In this exercise, you will be working with the Kinetic controller, which permits event-driven network control.

Kinetic is an SDN control module 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.

Kinetic 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 Kinetic—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.

Another benefit to Kinetic is that its dynamic behavior is verifiable. A network operator can write dynamic network policies in terms of a Finite State Machine (FSM), and these policies are automatically translated into a form that a model checker can verify for certain properties. For example, it is possible to verify certain properties such as “when an intrusion detection system indicates that a host is infected, the host will always be blocked from the network”

After the walkthrough, you will be asked to implement a simple server load-balancing application on Kinetic 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, you will use the Kinetic controller to express your network applications.

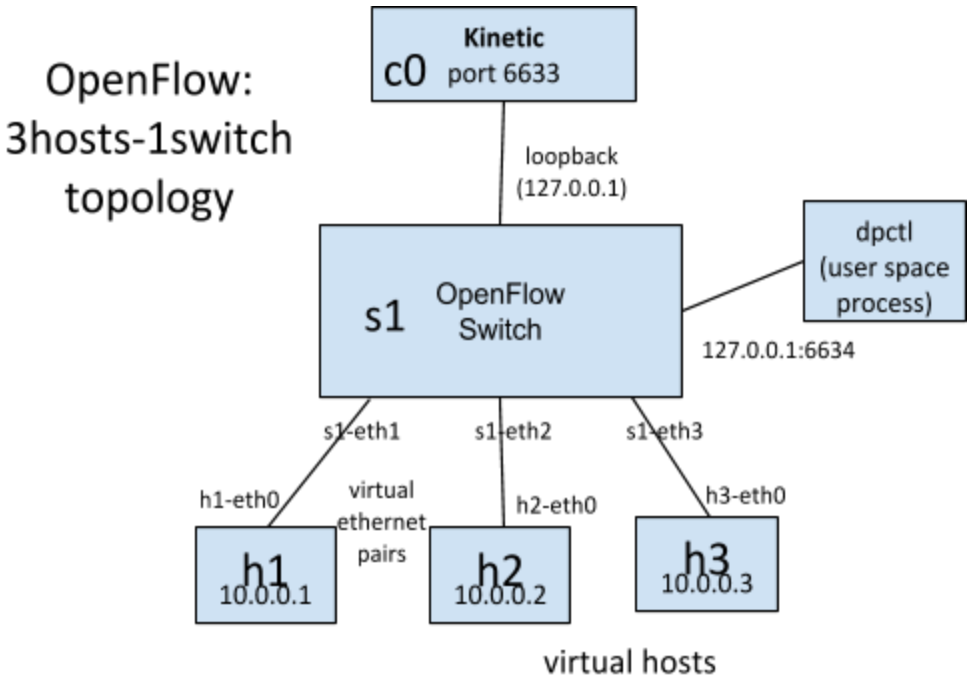


Figure 1: Topology for the Network under Test

What is Kinetic?

Kinetic is an SDN control framework where operators define a network policy as a Finite State Machine (FSM). Transitions between states are triggered by different 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. ***Kinetic*** is implemented as a Pyretic module. 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.

A Kinetic control program permits programmer-defined events to dynamically change forwarding behavior for an arbitrary set of flows. Such events can range from topology changes (generated by the Pyretic runtime) to security incidents (generated by an intrusion detection system). The programmer specifies an FSM description that contains set of states, each of which maps to some network behavior that are encoded using Pyretic's policy language; and a set of transitions between those states, each of which may be triggered by events that the operator defines.

For more details on Kinetic, see http://kinetic.noise.gatech.edu.

We will be using the Kinetic 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 Kinetic on Your VM

Make sure that you have Pyretic and 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/`".

```
$ cd ~/pyretic
```

Make sure you have the latest Kinetic branch.

```
$ git pull
$ git checkout kinetic
```

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

```
$ ls home/mininet/pyretic/pyretic/kinetic
```

Now run Kinetic module for testing.

