Load Balancing and Multicasting Using the Extended Dijkstra's Algorithm in Software Defined Networking

Jehn-Ruey Jiang[1], Widhi Yahya[1,2], and Mahardeka Tri Ananta[1,2]

[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. The extended Dijkstra's algorithm considers not only the edge weights but also the node weights for finding shortest paths from a source node to all other nodes in a given graph. To show the advantage of the extended Dijkstra's algorithm, this paper proposes a load-balancing algorithm and a multicast algorithm in Software Defined Networking (SDN) on the basis of the extended Dijkstra's algorithm for a graph derived from the underlying SDN topology. We use Pyretic to implement the proposed algorithms and compare them with related ones under the Abilene network topology with the Mininet emulation tool. As shown by the comparisons, the proposed algorithms outperform basic algorithms.

Keywords: Software Defined Networking (SDN); load-balancing; multicasting; shortest path; Dijkstra's algorithm

1 Introduction

Software Defined Networking (SDN) is a concept to decouple the control plane and data plane of network devices [1]. McKeown et al. proposed the OpenFlow protocol to realize the SDN concept to allow researchers to experiment novel network protocols [2]. In SDN, 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 run on top of the controller to monitor and manage the whole network in a centralized and real-time manner.

The emergence of the SDN technology brings many new network applications realized by programming the SDN controller. Typical examples include load balancing, multimedia multicast, intrusion detection, and so on. Some researchers developed programming languages, such as Frenetic [3] and Pyretic [4], to facilitate SDN application design. Frenetic is a declarative query language for classifying network traffic and providing a functional reactive combinator library for describing high-level pack-

adfa, p. 1, 2011.

et-forwarding policies [3]. Pyretic is a Python-base language that is extended from Frenetic. Pyretic raises the level of network abstraction and enables programmers to create modular software for SDN [4].

The paper [5] extends the well-known Dijkstra's shortest path algorithm 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 [5], the extended Dijkstra's algorithm outperforms the Dijkstra's algorithm and the non-weighted Dijkstra's algorithm under the Abilene network [6] 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.

To show the advantage of the extended Dijkstra's algorithm, this paper proposes a load-balancing algorithm and a multicast algorithm using the extended Dijkstra's 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 round-robin load-balancing algorithm and the randomized load-balancing algorithm, 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 [7]. As shown by the comparisons, the proposed algorithms outperform the others.

The remainder of this paper is organized as follows. In Section 2, we introduce some related work. Section 3 describes the proposed algorithms, and Section 4 shows the simulation results. Finally, this paper is concluded with Section 5.

2 RELATED WORK

2.1 The Extended Dijkstra's 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 [5] shows how to extend the original algorithm to consider both the edge weights and the node weights.

Fig. 1 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 $u\in V$.

Note that 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.

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 *Capacity*(v) be the *capacity* 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).

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

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

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

where *Bits*(f) stands for the number of flow 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.

```
Extended Dijkstra's Algorithm
Input: $G$=($V$, $E$), $ew$, $nw$, $s$
Output: $d[|V|]$, $p[|V|]$

$d[s] \leftarrow 0$; $d[u] \leftarrow \infty$, for each $u \neq s$, $u \in V$
insert $u$ with key $d[u]$ into the priority queue $Q$, for each $u \in V$
while ($Q \neq \emptyset$)
  $u \leftarrow$ Extract-Min($Q$)
  for each $v$ adjacent to $u$
    if $d[v] > d[u]+ew[u,v]+nw[u]$ then
      $d[v] \leftarrow d[u]+ew[u,v]+nw[u]$
      $p[v] \leftarrow d[u]$
```

Fig. 1. The extended Dijkstra's algorithm [5]

2.2 SDN-based Load-Balancing

Load balancing is an important concept in networking. The purpose of the load balancing application is to distribute loads among multiple servers in order to get the best performance [8]. Online services such as e-commerce, e-government, web sites, and social networks are often use multiple servers to get reliability and high availability. Those systems use a load balancer in the front-end to map requests from the client to the servers. The use of hardware load balancer can be a solution, but it suffers from the expensive price. Furthermore, the hardware load balancer is too rigid because the policy was designed by the company. The emergence of SDN enables users to design their own software load balancer that is suitable for their system and also has a lower price.

