# Module 7 Programming Assignment: Instructions

Help

In this exercise, you will be working with the PyResonance controller that permits event-driven network control. PyResonance is an SDN controller that can dynamically react to various types of network events (e.g., intrusion detection, bandwidth limit reached, etc) by changing the applied network policy dynamically. Event-driven SDN control makes networks easier to manage by automating many tasks that are currently performed by manually modifying multiple distributed device configuration files, which are expressed in low-level, vendor-specific CLI commands. PyResonance augments the original Pyretic code base with many useful features to present an attractive and engaging environment for implementing an event-driven SDN controller. Moreover, Pyretic's modular programming model allows programmers to build a complex network policy by composing multiple network policies together (sequential or parallel). Unlike previous weeks, this week you will be taken through the steps of writing reactive network applications using PyResonance --- giving operators access to dynamically changing network policies --- and testing them using Mininet. The purpose of this exercise is to show you yet another way of writing dynamic network applications.

After the walkthrough, you will be asked to implement a simple server load-balancing application on PyResonance and test it using mininet. More details on creating and submitting the code will be provided later on in the instructions. So, as always, make sure that you follow each step carefully.

## Walkthrough

The network you'll use in this exercise includes 3 hosts and a switch, this time with the PyResonance runtime to express your network applications.

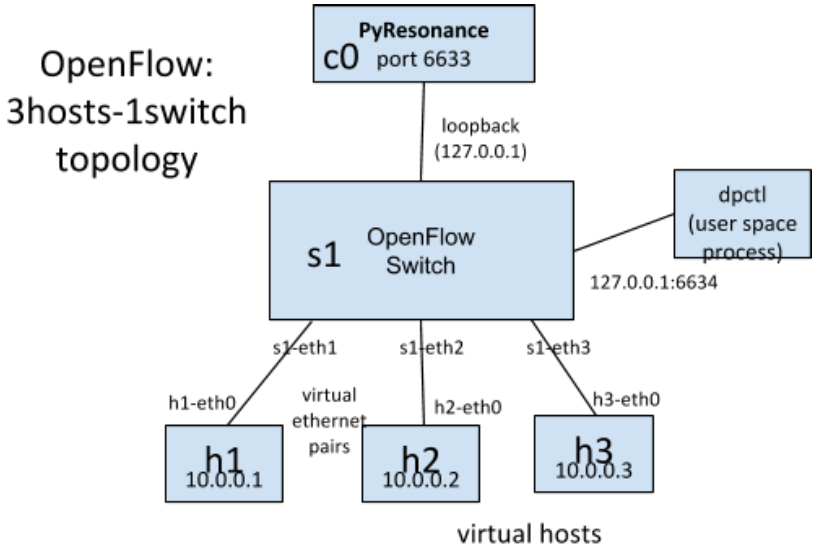


**Figure 1: Topology for the Network under Test**

## What is PyResonance?

```
Pyretic + Resonance = PyResonance
```

***Resonance*** is a SDN network management framework where operators define a network policy as a Finite State Machine (FSM). Transitions between states (or different forwarding decisions) are triggered by various types of dynamic events in the network, e.g., intrusion detection, authentication of hosts, data usage cap reached, etc. Based on different network events, operators can enforce different policies to the network using an intuitive FSM model.

***PyResonance*** is a Resonance platform implemented in Pyretic. You can build multiple network policies and compose them (sequential or parallel) together to express an overall network policy for the target network. For each network policy, you can have multiple states.

For more details on Pyretic, see http://resonance.noise.gatech.edu.

We will be using the PyResonance controller, so make sure that the default, POX or Pyretic controller is not running in the background. Also, confirm that the port '6633' used to communicate with OpenFlow switches by the runtime is not bounded:

```
$ sudo fuser -k 6633/tcp
```

This will kill any existing TCP connection, using this port.

You should also run `sudo mn -c` and restart Mininet to make sure that everything is clean.From you Mininet console:

```
mininet> exit
$ sudo mn -c
```

## Installing PyResonance on your VM

Make sure that you have Pyretic installed in your VM. We recommend using the VM, we have provided on the course website. It comes pre-installed with Mininet, POX and Pyretic.