Before running any kinetic application, we must issue this
`$export KINETICPATH=$HOME/pyretic/pyretic/kinetic`

or add the following line in ~/.bashrc to make it permanent.
`export KINETICPATH=$HOME/pyretic/pyretic/kinetic`

Move back to directory "`/home/mininet/pyretic`" and run:

```
$ python pyretic.py pyretic.kinetic.apps.ids
```

Check if this command produces any errors. It should start a controller and not output any error messages.

A Simple Kinetic 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 Kinetic "simple" application using the Pyretic runtime:

```
$ pyretic.py pyretic.kinetic.apps.ids
```

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` can reach host `h2`. This is because the traffic coming from these hosts is not authenticated.

Use the JSON client included with Kinetic to send JSON events to the Kinetic controller to indicate that host h1 is infected:

```
$ python json_sender.py -n infected -l True --flow="{srcip=10.0.0.1}"
-a 127.0.0.1 -p 50001
```

This command sends a JSON message to the Kinetic controller indicating that host `h1` is infected. The Kinetic control program you are running has a Finite State Machine (FSM) policy that says when a host corresponding to the flow space with a specific source IP address transitions to an infected state, the policy for that part of flow space should transition from "passthrough" to "drop".

Sending an event that indicates that the host is no longer infected should cause `h1` to no longer be blocked:

```
$ python json_sender.py -n infected -l False
--flow="{srcip=10.0.0.1}" -a 127.0.0.1 -p 50001
```

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

Let's have a closer look at the code:

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

```
from pyretic.kinetic.fsm_policy import *
from pyretic.kinetic.drivers.json_event import JSONEvent
from pyretic.kinetic.smv.model_checker import *

class auth(DynamicPolicy):
    def __init__(self):

        ### DEFINE THE LPEC FUNCTION
        def lpec(f):
            return match(srcip=f['srcip'])

        ## SET UP TRANSITION FUNCTIONS
        @transition
        def authenticated(self):
            self.case(occurred(self.event),self.event)

        @transition
        def policy(self):
            self.case(is_true(V('authenticated')),C(identity))
            self.default(C(drop))

        ### SET UP THE FSM DESCRIPTION
        self.fsm_def = FSMDef(
            authenticated=FSMVar(type=BoolType(),
                                 init=False,
                                 trans=authenticated),
            policy=FSMVar(type=Type(Policy,{drop,identity}),
                          init=drop,
                          trans=policy))

        ### SET UP POLICY AND EVENT STREAMS
        fsm_pol = FSMPolicy(lpec,self.fsm_def)
        json_event = JSONEvent()
        json_event.register_callback(fsm_pol.event_handler)

        super(auth,self).__init__(fsm_pol)

def main():
```

```
    pol = auth()
    return pol >> flood()
```

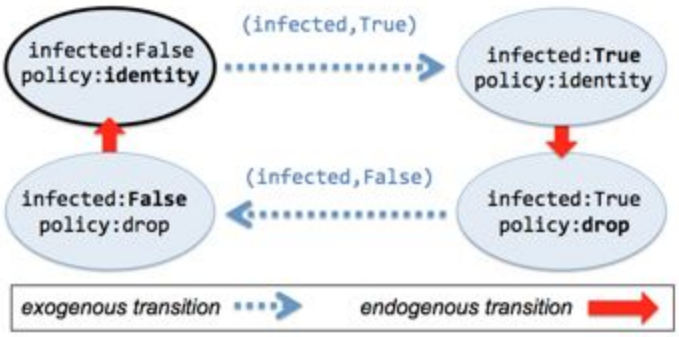
Table 1: Simple Kinetic IDS application.

Notice how short and concise the code is. We will now walk through the application to help explain how to write a Kinetic control program.

