# SDN controller clustering

Computer Networks module - SDN assignment

Michele Zanotti

Spring term 2018

# 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 use it to implement a certain topology.

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

- **Activity 1**: implement a network with multiple local controllers using a python script and the middle-level Mininet API.
- **Activity 2**: implement a network with multiple local controllers using a python script and the topology classes provided by the Mininet high-level API.
- **Activity 3**: implement a network with multiple controllers using *miniedit*, a tool provided by Mininet.

The paper also includes a fourth lab activity (**Activity 4**), which is a challange meant to let the reader test the knowledge acquired with the execution of the previous four activities.

It's important to point out that all the 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 has 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 [1], [2] and [].

The communication between different controllers can however be automatically managed by Mininet, the network emulator used in this 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. [ REFERENCE ???]

# Lab activity 1

## Topology diagram

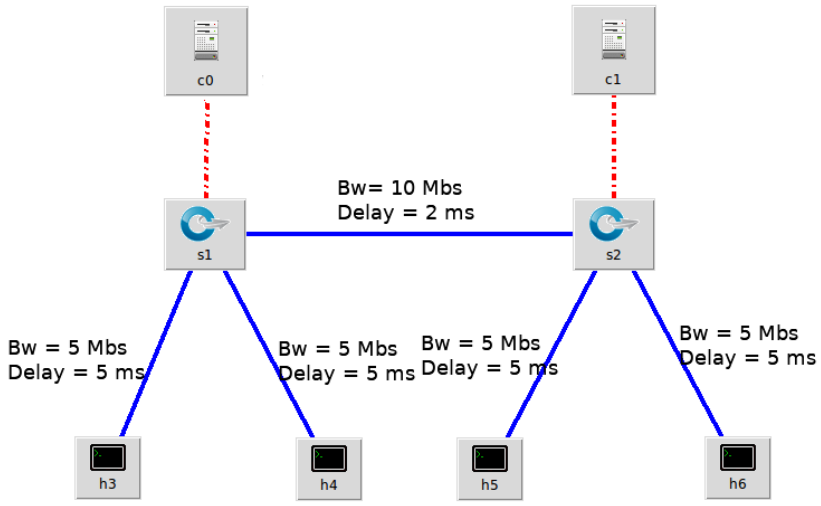


Figure 1: the topology that will be implemented during this lab activity. It is a simple linear topology with two switches connected to each other, each one connected to two hosts. Each switch is linked to a different SDN controller.

## Learning objectives

After finishing this lab activity you will be able to:

- Implement a cluster of local controllers inside Mininet using the Python middle-level Mininet API.
- Test the network connectivity and the performance of a network which includes a cluster of local controllers.
- Understand the main functions provided by the middle-level Mininet API required to implement a cluster of local 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 then 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. The two controllers shown in the topology diagram will be assumed local controllers. The topology has two different switches: each one will be connected to a different local controller.

Begin by creating a new Python script, then import 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 netowrk and add to it the required hosts, switches, links and controllers. After writing the script, execute it to create the network and test its connectivity and performance.

This lab activity assumes you are proficient in [...]. A basic knowledge of the Python programming language is also assumed.

## Task 1: write the skeleton of the Python script

### Step 1

Create a new Python script and edit with the text editor you prefer. If your 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
```

### 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 )
```

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

### 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 "*controllers-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

### Step 1

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

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

### Step 2

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

```
s1 = net.addSwitch('s1')
s2 = net.addSwitch('s2')
```

## Task 4: create links between nodes

### Step 1

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

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

### Step 2

Still inside the body of the function `multiControllerNet()`, create 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:

```
net.addLink( h3, s1, bw=10, delay='5ms' )
net.addLink( h4, s1, bw=10, delay='5ms' )
net.addLink( h5, s2, bw=10, delay='5ms' )
net.addLink( h6, s2, bw=10, delay='5ms' )
net.addLink( s1, s2, bw=20, 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 expresed 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).

## Task 5: create the controllers

### Step 1

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

```
info( "*** Creating (reference) controllers \n" )
```

### Step 2

Still inside the body of the function `multiControllerNet()`, add to the Mininet network the two required controller:

```
c0 = net.addController( 'c0', port=6633 )
c1 = net.addController( 'c1', port=6634 )
```

The function use to create the controllers is `addController`, which accept 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 wich 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()
```

## 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 controllers1.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 netowrk shown in figure 1 change if we used only one controller instead of two?**

