Analyzing and Comparing Data Forwarding Components in POX Software Defined Networking Controller

Mahmoud Khatib
Postgraduate Student (MSc)
Systems and Computer Networks Dept
University of Aleppo, Syria

Souheil Khawatmi
Assistant Professor
Systems and Computer Networks Dept
University of Aleppo, Syria

Fadel Sukkar
Professor
Artificial Intelligence and Natural Language Dept
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, where three components are compared, hub, l2_learning, and l2_multi, by measures the Round Trip Time (RTT) and CPU usage.

Keywords
Software Defined Network(SDN), OpenFlow protocol, POX controller, Mininet, data forwarding components, Round Trip Time (RTT), 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 analyze the POX controller and study the forwarding components that pox supports. 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 discuss the discovery metrology. Section eight introduce the simulation tool that used. Section nine analyze the forwarding components. Section ten is reserved to the results. Finally, conclusions and future works are 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, figure.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.

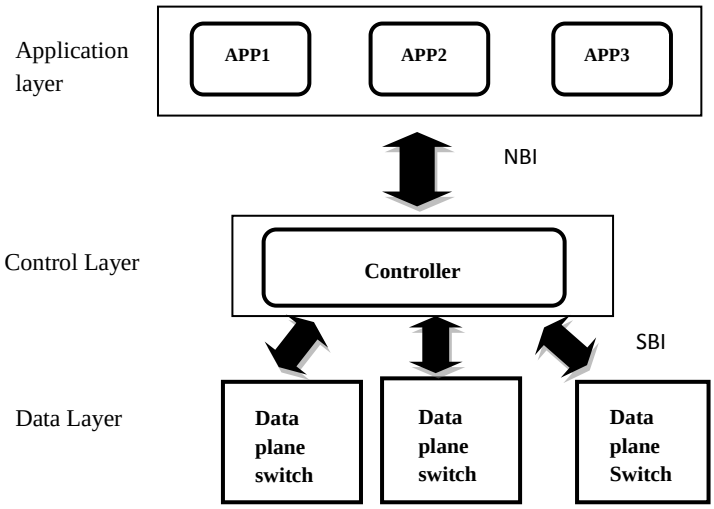

Fig 1: SDN paradigm

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].

3.1.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 figure. 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 Figure. 3 [11].

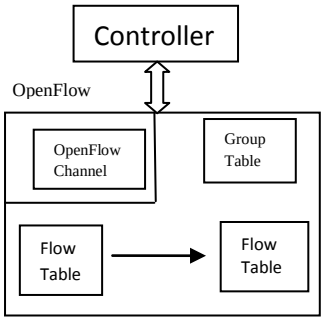

Fig 2: OpenFlow Architecture

3.2. OpenFlow Messages

OpenFlow Protocol has different messages (events) that are fired under certain condition [12]:

- Packet_In message: sent by the switch to the controller when the switch receives a flow that does not match with any rule in its flow table.
- Packet_Out: is initiated by the controller, used to configure the switch, manage the switch's flow table.
- FlowMod message: set from the controller to the switch in order to insert the necessary forwarding rules. The controller specify in this message an idle (The absolute timeout in which if there are no packets hitting the flow for the duration, then flow is removed from the device.) and a hard timeout (The absolute timeout after which the flow is removed from the device.).

4. FORWARDING MECHANISM USING OPENFLOW

The basic packet forwarding mechanism with OpenFlow is illustrated in figure. 3. When a switch receives a packet, it analyzes the packet header, which is then matched against the flow table. If there is a match found with the header field, then the flow table entry is considered. If several such entries are found, packets are matched based on prioritization, i.e., the most specific entry or the wildcard with the highest priority is selected. Then, the switch updates the counters of that particular flow table entry. Finally, the switch performs the actions specified by the flow table entry on the packet, e.g., the switch forwards the packet to a port. Otherwise, if no flow table entry matches the packet header, the switch generally notifies its controller about the packet (Table-Miss entry), which is buffered when the switch is capable of buffering. To that end, it encapsulates either the unbuffered packet or the first bytes of the buffered packet using a PACKET-IN message and sends it to the controller, The controller receives the PACKET-IN notification identifies the correct action for the packet and installs one or more appropriate entries in the requesting switch. Buffered packets are then forwarded according to these rules; this is triggered by setting the buffer ID in the flow insertion message or in explicit PACKET-OUT messages.

