Task One: Mac Tracker

Introduction:
MAC Tracker is small app which will watch for new addresses that have not been seen before and log the MAC and switch they were seen on.

Problem Analysis and Solution Design:
To able to track the MAC ADDRESS of all the hosts in the network.

Testing:
Testing is done using the Mininet software which creates a **realistic virtual network**.

The command used to simulate the network:
sudo mn --controller=remote --mac
This creates a simple network with one switch connected to our controller and 2 hosts connected to the switch.

```
12:47:28.653 WARN [n.f.c.i.C.s.notification:main] Switch 00:00:00:00:00:00:00:01 connected.
12:47:35.798 INFO [n.f.m.MACTracker:New I/O server worker #2-1] Host with MAC Address: 00:00:00:00:00:00:00:01 seen on switch: 1
12:47:35.812 INFO [n.f.m.MACTracker:New I/O server worker #2-1] Host with MAC Address: 00:00:00:00:00:00:00:02 seen on switch: 1
```

We can see that the MacTracker is able to determine the MAC address of host connected on a switch.

Another topology used to test the network is linear created using command:
sudo mn topo=linear,3 --controller=remote --mac

```
12:54:01.888 INFO [n.f.m.MACTracker:New I/O server worker #2-2] Host with MAC Address: 00:00:00:00:00:00:00:01 seen on switch: 1
12:54:02.040 INFO [n.f.m.MACTracker:New I/O server worker #2-3] Host with MAC Address: 00:00:00:00:00:00:00:02 seen on switch: 2
12:54:02.169 INFO [n.f.m.MACTracker:New I/O server worker #2-4] Host with MAC Address: 00:00:00:00:00:00:00:03 seen on switch: 3
```

The tracker is able to find the host associated with a switch as seen on the logs produced by the floodlight.

Observations:

- Initially all the switches have no rules installed on them. So, whenever any packet arrived on the switch, it forwards those packets to the controller.
- Controller at that point can gather all the information about the host

Task 2: SimpleSwitch

Introduction:
Simple Switch is an app which works at the IP level ie layer-3 in the TCP/IP protocol and allows all the hosts in the network to ping each other.

Algorithm/Implementation:
MySimpleSwitch class implements IOFMessageListener and IFloodlightModule
Receive function is the function which handles all the Packet In messages and install flow mod rules in the switches.

Data structures used :

1. ipToSwitchId : This Map is used to store ip to switchId mapping
2. switchToHostInfo: Map is used to store the information of all the hosts connected to it

Our Receiver listens for packet of Type : OFType.PACKET_IN

The Receiver Algorithm is:

```
Receive (IOFSwitch sw , PacketIn msg)
        sourceIp, destIp -- get sourceIp and destnIp from msg

        if source IP has never been seen before then
                store the switchID corresponding to sourceIp in ipToSwitchId

        if destination IP is not seen before then
                Flood the packet
                return
        else
                if packet data layer type is not ARP or IPV4 then
                        return ; //do nothing as we are supposed to handle only these types
                else
```

- install rules on the switches using FLOW MOD packets with wildcards matching on Data layer type, network source, network destination and in port
- set properties on rule like Timeout, length and output port
- send flow mod

Testing:

Testing is done by creating various topologies using mininet and sending ICMP messages to each other to test the ping of all the hosts.

Topology used:

1.Single , 2 (Single switch with 2 hosts)

