 メッセージオブジェクトの生成

各メッセージクラス、構造体クラスのインスタンスを適切な引数で生成します。

引数の名前は、基本的に OpenFlow プロトコルで定められたフィールドの名前と同じです。ただし、python の予約語と衝突する場合は、最後に「_」を付けます。以下の例では「**type_**」がこれに当たります。

```
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)
```

ノート: 定数モジュール、パーサーモジュールは直接 import して使っても良いですが、使用する OpenFlow バージョンを変更する際に最小限の修正で済むよう、できるだけ ProtocolDesc オブジェクトの ofproto, ofproto_parser 属性を使用することを推奨します。

7.3.4 メッセージオブジェクトの解析

メッセージオブジェクトの内容を調べることができます。

例えば OFPPacketIn オブジェクト pid の match フィールドには pin.match としてアクセスできます。

OFPMatch オブジェクトの各 TLV には、以下のように名前でアクセスできます。

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

7.3.5 JSON

メッセージオブジェクトを json.dumps 互換の辞書に変換する機能と、json.loads 互換の辞書からメッセージオブジェクトを復元する機能があります。

ノート: ただし、OpenFlow 1.0 に関しては実装が不完全です。(2014 年 2 月現在)

```
import json

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

7.3.6 メッセージの解析 (パース)

メッセージのバイト列から、対応するメッセージオブジェクトを生成します。スイッチから受信したメッセージについては、フレームワークが自動的にこの処理を行なうため、Ryu アプリケーションが意識する必要はありません。

具体的には以下のようになります。

1. `ryu.ofproto.ofproto_parser.header` 関数を使用して、バージョン非依存部分を解析
2. 1. の結果を `ryu.ofproto.ofproto_parser.msg` 関数に渡して残りの部分を解析

7.3.7 メッセージの生成 (シリアル化)

メッセージオブジェクトから、対応するメッセージのバイト列を生成します。スイッチに送信するメッセージについては、フレームワークが自動的にこの処理を行なうため、Ryu アプリケーションが意識する必要はありません。

具体的には以下のようになります。

1. メッセージオブジェクトの `serialize` メソッドを呼び出す
2. メッセージオブジェクトの `buf` 属性を読み出す

‘len’ などのいくつかのフィールドは、明示的に値を指定しなくとも `serialize` 時に自動的に計算されます。

第 8 章

パケットライブラリ

OpenFlow の Packet-In や Packet-Out メッセージには、生のパケット内容をあらわすバイト列が入るフィールドがあります。Ryu には、このような生のパケットをアプリケーションから扱いやすくするためのライブラリが用意されています。本章はこのライブラリについて紹介します。

8.1 基本的な使い方

8.1.1 プロトコルヘッダクラス

Ryu パケットライブラリには、色々なプロトコルヘッダに対応するクラスが用意されています。

以下のものを含むプロトコルがサポートされています。各プロトコルに対応するクラスなどの詳細は [API リファレンス](#) をご参照ください。

- arp
- bgp
- bpdu
- dhcp
- ethernet
- icmp
- icmpv6
- igmp
- ipv4
- ipv6
- llc

- lldp
- mpls
- ospf
- pbb
- sctp
- slow
- tcp
- udp
- wlan
- vrrp

各プロトコルヘッダクラスの`__init__`引数名は、基本的には RFC などで使用されている名前と同じになっています。プロトコルヘッダクラスのインスタンス属性の命名規則も同様です。ただし、`type`など、Python built-in と衝突する名前のフィールドに対応する`__init__`引数名には、`type_`のように最後に`_`が付きます。

いくつかの`__init__`引数にはデフォルト値が設定されており省略できます。以下の例では `version=4` 等が省略されています。

```
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 ネットワークアドレス

Ryu パケットライブラリの API では、基本的に文字列表現のネットワークアドレスが使用されます。例えば以下のようなものです。

アドレス種別	python 文字列の例
MAC アドレス	'00:03:47:8c:a1:b3'
IPv4 アドレス	'192.0.2.1'
IPv6 アドレス	'2001:db8::2'

8.1.3 パケットの解析(パース)

パケットのバイト列から、対応する python オブジェクトを生成します。

具体的には以下のようになります。

1. ryu.lib.packet.packet.Packet クラスのオブジェクトを生成 (data 引数に解析するバイト列を指定)
2. 1. のオブジェクトの get_protocol メソッド等を使用して、各プロトコルヘッダに対応するオブジェクトを取得

```
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 パケットの生成(シリализ)

python オブジェクトから、対応するパケットのバイト列を生成します。

具体的には以下のようになります。

1. ryu.lib.packet.packet.Packet クラスのオブジェクトを生成
2. 各プロトコルヘッダに対応するオブジェクトを生成 (ethernet, ipv4, ...)
3. 1. のオブジェクトの add_protocol メソッドを使用して 2. のヘッダを順番に追加
4. 1. のオブジェクトの serialize メソッドを呼び出してバイト列を生成

チェックサムやペイロード長などのいくつかのフィールドは、明示的に値を指定しなくても serialize 時に自動的に計算されます。詳細は各クラスのリファレンスをご参照ください。

```
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
```

Scapy ライクな代替 API も用意されていますので、お好みに応じてご使用ください。

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

8.2 アプリケーション例

上記の例を使用して作成した、ping に返事をするアプリケーションを示します。

ARP REQUEST と ICMP ECHO REQUEST を Packet-In で受けとり、返事を Packet-Out で送信します。IP アドレス等は`__init__`メソッド内にハードコードされています。

```
# 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.OFPIstructionActions(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_ip4 = pkt.get_protocol(ipv4.ipv4)
    pkt_icmp = pkt.get_protocol(icmp.icmp)
    if pkt_icmp:
        self._handle_icmp(datapath, port, pkt_ether, pkt_ip4, pkt_icmp)
        return

def _handle_arp(self, datapath, port, pkt_ether, pkt_arp):
    if pkt_arp.opcode != arp.ARP_REQUEST:
        return
    pkt = packet.Packet()
    pkt.add_protocol(ether.ethernet(ethertype=pkt_ether.ethertype,
                                   dst=pkt_ether.src,
                                   src=self.hw_addr))
    pkt.add_protocol(arp.arp(opcode=arp.ARP_REPLY,
                           src_mac=self.hw_addr,
                           dst_mac=pkt_arp.src))
    datapath.send_msg(pkt)

```

```

        src_ip=self.ip_addr,
        dst_mac=pkt_arp.src_mac,
        dst_ip=pkt_arp.src_ip))
self._send_packet(datapath, port, pkt)

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

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

```

ノート: OpenFlow 1.2 以降では、Packet-In メッセージの match フィールドから、パース済みのパケットヘッダーの内容を取得できる場合があります。ただし、このフィールドにどれだけの情報を入れてくれるかは、スイッチの実装によります。例えば Open vSwitch は最低限の情報しか入れてくれませんので、多くの場合コントローラー側でパケット内容を解析する必要があります。一方 LINC は可能な限り多くの情報を入れてくれます。

以下は ping -c 3 を実行した場合のログの例です。

```

EVENT ofp_event->IcmpResponder EventOFPSwitchFeatures
switch features ev version: 0x4 msg_type 0x6 xid 0xb63c802c OFPSwitchFeatures(auxiliary_id=0,capabilities=0)
move onto main mode
EVENT ofp_event->IcmpResponder EventOFPPacketIn
packet-in ethernet(dst='ff:ff:ff:ff:ff:ff', ethertype=2054, src='0a:e4:1c:d1:3e:43'), arp(dst_ip='192.0.0.1')
packet-out ethernet(dst='0a:e4:1c:d1:3e:43', ethertype=2054, src='0a:e4:1c:d1:3e:44'), arp(dst_ip='192.0.0.1')
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=47390, data='ping')
packet-out ethernet(dst='0a:e4:1c:d1:3e:43', ethertype=2048, src='0a:e4:1c:d1:3e:44'), ipv4(csum=14140, data='pong')

```

```
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, ...
packet-out ethernet(dst='0a:e4:1c:d1:3e:43', ethertype=2048, src='0a:e4:1c:d1:3e:44'), ipv4(csum=14140, ...
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, ...
packet-out ethernet(dst='0a:e4:1c:d1:3e:43', ethertype=2048, src='0a:e4:1c:d1:3e:44'), ipv4(csum=14140, ...
```

IP フラグメント対応は読者への宿題とします。OpenFlow プロトコル自体には MTU を取得する方法がありませんので、ハードコードするか、何らかの工夫が必要です。また、Ryu パケットライブラリは常にパケット全体をパース/シリアル化しますので、フラグメント化されたパケットを処理するための API 変更が必要です。

第9章

OF-Config ライブライ

本章では、Ryu に付属している OF-Config のクライアントライブラリについて紹介します。

9.1 OF-Config プロトコル

OF-Config は OpenFlow スイッチの管理のためのプロトコルです。NETCONF(RFC 6241) のスキーマとして定義されており、論理スイッチ、ポート、キューなどの状態取得や設定を行なうことができます。

OpenFlow と同じ ONF が策定したもので、以下のサイトから仕様が入手できます。

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

本ライブラリは OF-Config 1.1.1 に準拠しています。

ノート: 現在 Open vSwitch は OF-Config をサポートしていませんが、同じ目的のために OVSDB というサービスを提供しています。OF-Config は比較的新しい規格で、Open vSwitch が OVSDB を実装したときにはまだ存在していませんでした。

OVSDB プロトコルは RFC 7047 として仕様が公開されていますが、事実上 Open vSwitch 専用のプロトコルとなっています。OF-Config はまだ登場から日が浅いですが、将来的に多くの OpenFlow スイッチで実装されることが期待されます。

9.2 ライブライ構成

9.2.1 `ryu.lib.of_config.capable_switch.OFCapableSwitch` クラス

NETCONF セッションを扱うためのクラスです。

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

9.2.2 ryu.lib.of_config.classes モジュール

設定内容を python オブジェクトとして扱うためのクラス群を提供するモジュールです。

ノート: クラス名は基本的に OF-Config 1.1.1 の yang specification 上の grouping キーワードで使われている名前と同じです。例. OFPortType

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

9.3 使用例

9.3.1 スイッチへの接続

SSH トランスポートを使用してスイッチに接続します。unknown_host_cb には、不明な SSH ホスト鍵の処理を行なうコールバック関数を指定しますが、ここでは無条件に接続を継続するようにしています。

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

9.3.2 GET

NETCONF GET を使用して状態を取得する例です。全てのポートの/resources/port/resource-id と /resources/port/current-rate を表示します。

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

9.3.3 GET-CONFIG

NETCONF GET-CONFIG を使用して設定を取得する例です。

ノート: running というのは NETCONF のデータストアで、現在動作している設定です。実装によりますが、他にも startup(デバイスの起動時に読み込まれる設定) や candidate(候補設定) などのデータストアが利用できます。

全てのポートの/resources/port/resource-id と/resources/port/configuration/admin-state を表示します。

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

9.3.4 EDIT-CONFIG

NETCONF EDIT-CONFIG を使用して設定を変更する例です。基本的に、GET-CONFIG で取得した設定を編集して EDIT-CONFIG で送り返す、という手順になります。

ノート： プロトコル上は EDIT-CONFIG で設定の部分的な編集を行なうこともできますが、このような使い方が無難です。

全てのポートの/resources/port/configuration/admin-state を down に設定します。

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

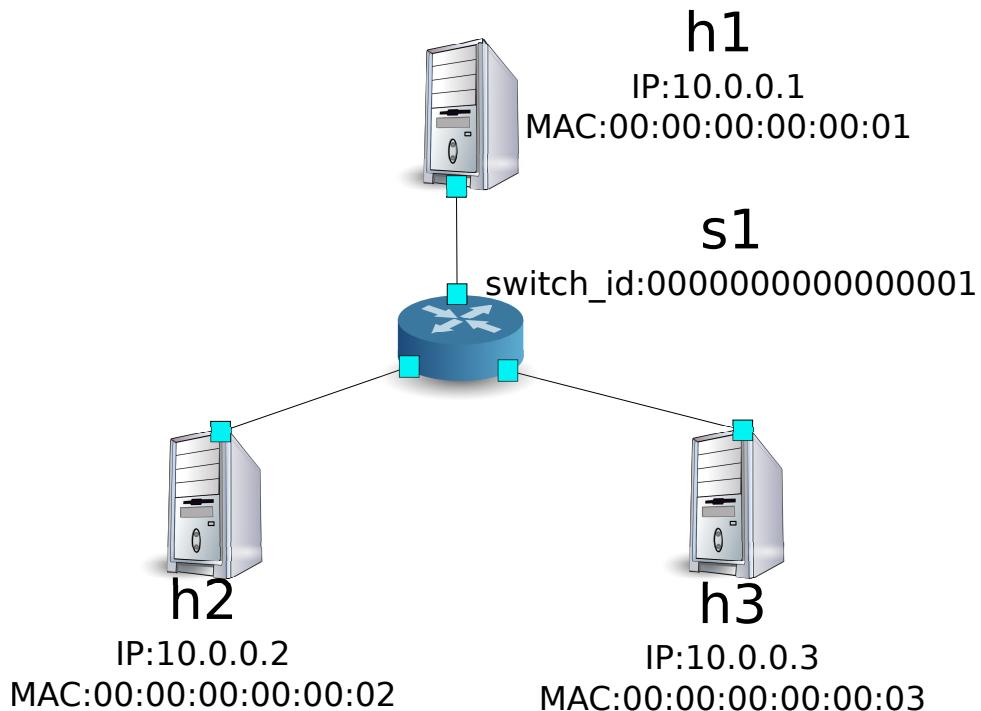

第 10 章

ファイアウォール

本章では、REST で設定が出来るファイアウォールの使用方法について説明します。

10.1 シングルテナントでの動作例

以下のようなトポロジを作成し、スイッチ s1 に対してルールの追加・削除を行う例を紹介します。



10.1.1 環境構築

まずは Mininet 上に環境を構築します。入力するコマンドは「[スイッチングハブ](#)」と同様です。