Some load balancing methods using the SDN technology are recently proposed. The paper [8] proposed a load balancing algorithm, named LABERIO (LoAd-BalancEd Routing wIth OpenFlow), to minimize latency and response time and to maximize the network throughput by better utilizing available resources. The algorithm uses ToR (Top of Rack) Switch-to-ToR Switch Paths Table (S2SPT) and Load Allocation Table (LAT). However, maintaining S2SPT becomes a problem for the LABERIO, if the network topology changes. So, LABERIO is not suitable in wide area network because in the wide area we cannot predict the topology changes. The paper [9] proposed the Plug-n-Serve system implementing a load balancing algorithm, called LOBUS (LOad-Balancing over UnStructure networks), using OpenFlow for unstructured networks. LOBUS maintains the network topology and link status, and greedily choses the client-server pair that yields the lowest total response time for each newly arriving request. The paper [10] developed a load balancing algorithm for handling multiple services (called LBMS) by the SDN technology. It uses the FlowVisor, an SDN device to achieve network virtualization, to coordinate multiple controllers each of which handles requests destined for different services. In this paper, we will not compare the proposed load-balancing algorithm with LABERIO, LOBUS, or LBMS, since they are intended for scenarios different from that of the proposed algorithm.

2.3 SDN-based Multicast

Recently, Aakash Iyer et al. [11] 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 paper focuses on general SDN-based wide area networks.

The multimedia data (e.g., video and audio data) has been a major source of data to be delivered by the multicast algorithm [12]. 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 [12]. Thanks to the SDN technology, the issues can be efficiently solved by constructing a multicast tree by an application program run on the controller.

3 PROPOSED ALGORITHMS

3.1 The Proposed Load-Balancing Algorithm

In this paper we adopt the concept of virtual IP (VIP) for achieving load-balancing. The client just sends a request to the VIP, and the request will be deflected to one of the multiple servers. Using naive algorithms, such as the round robin algorithm and the randomized algorithm, in wide-area-network load balancing has the possibility to forward a request to the farthest server. This is not efficient because the request and the replied data will go across the network and consume a lot of bandwidth of the whole network. In SDN, the controller has the global information of the whole network, and can decide to forward the request to the nearest server by finding the shortest path with the extended Dijkstra's algorithm from the client to a server, where the shortest path means the path with the smallest summation of node weights and edge weights.

```
Proposed Load-Balancing Algorithm
Input: $sw_{src}$, $S_{dst}$, $\theta$
Output: $s$, $s \in S_{dst}$
$P \leftarrow eDijkstra(sw_{src}, S_{dst})$; $Q \leftarrow \varnothing$ //$P$ and $Q$ are path sets
for every $p_i \in P$
    if $p_i.server.ll > \theta$ then move $p_i$ from $P$ to $Q$
if $P \neq \varnothing$ then
    $s \leftarrow min(P).server$
else
    $s \leftarrow min(Q).server$
return $s$
```

Fig. 2. The proposed load balancing algorithm

Fig. 2 shows the proposed load balancing algorithm. The basic idea is to forward each request to the nearest server with the link load (utilization of the link between the server and the switch) lower than a pre-specified threshold θ. However, if all the servers have link loads larger than the threshold, the algorithm still choose the nearest server. In this way, we can prevent congestion on the servers.

We assume server s_i is attached to switch sw_i and a switch is attached with at most one server. Later on, we use s_i and sw_i exchangeable for convenience. Given the source switch sw_{src} to which the request client is attached, the set S_{dst} of servers, and a prespecified threshold θ, the proposed algorithm will return the best server for load-balancing.

