SDN controller clustering

Computer Networks module - SDN assignment

Michele Zanotti

Spring term 2018

Assessment Feedback					
Aspect (& weighting)	Excellent	Very Good	Satisfacto ry	Needs some more work	Needs much more work
Content					
Critical Analysis					
Structure					
Referencing					
Presentation + Discussion					
Specific aspects of the assignment that the marker likes:		Specific aspects of the assignment that need more work:			
Tutor's Signature:		Date:		Grade	

Overview

In this paper are proposed four different lab activities meant to teach how to implement a cluster of controllers inside Mininet using some of the most common available methods. Each activity focuses on a single method and uses it to implement a given topology.

In particular, in each activity the following methods will be explained:

- **Activity 1**: implement a network with multiple controllers using a python script and the middle-level Mininet API
- **Activity 2**: implement a network with multiple local controllers using a python script, a custom switch class and the topology classes provided by the Mininet high-level API
- **Activity 3**: dynamically connect the switches of a Mininet network to different controllers using the tool ovs-vsctl.

The paper also includes a fourth lab activity (**Activity 4**), which is a conclusive challenge meant to let the reader test the knowledge acquired with the execution of the previous four activities. Each activity moreover contains some final questions aimed to make the reader reflect on what he did during that specific activity and possibly make him try to bring minor changes to the implemented topology. The solution for the activity 4 and the answers to the questions included in the previous activities can be found in Appendix A.

It's important to point out that all the proposed activities are meant to be only an introduction to SDN controller clustering, therefore this paper won't cover some of the aspects that have to be considered when implementing a cluster of controllers in a real-world context, such as load balancing and communication between controllers. More information about these topics can be found in references [2], [3] and [4]. The communication between different controllers can however be automatically managed by Mininet, the network emulator used in these paper activities. When in fact a network with multiple local controllers is implemented inside the emulator, all the local controllers are executed inside the kernel space on a single machine and therefore they can communicate with each other without an external communication mechanism being necessary.

Lab activity 1

Topology diagram

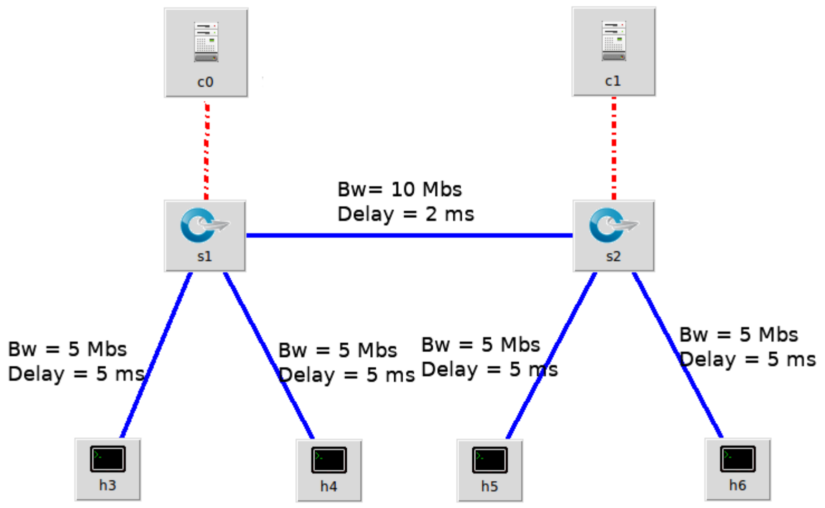


Figure 1: the topology that will be implemented during this lab activity. It is a simple linear topology with two switches connected to each other, each one connected to two hosts. Each switch is linked to a different SDN controller. The two controllers can be assumed local.

Learning objectives

After finishing this lab activity you will be able to:

- implement a cluster of local/remote controllers inside Mininet using the Python middle-level Mininet API
- test the network connectivity of a network which includes a cluster of controllers and verify that its performance meets the requirements
- understand the main functions provided by the middle-level Mininet API required to implement a cluster of controllers
- understand how to set performance parameters when implementing a network using a Python script and the middle-level Mininet API
- reflect about the reasons of using more than one controller in SDN networks.

Scenario

In this activity you will implement the simple topology shown in figure 1 using a Python script and the middle-level API provided by Mininet. Begin by creating a new Python script, then import the Mininet classes required for this activity and define the function that will be used to create the topology. Inside the body of this function create a new Mininet network and add to it the required hosts, switches, links and controllers. After writing the script, execute it to create the network, test its connectivity and verify that the performance meets the requirements. To conclude, answer the questions proposed in the last task.

This lab activity assumes you are proficient in SDN networks and Mininet network emulator A basic knowledge of the Python programming language is also required.

The activity is inspired by the script *controllers2.py* [6] included in the examples provided by Mininet, which can therefore be used as an additional example of how a network with multiple controllers can be implemented inside Mininet using a Python script and the middle-level API.

Task 1: write the skeleton of the Python script

Step 1

Create a new Python script and edit it with the text editor you prefer. If you're editing it inside the Mininet virtual machine, it is suggested to use Vim text editor.

Step 2

Import the required Python classes from the Mininet API:

```
#!/usr/bin/Python
from mininet.net import Mininet
from mininet.node import Controller, OVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink
```

Step 3

Define the function that will be used to create the topology:

```
def multiControllerNet():
```

Step 4

Inside the body of the function `multiControllerNet()` create a new Mininet network:

```
net = Mininet( controller=Controller, switch=OVSSwitch,
    link=TCLink )
```

The Mininet network is created invoking the Mininet constructor: the first two parameters passed to the constructor are the classes `Controller` and `OVSSwitch`, therefore Stanford/OpenFlow reference controllers and Open vSwitch switches will be used in the created network. Note that these two classes are the default parameters in the Mininet constructor, so it is not really necessary to specify them [5].