In the VM, go to "`/home/mininet/pyretic/pyretic`".

```
$ cd /home/mininet/pyretic/pyretic
```

Make a clone of PyResonance code from Github.

```
$ git clone https://github.com/Resonance-SDN/pyresonance.git
```

Check if "`home/mininet/pyretic/pyretic/pyresonance`" directory exists.

```
$ ls home/mininet/pyretic/pyretic/pyresonance
```

Now run PyResonace module for testing. Move back to directory "`/home/mininet/pyretic`" and run:

```
$ python pyretic.py pyretic.pyresonance.resonance_main
```

Check if the command produces any errors. It should not output any error messages.

## A simple PyResonance example

In this simple example, you'll fire up an authentication policy module, which drops traffic from any host that is not authenticated.

First, fire up mininet with the following topology:

```
$ sudo mn --controller=remote --topo=single,3 --mac --arp
```

This will create a network topology as shown in figure 1.

Now, start the PyResonance "simple" application using the Pyretic runtime system:

```
$ pyretic.py pyretic.pyresonance.resonance_simple
```

In mininet, send a ping to host `h2` (with IP address 10.0.0.2) from host `h1`:

```
mininet> h1 ping -c3 h2
```

You should see that host `h1` is not able to reach host `h2`. This is because the traffic coming from these hosts is not authenticated.

Use JSON client to send JSON events to the Resonance Controller, to authenticate the two hosts:

```
$ ./sendy_json.py -i 10.0.0.1 -e auth -V authenticated
$ ./sendy_json.py -i 10.0.0.2 -e auth -V authenticated
```

This time the ping will pass and you should see replies coming back from host `h2`.

Now, lets look at the code:

```
from pyretic.lib.corelib import *
from pyretic.lib.std import *
from pyretic.modules.mac_learner import learn

from resonance_policy import *
from resonance_states import *
from resonance_handlers import EventListener
from multiprocessing import Process, Queue
import threading

DEBUG = True

""" Dynamic resonance policy """
def resonance(self):

  # updating policy
  def update_policy(pkt=None):
    self.policy = self.authPolicy.default_policy()
    if DEBUG is True:
      print self.policy
  self.update_policy = update_policy

  # Listen for state transitions.
```

```
def transition_signal_catcher(queue):
  while 1:
    try:
      line = queue.get_nowait() # or q.get(timeout=.1)
    except:
      continue
    else: # Got line.
      self.update_policy()

def initialize():

  # Create queue for receiving state transition notification
  queue = Queue()

  self.authFSM = AuthStateMachine()

  # initialize policies
  self.authPolicy = AuthPolicy(self.authFSM)

  # Spawn an EventListener, which listens for events and keeps track of
  # the state of each host.
  # default is:


  self.eventListener = EventListener(self.authFSM)
  self.eventListener.start(queue)

  t = threading.Thread(target=transition_signal_catcher, args=(queue,))
  t.daemon = True
  t.start()

  # Set a default policy
  self.update_policy()

 initialize()

""" Main Method """
def main():
 return dynamic(resonance)() >> dynamic(learn)()
```

**Table 1: Simple Resonance application**

## Using PyResonance policies

- First, in initialization, we create a Finite State Machine (for authentication in this example), and create a policy out of it. After that, we initialize policy objects (type `AuthPolicy`). The policy objects return methods that ultimately return Pyretic methods. The policy objects will return different methods depending on the state of the host.

```
self.authFSM = AuthStateMachine()
       self.authPolicy = AuthPolicy(self.authFSM)
```

- Create and start an `EventListener`. By default, the `EventListener` will create a single finite state machine to update when it processes events. You can also initialize `EventListener` with a state machine (type is `ResonanceStateMachine`). For example,

```
self.eventListener = EventListener(self.authFSM)
self.eventListener.start(queue)
```

● Start a thread that listens to any dynamic state transition. The thread's target method is "transition_signal_catcher()", which listens for writes in the shared "queue" and calls "update_policy()" whenever there is a state transition. (A state transition event writes some string in this shared queue.)

```
t = threading.Thread(target=transition_signal_catcher,
args=(queue,))
```

● "update_policy()" sets the Pyresonance's current network policy to an updated one. In the code `"self.authPolicy.default_policy()"` is the policy, where authPolicy.default_policy() is defined in "`resonance_policy.py`".

