CMSC 417 Computer Networks **Spring 2016**

Project 5: Software-Defined Switching

Assigned: April 28 *Due: May 11th, 11:59:59 PM* **(no late submissions)**

1 Introduction

In this project you will write a program that talks to the OpenDaylight Software-Defined Networking controller, lets the OpenDaylight controller connect hosts at layer-2, and reads the forwarding statistics from the switches (via the controller) to discover and print the names of the switches (and ports) along the path carrying active traffic.

OpenDaylight will control a simulated network of OpenFlow-enabled switches created using Mininet.

You will interact with OpenDaylight via REST calls. This will enable you to query the controller for topology information and install OpenFlow rules in switches.

If you're curious about OpenFlow, you can read the OpenFlow 1.3.5 spec. Reading and understanding it shouldn't be required for this project.

1.1 Not using Vagrant

If you are not using Vagrant, there are several steps you need to make sure you follow to set things up in the same way.

- make sure your VM is configured with at least 2 GB of memory
- install mininet with some variant of `sudo apt-get install mininet`
- make sure to kill any copy of ovs-controller that might be listening on port 6633, which OpenDaylight needs, using some variant of `sudo killall ovs-controller`
- download and extract OpenDaylight Beryllium-SR1 from `https://nexus.opendaylight.org/content/repositories/public/org/opendaylight/integration/distribution-karaf/0.4.1-Beryllium-SR1/distribution-karaf-0.4.1-Beryllium-SR1.tar.gz`
- extract the provided `54-arphandler.xml` file to the directory `distribution-karaf-0.4.1-Beryllium-SR1/etc/opendaylight/karaf/`, creating the path if needed.

2 Mininet

Mininet is a tool for simulating OpenFlow networks. It can do a lot of different things, but for this project, we'll focus on using it to just create OpenFlow networks with switches and hosts so that OpenDaylight can discover and control them.

A relatively simple Mininet command you can use to play with is:

```
sudo mn --controller=remote,ip=127.0.0.1 --topo=linear,2
```

There are many other topologies that you can play with built into Mininet and you can create custom topologies using python as well. You can read more on the Mininet website.

If you're really interested, you can read about creating custom topologies, however you should be able to get by with testing on relatively simple topologies, e.g., `linear,<n>` and `tree,<n>` for relatively small `n`.

Another option for Mininet that is likely worth noting for this project is the `--mac` option which makes it so the MAC addresses of hosts and switches start at 00:00:00:00:00:01 and predictably correspond to their IDs in Mininet. *Note that while this is useful for debugging, you should not expect that MAC addresses or IP addresses will be allocated in this manner during tests.*

Once you have mininet running you can use the `pingall` command to check that OpenDaylight has established connectivity between the hosts. It typically takes 30 seconds or so after OpenDaylight has been running for things to work and the first time pingall "works", only the second ping will succeed.

After that, you can use `h1 ping h2` or `h2 ping h1` to generate traffic between the two hosts.

For example:

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> X
h2 -> h1
*** Results: 50% dropped (1/2 received)
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
mininet> h1 ping h2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.028 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.037 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.220 ms
...
```

3 OpenDaylight

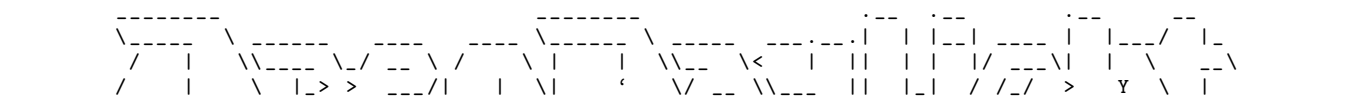
OpenDaylight is a Software-Defined Networking controller which supports controlling network devices using a variety of different protocols, but for this project we will focus on OpenFlow.

The provided Vagrant VM should automatically download and extract OpenDaylight for you including caching it across calls to `vagrant up` and `vagrant destroy`. It will be downloaded to a file called `odl-be-sr1.tar.gz`, which will be mounted in the vm at `/vagrant/odl-be-sr1.tar.gz` and automatically extracted into a `distribution-karaf-0.4.1-Beryllium-SR1` directory in the home folder of the VM.

You can then run OpenDaylight from the home directory like this:

```
./distribution-karaf-0.4.1-Beryllium-SR1/bin/karaf
```

Once it runs you should see something like[1]:



[1]Feel free to ignore warnings like `karaf: JAVA_HOME not set; results may vary`. They only apply if you have more than one version of Java installed.

