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 11 月 06 日

目次

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。

OpenFlow 1.3 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。

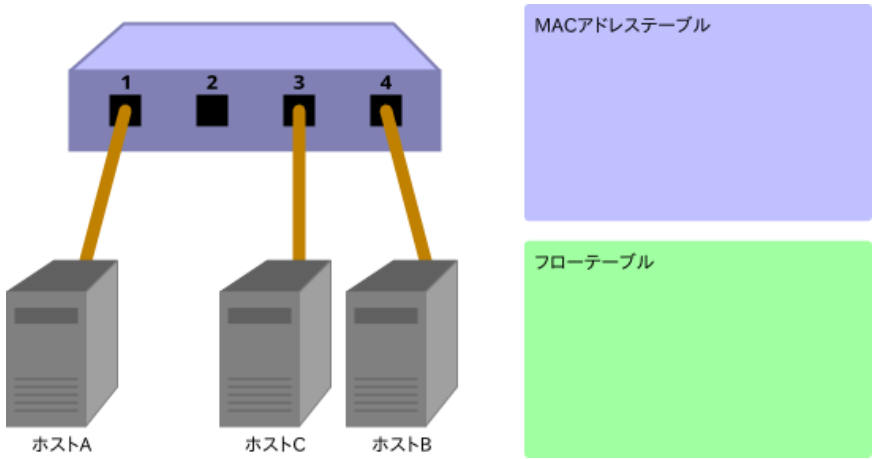
ヒント: OpenFlow では、NORMAL ポートという論理的な出力ポートがオプションで規定されており、出力ポートに NORMAL を指定すると、スイッチの L2/L3 機能を使ってパケットを処理するようになります。つまり、すべてのパケットを NORMAL ポートに出力するように指示するだけで、スイッチングハブとして動作するようにできます (スイッチが NORMAL ポートをサポートしている場合) が、ここでは各々の処理を OpenFlow を使って実現するものとします。

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。

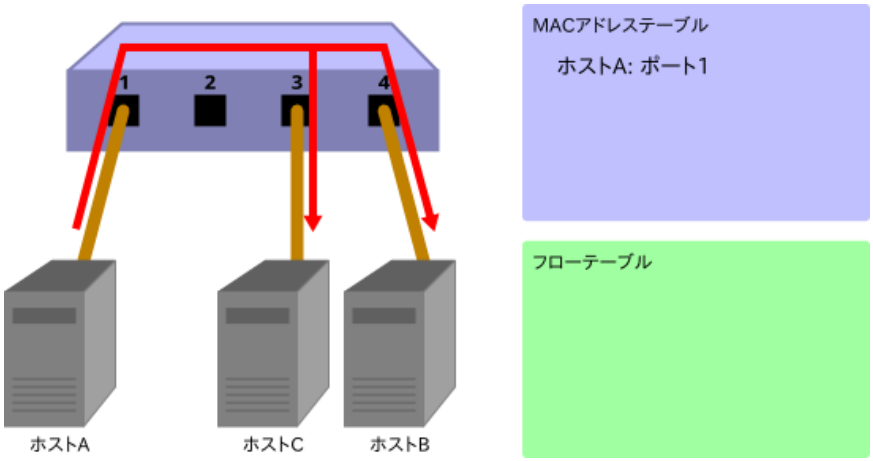
   ポート 1 にホスト A、ポート 4 にホスト B、ポート 3 にホスト C が接続されているものとします。



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:

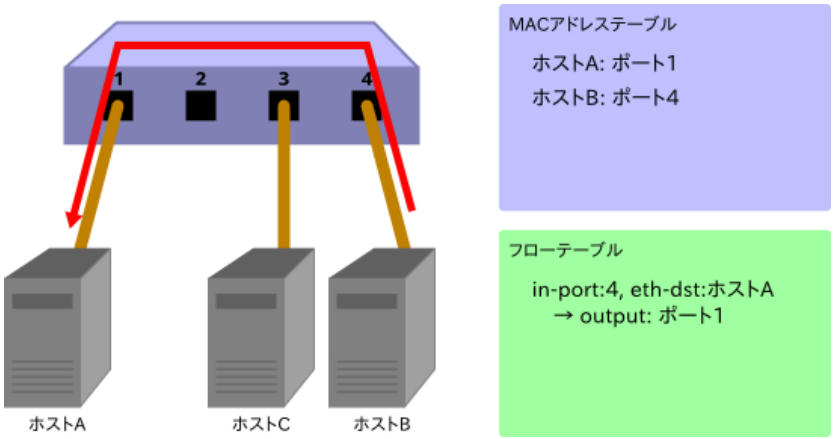
```
in-port: 1
eth-dst: ホスト B
eth-src: ホスト A
```