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
```

```
mininet> sh dpctl dump-flows tcp:127.0.0.1:6634
stats_reply (xid=0xdd55c330): flags=none type=1(flow)
  cookie=0, duration_sec=41s, duration_nsec=148000000s, table_id=0, priority=0, n_packets=1, n_bytes=98,
 idle_timeout=60,hard_timeout=0,icmp,in_port=1,nw_src=10.0.0.1,nw_dst=10.0.0.2,actions=output:2
  cookie=0, duration_sec=41s, duration_nsec=148000000s, table_id=0, priority=0, n_packets=1, n_bytes=98,
 idle_timeout=60,hard_timeout=0,icmp,in_port=2,nw_src=10.0.0.2,nw_dst=10.0.0.1,actions=output:1
  cookie=0, duration_sec=41s, duration_nsec=153000000s, table_id=0, priority=0, n_packets=1, n_bytes=42,
 idle_timeout=60,hard_timeout=0,arp,in_port=2,nw_src=10.0.0.2,nw_dst=10.0.0.1,actions=output:1
  cookie=0, duration_sec=41s, duration_nsec=153000000s, table_id=0, priority=0, n_packets=2, n_bytes=84,
 idle_timeout=60,hard_timeout=0,arp,in_port=1,nw_src=10.0.0.1,nw_dst=10.0.0.2,actions=output:2
```

We can see that the module installs rules associated with ARP and ICMP packets. And hence once a ping is done, our learning algorithm determines the path and installs the rules.

2.Linear, 2(Two switches with single host each)

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2
h2 -> h1
*** Results: 0% dropped (2/2 received)
```

```
mininet> sh dpctl dump-flows tcp:127.0.0.1:6634
stats_reply (xid=0x1c1fb255): flags=none type=1(flow)
  cookie=0, duration_sec=2s, duration_nsec=362000000s, table_id=0, priority=0, n_packets=1, n_bytes=42,
idle_timeout=60,hard_timeout=0,arp,in_port=2,nw_src=10.0.0.2,nw_dst=10.0.0.1,actions=output:1
  cookie=0, duration_sec=2s, duration_nsec=362000000s, table_id=0, priority=0, n_packets=0, n_bytes=0, i
dle_timeout=60,hard_timeout=0,arp,in_port=1,nw_src=10.0.0.1,nw_dst=10.0.0.2,actions=output:2
  cookie=0, duration_sec=2s, duration_nsec=342000000s, table_id=0, priority=0, n_packets=1, n_bytes=98,
idle_timeout=60,hard_timeout=0,icmp,in_port=1,nw_src=10.0.0.1,nw_dst=10.0.0.2,actions=output:2
  cookie=0, duration_sec=2s, duration_nsec=342000000s, table_id=0, priority=0, n_packets=2, n_bytes=196,
 idle_timeout=60,hard_timeout=0,icmp,in_port=2,nw_src=10.0.0.2,nw_dst=10.0.0.1,actions=output:1
```

```
mininet> sh dpctl dump-flows tcp:127.0.0.1:6635
stats_reply (xid=0x7ea09c3b): flags=none type=1(flow)
  cookie=0, duration_sec=5s, duration_nsec=623000000s, table_id=0, priority=0, n_packets=1, n_bytes=42,
idle_timeout=60,hard_timeout=0,arp,in_port=2,nw_src=10.0.0.1,nw_dst=10.0.0.2,actions=output:1
  cookie=0, duration_sec=5s, duration_nsec=623000000s, table_id=0, priority=0, n_packets=1, n_bytes=42,
idle_timeout=60,hard_timeout=0,arp,in_port=1,nw_src=10.0.0.2,nw_dst=10.0.0.1,actions=output:2
  cookie=0, duration_sec=5s, duration_nsec=606000000s, table_id=0, priority=0, n_packets=2, n_bytes=196,
 idle_timeout=60,hard_timeout=0,icmp,in_port=1,nw_src=10.0.0.2,nw_dst=10.0.0.1,actions=output:2
  cookie=0, duration_sec=5s, duration_nsec=606000000s, table_id=0, priority=0, n_packets=1, n_bytes=98,
idle_timeout=60,hard_timeout=0,icmp,in_port=2,nw_src=10.0.0.1,nw_dst=10.0.0.2,actions=output:1
```

The rules are installed on both the switches for forwarding.

Observation:

- Controller sometime takes a lot of time to get initialized, especially for the complex topologies like tree. In that case, sufficient time must be given for the controller to set up before start sending the ping messages.
- Flooding happens only for the first time, after then all the rules are installed and no flooding happens.

Task 3: Simple Firewall

Introduction:

A simple topology contains 3 hosts (H1, H2 and H3) and 1 switch (shown below). You need to modify the application in Task Two to achieve the following objectives: H1 and H2 can ping each other. H1 and H3 can ping each other. But H2 and H3 cannot ping each other.

Problem Analysis and Solution Design:
Firewall works on the same principle as Simple Switch discussed above but with a slight modification in the logic of the installation of the rules on the switches.

Since, we are supposed block host 2 and host 3 in the any kind of topology.

