Software Defined Mobile Multicast

Shunyi Xu*, Chuan Wu†, Zongpeng Li*
*Department of Computer Science, University of Calgary
{shunyi.xu, zongpeng}@ucalgary.ca
†Department of Computer Science, The University of Hong Kong
cwu@cs.hku.hk

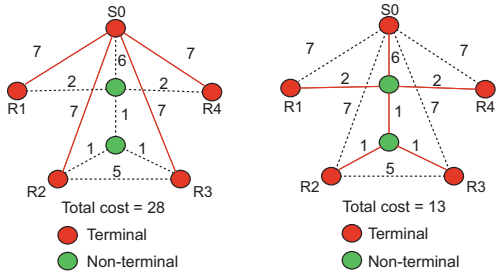
***Abstract*—Mobile multicast has been deployed in telecommunication networks for information dissemination applications such as IPTV and video conferencing. Recent studies of mobile multicast focused on fast handover protocols, and algorithms for multicast tree management have witnessed little improvement over the years. Shortest path trees represent the status quo of multicast topology in real-world systems. Steiner trees were investigated extensively in the theory community and are known to be bandwidth efficient, but come with an associated complexity. Recent developments in the Software Defined Networking (SDN) paradigm have shed light on implementing more sophisticated protocols for better routing performance. We propose an SDN-based design to combat the complexity *vs.* performance dilemma in mobile multicast. We construct low-cost Steiner trees for multicast in a mobile network, employing an SDN controller for coordinating tree construction and morphing. Highlights of our design include a set of efficient online algorithms for tree adjustment when nodes arrive and depart on the fly, and an SDN rule update framework based on constraints expressed by boolean logic to ensure loop-free rule updates. The algorithms are proven to achieve a constant competitive ratio against the offline optimal Steiner tree, with an amortized constant number of edge swaps per adjustment. Mininet-based implementation and evaluation further validate the efficacy of our design.**

I. Introduction

Mobile multicast has been deployed in real-world telecommunication networks, to support information dissemination applications such as IPTV and video conferencing. For example, the Multimedia Broadcast/Multicast Services (MBMS) brings point-to-multipoint TV service into 3G networks. The Advanced Television Systems Committee - Mobile/Handheld (ATSC-M/H), the standard to support mobile digital TV in the US, introduces multicast into television.

A key problem towards providing an efficient and robust multicast solution is the construction and maintenance of a multicast distribution tree. In existing systems, Shortest path trees (SPT) are built to distribute multicast traffic [3] [4]. SPTs connect multicast subscribers using their respective shortest paths to the source or the core, and are simple to manage. However, SPTs are sub-optimal in minimizing the consumption of network resources. In Figure 1a, to connect the four receivers (R1 to R4) with the source (S0), the SPT has a total cost of 28. In contrast, Steiner trees represent the optimal multicast topology, which has been extensively studied in the theory community [22]. A minimum Steiner tree has a lower cost of 13 in the same example network.

As demand for multicast services escalates in future mobile networks, an efficient multicast solution making the best use of network resources becomes imperative.



(a) Shortest Path Tree. (b) Minimum Steiner Tree.

Fig. 1: Multicast Distribution Trees.

The Steiner tree model consequently constitutes a better candidate for future mobile multicast networks.

However, constructing a Steiner tree as an efficient multicast topology still remains a theoretical subject up to today, mainly due to (i) the NP-hard computational complexity, and (ii) the non-smoothness of the Steiner tree — a small variation in multicast group membership results in dramatic tree structure changes.

The recent advent of Software Defined Networking (SDN) paradigm has provided an alternative in addressing the above problems. SDN [16] [5] advocates the separation of the control plane from the data plane, and aggregates decision-making functions to a logically centralized controller. Switches in an SDN network are freed from control plane computation, and specialize in simple actions such as packet forwarding. With this structural change, the controller is in possession of the complete topology information, and may execute sophisticated network algorithms such as one that constructs and maintains a Steiner tree for multicast routing.

Fig. 2 illustrates our vision of *software defined multicast* in an SDN-enabled cellular network. The backbone of the network comprises of OpenFlow switches, and a controller that is adapted for cellular networks, responsible for coordinating radio resources among base stations, tracking mobility of user equipments (UEs), managing policy and charging rules, and in our case, building steiner trees and making multicast routing decisions. A multicast source provider (*e.g.*, an IPTV streaming server) resides in the cellular network, and is connected to its subscribers in the same multicast group via a Steiner tree.

Besides, in a mobile network full of dynamics, the Steiner tree has to sustain abrupt changes of the multicast group due to user mobility, access technology change (*e.g.*, 3G to WiFi switch), etc. For example, in Fig. 2, when $UE1$ moves from $eNodeB1$ to $eNodeB2$, the Steiner tree needs to be adapted by first removing $eNodeB1$ and then adding $eNodeB2$ into the tree. The

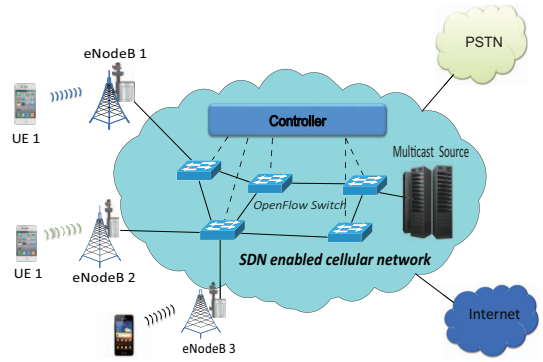


Fig. 2: Multicast in an SDN-enabled cellular network.

tree morphing process has to be smooth, as subscribers join/leave the group in real time. If too many changes are incurred, users will experience tangible glitches such as sporadic buffering in a mobile streaming application. In the worst case, even a minor change of the multicast group may result in $\mathcal{O}(n)$ [9] link changes to the previous Steiner tree (n is the total number of nodes). This translates into a significant level of control overhead to reshape the tree.

Furthermore, the process of reshaping the multicast routing tree in an SDN system may lead to transient configuration states, since updating forwarding rules in the affected switches cannot be done in a single atomic step. There is therefore a need to orchestrate the updates to ensure a consistent rule updating process.

The technical contributions of this work pivot on the above challenges. We present an online algorithm for multicast tree construction in mobile networks that **a)** approximates a Steiner tree in polynomial time and ***b)*** goes through a smooth tree morphing process with a low number of changes in reshaping the tree, as the network topology and multicast group evolves over time. We further design routing rule update procedures to guarantee consistent, loop-free SDN rule updating. Our contributions can be summarized as following:

The rest of the paper is organized as follows: Sec. II discusses the background and related work. Sec. III lays out the overall architecture of the proposed mobile multicast framework. The online algorithm to construct the Steiner tree in a full dynamic mobile network is discussed in Sec. IV, followed by a set of procedures to achieve loop-free switch update in Sec. V. The evaluation results are presented in Sec. VII. Finally we conclude this paper in Sec. VIII.

