International Journal of Applied Information Systems (IJAIS) – ISSN : 2249-0868
Foundation of Computer Science FCS, New York, USA
Volume 12 – No. 8, November 2017 – www.ijais.org

Improved Extended Dijkstra's Algorithm for Software Defined Networks

Abdul-hafiz Abdulaziz
Dept of Electrical & Computer Engineering
Ahmadu Bello University, Zaria, Kaduna State - Nigeria

Emmanuel Adewale Adedokun
Dept of Electrical & Computer Engineering
Ahmadu Bello University, Zaria, Kaduna State - Nigeria

Sani Man-Yahya
Dept of Electrical & Computer Engineering
Ahmadu Bello University, Zaria, Kaduna State - Nigeria

ABSTRACT
The existing Extended Dijkstra's algorithm for Software Defined Networks was developed to handle shortcomings associated with the traditional shortest path routing used in SDN. Today's SDN controllers route allocation mechanism is mainly based on Dijkstra algorithm. However, both approaches do not consider bandwidth utilization and do not take knowledge of the topology into consideration. This may result in network congestion and sub-optimal performance of applications. This paper presents a modified Extended Dijkstra's algorithm for Software Defined Networks using REST APIs and introduces a congestion control component responsible for handling traffic overhead in an SDN topology. The Abilene network topology was used to evaluate the performance of both approaches using throughput and latency as performance metrics.

Keywords
Software Defined Networking, REST APIs, Load Balancing, Congestion Control

1. INTRODUCTION
Software Defined Networking (SDN) is a newly emerging field in computer networks. The main goal of SDN is for a network to be open and programmable [2]. This brings many new network applications realized by programming the SDN controller. Typical examples include traffic engineering, security, Quality of Service (QoS), routing, load balancing and so on [6]. In recent years, various load balancing methods for Data Center Networks (DCNs) using the SDN paradigm have been introduced. A load balancing algorithm called LABERIO (LoAd-Balancing Routing wIth OpenFlow), to minimize latency and maximize the network throughput was proposed by Hui *et al.,* (2013). A Plug-n-Serve system implementing a load balancing algorithm called LOBUS (Load-Balancing over Unstructured networks), using OpenFlow for unstructured networks was proposed by Handigol *et al.,* (2013). LOBUS maintains the network topology and link status, and greedily chooses the client-server pair that yields the lowest total response time for each newly arriving request. Dijkstra algorithm, the classical shortest path algorithm used by many routing protocols for finding the shortest path two points in the networks was extended by Jiang *et al.,* (2014). With recent technology such as tunneling, overlay network and virtualized network, it is becoming increasingly difficult to find an appropriate path for the application, as a shortest path is not necessarily always the best path. The extended Dijkstra's algorithm can be applied to derive a pair of shortest path in an SDN topology. However, most load balancing approaches in the context of SDN allocate resources based on statically configured routes and therefore may experience uneven load balancing. In this paper, the Extended Dijkstra's algorithm for SDN is modified by utilizing REST API of the controller and introduces a congestion control component to handle traffic overhead in an SDN topology. The remainder of this paper is organized as follows. Section II, introduces SDN, OpenFlow Switches, REST APIs, OpenDaylight controller, Mininet and challenges with the extended Dijkstra's Algorithm for SDN. Section III describes the proposed algorithm. Section IV shows the simulation results and observations. Finally, this paper is concluded with Section V.

2. PRELIMINARIES
2.1 Software Defined Networking
SDN is a new paradigm that breaks the vertical integration between the network control plane and its data plane [2]. The core idea of SDN is to decouple network control from data transmission. OpenFlow switches implement data transmission function, so as to simplify the design of switches, and control functions are provided by controllers. The switches implement data transmission function according to flow tables allocated from controller [2]. Being the brain of SDN, controller acquires application information from upper layer through the unified northbound interface. Flow tables are generated in controller and allocated to OpenFlow switches through OpenFlow protocol.

