Multicasting with the Extended Dijkstra's Shortest Path Algorithm for Software Defined Networking

Mahardeka Tri Ananta[1, 2], Jehn-Ruey Jiang[1], and Muhammad Aziz Muslim[2]

[1]Department of Computer Science and Information Engineering
National Central University
Jhongli City, Taiwan
[2]Department of Electrical Engineering
University of Brawijaya
Malang City, Indonesia

***Abstract*—This work proposes a multicast algorithm on the basis of the extended Dijkstra's shortest path algorithm for Software Defined Networking (SDN) to run on top of the controller. The proposed muticast algorithm is used to generate a multicast tree for a data publisher to deliver data packets to all subscribers so that every node and every host on the multicast tree will receive every packet once and at most once for reducing bandwidth consumption. The extended Dijkstra's algorithm considers not only the edge weights, but also the node weights for a graph derived from the underlying SDN topology. We use Pyretic to implement a proposed algorithm over an SDN network, and compare it with related ones under the Abilene network topology with the Mininet emulation tool. As shown by the comparisons, the proposed algorithm achieves the best performance in terms of throughput, jitter, and packet loss.**

Keywords—Software Defined Networking (SDN); Multicast; network topology

I. INTRODUCTION

Software Defined Networking (SDN) is a hot topic in network research based on the concepts of control plane and data (forwarding) plane separation [1]. McKeown et al. proposed the OpenFlow protocol as a means to realize the SDN concept [1]. A logically centralized controller configures the forwarding tables (also called flow tables) of switches, which are responsible for forwarding the packets of communication flows. In this way, SDN users can composite application programs to run on top of the controller to monitor and manage the whole network. SDN offers several benefits such as ease of implementation and administration, no distributed states, a global network view, centrally at the control plane, no need to configure each forwarding plane device manually, and simple forwarding plane device configuration [2] [3].

The emergence of the SDN technology makes possible many new network applications realized by directly programming the SDN controller. One typical example of such applications is multicast. Some researchers developed SDN programming languages, such as Frenetic [4] and Pyretic [5], to facilitate SDN application implementation. Frenetic is a declarative query language for classifying and aggregating network traffic as well as a functional reactive combinator library for describing high-level packet-forwarding policies [4]. Pyretic is one member of the Frenetic family of SDN programming languages [5] and embedded in Python and the runtime system that implements programs written in the pyretic language on network switches. Pyretic can enable network programmers and operators to write succinct modular network applications by providing powerful abstractions.

Jehn-Ruey Jiang et al. [6] extended the well-known Dijkstra's shortest path algorithm [7] to consider not only the edge weights, but also the node weights for a graph derived from the underlying SDN topology. As shown by the simulation results in [6], the extended Dijkstra's algorithm outperforms the Dijkstra's algorithm and the non-weighted Dijkstra's algorithm under the Abilene network [8] in terms of end-to-end latency. This is because the extended Dijkstra's algorithm takes edge weights as transmission delays over edges and takes node weights as process delays over nodes, while the other two algorithms consider only edge weights or no weights.

Based on the extended Dijkstra's algorithm, this work proposes a multicast algorithm for SDN-based wide area networks. We use Pyretic to implement the proposed algorithms and compare it with related basic algorithms, i.e., the bread-first search tree multicast algorithm and the original Dijkstra's shortest path tree multicast algorithm, under the Abilene network topology with the Mininet emulation tool [9]. As shown by the comparisons, the proposed algorithms outperform the others.

The remainder of this work is organized as follows. In the next section, we present preliminaries on the subject of this work, i.e., Software Defined Networking, Multicast, Pyretic, and Mininet. In Section III, we describes the extended Dijkstra's algorithm and its implementation. Section IV presents and discusses the simulation settings and the results of our simulation experiments. Finally, we give our conclusion in Section V.

II. PRELIMINARIES

A. *Software Defined Networking*

Software-Defined Networking (SDN) is a new approach to networking and emerging architecture that is dynamic, manageable, cost-effective, and adaptable, making it ideal for the high-bandwidth, dynamic nature of today's applications. This architecture decouples the network control and forwarding functions enabling the network control to become directly programmable and the underlying infrastructure to be abstracted for applications and network services. The OpenFlow™ protocol is a foundational element for building SDN solutions [10]. With SDN, a researcher, network administrator, or third party can introduce a new capability by writing a software program that simply manipulates the logical map of a slice of the network also dictates the overall network behavior by using a controller [11] [12].