The third parameter passed to the constructor is the class `TCLink`, which provides performance limiting features. By using this class it is possible to set performance parameters when creating the links of the Mininet network.

Step 6

Make the script executable only as a program, set the CLI verbosity level to "info" and call the function `multiControllerNet()`:

```
if __name__ == '__main__':
    setLogLevel( 'info' )
    multiControllerNet()
```

Note that the lines of code above must be placed after the function `multiControllerNet()` outside its body.

Step 7

Save the text file as "*activity-1.py*" in your custom directory inside the mininet virtual machine.

Task 2: add hosts to the network

Step 1

Inside the body of the function `multiControllerNet()` add the following line of code in order to print to the console that hosts are being created:

```
info( "*** Creating hosts \n" )
```

Step 2

Still inside the body of the function `multiControllerNet()`, create the four hosts required for the topology by adding them to the mininet network previously created:

```
h1 = net.addHost('h3')
h2 = net.addHost('h4')
h3 = net.addHost('h5')
h4 = net.addHost('h6')
```

The function used to add the hosts to the network is `addHost('name')`, which accept as parameter the name of the host that will be created. The hosts names in this network therefore will be `h3`, `h4`, `h5` and `h6`.

Task 3: add switches to the network

Inside the body of the function `multiControllerNet()` print to the console that switches are being created and then add to the mininet network previously created the two switches required for the topology:

```
info( "*** Creating switches \n" )
s1 = net.addSwitch('s1')
s2 = net.addSwitch('s2')
```

Task 4: create links between nodes

Inside the body of the function `multiControllerNet()` print to the console that links are being created and then add the links between the hosts and the switches and the link between the two switches. For creating the links use the function `addLink`, specifying the bandwidth and the delay for each link:

```
info( "*** Creating links \n" )
net.addLink( h1, s1, bw=5, delay='5ms' )
net.addLink( h2, s1, bw=5, delay='5ms' )
net.addLink( h3, s2, bw=5, delay='5ms' )
net.addLink( h4, s2, bw=5, delay='5ms' )
net.addLink( s1, s2, bw=10, delay='2ms' )
```

Note that the bandwidth specified with the parameter `bw` is expressed in Mbit/s, while the delay specified with the parameter `delay` is expressed in milliseconds since we used the string "ms" (it is also possible to use the strings "us" and "s" in order to express the delay in microseconds and seconds) [1].

Task 5: create the controllers

Inside the body of the function `multiControllerNet()` print to the console that reference controllers are being created and then add to the Mininet network the two required controllers:

```
info( "*** Creating (reference) controllers \n" )
c0 = net.addController( 'c0', port=6633 )
c1 = net.addController( 'c1', port=6634 )
```

The function used to create the controllers is `addController`, which accepts as parameters the name of the controller which will be created and the TCP port that will be used by the switches for connecting to the controller.

Task 6: start the mininet network

Step 1

Append to the body the function `multiControllerNet()` the following line of code for printing to the console that the network is being started:

```
info( "*** Starting network \n" )
```

Step 2

Build the Mininet network:

```
net.build()
```

Step 3

Start the controllers:

```
c0.start()
c1.start()
```

Step 4

Start the switches, specifying for each switch the controller to which connect:

```
s1.start( [ c0 ] )
s2.start( [ c1 ] )
```

Step 5

Start the Mininet CLI:

```
info( "*** Running CLI\n" )
CLI( net )
```

Step 6

Stop the network so that after the user exits the Mininet CLI the network is stopped:

```
info( "*** Stopping network\n" )
net.stop()
```

Step 7

Save the file and exit the text editor.

Task 7: execute the script and test the network

After finishing the task 6 the script for implementing the required topology is completed. The full script is shown in listing 1 at the bottom of this activity.

Step 1

Execute the script as root:

```
$ sudo python activity-1.py
```

Step 2

Test the created topology: verify the network connectivity between all hosts. Write in the lines below the commands you used and the results you obtained.

Step 3

Verify that the bandwidth and the delay of each link comply with the values specified in the topology diagram shown in figure 1. Write in the lines below the commands you used and the results you obtained.

Task 8: reflection

1 - What are the advantages of having more controllers instead of one single controller which serves all the switches of the network?

2 - Would the fault tolerance of the network shown in figure 1 change if only one controller (linked to both the switches) was used instead of two?

3 - In this activity the topology shown in figure 1 was implemented assuming that the two controllers were local controllers. How would you have to change the Python script you created in this activity in order to use remote controllers instead of local ones? (Hint: see reference [9])

4 - How could we improve the fault tolerance of the netowork shown in figure 1 making minor changes to the Python script used?

