Received XX Month, XXXX; revised XX Month, XXXX; accepted XX Month, XXXX; Date of publication XX Month, XXXX; date of current version 11 January, 2024.

Digital Object Identifier 10.1109/OJCOMS.2024.011100

Segment-Encoded Explicit Trees (SEETs) for Stateless Multicast: P4-Based Implementation and Performance Study

Steffen Lindner[1], Thomas Stüber[1], Maximilian Bertsch[1], Toerless Eckert [2], and Michael Menth[1] ***(Senior Member, IEEE)***

[1]University of Tuebingen, Chair of Communication Networks, 72076 Tuebingen, Germany
[2]Futurewei Technologies, CA 95050, United States

CORRESPONDING AUTHOR: Michael Menth (e-mail: menth@uni-tuebingen.de).

The authors acknowledge the funding by the Deutsche Forschungsgemeinschaft (DFG) under grant ME2727/3-1 and by the Open Access Publishing Fund of the University of Tübingen, Germany. The authors alone are responsible for the content of the paper.

ABSTRACT IP multicast (IPMC) is used to efficiently distribute one-to-many traffic within networks. It requires per-group state in core nodes and results in large signaling overhead when multicast groups change. Bit Index Explicit Replication (BIER) and its tree engineering variant BIER-TE have been introduced as a stateless transport mechanism for IPMC. To utilize BIER or BIER-TE in a large domain, domains need to be subdivided into smaller sets of receivers or smaller connected subdomains, respectively. Sending traffic to receivers in different sets or subdomains necessarily implies sending multiple packets. While efficient algorithms exist to compute sets for BIER, algorithms for computing BIER-TE subdomains are still missing. In this paper, we present a novel stateless tree encoding mechanism called Segment-Encoded Explicit Tree (SEET). It encodes an explicit multicast distribution tree within a packet header so that tree engineering is supported and sets or subdomains are not needed for large domains. SEET is designed to be implementable on low-cost switching ASICs which we underline by a prototype for the Intel Tofino™. If explicit distribution trees are too large to be accommodated within a single header, multiple packets with different distribution trees are sent. For this purpose, we suggest an effective optimization heuristic. A comprehensive study compares the number of sent packets and resulting overall traffic for SEET and BIER in large domains. In our experiments, SEET outperforms BIER even for large multicast groups with up to 1024 receivers.

INDEX TERMS Segment-Encoded Explicit Trees (SEETs), Bit Index Explicit Replication (BIER), multicast, IP networks, performance evaluation, optimization

I. Introduction

IP multicast (IPMC) [1] is the default multicast service in IP networks and is used to reduce the traffic load of one-to-many traffic. Examples for IPMC services are Multicast VPN, streaming, content delivery networks, or financial stock exchange [2]. Receivers of multicast services are organized in multicast groups that are identified by unique IP addresses. Traffic is forwarded along a multicast distribution tree to all receivers of the multicast group. Thereby, only one packet is sent over each involved link. However, IPMC has two scalability issues. First, all forwarding nodes of the distribution tree need to maintain the forwarding state of the corresponding multicast groups, such as the list of links to which to copy packets for the group. In networks with large number of IP multicast groups, this leads to an equal large number of multicast forwarding states, which are expensive to ASICs. Second, when receivers of a multicast group change, or the network topology changes, the forwarding nodes need to update their forwarding state, which can result in excessive signaling overhead. Multicast mechanisms that rely on this kind of dynamic state are referred to as *stateful multicast* mechanisms. The Internet Engineering Task Force (IETF) is currently standardizing Bit Index Explicit Replication (BIER) [3] as a *stateless multicast* transport mechanism for IPMC traffic. BIER forwards IPMC traffic through a BIER domain without the need for dynamic forwarding state in core nodes. They leverage a so-called BIER bitstring that is added to the packets by ingress nodes of the domain. Each bit identifies an egress node of the BIER domain, i.e., a possible receiver of the multicast group. If a bit is set, the corresponding egress node requires a packet copy. Based on this bitstring, core nodes of the BIER domain are able to forward the traffic to the appropriate egress nodes. However, when BIER is scaled to networks with more egress nodes than bits permitted in the BIER bitstring, the receivers need to be split up into multiple sets of bitstrings. The packet header needs to carry a bitstring and a set identifier, and multiple copies of a multicast packet (with different set identifiers) might be forwarded over the same link, which mitigates the advantage of BIER over unicast forwarding. This is especially problematic for sparse multicast trees, i.e., multicast trees with a small number of receivers, in large networks. In that case, the ratio between receivers and redundant packets worsens.

