Ant Colony Optimization Algorithm for Finding Paths in POX Software Defined Networking Controller

Mahmoud Khatib[1], Souheil Khawatmi[2], Fadel Sukkar[3]

[1]Postgraduate Student (M.S), Systems and Computer Networks Department, University of Aleppo, Syria
[2]Associative Professor, Systems and Computer Networks Department, University of Aleppo, Syria
[3]Professor, Artificial Intelligence and Natural Language Department, University of Aleppo, Syria

---***---

Abstract *Software Defined Network (SDN) decouples networks control plane and data plane, make the controller gain the global network topology view which can be utilized by the controller's forwarding applications to forwards the packets between hosts with the helping of OpenFlow protocol. The POX controller and Mininet tool has been used to simulate the underlying SDN infrastructure. This paper analyze a different data forwarding components currently supported by the POX controller and present a protection approach by developing a new component (Ant_Pox), which implement Ant Colony Algorithm (ACO) to find a backup paths in case of failure occurs. Finally four components are compared, hub, l2_learning, l2_multi and Ant_Pox, by measures the Round Trip Time (RTT) and CPU usage.*

***Key Words*: Software Defined Network (SDN), OpenFlow Protocol, POX Controller, Mininet, Ant Colony Algorithm (ACO), Round Trip Time (RTT), and CPU usage.**

1.INTRODUCTION

The development of network technology has recently grown rapidly, where its development has made it easier for us to build, monitor or maintain a computer network. With the rapid development of network technology, it has created a new paradigm in network technology, namely software defined network (SDN). SDN is a term that refers to a new concept (paradigm) in designing, managing and implementing networks, especially to support the needs and innovations in this field, which are increasingly complex. In conventional networks, the data plane and the control plan are combined into one device, while in SDN networks, the data plane and control plane are separated [1]. With the separation between the control plane and the data plane on the SDN, network makes it easy to build, monitor or maintain a computer network with the provisions made. Many advanced development of SDN has been emerged nowadays [2][3]. The OpenFlow protocol, which allows the creation of applications for Software Defined Networks, has been a new standard to make a network programmable based on the protocol specification[1]. To do the network programming, an interface is needed. That interface is known as API (Application Programming Interface). POX Controller is one of the SDN controller which support the OpenFlow version 1.0 only. This is one of the first controller which developed to support SDN network.

The main goal of this paper is to develop a new protection component in POX controller The organization of this paper is constructed as follows: Section two present the basic concepts about SDN model. Section three discusses the OpenFlow architecture, messages Section four explain the matching process using OpenFlow. Section five introduce POX controller. Section six explain the data forwarding approaches. Section seven introduce the simulation tool that used. Section eight Implementation the forwarding Components. Section nine is reserved to the results. Section ten is evaluation of Pox forwarding approaches. Finally, conclusions is drawn in the section eleven

2. SOFTWARE DEFINED NETWORK (SDN)

The Open Networking Foundation (ONF) [3] defines the SDN as follows: " In the SDN architecture, the control and data planes are decoupled, network intelligence and state are logically centralized, and the underlying network infrastructure is abstracted from the applications."[4].

The SDN is an emerging network architecture that allows a centralized software program to control the behavior of an entire network, which consist three layers, Fig.1 illustrates the general SDN architecture, First layer (infrastructure layer) consists of both physical and virtual network devices. Second layer (control layer) involve of a centralized control plane, and considered the mid-layer that connects the application layer and infrastructure layer. It provides centralized global view to entire network. Third layer (application layer) contains of network services, application that used to interact with control layer [5]. The control layer bridges the application layer and the infrastructure layer, via its two interfaces. For the upward interacting with the application layer (i.e., the Nourthbound interface) or NBI, it provide an abstract of network functions (optimal network resources and paths) with a programmable interface for applications to consume the network services and configure the network dynamically. For the downward interacting with the infrastructure layer (the Southbound interface) or SBI, it allows a controller to define the behavior of the hardware in the network. The standard and most common Southbound API is OpenFlow.
Those interfaces are API is said to be used to define the software interaction among systems [6]. In SDN, these systems refer to network applications and hardware such as routers, switches and so on. The programming part of the API is what makes it necessary for SDN.

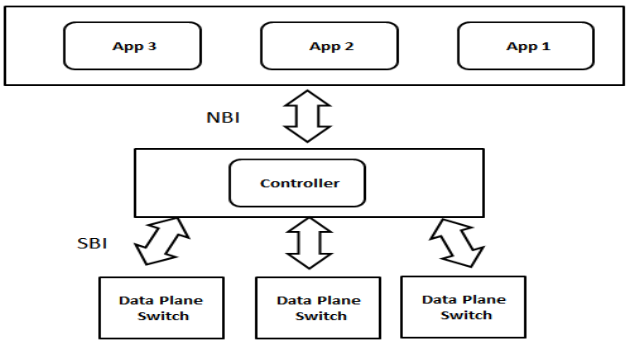


Fig 1: SDN paradigm

3. OPENFLOW PROTOCOL

For the southbound interface of SDN, the OpenFlow protocol is the most commonly used protocol which separates the data plane from the control plane, is the network abstraction layer which defines the standard protocol for communication in the network, in other words, SDN uses the OpenFlow protocol to allows the SDN controller to configure switches, i.e. via the installation of packet forwarding rules [7][8][9].The protocol also allows switches to notify the controller about special events, e.g. the receipt of a packet that does not match any installed rules. It allows both the controller and all the switches to understand each other [10].

4. OPENFLOW ARCHITIECTURE

An OpenFlow Switch consists of one or more flow tables and a group table, which perform packet lookups and forwarding, and an OpenFlow channel to an external controller as shown in Fig 2. The switch communicates with the controller and the controller manages the switch via the OpenFlow protocol. By using the OpenFlow protocol, the controller can add, update, and delete flow entries in flow tables, both reactively and proactively. Each flow table in the switch contains a set of flow entries, each flow entry consists of match fields, counters, and a set of instructions to apply for matching packets as shown in Fig.2 [11].