Algorithm:
The only change in the code of the Simple Switch is in two places:
1.Selection of Host 2 and Host 3:
To specifically block Host 2 and Host 3, the IP values are hardcoded into the system.

2.Rule installation:
If the source ip is of host 2 and destination ip of host 3 or vice versa, then the output port is left empty or 0 for the rule for which the condition is met. Hence, the packet is dropped and host 2 and host 3 are not able to ping each other.
Rest of the implementation is same as the Simple Switch.

Testing:
Testing is done by creating various topologies using mininet and sending ICMP messages to each other to test the ping of all the hosts.

Topology used:

1.Single , 3 (Single switch with 3 hosts)

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 X
h3 -> h1 X
*** Results: 33% dropped (4/6 received)
```

```
mininet> sh dpctl dump-flows tcp:127.0.0.1:6634
stats_reply (xid=0x47cb88fe): flags=none type=1(flow)
  cookie=0, duration_sec=24s, duration_nsec=755000000s, table_id=0, priority=0, n_packets=0, n_bytes=0, idle_timeout=50
0,hard_timeout=0,ip,in_port=3,nw_src=10.0.0.2,nw_dst=10.0.0.3,actions=output:0
  cookie=0, duration_sec=27s, duration_nsec=917000000s, table_id=0, priority=0, n_packets=1, n_bytes=98, idle_timeout=5
00,hard_timeout=0,ip,in_port=1,nw_src=10.0.0.1,nw_dst=10.0.0.3,actions=output:2
  cookie=0, duration_sec=27s, duration_nsec=965000000s, table_id=0, priority=0, n_packets=1, n_bytes=42, idle_timeout=5
00,hard_timeout=0,arp,in_port=3,nw_src=10.0.0.2,nw_dst=10.0.0.1,actions=output:1
  cookie=0, duration_sec=27s, duration_nsec=923000000s, table_id=0, priority=0, n_packets=1, n_bytes=42, idle_timeout=5
00,hard_timeout=0,arp,in_port=2,nw_src=10.0.0.3,nw_dst=10.0.0.1,actions=output:1
  cookie=0, duration_sec=27s, duration_nsec=965000000s, table_id=0, priority=0, n_packets=1, n_bytes=42, idle_timeout=5
00,hard_timeout=0,arp,in_port=1,nw_src=10.0.0.1,nw_dst=10.0.0.2,actions=output:3
  cookie=0, duration_sec=27s, duration_nsec=917000000s, table_id=0, priority=0, n_packets=2, n_bytes=196, idle_timeout=
500,hard_timeout=0,ip,in_port=2,nw_src=10.0.0.3,nw_dst=10.0.0.1,actions=output:1
  cookie=0, duration_sec=27s, duration_nsec=763000000s, table_id=0, priority=0, n_packets=4, n_bytes=168, idle_timeout=
500,hard_timeout=0,arp,in_port=2,nw_src=10.0.0.3,nw_dst=10.0.0.2,actions=output:0
  cookie=0, duration_sec=27s, duration_nsec=763000000s, table_id=0, priority=0, n_packets=5, n_bytes=210, idle_timeout=
500,hard_timeout=0,arp,in_port=3,nw_src=10.0.0.2,nw_dst=10.0.0.3,actions=output:0
  cookie=0, duration_sec=24s, duration_nsec=755000000s, table_id=0, priority=0, n_packets=0, n_bytes=0, idle_timeout=50
0,hard_timeout=0,ip,in_port=2,nw_src=10.0.0.3,nw_dst=10.0.0.2,actions=output:0
  cookie=0, duration_sec=27s, duration_nsec=956000000s, table_id=0, priority=0, n_packets=1, n_bytes=98, idle_timeout=5
00,hard_timeout=0,ip,in_port=1,nw_src=10.0.0.1,nw_dst=10.0.0.2,actions=output:3
  cookie=0, duration_sec=27s, duration_nsec=956000000s, table_id=0, priority=0, n_packets=2, n_bytes=196, idle_timeout=
500,hard_timeout=0,ip,in_port=3,nw_src=10.0.0.2,nw_dst=10.0.0.1,actions=output:1
  cookie=0, duration_sec=27s, duration_nsec=922000000s, table_id=0, priority=0, n_packets=1, n_bytes=42, idle_timeout=5
