
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

2013 年 12 月 17 日

目次

はじめに	1
第 1 章 スイッチングハブの実装	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	16
1.5 まとめ	22
第 2 章 トラフィックモニター	23
2.1 ネットワークの定期健診	23
2.2 トラフィックモニターの実装	23
2.3 トラフィックモニターの実行	30
2.4 まとめ	31
第 3 章 リンク・アグリゲーションの実装	33
3.1 リンク・アグリゲーション	33
3.2 Ryu アプリケーションの実行	34
3.3 Ryu によるリンク・アグリゲーション機能の実装	47
3.4 まとめ	57
第 4 章 REST API	59
4.1 Ryu における REST API の組み込み	59
4.2 実装する REST API	59
4.3 REST API 付きスイッチングハブの実装	59
4.4 SimpleSwitchRest13 クラスの実装	61
4.5 SimpleSwitchController クラスの実装	63
4.6 REST API 搭載スイッチングハブの実行	65
4.7 まとめ	67
第 5 章 スパニングツリーの実装	69
5.1 スパニングツリー	69
5.2 Ryu アプリケーションの実行	72
5.3 OpenFlow によるスパニングツリー	83
5.4 Ryu によるスパニングツリーの実装	84

5.5	まとめ	94
第 6 章	Ryu パケットライブラリ	95
6.1	基本的な使い方	95
6.2	アプリケーション例	98
第 7 章	OpenFlow プロトコル	103
7.1	マッチ	103
7.2	インストラクション	104
7.3	アクション	105
第 8 章	Ryu アーキテクチャ	107
8.1	アプリケーションプログラミングモデル	107
第 9 章	rest_firewall.py の使用例	109
9.1	シングルテナントでの動作例	109
9.2	マルチテナントでの動作例	122
9.3	REST API 一覧	126
第 10 章	OpenFlow スイッチテストツール	131
10.1	テストツールの概要	131
10.2	使用方法	133
10.3	テストツール使用例	135

はじめに

第1章

スイッチングハブの実装

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OpenFlowコントローラ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、ここでは一番単純な、必要最低限の機能を持ったスイッチングハブの実装をみてみます。

スイッチングハブの機能は次のようなものとします。

- 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
- 受信パケットの宛先ホストが接続されているポートへ、パケットを転送する
- 未知の宛先ホストへのパケットは、フラッディングする

これらの機能をOpenFlowで実現します。

1.2 OpenFlowによるスイッチングハブ

コントローラは、OpenFlowスイッチがパケット受信時に発行するPacket-Inメッセージから、ポートに接続されているホストのMACアドレスを学習します。

OpenFlow1.3では、OpenFlowスイッチにPacket-Inを発行させるために、Table-missフローエントリという特別なエントリをフローテーブルに追加する必要があります。

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: 現時点の Open vSwitch では、まだ 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。

コントローラは、受信した Packet-In メッセージから、パケットの受信ポートと送信元 MAC アドレスを得て、MAC アドレステーブルを更新します。

また、宛先 MAC アドレスを MAC アドレステーブルから検索し、見つかった場合は対応するポートへ転送するよう Packet-Out メッセージを OpenFlow スイッチに発行します。

さらに、対応する Modify Flow Entry(Flow Mod) メッセージを発行し、OpenFlow スイッチにフローエントリを設定することで、同一条件のパケットについては、Packet-In メッセージを発行せずにパケット転送するようにします。

宛先 MAC アドレスが MAC アドレステーブルに存在しないアドレスだった場合は、フラッディングを指定した Packet-Out メッセージを発行します。

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(スイッチが NORMAL ポートをサポートしている場合)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。

これらの動作を順を追って図とともに説明します。なお、ここでは Table-miss フローエントリについては省略しています。

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:

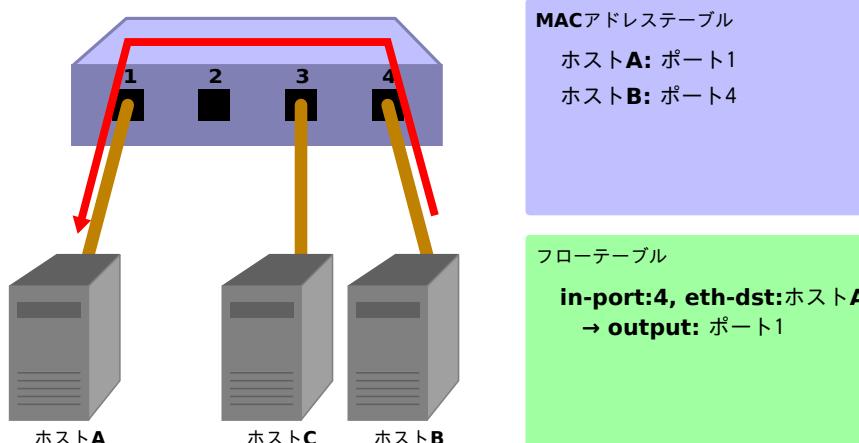
```
in-port: 1
eth-dst: ホストB
eth-src: ホストA
```

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。

```
# Copyright (C) 2011 Nippon Telegraph and Telephone Corporation.
#
# 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.

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(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)
```

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 スイッチに対応する `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
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 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 フローエントリを作成します。

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 OFPCML_NO_BUFFER を指定しています。

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

未知の受信パケットを受け付けるため、Packet-In メッセージを受け取ります。

```
@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
match	<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。
data	受信パケット自体を示すバイナリデータです。
total_len	受信パケットのデータ長です。
buffer_id	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 アドレスがみつかった場合は、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。優先度は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

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
OFPPFC_ADD	新しいフローエントリを追加します
OFPPFC MODIFY	フローエントリを更新します
OFPPFC MODIFY_STRICT	厳格に一致するフローエントリを更新します
OFPPFC_DELETE	フローエントリを削除します
OFPPFC_DELETE_STRICT	厳格に一致するフローエントリを更新します

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 OFPPFC_DELETE または OFPPFC_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)

OFPPFC_DELETE または OFPPFC_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。OFPPFC_ADD、OFPPFC_MODIFY、OFPPFC_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
OFPPF_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。
OFPPF_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。
OFPPF_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。
OFPPF_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。
OFPPF_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 スイッチに対応する 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 EventOFPErrorMsg
    CONSUMES EventOFPHello
    CONSUMES EventOFPEchoRequest
    CONSUMES EventOFPPortDescStatsReply
    CONSUMES EventOFPSwitchFeatures
connected socket:<eventlet.greenio.GreenSocket object at 0x2e2c050> address:(‘127.0.0.1’, 53937)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0x2e2a550>
move onto config mode
EVENT ofp_event->SimpleSwitch13 EventOFPSwitchFeatures
switch features ev version: 0x4 msg_type 0x6 xid 0xff9ad15b OFPSwitchFeatures(auxiliary_id=0, capabilities=71, datapath_id=1, n_buffers=256, n_tables=254)
move onto main mode
```

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 (OFL1.3) (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
=00:00:00:00:00:01 actions=output:1
  cookie=0x0, duration=48.402s, table=0, n_packets=1, n_bytes=42, priority=1,in_port=1,dl_dst
=00:00:00:00:00:02 actions=output:2
root@ryu-vm:~#
```

Table-miss 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 はブ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、ブ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 10.0.0.2 tell 10.0.0.1, length 28
20:38:04.678698 00:00:00:00:00:02 > 00:00:00:00:00:01, ethertype ARP (0x0806), length 42:
Reply 10.0.0.2 is-at 00:00:00:00:00:02, length 28
20:38:04.678731 00:00:00:00:00:01 > 00:00:00:00:00:02, ethertype IPv4 (0x0800), length 98:
10.0.0.1 > 10.0.0.2: ICMP echo request, id 3940, seq 1, length 64
20:38:04.722973 00:00:00:00:00:02 > 00:00:00:00:00:01, ethertype IPv4 (0x0800), length 98:
10.0.0.2 > 10.0.0.1: ICMP echo reply, id 3940, seq 1, length 64
```

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 10.0.0.2 tell 10.0.0.1, length 28
20:38:04.638059 00:00:00:00:00:02 > 00:00:00:00:00:01, ethertype ARP (0x0806), length 42:
Reply 10.0.0.2 is-at 00:00:00:00:00:02, length 28
20:38:04.722601 00:00:00:00:00:01 > 00:00:00:00:00:02, ethertype IPv4 (0x0800), length 98:
10.0.0.1 > 10.0.0.2: ICMP echo request, id 3940, seq 1, length 64
20:38:04.722747 00:00:00:00:00:02 > 00:00:00:00:00:01, ethertype IPv4 (0x0800), length 98:
10.0.0.2 > 10.0.0.1: ICMP echo reply, id 3940, seq 1, length 64
```

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 10.0.0.2 tell 10.0.0.1, length 28
```

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、「スイッチングハブの実装」の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、「スイッチングハブの実装」の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.EventOFPSwitchFeatures, [MAIN_DISPATCHER, DEAD_DISPATCHER])
    def _state_change_handler(self, ev):
        datapath = ev.datapath
        if ev.state == MAIN_DISPATCHER:
            if not datapath.id in self.datapaths:
                self.logger.debug('register datapath: %016x', datapath.id)
                self.datapaths[datapath.id] = datapath
        elif ev.state == DEAD_DISPATCHER:
            if datapath.id in self.datapaths:
                self.logger.debug('deregister datapath: %016x', datapath.id)
                del self.datapaths[datapath.id]

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

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

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

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

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

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

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

```

```

    self.logger.info('datapath      port      '
                     'rx-pkts  rx-bytes rx-error'
                     'tx-pkts  tx-bytes tx-error')
    self.logger.info('----- ----- '
                     '----- ----- '
                     '----- ----- ')
    for stat in sorted(body, key=attrgetter('port_no')):
        self.logger.info('%016x %8x %8d %8d %8d %8d %8d',
                         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.EventOFPSwitchFeatures,
            [MAIN_DISPATCHER, DEAD_DISPATCHER])
def _state_change_handler(self, ev):
    datapath = ev.datapath
    if ev.state == MAIN_DISPATCHER:
        if not datapath.id in self.datapaths:
            self.logger.debug('register datapath: %016x', datapath.id)
            self.datapaths[datapath.id] = datapath
    elif ev.state == DEAD_DISPATCHER:
        if datapath.id in self.datapaths:
            self.logger.debug('deregister 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` になった時に、そのスイッチを監視対象に登録、`DEAD_DISPATCHER` になった時に登録の削除を行っています。

```
# ...
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、対象となるスイッチに `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取得します。テーブル ID、出力ポート、cookie 値、マッチの条件などで取得対象のフローエントリを絞ることができます。ここではすべてのフローエントリを対称としています。

`OFPPortStatsRequest` は、スイッチのポートに関する統計情報を取得します。統計情報を取得するポートのポート番号を指定します。ここでは `OFPP_ANY` を指定し、すべてのポートの統計情報を取得しています。

2.2.2 FlowStats

`FlowStatsReply` メッセージを受信して、フローエントリの統計情報を出力します。

