CE321 Network Engineering:

SDN Tutorial

Dr Martin Reed v1.3 7th December 2022

**NOTE: this laboratory is a demonstration of the new SDN material introduced in the 2019/20 academic year. This supports the content regarding software defined networking in the Cisco Courseware. This material goes beyond that material so that you can see an SDN system in action. SDN is also now being widely deployed and employers are increasingly interested in graduates having this knowledge.**

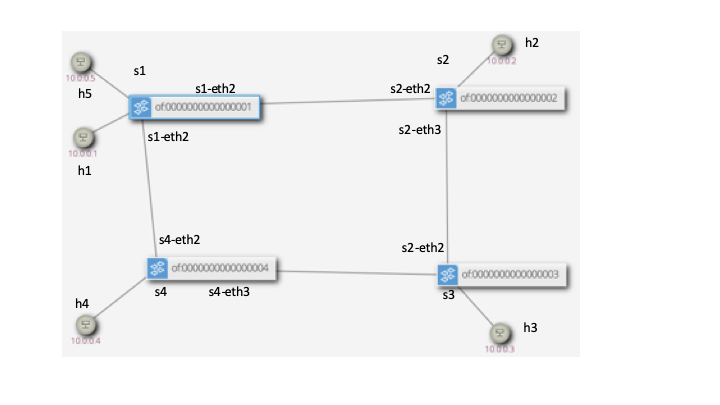
# **1. Introduction**

Software defined networking (SDN) can be an umbrella term encompassing a number of systems that configure networks using software (including something called SD-WAN which is quite different from the contents of this lab). Here we will use it in its a narrower sense to mean:

*Network switches under the control of a centralised software controller where the switches and controller communicate using the OpenFlow protocol. Through the use of this architecture existing protocols running on the switches (e.g. spanning tree protocol and/or routing protocols) are replaced by the central controller.*

This tutorial gives some examples of SDN in a small network to show its benefits and costs. It uses a Virtual Machine which contains a controller (ONOS <https://onosproject.org/>) and a network that is emulated using Mininet (<http://mininet.org/>). Although the network is emulated and running on a single Virtual Machine, this does not make it less “real,” in practice many cloud services running on Linux systems use the same Open vSwitch switches as used in the Mininet topology. This tutorial is meant to be performed in about two hours to:

* give a brief introduction to SDN through seeing an SDN network in action.
* perform some experiments to show the benefits and costs of SDN.
* show how the code in the SDN controller influences the action of switching.
* give opportunities for further exploration beyond this tutorial.



**Figure 1 Network Topology**

The test network is shown in Figure 1. Note that hosts have simple to remember MAC and IP addresses of the format 10.0.0.X and 0A:00:00:00:00:0X where X is replaced by the host number.

The virtual machine has a number of desktop icons which run the network and controller and provide shells for console access to the controller and running tests. It is also possible to run all the commands from the console and both graphical and command line versions of running the various components and tests will be given.

## 1.1. Passwords

**Virtual machine:**

Username: ce321

Password: networks

ce321 account has passwordless sudo privileges *i.e.* it can carry out root level commands using sudo <command> without a password, this would be a severe security hazard in a real computer!

**SDN controller**

Username: onos

Password: rocks

Again, not good examples of security!

# 2. Exploring the basic network

## 2.1. Starting the Controller and Basic Topology

**NOTE: in many cases both Graphical and Command Line versions are given, only use one (although you could use them interchangeably).**

Start the VM and start the controller:

Graphical: “Run ONOS Server” icon

Command line: cd ~/ONOS; bazel run ONOS-local --fetch=false

It is ready when you see:

195 - org.onosproject.onos-core-primitives - 2.7.0 | Updated node 127.0.0.1 state to READY

Wait until you see the ONOS server output stop (about 30 seconds, wait until the scrolling output stops), then start the basic topology (the -s means run it as SDN, later we will run it in STP mode):

Graphical: “Run SDN network” icon

Command line: cd ~/demo; sudo ./test-network.py -s

After some time to settle it will start the *Mininet* command line which allows you to interact with the Mininet network emulation system. This is feature-rich, but we will only be using the basic facilities.