BIER forwards traffic according to a so-called routing underlay, e.g., an IP network, and is not able to steer traffic on explicit paths. For that purpose, tree engineering for BIER (BIER-TE) has been designed [4]. Tree engineering is the capability of supporting explicit paths for stateless forwarding trees. The term has been introduced in [4] to distinguish this capability from traffic engineering. Traffic engineering in contrast also includes mechanisms like bandwidth reservation or algorithms to calculate the paths on which traffic will be forwarded. BIER-TE suffers from similar scaling issues as BIER.

The contributions of this paper are manifold. First, we show that BIER's scaling mechanism results in many redundant packet copies in large networks. Second, we present a novel mechanism for stateless multicast called Segment-Encoded Explicit Tree (SEET). It combines ideas of Segment Routing (SR) [5] and BIER [6], and supports tree engineering. Third, we present a P4-based implementation of SEET for the Intel Tofino™ ASIC. SEET encodes the complete distribution tree within a packet header. If the required SEET header is too large to be processed by high-speed switching ASICs, multiple packets with a smaller SEET header are needed. We present a simple, yet effective heuristic that computes a near-optimal header fragmentation to split a large SEET header into multiple small headers. We compare the efficiency of SEET with BIER in different topologies.

The remainder of the paper is structured as follows. In Section II we describe related work. Then, we introduce BIER and show that BIER's scaling mechanism results in many redundant packet copies in large networks in Section III. Section IV introduces SEET and Section V gives a brief introduction to P4. Afterward, we present a P4-based implementation of SEET in Section VI. We present a simple, yet effective heuristic that determines how size-constrained near-optimal headers for SEET can be constructed in Section VII. We evaluate the scalability of BIER and SEET in different topologies in Section VIII and conclude the paper in Section IX.

II. Related Work

We first review related work for stateful multicast solutions. Then we discuss existing approaches for stateless multicast.

A. Stateful Multicast

IPMC is the default multicast service in IP networks. It was introduced in 1986 [7] and defines the transmission of IP datagrams to a set of hosts. Hosts dynamically join multicast groups, and the membership information of multicast groups is propagated through the network with the help of multicast routing protocols, e.g., PIM [8]. Islam et al. [9] and Al-Saeed et al. [10] provide a broad overview of stateful multicast services. They discuss shortcomings of IPMC regarding scalability and signaling overhead. Iyer et al. [11] present the Avalanche Routing Algorithm (AvRA) that leverages properties of data center topologies to compute optimized multicast distribution trees. They present an OpenFlow-based controller module that improves data rate by up to 12% and reduces packet loss by 51% compared to traditional IPMC. Dual-Structure Multicast (DuSM) [12] leverages the SDN paradigm to remove multicast management logic from switches. An SDN-based controller manages multicast group state on forwarding devices and balances traffic among multiple shared forwarding trees to avoid congestion. Further, the controller applies a multicast-to-unicast translation for multicast groups with low bandwidth to reduce the required state on forwarding devices. Voyer et al. [13] propose a new segment routing type for multi-point service delivery. The so-called SR replication segment instructs nodes to replicate packets to a set of downstream nodes in a SR domain. The mapping between a replication segment and a set of downstream nodes, called replication state, is held by the corresponding replication nodes. The replication state may change over time if the leaf nodes of a multi-point service change. Therefore, it resembles the dynamic forwarding state of traditional IPMC.