Listing 1: the complete Python script required for Activity 1

```
#!/usr/bin/Python
from mininet.net import Mininet
from mininet.node import Controller, OVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink

def multiControllerNet():
    net = Mininet( controller=Controller, switch=OVSSwitch, link=TCLink )

    info( "*** Creating hosts\n" )
    h1 = net.addHost('h3')
    h2 = net.addHost('h4')
    h3 = net.addHost('h5')
    h4 = net.addHost('h6')

    info( "*** Creating switches\n" )
    s1 = net.addSwitch( 's1' )
    s2 = net.addSwitch( 's2' )

    info( "*** Creating links\n" )
    net.addLink( h1, s1, bw=5, delay='5ms' )
    net.addLink( h2, s1, bw=5, delay='5ms' )
    net.addLink( h3, s2, bw=5, delay='5ms' )
    net.addLink( h4, s2, bw=5, delay='5ms' )
    net.addLink( s1, s2, bw=10, delay='2ms' )

    info( "*** Creating (reference) controllers\n" )
    c0 = net.addController( 'c0', port=6633 )
    c1 = net.addController( 'c1', port=6634 )

    info( "*** Starting network\n" )
    net.build()
    c0.start()
    c1.start()
    s1.start( [ c0 ] )
    s2.start( [ c1 ] )

    info( "*** Running CLI\n" )
    CLI( net )

    info( "*** Stopping network\n" )
    net.stop()

if __name__ == '__main__':
    setLogLevel( 'info' )
    multiControllerNet()
```

Lab activity 2

Topology diagram

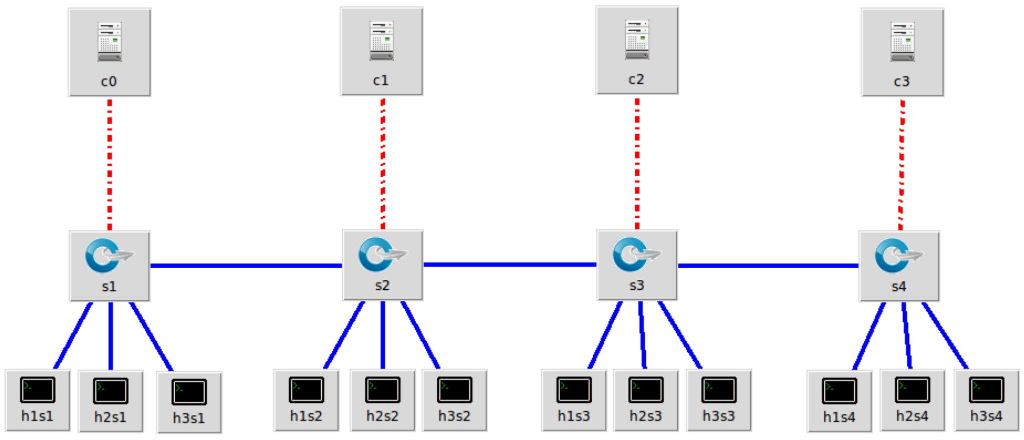


Figure 2: the topology that will be implemented during this lab activity. It is a linear topology with four switches connected to each other, each one connected to three hosts. Each switch is linked to a different SDN controller. All the controllers can be assumed local.

Learning objectives

After finishing this lab activity you will be able to:

- create a custom switch class that extends the OVSSwitch class provided by the Mininet Python API
- implement a cluster of local/remote controllers inside Mininet using a custom switch class and the topology classes provided by the Mininet high-level API
- implement more complex topologies with multiple controllers using a few lines of Python code
- reflect upon the method used in this activity for implementing a cluster of controllers and compare it with the method shown in Activity 1, focusing on the advantages and disadvantages of each method
- test the network connectivity and the performance of a network with multiple controllers.

Scenario

In this activity you will implement the topology shown in figure 2 using a Python script and the classes provided by the high-level Mininet API, which represent topology templates. In order to use these classes for creating a network with multiple controllers, you will have to define a new switch class that extends the standard OVSSwitch class provided by the API.

Begin by creating a new Python script and importing the required Mininet classes. Create the controllers shown in the topology diagram and define a custom switch class which extends the class `OVSSwitch`. Define a function for creating the network and inside its body use one of the topology templates provided by the API to implement the required topology. After writing the script, execute it and test the network connectivity and its performance. To conclude, answer the questions proposed in the last task.

This lab activity assumes that:

- you are proficient in SDN networks
- you are proficient in Mininet network emulator
- you have a basic knowledge of Python object-oriented
- you have already completed Activity 1.

The activity is inspired by the script *controllers.py* [7] included in the examples provided by Mininet, which can therefore be used as an additional example of how a topology with multiple controllers can be implemented inside Mininet using a Python script and defining a custom switch class.

Task 1: create the controllers

Step 1

Create a new Python script and edit it with the text editor you prefer. If you're editing it inside the Mininet virtual machine, it is suggested to use Vim text editor.

Step 2

Import the required Python classes from the Mininet API:

```
#!/usr/bin/Python
from mininet.net import Mininet
from mininet.node import OVSSwitch, Controller
from mininet.topo import LinearTopo
from mininet.log import setLogLevel
from mininet.cli import CLI
```

Step 3

Create the four required controllers, specifying for each one a different name and a different TCP port:

```
c0 = Controller( 'c0', port=6633 )
c1 = Controller( 'c1', port=6634 )
c2 = Controller( 'c2', port=6635 )
c3 = Controller( 'c3', port=6636 )
```

Step 4

Create a new array and initialize it with the four created controllers:

```
controllers = [c0, c1, c2, c3]
```

This array will be used later in the script to easily add all the controllers to the network using a for loop (see Step 4 of Task 3).

Step 5

Create a map that associates each switch to the relative controller:

```
cmap = { 's1': c0, 's2': c1, 's3': c2, 's4' : c3 }
```

Task 2: create a custom switch class

Step 1

Define a new class called `MultiSwitch` wich extends the class `OVSSwitch` provided by the Mininet API:

```
class MultiSwitch( OVSSwitch ):
```

Step 2

Overwrite the method `start` of the superclass `OVSSwitch`:

```
def start( self, controllers ):
  return OVSSwitch.start( self, [ cmap[ self.name ] ] )
```

The method simply returns the result of the call to the method `start` of the superclass passing as parameter the controller associated to the switch name, as defined by the map `cmap` previously created. In this way each switch will connect to the relative controller according to the map `cmap`.

