# 作ってわかる OpenFlow
# Trema チュートリアル & ハンズオン

Trema 開発チーム
鈴木一哉、高宮安仁

# OpenFlowとは

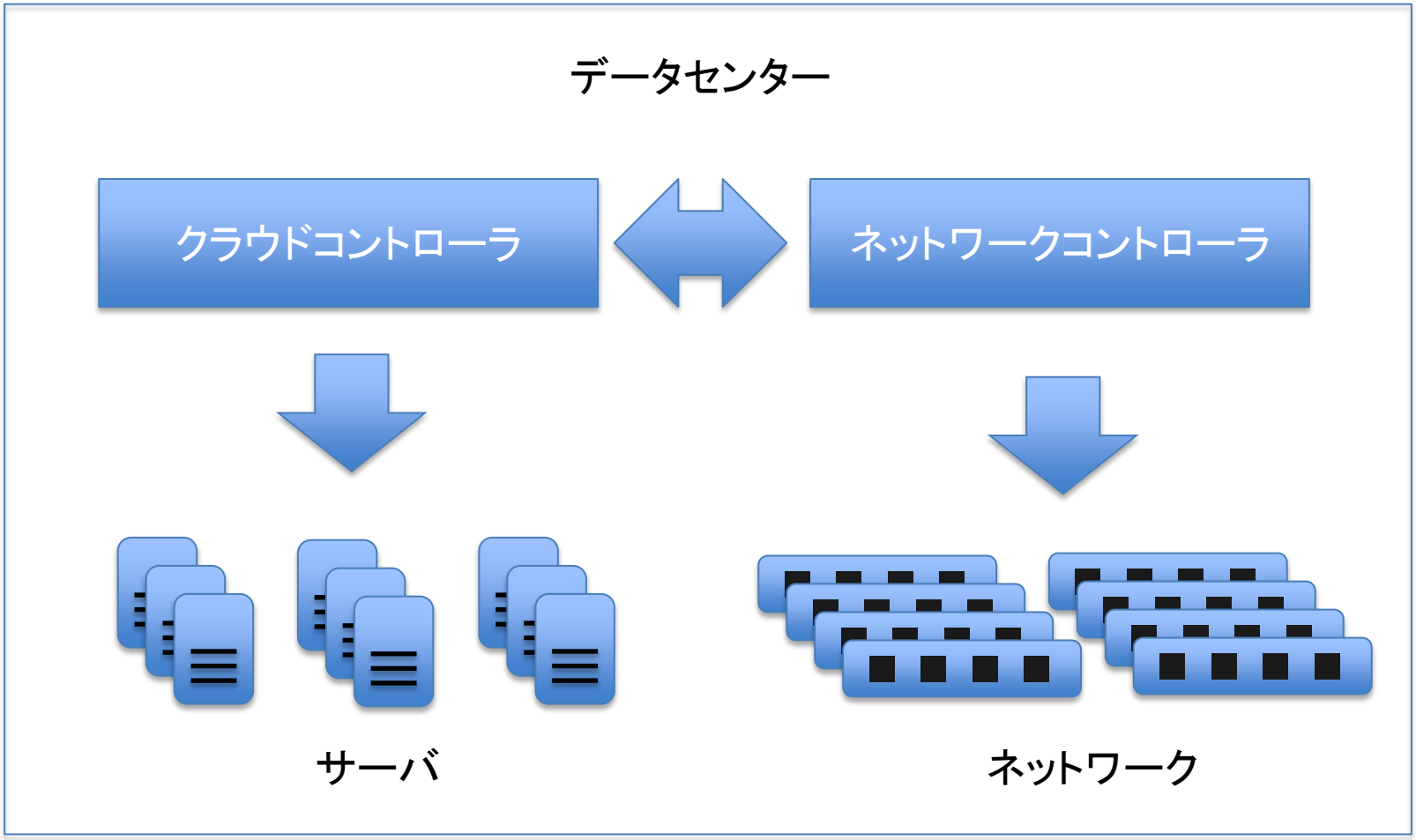
# 背景

- データセンター利用の拡大
  - データセンターの大規模化
  - 低コストに構築・運用する必要性
    - 人件費
    - 装置コスト

# データセンター低コスト化

- 人件費の削減
  - ソフトウェアによる運用の自動化
- コモディティハードウェアの利用
  - ソフトウェアによる障害回復

# 大規模データセンターの自動化



# 現状のネットワーク機器の課題(1)

- 多様なユーザニーズを満たすために、多様なプロトコル標準が存在
  - 装置仕様の肥大化
  - 装置コストの高騰

→ 実際に使われるのは、一部の機能のみ

# 現状のネットワーク機器の課題(2)

- 自律分散が故の動作の複雑さ

→ 外部から制御した方が、シンプルで効果的な場合も

# 現状のネットワーク機器の課題(3)

- ネットワークのオペレーションコスト
  - ネットワーク機器の増設
  - ファームウェアアップデート
  - 設定変更

  → 収容しているユーザに影響を与えないように行う必要がある。
  → 特に昨今のデータセンターでは、リードタイムの短縮も同時に求められる

- ネットワーク運用にも自動化が求められている

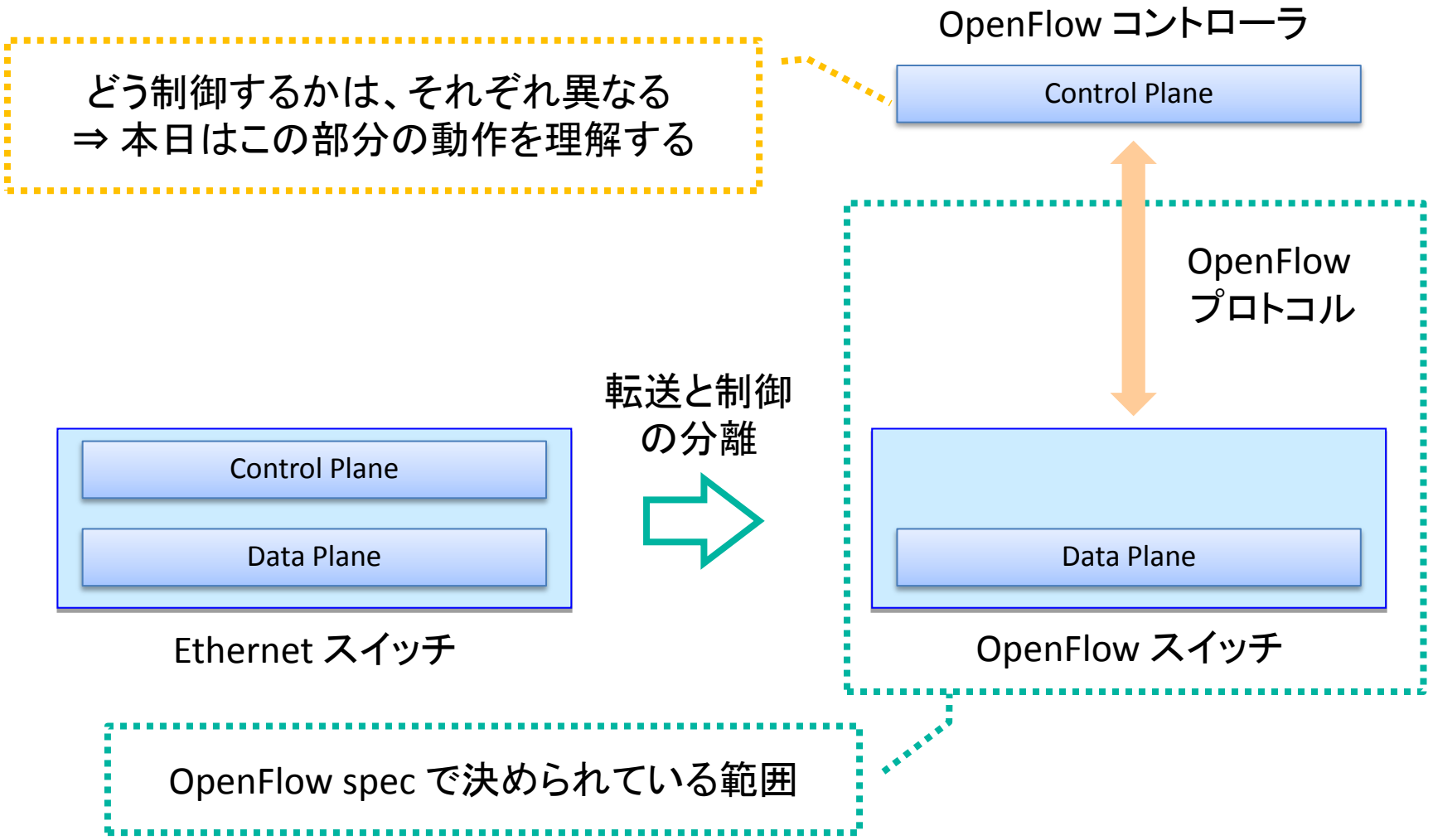
# Software Defined Network

- ネットワークも、ソフトウェアで制御

  ⇒ Software Defined Network

  - アプリケーションに合わせた最適化
  - 時間帯毎に異なるコンフィグレーションで運用
  - より柔軟な自己修復

# OpenFlow とは


どう制御するかは、それぞれ異なる
⇒ 本日はこの部分の動作を理解する
OpenFlow コントローラ
Control Plane
OpenFlow
プロトコル
転送と制御
の分離
Control Plane
Data Plane
Ethernet スイッチ
Data Plane
OpenFlow スイッチ
OpenFlow spec で決められている範囲

# 本日のゴール

- “Hello World” から始め、7 つの段階を踏んで開発を行います
- コントローラ開発サイクルの体験を通して、OpenFlow のしくみを学びます

Trema を使ったコントローラ開発の第一歩

# Task A : Hello Trema

# ここで学ぶこと

- Trema によるコントローラ開発入門
  - Trema の基本操作
  - コントローラのコーディング

- 必要となる Ruby の基礎知識

# Hello, Trema!

```
class HelloTrema < Controller
  def start
    puts “Hello, Trema!”
  end
end
```

# 演習: "Hello Trema!" を実行

- 以下のコマンドを入力し、Trema を実行

```
$ cd iw2012.files
$ trema run hello-trema.rb
Password: xxxxxxxx  # Enter your password here
Hello Trema!   # Ctrl-C to quit
```

# 基本コマンド : `trema run`

```
$ trema run [controller-file]
```

- 上記コマンドで、コントローラを実行します
- Ctrl-c で停止します
- `trema help run` でオプションリストを表示します

# 書いたコードをすぐ動かす

- `trema run` コマンドで、書いたコントローラをすぐ実行できます
- 書いたコントローラを、即座にテストできます
- 短いサイクルで "コーディング、テスト、デバッグ" を繰り返す開発スタイルを実現できます

Task A : Hello Trema

# RUBY 言語入門
# ～ まずは品詞を抑えよう～

# キーワード (予約語)

- alias and BEGIN begin break case class def defined do else elsif END end ensure false for if in module next nil not or redo rescue retry return self super then true undef unless until when while yield

# キーワード (予約語)

- alias and BEGIN begin break case class def defined do else elsif END end ensure false for if in module next nil not or redo rescue retry return self super then true undef unless until when while yield

# Hello, Trema!

```
class HelloTrema < Controller
  def start
    puts “Hello, Trema!”
  end
end
```

# 定数 (固有名詞)

- TokyoTower
  (Tokyo Tower)
- TheEmpireStateBuilding
  (The Empire State Building)
- HelloTrema
- Controller

# Hello, Trema!

```
class HelloTrema < Controller
  def start
    puts “Hello, Trema!”
  end
end
```

# 文法

- 品詞の組み合わせかた

# 文法 (クラス定義)

```
class HelloTrema < Controller
  def start
    puts “Hello, Trema!”
  end
end
```

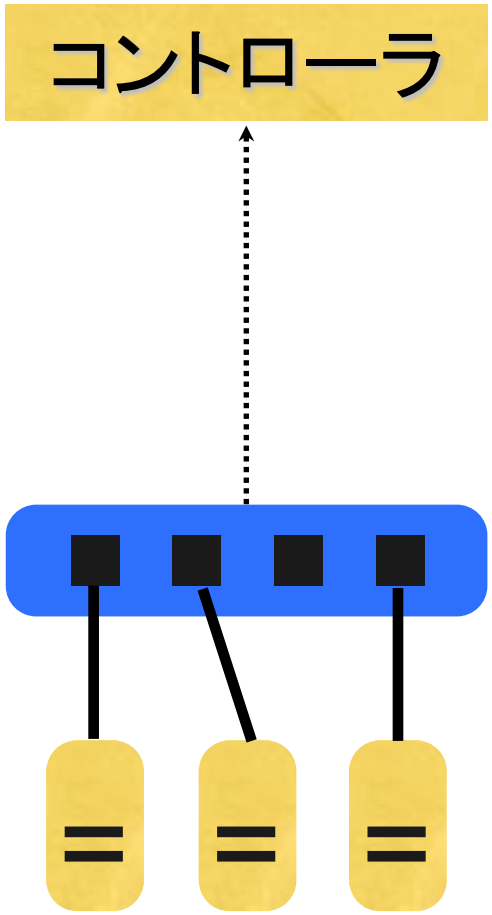
# コントローラクラス

```
class HelloTrema < Controller
  #...
end
```

- すべてのコントローラは、クラスとして実装します
  - `class HelloTrema`
- Trema クラスライブラリに用意されている `Controller` クラスのサブクラスとして実装します
- コントローラに必要なメソッドは、自動的に継承されます (flow-mod メッセージの送信等)

# OpenFlow フレームワーク

# OpenFlow フレームワークとは




ここを作る
ためのもの

# 主なフレームワーク

| 名前 | 開発言語 | 開発元 | ライセンス |
|---|---|---|---|
| Trema | Ruby, C | Trema プロジェクト | GPL2 |
| POX | Python | UC バークレイ | GPL3 |
| NOX | C++ | Nicira, スタンフォード大学, UC バークレイ | GPL3 |
| Floodlight | Java | Big Switch Networks | Apache |

# Trema でハブ

```
class RepeaterHub < Controller
  def packet_in datapath_id, message
    send_flow_mod_add(
      datapath_id,
      :match => ExactMatch.from( message ),
      :actions => SendOutPort.new( OFPP_FLOOD )
    )
    send_packet_out(
      datapath_id,
      :packet_in => message,
      :actions => SendOutPort.new( OFPP_FLOOD )
    )
  end
end
```

# POX (Python)

```
from pox.core import core
import pox.openflow.libopenflow_01 as of

class RepeaterHub (object):
  def __init__ (self, connection):
    self.connection = connection
    connection.addListeners(self)

  def send_packet (self, buffer_id, raw_data, out_port, in_port):
    msg = of.ofp_packet_out()
    msg.in_port = in_port
    if buffer_id != -1 and buffer_id is not None:
      msg.buffer_id = buffer_id
    else:
      if raw_data is None:
        return
      msg.data = raw_data
    action = of.ofp_action_output(port = out_port)
    msg.actions.append(action)
    self.connection.send(msg)

  def act_like_hub (self, packet, packet_in):
    self.send_packet(packet_in.buffer_id, packet_in.data,
                     of.OFPP_FLOOD, packet_in.in_port)

  def _handle_PacketIn (self, event):
    packet = event.parsed
    if not packet.parsed:
      return
    packet_in = event.ofp # The actual ofp_packet_in message.
    self.act_like_hub(packet, packet_in)

def launch ():
  def start_switch (event):
    RepeaterHub(event.connection)
  core.openflow.addListenerByName("ConnectionUp", start_switch)
```

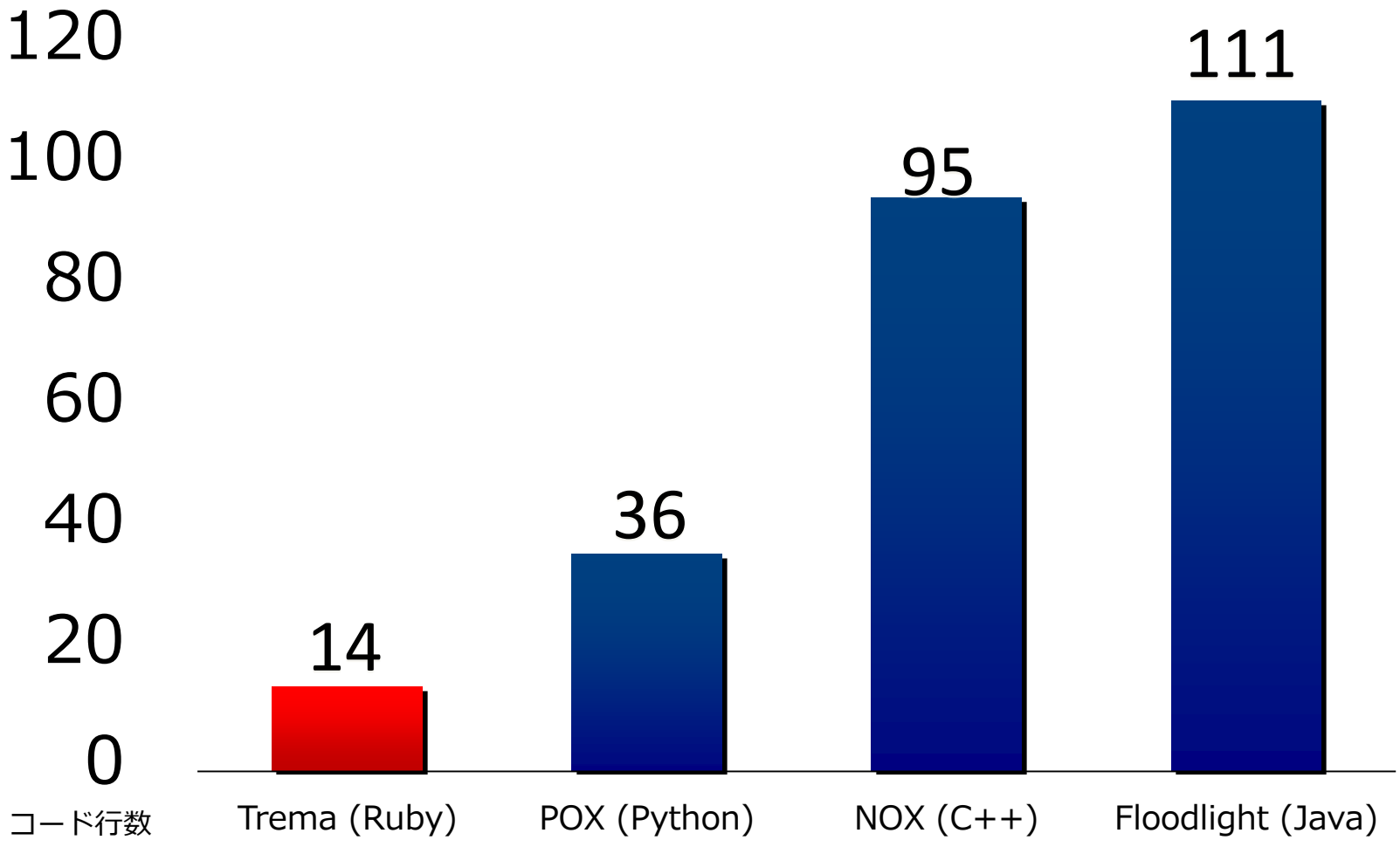
# NOX (C++)

```
#include <boost/bind.hpp>
#include <boost/shared_array.hpp>
#include "assert.hh"
#include "component.hh"
#include "flow.hh"
#include "packet-in.hh"
#include "vlog.hh"

#include "netinet++/ethernet.hh"

namespace {

using namespace vigil;
using namespace vigil::container;

Vlog_module lg("hub");

class Hub
    : public Component
{
public:
    Hub(const Context* c,
        const json_object*)
        : Component(c) { }

    void configure(const Configuration*) {
    }

    Disposition handler(const Event& e)
    {
        const Packet_in_event& pi = assert_cast<const Packet_in_event&>(e);
        uint32_t buffer_id = pi.buffer_id;
        Flow flow(pi.in_port, *(pi.get_buffer()));

        if (flow.dl_type == ethernet::LLDP){
            return CONTINUE;
        }

        ofp_flow_mod* ofm;
        size_t size = sizeof *ofm + sizeof(ofp_action_output);
        boost::shared_array<char> raw_of(new char[size]);
        ofm = (ofp_flow_mod*) raw_of.get();

        ofm->header.version = OFP_VERSION;
        ofm->header.type = OFPT_FLOW_MOD;
        ofm->header.length = htons(size);
        ofm->match.wildcards = htonl(0);
        ofm->match.in_port = htons(flow.in_port);
        ofm->match.dl_vlan = flow.dl_vlan;
        ofm->match.dl_vlan_pcp = flow.dl_vlan_pcp;
        memcpy(ofm->match.dl_src, flow.dl_src.octet, sizeof ofm->match.dl_src);
        memcpy(ofm->match.dl_dst, flow.dl_dst.octet, sizeof ofm->match.dl_dst);
        ofm->match.dl_type = flow.dl_type;
        ofm->match.nw_src = flow.nw_src;
        ofm->match.nw_dst = flow.nw_dst;
        ofm->match.nw_proto = flow.nw_proto;
        ofm->match.tp_src = flow.tp_src;
        ofm->match.tp_dst = flow.tp_dst;
        ofm->cookie = htonl(0);
        ofm->command = htons(OFPFC_ADD);
        ofm->buffer_id = htonl(buffer_id);
        ofm->idle_timeout = htons(5);
        ofm->hard_timeout = htons(5);
        ofm->priority = htons(OFP_DEFAULT_PRIORITY);
        ofm->flags = htons(0);
        ofp_action_output& action = *((ofp_action_output*)ofm->actions);
        memset(&action, 0, sizeof(ofp_action_output));
        action.type = htons(OFPAT_OUTPUT);
        action.len = htons(sizeof(ofp_action_output));
        action.port = htons(OFPP_FLOOD);
        action.max_len = htons(0);
        send_openflow_command(pi.datapath_id, &ofm->header, true);
        free(ofm);

        if (buffer_id == UINT32_MAX) {
            size_t data_len = pi.get_buffer()->size();
            size_t total_len = pi.total_len;
            if (total_len == data_len) {
                send_openflow_packet(pi.datapath_id, *pi.get_buffer(),
                        OFPP_FLOOD, pi.in_port, true);
            }
        }

        return CONTINUE;
    }

    void install()
    {
        register_handler<Packet_in_event>(boost::bind(&Hub::handler, this, _1));
    }
};

REGISTER_COMPONENT(container::Simple_component_factory<Hub>, Hub);

}
```

# Floodlight (Java)

```
package net.floodlightcontroller.hub;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.Map;
import net.floodlightcontroller.core.FloodlightContext;
import net.floodlightcontroller.core.IFloodlightProviderService;
import net.floodlightcontroller.core.IOFMessageListener;
import net.floodlightcontroller.core.IOFSwitch;
import net.floodlightcontroller.core.module.FloodlightModuleContext;
import net.floodlightcontroller.core.module.FloodlightModuleException;
import net.floodlightcontroller.core.module.IFloodlightModule;
import net.floodlightcontroller.core.module.IFloodlightService;
import org.openflow.protocol.OFMessage;
import org.openflow.protocol.OFPacketIn;
import org.openflow.protocol.OFPacketOut;
import org.openflow.protocol.OFPort;
import org.openflow.protocol.OFType;
import org.openflow.protocol.action.OFAction;
import org.openflow.protocol.action.OFActionOutput;
import org.openflow.util.U16;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Hub implements IFloodlightModule, IOFMessageListener {
  protected static Logger log = LoggerFactory.getLogger(Hub.class);
  protected IFloodlightProviderService floodlightProvider;

  public void setFloodlightProvider(IFloodlightProviderService floodlightProvider) {
    this.floodlightProvider = floodlightProvider;
  }

  @Override
  public String getName() {
    return Hub.class.getPackage().getName();
  }

  public Command receive(IOFSwitch sw, OFMessage msg, FloodlightContext cntx) {
    OFPacketIn pi = (OFPacketIn) msg;
    OFPacketOut po = (OFPacketOut) floodlightProvider.getOFMessageFactory()
        .getMessage(OFType.PACKET_OUT);
    po.setBufferId(pi.getBufferId())
      .setInPort(pi.getInPort());

    OFActionOutput action = new OFActionOutput()
      .setPort((short) OFPort.OFPP_FLOOD.getValue());
    po.setActions(Collections.singletonList((OFAction)action));
    po.setActionsLength((short) OFActionOutput.MINIMUM_LENGTH);

    if (pi.getBufferId() == 0xffffffff) {
      byte[] packetData = pi.getPacketData();
      po.setLength(U16.t(OFPacketOut.MINIMUM_LENGTH
          + po.getActionsLength() + packetData.length));
      po.setPacketData(packetData);
    } else {
      po.setLength(U16.t(OFPacketOut.MINIMUM_LENGTH
          + po.getActionsLength()));
    }
    try {
      sw.write(po, cntx);
    } catch (IOException e) {
      log.error("Failure writing PacketOut", e);
    }

    return Command.CONTINUE;
  }

  @Override
  public boolean isCallbackOrderingPrereq(OFType type, String name) {
    return false;
  }

  @Override
  public boolean isCallbackOrderingPostreq(OFType type, String name) {
    return false;
  }

  @Override
  public Collection<Class<? extends IFloodlightService>> getModuleServices() {
    return null;
  }

  @Override
  public Map<Class<? extends IFloodlightService>, IFloodlightService>
      getServiceImpls() {
    return null;
  }

  @Override
  public Collection<Class<? extends IFloodlightService>>
      getModuleDependencies() {
    Collection<Class<? extends IFloodlightService>> l =
        new ArrayList<Class<? extends IFloodlightService>>();
    l.add(IFloodlightProviderService.class);
    return l;
  }

  @Override
  public void init(FloodlightModuleContext context)
      throws FloodlightModuleException {
    floodlightProvider =
        context.getServiceImpl(IFloodlightProviderService.class);
  }

  @Override
  public void startUp(FloodlightModuleContext context) {
    floodlightProvider.addOFMessageListener(OFType.PACKET_IN, this);
  }
}
```