II. Related Work

Multicast in mobile networks typically uses shortest path trees to connect multicast group members, similar to that in static multicast network protocols [21] [19] [3] [4]. Smooth handover becomes a challenge under high node mobility, and low delay, time-sensitive handover has been the focus of recent studies [8].

A few existing studies address multicast in SDN-enabled networks. Castflow [15] explores implementing the current Shortest Path Tree (SPT) based multicast in a generic SDN network. Li *et al.* [14] focus on the multicast scalability in datacenter networks. In particular, their work aims to scale multicast up by distributing the multicast address better in datacenter topologies. Aakash *et al.* [10] investigate multicast routing in datacenter networks by using a randomized algorithm to connect new nodes to the established tree. Our work differs by focusing on minimizing the multicast distribution tree cost in a mobile network, and in particular, dealing with the abundant dynamics in such a network. It not only requires constructing a bandwidth efficient tree, but also efficient tree morphing that can adapt to multicast group changes in real time without incurring much control overhead.

The NP-hard Steiner Tree problem has been extensively studied in theoretical computer science [11] [13] [12] [18] [23] [20] [2], which aims to find a tree of minimum cost spanning a given set of nodes, known as *terminals*, which possibly also includes some non-terminal nodes, or Steiner points. The cost of the tree is the sum of link costs in the tree.

The dynamic variant of the Steiner Tree problem deals with a terminal set that evolves over time. In the *fully dynamic* version, terminal nodes can join and leave the multicast group. Imase *et al.* [9] is among the first to show that, in the fully dynamic scenario, δ-bounded tree algorithm (which mandates that the cost of any edge in the path between two nodes can not be δ times larger than the distance between them) can achieve an overall tree cost no worse than 4δ of the offline optimum. In addition, the total number of edge changes is $O(K^{3/2})$ in the first K node join/departure events. Studies of the fully dynamic scenario remain sporadic until Gupta *et al.* [7] recently proved that an amortized constant number of edge changes are sufficient to maintain a 4-competitive Steiner tree, a considerable improvement over Imase's bound of $O(K^{1/2})$ [9] in the amortized case. However, they assume the closest point to a newly added vertex is one of the undeleted terminals, which isn't necessarily realistic.

In the *partially dynamic* scenario, where nodes can only be added to, but not removed from the group, Megow *et al.* [17] show that, in total, $O(K)$ edge changes are sufficient to maintain a $O(2(1+\epsilon))$-competitive tree, where ϵ is an arbitrary parameter between 0 and 1. Gu *et al.* [6] show that only two edge changes are needed to maintain a tree of a constant competitive ratio. Our online algorithm design builds upon the above literature. We extend Megow *et al.* [17] and Gu *et al.* [6]'s work, which only address node additions, into the fully dynamic case. This work differs by making full use of edge swaps and edge tracing to achieve a similar competitive ratio to that in [9], but with an amortized constant number of link changes.

III. Architecture and Design Goals

We consider multicast in an SDN-enabled mobile network, as illustrated in Fig. 3. Base stations (*e.g.*, eNodeB) serve as the front ends connecting user equipments (UE) with the backbone network. Switches in the backbone are SDN-capable, communicating with the controller via OpenFlow interfaces. The controller is responsible for computing and maintaining the multicast tree, as well as translating it into routing rule updates and deciding the sequence of updates to enforce on the switches. Edge switches adjacent to base stations, to which multicast group members are connected, send membership updates (*join* or *departure*) to the controller. Multicast group information is passed through the Internet Group Management Protocol (IGMP), which is widely deployed to establish multicast group

membership of hosts. Multicast membership information and routing rules are passed between the switches and the controller in the events of group membership changes, in OpenFlow Protocol (OFP) messages.

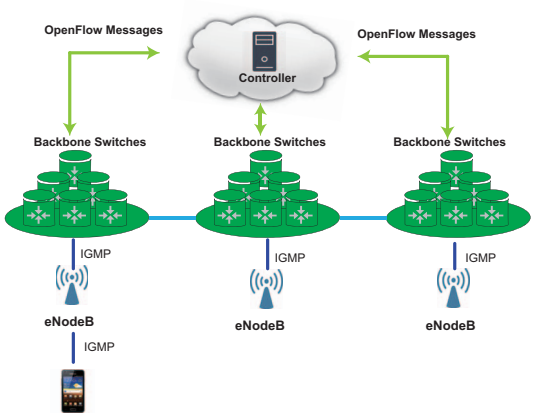


Fig. 3: Architecture of SDN-based Mobile Multicast.

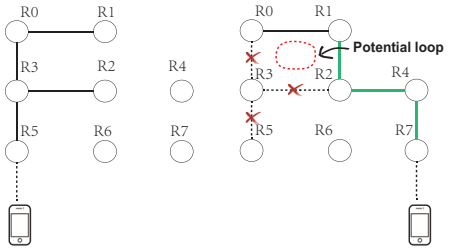
In a software defined multicast system, interested users initiate IGMP join messages destined to the *eNodeB*, and are further forwarded to the edge switch. The edge switch checks if the targeted multicast group is already subscribed by some hosts in its subnet (*i.e.*, if its *flow table* contains rules about the multicast group address). If so, it adds the inport that receives the IGMP message to its group of outports for the targeted multicast group, and drops the IGMP message. Otherwise, the switch forwards the *join* request up to the controller. The controller executes the Steiner tree maintenance algorithm for adding this edge switch, disseminating the latest multicast rules to affected switches via OpenFlow messages. After the rule updating process, switches on the path to the subscriber simply checks against their *flow tables* to forward multicast packets.

Upon departure, a subscriber sends an IGMP *leave* message. The corresponding edge switch checks if this is the last subscriber in this group. If not, it removes the inport that receives the IGMP message from the group of outports for the multicast group. Otherwise, the IGMP *leave* message is sent to the controller, which executes the Steiner tree algorithm to remove the edge switch. The decisions are disseminated to all other affected switches. Once the rule update process completes, the subscriber is detached from the multicast group.

Towards efficient software defined multicast in such a dynamic system, we aim to achieve the following three goals:

(i) ***Fast, incremental adjustment of the Steiner tree upon subscriber join/departure.*** A straightforward algorithm that recalculates a Steiner Tree upon each change is not practical: in the worst case, a total number of O(n^2) of changes happen upon the arrival of the nth member [9] (on average, O(n) per change), incurring significant control overhead. We seek to design an efficient online algorithm that adjusts the multicast tree smoothly, i.e., reducing the number of changes, while minimizing tree cost.