```
# ...
@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。

ここでは、プライオリティが 1 である、Table-miss フロー以外の通常のフローエントリのみを選択し、受信ポートと宛先 MAC アドレスでソートして、そ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、ここでは選択した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、OFPPFlowStatsReply 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": [
                        {
                            "OFPActionOutput": {
                                "len": 16,
                                "max_len": 65509,
                                "port": 1,
                                "type": 0
                            }
                        }
                    ],
                    "len": 24,
                    "type": 4
                }
            }
        ],
        "length": 96,
        "match": {
            "OFPMatch": {
                "length": 22,
                "oxm_fields": [
                    {
                        "OXMTlv": {
                            "field": "in_port",
                            "mask": null,
                            "value": 2
                        }
                    }
                ],
                "type": 1
            }
        }
    }
}
```

```
{
    "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

PortStatsReply メッセージを受信して、ポートの統計情報を出力します。

```
# ...
@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',
                         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、「[スイッチングハブの実装](#)」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 EventOFPPacketIn
    CONSUMES EventOFPPortStatsReply
    CONSUMES EventOFPSwitchFeatures
    CONSUMES EventOFPSwitchFeatures
    CONSUMES EventOFPEchoRequest
    CONSUMES EventOFPPortDescStatsReply
    CONSUMES EventOFPHello
    CONSUMES EventOFPErrorMsg
    CONSUMES EventOFPPacketIn
    CONSUMES EventOFPPortStatsReply
    CONSUMES EventOFPSwitchFeatures
    PROVIDES EventOFPStateChange TO {'SimpleMonitor': set(['main', 'dead'])}
    PROVIDES EventOFPPacketIn TO {'SimpleMonitor': set(['main'])}
    PROVIDES EventOFPPortStatsReply TO {'SimpleMonitor': set(['main'])}
    PROVIDES EventOFPSwitchFeatures TO {'SimpleMonitor': set(['config'])}
    CONSUMES EventOFPSwitchFeatures
    CONSUMES EventOFPPortDescStatsReply
    CONSUMES EventOFPHello
    CONSUMES EventOFPEchoRequest
    CONSUMES EventOFPSwitchFeatures
connected socket:<eventlet.greenio.GreenSocket object at 0x343fb10> address:(('127.0.0.1',
55598)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0x343fed0>
move onto config mode
EVENT ofp_event->SimpleMonitor EventOFPSwitchFeatures
switch features ev version: 0x4 msg_type 0x6 xid 0x7dd2dc58 OFPSwitchFeatures(auxiliary_id=0,
capabilities=71, datapath_id=1, n_buffers=256, n_tables=254)
move onto main mode
EVENT ofp_event->SimpleMonitor EventOFPStateChange
register datapath: 0000000000000001
send stats request: 0000000000000001
EVENT ofp_event->SimpleMonitor EventOFPPacketIn
datapath      in-port eth-dst          out-port packets  bytes
----- -----
EVENT ofp_event->SimpleMonitor EventOFPPacketIn
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
```

「[スイッチングハブの実装](#)」のスイッチングハブの時は、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。

ここで、ホスト 1 からホスト 2 へ ping を実行してみます。

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:~#
```

すると、パケットが転送されたり、フローエントリが設定されたりして、統計情報が変化します。

controller: c0:

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

上のフローエントリの統計情報では、受信ポート 1 のエントリ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。

下の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らないためです。

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 されています。そのため、ポートの統計情報では、ポート 1 の受信パケット数が 1、ポート 2 の受信パケット数が 2、フローエントリの統計情報より多くなっています。

2.4 まとめ

本章では、統計情報の取得機能の実装追加を題材として、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

リンク・アグリゲーションの実装

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。

3.1 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静态的方法と、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。

3.2 Ryu アプリケーションの実行

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実装](#)」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

```
# Copyright (C) 2013 Nippon Telegraph and Telephone Corporation.
#
# 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.

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.OFPIInstructionActions(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)

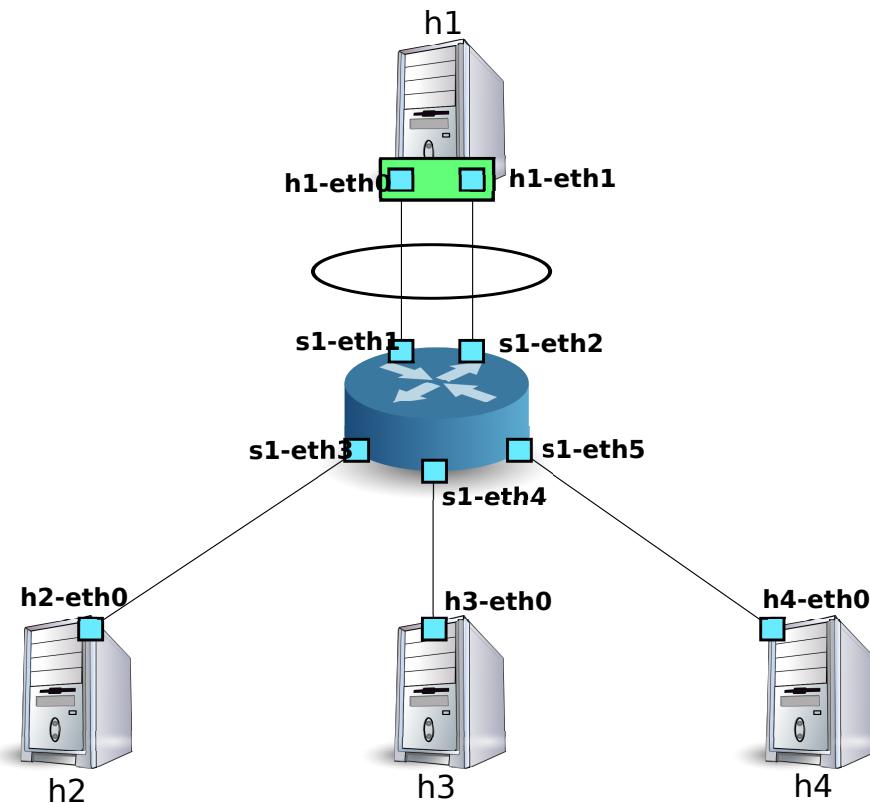
@set_ev_cls(lacplib.EventSlaveStateChanged, MAIN_DISPATCHER)
def _slave_state_changed_handler(self, ev):
    datapath = ev.datapath
    dpid = datapath.id
    port_no = ev.port
    enabled = ev.enabled
    self.logger.info("slave state changed port: %d enabled: %s",
                     port_no, enabled)
    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, {})
```

3.2.1 実験環境の構築

リンク・アグリゲーション・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「[スイッチングハブの実装](#)」を参照してください。

最初に「[スイッチングハブの実装](#)」と同様に Mininet を実行しますが、下図のようにトポロジが特殊なため、`mn` コマンドで作成することができません。



ここでは Mininet の低位クラスを直接使用するスクリプトを作成し、トポロジを作成することにします。

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()
```

このプログラム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 コマンドを実行することで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>
```

3.2.2 ホスト h1 でのリンク・アグリゲーションの設定

前節のコマンドを実行すると、コントローラ c0、ホスト h1～h4、およびスイッチ s1 の xterm が起動します。ホスト h1 の xterm で、ホスト側のリンク・アグリゲーションの設定を行います。本節でのコマンド入力は、すべてホスト 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は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。mn コマンドでホストを作成した場合にならって、10.0.0.1 を割り当てるにします。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や振り分けロジックの設定が確認できます。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 のリンク・アグリゲーションの設定は終了です。

3.2.3 OpenFlow バージョンの設定

「[スイッチングハブの実装](#)」で行ったのと同じように、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 の xterm 上で行ってください。

Node: s1:

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

3.2.4 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。

注意: 本 Ryu アプリケーションはライブラリとして lacplib.py を使用しますが、Ryu3.2 に含まれている lacplib.py には不具合があります。Ryu3.3 以降をご利用ください。

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

「LACP received.」は LACP データユニットを受信したことを、「the slave i/f has just been up.」は無効状態だったポートが有効状態に変更したことを、「the timeout time has changed.」は LACP データユニットの無通信監視時間が変更されたこと（今回の場合、初期状態の 0 秒から LONG_TIMEOUT_TIME の 90 秒）を、「LACP sent.」は応答用の LACP データユニットを送信したことを、それぞれ表します。「slave state changed」の行は、LACP ライブラリからの EventSlaveStateChanged イベントを受信したスイッチングハブが出力しています。

その後は定期的にホスト h1 から送られてくるたび、応答用 LACP データユニットを送信します。

Node: c0:

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

この時点で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=14.565s, table=0, n_packets=1, n_bytes=124, idle_timeout=90,
  send_flow_rem priority=65535,in_port=2,dl_src=00:00:00:00:00:12,dl_type=0x8809 actions=
CONTROLLER:65509
  cookie=0x0, duration=14.562s, table=0, n_packets=1, n_bytes=124, idle_timeout=90,
  send_flow_rem priority=65535,in_port=1,dl_src=00:00:00:00:00:11,dl_type=0x8809 actions=
CONTROLLER:65509
  cookie=0x0, duration=24.821s, table=0, n_packets=2, n_bytes=248, priority=0 actions=
CONTROLLER:65535
```

1 番めのフローエントリは「h1 の h1-eth1 から LACP データユニットが送られてきたら Packet-In メッセージを送信する」、2 番めのフローエントリは「h1 の h1-eth0 から LACP データユニットが送られてきたら Packet-In メッセージを送信する」という内容です。なお、3 番めのフローエントリは「スイッチングハブの実装」でも登録している Table-miss フローエントリです。

3.2.5 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。ただし、実際に通信性能の確認を行おうとすると大量のデータを間断なく送受信し続ける必要があり、試験が煩雑になってしまいますので、ここでは「スイッチングハブの機能により論理インターフェースを経由するフローエントリを複数登録し、その際特定の物理回線にのみフローが集中しないこと」をもって動作の確認を行います。