# コード行数




※ Trema (C) でも 45 行くらいです

# Trema とは

- Ruby と C 向けの OpenFlow コントローラ開発フレームワーク
  - GPL2
  - http://github.com/trema/trema
- “Post-Rails” 高い生産性を実現
  - 書いたコードをすぐ動かせる
  - よくある処理を短く書ける
  - 統合されたテスト環境

# Trema =

OpenFlow コントローラ向けライブラリ

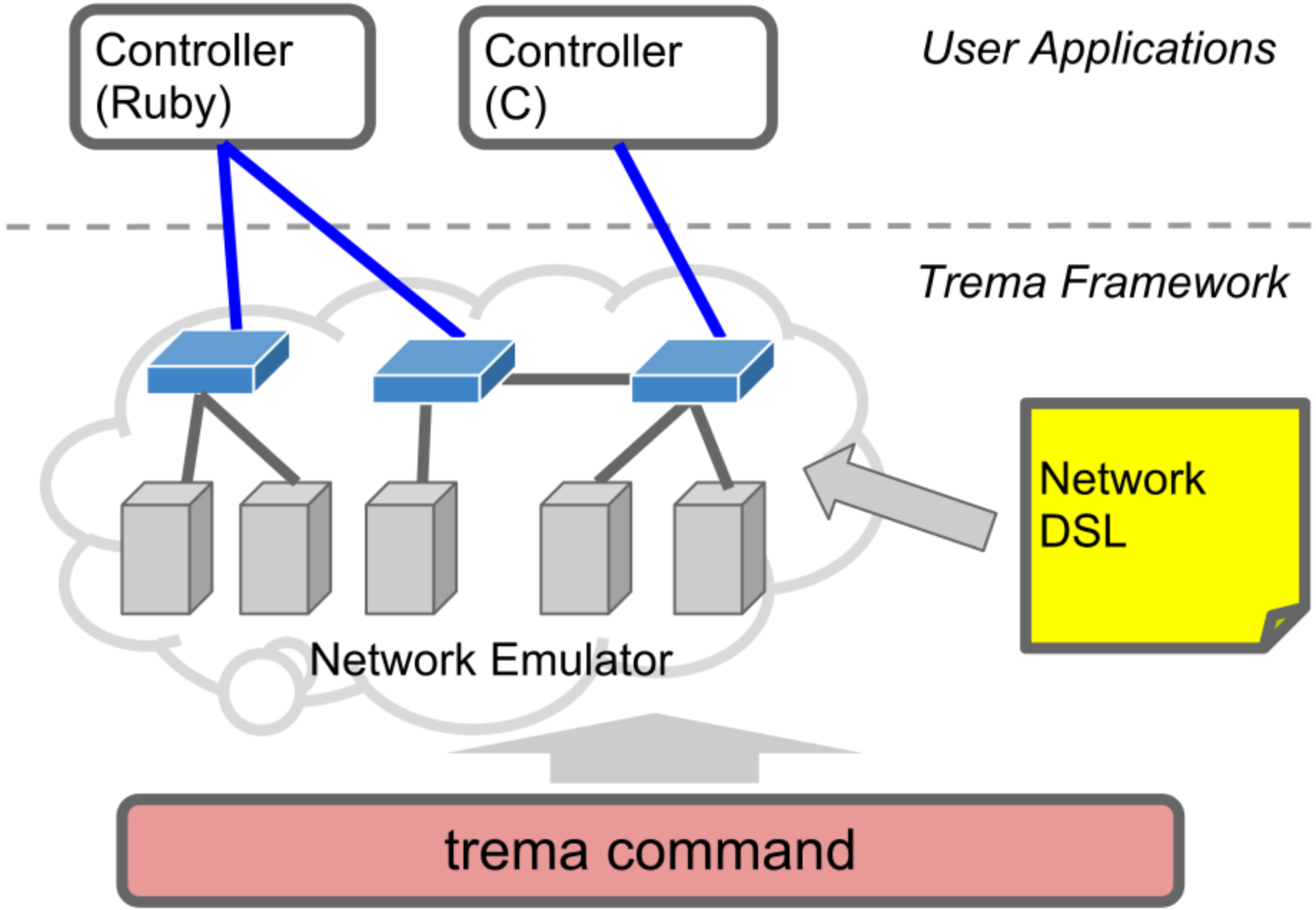
(Ruby and C)

+

ネットワークエミュレータ

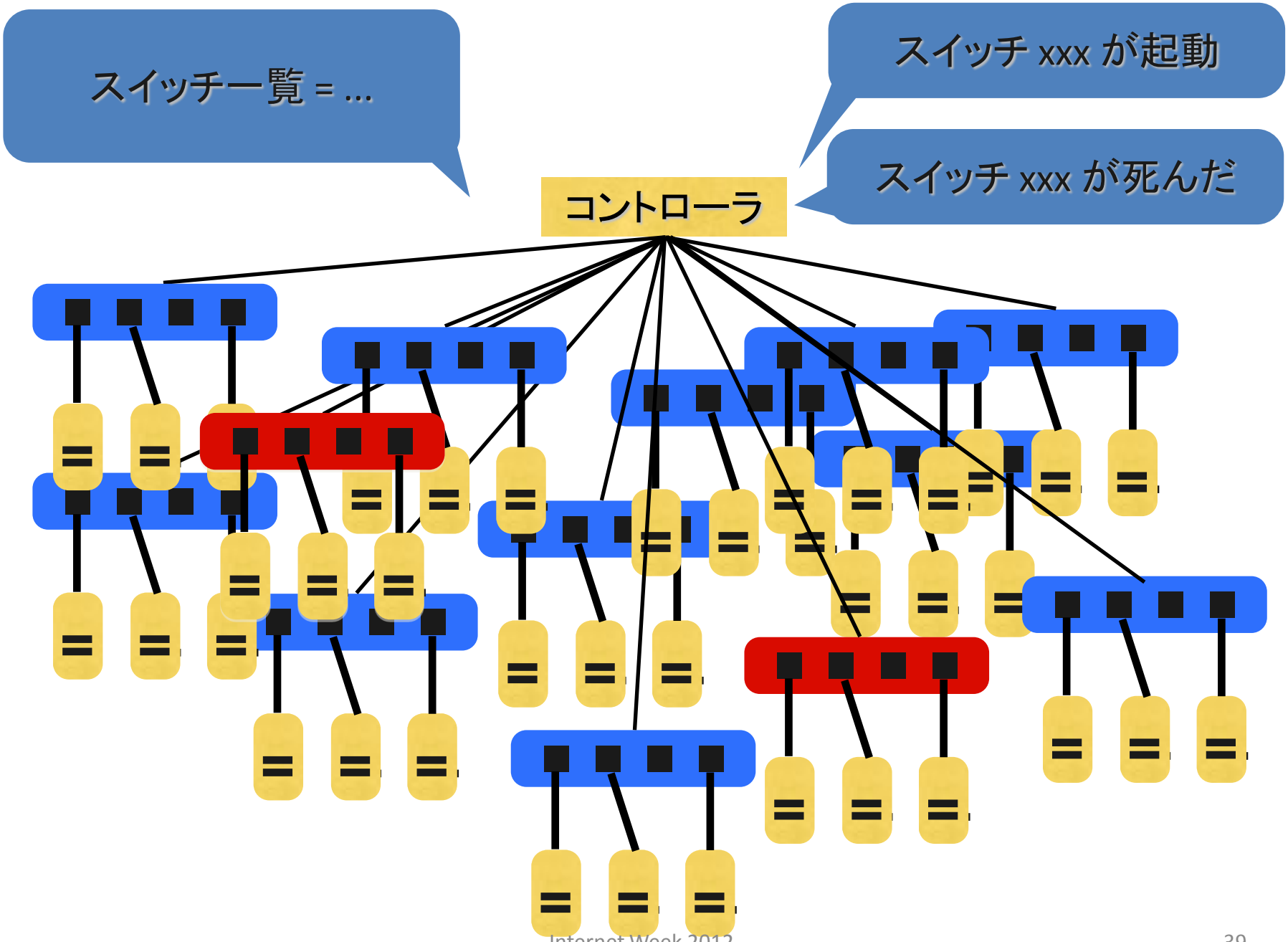
+

`trema` コマンド


Controller
(Ruby)
Controller
(C)
User Applications
Trema Framework
Network
DSL
Network Emulator
trema command

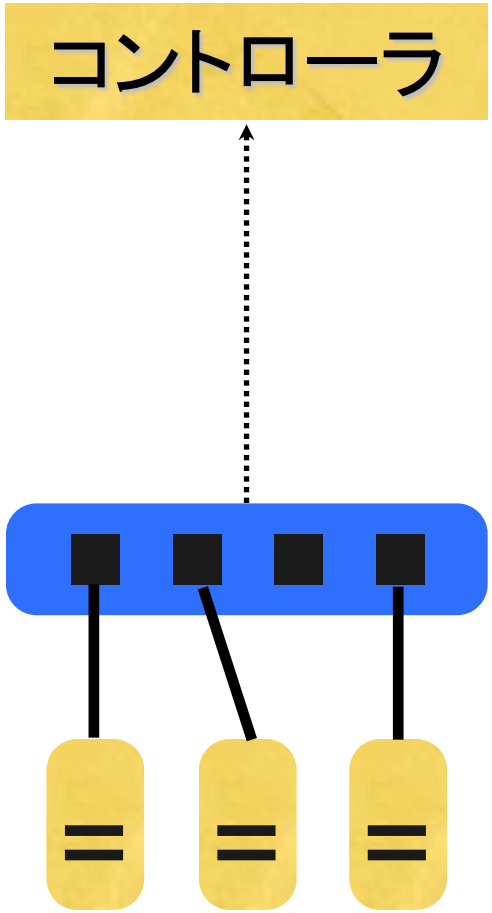
OpenFlow スイッチとコントローラを接続

# Task B : Hello Switch


スイッチ一覧 = …
スイッチ xxx が起動
スイッチ xxx が死んだ
コントローラ

# ここで学ぶこと

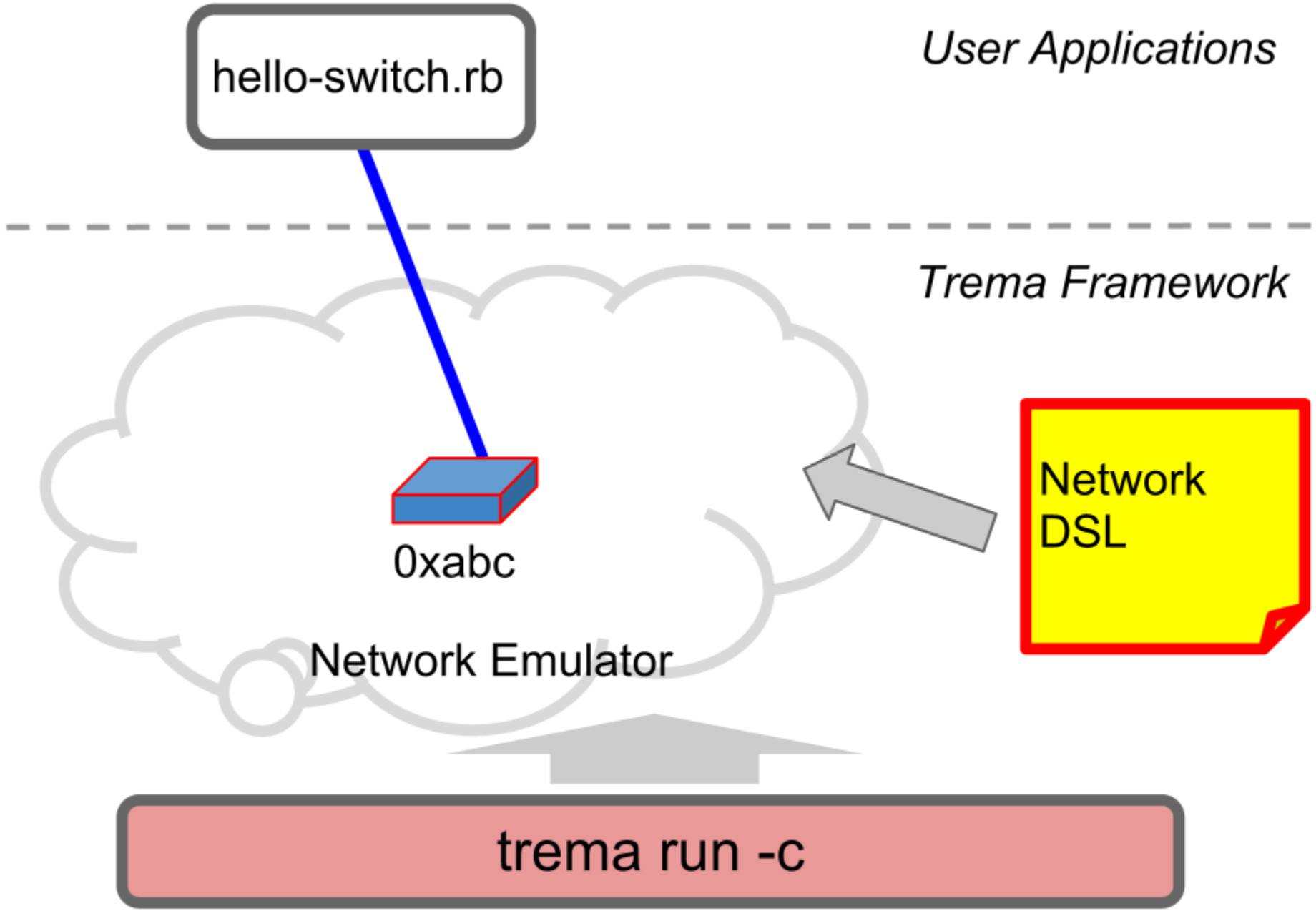
- スイッチ ⇔ コントローラ間の接続の仕組み



# 演習 : Hello Switch コントローラ

```
$ trema run hello-switch.rb -c hello-switch.conf
Password: xxxxxxxx  # Enter your password here
Hello 0xabc!  # Ctrl-c to quit
```

- ソフトウェア版 OpenFlow スイッチ (dpid = 0xabc) を起動し、コントローラと接続します
- コントローラは `“Hello 0xabc!”` と表示します
- ソフトウェア版 OpenFlow スイッチの起動は `hello-switch.conf` に定義します


hello-switch.rb
User Applications
Trema Framework
0xabc
Network Emulator
Network
DSL
trema run -c

# hello-switch.conf

```
#
# Add a switch with dpid == 0xabc
#
vswitch { dpid "0xabc" }
# or
vswitch { datapath_id "0xabc" }
```

- ソフトウェア版 OpenFlow スイッチが起動し、コントローラとのコネクションを確立します
- Trema は Full-stack の開発フレームワークです。ノート PC が一台あれば、物理スイッチを持っていなくても開発ができます

# `hello-switch.rb`

```
class HelloSwitch < Controller
  def switch_ready dpid
    puts "Hello #{ dpid.to_hex }!”
  end
end
```

- `switch_ready` は、スイッチがコントローラに接続したときに呼ばれるハンドラです
- 引数の `dpid` には接続したスイッチの ID が格納されます
- `.to_hex` をつけることで `dpid` を 16 進数で表示します

# 演習: スイッチの追加

```
# hello-switch.conf
vswitch { dpid "0x1" }
vswitch { dpid "0x2" }
vswitch { dpid "0x3" }
...

$ trema run hello-switch.rb -c hello-switch.conf
???
```

- `hello-switch.conf` にスイッチを追加して `trema run` したとき、何が表示される?
- 注 : 各スイッチの dpid はユニークである必要があります

# 演習: スイッチの切断

- hello-switch.rb にコードを追加し、スイッチ切断情報を表示
  - switch_disconnected ハンドラを使用
  - 引数は、switch_ready と同様に、dpid のみ

```
$ trema kill 0x1          # 仮想スイッチの切断
$ trema up 0x1            # 仮想スイッチの起動
```

Task B : Hello Switch

# 接続をハンドリング

# ハンドラー覧

- start switch_ready switch_disconnected packet_in flow_removed port_status openflow_error features_reply stats_reply barrier_reply get_config_reply queue_get_config_reply vendor

# ハンドラー覧

- start switch_ready switch_disconnected packet_in flow_removed port_status openflow_error features_reply stats_reply barrier_reply get_config_reply queue_get_config_reply vendor

# イベントハンドラ

```
class HelloSwitch < Controller
  def switch_ready dpid
    puts "Hello #{ dpid.to_hex }!”
  end
end
```

- イベントドリブン形式で、コントローラを記述します
- 各ハンドラを、インスタンスメソッドとして実装します

# イベントの振り分け

```
class MyController < Controller

  def start          # コントローラ起動時に自動的に呼ばれる
    # ...
  end

  def switch_ready dpid  # スイッチ接続時に自動的に呼ばれる
    # ...
  end
end
```

- Trema はイベントの振り分けにリフレクションという Ruby の機構を使っています
- そのため、複雑になりがちなディスパッチやハンドラ登録を行う必要はありません

# イベントハンドラ (Floodlight の場合)

```
public Command receive(IOFSwitch sw, ...) {
  switch (msg.getType()) {                          // イベントディスパッチ
    case PACKET_IN:
      return this.handlePacketIn(sw, ...);
    ...
private Command handlePacketIn(IOFSwitch sw, ...) {  // ハンドラ本体
    ...
```

- Floodlight では、複雑なイベント振り分けが必要です
- おまじないが多いため、コードの見通しが悪くなります

# コーディングのための工夫 (coding by convention)

- 簡潔なコードを書くための工夫
  - e.g., "handler name" == "message name”
- イベントディスパッチのような、おまじないを不要に
- 楽しいプログラミングのために、お約束事やつまらない部分を削減

# 短く書く

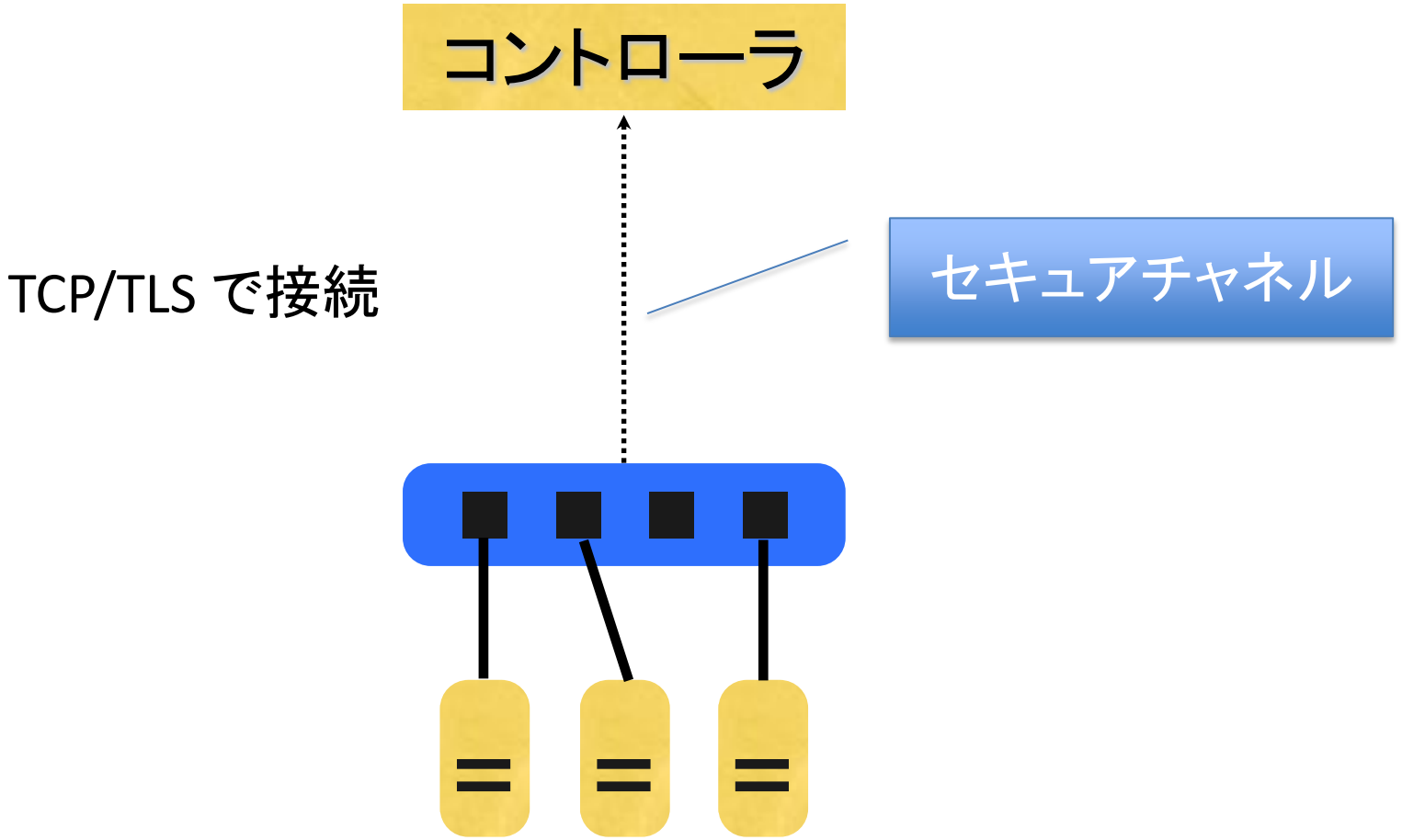
- コードの長さと生産性の間には強い相関関係
  - e.g. Arc Programming Language [Paul Graham]
- コードを短くすることで、
- お約束コードを書く時間を最小にする
- バグ混入の可能性を少なくする