(ii) ***Cost-competitive Steiner tree over the long run.*** We aim to maintain a low-cost Steiner tree on the fly, achieving a small competitive ratio in overall tree cost, as compared to the offline, optimal Steiner tree computed assuming full knowledge of node dynamics in the future. A cost minimizing multicast tree minimizes the consumption of network resources. As multicast data typically contribute to a substantial fraction of network traffic, high multicast bandwidth efficiency reduces the risk of congestion, and helps improve user experience.



(a) Before update. (b) After update.

Fig. 4: An example rule update pitfall.

(iii) ***Loop-free rule update.*** After the new Steiner tree is computed, a set of SDN rules need to be updated at switches. However, in a real-world network with switches distributed across a potentially a large geographical span, transient states arise due to transmission and processing delays. Consider the example in Fig. 4. When the UE moves away from R5 and wishes to reconnect to the multicast group through R7, the controller reshapes the distribution tree accordingly. As a result, edges R1-R2, R2-R4 and R4-R7 are added, while R0-R3, R2-R3 and R3-R5 are removed. If OpenFlow rules governing R0-R3 and R2-R3 are not removed when rules over R1-R2 are installed, there arises a transient routing loop R0-R1-R2-R3 in the network. Therefore, we need to carefully orchestrate the OpenFlow rules updating process to guarantee the robustness of a software defined multicast system.

IV. Online Steiner Tree Algorithm for Software Defined Multicast

TABLE I: Notation

Symbol	Definition
$G = (V, E)$	Complete metric graph with node set V, link set E
$V(H)$	The node set for a graph H
$E(H)$	The edge set for a graph H
r_i	The ith request arriving at round i.
p_i	The requested operation at round i, can be add or $remove$
T_i	The Steiner (multicast) tree at round i
P_i	Terminals in Steiner tree T_i.
D_n	Nodes that are added but then removed by round n
V_n	Nodes that appeared by round n, including P_n and D_n
SP	Steiner points in T_n
S	Swappable edges in transition from T_{n-1} to T_n
S'	Non-swappable edges in transition from T_{n-1} to T_n
G'	Subgraph of T_n formed by removing S' from T_n. $G' = (V, S)$
MST^*	The minimum spanning tree on V_n.
MST_i	The minimum spanning tree on P_i for round i.
ϵ	a tunable constant parameter in (0, 1)
$f_t^{n(t)}$	if not null, it is an edge in the current tree T_n, which is obtained by a series of swapping ($f_t^0 \rightarrow f_t^1 \rightarrow \cdots \rightarrow f_t^{n(t)}$); otherwise, it denotes $f_t^{n(t)-1}$ was deleted

A. Problem Formulation

Let $\dot{G} = (V, \dot{E})$ be the network of switches in our system, where V is the set of switches and $\dot{E}$ is the set of links connecting them. There is a link cost

function c mapping $\dot{E}$ to $\mathcal{N}^+$, *i.e.*, $c(\dot{e})$ is the cost of link $\dot{e} \in \dot{E}$. Where the context is clear, we will simply use $c(\mathcal{E})$ to denote the overall cost of links in a set $\mathcal{E} \subseteq \dot{E}$. Formally, let $R_k = \{r_1, r_2, ..., r_k\}$ denote the sequence of join/departure requests, which are revealed one after another. Each time a new request arrives is taken as a *round*, *i.e.*, request r_i arrives at round i. r_i is a pair (v_i, p_i), where $v_i \in V$ and $p_i \in \{add, remove\}$, indicating whether edge switch v_i is to be added into or removed from the multicast tree.

Our goal is to compute a low-cost tree connecting all the edge switches that have been added but not deleted (terminals, denoted by set P), which may span other switches (Steiner points, denoted by set SP) as well. Essentially, we are solving the following optimization problem.

$$\min_{e \in T} c(e)$$
$$subject\ to\ \forall v \in P \rightarrow v \in V(T)$$

The above optimization is NP-hard, and we resort to an online algorithm that approximates the Minimum Steiner Tree. We convert the network topology $\dot{G}$ = (V, $\dot{E}$) to its metric space representation G = (V, E), where V remains intact and each edge $e = (v_1, v_2) \in E$ represents the shortest path between v_1 and v_2 of cost $c(e)$ in the original topology $\dot{G}$. The conversion can be done in almost-quadratic time by running an all-pairs shortest path algorithm on $\dot{G}$. In a metric space, the distance between any pair of nodes is well defined, and satisfies the triangle inequality, which facilitates the design and analysis of efficient graph algorithms. We seek to build a min-cost tree connecting all the terminals in P in the metric space G, and then convert it to the multicast tree in $\dot{G}$ for routing rule installation in the actual switch network. The conversion can again be done in polynomial time, by replacing the metric space edge with a physical shortest path between the two end points. We will then show that the multicast tree in the original topology $\dot{G}$ is cost competitive, as compared to the offline mininum-cost tree. Table I summarizes notation for ease of reference.

B. Online Tree Morphing Algorithm

Let T be an existing tree at any time, $e \in T$ and $f \in E \backslash T$. The key idea of our online algorithm is based on *edge swaps*, which removes an in-tree edge e from T while at the same time adds an out-tree edge f to T. This happens only if $c(f) < c(e)$ and $T_n \setminus e \cup f$ is still a connected tree. If an unlimited number of such swaps are allowed, the optimal tree can be obtained when no further cost reduction can be achieved. However, the number of swaps may not be polynomial (recall the NP-hardness of the Steiner tree problem). To achieve an efficient algorithm by removing unnecessary swaps, we resort to the following heuristics: (I) A swap is taken only if $c(f) < \frac{c(e)}{1+\epsilon}$, where $\epsilon \in (0, 1)$. With this, we can adjust the threshold of swapping. (II) We differentiate between the set of *swappable edges* and the set of *non-swappable edges*, and only do swaps on *swappable edges*. Specifically, we associate each edge e in the current tree with an MST_i in previous round i ($i < n$). If $c(MST_i) \leq \epsilon \cdot c(MST_n)$, we categorize e as non-swappable, and vice versa. The intuition is that if an edge's associated MST_i is already very cheap compared with MST_n. It's an indication that the edge is also very cheap. Hence it would not bring sufficient benefits to swap them out.

The first heuristics is relatively easy to understand; we next explain our second heuristic with rigorous analysis below.

Every edge in the tree T_n of the current round n is obtained by either connecting a newly added terminal to the nearest existing terminal, or replacing some existing edge in the tree. Inspired by edge tracking techniques from latest literature on Steiner trees [17] and [6], we can generally present the 'trace' of the edge using a sequence of edges $\mathscr{F}_t = \{f_t^0, f_t^1, ..., f_t^i, ..., f_t^{n(t)}\}$. Here f_t^0 is the origin edge added to multicast tree T_t in an earlier round t ($t \leq n$) by connecting a new terminal v_t to the closest terminal in the tree. f_t^i denotes the edge that replaces f_t^{i-1} in a swap at some round $k \in (t, n]$. $f_t^{n(t)}$, if not null, is the outcome after this series of edge swap, and is the only existing edge (of the sequence $\mathscr{F}_t$) in the tree T_n. The total number of swaps in this series of swapping is recorded by $n(t)$. The edges in tree T_n can be expressed as $E(T_n) = \{f_t^{n(t)}, \text{for all } t \in [1, n], \text{ where } f_t^{n(t)} \neq null\}$.

