解説

基応
専般

OpenFlowプログラミングへの招待

ネットワークインフラのプログラミングを可能とするSoftware-Defined Networking技術

鈴木一哉（NEC情報・ナレッジ研究所）

⇒ OpenFlowの誕生

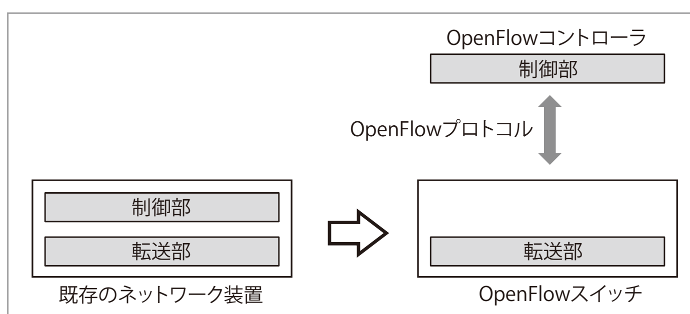

図-1　転送と制御の分離

現在広く使用されているネットワーク機器は，自律的に収集した情報を元に，パケットを転送する．この自律的動作のための制御機能が，それぞれのネットワーク機器に搭載されている．利用者は，ベンダによりあらかじめ用意されている範囲でこの制御機能を使用するしかなかった．

一方，ネットワークサービス事業者は，ネットワーク技術の複雑化に加え，事業環境の急激な変化や，ユーザニーズの多様化といった課題に直面している．このような課題解決のため，これまで以上に自由にネットワーク機器を扱うための技術が求められている．この技術は，ソフトウェアのように柔軟な制御を実現する技術ということで，Sofware-Defined Networking 技術と呼ばれている．

これらの動きの中で生み出された技術がOpenFlowである．OpenFlowでは，従来のネットワーク機器とは異なり，制御部と転送部を分離したアーキテクチャを採用している（**図-1**）．OpenFlowに対応したスイッチはパケットをどのように転送するかを決定するための制御部を持たない．OpenFlowスイッチは，OpenFlowコントローラと呼ばれる制御部から，OpenFlowプロトコルを使って送られた指示に従って，パケットを転送する．そのため，ネットワーク管理者は，このOpenFlowコントローラを自ら設計・実装することで，必要な制御機能を自由に実現できる．

本稿ではOpenFlowの概要を説明した後，OpenFlowコントローラのプログラミング方法を紹介する．

⇒ OpenFlowとは

OpenFlowでできること

OpenFlowは，柔軟なネットワーク制御を実現するために，以下に示す3つの特徴的な仕組みを持っている．

1つ目は，スイッチがパケットを転送するための規則を，コントローラから設定できる点である（**図-2**）．OpenFlowでは，このパケット転送のための規則を，フローと呼ぶ．コントローラが設定するフロー次第で，スイッチにさまざまな動作をさせることができる．

フローは，条件（Match）と処理（Action）から構成される．条件は，**表-1**に示す12 種のフィールドを使用でき，受信したパケットの識別に用いられる．識別に使用しないフィールドにはワイルドカードを指定することで，任意のフィールドのみを用いることができる．たとえば，“受信ポート番号が2であり，かつ送信元MACアドレスが

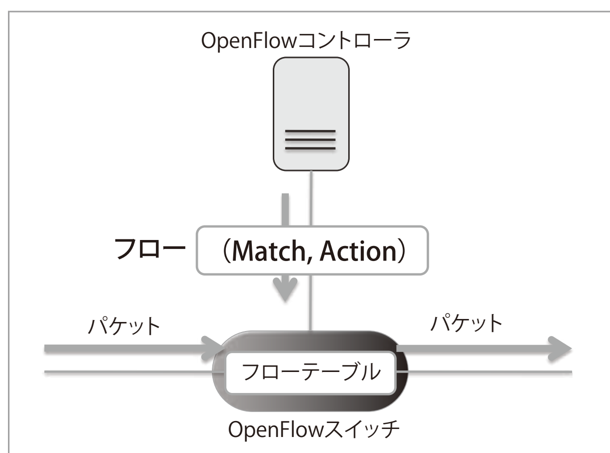

図 -2　フローの設定

フィールド	説明	Trema 上での表記
Ingress port	受信ポート	in_port
Ethernet src address	送信元 MAC アドレス	dl_src
Ethernet dst address	宛先 MAC アドレス	dl_dst
Ethernet type	プロトコル種別	dl_type
VLAN id	VLAN ID	dl_vlan
VLAN priority	VLAN PCP 値	dl_vlan_pcp
IP src address	IP 送信元アドレス	nw_src
IP dst address	IP 宛先アドレス	nw_dst
IP protocol number	プロトコル番号	nw_proto
IP ToS bits	ToS 値	nw_tos
Transport src port	送信元ポート番号	tp_src
Transport dst port	宛先ポート番号	tp_dst