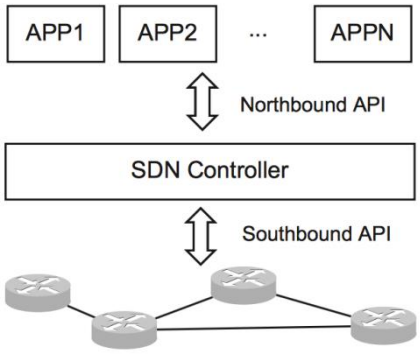

Figure 1. Software-Defined Network Architecture [2]

By acquiring network topology information, SDN controller provides the global network view for OpenFlow switches and implements the flexible network configuration and network management. SDN has gained a lot of attention in recent years, because it addresses the lack of programmability in existing networking architectures and faster network innovation [10].

2.2 OpenFlow Switches

OpenFlow switches are like standard hardware switches with a flow table performing packet lookup and forwarding. The difference between OpenFlow switches and standard switches, lies in how the flows rules are inserted and updated inside the switch's flow table [12]. A standard switch can have static rules inserted into the switch or can be a learning switch where the switch inserts rules into its flow table as it learns on which interface (switch port) a machine is. The OpenFlow switch on the other hand uses an external controller to add rules into its flow table.

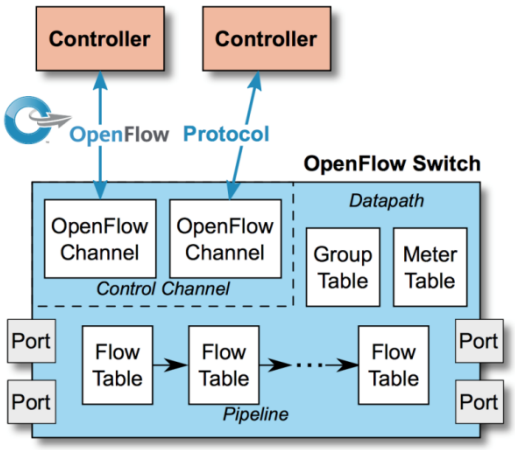

Figure 2. Components of an OpenFlow Switch [5]

2.3 OpenDaylight

OpenDaylight [14] is an open source project with a modular, pluggable, and flexible controller platform at its core. The core of the OpenDaylight platform is the Model-Driven Service Abstraction Layer (MD-SAL). The OpenDaylight controller can execute modules that describe how a new flow should be handled [16]. This provides us an interface to write python modules that dynamically add or delete routing rules into the switch and can use different policies for handling flows. Figure 3 presents the components of the OpenDaylight controller.

Mininet

Mininet [10] is an open source network emulator that supports the OpenFlow protocol for the SDN architecture. It is one of the most popular tools used by the SDN research community. It uses the virtualization approach to create a network of virtual hosts, switches, controllers, and links. Mininet hosts run standard Linux network software, and it supports the OpenFlow protocol for highly flexible custom routing and Software-Defined Networking. Just as an operating system (OS) virtualizes computing resources with process abstraction, Mininet uses process-based virtualization to emulate entities on a single OS kernel by running real code, including standard network applications, the real OS kernel and the network stack. Therefore, a design that works properly in Mininet can usually move directly to practical networks composed of real hardware devices.

2.4 REST APIs in the context of SDN

REST is an architecture style for designing networked applications. As REST architectural style has gained more popularity in implementing loosely-coupled systems, RESTful services are becoming the style of choice for northbound API and gaining increasingly importance in SDN architecture. Today's controllers utilize the REST API technology, which is an effective mechanism to communicate with various components in an SDN network [9]. The REST API uses HTTP messages to send and receive information between the SDN controller and another application. When the SDN controller receives the HTTP GET request, it will reply with an HTTP GET response with the information that was requested.