```
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>
```

また、コントローラ用の xterm をもうひとつ起動しておきます。

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

続いて、使用する OpenFlow のバージョンを 1.3 に設定します。

switch: s1 (root):

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

最後に、コントローラの xterm 上で rest_firewall を起動させます。

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 とスイッチの間の接続に成功すると、次のメッセージが表示されます。

controller: c0 (root):

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

10.1.2 初期状態の変更

firewall の起動直後は、すべての通信を遮断するよう無効状態となっています。次のコマンドで有効 (enable) にします。

ノート: 以降の説明で使用する REST API の詳細は、章末の「REST API 一覧」を参照してください。

Node: c0 (root):

```
root@ryu-vm:~# curl -X PUT http://localhost:8080/firewall/module/enable/00000000000000000000
[
  {
    "switch_id": "00000000000000000001",
    "command_result": {
      "result": "success",
      "details": "firewall running."
    }
  }
]
```

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

ノート: REST コマンドの実行結果は見やするように整形しています。

h1 から h2 への ping の疎通を確認してみます。しかし、アクセス許可のルールを設定していないため遮断されてしまいます。

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
```

遮断されたパケットはログに出力されます。

controller: c0 (root):

```
[FW] [INFO] dpid=0000000000000001: Blocked packet = ethernet(dst='00:00:00:00:00:02', ethertype=2048, s...
```

10.1.3 ルール追加

h1 と h2 の間で ping を許可するルールを追加します。双方向にルールを追加をする必要があります。

次のルールを追加してみましょう。ルール ID は自動採番されます。

送信元	宛先	プロトコル	可否	(ルール ID)
10.0.0.1/32	10.0.0.2/32	ICMP	許可	1
10.0.0.2/32	10.0.0.1/32	ICMP	許可	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"}' [ { "switch_id": "0000000000000001", "command_result": [ { "result": "success", "details": "Rule added. : rule_id=1" } ] } ]
```

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

追加したルールがフローエントリとしてスイッチに登録されます。

switch: s1 (root):

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

また、h2 と h3 の間で、ping を含むすべての IPv4 パケットを許可するようルールを追加します。

送信元	宛先	プロトコル	可否	(ルール ID)
10.0.0.2/32	10.0.0.3/32	any	許可	3
10.0.0.3/32	10.0.0.2/32	any	許可	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/api/v1/rules
[{"switch_id": "0000000000000001", "command_result": [{"result": "success", "details": "Rule added. : rule_id=3"}]}]

root@ryu-vm:~# curl -X POST -d '{"nw_src": "10.0.0.3/32", "nw_dst": "10.0.0.2/32"}' http://localhost:8080/api/v1/rules
[{"switch_id": "0000000000000001", "command_result": [{"result": "success", "details": "Rule added. : rule_id=4"}]}]
```

追加したルールがフローエントリとしてスイッチに登録されます。

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
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
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
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
```

ルールには優先度を設定することが出来ます。

h2 と h3 の間で ping(ICMP) を遮断するルールを追加してみましょう。優先度としてデフォルト値の 1 より大きい値を設定します。

(優先度)	送信元	宛先	プロトコル	可否	(ルール ID)
10	10.0.0.2/32	10.0.0.3/32	ICMP	遮断	5
10	10.0.0.3/32	10.0.0.2/32	ICMP	遮断	6

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"nw_src": "10.0.0.2/32", "nw_dst": "10.0.0.3/32", "nw_proto": "ICMP", "priority": 10, "action": "deny", "rule_id": 5}' http://127.0.0.1:8080/api/v1/rules
[{"switch_id": "0000000000000001", "command_result": [{"result": "success", "details": "Rule added. : rule_id=5"}]}]

root@ryu-vm:~# curl -X POST -d '{"nw_src": "10.0.0.3/32", "nw_dst": "10.0.0.2/32", "nw_proto": "ICMP", "priority": 10, "action": "deny", "rule_id": 6}' http://127.0.0.1:8080/api/v1/rules
[{"switch_id": "0000000000000001", "command_result": [{"result": "success", "details": "Rule added. : rule_id=6"}]}]
```

追加したルールがフローエントリとしてスイッチに登録されます。

switch: s1 (root):

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x3, duration=242.155s, table=0, n_packets=0, n_bytes=0, priority=1, ip, nw_src=10.0.0.2, nw_dst=10.0.0.3
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
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=10.0.0.1
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
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
cookie=0x2, duration=564.793s, table=0, n_packets=0, n_bytes=0, priority=1, icmp, nw_src=10.0.0.2, nw_dst=10.0.0.3
```

10.1.4 ルール確認

設定されているルールを確認します。

Node: c0 (root):

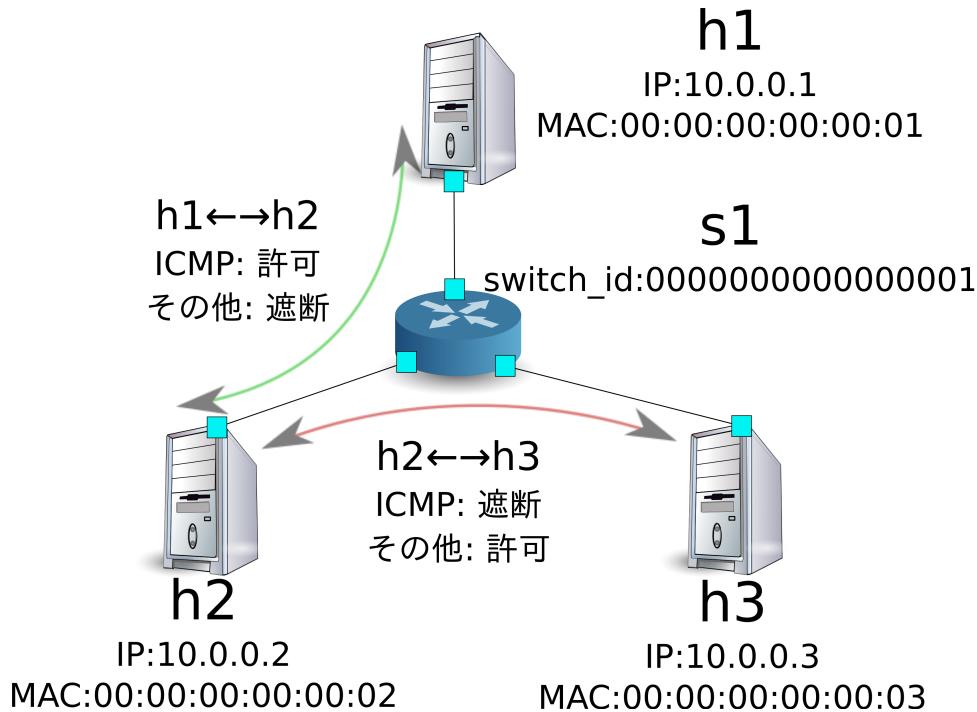
```
root@ryu-vm:~# curl http://localhost:8080/firewall/rules/00000000000000000001
[
  {
    "access_control_list": [
      {
        "rules": [
          {
            "priority": 1,
            "dl_type": "IPv4",
            "nw_dst": "10.0.0.3",
            "nw_src": "10.0.0.2",
            "rule_id": 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"
          },
          {
            "priority": 1,
            "dl_type": "IPv4",
            "nw_proto": "ICMP",
            "nw_dst": "10.0.0.2",
            "nw_src": "10.0.0.1",
            "rule_id": 1,
            "actions": "ALLOW"
          },
          {
            "priority": 10,
            "dl_type": "IPv4",
            "nw_proto": "ICMP",
            "nw_dst": "10.0.0.2",
            "nw_src": "10.0.0.3",
            "rule_id": 6,
            "actions": "DENY"
          }
        ]
      }
    ]
  }
]
```

```

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

```

設定したルールを図示すると以下のようになります。



h1 から h2 に ping を実行してみます。許可するルールが設定されているので、ping が疎通します。

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
...

```

h1 から h2 への ping 以外のパケットは firewall によって遮断されます。例えば h1 から h2 に wget を実行すると、パケットが遮断された旨ログが出力されます。

host: h1:

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

controller: c0 (root):

```
[FW] [INFO] dpid=0000000000000001: Blocked packet = ethernet(dst='00:00:00:00:00:02', ethertype=2048, si...
```

h2 と h3 の間は ping 以外のパケットの疎通が可能となっています。例えば h2 から h3 に ssh を実行すると、パケットが遮断された旨のログは出力されません (h3 で sshd が動作していないため、ssh での接続には失敗します)。

host: h2:

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

h2 から h3 に ping を実行すると、パケットが firewall によって遮断された旨ログが出力されます。

host: h2:

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

controller: c0 (root):

```
[FW] [INFO] dpid=0000000000000001: Blocked packet = ethernet(dst='00:00:00:00:00:03', ethertype=2048, si...
```

10.1.5 ルール削除

“rule_id:5” および “rule_id:6” のルールを削除します。

Node: c0 (root):

```
root@ryu-vm:~# curl -X DELETE -d '{"rule_id": "5"}' http://localhost:8080/firewall/rules/00000000000000000000000000000000
[
  {
    "switch_id": "0000000000000001",
    "command_result": [
      {
        "rule_id": "5"
      }
    ]
  }
]
```

