2016 IEEE 36th International Conference on Distributed Computing Systems Workshops

Towards Efficient Multicast Communication in Software-Defined Networks

Tim Humernbrum
University of Münster, Germany
Email: t_hume01@uni-muenster.de

Bastian Hagedorn
University of Münster, Germany
Email: b.hagedorn@uni-muenster.de

Sergei Gorlatch
University of Münster, Germany
Email: gorlatch@uni-muenster.de

***Abstract*—The emerging Software-Defined Networks (SDN) offer several advantages over traditional networking technologies. We explore how multicast communication required in many important applications can be organized and implemented efficiently in SDN. Our contributions in this paper are as follows: 1) We propose and implement a simple yet powerful multicast scheme for SDN that gives the sender full control over the members of multicast groups which is absent in traditional IP multicast; 2) We develop algorithms for calculating multicast trees based on Branch-Aware Modification (BAM) and Early Branching (EB) techniques that together allow for maximally reusing unicast flow table entries in SDN switches, thereby improving the scalability of our SDN multicast; 3) We implement our approach as an extendable module within the open-source SDN controller Floodlight. Our approach and its implementation are evaluated using Mininet simulation and experiments in a real OpenFlow-enabled network.**

I. Introduction

Multicast communication is useful for many classes of networked applications, especially when the same data has to be sent frequently to several receivers, such as in video streaming or distributed datacenter applications. In traditional networks, the logic of packet forwarding is distributed across the network components like switches and routers, with only limited possibilities for a centralized control over forwarding decisions. Hence, current standards and algorithms for multicast mostly work in a distributed, receiver-initiated manner: receivers must know the multicast address of a desired service in advance and they explicitly subscribe to the corresponding multicast group at specific routers by using the IGMP protocol [1]. This design of the IP multicast is not only intransparent for the receivers but it has also other drawbacks for important applications like teleconferencing, file transfer, etc.: a) the sender has no control over the members of a multicast group, e. g., it cannot exclude a certain receiver from the multicast communication; b) computation-intensive approaches like Steiner trees are difficult to compute in a distributed manner [2] and, thus, are avoided in current multicast standards, e. g., PIM-SM [3], which mostly use simpler but less scalable *Shortest-Path Trees (SPT)*.

Software-defined Networking (SDN) is an emerging network architecture in which a central control instance, the so-called *SDN controller*, takes over the forwarding logic of the underlying network components by installing flow table entries in them through interfaces like OpenFlow [4]. While also the traditional, IP-based multicast using IGMP can be implemented in SDN (e. g., as described in [5]), a major bottleneck for both, unicast and multicast communication in SDN, is the limited capacity of flow tables as illustrated in [6]. Since using multicast incurs additional flow table entries besides unicast entries, this bottleneck becomes a major hurdle for the efficient utilization of multicast in SDN.

Previous work has addressed possibilities to improve the scalability of multicast (i. e., to increase the number and size of multicast groups that can be accommodated in the network) by efficiently using the available flow table space. Kanzoi et al. [6] introduce a framework for the distribution of flow table entries. Li et al. [7] demonstrate how flow tables can be compressed using hash functions. Another approach to the scalable SDN multicast is to minimize the number of switches and/or the number of needed multicast entries in the switches' flow tables by utilizing advanced multicast trees. Yang et al. [8] introduce so-called *branch forwarding* to minimize flow table entries. This technique is applied by Huang et al. [9] in the *Branch-aware Steiner Tree (BST)*. However, their approach introduces an additional overhead by relying on IP tunneling [10].

The contributions and structure of this paper are as follows:

1) We propose a simple yet powerful scheme for SDN multicast which overcomes the drawbacks of traditional IP multicast by exploiting the centralized architecture of SDN. Our approach is sender-initiated and transparent for the receiver, i. e., the sender has full control over the receiving endpoints and no action is required from the receiver to participate in multicast communication (Section II).
2) We develop algorithms for calculating multicast trees that allow for maximally reusing unicast flow table entries for multicast, thereby improving the scalability of SDN multicast without IP tunneling (Section III).
3) We implement our approach as an extendable module within the open-source SDN controller Floodlight [11] and we develop a C++ library for the application-side management of multicast groups (Section IV).

Section V concludes the paper with an experimental evaluation of our multicast approach in both a Mininet-simulated network environment and a real, SDN-enabled testbed.

2332-5666/16 $31.00 © 2016 IEEE
DOI 10.1109/ICDCSW.2016.15

II. SDN-based Multicast

Figure 1 illustrates the idea of our SDN multicast scheme. If Host 1 *(sender)* is going to send data to the hosts 2 and 3 *(receivers)* using multicast, it specifies a *multicast group* comprising the IP addresses of the receivers (step ① in Figure 1). The addresses are then sent to the SDN controller (step ②) via its so-called *Northbound API*. Subsequently, in step ③, the controller defines a new multicast address for this group (20.0.0.1 in Figure 1) and *installs* this multicast group in the network by adding a new entry to the flow table of each switch on the routes to the receivers (Switch A and Switch B in the figure). Which routes are chosen by the SDN controller to forward multicast packets to the members of a multicast group is calculated in form of a *multicast tree* which we discuss in Section III. After installing the new multicast group in the network, the controller returns the multicast address of the group to the sender (step ④). The sender can then send data (packet `d` in step ⑤) to the members of the multicast group by using this address. When a packet sent to the multicast address arrives at Switch A, it is matched to the corresponding entry in A's flow table and is forwarded to Switch B. Switch B then transforms the packet's header: the multicast address in the packet is replaced by the unicast address of the corresponding receiver (UDP ports and MAC addresses are omitted for simplicity in Figure 1). Finally, Switch B forwards the packet to the ports on which the receivers are connected to B (step ⑥ with transformed data packets `d*` and `d+`).

