SDN101

Software Defined Networking and You by Timothy Serewicz

Version C-ALE

The C-ALE (Cloud & Container Apprentice Linux Engineer) is a series of seminars held at existing conferences covering topics which are fundamental to a Linux professional in the Linux Cloud and Container field of computing.

This seminar will spend equal time on lecture and hands on labs at the end of each seminar which allow you to practice the material you've learned.

> This material makes the assumption that you have minimal experience with using Linux in general, and a basic understanding of general industry terms. The assumption is also made that you have access to your own computers upon which to practice this material.

More information can be found at **https://c-ale.org/**

Contents

Chapter 1

Software Defined Networking

1.1 Why Software Defined Networking

Software Defined Networking and You

- What is Software Defined Networking?
- Why does SDN matter?
- Basic Functionality
- Common Solutions
- Future trends

Why Should I Care?

- Flexibility
- Speed of configuration
- Money

Flexibility

- Multiple hardware vendors
- Whole network view
- Wire once, deploy many
- Open APIs

Speed of Configuration

- Rapid provisioning
- Meet needs of data center virtualization
- Container Orchestration

Lower Cost

- Operational cost reduction
- Security costs
- Migration away from non-SDN solutions

1.2 Software Defined Networking Explained

What is Software Defined Networking?

- Decouple the control plane
- Centralize management and reporting
- Programmable network infrastructure

Decoupled Control Plane

- Packet handling rules
- Locality of data
- Greater flexibility

Centralize Management and Reporting

- Whole network control
- Virtual view of complete network
- Consolidate multiple products

Programmable Network Infrastructure

- Optimize the data center
- Leverage Network Function Virtualization
- Easily detect and respond to issues