Start by trying to ping between h1 and h2 and note it fails (use Ctrl-c to finish a ping):

mininet> h1 ping h2

PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.

From 10.0.0.1 icmp\_seq=1 Destination Host Unreachable

….

This fails as we have a running controller, but it does not have any running SDN applications yet.

Open the GUI for the controller using (username ONOS; password rocks):

Graphical: “ONOS GUI” icon

Command line: firefox <http://localhost:8181/onos/ui> &

This may say “No devices connected yet” (depending on what is running by default from the last time it was run).

Open the ONOS console (if you want to quit it, use Ctrl-d) (password: rocks)app:

Graphical: “ONOS Console”

Command line: ssh -q -p 8101 onos@127.0.0.1

In the ONOS console start the applications that we need (note the onos@root prompt):

onos@root> app activate org.onosproject.openflow

onos@root> app activate org.onosproject.fwd

The first command enables the controller to use the OpenFlow protocol to communicate with the switches. The second implements a basic forwarding application that will insert *flow rules* into the controller to implement basic Ethernet MAC based switching (but not the same as the basic MAC table implemented in traditional Ethernet switching). Note loading these applications will also automatically load other needed SDN applications.

You should now see some devices in the Firefox window. Use the key-strokes **h** and **l** (lower case L) to toggle showing hosts and labels (this will be useful later).

Again ping between h1 and h2 and note it succeeds (use Ctr**l**-c to finish a ping):

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=14.6 ms

64 bytes from 10.0.0.2: icmp\_seq=2 ttl=64 time=10.0 ms

64 bytes from 10.0.0.2: icmp\_seq=3 ttl=64 time=0.500 ms

64 bytes from 10.0.0.2: icmp\_seq=4 ttl=64 time=0.051 ms

^C

--- 10.0.0.2 ping statistics ---

4 packets transmitted, 4 received, 0% packet loss, time 3012ms

rtt min/avg/max/mdev = 0.051/6.312/14.652/6.254 ms

This should succeed, as above, if it does not please see a lab demonstrator.

Now you should see the network topology in the ONOS GUI in Firefox. You will probably only see hosts h1 and h2 unless you have sent traffic to other hosts. Note you may need to use **h** and **l** keys to toggle showing the hosts and labels.

## 2.2. Observing the data plane flow rules

Wait at least 15 seconds without any traffic running in the network (ie after your ping above, or any other tests you perform later), now look at the flow rules in the switch s1 in an terminal (you will need to open a new terminal for this, it will **not work** in the ONOS console nor the Mininet console.)

sdn@sdn:~$ sudo ovs-ofctl dump-flows s1

cookie=0x10000021b41dc, duration=397.615s, table=0, n\_packets=5, n\_bytes=490,

priority=5,ip actions=CONTROLLER:65535

cookie=0x100009465555a, duration=397.615s, table=0, n\_packets=257, n\_bytes=35723, priority=40000,dl\_type=0x88cc actions=CONTROLLER:65535

cookie=0x10000ea6f4b8e, duration=397.615s, table=0, n\_packets=8, n\_bytes=336,

priority=40000,arp actions=CONTROLLER:65535

cookie=0x100007a585b6f, duration=397.614s, table=0, n\_packets=257, n\_bytes=35723, priority=40000,dl\_type=0x8942 actions=CONTROLLER:65535

This contains only four flow-rules, the first line of each is basically some statistics and identifiers, please ignore this, for now. The second part is more interesting. The first means “if an IP packet arrives in the switch then send it to the ONOS controller.” The others are similar except for other types of Ethernet frames, the third is for ARP packets, the other two are for two discovery protocols used in the network. These four rules are placed in all the network switches start up by the controller so that frames from all “unknown” flows are sent to the controller. Note that the flow rules have a priority: a frame is compared to the list of flow rules in order (highest number priority first) until one matches, the associated action is performed, then the frame is not compared to any other rules.

