# Programming Assignments for Graduate Students using GENI

## Flow Management using OpenFlow on GENI



Please direct comments regarding this document to `fahmy@cs.purdue.edu`.

# 1 Introduction

This project leverages resources on the ProtoGENI aggregate, using the Omni GENI client [5]. The ProtoGENI tutorial [10] is a good starting point to become familiar with the ProtoGENI aggregate. General documentation on the GENI project and its available resources is found on the GENI wiki [6].

GENI resources are shared resources provided by members of the networking community. Please release your slivers when you are done with them, or if you are going to have to leave them for an extended period of time. Remember that in many cases you may be able to perform an experiment, copy the data off to another host, and release the sliver.

A moderately deep understanding of the OpenFlow [7] controller model and API is required for this project. The OpenFlow specification, version 1.1.0 [9] is a valuable reference for OpenFlow and the OpenFlow controller model.

## 1.1 Objectives

The objective of this assignment is to familiarize you with the software-defined networking, as well OpenFlow technology. You will specifically learn about:

- Basic OpenFlow principles, capabilities, and limitations
- OpenFlow controller/switch interactions
- Implementation of a specific OpenFlow controller application
- Integration of external information with OpenFlow decision making
- Application of control loop theory to practical networking problems

## 1.2 Assignment Material

The assignment material for this assignment can be found at:
`http://www.cs.purdue.edu/homes/fahmy/geni/geni-openflow.tar.gz`

## 1.3 Tools

**Trema** Each exercise in this assignment will require you to design and implement an OpenFlow controller. Numerous libraries and controller frameworks are available for this task. The guidelines in this assignment assume that you are using the Trema [11] controller framework. The Trema project web site, `http://trema.github.com/trema/`, contains documentation, a number of helpful examples, and some tutorial material for learning to use Trema.

**Traffic Control (`tc`)** The `tc` command is available in the GNU/Linux distributions on ProtoGENI nodes, found in the `/sbin` directory. This command manipulates the Linux network forwarding tables, allowing for configuration of *network emulation*, which allows the Linux kernel to emulate various network conditions such as delay or loss. This effect is provided by the `netem` subcommand. In these exercises, `tc` will be used to modify network conditions. Example command lines will be provided.

**Iperf** Iperf [2] is used to measure the bandwidth performance of Internet links. In these exercises, it is used to study the behavior of TCP in the face of changing link characteristics. It is available on ProtoGENI nodes, located at `/usr/local/etc/emulab/emulab-iperf`. Iperf runs as both a server and a client. The server is started with the `-s` command line option, and listens for connections from the client. The client is started with the `-c <server>` command line option, and connects to the server and sends data at either the fastest possible rate (given the underlying network) or a user-specified rate. The `-p <port>` option to either the client or server mode of Iperf specifies that it should use the specified port number. You will find Iperf useful in testing and evaluating your implementations.

**Telnet** The Unix `telnet <host> <port>` command allows you to easily connect to a specified TCP port on a remote host and manually input data. You may find it useful for testing and evaluating your implementations.

# 2 Debugging an OpenFlow Controller

You will find it helpful to know what is going on inside your OpenFlow controller and its associated switch when implementing these exercises. This section

contains a few tips that may help you out if you are using the Open vSwitch implementation provided with this handout. If you are using a hardware OpenFlow switch, your instructor can help you find equivalent commands.

The Open vSwitch installation provided by the RSpec included in the handout materials is located in `/opt/openvswitch-1.6.1-F15`. You will find Open vSwitch commands in `/opt/openvswitch-1.6.1-F15/bin` and `/opt/openvswitch-1.6.1-F15/sbin`. Some of these commands may be helpful to you. If you add these paths to your shell's `$PATH`, you will be able to access their manual pages with `man`. Note that `$PATH` will not affect `sudo`, so you will still have to provide the absolute path to `sudo`; the absolute path is omitted from the following examples for clarity and formatting.

## 2.1 ovs-vsctl

Open vSwitch switches are primarily configured using the `ovs-vsctl` command. For exploring, you may find the `ovs-vsctl show` command useful, as it dumps the status of all virtual switches on the local Open vSwitch instance. Once you have some information on the local switch configurations, `ovs-vsctl` provides a broad range of capabilities that you will likely find useful for expanding your network setup to more complex configurations for testing and verification. In particular, the subcommands `add-br`, `add-port`, and `set-controller` may be of interest.