Trema は、実行時の効率性よりも
プログラマーの生産性に重きをおいています

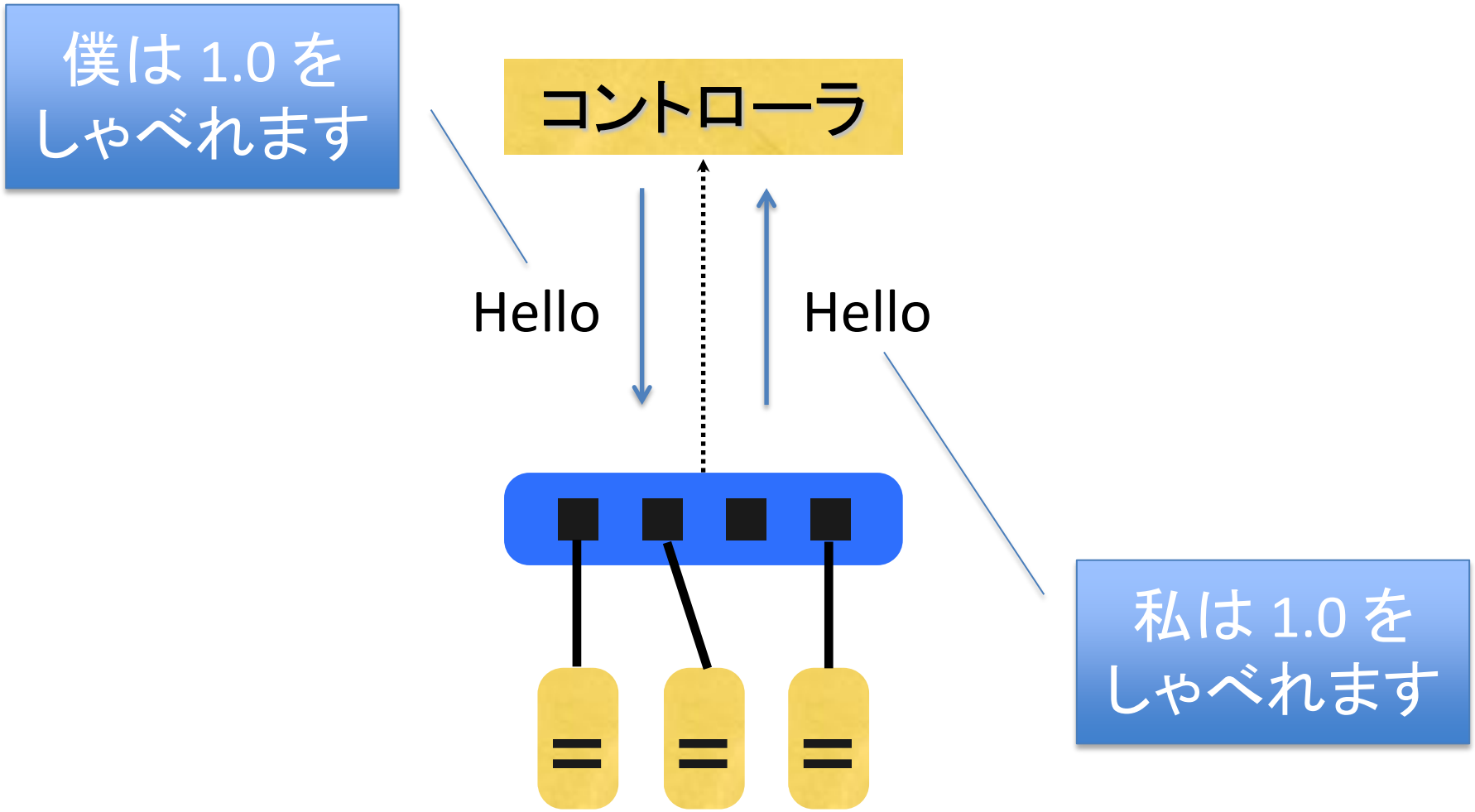
Task B : Hello Switch

# OpenFlow メッセージ(スイッチ接続)

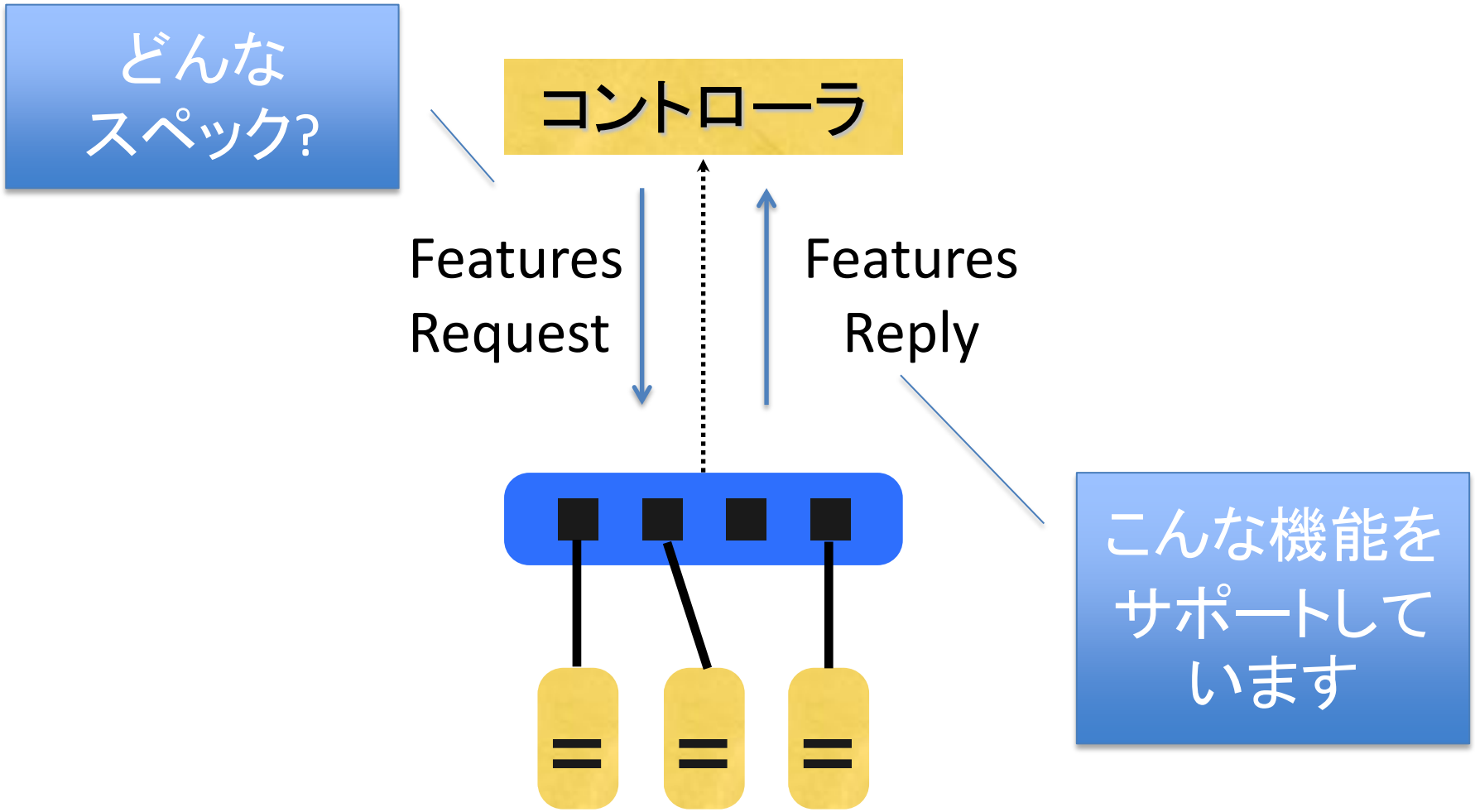
# スイッチ接続時の流れ



# Hello メッセージ



# Features Request/Reply



# 主な OpenFlow メッセージ

- パケット
  - Packet in : 受信パケットをコントローラへ送る
  - Packet out : コントローラ側で作ったパケットをスイッチから出力させる
- フロー関連
  - Flow mod : スイッチに、フローを設定する
  - Flow removed : フローが消えたことをスイッチからコントローラへ通知
- マネージメント
  - Hello : 接続時に用いられる
  - Features request/reply : スイッチのケイパビリティを取得
  - …

# ここまでのまとめ

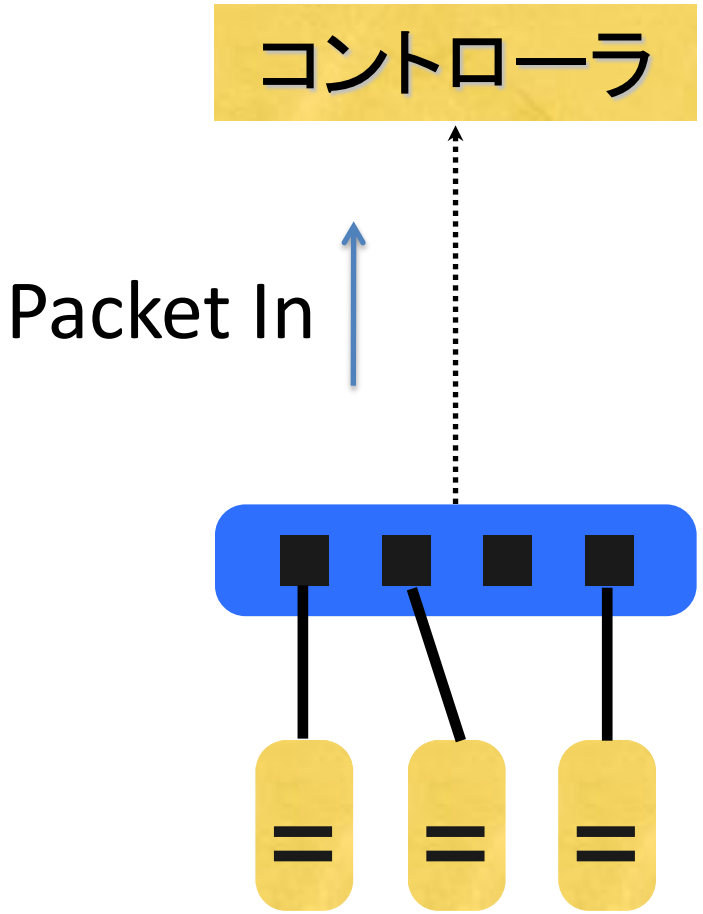
- Trema は “Post-Rails” なモダンなフレームワーク
  - 書いたコードをすぐ動かせる : `trema run`
  - Coding by Convention : 分かりやすく名付けられた各種メソッド
  - Full-Stack : ネットワーク DSL によるエミュレーション
  - 便利なサブコマンド : `trema ruby`
- スイッチ接続時に使われる OpenFlow メッセージ
  - Hello
  - Features Request/Reply

Packet-In メッセージの扱い

# Task C : Packet-In Dumper

# ここで学ぶこと

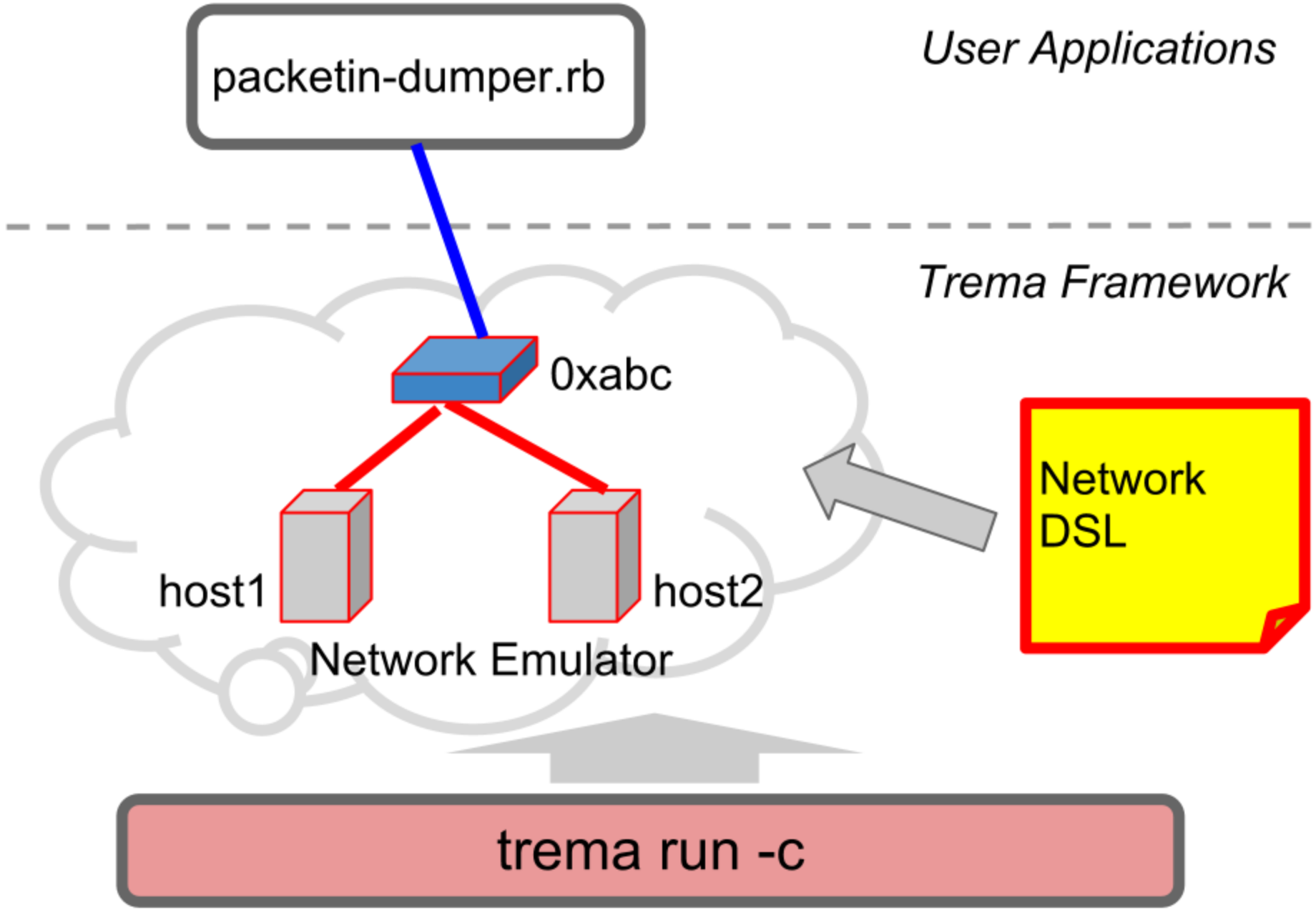
- Packet In を発生させる
- その内容を表示する



# 演習:Packet-In メッセージの内容表示

```
$ trema run packetin-dumper.rb -c packetin-dumper.conf
```

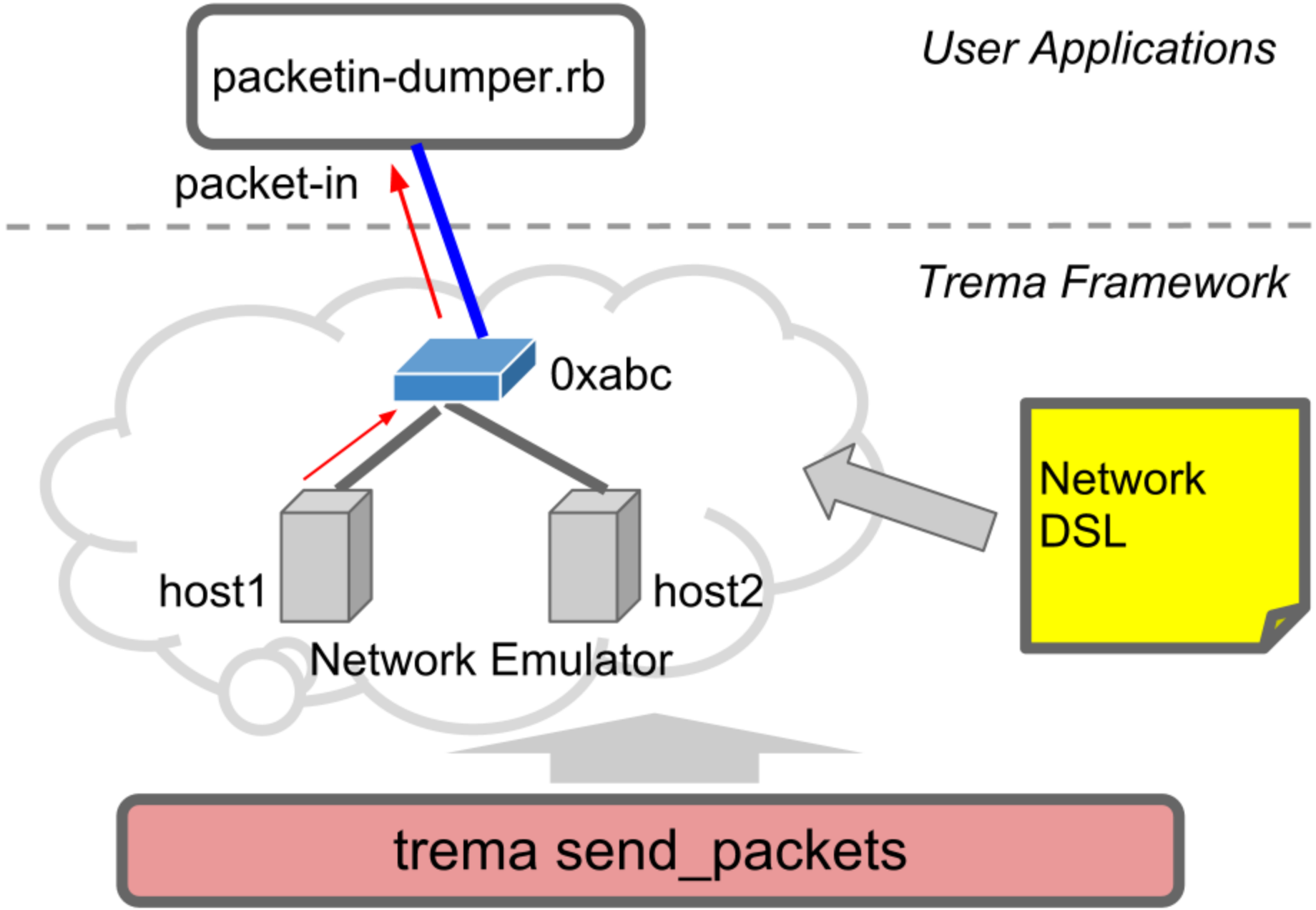
- Packet-In dumper コントローラを起動します
- 仮想ネットワークも同時に起動します
  - 仮想スイッチ 1 台
  - 仮想ホスト 2 台 (host1, host2)


packetin-dumper.rb
User Applications
Trema Framework
0xabc
Network
DSL
host1
host2
Network Emulator
trema run -c

# 演習:Packet-In メッセージの内容表示

```
$ trema send_packets --source host1 --dest host2
```

- 別のターミナルを開き、host1 から host2 へとパケットを送信します
- その結果、コントローラに送られた Packet In メッセージがダンプ表示されます


packetin-dumper.rb
User Applications
packet-in
Trema Framework
0xabc
Network
DSL
host1
host2
Network Emulator
trema send_packets

# 演習: Packet-In の各種アトリビュートを参照

```
# packetin-dumper.rb
class PacketinDumper < Controller
  def packet_in dpid, message
    puts "received a packet_in”
    puts "dpid: #{ datapath_id.to_hex }”
    puts "in_port: #{ message.in_port }”
    puts "total_len: #{ message.total_len }"
    ...
  end
end
```

- コードを修正し、他の Packet-In アトリビュートを表示してみる
  - total_len, macsa, macda ...
- ヒント: `trema ruby` を使い、Packet In クラス API を参照してみよう

Task C : Packet-In Dumper

# Q: テストパケットを出すにはどうすればよい？

# 仮想ホストと仮想リンク

- 仮想ホスト (host1, host2) を作り、仮想スイッチ 0xabc に接続

```
# Add one virtual switch
vswitch { dpid "0xabc" }
# Add two virtual hosts
vhost "host1”
vhost "host2”
# Then connect them to the switch 0xabc
link "0xabc", "host1”
link "0xabc", "host2"
```

- 一方の仮想ホストから他方へ、テストパケットを送る

```
$ trema send_packets --source host1 --dest host2
```

# ネットワークコンフィグレーションファイル

- シンプルな記述で、テスト環境を構築
- DSL を使って記述することで任意のネットワーク構成を実現
- シンプルなコマンドでテストパケットを送信

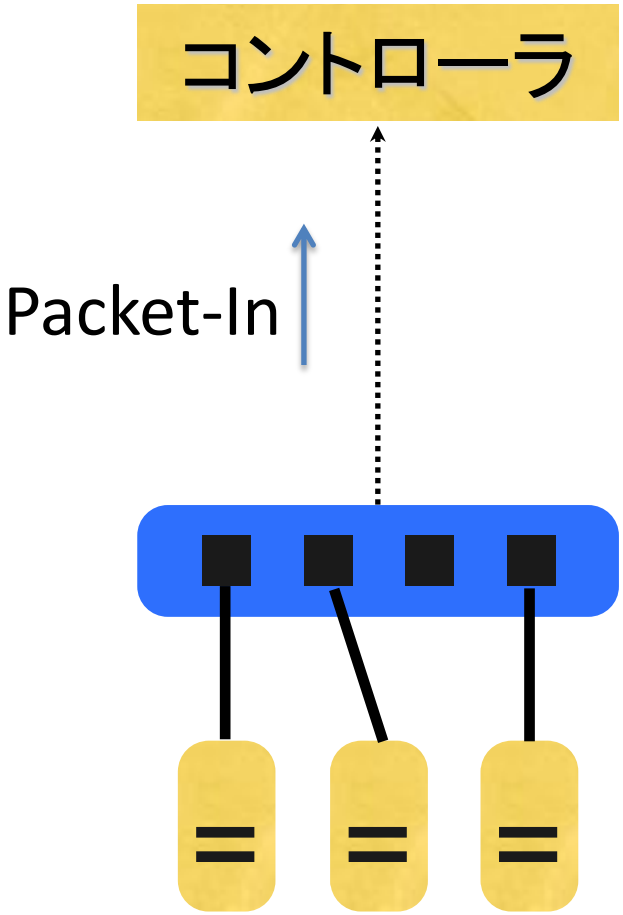
# 例: より複雑なネットワーク

```
vswitch { dpid "0x1" }
vswitch { dpid "0x2" }
...
vhost "host1”
vhost "host2”
vhost "host3”
vhost "host4"
 ...
link "0x1", "0x2"
...
link "0x1", "host1”
link "0x1", "host2”
link "0x2", "host3”
link "0x2", "host4”
...
```

Task C : Packet-In Dumper

# Packet-In をハンドリング

# Controller クラスの仕事




1. 自分が packet_in ハンドラを持っているか?
2. あれば呼ぶ
3. なければ呼ばない

# packet_in ハンドラ

```
# packetin-dumper.rb
class PacketinDumper < Controller
  def packet_in dpid, message
    puts "received a packet_in”
    puts "dpid: #{ datapath_id.to_hex }”
    puts "in_port: #{ message.in_port }”
  end
end
```

- 引数は以下の二つ
  - dpid : スイッチの Datapath ID
  - message : Packet-In メッセージオブジェクト

# Packet-In メッセージ

```
# packetin-dumper.rb
class PacketinDumper < Controller
  def packet_in dpid, message
    puts "received a packet_in”
    puts "dpid: #{ datapath_id.to_hex }”
    puts "in_port: #{ message.in_port }”
  end
end
```

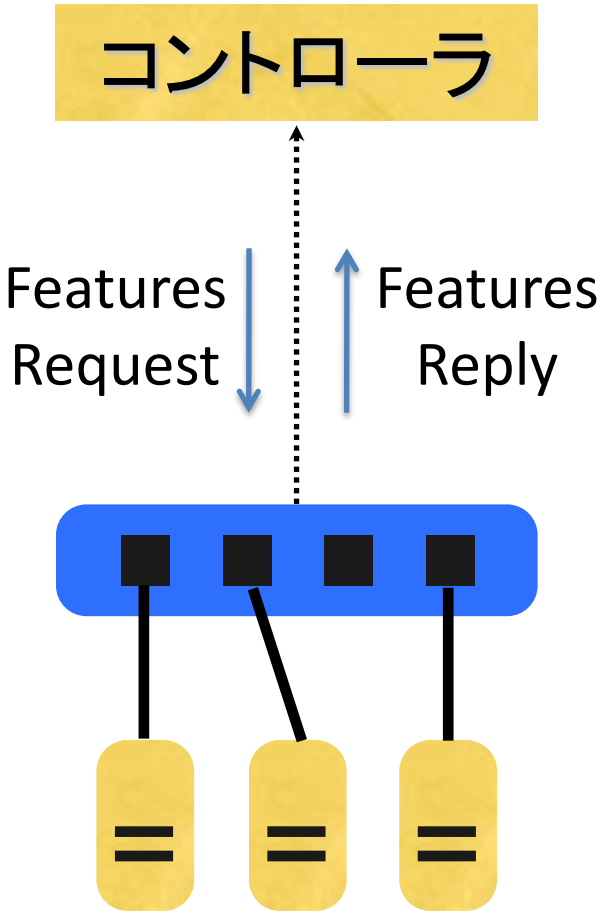
- Packet-In メッセージ中の値の参照が可能
  - message.in_port : 受信ポート
- Packet-In メッセージで送られてくるパケット中のヘッダフィールドのアクセスも可能
  - message.macsa : 送信元 MAC アドレス
  - message.ipv4_daddr : 宛先 IPv4 アドレス

スイッチから情報を取得する

# Task D : Show Features

# ここでのゴール

- OpenFlow プロトコルで、スイッチから情報を取得する



# 演習 : Features Reply メッセージの表示

```
$ trema run show-features.rb -c show-features.conf
```

- Show Features コントローラを起動します
- 仮想ネットワークも同時に起動します (= 1 台の仮想スイッチと 2 台の仮想ホスト host1, host2)

# 演習 : Description を表示

- スイッチの Description を取得し、表示するコントローラを作成する

- ヒント
  - まず Stats Request を送ります
    - メッセージは、DescStatsRequest クラスで作ります
  - ハンドラには stats_reply を使います
  - pp を使うことで、メッセージの内容を表示できます

Task D : Show Features

# スイッチからの情報取得

# Features request / reply

- スイッチの Features, Port info を取得
  - サポートしている統計情報
  - サポートしているアクション
  - ポート情報

# ポート情報

- ポート番号
- ハードウェアアドレス
- 名前
- Config
  - Administrative down?
  - Packet In させる?
- Status
  - リンクアップしているか？
- Features
  - リンク速度、オートネゴの有無等

# ハンドラー覧

- start switch_ready switch_disconnected packet_in flow_removed port_status openflow_error features_reply stats_reply barrier_reply get_config_reply queue_get_config_reply vendor

# show-features.rb

```
class ShowFeatures < Controller
  def switch_ready datapath_id
    send_message datapath_id, FeaturesRequest.new
  end

  def features_reply datapath_id, message
     puts "Datapath ID: #{ datapath_id.to_hex }"

    message.ports.each do | each |
      puts  "Port no: #{ each.number }"
      puts  " Hardware address: #{ each.hw_addr.to_s }"
      puts  " Port name: #{ each.name }"
    end
  end
end
```

# Stats request / reply

- スイッチの State の取得
  - Description
  - Flow statistics
    - Individual flow
    - Aggregate flow
    - Flow table
    - Port
    - Queue
  - Vendor extension

# ハンドラー覧

- start switch_ready switch_disconnected packet_in flow_removed port_status openflow_error features_reply stats_reply barrier_reply get_config_reply queue_get_config_reply vendor