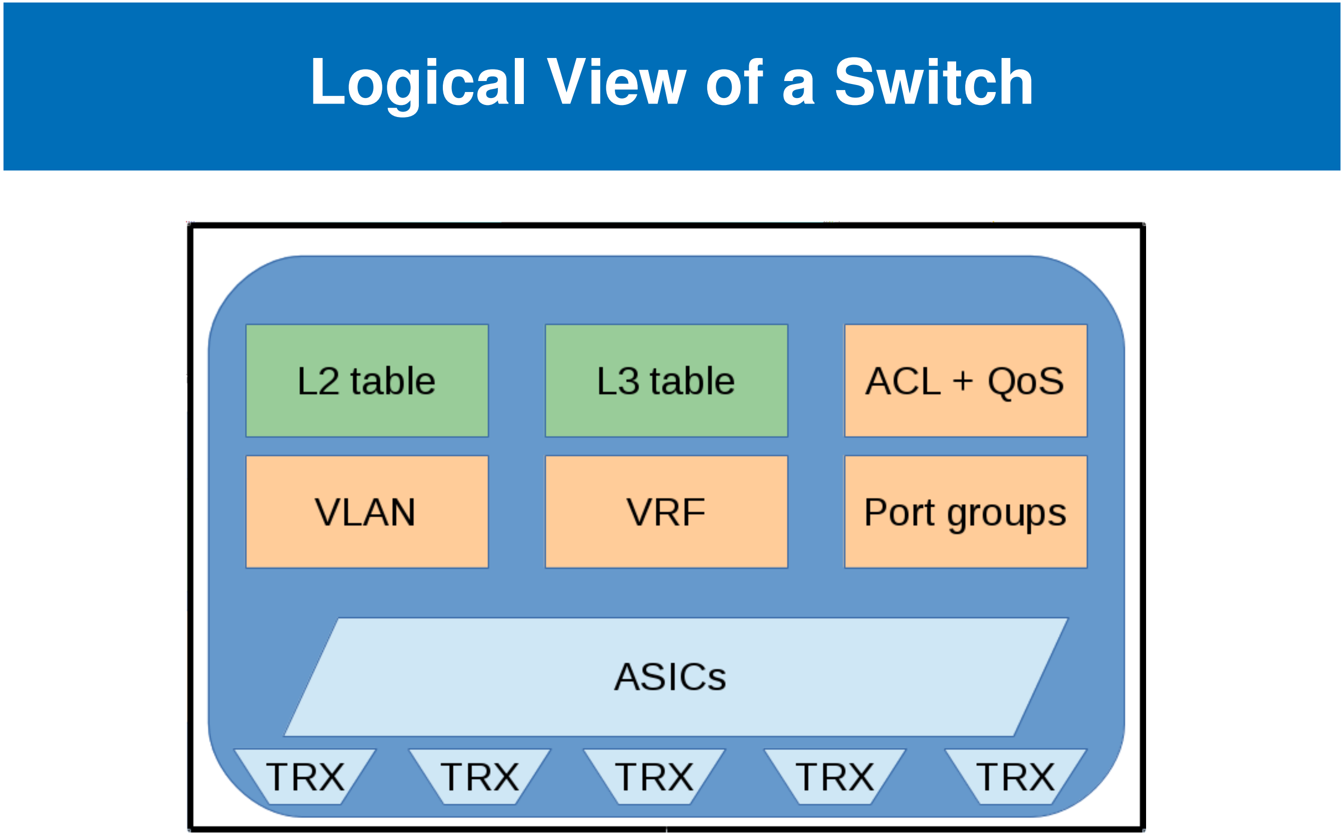


Figure 1.1: **Logical view of a switch**

Control, Management, and Forwarding Planes

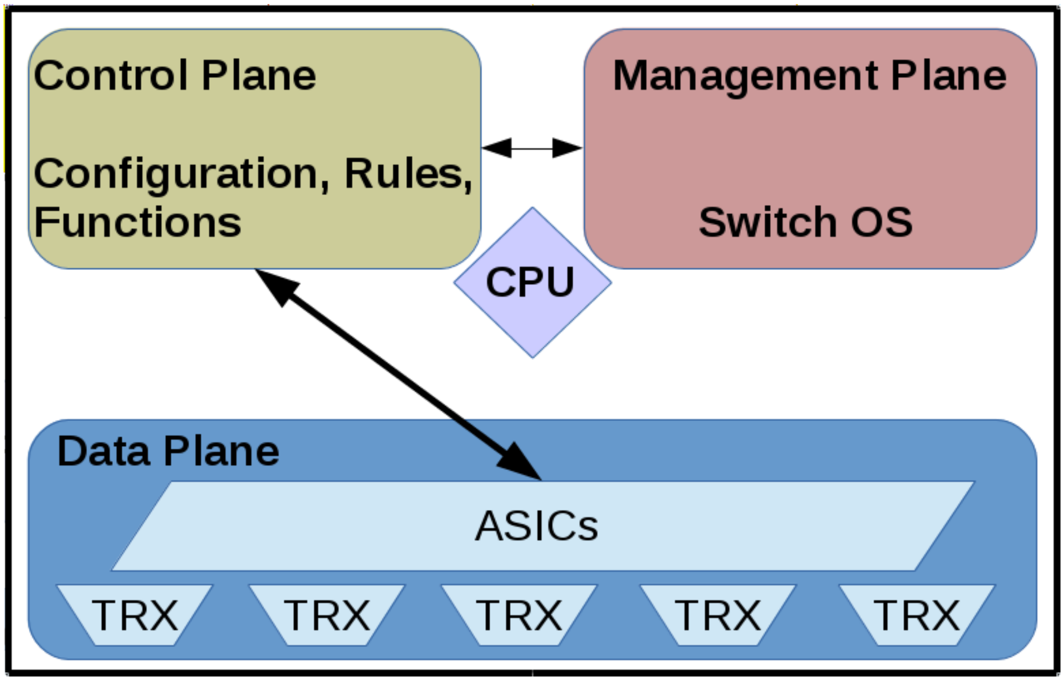


Figure 1.2: **Relationship between switch planes**

New Packet Steps

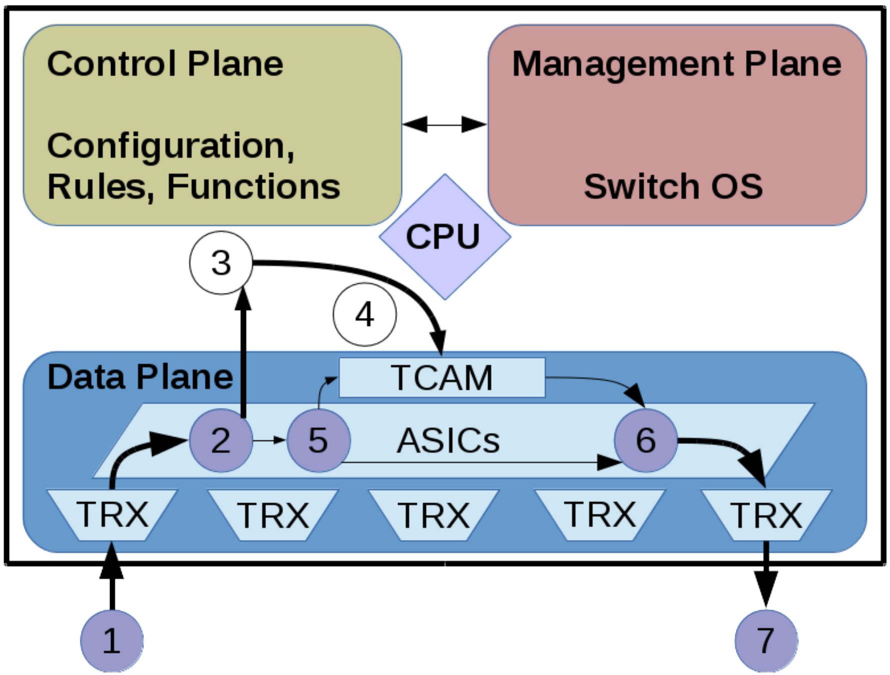


Figure 1.3: **Detailed Steps for a new packet**

Known Packet Steps

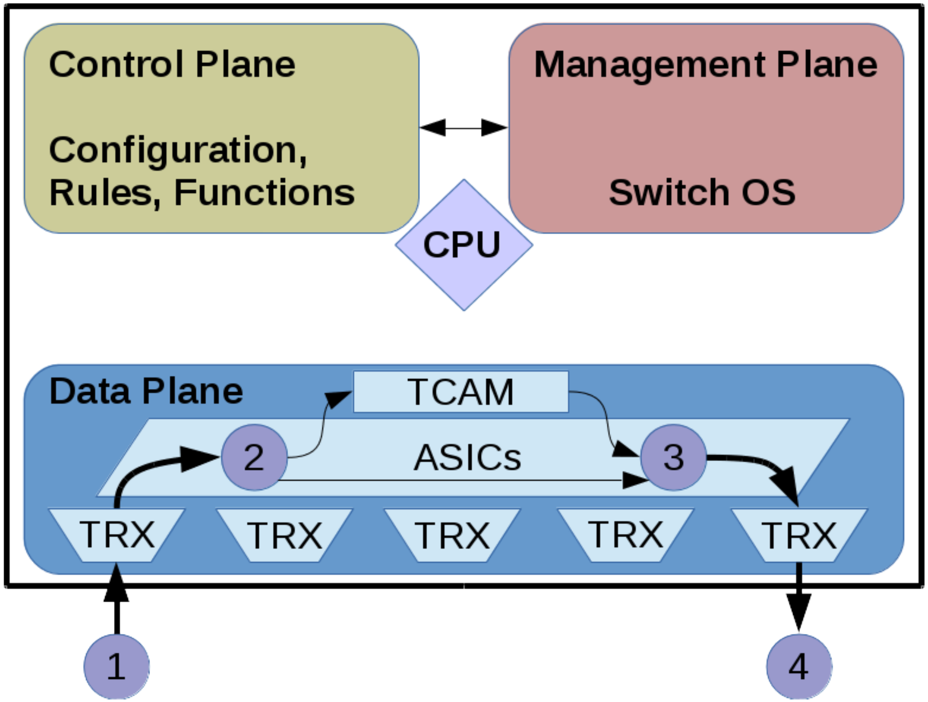


Figure 1.4

Updating Rules

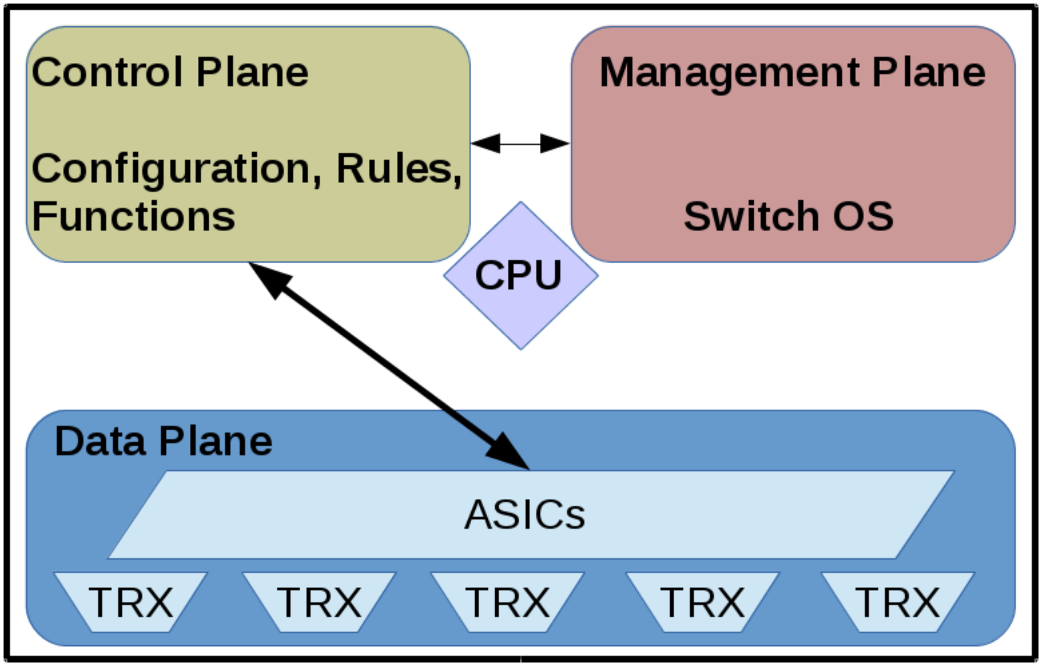


Figure 1.5

Abstraction of Control Plane

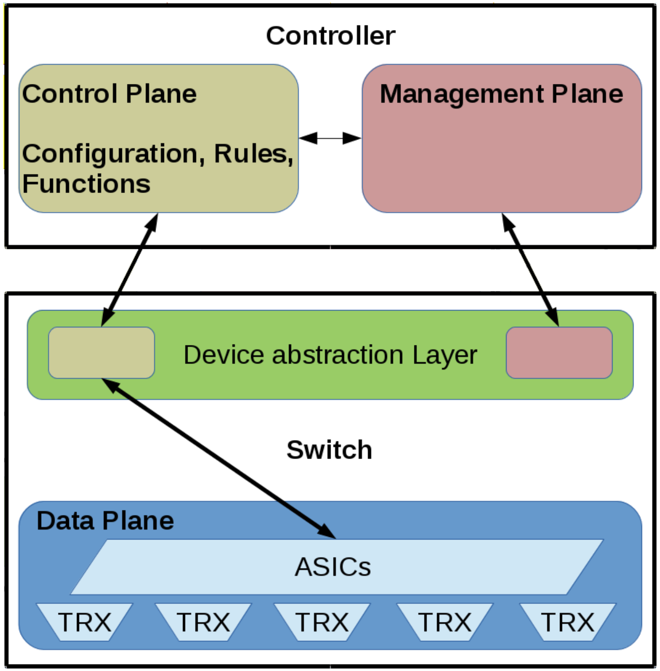


Figure 1.6: **Use DAL for remote information**

Whole Network Monitoring and Control

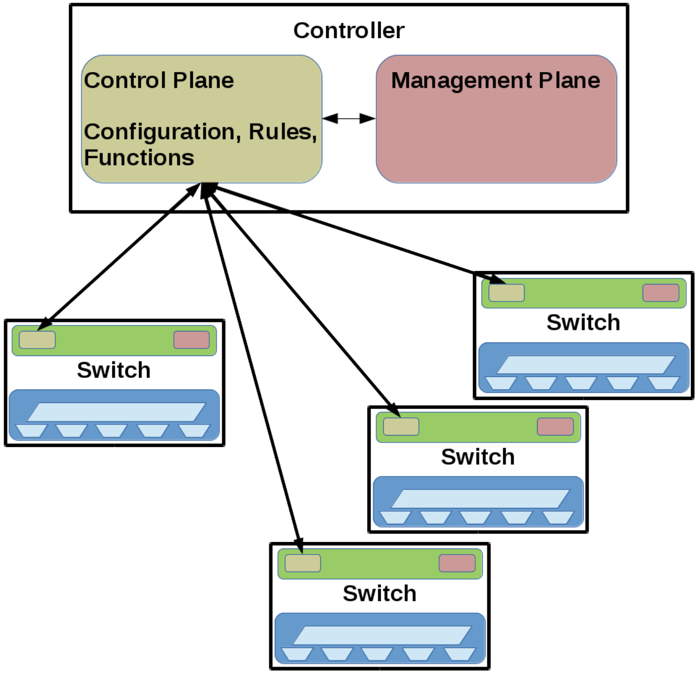
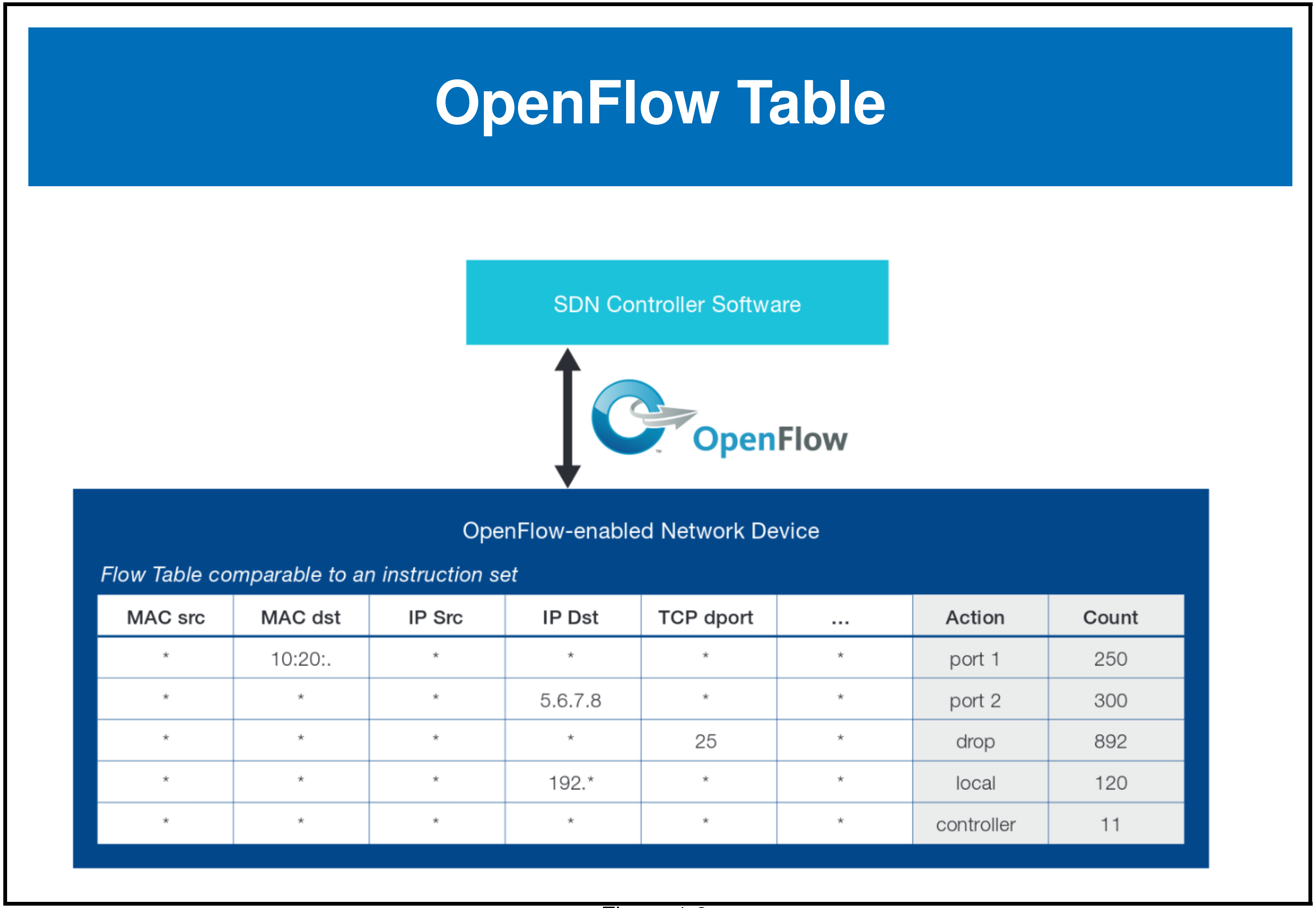


Figure 1.7: **Control and Monitor Whole Network**

1.3 OpenFlow

OpenFlow

- Specifications from **Open Networking Forum**
- Message layer
- State Machine
- System Interface
- Configuration