B. Stateless Multicast

Small Group Multicast (SGM) [14] proposed the idea to explicitly encode the multicast destinations in an IPMC header to enable stateless forwarding. Although the concept was not pursued any further due to the limited processing capabilities of ASICs at that time, it laid the foundations for subsequent work. Elmo [15] aims to improve the scalability of IPMC in data center environments. Multicast group information is encoded in the packet header, which eliminates the need for a dynamic state in forwarding devices. They claim to support up to one million different multicast groups in a three-tier data center topology with 27.000 hosts with an average packet header size of 114 B (bytes). Several works leverage Bloom filters to efficiently encode multicast traffic [16] [17] [18]. However, due to the inherent false-positive nature of Bloom filters, redundant or wrong forwarding decisions may be taken, which makes it unsuitable for a reliable multicast service. BIER [3] is currently standardized by the IETF and proposes a novel stateless transport mechanism for IPMC. It is based on the notion of a bit string, in the following referred to as bitstring, that indicates the recipients of a multicast group. Forwarding devices within a BIER domain are able to forward BIER packets according to the bitstring without the need for dynamic state. Merling et al. [19] [20] and Lindner et al. [21] present a P4-based implementation of BIER on high-performance switching hardware. The presented prototypes are able to forward BIER-based multicast with 100 Gb/s per egress port. In subsequent work, Merling et al. [22] investigate the efficiency of BIER multicast in large networks. They compare the traffic savings of IPMC and BIER relative to unicast forwarding in a wide range of network topologies. Further, they present algorithms to build optimal BIER subdomains for large networks. BIER with tree engineering (BIER-TE) [4] augments BIER with tree engineering capabilities. It is based on the same header format as BIER, i.e., a bitstring that indicates the recipients of a multicast group. Further, the bitstring contains a bit for each adjacency in the network. If the corresponding bit is set, the packet is forwarded over this adjacency. Flüchter et al. [23] present a P4-based implementation of BIER-TE. They introduce a novel scalability concept to scale BIER-TE to large networks. Further, the show how node and link failures can be tackled within their concept and present evaluations regarding throughput on the Intel Tofino™. Hawkeye [24] enhances BIER-TE with a deep reinforcement learning agent that builds multicast distribution trees. Hawkeye can proactively compute multicast trees based on historical traces. MSR6 [25] implements BIER and BIER-TE based on the SRv6 [26] forwarding plane. It introduces a so-called RGB segment that contains the BIER bitstring and leverages unicast IPv6 forwarding between replication nodes. Eckert et al. [6] proposes Recursive Bit-String Structure (RBS) for BIER and MSR6 to improve the scalability for sparse multicast trees in large networks. They encode the forwarding tree in a hop-by-hop fashion using local bitstrings. Diab et al. [27] present YETI, a stateless and generalized multicast forwarding scheme. It is based on label and bitstring-based forwarding, similar to SEET, and can be used to encode an arbitrary multicast distribution tree. They compare it with existing rule-based multicast solutions as well as BIER-TE. However, they do not consider header limitations of modern forwarding ASICs and evaluate their solution only on small ISP backbone topologies with at most 197 routers and 486 links. In contrast, we consider realistic header limitations for SEET, propose an effective optimization heuristic that derives how a SEET header is fragmented into multiple packets, and evaluate SEET in large access network topologies with several thousand receivers.

III. Bit Index Explicit Replication (BIER)

We first give an overview of BIER(-TE) and explain its scaling mechanisms for large networks. Then we explain performance issues of BIER(-TE) in large networks with sparse multicast trees.

A. Overview

Bit Index Explicit Replication (BIER) [3] is a stateless transport mechanism for IPMC that has been standardized by the IETF. It is based on a domain concept and introduces three different types of BIER devices: Bit-Forwarding Ingress Routers (BFIRs), Bit-Forwarding Routers (BFRs), and Bit-Forwarding Egress Routers (BFERs). Figure 1 illustrates the concept of BIER.

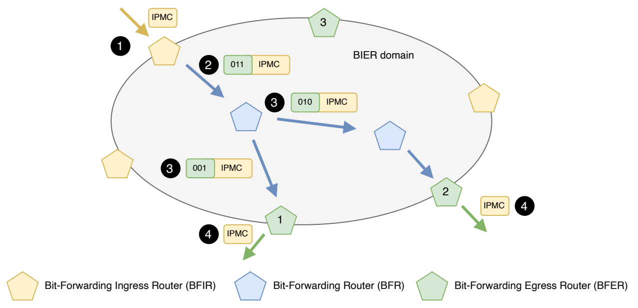

IPMC
❶
3
BIER domain
❷ 011 IPMC
❸ 010 IPMC
❸ 001 IPMC
2
IPMC ❹
1
❹ IPMC
Bit-Forwarding Ingress Router (BFIR)
Bit-Forwarding Router (BFR)
Bit-Forwarding Egress Router (BFER)

FIGURE 1. A BIER domain is composed of Bit-Forwarding Ingress Routers (BFIRs), Bit-Forwarding Routers (BFRs), and Bit-Forwarding Egress Routers (BFERs).