Writing Kinetic Control Programs

A Kinetic program has several parts:

- **The LPEC function.** An LPEC refers to the maximal set of located packets (i.e., flowspace) that one independent FSM instance will handle. A programmer can represent a LPEC using any Pyretic filter. For example, the LPEC containing all packets whose source IP address is 10.0.0.1 can be expressed simply as `match(srcip=IPAddr('10.0.0.1'))`.
- **Transition functions.** A transition function encodes logic that indicates the new value a variable should take when a particular event arrives at the controller. For example, the that the infected transition function encodes contains a single case: when an infected event occurs, the new value taken by the infected variable is the value of that event. Changes to any given FSM state variable (i.e., transitions) are triggered for one of two reasons:
    - exogenous, the arrival of an external event upon which this variable's value depends; and
    - endogenous, the change of another variable upon which this variable's value depends.
- **FSM definition.** The FSM associates the transition functions that we define with the appropriate state variables. The FSM definition consists of a set of state variable definitions. Each variable definition simply specifies the variable's type (i.e., set of allowable values), initial value, and associated transition functions. The `infected` variable is a boolean whose initial value is False (representing the assumption that hosts are initially uninfected), and transitions based on the infected function defined previously. Likewise, the `policy` variable can take the values drop or identity, initially starts off in the identity state, and transitions based on the policy function defined previously.
- **Policy and event streams.** The `FSMPolicy` that Kinetic provides automatically directs each incoming event to the appropriate LPEC FSM, where it will be handled by the exogenous transition function specified in the FSM description (e.g., `fsm_desc` above).



Example with Sequential Composition

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 Kinetic “main” application:

```
$ pyretic.py pyretic.kinetic.examples.auth_ml_ids
```

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 yet authenticated.

Use JSON client to send JSON events to the Kinetic controller, to authenticate the two hosts:

```
$ cd ~/pyretic/pyretic/kinetic
$ python json_sender.py -n authenticated -l True
--flow="{srcip=10.0.0.1}" -a 127.0.0.1 -p 50001
$ python json_sender.py -n authenticated -l True
--flow="{srcip=10.0.0.2}" -a 127.0.0.1 -p 50001
```

You will see that sending a ping this time to host `h2` from `h1` now succeeds.

Similarly, you can block traffic as before, with an IDS event:

```
$ python json_sender.py -n infected -l True --flow="{srcip=10.0.0.1}"
-a 127.0.0.1 -p 50002
```

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.

Note: The "infected" event should be sent to port 50002. Port 50001 is solely used by the auth application in this case. There is one TCP port number assigned per FSM, which is incremented by one as new FSMs that take external events are instantiated. (Yes, this is not the most intuitive behavior, and we are working to address this in future versions!)

Resetting the infected variable to false will again allow traffic to pass from `h1` to other hosts on the network.

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

```
from pyretic.kinetic.apps.ids import ids
from pyretic.kinetic.apps.mac_learner import mac_learner
from pyretic.kinetic.apps.auth import auth


def main():
    return auth() >> mac_learner() >> ids()
```

Table 2: Kinetic application with sequential composition.

Composing event-driven FSM policies is as easy as it was in Pyretic. Simply use Pyretic's sequential composition policies to compose policies that you have already specified. In this particular example, we compose the authentication policy above with a MAC learning switch, followed by the IDS policy. As it turns out, you can also write a MAC learning switch in Kinetic. The policy above is actually a Kinetic policy that changes states in conjunction with topology events. You can see that policy here.

Model Checking and Verification with Kinetic

You'll also note that the example IDS application has some logical assertions that the Kinetic controller attempts to verify using a model checker called NuSMV. There are three statements that are listed in the example code.