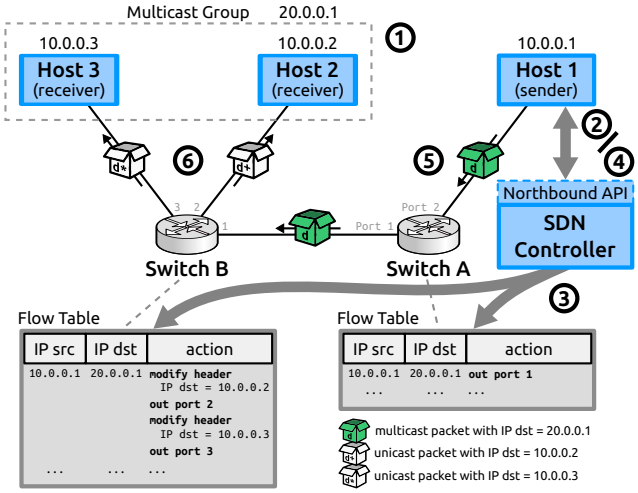


Fig. 1. Our SDN-based multicast scheme

In our multicast scheme in Figure 1, when the transformed packets sent to a multicast address finally arrive at the receiver, they look like normal, directly addressed unicast packets because the multicast address is replaced by the receiver's IP address as described above. Thereby, our multicast scheme overcomes the drawbacks of the traditional IP multicast in which receivers must subscribe for a multicast group at specific multicast routers and potentially every member of the group can send data to the multicast address, which is usually not desirable. Indeed, our multicast approach is transparent for receivers: they cannot differentiate multicast packets from unicast packets, and the multicast is initiated by the sender – no action is required from the receivers to participate in multicast communication. At the same time, the sender has full control over the receiving endpoints, i. e., it can add or remove receivers to or from a multicast group at any time. Furthermore, only the sender that has specified a multicast group is able to send data to the corresponding multicast address and reach the members of this group, because the sender's IP address is referenced in the switches' flow table entries. If a network node other than the sender of a multicast group tries to send data to the group's multicast address, the packets will not match the flow table entries and will be dropped or forwarded to the controller (e. g., to trigger access violation routines) depending on the controller's configuration.

Our SDN-based multicast utilizes the OpenFlow protocol [12] for the communication between the controller and switches, which is the de-facto standard for SDN. Every switch must support OpenFlow's optional action for the transformation of packet headers in step ⑥ in Figure 1.

We will show in Section III that, in contrast to Figure 1 (which illustrates the general concept of our multicast approach for SDN), it is preferable that the transformation of multicast packets into unicast packets takes place near to the root (sender) of the multicast tree, in order to reuse unicast entries which might already exist in the switches' flow tables. For instance, this transformation could be performed in Figure 1 by Switch A instead of Switch B, in order to reuse the unicast entries in B's flow table. Even if such unicast entries must be newly created for the multicast, they can be used afterwards for unicast and multicast communication between the sender and a specific receiver. Our experimental evaluation in Section V shows that this reuse by far compensates the fact that flow table entries for the transformation of packet headers are larger than "normal" forwarding entries and, therefore, consume more space. The size of such a transformation entry depends on the number of receivers in a multicast group for which multicast packets have to be transformed into unicast packets: for every receiver, its IP and MAC addresses as well as the corresponding output port have to be saved in the flow table. In Figure 1, the transformation entry in Switch B's flow table comprises the destination addresses of hosts 2 and 3 as well as the output ports 2 and 3 on which the hosts are connected to the switch.

Minimizing flow table entries by shifting the transformation closer to the root of a multicast tree increases the overall bandwidth utilization of the network, which contradicts the very reason to use multicast for group communication. Therefore, we introduce an algorithm in Section III-A which performs such shifting in a way which preserves the bandwidth utilization. Additionally, we introduce a multicast tree with properties which pose a reasonable compromise between minimizing flow table entries on the one hand and reducing bandwidth utilization on the other hand.

III. Multicast Trees

In traditional IP multicast, the calculation of multicast trees is based on distributed algorithms which incur a communication overhead and have to suit the rather limited computational capacity of switches. Hence, complex algorithms for constructing advanced multicast trees (e. g., Steiner trees [2]) are often avoided in traditional IP multicast and rather Shortest-Path Trees (SPT) are used which limits the number and size of multicast groups that can be accommodated in the network. In our SDN multicast scheme, multicast trees are calculated centrally by the SDN controller which has a global view over the network topology and is usually deployed on a powerful server machine. Hence, there is no communication overhead and the computational expense for calculating multicast trees is no more a limiting factor in our approach; therefore, we concentrate on another potential limiting factor – the capacity of the switches' flow tables. For instance, common OpenFlow switches like Dell PowerConnect 8132F, HP ProCurve 5406zl, and Pica8 P-3290 can store about 750, 1500, and 2000 entries, respectively [13].