SDN encourages the separation of control and data planes, where underlying switching hardware is controlled via software that runs in an external, decoupled automated control plane [13]. So in short, SDN separates the network control (Learning, routing and forwarding packets) from Network topology (Routers, switches, Hubs, etc.) [14]. So basically, SDN architecture consist of three functional layers: the data plane, the control plane, and the applications. In addition, it also contains a set of APIs that enable network administrator easily manage network services, including routing, access list, multicast, and other traffic engineering to meet the business goal [15]. Fig.1 depicts a logical view of the SDN architecture.

The communications between the control and forwarding layers of an SDN architecture are conducted by OpenFlow protocol. OpenFlow provides an open protocol to program the flow table in different switches and routers. A network administrator can partition traffic into production and research flows. Researchers can control their own flows - by choosing the routes their packets follow and the processing they receive. In this way, researchers can try new routing protocols, security models, addressing schemes, and even alternatives to IP. On the same network, the production traffic is isolated and processed in the same way as today [1].

OpenFlow provide an open, programmable, virtualized platform on their switches and routers so that researchers can deploy new protocols, while network administrators can take comfort that the equipment is well supported. 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 (in response to packets) 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. 3 [16].

Basically matching starts at the first flow table and may continue to additional flow tables. We can see Fig. 4, flow entries match packets in priority order, with the first matching entry in each table being used. If a matching entry is found, the instructions associated with the specific flow entry are executed. If no match is found in a flow table, the outcome depends on configuration of the table-miss flow entry: for example, the packet may be forwarded to the controller over the OpenFlow channel, dropped, or may continue to the next flow table [16].

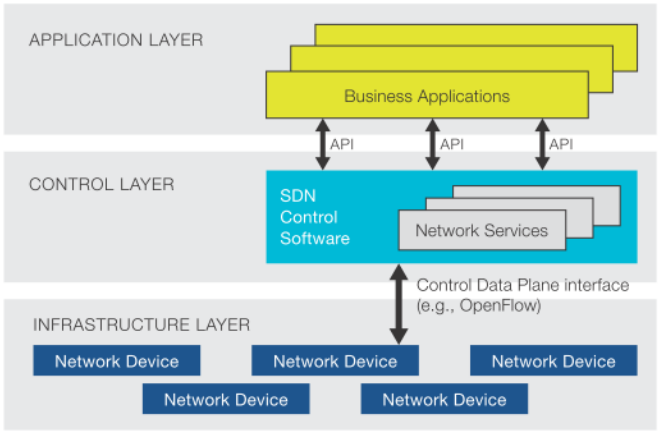

Fig. 1. The illustration of the SDN architecture [16]

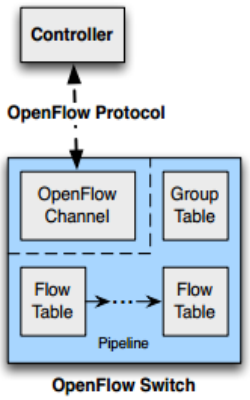

Fig. 2. The OpenFlow controller and the switch [16]

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie

Fig. 3. The flow table entry of the OpenFlow switch [16]

B. *Pyretic*

N. Foster et al. [4] introduced high-level for SDN network. Frenetic provide a domain specific sub-language for specifying the dataplane packet processing in terms of packet functions and combinators inside of a general purpose programming language for describing high-level packet-forwarding policies. Modularity is the important key for managing complexity in any software system, and SDNs are no exception. Joshua Reich et.al [17], introduced Pyretic as a programming platform that raise the level of abstraction and enable to create modular software. Pyretic is one member of the Frenetic family of SDN programming languages (Python + Frenetic = Pyretic) that is

extended from Frenetic [4]. It has two policy composition operators, parallel composition and sequential composition, to allow programmers to combine multiple policies together.

As such Pyretic enables network programmers and operators to write shorter modular network applications by providing powerful abstractions. Pyretic is both a programmer-friendly domain-specific language embedded in Python and the runtime system that implements programs written in the Pyretic language on network switches [5].