Now start the ping between h1 and h2 again, leaving it going for the moment while you look again at the flows in s1, you will see two new rules (the other four will still be there):

cookie=0x1300006e01dc4b, duration=18.437s, table=0, n\_packets=17, n\_bytes=1666, priority=10,in\_port="s1-eth1",dl\_src=0a:00:00:00:00:01,dl\_dst=0a:00:00:00:00:02 actions=output:"s1-eth2"

cookie=0x13000052014fde, duration=18.434s, table=0, n\_packets=17, n\_bytes=1666, priority=10,in\_port="s1-eth2",dl\_src=0a:00:00:00:00:02,dl\_dst=0a:00:00:00:00:01 actions=output:"s1-eth1"

These two rules were inserted by the ONOS *fwd* application as a result of the first frame being sent to the controller. We can now describe the rules in more detail. Consider the following portion of the first of the above rules:

in\_port="s1-eth1",dl\_src=0a:00:00:00:00:01,dl\_dst=0a:00:00:00:00:02 actions=output:"s1-eth2"

This has two parts: the first is the *selector:*

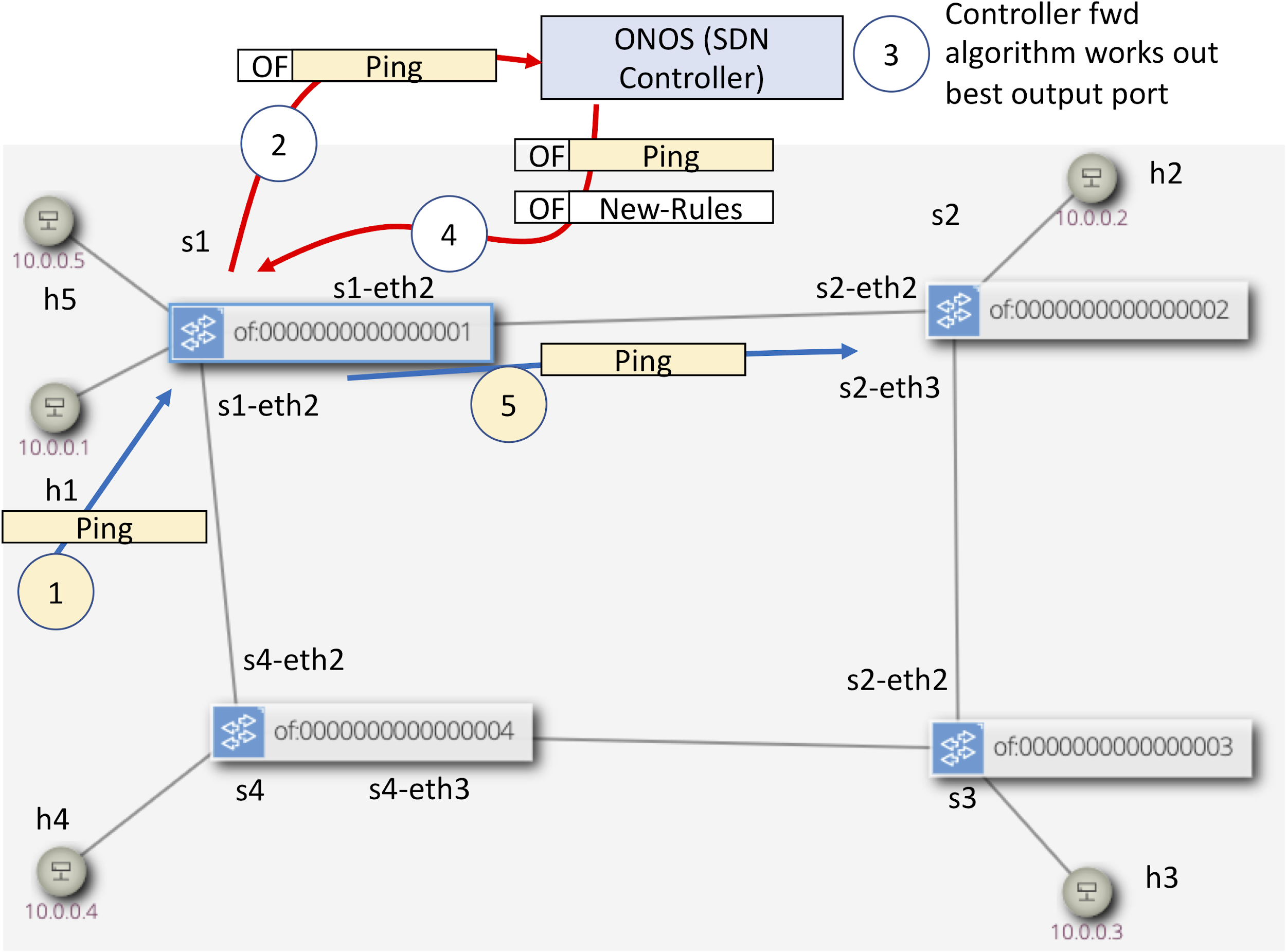
in\_port="s1-eth1",dl\_src=0a:00:00:00:00:01,dl\_dst=0a:00:00:00:00:02