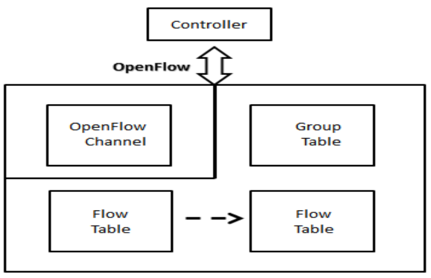


Fig 2: OpenFlow Architecture

5. POX CONTROLLER

POX is a software platform developed in Python, it is began early as a controller for an OpenFlow protocol[12][13]. However, it can nowadays, act as an OpenFlow switch, and can be used for developing networking software (i.e. Load Balancing, Firewall). POX controller worked as publish-subscribe model, There are some objects which generate events and there are some subscribers which subscribe event through event handler. The communication between switch to controller is coordinated through events. There are collections of events and each event will fired under certain condition. POX uses OpenFlow Protocol for sounthbound interface. OpenFlow has different events (Packet_In, Packet_OUT,Port-status, Flow_Remove, connectionUp, etc). POX work with Python 2.7 (it can also work fine with Python 2.6), and can run under Linux OS, Mac OS, and Windows. Pox comes with a number of components called stock components, some provide basic functionality, some provide convenient features, and some are just examples, some of which depend on each other to do the work

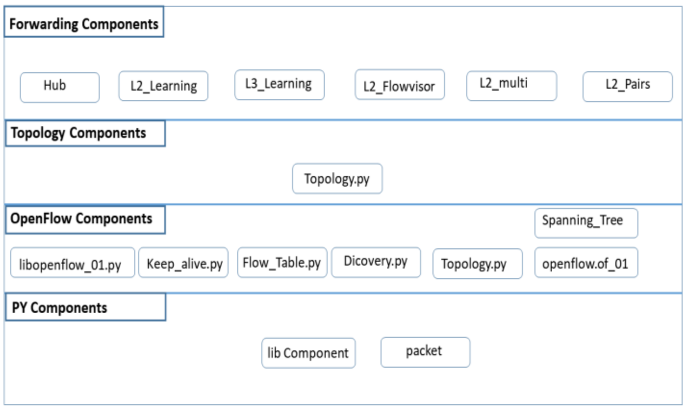


Fig 3: POX Architecture Component

Fig.3 shows the modular structure of the POX controller, which is divided into four main components [12]:

- Py components: this component cause pox to start the interactive python interpreter and can be useful for debugging and interactive experiments.
- OpenFlow components: is used to discover connections between OpenFlow switches by periodically sending LLDP packets for link discovery.
- Topology components: is responsible for creating a graph of the physical network and keeping it up to date.
- Forwarding components: each switch object listens for events related to the adjacent switch, this components care about one event called "packet_in", when the switch receives a new packets and does not find a match in the flow table, then a request is sent to the controller which contain the packet header. This event in turn indicates to the controller that there is a new flow in the network. The process of calculating the path occurs in one of the forwarding components. After finding the path, the controller assigns a flow by sending the appropriate rules via Ofp_flow messages to the flow table in all the switches on the path.

6. POX FORWARDING APPROACHES

This section presents an overview of the three main components of the forwarding functionality used in the current POX [12]. The forwarding components requires one specific event called "Packet In". Every time an edge switch registers a new packet and does not have a matching table entry for it, it sends a request to the controller, which contains the packet header and a buffer ID. This event indicates to the controller that there is a new flow in the network. The path calculation can be done using any data forwarding algorithm. It can be done in a reactive or a proactive way. After the path had been found. The controller assigns it to the flow, then installing table rules to match on every switch on the path by sending a Packet_Out (hub) or ofp_flow_mod (L2_learning, L2_multi) commands. Additionally, the forwarding component is responsible to track every new flow together with its route. It keeps information locally about every flow until a "*FlowRemoved*" event fires up. This happens when a switch removes a flow entry from its table, because it was deleted or expired (idle or hard time out).

In this Paper, four data forwarding components in POX controller are discussed, Hub, L2_learning, and L2_multi are built-in components in pox controller, and one has been developed after modified L2_multi component, called Ant_Pox. Hub and L2_learning components works like traditional network devices. Fig.4, illustrate the steps that L2_multi and Ant_Pox follow:

1- Host 1 generate a new request packet the to destination (Host 2).
2- First of all, the Discovery Component its imported in the POX controller, so that the L2_multi can utilize the topology information.
3- The Controller's data forwarding component use the information topology to calculate the path for entire underlying topology.
4- The controller insert the whole paths for the packets in the network by modifying the flow tables of all data plane switches on the path by sent Flow_Mod messages, where each entry is contains hard and idle timeout fields.

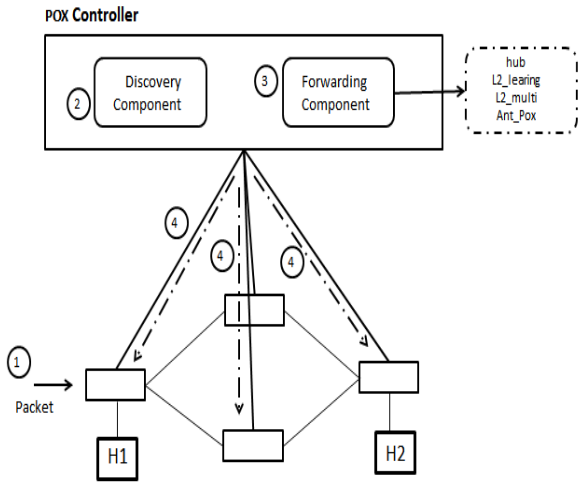


Fig 4: l2_multi and Ant_Pox Approach

7. MININET TOOL

Mininet [14] is an open source network emulator that supports the OpenFlow protocol for SDN architecture. With Python language, Mininet is simple to use and has a great flexibility. It is very powerful LINUX based, uses virtualization approach to create a realistic network of virtual hosts, switches, controllers, and links, and uses process-based virtualization to emulate entities on a single OS kernel by running real code. Moreover, it is used by many researchers because the design that works properly in the Mininet can usually move directly to practical networks composed of real hardware devices.

