こんな夜中にOpenFlowでネットワークをプログラミング!! Trema編

Tremaプロジェクト　http://trema.github.com/trema/
高宮 安仁（TAKAMIYA Yasuhito）@yasuhito、すぎょう かずし（SUGYO Kazushi）、
千葉 靖伸（CHIBA Yasunobu）、鈴木 一哉（SUZUKI Kazuya）、小出 俊夫（KOIDE Toshio）@effy55

第8回 いよいよ本格的なOpenFlowプログラミングに突入！トラフィック集計スイッチを作ろう

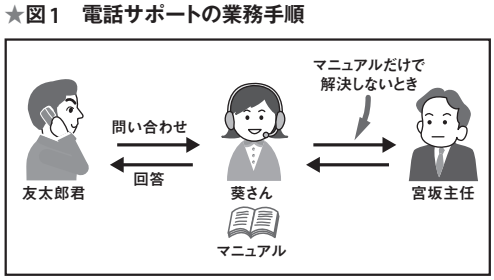
はじめに

「そろそろ頃合い」セイウチは言った
「あれやこれやの積もる話」
ルイス・キャロル
「不思議の国のアリス」

今回は盛りだくさんです！ まずは身近な例を使って、OpenFlowの動作モデルを説明します。これが理解できれば、OpenFlowの基本概念はバッチリです。次に、「トラフィック集計付きスイッチ」を実現するコントローラを実際に作ります。これはOpenFlowの重要な処理をすべて含んでいるので、応用するだけでさまざまなタイプのコントローラが作れるようになります。最後に、作成したコントローラをTremaの仮想ネットワーク上で実行します。すばらしいことに、Tremaを使えば開発から動作テストまでを開発マシン1台だけで完結できます！

では前置きはこのぐらいにして、まずはOpenFlowでスイッチを制御するしくみを理解しましょう。

★図1　電話サポートの業務手順



OpenFlowの動作モデル

OpenFlowの動作を現実世界にたとえると、製品の電話サポートサービスに似ています。

電話サポートの業務手順

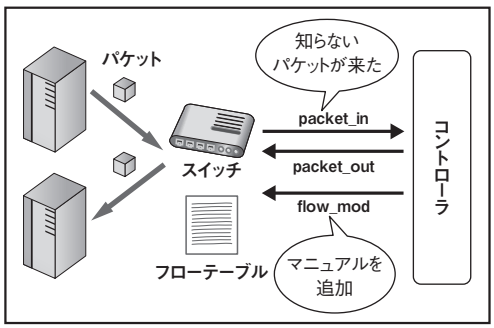
友太郎（ゆうたろう）君は、エアコンが故障したので修理に出そうと考えました（**図1**）。電話サポートに問い合わせると、サポート係の葵（あおい）さんはエアコンの症状を聞き、手元のマニュアルに対処方法が載っている場合にはこれをすぐに教えてくれます。問題は、マニュアルに対処法が載っていない場合です。このようなときは少し時間がかかりますが、上司の宮坂主任にどうしたらよいか聞きます。そして、宮坂主任からの回答が得られたら、葵さんは友太郎君に折り返し電話をします。また、次からの同じ問い合わせにはすばやく答えられるようにするため、葵さんは教わった対処法を手元のマニュアルに追加しておきます。

簡単ですね？ 信じられないかもしれませんが、あなたはすでにOpenFlowの95%を理解したも同然なのです。

OpenFlowに置き換えると……

OpenFlowでは、お客さんがパケットを発生させるホスト、電話サポート係がスイッチ、上

★図2　OpenFlowの動作モデル



司がコントローラ、マニュアルがスイッチのフローテーブル(後述)に対応します(**図2**)。

スイッチはホストからのパケットを受信すると、最初はその処理方法がわかりません。そこで、上司にあたるコントローラに問い合わせます。この問い合わせをpacket_inメッセージと呼びます。コントローラはこれを受け取ると、同様のパケットが届いた場合にスイッチでどう処理すべきか(パケットを転送する、書き換えるなど)を決めます。これをアクションと呼びます。そして「スイッチで処理すべきパケットの特徴」+「アクション」の組(フローと呼びます)をスイッチのマニュアルに追加します。この命令をflow_modメッセージと呼び、スイッチのマニュアルをフローテーブルと呼びます。処理すべきパケットの特徴とアクションをフローテーブルに書いておくことで、以後、これに当てはまるパケットはスイッチ側だけですばやく処理できます。忘れてはならないのが、packet_inメッセージで上がってきた最初のパケットです。これはコントローラに上がってきて処理待ちの状態になっているので、packet_outメッセージで適切な宛先に転送してあげます。