# show-description.rb

```
class ShowDescription < Controller
  def switch_ready datapath_id
    send_message datapath_id, DescStatsRequest.new
  end

  def stats_reply datapath_id, message
    desc = message.stats.find do | each |
      each.is_a?( DescStatsReply )
    end

    if not desc.nil?
      puts "Manufacturer description: #{ desc.mfr_desc }"
      puts "Hardware description: #{ desc.hw_desc }"
      puts "Software description: #{ desc.sw_desc }"
      puts "Serial number: #{ desc.serial_num }"
      puts "Human readable description of datapath: #{ desc.dp_desc }"
    end
  end
end
```

# スイッチからの情報取得 – まとめ

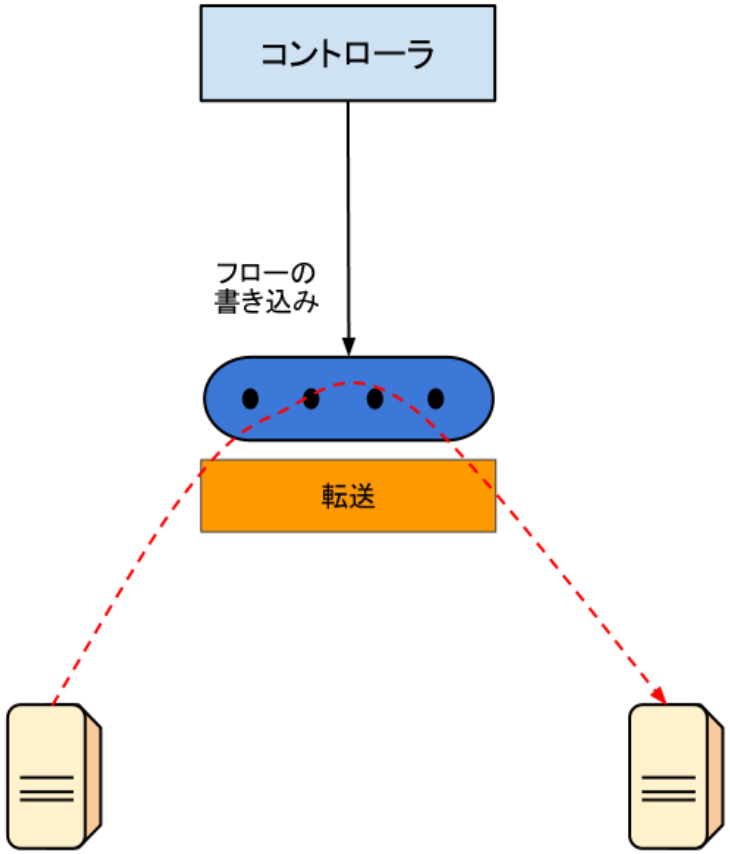
- スイッチから情報取得
  - Features Request / Reply
  - Stats Request / Reply

# OpenFlow ユースケース

# フロー = 水道管

- パケットを転送する
- 書き変える
- 流量を調べる
- 分岐する

# OpenFlow でスイッチを実現

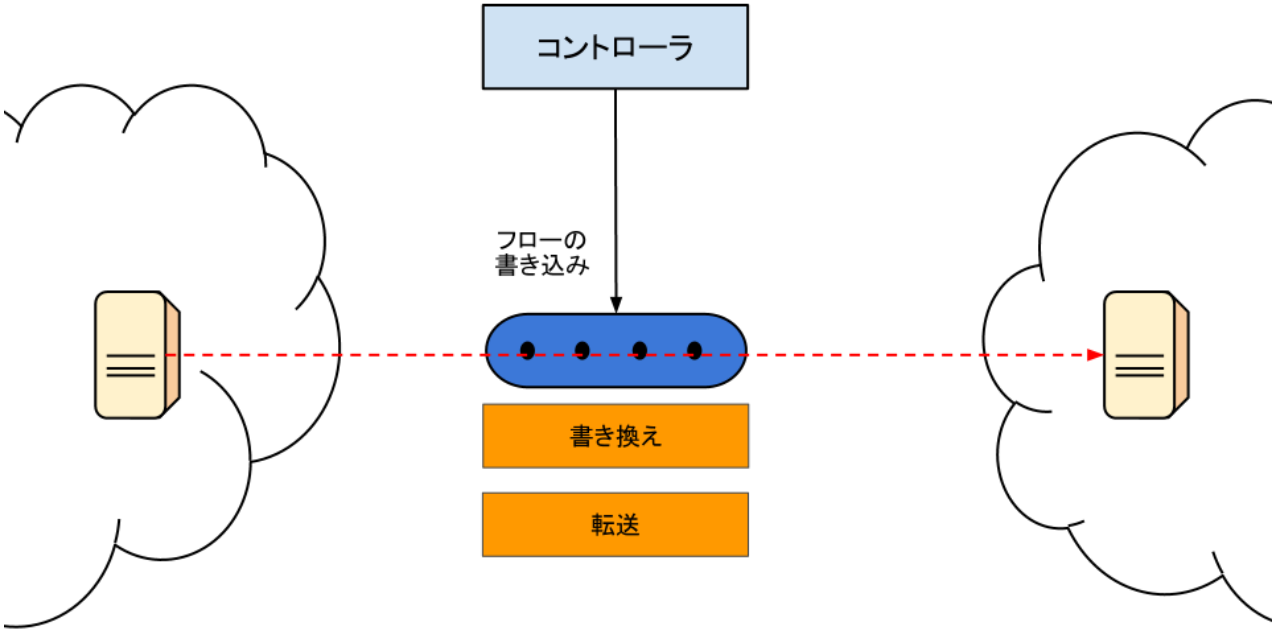



※ 演習で詳しく動作を見ていきます。

# OpenFlow で指定可能なアクション

- 指定のポートから出力する
  - 物理ポート
  - 論理ポート
    - All, Controller, Local, Table, IN_PORT (Required)
    - Normal, Flood (Required)
- キューにいれる
- 破棄する
- ヘッダ中のフィールドを書き換える

# OpenFlow でルータを実現

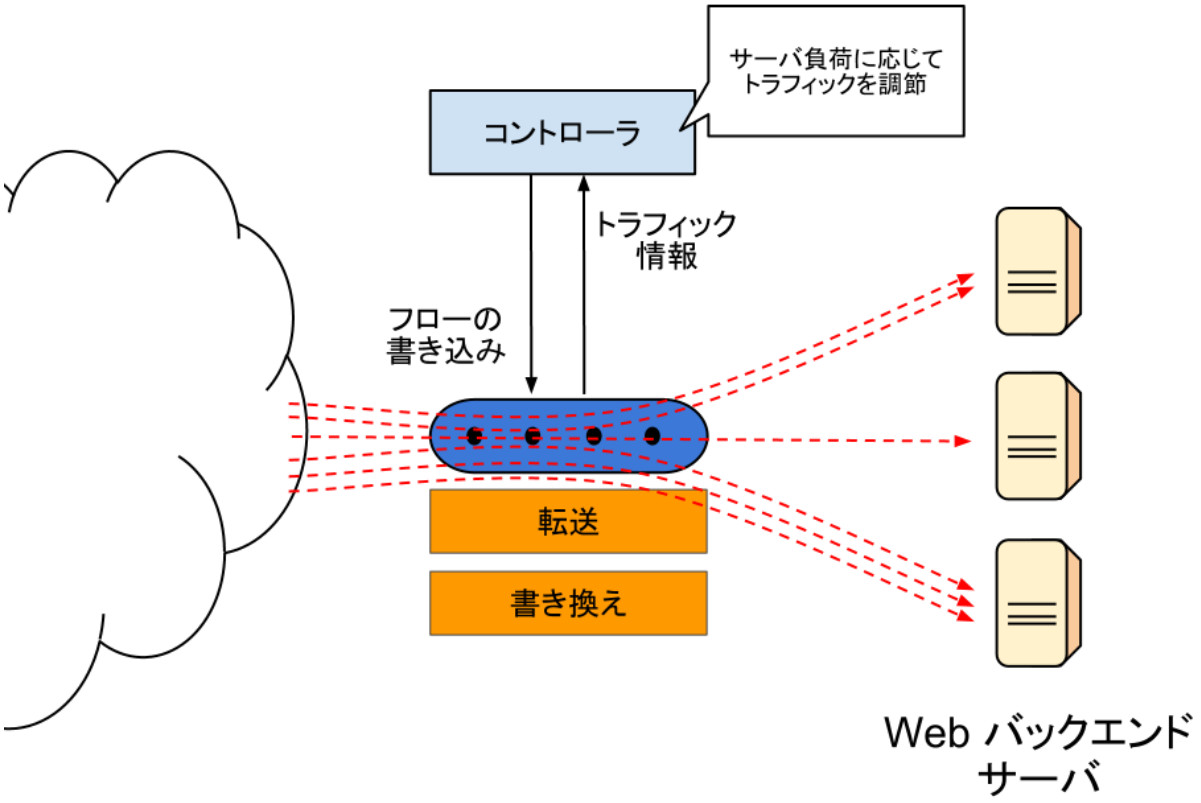



- ルータや NAT で必要となるヘッダ中の値の書換も可能
  - MAC アドレス書き換え (ルータ)
  - IP アドレス書き換え (NAT)
  - TCP ポート番号書き換え (NAT)

# 書き換えアクション

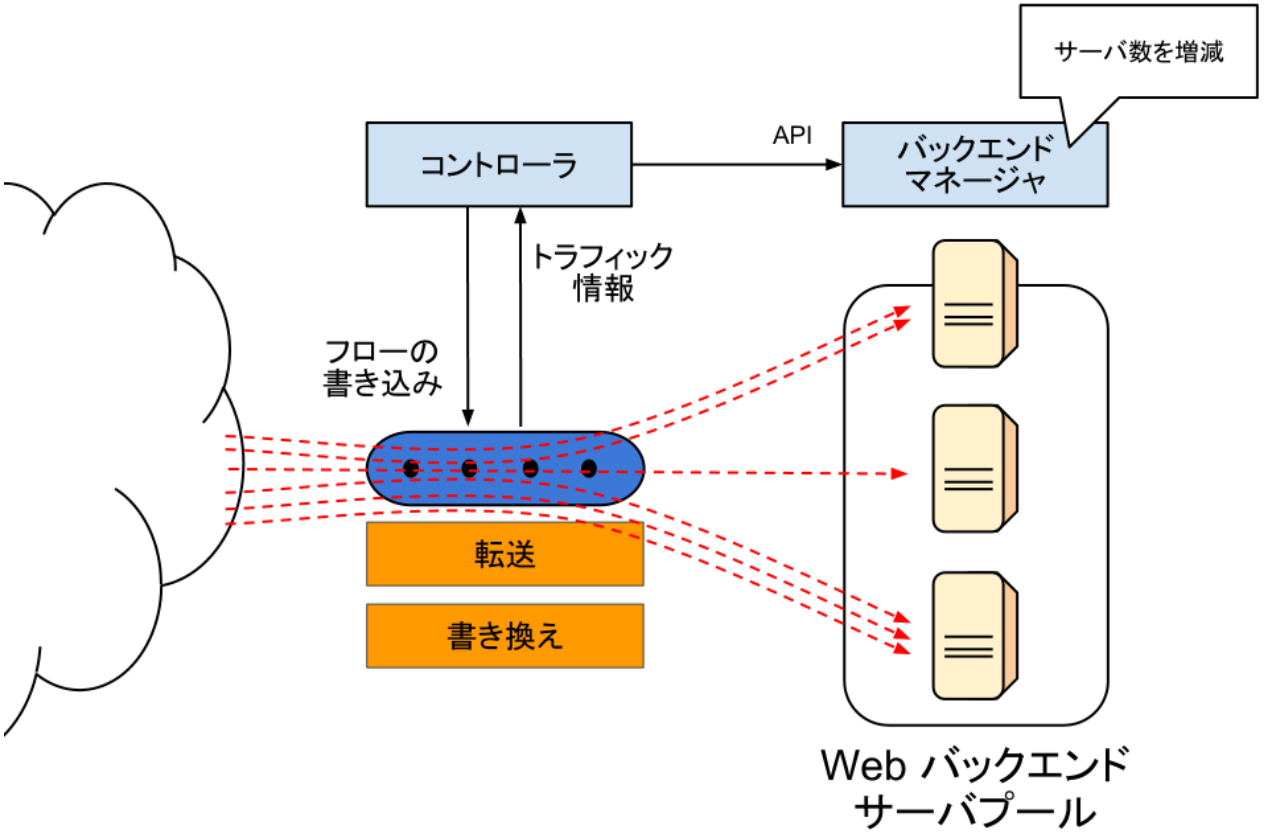
- Set/Add VLAN ID
- Set VLAN priority
- Strip VLAN Header
- Modify Ethernet source/destination address
- Modify IPv4 source/destionation address
- Modify IPv4 type of service bits
- Modify IPv4 TCP/UDP source/destination port

# OpenFlow でロードバランサを実現




- フロー単位での制御が可能
- 注 : 専用機の全機能を実現できるわけではない

# トラフィックに応じて バックエンドサーバの数を調節



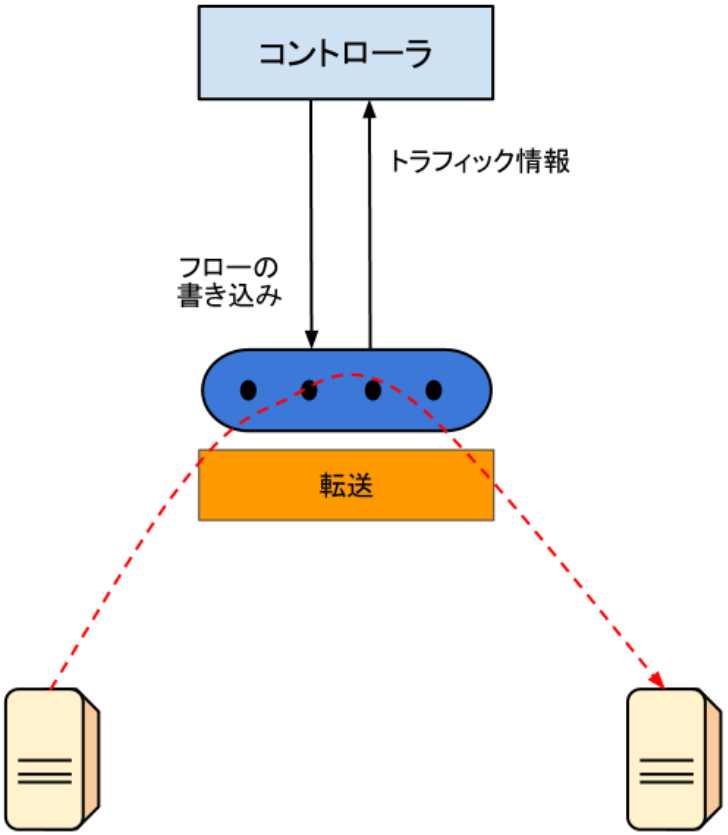

- 連携動作は、OpenFlow の方がやりやすい

# マッチ条件

- Ingress port
- Ethernet source/destination address
- Ethernet type
- VLAN ID
- VLAN priority
- IPv4 source/destination address
- IPv4 protocol number
- IPv4 type of service
- TCP/UDP source/destination port
- ICMP type/code

L1 から L4 まで 12 tuple のヘッダフィールドを利用可能

# トラフィック集計機能つきスイッチ

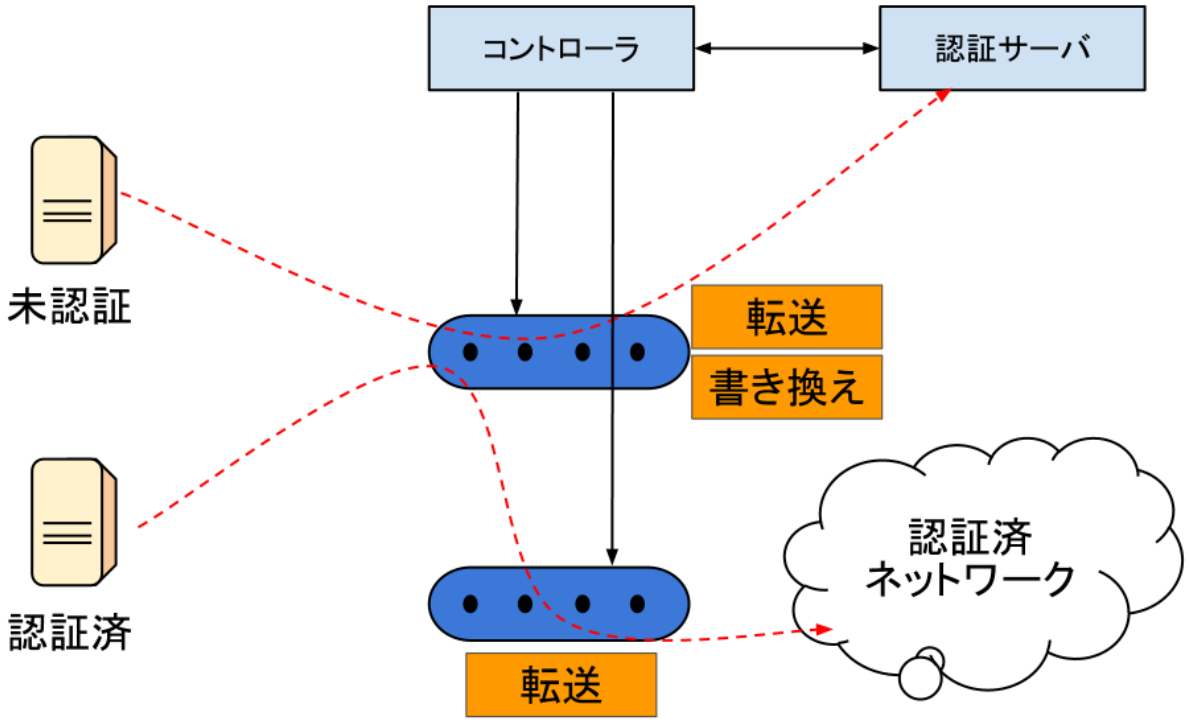



※ 後で詳しく動作を見ていきます。

# OpenFlow スイッチの持つカウンタ

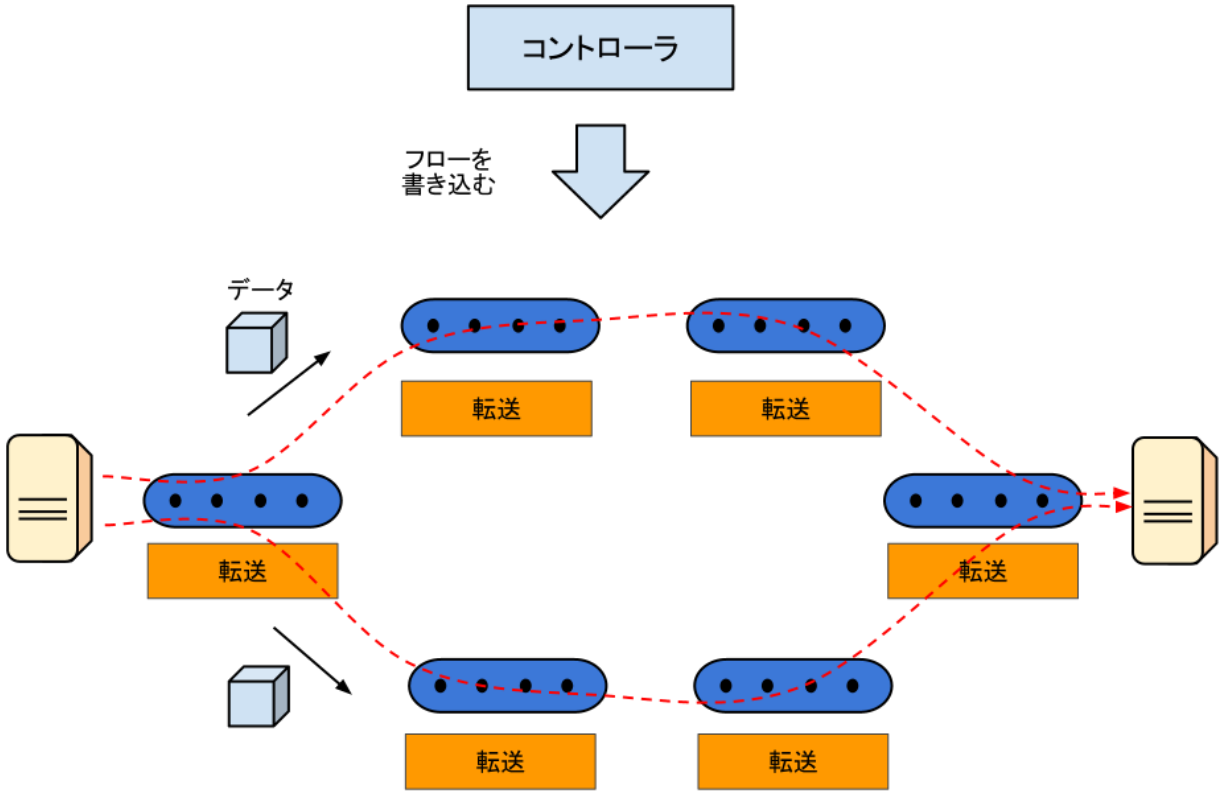
- テーブル全体のカウンタ
  - 有効なエントリ数
  - 参照パケット数
  - マッチしたパケット数
- フロー毎のカウンタ
  - パケット数/バイト
  - エントリの生存時間
- ポート毎のカウンタ
  - 送受信パケット数/バイト
  - 破棄パケット数
  - …
- キュー毎のカウンタ
  - 送信パケット数/バイト

# ホストを認識し認証するネットワーク



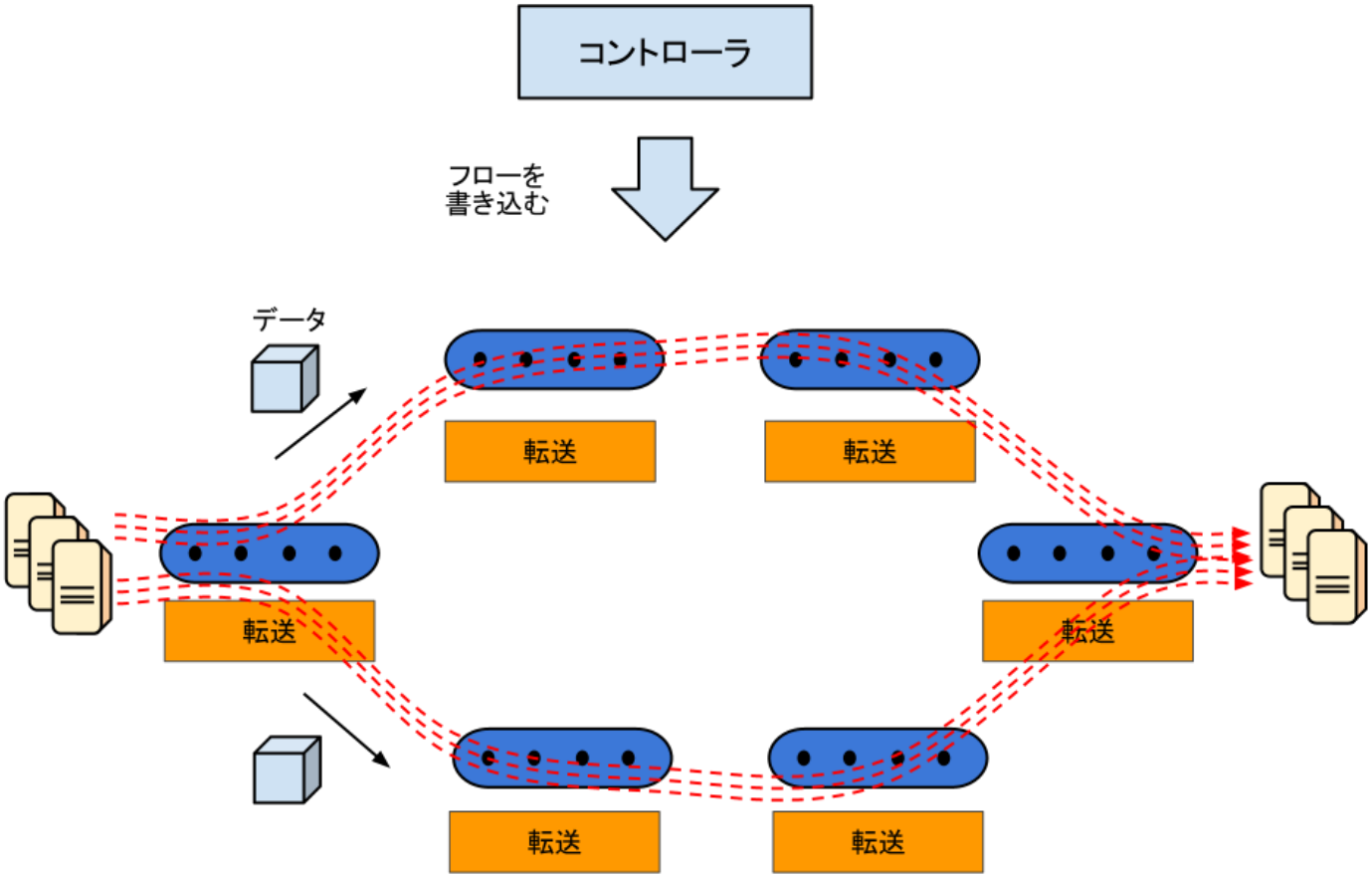

- チームー人旅 (オープンルータ・コンペティション出展)
- https://sites.google.com/site/orcmatome/list/tabi