Task 3: define a function that creates the topology

Step 1

Define a new function called "multiControllerNet":

```
def multiControllerNet():
```

Step 2

Create a new linear topology using the class `LinearTopo` provided by the API, specifying as parameters the number of switches `k` and the number of hosts per switch `n`:

```
topo = LinearTopo( k=4, n=3 )
```

Step 3

Create a new Mininet network, specifying as constructor parameters the topology called `topo` created in the previous step, the class `MultiSwitch` defined in task 2 and the value `build=False` for preventing Minined to build the network immediately:

```
net = Mininet( topo=topo, switch=MultiSwitch, build=False)
```

Step 4

Add the controllers to the network:

```
for c in controllers:
  net.addController(c)
```

Step 5

Build the network and start it:

```
net.build()
net.start()
```

Step 6

Start the CLI and stop the network:

```
CLI( net )
net.stop()
```

Task 4: finalize the script and save it

Step 1

Make the script executable only as a program and set the CLI verbosity level to “info”:

```
if __name__ == '__main__':
    setLogLevel( 'info' )
    multiControllerNet()
```

Step 2

Save the text file as “*activity-2.py*” in your custom directory inside the mininet virtual machine.

Task 5: execute the script and test the network

After finishing the task 4 the script for implementing the required topology is completed. The full script is shown in listing 2 at the bottom of this activity.

Step 1

Execute the script as root:

```
$ sudo python activity-2.py
```

Step 2

Test the created topology: verify the network connectivity between all hosts. Write in the lines below the commands you used and the results you obtained.

Step 3

Verify the bandwidth and the delay between the hosts `h1s1` and `h3s4`. Write in the lines below the commands you used and the results you obtained.

Task 6: reflection

1 - In this activity you implemented a network with multiple controllers using a custom switch class and the high-level API. Which are the advantages of using this method insted of the one used in Activity 1? What about the disadvantages?

2 - In this activity the topology shown in figure 2 was implemented assuming that all the controllers were local controllers. How would you have to change the Python script you created in this activity in order to use remote controllers instead of local ones?

3 - Try to modify the script you realized in order to have only two controllers instead of four, each one linked to two different switches. Each switch must be linked to a controller.

Listing 2: the complete Python script required for Activity 2

```
#!/usr/bin/Python
from mininet.net import Mininet
from mininet.node import OVSSwitch, Controller
from mininet.topo import LinearTopo
from mininet.log import setLogLevel
from mininet.cli import CLI

c0 = Controller( 'c0', port=6633 )
c1 = Controller( 'c1', port=6634 )
c2 = Controller( 'c2', port=6635 )
c3 = Controller( 'c3', port=6636 )

controllers = [c0, c1, c2, c3]
cmap = { 's1': c0, 's2': c1, 's3': c2, 's4' : c3 }


class MultiSwitch( OVSSwitch ):
  def start( self, controllers ):
    return OVSSwitch.start( self, [ cmap[ self.name ] ] )


def multiControllerNet():
  topo = LinearTopo( k=4, n=3 )
  net = Mininet( topo=topo, switch=MultiSwitch, build=False)

  for c in controllers:
    net.addController(c)

  net.build()
  net.start()
  CLI( net )
  net.stop()

if __name__ == '__main__':
  setLogLevel( 'info' )
  multiControllerNet()
```

Lab activity 3

Topology diagram

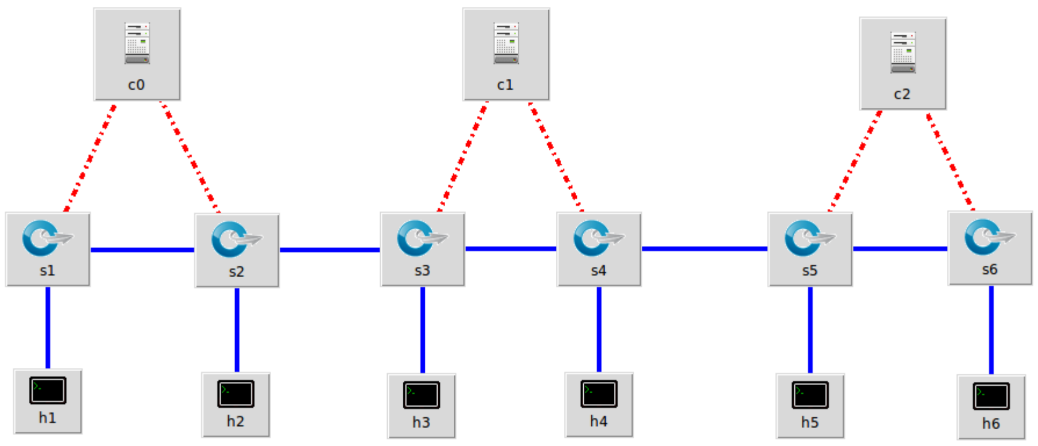


Figure 3: the simple linear topology that will be implemented in this activity. It is assumed that c1 and c2 are remote controllers running on the same machine on which Mininet is running, respectively on the TCP ports 6634 and 6635, while c0 is a local controller.

Learning objectives

After finishing this lab activity you will be able to:

- use the tool ovs-vsctl for implementing a cluster of controllers inside Mininet
- test the network connectivity and the performance of a network with multiple controllers
- dynamically set the controller for any switch of a generic running Mininet network using the tool ovs-vsctl.