電話サポートとの大きな違いは、フローテーブルに書かれたフローには期限があり、これを過ぎると消えてしまうということです。これは、「マニュアルに書かれた内容は徐々に古くなるので、古くなった項目は消す必要がある」と考えるとわかりやすいかもしれません。フローが消えるタイミングでコントローラにはflow_removedメッセージが送信されます。これには、あるフローに従ってパケットがどれだけ転送されたか——電話サポートの例で言うと、マニュアルのある項目が何回参照されたか——つまり、トラフィックの集計情報が記録されています。

それではしくみの話はこのぐらいにして、早速実践に移りましょう。もし途中でわからなくなったら、この節の頭から読み直してください。

「トラフィック集計スイッチ」コントローラの概要

トラフィック集計スイッチは、パっと見は普通のL2スイッチとして動作します。しかし、裏では各ホストが送信したトラフィックをカウントしており、定期的に集計情報を表示してくれます。これを使えば、ネットワークを無駄に使いすぎているホストを簡単に特定できます。

設計と実装

「L2スイッチ機能」と「トラフィックの集計機能」のためにはどんな部品が必要でしょうか？まずは、スイッチに指示を出す上司にあたるコントローラクラスが必要です。これをTraffic Monitorクラスと名付けましょう。また、パケットを宛先のスイッチポートへ届けるためのFDBクラス[注1]、あとはトラフィックを集計するためのCounterクラスの3つが最低限必要です。

❶FDBクラス

FDBクラス(**リスト1**)は、ホストのMACアドレスとホストが接続しているスイッチポートの対応を学習するデータベースです。このデータベースを参照することで、packet_inメッセージで入ってきたパケットの宛先MACアドレスからパケット送信先のスイッチポートを決定できます。

注1) FDBとはForwarding DataBaseの略で、スイッチの一般的な機能です。詳しくは続く実装で説明します。

❷ Counterクラス

Counterクラス(**リスト2**)は、ホスト(MACアドレスで区別します)ごとの送信パケット数およびバイト数をカウントします。また、カウントした集計情報を表示するためのヘルパメソッドを提供します。

★リスト1　MACアドレス→スイッチポートのデータベースFDBクラス(fdb.rb)

```
class FDB
  def initialize
    @db = {}
  end

  def lookup mac
    if @db[ mac ]
      @db[ mac ][ :port_number ]
    else
      nil
    end
  end

  def learn mac, port_number
    if @db[ mac ]
      @db[ mac ][ :port_number ] = port_number
    else
      @db[ mac ] = { :mac => mac, :port_number => port_number }
    end
  end
end
```

連想配列(MACアドレス→スイッチポート番号)

MACアドレスからスイッチポート番号を引く

MACアドレス＋スイッチポートを学習

★リスト2　トラフィックを記録し集計するCounterクラス(counter.rb)

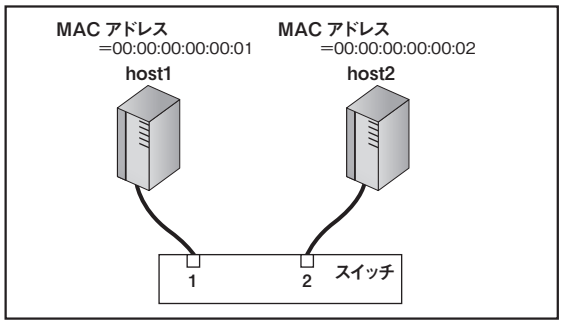
```
class Counter
  def initialize
    @db = {}
  end

  def add mac, packet_count, byte_count
    @db[ mac ] ||= { :packet_count => 0, :byte_count => 0 }
    @db[ mac ][ :packet_count ] += packet_count
    @db[ mac ][ :byte_count ] += byte_count
  end

  def each_pair &block
    @db.each_pair &block
  end
end
```