The link load ll_i of serve s_i (the utilization of the link $<s_i, sw_i>$ between the server s_i and the switch sw_i) is defined as follows:

$$ll_i = \frac{current\ traffic\ of\ link\ < s_i, sw_i >}{maximum\ bandwidth\ of\ < s_i, sw_i >} \qquad (3)$$

Since the proposed algorithm is based on the extended Dijkstra's algorithm [6], we also take the same mechanisms to obtain the node weights and the edge weights. Note that eDijkstra(sw_{src}, S_{dst}) will use the extended Dijkstra's algorithm to return a set *P* of shortest paths from the source switch sw_{src}, to every server in the server set the S_{dst}. Also note that p_i*.server* stands for the server associated with the path p_i, and hence p_i*.server.ll* stands for the link load of the server associated with the path p_i. Furthermore, the function *min*(*P*) (resp., min(*Q*)) will return the shortest one among all shortest paths in *P* (resp., *Q*).

3.2 The proposed multicast algorithm

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. 3. 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: $MT$

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

Fig. 3. The extended Dijkstra's shortest path tree (EDSPT) algorithm

4 SIMULATION

4.1 Simulation for the Proposed Load-Balancing algorithm

According to the Abilene core topology, we set up an OpenFlow POX controller and 11 OpenFlow switches as SDN nodes, each of which was linked to the controller logically, as shown in Fig. 4. For the load balancing testing, we assumed two web servers are placed and spread in two different locations in the Abilene network which clients will send requests to. We implemented the proposed load-balancing algorithm and two basic algorithms using Pyretic for the purpose of simulation.

In the simulation testing, we generated Transmission Control Protocol (TCP) data stream from the clients to the servers using Iperf. To extend more information, we also used the Netperf to generate request from clients. We defined the request size to be 1024 bytes and the response size to be 65536 bytes. For every testing we set the number of clients to be 4, 8, and 12. This simulation was run on a PC with AMD Phenom(tm) 9650 Quad-Core Processor and 8GB of RAM. We compared the proposed algorithm with the round robin and randomized algorithms which do not consider the shortest paths.

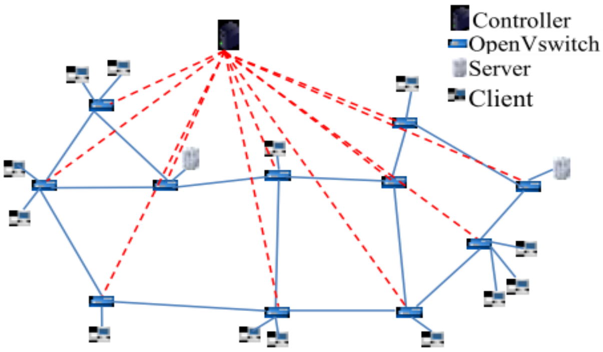

Fig. 4. The scenario of the Abilene network used in the simulation

The simulation results show that the proposed algorithm is better than the two naive algorithms in the term of end-to-end latency, as shown in Fig. 5. The end-to-end latency is measured by using the ping tool to send 30 packets whose packet size is 65507 bytes from clients to the servers for 30 seconds. The superiority is because the proposed algorithm considers the shortest paths and also the congestion control. On the contrary, it is possible for the naive algorithms to forward requests to the far server to go through some switches. In the real network, a high performance IP routers and switches add approximately 200 microseconds of latency to the link due to packet processing. It means, if the request is deflected to the farthest server to go through a lot of switches, the latency will increase significantly.

We also simulate the response time for each algorithm, where the response time is the time that is needed for a client sending a request to have the reply from the server. To get this response time we generated requests using the Netperf for 30 seconds. The request size is set to 1024 bytes and the response size is set to 512 kilobytes. As shown in Fig. 6, the proposed algorithm has the best average response time.