## 2.2 ovs-ofctl

The switch host configured by the handout materials listens for incoming OpenFlow connections on localhost port 6634. You can use this to query the switch state using the `ovs-ofctl` command. In particular, you may find the `dump-tables` and `dump-flows` subcommands useful. For example, `sudo ovs-ofctl dump-flows tcp:127.0.0.1:6634` will output lines that look like this:

```
cookie=0x4, duration=6112.717s, table=0, n_packets=1,
n_bytes=74, idle_age=78,priority=5,tcp,
nw_src=10.10.10.0/24 actions=CONTROLLER:65535
```

This indicates that any TCP segment with source IP in the 10.10.10.0/24 subnet should be sent to the OpenFlow controller for processing, that it has been 78 seconds since such a segment was last seen, that one such segment has been seen so far, and the total number of bytes in packets matching this rule is 74. The other

fields are perhaps interesting, but you will probably not need them for debugging. (Unless, of course, you choose to use multiple tables — an exercise in OpenFlow 1.1 functionality left to the reader.)

### 2.3 Unix utilities

You will want to use a variety of Unix utilities, in addition to the tools listed in Section 1.3, to test your controllers. The standard `ping` and `/usr/sbin/arping` tools are useful for debugging connectivity (but make sure your controller passes ICMP ECHO REQUEST and REPLY packets and ARP traffic, respectively!), and the command `netstat -an` will show all active network connections on a Unix host; the TCP connections of interest in this exercise will be at the top of the listing. The format of `netstat` output is out of the scope of this document, but information is available online and in the manual pages.

## 3 Exercises

### 3.1 Building a Firewall with OpenFlow

A firewall observes the packets that pass through it, and uses a set of rules to determine whether any given packet should be allowed to pass. A *stateless* firewall does this using only the rules and the current packet. A *stateful* firewall keeps track of the packets it has seen in the past, and uses information about them, along with the rules, to make its determinations.

In this exercise, you will build a stateful firewall controller for TCP [1] connections in OpenFlow. The first packet of each connection will be handled by the controller, but all other connection packets will be handled by the OpenFlow-enabled router or switch without contacting your controller. This design will allow you to write powerful firewall rule sets without unduly impacting packet forwarding speeds. Your controller will parse a simple configuration file to load its rules. Complete stateful firewalls often handle multiple TCP/IP protocols (generally at least both TCP and UDP), track transport protocol operational states, and often understand some application protocols, particularly those utilizing multiple transport streams (such as FTP, SIP, and DHCP). The firewall you will implement for this exercise, however, needs handle only TCP, and will not directly process packet headers or data.

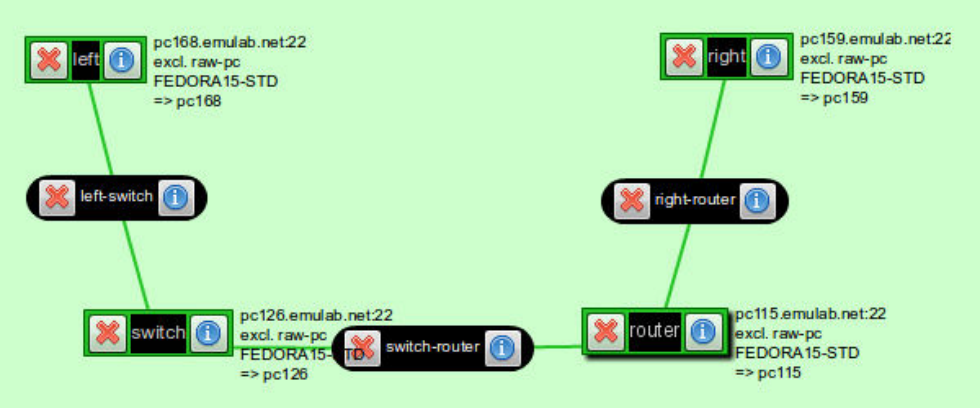


Figure 1: Simple firewall testing topology.

### Network Setup

In the assignment materials tarball provided by your instructor, you will find an RSpec called `fw.rspec`. It implements the topology shown in Fig. 1. The specific host names allocated for your experiment will be different, but the topology will be isomorphic to Fig. 1. The host labeled `left` in the figure is "behind" the firewall, implemented by the Open vSwitch host labeled `switch`. The host labeled `router` handles IP routing for the firewalled network, and every host on the other side of this router (the host labeled `right` being the only example on this topology; you may wish to add others for your testing and experimentation) are "outside" of the firewall.