This matches the input port, Ethernet source and Ethernet destination, but it could also match things like Ethertype, IP addresses, TCP/UDP ports (ie most things in the headers).

The second part is the *action:*

actions=output:"s1-eth2"

Which means send it out the second port (called s1-eth2), but it could have other actions such as changing the addresses (e.g. for network address translation), adding a VLAN tag, or even resubmit the frame to another switch table for further processing.



**Figure 2 SDN steps on the first Ping packet between h1 and h2, note data plane and control plane actions are coloured differently**

Figure 2 illustrates and breaks these steps down:

1. The ping is sent from h1 to s1 (we are skipping the ARP message, assuming this has already happened)
2. S1 looks in the flow table and finds no specific rule so it matches the “cath all” ip rule to send frames to the controller. It sends the whole ping packet to the controller as the contents of an OpenFlow message.
3. The application in the controller (in this case called fwd) works out the best output port
4. The controller sends the ping packet back to the switch, again in an OpenFlow message, but this time it also has the instruction on what the switch should do with it (send out port 2). It also sends some other OpenFlow messages to insert new rules in the switch (as you have seen above).
5. The switch now sends the frame out of the indicated port (2) and inserts the rules in the switch.
6. (not shown) the same as above will be performed in s2
7. (not shown) future frames in the flow will not be sent to the controller while the rules exist.
8. (not shown) when the flow of packets stops the switches delete the flows without telling the controller. (The timeout is 10s).

If this sounds complex, well yes, it is. However, note that it only happens on the first frames, once the two rules have been inserted at priority 10 any subsequent frames are switched directly by the rules without intervention from the switch; the catch-all IP rule is at a lower priority of 5.

There are some key differences between this method and “traditional” Ethernet switching apart from the obvious steps above:

* SDN separates the *control plane* from the *data plane*. We no longer have spanning tree protocol (STP) and routing protocols performing the control plane in the same links as the data plane, OpenFlow goes over a separate network (at least in this simple story). Instead of the control plane being distributed, it runs in a central controller.
* The switching (in this application) inserts rules that match both source and destination MAC addresses; whereas in traditional Ethernet switches the MAC address table only contains destination addresses. You may see why this is important later.

While the ping is still going on look at the flow tables in the other three switches and record the path that the pings between h1 and h2 go over.

So now you have seen the basics of SDN. However, we cannot use this to say how every SDN network operates, it will depend on the application loaded into the controller, in this case one called *fwd*. If a different application is run the story might be slightly (or even wildly) different.

# 3. Comparing the performance of SDN vs STP controlled network

Now we will compare both and SDN and STP controlled network by carrying out some very basic experiments (each test is available from the Mininet command line in the test\_network script. Here is a summary of the tests, we will use them in slightly different ways and running the tests can be automated as described after the summary:

test1 : send a TCP flow between h1 and h2 as quickly as possible and report throughput.

test2 : send a TCP flow between h1 and h2 as quickly as possible while simultaneously sending between h4 and h3 and report throughput.

test3: send a TCP flow between h1 and h2 as quickly as possible while simultaneously sending between h5 and h2 and report throughput.

test4: clear the ARP caches in h4 and h3, wait for the flow-tables to clear in the switches, then send 10 pings between h4 and h3.

test5 : fill the ARP caches in h4 and h3 (by sending an initial ping which is not shown), wait for the flow-tables to clear in the switches, then send 10 pings between h4 and h3.