Mininet provide two ways to use:

- Commad Line Interface (CLI): To control and manage the virtual network from a single console.
- Application programming Interface (API): The Python API allows to createn custom topologies based on scripts.

8 . IMPLEMENTATION

Firstly, the underlying topology that contain data plane switches will simulate.

a. Implementation of the simulation scenario

Firstly, the underlying topology that contain data plane switches will simulate, Mininet API was used to create the topology by writing python script and calling the following functions:
AddLink(): to add link between two switches or switch-host connection.
AddHost(): to add host.
AddSwitch(): to add openflow switch.
The simulation scenario consists of a Five OpenFlow switches connected to Two hosts (host1, host2) and to a controller POX. This controller has four created components called Hub and l2_learning, l2_milti, and Ant_Pox. Fig.5

shows the topology of implementation the simulation scenario.

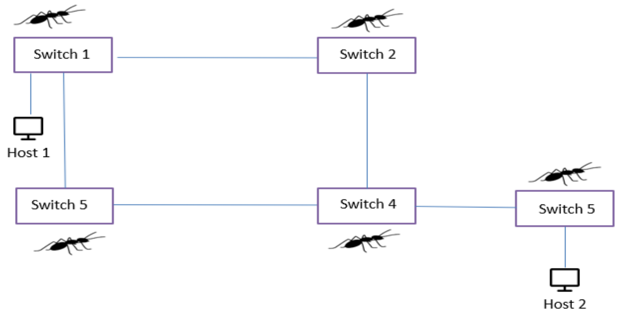
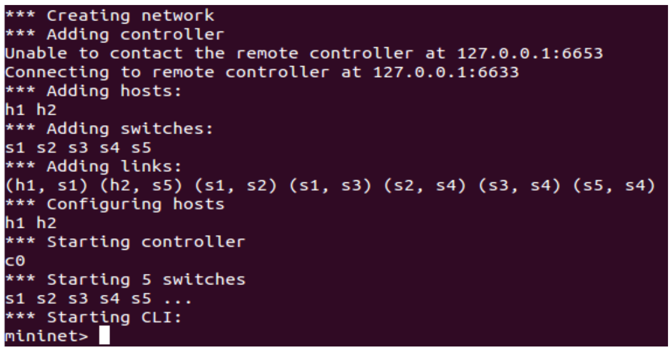


Fig 5: l2_Network Topology

Fig.6 shows the topology implementation using Mininet.



```
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Connecting to remote controller at 127.0.0.1:6633
*** Adding hosts:
h1 h2
*** Adding switches:
s1 s2 s3 s4 s5
*** Adding links:
(h1, s1) (h2, s5) (s1, s2) (s1, s3) (s2, s4) (s3, s4) (s5, s4)
*** Configuring hosts
h1 h2
*** Starting controller
c0
*** Starting 5 switches
s1 s2 s3 s4 s5 ...
*** Starting CLI:
mininet>
```

Fig 6: l2_Network Topology

To create the simulation scenario, two terminals are open, one for Mininet and another for the POX. In the Mininet terminal it was used the commands of Table 1 to build a topology.

Table 1. Implementing the topology

```
# sudo mn –custom topo.py –topo mytopo –controller=remote, ip=127.0.0.1, port=6633
```

The parameters used in Table.2 are described in the table below:

Table 2. Implementing the topology

description	command
Run mininet	mn
Create topology	--topo
Run custom topology	--custom
Use random mac addresses	--mac
Identify the controller	--controller=remote
Identify the controller' ip	--ip
Identify the controller' port	--port

b. Implementing of data forwarding components

Secondly, POX controller will run by implement the data forwarding components.

I. Hub

In this section, a hub component has been present, its works at reactive mode, where every packet come to a data plane, i,e. Switch is sent to the controller by ConnectionUP event that represent the a moment when connection between the controller and switch was established after a handshake process, At this point, the controller requests the switch to egress this packet from all ports except the port where it was received, it generate OpenFlow OFPT_PACKET_OUT message on each received PacketIn event. Table 3 shows the hub application code.

Table 3 . Hub Application Code In POX

```
from pox.core import core
import pox.openflow.libopenflow_01 as of
from pox.lib.util import dpidToStr
log = core.getLogger()
def _handle_ConnectionUp (event):
msg = of.ofp_flow_mod()
msg.actions.append(of.ofp_action_output(port
of.OFPP_FLOOD))
event.connection.send(msg)
log.info("Hubifying %s", dpidToStr(event.dpid))
def launch ():
core.openflow.addListenerByName("ConnectionUp",
_handle_ConnectionUp)
    log.info("Hub running.")
```

In the terminal for the POX, previously opened, you must access the directory/pox and run the Hub component, as shown in Table 4. The file should be in the folder /pox/forwarding/hub.py, and run the following instruction:

Table 4 . Implementation of Hub Component In POX

```
Sudo ~/pox/pox.py forwarding.hub
openflow.discovery
```

II. L2_learning

The L2_Learning component in POX acts as a layer 2 switch, this means that it able to deals and learns the different sources based on their MAC addresses and maps them to their corresponding incoming port, thus it is learns the paths to the hosts, checks the parameters and destination address then forwards the packets accordingly, as well as its keeps tracks of where the host with MAC address is located and accordingly sends packets towards the destination and does not flood it out through all ports.

The absorbing thing that must be noticed in this component is how it work with Flow_Mod messages that inserts entries to the flow table of an OpenFlow Switch, Table 5 shows the L2_learning application code.