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
...
...
```

ノート: 後述するとおり、Ryu のリンク・アグリゲーション・アプリケーションは振り分けロジックを実装していません。振り分けは対向装置の処理に依存します。

ホスト h1 のボンディングドライバは、振り分けロジックとして宛先の MAC アドレスを使用するよう設定されています。このロジックは単純に言えば「宛先 MAC アドレスの下 1 バイトを有効なポート数で除算し、その剩余を元にどのポートから出力するかを決定する」といったものです。

ホスト h1 からホスト h2 に対し ping を実行した場合、ICMP echo request の前に ARP request が送信されます。ARP request の宛先 MAC アドレスは ff:ff:ff:ff:ff 固定であるため、振り分けロジックを適用すると使用する物理インターフェースが固定されてしまいます。そういう事態を避けるため、ここではホスト h2 からホスト h1 に対し ping を実行しています。以降の例も同様です。

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=65535,in_port=2,dl_src=00:00:00:00:00:12,dl_type=0x8809 actions=
CONTROLLER:65509
  cookie=0x0, duration=22.046s, table=0, n_packets=1, n_bytes=124, idle_timeout=90,
  send_flow_rem priority=65535,in_port=1,dl_src=00:00:00:00:00:11,dl_type=0x8809 actions=
CONTROLLER:65509
  cookie=0x0, duration=33.046s, table=0, n_packets=6, n_bytes=472, priority=0 actions=
CONTROLLER:65535
  cookie=0x0, duration=3.259s, table=0, n_packets=3, n_bytes=294, priority=1,in_port=3,dl_dst
=02:01:02:03:04:08 actions=output:1
  cookie=0x0, duration=3.262s, table=0, n_packets=4, n_bytes=392, priority=1,in_port=1,dl_dst
=00:00:00:00:00:22 actions=output:3
```

先ほど確認した時点から、4 番めと 5 番めのフローエントリが追加されています。4 番めのフローエントリは「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」、5 番めのフローエントリは「1 番ポート (s1-eth2) 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=65535,in_port=2,dl_src=00:00:00:00:00:12,dl_type=0x8809 actions=
CONTROLLER:65509
  cookie=0x0, duration=99.761s, table=0, n_packets=4, n_bytes=496, idle_timeout=90,
  send_flow_rem priority=65535,in_port=1,dl_src=00:00:00:00:00:11,dl_type=0x8809 actions=
CONTROLLER:65509
  cookie=0x0, duration=110.761s, table=0, n_packets=10, n_bytes=696, priority=0 actions=
CONTROLLER:65535
  cookie=0x0, duration=80.974s, table=0, n_packets=82, n_bytes=7924, priority=1,in_port=3,
  dl_dst=02:01:02:03:04:08 actions=output:1
  cookie=0x0, duration=2.677s, table=0, n_packets=2, n_bytes=196, priority=1,in_port=2,dl_dst
=00:00:00:00:00:23 actions=output:4
  cookie=0x0, duration=2.675s, table=0, n_packets=1, n_bytes=98, priority=1,in_port=4,dl_dst
=02:01:02:03:04:08 actions=output:2
  cookie=0x0, duration=80.977s, table=0, n_packets=83, n_bytes=8022, priority=1,in_port=1,
  dl_dst=00:00:00:00:00:22 actions=output:3
```

先ほど確認した時点から、5番めと6番めのフローエントリが追加されています。6番めのフローエントリは「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」、5番めのフローエントリは「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」というフローエントリです。h3 との通信には s1-eth2 が使用されていることがわかります。

同様にホスト h4 からホスト h1 に対し ping を実行します。

Node: h4:

```
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=86.3 ms
64 bytes from 10.0.0.1: icmp_req=2 ttl=64 time=0.397 ms
64 bytes from 10.0.0.1: icmp_req=3 ttl=64 time=0.136 ms
64 bytes from 10.0.0.1: icmp_req=4 ttl=64 time=0.035 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=199.287s, table=0, n_packets=7, n_bytes=868, idle_timeout=90,
  send_flow_rem priority=65535,in_port=2,dl_src=00:00:00:00:00:12,dl_type=0x8809 actions=
CONTROLLER:65509
  cookie=0x0, duration=199.283s, table=0, n_packets=7, n_bytes=868, idle_timeout=90,
  send_flow_rem priority=65535,in_port=1,dl_src=00:00:00:00:00:11,dl_type=0x8809 actions=
CONTROLLER:65509
  cookie=0x0, duration=210.283s, table=0, n_packets=14, n_bytes=920, priority=0 actions=
CONTROLLER:65535
```

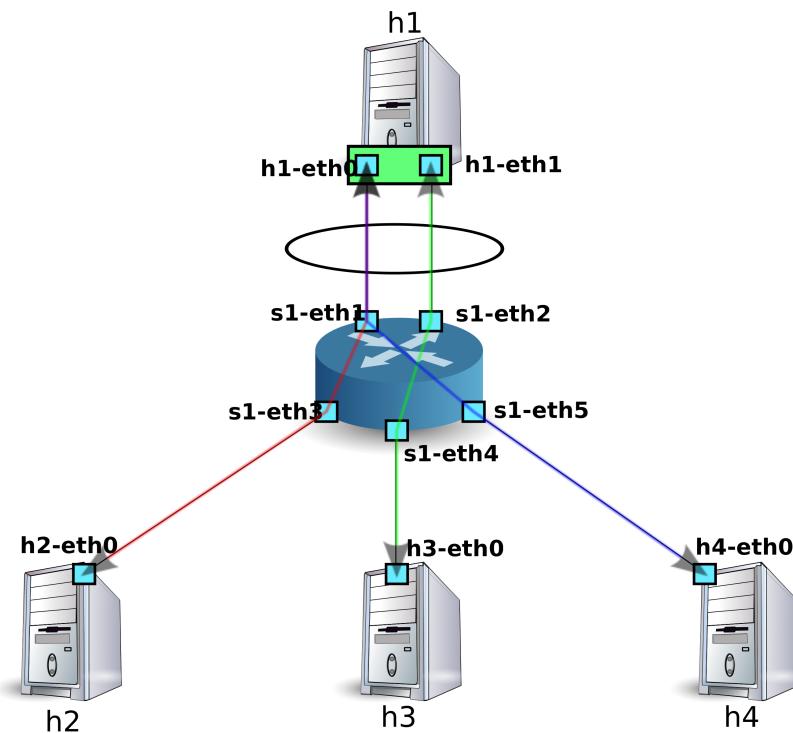
```

cookie=0x0, duration=2.807s, table=0, n_packets=3, n_bytes=294, priority=1,in_port=1,dl_dst
=00:00:00:00:00:24 actions=output:5
cookie=0x0, duration=180.496s, table=0, n_packets=185, n_bytes=17850, priority=1,in_port=3,
dl_dst=02:01:02:03:04:08 actions=output:1
cookie=0x0, duration=2.804s, table=0, n_packets=2, n_bytes=196, priority=1,in_port=5,dl_dst
=02:01:02:03:04:08 actions=output:1
cookie=0x0, duration=102.199s, table=0, n_packets=105, n_bytes=10122, priority=1,in_port=2,
dl_dst=00:00:00:00:00:23 actions=output:4
cookie=0x0, duration=102.197s, table=0, n_packets=104, n_bytes=10024, priority=1,in_port=4,
dl_dst=02:01:02:03:04:08 actions=output:2
cookie=0x0, duration=180.499s, table=0, n_packets=186, n_bytes=17948, priority=1,in_port=1,
dl_dst=00:00:00:00:00:22 actions=output:3

```

追加されたフローエントリは、「5番ポート(s1-eth5、h4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 h4 宛のパケットを受信したら 5番ポート(s1-eth5)から出力する」です。

宛先ホスト	使用ポート
h2	1
h3	2
h4	1



以上のように、フローが分散することが確認できました。

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。そのまま無通信監視時間の上限である 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=0000000000000001 PORT=1 LACP received.
[LACP] [INFO] SW=0000000000000001 PORT=1 LACP sent.
[LACP] [INFO] SW=0000000000000001 PORT=2 LACP received.
[LACP] [INFO] SW=0000000000000001 PORT=2 LACP sent.
[LACP] [INFO] SW=0000000000000001 PORT=2 LACP received.
[LACP] [INFO] SW=0000000000000001 PORT=2 LACP sent.
[LACP] [INFO] SW=0000000000000001 PORT=2 LACP received.
[LACP] [INFO] SW=0000000000000001 PORT=2 LACP sent.
[LACP] [INFO] SW=0000000000000001 PORT=1 LACP exchange timeout has occurred.
slave state changed port: 1 enabled: False
...
```

「LACP exchange timeout has occurred.」は無通信監視時間の上限に達し、LACP データユニットを Packet-In するフローエントリが削除されたことを表します。EventSlaveStateChanged イベントを受信したスイッチングハブは、学習した MAC アドレスをすべて忘却し、転送用のフローエントリをすべて削除します。

すべての学習結果を忘れた状態でも、ホスト h2～h4 ではまだ ping が実行されているため、通常のスイッチングハブの動作によって再度 MAC アドレスを学習し、転送用のフローエントリを登録します。このとき、停止している h1-eth0 ではパケットの送受信が行われないため、ホスト h2～h4 とホスト h1 との間の通信はすべて h1-eth1 が使用されます。

Node: s1:

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

```
cookie=0x0, duration=5.506s, table=0, n_packets=5, n_bytes=434, priority=1,in_port=4,dl_dst
=02:01:02:03:04:08 actions=output:2
cookie=0x0, duration=5.736s, table=0, n_packets=5, n_bytes=490, priority=1,in_port=2,dl_dst
=00:00:00:00:00:21 actions=output:3
cookie=0x0, duration=6.504s, table=0, n_packets=6, n_bytes=532, priority=1,in_port=2,dl_dst
=00:00:00:00:00:22 actions=output:4
```

フローエントリが再登録されたことにより、ホスト h3 で停止していた ping が再開されます。

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、一部の物理インターフェースに故障が発生した場合でも、他の物理インターフェースを用いて自動的に復旧できることが確認できました。

3.3 Ryu によるリンク・アグリゲーション機能の実装

Ryu のリンク・アグリゲーション・アプリケーションにおいて、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。

冒頭でも述べたとおり、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つ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「[スイッチングハブの実装](#)」のスイッチングハブ機能で学習・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実装](#)」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。

ACTIVE の実装には定期送信のためのタイマー処理が必要となります。リンク・アグリゲーションとは本来無関係な処理が増えてしまうのを避けるため、ACTIVE モードを非対応としています。

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 メッセージ処理中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 アドレスの剩余额」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 アドレスの剩余额:0	MAC アドレスの剩余额:1	MAC アドレスの剩余额: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 アドレスの剩余额」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 アドレスの剩余额:0	MAC アドレスの剩余额: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、ということになります。

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するものとします。

LACP データユニット以外のフローエントリは「[スイッチングハブの実装](#)」のスイッチングハブ機能で登録しているため、それらのフローエントリを削除するのもスイッチングハブで行うべきです。

以上のことを整理すると、リンク・アグリゲーション機能として実装すべき項目は

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
- FlowRemoved メッセージ受信時、当該物理インターフェースを無効とみなし、所属する論理インターフェースのフローエントリをスイッチングハブに削除させる
- 無効状態の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、当該物理インターフェースを有効とみなし、所属する論理インターフェースのフローエントリをスイッチングハブに削除させる

スイッチングハブ

- LACP データユニット以外を受信したら従来どおり学習・転送する
- LACP ライブラリの指示により、登録済みのフローエントリを削除する

となります。

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/lacliblib.py`

`ryu/app/simple_switch_lacp.py`

ノート: 前述のとおり、`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。

注意: Ryu3.2に含まれている lacplib.py には不具合があります。Ryu3.3以降をご利用ください。

3.3.1 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具体的に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