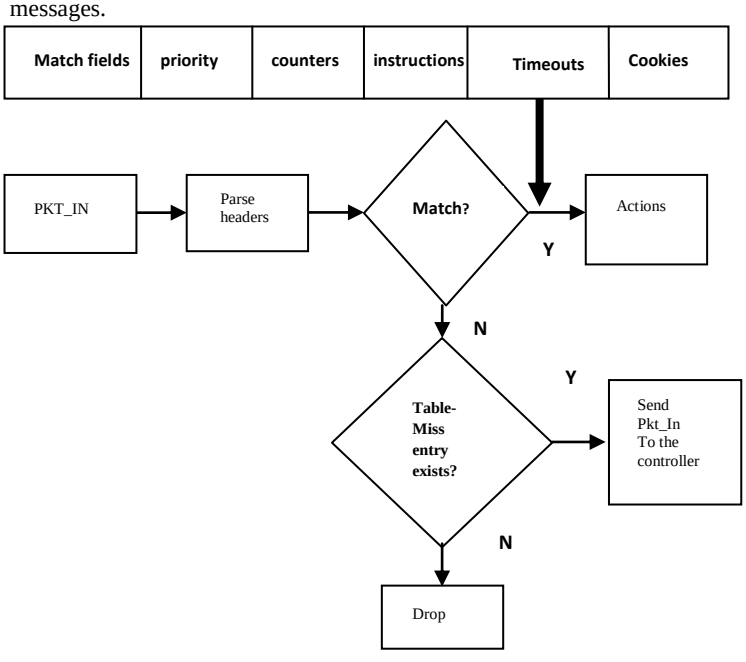

Fig 3: Forwarding Mechanism

5. POX Controller

POX is a software platform developed in Python, it is began early as a controller for an OpenFlow protocol[13][14]. 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.

6. UNDERLYING TOPOLOGY DISCOVRY

The topology discovery is a unique feature of SDN which allow the controller to facilitate the applications in the application layer [15], For instance, a forwarding application uses the network topology to forward the network traffic to its destination [16][17][18], In that way, A POX controller incorporates various core components that assist in executing various SDN applications [19]. One of the main responsibilities of POX is to interact with OpenFlow switches which do not support any topology discovery functionality, and it therefore needs to be implemented as a service at the controller. For this purpose, a separate component (openflow.of_01) is registered to the Core as soon as the controller is fired up (ConnectionUp event). Another component (openflow.discovery) uses for topology Discovery, this component use the OFDP (OpenFlow Discovery Protocol) which is the protocol used by OpenFlow controllers to discover the underlying topology, that sends specially LLDP (Link Layer Discovery Protocol) packets out of the switches, as well as packet_In and Packet_out events are needed to discover links in a network.

7. POX FORWARDING APPROACHES

This section presents an overview of the three main components of the forwarding functionality used in the current POX [20]. 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 research, there data forwarding components in POX controller are discussed, Hub, L2_learning, and L2_multi components. Figure 4, illustrate the steps that hub and L2_learning data forwarding components follow :

1- Host 1 generate a new request packet the to destination (Host 2).
2- Initially, the flow table will be empty (no entry found), so the data plane switch will encapsulate the packet inside Packet_In message (apply Table-miss entry), then inform that controller about that flow.
3- The controller will run the data forwarding component.
4- The controller install new flow in the switch for that flow into the flow table.
5- sent the path through Packet_Out messages to every intermediate data plane switches on the path to insert the entry inside the flow table.