Bit-Forwarding Ingress Routers (BFIRs) are the ingress nodes of the BIER domain. They receive IPMC packets ❶ and encapsulate them with a BIER header ❷. The BIER header contains a bit string, which we call BIER bitstring, that indicates the destinations of the packet within the domain. Each BFER is assigned to a bit position in the BIER bitstring. For simplicity, BFER n has been assigned to bit position[1] n in Figure 1. A bit is activated in the bitstring if the corresponding BFER should receive a copy of the packet. Within the BIER domain, Bit-Forwarding Routers (BFRs) forward the BIER packet solely according to the BIER bitstring in the header. To that end, a BFR sends a packet copy to each next-hop over which at least one destination is reached. The bitstring is altered in each packet copy, such that it only contains the activated bits for BFERs that are reached via this next-hop ❸. This prevents duplicates at the receiver. Packets are forwarded according to the forwarding information from the routing underlay. Finally, Bit-Forwarding Egress Routers (BFERs) remove the BIER header and forward the IPMC packet as usual ❹.

BIER-TE augments the concept of BIER with tree engineering capabilities. It is based on the same header format

[1]Bit position 1 corresponds to the lowest-significant bit.

as BIER, i.e., a bitstring that indicates the recipients of a multicast group. Further, the bitstring contains a bit for each adjacency in the network. If the corresponding bit is set, the packet is forwarded over this adjacency.

B. Scaling BIER(-TE) to Large Networks

The number of BFERs is limited by the size of the BIER bitstring. Common bitstring lengths are 256-, 512-, and 1024-bit. We refer to a BIER domain with bitstring length x as BIER-x. The bitstring length might be limited due to different reasons, e.g., technical restrictions in forwarding ASICs or header overhead tradeoffs. For example, a network with 10.000 receivers requires a 1250 B bitstring, which is not feasible in practice. BIER introduces subsets of BFERs to scale to larger networks. A BIER subset S_i is identified by the so-called Set Identifier (SI) in the BIER header. The SI remaps a bit position of the BIER bitstring to a different BFER for each subset, i.e., the first bit position identifies BFER 1 in S_1 and BFER 5600 in S_2. With this approach, a BIER domain can support 10.000 BFERs with a 256-bit BIER bitstring and 40 subsets. Optimal SI selection, i.e., assigning a BFER to a SI in an optimal manner[2], is an NP-hard problem [22].

Scaling BIER-TE also leverages subsets, but these subsets comprise both BFERs and links, and the elements in the subset must be connected. Thus, covering a domain with subsets requires more subsets for BIER-TE than for BIER, leading to worse scaling properties. Moreover, due to the connectivity constraint, it is harder to find appropriate subsets for BIER-TE than for BIER, in particular in the presence of resilience requirements [23]. Therefore, no algorithm for finding suitable subsets for BIER-TE has been published, yet.

C. Performance Issues

Large BIER domains may require multiple subsets to reach all BFERs. If a BIER packet is destined to BFERs in several subsets, multiple BIER packets (with different BIER headers) are sent by the BFIR, one packet for each subset. Consequently, the same IPMC packet may be sent over a link multiple times. If the same IPMC packet is sent five times over the same link, four of these packets are redundant. This may reduce the advantage of BIER over native IPMC. Figure 2 shows an example where an IPMC packet is sent over the same link multiple times.

In the example, a BIER domain with two BFERs is divided into two subsets, i.e., subset one and subset two. Both subsets contain a single BFER that is identified within its subset by the least-significant bit. When an IPMC packet that needs to be forwarded to both BFERs enters the BIER domain, the BFIR creates a BIER packet for each subset. Based on the

[2]An objective function might be to minimize the overall traffic rate or number of redundant packets.

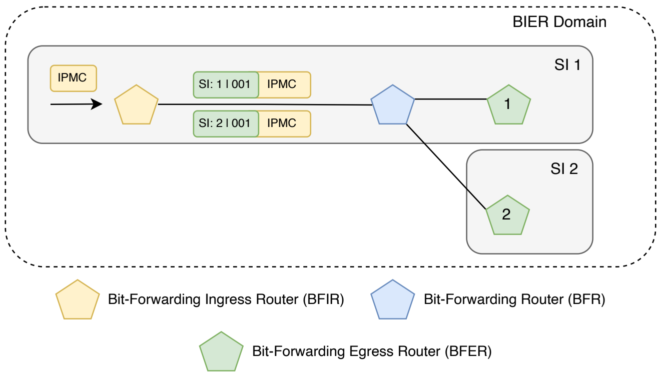

BIER Domain
IPMC
SI: 1 I 001
IPMC
SI: 2 I 001
IPMC
SI 1
1
SI 2
2
Bit-Forwarding Ingress Router (BFIR)
Bit-Forwarding Router (BFR)
Bit-Forwarding Egress Router (BFER)