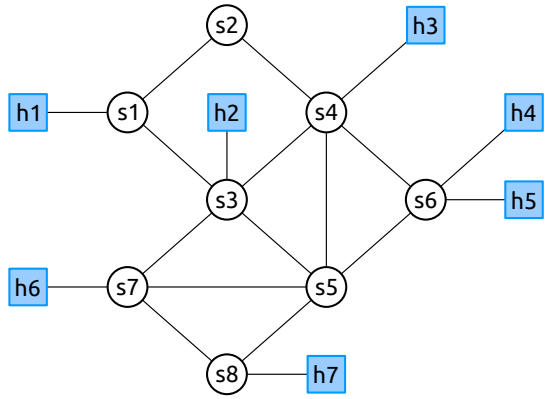


Fig. 2. Example network topology for a multicast scenario

As our running example, we will consider the topology depicted in Figure 2 to illustrate the algorithms for calculating multicast trees as subgraphs of this topology. In the figure, square nodes represent hosts while circle nodes represent switches, e. g., *h1* denotes Host 1, and *s1* denotes Switch 1. We further differentiate between two types of switch nodes in a multicast tree: a) *unicast nodes* store unicast entries matching against unicast IP addresses only; b) *multicast nodes* store multicast entries matching against multicast IP addresses and, thus, these nodes can forward multicast packets or transform them into unicast packets. According to this definition, both switches A and B in Figure 1 are multicast nodes, because their flow tables each comprise an entry for matching packets against the multicast address 20.0.0.1. However, if the transformation into unicast packets would take place in Switch A, then Switch B would become a unicast node that forwards the transformed unicast packets `d*` and `d+` to the corresponding receivers.

A. Branch-aware Modification

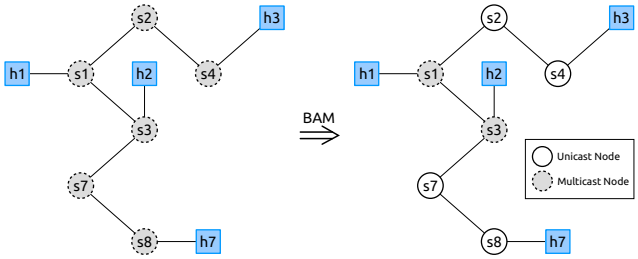
Our approach to improve multicast scalability (i. e., to increase the number and/or size of multicast groups despite the limited capacity of flow tables) is to reduce the number of multicast switch nodes in an arbitrary multicast tree by applying the so-called Branch-aware Modification (BAM) to this tree. BAM takes a tree that contains only multicast nodes and it identifies nodes between a receiver and its nearest branch node (i. e., a node with more than two edges to the adjacent switch and/or host nodes) as potential unicast nodes.

Algorithm 1 BAM: identifying unicast nodes

```
for all v ∈ R do
    u ← get_parent(v)
    while node_degree(u) ≤ 2 & u ≠ root do
        mark u as unicast_node
        u ← get_parent(u)
    end while
end for
```

Algorithm 1 shows the pseudocode for the Branch-aware Modification. R denotes the set of receiving hosts in the input tree (the leaves of the tree), *root* is the switch node directly connected to the sender. Although our algorithm uses a nested loop, its computational complexity is only $\mathcal{O}(|V|)$, i. e., linear in the number of network nodes, because every node is visited not more than once; depending on the topology there may be nodes which are not visited at all.

We usually take Shortest-Path Trees (SPT) as input for BAM, because they are most widely used in multicast, but the BAM algorithm is applicable to an arbitrary tree. Trees in which all nodes identified by the BAM are modified into unicast nodes gain the prefix Branch-aware: for example, a BAM-modified SPT is called Branch-aware SPT or BSPT.



(a) SPT (b) BSPT

Fig. 3. Example of the Branch-aware Modification

Figure 3 (a) depicts an example SPT for the topology of Figure 2, with *h1* as sender and *h2, h3* and *h7* as receivers; all switch nodes in this tree are multicast nodes. Applying the BAM algorithm to this SPT results in the BSPT depicted in Figure 3 (b). Here, *s3* is the first branch node on the path from the receivers *h2* and *h7* to the sender *h1*. Therefore, all nodes on the paths from *s3* to *h2* and *h7* are modified to unicast nodes. The same applies to the nodes on the path from *h3*

to *s1*. Summarizing, the BAM algorithm modifies *s2, s4, s7* and *s8* to unicast nodes. The resulting multicast tree contains only two nodes which need to store multicast entries while the original SPT contains six multicast switches.

At the time of the multicast group creation as described in Section II, unicast entries may already exist in some switches due to former unicast communication in the network; in this case, they will be simply re-used after BAM. If no unicast entries are present, then our approach installs them instead of multicast entries; the advantage is that they can be reused afterwards in both multicast and unicast communication, whereas multicast entries can only be used for a specific multicast group and always need to be explicitly installed for every new multicast tree.