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:

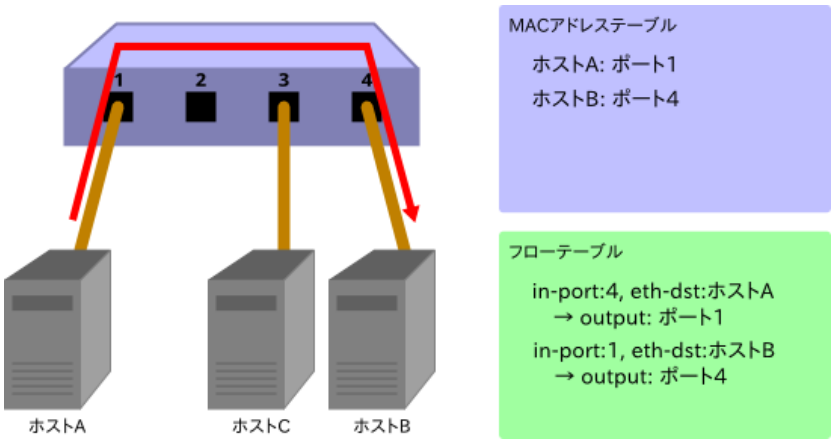
```
in-port: 4
eth-dst: ホスト A
eth-src: ホスト B
```

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.OFPIT_APPLY_ACTIONS,
                                             actions)]

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

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

```
        in_port = msg.match['in_port']

        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
ryu.controller.handler.HANDSHAKE_DISPATCHER	HELLO メッセージの交換
ryu.controller.handler.CONFIG_DISPATCHER	SwitchFeatures メッセージの受信待ち
ryu.controller.handler.MAIN_DISPATCHER	通常状態
ryu.controller.handler.DEAD_DISPATCHER	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。詳細については後述します。

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。

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、ここでは OFPCML_NO_BUFFER を指定してパケット全体を添付するようにしています。

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	ryu.ofproto.ofproto_v1_3_parser.OFPMatch 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、ryu.ofproto.ofproto_v1_3.OFP_NO_BUFFER 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(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

    # ...
```

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。

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
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

次のページに続く

表 1.1 – 前のページからの続き

Match フィールド名	説明
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。

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いきます。

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

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

```
    # ...
```

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

マッチについては既に説明しました。

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。

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。GotoTable 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、OPFInstructionActions にまとめられていて、インスタンス生成時に選択します。

スイッチングハブの実装では、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。

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。

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

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

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

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

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

datapath

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

cookie (0)

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

cookie_mask (0)

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

table_id (0)

> 操作対象のフローテーブルのテーブル ID を指定します。

command (ofproto_v1_3.OFPFC_ADD)

> どのような操作を行うかを指定します。

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

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 OFPFC_DELETE または OFPFC_DELETE_STRICT の場合は無視されます。

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

out_port (0)

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

出力ポートでのフィルタを無効にするには、`OFPP_ANY` を指定します。

out_group (0)

out_port と同様に、出力グループでフィルタします。

無効にするには、`OFPG_ANY` を指定します。

flags (0)

以下のフラグの組み合わせを指定することができます。

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

match (None)

マッチを指定します。

instructions ([])

インストラクションのリストを指定します。

パケットの転送

Packet-In ハンドラに戻り、最後の処理の説明です。

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

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

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

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

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

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

datapath

OpenFlow スイッチに対応する 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、自身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 (OF1.3) (xid=0x2):
 cookie=0x0, duration=105.975s, table=0, n_packets=0, n_bytes=0, priority=0 actions=CONTROLLER
:65535
root@ryu-vm:~#
```

優先度が 0 で、マッチがなく、アクションに CONTROLLER、送信データサイズ 65535(0xffff = OFPCML_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と思いますが、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.EventOFPStateChange,
                [MAIN_DISPATCHER, DEAD_DISPATCHER])
    def _state_change_handler(self, ev):
        datapath = ev.datapath
        if ev.state == MAIN_DISPATCHER:
            if not datapath.id in self.datapaths:
                self.logger.debug('register datapath: %016x', datapath.id)
                self.datapaths[datapath.id] = datapath
        elif ev.state == DEAD_DISPATCHER:
            if datapath.id in self.datapaths:
                self.logger.debug('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 %8d',
                             ev.msg.datapath.id, stat.port_no,
                             stat.rx_packets, stat.rx_bytes, stat.rx_errors,
                             stat.tx_packets, stat.tx_bytes, stat.tx_errors)
```