FIGURE 2. The BFIR sends a BIER packet for each subset that contains a receiver for that packet. This might lead to multiple packets on a link.

topology, both BIER packets are forwarded through the same link from the BFIR to the BFR. Therefore, the same IPMC packet is forwarded over the same link two times instead of a single time as in native IPMC, leading to a redundant BIER packet on that link.

We quantify the advantage of BIER over native IPMC with the following experiment. We consider a BIER domain with $n = 1024$ BFERs, an average node degree of eight, bitstring lengths of $b \in \{64, 128, 256, 512, 1024\}$ bits, and $s \in \{16, 8, 4, 2, 1\}$ subsets, respectively. Then, we send BIER packets from a random source to n random receivers and repeat the experiment 50 times. We count the number of packets on all links p_{l_i} for both IPMC and BIER and report their difference, i.e., #redundant packets = $\sum_i p_{l_i}^{BIER} - \sum_i p_{l_i}^{IPMC}$. BFERs are randomly assigned to a single subset with equal probability. Figure 3 shows the number of redundant BIER packets that are sent to reach all receivers.

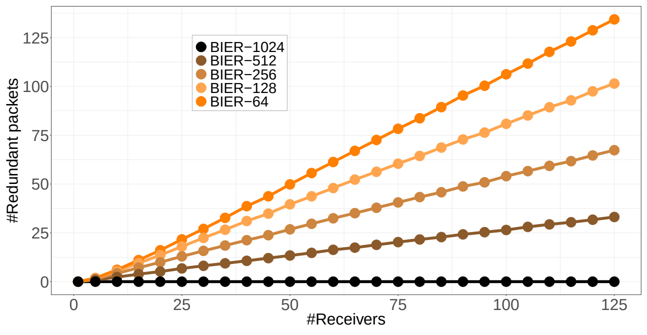


FIGURE 3. Number of redundant BIER packets to reach all receivers.

Native IPMC and BIER-1024, i.e., BIER with a bitstring length of 1024 bit, send at most one packet over a link and do not require redundant packets to reach all destinations. BIER variants with smaller bitstring lengths require more redundant packets. The number of redundant packets scales with the number of receivers of an IPMC packet and is more severe in larger networks.

IV. Segment-Encoded Explicit Trees (SEETs)

In this section, we introduce a novel stateless tree encoding mechanism, which we call Segment-Encoded Explicit Trees (SEETs). It is based on ideas of Segment Routing (SR) and RBS/BIER but uses its own encoding for better efficiency. First, we explain the general concept of a generic multicast tree encoding. Then we give an overview of SEET and explain its encoding in detail. Finally, we provide pseudocode for the forwarding logic of SEET-enabled devices.

A. Generic Multicast Tree Encoding

Stateless multicast solutions require the multicast distribution tree to be encoded within the packet. We propose the following generic encoding scheme that translates a recursive tree structure to a linear sequence of instructions that is agnostic to a specific protocol implementation. Figure 4 illustrates the generic encoding concept.

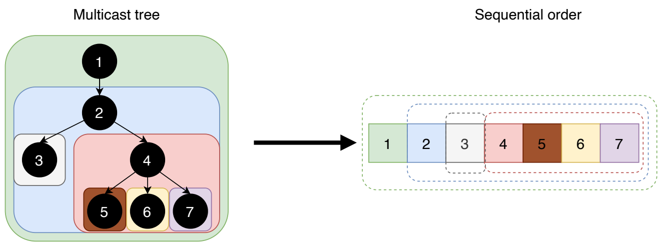


FIGURE 4. Concept for stateless multicast source routing. A generic multicast distribution tree is translated into a sequential list structure.

A generic multicast distribution tree is encoded in a sequential list structure. Thereby, the tree structure is serialized into a list of elements following a depth-first search pre-order traversal. The forwarding principle of the stateless multicast source routing is illustrated in Figure 5. When a node receives an encoded multicast packet, it first partitions the encoded tree into its subtrees. Then, a packet copy is created for each next-hop. The packet copy contains only the relevant subtree, i.e., the subtree that starts with the next-hop. Thereby, the packet header shrinks along the forwarding path.