- Not just the destination IP or MAC
- Multiple tables may be referenced



MAC src	MAC dst	IP Src	IP Dst	TCP dport	...	Action	Count
*	10:20:.	*	*	*	*	port 1	250
*	*	*	5.6.7.8	*	*	port 2	300
*	*	*	*	25	*	drop	892
*	*	*	192.*	*	*	local	120
*	*	*	*	*	*	controller	11

Figure 1.8

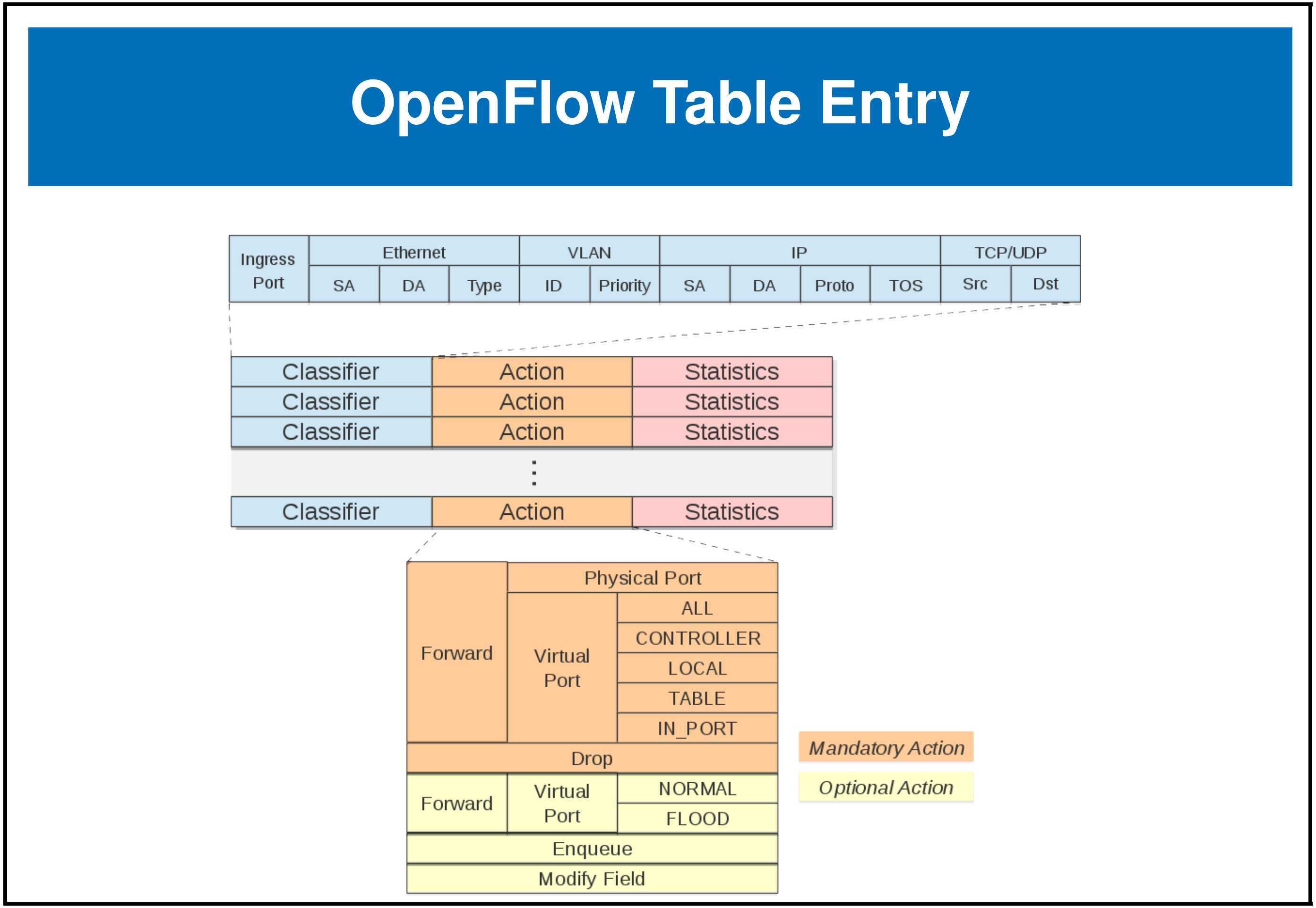

OpenFlow Table Entry
Ingress Port
Ethernet
SA
DA
Type
VLAN
ID
Priority
IP
SA
DA
Proto
TOS
TCP/UDP
Src
Dst
Classifier
Action
Statistics
Classifier
Action
Statistics
Classifier
Action
Statistics
Classifier
Action
Statistics
Forward
Physical Port
Virtual Port
ALL
CONTROLLER
LOCAL
TABLE
IN_PORT
Drop
Forward
Virtual Port
NORMAL
FLOOD
Enqueue
Modify Field
Mandatory Action
Optional Action

Figure 1.9

Open vSwitch

- Common in-memory switch
- Fully functional, focused on automated and dynamic network control
- Security, monitoring, QoS, and network protocol interaction of vendor-provided switches