表 -1　Match に用いられるフィールド

00:53:FF:00:01:01”や“IP パケットであり，その宛先 IP アドレスが 192.168.10.2”といった条件を指定できる．表 -1 中の Trema については，後述する．

OpenFlow スイッチは，フロー中の条件にマッチしたパケットに対して，そのフローで指定された処理を適用する．これらの処理は，1 つのフローに対して複数指定することができる．たとえば,“宛先 IP アドレスを書き換えた上での指定のポートからの出力”や,“指定のポートから出力させた後，さらに別のポートからも出力”といった動作を指定できる．

コントローラが作成したフローは，コントローラからスイッチへ，Flow Mod メッセージを用いて送られる．

2 つ目は，OpenFlow スイッチが受信したパケットを OpenFlow コントローラへ送ることができる点である（**図 -3**）．その際に用いられるのが，Packet In メッセージである．このメッセージを用いることで，スイッチが受信パケットの処理方法を知らない場合，コントローラ側で判断させることができる．たとえば，Packet In で送られてきたパケットを元に，そのパケットを転送するためのフローを生成するという動作を，コントローラにさせることができる．

3 つ目の仕組みは，コントローラが作成したパケットを，スイッチから出力させることが可能な点である．Packet Out メッセージを用いて，コントローラからスイッチへパケットを送ることで，そのパケットをスイッチから出力させることができる．図 -3 のように，Packet In でコントローラに送られてきたパケットを本来の宛先に送るために，スイッチ側へと送り返す際に用いることもできる．

OpenFlow の活用領域

OpenFlow は，以下に示す 2 つの領域での活用が有望視されている．

1 つ目の領域は，データセンタである．データセンタは，サーバに加え，スイッチやファイアウォール，ロードバランサなど多数の機器から構成されている．データセンタの運用コストを削減するためには，これらの機器を統一して管理する必要がある．しかし，ネットワーク機器の管理方法はベンダごとに異なっており，統一された方法は従来存在しなかった．さまざまなベンダのスイッチを統一的に扱うために，仕様が標準化されている OpenFlow の活用が有望視されている．

2 つ目の領域は，広域網である．世界中に分散してデータセンタを保有している事業者にとって，それらのデータセンタ間接続に用いられる広域網の有効活用が課題である．通常 IP パケットは，宛先まで最短経路で転送される．このとき，複数の最短経路が特定のリンクに集中することにより，リンク帯域の不足が発生する可能性がある．たとえば**図 -4** 中のデータセンタ B がデータセンタ C および D と通信する場合，リンク 1, 4, 5 には帯域に余