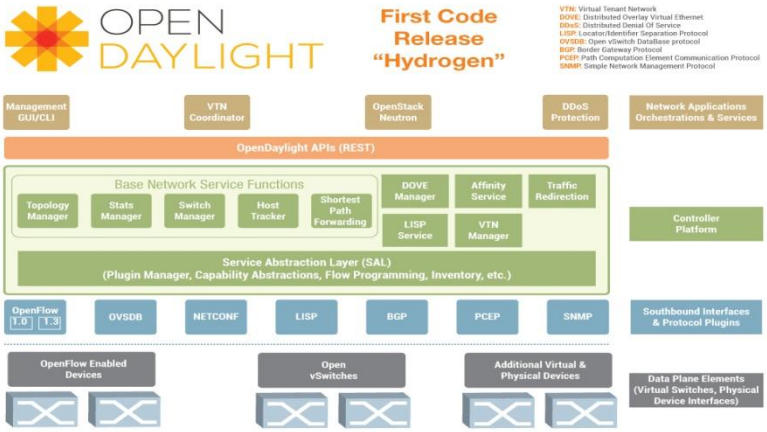

Figure 3: OpenDaylight Architecture [16]

2.6 Extended Dijkstra's Algorithm for SDN

Given a weighted, directed graph G = (V, E) and a single source node *s*, the classical Dijkstra's algorithm can return a shortest path from the source node *s* to every other node, where V is a set of nodes and E is the set of edges, each of which is associated with no weight. In the original Dijkstra algorithm nodes are associated with no weight. However, ED-SDN returns the shortest path from the single source node to every other with the consideration of the edge weight and the node weight [6].

Extended Dijkstra's Algorithm

Input: $G=(V, E)$, ew, nw, s

Output: $d[|V|]$, $p[|V|]$

1: $d[s] \leftarrow 0$; $d[u] \leftarrow \infty$, for each $u \neq s$, $u \in V$
2: **insert** u with key $d[u]$ into the priority queue Q, for each $u \in V$
3: **while** $(Q \neq \emptyset)$
4: $u \leftarrow$ Extract-Min(Q)
5: **for** each v adjacent to u
6: **if** $d[v] > d[u]+ew[u,v]+nw[u]$ **then**
7: $d[v] \leftarrow d[u]+ew[u,v]+nw[u]$
8: $p[v] \leftarrow d[u]$

Figure 4: Extended Dijkstra's Algorithm [6]

3 IMPROVED EXTENDED DIJKSTRA ALGORITHM (mED-SDN)

This sections provides an overview of the improved Extended Dijkstra's algorithm for SDN.

3.1. Description of Improved Extended Dijkstra's Algorithm for SDN

The main component of the Improved Extended Dijkstra's Algorithm for SDN (mED-SDN) is REST. To make REST API calls to the controller, the application has to be authenticated against the controller.

```
curl –sk –H "X-Auth-Token:$token" –X POST –H
'Content-Type:application/json
$token –d
https://192.168.56.1/sdn/v2.0/of/datapats/00:00:00:0
0:00:0e/flows
echo "Finished"
printf "\n\n"
```

Figure 5: Script to Authenticate with the Controller

Figure 5 shows a section of the python script that uses HTTP POST to fetch the URL through the API and retrieve some of the variables that are available, for example, information about all nodes (hosts) on the network. Once the API receives this, it will respond with an HTTP GET response message. As shown in Figure 6, the HTTP traffic from host with an IP address *10.0.0.1* to destination of *10.0.0.12* will be sent out of port 1 on the switch. The flow priority is set to 3200, however the default priority is 2990 on the controller. The script puts the flow priority higher than the default, this is to ensure that the traffic will use the path as set by this script rather than the default from the controller. The idle timeout for flow entries were set to sixty (60) seconds. This can be set to a higher value however. The idle timeout signifies how long the flow entry will stay in the switch when there is no traffic matching the flow entry.

```
payload0b='{
     "flow": {
          "priority": 3000,
          "idle_timeout": 60,
          "match": [
               {"eth_type": "ipv4"},
               {"ipv4_src": "10.0.0.1"},
               {"ipv4_dst": "10.0.0.12"},
               {"ip_proto": "tcp"},
               {"tcp_dst": "80"},
     ],
     "actions": [("output":1)]
       }
}
```

Figure 6: Updating an OpenFlow Switch Table

3.2 Congestion Prevention Mechanism

This sub-section explains the congestion prevention mechanism which is a component of the load balancer. The congestion control component of this algorithm uses the bandwidth utilization as its evaluation criterion for improving congestion in an SDN topology. When the bandwidth utilization exceeds the threshold value set by the algorithm, it reverts to the controller to search for a new path. In other to obtain the link bandwidth utilization, the congestion congestion component proactively measures the bandwidth in the topology and utilizes the REST API on the controller to collect cumulative transmitted bytes at corresponding OpenFlow switches port.