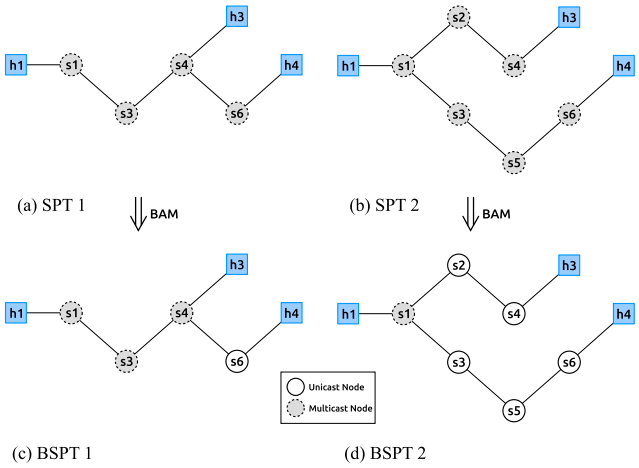


Fig. 4. Comparison of two branch-aware SPTs

The BAM procedure is obviously most advantageous for trees which have long paths connecting single receivers with the sender. In this case, branching happens early (near to the sender), such that only few multicast nodes are needed. For example, Figure 4 depicts two different SPTs, (a) and (b), for a multicast group with *h1* as sender and *h3* and *h4* as receivers. The BAM (Algorithm 1) applied to these trees produces output trees shown in Figure 4 (c) and (d), correspondingly. We observe that the "early branching" tree SPT 2 in Figure 4 (b) greatly benefits from BAM: the number of multicast nodes is reduced from 6 to 1. The "late branching" tree in Figure 4 (a) also benefits from BAM, but on a smaller scale: the number of multicast nodes decreases from 4 to 3. Note that BAM always identifies some nodes as unicast nodes, unless every receiver is directly connected to a branch node as for example in a full binary tree. The BSPT in Figure 4 (d) requires more bandwidth of the whole network than the BSPT depicted in (c). In scenarios where bandwidth is a limiting factor, the BSPT depicted in (c) is favorable. However, to achieve scalability in the context of SDN multicast, one should consider trees with the "early branching" property.

If a multicast group contains only one receiver, then the multicast tree does not branch. In this very special case, every node except the root is marked as a unicast node by the BAM algorithm. The sender then sends its data via multicast to the root switch which directly transforms the multicast packets to unicast packets, i. e., the only multicast entry in the whole tree is installed in the flow table of the root.

B. Early Branching Multicast Tree

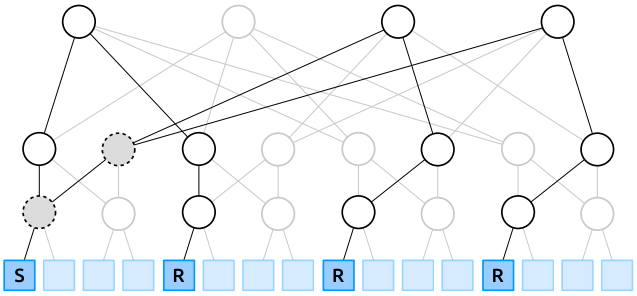
In this section, we develop an algorithm which, for a given network topology graph, finds an SPT with the early branching property that is especially advantageous for applying BAM. For a given topology graph, a given sender and a set of multicast receivers, we define the *Early Branching Shortest Path Tree (EBSPT)* as the SPT of this graph in which the branch node nearest to the root has a minimal distance to the root among all SPTs. For calculating this tree, we modify the algorithm originally proposed in [14] for finding a Destination Driven Shortest Path Tree (DDSPT). The DDSPT is exactly the opposite of the EBSPT – it branches as far as possible from the sender node.

Figure 4 (a) illustrates the DDSPT (equal to SPT 1 in the figure) as a subgraph of the topology in Figure 2 for the multicast group with *h1* as sender and *h3* and *h4* as receivers. This tree branches quite late because both receivers share the path *h1-s1-s3-s4*. Applying the BAM to this DDSPT only reduces the number of multicast nodes by one as depicted in Figure 4 (c). Figure 4 (b) shows the EBSPT for the same multicast group. As already discussed, applying the BAM to this EBSPT significantly reduces the number of multicast nodes as depicted in Figure 4 (d).

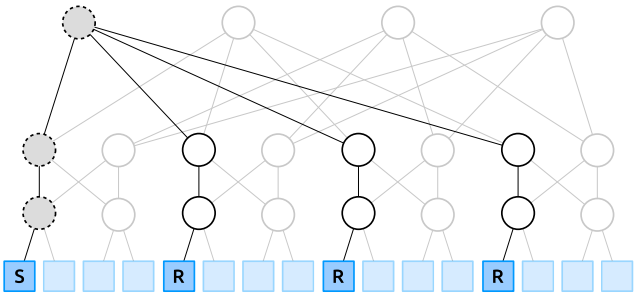
Algorithm 2 Computation of an EBSPT

$T \leftarrow \emptyset$
$R \leftarrow$ set of receivers
while $\exists r \in R : r \notin T$ **do**
  $U \leftarrow \{u \in R \setminus T : min\{\text{distance\_to\_sender}(u)\}\}$
  $v \leftarrow v \in U$
  if $|U| > 1$ **then**
    $v \leftarrow \{u \in U : max\{\text{distance\_to\_a\_recv\_in\_T}(u)\}\}$
  end if
  $T \leftarrow T \cup path(sender, v)$
end while

The pseudocode to calculate the EBSPT is shown in Algorithm 2. This greedy algorithm is similar to the Dijkstra algorithm for calculating an SPT. It uses two criteria to choose a node at each step: the distance to the sender ($distance\_to\_sender$) and the distance to the nearest receiver that is already connected to the tree ($distance\_to\_a\_recv\_in\_T$). If two receivers have an equal distance to the sender, then the original DDSPT algorithm [14] would choose the receiver with the minimal second criterion. To calculate the EBSPT, we choose the receiver with the maximal second criterion: this causes the tree to branch early.