The first step is to update the address/port hash table (that is self.macToPort[packet.src] = event.port). This will associate the MAC address of the sender to the switch port on which the packet has been received by the switch. Certain types of the packets are dropped. Multicast traffic is properly flooded. If the destination of the packet is not available in the address/port hash table, the packet is also flooded. If the input and output ports are the same, then the packet will be dropped to avoid loop (if port==event.port:). Finally, a proper flow table entry gets installed inside the flow table of the OpenFlow switch. In summary, the l2_learning.py program implements the required logic and algorithm to change the behavior of our OpenFlow switch to an Ethernet learning switch one

Table 5 . l2_learning Application Code In POX

```
self.macToPort[packet.src] = event.port
if not self.transparent:
if packet.type == packet.LLDP_TYPE or
packet.dst.isBridgeFiltered():
drop()
return
if packet.dst.isMulticast():
flood()
else:
if packet.dst not in self.macToPort:
log.debug("Port for %s unknown -- flooding" %
(packet.dst,))
flood()
else:
port = self.macToPort[packet.dst]
if port == event.port:
log.warning("Same port for packet from %s -> %s on
%s.
Drop." %
(packet.src, packet.dst, port), dpidToStr(event.dpid))
drop(10)
return
log.debug("installing flow for %s.%i -> %s.%i" %
(packet.src, event.port, packet.dst, port))
msg = of.ofp_flow_mod()
msg.match = of.ofp_match.from_packet(packet)
msg.idle_timeout = 10
msg.hard_timeout = 30
msg.actions.append(of.ofp_action_output(port =
port))
msg.buffer_id = event.ofp.buffer_id
self.connection.send(msg)
```

The instruction of execution is the same of the Hub component, in the terminal for the POX; you must access the directory /pox and run the L2_learning component, as shown in Table 6. The file should be in the folder /pox/forwarding/l2_learning.py, and run the following instruction:

Table. 6 Implementation of L2_learning Component In POX

```
Sudo ~/pox/pox.py forwarding.L2_Learning
openflow.discovery
```

III. L2_mulity

The idea behind this module is to have a forwarding Database (forwarding map) for a whole underlying topology. In order to build that map, this module dependent on Discovery module (openflow.discovery) to creates a full map of all the network links (path_map). To avoid having to rebuild the forwarding map on each time the link goes down; L2_multi component does not creates any routes, and openflow packet-forwarding rules are set up on demand, when traffic between two hosts is first seen (not counting LLDP packets). After learn the topology, the controller will install openflow rules so that all the traffic is forward by shortest path, by that point, the network is stable as well as all the routes between each pair has been found.

L2_multi component uses The Floyd-Warshall to calculate the shortest path between each pairs, which is a form of the distance-vector algorithm optimized when a full network map is available. The Floyd-Warshall is an algorithm for calculating the shortest path where the algorithm can find all the distances from each node (all pairs shortest path) which means that it can be used to calculate the smallest weight of all paths connecting a pair of points, and do it all at once for all pairs of points. Table 8 shows the pseudo code for floyed-warshall algorithm that finds the intermediate nodes such that the distance between all the source–destination pairs is minimized.

Table 8. Floyed-Warshall algorithm

```
Floyd-Warshall
SWs = switches.values()
Path_map= D
initialization
for k = 1 to SWs
for i = 1 to SWs
for j = 1 to SWs
if dij > dik + dkj
then dij = dik + dkj
return D
```

L2_multi component contains several dictionaries that handle work, Firstly, store the topology in multi dimension Dictionary (adjacency):

adjacency = defaultdict(lambda:defaultdict(lambda:None))

secondly, store data plane switches in dictionary (switches), which we can get the topology size

switches = {}

thirdly, store the connections between each switch and its ports in dictionary (switches), which we can get the topology size.

Mac_map = {}

Finally, after apply the Floyd-Warshall algorithm, it will store the routes between all switches in the topology in multi dimension dictionary, as shown in table 7:

path_map= defaultdict(lambda:defaultdict(lambda:(None,None)))

Fig.7 illustrate the flowchart for L2_multi component.

Table 7. Path_map dictionary

	Switch 1	**Switch 2**
Switch 1	**[distance, intermediate]**	**[distance, intermediate]**
Switch 2	**[distance, intermediate]**	**[distance, intermediate]**

The previous dictionary structure is an example of a topology composed of two switches, we note that it is three-dimensional, the first dimension represents the source switch and the second dimension represents the target switch, and the third dimension is a tuple composed of two elements, the first element represents the distance between the source and target switch and the second element represents a list of switches or intermediate nodes that exist

between the source and target switch. The Path_map is populated by applying the Floyed-Warshall algorithm, caching distances and intermediate nodes, so in the event of a topology failure, the Path_map dictionary is recalculated again as we explained earlier. This is where the ant colony algorithm comes into play by proactively calculating additional paths, thus improving network stability if the failure occurs.
To implements the L2_multi component:

Table 9 . Implementation of L2_multi Component In POX

Sudo ~/pox/pox.py forwarding.L2_multi openflow.discovery

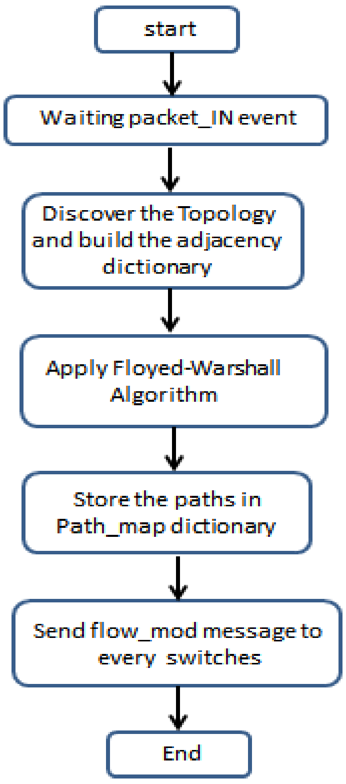

start
Waiting packet_IN event
Discover the Topology and build the adjacency dictionary
Apply Floyed-Warshall Algorithm
Store the paths in Path_map dictionary
Send flow_mod message to every switches
End

Fig 7:L2_multi Component Flowchart

IV. Ant_Pox component:

The problem appears in the previous component when a link goes down in the network topology and then the network will becomes unstable, in this case the controller deletes all the paths stored in the path_map dictionary and re-calculates the paths between each pair of switches again and all the packets that sent in the network would not reach its destination and will be dropped, therefore from the previous explanation we conclude that if we can add mechanism to prevent the process of re-calculating the paths from the beginning when the link failure happens, and thus we can reduce the stability time of the network as well as there will be less computation overhead. In this component, the floyed_warshall will be applied first to calculate the shortest paths between each pair of switches, and then the ACO algorithm [15] will be applied, which takes the output of the first algorithm as its input, instead of assigning random values at the beginning. Therefore we will assigns number of ants equals to the numbers of switches in the topology and calculate the backup path for each switch.