```
    \_______  /   __/ \___  >___|  /_______  (____  / ____||____/__\___  /|___|  /__|
            \/|__|        \/     \/        \/     \/\/            /_____/      \/


Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown OpenDaylight.

opendaylight-user@root>
```

3.1 Installing the L2 Switch Application in OpenDaylight

For the first part of the project, you will want to enable the L2 Switch application with its REST APIs. This is done by installing the `odl-l2switch-switch-rest` *feature* that will provide basic layer-2 forwarding over the switches in Mininet. You can do that by typing the following at the OpenDaylight command prompt:

```
opendaylight-user@root> feature:install odl-l2switch-switch-rest
```

It might take a bit for the the feature to full install and the command prompt to come back. At the same time, run Mininet as described above so that the switches can connect to the controller.

3.2 Interacting with OpenDaylight via REST

OpenDaylight provides a REST interface (actually it's RESTCONF[2]) to read and write data to control the network.

As an example, here is a way to use `curl` to make a REST call to get the topology of the network from OpenDaylight. Whatever language you decide to use should include both tools to make REST calls (really just HTTP connections) and parse the JSON and/or XML that comes back. If you want XML instead of JSON as output, you can change the `Accept:` header to `application/xml` instead of `application/json`.

```
curl --user "admin":"admin" -H "Accept:  application/json"
http://127.0.0.1:8181/restconf/operational/network-topology:network-topology/
```

It's often useful to be able to pretty print the JSON output from REST call. Python provides a really simple one-line way to do that:

```
curl --user "admin":"admin" -H "Accept:  application/json"
http://127.0.0.1:8181/restconf/operational/network-topology:network-topology/
| python -m json.tool
```

The output should look something like

```
{
    "network-topology": {
        "topology": [
            {
                "link": [
                    {
                        "destination": {
                            "dest-node": "openflow:1",
                            "dest-tp": "openflow:1:2"
                        },
```

[2]You can read more about RESTCONF in its IETF specification if you're curious.

```
                        "link-id": "openflow:2:2",
                        "source": {
                            "source-node": "openflow:2",
                            "source-tp": "openflow:2:2"
                        }
                    },
                    {
                        "destination": {
                            "dest-node": "openflow:2",
                            "dest-tp": "openflow:2:2"
                        },
                        "link-id": "openflow:1:2",
                        "source": {
                            "source-node": "openflow:1",
                            "source-tp": "openflow:1:2"
                        }
                    }
                ],
                "node": [
                    {
                        "node-id": "openflow:2",
                        "opendaylight-topology-inventory:inventory-node-ref": ...,
                        "termination-point": [
                            {
                                "opendaylight-topology-inventory:inventory-node-connector-ref": ...,
                                "tp-id": "openflow:2:1"
                            },
                            {
                                "opendaylight-topology-inventory:inventory-node-connector-ref": ...,
                                "tp-id": "openflow:2:LOCAL"
                            },
                            {
                                "opendaylight-topology-inventory:inventory-node-connector-ref": ...,
                                "tp-id": "openflow:2:2"
                            }
                        ]
                    },
                    {
                        "node-id": "openflow:1",
                        "opendaylight-topology-inventory:inventory-node-ref": ...,
                        "termination-point": [
                            {
                                "opendaylight-topology-inventory:inventory-node-connector-ref": ...,
                                "tp-id": "openflow:1:1"
                            },
                            {
                                "opendaylight-topology-inventory:inventory-node-connector-ref": ...,
                                "tp-id": "openflow:1:2"
                            },
                            {
                                "opendaylight-topology-inventory:inventory-node-connector-ref": ...,
                                "tp-id": "openflow:1:LOCAL"
                            }
                        ]
                    }
                ],
                "topology-id": "flow:1"
            }
        ]
    }
}
```


You should note that the general structure for the topology consists of a list of *link*s and a list of *node*s. Each link consists of a *link-id*, *source* and *destination*. The source and destination include not only the switch, but the port (called a *termination point* or *tp*) on that switch. Each

node consists of a *node-id* and a list of *termination points*. The fields starting with *opendaylight-topology-inventory:* exist only to aid in mapping between topology data and inventory data. You may or may not find them useful.

In addition to the topology data, there is inventory data, which does not contain information about the links and connectivity between switches, but does contain information about the OpenFlow rules in each switch. You can access that in the same way as the topology, but using a different URL:

`http://127.0.0.1:8181/restconf/operational/opendaylight-inventory:nodes/`

When it comes to working with OpenFlow and the inventory model, there are two potentially useful tutorials. One covers how OpenFlow switches show up as nodes in the inventory and another covers how to program and read already-installed OpenFlow rules from switches.

4 Your Program

Your goal is to use the REST APIs of OpenDaylight to determine the path of active traffic in a simulated network where a single pair of hosts will be exchanging traffic. *Note that this will be complicated by the fact that OpenDaylight itself exchanges traffic between switches to keep an updated view of the topology.*

Your output will be directed to a file that will be passed as the first argument to your `run.sh` script. It should consist of four parts as described in the next four subsections. *If you follow the output guidelines, your output should be a valid YAML*[3] *file, but you shouldn't need to understand any of the details of YAML. You might find it useful to open the output using a YAML parser to verify it's format.*

With the exception of outputting the `Path:` part described in Section 4.4, the order of the elements within each part does not matter.

We will give your program 30 seconds to discover all of the relevant information and produce the output file.

4.1 Discovering Involved Switches

For this part, your program should discover the switches involved in carrying the active traffic and print them out like:

```
Switches:
  - openflow:2
  - openflow:1
```

Where the names of the switches are the *node-id*s from the REST calls to the topology.

4.2 Discovering Involved Ports

For this part, your program should discover the ports (or termination points) involved in carrying the active traffic and print them out like:

```
Ports:
  - openflow:2:1
```

[3] http://yaml.org/

```
  - openflow:2:2
  - openflow:1:1
  - openflow:1:2
```

Where the names of the ports are the *tp-id*s from the REST calls to the topology.

4.3 Discovering Involved Hosts

For this part, your program should discover the hosts exchanging traffic and print them out like:

```
Hosts:
  - 00:00:00:00:00:01
  - 00:00:00:00:00:02
```

Where the names of the hosts are the MAC addresses of the hosts.

4.4 Discovering The Path

For this part you should aggregate all of the information from each of the above parts to form a complete path through the network. Since the two hosts will be exchanging traffic symmetrically, there are two valid paths: one forward and one backward. For convenience, print the path starting with the host with the lower MAC address.

The list should be ordered, start and end with a host and have a series of one or more triples of port-switch-port in between. For example:

```
Path:
  - 00:00:00:00:00:01
  - openflow:1:1
  - openflow:1
  - openflow:1:2
  - openflow:2:1
  - openflow:2
  - openflow:2:2
  - 00:00:00:00:00:02
```

5 Solution Implementation

As long as you comply with the requirements in Section 6, you may write this project in a language of your choosing. We will be installing support for python, ruby, java, scala, C and C++ on the VM. If you feel that you need additional language support and the VM doesn't support it, please let us know and we may (but also may not) accommodate your request. Make sure to test that your code compiles and runs in the VM using `make` and `bash run.sh` so that our tests will also be able to compile and run it.

The URL to accept the assignment is:

https://ter.ps/biv

The process of adding the Java OpenJDK to the VM has made vagrant up take longer than before. Even with the trusty64 base box pre-cached, it can take 5–10 minutes or for it to come up.

Once you get it up, the current versions of ruby, python, java, gcc and g++ installed are as follows:

```
vagrant@vagrant-ubuntu-trusty-64: $ ruby -v
ruby 1.9.3p484 (2013-11-22 revision 43786) [x86_64-linux]
vagrant@vagrant-ubuntu-trusty-64: $ python --version
Python 2.7.6
vagrant@vagrant-ubuntu-trusty-64: $ javac -version
javac 1.7.0_95
vagrant@vagrant-ubuntu-trusty-64: $ gcc --version
gcc (Ubuntu 4.8.4-2ubuntu1 14.04.1) 4.8.4
vagrant@vagrant-ubuntu-trusty-64: $ g++ --version
g++ (Ubuntu 4.8.4-2ubuntu1 14.04.1) 4.8.4
vagrant@vagrant-ubuntu-trusty-64: $ scala -version
Scala code runner version 2.9.2 -- Copyright 2002-2011, LAMP/EPFL
```

6 Additional Requirements

1. Your code must be submitted as a series of commits that are pushed to the origin/master branch of your Git repository. We consider your latest commit prior to the due date/time to represent your submission.

2. You must provide a `Makefile` that is included along with the code that you commit. We will run `make` inside the root of your repository to compile your code. It is your responsibility to write a `Makefile` that compiles your code (if necessary) regardless of the language, e.g., call `javac` appropriately if you're using java. *If your code doesn't need to be compiled, e.g., python or ruby, you can use a makefile with a single line "skip:" that will do nothing when we call make in our tests.*

3. You must provide an executable `run.sh` shell script that calls your code. We will call `bash run.sh <file>`, so make sure it works with bash. *Note: a run.sh with the single line "<command> > $1" will call that command and pass through the first argument to redirect standard out to the specified file. This will allow you to just print to standard out rather than dealing with file I/O.*

4. You must submit code that compiles and runs in the provided VM, otherwise your assignment will not be graded.

5. You are not allowed to work in teams or to copy code from any source.