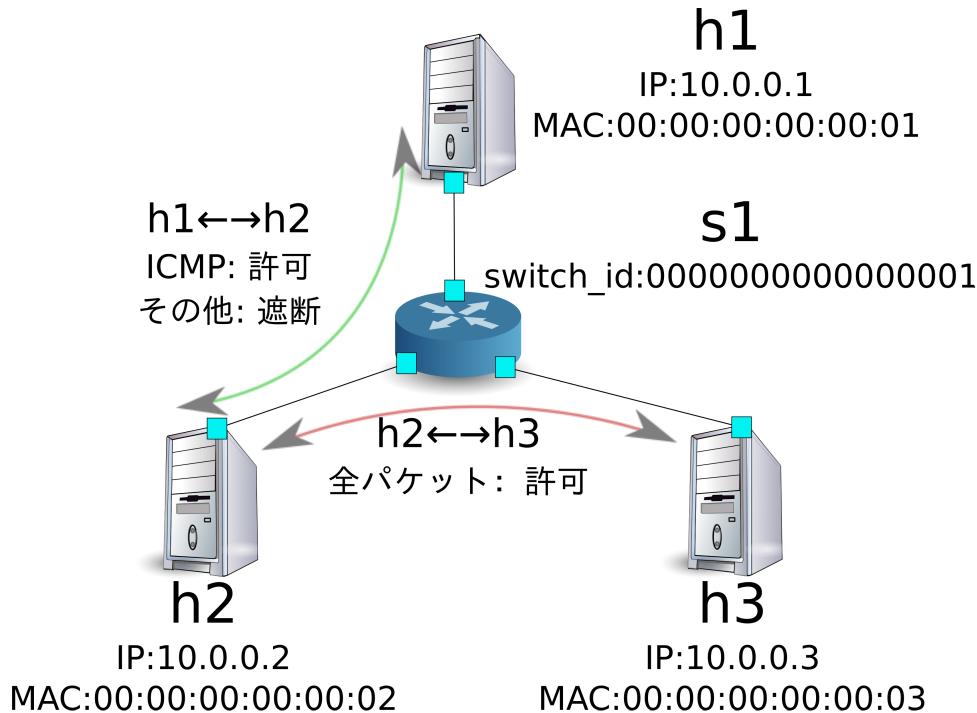
```

        "result": "success",
        "details": "Rule deleted. : ruleID=5"
    }
]
}
]

root@ryu-vm:~# curl -X DELETE -d '{"rule_id": "6"}' http://localhost:8080/firewall/rules/0000000000000000
[
{
    "switch_id": "0000000000000001",
    "command_result": [
        {
            "result": "success",
            "details": "Rule deleted. : ruleID=6"
        }
    ]
}
]

```

現在のルールを図示すると以下のようになります。



実際に確認します。h2 と h3 の間の ping(ICMP) を遮断するルールが削除されたため、ping が疎通できるようになりましたことがわかります。

host: h2:

```

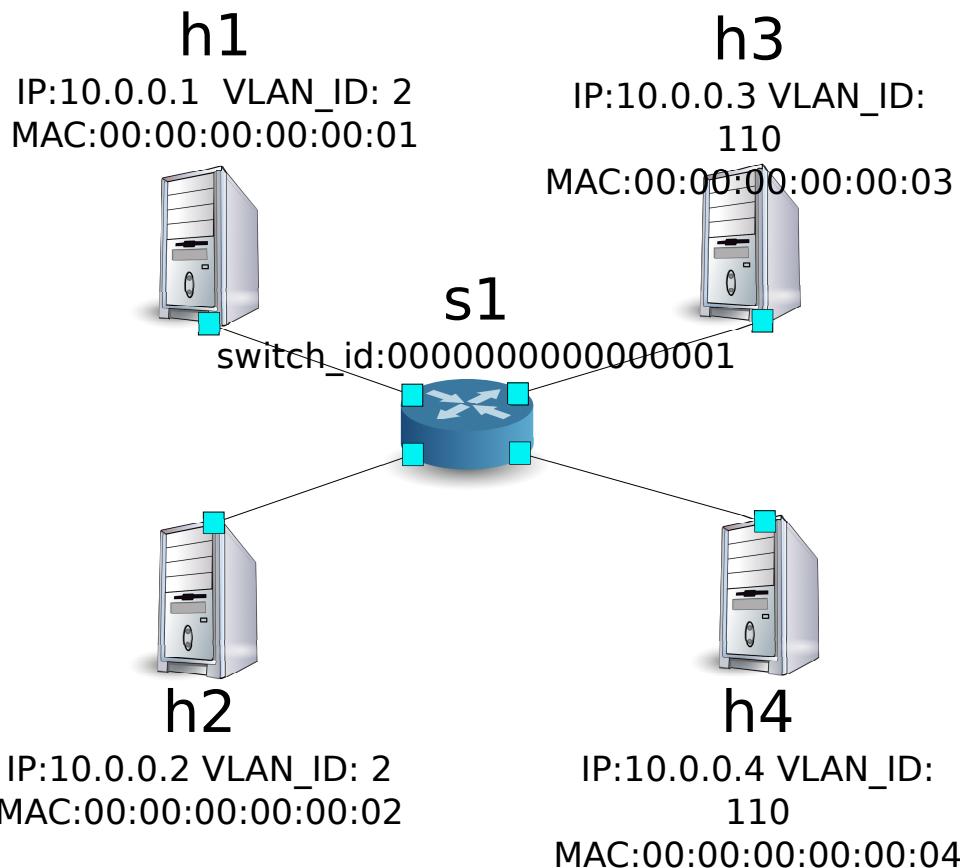
root@ryu-vm:~# ping 10.0.0.3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_req=1 ttl=64 time=0.841 ms

```

```
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 マルチテナントでの動作例

続いて、VLAN によるテナント分けが行われている以下のようなトポロジを作成し、スイッチ s1 に対してルールの追加・削除を行い、各ホスト間の疎通可否を確認する例を紹介します。



10.2.1 環境構築

シングルテナントでの例と同様、Mininet 上に環境を構築し、コントローラ用の xterm をもうひとつ起動しておきます。使用するホストがひとつ増えていることにご注意ください。

```
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>
```

続いて、各ホストのインターフェースに VLAN ID を設定します。

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
```

さらに、使用する OpenFlow のバージョンを 1.3 に設定します。

switch: s1 (root):

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

最後に、コントローラの xterm 上で rest_firewall を起動させます。

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 とスイッチの間の接続に成功すると、次のメッセージが表示されます。

controller: c0 (root):

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

10.2.2 初期状態の変更

firewall を有効 (enable) にします。

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 ルール追加

vlan_id=2 に 10.0.0.0/8 で送受信される ping(ICMP パケット) を許可するルールを追加します。双方向にルールを設定する必要がありますので、ルールを二つ追加します。

(優先度)	VLAN ID	送信元	宛先	プロトコル	可否	(ルール ID)
1	2	10.0.0.0/8	any	ICMP	許可	1
1	2	any	10.0.0.0/8	ICMP	許可	2

Node: c0 (root):

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

10.2.4 ルール確認

設定されているルールを確認します。

Node: c0 (root):

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

```

        "dl_vlan": 2,
        "nw_src": "10.0.0.0/8",
        "rule_id": 1,
        "actions": "ALLOW"
    },
    {
        "priority": 1,
        "dl_type": "IPv4",
        "nw_proto": "ICMP",
        "nw_dst": "10.0.0.0/8",
        "dl_vlan": 2,
        "rule_id": 2,
        "actions": "ALLOW"
    }
],
"vlan_id": 2
}
],
"switch_id": "0000000000000001"
}
]

```

実際に確認してみます。vlan_id=2 である h1 から、同じく vlan_id=2 である h2 に対し、ping を実行すると、追加したルールのとおり疎通できることがわかります。

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 同士である h3 と h4 の間は、ルールが登録されていないため、ping パケットは遮断されます。

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

```

パケットが遮断されたのでログが出力されます。

controller: c0 (root):

```

[FW] [INFO] dpid=0000000000000001: Blocked packet = ethernet(dst='00:00:00:00:00:04', ethertype=33024, s
...

```

本章では、具体例を挙げながらファイアウォールの使用方法を説明しました。

10.3 REST API 一覧

本章で紹介した rest_firewall の REST API 一覧です。

10.3.1 全スイッチの有効無効状態の取得

メソッド	GET
URL	/firewall/module/status

10.3.2 各スイッチの有効無効状態の変更

メソッド	PUT
URL	/firewall/module/{op}/{switch} -op: [“enable” “disable”] -switch: [“all” スイッチ ID]
備考	各スイッチの初期状態は”disable” になっています。

10.3.3 全ルールの取得

メソッド	GET
URL	/firewall/rules/{switch}[/{vlan}] -switch: [“all” スイッチ ID] -vlan: [“all” VLAN ID]
備考	VLAN ID の指定はオプションです。

10.3.4 ルールの追加

メソッド	POST
URL	/firewall/rules/{switch}[/{vlan}] -switch: [“all” スイッチ ID] -vlan: [“all” VLAN ID]
データ	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”]
備考	登録に成功するとルール ID が生成され、応答に記載されます。 VLAN ID の指定はオプションです。

10.3.5 ルールの削除

メソッド	DELETE
URL	/firewall/rules/{switch}[/{vlan}] -switch: [“all” スイッチ ID] -vlan: [“all” VLAN ID]
データ	rule_id:[“all” 1 - ...]
備考	VLAN ID の指定はオプションです。

10.3.6 全スイッチのログ出力状態の取得

メソッド	GET
URL	/firewall/log/status

10.3.7 各スイッチのログ出力状態の変更

メソッド	PUT
URL	/firewall/log/{op}/{switch} -op: [“enable” “disable”] -switch: [“all” スイッチ ID]
備考	各スイッチの初期状態は”enable” になっています。

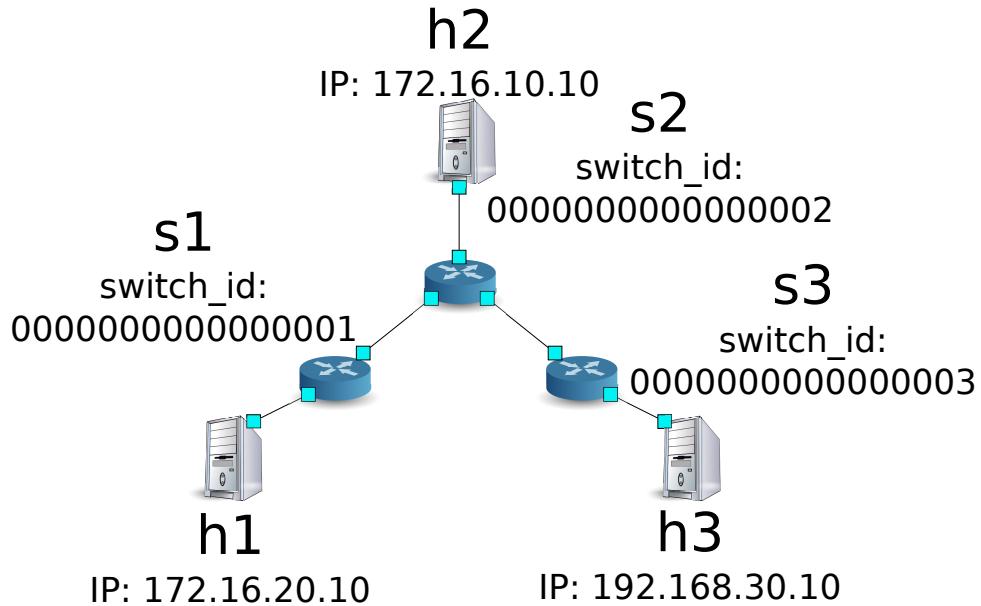
第 11 章

ルータ

本章では、REST で設定が出来るルータの使用方法について説明します。

11.1 シングルテナントでの動作例

以下のようなトポロジを作成し、各スイッチ（ルータ）に対してアドレスやルートの追加・削除を行い、各ホスト間の疎通可否を確認する例を紹介します。



11.1.1 環境構築

まずは Mininet 上に環境を構築します。mn コマンドのパラメータは以下のようになります。

パラメータ	値	説明
topo	linear,3	3 台のスイッチが直列に接続されているトポロジ
mac	なし	自動的にホストの MAC アドレスをセットする
switch	ovsk	Open vSwitch を使用する
controller	remote	OpenFlow コントローラは外部のものを利用する
x	なし	xterm を起動する

実行例は以下のようになります。