00,hard_timeout=0,arp,in_port=1,nw_src=10.0.0.1,nw_dst=10.0.0.3,actions=output:2
```

2.Linear,3

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 X
h3 -> h1 X
*** Results: 33% dropped (4/6 received)
```

```
mininet> sh dpctl dump-flows tcp:127.0.0.1:6634
stats_reply (xid=0x7c957584): flags=none type=1(flow)
  cookie=0, duration_sec=17s, duration_nsec=763000000s, table_id=0, priority=0, n_packets=1, n_bytes=42, idle_timeout=5
00,hard_timeout=0,arp,in_port=2,nw_src=10.0.0.2,nw_dst=10.0.0.1,actions=output:1
  cookie=0, duration_sec=17s, duration_nsec=756000000s, table_id=0, priority=0, n_packets=2, n_bytes=196, idle_timeout=
500,hard_timeout=0,ip,in_port=2,nw_src=10.0.0.2,nw_dst=10.0.0.1,actions=output:1
  cookie=0, duration_sec=17s, duration_nsec=701000000s, table_id=0, priority=0, n_packets=1, n_bytes=98, idle_timeout=5
00,hard_timeout=0,ip,in_port=1,nw_src=10.0.0.1,nw_dst=10.0.0.3,actions=output:2
  cookie=0, duration_sec=17s, duration_nsec=705000000s, table_id=0, priority=0, n_packets=1, n_bytes=42, idle_timeout=5
00,hard_timeout=0,arp,in_port=2,nw_src=10.0.0.3,nw_dst=10.0.0.1,actions=output:1
  cookie=0, duration_sec=17s, duration_nsec=763000000s, table_id=0, priority=0, n_packets=1, n_bytes=42, idle_timeout=5
00,hard_timeout=0,arp,in_port=1,nw_src=10.0.0.1,nw_dst=10.0.0.2,actions=output:2
  cookie=0, duration_sec=17s, duration_nsec=701000000s, table_id=0, priority=0, n_packets=2, n_bytes=196, idle_timeout=
500,hard_timeout=0,ip,in_port=2,nw_src=10.0.0.3,nw_dst=10.0.0.1,actions=output:1
  cookie=0, duration_sec=17s, duration_nsec=656000000s, table_id=0, priority=0, n_packets=0, n_bytes=0, idle_timeout=50
0,hard_timeout=0,arp,in_port=2,nw_src=10.0.0.3,nw_dst=10.0.0.2,actions=output:0
  cookie=0, duration_sec=14s, duration_nsec=628000000s, table_id=0, priority=0, n_packets=0, n_bytes=0, idle_timeout=50
0,hard_timeout=0,ip,in_port=2,nw_src=10.0.0.2,nw_dst=10.0.0.3,actions=output:0
  cookie=0, duration_sec=14s, duration_nsec=628000000s, table_id=0, priority=0, n_packets=0, n_bytes=0, idle_timeout=50
0,hard_timeout=0,ip,in_port=2,nw_src=10.0.0.3,nw_dst=10.0.0.2,actions=output:0
  cookie=0, duration_sec=17s, duration_nsec=756000000s, table_id=0, priority=0, n_packets=1, n_bytes=98, idle_timeout=5
00,hard_timeout=0,ip,in_port=1,nw_src=10.0.0.1,nw_dst=10.0.0.2,actions=output:2
  cookie=0, duration_sec=17s, duration_nsec=656000000s, table_id=0, priority=0, n_packets=0, n_bytes=0, idle_timeout=50
0,hard_timeout=0,arp,in_port=2,nw_src=10.0.0.2,nw_dst=10.0.0.3,actions=output:0
  cookie=0, duration_sec=17s, duration_nsec=705000000s, table_id=0, priority=0, n_packets=1, n_bytes=42, idle_timeout=5
00,hard_timeout=0,arp,in_port=1,nw_src=10.0.0.1,nw_dst=10.0.0.3,actions=output:2
```

Summary/Conclusion:

- Not specifying output during rule installation can lead to packet drop and hence that has been utilized here.

Task 4: MyNewApp

Introduction:

SDN apps has opened a whole new paradigm in the field of management and security of networks. Centralized controller has allowed to collect, store and visualize network data for the real time monitoring and determining various anomalies associated with the data.