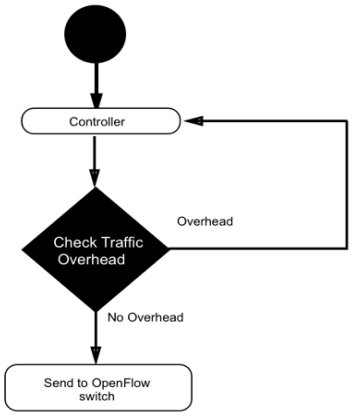

Figure 7: Congestion Control Component

4 EVALUATION

4.1 Simulation Settings

This research adopted the Abilene network topology and utilizes the Mininet network emulator to perform simulations. The Abilene network is a high-performance backbone network suggested by the Internet2 project. Figure 8 shows a historical Abilene (network) core topology emulated in the Mininet network emulator, connecting 11 regional sites or nodes across the United States. The Abilene network has 10 Gbps connectivity between neighboring nodes and 100 Mbps connectivity between a host and a node. Based on the Abilene core topology, a Mininet topology was set up consisting of an OpenDaylight SDN controller and 11 switches as nodes, where each switch is linked to the controller logically and is attached with at least one host. The simulation parameter settings are shown in Table I.

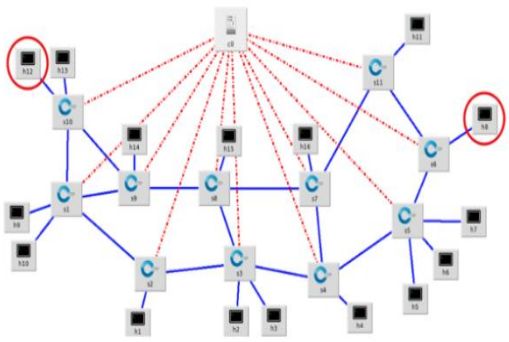

Figure 8: Setup of Abilene Topology in Mininet

Table 1: Simulation Settings

Parameter	Setting
Bandwidth on Edges	100Mbps ~ 1Gps
Capacity of nodes	3Gbps ~ 7Gbps
Number of switches	12
Number of edges	25
Controller	OpenDaylight controller
Testing tool	iPerf
Testing time	30 secs

4.2 Throughput Test

The Iperf network testing utility was used to study the throughput utilization on the Abilene network topology. As depicted in Figure 9 and Table 2, when the number of nodes are relatively light, all three approaches perform similarly. However, the TCP control mechanisms such as flow control, congestion control and error control mechanisms limit the throughput. The results suggest that the proposed approach shows a better performance in comparison to the other two. The average throughput recorded for the 12-node scenario for DA, ED-SDN, mED-SDN was 564, 589 and 610 Mbps respectively.

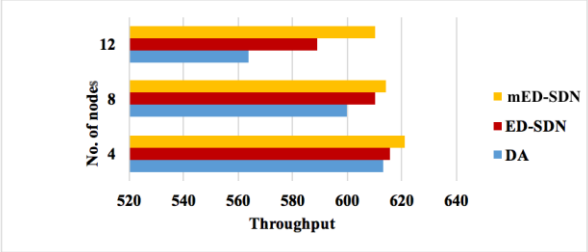

Figure 9: Throughput Test

Table 2. Throughput test results

Nodes	mED-SDN	ED-SDN	DA
4	620 Mbps	615 Mbps	612 Mbps
8	614 Mbps	610 Mbps	599 Mbps
12	610 Mbps	589 Mbps	564 Mbps

4.3 Throughput Test on Larger Abilene Network

To test for the effectiveness of the algorithm and further study the impact of the number of nodes on the performance of DA, ED-SDN and mED-SDN under the Abilene network topology, the number of nodes were increased by four (6) more nodes. The Figure 10 illustrates the output from Iperf on the new network scenario. Throughput tests were carried out for the three approaches. An arbitrary node was selected as the server to carry out the throughput tests using Iperf.