```
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>
```

また、コントローラ用の xterm をもうひとつ起動しておきます。

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

続いて、各ルータで使用する OpenFlow のバージョンを 1.3 に設定します。

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
```

その後、各ホストで自動的に割り当てられている IP アドレスを削除し、新たに IP アドレスを設定します。

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
```

最後に、コントローラの xterm 上で rest_router を起動させます。

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/
```

Ryu とルータの間の接続に成功すると、次のメッセージが表示されます。

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.
...
```

上記ログがルータ 3 台分表示されれば準備完了です。

11.1.2 アドレスの設定

各ルータにアドレスを設定します。

まず、ルータ s1 にアドレス「172.16.20.1/24」と「172.16.30.30/24」を設定します。

ノート: 以降の説明で使用する REST API の詳細は、章末の「REST API 一覧」を参照してください。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"address":"172.16.20.1/24"}' http://localhost:8080/router/00000000000000000000000000000000
[{"switch_id": "00000000000000000000000000000001", "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/00000000000000000000000000000000
[{"switch_id": "00000000000000000000000000000001", "command_result": [{"result": "success", "details": "Add address [address_id=2]"}]}]
```

ノート: REST コマンドの実行結果は見やすいように整形しています。

続いて、ルータ s2 にアドレス「172.16.10.1/24」「172.16.30.1/24」「192.168.10.1/24」を設定します。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"address": "172.16.10.1/24"}' http://localhost:8080/router/00000000000000000000000000000002
[{"switch_id": "00000000000000000000000000000002", "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/00000000000000000000000000000002
[{"switch_id": "00000000000000000000000000000002", "command_result": [{"result": "success", "details": "Add address [address_id=2]"}]}
```

```

        "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/00000000
[
{
    "switch_id": "00000000000000000002",
    "command_result": [
        {
            "result": "success",
            "details": "Add address [address_id=3]"
        }
    ]
}
]

```

さらに、ルータ s3 にアドレス「192.168.30.1/24」と「192.168.10.20/24」を設定します。

Node: c0 (root):

```

root@ryu-vm:~# curl -X POST -d '{"address": "192.168.30.1/24"}' http://localhost:8080/router/00000000
[
{
    "switch_id": "00000000000000000003",
    "command_result": [
        {
            "result": "success",
            "details": "Add address [address_id=1]"
        }
    ]
}
]

```

```

root@ryu-vm:~# curl -X POST -d '{"address": "192.168.10.20/24"}' http://localhost:8080/router/00000000
[
{
    "switch_id": "00000000000000000003",
    "command_result": [
        {
            "result": "success",
            "details": "Add address [address_id=2]"
        }
    ]
}
]
```

ルータへの IP アドレスの設定ができたので、各ホストにデフォルトゲートウェイとして登録します。

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 デフォルトルートの設定

各ルータにデフォルトルートを設定します。

まず、ルータ s1 のデフォルトルートとしてルータ s2 を設定します。

Node: c0 (root);

```
root@ryu-vm:~# curl -X POST -d '{"gateway": "172.16.30.1"}' http://localhost:8080/router/0000000000000000
[
{
  "switch_id": "0000000000000001",
  "command_result": [
    {
      "result": "success",
      "details": "Add route [route_id=1]"
    }
  ]
}
]
```

ルータ s2 のデフォルトルートにはルータ s1 を設定します。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"gateway": "172.16.30.30"}' http://localhost:8080/router/00000000000000000000000000000000
[{"switch_id": "00000000000000000000000000000002", "command_result": [{"result": "success", "details": "Add route [route_id=1]"}]}]
```

ルータ s3 のデフォルトルートにはルータ s2 を設定します。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"gateway": "192.168.10.1"}' http://localhost:8080/router/00000000000000000000000000000000
[
{
  "switch_id": "00000000000000000000000000000003",
  "command_result": [
    {
      "result": "success",
      "details": "Add route [route_id=1]"
    }
  ]
}
]
```

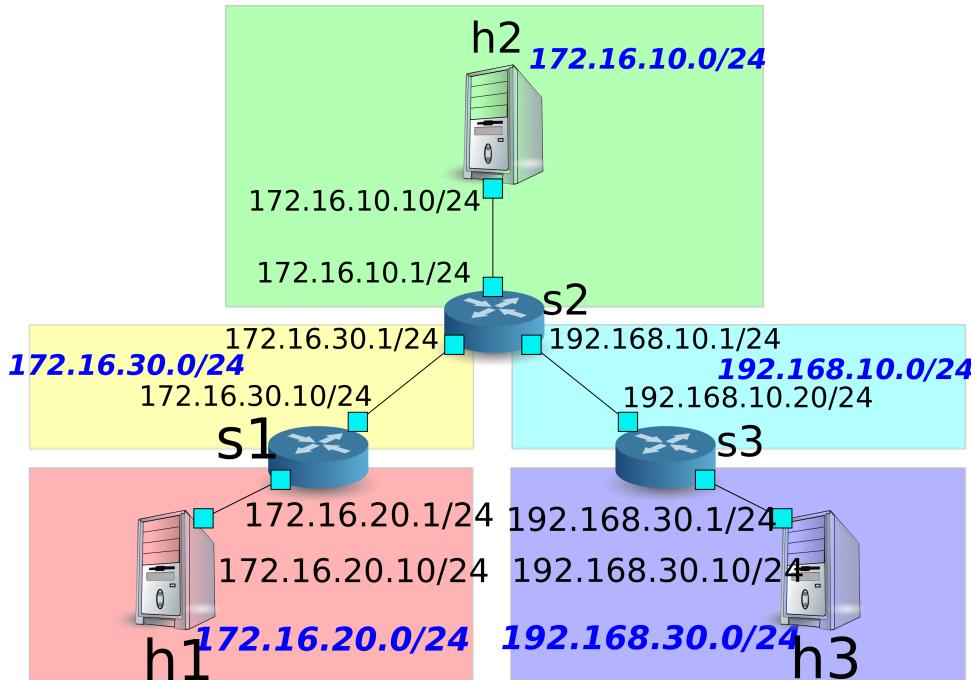
11.1.4 静的ルートの設定

ルータ s2 に対し、ルータ s3 配下のホスト (192.168.30.0/24) へのスタティックルートを設定します。

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/00000000000000000000000000000002
[
{
  "switch_id": "00000000000000000000000000000002",
  "command_result": [
    {
      "result": "success",
      "details": "Add route [route_id=2]"
    }
  ]
}
]
```

アドレスやルートの設定状態は、次のようにになります。



11.1.5 設定内容の確認

各ルータに設定された内容を確認します。

Node: c0 (root):

```
root@ryu-vm:~# curl http://localhost:8080/router/00000000000000000000000000000001
[
  {
    "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": "00000000000000000001"
}
]

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": "00000000000000000002"
}
]

root@ryu-vm:~# curl http://localhost:8080/router/00000000000000000003
[
{
  "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"
}
]
```

この状態で、ping による疎通を確認してみます。まず、h2 から h3 へ ping を実行します。正常に疎通できることが確認できます。

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
...
```

また、h2 から h1 へ ping を実行します。こちらも正常に疎通できることが確認できます。

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 静的ルートの削除

ルータ s2 に設定したルータ s3 へのスタティックルートを削除します。

Node: c0 (root):

```
root@ryu-vm:~# curl -X DELETE -d '{"route_id": "2"}' http://localhost:8080/router/00000000000000000002
[
{
  "switch_id": "0000000000000002",
  "command_result": [
    {
      "result": "success",
      "details": "Delete route [route_id=2]"
    }
  ]
}
]
```

ルータ s2 に設定された情報を確認してみます。ルータ s3 へのスタティックルートが削除されていることがわかります。

Node: c0 (root):

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

この状態で、ping による疎通を確認してみます。h2 から h3 へはルート情報がなくなったため、疎通できない

ことがわかります。

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 アドレスの削除

ルータ s1 に設定したアドレス「172.16.20.1/24」を削除します。

Node: c0 (root):

```
root@ryu-vm:~# curl -X DELETE -d '{"address_id": "1"}' http://localhost:8080/router/00000000000000000000000000000001
[
  {
    "switch_id": "00000000000000000000000000000001",
    "command_result": [
      {
        "result": "success",
        "details": "Delete address [address_id=1]"
      }
    ]
  }
]
```

ルータ s1 に設定された情報を確認してみます。ルータ s1 に設定された IP アドレスのうち、「172.16.20.1/24」が削除されていることがわかります。

Node: c0 (root):

```
root@ryu-vm:~# curl http://localhost:8080/router/00000000000000000000000000000001
[
  {
    "internal_network": [
      {
        "route": [
          {
            "route_id": 1,
            "destination": "0.0.0.0/0",
            "gateway": "172.16.30.1"
          }
        ],
        "address": [
          {
            "address_id": 2,
            "address": "172.16.30.30/24"
          }
        ]
      }
    ]
  }
]
```

```

        }
    ]
}
],
"switch_id": "0000000000000001"
}
]

```

この状態で、ping による疎通を確認してみます。h2 から h1 へは、h1 の所属するサブネットに関する情報がルータ s1 から削除されたため、疎通できないことがわかります。

host: h2:

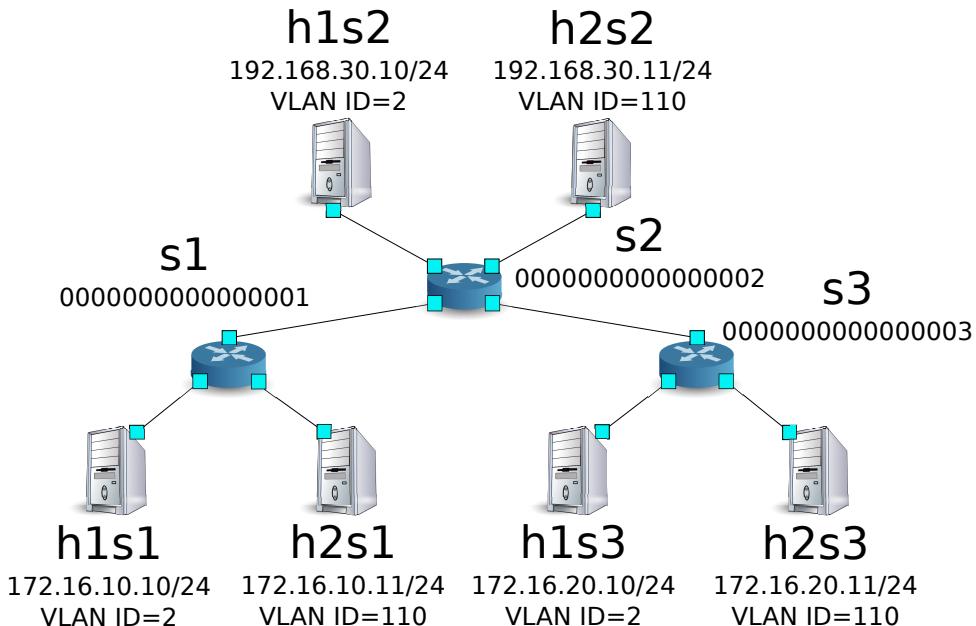
```

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 マルチテナントでの動作例

続いて、VLAN によるテナント分けが行われている以下のようなトポロジを作成し、各スイッチ（ルータ）に対してアドレスやルートの追加・削除を行い、各ホスト間の疎通可否を確認する例を紹介します。



11.2.1 環境構築

まずは Mininet 上に環境を構築します。mn コマンドのパラメータは以下のようになります。

パラメータ	値	説明
topo	linear,3,2	3 台のスイッチが直列に接続されているトポロジ (各スイッチに 2 台のホストが接続される)
mac	なし	自動的にホストの MAC アドレスをセットする
switch	ovsk	Open vSwitch を使用する
controller	remote	OpenFlow コントローラは外部のものを利用する
x	なし	xterm を起動する