A Simple Switch and Hosts

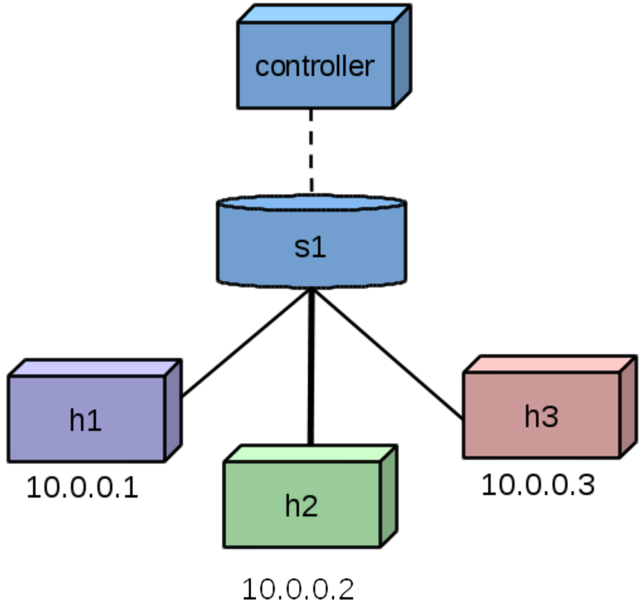


Figure 1.10

1.4 OpenDaylight

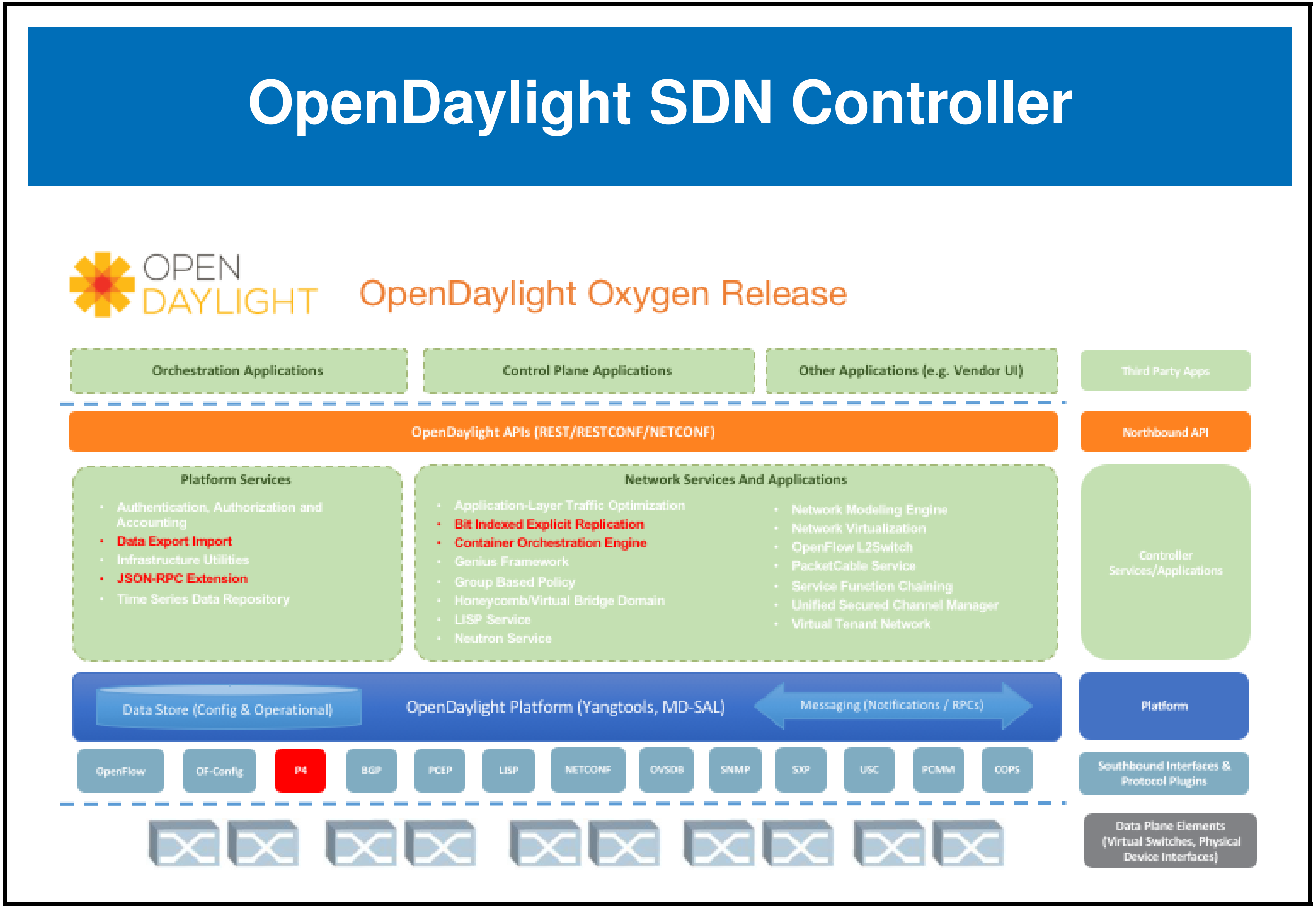


Figure 1.11: OpenDaylight Stack

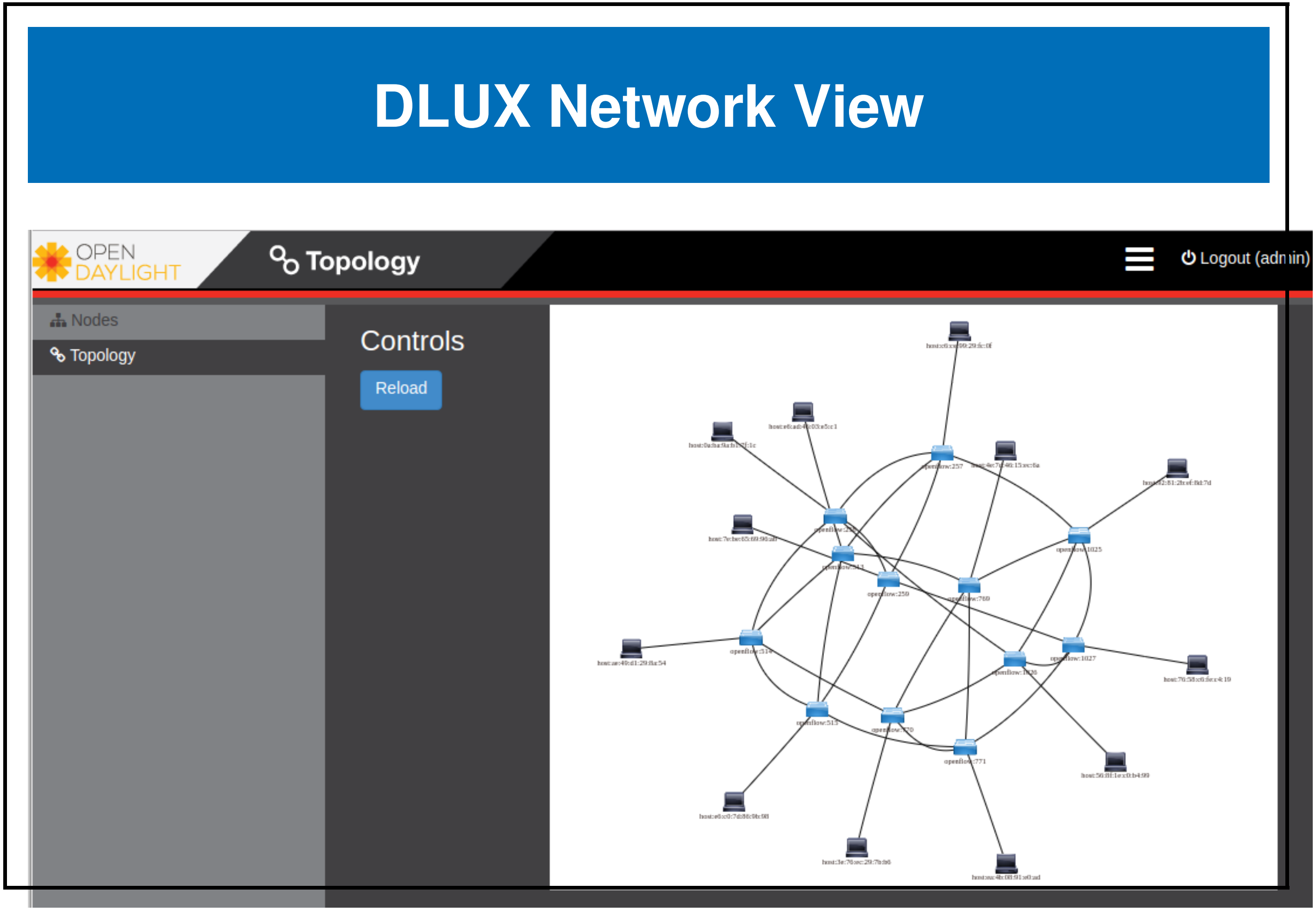


Figure 1.12: **OpenDaylight DLUX Torus**

1.5 Open Source SDN Options

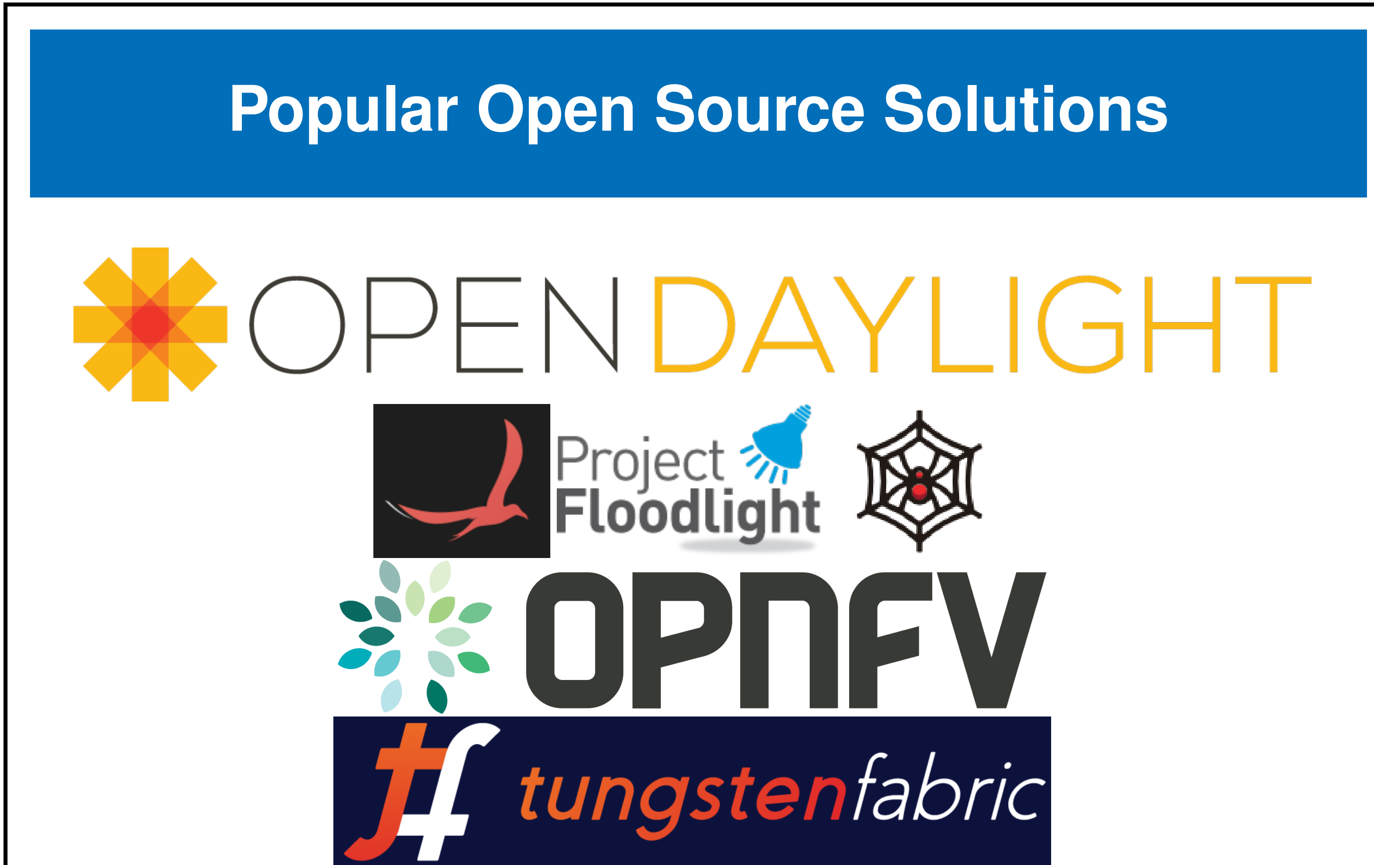


Figure 1.13: **Official Project Logos**

1.6 Future Trends

The Future of Open Source SDN

- More collaboration
- Deeper reach into traditional network spaces
- Open source and customer driven solutions

1.7 Labs

The following lab was completed using an AWS Ubuntu 16.04 m5.large instance. It had 2vCPU and 8GB of memory. The **OpenDaylight** controller often will not start properly if the instance does not have at least 8GB. The errors often show in red in the controller output and will detail the inability of **Java** to complete a task. I have also removed a firewall as it can add to the complexity in understanding if a network operation fails because of configuration or being blocked by an outside agent.

The steps should be very similar if you choose a different **Linux** operating system. Some changes will be necessary, the use of **yum** or **dnf** instead of **apt** for example. There are several command used in the labs which have lots of sub-commands, which would be good to investigate after completing the labs.

The use of multiple connections to the node can be helpful. This will allow you to view output immediately in a different terminal from the one executing the command.

Exercise 1.1: Deploy A Switch Using OVS and Mininet

We will start by using **Mininet** and **Open vSwitch**, which are both easy to deploy and configure.

1. Begin by updating the node software. While it is a better practice to execute each command via **sudo**, for simplicity we will become and remain root for the exercise.

```
node-term1$ sudo -i

node-term1# apt-get update
<output_omitted>
Get:37 http://security.ubuntu.com/ubuntu xenial-security/multiverse amd64 Packages [3,456 B]
Get:38 http://security.ubuntu.com/ubuntu xenial-security/multiverse Translation-en [1,744 B]
Fetched 25.5 MB in 4s (5,980 kB/s)
Reading package lists... Done
```

2. There are several packages we will need to install for this and following labs. We will install them all at once. Depending on how you built your instance you may need to manage your repositories and install several dependencies.

   During the installation you may see a pop-up window asking if `Configuring wireshark-common` should allow non-superusers to capture packets. Please use the arrow keys and select **yes**.


```
node-term1# apt-get install -y mininet wireshark default-jdk \
    openvswitch-common openvswitch-testcontroller