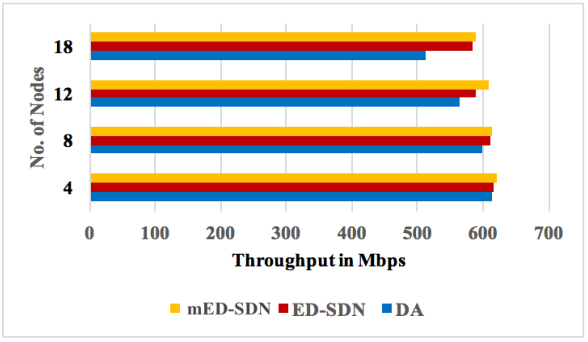

Figure 10: Throughput Test on Larger Abilene Topology

Under the new traffic conditions, DA experiences the most throughput set back, with an average of 512 Mbps for the 18-node scenario, ED-SDN and mED-SDN however performed similarly with a throughput of 585.5 Mbps and 590.4 Mbps respectively. In a typical network however, it is very unlikely that all the switches are highly active at the same time, but protocols such as the CDP (control datagram packet) and OpenFlow messages exchanged between the switches and the controller transmitted across the links can reduce the network throughput. Furthermore, DA experienced significant throughput set back also due to the fact that it only considers distance as the single factor for determining the best path, as the number of nodes increase, the number of hops to reach the destination also increases, therefore a packet has to transverse through more network hops and subsequently resulting to drop in the transmission rate.

4.4 Latency Test on Abilene Network Topology

The bandwidth of an edge and the capacity of a node were set randomly to be within the range shown in Table I. The simulation results of the latency tests are shown in Figure 11. By the simulation results, we can notice that mED- SDN has less end-to-end latency than the original ED-SDN, DA experiences significant degradation of latency when the nodes increase, partly due to the increase in the number of hops a packet travels from the server node. The average latency experienced on the 12-node Abilene topology for DA, ED-SDN and mED-SDN were recorded at 7.5, 6.8 and 5.4 milliseconds respectively.

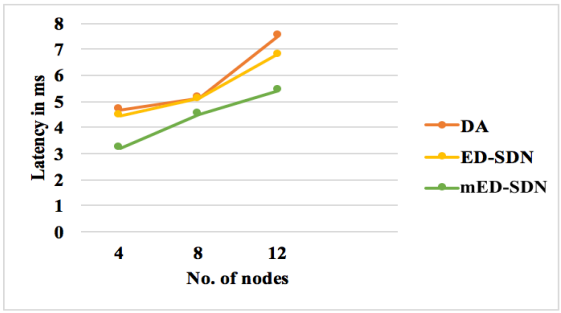

Figure 11: Latency Test

Table 3. Latency test results

Nodes	mED-SDN	ED-SDN	DA
4	5.4 ms	6.7 ms	7.5 ms
8	4.5 ms	5.1 ms	5.2 ms
12	3.2 ms	4.4 ms	4.6 ms

4.5 Comparison with Round Robin Approach

Further comparison tests were conducted with a common load balancing method which is the round robin approach for SDN. As seen in Figure 12., the round robin approach has a significant low throughput around 50% less than the proposed algorithm. Furthermore, the round robin approach may deflect a request to a farther server which would cause significant latency. Therefore, if the requests are deflected to the farther servers through which packets transverse switches and routers, the throughput would reduce therefore causing the latency to increase significantly. The proposed algorithm is superior with respect to network throughput. mED-SDN attained an average of 612Mbps under 12-node Abilene network topology while round robin was at 315Mbps.