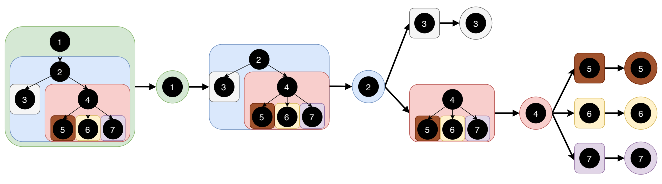


FIGURE 5. Forwarding principle of the encoding concept for stateless multicast source routing. Only the relevant subtree of the packet header is forwarded to a downstream node in the multicast tree.

B. SEET Overview

SEET is a forwarding scheme to steer a multicast packet along an explicit or implicit multicast tree. It supports both shortest-path forwarding as well as tree engineering. SEET's encoding scheme has been designed to be implementable on low-cost switching ASICs, e.g., with P4 [28] on the Intel Tofino™. A proof of concept implementation of SEET for the Intel Tofino™ is described in Section VI. Figure 6 illustrates the concept of SEET.

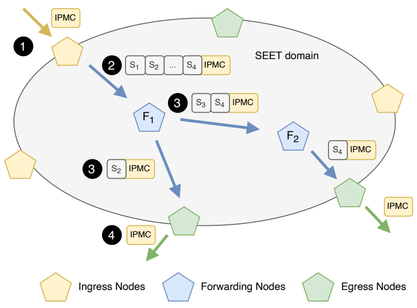


FIGURE 6. A SEET domain is composed of ingress nodes, forwarding nodes, and egress nodes.

SEET is based on a domain concept similar to BIER and introduces three different types of devices: ingress nodes, forwarding nodes, and egress nodes. An ingress node receives an IPMC packet and prepends a list of ordered segments to the packet ❶. We refer to this list of segments as forwarding stack (fs) ❷. Each segment encodes a SEET-specific identifier that is used by nodes to forward the packet along the distribution tree. Node identifiers can be either derived through the routing underlay, e.g., exchanged in IGP messages, or configured by a central configuration unit, e.g., a PCE. Figure 6 illustrates how a SEET packet with four segments is received by the first forwarding node F_1. The initial segment S_1 identifies F_1 itself and instructs it to process the forwarding stack while the rest of the forwarding stack encodes the downstream multicast distribution tree. The first part of the forwarding stack, i.e., $\{S_2, ..., S_i\}$, encodes the downstream distribution tree for the first next-hop of F_1, and the second part of the forwarding stack, i.e., $\{S_{i+1}, ..., S_n\}$, encodes the downstream distribution tree for the second next-hop. When F_1 forwards the SEET packet to its neighbors, only the corresponding downstream forwarding stack is kept on the packet, the other part is removed. Finally, the egress nodes remove the SEET header and forward the underlying IPMC packet ❹.

C. SEET Encoding

The multicast distribution tree is recursively encoded in the forwarding stack. Figure 7 illustrates the encoding.

A SEET header consists of a 16 bit next protocol field and a list of segments, i.e., a forwarding stack (fs). The next protocol field is used to identify the protocol of the payload. A segment consists of a n-bit identifier, a deliver bit (D), a 1-bit bitstring indicator (B), y-bit padding (P), and an 8-bit length (L) field that indicates how many bytes are left in the current distribution tree. A byte alignment for segments is required to facilitate parsing in low-cost forwarding ASICs, i.e., $(n + 1 + 1 + y) \ mod \ 8 = 0$ has to hold. The identifier is used by forwarding nodes to determine the next-hop. For

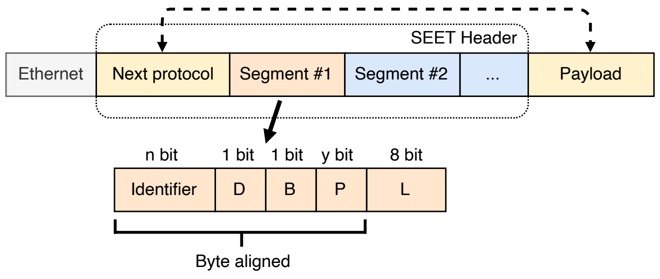


FIGURE 7. **A SEET header consists of a next protocol field and several segments that form the forwarding stack.**

example, in a network with 100 nodes, seven bits suffice to identify all nodes in the network. The D-bit indicates whether the node that is identified through the identifier is a destination of the multicast tree.