実行例は以下のようになります。

```
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>
```

また、コントローラ用の xterm をもうひとつ起動しておきます。

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

続いて、各ルータで使用する OpenFlow のバージョンを 1.3 に設定します。

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
```

その後、各ホストのインターフェースに VLAN ID を設定し、新たに IP アドレスを設定します。

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
```

最後に、コントローラの xterm 上で rest_router を起動させます。

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 とルータの間の接続に成功すると、次のメッセージが表示されます。

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.
...
```

上記ログがルータ 3 台分表示されれば準備完了です。

11.2.2 アドレスの設定

各ルータにアドレスを設定します。

まず、ルータ s1 にアドレス「172.16.20.1/24」と「10.10.10.1/24」を設定します。それぞれ VLAN ID ごとに設定する必要があります。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"address": "172.16.10.1/24"}' http://localhost:8080/router/0000000000000001
[
  {
    "switch_id": "0000000000000001",
    "command_result": [
      {
        "result": "success",
        "vlan_id": 2,
        "details": "Add address [address_id=1]"
      }
    ]
  }
]
```

```
root@ryu-vm:~# curl -X POST -d '{"address": "10.10.10.1/24"}' http://localhost:8080/router/0000000000000001
[
  {
    "switch_id": "0000000000000001",
    "command_result": [
      {
        "result": "success",
        "vlan_id": 2,
        "details": "Add address [address_id=2]"
      }
    ]
  }
]
```

```

        }
    ]
}
]
]

root@ryu-vm:~# curl -X POST -d '{"address": "172.16.10.1/24"}' http://localhost:8080/router/00000000000000000000000000000000
[
{
  "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/00000000000000000000000000000000
[
{
  "switch_id": "00000000000000000000000000000001",
  "command_result": [
    {
      "result": "success",
      "vlan_id": 110,
      "details": "Add address [address_id=2]"
    }
  ]
}
]
]
```

続いて、ルータ s2 にアドレス「192.168.30.1/24」と「10.10.10.2/24」を設定します。

Node: c0 (root):

```

root@ryu-vm:~# curl -X POST -d '{"address": "192.168.30.1/24"}' http://localhost:8080/router/00000000000000000000000000000000
[
{
  "switch_id": "00000000000000000000000000000002",
  "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/00000000000000000000000000000000
[{"switch_id": "00000000000000000000000000000002", "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/00000000000000000000000000000000
[{"switch_id": "00000000000000000000000000000002", "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/00000000000000000000000000000000
[{"switch_id": "00000000000000000000000000000002", "command_result": [{"result": "success", "vlan_id": 110, "details": "Add address [address_id=2]"}]}]
```

さらに、ルータ s3 にアドレス「172.16.20.1/24」と「10.10.10.3/24」を設定します。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"address": "172.16.20.1/24"}' http://localhost:8080/router/00000000000000000000000000000000
[{"switch_id": "00000000000000000000000000000003", "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/0000000000000000  
[  
{  
    "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/0000000000000000  
[  
{  
    "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/0000000000000000  
[  
{  
    "switch_id": "0000000000000003",  
    "command_result": [  
        {  
            "result": "success",  
            "vlan_id": 110,  
            "details": "Add address [address_id=2]"  
        }  
    ]  
}  
]
```

ルータへの IP アドレスの設定ができたので、各ホストにデフォルトゲートウェイとして登録します。

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
```

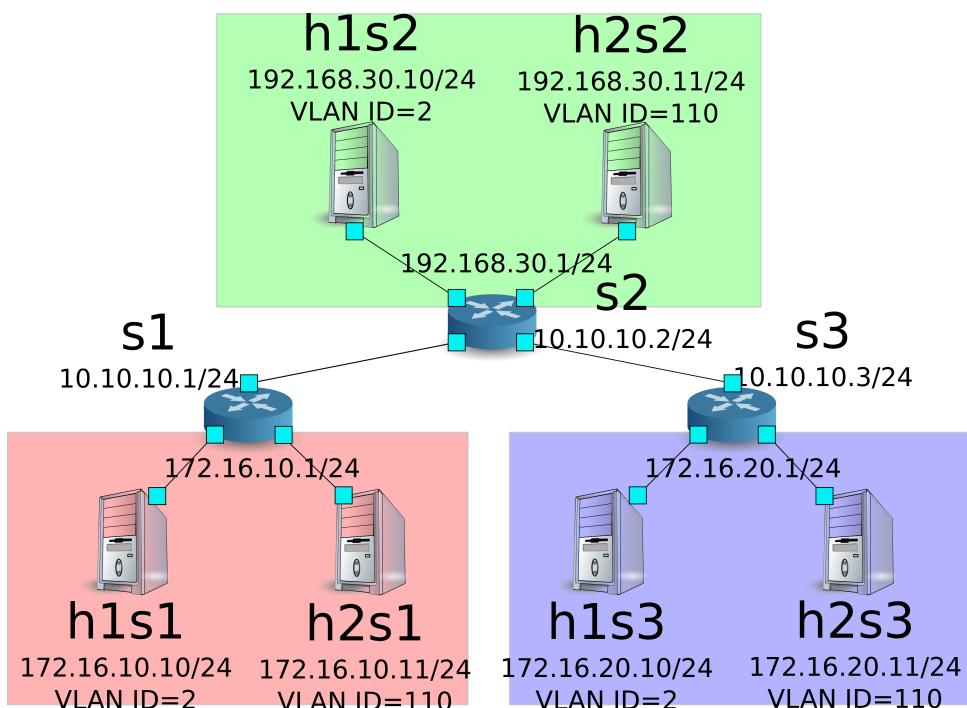
host: h1s3:

```
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
```

設定されたアドレスは、次の通りです。



11.2.3 デフォルトルートと静的ルートの設定

各ルータにデフォルトルートと静的ルートを設定します。

まず、ルータ s1 のデフォルトルートとしてルータ s2 を設定します。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"gateway": "10.10.10.2"}' http://localhost:8080/router/0000000000000000
[{"switch_id": "0000000000000001", "command_result": [{"result": "success", "vlan_id": 2, "details": "Add route [route_id=1]"}]}]

root@ryu-vm:~# curl -X POST -d '{"gateway": "10.10.10.2"}' http://localhost:8080/router/0000000000000000
[{"switch_id": "0000000000000001", "command_result": [{"result": "success", "vlan_id": 110, "details": "Add route [route_id=1]"}]}]
```

ルータ s2 のデフォルトルートにはルータ s1 を設定します。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"gateway": "10.10.10.1"}' http://localhost:8080/router/0000000000000000
[{"switch_id": "0000000000000002", "command_result": [{"result": "success", "vlan_id": 2, "details": "Add route [route_id=1]"}]}]
```

```
}
```

```
]
```

```
root@ryu-vm:~# curl -X POST -d '{"gateway": "10.10.10.1"}' http://localhost:8080/router/0000000000000000
```

```
[
```

```
{
```

```
    "switch_id": "0000000000000002",
```

```
    "command_result": [
```

```
        {
```

```
            "result": "success",
```

```
            "vlan_id": 110,
```

```
            "details": "Add route [route_id=1]"
```

```
        }
```

```
    ]
```

```
}
```

```
]
```

ルータ s3 のデフォルトルートにはルータ s2 を設定します。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"gateway": "10.10.10.2"}' http://localhost:8080/router/0000000000000000
```

```
[
```

```
{
```

```
    "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/0000000000000000
```

```
[
```

```
{
```

```
    "switch_id": "0000000000000003",
```

```
    "command_result": [
```

```
        {
```

```
            "result": "success",
```

```
            "vlan_id": 110,
```

```
            "details": "Add route [route_id=1]"
```

```
        }
```

```
    ]
```

```
}
```

```
]
```

続いてルータ s2 に対し、ルータ s3 配下のホスト (172.16.20.0/24)へのスタティックルートを設定します。
vlan_id=2 の場合のみ設定します。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"destination": "172.16.20.0/24", "gateway": "10.10.10.3"}' http://10.10.10.1:8080/router/1/1
[{"switch_id": "0000000000000002", "command_result": [{"result": "success", "vlan_id": 2, "details": "Add route [route_id=2]"}]}]
```

11.2.4 設定内容の確認

各ルータに設定された内容を確認します。

Node: c0 (root):

```
root@ryu-vm:~# curl http://localhost:8080/router/all/all
[{"internal_network": [{"route": [{"route_id": 1, "destination": "0.0.0.0/0", "gateway": "10.10.10.2"}], "vlan_id": 2, "address": [{"address_id": 2, "address": "10.10.10.1/24"}, {"address_id": 1, "address": "172.16.10.1/24"}]}, {"route": []}], "switch_id": "0000000000000002", "command_result": [{"result": "success", "details": "Add route [route_id=2]"}]}]
```

```
        "route_id": 1,
        "destination": "0.0.0.0/0",
        "gateway": "10.10.10.2"
    }
],
"vlan_id": 110,
"address": [
{
    "address_id": 2,
    "address": "10.10.10.1/24"
},
{
    "address_id": 1,
    "address": "172.16.10.1/24"
}
]
},
"switch_id": "0000000000000001"
},
{
"internal_network": [
{
},
{
"route": [
{
"route_id": 2,
"destination": "172.16.20.0/24",
"gateway": "10.10.10.3"
},
{
"route_id": 1,
"destination": "0.0.0.0/0",
"gateway": "10.10.10.1"
}
],
"vlan_id": 2,
"address": [
{
    "address_id": 2,
    "address": "10.10.10.2/24"
},
{
    "address_id": 1,
    "address": "192.168.30.1/24"
}
]
},
{
"route": [
{

```

```
        "route_id": 1,
        "destination": "0.0.0.0/0",
        "gateway": "10.10.10.1"
    }
],
"vlan_id": 110,
"address": [
{
    "address_id": 2,
    "address": "10.10.10.2/24"
},
{
    "address_id": 1,
    "address": "192.168.30.1/24"
}
]
},
"switch_id": "0000000000000002"
},
{
    "internal_network": [
{
},
{
        "route": [
{
        "route_id": 1,
        "destination": "0.0.0.0/0",
        "gateway": "10.10.10.2"
}
],
"vlan_id": 2,
"address": [
{
    "address_id": 1,
    "address": "172.16.20.1/24"
},
{
    "address_id": 2,
    "address": "10.10.10.3/24"
}
]
},
{
        "route": [
{
        "route_id": 1,
        "destination": "0.0.0.0/0",
        "gateway": "10.10.10.2"
}
]
},
```

```

        "vlan_id": 110,
        "address": [
            {
                "address_id": 1,
                "address": "172.16.20.1/24"
            },
            {
                "address_id": 2,
                "address": "10.10.10.3/24"
            }
        ]
    },
    "switch_id": "0000000000000003"
}
]