The provided RSpec and the files it installs on the hosts it allocates will configure a complete, working network with an Open vSwitch running on the host labeled `switch`. The Open vSwitch switch is configured to connect to a controller on localhost (that is, the `switch` host), but no controller is started; until a controller is started on localhost, the Open vSwitch will act like a normal learning switch, forwarding all packets to the appropriate interface based on MAC address. Trema is installed in `/opt/trema-trema-8e97343`. Once you have implemented your switch, you can simply use this Trema install to run it and the Open vSwitch will obey its configuration.

You can test that the network configured correctly by waiting a few moments after Flack or Omni (or whatever GENI tool you are using) suggests that the sliver is ready, and running `ping right` from the host allocated for `left` or vice-versa. Since the fallback switch configuration will act like a normal learning

switch, the ping packets should go through.

### Firewall Configuration

The firewall configuration language is very simple. All flows not specified in the configuration are assumed to be forbidden, and the default packet processing policy on the OpenFlow device you are managing should be to drop packets. The configuration language will specify, one flow to a line, the TCP flows that should be permitted to pass the firewall. The syntax is:

```
<ip>[/<netmask>] <port> <ip>[/<netmask>] <port>
```

Items in angle brackets (<>) represent variable values, items in square brackets (`[]`) represent optional syntax, and unquoted characters (*e.g.*, the slash characters) represent themselves. The first subnet (IP address plus mask length) and port number are the subnet and port number of the host initiating the connection (that is, sending the first bare SYN), and the second subnet and port number are those of the host accepting the connection. IP addresses are specified as dotted quads (*e.g.*, `192.168.1.0`) and netmasks as bit lengths (*e.g.*, `24`). If a netmask is missing (the IP address for a given subnet is not followed by a slash and an integer), it is equivalent to `/32`. Port numbers are integers. Either or both of the IP address or port numbers may be replaced by the word `any`, equivalent to `0.0.0.0/0` in the case of IP address, or to any port number, in the case of port numbers.

A sample configuration that implements a firewall permitting inbound connections to a web server at IP address 192.168.1.1 on port 80, and any outbound connections initiated by hosts inside the firewall (protecting 192.168.1.0/24) is as follows:

```
any any 192.168.1.1 80
192.168.1.0/24 any any any
```

All whitespace will be a single ASCII space character, all newlines will be a single ASCII newline character (`0x0a`) Empty lines (two newlines back-to-back) are permitted.

A connection is allowable if it matches *any* rule. A connection matches a rule if all four elements of the four-tuple match. Subnet matching uses standard rules, expressed in this pseudocode:

```
boolean subnet_match(IP subnet, int bits, IP addr) {
    int32 bitmask = ~((1 << 32 - bits) - 1);
    IP addrnet = addr & bitmask;
    return addrnet ^ subnet == 0;
}
```

Note that rules are not bidirectional; the presence of the first rule in this set does *not* imply the second:

```
192.168.1.0/24 any any any
any any 192.168.1.0/24 any
```

This means that the *first packet the controller sees* that matches a flow causes the flow to be allowed. Packets that would trigger a *reply* that would be allowed are not necessarily allowed.

The name of a firewall configuration file will be provided on the controller command line. To provide an argument to your controller application, it must be included with the controller file name. For example, to configure your firewall found in `firewall.rb` to load `fw.conf`, you would invoke:
`trema run 'firewall.rb fw.conf'`

You will then find `['firewall.rb', 'fw.conf']` in `ARGV` when your controller's `start` method is invoked.

### Firewall Semantics

When an OpenFlow device connects to your controller (that is, you receive a `switch_ready` controller event), your controller should send it instructions to:

- Pass all packets matching allowed connections to your controller
- Drop all other packets

Priorities are going to be critical to the correct operation of your controller, so set them carefully. Higher priority rules match before lower priority rules, and the first matching rule is followed. See Section 3.4 of the OpenFlow specification [8] for more details on flow matching.

Upon receiving a packet from the OpenFlow device (via a `OFPT_PACKET_IN` message), your controller should:

- Ensure that the packet matches a rule in the configuration

- Insert a flow match in the OpenFlow device for the *complete four-tuple* matching the incoming packet
- Insert a flow match in the OpenFlow device for the *complete four-tuple* matching the opposite direction of the same connection
- Instruct the OpenFlow device to forward the incoming packet normally (using `OFPP_NORMAL`)

Packets which do not match a rule on the controller should be denied. Because your initial device configuration eliminates most of these packets outright, your controller should not see a large number of these packets.