## Example with Sequential Composition (Advanced)

A more complicated example shows the power of having multiple FSMs to process events and then using sequential composition to apply policies to the traffic.

Start the Mininet Topology, as in the previous example:

```
$ sudo mn --controller=remote --topo=single,3 --mac --arp
```

Run the resonance "main" application:

```
$ pyretic.py pyretic.resonance.resonance_main
```

In mininet, send a ping to host `h2` (with IP address 10.0.0.2) from host `h1`:

```
mininet> h1 ping -c3 h2
```

You should see that host `h1` is not able to reach host `h2`. This is because the traffic coming from these hosts is not authenticated.

Use JSON client to send JSON events to the Resonance Controller, to authenticate the two hosts:

```
$ ./sendy_json.py -i 10.0.0.1 -e auth -V authenticated
$ ./sendy_json.py -i 10.0.0.2 -e auth -V authenticated
```

You will see that, sending a ping this time to host `h2` from `h1` also fails. This is because the policy on the packets is a sequential composition of the authentication policy and the IDS policy, no packets go through until both of these policies have a passthrough policy.

Send the following JSON events to the Resonance controller to mark the two hosts as clean:

```
$ ./sendy_json.py -i 10.0.0.1 -e ids -V clean
$ ./sendy_json.py -i 10.0.0.2 -e ids -V clean
```

This time the ping will pass and you should see replies coming back from host `h2`.

Here's the code for the "main" application:

```
from pyretic.lib.corelib import *
```

```
from pyretic.lib.std import *
from pyretic.modules.mac_learner import learn

from resonance_policy import *
from resonance_states import *
from resonance_handlers import EventListener
from multiprocessing import Process, Queue
import threading

DEBUG = True

""" Dynamic resonance policy """
def resonance(self):

 # updating policy
 def update_policy(pkt=None):
  # sequential composition of auth policy, then IDS policy
  self.policy = (self.authPolicy.default_policy() >> self.idsPolicy.default_policy())
 self.update_policy = update_policy

 # Listen for state transitions.
 def transition_signal_catcher(queue):
  while 1:
   try:
    line = queue.get_nowait() # or q.get(timeout=.1)
   except:
    continue
   else: # Got line.
    self.update_policy()

 def initialize():

  # Create queue for receiving state transition notification
  queue = Queue()

  # Spawn an EventListener, which listens for events and keeps track of
  # the state of each host.

  self.authFSM = AuthStateMachine()
  self.eventListener = EventListener(self.authFSM)

  self.IDSFSM = IDSStateMachine()
  self.eventListener.add_fsm(self.IDSFSM)

  self.eventListener.start(queue)

  # initialize policies
  self.authPolicy = AuthPolicy(self.authFSM)
  self.idsPolicy = IDSPolicy(self.IDSFSM)

  t = threading.Thread(target=transition_signal_catcher, args=(queue,))
  t.daemon = True
  t.start()
```

```
    # Set a default policy
    self.update_policy()

  initialize()


""" Main Method """
def main():
  return dynamic(resonance)() >> dynamic(learn)()
```

**Table 2: Resonance application with Sequential composition**

## Assignment

Server Load balancing improves network performance by distributing traffic efficiently so that individual servers are not overwhelmed by sudden fluctuations in activity. In this assignment, your task is to implement a simple server load balancing application using PyResonance. The network, as shown in the figure below, will have three hosts (`h1`, `h2` and `h3`) directly connected to the switch `s1`. `h1` and `h3`, are connected to the switch over a very low latency link (i.e. delay of approx., 0 ms). Whereas, `h2` is connected using a link with some significant delays (i.e., 100ms). The host `h1` sends a ping request to a public IP address (10.0.0.100) which based on the policy configured on the PyResonance controller (details discussed later in the instructions) will direct the traffic to either host `h2` (10.0.0.2) or `h3` (10.0.0.3).