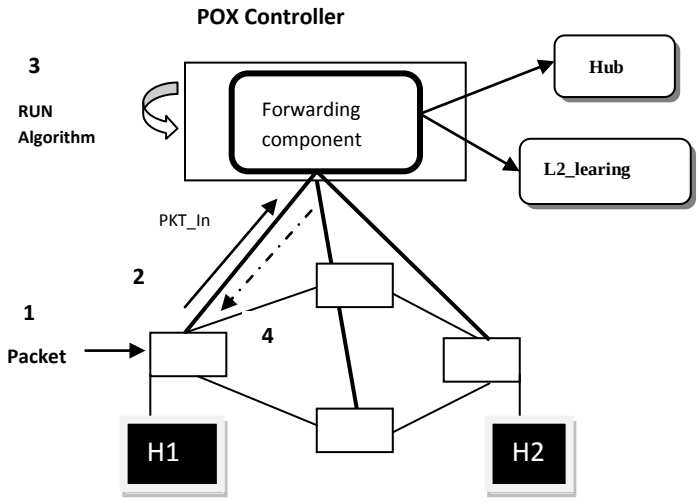

Fig 4: Hub and l2_learning Approach

Figure 5, illustrate the steps L2_multi data forwarding component 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

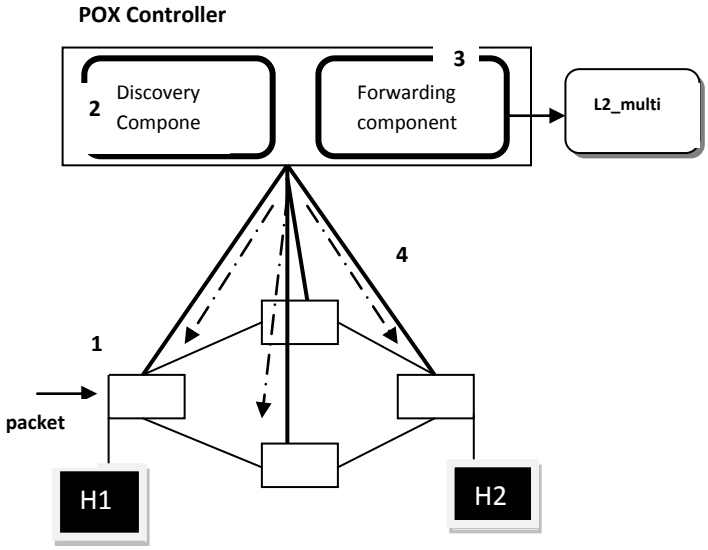

Fig 5: l2_multi Approach

8. MININET TOOL

Mininet [21] 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:

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

9. IMPLEMENTATION

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

a. Implementation of the simulation scenario

Mininet'CLI was used to create a Tree topology, Mininet implementation of the simulation scenario. The simulation scenario consists of a six OpenFlow switches with linear topology (S1, S2, S3,S4,S5,S6,S7) connected to six hosts (h1, h2, h3, h4, h5, h6,h7,h8) and to a controller POX.

This controller has three created components called Hub and l2_learning, and l2_milti. Figure 6 shows the topology of implementation the simulation scenario.

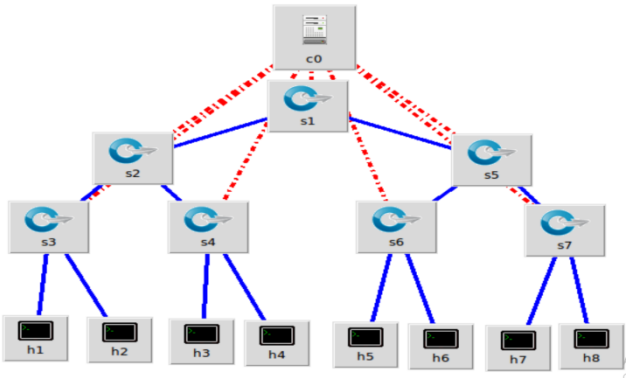

Fig 6: Network Topology

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 . Implementation of Hub Component In POX

```
$ sudo mn –topo tree,3 -- controller remote ,ip=127.0.0.1,
port=6533
```

The parameters used in table are described below:

- **mn**: it starts the CLI Mininet.
- **--topo tree,3**: it creates a topology with 7 switches and 8 virtual hosts.
- **--controller remote**: it sets the openflow switch to connect to a remote controller.
- **-- ip=127.0.0.1**: its loopback address which means that the POX controller running on the same VM.
- **--port=6533**: POX controller's port number.

b. Implementation 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 2 shows the hub application code.