Because this firewall implementation cannot track the actual state of the TCP connections it is managing, removing accepted connections from the forwarding tables on the OpenFlow device must be handled by timers. OpenFlow rules can be removed by an idle timer as well as expired a fixed period after insertion. For this firewall, use an idle timer of 300 seconds.

### Limitations of this Approach

Note that this approach to implementing a firewall has drawbacks. Because the OpenFlow controller does not, and can not efficiently, track the precise state of the TCP flow it is forwarding, the rules are a little bit sloppy. In particular, connections "in progress" when the firewall comes online are not differentiated from new connections created after the firewall is initialized, and connection closings cannot be detected by the controller. The former can be managed by inspecting the packet headers included in the `OFPT_PACKET_IN` message when a connection is opened, but the latter cannot easily be mitigated. This means that connections with long idle times (and 300 s is not particularly unusual, in the long tail of TCP connection statistics!) will be disconnected unnecessarily, and new connections reusing recent four-tuples may be passed through the firewall without examination by the controller.

### Hints

The following list of hints may help you design and debug your implementation more rapidly.

- Remember that OpenFlow switches are an *Ethernet switch* first and foremost, and that not all packets on an Ethernet are IP. In particular, your hosts will require ARP in order to pass IP traffic through the switch!
- You may pass ICMP packets without limitation, to make debugging easier.
- The Trema `Match` class has a `compare()` method that accepts a `Match` argument and may be useful to you — consider the `ExactMatch#from()` method in conjunction.

### Extra Credit

For extra credit (if permitted by your instructor), generate TCP reset segment at the firewall to reset rejected connections.

## 3.2 Extending the Firewall

OpenFlow controllers can also make complex flow decisions based on arbitrary state. This is one benefit to removing the controller from the network device — the controller is free to perform any computation required over whatever data is available when making decisions, rather than being constrained to the limited computing power and storage of the network device. For this exercise, you will extend the firewall described in Section 3.1 to include rudimentary denial of service prevention using this capability.

### Extended Firewall Configuration

You will extend the firewall configuration language to accept an additional final parameter, an integer representing the number of allowable connections matching a given rule at any point in time. As before, the keyword `any` will be used to indicate that no limiting is to be performed on the rule. The new firewall configuration syntax is:

```
<ip>/<netmask> <port> <ip>/<netmask> <port> <limit>
```

### Connection Limiting Semantics

The extended firewall will perform flow matching as before, with one added check: if the number of existing flows allowed by a given rule exceeds the limit

specified in the configuration, a new flow matching that rule will be denied. The number of existing flows matching a given rule is computed as the number of currently active flow matches in the OpenFlow device for that rule. You may wish to look into the `OFPT_FLOW_REMOVED` message for help in implementing this. If a connection rule specifies `any` as the flow limit, no limiting will be performed by the controller.

Note that the timeout-based nature of flow removal dictates that small connection limits will be *quite* limiting. Keep this in mind when testing your firewall!

## 3.3 Load Balancing

Load balancing in computer networking is the division of network traffic between two or more network devices or paths, typically for the purpose of achieving higher total throughput than either one path, ensuring a specific maximum latency or minimum bandwidth to some or all flows, or similar purposes. For this exercise, you will design a load-balancing OpenFlow controller capable of collecting external data and using it to divide traffic between dissimilar network paths so as to achieve full bandwidth utilization with minimal queuing delays.

An interesting property of removing the controller from an OpenFlow device and placing it in an external system of arbitrary computing power and storage capability is that decision-making for network flows based on external state becomes reasonable. Traditional routing and switching devices make flow decisions based largely on local data (or perhaps data from adjacent network devices), but an OpenFlow controller can collect data from servers, network devices, or any other convenient source, and use this data to direct incoming flows.

For the purpose of this exercise, data collection will be limited to the *bandwidth* and *queue* occupancy of two emulated network links.

### Experimental Setup

Use the supplied `lb.rspec` to instantiate a sliver, via either Omni or Flack. Your GENI resources will be configured in a manner similar to Fig. 2. The various parts of the diagram are as follows:

- **Inside and Outside Nodes:** These nodes can be any exclusive ProtoGENI PCs.