(a) BEBSPT



(b) BDDSPT

Fig. 5. BEBSPT and BDDSPT in a Fat-Tree topology

Our modification of DDSPT does not change the computational complexity of the algorithm which, as shown in [14], is $\mathcal{O}(|E| \cdot log|V|)$ where $|E|$ denotes the number of edges and $|V|$ is the number of nodes.

As a final example we will analyze the effectiveness of early branching and branch awareness on the *Fat-Tree* topology which is commonly used in datacenters, industrial, and campus networks [15]. Figure 5 depicts a k-ary Fat-Tree (in our case $k = 4$) in which switches are split in horizontal levels, from top to bottom: core, aggregation, and edge level. Let us consider for this topology a multicast group with three receivers (denoted by R) and a sender denoted by S. Figure 5 (a) depicts the BEBSPT and Figure 5 (b) depicts the BDDSPT for this multicast group. As explained before, these trees are built in two steps: 1) by executing the EBSPT or the DDSPT algorithm, correspondingly – in the resulting trees, all switch nodes will be multicast nodes; and then 2) by applying the BAM transformation to the EBSPT or DDSPT tree. We observe in Figure 5 that the BAM transformation is beneficial in both cases: applying BAM to the DDSPT reduces the amount of multicast nodes by 66 % as in Figure 5 (b), while applying BAM to the EBSPT as in Figure 5 (a) is even more beneficial: the total number of multicast nodes is reduced by 83 % from 12 to only 2 multicast nodes.

IV. Implementation

For implementing the SDN-based multicast introduced in Section II and the algorithms for multicast tree calculation introduced in Section III, only standard OpenFlow features

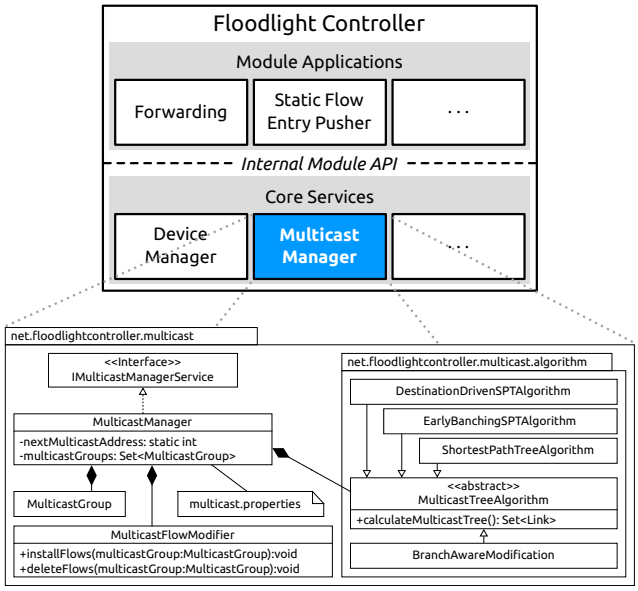


Fig. 6. The `MulticastManager` module in the Floodlight Controller

are required. Therefore, most modern SDN controllers and platforms can be easily extended to support our multicast scheme. We decided to use the open-source SDN controller Floodlight [11] which is widely used and benefits from an active developer community. In its standard configuration, Floodlight supports neither IPv4 multicast nor SDN-based multicast; there exist third-party modules enabling multicast with IGMP [16] [17].

We extended Floodlight by adding to the controller a new module called `MulticastManager`. Figure 6 (upper half) shows how the `MulticastManager` is integrated in the Floodlight architecture as an additional core service that can be used like any other module of Floodlight. It uses some of Floodlight's module applications through an internal API, e. g., the Static Flow Pusher [11] to install flow table entries.

Figure 6 (lower half) shows the implementation of our `MulticastManager` module in a modular, extendable manner. An arbitrary multicast tree algorithm can be added to the module to replace or extend the existing algorithms by implementing the abstract class `MulticastTreeAlgorithm`. The network administrator specifies which particular algorithm should be used via a configuration file (`multicast.properties`). For example, the entry `algorithm=BEBSPT` specifies that the Branch-aware EBSPT should be used; this would lead to the execution of Algorithm 2 followed by Algorithm 1, both presented in the previous section. Internally, this is realized by providing an EBSPT as input to the BAM algorithm.