Figure 8 shows an example for a SEET forwarding stack that instructs a node with identifier 1052 to replicate a packet to two neighbors R_1 and R_3 that are receivers of the multicast distribution tree.

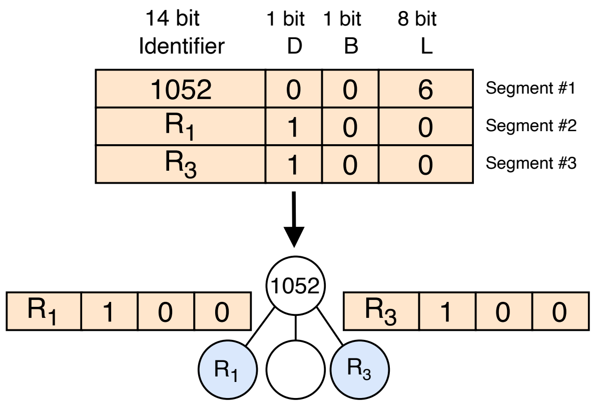


FIGURE 8. **Example SEET forwarding stack encoding two receivers of node with node identifier 1052.**

The example uses 14-bit global node identifiers. The node with identifier 1052 receives the forwarding stack and detects that the identifier in the first segment is its own identifier. As the D-bit is not set, it simply removes the first segment and begins to process the second segment. The second segment instructs node 1052 to create a packet copy with the next zero bytes (only segment #2) and send the packet copy to the next-hop identified by the identifier R_1. Then the second segment is removed and the same procedure is applied for the third segment. When R_1 and R_3 receive the respective packets, they detect that the identifier in the first segment is their own identifier. As the D-bit is set, they pass a packet copy without the SEET header to their upper layer. As the length field is zero, processing stops afterward.

D. Efficient Replication at Leaf Nodes

SEET requires for each recipient of the multicast distribution tree at least one segment[3]. Therefore, SEET can be efficiently used to encode paths that span multiple hops but it is less efficient if a node should replicate a packet to multiple neighbors. The example of Figure 8 requires 6 B (2 segments) to encode the receivers of node 1052. Therefore, we propose to use an optional BIER-like bitstring to address multiple neighbors efficiently at the penultimate[4] hop in a multicast distribution tree. The B-bit indicates if a segment is followed by a BIER-like bitstring. In that case, the length field (L) is split into two 4-bit fields: bitstring length (BL) and bitstring set identifier (BSI). The first 4-bit indicate the length of the bitstring in bytes. The second 4-bit build an identifier with the same purpose as the SI in BIER. Bitstrings can only be used at the penultimate hop of a branch in the distribution tree. Figure 9 illustrates the forwarding stack for the same example as Figure 8 but with efficient replication at a leaf node. The example uses 14-bit global node identifiers and shows a distribution subtree where node #1052 is the penultimate hop with two receivers R_1 and R_3.

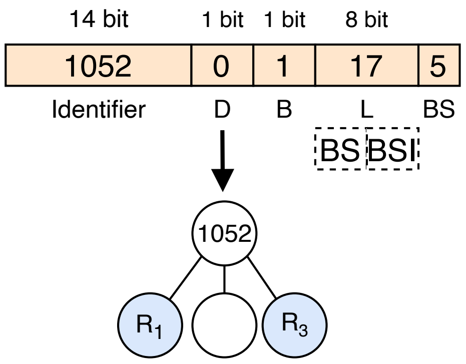


FIGURE 9. **The penultimate hop is addressed with a SEET identifier and carries a bitstring that is used to efficiently replicate the packet to multiple neighbors.**

The identifier 1052 in the SEET segment addresses the penultimate hop. The D-bit is set to 0 as the node is not a receiver of the distribution tree. The B-bit is set to 1 as the segment is followed by a bitstring. Therefore, the length field with value 17 (`0b00010001`) is split into two 4 bit values, i.e., BL = 1 and BSI = 1. It is followed by a 1-byte long bitstring corresponding to the first SI. Finally, the bitstring[5] 5 = `0b00000101` has the bits for receiver[6] R_1 and R_3 set. With the efficient replication at leaf nodes, both

[3]Multiple segments if an explicit path is encoded.
[4]Penultimate hop refers to the last hop before the actual destination.
[5]The bitstring has the same semantic as in BIER, i.e., each bit identifies a potential receiver.
[6]For simplicity, we assume that the least significant bit corresponds to R_1 and the third least significant bit to R_3.

receivers R_1 and R_3 in Figure 9 are encoded with a single byte instead of 6 B as in Figure 8.