We have the following property regarding the origin edge that an edge in T_n is traced back to. All the proofs of lemmas and theorems in this paper can be found in the technical report [1].

Lemma 1: f_t^0, the origin edge for some edge $f_t^{n(t)}$ in T_n, is an edge in MST_t.

From Lemma 1, we know that the cost of edge f_t^0 cannot be larger than $c(MST_t)$, because f_t^0 is in the tree MST_t. Since the swaps in sequence $\mathscr{F}_t$ proceed in the direction of reducing the edge cost, we know that $c(f_t^{n(t)}) < c(f_t^0) \leq c(MST_t)$ holds. In this sense, we associate $f_t^{n(t)}$ with MST_t. If $c(MST_t) \leq \epsilon \cdot c(MST_n)$, and we know that $c(f_t^{n(t)})$ is even smaller from the above analysis, we can deduce that the edge $f_t^{n(t)}$ is already very cheap. As a result, it wouldn't bring significant benefits to swap it out.

Formally, we define the set of *swappable edges* in round n as following,

$$S = \{f_t^{n(t)} | f_t^{n(t)} \in E(T_n), c(MST_t) > \epsilon \cdot c(MST_n)\}. \quad (1)$$

The set of *non-swappable edges* is just the complement of S.

Furthermore, our algorithm seeks to maintain an *extension tree*, which satisfies the following two conditions: (a) it is a Steiner tree and (b) all Steiner points in the tree has a degree of at least 3. There are nice properties with such an extension tree, which we will show in our analysis. Intuitively, we do not want to make changes to Steiner points with degree at least 3, since removing them would incur much more rewiring work than removing Steiner points with degrees 1 or 2. Edge swapping or node removal may lead to Steiner points with degrees smaller than 3. In that case, we can convert the tree back into an extension tree with the following two operations: (a) delete Steiner Points of degree 1 and its incident edge; and (b) delete Steiner Points of degree 2 as well as the two incident edges, and add back the edge connecting its two neighbors.

Our complete algorithms for morphing the tree in round n are given in Alg. 1 and Alg. 2, in the cases that a terminal is added or removed, respectively.

Algorithm 1 Adding a Terminal v_n

1: $P_n \leftarrow P_{n-1} \cup v_n$
2: $f_n^0 \leftarrow$ shortest path edge from v_n to P_{n-1}
3: $T_n \leftarrow T_{n-1} \cup f_n^0$
4: $S \leftarrow$ CALCSWAPPABLEEDGE($\mathscr{F}$)
5: **while** $\exists (e, e')$ s.t. $e \in S, e' \in E \setminus T_n$ and ISVALIDPAIR(e, e') **do**
6: EDGESWAP(e, e')
7: **if** T_n is not an extension tree **then**
8: convert T_n into an extension tree
9: **return** T_n
10:
11: **function** ISVALIDPAIR(e, e')
12: **if** $\frac{c(e)}{c(e')} > 1 + \epsilon$ and $T_n \cup e' \setminus e$ is a connected tree **then**
13: **return** True
14: **return** False
15:
16: **function** CALCSWAPPABLEEDGE($\mathscr{F}$)
17: $S = \emptyset$
18: **for all** $t \leftarrow 1$ to $n-1$ **do**
19: **if** $c(MST_t) > \epsilon \cdot c(MST_n)$ **then**
20: $S \leftarrow S \cup f_t^{n(t)}$
21: **return** S
22:
23: **function** EDGESWAP(e, e')
24: $T_n \leftarrow T_n \cup e' \setminus e$
25: find s such that $f_s^{n'(s)} \rightarrow e$
26: $n(s) \leftarrow n'(s) + 1$
27: $f_s^{n(s)} \leftarrow e'$

Algorithm 2 Removing a Terminal v_n

1: $P_n \leftarrow P_{n-1} \setminus v_n$
2: $T_n \leftarrow T_{n-1}$
3: TOEXTENSIONTREE(v_n)
4: $S \leftarrow$ CALCSWAPPABLEEDGE($\mathscr{F}$)
5: **for all** pair (e, e'), $e \in S, e' \in E \setminus E(T_n)$ **do**
6: **if** $\frac{c(e)}{c(e')} > 1 + \epsilon$ **then**
7: EDGESWAP(e, e')
8:
9: **function** TOEXTENSIONTREE(v_n)
10: **if** degree of v_n == 1 **then**
11: find neighbor node v_i of v_n in T_n
12: $T_n \leftarrow T_n \setminus v_n v_i$
13: find s such that $f_s^{n'(s)} \rightarrow$ edge $v_n v_i$
14: $f_s^{n'(s)+1} \leftarrow null$
15: **if** v_i is a Steiner Point **then**
16: TOEXTENSIONTREE(v_i)
17: **else if** degree of v_n == 2 **then**
18: find neighbor nodes v_i, v_w of v_n in T_n
19: $T_n \leftarrow T_n \setminus (v_n v_s \cup v_n v_w) \cup v_i v_w$
20: find s such that $f_s^{n'(s)} \rightarrow$ edge $v_n v_i$
21: $f_s^{n'(s)+1} \leftarrow null$
22: find y such that $f_y^{n'(y)} \rightarrow$ edge $v_n v_w$
23: $f_y^{n'(y)+1} \leftarrow null$
24: $f_n^0 \leftarrow v_i v_w$
25: **else** ▷ do nothing for Steiner points of degree ≥ 3
26: **return**

(1) Add Request (Alg. 1). Suppose a terminal v_n is added in round n. We first find a shortest-path edge $v_n v_k$ to connect v_n to an existing terminal, where

$$v_k = \arg\min_{v_i \in P_{n-1}} c(v_n v_i). \tag{2}$$

the edge $v_n v_k$ is called the *greedy edge*, because it's achieved by choosing the closest terminal to connect v_n. It's also called the *origin edge* in the sense that it marks the beginning of edge tracking for a new round n, in the series $\mathscr{F}$. Accordingly, we set f_n^0 to be $v_n v_k$ (line 2). We proceed to perform edge swaps, if possible, to reduce the overall tree cost, following the two heuristics explained earlier. We first find out the set of swappable edges S (line 4), by determining for each edge in T_{n-1}, whether the cost of its associated MST is ϵ times smaller than $c(MST_n)$. Subsequently, we check if there exists a pair $e \in S$, and $e' \in E \setminus E(T_n)$ such that $\frac{c(e)}{c(e')} > (1+\epsilon)$ and $T_n \setminus e \cup e'$ is still a connected tree (lines 5). If so, we replace edge e by e'. We also find round s when e's associated origin edge was added into the tree, and update $\mathscr{F}_s$ to track e' (lines 6), by adding $e' = f_s^{n(s)}$ to the end of the sequence $\mathscr{F}_s$, where $n(s) = n'(s)+1$ and $n'(s)$ is the number of swaps from the origin edge to e.