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

2.2.1 定周期処理

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

```
from operator import attrgetter

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


class SimpleMonitor(simple_switch_13.SimpleSwitch13):

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

ryu.lib.hub には、いくつかの eventlet のラッパーや基本的なクラスの実装があります。ここではスレッドを生成する hub.spawn() を使用します。実際に生成されるスレッドは eventlet のグリーンスレッドです。

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

```
                        {
                            "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 %8d',
                         ev.msg.datapath.id, stat.port_no,
                         stat.rx_packets, stat.rx_bytes, stat.rx_errors,
                         stat.tx_packets, stat.tx_bytes, stat.tx_errors)
```

OPFPortStatsReply 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 EventOFPFlowStatsReply
  CONSUMES EventOFPPortStatsReply
  CONSUMES EventOFPPacketIn
  CONSUMES EventOFPSwitchFeatures
BRICK ofp_event
  PROVIDES EventOFPStateChange TO {'SimpleMonitor': set(['main', 'dead'])}
  PROVIDES EventOFPFlowStatsReply TO {'SimpleMonitor': set(['main'])}
  PROVIDES EventOFPPortStatsReply TO {'SimpleMonitor': set(['main'])}
  PROVIDES EventOFPPacketIn TO {'SimpleMonitor': set(['main'])}
  PROVIDES EventOFPSwitchFeatures TO {'SimpleMonitor': set(['config'])}
  CONSUMES EventOFPErrorMsg
  CONSUMES EventOFPPortDescStatsReply
  CONSUMES EventOFPHello
  CONSUMES EventOFPEchoRequest
  CONSUMES EventOFPSwitchFeatures
connected socket:<eventlet.greenio.GreenSocket object at 0x343fb10> address:('127.0.0.1',
55598)
hello ev <ryu.controller.ofp_event.EventOFPHello object at 0x343fed0>
move onto config mode
EVENT ofp_event->SimpleMonitor EventOFPSwitchFeatures
switch features ev version: 0x4 msg_type 0x6 xid 0x7dd2dc58 OFPSwitchFeatures(auxiliary_id=0,
capabilities=71,datapath_id=1,n_buffers=256,n_tables=254)
move onto main mode
EVENT ofp_event->SimpleMonitor EventOFPStateChange
register datapath: 0000000000000001
send stats request: 0000000000000001
EVENT ofp_event->SimpleMonitor EventOFPFlowStatsReply
datapath         in-port  eth-dst           out-port packets  bytes
---------------- -------- ----------------- -------- -------- --------
EVENT ofp_event->SimpleMonitor EventOFPPortStatsReply
datapath         port     rx-pkts  rx-bytes rx-error tx-pkts  tx-bytes tx-error
---------------- -------- -------- -------- -------- -------- -------- --------
0000000000000001        1        0        0        0        0        0        0
0000000000000001        2        0        0        0        0        0        0
0000000000000001        3        0        0        0        0        0        0
0000000000000001 fffffffe        0        0        0        0        0        0
```

「**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 rx-error tx-pkts  tx-bytes tx-error
---------------- -------- -------- -------- -------- -------- -------- --------
0000000000000001        1        3      182        0        3      182        0
0000000000000001        2        3      182        0        3      182        0
0000000000000001        3        0        0        0        1       42        0
0000000000000001 fffffffe        0        0        0        1       42        0
```

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っていないためです。

このケースでは、ホスト 1 が最初にブロードキャストした ARP リクエスト、ホスト 2 がホスト 1 に返した ARP リプライ、ホスト 1 がホスト 2 へ発行した echo request の 3 パケットが Packet-Out 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

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。

3.1 基本的な使い方

3.1.1 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
- vlan
- 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_ipvr.proto
```

3.1.2 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'

3.1.3 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_ethernet = pkt.get_protocol(ethernet.ethernet)
if not pkt_ethernet:
    # non ethernet
    return
print pkt_ethernet.dst
print pkt_ethernet.src
print pkt_ethernet.ethertype
```

3.1.4 パケットの生成 (シリアライズ)

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
```

3.2 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.OFPInstructionActions(type_=ofproto.OFPIT_APPLY_ACTIONS,
                                             actions=actions)]
        mod = parser.OFPFlowMod(datapath=datapath,
                                priority=0,
                                match=parser.OFPMatch(),
                                instructions=inst)
        datapath.send_msg(mod)