前述のとおり、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インターフェースとして扱うようになります。複数のグループを作成したい場合、その都度 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

「スイッチングハブの実装」の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専用の動作を行い、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」という処理分岐を行い、スイッチングハブに LACP データユニットを処理さ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「[スイッチングハブの実装](#)」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。

「[スイッチングハブの実装](#)」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利用不能状態であれば利用可能状態に変更し、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タリを登録します。

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種類があります。リンク・アグリゲーションの仕様によれば、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.OFFActionOutput(
        ofproto.OFPP_CONTROLLER, ofproto.OFPCML_MAX)]
    inst = [parser.OFPIInstructionActions(
        ofproto.OFPIAT_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。その中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。

3.3.2 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実装」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。

す。ここでは LACP ライブリの 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 ライブリの初期設定を行います。初期設定には LACP ライブリの提供する `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 ライブリの実装で説明したとおり、LACP ライブリは 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)
        del self.mac_to_port[dpid]
    self.mac_to_port.setdefault(dpid, {})

```

本節の冒頭で整理したとおり、ポートの有効/無効状態が変更され、パケットの転送に使用可能な物理インターフェースの個数が増減すると、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。パケットの経路が変更される場合、すでに登録されているフローエントリを削除し、新たにフローエントリを登録する必要があります。この例では処理を簡略化するため、「ポートの有効/無効状態が変更された場合、当該 OpenFlow スイッチの全学習結果を削除する」という実装となっています。

```

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。

3.4 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4章

REST API

本章では、「スイッチングハブの実装」で説明したスイッチングハブに、REST API を追加します。

4.1 Ryu における REST API の組み込み

Ryu には、WSGI に対応した Web サーバの機能が含まれています。コントローラに REST API を付加することで、ブラウザや curl コマンド等からコントローラの振る舞いを変えるようなアプリケーション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。

4.2 実装する REST API

実装する REST API は以下の二つとします。

1. MAC アドレステーブル取得 API

スイッチングハブで保持されている 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

REST API へのパラメータとして入力された 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ます。また、あわせてフローエントリの追加も行います。

4.3 REST API 付きスイッチングハブの実装

「スイッチングハブの実装」のスイッチングハブに REST API を追加したソースコードです。

```
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ークラス (SimpleSwitchController) です。また、二つ目は「スイッチングハブの実装」の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、任意のタイミングでフローエントリの追加ができるように、FeatureReply メソッドをオーバーライドし、スイッチングハブのインスタンス内で datapath オブジェクトを保持するようにしています。

4.4 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、第 2 引数で、「simple_switch_api_app」というキー名でインスタンスを保持する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 アドレステーブルに初期値(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
...
...
```

REST API へのリクエスト時に渡された datapath ID に対応するスイッチに、MAC アドレスとポートを格納するためのメソッドです。REST API が PUT で呼ばれた際に実行このメソッドが実行されます。このメソッドでは、引数 entry に格納された MAC アドレスと接続ポートのペアと、self.mac_to_port にすでに保持されている MAC アドレスと接続ポートとを組み合わせてフローエントリを作成し、親クラスの add_flow メソッドを用いてそのフローエントリをスイッチに登録します。また続いて、引数 entry の MAC アドレスとポートは、スイッチングハブ内の MAC アドレスとポートアドレステーブルに格納します。

作成されるフローエントリについて、少し説明します。例えば、すでに格納されている MAC アドレスと接続ポートが、

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

また、このメソッドに渡された MAC アドレスとポートの組が、

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

である場合、スイッチに登録されるフローエントリは以下となります。

(1) 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

(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

なお、今回の実装では、self.mac_to_port にすでに登録済みのポートがこのメソッドに渡された場合は、フローエントリの設定は行わず、その時点で格納済みの MAC アドレスと接続ポートの組を返却するのみとしています。

4.5 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。なお、このコンストラクタは WSGIApplication インスタンスから呼び出されます。

```
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]
    ...
```

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.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)
...
```

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。このデコレータを用いることで、HTTP リクエストがデコレータの第 2 引数で指定した URL にマッチした時、list_mac_table メソッドが呼ばれるようになります。

なお、デコレータで指定する内容は、それぞれ以下となります。

1. 第1引数

任意の名前

2. 第2引数

URLを指定します。ここでは、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。

3. 第3引数

HTTPメソッドを指定します。ここではGETメソッドを指定しています。

4. 第4引数

指定箇所の形式を指定します。ここでは、URL(/simpleswitch/mactable/{dpid})の{dpid}の部分が、ryu/lib/dpid.pyのDPID_PATTERNで規定されたパターン(16桁の16進数値)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、このlist_mac_tableメソッドが呼ばれます。このメソッドでは、{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形式に変換後クライアントに返却しています。

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.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)
    ...
```

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の場合にこの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のレスポンスを返却します。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。

4.6 REST API 搭載スイッチングハブの実行

実際に、REST API 搭載スイッチングハブを実行してみます。最初に「[スイッチングハブの実装](#)」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=71,datapath_id=1,n_buffers=256,n_tables=254)
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 へのパケットインは 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
```

第4章 REST API

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。REST API の呼び出しには、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/00000000000000000000000000000001
{"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дрес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=71,datapath_id=1,n_buffers=256,n_tables=254)
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/simpleswitch/mactable/00000000000000000000000000000001
{"00:00:00:00:00:01": 1}
ryu@ryu-vm:~$ curl -X PUT -d '{"mac" : "00:00:00:00:00:02", "port" : 2}' http
://127.0.0.1:8080/simpleswitch/mactable/00000000000000000000000000000001
{"00:00:00:00:00:02": 2, "00:00:00:00:00:01": 1}
```

これらのコマンド実行の際に、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 0.002734
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、パケットインは 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については発生していません。

4.7 まとめ

本章では、MAC アドレステーブルの参照/更新機能の実装追加を題材として、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かでしょうか。

第5章

スパニングツリーの実装

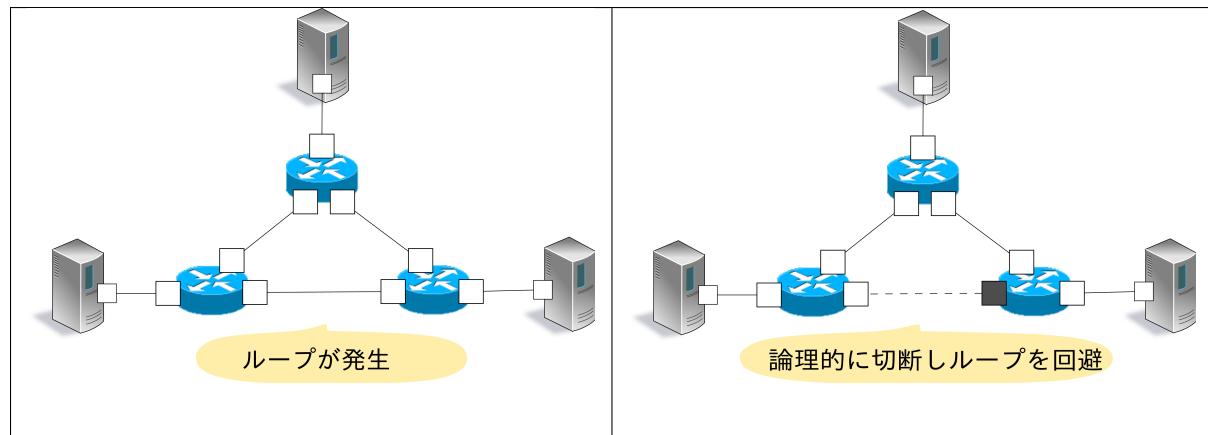
本章では、Ryu を用いたスパニングツリーの実装方法を解説していきます。

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、各ブリッジ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における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各ポートの情報を比較することで、論理ツリーの頂点であるルートブリッジを決定し、さらに各ブリッジの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、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

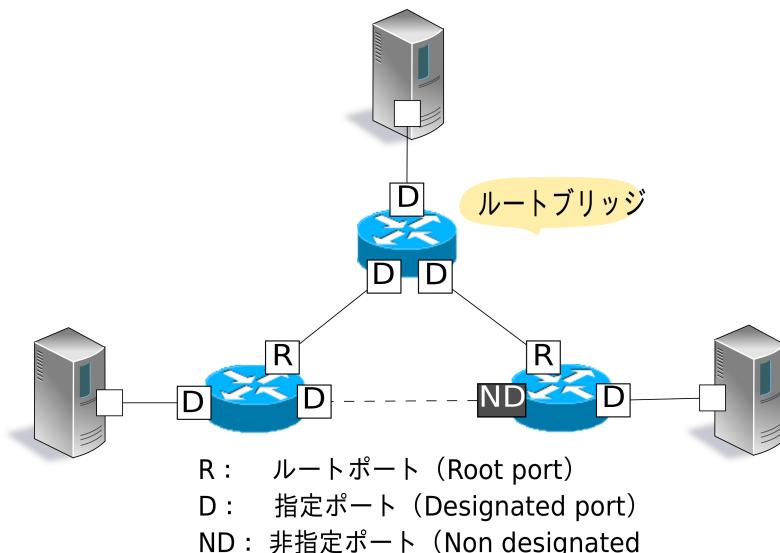
上位 2byte	下位 6byte
ブリッジ priority	MAC アドレス

ブリッジ priority は、Ryu の STP ライブラリではコンフィグ設定により変更が可能です。

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

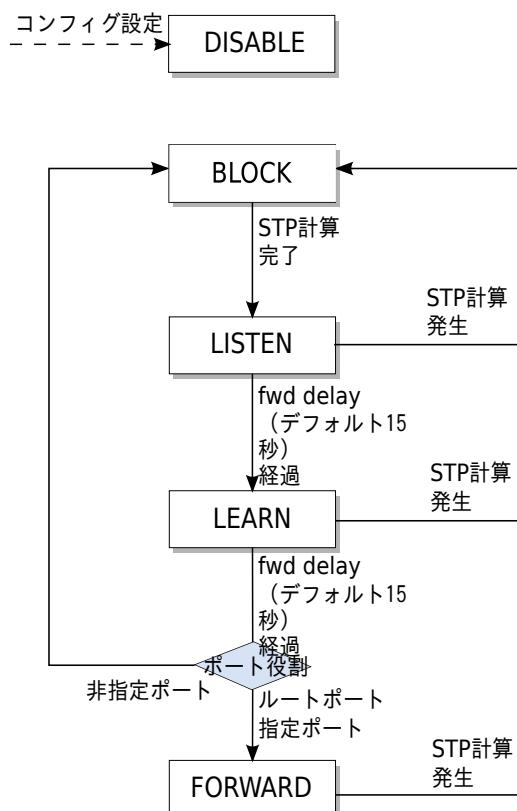
ポート ID

上位 2byte	下位 2byte
ポート priority	ポート番号

ポートの path cost 値やポート priority は、Ryu の STP ライブラリではコンフィグ設定により変更が可能です。

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。コンフィグ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。