Table 2 . 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.")
```

connectionUP function : a handler for connectionup events its invoked when a switch first connects to the controller.

In POX any component is invoked by the Launch functionWhen the application is started, this function is automatically invoked, and is a function that POX calls to tell the component to initialize itself.

In the terminal for the POX, previously opened, you must access the directory/pox and run the Hub component, as shown in Table 3. The file should be in the folder /pox/forwarding/hub.py, and run the following instruction:

Table 3 . Implementation of Hub Component In POX

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

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 4 shows the L2_learning application code.

Table 4 . 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 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.

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 5. The file should be in the folder /pox/forwarding/l2_learning.py, and run the following instruction:

Table 5 . Hub Application Code In POX

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

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.

After the created Underlying topology, the POX controller will be able to remotely connect to It. Table 6 shows the topology discovery process that happened after connects the Mininet'topology to the POX controller, it can be notice that after the connection is established (connection_up event is fired up) openflow.of_01component has runs, as well as OFDP protocol that discovers the links between data plane switches.

Table 6. Underlying topology discovery

```
INFO:openflow.of_01:[00-00-00-00-00-04 1] connected
INFO:openflow.of_01:[00-00-00-00-00-06 2] connected
INFO:openflow.of_01:[00-00-00-00-00-03 4] connected
INFO:openflow.of_01:[00-00-00-00-00-01 3] connected
INFO:openflow.of_01:[00-00-00-00-00-05 6] connected
INFO:openflow.of_01:[00-00-00-00-00-02 5] connected
INFO:openflow.discovery:link detected: 00-00-00-00-00-04.1 -> 00-
00-00-00-00-01.2
INFO:openflow.discovery:link detected: 00-00-00-00-00-04.2 -> 00-
00-00-00-00-03.1
INFO:openflow.discovery:link detected: 00-00-00-00-00-04.3 -> 00-
00-00-00-00-05.1
INFO:openflow.discovery:link detected: 00-00-00-00-00-06.1 -> 00-
00-00-00-00-02.2
INFO:openflow.discovery:link detected: 00-00-00-00-00-06.2 -> 00-
00-00-00-00-03.2
INFO:openflow.discovery:link detected: 00-00-00-00-00-06.3 -> 00-
00-00-00-00-05.2
INFO:openflow.discovery:link detected: 00-00-00-00-00-03.1 -> 00-
00-00-00-00-04.2
INFO:openflow.discovery:link detected: 00-00-00-00-00-03.2 -> 00-
00-00-00-00-06.2
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-
00-00-00-00-04.1
INFO:openflow.discovery:link detected: 00-00-00-00-00-05.1 -> 00-