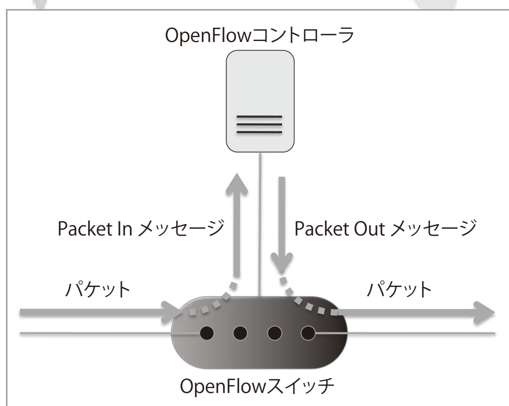

図 -3　Packet In / Packet Out メッセージ

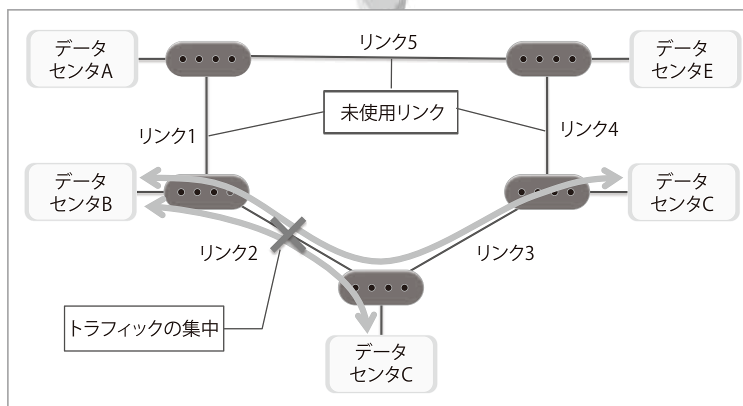

図 -4　特定リンクへの最短経路の集中

裕があるにもかかわらず，リンク 2 にトラフィックが集中する．

このようなトラフィック集中を防ぐために，広域網の制御に OpenFlow を活用しようという動きがある．トラフィックを帯域に余裕のあるリンクに迂回させるためには，ネットワーク中の各リンク帯域の使用状況を把握した上で，トラフィックを制御する必要がある．そのためには，OpenFlow のような集中制御方式が有効である．また，OpenFlow を使うことで，表 -1 に示すパケットの各フィールドを用いて，迂回させるトラフィック種別の識別・制御が可能である．

⇒ OpenFlow プログラミング

ここでは，Trema を使った OpenFlow のプログラミング方法について紹介する．

Trema とは

Trema とは，OpenFlow コントローラを開発するためのプログラミングフレームワークである．Trema は GPLv2 のライセンスでオープンソースとして公開されており，国内外の企業や大学，研究機関にて広く利用されている．

Trema は「ノート PC 1 台で簡単に OpenFlow コントローラ開発が可能」というコンセプトの元に開発が進められており，以下に示す 3 つの特徴がある．

- Trema を使うことで，必要な処理を短く記述することができる．
- Trema には，テストのためのネットワークエミュレータ機能が含まれている．開発したコントローラのテストを，開発マシン上で行うことができる．
- オリジナルのコントローラを開発する際に参考となる多数のサンプルアプリケーションが，Trema とともに提供されている．仮想ネットワークを構成するためのコントローラ SliceableSwitch 等が公開されている．

Trema のインストール

ここでは Trema（Version 0.4.7）を Ubuntu 14.04 にインストールする方法について説明する（他のディストリビューションを使う場合は，公式サイト http://trema.github.io/trema/ を参照のこと）．

まずは，必要なパッケージをインストールする．

```
$ sudo apt-get install gcc make libpcap-dev libglib2.0-dev
$ sudo apt-get install git ruby ruby-dev libsqlite3-dev
$ sudo gem install rubygems-update
$ sudo update_rubygems
```

次に，以下のコマンドで Trema 本体をインストールする．

```
$ sudo gem install trema
```

無事インストールができたかを，以下のコマンドで確認する．Trema のバージョン番号が表示されれば成功である．

```
$ trema --version
trema version 0.4.7
```

Trema プログラミング第一歩

Trema でオリジナルのコントローラを開発するとき，**リスト 1** (1) のように，Controller クラスを継承したクラスを定義する．この Controller クラスを継承することで，コントローラの基本機能が MyController クラスに組み込まれる．

リスト 1：mycontroller.rb

```
class MyController < Controller ← (1)
  def switch_ready datapath_id  ← (2)
    send_message datapath_id, FeaturesRequest.new
  end

  def features_reply datapath_id, message ← (3)
    puts "Datapath ID : #{ datapath_id.to_hex }"
    puts "# of ports : #{ message.ports.size }"
  end
end
```

Trema はイベントドリブンなプログラミングモデルを採用している．Trema には，コントローラ上でのイベント発生時に呼び出されるハンドラが，**表 -2** のように，定義されている．Trema を用いたプログラミングでは，これらのハンドラの中身を 1 つずつ具体化していくこととなる．

リスト 1 のコントローラには，`switch_ready` と `features_reply` の 2 つのハンドラが使われている．スイッチ接続時に呼び出される `switch_ready` ハンドラ（リスト 1 (2)）では，作成した Features Request メッセージをスイッチへと送る．Features Request メッセージを送ると，スイッチはバッファのサイズやポート情報などのスイッチ固有の情報を含む Features Reply メッセージをコントローラへと送り返す．リスト 1 のコントローラでは，Features Reply メッセージ受信時に呼び出される `features_reply` ハンドラ（リスト 1 (3)）にて，Features Reply メッセージの送り元であるスイッチの ID（Datapath ID と呼ばれる）とポート数を表示する．