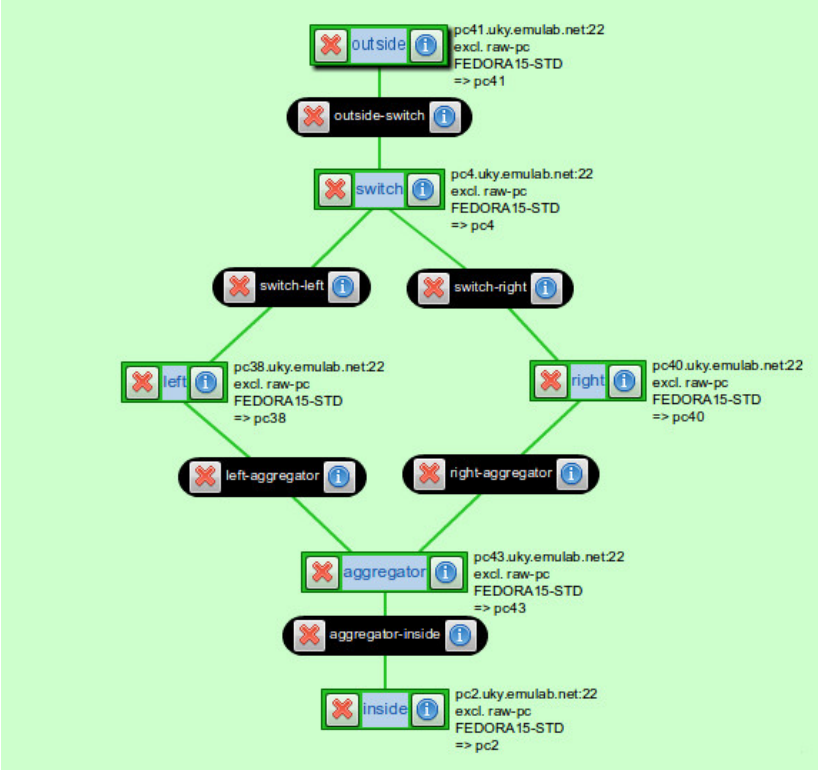


Figure 2: Example load balancing topology

- **Switch:** The role of the Open vSwitch node may be played either by a software Open vSwitch installation on a ProtoGENI node, or by the OpenFlow switches available in GENI — consult your instructor.

- **Traffic Shaping Nodes (Left and Right):** These are Linux hosts with two network interfaces. You can configure `netem` on the two traffic shaping nodes to have differing characteristics; the specific values don't matter, as long as they are reasonable. (No slower than a few hundred kbps, no faster than a few tens of Mbps with 0-100 ms of delay would be a good guideline.) Use several different bandwidth combinations as you test your load balancer.

- **Aggregator:** This node is a Linux host running Open vSwitch with a switch controller that will cause TCP connections to "follow" the decisions made by your OpenFlow controller on the Switch node.

**Linux netem** Use the `tc` command to enable and configure delay and bandwidth constraints on the *outgoing* interfaces for traffic traveling from the OpenFlow switch to the Aggregator node. To configure a path with 20 Mbps bandwidth and a 20 ms delay on `eth2`, you would issue the command:

```
sudo tc qdisc add dev eth2 root handle 1:0 \
netem delay 20ms
sudo tc qdisc add dev eth2 parent 1:0 \
tbf rate 20mbit buffer 20000 limit 16000
```

See the *tc*(8) and *tc-tbf*(8) manual pages for more information on configuring `tc` token bucket filters as in the second command line. Use the `tc qdisc change` command to reconfigure existing links, instead of `tc qdisc add`.

The outgoing links in the provided `lb.rspec` are numbered 192.168.4.1 and 192.168.5.1 for `left` and `right`, respectively.

### Balancing the Load

The goal of your OpenFlow controller will be to achieve *full bandwidth utilization* with *minimal queuing delays* of the two links between the OpenFlow switch and the Aggregator host. In order to accomplish this, your OpenFlow switch will intelligently divide TCP flows between the two paths. The intelligence for this decision will come from bandwidth and queuing status reports from the two traffic shaping nodes representing the alternate paths.

When the network is lightly loaded, flows may be directed toward either path, as neither path exhibits queuing delays and both paths are largely unloaded. As network load increases, however, your controller should direct flows toward the least loaded fork in the path, as defined by occupied bandwidth for links that are not yet near capacity and queue depth for links that are near capacity.

Because TCP traffic is bursty and unpredictable, your controller will not be able to perfectly balance the flows between these links. However, as more TCP flows are combined on the links, their combined congestion control behaviors will allow you to utilize the links to near capacity, with queuing delays that are roughly balanced. Your controller need not re-balance flows that have previously been assigned, but you may do so if you like.