```

各ルータの設定内容を表にすると、下記のようになります。

ルータ	VLAN ID	IP アドレス	デフォルトルート	静的ルート
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)	宛先:172.16.20.0/24, ゲートウェイ: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)	

h1s1 から h1s3 に対し ping を送信してみます。同じ vlan_id=2 のホスト同士であり、ルータ s2 に s3 宛の静的ルートが設定されているため、疎通が可能です。

host: h1s1:

```

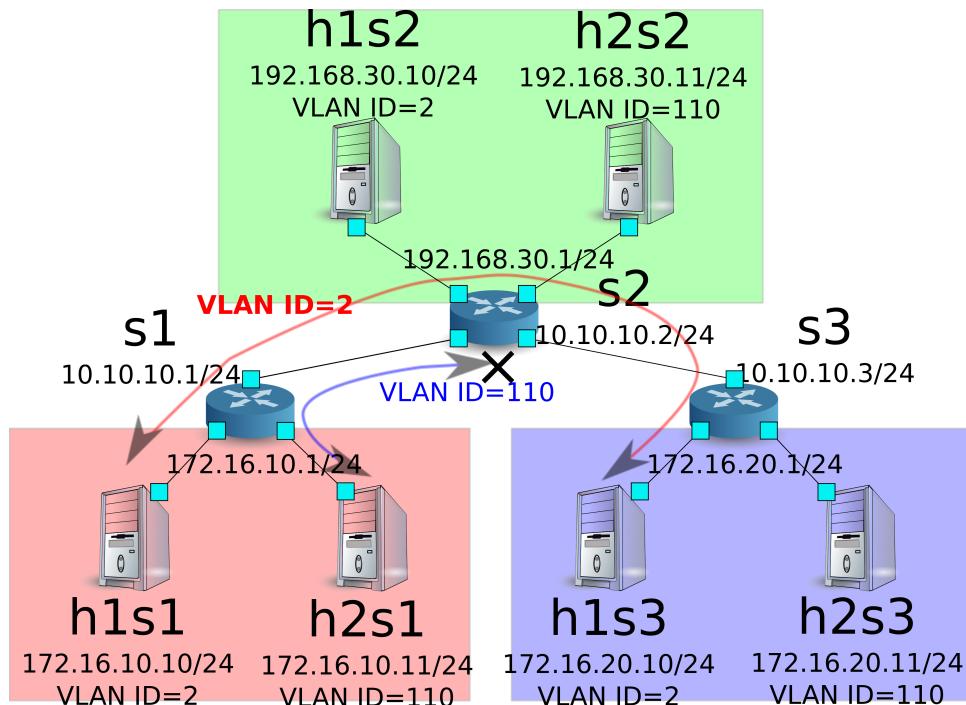
root@ryu-vm:~# ping 172.16.20.10
PING 172.16.20.10 (172.16.20.10) 56(84) bytes of data.
64 bytes from 172.16.20.10: icmp_req=1 ttl=61 time=45.9 ms
64 bytes from 172.16.20.10: icmp_req=2 ttl=61 time=0.257 ms
64 bytes from 172.16.20.10: icmp_req=3 ttl=61 time=0.059 ms
64 bytes from 172.16.20.10: icmp_req=4 ttl=61 time=0.182 ms

```

h2s1 から h2s3 に対し ping を送信してみます。同じ vlan_id=110 のホスト同士ですが、ルータ s2 に s3 宛の静的ルートが設定されていないため、疎通が不可能です。

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
```



本章では、具体例を挙げながらルータの使用方法を説明しました。

11.3 REST API 一覧

本章で紹介した rest_router の REST API 一覧です。

11.3.1 設定の取得

メソッド	GET
URL	/router/{switch}[/{vlan}] -switch: [“all” スイッチ ID] -vlan: [“all” VLAN ID]
備考	VLAN ID の指定はオプションです。

11.3.2 アドレスの設定

メソッド	POST
URL	/router/{switch}[/{vlan}] -switch: [“all” スイッチ ID] -vlan: [“all” VLAN ID]
データ	address: ”<xxx.xxx.xxx.xxx/xx>”
備考	アドレス設定はルート設定前に行ってください。 VLAN ID の指定はオプションです。

11.3.3 静的ルートの設定

メソッド	POST
URL	/router/{switch}[/{vlan}] -switch: [“all” スイッチ ID] -vlan: [“all” VLAN ID]
データ	destination: ”<xxx.xxx.xxx.xxx/xx>” gateway: ”<xxx.xxx.xxx.xxx>”
備考	VLAN ID の指定はオプションです。

11.3.4 デフォルトルートの設定

メソッド	POST
URL	/router/{switch}[/{vlan}] -switch: [“all” スイッチ ID] -vlan: [“all” VLAN ID]
データ	gateway: ”<xxx.xxx.xxx.xxx>”
備考	VLAN ID の指定はオプションです。

11.3.5 アドレスの削除

メソッド	DELETE
URL	/router/{switch}[/{vlan}] -switch: [“all” スイッチ ID] -vlan: [“all” VLAN ID]
データ	address_id: [1 - ...]
備考	VLAN ID の指定はオプションです。

11.3.6 ルートの削除

メソッド	DELETE
URL	/router/{switch}[/{vlan}] -switch: [“all” スイッヂ ID] -vlan: [“all” VLAN ID]
データ	route_id:[1 - ...]
備考	VLAN ID の指定はオプションです。

第 12 章

OpenFlow スイッチテストツール

本章では、OpenFlow スイッチの OpenFlow 仕様への準拠の度合いを検証する、テストツールの使用方法を解説します。

12.1 テストツールの概要

本ツールは、テストパターンファイルに従って試験対象の OpenFlow スイッチに対してフローエントリ登録 / パケット印加を実施し、OpenFlow スイッチのパケット書き換えや転送（または破棄）の処理結果と、テストパターンファイルに記述された「期待する処理結果」の比較を行うことにより、OpenFlow スイッチの OpenFlow 仕様への対応状況を検証するテストツールです。

ツールは、OpenFlow バージョン 1.3 の FlowMod メッセージの試験に対応しています。

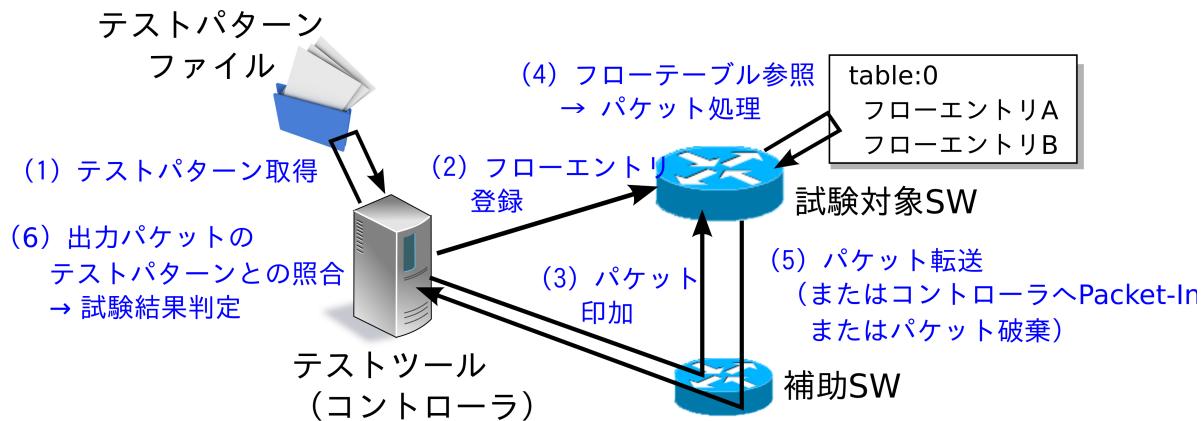
試験対象メッセージ	対応パラメータ
OpenFlow1.3 FlowMod メッセージ	match (IN_PHY_PORT を除く) actions (SET_QUEUE、GROUP を除く)

印加するパケットの生成やパケット書き換え結果の確認などに「*ch_packet_lib*」を利用しています。

12.1.1 動作概要

試験実行イメージ

テストツールを実行した際の動作イメージを示します。テストパターンファイルには、「登録するフローエントリ」「印加パケット」「期待する処理結果」が記述されます。また、ツール実行のための環境設定については後述（ツール実行環境を参照）します。



試験結果の出力イメージ

指定されたテストパターンファイルのテスト項目を順番に実行し、試験結果 (OK / ERROR) を出力します。
試験結果が ERROR の場合はエラー詳細を併せて出力します。

--- 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 ---

```

12.2 使用方法

テストツールの使用方法を解説します。

12.2.1 テストパターンファイル

試験したいテストパターンに応じたテストパターンファイルを作成する必要があります。

テストパターンファイルは拡張子を「.json」としたテキストファイルです。以下の形式で記述します。

```
[
    "xxxxxxxxxx",                                # 試験項目名
    {
        "description": "xxxxxxxxxx",   # 試験内容の説明
        "prerequisite": [
            {
                "OFPFlowMod": {...}  # 登録するフローエントリ
            },
            {...},                  # (Ryu の OFPFlowMod を json 形式で記述)
            {...},                  # パケット転送 (actions=output) の場合は
            {...}                   # 出力ポート番号に「2」を指定してください
        ],
        "tests": [
            {
                "ingress": [           # 印加するパケット
                    "ethernet(...)",  # (Ryu パケットライブラリのコンストラクタの形式で記述)
                    "ipv4(...)",
                    "tcp(...)"
                ],

                # 期待する処理結果
                # 処理結果の種別に応じて (a) (b) (c) のいずれかを記述
                # (a) パケット転送 (actions=output:X) の確認試験
                "egress": [           # 期待する転送パケット
                    "ethernet(...)",
                    "ipv4(...)",
                    "tcp(...)"
                ]
                # (b) パケットイン (actions=CONTROLLER) の確認試験
                "PACKET_IN": [         # 期待する Packet-In データ
                    "ethernet(...)",
                    "ipv4(...)",
                    "tcp(...)"
                ]
                # (c) table-miss の確認試験
                "table-miss": [        # table-miss となることを期待するフローテーブル ID
                    0
                ]
            },
            {...},
            {...}
        ]
    },
    {...},                                         # 試験 1
    {...},                                         # 試験 2
    {...}                                          # 試験 3
]

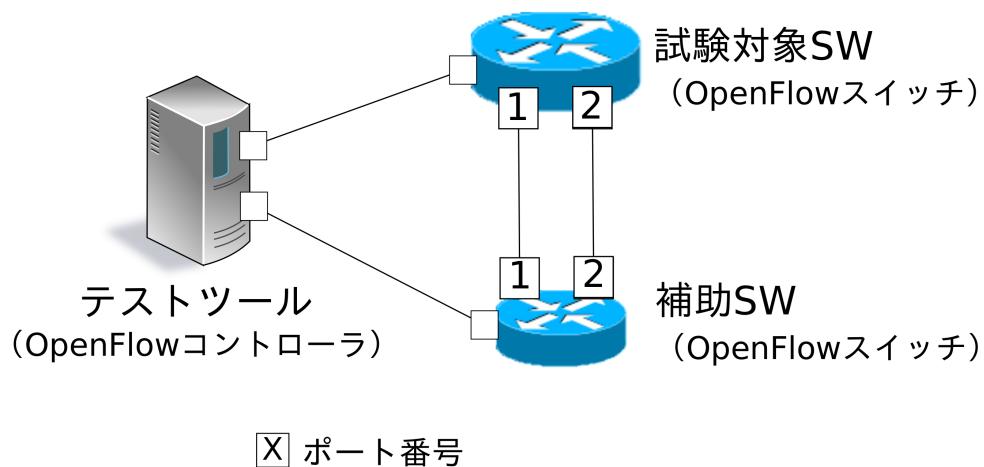
```

ノート: Ryu のソースツリーにはサンプルテストパターンとして、OpenFlow1.3 FlowMod メッセージの match / actions に指定できる各パラメータがそれぞれ正常に動作するかを確認するテストパターンファイルが用意されています。

ryu/tests/switch/of13

12.2.2 ツール実行環境

テストツール実行のための環境は次のとおりです。



補助スイッチとして、以下の動作を正常に行うことが出来る OpenFlow スイッチが必要です。

- actions=CONTROLLER のフローエントリ登録
- actions=CONTROLLER のフローエントリによる Packet-In メッセージ送信
- Packet-Out メッセージ受信によるパケット送信

ノート: Open vSwitch を試験対象スイッチとしたツール実行環境を mininet 上で実現する環境構築スクリプトが、Ryu のソースツリーに用意されています。