<output_omitted>
```

3. Use the **Mininet** utility to create a switch. Note that it is unable to find a default **OpenFlow** controller and leverages an **OVS** bridge instead. Use **exit** to return to a system prompt. The default topology creates a controller, a switch and two hosts. Similar to a graphic from the chapter.

4. Run the **Mininet** utility again with greater verbosity.

```
node-term1# mn
*** No default OpenFlow controller found for default switch!
*** Falling back to OVS Bridge
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1)
*** Configuring hosts
h1 h2
*** Starting controller

*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet> exit
```

5. If you encounter a problem you can pass an option for greater verbosity, via the **–verbosity=debug** command. This time you can see that the **which** command is unable to find the program among the three tried. Exit back to the shell when done working through the output.

```
node-term1# mn --verbosity=debug
*** errRun: ['which', 'controller']
  1*** errRun: ['which', 'ovs-controller']
  1*** errRun: ['which', 'test-controller']
  1*** No default OpenFlow controller found for default switch!
*** Falling back to OVS Bridge
<output_omitted>
```

6. We installed the controller in a previous step, but the name of the program in **Ubuntu** does not match the name searched by **Mininet**. Create a symbolic link so that the controller can be called by **Mininet**.

```
node-term1# ln -s /usr/bin/ovs-testcontroller /usr/bin/ovs-controller
```

7. Run the **Mininet** utility again. This time you the controller should be found. Exit back to the node prompt when done.

```
node-term1# mn
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2
<output_omitted>
```

8. Use the `debug` option again. As you slowly work through the output you will see an attempt to connect to the controller via telnet which fails. Further along you will find a series of **ovs-vsctl** commands to create the controller, switches, interface, and other components. Note that the controller is running

```
node-term1# mn --verbosity=debug
*** errRun: ['which', 'controller']
  1*** errRun: ['which', 'ovs-controller']
/usr/bin/ovs-controller
  0*** errRun: ['grep', '-c', 'processor', '/proc/cpuinfo']

<output_omitted>

added intf lo (0) to node s1
```

```
*** s1 : ('ifconfig', 'lo', 'up')
s1
*** Adding links:
*** h1 : ('ip link add name h1-eth0 address 06:b5:9b:d5:a7:12 type veth peer name s1-eth1 address 56:12:e4:cb:3c:

<output_omitted>

*** Starting 1 switches
s1 ...*** errRun: ovs-vsctl -- --id=@s1c0 create Controller target=\"tcp:127.0.0.1:6633\"
max_backoff=1000 -- --id=@s1-listen create Controller target=\"ptcp:6634\" max_backoff=1000

<output_omitted>
```