**3 - How could we improve the fault tolerance of the netowrk shown in figure 1?**

Listing 1: 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

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

    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( h3, s1, bw=10, delay='5ms' )
    net.addLink( h4, s1, bw=10, delay='5ms' )
    net.addLink( h5, s2, bw=10, delay='5ms' )
    net.addLink( h6, s2, bw=10, delay='5ms' )
    net.addLink( s1, s2, bw=20, 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

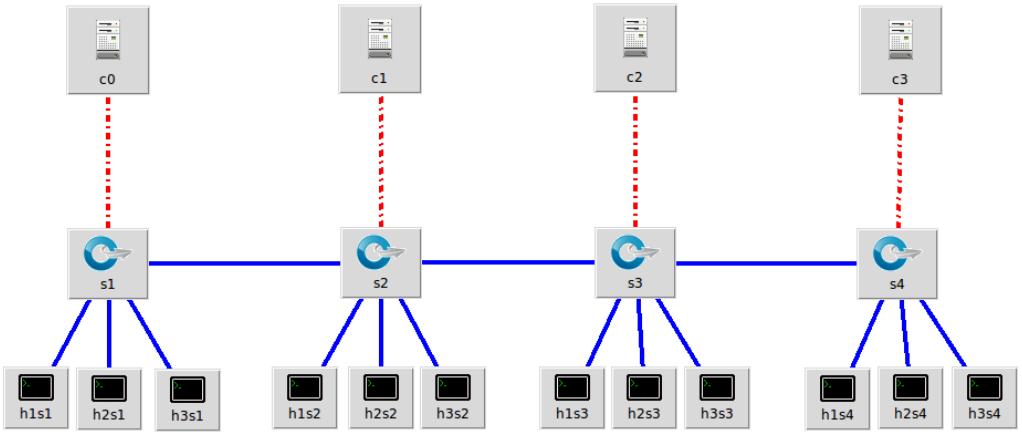


Figure 2: the topology that will be implemented during this lab activity. It is a linear topology with four switches connected to each other, each one connected to three hosts. Each switch is linked to a different SDN controller.

## Learning objectives

After finishing this lab activity you will be able to:

- Implement a cluster of local controllers inside Mininet using the topology classes provided by the Mininet high-level API.
- Define a custom switch class and use it to implement a network with multiple controllers.
- Reflect on the reasons for implementing a cluster of local controllers using the Mininet high-level API instead of the way shown in Activity 1.

## 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. The

two controllers shown in the topology diagram will be local controllers for this activity. The topology has two different switches: each one will be connected to a different local controller.

Begin by creating a new Python script, then import 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 netowrk and add to it the required hosts, switches, links and controllers. After writing the script, execute it to create the network and test its connectivity and performance.

This lab activity assumes you are proficient in [...]. A basic knowledge of the Python programming language is also assumed.

## Task 1: create the controllers

### Step 1

Create a new Python script and edit with the text editor you prefer. If your 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: create 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 `LinearTopology` provided by the Mininet 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

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

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

## 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.

Listing 2: complete Python script required for Activity 2

```
1  #!/usr/bin/Python
2  from mininet.net import Mininet
3  from mininet.node import OVSSwitch, Controller
4  from mininet.topo import LinearTopo
5  from mininet.log import setLogLevel
6  from mininet.cli import CLI
7
8  c0 = Controller( 'c0', port=6633 )
9  c1 = Controller( 'c1', port=6634 )
10 c2 = Controller( 'c2', port=6635 )
11 c3 = Controller( 'c3', port=6636 )
12
13 controllers = [c0, c1, c2, c3]
14 cmap = { 's1': c0, 's2': c1, 's3': c2, 's4' : c3 }
15
16
17 class MultiSwitch( OVSSwitch ):
18   def start( self, controllers ):
19     return OVSSwitch.start( self, [ cmap[ self.name ] ] )
20
21
22 def multiControllerNet():
23   topo = LinearTopo( k=4, n=3 )
24   net = Mininet( topo=topo, switch=MultiSwitch, build=False)
25
26   for c in controllers:
27     net.addController(c)
28
29   net.build()
30   net.start()
31   CLI( net )
32   net.stop()
33
34 if __name__ == '__main__':
35   setLogLevel( 'info' )
36   multiControllerNet()
```

# References

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

[2] K. Phemius, M. Bouet and J. Leguay, "DISCO: Distributed SDN controllers in a multi-domain environment", *2014 IEEE Network Operations and Management Symposium (NOMS)*, Krakow, 2014, pp. 1-2.