ホストごとの集計情報を記録する連想配列

ホスト(MACアドレス=mac)の送信パケット数、バイト数を追加

集計情報の表示用

★図3　TrafficMonitorを動作させるネットワーク構成の例



❸ TrafficMonitorクラス

TrafficMonitorクラスはコントローラの本体です(**リスト3**)。メインの処理はリスト3①～③の3つになります。

①packet_inメッセージが到着したとき、パケットを宛先のスイッチポートに転送し、フローテーブルを更新する部分
②flow_removedメッセージが到着したとき、トラフィック集計情報を更新する部分
③タイマーで10秒ごとにトラフィックの集計情報を表示する部分

それでは、とくに重要な①の処理を詳しく見ていきましょう。なお、リスト3中で使われているメソッドの引数などAPIの詳細については、「Trema Ruby APIドキュメント」(http://rubydoc.info/github/trema/trema/master/frames)を参照してください。

以下の説明では**図3**に示すホスト2台＋スイッチ1台からなるネットワーク構成を使います。host1からhost2にパケットを送信したときの動作シーケンスは**図4**のようになります。

❶host1からhost2を宛先としてパケットを送信すると、まずはスイッチにパケットが届く
❷スイッチのフローテーブルは最初はまっさらで、どう処理すればよいかわからない状態なので、コントローラであるTrafficMonitorにpacket_inメッセージを送る
❸TrafficMonitorのpacket_inメッセージハンドラでは、packet_inメッセージのin_port(host1のつながるスイッチポート)とhost1

のMACアドレスをFDBに記録する

❹また、Counterに記録されたhost1の送信トラフィックを1パケット分増やす

❺packet_inメッセージの宛先MACアドレスから転送先のスイッチポート番号をFDBに問い合わせる。この時点ではhost2のスイッチポー

★リスト3　本体TrafficMonitorクラス（traffic-monitor.rb）

```
require "counter"
require "fdb"

class TrafficMonitor < Controller
  periodic_timer_event :show_counter, 10 ―――――――――――――― ③

  def start
    @counter = Counter.new ←――― Counterオブジェクト
    @fdb = FDB.new ←――― FDBオブジェクト
  end

  def packet_in datapath_id, message ―――――――――――――― ①
    macsa = message.macsa ←――― パケットを送信したホストのMACアドレス
    macda = message.macda ←――― パケットの宛先ホストのMACアドレス

    @fdb.learn macsa, message.in_port
    @counter.add macsa, 1, message.total_len
    out_port = @fdb.lookup( macda )
    if out_port
      packet_out datapath_id, message, out_port
      flow_mod datapath_id, macsa, macda, out_port
    else
      flood datapath_id, message
    end
  end

  def flow_removed datapath_id, message ―――――――――――――― ②
    @counter.add message.match.dl_src,message.packet_count, message.byte_count
  end

  private ←――― 以下、プライベートメソッド

  def show_counter ←――― カウンタを表示
    puts Time.now
    @counter.each_pair do | mac, counter |
      puts "#{ mac } #{ counter[ :packet_count ] } packets (#{ counter[ :byte_count ] } bytes)"
    end
  end

  def flow_mod datapath_id, macsa, macda, out_port ←――― macsaからmacdaへのパケットをout_portへ転送するflow_modを打つ
    send_flow_mod_add(
      datapath_id,
      :hard_timeout => 10, ←――― flow_modの有効期限は10秒
      :match => Match.new( :dl_src => macsa, :dl_dst => macda ),
      :actions => Trema::ActionOutput.new( out_port )
    )
  end

  def packet_out datapath_id, message, out_port ←――― packet_inしたメッセージをout_portへ転送
    send_packet_out(
      datapath_id,
      :packet_in => message,
      :actions => Trema::ActionOutput.new( out_port )
    )
  end

  def flood datapath_id, message ←――― packet_inしたメッセージをin_port以外の全スイッチポートへ転送
    packet_out datapath_id, message, OFPP_FLOOD
  end
end
```