# 複数経路を使って帯域をかせぐ

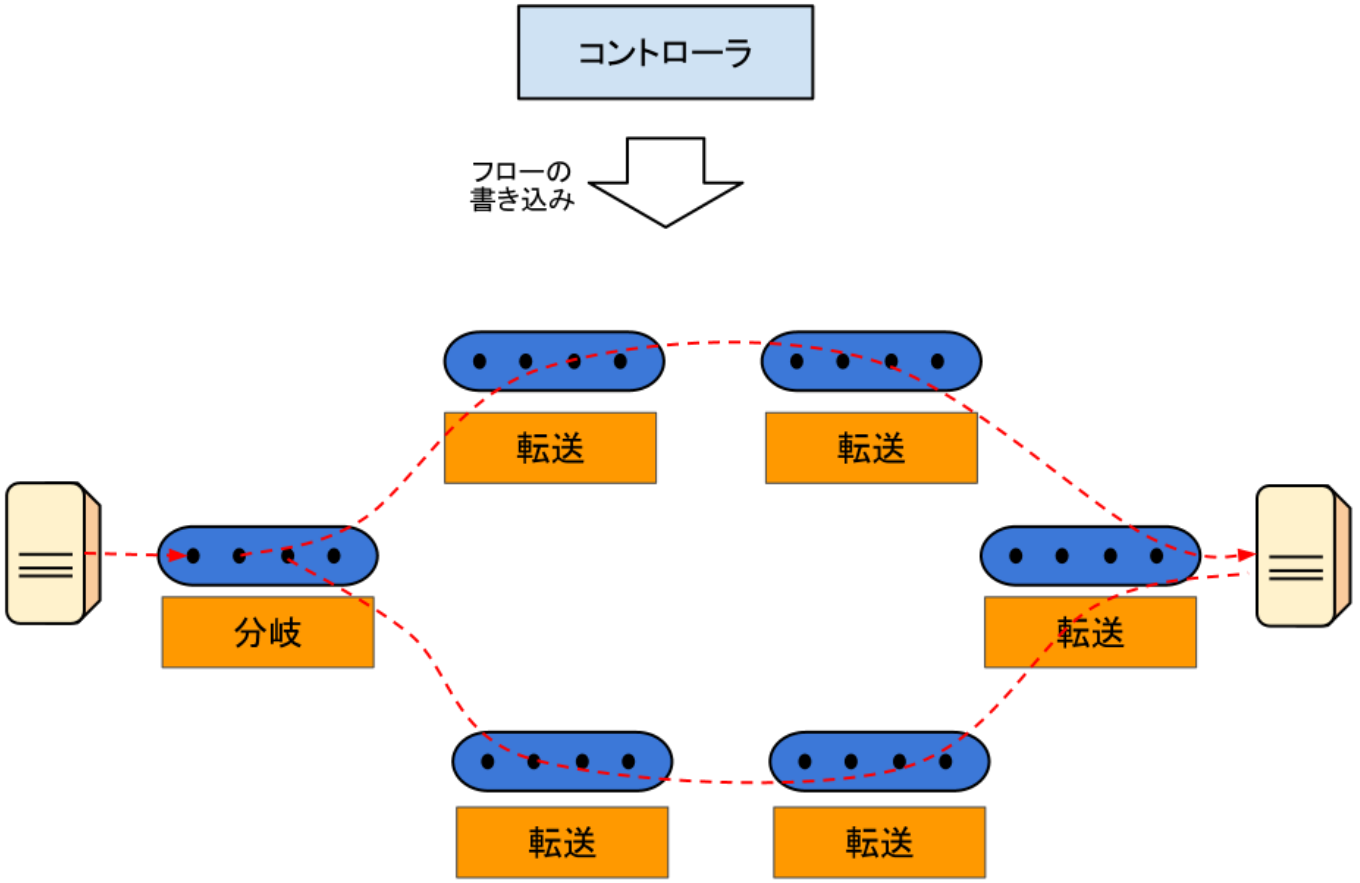



- Google データセンター間接続 (G-Scale) では、この目的のために、OpenFlow を使用している

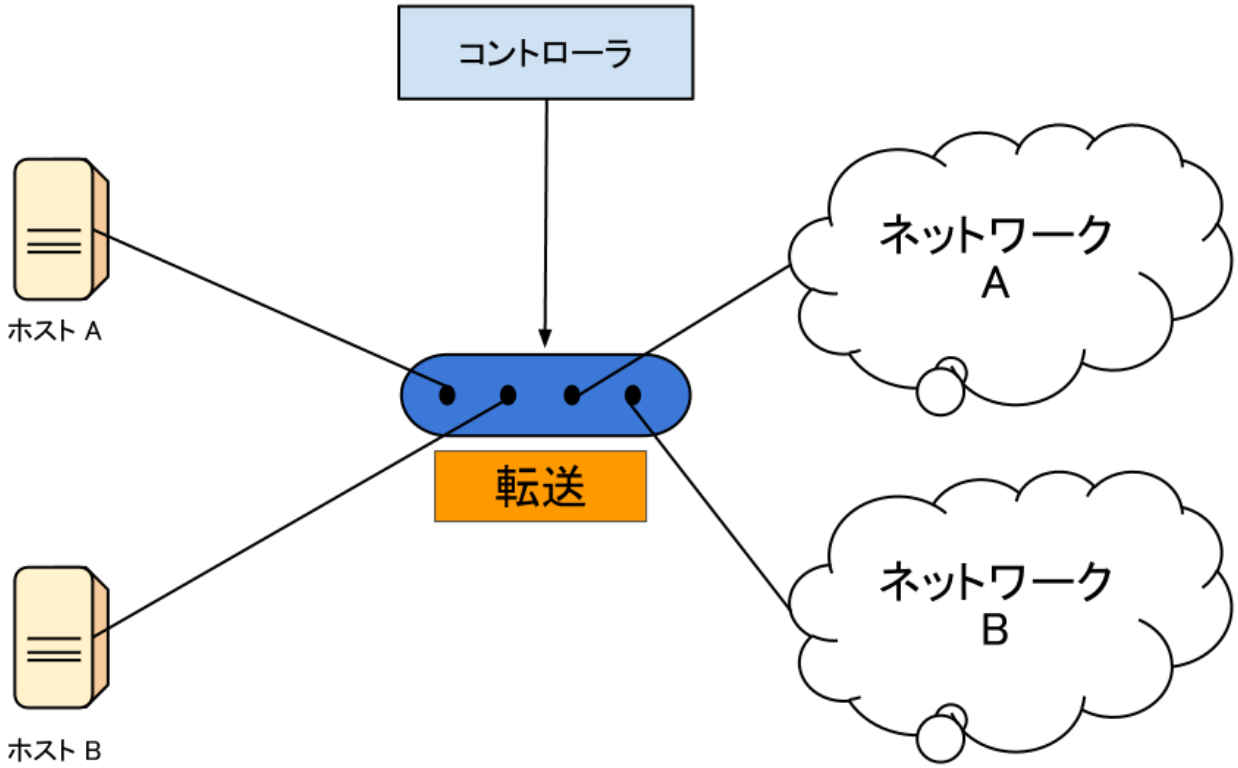
# アプリ毎に配信パスを分散



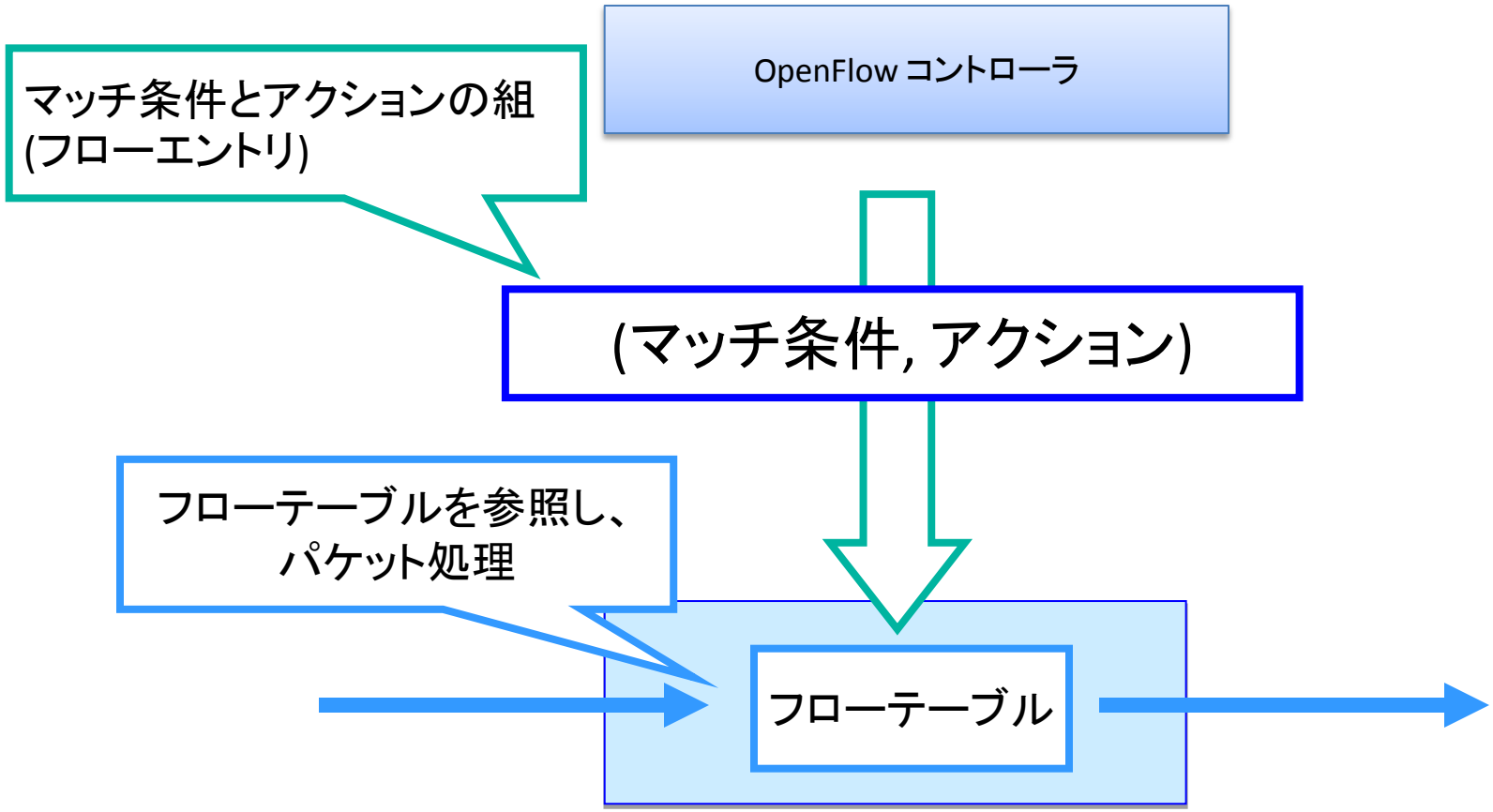
# OpenFlow で冗長な経路を作る



# 各ホストの所属する
# ネットワークを切り替える



# OpenFlow スイッチの動作



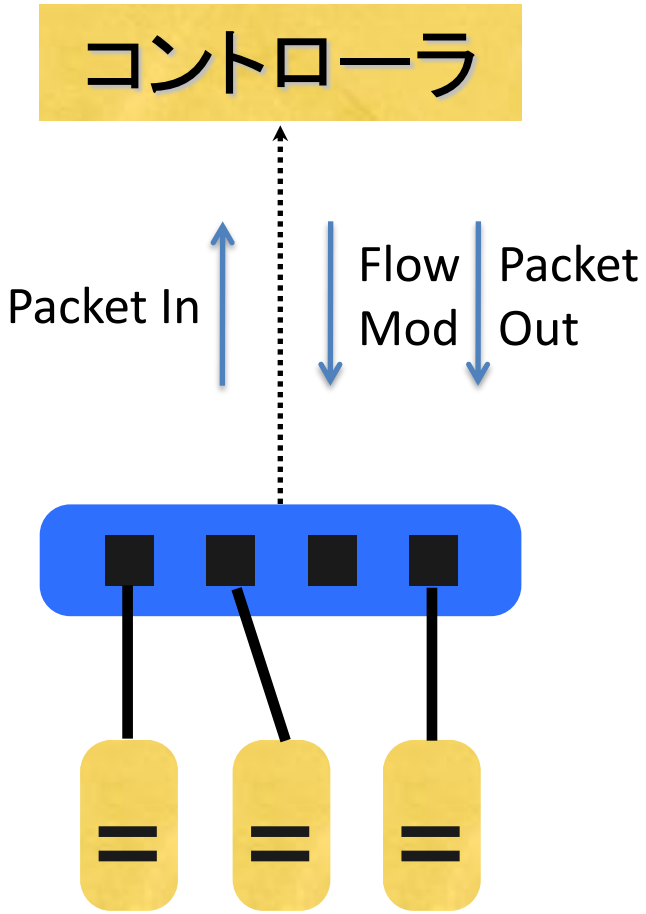

OpenFlow スイッチ

flow_mod と packet_out を送る

# Task E : Learning Switch

# ここで学ぶこと

- L2 スイッチの実現方法について学ぶ
  - Flow Mod
  - Packet Out
- Trema のサブコマンド
  - 内部状態
  - 統計情報



# 演習: 送受信パケット量を表示する

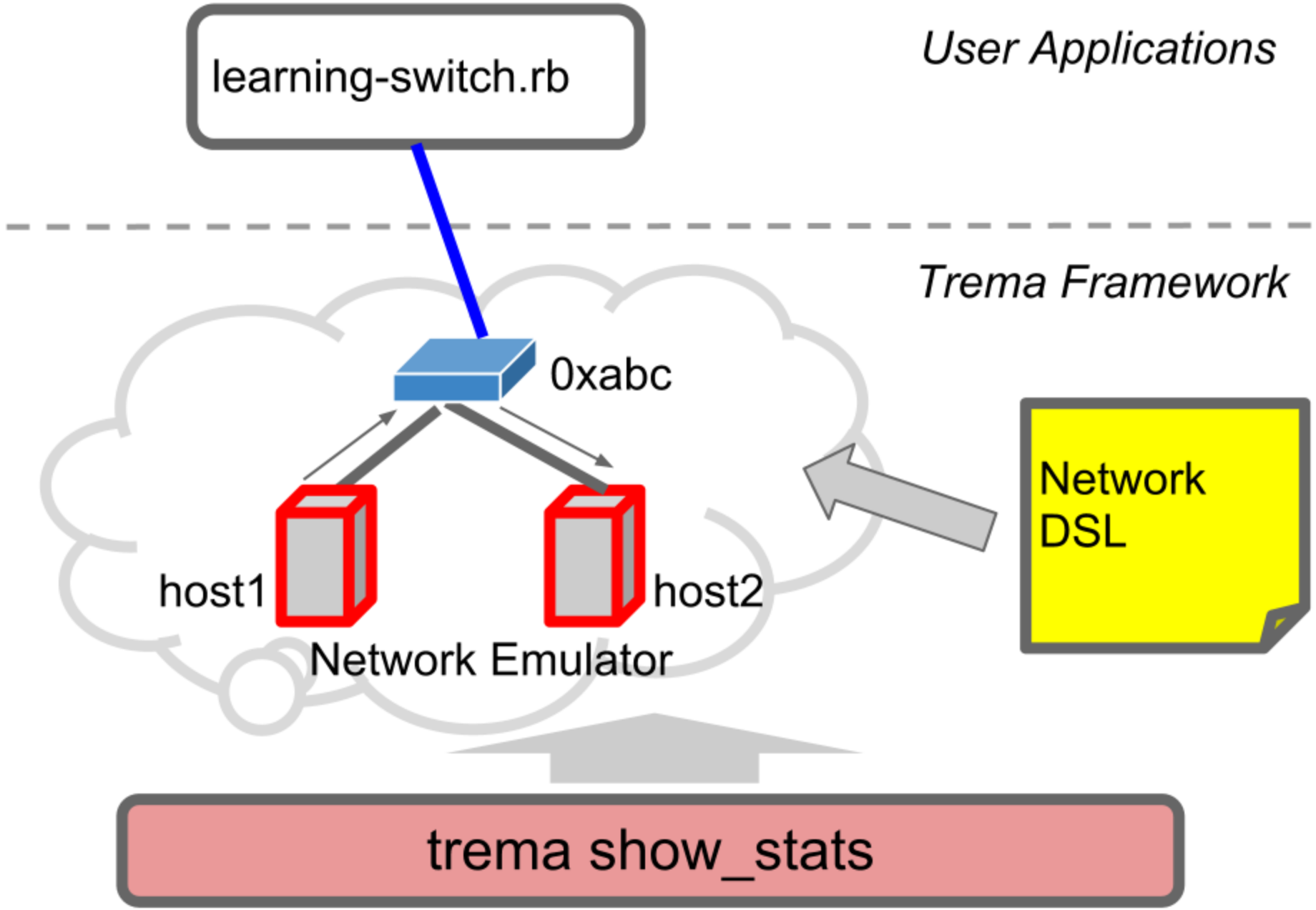
```
$ trema run learning-switch.rb -c learning-switch.conf
```

- L2 スイッチコントローラ (learning_switch) を起動する

# 演習: 送受信パケット量を表示する

```
$ trema send_packet --source host1 --dest host2
$ trema show_stats host1
$ trema show_stats host2
```

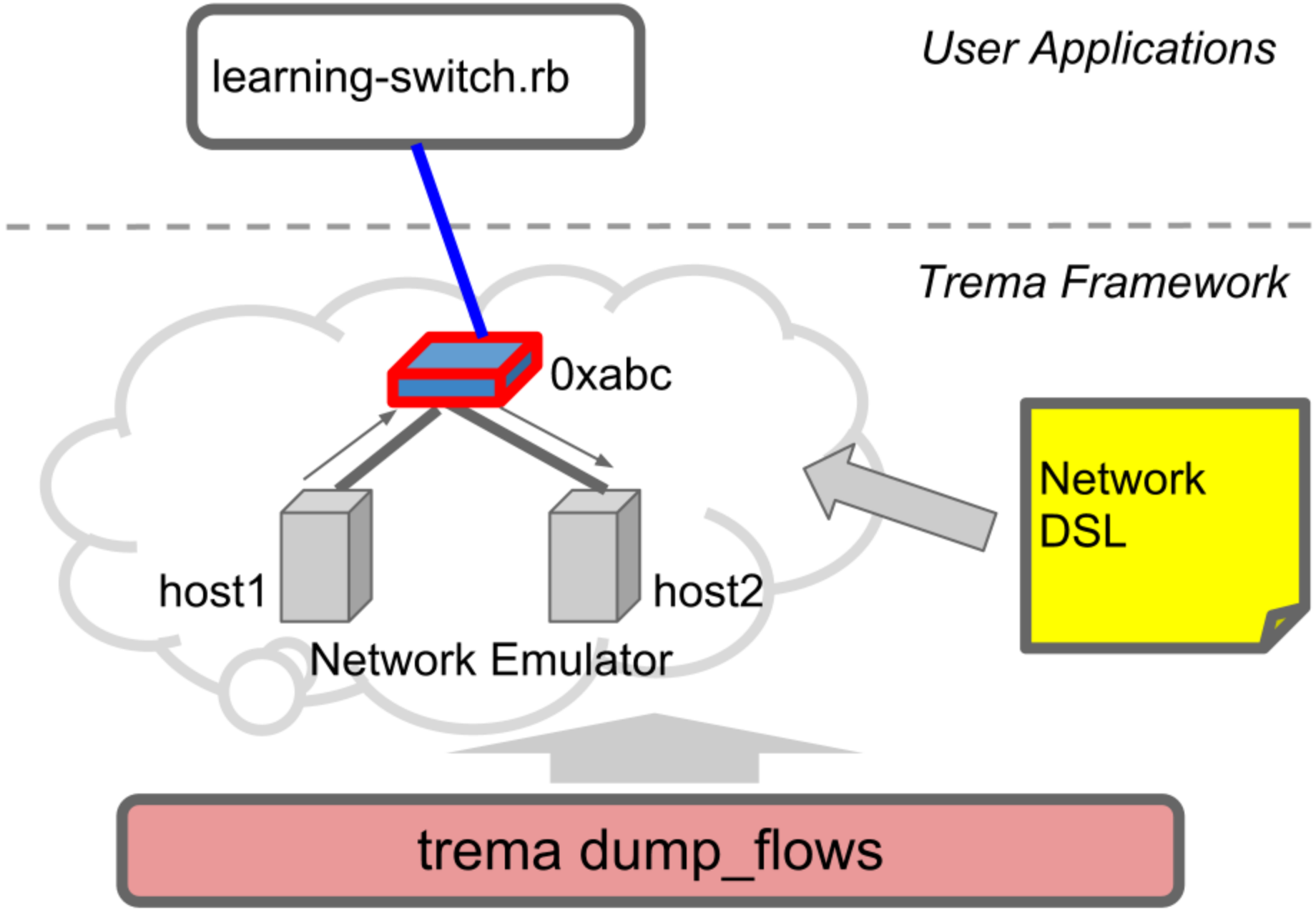
- 別のターミナルを開き、テストパケットを送る
- `show_stats` で送受信パケット量に関する情報を表示する


learning-switch.rb
User Applications
Trema Framework
0xabc
Network
DSL
host1
host2
Network Emulator
trema show_stats

# 演習: フローテーブルの表示

```
$ trema send_packet --source host2 --dest host1
$ trema dump_flows 0xabc
```

- 上記のコマンドで、スイッチ 0xabc のフローテーブルを表示する


learning-switch.rb
User Applications
Trema Framework
0xabc
Network
DSL
host1
host2
Network Emulator
trema dump_flows

# 今回使用したTremaのサブコマンド

`trema show_stats HOST_NAME`

`trema dump_flows SWITCH_NAME`

- 様々な統計情報と内部情報を表示

Task E : Learning Switch

# Learning Switch のソースコード

# Learning Switch

```
class LearningSwitch < Controller
# ...
  def packet_in dpid, message
    @fdb.learn message.macsa, message.in_port
    port_no = @fdb.lookup( message.macda )
    if port_no
      flow_mod dpid, message, port_no
      packet_out dpid, message, port_no
    else
      flood dpid, message
    end
  end
# ...
end
```

- 擬似コードのように簡単に読むことができるはず？

# 詳しく見ていこう

```
def packet_in dpid, message
  @fdb.learn message.macsa, message.in_port        # 1
  port_no = @fdb.lookup( message.macda )           # 2
  if port_no
    flow_mod dpid, message, port_no                # 3
    packet_out dpid, message, port_no
  else
    flood dpid, message                            # 4
  end
end
```

1. Packet In メッセージが送られてきた時に、送信元 MAC アドレス (macsa) と受信ポート (in_port) を Forwarding DB (FDB) に記録する
2. 宛先 MAC アドレス (macda) から送出ポートを検索する
3. もし見つかれば、スイッチのフローテーブルを更新し、パケットを Packet-Out する
4. 見つからなければ、パケットを flood する

# プライベートメソッド

- `flow_mod`, `packet_out`, `flood` は、learning_switch のプライベートメソッド
  - Trema API ではありません

- 適切なネーミングは、コードの可読性を高めます

# Syntactic Sugar: `ExactMatch.from()`

```
ExactMatch.from( message )
```

**vs**

```
Match.new(
  :in_port => message.in_port,
  :nw_src => message.nw_src,
  :nw_dst => message.nw_dst,
  :tp_src => message.tp_src,
  :tp_dst => message.tp_dst,
  :dl_src => message.dl_src,
  :dl_dst => message.dl_dst,
  ...
)
```

# Trema vs. NOX Python

```
# Trema
send_flow_mod_add(
  dpid,
  :match => ExactMatch.from( message ),
  :actions => ActionOutput.new( port_no )
)
```

vs

```
# NOX Python
inst.install_datapath_flow(
  dpid,
  extract_flow(packet),
  CACHE_TIMEOUT,
  openflow.OFP_FLOW_PERMANENT,
  [[openflow.OFPAT_OUTPUT, [0, prt[0]]]],
  bufid,
  openflow.OFP_DEFAULT_PRIORITY,
  inport,
  buf
)
```