To utilize our multicast scheme in networked applications written in C++, we develop a C++ library which we call *SDN Module*. Listing 1 demonstrates an example: the usage of the SDN Module by an application developer for the specification of the example multicast group shown in Figure 1. In lines 4–8 of Listing 1, the SDN Module is

activated with the address and port of the SDN controller. The controller type is set to `REST` [18], so that the application communicates with the controller via its REST-based Northbound API as, e. g., required by Floodlight. After activating the SDN Module, a new multicast group is created in lines 11–13, and hosts 2 and 3 are added to this group. The SDN Module provides a callback mechanism, e. g., for receiving a multicast address from the controller when multicast has been installed in the network. To use this mechanism, an implementation of the `MulticastListener` interface is registered for the multicast group as shown in line 16. Finally, in line 17, multicast for the group is activated by calling function `activateMulticast`. This triggers the communication between the application and the controller and transfers the addresses of the multicast group to the controller as in step ② of Figure 1. Subsequently, the controller calculates a multicast tree for the group and adds the corresponding flow table entries in the involved switches (step ③).

```
1  using namespace sdn;
2
3  // initialize SDN Module
4  SDNModuleProperties props{};
5  props.controller = Endpoint("10.0.0.254", 6300);
6  props.controller_type = ControllerType::REST;
7  SDNModule sdn_mod{};
8  sdn_mod.activate(props);
9
10 // specify multicast group
11 auto& grp = sdn_mod.createMulticastGroup();
12 grp.addDstEndpoint(Endpoint{"10.0.0.2", 1234});
13 grp.addDstEndpoint(Endpoint{"10.0.0.3", 1234});
14
15 // register listener an activate multicast
16 grp.registerListener(listener);
17 grp.activateMulticast();
```

Listing 1. Example usage of the SDN Module

V. Experimental Evaluation

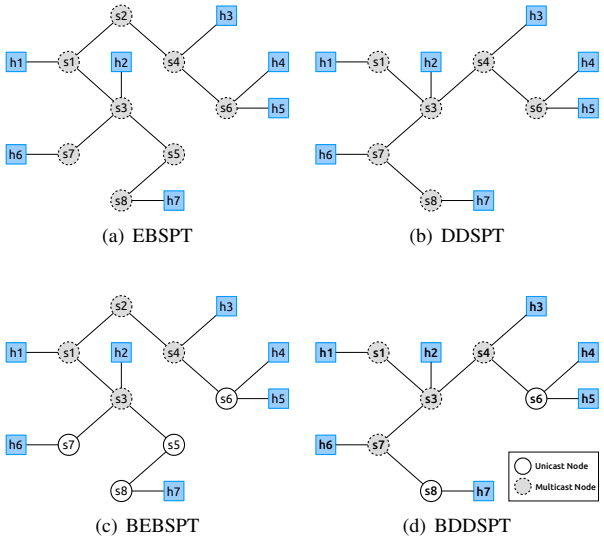
We evaluate our SDN multicast approach, the suggested algorithms for building multicast trees, and their implementation in the Floodlight controller, on two evaluation platforms:

- the network emulation software Mininet [19] which creates a virtual SDN-enabled network on a single computer;
- the Network Laboratory *NetLab* at the University of Münster which is an SDN network with hosts connected via OpenFlow-enabled switches.

To generate network load in SDN multicast experiments, we transfer 100 MB of random data via UDP from the sender to the receivers. All results regarding time and CPU usage are the arithmetic average of five experiment cycles (the observed differences across several runs did not exceed 5 %).

A. Mininet Simulation

The example topology depicted in Figure 2 is emulated in Mininet to evaluate the behavior of different multicast trees in SDN-enabled networks. The evaluated multicast trees are depicted in Figure 7. Mininet emulates OpenFlow-enabled switches which we connect to the Floodlight controller extended by our multicast module (Section IV) and installed on a separate machine. In our experiments, *h1* is the sender and *h2–h7* are the six receivers of the multicast group.



(a) EBSPT (b) DDSPT

(c) BEBSPT (d) BDDSPT

Fig. 7. Multicast trees evaluated in Mininet

Since flow tables are a potential bottleneck in SDN, we measure the number of required flow table entries per multicast tree. We differentiate between unicast and multicast entries and we consider trees with fewer multicast entries superior to trees which require more such entries, because fewer multicast entries mean that more unicast entries can be reused afterwards for either multicast or unicast communication whereas multicast entries are not reusable.

Figure 8 shows the number of required multicast (black) and unicast (gray) flow table entries for the multicast trees depicted in Figure 7. We observe that DDSPT minimizes the total number of flow table entries (6 entries instead of 20 as compared to unicast) while BEBSPT minimizes the number of required multicast entries (4 entries instead of 8 as compared to the EBSPT).