コントローラの起動

ネットワークエミュレータ機能を使い，リスト 1 のコントローラを起動する．まずエミュレートするネットワークの構成を設定ファイルとして用意する．たとえば，仮想スイッチ 2 台に，仮想ホスト 2 台が接続するネットワークを作る場合，**リスト 2** のように記述する．

イベント	呼び出されるハンドラ
コントローラ起動	start
OpenFlow スイッチ接続	switch_ready
OpenFlow スイッチ切断	switch_disconnected
Packet In メッセージ受信	packet_in
Port Status メッセージ受信	port_status
Features Reply メッセージ受信	features_reply

表 -2　主なハンドラ

リスト2：network.conf

```
vswitch { dpid "0x1" } # 仮想スイッチの定義
vswitch { dpid "0x2" }
vhost ( "host1" ) # 仮想ホストの定義
vhost ( "host2" )
link "0x1", "host1" # 仮想リンクの定義
link "0x1", "0x2"
link "0x2", "host2"
```

コントローラの起動には，trema run コマンドを使う．-c オプションで設定ファイルを指定することで，エミュレートしたネットワーク上でコントローラを動作させることができる．

```
$ trema run ./mycontroller.rb -c ./network.conf
Datapath ID : 0x1
The number of ports : 3
Datapath ID : 0x2
The number of ports : 3 ← Ctrl+C で停止
```

スイッチ 2 台分の Datapath ID とポート数が表示されれば成功である．表示されたら，Ctrl+C でコントローラを停止する．

⇨ フィルタリングタップを作ってみよう

本章では，フィルタリングタップを実現するコントローラを，Trema を使ってプログラミングする方法を紹介する．

フィルタリングタップとは？

フィルタリングタップとは，ネットワーク中を流れるパケットをコピーしてネットワーク監視装置へと送り込む装置である．その際に，指定された条件に合うパケットのみをコピーし，ネットワーク監視装置へと送る．

図 -5 は，フィルタリングタップの配置例である．フィルタリングタップ内部では A ポートから入ったパケットは B ポートへ，また B ポートから入ったパケットは A ポートへと通り抜ける構造となっている．受信したパケットのうち，条件にあうパケットだけ，コピーされ，モニタリングポートから出力される．

コントローラの設計

OpenFlow スイッチをフィルタリングタップとして動作させるためには，**図 -6** のように 2 種類フローを設定する必

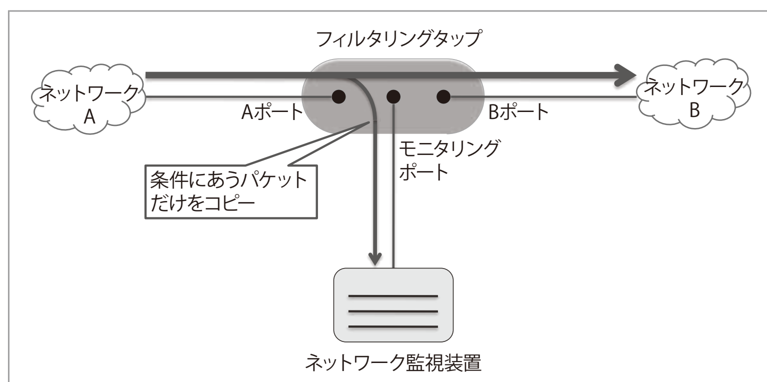

図 -5　フィルタリングタップの配置例

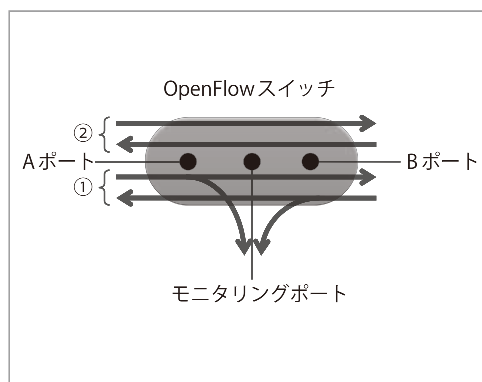

図 -6　フィルタリングタップのためのフロー

要がある．

1つ目は，フィルタリング対象パケットを受信した時，受信ポートではないポートとモニタリングポートの両方に，出力するフローである（図-6 ①）．2つ目は，フィルタリング対象ではないパケットを，受信ポートではないポートから出力するためのフローである（図-6 ②）．