In order to clarify the concept of the ACO algorithm, the topology that build is shown in figure(5), the network topology (switches, and links) has been represented by a diagram compose links and edges, every ant in the real world is a packet in Graph. The algorithm is start randomly from any node in the topology, let suppose that the variables m and n are represents the number of switches and the number of ants respectively , and bi(t) is a function represents the number of ants on the switch(i), so:

$$m = \sum_{i=1}^{n} b_i \ (t) \qquad (1)$$

Each ant has a list of previously visited switches in order to avoid adding them again. The probability of the state transition between two switches is calculated based on the remaining pheromone on each path, and the next switch is chosen according to the probability equation in the ACO algorithm, which expresses the probability of the ant choosing (k) the next node from node (i) to node (j), where the node represents OpenFlow switch and edges are links.

$$p_{ij}^{k}(t) = \begin{cases} \dfrac{[\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}]^{\beta}}{\sum_{ak} [\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}]^{\beta}} & \text{if } j \in a_k \\ 0 & \textit{otherwise} \end{cases} \qquad (2)$$

The function indicates the probability of the ant visiting the path K from the switch Si to Sj. Where:
a_k : represents the switches available to be visited when the Ak ant is in the Sj switch.
Tij(t): represents the total pheromone present in the path between Si-Sj, where tij(0) denotes the initial amount of pheromone.
α and β represent the weight for the pheromone remaining in the path and the path distance in the nezt switch selection process respectively. While the indicative function is represented as follows:

$$1/d_{ij} = \eta_{ij} \qquad (3)$$

d_{ij} represents the distance between Si, Sj and thus the probability of visiting the switch Si to Sj is directly proportional to the value of dij. In our case, the distance (metric) was represented by the bandwidth by the shortest path algorithm[15], the link cost is inversely proportional to the bandwidth of the link (the higher the bath)

$$C(u{\cdot}v) = \frac{1}{BW(u{\cdot}v)} \qquad (4)$$

Where c(u,v) is the cost between the switch u and the switch v, and BW(u.v) is the bandwidth value on the link. Two issues must be discussed in the current Ant algorithm, first the selection of the following switch: in the case of a small value of α , the selection of the next switch is mainly based on the value of dij. And second the case of a small value of β , the next switch is chosen randomly. The pheromone on the path will be updated when the ant bypasses a switch (partial update) or when visit an entire path (global update), the pheromone update equation is:

$$\tau_{ij}(t+n) = (1-p) * \tau_{ij}(t) + \Delta\tau_{ij}(t) \qquad (5)$$

$$\Delta\tau_{ij}^{k}(t) = \begin{cases} 1/Lk & \text{A\_k moves from S\_i to S\_j} \\ 0 & \textit{otherwise} \end{cases} \qquad (6)$$

P represents the pheromone evaporation rate on the track while (1-p) represents the remaining pheromone fraction. The evaporation rate factor in the equation affects the search ability, if the value of p is large then the pheromone evaporation rate will be high, leading to the random selection of the next switch, while if it is a small value, the evaporation rate will be low, which results in an optimal local solution.

$\Delta\tau_{ij}^{k}$ (t) represents the value of the pheromone that added by the ant on the path in one cycle

$\Delta\tau_{ij}^{k}$ (0) represents the initial state of all the paths before runs the algorithm. Each ant in the searching process for the optimal path will use the ACO algorithm to determines the next switch based on the equation (2), and at the same time the positive feeding mechanism is adopted to add more weight to the current optimal path, in other words, the information on the path that has been passed it means giving more weight after each repetition.

Ant_Pox application steps:
Ant is placed on each of the openflow switches.
1- Initialization parameters.
2- Ant k calculate p_{ij}^{k} ,then:
 2-1- select the next openflow switch.
3- If no reach to destination openflow switch, then repeat step2.
4- When cycle path is completed, update pheromone.
5- According to number of iteration end the cycle and complete the routing process.

L2_multi component has been modified to develop ACO algorithm, therefore additional dictionary has been added: Dictionary for fitness function, dictionary for store the probabilities of the ants, and dictionary for the pheromone values which is equal to 1 for all edges at first, then it will be updated at each generation.

```
Fit_map=
defaultdict(lambda:defaultdict(lambda:(None,None))
pher_map=
defaultdict(lambda:defaultdict(lambda:(None,None)))
prob_map=
defaultdict(lambda:defaultdict(lambda:(None,None)))
```

Table 10 shows the pseudo code that added to L2_mutli component to implements ACO algorithm.

Table 10. Floyed_Warshall algorithm

```
//Floyed_Warshall algorithm
For k in sws:
 For i in sws:
   For j in sws:
       If path_map [i][k][0] is not none:
       If path_map [k][j][0] is not none:
          Ikj_dist =path_map [i][k][0] + path_map [k][j][0]
          If path_map [i][j][0] is none  or   Ikj_dist < path_map [i][j][0]
              Path_map [i][j] = (ikj_dist,k) // calculate the paths
              Fit_map[i][j]= 1/path_map[i][j][0] // fitness fun
              Pher_map[i][j]=1.0 // assigns initial pheromone to edges
```

Table 11 shows the pseudo code for equation (2) to calculate the probabilities for each ant.