These are some basic tests that will allow us to test some basic differences between SDN and STP in practice, they are by no means exhaustive. Note that the timings and bit rates observed will be highly dependent on what else is running and the OS/CPU caching mechanisms. Try not to run anything else while running the tests and keep the virtual machine desktop in focus all the time. The capacity of all the links between the switches has been limited to approximately 10 Mb/s, but the mechanism to do this limitation is not exact and you may see traffic above this level.

## 3.1. SDN performance experiments

Close any existing Mininet consoles or windows (the script will disable old scripts but this is not obvious in the old scripts you might leave lying around).

You can either run the tests from an existing Mininet window (test1, test2 etc.) or simply run the automated tests in order:

Graphical: “Test SDN scenario” icon

Command line: cd ~/demo; sudo ./test-network.py -s -t

The -t flag (“normal” mode) tells the script to run all five tests in order.

Record the output of the five tests (copying from the command line will be easiest).

When they have finished confirm the path taken by the (h1, h2), (h4,h3) and (h5,h2) traffic flows by launching a ping between one pair at a time and looking at the switch flow tables.

## 3.2. STP performance experiments

Again, close any existing Mininet consoles or windows.

You can either run the tests from an existing Mininet window (test1, test2 etc.) or simply run the automated tests in order:

Graphical: “Test STP scenario” icon

Command line: cd ~/demo; sudo ./test-network.py -n -t

The -n flag (“normal” mode) tells the script to run the switches in “standalone” mode without the controller and instead uses Spanning Tree Protocol (actually RSTP) as the control plane.

Record the output of the five tests (copying from the command line will be easiest).

Note that when the script started (ie above the tests), it lists the STP role of each port. Record this and thus work out the path taken by the (h1, h2), (h4,h3) and (h5,h2) traffic flows. You can also see the switching tables by looking at the MAC address tables in the switches, for example in s1 (only showing relevant entries):

sdn@sdn:~$ sudo ovs-appctl fdb/show s1

port VLAN MAC Age

2 0 0a:00:00:00:00:04 0

3 0 0a:00:00:00:00:01 0

2 0 0a:00:00:00:00:05 0

2 0 0a:00:00:00:00:03 0

3 0 0a:00:00:00:00:02 0

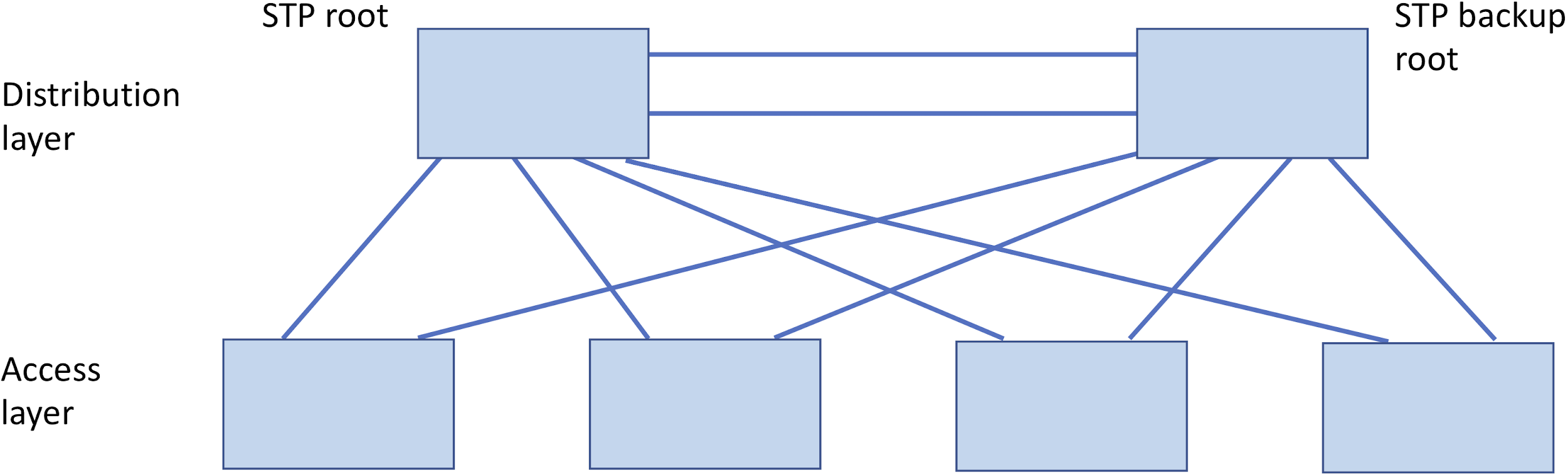
If you need a reminder about STP port roles and how Ethenet uses MAC address tables, look it up in your previous course notes or look it up on a source such as Wikipedia. In brief: STP works to stop any switching loops by “turning off” selected ports to stop the loops (alternate ports in RSTP); the MAC address table is populated as frames appear in the switches and records which port to send frames that match that particular destination MAC address.