フィルタリングタップをプログラミング

リスト3は，FilteringTap コントローラのソースコードである．

リスト3：フィルタリングタップ (filtering-tap.rb) のソースコード

```
 1: class FilteringTap < Controller
 2:   def start
 3:     load 'filtering-tap.conf'
 4:   end
 5:
 6:   def switch_ready dpid
 7:     make_path( dpid, $port_a, $port_b )
 8:     make_path( dpid, $port_b, $port_a )
 9:
10:     $conditions.each do | each |
11:       make_branch( dpid, $port_a, [ $port_m, $port_b ], each )
12:       make_branch( dpid, $port_b, [ $port_m, $port_a ], each )
13:     end
14:   end
15:
16: private
17:
18:   def make_path dpid, in_port, out_port
19:     send_flow_mod_add( dpid,
20:       :match => Match.new( :in_port => in_port ),
21:       :actions => [ SendOutPort.new( out_port ) ],
22:       :priority => 50000
23:     )
24:   end
25:
26:   def make_branch dpid, in_port, out_ports, condition
27:     condition[ :in_port ] = in_port
28:     actions = out_ports.collect do | each |
29:       SendOutPort.new( each )
30:     end
31:     send_flow_mod_add( dpid,
32:       :match => Match.new( condition ),
33:       :actions => actions,
34:       :priority => 65535
35:     )
36:   end
37: end
```

コントローラの起動時に呼び出される `start` ハンドラでは，設定ファイルである `filtering-tap.conf`（**リスト4**）を読み込む．このファイルでは，OpenFlow スイッチのどのポートを，図-6 における A ポート，B ポート，モニタリングポートとして動作させるかを指定する（リスト4の1～3 行目）．さらに，モニタリング対象を，リスト4の5～8 行目のように指定する．ここで指定した内容は，最終的に `Match` の引数として用いられる．この例では，`dl_type` と `nw_proto` のみを使用しているが，ほかにも表-1 に示す各フィールドを使用できる（ただし，`in_port` は除く）．

リスト 4：filtering-tap.conf

```
1: $port_a = 1
2: $port_b = 2
3: $port_m = 3
4:
5: $conditions = [
6:   { :dl_type => 0x0800,
7:     :nw_proto => 1 }
8: ]
```

リスト 3 中の switch_ready ハンドラは，コントローラへ接続したスイッチに対して，フローを設定する．前節で説明した図 -6 ②のフローを 7 ～ 8 行目の make_path を使って設定する．また，図 -6 ①のフローは，11 ～ 12 行目の make_branch にて設定する．

make_path（18 ～ 24 行目）を見ると，send_flow_mod_add を用いて，Flow Mod メッセージを送っている．ここで送られるフロー中の条件には，in_port のみを指定した Match を用いている．このフローの処理には，指定のポートからの出力を指示する SendOutPort を指定している．また，このフローの優先度を 50000 と設定している．

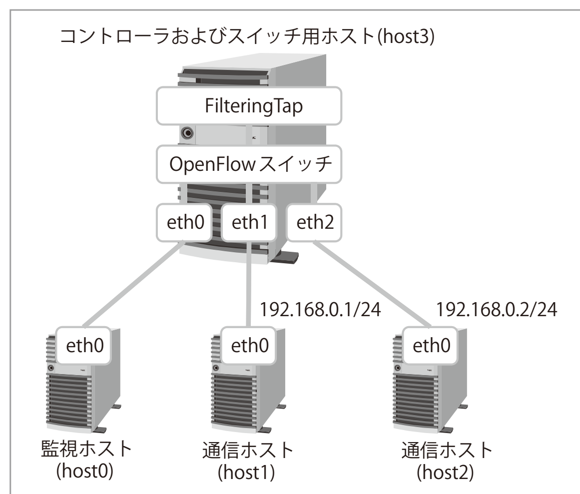

図 -7　動作構成

次に make_branch（26 ～ 36 行目）を見てみる．こちらも make_path と同様に，send_flow_mod_add を用いているが，送るフローが異なる．このフローでは，リスト 4 に記述された内容に，in_port を追加した条件を用いる．このフロー中の処理は，条件に合うパケットをコピーして 2 つのポートから出力させる必要がある．そのため，リスト 3 の 28 ～ 30 行目にて，Ruby における collect イテレータを用い，複数の SendOutPort を含むリスト actions を作る．このフローの優先度には 65535 を指定する．受信パケットに対して，条件に合うフローが複数ある場合，優先度が一番高いフローの処理が適用される．そのため，make_branch が作るフローの優先度を 65535 とすることで，このフローは make_path が作るフロー（優先度 50000）より優先される．