C. The Extended Dijkstra's Shortest Path Algorithm

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 the set of nodes and E is the set of edges, each of which is associated with a non-negative *weight* (or *length*). In the original Dijkstra's algorithm, nodes are associated with no weight. The paper [6] shows how to extend the original algorithm to consider both the edge weights and the node weights.

Fig. 4 shows the extended Dijkstra's algorithm, whose input is a given graph $G=(V, E)$, the edge weight setting ew, the node weight setting nw, and the single source node s. The extended algorithm uses $d[u]$ to store the *distance* of the current shortest path from the source node s to the destination node u, and uses $p[u]$ to store the *previous* node preceding u on the current shortest path. Initially, $d[s]=0$, $d[u]=\infty$ for $u\in V$, $u\neq s$, and $p[u]$=null for $u\in V$.

Extended Dijkstra's Algorithm

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

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

```
1: d[s]←0; d[u]←∞, for each u≠s, u∈V
2: insert u with key d[u] into the priority queue Q, for each u∈V
3: while (Q≠∅)
4:   u←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]←d[u]+ew[u,v]+nw[u]
8:       p[v]←d[u]
```

Fig. 4. The extended Dijkstra's algorithm [6]

Note that the extended Dijkstra's algorithm is similar to the original Dijkstra's algorithm. The difference is that the extended version adds the node weight in line 6 and line 7 of the algorithm. The original Dijkstra's algorithm cannot achieve the same result just by adding node weights into edge weights. This is because the node weight should be considered only at the outgoing edge of an intermediate node on the path. Adding node weights into edge weights implies that an extra node weight of the destination node is added into the total weight of every shortest path, making the algorithm return the wrong result.

The extended Dijkstra's algorithm is very useful in deriving the best routing path to send a packet from a specific source node to another node (i.e., the destination node) for the SDN environment in which significant latency occurs when the packet goes through intermediate nodes and edges (or links). Below, we show how to define the edge weights and node weights so that the extended Dijkstra's algorithm can be applied to derive routing path for some specific SDN environment.

A. Rus et al. [18] has addressed the implementation issues for the modified Dijkstra's algorithm [19] and the modified Floyd-Warshall shortest path algorithm in OpenFlow. However, the modified Dijkstra's algorithm is different from the extended Dijkstra's algorithm proposed in [6] in the sense that the former is modified to solve the multi-source single-destination shortest path problem and the latter are extended from the Dijkstra's algorithm to consider both edge weights and node weights for solving the single-source shortest path problem. It is worth mentioning that the extension concept proposed in this work can also be applied to the modified Dijkstra's algorithm.

Assume that we can derive from the SDN topology a graph $G=(V, E)$, which is weighted, directed, and connected. For a node $v\in V$ and an edge $e\in E$, let $Flow(v)$ and $Flow(e)$ denote the set of all the flows passing through v and e, respectively, let $Capability(v)$ be the *capability* of v (i.e., the number of bits that v can process per second), and let $Bandwidth(e)$ be the bandwidth of e (i.e., the number of bits that e can transmit per second). The node weight $nw[v]$ of v is defined according to Eq. (1), and the edge weight $ew[e]$ of e is defined according to Eq. (2).

$$nw[v] = \frac{\sum_{f\in Flow(v)} Bits(f)}{Capacity(v)}, \quad (1)$$

where $Bits(f)$ stands for the number of f's bits processed by node v per second.

$$ew[e] = \frac{\sum_{f\in Flow(e)} Bits(f)}{Bandwidth(e)}, \quad (2)$$

where $Bits(f)$ stands for the number of f's bits passing through edge e per second.

Note that we can easily obtain the number of a flow's bits processed by a node or passing through an edge with the help of the "counters field" of the OpenFlow switches' flow tables. Also note that the numerators in Eq. (1) and Eq. (2) are of the unit of "bits", and the denominators are of the unit of "bits per second". Therefore, the node weight $nw[v]$ and the edge weight $ew[e]$ are of the unit of "second". When we accumulate all the node weights and all the edge weights along a path, we can obtain the end-to-end latency from one end to the other end of the path.