9. In a second terminal session view the active connections using the **netstat** command.

```
node-term2# netstat -tulpn
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address   Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:6653    0.0.0.0:*               LISTEN      10459/ovs-testcontr
tcp        0      0 0.0.0.0:6633    0.0.0.0:*               LISTEN      12518/ovs-controlle
tcp        0      0 0.0.0.0:6634    0.0.0.0:*               LISTEN      10292/ovs-vswitchd
tcp        0      0 0.0.0.0:22      0.0.0.0:*               LISTEN      1262/sshd
tcp6       0      0 :::22           :::*                    LISTEN      1262/sshd
udp        0      0 0.0.0.0:68      0.0.0.0:*                           896/dhclient
```

10. Return to the terminal running **Mininet** and look at available commands. From the list then run **dump** to view current configuration information. Note the IP and port in use by `OVSController c0` which defaults to 127.0.0.1:6633.

```
mininet> help

Documented commands (type help <topic>):
========================================
EOF    gterm  iperfudp  nodes        pingpair      py      switch
dpctl  help   link      noecho       pingpairfull  quit    time
```

```
dump   intfs  links     pingall      ports        sh      x
exit   iperf  net       pingallfull  px           source  xterm
<output_omitted>


mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=12457>
<Host h2: h2-eth0:10.0.0.2 pid=12460>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None pid=12466>
<OVSController c0: 127.0.0.1:6633 pid=12450>
```

11. Return to the second terminal session and use the **ovs-vsctl show** command to view switch information from the **OVS** perspective. Note there are two lines with controller information. One uses parallel TCP to port 6634 and the other to 6633. A view of current processes show two separate processes are running. Also note the bridge for this configuration is `s1`.

```
node-term2# ovs-vsctl show
18f7d986-fdc0-43e9-8fcc-dbb699a18b5f
    Bridge "s1"
        Controller "ptcp:6634"
        Controller "tcp:127.0.0.1:6633"
            is_connected: true
        fail_mode: secure
        Port "s1-eth1"
            Interface "s1-eth1"
        Port "s1"
            Interface "s1"
                type: internal
        Port "s1-eth2"
            Interface "s1-eth2"
    ovs_version: "2.5.4"
```

12. View the **OpenFlow** information of the `s1` switch. We can see the capabilities of the **OVS** switch as well as port

information.

```
node-term2#  ovs-ofctl show s1
OFPT_FEATURES_REPLY (xid=0x2): dpid:0000000000000001
n_tables:254, n_buffers:256
capabilities: FLOW_STATS TABLE_STATS PORT_STATS QUEUE_STATS ARP_MATCH_IP
actions: output enqueue set_vlan_vid set_vlan_pcp strip_vlan mod_dl_src
mod_dl_dst mod_nw_src mod_nw_dst mod_nw_tos mod_tp_src mod_tp_dst
 1(s1-eth1): addr:56:12:e4:cb:3c:24
     config:     0
     state:      0
     current:    10GB-FD COPPER
     speed: 10000 Mbps now, 0 Mbps max
 2(s1-eth2): addr:32:d0:20:f7:71:f7
<output_omitted>
```

13. View the current flow tables. As we have not yet done anything with the environment there should only be a `CONTROLLER` entry.

```
node-term2# ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=2198.442s, table=0, n_packets=13, n_bytes=1026,
idle_age=2188, priority=0 actions=CONTROLLER:128
```

14. Now use the **snoop** command to see the current activity of the switch. We should see some OFPT echo request and replies. Also note the version of **OpenFlow** is version 1.3. Leave the command running, and capturing the window, and return to the other terminal window. Eventually you can use **ctrl-c** to quit the snoop.

```
node-term2# ovs-ofctl snoop s1
OFPT_ECHO_REQUEST (OF1.3) (xid=0x0): 0 bytes of payload
OFPT_ECHO_REPLY (OF1.3) (xid=0x0): 0 bytes of payload
```

15. Return to the terminal running **Mininet** and use the **pingall** command to cause all hosts to ping all other hosts.


```
mininet> pingall
*** Ping: testing ping reachability
h1 -> *** h1 : ('ping -c1  10.0.0.2',)
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=3.70 ms
<output_omitted>
```

16. Return to the terminal which continues to run the snoop. You should see new traffic which starts with a `OFPT_PACKET_IN` statement and then shows a series of `OFPT_FLOW_MOD` statements which show an **ADD** to the flow table.

```
<output_omitted>
OFPT_PACKET_IN (OF1.3) (xid=0x0): cookie=0x0 total_len=42 in_port=1
(via no_match) data_len=42 buffer=0x0000010d
arp,vlan_tci=0x0000,dl_src=06:b5:9b:d5:a7:12,dl_dst=ff:ff:ff:ff:ff:ff
,arp_spa=10.0.0.1,arp_tpa=10.0.0.2,arp_op=1,arp_sha=06:b5:9b:d5:a7:12,
arp_tha=00:00:00:00:00:00
OFPT_PACKET_OUT (OF1.3) (xid=0x12): in_port=1 actions=FLOOD buffer=0x0000010d
OFPT_PACKET_IN (OF1.3) (xid=0x0): cookie=0x0 total_len=42 in_port=2 (via no_match)
 data_len=42 buffer=0x0000010e
arp,vlan_tci=0x0000,dl_src=52:ac:d2:a2:e1:d7,dl_dst=06:b5:9b:d5:a7:12,arp_spa=10.0.0.2,
arp_tpa=10.0.0.1,arp_op=2,arp_sha=52:ac:d2:a2:e1:d7,arp_tha=06:b5:9b:d5:a7:12
OFPT_FLOW_MOD (OF1.3) (xid=0x13): ADD priority=1,arp,in_port=2,vlan_tci=0x0000/0x1fff,dl_src=52:ac:d2:a2:e1:d7,dl_dst
06:b5:9b:d5:a7:12,arp_spa=10.0.0.2,arp_tpa=10.0.0.1,arp_op=2 idle:60 buf:0x10e
actions=output:1
<output_omitted>
```

17. Interrupt the snoop using **ctrl-c**. Look at the current flow tables for the switch again. Note that the entries are removed when they become stale. If you don't see any new rules, return to the **Mininet** terminal and run the **pingall** again. You should see the rules then. The rule we saw before is the last among several. Each new rule should match one of the **ADD** statements we saw in the snoop. Note the differences between `in_port`, `dl_src` and `output` parts of each line.

```
node-term2# ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=2.327s, table=0, n_packets=1, n_bytes=98, idle_timeout=60, idle_age=2, priority=1,icmp,in_port=
```

```
 cookie=0x0, duration=2.326s, table=0, n_packets=1, n_bytes=98, idle_timeout=60, idle_age=2, priority=1,icmp,in_po
 cookie=0x0, duration=2.324s, table=0, n_packets=1, n_bytes=98, idle_timeout=60, idle_age=2, priority=1,icmp,in_po
 cookie=0x0, duration=2.324s, table=0, n_packets=1, n_bytes=98, idle_timeout=60, idle_age=2, priority=1,icmp,in_po
 cookie=0x0, duration=2660.668s, table=0, n_packets=25, n_bytes=1978, idle_age=2, priority=0 actions=CONTROLLER:12
```

18. Use the above output and compare to the output of **ovs-ofctl show s1** and **ovs-vsctl show**. You should find the ports and MAC addresses align with the rules added to the switch.

19. Wait for a couple of minutes. Check the flow table again. Only the controller should be found.

```
node-term2#  ovs-ofctl dump-flows s1
NXST_FLOW reply (xid=0x4):
 cookie=0x0, duration=3297.640s, table=0, n_packets=27, n_bytes=2062,
idle_age=634, priority=0 actions=CONTROLLER:128
```

20. Return to the **Mininet** terminal and shut down the switch and hosts by exiting.

```
mininet> exit
*** Stopping 1 controllers
c0 *** c0 : ('kill %ovs-controller',)
*** c0 : ('wait %ovs-controller',)

*** Stopping 2 links
.*** h1 : ('ip link del h1-eth0',)
.*** h2 : ('ip link del h2-eth0',)

*** Stopping 1 switches
*** errRun: ['ovs-vsctl', '--if-exists', 'del-br', 's1']
  0*** errRun: ['kill', '-HUP', '12466']
  0s1
*** Stopping 2 hosts
h1 h2
*** Done
completed in 348.787 seconds
node-term1#
```

Exercise 1.2: Install OpenDaylight Controller In this exercise we will deploy the **OpenDaylight** SDN controller. While this is a basic exercise you can view full documentation and learn about all of its features by visiting the opendaylight.org website. The software downloaded is about 340MB in size and may take a while depending on your network speed.