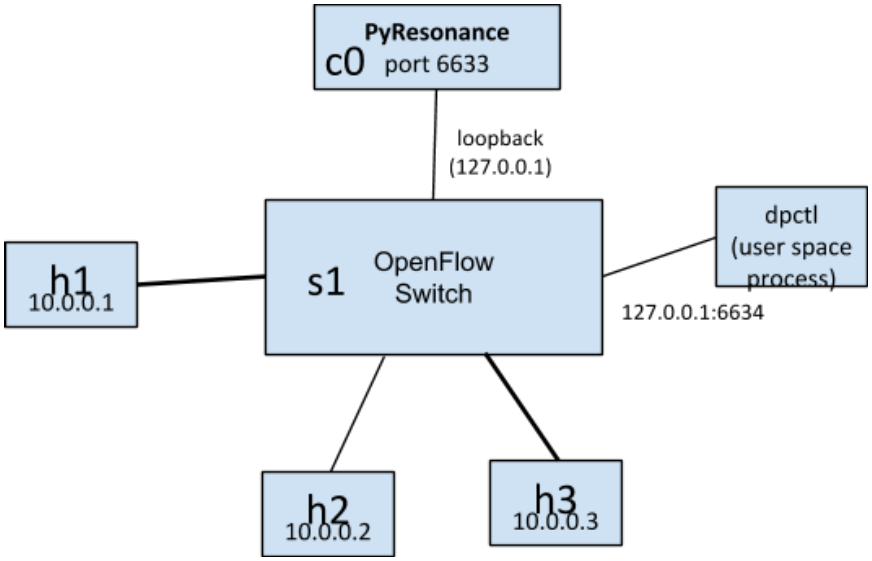


Figure 2: Topology for the Network under Test used in this assignment

### Understanding the Code

To start this exercise, download module7-assignment1.zip. It consists of following files:

- `resonance_main.py`: the main module for the resonance application.
- `resonance_handler.py`: the module for receiving policies as JSON messages from `sendy_json.py`
- `resonance_policy.py`: placeholder for the resonance policy classes and modules

- `resonance_states.py`: implements the resonance Finite State Machine
- `sendy_json.py`: used for sending policies as JSON messages to the resonance application.
- `MyTopo.py`: Creates a simple topology and sends a ping request to public IP (10.0.0.100)
- `submit.py`: used to submit your code and output to the coursera servers for grading.

You don't have to do any modifications in `resonance_main.py`, `resonance_hanlder.py`, `sendy_json.py`, `MyTopo.py` and `submit.py`.

You will update the `resonance_policy.py` and `resonance_states.py`. They are populated with a skeleton code.

The `resonance_policy.py` contains a `default_policy` fucntion in the `LBPolicy` class. This class also contains two more functions for portA and portB that contain policies which will be applied based on the port selected by the `default_policy` function. Here, your task is to update the `default_policy` function to select the right port policy for a given packet by matching against the list of hosts configured for each port.

The `resonance_states.py` has a class `LBStateMachine`, that contains three functions. The `handleMessage` function parses the JSON messages and based on the output, performs the state transition and assigns a host to either portA or portB. The two other functions `get_portA_hosts` and `get_portB_hosts` are used to get the list of hosts assigned to a given state (i.e., portA or portB state). Your task is to update the `handleMessage` function to perform the state transition based on the input received in the JSON message. Also, you have to update the `get_portA_hosts` and `get_portB_hosts` to return the list of hosts associated with each port (i.e., state).

## Testing your Code

Once you have your code, copy its content under the `~/pyretic/pyretic/pyresonance` directory on your VM.

Run PyResonance controller application:

```
$ cd ~/pyretic
$ pyretic.py pyretic.pyresonance.resonance_main &
```

Now, open another terminal and run mininet:

```
$ cd ~/pyretic/pyretic/pyresonance
$ sudo python MyTopo.py
```

This will run a simple mininet topology. The `MyTopo.py` script will ping the public IP (10.0.0.100) from host (`h1`):

```
mininet> h1 ping -c9 10.0.0.100
```