D. SDN-based Multicast

Recently, Aakash Iyer et al. [20] developed a new multicast algorithm, called Avalanche Routing Algorithm (AvRA), attempting to minimize the size of the routing tree created for each multicast group. Instead of trying to find the shortest path from a group member to the source node of the group, the AvRA tries to find the shortest path to the existing

multicast tree node. AvRA is designed for typical data center topologies like the FatTree structure. However, we will not compare the proposed algorithm with AvRA, because AvRA is designed for special topologies used in data centers, while the work focuses on general SDN-based wide area networks.

The multimedia data (e.g., video and audio data) have been a major source of data to be delivered by the multicast algorithm [21]. The growth and popularity of the Internet in the mid-1990's motivated multimedia data delivery over best-effort packet networks. Such multimedia data delivery is affected by a number of factors, including unknown and time-varying bandwidth, jitter, and losses. There raise issues such as how to fairly share the network resources among many flows and how to efficiently perform one-to-many communication for popular content [21]. Thanks to the SDN technology, the issues can be efficiently solved.

E. Mininet

Mininet is either a network simulation tool or network emulation tool that runs a collection of end-hosts, switches, routers, and links on a single Linux kernel. It uses lightweight virtualization to make a single system look like a complete network, running the same kernel, system, and user code. Mininet hosts run standard Linux network software, and its switches support OpenFlow for highly flexible custom routing and Software-Defined Networking [9]. By using Mininet We can emulates an arbitrary OpenFlow network on our machine.

Mininet also enable us to use client servers program such as ping and iperf. In this work, we use iperf to generate TCP and UDP packets from clients to servers. Iperf is a tool for measuring throughput, reminiscent of ttcp and nettest. It allows the tuning of various parameters and UDP characteristics. Iperf reports throughput, delay jitter, packet loss. The Iperf code is also designed to compile easily on any POSIX compliant platform. Iperf can run as a server mode or client mode and also specify the durations of testing [22].

III. The Proposed Multicast Algorithms

The proposed multicast algorithm is based on the multicast tree construction algorithm using the extended Dijkstra's algorithm for a multicast group publisher p to send data packets to all members in the multicast group MG of subscribers. The multicast tree construction algorithm for the proposed multicast algorithm is called the EDSPT (Extended Dijkstra's Shortest Path Tree) algorithm, as shown in Fig. 5. We just add an array $pred[i]$ to keep track the predecessor of every node i so that we can construct a tree T rooted at p to span all nodes, in term deriving the subtree MG of T associated with MG to make all subscribers in the multicast group MG reachable from the publisher p.

Extended Dijkstra's Shortest Path Tree Algorithm

Input: $G = (V, E)$, ew, nw, p, MG
Output: MT

1: $T=\{p\}$;$d[p]\leftarrow 0$; $d[u]\leftarrow\infty$ and $pred[i]\leftarrow$nil for each $u\neq p$, $u\in V$
2: **insert** u with key $d[u]$ into the priority queue Q, for each $u\in V$
while ($Q\neq\varnothing$)
3: $j \leftarrow$ Extract-Min(Q)
4: **for** every node i, $i\notin T$ and i is adjacent to j
5: $alt = d[j] + ew(j, i) + nw(j)$
6: **if** alt < $d[i]$ **then**
7: $d[i] \leftarrow$ alt
8: $pred[i] \leftarrow j$ // set i as a child node of j
9: add i into T
10: **return** MT, the subtree of T rooted at p associated with MG

Fig. 5. The extended Dijkstra's shortest path tree (EDSPT) algorithm

IV. Simulation for the Proposed Multicast Algorithm

We set up one POX OpenFlow controller and 11 OpenFlow switches as nodes based on the Abilene topology in the Mininet simulator, each of switch is linked to the controller logically. The Abilene network [8] is a high-performance backbone network suggested by the Internet2 project. Fig. 6 shows a historical Abilene (network) core topology [23], 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.