トは学習していないので、結果は「不明」

❻そこで、パケットをin_port以外のすべてのスイッチポートに出力するpacket_outメッセージ(FLOODと呼ばれる)をスイッチに送り、host2が受信してくれることを期待する

❼スイッチは、パケットをin_port以外のすべてのポートに出す

これで、最終的にhost2がパケットを受信できます。逆に、この状態でhost1を宛先としてhost2からパケットを送信したときの動作シーケンスは次のとおりになります(図5)。❹までの動作は図4と同じですが、❺からの動作が次のように異なります。

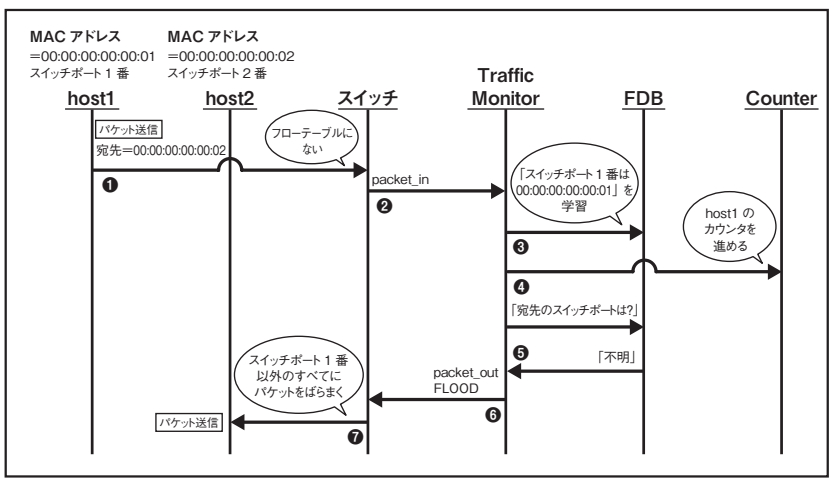
❺packet_inメッセージの宛先MACアドレスから、転送先のスイッチポート番号をFDBに問い合わせる。これは、先ほどhost1からhost2にパケットを送った時点でFDBに学習させているので、送信先はスイッチポート1番ということがわかる

❻そこで、TrafficMonitorはパケットをスイッチポート1番へ出力するpacket_outメッセージをスイッチに送る。スイッチはこれを受け取ると、パケットをスイッチポート1番に出し、最終的にhost1がパケットを受信する

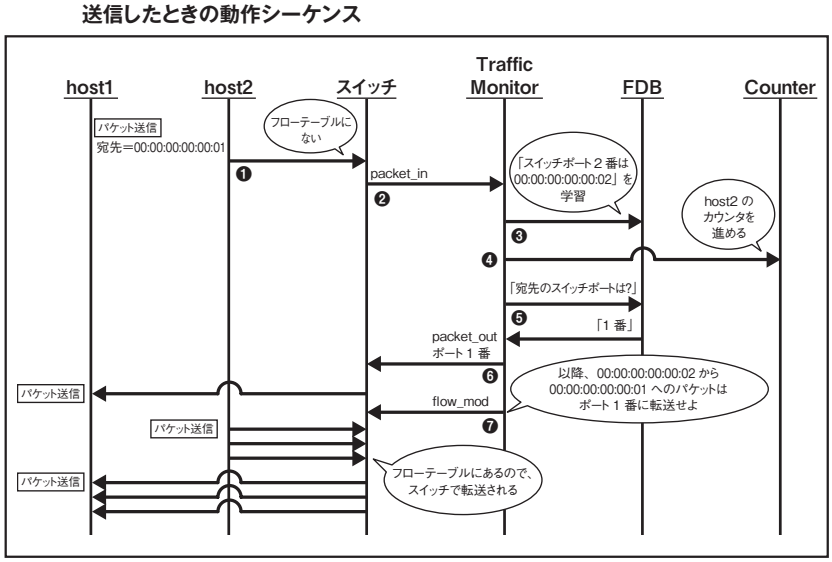
❼「送信元=00:00:00:00:00:02、送信先=00:00:00:00:00:01となるパケットはスイッチポート1番に転送せよ」というflow_modメッセージをスイッチに送信する