Table 11. ACO Algorithm

```
// calculate the probabilities matrix based on pheromone
Def calc_prop():
 L=[]
 Sws-switches.values()
 Prop_sum={}
 For i in sws:
    Prop_sum[i]=0
    For j in sws:
      Prop_sum[i]= prop_sum[i]+fit_map[i][j]*pher_map[i][j]
 For I in sws:
   For j in sws:
     Pro[i][j]= fit_map[i][j]*pher_map[i][j] / prop_sum[i]

// print probabilities map
Calc_prob()
For i in sws:
   For j in sws:
      Print i,j,prop[i][j]
```

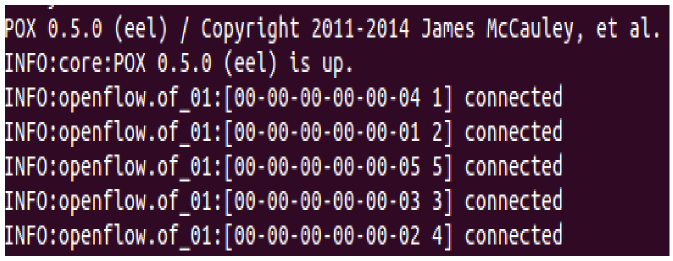
In order to run the Ant_Pox application in the controller, the following instruction is executed:

Table 12 . Implementation of Hub Component In POX

```
Sudo ~/pox/pox.py forwarding.AntPOX.Py
openflow.discovery
```

9. RESULTS AND DISCUSSION

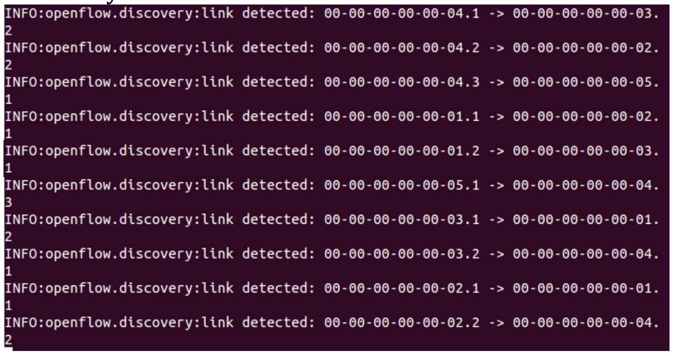
In Ant_Pox Component we developed a protection approach which does not require controller-switch communications when the failure occurs. The Ant_Pox component pre-establish backup paths as well as the paths that already has been stored in Path_Map dictionary. In this case the network resilience has improved. This mechanism allows the restoration of traffic without sending the messages at the time of failure.
After running the Ant_Pox Component, the controller will discover the topology, we can notice that every switch has its own DPID, Fig.8 shows the detected topology which contain five switches:



```
POX 0.5.0 (eel) / Copyright 2011-2014 James McCauley, et al.
INFO:core:POX 0.5.0 (eel) is up.
INFO:openflow.of_01:[00-00-00-00-00-04 1] connected
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
INFO:openflow.of_01:[00-00-00-00-00-05 5] connected
INFO:openflow.of_01:[00-00-00-00-00-03 3] connected
INFO:openflow.of_01:[00-00-00-00-00-02 4] connected
```

Fig 8: Discovered Topology using Discovery component

Fig.9 illustrate the discovered links between switches in the topology and it ports, where they are stored in the mac_map dictionary.



```
INFO:openflow.discovery:link detected: 00-00-00-00-00-04.1 -> 00-00-00-00-00-03.
2
INFO:openflow.discovery:link detected: 00-00-00-00-00-04.2 -> 00-00-00-00-00-02.
2
INFO:openflow.discovery:link detected: 00-00-00-00-00-04.3 -> 00-00-00-00-00-05.
1
INFO:openflow.discovery:link detected: 00-00-00-00-00-01.1 -> 00-00-00-00-00-02.
1
INFO:openflow.discovery:link detected: 00-00-00-00-00-01.2 -> 00-00-00-00-00-03.
1
INFO:openflow.discovery:link detected: 00-00-00-00-00-05.1 -> 00-00-00-00-00-04.
3
INFO:openflow.discovery:link detected: 00-00-00-00-00-03.1 -> 00-00-00-00-00-01.
2
INFO:openflow.discovery:link detected: 00-00-00-00-00-03.2 -> 00-00-00-00-00-04.
1
INFO:openflow.discovery:link detected: 00-00-00-00-00-02.1 -> 00-00-00-00-00-01.
1
INFO:openflow.discovery:link detected: 00-00-00-00-00-02.2 -> 00-00-00-00-00-04.
2
```

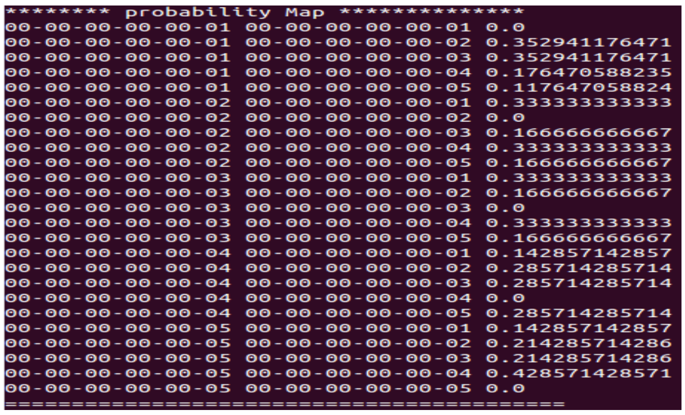
Fig 9: Discovered links using Discovery component

ACO algorithm is implemented with the following parameters:

Table 13. ACO parameters

Parameter	value
Number of ants	30
Number of Generations	10
Alpha value	0.5
Beta value	0.01
Initial pheromone	1