Network traffic anomalies exhibit abnormal changes in network traffic. Examples are UDP Flood, ICMP Flood, TCP SYN attack, PortScan etc. This app focuses on the ICMP flood attack only for now but can easily be extended to detect anomaly in UDP, TCP or any other packet type of protocol.

This app monitors the network traffic flow to detect abnormal behaviour using mathematical model from some malicious host and ultimately block the host. One such mathematical model is Triple Exponential Smoothing , also known as Holt-Winters algorithm, which can be applied to a time series data to make forecasts on some flow metric. There can be a large number of flow metrics associated with network flow like flow size, number of packets etc. This app considers only number of packets as metrics for detecting anomaly.

App tries to predict the ICMP packet anomalies well in advance, exposing REST api to the network administrator about the anomalies and the possible malicious hosts. It's upto network administrator to block the host or ignore, as soon as the violation occurs or wait for the system to block the host when some threshold number of violations are done by the user.

Problem Analysis and Solution Design:

ICMP Flood:

ICMP flood is producing a large number of Echo Requests, making flood victim too busy replying to the requests and ending up consuming bandwidth and slowing down the victim.

We can model this attack using either Total Bytes sent or Total Packets sent or both. For now, this app models only the Total Packets sent over the network from one host to another.

Overview of the Triple Exponential Smoothing:

Exponential Smoothing can be used to predict some future values related to some metric. Holt-Winters Forecasting Algorithm uses exponential smoothing technique and considers 3 factors to predict the values: Baseline, Trend factor and Seasonal Index

The basic equations are:

$S_t = \alpha y_t / I_{t-L} + (1-\alpha)(S_{t-1} + b_{t-1})$

$b_t = \gamma (S_t - S_{t-1}) + (1-\gamma) b_{t-1}$

$I_t = \beta y_t / S_t + (1-\beta) I_{t-L}$

$F_{t+m} = (S_t + m b_t) I_{t-L+m}$

where

- y_t is the traffic count in the time t
- S_t is the prediction made for the time t
- b is the trend factor
- I is the seasonal index
- F is the forecast and m periods ahead
- t is the time frame used for collecting and predicting traffic
- $\alpha\beta\gamma$ are model parameters

Confidence Bands:

Concept of confidence bands is used to measure the deviation of actual traffic from the predicted value and consider it as anomaly.

The deviation is defined using the equation:

$d_t = | y_t - \delta S_t |$

where

- d_t is the deviation
- y_t is the actual traffic count
- S_t is the predicted traffic value
- t is the time frame for which the data is used
- δ is known as the scaling factor, also define the width of the confidence band

Anomaly Detection:

The app only consider positive deviation while using the above equation. Thus, if the actual traffic crosses the upper bound of the confidence band only then it is considered an anomaly and recorded in the memory. The upper bound is represented as THRESHOLD variable in the code.

Choosing Model Parameters:

$\alpha\beta\gamma$

When α is close to 1. dampening is quick and the equation responds more to the changes in the current time. When it is close to 0, this dampening is slow.

We choose α such that MSE (Mean Squared Error) of the actual and predicted value is less.

δ

The reasonable values lies between [2,3] using the statistical distribution theory. For this app, value is 2 to detect anomalies very aggressively for demonstration purposes .

Data Structure and Important Variables:
Several Mapping structures are used to capture a lot of information regarding the traffic in the network.

Important Constants:

1. TimeWindow : The time for monitoring the traffic can be used either in seconds or miniutes or even days. For this app and given a week time, the app is tested for detection anomaly in seconds. So, the time window chosen is 60 seconds
2. Period: The Time window is divided in to 6 periods and hence giving us a time frame of 10 seconds, for which the traffic collected.
3. Scaling Factor: Used to define the width of the confidence bands
4. Threshold: The number of times a host can violate the network traffic trend before being blocked to send any more traffic on the net.

Mapping Structures:
1.For each host, traffic information is maintained in structure known as “HostTrafficVector” which contains information, as

- currSw: Current Switch to which it is connected
- currFrame: Current time frame among the 10 possible frames
- icmpPacketFreq: stores the actual traffic count of a host in a particular time frame, used later to predict the trafic in the next time frame
- violated : A boolean variable to determine if the host in the current time frame has already crossed the confidence band’s upper limit. Used to avoid multiple violation detection in the same time frame.