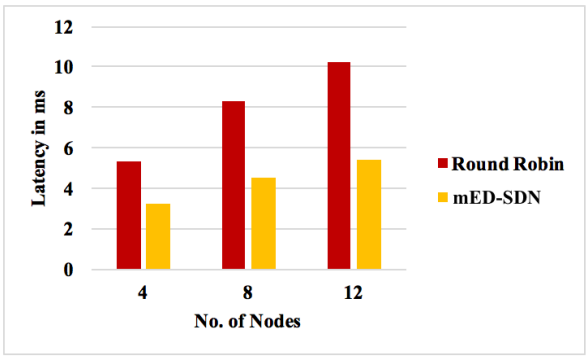

Figure 12: Comparison with Round Robin

5 CONCLUSION

This research is aimed at the development of an efficient load balancer in the context of SDN, by modifying the existing Extended Dijkstra's algorithm for SDN using a northbound plug-in that performs REST requests to update flow tables in switches and a function that handles control congestion in the SDN topology. The mED-SDN is a python-based script, which is imported as a component into the controller. Using Iperf, throughput and latency tests were carried out on the Abilene network topology. To further test for the effectiveness of the mED-SDN, this research performed comparison tests with the Round Robin approach and the traditional Dijkstra's algorithm used by many controllers and achieved better performance in terms of throughput and latency.

6 REFERENCES

[1] B. Wolfgang and M. Michael. 2014. Software-defined Networking: Using OpenFlow: Protocols, Applications and Architectural Design Choices. *IEEE 13th International Conference on Trust, Security and Privacy in Computing Communications, 1-6.*

[2] Dijkstra, E.W. 1959. A note on two problems in connexion with graphs, *Numerishe mathematik 1* (1), 269-271.

[3] G. Deep and J. Hong. 2016. Round Robin Load Balancer using Software Defined Networking (SDN). *Capstone Team Research Project, Vol. 5, 1-9, 2016.*

[4] G. Senthil, and S. Rajani. 2015. Dynamic Load Balancing using Software Defined Networks. *International Journal of Electrical and Computer Engineering (IJECE), Vol. 3, 203-256*

[5] H. Tim. 2011. Path computation enhancement in Software Defined Networks. *ChenDu College of University of Electronic Science and Technology.*

[6] J. Jiang, H. Hsin-Wen, and C. Szu-Yuan. 2013. Extending Dijkstra's shortest path algorithm for Software Defined Networks. *Department of Computer Science and Information Engineering National Central University.*

[7] K. Diego and M. Fernando. 2014. Software-Defined Networking: *A Comprehensive Survey. International Conference on Advanced Information Networking and Applications, Vol. 103(1), 1-63.*

[8] L.Hui, S.Yao, and T.Fin. 2013. Dynamic Load Balance Routing in OpenFlow Enabled Networks. *IEEE International Conference on Advanced Information Networking and Applications, 290-297.*

[9] Michael, D. 2011. The REST API Lifecycle: From Planning to Production. *International Journal for Advances in Engineering Research, Vol. 2, 333-339.*

[10] Mininet Website, *http://mininet.org/, last accessed on May 2016.*

[11] N. Handigol, M. Heller and M. Nick. 2013. Plug-n-Serve: Load-Balancing Web Traffic using OpenFlow. *Networking Report, University of Washington, Vol. 7.*

[12] N. Mckeown, T. Anderson. 2008. OpenFlow enabling innovation on campus networks. *ACM SIGCOMM Computer Communication.*

[13] Open Networking Foundation., *OpenFlow Switch Specification version 1.4.0. Accessed on October 14, 2016*

[14] OpenDaylight Developer Guide., *OpenDaylight community. Accessed on October 14, 2015*

[15] Y. Widhi, J. Jehn-Ruey, & B. Achmad. 2015. The Extended Dijkstra-based Load Balancing for OpenFlow Network. *International Journal of Electrical and Computer Engineering (IJECE), Vol. 5(2)., 289-296*

[16] Z. Khan, M.and Awais. 2014. Performance Evaluation of OpenDaylight SDN controller. *Department of Computer Science, Namal College, Pakistan..*