状態	動作
DISABLE	無効ポート。全ての受信パケットを無視します。
BLOCK	BPDU 受信のみを行います。
LISTEN	BPDU 送受信を行います。
LEARN	BPDU 送受信/MAC 学習を行います。
FORWARD	BPDU 送受信/MAC 学習/フレーム転送を行います。

これらの動作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トポジ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、「[スイッチングハブの実装](#)」の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

```
# Copyright (C) 2013 Nippon Telegraph and Telephone Corporation.
#
# 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.

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.OFPIInstructionActions(ofproto.OFPI_APPLY_ACTIONS,
                                              actions)]

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

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

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

```

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

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

    dst = eth.dst
    src = eth.src

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

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

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

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

    actions = [parser.OFPActionOutput(out_port)]

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

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

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

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

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

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

```

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

```

5.2.1 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「[スイッチングハブの実装](#)」を参照してください。

ループ構造を持つ特殊なトポジで動作させるため、「[リンク・アグリゲーションの実装](#)」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))

```

```

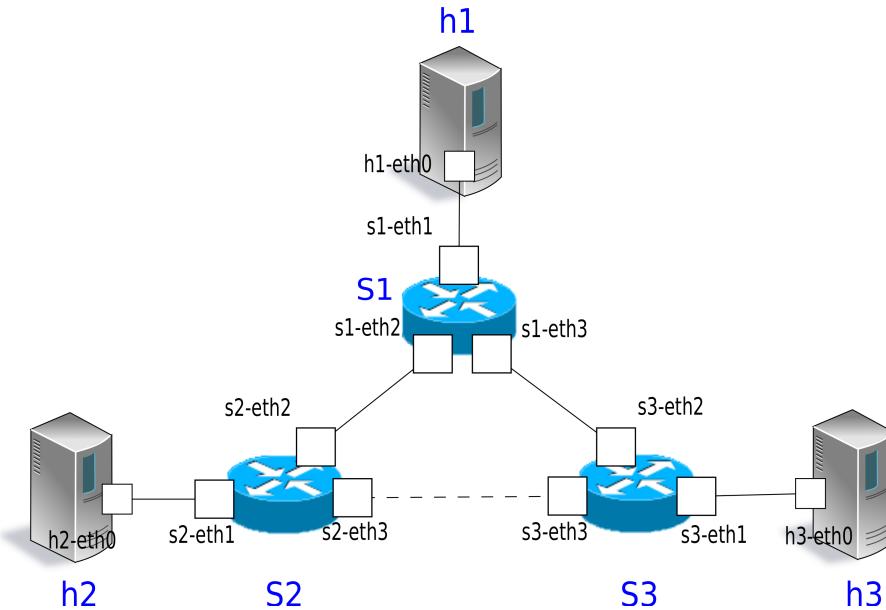
net.terms.append(makeTerm(h2))
net.terms.append(makeTerm(h3))

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:

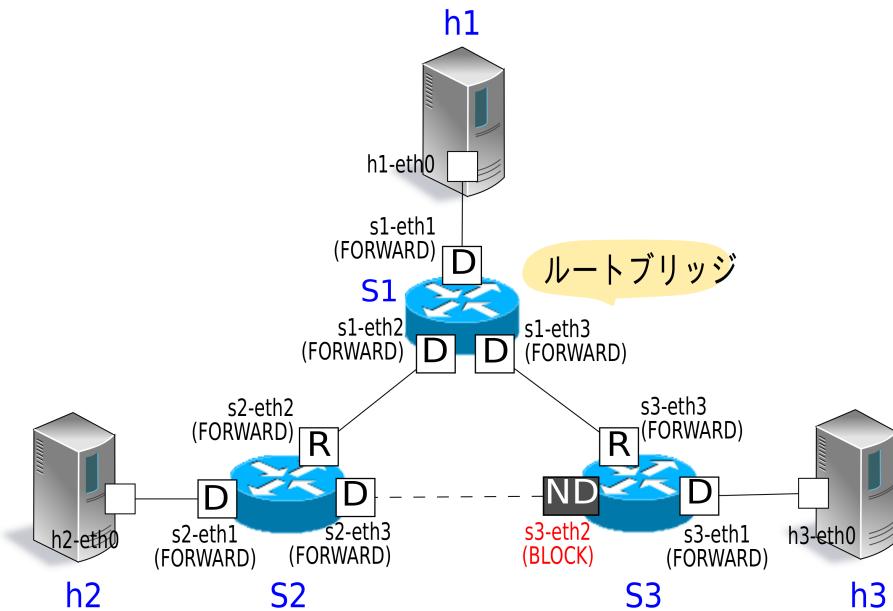
```
root@ryu-vm:~$ ryu-manager ./simple_switch_stp_13.py
loading app simple_switch_stp_13.py
loading app ryu.controller.ofp_handler
loading app ryu.controller.ofp_handler
instantiating app None of Stp
creating context stplib
instantiating app simple_switch_stp_13.py of SimpleSwitch13
instantiating app 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、ルートプリッジの選出・ポート役割の設定・ポート状態遷移が行われます。

```
[STP] [INFO] dpid=0000000000000000: Join as stp bridge.
[STP] [INFO] dpid=0000000000000000: [port=1] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=0000000000000000: [port=2] DESIGNATED_PORT      / LISTEN
[STP] [INFO] dpid=0000000000000000: [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
```


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、各スイッチでパケットがループしていないことを確認できるように 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
```

各スイッチで実行した 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:

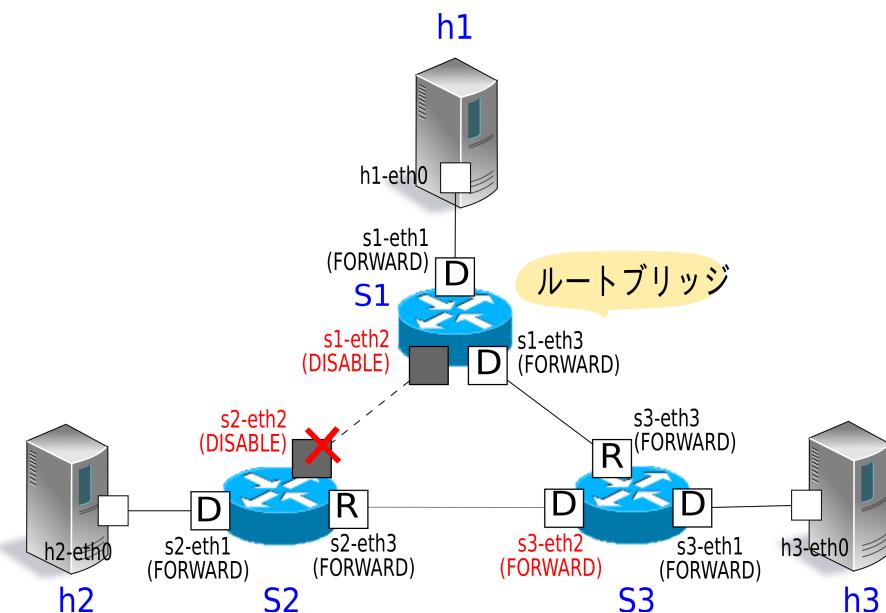
```
root@ryu-vm:~# ifconfig s2-eth2 down
```

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=0000000000000002: [port=1] DESIGNATED_PORT      / LEARN
[STP] [INFO] dpid=0000000000000002: [port=3] ROOT_PORT          / LEARN
[STP] [INFO] dpid=0000000000000003: [port=1] DESIGNATED_PORT      / FORWARD
[STP] [INFO] dpid=0000000000000003: [port=2] DESIGNATED_PORT      / FORWARD
[STP] [INFO] dpid=0000000000000003: [port=3] ROOT_PORT          / FORWARD
[STP] [INFO] dpid=0000000000000002: [port=3] DESIGNATED_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つ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
```

OpenFlow スイッチの初回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るため、各ポートの状態に応じて 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: 前述の通り、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 クラスオブジェクト) が OpenFlow スイッチのコントローラへの接続を検出すると、Bridge クラスオブジェクト・Port クラスオブジェクトが生成されます。各クラスオブジェクトが生成・起動された後は、Stp クラスオブジェクトからの OpenFlow メッセージ受信通知、Bridge クラスオブジェクトの STP 計算 (ルートプリッジ選択・各ポートの役割選択)、Port クラスオブジェクトのポート状態遷移・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() メソッドによりプリッジ・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
priority	プリッジ優先度	0x8000
sys_ext_id	VLAN-ID を設定 (*現状の STP ライブリは VLAN 未対応)	0
max_age	BPDU パケットの受信待ちタイマー値	20[sec]
hello_time	BPDU パケットの送信間隔	2 [sec]
fwd_delay	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。ポート状態が LISTEN となった際にスレッド処理が開始され、ポート役割が DESIGNATED_PORT の場合にのみ、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 パケットの生成 (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` クラスオブジェクトを経由して `Port` クラスオブジェクトに通知されます。イベントハンドラの実装は「[スイッチングハブの実装](#)」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、「新たなルートブリッジが追加された」などのネットワークトポジ变更が発生したことを意味するため、`Bridge` クラスオブジェクトに通知され 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 クラスオブジェクトへ通知されます。

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 クラスオブジェクトへ通知されます。

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次の 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実装」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・アグリゲーションの実装」と同様に STP ライブラリを利用するため CONTEXT 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 スイッチのブリッジ ID が常に最小の値となり、ルートブ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・アグリゲーションの実装」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 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出来たため、「スイッチングハブの実装」と同様の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.OFPPFlowMod(
                datapath, command=ofproto.OFPFC_DELETE,
                out_port=ofproto.OFPP_ANY, out_group=ofproto.OFGG_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を受け取り、ポート状態のデバッガ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

Ryu 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。

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。

6.1 基本的な使い方

6.1.1 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
- 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
```

6.1.2 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'

6.1.3 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
```

6.1.4 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
```

6.2 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_etheren = pkt.get_protocol(etheren.etheren)
    if not pkt_etheren:
        return
    pkt_arp = pkt.get_protocol(arp.arp)
    if pkt_arp:
        self._handle_arp(datapath, port, pkt_etheren, 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_etheren, pkt_ip4, pkt_icmp)
        return

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

def _handle_icmp(self, datapath, port, pkt_etheren, pkt_ip4, pkt_icmp):
    if pkt_icmp.type != icmp.ICMP_ECHO_REQUEST:
        return
    pkt = packet.Packet()
    pkt.add_protocol(etheren.etheren(ethertype=pkt_etheren.ethertype,
                                    dst=pkt_etheren.src,
                                    src=self.hw_addr))
    pkt.add_protocol(ipv4.ipv4(dst=pkt_ip4.src,
                             src=self.ip_addr,
                             proto=pkt_ip4.proto))
    pkt.add_protocol(icmp.icmp(type_=icmp.ICMP_ECHO_REPLY,
                               code=icmp.ICMP_ECHO_REPLY_CODE,
                               csum=0,

```

以下は ping -c 3 を実行した場合のログの例です。