Scenario

In this activity you will implement the topology shown in figure 3 using the command `mn` to create and start a new Mininet network, passing `--topo` as parameter to specify the required topology. You will then use the tool ovs-vsctl in order to set the controller for each switch according to the topology diagram shown in figure 3.

Begin by creating and starting a new Mininet network using the command `mn`. Once the network is running, start the remote controllers and use the tool ovs-vsctl for setting the controller for each switch according to the topology diagram. To conclude, test the network verifying the connectivity between all hosts.

This lab activity assumes that:

- you are proficient in SDN networks
- you are proficient in Mininet network emulator
- you have already completed the previous lab activities of this paper.

Task 1: create and start the network

Create and start a new Mininet network, using the parameter `--topo` to specify the required topology:

```
$ sudo mn --topo linear,6
```

After executing this command the network will be running and a single local controller (called `c0`) will be used for all the switches.

Task 2: start the remote controllers

Step 1

Open a new terminal and move to the directory `/home/mininet/pox`

Step 2

Start the first POX controller, using the component `forwarding.l2_learning` for making the OpenFlow switches act as L2 learning switches and the component `openflow.of_01` for specifying the TCP port to listen for connections on.

```
$ sudo ./pox.py forwarding.l2_learning openflow.of_01 --port=6634
```

Step 3

Open a new terminal and repeat step 1 and step 2 for starting the second controller. Remember to specify the correct TCP port, which this time will be 6635 instead of 6634.

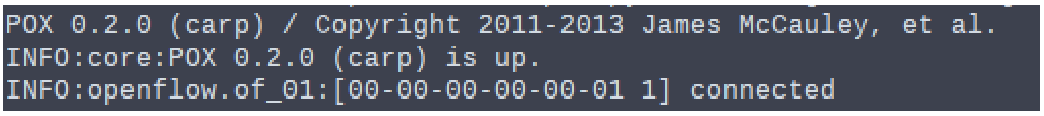
Task 3: set the controller for each switch

Step 1

Open a new terminal and use ovs-vsctl to connect the switch `s3` to the POX controller listening on port 6634 [8]:

```
$ sudo ovs-vsctl set-controller s3 tcp:127.0.0.1:6634
```

On the terminal with which you started that controller you should see that a new switch has connected, like in the screenshoot below:



```
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
INFO:core:POX 0.2.0 (carp) is up.
INFO:openflow.of_01:[00-00-00-00-00-01 1] connected
```

Step 2

Use ovs-vsctl to connect the switch `s4` to the POX controller listening on port 6634:

```
$ sudo ovs-vsctl set-controller s4 tcp:127.0.0.1:6634
```

Again, on the controller's terminal you should se that the switch has connected to the controller.

Step 3

Apply the same proceeding showed in the first two step to connect the switches `s5` and `s6` to the POX controller listening on port 6635.

Task 4: test the network

Test the connectivity between all hosts.

Lab activity 4: final challenge

Topology diagram

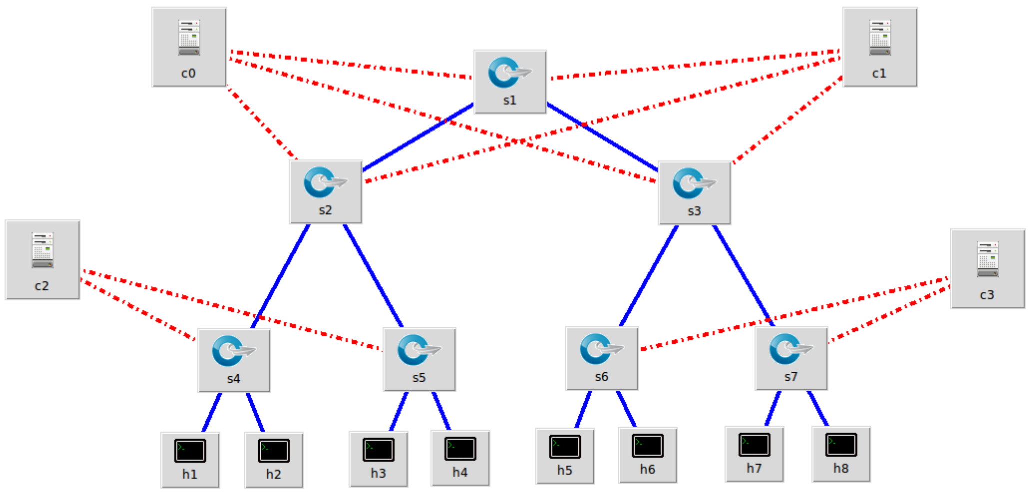


Figure 4: the topology that you will have to implement in this activity, or rather a tree topology with multiple SDN controllers. The controllers `c2` and `c3` are local while `c0` and `c1` are remote controllers running on the same machine on which Mininet is running, listening respectively on TCP ports 6635 and 6636.

Learning objectives

After finishing this lab activity you will be able to apply what you have learnt through the previous lab activities in order to implement on your own a given topology which includes multiple controllers.

Scenario

In this activity you will have to implement on your own the topology shown in figure 4 using the method you think is most appropriate among those previously shown in this paper.

The activity is meant to let you test the knowledge you acquired through the previous three activities proposed in this paper, therefore it assumes that you have already completed them successfully.

The solution for this activity is available in the appendix A of this paper.

Task 1: implement the topology

Implement the topology shown in figure 4 using the method you think is most appropriate.

Task 2: fix the network

Assume that the controller `c3` failed and it's not working anymore: without shutting down the whole network, fix the problem by connecting the switches served by `c3` to the other available remote controller.