## 3.3. Your conclusions from your experiments?

Compare your results from the tests run in the two scenarios SDN vs STP, what do you conclude about:

* Choice of paths and advantages/disadvantages this may bring?
* Why does SDN need to take note of both source and destination frame MAC addresses while STP only records the destination address in its MAC address table?
* If there are N devices in the network how big can the following become:
  + the number of flow entries in each switch in the case of SDN?
  + the number of MAC addresses in the MAC address table in the case of STP?
* Delays, in particular in the first packet in a flow?

One of the big reasons SDN has been deployed in data centres is that it can get approximately 40% more traffic in the network compared to a simple STP deployment. Can you see why from the (highly contrived) example network shown in Figure 3? You will need to consider which ports are in alternate mode (for RSTP) and add up the traffic available from links that do not have an alternate port on one end.



**Figure 3 Stylised Data Centre network with four access layer switches and two distribution layer switches, consider which ports will be alternate in this topology?**

# **4. More advanced investigation**

## **4.1. The SDN controller**

One might say: what you do with SDN is simply limited by your imagination, it allows a complete rethinking of networking by allowing the programmer of an SDN controller to do whatever they need to do with the network.

We said earlier that SDN might look complex. Consider what the controller needs to do, this is indeed complex, it needs to:

* Understand a wide variety of network protocols
* Implement the OpenFlow protocol to communicate between switches and the controller
* Run algorithms to “do the right thing” to tell switches how to forward frames (e.g. in our simple fwd application shown above)
* Have management interfaces (e.g. the Web based GUI you have seen)
* Allow programmatic interfaces for controlling the system, typically these are over a REST API (ie using HTTP GET and POST messages)

The ONOS controller you are using is the basic open-source version that a real application developer would need to adapt to their own situation. Even this version is vast with over one million lines of code (see the Appendix for some stats). You will now be editing this code. This might be the first time you have worked with such a large code base, luckily it is well organised. We will be looking at a copy of the fwd application which is called (rather unimaginatively) *fwdmod*. During this experiment we will see how such applications can be loaded and unloaded from Onos while it is running. The scenario will use quality of service (QoS) to investigate controller capabilities.