Alg. 1 is called each time a terminal is to be added. As a result, a series of sequences $\mathscr{F} = \{\mathscr{F}_1, \mathscr{F}_2, ..., \mathscr{F}_n\}$ are maintained that keep tack of the evolution of all the edges which once appear in our tree.

(2) Removal Request (Alg. 2). Upon the removal request for v_n, our algorithm removes v_n from the terminal set P. If the degree of the node is 1, we delete its incident edges; if the degree is 2, we rewire its incident edges to be directly connected. Otherwise, we take no action, to avoid expensive reshaping efforts. When we delete any edge e, we mark its relevant $\mathscr{F}$ null (lines 13-14, lines 20-23): find round s such that $e = f_s^{n'(s)}$, and set $f_s^{n'(s)+1}$ to be $null$, such that $\mathscr{F}_t$ stops tracking edges onwards.

If a non-extension tree results from the deletion, we convert it into an extension tree (line 3,16). The process of conversion might introduce some edge pairs whose cost ratios exceed $1+\epsilon$. If so, we perform edge swaps to remove such pairs (lines 4-7). As in the *addition* algorithm, unswappable edges are kept from swapping.

C. Performance Analysis

We next benchmark the performance of our online algorithm by evaluating its competitive ratio, the worst-case ratio between the tree cost of our algorithm compared to the cost of the optimal, offline Steiner tree. We also analyze edge swap complexity throughout the tree morphing process.

Competitive Ratio. In any round n, let S and S' denote the set of edges that are swappable and non-swappable, respectively. By removing S' from T_n, we divide the tree T_n into two parts, a forest G' and the non-swappable edges S'. We give the following lemmas, whose analysis can be found in the technical report [1].

Lemma 2: Let $G' = G(V, S)$ be the forest induced on the nodes V and swappable edges S. In other words, G' is formed by removing S' from T_n. Then $c(G') \leq 2(1+\epsilon)c(MST_n)$.

Lemma 3: For any round n, let T_n be the tree built by our algorithm, and MST_n be the minimum spanning tree over all terminals P_n at round n. Then

$$c(T_n) \leq \frac{2(1+\epsilon)}{1-\epsilon} \cdot c(MST_n)$$

The above lemma implies the following bound.

Theorem 1: Our online algorithm described in Alg. 1 and Alg. 2 achieves a competitive ratio of $4(1+\mathcal{O}(\epsilon)$ in the overall tree cost of the physical space, as compared to the offline optimal Steiner tree, where $\epsilon \in (0,1)$.

Edge Swap Complexity. We next derive an upper bound on the total number of edge swaps in each round with our online algorithm.

Define $\mathcal{K}$ to be the largest cost ratio of any edge pair in our metric space network $G(V,E)$, *i.e.*,

$$\mathcal{K} = \max_{e_1,e_2 \in E} \frac{c(e_1)}{c(e_2)}.$$

In physical space, $\mathcal{K}$ can be interpreted as the length of the longest path.

Theorem 2: Our online algorithm performs an amortized number of $\frac{2+\log \mathcal{K}}{\log(1+\epsilon)}$ edge swaps in each round, in the metric space $G=(V,E)$.

Note that rules update occurs in the physcial space, so we need to map Theorem 2 to the phsysical space. This is given by the following theorem.

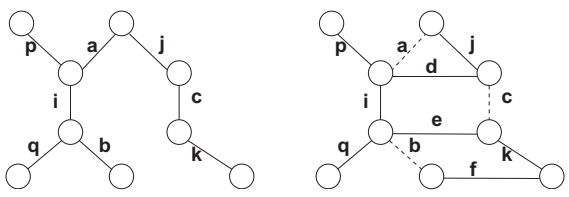
Theorem 3: Our online algorithm induces an amortized number of $\frac{2\mathcal{K}(2+\log \mathcal{K})}{\log(1+\epsilon)}$ link changes in the physical switch network.

In a typical multicast application, the network topology is fixed. Hence the topology parameter $\mathcal{K}$ is fixed. The number of link changes is not related to the multicast group size, the number of multicast groups, or the arrival patterns of multicast nodes. In this scenario, the number of link swaps is a constant in amortized sense.

V. Loop-free Multicast Rule Update

When a new multicast tree is decided, link change decisions need to be disseminated to the affected SDN switches. Since we are maintaining a metric space distribution tree, the first task is to resolve the tree into a physical space distribution tree, by replacing each edge with a shortest path between its end nodes, removing duplications of links, if there exists any.

After getting the resolution above, we obtain an actual hop-by-hop multicast tree topology. We now explain our rule updating approach on the physical space with a specific example. Figure 5a and Figure 5b shows the tree structure before and after the update: edges a,b,c are deleted and edges d,e,f are added. In the transition stage, several loops might appear, including $a-d-j$ and $d-i-e-c$.



(a) Before Update (b) After Update

Fig. 5: Loop-free Rule Update

We first find the potential loops by logically adding all to-be-added links to the tree without removing any of the to-be-deleted links. This can be done with DFS traversal and a loop is found if we step onto a node that lies in the visited stack. In each loop detected, we identify a dependence relationship. For example, in loop $a-d-j$, we have dependence pair $(a \to d)$, meaning that edge d needs to be added after a is removed. In loop $b-e-k-f$, we have dependence $(b \to e \wedge f)$, where we use $\wedge$ to denote that after removing b both e and f can be safely added. In loop $i-d-c-k-f-b$, we have dependence $(c \vee b \to f \wedge d)$, where we use $\vee$ to denote that after removing either c or b we can safely add f and d. All such dependencies in this example topology are given in Table II.

For each to-be-added link, we find the dependence relations it is involved, and aggregate all such dependence relations into a compact logic expression. For example, edge f is involved in the dependencies $(b \to e \wedge f)$, $(c \vee b \to f \wedge d)$ and $(a \vee c \vee b \to f)$. Since we are only concerned with f right now, those three dependencies can be simplified to $(b \to f)$, $(c \vee b \to f)$ and $(a \vee c \vee b \to f)$. We aggregate them into a logical expression as follows:

$$\begin{aligned} & b \wedge (c \vee b) \wedge (a \vee c \vee b) \to f \\ \Leftrightarrow\ & (b \wedge c \vee b) \wedge (a \vee c \vee b) \to f \\ \Leftrightarrow\ & b \wedge (a \vee c \vee b) \to f \\ \Leftrightarrow\ & b \to f \end{aligned}$$

which implies that edge f is exclusively dependent on edge b. Similarly, the aggregated dependence for edge d is:

$$a \wedge c \wedge (c \vee b) = (a \wedge c) \vee (a \wedge b) = a \wedge c \wedge (1 \vee b) = a \wedge c \to d$$

which implies that edge d can only be added after edge a and c are both removed. Table II summarizes aggregate dependency for each to-be-added edge.