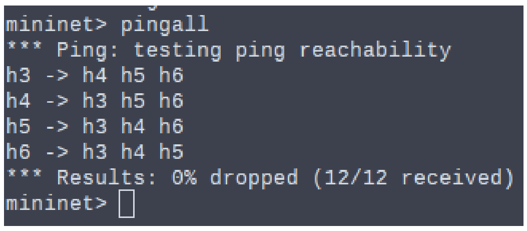
Appendix A

In this appendix are reported the answers to the questions proposed in all the previous lab activities.

Activity 1

Task 7 - Step 2

Test the created topology: verify the network connectivity between all hosts. Write in the lines below the commands you used and the results you obtained.



```
mininet> pingall
*** Ping: testing ping reachability
h3 -> h4 h5 h6
h4 -> h3 h5 h6
h5 -> h3 h4 h6
h6 -> h3 h4 h5
*** Results: 0% dropped (12/12 received)
mininet> 
```

Task 7 - Step 3

Verify that the bandwidth and the delay of each link comply with the values specified in the topology diagram shown in figure 1. Write in the lines below the commands you used and the results you obtained.

- `h3 ping h4 -c 10`

  Output:

  ```
  --- 10.0.0.2 ping statistics ---
  5 packets transmitted, 5 received, 0% packet loss, time 4006ms
  rtt min/avg/max/mdev = 23.186/25.264/26.760/1.507 ms
  ```

  The average RTT is approximately 20ms, which complies the link delays values specified in the topology diagram. The same proceeding can be used to test the delay of all the others links.

  It's important to point out that the link delay between the two switches doesn't comply with the topology diagram: this behaviour is however acceptable because it's due to the fact that in Mininet the switches are by default run in the same network namespace inside the kernel space, therefore they communicate to each other without using the link which has the performance parameters specified in the Python script.

- `iperf h3 h4`

  Output:

  ```
  *** Iperf: testing TCP bandwidth between h3 and h6
  *** Results: ['4.75 Mbits/sec', '6.23 Mbits/sec']
  ```

  The same proceeding can be used to test the bandwidth between all nodes.

Task 8 - Question 1

What are the advantages of having more controllers instead of one single controller which serves all the switches of the network?

- **More performance**: having more controllers makes each controller serve fewer nodes, therefore reducing its workload and improving the network performance. Moreover using more controllers makes it possible to choose where to position each controller: choosing a good placement increases the performance of the network by reducing the delay between each switch and the relative controller.
- **More fault tolerance**: the SDN controller is not a SPOF [1] anymore because the network includes more controllers, so if one of them fails the network can still work if there is at least another controller which can replace it. This can be achieved for example connecting each switch to multiple controllers or connecting the switches to a load balancer which acts like a "man in the middle" and forwards the requests that receives only to working controllers.

Task 8 - Question 2

Would the fault tolerance of the network shown in figure 1 change if only one controller (linked to both the switches) was used instead of two?

No because in this topology each switch is connected only to a single controller, therefore a failure of one of the two controllers prevents the network from working properly, just like in the case of a single controller connected to both switches.

[1] SPOF = single point of failure

Task 8 - Question 3

In this activity the topology shown in figure 1 was implemented assuming that the two controllers were local controllers. How would you have to change the Python script you created in this activity in order to use remote controllers instead of local ones? (Hint: see reference [5])

The changes that has to be applied to the Python script are marked with the red colour in the listing below.

```python
#!/usr/bin/Python
from mininet.net import Mininet
from mininet.node import Controller, OVSSwitch, RemoteController
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink

def multiControllerNet():
    net = Mininet( controller=Controller, switch=OVSSwitch, link=TCLink )

    info( "*** Creating hosts\n" )
    h1 = net.addHost('h3')
    h2 = net.addHost('h4')
    h3 = net.addHost('h5')
    h4 = net.addHost('h6')

    info( "*** Creating switches\n" )
    s1 = net.addSwitch( 's1' )
    s2 = net.addSwitch( 's2' )

    info( "*** Creating links\n" )
    net.addLink( h1, s1, bw=5, delay='5ms' )
    net.addLink( h2, s1, bw=5, delay='5ms' )
    net.addLink( h3, s2, bw=5, delay='5ms' )
    net.addLink( h4, s2, bw=5, delay='5ms' )
    net.addLink( s1, s2, bw=10, delay='2ms' )

    info( "*** Creating (reference) controllers\n" )
    c0 = net.addController( 'c0', controller=RemoteController, ip='127.0.0.1',
        port=6633 )
    c1 = net.addController( 'c1', controller=RemoteController, ip='127.0.0.1',
        port=6634 )

    info( "*** Starting network\n" )
    net.build()
    c0.start()
    c1.start()
    s1.start( [ c0 ] )
    s2.start( [ c1 ] )

    info( "*** Running CLI\n" )
    CLI( net )

    info( "*** Stopping network\n" )
    net.stop()

if __name__ == '__main__':
    setLogLevel( 'info' )
    multiControllerNet()
```

For simplicity it has been assumed that the two remote controllers are running on the same machine on which Mininet is running, therefore the ip `127.0.0.1` has been used (if the controllers had been on different machines the relative IP should have been used).

Task 8 - Question 4

How could we improve the fault tolerance of the network shown in figure 1 making minor changes to the Python script used?