00-00-00-00-04.3
INFO:openflow.discovery:link detected: 00-00-00-00-00-05.2 -> 00-
00-00-00-00-06.3
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-
00-00-00-00-06.1
```

L2_mutli 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-Wallshall 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 7 shows the psudo code for floyed-warshall algorithm that finds the intermediate nodes such that the distance between all the source–destination pairs is minimized.

Table 7. 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

10. RESULTS AND DISCUSSION

After running the data forwarding components, the network turned to be monitored by the POX controller. Analyzing network behavior with some commands in the CLI of Mininet, different results has been obtained as below:

Ping tool was used to measure the Round Trip Time (RTT), also known as ping time, that tells the time required to send a packet towards a specific destination and receive a response.

In this experiment, the ping command has execute where ICMP (Internet Control Message Protocol) packets have transmitted in three different scenarios for three data forwarding components, hub, l2_learnining, and l2_multi, from host1 to host4, host1 to host6, and host1 to host8. The purpose of the ICMP measurement is to evaluate the additional delay introduced by the controller for the first Packet_in messages of a flow, this additional delay comes from the communication between switch and controller, that because at the beginning data plane switch's flow tables are empty, therefore first response is always the longest in terms of delay compared to others delay, But its variable for each component. The POX's behavior has been Analyzed by measure RTT through a ping transaction In two cases:

- **Case 1**: When ARP Tables and Flow Tables are empty.
- **Case 2**: When ARP Tables are empty and Flow Tables initialized.

Figure 7 illustrate the RTT time for the first packet response for the three components In case 1:

- hub component: the delay is the smallest, which are host1 to host4 : 14.5 ms, host1 to host6 : 79 ms, and host1 to host8 : 83.1, that because there is no complicated processes, it just flooding frames, thought it will be bigger as the topology grows.
- l2_learninng component: the delay is bigger than hub component, which are host1 to host4 : 86.1 ms, host1 to host6 : 184 ms, and host1 to host8 : 271. Since this component act like a switch behavior, that learns the path as the packets arrive at the switch (learns mac addresses) as well as installing entries in flow tables through Flow-Mod messages, as Table 2 and figure 4 shown. In a nutshell this component have some logic that incuse the response time for first packet.