Throughput is the rate of successful message delivery over a communication channel. In our simulation, the throughput was measured by generating TCP data packets from the clients to the servers with the Iperf tool for 30 seconds. As shown in Fig. 7, the proposed algorithm has higher throughput than the round robin and the randomized algorithms. By considering the shortest paths and the link utilization, the proposed algorithm achieves the highest throughput. The round robin and the randomized algorithms may deflect requests to the far server and even may cause links to be congested. This is why the throughput of the two naïve algorithms is lower than that of the proposed algorithm.

The Iperf has the client and the server side program. In the server side program, we can get the value of server loads. This information was used to depict the server load variation by calculating the standard deviation of each server's loads. The standard deviation measures the amount of variation or dispersion from the average server load. As shown in Fig. 8, the proposed algorithm also has the best (smallest) standard server load deviation.

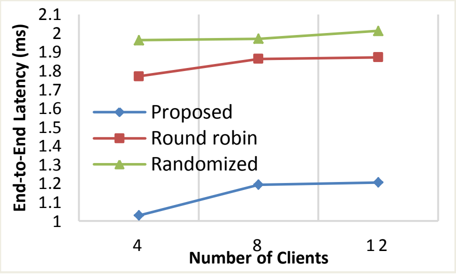

Fig. 5. The end-to-end latency comparisons

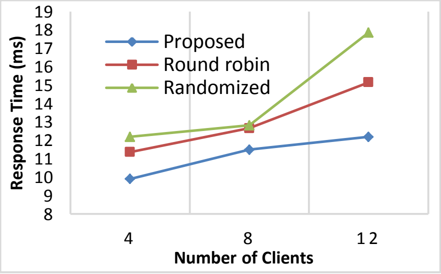

Fig. 6. The response time comparisons

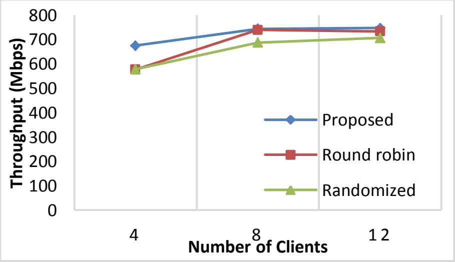

Fig. 7. The throughput comparisons

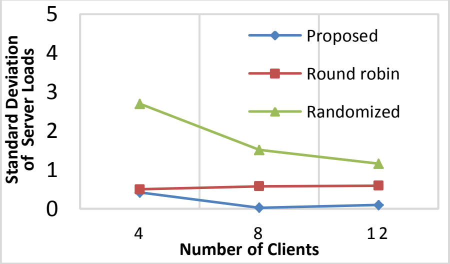

Fig. 8. Standard deviation comparisons

4.2 Simulation for the Proposed Multicast Algorithm

We simulate the multicast algorithms 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. 9. The bandwidth of the edges (links) was set randomly within the range from 100Mbps to 1Gbps, and the capability of nodes was set randomly from 3Gbps to 7Gbps. However the BFST algorithm considered all edge weights as 1.

We used POX as the OpenFlow controller and implemented the multicast tree algorithms using Pyretic. We ran to measure the following network performance metrics, namely, throughput and jitter, for the multicast algorithms using BFST, DSPT, and EDSPT. We used Iperf to create Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) data stream packets. The experiment time for every test case was 30 seconds.

Fig. 10 shows the throughput for different numbers of multicast group subscribers We can see that the multicast algorithm using EDSPT (i.e., the proposed algorithm) outperforms the algorithms using BFST and DSPT.