最後の❼によって、以降のhost2からhost1へのパケットはすべてスイッチ側だけで処理されるようになります。

★図4　host1からhost2宛にパケットを送信したときの動作シーケンス



★図5　host1からhost2宛にパケットを送信したあと、host2からhost1宛にパケットを送信したときの動作シーケンス



実行してみよう

それでは、早速実行してみましょう[注2]。**リスト4**の内容の仮想ネットワーク設定をtraffic-monitor.confとして保存し、次のように実行してください。

```
% ./trema run ./traffic-monitor.rb ↵
-c ./traffic-monitor.conf
```

実行すると、図3に示した仮想ネットワークが構成され、TrafficMonitorコントローラが起動します。

それでは、実際にトラフィックを発生させて集計されるか見てみましょう。Tremaのsend_packetsコマンドを使うと、仮想ホスト間で簡単にパケットを送受信できます。別ターミナルを開き、次のコマンドを入力してください。

```
% ./trema send_packets --source host1 ↵
--dest host2 --n_pkts 10 --pps 10
```

host1からhost2宛にパケットを10個送る

```
% ./trema send_packets --source host2 ↵
--dest host1 --n_pkts 10 --pps 10
```

host2からhost1宛にパケットを10個送る

trema runを実行した元のターミナルに次のような出力が出ていれば成功です[注3]。

```
.....
00:00:00:00:00:01 10 packets (640 bytes)
```

host1からパケットが10個送信された

```
00:00:00:00:00:02 10 packets (640 bytes)
.....
```

host2からパケットが10個送信された

★リスト4　仮想スイッチ0xabcに仮想ホストhost1、host2を接続する設定

```
vswitch {                                  ← 仮想スイッチ0xabcを定義
  datapath_id 0xabc
}

vhost ("host1") {                          ← 仮想ホストhost1を定義
  ip "192.168.0.1"
  mac "00:00:00:00:00:01"
}
vhost ("host2") {                          ← 仮想ホストhost2を定義
  ip "192.168.0.2"
  mac "00:00:00:00:00:02"
}

link "0xabc", "host1"                      ← ホストhost1,host2をスイッチ0xabcに接続
link "0xabc", "host2"
```

まとめ

今回は「トラフィック集計機能付きスイッチ」を実現するコントローラを書きました。学んだことは次の2つです。

- 電話サポートの例を使ってOpenFlowの動作モデルを学びました。パケットの転送はスイッチ上のフローテーブルによって行われ、flow_modメッセージによって書き換えることができます。また、フローテーブルに登録されていないパケットによってpacket_inメッセージがコントローラに届きます。
- 仮想ネットワークを使ったコントローラの動作テスト方法を学びました。仮想スイッチと仮想ホストを起動してつなぎ、send_packetsコマンドを使って仮想ホスト間でパケットを送受信することで、コントローラの簡単な動作テストができます。

次回はTremaを使ったテストファースト開発を紹介します。RailsやSinatraを使ったWebアジャイル開発ではお馴染みのテストファースト開発ですが、もちろんTremaでもサポートしています。とくにOpenFlowコントローラのように複雑な動作シーケンスを持つソフトウェアの実装には、テストファーストによるインクリメンタルな開発が有効です。SD

注2）Tremaのセットアップが済んでいない人は、先月号(11月号)の本連載記事またはTremaのドキュメントを参考にセットアップしておいてください。なお、Tremaは頻繁に更新されていますので、すでにインストールしている人も最新版にアップデートすることをお勧めします。

注3）その他のトラフィック情報も出るかもしれませんが、これはLinuxカーネルが送っているIPv6のパケットなので、host1、host2とは関係ありません。