```
=1,src='192.0.2.99',tos=0,total_length=84,ttl=255,version=4), icmp(code=0,csum=26863,data=echo
(data='U,B\x00\x00\x00\x00\x00!\xa26(\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\
\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&%' () *+, -./\x00\x00\x00\x00\x00\x00\x00\x00\x00',id=44565,seq=2),type
=8)
packet-out ethernet(dst='0a:e4:1c:d1:3e:43',ethertype=2048,src='0a:e4:1c:d1:3e:44'), ipv4(csum
=14140,dst='192.0.2.99',flags=0,header_length=5,identification=0,offset=0,option=None,proto=1,
src='192.0.2.9',tos=0,total_length=84,ttl=255,version=4), icmp(code=0,csum=28911,data=echo(
data='U,B\x00\x00\x00\x00\x00!\xa26(\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\
\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&%' () *+, -./\x00\x00\x00\x00\x00\x00\x00\x00\x00',id=44565,seq=2),type
=0)
```

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 7 章

OpenFlow プロトコル

本章では、これまで詳しく説明していなかった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。

7.1 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 7.1 – 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。

7.2 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。

7.3 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。

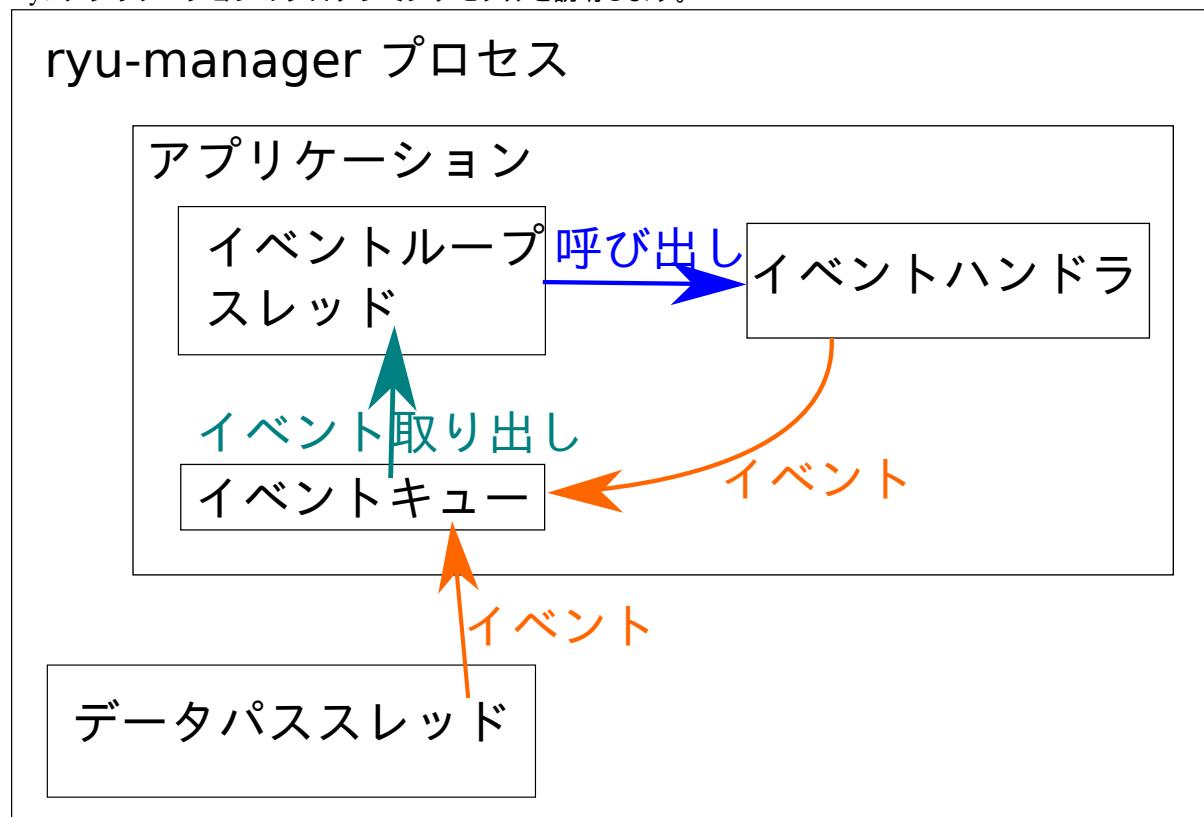
ノート: max_len には通常、Packet-In ハンドラ内の処理で必要となるバイト数を指定します。 OFPCML_NO_BUFFER を指定するとパケット全体が送られるため、パフォーマンスが悪くなります。ところが、現時点の Open vSwitch の実装では、例えば max_len に 128 を指定した場合、バッファリングせずにパケットの先頭 128 バイトのみを添付した Packet-In が発行されます。これでは 128 バイトを越えるサイズのパケットが正しく転送できなくなってしまいますので、 OFPCML_NO_BUFFER を指定してパケット全体を添付させるなどの対策が必要です。

第8章

Ryu アーキテクチャ

8.1 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。



8.1.1 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。

8.1.2 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。

8.1.3 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。

8.1.4 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ができますが、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。

8.1.5 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。

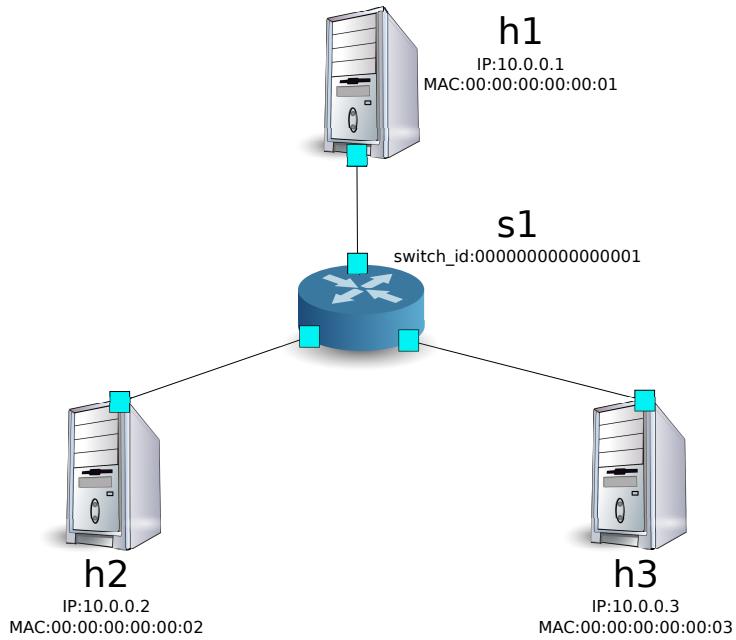
第 9 章

rest_firewall.py の使用例

本章では、「[REST API](#)」を拡張して実装され、Ryu のソースツリーに登録されている ryu/app/rest_firewall.py の使用方法について説明します。

9.1 シングルテナントでの動作例

まず、「[スイッチングハブの実装](#)」で作成した環境を使用して、VLAN によるテナント分けのされていない以下のトポロジを作成し、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。



9.1.1 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「[スイッチングハブの実装](#)」と同様です。

第9章 rest_firewall.py の使用例

```
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
```

注意: Ryu3.2 に含まれている rest_firewall.py は OpenFlow1.3 以降に対応していません。Ryu3.4 以降をご利用ください。

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:~# cd ryu
root@ryu-vm:~/ryu# ryu-manager ryu/app/rest_firewall.py
loading app ryu/app/rest_firewall.py
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
creating context wsgi
instantiating app ryu/app/rest_firewall.py of RestFirewallAPI
instantiating app ryu.controller.ofp_handler of OFPHandler
(1433) 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
```

9.1.2 初期状態の確認

firewall の状態を確認します。初期状態は無効 (disable) になっています。

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 http://localhost:8080/firewall/module/status
[
  {
    "status": "disable",
    "switch_id": "00000000000000000001"
  }
]
```

ノート: REST コマンドの実行結果は見やすいうように整形しています。

この時点でのフローエントリは以下のようになっています。最高優先度で全パケットの破棄が登録されていることがわかります。

switch: s1 (root):

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
  cookie=0x0, duration=32.538s, table=0, n_packets=0, n_bytes=0, priority=65534,arp actions=NORMAL
  cookie=0x0, duration=32.575s, table=0, n_packets=0, n_bytes=0, priority=65535 actions=drop
  cookie=0x0, duration=32.538s, table=0, n_packets=0, n_bytes=0, priority=0 actions=CONTROLLER
    :128
```

この状態で h1 から h2 への ping の疎通を確認してみます。全パケットを破棄するフローエントリが登録されているため、ping は届きません。

host: h1:

```
root@ryu-vm:~# ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
From 10.0.0.1 icmp_seq=4 Destination Host Unreachable
...
```

9.1.3 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/00000000000000000001
[
```

第9章 rest_firewall.py の使用例

```
"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"
    }
]
```

firewall を有効にしたことにより、最高優先度で登録されていた破棄の指示が削除されます。

switch: s1 (root):

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
  cookie=0x0, duration=110.148s, table=0, n_packets=0, n_bytes=0, priority=65534,arp actions=NORMAL
  cookie=0x0, duration=110.148s, table=0, n_packets=0, n_bytes=0, priority=0 actions=CONTROLLER:128
```

この状態で再度 h1 から h2 への ping の疎通を確認してみます。全パケットを破棄するフローエントリはなくなりましたが、h1 から h2 への ping パケットを転送するためのフローエントリが登録されていないため、やはり ping は届きません。

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
```

firewall が無効であった場合と異なり、ルールにマッチしなかったパケットは最低優先度で登録されている PacketIn のフローエントリによって通知されます。この通知によって、破棄されたパケットが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,src='00:00:00:00:00:01'), ipv4(csum=9895,dst='10.0.0.2',flags=2,header_length=5,identification=0,offset=0,option=None,proto=1,src='10.0.0.1',tos=0,total_length=84,ttl=64,version=4), icmp(code=0,csum=55644,data=echo(data='K\x8e\xaeR\x00\x00\x00=\xc6\r\x00\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !#$%&\'()*+,-./01234567',id=6952,seq=1),type=8)
...
```

9.1.4 ログ出力機能の設定変更

firewall のログ出力を無効にします。

Node: c0 (root):

```
root@ryu-vm:~# curl -X PUT http://localhost:8080/firewall/log/disable/00000000000000000000000000000001
[
  [
    {
      "command_result": {
        "result": "success",
        "details": "Log collection stopped."
      }
    }
  ]
]

root@ryu-vm:~# curl http://localhost:8080/firewall/log/status
[
  {
    "log_status": "disable",
    "switch_id": "00000000000000000000000000000001"
  }
]
```