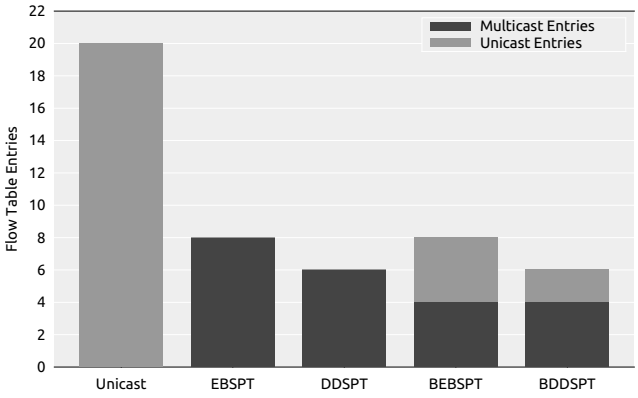


Fig. 8. Mininet: number of flow table entries for the trees of Fig. 7

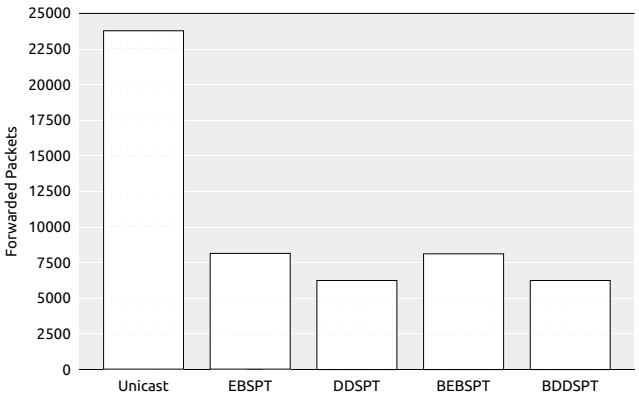


Fig. 9. Mininet: number of forwarded packets for the trees of Fig. 7

Figure 9 demonstrates that multicast significantly reduces the bandwidth utilization (number of forwarded packets) as compared to unicast which generates much higher load because the same data has to pass the same switches multiple times. We observe that the bandwidth utilization is not increased when applying the BAM to multicast trees (e. g., BDDSPT vs. DDSPT, and BEBSPT vs. EBSPT) because every packet passes a switch only once. Although the EBSPT maximizes the amount of edges in the multicast tree (which allows to reduce the number of multicast entries), the bandwidth utilization is still reduced by $\sim 66\,\%$ as compared to unicast. As expected, the DDSPT reduces the bandwidth utilization stronger than the EBSPT ($\sim 25\,\%$) because it uses fewer edges in the multicast tree.

In Figure 10, we evaluate the difference between BDDSPT and BEBSPT by examining the scenario of Figure 4: *h1* creates a multicast group with *h3* and *h4* as members. The flow table entry requirements of the BDDSPT (in Figure 4 (c)) and the BEBSPT (in Figure 4 (d)) confirm that the DDSPT decreases the total number of flow table entries while the BEBSPT decreases the number of multicast entries.

Figure 11 shows experiments on the Mininet-emulated topology of the pan-European OFELIA testbed [20] with OpenFlow islands in Zurich, Ghent, and Trento. Figure 12 shows the BEBSPT calculated for a multicast group with host *S* as sender and all other hosts as receivers. As illustrated

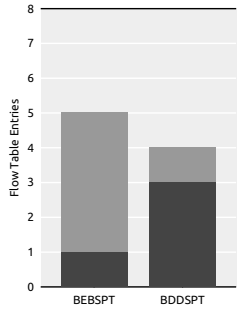


Fig. 10. Number of flow table entries of BEBSPT and BDDSPT from Fig. 4

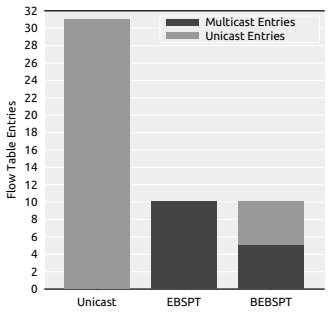


Fig. 11. Comparing unicast and BEBSPT in the OFELIA topology from Fig. 12

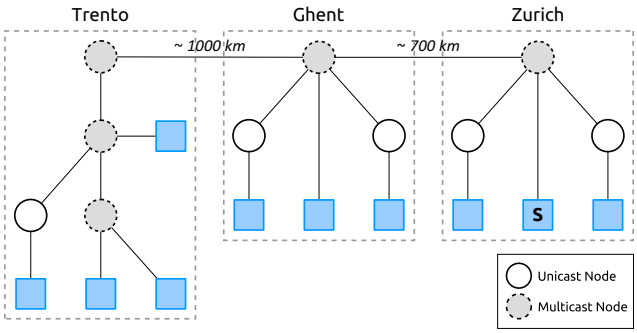


Fig. 12. BEBSPT for the pan-European SDN network OFELIA

in Figure 11, the total number of required flow table entries has been reduced by 66 % by using multicast with EBSPT instead of unicast. The number of required multicast entries is even further reduced by 50 % when BEBSPT is employed.

B. The NetLab OpenFlow Network

Figure 13 shows the NetLab's network topology that consists of three OpenFlow-enabled software switches *s1–s3* using Open vSwitch 1.10.2, each running on an Intel Xeon E3-1240v2 server machine with Intel Gigabit ET dual port server adapters. In our experiments, the receivers *h2–h4* run on identically equipped machines while the sender *h1* uses a machine with two Intel Xeon E5-2620 processors and 26 GB RAM. The traffic is forwarded with a link speed of 1 Gbit/s.

For this very simple topology, the implemented algorithms (EBSPT, DDSPT), including the branch-aware versions of both, compute the same multicast tree depicted in Figure 13. The BAM version only transforms multicast packets to unicast packets at *s3* which does not change the results regarding time and CPU utilization. Because of this, we only evaluate our SDN multicast scheme and its implementation without considering different multicast trees.

Figures 14 and 15 show the results of the experiment in the NetLab: to send 100 MB from *h1* to all three receivers using links with 1 GB/s bandwidth, multicast takes about 30 % of the time needed by unicast, while the CPU load of multicast is also slightly lower.