- l2_multi component: the delay is the highest, which are host1 to host4 : 304 ms, host1 to host6 : 841 ms, and host1 to host8 : 1100 ms. Considering that this component has to apply a shortest path algorithm (Floyed-Warshall algorithm) to calculate the shortest paths between each pairs, and store the path in n-multi dimensional dictionary (path_map), along with installing the paths to each intermediate switches through Flow-Mod messages, as figure 5 shown.

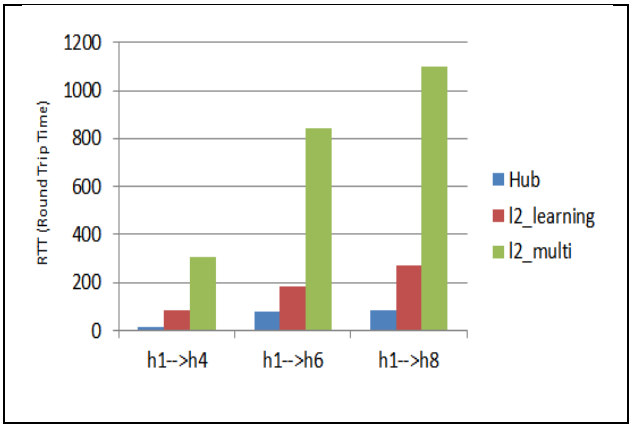

Fig 7. Case 1

Figure 8 illustrate the RTT time for the first packet response for the three components In case 2, After delete the ARP Tables for host1, different results has been obtained as below:

- hub component: the delay had increased, which are host1 to host4 : 10.2 ms, host1 to host6 : 5 ms, and host1 to host8 : 3.37, because the switch have to ask the POX controller via Packet_In message to rebuilt the flow table.
- l2_learninng component: the delay is smaller than Hub component, which are host1 to host4 : 0.118 ms, host1 to host6 : 0.293 ms, and host1 to host8 : 0.513 ms. Since this component learnt already the paths, there is no need to interact with the POX controller.
- l2_multi component: the delay is nearly as l2_learming component, which are host1 to host4 : 0.182 ms, host1 to host6 :0.264 ms, and host1 to host8 : 0.296 ms. The Floyed-Warshall is early calculates the paths between hosts and stories them, hence the switch does have to consulates the POX controller, it just forward the packets according to flow table entries.

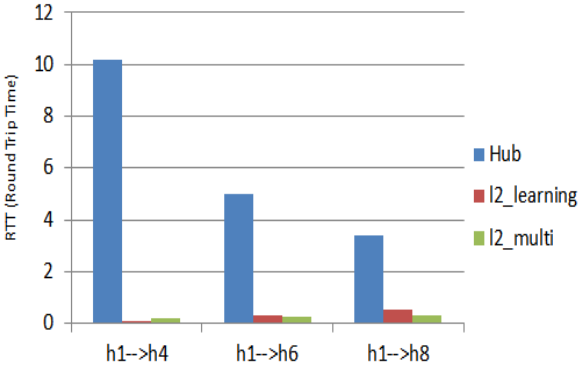

Fig 8. Case 2

In the second experiment, ten ICMP packets have transmitted from host1 to host4, host1 to host6, and host1 to host8, and the minimum, maximum, and average values determined

Figure 9 demonstrate Hub component where minimum RTTs are (host1-host4) 0.125ms, (host1-host6) 0.072ms (host1-host8) 0.098ms: average RTTs are (host1-host4) 0.165 ms, (host1-host6) 0.181 ms, and (host1-host8) 0.208 ms, and maximum RTTs are (host1-host4) 0.436 ms, (host1-host6) 0.751 ms, and (host1-host8) 0.766 ms.

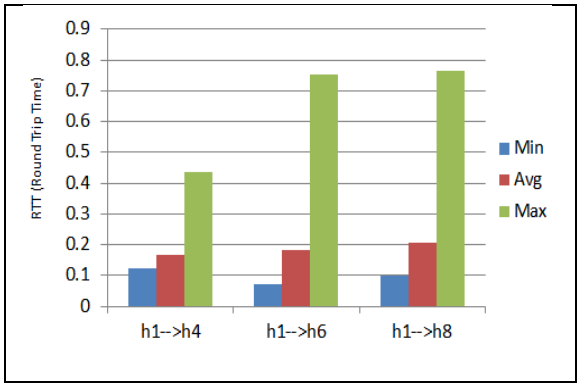

Fig 9. Hub component

Figure 10 indicate l2_learning component minimum RTTs are (host1-host4) 0.12ms, (host1-host6) 0.132ms (host1-host8), 0.069 ms: average RTTs are (host1-host4) 0.227 ms, (host1-host6) 0.212 ms, and (host1-host8) 0.239 ms, and maximum RTTs are (host1-host4) 1.078 ms, (host1-host6) 0.855 ms, and (host1-host8) 1.236 ms.

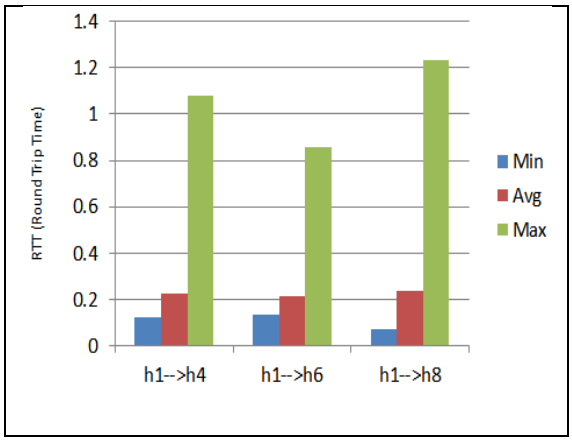

Fig10. l2_learning

Figure 11 illustrate l2_multi component, where the minimum RTTs are (host1-host4) 0.059 ms, (host1-host6) 0.104 ms (host1-host8) 0.101, average RTTs are (host1-host4) 15,842 ms, (host1-host6) 18.593 ms, and (host1-host8) 0.208 ms, and maximum RTTs are (host1-host4) 157.335 ms, (host1-host6) 184.614 ms, and (host1-host8) 196.085 ms.