What do you see? If everything has be done and setup correctly then you should see the following output:

```
*** h1 : ('ping', '-c9', '10.0.0.100')
```

```
PING 10.0.0.100 (10.0.0.100) 56(84) bytes of data.
64 bytes from 10.0.0.100: icmp_req=1 ttl=64 time=1190 ms
64 bytes from 10.0.0.100: icmp_req=2 ttl=64 time=222 ms
64 bytes from 10.0.0.100: icmp_req=3 ttl=64 time=21.8 ms
64 bytes from 10.0.0.100: icmp_req=4 ttl=64 time=0.037 ms
64 bytes from 10.0.0.100: icmp_req=5 ttl=64 time=0.024 ms
64 bytes from 10.0.0.100: icmp_req=6 ttl=64 time=0.094 ms
64 bytes from 10.0.0.100: icmp_req=7 ttl=64 time=0.097 ms
64 bytes from 10.0.0.100: icmp_req=8 ttl=64 time=0.037 ms
64 bytes from 10.0.0.100: icmp_req=9 ttl=64 time=0.001 ms

--- 10.0.0.100 ping statistics ---
9 packets transmitted, 9 received, 0% packet loss, time 8007ms
```

By default, the public IP (10.0.0.100) is mapped to host h3 (10.0.0.3) which is connected to the switch over a link with a delay value of approx., 0ms. That's why the RTT value settles down around approx., 0 ms.

Now, on a different terminal, execute the following command using `sendy_json.py` to shift the traffic coming from host `h1` (10.0.0.1) to be receveid by host `h2` (10.0.0.2), when `h1` sends a ping request to public IP address (10.0.0.100).

```
$ ./sendy_json.py -i 10.0.0.1 -e lb -V portA
```

Run the `MyTopo.py` again and this time the output will be:

```
*** h1 : ('ping', '-c9', '10.0.0.100')
PING 10.0.0.100 (10.0.0.100) 56(84) bytes of data.
64 bytes from 10.0.0.100: icmp_req=1 ttl=64 time=1531 ms
64 bytes from 10.0.0.100: icmp_req=2 ttl=64 time=558 ms
64 bytes from 10.0.0.100: icmp_req=3 ttl=64 time=250 ms
64 bytes from 10.0.0.100: icmp_req=4 ttl=64 time=201 ms
64 bytes from 10.0.0.100: icmp_req=5 ttl=64 time=200 ms
64 bytes from 10.0.0.100: icmp_req=6 ttl=64 time=204 ms
64 bytes from 10.0.0.100: icmp_req=7 ttl=64 time=204 ms
64 bytes from 10.0.0.100: icmp_req=8 ttl=64 time=203 ms
64 bytes from 10.0.0.100: icmp_req=9 ttl=64 time=200 ms

--- 10.0.0.100 ping statistics ---
9 packets transmitted, 9 received, 0% packet loss, time 8020ms
```

The link connecting `h2` to the switch has a very high delay value of 100ms, which produces an RTT of approx., 200 ms. Note, h1 connects to the switch with a delay value of approx., 0ms, thus doesn't add much to the overall RTT.

## Submitting your Code

Run PyResonance controller application:

```
$ cd ~/pyretic
$ pyretic.py pyretic.pyresonance.resonance_main &
```

Configure the following policy using `sendy_json.py`:

```
$ ./sendy_json.py -i 10.0.0.1 -e lb -V portA
```

### Method 1:

To submit your code, run the submit.py script, under the `~/pyretic/pyretic/pyresonance` directory:

```
$ cd ~/pyretic/pyretic/pyresonance
$ sudo python submit.py
```

Your mininet VM should have internet access by default, but still verify that it has internet connectivity (i.e., eth0 set up as NAT). Otherwise `submit.py` will not be able to post your code and output to our coursera servers.

The submission script will ask for your login and password. This password is not the general account password, but an assignment-specific password that is uniquely generated for each student. You can get this from the assignments listing page.

Once finished, it will prompt the results on the terminal (either passed or failed).

### Method 2: (alternative for those not able to submit using the above script)

Download and run the following script:

```
$ wget https://d396qusza40orc.cloudfront.net/sdn/srcs/m7-output.py
$ sudo python m7-output.py
```

This will create an `output.log` file in the same folder. Upload this log file on `coursera` using the `submit` button for Module 7 under the Week 6 section on the Programming Assignment page.

Once uploaded, it will prompt the results on the new page (either passed or failed).

Note, if during the execution `submit.py` script crashes for some reason or you terminate it using CTRL+C, make sure to clean mininet environment using:

```
$ sudo mn -c
```

Also, if it still complains about the controller running. Execute the following command to kill it:

```
$ sudo fuser -k 6633/tcp
```

---

* Part of these instructions are adapted from http://resonance.noise.gatech.edu.