2. A mapping is maintained of the total number of violation known as “HostViolationVector”, for a host in the network which contains information like

- Host Mac String: In human readable form, used by the network admin
- List of violation data, where each violation data contains
    - time : when the violation occurred
    - predicted: Predicted traffic value for that time frame
    - actual: Actual traffic recorded for the user

3.hostStatusMap is maintained to store the connection information for each host where each HostConnectionData structure stores

- swID: switch Id on which the host is connected
- port : the port on the switch on which the host is connected
- status: status of the host whether it is CONNECTED or BLOCKED

4. hostListToBlock: list which stores the macID of the hosts which have been selected to be blocked

Two data structures are same as used on the L3 switch for the packet forwarding -

- ipToSwitch
- switchToHost

Algorithm:
Our Module listens to two types of Open flow message Types

1. PacketIn
2. Flow Removed

So we have three paths in our algorithm:
Path1: PacketIn Message Received
When this packet is received then it works as L3 Switch and install rules like before.

We want to maintain the total packet sent count by each host in the network. This can be achieved by 2 ways :

1. Making each packet sent by the host in the network to pass through the controller and increase the count
2. Install the rules such that they expire with same time length as the time frame used in the app and get the packet count that passed through this rule for each host.

The app uses the later method as this increases the performance of the app and the network significantly. For this we made two changes in the rules properties :

1. Make sure that the OFPFF_SEND_FLOW_REM flag is set for the Flow mod installed on the switch.
2. value of FLOWMOD_DEFAULT_HARD_TIMEOUT is set equal to the time frame

The reason for this is because we want the total packet sent by each host in the network. This can be achieved by either making each packet pass throug

Path 2: Flow Removed Message is recived
Receiver Algorithm:

parse the FlowRemoved msg and get source IP
if source IP is not 0 and packet type is IPV4 then
Update the traffic vector and calculate the prediction value

- get the traffic vector for this host and for current time frame increase the icmp count for this
- if time frame switched to a new time frame ie we have moved on to the counting of traffic for new time frame then get the new predicted value for this time frame using Holt Winters Algorithm

Check for violation

- get the diff as (current traffic count - SCALING_FACTOR * prediction), for the current time frame

if the diff > 0 and there has been no violation in the current time frame

- make an entry in the host's violation data with current time when the violation happened, total count and the expected value

Check And Add Mac in the blocking list:

- If the number of violations for the particular host has crossed some THRESHOLD value then
  - add the source mac to the "hostListToBlock" list

call Block Host for Switch in all the cases:

check that there are entries in the list to block and host is on the same switch for which the FlowRemoved message is received then

- send Flow mod with source mac as the hosts mac and output port as NONE, as a result all the packets will drop which originate from this host

Path 3:

We have exposed a Rest Api to the administrator to block the host using the call like http://localhost:8080/wm/mynewapp/devices/{macId}/block

This call comes from another Thread in the java code and hence all the maps and list used in the system have been taken to be thread safe and synchronized.

Function called is DeviceBlockResponse

Algorithm:

get the macId from the http call and check that whether the mac provided is valid and present in the network or not

if not a valid mac

send appropriate response to the admin

else

add the mac to the listToBlock

Now our system is reactive blocking system not proactive, hence, whenever any flow removed packet arrives on the same switch where this host is located, then the rule will be added to block this host.

Important Note: Once the host is blocked the violations data for the host is removed, so that if the host is unblocked it does not get blocked again. And starts with no violation at all.

Path 4:

The app also exposes another api to unblock the host in the system.The Api is http://localhost:8080/wm/mynewapp/devices/{swId}/unblock

Function used is getHostToUnBlock in DeviceUnblockResource clas:

Algorithm:

get the macId from the http call and check that whether the mac provided is valid and present in the network or not

if not a valid mac

send appropriate response to the admin
else
remove mac from the listToBlock

Since the rule on the switch to block the packets from the host will TimeOut and then it will not reinstall the rule, as the host will no be present on the listToBlock

REST Interface

The MyNewApp Module exposes REST interface implemented as RestletRoutable using Rest API Service. Following is a list of REST methods exposed:

URI /wm/mynewapp	**Method**	**Description**
/devices/violations	Get	Get a detailed layout of the violations done by any host in JSON format
/devices/list	Get	Gived the status whether CONNECTED or BLOCKED list of all the hosts in the system
/devices/{macId}/block	Get	eventually blocks the device with the given macID (in long)
/devices/{macId}/unblock	Get	eventually unblock the host with given macID

Testing:

IMPORTANT NOTE: The app initializes the traffic count for each host initially with value 3, so that the initial prediction does not come out to be 0 and hence ending up giving violation warning even for a single ping.