TABLE II: Dependencies in Fig. 5

Dependence Relations
$a \to d$
$a \vee c \to e$
$c \to d \wedge e$
$a \vee b \vee c \to f$
$b \vee c \to d \wedge f$
$b \to f$

Aggregate Dependency for Each to-be-Added Edge
$a \wedge c \to d$
$b \wedge c \to e$
$b \to f$

Loop-free rule update algorithm. We present our algorithm in Alg. 3. We first detect all potential loops, and store dependence relations in a list (lines 2-8). We then take a second pass of the list, to aggregate any dependencies induced on one to-be-added link to a logic expression (lines 10-11). The actual rule update procedure is initiated by first sending OpenFlow remove messages to all switches induced on the entire set of to-be-removed links (line 13). We check if there is any ACK returned from the switches. If all ACKs for one link removal are received, the logic expressions induced on that link are re-evaluated (lines 15-17). If the value of the dependency logic expression for some to-be-added link turns 1, which implies that the dependence is gone, we can immediately add that link to the network (line 18). We do so by sending out OpenFlow *add* messages, and continue the process until all dependency logic expressions are evaluated to be 1 (such that all new links are added).

Correctness. In Alg. 3, a new edge is added only when its dependencies with the to-be-removed edges are gone. The new edge will not induce a loop together with other existing tree edges or other to-be-added edges, since altogether they constitute the new multicast tree. It is therefore easy to see that our algorithm can correctly achieve loop-free rule updates.

Algorithm 3 Loop-free Rule Update Algorithm

Input: T_{n-1} - previous tree; S_r - set of edges to be removed; S_a - set of edges to be added

1: $T = T_{n-1} \cup S_a$
2: $dependencePairs$ = [ ]
3: $logicEquations$ = [ ]
4: // find dependence relations
5: **for all** loop in T **do**
6: dependants = loop $\cap S_a$
7: hosts = loop $\cap S_r$
8: $dependencePairs$.add($(hosts \rightarrow dependents)$)
9: // aggregate dependence relations for each to-be-added edge
10: **for all** $link$ in S_a **do**
11: aggregate $dependencePairs$ relevant to the $link$ into one logical expression and add it to $logicEquations$
12:
13: send OpenFlow remove messages to switches incident on S_r
14: **while** $logicEquations$ not empty **do**
15: **if** ACKs are received for removing link e from its incident switches **then**
16: find logic expression set L in $logicEquations$ which contain e in their left-hand sides
17: re-evaluate each logic expression in L; let set M include to-be-added links whose dependence logic expressions turn 1
18: send OpenFlow add messages to all switches incident on M
19: delete dependencies defined for links in M from $logicEquations$

Time complexity. We use DFS to detect loops, and use a hashtable to store the dependencies. Since a potential cycle always contains at least one to-be-added link, and a to-be-added link cannot be shared by two cycles (otherwise, the union of the two cycles is a larger cycle, contradicting the tree topology), the total number of potential cycles is upperbounded by $|S_a|$. Using a DFS to find a loop takes $\mathcal{O}(|V| + |E|)$ operations. Therefore, lines 5-8 take a total of $\mathcal{O}((|S_a|)(|V|+|E|)(|S_a|+|S_r|))$ operations. Lines 10-11 take $\mathcal{O}(|S_a|)$ time since we need to go through $|S_a|$ loops. Lines 13-19 take $\mathcal{O}(|S_a| + |S_r|)$ operations in the worst case, which happens when one needs to wait for all to-be-deleted edges to be removed before inserting any to-be-added edge. It's known from Theorem 2 that the amortized number of edge changes is constant. Hence $|S_a| + |S_r|$ is a constant number in the amortized sense, which implies $|S_a|$ is a constant in amortized sense as well. Therefore the algorithm takes $\mathcal{O}(|V| + |E|)$ time in amortized sense. Note $V = V(T_{n-1}), E = E(T_{n-1}) + S_a$, and because S_a is a constant in amortized sense, we can see that $\mathcal{O}(|V|+|E|)$ is essentially $\mathcal{O}(|V(T_{n-1})|+|E(T_{n-1})|)$, which is linear in the tree size in the previous round.

VI. System Implementation

We have implemented a prototype of the proposed multicast framework with 2K lines of Java code. The system relies on Floodlight as the controller, and switches are emulated in a Mininet network. The multicast functionality is implemented in two modules on top of Floodlight. Fig. 6 shows the major components.

▷**Multicast Handler.** This is the frontline that processes multicast routing requests from switches. It listens to the port where the OpenFlow messages enter, and checks whether a message encapsulates an IGMP report packet If the message contains an IGMP report packet, relevant information is passed to the multicast module. If the message contains an ACK for rule update, it is passed to the loop-free update module. Otherwise, the message is left to be processed by other modules in the controller.

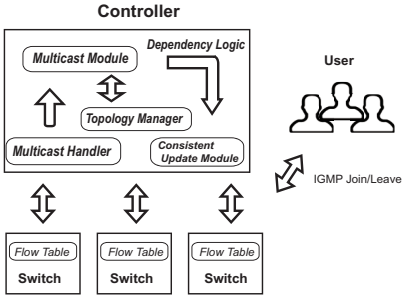


Fig. 6: Main Components in the Prototype System

▷**Multicast Module.** This module implements the tree morphing algorithms in Sec. IV. It identifies the edge switch that intends to join or leave the multicast group by its global switch ID encapsulated in the OFP message. It fetches the distances between switches from the topology manager module that comes with Floodlight, and transforms the physical space graph to its metric space representation by calculating the cost on metric space edges. It then carries out our online algorithm to build the Steiner tree depending on the request. The outcome is a set of edges to be added or deleted in metric-space, which are further resolved into physical space links. These link changes are then fed as input to the loop-free update module.

▷**Loop-free Update Module.** This module carries out the loop-free update algorithm in Sec. V. Dependency relations are derived based on the links to delete and to add, in the form of logic expressions. The module dispatches OpenFlow *remove* messages to relevant switches. When an ACK is received, relevant logic expressions are re-evaluated, and new edges whose dependencies are cleared are decided. The module then composes OpenFlow messages that add relevant rules to the incident switches. It pushes out the rule-installing OpenFlow messages and continues to wait for more ACKs.