The first statement says that an infected event for some LPEC should result in the next policy being "drop" (i.e., `policy_1`). The second statement says that if an infected variable transitions to false, the next policy should be "passthrough" (i.e., `policy_2`). The final statement says that the default policy should be "allow" until the infected variable becomes true.

```
    ### If infected event is true, next policy state is 'drop'
```

```
mc.add_spec("SPEC AG (infected -> AX policy=policy_1)")

### If infected event is false, next policy state is 'allow'
mc.add_spec("SPEC AG (!infected -> AX policy=policy_2)")

### Policy state is 'allow' until infected is true.
mc.add_spec("SPEC A [ policy=policy_2 U infected ]")
```

Below is the set of NuSMV logical operators that can help you understand the modifiers A, G, X, and U above. The logic is a little tricky and takes some getting the hang of. The trick with reasoning about CTL is to realize that all computation paths are represented as a tree from the current state. Any transition will result in transitioning down the tree to another node. A "path" in CTL is not a network path, but rather an execution path in this tree. So, in the table below, "all paths from the current state" basically means "for all nodes on paths in the tree rooted at the current state".

	(Quantifiers over Groups of Paths)
A ϕ	ϕ holds for all possible paths from the current state.
E ϕ	There exists a paths from the current state where ϕ holds.
	(Quantifiers over a Specific Path)
X ϕ	ϕ holds for ne**X**t state.
F ϕ	ϕ eventually holds sometime in the **F**uture.
G ϕ	ϕ holds for all current and following states, **G**lobally.
ϕ U ψ	ϕ holds at least **U**ntil ψ.

So, reading for example the first SMV specification literally, you might say "for all paths globally from the current state, an infected transition should always result in the policy becoming `policy_2` as the next state from the current state". You should practice reading and writing these logical statements. This additional background on computation tree logic (CTL) may also be useful.

The other inconvenience is that you first have to write your policy to determine that `policy_1` corresponds to drop and `policy_2` corresponds to allow.

Assignment

The assignment has two parts. You'll first use Kinetic to make small modifications to the authentication module (and write some rules to verify that it is correct). You'll then use Kinetic to implement some server load balancing logic (and verify the correctness of your logic). To start this exercise, download module8-assignment1.zip. All parts of this assignment use the topolgy below. Put these files in a directory called `~/pyretic/pyretic/kinetic/ext`.

You will use the following files:

- `gardenwall.py`: Skeleton code for the gardenwall application.

- `primary_backup.py`: Skeleton code for the primary/backup
- `submit.py`: used to submit your code and output to the coursera servers for grading.