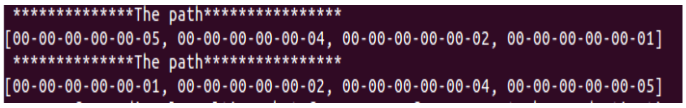
Fig.10 shows the implementation of the ACO algorithm and the probabilities (according to the equation (2)) of each ant to move from a switch to all switches in the topology, where a number of ants is equal to the number of switches. First line means that the probabilities from switch1 to switch1 is zero while the second line means that the probabilities from switch 1 to switch 2 is equal to 3.529, and so on.



```
********* probability Map *****************
00-00-00-00-00-01 00-00-00-00-00-01 0.0
00-00-00-00-00-01 00-00-00-00-00-02 0.352941176471
00-00-00-00-00-01 00-00-00-00-00-03 0.352941176471
00-00-00-00-00-01 00-00-00-00-00-04 0.176470588235
00-00-00-00-00-01 00-00-00-00-00-05 0.117647058824
00-00-00-00-00-02 00-00-00-00-00-01 0.333333333333
00-00-00-00-00-02 00-00-00-00-00-02 0.0
00-00-00-00-00-02 00-00-00-00-00-03 0.166666666667
00-00-00-00-00-02 00-00-00-00-00-04 0.333333333333
00-00-00-00-00-02 00-00-00-00-00-05 0.166666666667
00-00-00-00-00-03 00-00-00-00-00-01 0.333333333333
00-00-00-00-00-03 00-00-00-00-00-02 0.166666666667
00-00-00-00-00-03 00-00-00-00-00-03 0.0
00-00-00-00-00-03 00-00-00-00-00-04 0.333333333333
00-00-00-00-00-03 00-00-00-00-00-05 0.166666666667
00-00-00-00-00-04 00-00-00-00-00-01 0.142857142857
00-00-00-00-00-04 00-00-00-00-00-02 0.285714285714
00-00-00-00-00-04 00-00-00-00-00-03 0.285714285714
00-00-00-00-00-04 00-00-00-00-00-04 0.0
00-00-00-00-00-04 00-00-00-00-00-05 0.285714285714
00-00-00-00-00-05 00-00-00-00-00-01 0.142857142857
00-00-00-00-00-05 00-00-00-00-00-02 0.214285714286
00-00-00-00-00-05 00-00-00-00-00-03 0.214285714286
00-00-00-00-00-05 00-00-00-00-00-04 0.428571428571
00-00-00-00-00-05 00-00-00-00-00-05 0.0
========================================
```

Fig 10: probabilities calculations

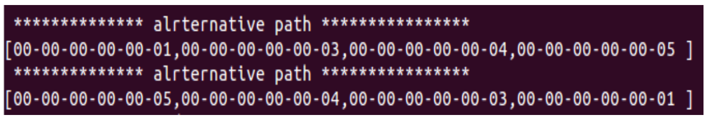
In this evaluation, host h1 which is connected to switch S1 ping to reach the target host h2 which is connected to switch S5. Accoutering to the Ant_Pox application the minimum path is s1-s2-s4-s5 as shown in Fig.11.



```
*************The path***************
[00-00-00-00-00-05, 00-00-00-00-00-04, 00-00-00-00-00-02, 00-00-00-00-00-01]
*************The path***************
[00-00-00-00-00-01, 00-00-00-00-00-02, 00-00-00-00-00-04, 00-00-00-00-00-05]
```

Fig 11: the primary path between h1 and h2

the Fig.12 shows The Alternative path.



```
************* alrternative path ***************
[00-00-00-00-00-01,00-00-00-00-00-03,00-00-00-00-00-04,00-00-00-00-00-05 ]
************* alrternative path ***************
[00-00-00-00-00-05,00-00-00-00-00-04,00-00-00-00-00-03,00-00-00-00-00-01 ]
```

Fig 12: the backup path between h1 and h2

10. Evaluation of POX forwarding Approaches

For performance evaluation and comparison of different forwarding algorithms depicted in section 8, a topology as shown in Fig.5 is selected. All the forwarding algorithms presented in Section 8 have been evaluated with Discovery Component. For comparison of different algorithms, different attributes or metrics are identified. These attributes are Round Trip Time (RTT) and CPU usage for initial flow setup.

- **CPU usage of forwarding algorithm**

The CPU usage is measured using Linux top command for the entire forwarding algorithm to find the computational overhead of different algorithms. As we can see in Fig.13, hub and L2_learning have lower overhead than that of the other algorithms, this deviation is caused by the way they find the paths, when L2_multi and Ant_Pox algorithms is started, it requires more overhead due to more initial processing. After finding the paths, they have normal overhead, that because they do no needs run any calculations.

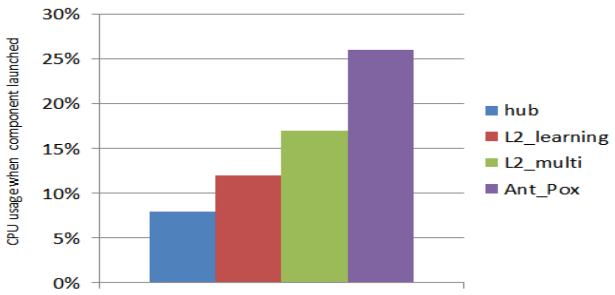


Fig 13: CPU usage when component launched

- **CPU usage for initial flow setup**

When a switch receives a packet from any source the first time it will install the flow entry in the flow table. Thus after learning all the nodes in the network, the complete flow establishment has taken place. The pingall command in mininet is used to check the connectivity of the entire network. The pingall command is sent from every host to all the other hosts. The CPU usage is measured for the initial flow establishment. Fig.14 shows that Hub and L2_learning have higher CPU usage than the other. The reason is that

L2_multi and Ant_Pox calculate the paths and store them in Path_Map dictionary once the links and nodes are detected, thus when runs the pingall command these two algorithms requires less packets to find the destination, instead of just flooding the packets, for that they have low CPU usage.