VII. Performance Evaluation

We emulate multicast networks of varying sizes in Mininet with the Floodlight controller. For each experiment, we make a certain percentage (20 percent by default) of the network nodes participate in the same multicast group. They join the multicast group in a fixed order every two seconds. We further specify a portion (20 percent by default) of the network nodes to come and go on the fly. For those dynamic nodes, add requests arrive following a Poisson distribution, where λ is set to be 20 per minute (join events 3 seconds apart on average), and each added node stays for a lifetime following a Zipf distribution (with the exponent set to 2) before sending out a removal request. Add requests and removal requests are implemented with a control snippet using *iperf*. The topologies of the switch networks are generated with the topology generator BRITE, under the Waxman's model. However, since there's no easy way to generate topologies that takes

$\mathcal{K}$ (the length of the longest path) as a parameter, we take an indirect way by modifying the topology configuration files generated by BRITE, and connect a quarter of the nodes via a simple path. For example, in a topology of 400 nodes, we manually create a path of length 100, so we know $\mathcal{K}$ is at least 100. We would see the impact of $\mathcal{K}$ on link changes later. By default, we set ϵ to be 0.8.

We also implement two other algorithms for comparison: (i) The Shortest Path Tree (SPT) approach commonly used in the existing multicast networks. (ii) KMB [13], a well-known Steiner Tree approximation algorithm, which calculates a new Steiner Tree from scratch upon each add/removal request. KMB achieves a competitive ratio of 2 against the optimal Steiner Tree. It therefore serves as a good benchmark to compare with classical Steiner tree approximation algorithms.

Cost of Multicast Tree vs. Topology Size. We first vary the total number of switches in the network to investigate the costs of the distribution trees. Fig. 7 shows that the tree cost from our algorithm is consistently lower than that of SPT. The cost reduction is especially abundant when the size of the topology is larger, *e.g.*, a 19% cost reduction is achieved when there are 400 nodes in the network.

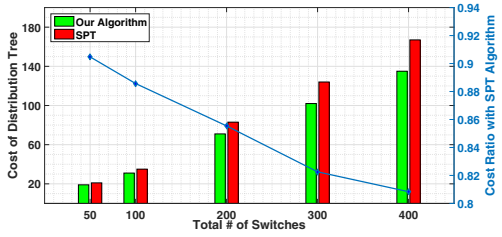


Fig. 7: Tree Costs as Size of Topology Changes

Number of link changes vs. Topology Size. We investigate the number of link changes as the size of the topology changes. Fig. 8 reveals that the amortized number of link changes per round with our algorithm is similar to that with SPT. SPT simply connects the new node via the shortest path to the multicast source, while our algorithm connects the new node to the closest node in the established tree and does extra edge swaps to reduce the cost. The results show that the number of changes with both algorithms are comparable. It's also noted that the value of $\mathcal{K}$ has small effects on the result. For example, in the topology of 400 switches, we know $\mathcal{K}$ is at least 100. The amortized number of link changes is barely around 12, which is much smaller than its theoretical upper bound.

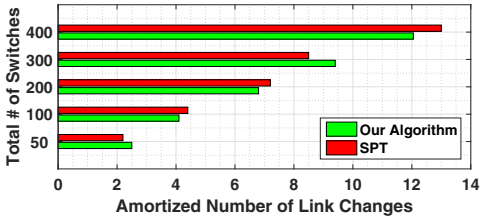


Fig. 8: Amortized Number of Link Changes per Round

Impact of ϵ. In this set of experiments, we fix the topology size to 100. There are 20 switches joining in a predefined order and they never leave. There are another 20 nodes which join and leave following the above mentioned distribution. So there are in total 40 add requests and 20 removal requests. The tree cost ratios between our algorithm and those of SPT and KMB are shown in Fig. 9. We first observe that the tree cost with our algorithm is slightly smaller when ϵ is smaller. This is because the threshold of performing a swap is lower with smaller ϵ, and more cheaper edges can be swapped in, which is consistent with Theorem 3. Secondly, although in theory our algorithm is at least two times worse than KMB (our algorithm achieves a competitive ratio larger than 4 and KMB has a competitive ratio of 2), the results show that the cost ratio between our algorithm and KMB is around 1.2 and 1.3 in practical scenarios.

Fig. 10 illustrates the impact of ϵ on the number of link changes. For better visualization, we organize every 10 rounds out of the 60 rounds into one epoch, and plot the total number of link changes in each epoch. When ϵ is smaller, the number of link changes is larger with our algorithm, due to the lower edge swapping threshold. Therefore, the value of ϵ decides a trade-off between the tree cost and the number of link changes. It is also observed that many more link changes are needed in KMB in order to achieve the slightly lower tree cost. This is because KMB recalculates a new Steiner tree from scratch, disregarding informations about previously built tree, and hence would incur abrupt changes.

Packet Loss Reduction. The reason for packet losses is two fold. Without orchestration, rules in the switch may be long removed before their counterparts are added. If packets enter the switch during the gap, they will get dropped. On the other hand, if a loop is formed, it will likely lead to congestion on low-bandwidth links, especially when there is a large volume of traffic in the network. Packets will also get dropped in this case. Therefore, we investigate how effective our loop-free module help orchestrate the rule-updating process, and avoid transient loops by measuring packet loss rates. We carry out the experiments under different network sizes, and two different densities: 80% and 60% of all switches join the multicast group over time, respectively, and half of them will leave after a Zipf lifetime. Fig. 11 shows the packet loss reduction ratios, *i.e.*, the total number of packet losses without the loop-free rule update module minus the total number of packet losses with the loop-free rule update module , divided by the former. We observe that the reduction ratios increase when the network is larger and more switches in the multicast group, revealing the benefit of our algorithm in larger and denser multicast networks.

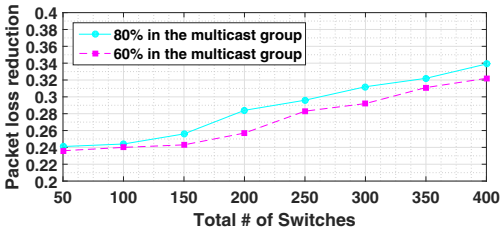
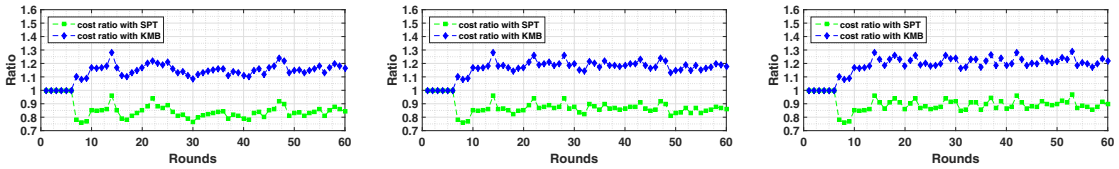