Please create package with name: net.floodlightcontroller.mynewapp
and include all the files in that.

1.Single , 3 (Single switch with 3 hosts)

Steps taken:
1.Command Fired: pingall

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3
h2 -> h1 h3
h3 -> h1 h2
*** Results: 0% dropped (6/6 received)
```

Rest Api output: http://localhost:8080/wm/mynewapp/devices/list


```
{

   "2":{
      "port":3,
      "status":"CONNECTED",
      "swId":1
   },
   "1":{
      "port":2,
      "status":"CONNECTED",
      "swId":1
   },
   "3":{
      "port":1,
      "status":"CONNECTED",
      "swId":1
   }

}
```


and also the violations:
http://localhost:8080/wm/mynewapp/devices/violations


```
{

   "2":{
      "macAddress":"00:00:00:00:00:00:00:02",
      "violations":[  ]
   },
   "1":{
      "macAddress":"00:00:00:00:00:00:00:01",
      "violations":[   ]
   },
   "3":{
      "macAddress":"00:00:00:00:00:00:00:03",
      "violations":[    ]
   }
}
```


Now let say Host 1 starts pinging a lot of ICMP packets.
using command: h1 ping h2

and setting very low Threshold and initialization values for the traffic vector for any host, we can easily cross the Confidence Values

The logs produced by the floodlight logger module will be like:
19:41:48.219 INFO [n.f.m.MACTracker:New I/O server worker #2-2] Values used for prediction for source Mac 00:00:00:00:00:00:00:01 with index 8
19:41:48.220 INFO [n.f.m.MACTracker:New I/O server worker #2-2] Prediction value is:1
19:41:48.220 INFO [n.f.m.MACTracker:New I/O server worker #2-2] WARNING:::::: Expected 1 Actual:7

Using Api: http://localhost:8080/wm/mynewapp/devices/violations


```
{

   "2":{
      "macAddress":"00:00:00:00:00:00:00:02",
      "violations":[
         {
            "time":"2014/10/02 19:41:32",
            "actual":4,
            "predicted":0
         },
         {
            "time":"2014/10/02 19:41:48",
            "actual":8,
            "predicted":1
         }
      ]
   },
   "1":{
      "macAddress":"00:00:00:00:00:00:00:01",
      "violations":[
         {
            "time":"2014/10/02 19:41:32",
            "actual":5,
            "predicted":0
         },
         {
            "time":"2014/10/02 19:41:48",
            "actual":7,
            "predicted":1
         }
      ]
```

```
    },
    "3":{
        "macAddress":"00:00:00:00:00:00:00:03",
        "violations":[  ]
    }

}
```


Eventually, the threshold value for the total violation a user can do will exceed and the logger will print logs like this
19:53:03.299 INFO [] WARNING:::::: Expected0 Actual:7
19:53:03.299 INFO [] !!!!!!!!!!!!!!!Mac:2 has reached the threshold limit. Blocking!!!!
19:53:03.299 INFO [] Sending blocking Rule

Looking at the status using API: http://localhost:8080/wm/mynewapp/devices/list

{"2":{"swId":1,"port":3,"status":"**BLOCKED**"},"1":{"swId":1,"port":1,"status":"**BLOCKED**"},"3":{"swId":1,"port":2,"status":"CONNECTED"}}

Both 2 and 1 got blocked because both of them were pinging to each other with very high rate, and hence both of them ended up violating and crossing the threshold and hence the app blocked both of them.

Output of pingall :

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> X X
h2 -> X X
h3 -> X X
*** Results: 100% dropped (0/6 received)
```

Let say the admin decided to unblock the host1, because it was a false alarm or it was expected, he can call the API:
http://localhost:8080/wm/mynewapp/devices/1/unblock