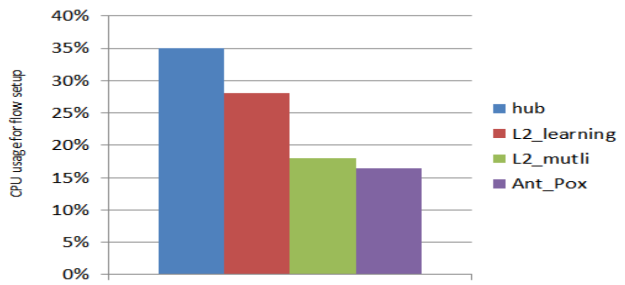


Fig 14: CPU usage for flow setup

- **Round Trip Time**

Ping tool was used to measure the RTT for the first packet, which can indicate the delay that each component takes to find the path. Fig.15 shows the RTT time for first packet, where we can notice that the time that L2_multi and Ant_Pox component takes more than the other, in spite of the flooding that happens in hub and L2_leaning, that due to L2_multi and Ant_Pox implements more complex calculations at first to find and store the paths at Path_Map dictionary, Ant_Pox component will have the higher time for first packet because it finds the backup paths and store them too in case of the failure happens.

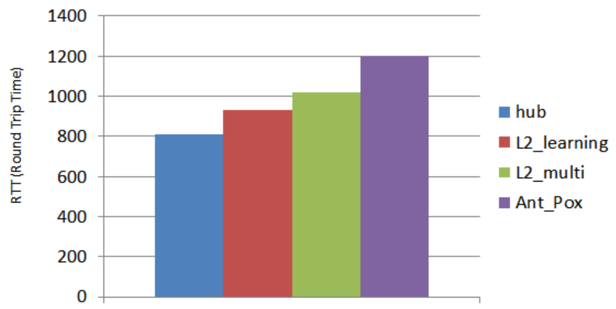


Fig 15: RTT time for first packet

Table 14. illustrate the main characteristic for POX Forwarding algoritms, Failure detection is facilitated by the Discovery component as it triggers the link events, the first two algorithm, eg., hub, L2_leaening do not react against the failure, they simply repeat its processes and eventually establish the new flow, while both L2_multi and Ant_Pox immediately react to the failure. complex configuration computation is initially performed using approaches L2_mulit, Ant_Pox, they both calculate the shortest paths, but the different is that the developed Ant_Pox component is calculates backup paths, while L2_multi it will wipes out the entire stored paths (Path_map) and starts the process of calculation paths again in case the failure happens, therefore we added mechanism to pre-calculate the paths for all the sources and destinations.

Table 14. summary of main characteristic for POX forwarding algorithms

Feature	Hub component	L2_learning component	L2_multi component	Ant_POX component
Failure Detection	✓	✓	✓	✓
Failure Recovery	☒	☒	✓	✓
Initial configuration computation	☒	☒	✓	✓
Shortest Path calculation	☒	☒	✓	✓
Backup path	☒	☒	☒	✓

11. CONCLUSION

In this paper, a new protection approach has been introduced by modifying the L2_multi component, and develop ACO algorithm to find backup paths in case the failure occurs. Four data forwarding algorithm in POX controller components has been investigated and compared. This comparison helps better understand the forwarding approaches in POX and future enhancement.

12. REFERENCES

[1] Mulyana, E. SDN-RG Community Books. Bandung: GitBook, 2014.

[2] Marcel Caria, Admela Jukan, and Marco Hoffman," A performance study of network migration to SDN-enabled Traffic Engineering ", Globecom 2013-Communication Qos, Reliability and Modeling Symposium 2012.

[3] Heleno Isolani p, "Interactive Monitoring, Visualization, and Configuration of OpenFlow-based SDN" IEEE International Symposium on Integrated Network Management, 2015.

[4] Open Networking Foundation. Available from: https://www.opennetworking.org/, last online : 2019/3/4.

[5] Azodolmolky S. Software defined networking with OpenFlow: Packt Pub, Birmingham, UK . 2013

[6] Fishnet Security, "SDN APIs: A New Vocabulary for Network Engineers",https://www.fishnetsecurity.com/6labs/blog/sdn-apis-new-vocabulary-network-engineers

[7] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, S. Uhlig, Softwaredefined networking: A comprehensive survey, Proc. of the IEEE 103 (1) (2015) 14-76.

[8] H. Kim, N. Feamster, Improving network management with software defined networking, IEEE Communications Magazine 51 (2) (2013) 114-119.

[9] F. Hu, Q. Hao, K. Bao, A survey on software-defined network and openflow: From concept to

implementation, IEEE Communications Surveys & Tutorials 16 (4) (2014) 2181-2206

[10] Sumanth B. Designing an Openflow Controller for data delivery with end-to-end QoS over Software Defined Networks: Computer Science and Engineering; Conference in Hollywood, CA, USA 2016.

[11] Lara, A.; Kolasani, A.; Ramamurthy, B. Network Innovation Using OpenFlow: A Survey.IEEE Commun. Surv. Tutor. 2013,16, 1–20.

[12] POX, "Pox openflow controller," 2014, Accessed: Sept.2014.[Online].Available

[13] Python Software Foundation, "Python language reference, version

[14] Mininet. An Instant Virtual Network on your Laptop.2014, Accessed: Sept. 2014[Online]Available: http://mininet.org.

[15] M. Dorigo, and C. Blum, "Ant colony optimization theory: a survey", in Theoretical. Computer Science, vol. 344, no. 2-3, pp. 243-278, 2005.

BIOGRAPHIES

Dr.Eng. Souheil Khawatmi
Associative Professor,
Computer Networks Department,
Faculty of Informatics Engineering
University of Aleppo, Syria.

Dr.Eng. Fadel Sukkar
Professor, Artificial Intelligence and Natural Language Department,
University of Aleppo, Syria

Eng. Mahmoud Khatib
Postgraduate Student (M.S),
Computer Networks Department,
Faculty of Informatics Engineering
University of Aleppo, Syria.