# Learning Switch - まとめ

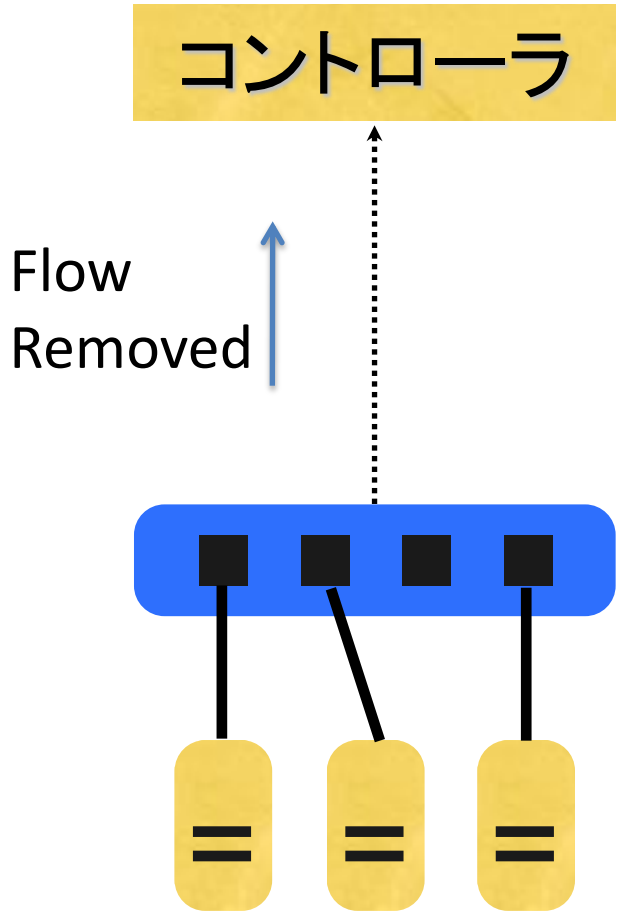
- 内部の状態表示
  - `trema show_stats`
  - `trema dump_flows`
- 短く書くための API
  - `ExactMatch.from`
  - `send_flow_mod_add`

flow_removed メッセージからトラフィックデータを取得

# Task F : Traffic Monitor

# ここで学ぶこと

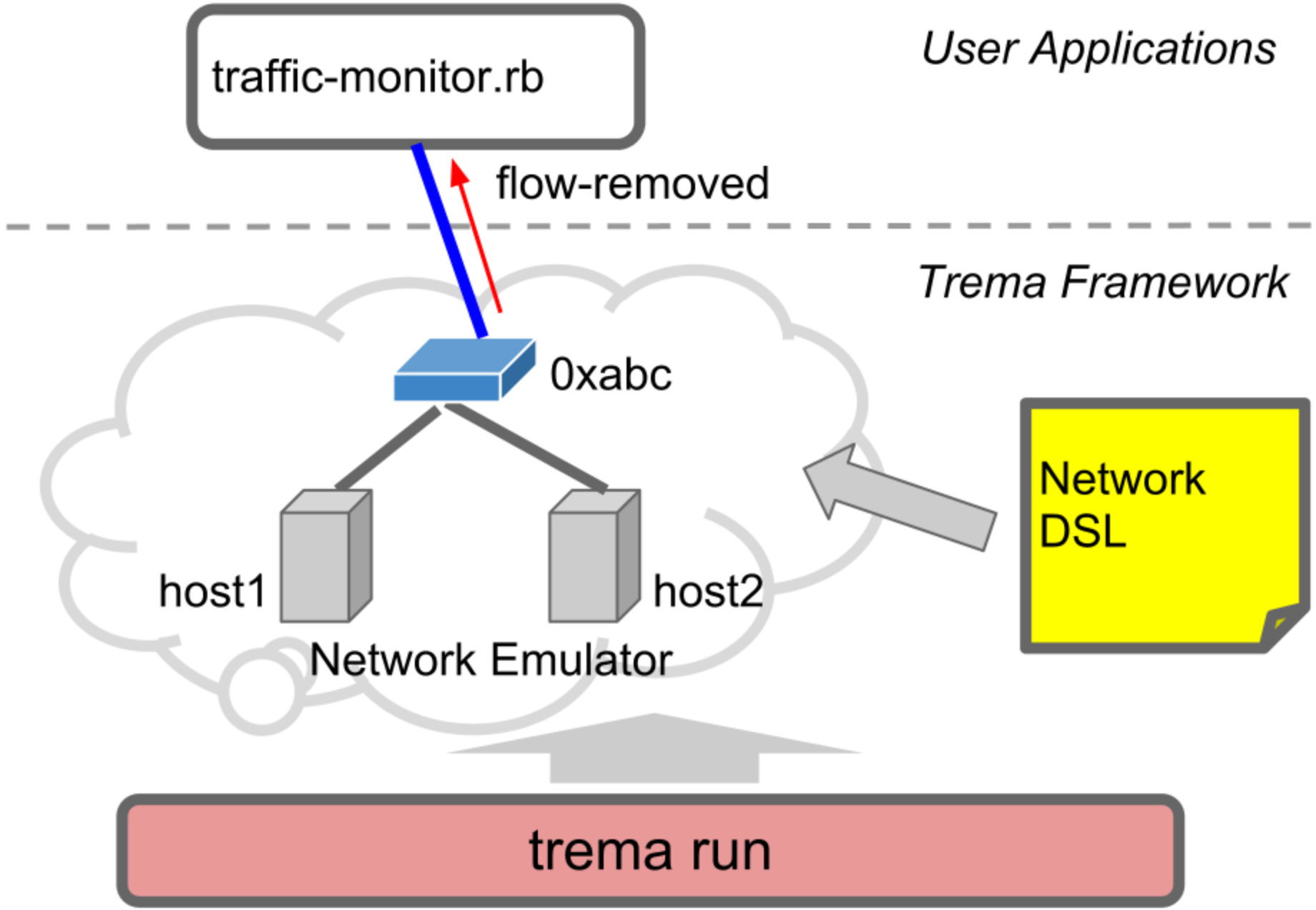
- トラフィック量の取得
  - Flow Removed の中身を表示する



# 演習:トラフィックデータを表示する

```
$ trema run traffic-monitor.rb -c traffic-monitor.conf

# (別のターミナルで、)
$ trema send_packets --source host1 --dest host2
$ trema send_packets --source host1 --dest host2
$ trema send_packets --source host2 --dest host1
```

- “トラフィックモニター付き L2 スイッチ” コントローラを起動
- テストパケットをランダムに送る
- コントローラは、各ホストのトラフィック情報を表示する


traffic-monitor.rb
User Applications
flow-removed
Trema Framework
0xabc
Network
DSL
host1
host2
Network Emulator
trema run

# トラフィック量を取得する

```
def flow_mod dpid, macsa, macda, out_port
    send_flow_mod_add(
      dpid,
      :hard_timeout => 10,  # 設定後、10 秒でタイムアウトさせる
      :match => Match.new( :dl_src => macsa, :dl_dst => macda ),
      :actions => ActionOutput.new( out_port )
    )
 end
```

- 各フローを 10 秒でタイムアウトさせる

# トラフィック量を取得する

```
class TrafficMonitor < Controller
# ...
  def flow_removed dpid, message
    @counter.add message.match.dl_src,
                 message.byte_count
  end
# ...
end
```

- フローがタイムアウトした時に送られる flow_removed メッセージをハンドリングする
- フローにより転送されたトラフィック量を記録する

# トラフィック量を表示する

```
class TrafficMonitor < Controller
  periodic_timer_event :show_counter, 10
  # ...
  private

  def show_counter
    puts Time.now
      @counter.each_pair do | mac, nbytes |
      puts "#{ mac } #{ nbytes } bytes”
    end
  end
  # ...
end
```

- 現在時刻と、`@counter` に記録されているトラフィック量を 10 秒ごとに表示

# Timer Attribute

```
class TrafficMonitor < Controller
  periodic_timer_event :show_counter, 10

  # ...

  def show_counter ...
  end
   # ...
end
```

- クラスアトリビュートのようにタイマーハンドラを定義
- スレッドを使った実装などを独自に行う必要がない
- coding by convention の一例

# Traffic Monitor まとめ

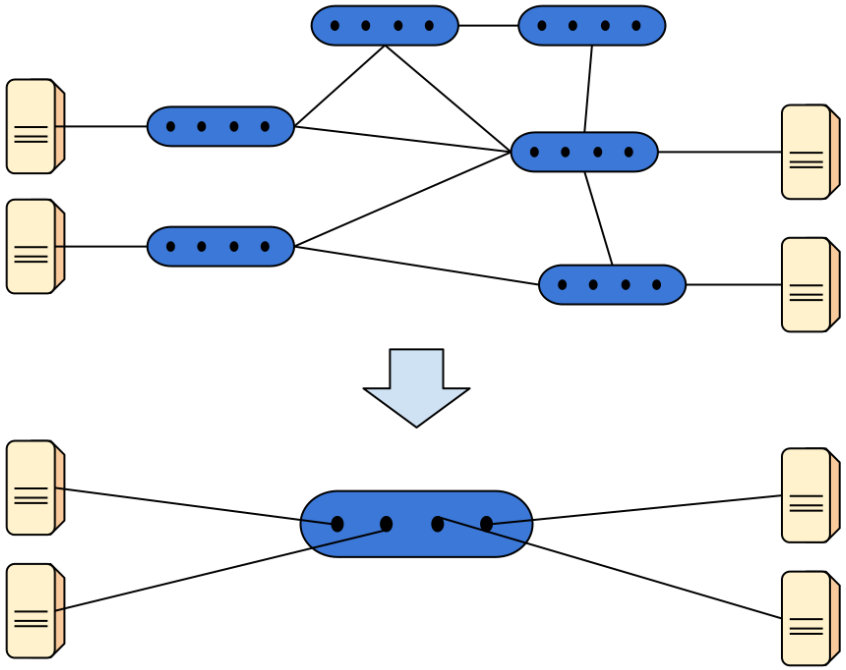
- flow_removed メッセージ中のトラフィックデータの取り扱い
- learning_switch よりも一歩進んだコントローラ

複数のスイッチを制御する

# Task G : Routing Switch

# ここで学ぶこと

- Routing Switch の動作について学習
  - 複数の OpenFlow スイッチを制御
  - L2 ネットワークを実現

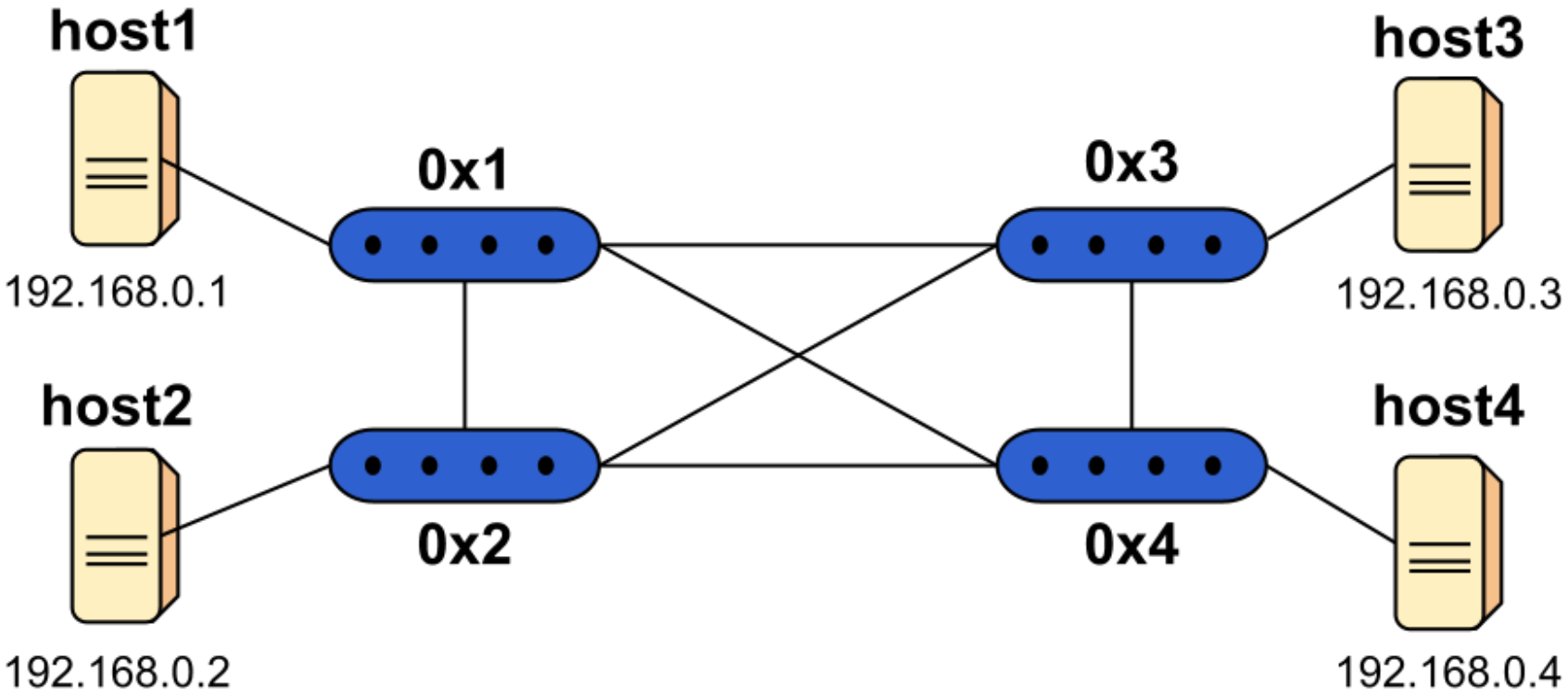


# 演習：Routing Switch の起動

```
$ trema run –c routing-switch.conf
```

- Routing Switch を起動する

# routing-switch.conf



# 演習: フローテーブルの表示

```
$ trema send_packet --source host1 --dest host4
$ trema send_packet --source host4 --dest host1
$ trema dump_flows 0x1
...
$ trema dump_flows 0x2
...
```

- 別のターミナルで以下を行う
  - host1 から host4 へ、さらに逆方向にパケットを送る
  - その後、各スイッチ (0x1 〜 0x4) のフローエントリを確認する
  - 最短パス上のスイッチにフローエントリができている？

# 演習：トポロジーを表示する

```
$ ./show_topology
...
```

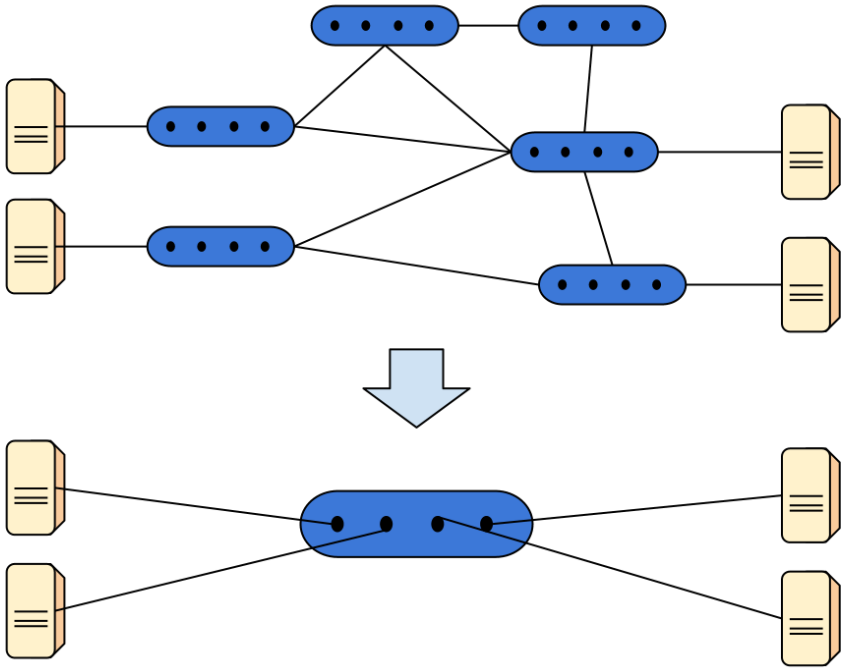
- 上記コマンドを実行する
  - OpenFlow コントローラが認識しているトポロジーを表示する

Task G : Routing Switch

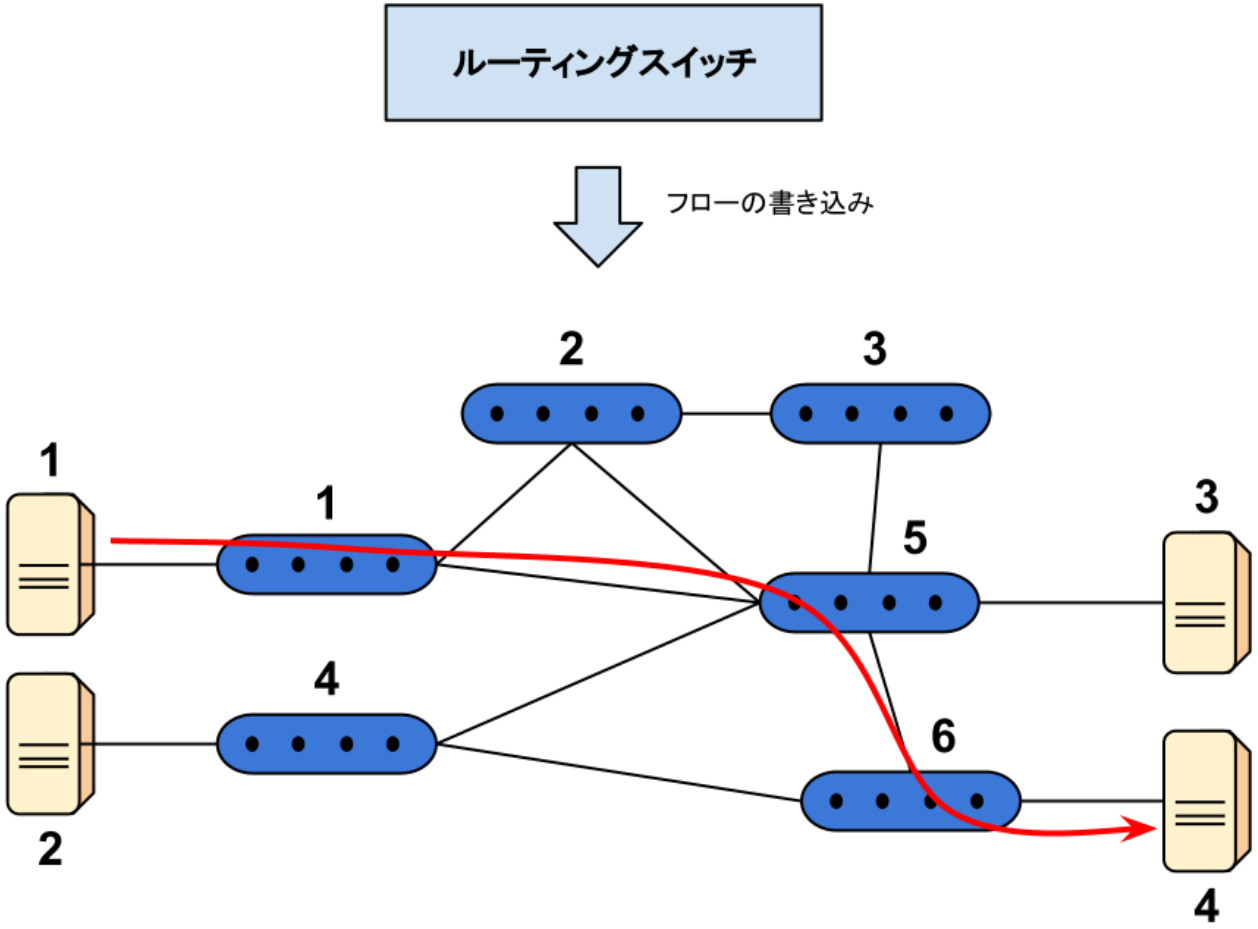
# 複数のスイッチを制御する

# Routing Switch

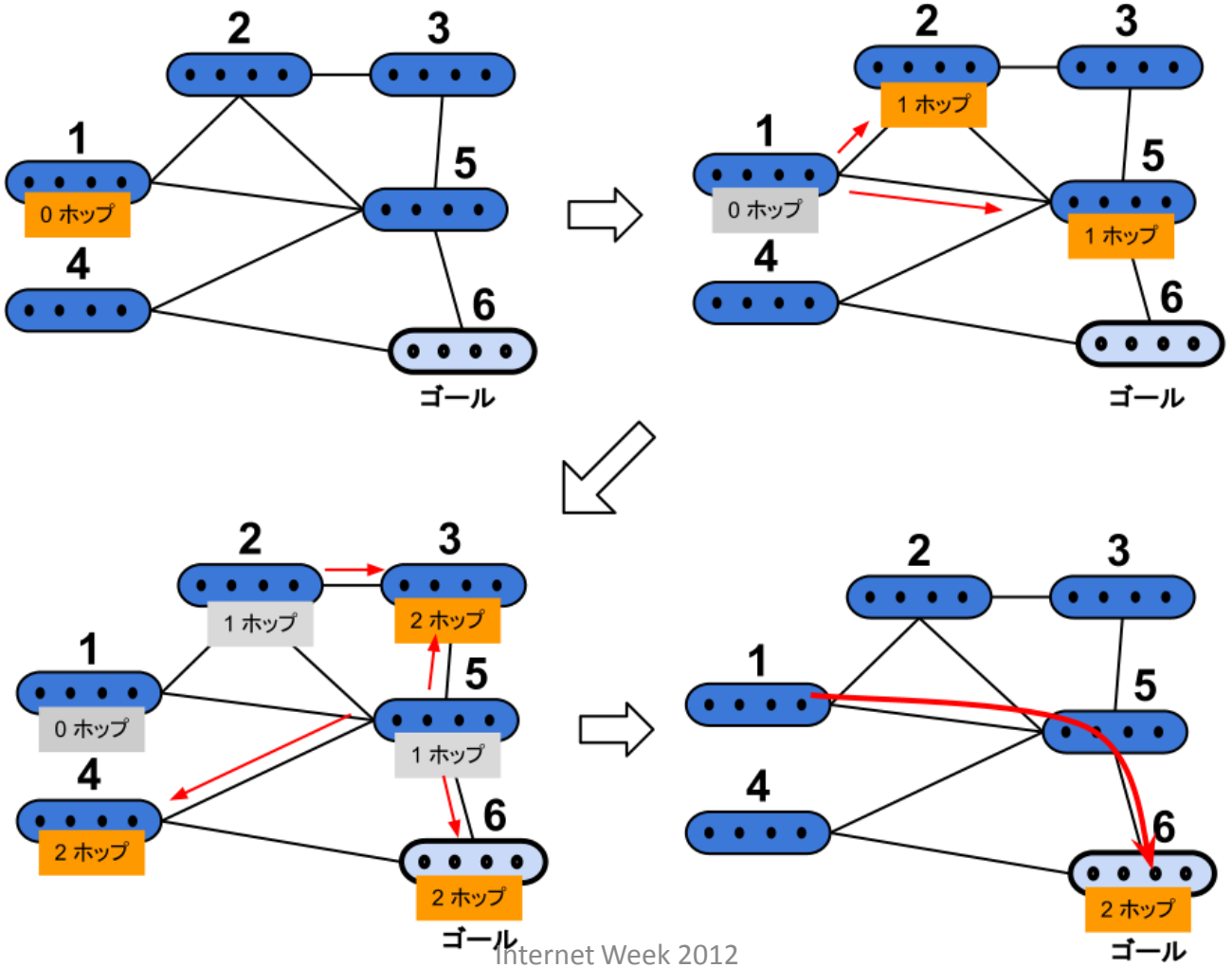
- L2 ネットワークを実現する OpenFlow コントローラの一種
  - 複数の OpenFlow スイッチを制御