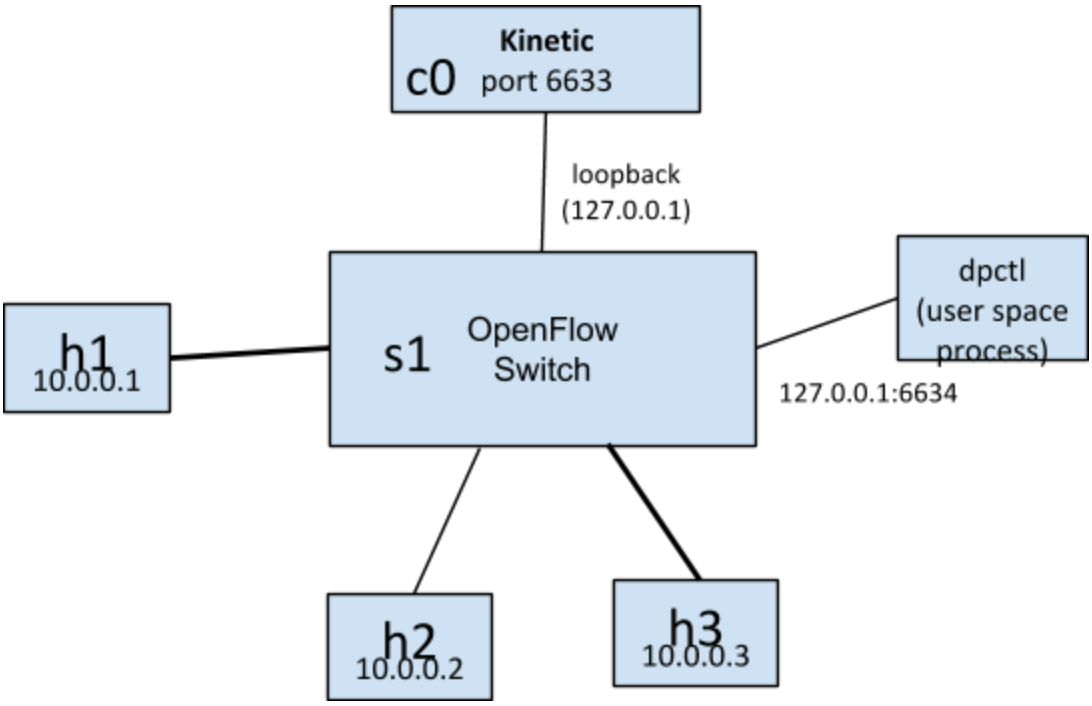


Figure 2: Topology for the network under test for this assignment.

Part 1: Host Gardenwall

In this part of the assignment, you will augment the IDS module that we have been studying so that some hosts might be classified as “exempt” from blocking, even if they transition to an infected state. In our simple example, the hosts that are in an “exempt” but “infected” state will be redirected to different server whenever they attempt to access any destination. An application of this example might be in an enterprise network setting, where infected hosts belonging to guests are simply cut off of the network entirely, but hosts belonging to employees, students, etc. are instead redirected to a “garden wall”.

Your example application should have the following modifications:

- Modify the finite state machine to have an “exempt” state, in addition to the “infected” state. (So, a host can either be infected or not, and exempt or not.)
- Create a new policy using Pyretic, gardenwall, which modifies traffic in the example topology to redirect all traffic from an infected but exempt host to *any* destination to host 10.0.0.3 (let’s assume that 10.0.0.3 is a host where the gardenwall server would be located).
- Write and check your CTL logic to verify that your host machine satisfies the following table of policies:

	Infected	**Not Infected**
Exempt	“gardenwall”: Redirect to 10.0.0.3 (rewrite dst IP)	“identity” (allow)
Not Exempt	“drop”: Drop	“identity” (allow)

The following function will be helpful:

- rewriteDstIPAndMAC(): This is to change the dstip addres to a specific IP address *__whatever the dstip is__* (more exactly, if the dstip matches any of the given client IPs).

Testing Your Code

Run your Kinetic controller application:

```
$ cd ~/pyretic
$ pyretic.py pyretic.kinetic.ext.gardenwall &
```

Start a ping from h1 to h2
```
mininet> h1 ping h2
```

Send Event to block traffic "h1 ping h2" (in "~/pyretic/pyretic/kinetic" directory)
```
$ python json_sender.py -n infected -l True --flow="{srcip=10.0.0.1}"
-a 127.0.0.1 -p 50001
```

Make h1's flow not be affected by IDS infection event, h1's traffic should be forwarded to 10.0.0.3 after issuing this command:
```
$ python json_sender.py -n exempt -l True --flow="{srcip=10.0.0.1}"
-a 127.0.0.1 -p 50001
```

Events to now allow traffic again:
```
$ python json_sender.py -n infected -l False
--flow="{srcip=10.0.0.1}" -a 127.0.0.1 -p 50001
```

Part 2: Server Load Balancing / Backup

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 will implement a simple server load balancing application using Kinetic, where the load

balancer is implemented as a primary/backup. The load balancer that we are asking you to implement is a load balancer that responds to events that explicitly redirect a source's traffic to a particular server, as opposed to the load balancer that we included in the Kinetic example applications that is based on random assignment. In this sense, the controller application you are writing could be used to process different types of events that might occur in the network (e.g., server failure, overload, etc.)

The network topology that you are using has three hosts (`h1`, `h2` and `h3`) directly connected to the switch `s1`. For this part of the assignment, let's assume that `h1` is a client, and `h2` and `h3` are servers for which you will do load balancing.