この状態で先ほどと同様に ping を送信し、遮断されることを確認します。

host: h1:

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

ログ出力を無効にしたため、パケットを遮断した旨のログが出力されないことがわかります。

あとの動作確認のため、ログ出力を有効に戻しておきます。

Node: c0 (root):

```
root@ryu-vm:~# curl -X PUT http://localhost:8080/firewall/log/enable/00000000000000000000000000000001
[
  [
    {
      "command_result": {
        "result": "success",
        "details": "Log collection started."
      }
    }
  ]
]

root@ryu-vm:~# curl http://localhost:8080/firewall/log/status
[
  {
    "log_status": "enable",
    "switch_id": "00000000000000000000000000000001"
  }
]
```

9.1.5 ルール追加

h1 と h2 の間で ping による疎通が可能になるようルールを追加します。双方向にルールを設定をする必要がありまますので、ルールをふたつ追加します。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"nw_src": "10.0.0.1/32", "nw_dst": "10.0.0.2/32", "nw_proto": "ICMP"}' http://localhost:8080/firewall/rules/000000000000000000000001
[{"switch_id": "0000000000000001", "command_result": [{"result": "success", "details": "Rule added. : rule_id=1"}]}]
[{"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 actions=NORMAL
  cookie=0x2, duration=118.265s, table=0, n_packets=0, n_bytes=0, priority=1,icmp,nw_src=10.0.0.2,nw_dst=10.0.0.1 actions=NORMAL
```

また、h2 と h3 の間では、ping を含むすべての IPv4 パケットの疎通が可能になるようルールを追加します。先ほどと同様双方向にルールを設定します。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"nw_src": "10.0.0.2/32", "nw_dst": "10.0.0.3/32"}' http://localhost:8080/firewall/rules/000000000000000000000001
```

```
[{"switch_id": "00000000000000000001", "command_result": [{"result": "success", "details": "Rule added. : rule_id=3"}]}, {"switch_id": "00000000000000000001", "command_result": [{"result": "success", "details": "Rule added. : rule_id=4"}]}]
```

root@ryu-vm:~# curl -X POST -d '{"nw_src": "10.0.0.3/32", "nw_dst": "10.0.0.2/32"}' http://localhost:8080/firewall/rules/00000000000000000001

```
[{"switch_id": "00000000000000000001", "command_result": [{"result": "success", "details": "Rule added. : rule_id=4"}]}]
```

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 actions=NORMAL
 cookie=0x4, duration=3.668s, table=0, n_packets=0, n_bytes=0, priority=1, ip,nw_src=10.0.0.3,
 nw_dst=10.0.0.2 actions=NORMAL
 cookie=0x0, duration=1040.802s, table=0, n_packets=10, n_bytes=420, priority=65534, arp
 actions=NORMAL
 cookie=0x0, duration=759.569s, table=0, n_packets=20, n_bytes=1960, priority=0 actions=
 CONTROLLER:128
 cookie=0x1, duration=362.147s, table=0, n_packets=0, n_bytes=0, priority=1, icmp,nw_src
 =10.0.0.1,nw_dst=10.0.0.2 actions=NORMAL
 cookie=0x2, duration=335.362s, table=0, n_packets=0, n_bytes=0, priority=1, icmp,nw_src
 =10.0.0.2,nw_dst=10.0.0.1 actions=NORMAL
```

先ほど追加した h2 と h3 の間のルールに、ping(ICMP) のパケットを遮断するルールを追加します。先ほどと同様双方向にルールを設定します。また、先ほど追加したルールよりも優先度を高く設定します。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"nw_src": "10.0.0.2/32", "nw_dst": "10.0.0.3/32", "nw_proto
 ":"ICMP", "actions": "DENY", "priority": "10"}' http://localhost:8080/firewall/rules
 /00000000000000000001
 [{"switch_id": "00000000000000000001", "command_result": [{"result": "success", "details": "Rule added. : rule_id=5"}]}]
```

第9章 rest_firewall.py の使用例

```
        }
    ]
}

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

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 actions=NORMAL
  cookie=0x4, duration=233.099s, table=0, n_packets=0, n_bytes=0, priority=1, ip, nw_src
  =10.0.0.3, nw_dst=10.0.0.2 actions=NORMAL
  cookie=0x0, duration=1270.233s, table=0, n_packets=10, n_bytes=420, priority=65534, arp
actions=NORMAL
  cookie=0x0, duration=989s, table=0, n_packets=20, n_bytes=1960, priority=0 actions=CONTROLLER
:128
  cookie=0x5, duration=26.984s, table=0, n_packets=0, n_bytes=0, priority=10, icmp, nw_src
  =10.0.0.2, nw_dst=10.0.0.3 actions=drop
  cookie=0x1, duration=591.578s, table=0, n_packets=0, n_bytes=0, priority=1, icmp, nw_src
  =10.0.0.1, nw_dst=10.0.0.2 actions=NORMAL
  cookie=0x6, duration=14.523s, table=0, n_packets=0, n_bytes=0, priority=10, icmp, nw_src
  =10.0.0.3, nw_dst=10.0.0.2 actions=drop
  cookie=0x2, duration=564.793s, table=0, n_packets=0, n_bytes=0, priority=1, icmp, nw_src
  =10.0.0.2, nw_dst=10.0.0.1 actions=NORMAL
```

9.1.6 ルール確認

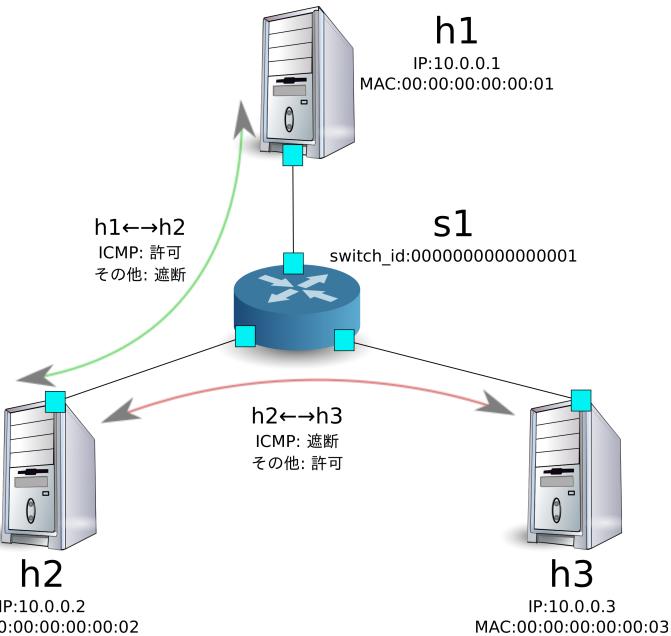
設定されているルールを確認します。

Node: c0 (root):

```
root@ryu-vm:~# curl http://localhost:8080/firewall/rules/0000000000000001
[
  {
    "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。



実際に h1 から h2 に ping を実行して確認します。設定したルールにより、ping が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=0.419 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の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, src='00:00:00:00:00:01'), ipv4(csum=4812, dst='10.0.0.2', flags=2, header_length=5, identification=5102, offset=0, option=None, proto=6, src='10.0.0.1', tos=0, total_length=60, ttl=64, version=4), tcp(ack=0, bits=2, csum=45753, dst_port=80, offset=10, option='\x02\x04\x05\xb4\x04\x02\x08\n\x00H:\x99\x00\x00\x00\x00\x01\x03\x03\t', seq=1021913463, src_port=42664, urgent=0, window_size=14600)
...
...
```

h2 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 によって遮断されます。ただしこの遮断は PacketIn のフローエントリによって通知されないため、ログは出力されません。

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
```

9.1.7 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/00000000000000000001
[{"switch_id": "00000000000000000001", "command_result": [{"result": "success", "details": "Rule deleted. : ruleID=5"}]}

root@ryu-vm:~# curl -X DELETE -d '{"rule_id": "6"}' http://localhost:8080/firewall/rules/00000000000000000001
[{"switch_id": "00000000000000000001", "command_result": [{"result": "success", "details": "Rule deleted. : ruleID=6"}]}]
```

再度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": 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": 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"
}
```

現在のルールを図示すると以下のようになります。



フローを確認すると、”rule_id:5” と”rule_id:6” に該当するフローエントリが削除されていることがわかります。

switch: s1 (root):

```
root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
  cookie=0x3, duration=300.883s, table=0, n_packets=0, n_bytes=0, priority=1, ip,nw_src
=10.0.0.2,nw_dst=10.0.0.3 actions=NORMAL
  cookie=0x4, duration=292.668s, table=0, n_packets=0, n_bytes=0, priority=1, ip,nw_src
=10.0.0.3,nw_dst=10.0.0.2 actions=NORMAL
  cookie=0x0, duration=431.556s, table=0, n_packets=0, n_bytes=0, priority=65534,arp actions=
NORMAL
  cookie=0x0, duration=431.556s, table=0, n_packets=0, n_bytes=0, priority=0 actions=CONTROLLER
:128
  cookie=0x1, duration=345.616s, table=0, n_packets=0, n_bytes=0, priority=1,icmp,nw_src
=10.0.0.1,nw_dst=10.0.0.2 actions=NORMAL
  cookie=0x2, duration=336.091s, table=0, n_packets=0, n_bytes=0, priority=1,icmp,nw_src
=10.0.0.2,nw_dst=10.0.0.1 actions=NORMAL
```

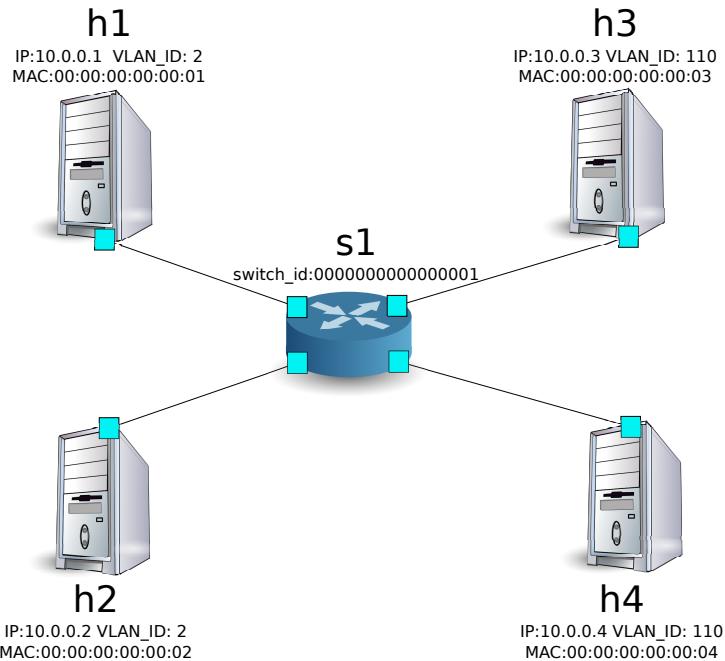
実際に確認します。h2 と h3 の間の ping(ICMP) を遮断するルールが削除されたため、ping が疎通できるようになりましたことがわかります。

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
...
```