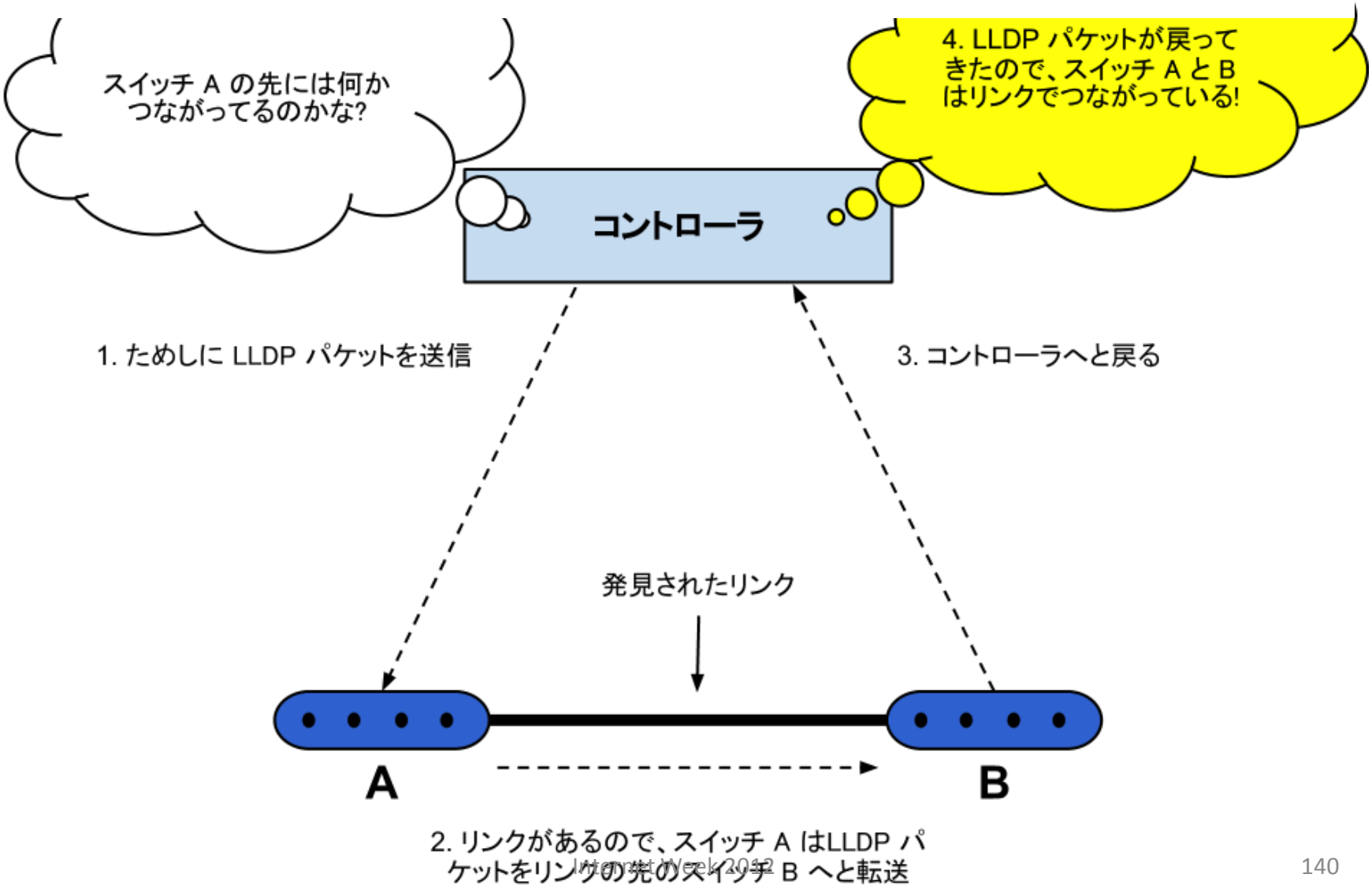
# 最短パスでのフローエントリの設定



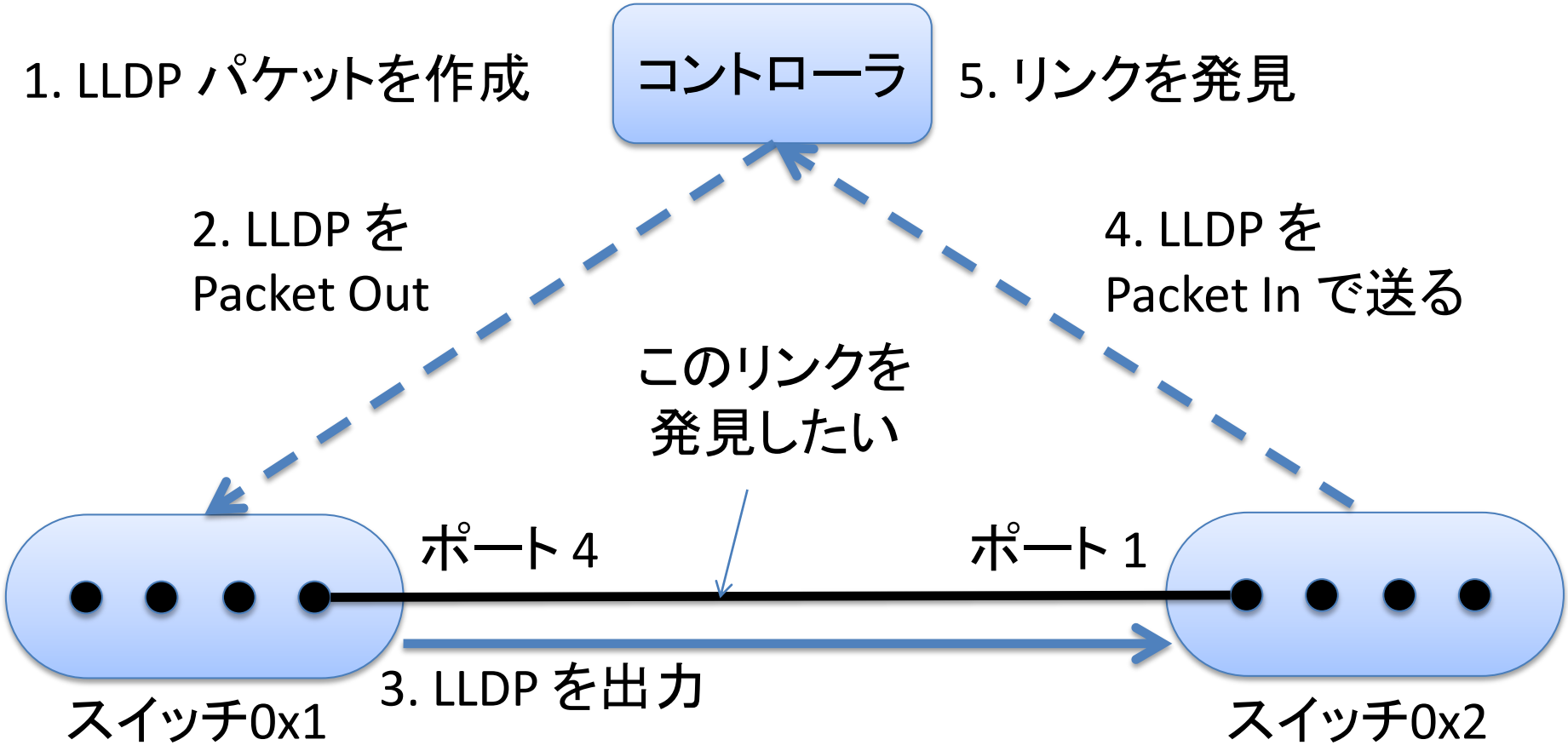
# ダイクストラ法で最短パスの計算



# LLDP を使ったリンクの検出



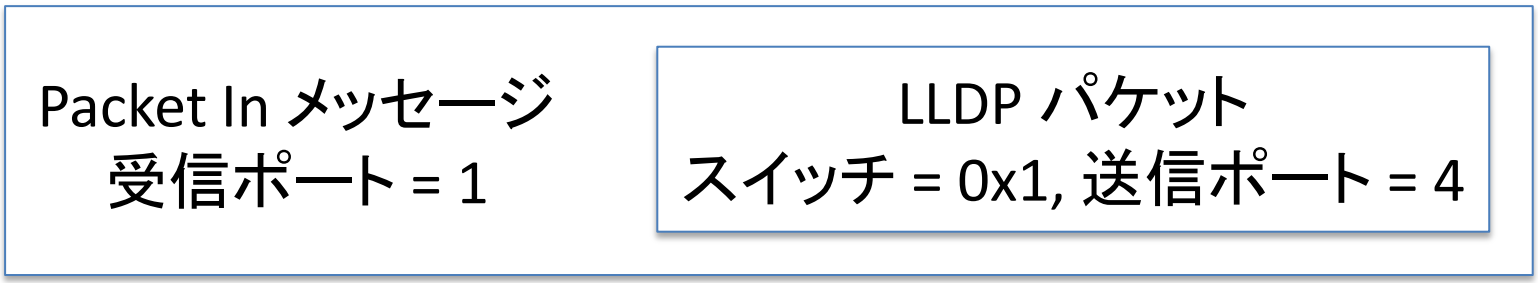
# OpenFlow を使ったリンク検出



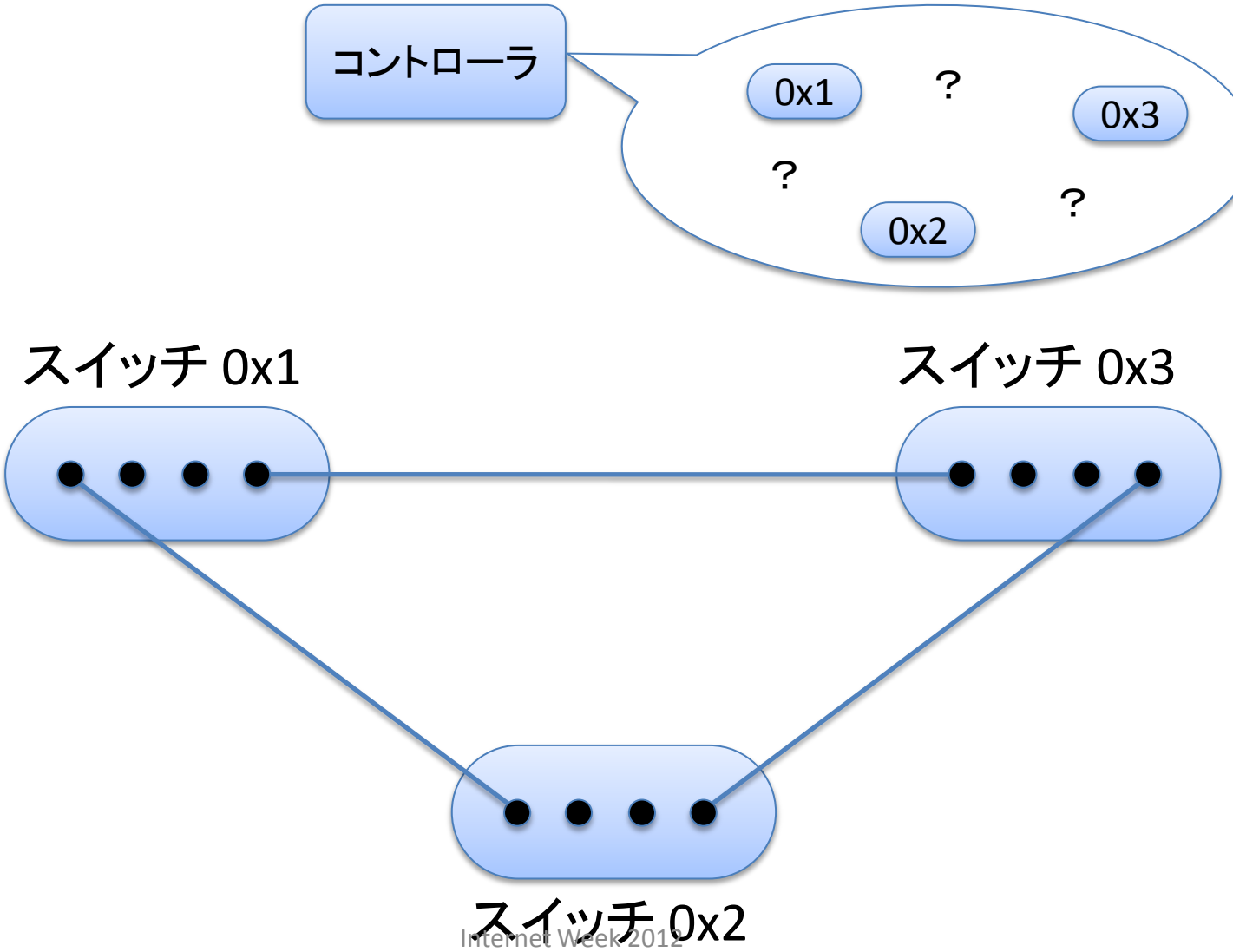
# LLDP パケットの解析

- LLDP パケット
  - Packet Out するスイッチの ID とポート番号
- Packet In メッセージ
  - LLDP の受信ポート番号

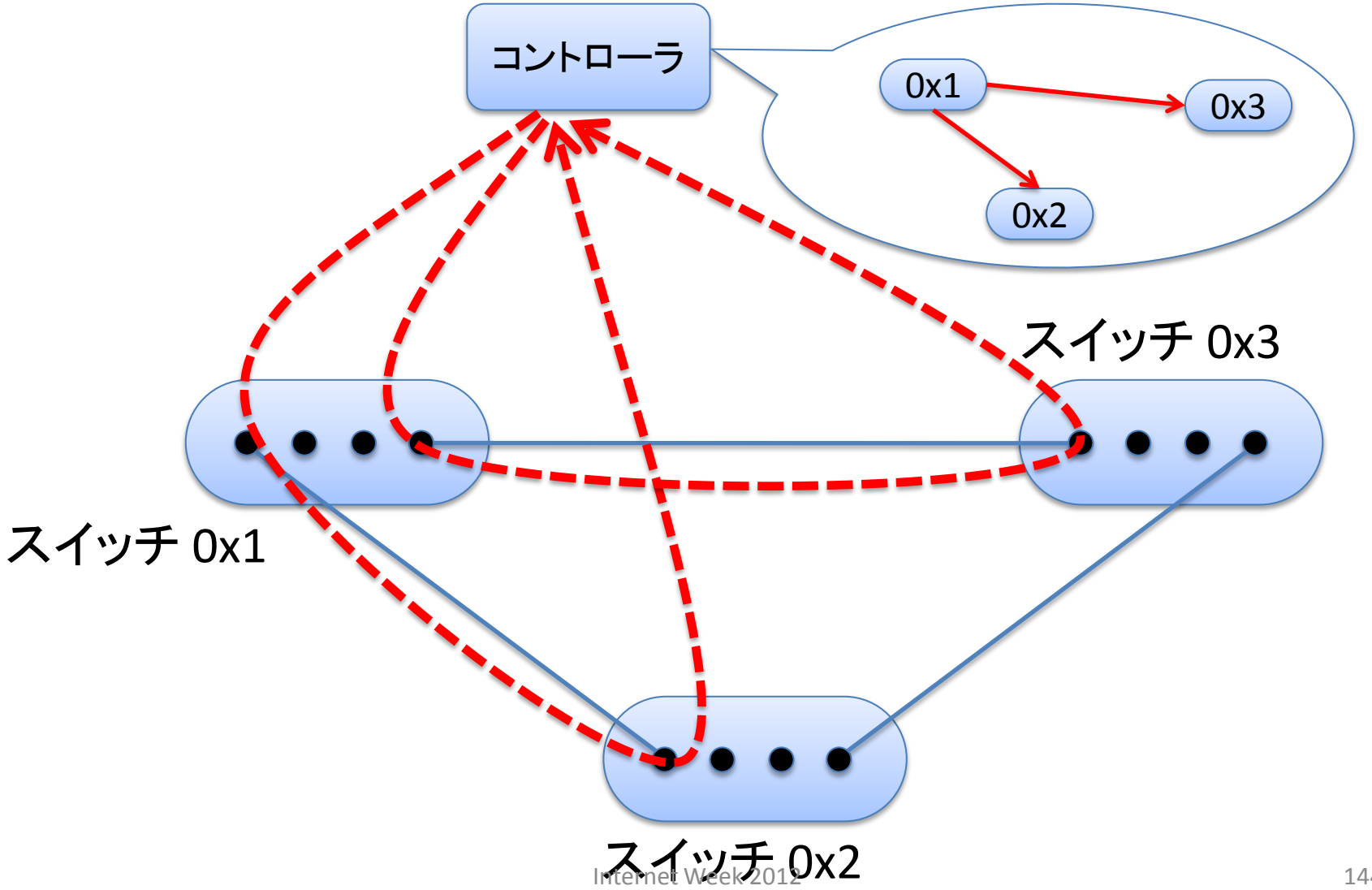
スイッチ 0x2 から送られてくる Packet In メッセージ



# トポロジー検出前

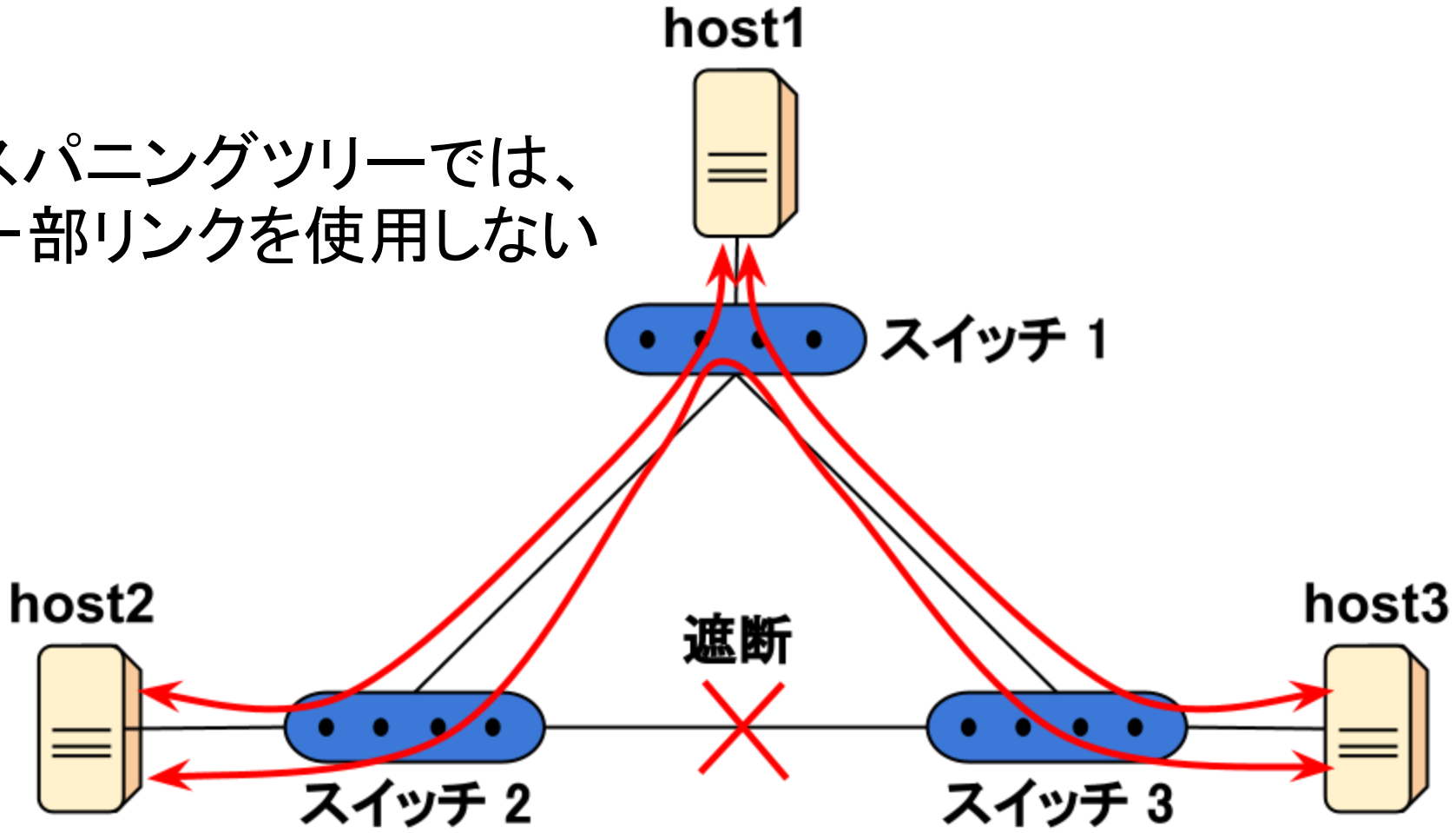


# トポロジー検出後



# Routing Switch の利点

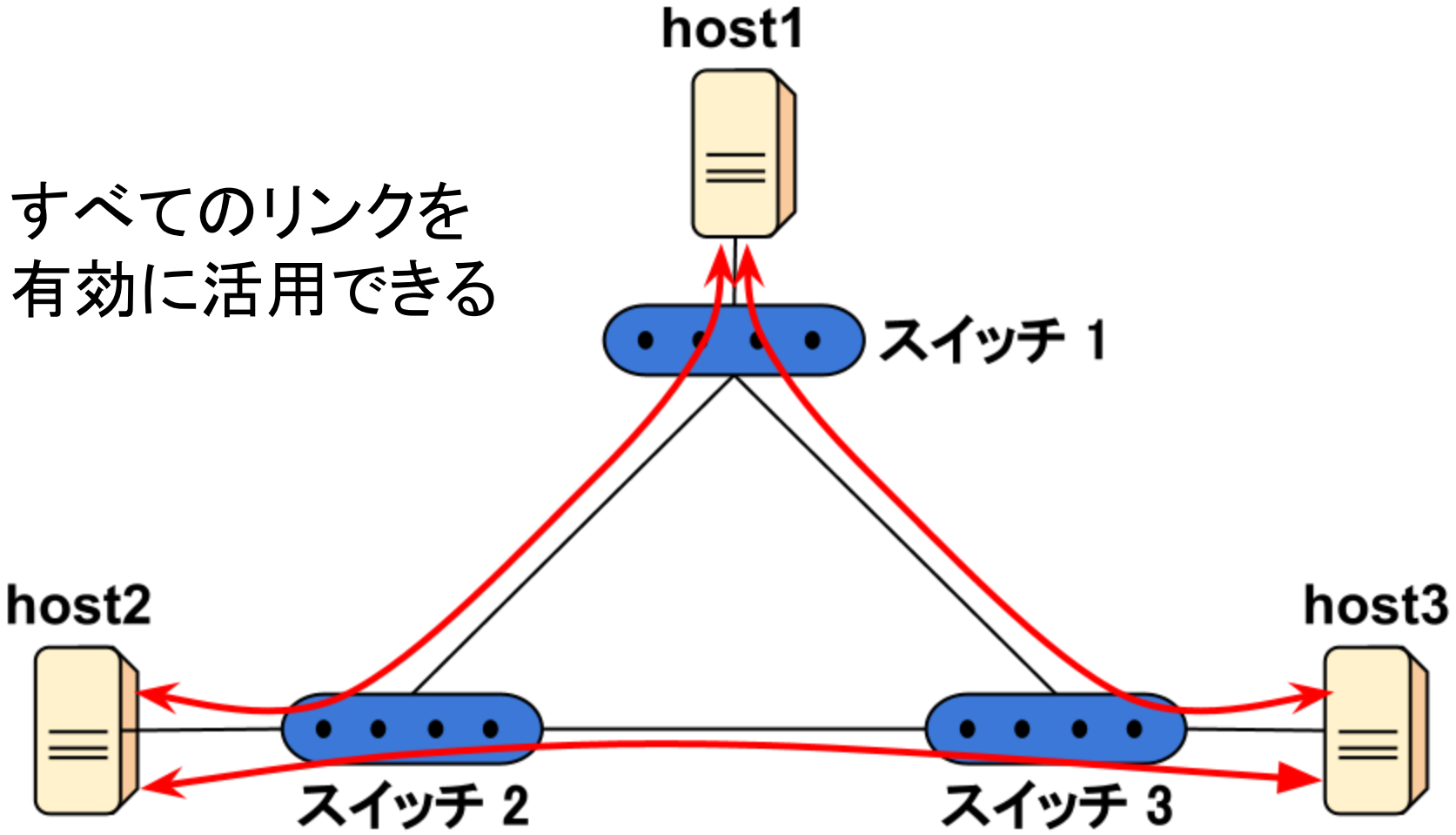
スパニングツリーでは、
一部リンクを使用しない




従来機器を使う場合

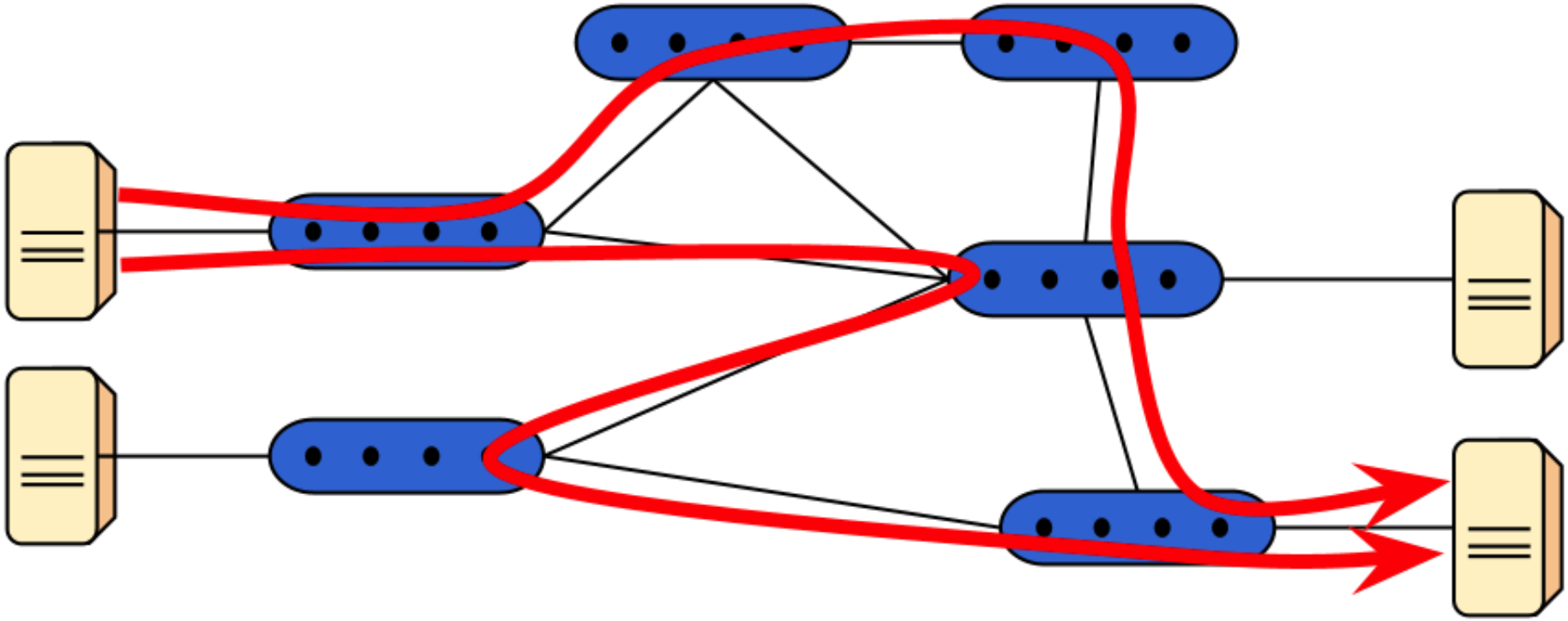
# Routing Switch の利点

すべてのリンクを
有効に活用できる




Routing Switch を使う場合

# Routing Switch の利点



マルチパスの実現が容易

# Routing Switch の動作 – まとめ

- 複数のスイッチを扱う
  - トポロジーの検出
  - 最短パスの計算
  - パスにそって、フローエントリを設定

- メリット
  - すべてのリンクを有効活用可能
    - スパニングツリーが不要
  - マルチパスの実現が容易
    - ただしマルチパスを計算するアルゴリズムの実装が必要