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

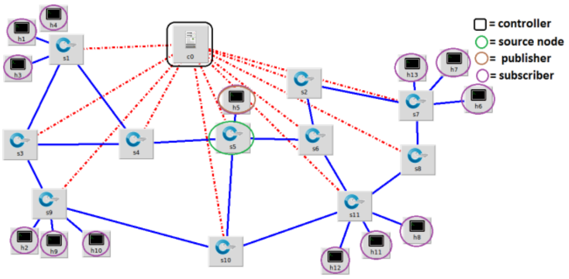

Fig. 9. The network environment for simulating multicast algorithms

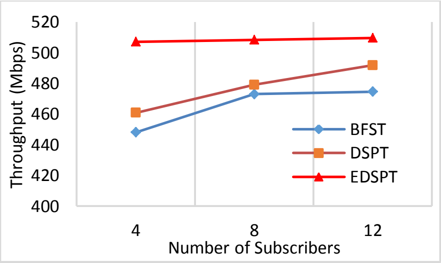

Fig. 10. The throughput comparisons

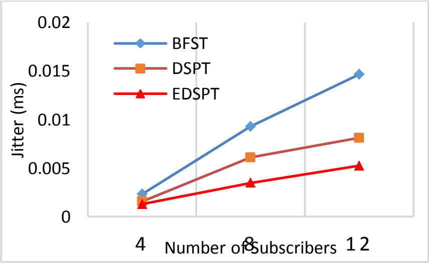

Fig. 11. The jitter comparisons

5 CONCLUSION

This paper proposes a load-balancing algorithm and 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 proposed load-balancing algorithm and compare them with the round robin and the randomized algorithm under the Abilene network topology with the Mininet emulation tool. The simulation results show that the proposed load-balancing algorithm outperforms others in terms of the end-to-end latency, response time, throughput, and standard deviation. We also use Pyretic to implement the multicast algorithms using BFST, DSPT, and EDSPT and compare them in terms of throughput, and jitter under the Abilene network topology with the Mininet emulation tool. The simulation results show that the proposed multicast algorithm outperforms others.

In the future, we plan to investigate more related load-balancing algorithms and multicast algorithms for SDN. We also plan to use more sophisticated tools, such as EstiNet, to simulate and compare investigated SDN algorithms for the sake of comprehensive performance comparisons.

REFERENCES

1. B. Nunes, M. Mendonça, X. Nguyen, K. Obraczk, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," to appear in IEEE Communications Surveys & Tutorials, 2014.
2. Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner, "OpenFlow: Enabling Innovation in Campus Networks," ACM SIGCOMM Computer Communication, 2008.
3. N. Foster, R. Harrison, M. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A Network Programming Language", in Proc. of the 16th ACM SIGPLAN International Conference on Functional Programming, 2011, pp 279-291.
4. 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.
5. Jehn-Ruey Jiang, Hsin-Wen Huang, Ji-Hau Liao, and Szu-Yuan Chen, "Extending Dijkstra's Shortest Path Algorithm for Software Defined Networking," in *Proc. of the 16th Asia-Pacific Network Operations and Management Symposium (APNOMS 2014)*, 2014.
6. Abilene Network, http://en.wikipedia.org/wiki/Abilene_Network-#cite_note-line-1, last accessed on March 4, 2014.
7. Mininet Website, http://mininet.org/, last accessed on May 2014.
8. H. Long, Y. Shen, M. Guo, and F. Tang, "LABERIO: Dynamic load-balanced routing in OpenFlow-enabled networks," IEEE 27th International Conference on Advanced Information Networking and Applications, 2013.
9. N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, "Plug-n-Serve: Load-balancing web traffic using OpenFlow," Demo at ACM SIGCOMM, Aug. 2009.
10. M. Koerner and O. Kao, "Multiple Service Load-Balancing with OpenFlow", IEEE 13th International Conference on High Performance Switching and Routing, 2012.
11. Aakash Iyer, Praveen Kumar, Vijay Mann, "Avalanche: Data center Multicast using Software Defined Networking", IEEE Communication Systems and Networks (COMSNETS), Sixth International Conference, 2014
12. John G. Apostolopoulos, Wai-tian Tan, Susie J. Wee, "Video Streaming: Concepts, Algorithms, and System," Streaming Media Systems Group Hewlett-Packard Laboratories, 2002.