`ryu/tests/switch/run_mininet.py`

スクリプトの使用例を「[テストツール使用例](#)」に記載しています。

12.2.3 テストツールの実行方法

テストツールは Ryu のソースツリー上で公開されています。

ソースコード	説明
<code>ryu/tests/switch/tester.py</code>	テストツール
<code>ryu/tests/switch/of13</code>	テストパターンファイルのサンプル
<code>ryu/tests/switch/run_mininet.py</code>	試験環境構築スクリプト

テストツールは次のコマンドで実行します。

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

オプション	説明	デフォルト値
-test-switch-target	試験対象スイッチのデータパス ID	00000000000000000001
-test-switch-tester	補助スイッチのデータパス ID	00000000000000000002
-test-switch-dir	テストパターンファイルのディレクトリパス	ryu/tests/switch/of13

ノート: テストツールは Ryu アプリケーションとして ryu.base.app_manager.RyuApp を継承して作成されているため、他の Ryu アプリケーションと同様に -verbose オプションによるデバッグ情報出力等にも対応しています。

テストツールの起動後、試験対象スイッチと補助スイッチがコントローラに接続されると、指定したテストパターンファイルを元に試験が開始されます。

12.3 テストツール使用例

サンプルテストパターンやオリジナルのテストパターンファイルを用いたテストツールの実行手順を紹介します。

12.3.1 サンプルテストパターンの実行手順

Ryu のソースツリーのサンプルテストパターン (ryu/tests/switch/of13) を用いて、FlowMod メッセージの match / actions の一通りの動作確認を行う手順を示します。

本手順では、試験環境を試験環境構築スクリプト (ryu/tests/switch/run_mininet.py) を用いて構築することとします。このため試験対象スイッチは Open vSwitch となります。VM イメージ利用のための環境設定やログイン方法等は「[スイッチングハブ](#)」を参照してください。

1. 試験環境の構築

VM 環境にログインし、試験環境構築スクリプトを実行します。

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

net コマンドの実行結果は次の通りです。

```
mininet> net
c0
s1 lo:  s1-eth1:s2-eth1 s1-eth2:s2-eth2
s2 lo:  s2-eth1:s1-eth1 s2-eth2:s1-eth2
```

2. テストツール実行

テストツール実行のため、コントローラの xterm を開きます。

```
mininet> xterm c0
```

「Node: c0 (root)」の xterm から、テストツールを実行します。この際、テストパターンファイルのディレクトリとして、サンプルテストパターンのディレクトリ (ryu/tests/switch/of13) を指定し

ます。なお、mininet 環境の試験対象スイッチと補助スイッチのデータパス ID はそれぞれ--test-switch-target / --test-switch-tester オプションのデフォルト値となっているため、オプション指定を省略しています。

Node: c0:

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

ツールを実行すると次のように表示され、試験対象スイッチと補助スイッチがコントローラに接続されるまで待機します。

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

試験対象スイッチと補助スイッチがコントローラに接続されると、試験が開始されます。

```
root@ryu-vm:~$ ryu-manager --test-switch-dir ryu/ryu/tests/switch/of13/ ryu/ryu/tests/switch/
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'
        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'
        Failed to add flows: OFPErrorMsg[type=0x02, code=0x00]
...

```

ryu/tests/switch/of13 配下の全てのサンプルテストパターンファイルの試験が完了すると、テストツールは終了します。

<参考>

サンプルテストパターンファイル一覧

match / actions の各設定項目に対応するフローエントリを登録し、フローエントリに match する
(または match しない) 複数パターンのパケットを印加するテストパターンが用意されています。

```

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 28_PUSH_PBB.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/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

12.3.2 オリジナルテストパターンの実行手順

次に、オリジナルのテストパターンを作成してテストツールを実行する手順を示します。

例として、OpenFlow スイッチがルータ機能を実現するために必要な match / actions を処理する機能を備えているかを確認するテストパターンを作成します。

1. テストパターンファイル作成

ルータがルーティングテーブルに従ってパケットを転送する機能を実現する以下のフローエントリが正しく動作するかを試験します。

match	actions
宛先 IP アドレス帯 「192.168.30.0/24」	送信元 MAC アドレスを「aa:aa:aa:aa:aa:aa」に書き換え 宛先 MAC アドレスを「bb:bb:bb:bb:bb:bb」に書き換え TTL 減算 パケット転送

このテストパターンを実行するテストパターンファイルを作成します。

ファイル名：sample_test_pattern.json

```
[  
    "sample: Router test",  
    {  
        "description": "static routing table",  
        "prerequisite": [  
            {  
                "OFPFlowMod": {  
                    "table_id": 0,  
                    "match": {  
                        "OFPMatch": {  
                            "oxm_fields": [  
                                {  
                                    "OXMTlv": {  
                                        "field": "eth_type",  
                                        "value": 2048  
                                    }  
                                },  
                                {  
                                    "OXMTlv": {  
                                        "field": "ip_dst",  
                                        "value": "192.168.30.0/24"  
                                    }  
                                }  
                            ]  
                        }  
                    }  
                }  
            }  
        ]  
    }  
]
```

```

        "field": "ipv4_dst",
        "mask": 4294967040,
        "value": "192.168.30.0"
    }
}
]
},
"instructions": [
{
    "OFPInstructionActions": {
        "actions": [
{
            "OFPActionSetField": {
                "field": {
                    "OXMTlv": {
                        "field": "eth_src",
                        "value": "aa:aa:aa:aa:aa:aa"
                    }
                }
            }
        ],
        {
            "OFPActionSetField": {
                "field": {
                    "OXMTlv": {
                        "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. 試験環境構築

試験環境構築スクリプトを用いて試験環境を構築します。手順は[サンプルテストパターンの実行手順](#)を参照してください。

3. テストツール実行

コントローラの xterm から、先ほど作成したオリジナルのテストパターンファイルを指定してテストツールを実行します。なお、`--test-switch-dir` オプションはディレクトリだけでなくファイルを直接指定することも可能です。また、送受信パケットの内容を確認するため`--verbose` オプションを指定しています。

Node: c0:

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

試験対象スイッチと補助スイッチがコントローラに接続されると、試験が開始されます。

「`dpid=0000000000000002 : receive_packet...`」のログ出力から、テストパターンファイルの egress パケットとして設定した、期待する出力パケットが送信されたことが分かります。なお、ここではテストツールが出力したログのみを抜粋しています。

```
root@ryu-vm:~$ ryu-manager --verbose --test-switch-dir ./sample_test_pattern.json ryu/ryu/te
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=00000000000000000002 : Join tester SW.  
dpid=00000000000000000001 : 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(  
egress:[ethernet(dst='bb:bb:bb:bb:bb:bb', ethertype=2048, src='aa:aa:aa:aa:aa:aa'), ipv4(csum=  
packet_in:[]  
dpid=00000000000000000002 : receive_packet[ethernet(dst='bb:bb:bb:bb:bb:bb', ethertype=2048, src='  
static routing table                                     OK  
--- Test end ---
```

実際に OpenFlow スイッチに登録されたフローエントリは以下の通りです。テストツールによって印加されたパケットがフローエントリに match し、n_packets がカウントアップされていることが分かります。