The binding of OpenFlow port numbers to logical topology links can be found in the file `/tmp/portmap` on the switch node when the provided RSpec boots. It consists of three lines, each containing one logical link name (*left*, *right*, and *outside*) and an integer indicating the port on which the corresponding link is connected. You may use this information in your controller configuration if it is

helpful.

You will find an example OpenFlow controller that arbitrarily assigns incoming TCP connections to alternating paths in the assignment materials in the file `load-balancer.rb`. This simple controller can be used as a starting point for your controller if you desire. Examining its behavior may also prove instructive; you should see that its effectiveness at achieving the assignment goals falls off as the imbalance between balanced link capacities or delays grows.

### Gathering Information

The information you will use to inform your OpenFlow controller about the state of the two load-balanced paths will be gathered from the traffic shaping hosts. This information can be parsed out of the file `/proc/net/dev`, which contains a line for each interface on the machine, as well as the `tc -p qdisc show` command, which displays the number of packets in the token bucket queue. As TCP connections take some time to converge on a stable bandwidth utilization, you may want to collect these statistics once every few seconds, and smooth the values you receive over the intervening time periods.

You may find the file `/tmp/ifmap` on the traffic shaping nodes useful. It is created at system startup, and identifies the inside- and outside-facing interfaces with lines such as:

```
inside eth2
outside eth1
```

The first word on the line is the "direction" of the interface — toward the inside or outside of the network diagram in Fig. 2. The second is the interface name as found in `/proc/net/dev`.

You are free to communicate these network statistics from the traffic shaping nodes to your OpenFlow controller in any fashion you like. You may want to use a web service, or transfer the data via an external daemon and query a statistics file from the controller. Keep in mind that flow creation decisions need to be made rather quickly, to prevent retransmissions on the connecting host.

### Hints

- Remember that the TCP control loop is rather slow — on the order of several round trip times for the TCP connection. This means your load balancing control loop should be slow.

- You may wish to review control theory, as well as TCP congestion control and avoidance principles.
- Without rebalancing, "correcting" a severe imbalance may be difficult or impossible. For testing purposes, add flows to the path *slowly* and wait for things to stabilize.
- Some thoughts on reducing the flow count when load balancing via OpenFlow can be found in Wang et al. [12] You are not required to implement these techniques, but may find them helpful.
- Remember that the default OpenFlow policy for your switch or Open vSwitch instance will likely be to send any packets that do not match a flow spec to the controller, so you will have to handle or discard these packets.
- You will want your load balancer to communicate with the traffic shaping nodes via their administrative IP address, available in the slice manifest.
- If packet processing on the OpenFlow controller blocks for communication with the traffic shaping nodes, TCP performance may suffer. Use `require 'threads'`, `Thread`, and `Mutex` to fetch load information in a separate thread.
- The OpenFlow debugging hints from Section 3.1 remain relevant for this exercise.

# References


[1] Information Sciences Institute. Transmission control protocol. RFC 793. Available at `http://www.ietf.org/rfc/rfc793.txt`. Edited by Jon Postel.

[2] Iperf. `http://iperf.sourceforge.net/`.

[3] Net-SNMP. `http://net-snmp.sourceforge.net/`.

[4] Ruby SNMP. `http://snmplib.rubyforge.org/`.

[5] The Geni Project Office Wiki. Omni. `http://trac.gpolab.bbn.com/gcf/wiki/Omni`.

[6] The GENI wiki. http://groups.geni.net/geni.

[7] The Open Networking Foundation. OpenFlow. http://www.openflow.org/.

[8] The Open Networking Foundation. OpenFlow switch specification: Version 1.0.0 (wire protocol 0x01). http://www.openflow.org/documents/openflow-spec-v1.0.0.pdf.

[9] The Open Networking Foundation. OpenFlow switch specification: Version 1.1.0 implemented (wire protocol 0x02). http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf.

[10] The ProtoGENI Project Wiki. ProtoGENI tutorial. http://www.protogeni.net/trac/protogeni/wiki/Tutorial.

[11] Trema: Full-Stack OpenFlow Framework in Ruby and C. http://trema.github.com/trema/.

[12] R. Wang, D. Butnariu, and J. Rexford. OpenFlow-based server load balancing gone wild. In *Proceedings of the Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE)*, Mar. 2011.