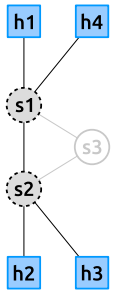


Fig. 13. The Network Laboratory *NetLab*

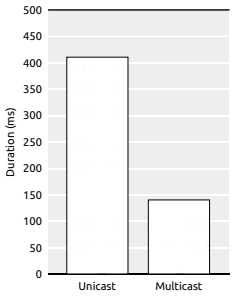


Fig. 14. NetLab: time required to send data

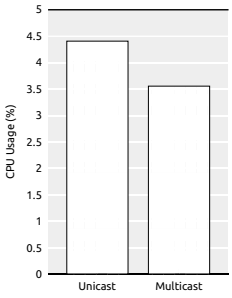


Fig. 15. NetLab: CPU load of sender *h1*

VI. Conclusion

Our contributions in this paper are three-fold. Firstly, we propose a simple yet powerful multicast scheme for SDN that exploits its controller-centered architecture to overcome the drawbacks of the traditional IP multicast – the lack of multicast group membership control for the sender and limited scalability. Secondly, we develop an approach for calculating multicast trees based on Branch-Aware Modification (BAM) and Early Branching (EB) techniques to improve the scalability of the SDN multicast. Thirdly, we implement the multicast scheme for SDN and the algorithms for calculating multicast trees in an extendable module for the SDN controller Floodlight and we evaluate our approach in both, simulated networks using Mininet and in a real, OpenFlow-enabled network.

The key feature of our EBSPT multicast tree is that it comprises long paths connecting the sender to single receivers while avoiding paths shared by several receivers. This causes the tree to branch early which is advantageous for applying BAM afterwards. Such trees have not been previously considered for the traditional IP multicast, because they may potentially increase the bandwidth utilization in the network, since the number of edges in EBSPT is maximized. Our evaluation in Section V shows that, for SDN multicast, the increase in bandwidth utilization is negligible as compared to the advantages of reducing the required number of flow table entries.

References

[1] B. Cain, S. Deering, I. Kouvelas *et al.*, "Internet Group Management Protocol, Version 3," *IETF RFC 3376*, October 2002.

[2] F. K. Hwang, D. S. Richards, and P. Winter, *The Steiner Tree Problem*, ser. Annals of Discrete Mathematics. North-Holland, 1992.

[3] D. Estrin, D. Farinacci, A. Helmy *et al.*, "Protocol Independent Multicast-Sparse Mode (PIM-SM): Protocol Specification," *IETF RFC 2362*, June 1998.

[4] N. McKeown, T. Anderson, H. Balakrishnan *et al.*, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[5] A. Iyer, P. Kumar, and V. Mann, "Avalanche: Data Center Multicast using Software Defined Networking," in *Sixth International Conference on Communication Systems and Networks (COMSNETS), 2014*, Jan 2014, pp. 1–8.

[6] Y. Kanizo, D. Hay, and I. Keslassy, "Palette: Distributing Tables in Software-Defined Networks," in *IEEE Proceedings of INFOCOM*, 2013, pp. 545–549.

[7] D. Li, H. Cui, Y. Hu *et al.*, "Scalable Data Center Multicast using Multi-Class Bloom Filter," in *19th IEEE International Conference on Network Protocols (ICNP)*, Oct 2011, pp. 266–275.

[8] D.-N. Yang and W. Liao, "Protocol Design for Scalable and Adaptive Multicast for Group Communications," in *IEEE International Conference on Network Protocols (ICNP 2008)*, Oct 2008, pp. 33–42.

[9] L.-H. Huang, H.-J. Hung, C.-C. Lin *et al.*, "Scalable Steiner Tree for Multicast Communications in Software-Defined Networking," *ArXiv e-prints*, 2014.

[10] W. Simpson, "IP in IP Tunneling," Internet Requests for Comments, RFC 1853, 1995. [Online]. Available: https://tools.ietf.org/html/rfc1853

[11] (2015) Floodlight OpenFlow Controller. [Online]. Available: http://www.projectfloodlight.org

[12] (2012) OpenFlow Switch Specification Version 1.3.0 (Wire Protocol 0x04). [Online]. Available: https://www.opennetworking.org/sdn-resources/technical-library

[13] M. Kuźniar, P. Perešíni, and D. Kostić, "What You Need to Know About SDN Flow Tables," in *Passive and Active Measurement*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2015, vol. 8995, pp. 347–359.

[14] B. Zhang and H. T. Mouftah, "Destination-driven shortest path tree algorithms," in *Journal of High Speed Networks*, Jan 2006, pp. 123–130.

[15] M. Al-Fares, A. Loukissas, and A. Vahdat, "A Scalable, Commodity Data Center Network Architecture," in *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, ser. SIGCOMM '08. ACM, 2008, pp. 63–74. [Online]. Available: http://doi.acm.org/10.1145/1402958.1402967

[16] (2012) Suzuki Kazuya, simple-multicast.rb. [Online]. Available: https://github.com/daisuke-k/trema-apps/blob/master/simple_multicast/simple-multicast.rb

[17] (2013) Ito Yuichi, simple_switch_igmp.py. [Online]. Available: https://github.com/osrg/ryu/blob/master/ryu/app/simple_switch_igmp.py

[18] R. T. Fielding and R. N. Taylor, "Principled Design of the Modern Web Architecture," vol. 2, no. 2. New York, NY, USA: ACM, May 2002, pp. 115–150.

[19] B. Lantz, B. Heller, and N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX. New York, NY, USA: ACM, 2010, pp. 19:1–19:6.

[20] (2014) OFELIA: OpenFlow in Europe: Linking Infrastructure and Applications. [Online]. Available: http://www.fp7-ofelia.eu