1. Become root, if not already.

```
node-term1$ sudo -i
node-term1#
```

2. Download the software. In this case as a compressed tar file. Both a short URL and the long URL have been included. To download an older version you can visit the main product website. The longer URL is: https://nexus.opendaylight.org/content/repositories/public/org/opendaylight/integration/karaf/0.8.3/karaf-0.8.3.tar.gz

```
node-term1#  wget https://tinyurl.com/yaujslvx -O karaf-0.8.3.tar.gz
<output_omitted>
karaf-0.8.3.tar.gz 100%[===================================>] 336.81M  34.4MB/s    in 10s

2018-08-26 21:25:24 (33.2 MB/s) - karaf-0.8.3.tar.gz saved [353170921/353170921]
```

3. Use the **tar** command to extract the tarball.

```
node-term1# tar -xf karaf-0.8.3.tar.gz
```

4. Change into the new directory. Look at the files and directories available.

```
node-term1# cd karaf-0.8.3/


node-term1# ls -l
total 56
drwxr-xr-x  3 root root  4096 Aug  8 02:59 bin
-rw-r--r--  1 root root    76 Aug  8 02:59 build.url
drwxr-xr-x  2 root root  4096 Aug  8 02:59 configuration
-rw-r--r--  1 root root  1126 Aug  8 02:59 CONTRIBUTING.markdown
```

```
drwxr-xr-x  3 root root  4096 Aug  8 02:59 data
drwxr-xr-x  2 root root  4096 Aug  8 02:59 deploy
drwxr-xr-x  3 root root  4096 Aug  8 02:59 etc
drwxr-xr-x  5 root root  4096 Aug  8 02:59 lib
-rw-r--r--  1 root root 11266 Aug  8 02:59 LICENSE
-rw-r--r--  1 root root   172 Aug  8 02:59 README.markdown
drwxr-xr-x 25 root root  4096 Aug  8 02:59 system
-rw-r--r--  1 root root  1987 Aug  8 02:59 taglist.log
```

5. Take a closer look at the files in the etc/ subdirectory.

```
node-term1# ls etc/
2c92bff6-6022-4058-97d9-a1edc82fc8d8.xml   org.apache.karaf.command.acl.jaas.cfg
all.policy                                 org.apache.karaf.command.acl.kar.cfg
config.properties                          org.apache.karaf.command.acl.scope_bundle.cfg
custom.properties                          org.apache.karaf.command.acl.shell.cfg
distribution.info                          org.apache.karaf.command.acl.system.cfg
equinox-debug.properties                   org.apache.karaf.features.cfg
java.util.logging.properties               org.apache.karaf.features.repos.cfg
jetty.xml                                  org.apache.karaf.jaas.cfg
<output_omitted>
```

6. Look through the etc/jetty.xml command. Around line 86 you should see a stanza which configures the http-default settings. Among the settings after you will find jetty.port set to 8181

```
node-term1# less etc/jetty.xml
<output_omitted>
                    <Property name="jetty.host"/>
                </Set>
                <Set name="port">
                    <Property name="jetty.port" default="8181"/>
                </Set>
<output_omitted>
```

7. While we installed JAVA in the earlier lab we also need to set the JAVA_HOME parameter. You may want to make this a persistent setting as well.

```
node-term1# export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64/
```

8. Start the **OpenDaylight** controller. This may take a bit to fully start. Remember if you don't have enough memory it may still appear to function but will be slow and/or not function properly.

```
node-term1# ./bin/karaf
Apache Karaf starting up. Press Enter to open the shell now...
100% [========================================================================]
Karaf started in 1s. Bundle stats: 54 active, 55 total

    ________                       ________                .__  .__       .__     __
    \_____  \ ______   ____   ____ \______ \ _____  ___.__.|  | |__| ____ |  |___/  |_
     /   |   \\____ \_/ __ \ /    \ |    |  \\__  \<   |  ||  | |  |/ ___\|  |  \   __\
    /    |    \  |_> >  ___/|   |  \|    `   \/ __ \\___  ||  |_|  / /_/  >   Y  \  |
    \_______  /   __/ \___  >___|  /_______  (____  / ____||____/__\___  /|___|  /__|
            \/|__|        \/     \/        \/     \/\/            /_____/      \/


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>
```

9. You now have a basic controller installed. With a modular approach we can choose to add features. Be aware you may get a prompt back before the feature has been fully installed and able to respond to requests.

```
opendaylight-user@root>feature:install odl-restconf odl-l2switch-switch \
  odl-mdsal-apidocs odl-dlux-core odl-dluxapps-nodes odl-dluxapps-topology
```

10. Change to a second terminal session. Use the **netstat -tulpn** command to view active connections. You should see a series of Java processes listening, including one listening on port 6633.

```
node-term2# netstat -tulpn
Active Internet connections (only servers)
```

```
Proto Recv-Q Send-Q Local Address           Foreign Address         State       PID/Program name
tcp        0      0 0.0.0.0:6653            0.0.0.0:*               LISTEN      10459/ovs-testcontr
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN      1262/sshd
tcp6       0      0 :::44444                :::*                    LISTEN      13850/java
tcp6       0      0 :::35073                :::*                    LISTEN      13850/java
tcp6       0      0 :::8101                 :::*                    LISTEN      13850/java
tcp6       0      0 :::6886                 :::*                    LISTEN      13850/java
tcp6       0      0 127.0.0.1:46662         :::*                    LISTEN      13850/java
tcp6       0      0 :::6633                 :::*                    LISTEN      13850/java
<output_omitted>
```

11. Create another controller, switch and hosts using **Mininet**. This time we will pass a **–controller** reference using the IP address of the node and port 6633. Ensure you don't see an error or delay as it talks to the ODL controller. We can also pass various topology options.

```
node-term2# mn --controller=remote,ip=172.31.xx.yy,port=6633 --topo=tree,2
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1 s2 s3
*** Adding links:
(s1, s2) (s1, s3) (s2, h1) (s2, h2) (s3, h3) (s3, h4)
<output_omitted>
```

12. Use a web browser and navigate to the IP address of the node. You must include the 8181 port as well as `/index.html` to the path. The default user name and password is **admin**.

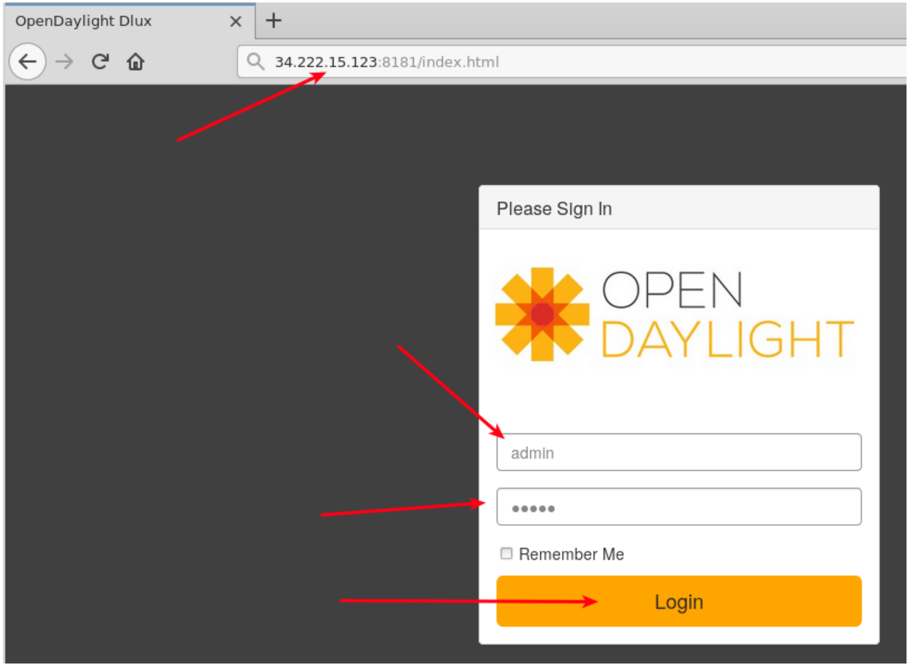


Figure 1.14: **Login Page**

13. Upon logging in you are presented with the current `nodes` known by the controller. On the left of the page you should see a link to view the `Topology`. Upon selecting that link you should see the three switches, but no hosts. The hosts have not generated any traffic and do not currently have rules.