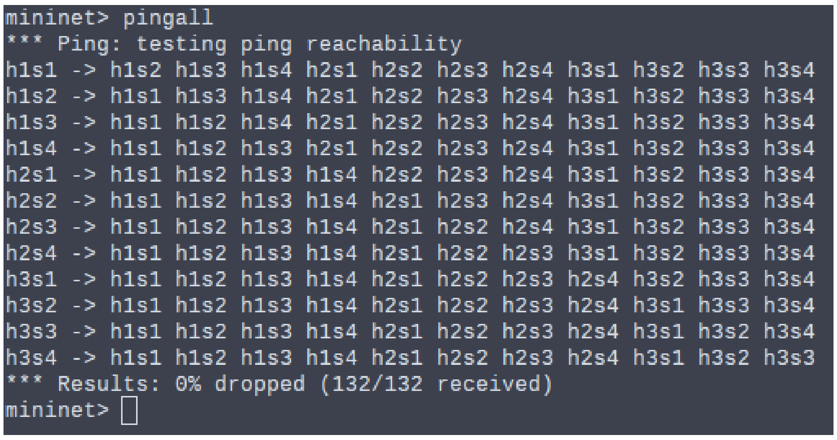
The easiest way is to connect each switch to both the controllers, so that if one controller fails the other one can still serve the switches keeping therefore the network working. It is possible to do that simply modifying the lines 36 and 37 of the Python script used in activity 1 (shown in listing 1) like this:

```
s1.start( [ c0, c1 ] )
s2.start( [ c0, c1 ] )
```

Activity 2

Task 5 - Step 2

Test the created topology: verify the network connectivity between all hosts. Write in the lines below the commands you used and the results you obtained.



```
mininet> pingall
*** Ping: testing ping reachability
h1s1 -> h1s2 h1s3 h1s4 h2s1 h2s2 h2s3 h2s4 h3s1 h3s2 h3s3 h3s4
h1s2 -> h1s1 h1s3 h1s4 h2s1 h2s2 h2s3 h2s4 h3s1 h3s2 h3s3 h3s4
h1s3 -> h1s1 h1s2 h1s4 h2s1 h2s2 h2s3 h2s4 h3s1 h3s2 h3s3 h3s4
h1s4 -> h1s1 h1s2 h1s3 h2s1 h2s2 h2s3 h2s4 h3s1 h3s2 h3s3 h3s4
h2s1 -> h1s1 h1s2 h1s3 h1s4 h2s2 h2s3 h2s4 h3s1 h3s2 h3s3 h3s4
h2s2 -> h1s1 h1s2 h1s3 h1s4 h2s1 h2s3 h2s4 h3s1 h3s2 h3s3 h3s4
h2s3 -> h1s1 h1s2 h1s3 h1s4 h2s1 h2s2 h2s4 h3s1 h3s2 h3s3 h3s4
h2s4 -> h1s1 h1s2 h1s3 h1s4 h2s1 h2s2 h2s3 h3s1 h3s2 h3s3 h3s4
h3s1 -> h1s1 h1s2 h1s3 h1s4 h2s1 h2s2 h2s3 h2s4 h3s2 h3s3 h3s4
h3s2 -> h1s1 h1s2 h1s3 h1s4 h2s1 h2s2 h2s3 h2s4 h3s1 h3s3 h3s4
h3s3 -> h1s1 h1s2 h1s3 h1s4 h2s1 h2s2 h2s3 h2s4 h3s1 h3s2 h3s4
h3s4 -> h1s1 h1s2 h1s3 h1s4 h2s1 h2s2 h2s3 h2s4 h3s1 h3s2 h3s3
*** Results: 0% dropped (132/132 received)
mininet> 
```

Task 5 - Step 3

Verify the bandwidth and the delay between the hosts h1s1 and h3s4. Write in the lines below the commands you used and the results you obtained.

- `iperf h1s1 h3s4`

  Result:

```
*** Iperf: testing TCP bandwidth between h1s1 and h3s4
*** Results: ['4.25 Gbits/sec', '4.26 Gbits/sec']
```

- `h1s1 ping h3s4 -c 10`

  Result:

  ```
  --- 10.0.0.12 ping statistics ---
  10 packets transmitted, 10 received, 0% packet loss, time 9006ms
  rtt min/avg/max/mdev = 0.107/4.042/24.062/7.490 ms
  ```

Task 6 - Question 1

In this activity you implemented a network with multiple controllers using a custom switch class and the high-level API. Which are the advantages of using this method instead of the one used in Activity 1? What about the disadvantages?

The advantage of the high-level API is that it allows to write reusable code by giving the possibly to create parametrized topology templates. Each template can be used for easily creating different (and eventually complex) topologies based on it.

The main disadvantages are the higher complexity and the limitations in configuring the network's nodes and links.

Task 6 - Question 2

In this activity the topology shown in figure 2 was implemented assuming that all the controllers were local controllers. How would you have to change the Python script you created in this activity in order to use remote controllers instead of local ones?

It is necessary only to import the class `RemoteController` and modify the lines 8,9,10,11 like this:

```
c0 = RemoteController( 'c0', ip='127.0.0.1', port=6633 )
c1 = RemoteController( 'c1', ip='127.0.0.1', port=6634 )
c2 = RemoteController( 'c2', ip='127.0.0.1', port=6635 )
c3 = RemoteController( 'c3', ip='127.0.0.1', port=6636 )
```

Note that it has been assumed that the two remote controllers are running on the same machine on which Mininet is running, therefore the ip `127.0.0.1` has been used.

Task 6 - Question 3

Try to modify the script you realized in order to have only two controllers instead of four, each one linked to two different switches. Each switch must be linked to a controller.