## 4.2. Running a QoS example

Start Onos as before (or keep using the one from earlier) and make sure the fwd and openflow applications are running by opening the Onos Console:

Graphical: “ONOS Console”

Command line: ssh -q -p 8101 ONOS@127.0.0.1

and in this console listing the running applications (some will have been automatically loaded by the fwd and openflow applications):

onos@root > apps -a -s

\* 9 org.onosproject.gui2 2.2.1.SNAPSHOT ONOS GUI2

\* 19 org.onosproject.fwd 2.2.1.SNAPSHOT Reactive Forwarding

\* 35 org.onosproject.optical-model 2.2.1.SNAPSHOT Optical Network Model

\* 52 org.onosproject.drivers 2.2.1.SNAPSHOT Default Drivers

\* 56 org.onosproject.openflow-base 2.2.1.SNAPSHOT OpenFlow Base Provider

\* 57 org.onosproject.lldpprovider 2.2.1.SNAPSHOT LLDP Link Provider

\* 58 org.onosproject.hostprovider 2.2.1.SNAPSHOT Host Location Provider

\* 63 org.onosproject.openflow 2.2.1.SNAPSHOT OpenFlow Provider Suite

If the fwd or openflow applications are not running, then activate them as above in Section 2.1.

Now start a slightly different Mininet scenario:

Graphical: “Test SDN with QoS” icon

Command line: cd ~/demo; sudo ./test-network.py -q -s

The -q flag (“qos” mode) tells the script to run the switches with some quality of service (QoS) queues enabled. In fact they are implementing *class based queuing*. You will see a description of the QoS queues in one of the ports in s1 as shown in the Mininet script (all switch-to-switch output ports have the same types of queues):

\*\*\* Showing Queues in s1-eth2

\*\*\* s1 : ('tc -g class show dev s1-eth2',)

+---(1:1) htb rate 10Mbit ceil 10Mbit burst 1600b cburst 1600b

+---(1:2) htb prio rate 9Mbit ceil 10Mbit burst 1598b cburst 1600b

+---(1:3) htb prio rate 1Mbit ceil 10Mbit burst 1600b cburst 1600b

The numbering of these queues is a little confusing, for the SDN switches (and controller) they are 0,1 and 2, for the linux subsystem that implements them they are respectively (1:1), (1:2)and(1:3)

The meaning of this is that the link (1:1, queue 0) is the main output queue with maximum (ceil) of 10 Mb/s. Then there are two queues that feed this main output queue:

* (1:2) queue 1, which has a guaranteed rate of 9 Mb/s but which can go up to 10 Mb/s if no other queue wants the capacity of the parent queue 0
* (1:3) queue 2, which has a guaranteed rate of 1 Mb/s but which can go up to 10 Mb/s if no other queue wants the capacity of the parent queue 0.

**Note: queue 2** (1:3) is the **default queue**, unless the switch is told otherwise, all traffic is sent to this queue.

In the Mininet console run test3 (send a TCP flow between h1 and h2 as quickly as possible while simultaneously sending between h5 and h2 and report throughput). Record the throughput of both flows. Additionally, record the flow tables from s1 as shown in Section 2.2. If the test has finished re-run it while you record the flow table.

You should have noted that the two flows have (approximately) the same throughput. Note that the total might not add up to 10 Mb/s as the class based queueing being used is not very accurate. The reason that there is no difference between the two is that we have not treated them differently and all traffic goes to the default queue 2. So although QoS is enabled in the switches the control plane (the controller) does not know how to treat traffic differently.

## 4.3. Running a QoS aware SDN applicatioon

Now you will change the SDN application to one called *fwdmod* which manages traffic differently based upon the source MAC address.

Insert the new fwdmod application by running the following script (it must be run from an LXTerminal, not the Mininet or ONOS console):

sdn@sdn:~$ cd onos/myscripts/

sdn@sdn:~/onos/myscripts$ ./load-fwdmod.sh

This script uses a REST API to send all the compiled code in the fwdmod application to ONOS, tells it to deactivate fwd and instead activate fwdmod.