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

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

    def _handle_icmp(self, datapath, port, pkt_ethernet, pkt_ipv4, pkt_icmp):
        if pkt_icmp.type != icmp.ICMP_ECHO_REQUEST:
            return
        pkt = packet.Packet()
        pkt.add_protocol(ethernet.ethernet(ethertype=pkt_ethernet.ethertype,
                                           dst=pkt_ethernet.src,
```

```
                                        src=self.hw_addr))
        pkt.add_protocol(ipv4.ipv4(dst=pkt_ipv4.src,
                                   src=self.ip_addr,
                                   proto=pkt_ipv4.proto))
        pkt.add_protocol(icmp.icmp(type_=icmp.ICMP_ECHO_REPLY,
                                   code=icmp.ICMP_ECHO_REPLY_CODE,
                                   csum=0,
                                   data=pkt_icmp.data))
        self._send_packet(datapath, port, pkt)

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

以下は ping -c 3 を実行した場合のログの例です。

```
EVENT ofp_event->IcmpResponder EventOFPSwitchFeatures
switch features ev version: 0x4 msg_type 0x6 xid 0xb63c802c OFPSwitchFeatures(auxiliary_id=0,
capabilities=71,datapath_id=11974852296259,n_buffers=256,n_tables=254)
move onto main mode
EVENT ofp_event->IcmpResponder EventOFPPacketIn
packet-in ethernet(dst='ff:ff:ff:ff:ff:ff',ethertype=2054,src='0a:e4:1c:d1:3e:43'), arp(dst_ip
='192.0.2.9',dst_mac='00:00:00:00:00:00',hlen=6,hwtype=1,opcode=1,plen=4,proto=2048,src_ip
='192.0.2.99',src_mac='0a:e4:1c:d1:3e:43'), '\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
x00\x00\x00\x00\x00\x00'
packet-out ethernet(dst='0a:e4:1c:d1:3e:43',ethertype=2054,src='0a:e4:1c:d1:3e:44'), arp(
dst_ip='192.0.2.99',dst_mac='0a:e4:1c:d1:3e:43',hlen=6,hwtype=1,opcode=2,plen=4,proto=2048,
src_ip='192.0.2.9',src_mac='0a:e4:1c:d1:3e:44')
EVENT ofp_event->IcmpResponder EventOFPPacketIn
packet-in ethernet(dst='0a:e4:1c:d1:3e:44',ethertype=2048,src='0a:e4:1c:d1:3e:43'), ipv4(csum
=47390,dst='192.0.2.9',flags=0,header_length=5,identification=32285,offset=0,option=None,proto
=1,src='192.0.2.99',tos=0,total_length=84,ttl=255,version=4), icmp(code=0,csum=38471,data=echo
(data='S,B\x00\x00\x00\x00\x00\x03L)(\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\
x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./\x00\x00\x00\x00\x00\x00\x00\x00',id=44565,seq=0),type
=8)
packet-out ethernet(dst='0a:e4:1c:d1:3e:43',ethertype=2048,src='0a:e4:1c:d1:3e:44'), ipv4(csum
=14140,dst='192.0.2.99',flags=0,header_length=5,identification=0,offset=0,option=None,proto=1,
src='192.0.2.9',tos=0,total_length=84,ttl=255,version=4), icmp(code=0,csum=40519,data=echo(
data='S,B\x00\x00\x00\x00\x00\x03L)(\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\
x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./\x00\x00\x00\x00\x00\x00\x00\x00',id=44565,seq=0),type
=0)
EVENT ofp_event->IcmpResponder EventOFPPacketIn
packet-in ethernet(dst='0a:e4:1c:d1:3e:44',ethertype=2048,src='0a:e4:1c:d1:3e:43'), ipv4(csum
=47383,dst='192.0.2.9',flags=0,header_length=5,identification=32292,offset=0,option=None,proto
=1,src='192.0.2.99',tos=0,total_length=84,ttl=255,version=4), icmp(code=0,csum=12667,data=echo
(data='T,B\x00\x00\x00\x00\x00Q\x17?(\x00\x00\x00\x00\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\
x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./\x00\x00\x00\x00\x00\x00\x00\x00',id=44565,seq=1),type
=8)
packet-out ethernet(dst='0a:e4:1c:d1:3e:43',ethertype=2048,src='0a:e4:1c:d1:3e:44'), ipv4(csum
=14140,dst='192.0.2.99',flags=0,header_length=5,identification=0,offset=0,option=None,proto=1,
```

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

IP 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/シリアライズしますので、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。