We can see the output of command:h3 ping h2

```
mininet> h3 ping h1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
From 10.0.0.3 icmp_seq=10 Destination Host Unreachable
From 10.0.0.3 icmp_seq=11 Destination Host Unreachable
From 10.0.0.3 icmp_seq=12 Destination Host Unreachable
From 10.0.0.3 icmp_seq=13 Destination Host Unreachable
From 10.0.0.3 icmp_seq=14 Destination Host Unreachable
From 10.0.0.3 icmp_seq=15 Destination Host Unreachable
From 10.0.0.3 icmp_seq=16 Destination Host Unreachable
From 10.0.0.3 icmp_seq=17 Destination Host Unreachable
From 10.0.0.3 icmp_seq=18 Destination Host Unreachable
64 bytes from 10.0.0.3: icmp_seq=19 ttl=64 time=4.76 ms
64 bytes from 10.0.0.3: icmp_seq=20 ttl=64 time=0.161 ms
64 bytes from 10.0.0.3: icmp_seq=21 ttl=64 time=0.155 ms
64 bytes from 10.0.0.3: icmp_seq=22 ttl=64 time=0.162 ms
64 bytes from 10.0.0.3: icmp_seq=23 ttl=64 time=0.094 ms
64 bytes from 10.0.0.3: icmp_seq=24 ttl=64 time=0.217 ms
64 bytes from 10.0.0.3: icmp_seq=25 ttl=64 time=0.105 ms
^C
--- 10.0.0.1 ping statistics ---
25 packets transmitted, 7 received, +9 errors, 72% packet loss, time 24007ms
rtt min/avg/max/mdev = 0.094/0.807/4.760/1.614 ms, pipe 3
```

Eventually the rule on host 1 to block it, will get removed when the Timeout will occur and it has again become reachable.

There can also be cases when the Admin decided to block some user which is producing a lot of traffic in the system or have produced enough violations for the Admin to suspect that he making the network slow. In that case, admin can use the api like:
http://localhost:8080/wm/mynewapp/devices/2/block

{"2":{"swId":1,"port":3,"status":"**BLOCKED**"},"1":{"swId":1,"port":1,"status":"CONNECTED"},"3":{"swId":1,"port":2,"status":"CONNECTED"}}

```
From 10.0.0.2 icmp_seq=56 Destination Host Unreachable
From 10.0.0.2 icmp_seq=57 Destination Host Unreachable
From 10.0.0.2 icmp_seq=58 Destination Host Unreachable
From 10.0.0.2 icmp_seq=59 Destination Host Unreachable
From 10.0.0.2 icmp_seq=60 Destination Host Unreachable
From 10.0.0.2 icmp_seq=61 Destination Host Unreachable
64 bytes from 10.0.0.2: icmp_seq=62 ttl=64 time=1006 ms
64 bytes from 10.0.0.2: icmp_seq=63 ttl=64 time=7.14 ms
64 bytes from 10.0.0.2: icmp_seq=64 ttl=64 time=0.209 ms
64 bytes from 10.0.0.2: icmp_seq=65 ttl=64 time=0.152 ms
64 bytes from 10.0.0.2: icmp_seq=66 ttl=64 time=0.162 ms
```

Evaluation/Observation:

- App is able to block hosts based on the anomaly detection.
- Depending on the model parameters chosen, app can be made to adapt to more recent traffic or be more rigid to changes.

- Depending on the scaling factor value, app can be made more aggressive towards the anomaly detection.

Further work:

- Test the app in the real world and not on the virtual network with heavy traffic to test the app's effectiveness.
- Come up with better $\alpha\beta\gamma$ ie the model parameters to mimic the real world traffic scenario.
- Flexibility to detect the anomalies in all kind of network packets like TCP SYN, UDP etc, so that all kind of flood attacks can be blocked using the app.
- Correct the initialization of the traffic vector for each host. Either make a more calculated guess or collect some previous traffic data for the host before starting the whole prediction process.
-

Bibliography:

[1] **http://en.wikipedia.org/wiki/Exponential_smoothing**
[2] **http://arxiv.org/pdf/1007.1264.pdf**
[3] **https://www.usenix.org/legacy/events/lisa00/full_papers/brutlag/brutlag.pdf**
[4] **http://www.itl.nist.gov/div898/handbook/pmc/pmc.htm**
[5] **http://repo.hackerzvoice.net/depot_cehv6/CEHv6%20Module%2014%20Denial%20of%20Service/dos.pdf**