In the setup script that we have provided, host `h1` sends a ping request to a "public" IP address (10.0.0.100), which will direct the traffic to either host `h2` (10.0.0.2) or `h3` (10.0.0.3) based on the policy configured on the Kinetic controller.
Specifically, when you issue the following command:

```
$ python json_sender.py -n backup -l True --flow="{srcip=10.0.0.1}"
-a 127.0.0.1 -p 50001
```

Your traffic from `h1` should be redirected to host `h3` (the backup). When the backup variable is false, the traffic should be directed to `h2` (the "primary").

Your example application should have the following modifications:

- Create a finite state machine where host `h1`'s flows are directed either to primary or backup, using a variable called "`backup`".
- Create a new policy using Pyretic, `fwdtobackup`, which modifies traffic in the example topology to redirect all traffic from host `h1` to the backup server, `h3`.
- Write and check your CTL logic to verify that your host machine satisfies the following policy: "When h1 receives a `backup` event, the controller transitions to run the `fwdtobackup` policy."

The following function will be helpful:

- `rewriteDstIPAndMAC_Public()`. This is to change the dstip address to a specific IP address if the dstip matches a public IP (e.g., 10.0.0.100). This is used in the "primary-backup" application where the src host knows that it has to send traffic to a specific public IP.

Testing Your Code

Run your Kinetic controller application:

```
$ cd ~/pyretic
$ pyretic.py pyretic.kinetic.ext.primary_backup &
```

Now, open another terminal and run mininet to create your topology:

```
$ sudo mn --controller=remote,ip=127.0.0.1 --mac --arp --switch ovsk
--link=tc --topo=single,3
```

Start a ping from h1 to the public IP address:

```
mininet> h1 ping 10.0.0.100
```

Now, make h1's flow use backup server:

```
$ python json_sender.py -n backup -l True --flow="{srcip=10.0.0.1}"
-a 127.0.0.1 -p 50001
```

You can make h1's flow use the primary server again as follows:

```
$ python json_sender.py -n backup -l False --flow="{srcip=10.0.0.1}"
-a 127.0.0.1 -p 50001
```

Submitting Your Code

Part 1:

Run Kinetic controller application:

```
$ pyretic.py pyretic.kinetic.ext.gardenwall &
```

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/kinetic` directory:

```
$ cd ~/pyretic/pyretic/kinetic
$ 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 m7a-output.py
```

This will create an `output-1.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 results the results on a 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
```

Appendix: Steps to Make an Kinetic Application

1. **Define class for the application, subclassed from DynamicPolicy**
2. **Define LPEC**
    a. Define the packet space fields that will used to categorize packets to have equivalent policies applied. Unspecified fields will be wildcarded.
3. **Set up transition functions**
    a. Use “@transition” decorator
        i. Ensures it is fed into the NuSMV input automatic translator
    b. For events
    c. For policies
4. **Set up the FSM description**
    a. Define variable’s type, initial value, and transition function
    b. Both event and policy are variables
        i. **type**: Type of this variable (e.g., boolean, integer, Pyretic policy)
        ii. **init**: Initial value for this variable

iii. **trans**: transition function for this variable. Whenever an event arrives, the specified transition function is called.

5. **Set up policy and event streams**
    a. Create FSMPolicy.
    b. Register for events (e.g,. JSON events), so that it's called every time an event arrives.
    - **Note:** Normally, this part pretty much remains the same for any application, so no need to add/modify unless your application listens to non-JSON events.
6. **Define "main" method**
    a. Add methods to get ready for NuSMV model checker, if you want verification.
        - smv_str = fsm_def_to_smv_model(pol.fsm_def)
        - mc = ModelChecker(smv_str,'ids')
7. **Add SPEC lines, which is in CTL (Computation Tree Logic) language.**
    a. Add statements that should be verified.
    b. Some IDS examples:
        i. **If infected event is true, next policy state is 'drop':**
            - mc.add_spec("SPEC AG (infected -> AX policy=drop)")
        ii. **Policy state is 'allow' until infected is true.**
            - mc.add_spec("SPEC A [policy=allow U infected]")
8. **Return DynamicPolicy instance.**