フィルタリングタップを動かす

ここでは，Trema のネットワークエミュレータ機能に組み込まれたソフトウェア版の OpenFlow スイッチを使って，リスト 3 のコントローラを動かす方法を説明する．**図 -7** に示すように，4 台のホストを用意する．host1, host2 間で通信できるように，それぞれに IP アドレスを割り当てておく．

この構成では，リスト 2 で説明した仮想ホストの代わりに実際のホストを使っている．そのため，仮想リンクの接続先には，OpenFlow スイッチを動作させるホスト（host3）のネットワークインタフェース名（eth0，eth1，eth2）を指定する（**リスト 5**）．

リスト 5：network.conf

```
vswitch { dpid "0xabc" }
link "0xabc", "eth0"
link "0xabc", "eth1"
link "0xabc", "eth2"
```

次のように host3 上でリスト 3 のコントローラとスイッチを起動する．

```
host3$ trema run ./filtering-tap.rb -c ./network.conf
```

コントローラが起動したら，host1 から host2 宛てに ping を打つ．

```
host1$ ping 192.168.0.2
PING 192.168.0.2 (192.168.0.2) 56(84) bytes of data.
64 bytes from 192.168.0.2: icmp_req=1 ttl=64 time=0.263 ms
64 bytes from 192.168.0.2: icmp_req=1 ttl=64 time=0.138 ms
...
```

監視ホスト側への ICMP パケット到達を確認する．次のように tcpdump コマンドを使い，インタフェース (eth0) に届くパケットを覗いてみる．

```
host0$ sudo tcpdump -ni eth0
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 65535 bytes
12:11:37.190729 IP 192.168.0.1 > 192.168.0.2: ICMP echo request, id 15129, seq 1, ...
12:11:37.190856 IP 192.168.0.2 > 192.168.0.1: ICMP echo reply, id 15129, seq 1, ...
...
```

上記のように host1, host2 間の ICMP パケットが表示されれば成功である．ping 以外にも，ssh や ftp 等の通信も試してみよう．ICMP 以外の通信は観測できないはずである．

ここで説明したソフトウェア版の OpenFlow スイッチの代わりに，ハードウェアの OpenFlow スイッチを使う場合の注意点は以下のとおりである．

1. OpenFlow コントローラとスイッチの間では，TCP を使った通信が行われる．そのため，コントローラが動作するホストとスイッチとの間で通信ができるよう，それぞれに適切な IP アドレスを割り当てる．
2. ネットワークエミュレータ機能を使わないため, `-c` オプションを用いずに, `filtering-tap.rb` コントローラを起動する．

⇒ 今後に向けて

OpenFlow プログラミングにより，これまでより自由にネットワーク機器を制御できる．そのため OpenFlow は，アイディア次第で，さまざまな場面で活用できる可能性を秘めている．今回紹介した Trema には，多数のサンプルコントローラが公開されている (https://github.com/trema/apps)．また OpenFlow を仕様から詳しく解説している文献 1) や，Trema でのプログラミングを解説した文献 2)，3) 等が出版されている．これらを参考に，オリジナルの OpenFlow コントローラの作成にチャレンジしてみてほしい．

参考文献

1) あきみち, 宮永直樹 , 岩田　淳 : マスタリング TCP/IP OpenFlow 編 , オーム社 (2013)．
2) 高宮安仁，鈴木一哉 : OpenFlow 実践入門 , 技術評論社 (2013)．
3) 石井秀治, 大山裕泰, 河合栄治 : 次世代ネットワーク制御技術 OpenFlow 入門, アスキー・メディアワークス (2013)．

(2014 年 4 月 7 日受付)

鈴木一哉　kazuya@ax.jp.nec.com

博士（システムズ・マネジメント）．1997 年日本電気（株）入社．現在情報・ナレッジ研究所主任研究員．2014 年より電気通信大学大学院 情報システム学研究科 客員准教授．