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=192.168.1.100
```

12.3.3 エラーメッセージ一覧

本ツールで出力されるエラーメッセージの一覧を示します。

エラーメッセージ	説明
Failed to initialize flow tables: barrier request timeout.	前回試験のフローエントリ削除に失敗 (Barrier Request のタイムアウト)
Failed to initialize flow tables: [err_msg]	前回試験のフローエントリ削除に失敗 (FlowMod に対する Error メッセージ受信)
Failed to add flows: barrier request timeout.	フローエントリ登録に失敗 (Barrier Request のタイムアウト)
Failed to add flows: [err_msg]	フローエントリ登録に失敗 (FlowMod に対する Error メッセージ受信)
Added incorrect flows: [flows]	フローエントリ登録確認エラー (想定外のフローエントリが登録された)
Failed to add flows: flow stats request timeout.	フローエントリ登録確認に失敗 (FlowStats Request のタイムアウト)
Failed to add flows: [err_msg]	フローエントリ登録確認に失敗 (FlowStats Request に対する Error メッセージ受信)
Failed to request port stats from target: request timeout.	試験対象 SW の PortStats 取得に失敗 (PortStats Request のタイムアウト)
Failed to request port stats from target: [err_msg]	試験対象 SW の PortStats 取得に失敗 (PortStats Request に対する Error メッセージ受信)
Failed to request port stats from tester: request timeout.	補助 SW の PortStats 取得に失敗 (PortStats Request のタイムアウト)
Failed to request port stats from tester: [err_msg]	補助 SW の PortStats 取得に失敗 (PortStats Request に対する Error メッセージ受信)
Received incorrect [packet]	期待した出力パケットの受信エラー (異なるパケットを受信)
Receiving timeout: [detail]	期待した出力パケットの受信に失敗 (タイムアウト)
Faild to send packet: barrier request timeout.	パケット印加に失敗 (Barrier Request のタイムアウト)
Faild to send packet: [err_msg]	パケット印加に失敗 (Packet-Out に対する Error メッセージ受信)
Table-miss error: increment in matched_count.	table-miss 確認エラー (フローに match している)
Table-miss error: no change in lookup_count.	table-miss 確認エラー (パケットが確認対象のフローテーブルで処理されていない)
Failed to request table stats: request timeout.	table-miss の確認に失敗 (TableStats Request のタイムアウト)
Failed to request table stats: [err_msg]	table-miss の確認に失敗 (TableStats Request に対する Error メッセージ受信)

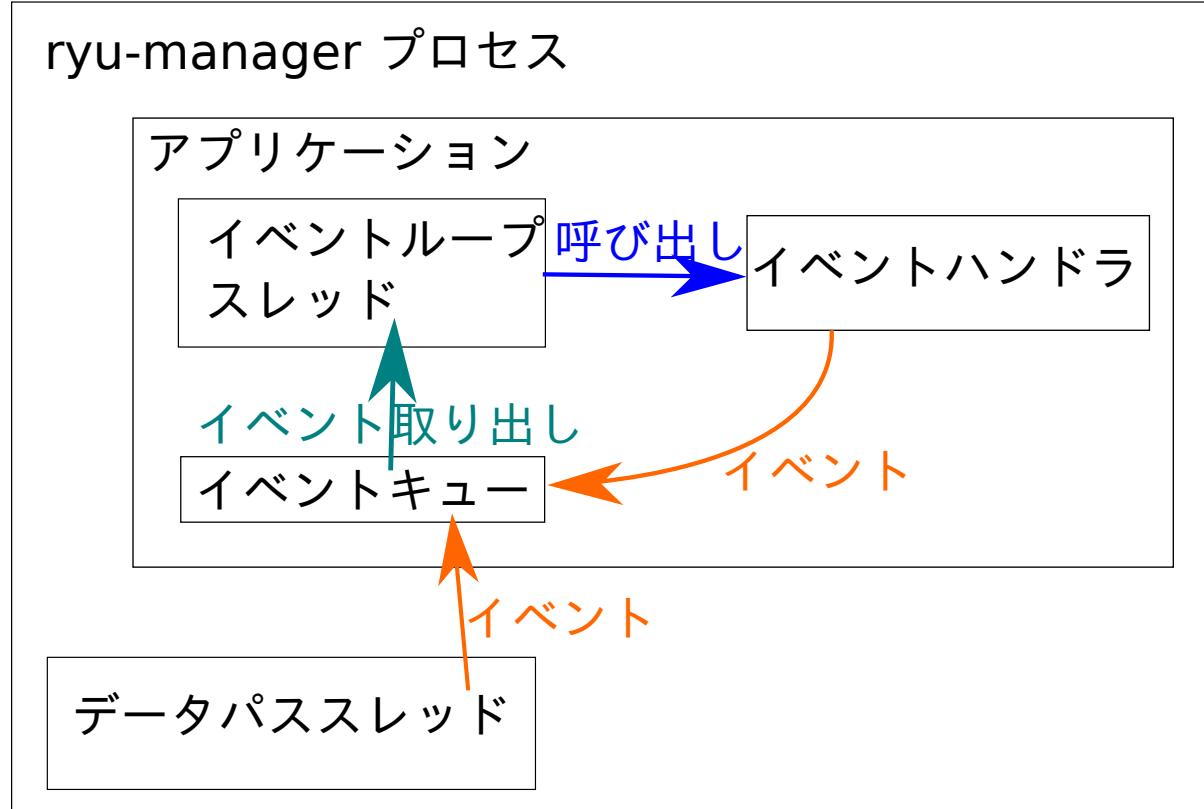
第 13 章

アーキテクチャ

Ryu のアーキテクチャを紹介します。各クラスの使い方など [API リファレンス](#) もご参照ください。

13.1 アプリケーションプログラミングモデル

Ryu アプリケーションのプログラミングモデルを説明します。



13.1.1 アプリケーション

アプリケーションとは `ryu.base.app_manager.RyuApp` を継承したクラスです。ユーザーロジックはアプリケーションとして記述します。

13.1.2 イベント

イベントとは `ryu.controller.event.EventBase` を継承したクラスのオブジェクトです。アプリケーション間の通信はイベントを送受信することで行ないます。

13.1.3 イベントキュー

各アプリケーションはイベント受信のためのキューを一つ持っています。

13.1.4 スレッド

Ryu は `eventlet` を使用したマルチスレッドで動作します。スレッドは非ブリエンプトですので、時間のかかる処理を行なう場合は注意が必要です。

イベントループ

アプリケーションにつき一個のスレッドが自動的に作成されます。このスレッドはイベントループを実行します。イベントループは、イベントキューにイベントがあれば取り出し、対応するイベントハンドラ（後述）を呼び出します。

追加のスレッド

`hub.spawn` 関数を使用して追加のスレッドを作成し、アプリケーション固有の処理を行なうことができます。

eventlet

`eventlet` の機能をアプリケーションから直接使用することもできますが、非推奨です。可能なら `hub` モジュールの提供するラッパーを使用するようにしてください。

13.1.5 イベントハンドラ

アプリケーションクラスのメソッドを `ryu.controller.handler.set_ev_cls` デコレータで修飾することでイベントハンドラを定義できます。イベントハンドラは指定した種類のイベントが発生した際に、アプリケーションのイベントループから呼び出されます。

第 14 章

コントリビューション

オープンソース・ソフトウェアの醍醐味の一つは、自ら開発に参加できることでしょう。この章では、Ryu の開発に参加する方法について紹介します。

14.1 開発体制

Ryu の開発はメーリングリストを中心に進められています。まずはメーリングリストに参加することから始めましょう。

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

メーリングリストでのやり取りは、基本的に英語で行われます。使い方などで疑問があつたり、不具合と思われるような挙動に遭遇した際には、メールを送ることをためらう必要はありません。オープンソース・ソフトウェアを使うこと自体が、プロジェクトにとって重要なコントリビューションだからです。

14.2 開発環境

このセクションでは、Ryu の開発で必要な環境と留意事項について説明します。

14.2.1 Python

Ryu は Python 2.6 以上をサポートしています。すなわち、Python 2.7 でのみ使用可能な構文などは使ってはいけません。

Python 3.0 以上については、今のところサポートされていません。ですが、ソースコードは将来的な変更がなるべく少なく済むような記述を心がけると良いでしょう。

14.2.2 コーディングスタイル

Ryu のソースコードは PEP8 というコーディングスタイルに準拠しています。後述するパッチの送付の際には、その内容が PEP8 に準拠していることをあらかじめ確認してください。

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

尚、ソースコードが PEP8 に準拠しているか確認するには、テストのセクションで紹介するスクリプトと共にチェックマークが利用できます。

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

14.2.3 テスト

Ryu には幾つかの自動化されたテストが存在しますが、最も単純で多用されるものは Ryu のみで完結するユニットテストです。後述するパッチの送付の際には、加えた変更によってユニットテストの実行が失敗しないことをあらかじめ確認してください。また、新たに追加したソースコードについては、なるべくユニットテストを記述することが望ましいでしょう。

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

14.3 パッチを送る

機能の追加や、不具合の修正などでリポジトリのソースコードを変更する際には、変更内容をパッチにした上で、メーリングリストに送ります。大きな変更は、あらかじめメーリングリストで議論されていると望ましいでしょう。

ノート: Ryu のソースコードのリポジトリは GitHub 上に存在しますが、プルリクエストを用いた開発プロセスではないことに注意してください。

送付するパッチの形式は Linux カーネルの開発で使われるスタイルが想定されています。このセクションでは、同スタイルのパッチをメーリングリストに送るまでの一例を紹介していますが、より詳しくは関連するドキュメントを参照してください。

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

それでは手順を紹介します。

1. ソースコードをチェックアウトする

まずは Ryu のソースコードをチェックアウトします。GitHub 上でソースコードを fork して自分の作業用リポジトリを作っても構いませんが、単純にするためオリジナルをそのまま使った例になっています。

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

2. ソースコードに変更を加える

Ryu のソースコードに必要な変更を加えます。作業に区切りがついたら、変更内容をコミットしましょう。

```
$ git commit -a
```

3. パッチを作る

変更内容の差分をパッチにします。パッチには Signed-off-by: 行を付けることを忘れないでください。この署名は、あなたが提出したパッチがオープンソース・ソフトウェアのライセンス上、問題ないことの宣言になります。

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

4. パッチを送る

完成したパッチの内容が正しいことを確認した後に、メーリングリストに送ります。お使いのメールで直接送ることもできますが git-send-email(1) を使うことで対話的に扱うこともできます。

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

5. 応答を待つ

パッチに対する応答を待ちます。そのまま取り込まれる場合もありますが、指摘事項などがあれば内容を修正して再度送る必要があるでしょう。

第 15 章

導入事例

本章では、Ryu を利用したサービス / 製品の事例について紹介します。

15.1 Stratosphere SDN Platform (ストラトスフィア)

Stratosphere SDN Platform(以下 SSP) は、ストラトスフィア社の開発するソフトウェア製品です。SSP を用いることで VXLAN, STT, MPLS といったトンネリングプロトコルを用いて、エッジオーバレイ型の仮想ネットワークを構築できます。

各トンネリングプロトコルは VLAN と相互に変換されます。各トンネリングプロトコルの識別子は VLAN の 12 ビットよりも大きいことから、VLAN を直接使うよりも多くの L2 セグメントが管理できます。また SSP は OpenStack や CloudStack といった IaaS ソフトウェアと組み合わせて使用することができます。

SSP では機能の実現に OpenFlow を用いており、バージョン 1.1.4 ではコントローラに Ryu を採用しています。理由としては、まず OpenFlow1.1 以降への対応が挙げられます。SSP を MPLS に対応させる上で、プロトコルレベルでのサポートがある OpenFlow1.1 以降に対応したフレームワークの導入が考えられました。

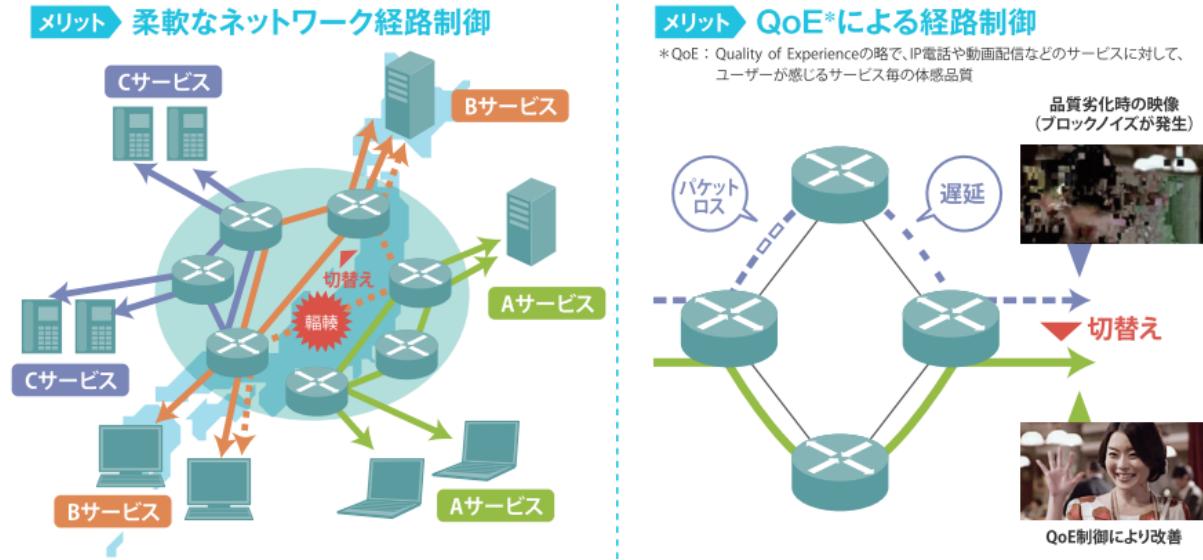
ノート： OpenFlow プロトコル自体のサポートとは別に、実装がオプショナルな項目については、利用する OpenFlow スイッチ側のサポート状況も十分に考慮する必要があります。

また、開発言語として Python が利用できる点も挙げられます。ストラトスフィアの開発では Python を積極的に用いており、SSP も多くの箇所が Python で記述されています。Python 自体の記述力の高さと、使い慣れた言語を利用できることで開発効率の向上が見込めました。

ソフトウェアは複数の Ryu アプリケーションから成り、REST API を通して SSP の他のコンポーネントとやり取りします。ソフトウェアを機能単位で複数のアプリケーションに分割できることは、見通しの良いソースコードを保つ上で不可欠でした。

15.2 SmartSDN Controller (NTT コムウェア)

「SmartSDN Controller」は、従来の自律分散制御にかわるネットワークの集中制御機能（ネットワーク仮想化/最適化等）を提供する SDN コントローラです。



「SmartSDN Controller」は以下の 2 点の特徴を有しています。

1. 仮想ネットワークによる柔軟なネットワーク経路制御

同一の物理ネットワーク上に複数の仮想ネットワークを構築することにより、ユーザからの要望に対し柔軟なネットワーク環境を提供し、設備有効活用による設備コストの低減を可能とします。また、これまで個々に情報を参照、設定していたスイッチ・ルーターを一元管理することで、ネットワーク全体を把握し、故障やネットワークのトラヒック状況に応じた柔軟な経路変更を可能にします。

サービス利用者の体感品質（「QoE」: Quality of Experience）に注目し、通信が流れているネットワークの品質（帯域、遅延、ロス、ゆらぎなど）から体感品質（QoE）を判断し、より良い経路へ迂回することで、サービス品質の安定維持を実現します。

2. 高度な保守運用機能でネットワークの信頼性確保

コントローラの故障発生時にもサービスを継続するため、冗長化構成を実現しています。また、拠点間を流れる通信パケットを疑似的に作成し、経路上に流すことで OpenFlow の仕様で規定される標準的な監視機能では検知出来ない経路上の故障の早期発見や、各種試験（疎通確認、経路確認等）を可能にします。

また、ネットワーク設計、ネットワークの状態確認は GUI により可視化し、保守者のスキルレベルに依らない運用を可能とし、ネットワーク運用コストを低減します。

「SmartSDN Controller」の開発にあたっては、以下の条件を満たす OpenFlow のフレームワークを選定する必要がありました。

- OpenFlow仕様を網羅的にサポートできるフレームワークであること
- OpenFlowのバージョンアップへの追従を計画しているため、比較的早く追従対応がされるフレームワークであること

その中で Ryu は

- OpenFlowの各バージョンにおける機能を満遍なくサポートしている
- OpenFlowのバージョンアップへの追従対応が早い。また、開発コミュニティが活発であり、バグへの対応が早い
- サンプルコード／ドキュメントが充実している

等の特徴を有していることからフレームワークとして適切と判断し、採用しました。