9.2 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。



9.2.1 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
```

注意: Ryu3.2 に含まれている rest_firewall.py は OpenFlow1.3 以降に対応していません。Ryu3.4 以降をご利用ください。

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:~# cd ryu
root@ryu-vm:~/ryu# ryu-manager ryu/app/rest_firewall.py
loading app ryu/app/rest_firewall.py
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
creating context wsgi
instantiating app ryu/app/rest_firewall.py 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
```

9.2.2 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/00000000000000000000000000000001
[
  {
    "switch_id": "00000000000000000000000000000001",
    "command_result": {
      "result": "success",
      "details": "firewall running."
    }
  }
]

root@ryu-vm:~# curl http://localhost:8080/firewall/module/status
[
  {
    "status": "enable",
    "switch_id": "00000000000000000000000000000001"
  }
]
```

9.2.3 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をする必要がありますので、ルールをふたつ追加します。

Node: c0 (root):

```
root@ryu-vm:~# curl -X POST -d '{"nw_src": "10.0.0.0/8", "nw_proto": "ICMP"}' localhost:8080/firewall/rules/00000000000000000000000000000001/2
[
  {
    "switch_id": "00000000000000000000000000000001",
    "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"}' localhost:8080/firewall/rules/00000000000000000000000000000001/2
[
  {
    "switch_id": "00000000000000000000000000000001",
    "command_result": [
      {
        "result": "success",
        "vlan_id": 2,
        "details": "Rule added. : rule_id=2"
      }
    ]
  }
]
```

```

        "details": "Rule added. : rule_id=2"
    }
]
}
]
```

9.2.4 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",
                    "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"
}
```

switch: s1 (root):

```

root@ryu-vm:~# ovs-ofctl -O openflow13 dump-flows s1
OFPST_FLOW reply (OF1.3) (xid=0x2):
cookie=0x200000001, duration=190.226s, table=0, n_packets=0, n_bytes=0, priority=1,icmp,
dl_vlan=2,nw_src=10.0.0.0/8 actions=NORMAL
cookie=0x0, duration=329.515s, table=0, n_packets=0, n_bytes=0, priority=65534,arp actions=
NORMAL
cookie=0x0, duration=329.515s, table=0, n_packets=0, n_bytes=0, priority=0 actions=CONTROLLER
:128
cookie=0x200000002, duration=174.986s, table=0, n_packets=0, n_bytes=0, priority=1,icmp,
dl_vlan=2,nw_dst=10.0.0.0/8 actions=NORMAL
```

第9章 rest_firewall.py の使用例

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=2 である h1 から vlan_id=110 である h3 へは、VLAN ID が異なるため ping パケットは到達しません。

host: h1:

```
root@ryu-vm:~# ping 10.0.0.3
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
From 10.0.0.1 icmp_seq=1 Destination Host Unreachable
From 10.0.0.1 icmp_seq=2 Destination Host Unreachable
From 10.0.0.1 icmp_seq=3 Destination Host Unreachable
^C
--- 10.0.0.3 ping statistics ---
6 packets transmitted, 0 received, +3 errors, 100% packet loss, time 5032ms
```

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
```

firewall でパケットが遮断されるとログが output されます。

controller: c0 (root):

```
[FW] [INFO] dpid=0000000000000001: Blocked packet = ethernet(dst='00:00:00:00:00:04', ethertype=33024, src='00:00:00:00:00:03'), vlan(cfi=0, ethertype=2048, pcp=0, vid=110), ipv4(csum=9891, dst='10.0.0.4', flags=2, header_length=5, identification=0, offset=0, option=None, proto=1, src='10.0.0.3', tos=0, total_length=84, ttl=64, version=4), icmp(code=0, csum=58104, data=echo(data='\xb8\x9\xaeR\x00\x00\x00\xce\xe3\x02\x00\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\' ()*+, -./01234567', id=7760, seq=4), type=8)
...
```

9.3 REST API 一覧

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

9.3.1 全スイッチの有効無効状態の取得

メソッド	GET
URL	/firewall/module/status

9.3.2 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。

9.3.3 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。

9.3.4 新規ルールの追加

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。

9.3.5 既存ルールの削除

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。

9.3.6 全スイッチのログ出力状態の取得

メソッド	GET
URL	/firewall/log/status

9.3.7 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 10 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、Ryu の OpenFlow スイッチテストツールの使用方法を解説します。

10.1 テストツールの概要

本ツールは、

1. 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
2. OpenFlow スイッチのパケット編集や転送（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 仕様への準拠の度合い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 の以下の OpenFlow メッセージの試験に対応しています。

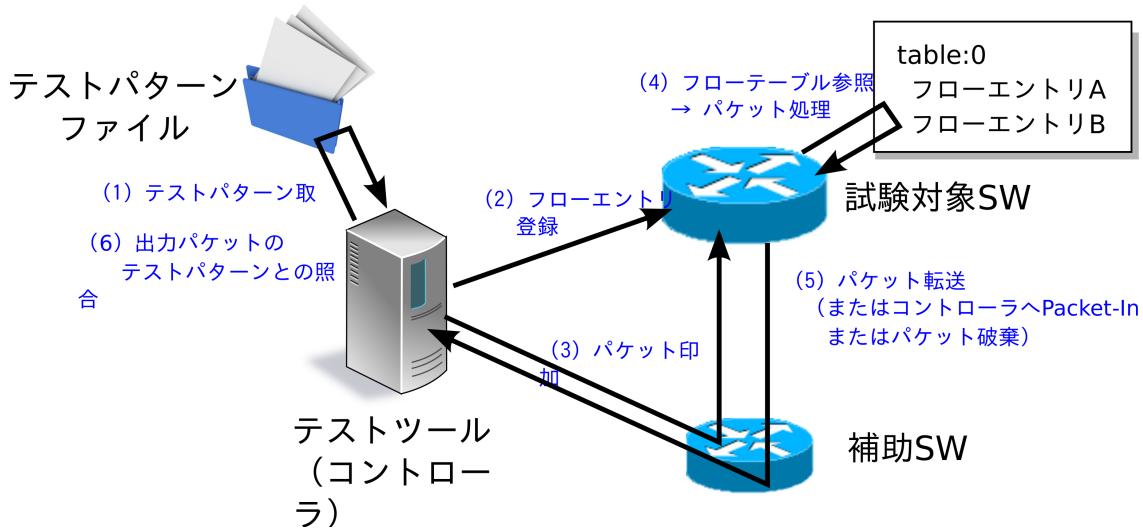
試験対象メッセージ	対応パラメータ
FlowMod メッセージ	match (IN_PHY_PORT を除く) actions (SET_QUEUE、GROUP を除く)

また印加パケットとして、Ryu のパケットライブラリで定義されたプロトコルを用いることが出来ます。

10.1.1 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 ---
```

10.1.2 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.	初期化タイムアウトエラー
Failed to initialize flow tables: [err_msg]	初期化時に OpenFlow エラーメッセージ受信
...	...

10.2 使用方法

テストツールの使用方法を解説します。

10.2.1 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。

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

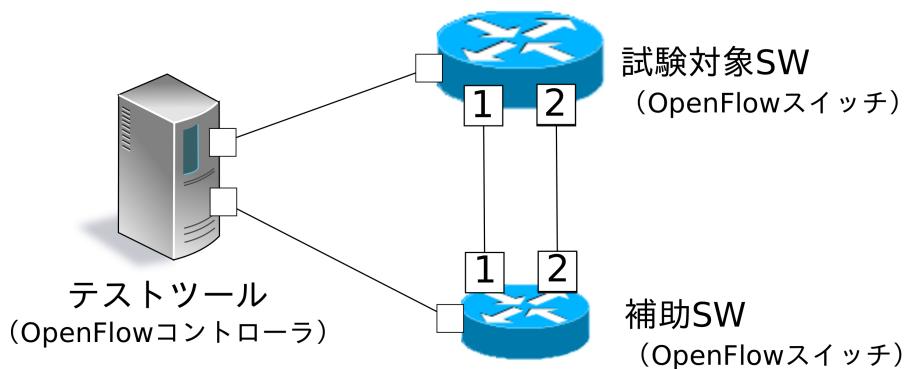
テストパターンファイルは拡張子を「.json」としたテキストファイルです。以下の形式で記述します。

```
[
    "xxxxxxxxxxxx",           # 試験項目名
    {
        "description": "xxxxxxxxxxxx", # 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)
            {...},
            {...}
        ],
        "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
]
```

10.2.2 ツール実行環境

テストツール実行のための環境を構築する必要があります。



ポート番号

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: この環境を 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。

10.2.3 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
ryu/tests/switch/tester.py	テストツール
ryu/tests/switch/of13	テストパターンファイルのサンプル
ryu/tests/switch/run_mininet.py	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

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。

```
$ ryu-manager ryu/tests/switch/tester.py
loading app ryu/tests/switch/tester.py
loading app ryu.controller.ofp_handler
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ryu/tests/switch/tester.py of OfTester
target_dpid=0000000000000001
tester_dpid=0000000000000002
Test files directory = ryu/tests/switch/of13
--- 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。

```
$ ryu-manager ryu/tests/switch/tester.py
loading app ryu/tests/switch/tester.py
loading app ryu.controller.ofp_handler
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ryu/tests/switch/tester.py of OfTester
target_dpid=0000000000000001
tester_dpid=0000000000000002
Test files directory = ryu/tests/switch/of13
--- 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
```

10.3 テストツール使用例

XXX を例に、オリジナルのテストパターンファイルを作成しテストツールを実行する手順を示します。

10.3.1 テストパターンファイルの作成

テストパターンファイルのサンプルを示します。

OpenFlow スイッチが match 条件として eth_dst パラメータに対応しているかを確認する試験サンプルです。match 条件に eth_dst を指定するフローエントリを登録し、match 条件に合致する eth_dst が設定されたパケットを印加することでパケットがフローエントリに match し転送されることを期待する試験を記述しています。

ファイル名 : sample_test_pattern.json

```
[
    "match": "03_ETH_DST",
    {
        "description": "ethernet(dst='22:22:22:22:22:22')/ipv4/tcp-->'eth_dst
=22:22:22:22:22:22,actions=output:2'",
        "prerequisite": [
            {
                "OFPFlowMod": {
                    "table_id": 0,
                    "match": {
                        "OFPMatch": {
                            "oxm_fields": [
                                {
                                    "OXMTlv": {
                                        "field": "eth_dst",
                                        "value": "22:22:22:22:22:22"
                                    }
                                }
                            ]
                        }
                    }
                }
            },
            "instructions": [
                {
                    "OFPInstructionActions": {
                        "actions": [
                            {
                                "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='192.168.20.20', 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'"
            ],
            "egress": [

```

```
"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='192.168.20.20', 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'"  
]
```

10.3.2 環境構築

10.3.3 テストツール実行