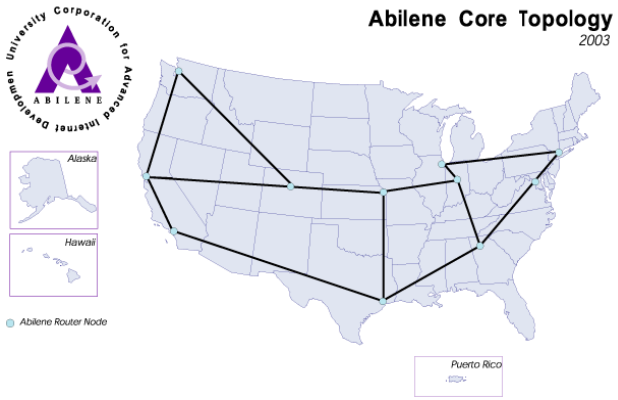

Fig. 6. The Abilene network core topology [23]

We simulate the multicast algorithms based on the based on the multicast tree construction algorithms using the breadth first search algorithm, the original Dijkstra's algorithm, and the extended Dijkstra's algorithm, respectively. Those multicast tree construction algorithms are called BFST, DSPT, and EDSPT algorithms. We assume 1 publisher as the source node (host5) located at switch 5, and 12 subscribers located in different areas of the Abilene network topology shown in Fig. 7. The bandwidth of the edges (links) were set randomly within the range from 100Mbps to 1Gbps, and the capability of nodes were set randomly from 3Gbps to 7Gbps. However the BFST algorithm considered all edge weights as 1.

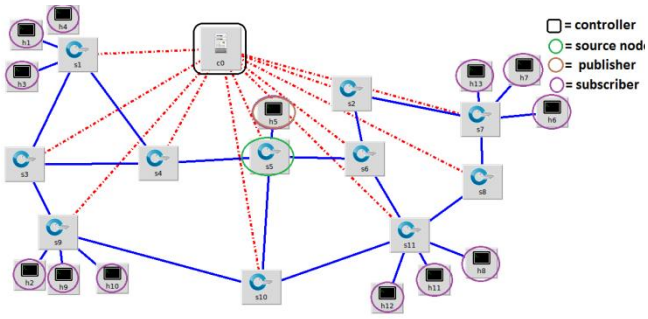

Fig. 7. Topology Setting

Table I describes the details of our simulation settings. We used POX as the OpenFlow controller and implemented the multicast tree algorithms using Pyretic. We ran this simulation on a PC with Pentium(R) Dual-Core CPU E5400@2.70GHz and 4GB of RAM.

TABLE I. SIMULATION SETTINGS

Parameter	Setting
Number of controller	1
Number of switches	11
Number of publishers	1
Number of subscribers	12
Number of edges	25
Controller	POX 2.0 supporting Pyretic
OpenFlow switch	Openvswitch 1.0
Testing tool	Iperf
Testing time per case	30 sec

In the simulation experiments, we measure the following network performance metrics, namely, throughput, jitter, and packet loss, for the multicast algorithms using BFST, DSPT, and EDSPT. We used Iperf to create the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) data stream packets. The experiment time for every test case was 30 seconds.

Fig. 8 shows the throughput for different numbers of multicast group subscribers, and Fig. 9 shows the average throughput. We can see that the multicast algorithm using EDSPT (i.e., the proposed algorithm) outperforms the algorithms using BFST and DSPT.