Check that this new module is now running in ONOS (it should have replaced fwd):

onos@root > apps -a -s

\* 9 org.onosproject.gui2 2.2.1.SNAPSHOT ONOS GUI2

\* 19 org.onosproject.fwdmod 2.2.1.SNAPSHOT Reactive Forwarding

\* 35 org.onosproject.optical-model 2.2.1.SNAPSHOT Optical Network Model

\* 52 org.onosproject.drivers 2.2.1.SNAPSHOT Default Drivers

\* 56 org.onosproject.openflow-base 2.2.1.SNAPSHOT OpenFlow Base Provider

\* 57 org.onosproject.lldpprovider 2.2.1.SNAPSHOT LLDP Link Provider

\* 58 org.onosproject.hostprovider 2.2.1.SNAPSHOT Host Location Provider

\* 63 org.onosproject.openflow 2.2.1.SNAPSHOT OpenFlow Provider Suite

Now re-run (in the Mininet console) test3 (send a TCP flow between h1 and h2 as quickly as possible while simultaneously sending between h5 and h2 and report throughput). Record the throughput of both flows. Additionally, record the flow table from s1 while this test is running as shown in Section 2.2. If the test has finished re-run it while you record the flow table.

How are the flow tables different from the previous time you ran test3? Does this explain the throughput results?

To see how the code works to perform this we will look at the source code. Open the editor Geany:

Graphical: “Geany” icon

Command line: geany

By default it will open to the relevant part of the code which is the file:

/home/sdn/onos/apps/fwdmod/src/main/java/org/onosproject/fwdmod/ReactiveForwarding.java

This version of the fwd application has been simplified as much as possible so that you can understand the basics of how the code operates. There are two core functions:

* process(...) which is called when the packet arrives and which calls the controller topology service to find out the best port to send the frame out of, then it calls the next function of interest ...
* installRule(...) which actually installs the rule (and tells the switch to forward the packet as well)

Look at the lines of code in installRule() just below the comment

// demo inserted change to send to queue 1 for certain mac source

You should be able to see how the code treats the different source MAC addresses differently from the code.

There are some other parts of the code highlighted for interest in this file, just search for any mention of “demo.”

Now edit the code to change the MAC address of interest to that of h5 instead of h1. You then can compile the code using the menu Build -> Compile. Then you need to load this new version of the application into ONOS using the load-fwdmod.sh script again (it actually reinstalls it if it was previously loaded). Again now run test3 and observe if this changes how the flows are queued.

You have seen how the code actually inspects incoming packets, makes a decision and then implements a flow rule in a switch. In practice, deployers of SDN tend to work on higher layer interfaces to the SDN controller using things like the REST API to configure it and using *Intents* to perform *Intent Based Networking*. This will be covered in a later addition to this tutorial.

# 5. Conclusions and Further Work

This has given a brief and low-level introduction to SDN and using Mininet to test an SDN network. There are many more things that can be done. Here are some ideas:

* Build on the test network given here towards more complex scenarios. See the script from this lab here on github <https://github.com/martinjreed/sdn-demo> which you could build upon
* Investigate more complex QoS scenarios and test them
* Investigate the latency in SDN and look at ways of reducing it (e.g. by moving from reactive based forwarding towards using Intents, or by investigating adaptive timeouts)
* Investigate resiliency in the SDN controller (ONOS supports a cluster of controllers)
* Write your own SDN application by forking the ONOS codebase available at <https://github.com/opennetworkinglab/onos>

# Appendix

**Statistics on the code in ONOS (as at version 2.2):**

**--------------------------------------------------------------------------------**

**Language files blank lines comment lines code lines**

**--------------------------------------------------------------------------------**

**Java 9103 166144 376389 766554**

**JSON 852 190 0 188466**

**JavaScript 265 9965 11363 45458**

**TypeScript 248 2765 6056 17325**

**XML 244 1647 3700 14597**

**CSS 174 1800 3192 9762**

**Python 91 2102 1604 9049**

**HTML 169 505 1397 5477**

**Bourne Again Shell 188 1428 1198 5095**

**Markdown 95 1551 0 4624**

**Bourne Shell 37 495 390 1981**

**Maven 22 177 384 1473**

**YAML 25 108 80 934**

**Protocol Buffers 55 196 985 734**

**Dockerfile 1 13 21 48**

**make 3 13 0 39**

**C Shell 1 4 10 27**

**zsh 1 2 4 5**

**--------------------------------------------------------------------------------**

**SUM: 11574 189105 406773 1071648**

**--------------------------------------------------------------------------------**