OpenFlow の活用例

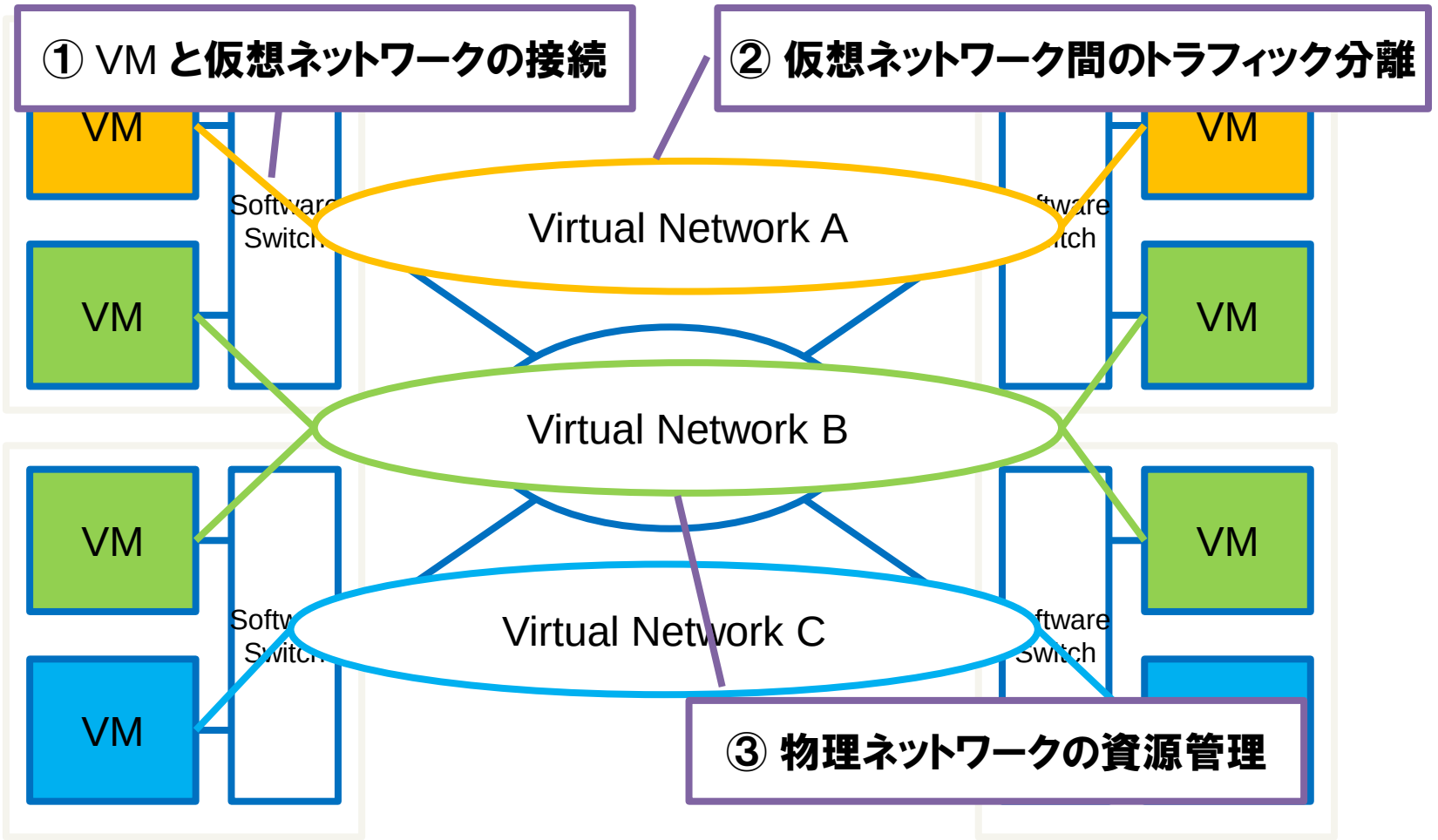
# ケーススタディ
# データセンターへの適用

# ここで学ぶこと

- データセンターネットワークにおける課題

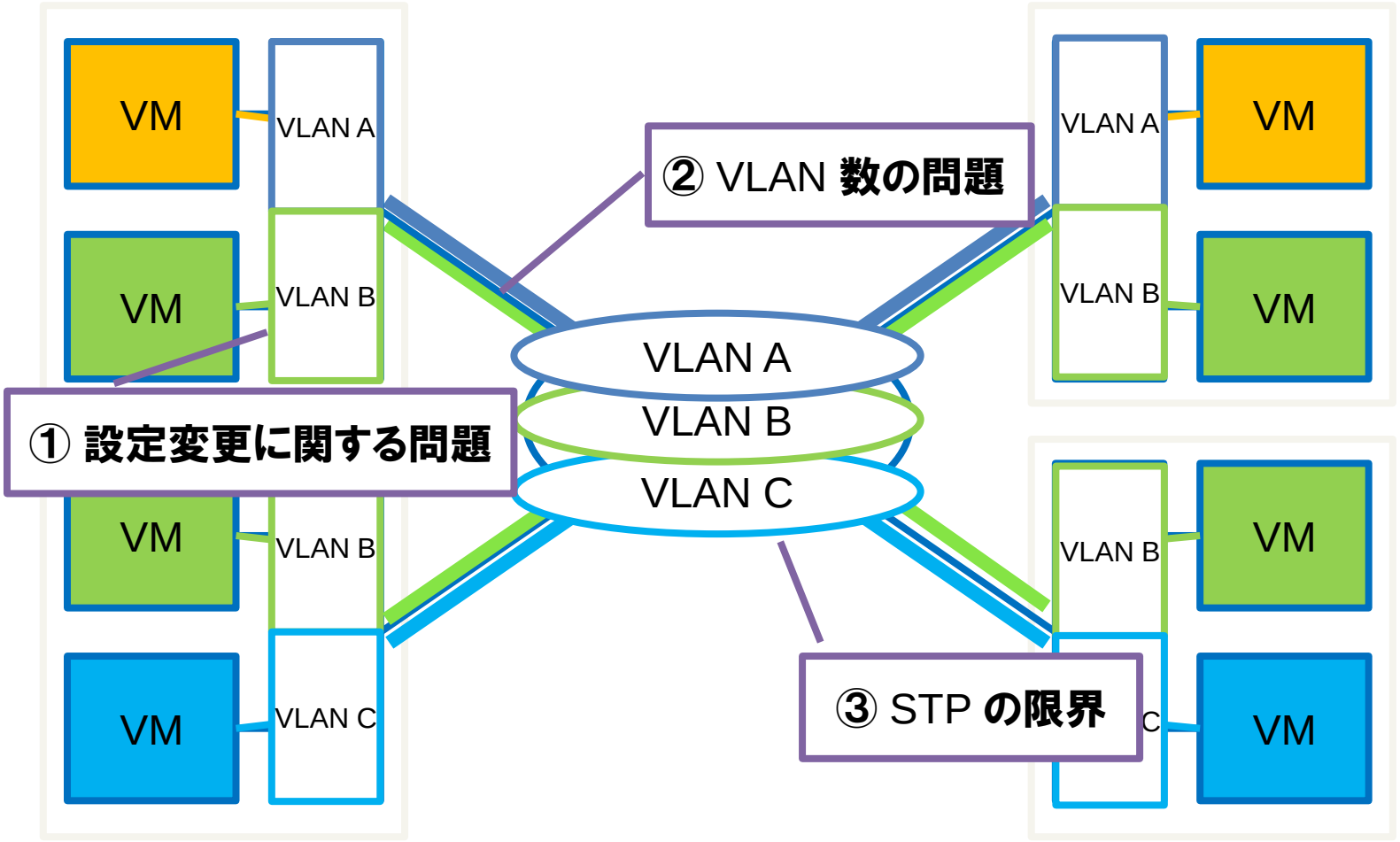
- OpenFlow を使いどのように解決を行うか？
  - Hop-by-hop モデル
  - Overlay モデル

# データセンターにおけるネットワーク仮想化



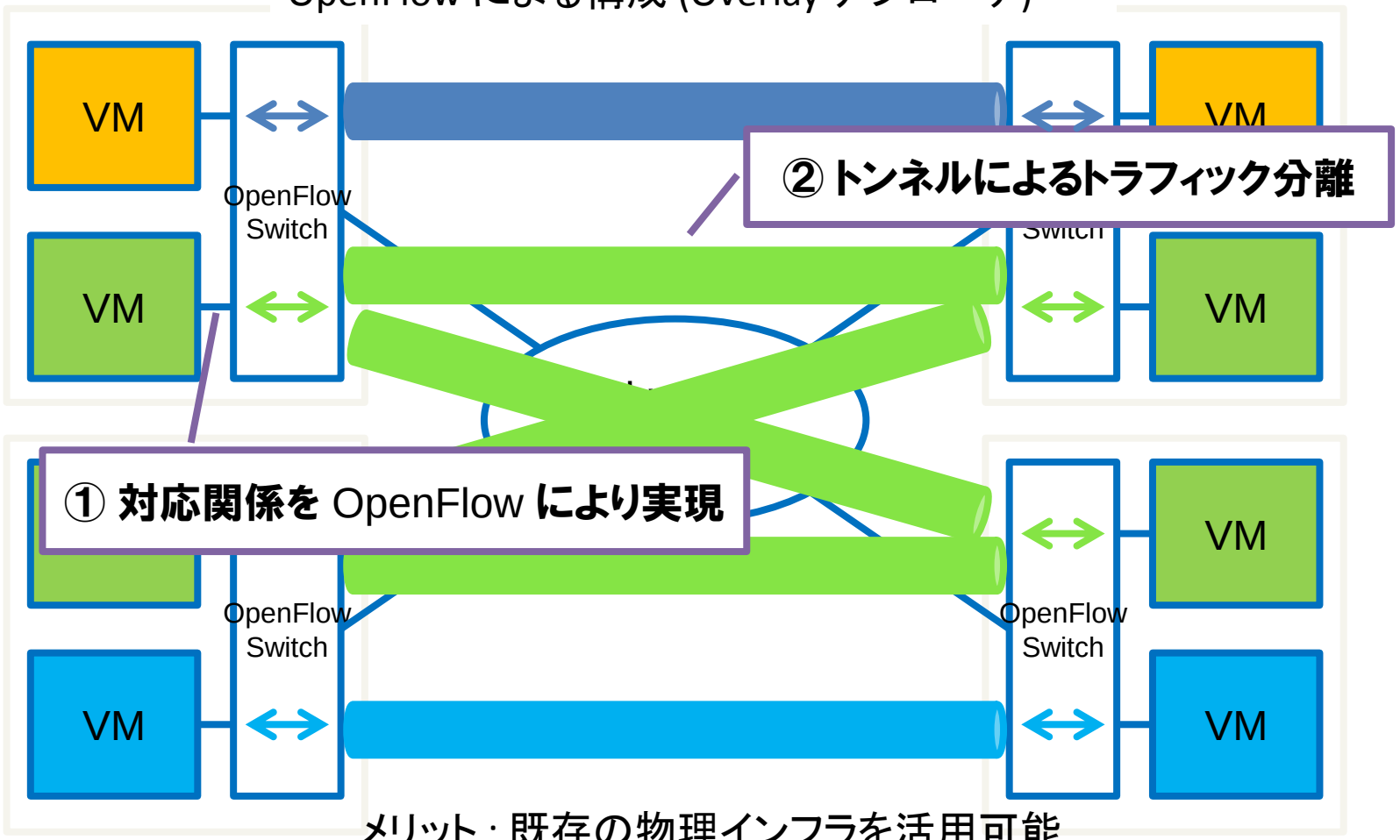
# データセンターにおけるネットワーク仮想化

従来技術 : VLAN による構成



# データセンターにおけるネットワーク仮想化

OpenFlow による構成 (Overlay アプローチ)

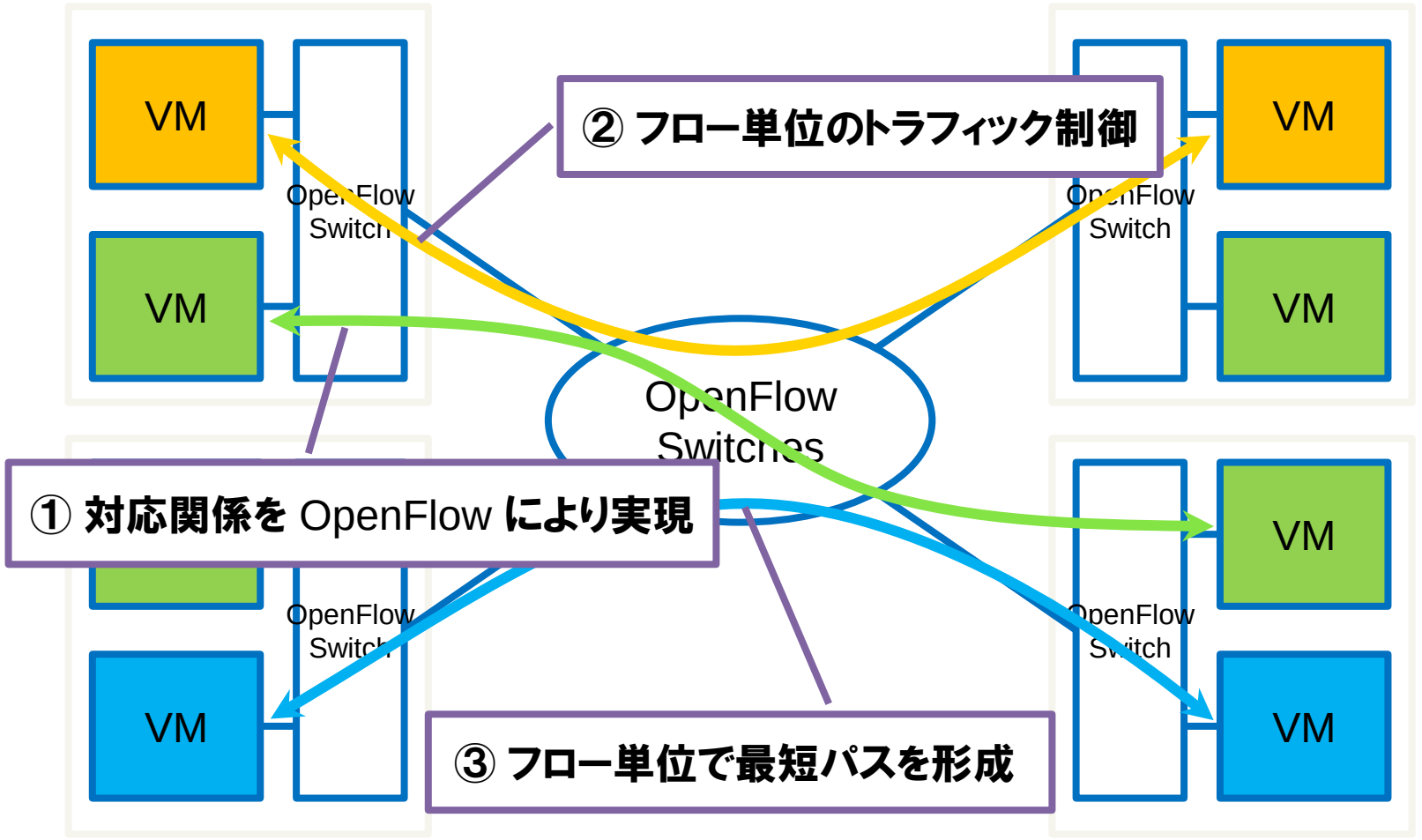



メリット : 既存の物理インフラを活用可能

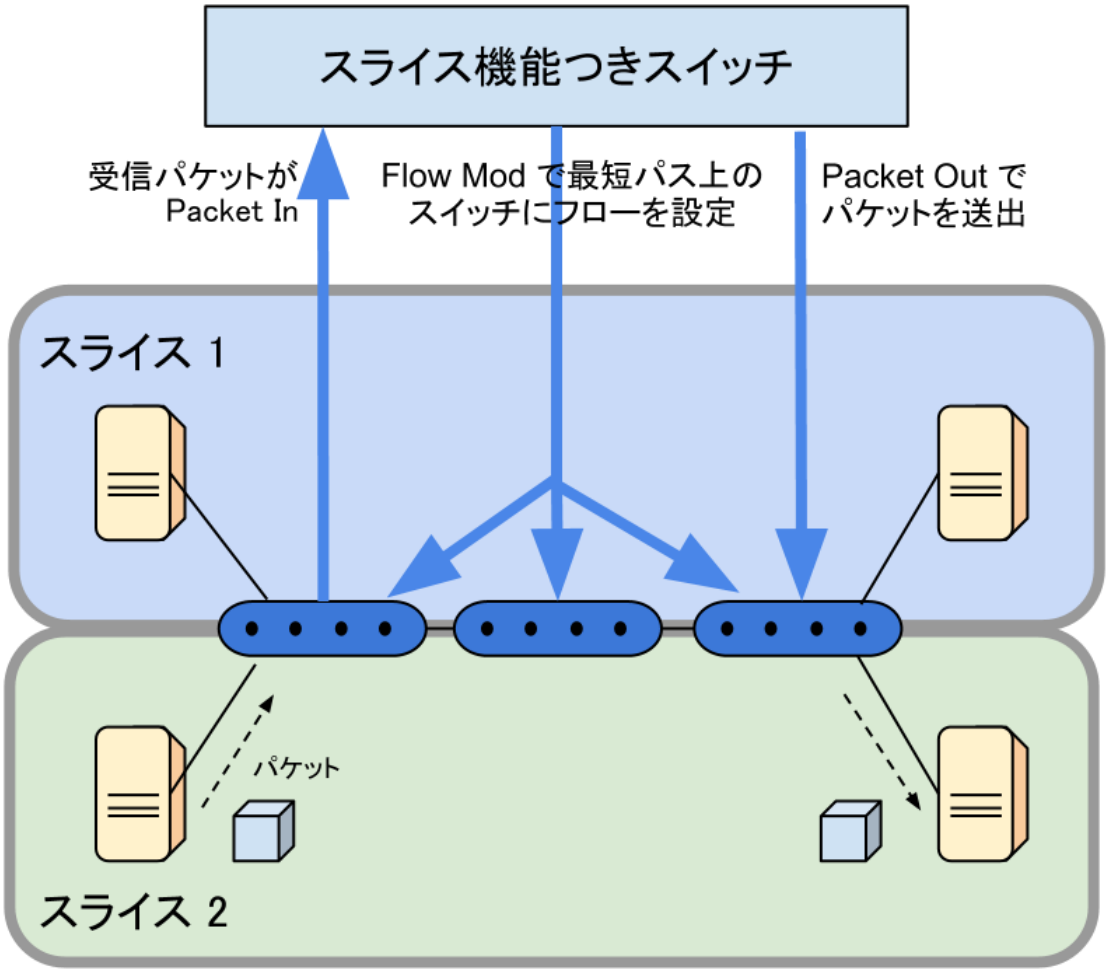
デメリット : トンネル化によるオーバーヘッド (MTU長、処理遅延)

# データセンターにおけるネットワーク仮想化

OpenFlow による構成 (Hop-by-hop アプローチ)

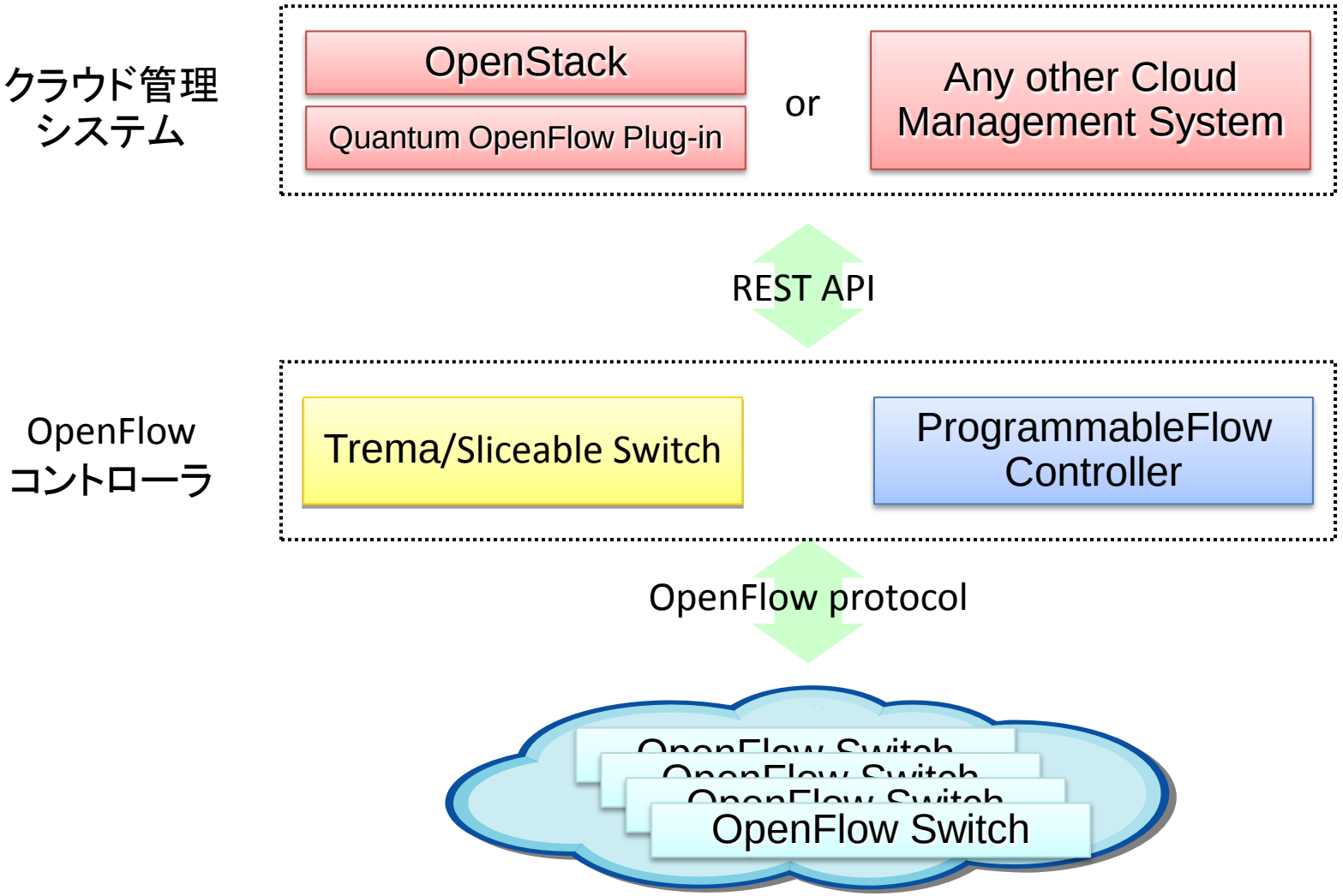


# Hop-by-hop 型の一実装例




フロー単位で、コントローラが判断を行い、パスを構成
→ 設定変更をスイッチに落とし込む必要がない

# クラウド管理システムとの連携例



# まとめ

# Trema: "Rails のように OpenFlow を"

- 書いたコードをすぐ動かせる: コーディング、実行、デバッグのループを短いサイクルで
  - 仮想ネットワーク DSL
  - `trema {run, send_packets, show_stats, up, kill}`
- Coding by Convention: 短く書く
  - naming coversion によるイベントの自動振り分け
  - Class アトリビュート: `periodic_timer_event`
  - Syntactic sugars: `ExactMatch.from`
  - デフォルトオプション: `send_flow_mod_add`
- Trema のサブコマンド
  - `trema {dump_flows, ruby}`

# 開発者、次の一歩のために

- `[trema]/src/examples`
  - API の使い方を示すシンプルなサンプルアプリケーション
  - Ruby と C のAPIを理解するために最適なリファレンス
- Trema/Apps http://github.com/trema/apps
  - より実用的・実験的なコントローラアプリケーション
  - 実用コントローラ開発の出発点として最適

# Trema C

- Trema は Ruby と C 両方のライブラリを提供
  - 開発者が選択可能
- Trema C もまた Trema Ruby のようにシンプル

```
$ gcc myapp.c `trema-config -c -l` -o myapp
$ trema run myapp
```

# OpenFlow ユースケース

- データセンターネットワークの仮想化
- データセンター間接続での TE

- どのような課題を、どのように解決しているか？

# Sources

- Trema: http://github.com/trema/
- Trema/Apps: http://github.com/trema/apps/
- Web Page: http://trema.github.com/trema/

- Book : http://yasuhito.github.com/trema-book/

- Mailing List: https://groups.google.com/group/trema-dev
- Blog : http://trema.info/, http://trema.hatenablog.jp/