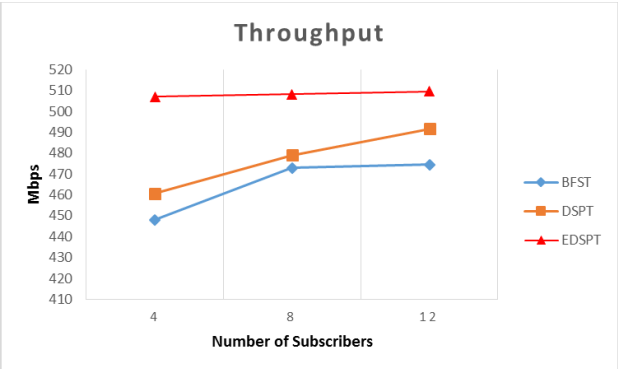

Fig. 8. The throughput comparisons

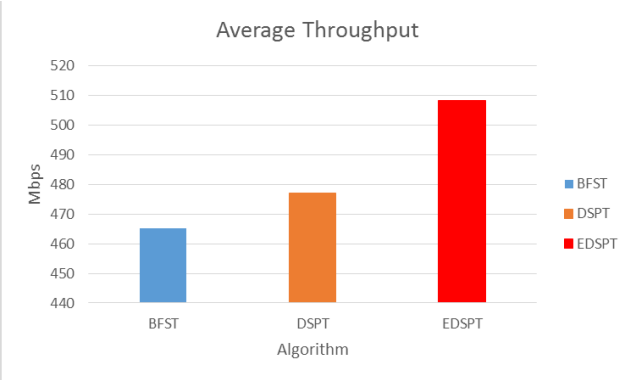

Fig. 9. The average throughput comparisons

We also conducted the jitter measurement. By using Iperf the publisher sends UDP packets to the subscribers for 30 seconds. Fig.10 shows the jitter for different numbers of subscribers, and Fig. 11 shows the average jitter. We can see that the multicast algorithm using EDSPT outperforms the algorithms using BFST and DSPT.

Fig. 10. The jitter comparisons

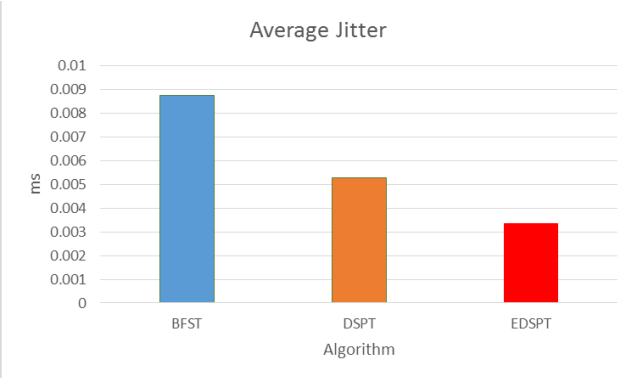

Fig. 11. Average Jitter

We also measured packet loss to verify the performance of several multicast algorithms. The measurement is based on the Iperf tool generating UDP packets for the publisher to send to subscribers for 30 seconds. Fig.12 shows the packet loss rates for different numbers of subscribers, and Fig. 13 shows the average packet loss rates. By the simulation results, we can see that the multicast algorithm using EDSPT is more suitable for dense networks and yields the highest throughput, the

smallest jitter and the packet loss rate. This is because the EDSPT algorithm considers both the edge weights and node weights, and the DSPT algorithm only considers the edge weights, and the BFST algorithm only considers the adjacent nodes to generate the multicast tree.

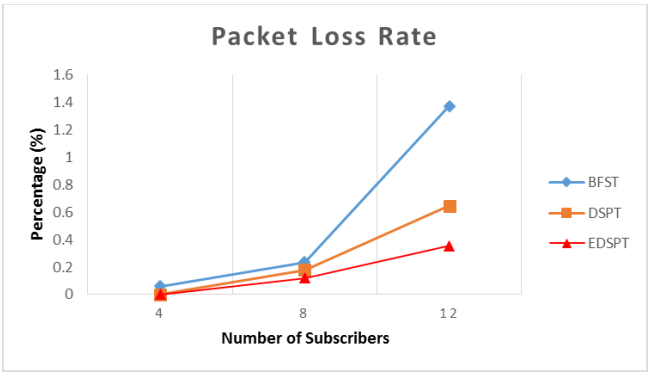

Fig. 12. The packet loss rate comparisons

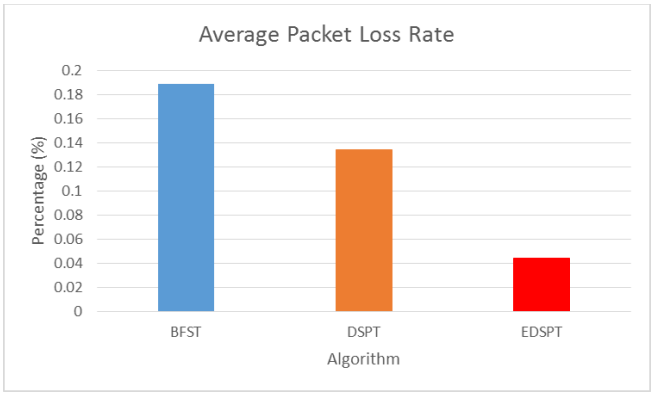

Fig. 13. The average packet loss rate comparisons

V. Conclusion

This work proposes a multicast algorithm on the basis of the extended Dijkstra's shortest path algorithm for SDN. The extended Dijkstra's algorithm considers not only the edge weights, but also the node weights for a graph derived from the underlying SDN topology. We use Pyretic to implement the multicast algorithms using BFST, DSPT, and EDSPT and compare them in terms of throughput, jitter, and packet loss under the Abilene network topology with the Mininet emulation tool. The simulation results show that the proposed multicast algorithm outperforms others.