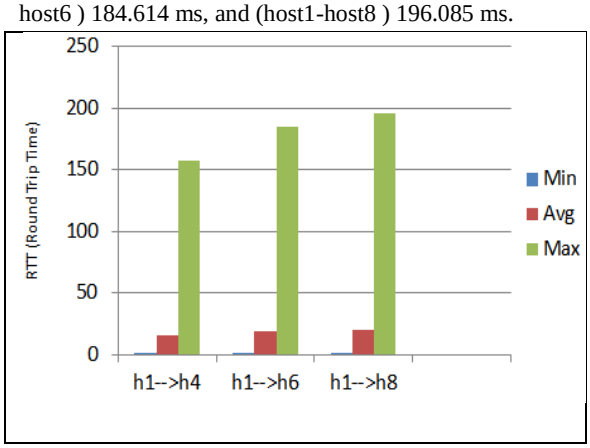

Fig 11. l2_multi

We can conclude that multi component have the largest delay, that due to the complex logic, then l2_learning, and finally the hub component that floods the packets.

In the third experiment, The CPU usage is measured for the initial flow establishment for tree topology contain 31 data plane switch and 32 hosts. Figure 12 shows that l2_learning and l2_multi have higher CPU usage the hub component. The possible reason is that those components have to do some logic, not just floods the packets as hub component do.

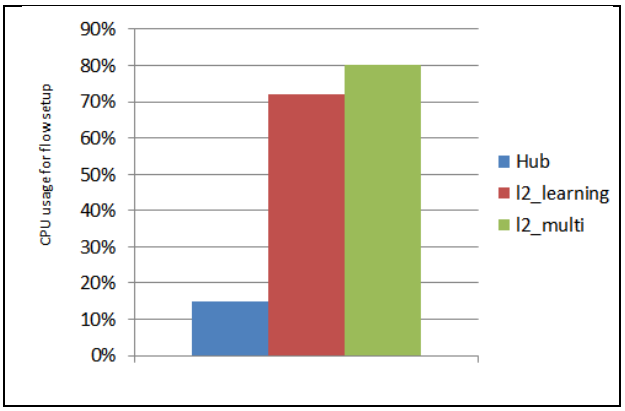

Fig 12. CPU usage for flow setup

11. CONCLUSIONS AND FUTURE WORK

In this paper, the three existing 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. The first experimental shows that l2_multi component has better delay because its uses a shortest path algorithm, although it needs high time to setup the paths. Second experimental emphasize the previous results, l2_learning and l2_multi have high delay due to it require to implement a algorithm to find the paths, while hub component only flood the packets to every ports. Finally, in the third experimental shows the CPU usage that needs each component for initial flow setup. In future, l2_multi component will be expanded, so it can find alternative paths, and simulate much bigger topology.

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] https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.2.pdf

[13] POX, "Pox openflow controller," 2014, Accessed: Sept.2014.[Online].Available: http://www.noxrepo.org/pox/about-pox.

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

[15] S. Shenker, M. Casado, T. Koponen, and N. McKeown, "The future of networking, and the past of protocols," *Open Networking Summit*, vol. 20, 2011.

[16] V. Kotronis, X. Dimitropoulos, and B. Ager, "Outsourcing the routing control logic: Better Internet routing based on SDN principles," in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, 2012, pp. 55-60.

[17] C. Staff, "A purpose-built global network: Google's move to SDN," *Communications of the ACM*, vol. 59, pp. 46-54, 2016.

[18] O. A. Mahdi, A. W. A. Wahab, M. Y. I. Idris, A. A. Znaid, Y. R. B. Al-Mayouf, and S. Khan, "WDARS: A Weighted Data Aggregation Routing Strategy with Minimum Link Cost in Event-Driven WSNs."

[19] J. C. Mogul, A. AuYoung, S. Banerjee, L. Popa, J. Lee, J. Mudigonda, *et al.*, "Corybantic: towards the modular composition of SDN control programs," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, 2013, p. 1

[20] Pox Source code at https://github.com/noxrepo/pox.

[21] [21]Mininet. An Instant Virtual Network on your Laptop.2014, Accessed: Sept. 2014[Online]Available: http://mininet.org.