The lines of code that has to be changed are showed in red in the script below.

```
#!/usr/bin/Python
from mininet.net import Mininet
from mininet.node import OVSSwitch, Controller
from mininet.topo import LinearTopo
from mininet.log import setLogLevel
from mininet.cli import CLI


c0 = Controller( 'c0', port=6633 )
c1 = Controller( 'c1', port=6634 )

controllers = [c0, c1]
cmap = { 's1': c0, 's2': c0, 's3': c1, 's4' : c1 }



class MultiSwitch( OVSSwitch ):
  def start( self, controllers ):
    return OVSSwitch.start( self, [ cmap[ self.name ] ] )


def multiControllerNet():
  topo = LinearTopo( k=4, n=3 )
  net = Mininet( topo=topo, switch=MultiSwitch, build=False)

  for c in controllers:
    net.addController(c)

  net.build()
  net.start()
  CLI( net )
  net.stop()

if __name__ == '__main__':
  setLogLevel( 'info' )
  multiControllerNet()
```

Challenge solution

Task 1

One possible solution to the task 1 is implementing the network using a Python script and the Mininet middle-level API. The script is shown in the listing below.

```
#!/usr/bin/Python
from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info

def multiControllerNet():
    net = Mininet( controller=Controller, switch=OVSSwitch )

    info( "*** Creating hosts\n" )
    h1 = net.addHost('h1')
    h2 = net.addHost('h2')
    h3 = net.addHost('h3')
    h4 = net.addHost('h4')
    h5 = net.addHost('h5')
    h6 = net.addHost('h6')
    h7 = net.addHost('h7')
    h8 = net.addHost('h8')

    info( "*** Creating switches\n" )
    s1 = net.addSwitch( 's1' )
    s2 = net.addSwitch( 's2' )
    s3 = net.addSwitch( 's3' )
    s4 = net.addSwitch( 's4' )
    s5 = net.addSwitch( 's5' )
    s6 = net.addSwitch( 's6' )
    s7 = net.addSwitch( 's7' )

    info( "*** Creating links\n" )

    net.addLink( h1, s4 )
    net.addLink( h2, s4 )

    net.addLink( h3, s5 )
    net.addLink( h4, s5 )

    net.addLink( h5, s6 )
    net.addLink( h6, s6 )

    net.addLink( h7, s7 )
    net.addLink( h8, s7 )


    net.addLink( s4, s2 )
    net.addLink( s5, s2 )
    net.addLink( s6, s3 )
    net.addLink( s7, s3 )


    net.addLink( s2, s1 )
    net.addLink( s3, s1 )



    info( "*** Creating (reference) controllers\n" )
    c0 = net.addController( 'c0', controller=RemoteController, ip='127.0.0.1',
        port=6635 )
    c1 = net.addController( 'c1', controller=RemoteController, ip='127.0.0.1',
        port=6636 )
    c2 = net.addController( 'c2', port=6633 )
    c3 = net.addController( 'c3', port=6634 )


    info( "*** Starting network\n" )
    net.build()
    c0.start()
    c1.start()
    c2.start()
    c3.start()

    s1.start( [ c0, c1 ] )
    s2.start( [ c0, c1 ] )
    s3.start( [ c0, c1 ] )
    s4.start( [ c2 ] )
    s5.start( [ c2 ] )
    s6.start( [ c3 ] )
    s7.start( [ c3 ] )

    info( "*** Running CLI\n" )
    CLI( net )

    info( "*** Stopping network\n" )
    net.stop()

if __name__ == '__main__':
    setLogLevel( 'info' )
    multiControllerNet()
```

Task 2

These are the commands for connecting the switches served by `c3` to the controller `c2`:

```
sudo ovs-vsctl set-controller s6 tcp:127.0.0.1:6635
sudo ovs-vsctl set-controller s7 tcp:127.0.0.1:6635
```

References

[1] Allouzi, M. (2015), *Programming Assignment 2: Using Mininet and Mininet Python API: Instructions*, lecture notes, Software defined networking Kent State University, Available at: http://www.cs.kent.edu/~mallouzi/Software%20Defined%20Networking/Assignment2.pdf

[2] Blial, O., Ben Mamoun, M. and Benaini, R. (2016). An Overview on SDN Architectures with Multiple Controllers. *Journal of Computer Networks and Communications*, pp.1-8.

[3] Hai, N. and Kim, D. (2016). Efficient load balancing for multi-controller in SDN-based mission-critical networks. *2016 IEEE 14th International Conference on Industrial Informatics (INDIN)*, pp.420-425

[4] Phemius, K., Bouet, M. and Leguay, J. (2014). DISCO: Distributed SDN controllers in a multi-domain environment. *2014 IEEE Network Operations and Management Symposium (NOMS).*

[5] GitHub. (n.d.). *Introduction to Mininet.* [online]. Available at: https://github.com/mininet/mininet/wiki/Introduction-to-Mininet#controllers [Accesed 3 Apr. 2018].

[6] GitHub. (2016). *mininet.* [online]. Available at: https://github.com/mininet/mininet/blob/master/examples/controllers2.py [Accessed 28 Mar. 2018].

[7] GitHub. (2016). *mininet.* [online]. Available at: https://github.com/mininet/mininet/blob/master/examples/controllers.py [Accessed 28 Mar. 2018].

[8] Wiki.opendaylight.org. (n.d.). *OpenDaylight OpenFlow Plugin::Mininet with multiple controllers.* [online]. Available at: https://wiki.opendaylight.org/view/OpenDaylight_OpenFlow_Plugin::Mininet_with_multiple_controllers [Accessed 12 Apr. 2018].

[9] GitHub. (2014). *Reg:Mininet with multiple external Remote controllers - Issue #319.* [online]. Available at: https://github.com/mininet/mininet/issues/319 [Accessed 30 Mar. 2018]