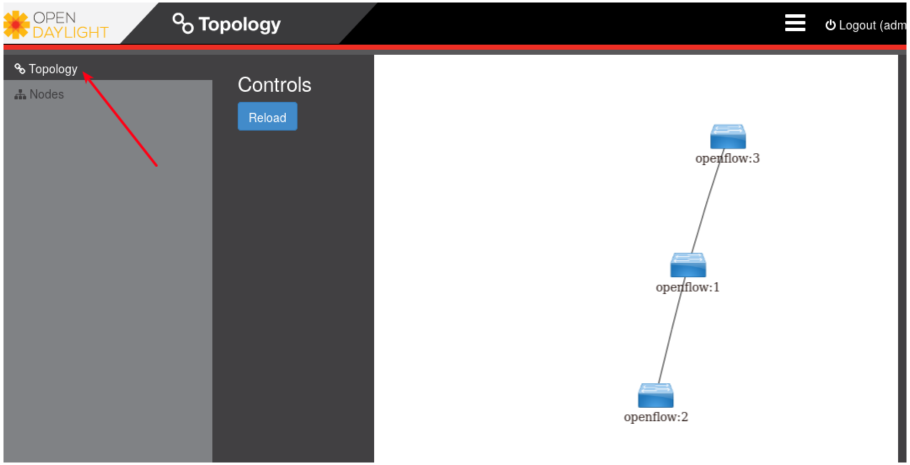


Figure 1.15: **Login Page**

14. Return to the second terminal where the **Mininet** continues to run. Use the **pingall** command to generate traffic among the hosts. Then return to the web page. Use the **Reload** button. You should see the hosts have been added.

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4
h2 -> h1 h3 h4
h3 -> h1 h2 h4
h4 -> h1 h2 h3
*** Results: 0% dropped (12/12 received)
```

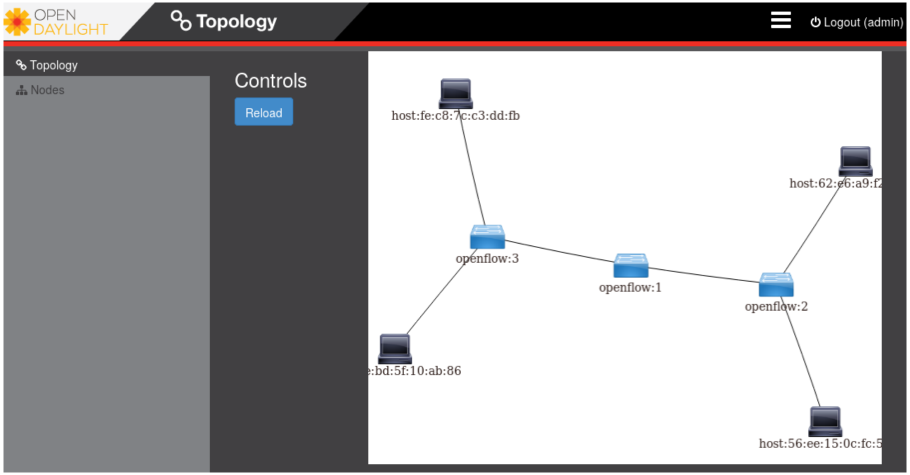



Figure 1.16: **Login Page**

Exercise 1.3: Use Wireshark to Examine Flow Modifications

We installed the **Wireshark** tool which allows us to capture and examine every packet from a graphical interface. As there may be a lot of traffic you may want stop the traffic while you investigate the details.

1. Log into the instance in a third terminal. Because I am accessing the node remotely I need to export the graphical display back to my local node. The use of the **SSH** `-X` or `-Y` may be necessary depending on your connection to the node.

```
[laptop ~]$ ssh -Y -i LFS452.pem ubuntu@34.222.15.123
Warning: No xauth data; using fake authentication data for X11 forwarding.
Welcome to Ubuntu 16.04.5 LTS (GNU/Linux 4.4.0-1065-aws x86_64)
<output_omitted>
node-term3$
```


2. We allowed non-root users to capture packets, but you may still get some errors about certain output files. As you we are running **Mininet** in another window you will see several interfaces when **wireshark** starts. Chose `any` to see all traffic. You can also return to this lab later and experiment with various interfaces to learn which may handle various types of traffic.

```
node-term3$ sudo wireshark
```

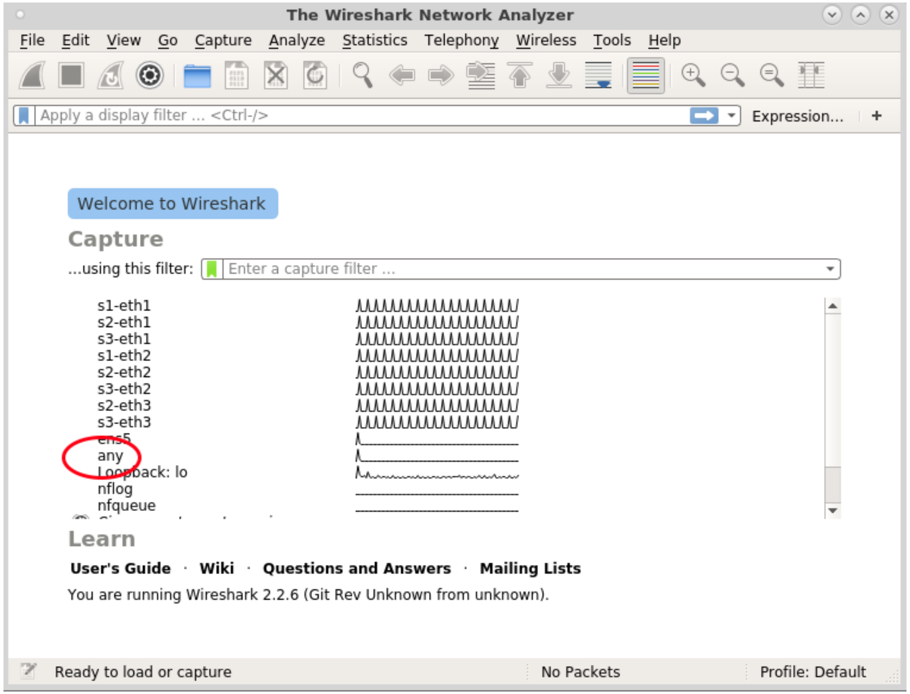


Figure 1.17: **Login Page**

3. Note with **Wireshark** that the version of **OpenFlow** can determine if you see output. Start with `openflow_v4`. After

adding the filter return to the second terminal and cause some rules to be made.

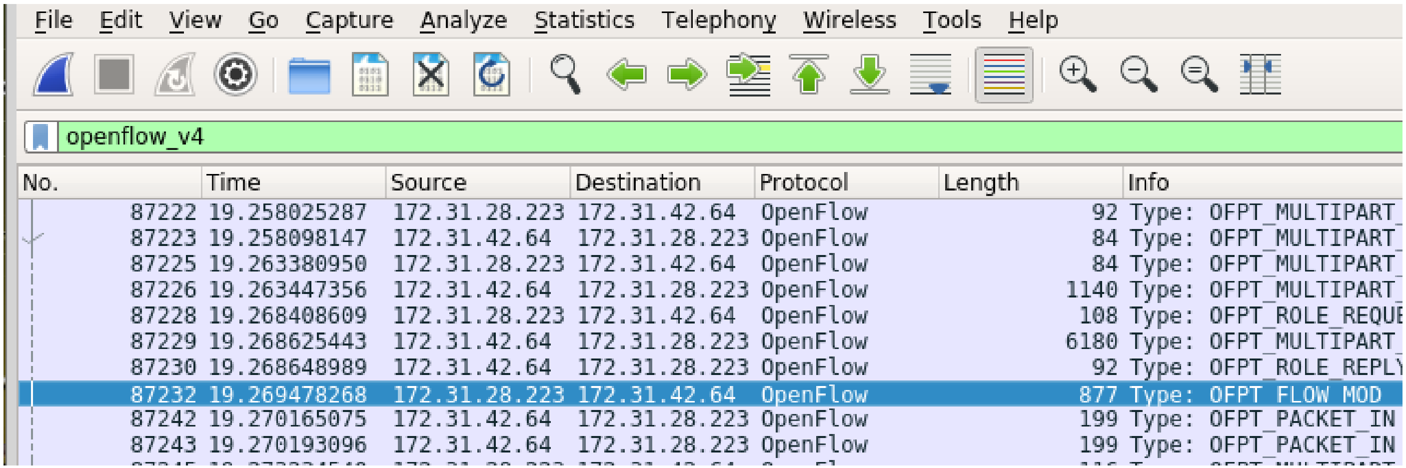


Figure 1.18: **Wireshark with Filter**

```
mininet> exit
*** Stopping 1 controllers
<output_omitted>

node-term2$ mn --controller=remote,ip=172.31.28.5,port=6633 --topo=tree,2
<output_omitted>
mininet> pingall
<output_omitted>
```

4. Experiment with various switch topologies. Use multiple terminals to snoop the **OpenFlow** traffic while creating the switches as well as viewing the configuration via the browser, the **ovs-ofctl**, and **ovs-vsctl** commands.

```
node-term2$ mn --controller=remote,ip=172.31.28.5,port=6633 --topo=torus,3,3
```

```
node-term2$ mn --controller=remote,ip=172.31.28.5,port=6633 --topo=linear,4

node-term2$ mn --controller=remote,ip=172.31.28.5,port=6633 --topo=single,9
```