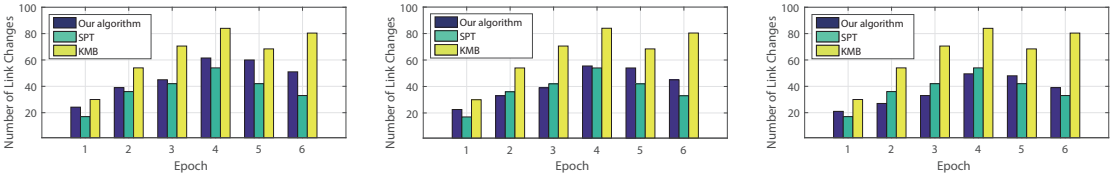
Fig. 11: Packet Loss Reduction

Rule Updating Latency. We also investigate the latencies incurred when updating switch rules in each join/departure event with our loop-free update algorithm, measured as the time difference between when the first rule update OpenFlow message is sent from



(a) $\epsilon = 0.2$ (b) $\epsilon = 0.5$ (c) $\epsilon = 0.8$

Fig. 9: Tree Cost Comparisons with Different ϵ



(a) $\epsilon = 0.2$ (b) $\epsilon = 0.5$ (c) $\epsilon = 0.8$

Fig. 10: Link Change Comparisons with Different ϵ

the controller and when the last ACK for adding link is received. Fig. 12 shows the average rule update latency per round, with the range of latencies in different rounds plotted as well. Considering rounds (join and leave events) are seconds apart, we can see that our rule updates incur small overhead, with updates in most rounds finished in around 1 millisecond.

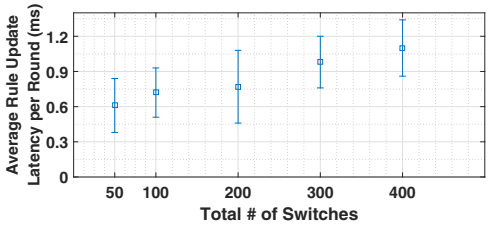


Fig. 12: Rule Update Latency

VIII. Concluding Remarks

We presented a new multicast framework in SDN-enabled mobile networks, based on Steiner trees. Differing from the tradition SPT, the new framework doesn't simply connect receivers via shortest paths to the source. It leverages the global view provided by SDN to approximate the minimum Steiner tree. We design an efficient online algorithm for computing and morphing the Steiner tree in a fully dynamic scenario, where users can join and leave the multicast group any time. A set of multicast rule update procedures is also proposed to ensure that no transient loop occurs. We demonstrated with SDN prototyping that an online Steiner Tree algorithm can be constructed to enable efficient and robust multicast in dynamic networks.

References

[1] "Software Defined Mobile Multicast" Tech. Rep. https://www.dropbox.com/s/55comxkh90wpytq/TechReport.pdf?dl=0.

[2] J. Byrka, F. Grandoni, T. Rothvoß, and L. Sanità. An Improved LP-based Approximation for Steiner Tree. In *Proceedings of the Forty-second ACM Symposium on Theory of Computing*, STOC '10, pages 583–592, New York, NY, USA, 2010. ACM.

[3] S. Deering, D. L. Estrin, D. Farinacci, V. Jacobson, C.-G. Liu, and L. Wei. The PIM Architecture for Wide-area Multicast Routing. *IEEE/ACM Trans. Netw.*, 4(2):153–162, Apr. 1996.

[4] D. Estrin, D. Farinacci, A. Helmy, D. Thaler, S. Deering, M. Handley, V. Jacobson, C. Liu, P. Sharma, and L. Wei. RFC 2362: Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification, June 1998.

[5] O. N. Foundation. Software Defined Networking definition.

[6] A. Gu, A. Gupta, and A. Kumar. The Power of Deferral: Maintaining a Constant-Competitive Steiner Tree Online. *CoRR*, abs/1307.3757, 2013.

[7] A. Gupta and A. Kumar. Online Steiner Tree with Deletions. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '14, pages 455–467. SIAM, 2014.

[8] T. G. Harrison, C. L. Williamson, W. L. Mackrell, and R. B. Bunt. Mobile Multicast (MoM) Protocol: Multicast Support for Mobile Hosts. In *Proceedings of the 3rd Annual ACM/IEEE International Conference on Mobile Computing and Networking*, MobiCom '97, pages 151–160, New York, NY, USA, 1997. ACM.

[9] M. Imase and B. M. Waxman. Dynamic Steiner Tree Problem. *SIAM J. Discrete Math.*, pages 369–384, 1991.

[10] A. Iyer, P. Kumar, and V. Mann. Avalanche: Data center Multicast using software defined networking. In *Communication Systems and Networks (COMSNETS), 2014 Sixth International Conference on*, pages 1–8, Jan 2014.

[11] R. Karp. Reducibility among Combinatorial Problems. In R. Miller, J. Thatcher, and J. Bohlinger, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Springer US, 1972.

[12] M. Karpinski and A. Zelikovsky. New Approximation Algorithms for the Steiner Tree Problems. *Journal of Combinatorial Optimization*, 1:47–65, 1995.

[13] L. T. Kou, G. Markowsky, and L. Berman. A Fast Algorithm for Steiner Trees. *Acta Inf.*, pages 141–145, 1981.

[14] X. Li and M. J. Freedman. Scaling IP Multicast on Datacenter Topologies. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 61–72, New York, NY, USA, 2013. ACM.

[15] C. Marcondes, T. Santos, A. Godoy, C. Viel, and C. Teixeira. CastFlow: Clean-slate multicast approach using in-advance path processing in programmable networks. In *Computers and Communications (ISCC), 2012 IEEE Symposium on*, pages 000094–000101, July 2012.

[16] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[17] N. Megow, M. Skutella, J. Verschae, and A. Wiese. The Power of Recourse for Online MST and TSP. In *Proceedings of the 39th International Colloquium Conference on Automata, Languages, and Programming - Volume Part I*, ICALP'12, pages 689–700, Berlin, Heidelberg, 2012. Springer-Verlag.

[18] K. Mehlhorn. A Faster Approximation Algorithm for the Steiner Problem in Graphs. *Inf. Process. Lett.*, 27(3):125–128, Mar. 1988.

[19] J. Moy. RFC 1584: Multicast Extensions to OSPF, Mar. 1994.

[20] G. Robins and A. Zelikovsky. Tighter Bounds for Graph Steiner Tree Approximation. *SIAM J. Discret. Math.*, 19(1):122–134, May 2005.

[21] D. Waitzman, C. Partridge, and S. E. Deering. RFC 1075: Distance Vector Multicast Routing Protocol, Nov. 1988.

[22] P. Winter. Steiner Problem in Networks: A Survey. *Netw.*, 17(2):129–167, Apr. 1987.

[23] A. Zelikovsky. An 11/6-approximation Algorithm for the Network Steiner Problem. *Algorithmica*, 9(5):463–470, 1993.