References

[1] McKeown, Nick, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication*, 2008.

[2] Software-Defined Networking (SDN) Definition. Website https://www.opennetworking.org/sdn-resources/sdn-definition, last accessed on January 2014.

[3] Software-Defined Networking Research Project. http://www.ipvs.uni-stuttgart.de/abteilungen/vs/forschung/projekte/sdn, last accessed on March 2014.

[4] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. WalkerFoster, "Frenetic: A Network Programming Language", ACM, 2013.

[5] Python + Frenetic = Pyretic. http://frenetic-lang.org/pyretic/ , last accessed on March 2014.

[6] Jehn-Ruey Jiang, Hsin-Wen Huang, Ji-Hau Liao, and Szu-Yuan Chen, "Extending Dijkstra's Shortest Path Algorithm for Software Defined Networking," Technical Report, National Central University, 2014.

[7] E. . Dijkstra, "A note on two problems in connexion with graphs," Numerische mathematik, vol. 1, no.1, 1959, pp. 269-271.

[8] Abilene Network, http://en.wikipedia.org/wiki/Abilene_Network-#cite_note-line-1, last accessed on March 4, 2014.

[9] Mininet, An Instant Virtual Network on your Laptop (or other PC), http://mininet.org/, last accessed on June 2014.

[10] Software-Defined Networking (SDN) Definition. Website https://www.opennetworking.org/sdn-resources/sdn-definition, last accessed on January 2014.

[11] Kobayashi, Masayoshi, Et al., "Maturing of OpenFlow and Software-defined Networking through deployments," Science Direct Computer Networks, 2013.

[12] Hyojoon, Kim Feamster, N, "Improving Network Management With Software Defined Networking," Communications Magazine, IEEE, 2013.

[13] Saurav Das, Et al., "Handbook of Fiber Optic Data Communication a Practical Guide to Optical Networking Chapter 17, 4th edition".

[14] Sugam Agarwal, Murali Kodialam, T. V. Lakshman, "Traffic Engineering in Software Defined Networks," Proceedings IEEE INFOCOM, Bell Labs Alcatel-Lucent Holmdel, 2013.

[15] The Open Networking Foundation, "Software-Defined Networking: The New Norm for Networks," April 13, 2012.

[16] The Open Networking Foundation, "OpenFlow Switch Specification version 1.4.0," October 14, 2013.

[17] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, "Modular SDN Programming with Pyretic", Technical Reprot of USENIX, available at http://www.usenix.org, 2013.

[18] A. Rus, V. Dobrota, A. Vedinas, G. Boanea, and M. Barabas, "Modified Dijkstra's algorithm with cross-layer QoS," ACTA TECHNICA NAPOCENSIS, Electronics and Telecommunications, vol. 51, no. 3, 2010, pp. 75-80.

[19] A. Furculita, M. Ulinic, A. Rus, and V. Dobrota, "Implementation issues for Modified Dijkstra's and Floyd-Warshall algorithms in OpenFlow," in Proc. of 2013 RoEduNet International Conference 12th Edition: Networking in Education and Research, 2013, pp. 141-146

[20] A akash Iyer, Praveen Kumar, Vijay Mann, "Avalanche: Data center Multicast using Software Defined Networking", IEEE Communication Systems and Networks (COMSNETS), Sixth International Conference, 2014

[21] John G. Apostolopoulos, Wai-tian Tan, Susie J. Wee, "Video Streaming: Concepts, Algorithms, and, System," *Streaming Media Systems Group Hewlett-Packard Laboratories*, 2002.

[22] Iperf Website, http://iperf.fr/, last accessed on June 2014.

[23] Historical Abilene Connection Traffic Statistics, http://stryper.uits.iu.edu/abilene/, last accessed in March 2014.