Additionally, if BL equals zero, the BSI can encode up to 16 (2^4) static multicast groups. For example, BL = 0 and BSI = 0 may be configured to trigger a local broadcast to a pre-defined set of neighbors.

E. SEET Forwarding Algorithm

We formalize the above sketched forwarding algorithm using pseudocode.

Algorithm 1 shows the forwarding logic for SEET in pseudocode for a packet p that has been received by a $node$ with forwarding stack $p.fs$. It uses the following methods without further formalization:

- $p.fs.pop()$: Removes the first segment in the forwarding stack fs of a packet p.
- $node.getNextHop(identifier)$: Returns the next-hop for a $node$ identified through the $identifier$.
- $p.fs.pop(l)$: Removes the first segment and the next lB in the forwarding stack fs of a packet p.

The first segment in the forwarding stack ($p.fs[0]$) identifies the next-hop in the SEET domain. If a node receives a SEET packet, it first checks if the identifier of the first segment identifies the node itself (line 1). If the first segment does not identify the node, the packet is forwarded according to the identifier to the next-hop (line 33).If the first segment identifies the node itself, it is first checked if the destination bit (D-bit) is set (line 2). An activated D-bit indicates that the node is a destination in the multicast tree. In that case, the packet is copied and passed to the upper layer without the SEET header for native IPMC processing (line 3).

If the first segment has the B-bit set (line 5), then the node is a penultimate hop that uses a bitstring for efficient replication. In that case, the packet without SEET header is forwarded to all neighbors[7] identified through the bitstring and the forwarding algorithm stops. If the B-bit is not set, the first segment is removed as it has been processed (line 10).

The following steps are performed as long as the forwarding stack is not empty. First, the next-hop of the packet is derived through the identifier of the first segment (line 13). Afterward, a packet copy is created that contains the first segment and, depending on the B-bit of the first segment, the next $p_{cp}.l$ or $p_{cp}.bs$ B of the forwarding stack (lines 15-23). Then, the packet copy is forwarded to the $next\_hop$ (line 25). Finally, the processed segments are removed from the original packet (line 27). The forwarding algorithm stops when all segments have been processed.

Tree engineering can be achieved by encoding each hop on the path as a segment in the forwarding stack.

[7]In this context, a neighbor identified in the bitstring is a receiver of the multicast distribution tree.

Algorithm 1: SEET forwarding algorithm.

```
Input: packet: $p$
       current node: $node$
1  if $p.fs[0].identifier == node.identifier$ then
       /* Check if destination bit is set */
2      if $p.fs[0].D$ then
3          copy packet to upper layer without SEET header;
4
       /* Check if B bit is set */
5      if $p.fs[0].B$ then
6          forward packet without SEET header according to bitstring;
7
8          return
9
10     $p.fs.pop()$ ;   /* Remove first segment */
11
12     while $p.fs$ is not empty do
           /* next-hop of the packet */
13         $next\_hop = node.getNextHop(p.fs[0].identifier)$ ;
14
           /* Keep relevant segments */
15         create packet copy $p_{cp}$ ;
16
           /* If bitstring is used, use BS field as length field */
17         if $p_{cp}.fs[0].B$ then
18             n_bytes = $p_{cp}.bs$
19         else
20             n_bytes = $p_{cp}.l$
21         end
22
23         $p_{cp}.fs = p_{cp}.fs[0]$ ++ next $n\_bytes$ B ;
24
25         forward $p_{cp}$ to $next\_hop$ ;
26
           /* Remove processed segments */
27         $p.fs.pop(n\_bytes)$ ;
28     end
29 end
30 else
       /* next-hop of the packet */
31     $next\_hop = node.getNextHop(p.fs[0].identifier)$ ;
32
33     forward $p$ to $next\_hop$;
34 end
```

V. Introduction to P4

We review fundamentals of P4 that are relevant for the implementation of SEET. First, we give an overview of the general P4 pipeline of the Intel Tofino™. Then, we discuss the concept of recirculation and the capabilities of the packet parser. Details of the P4 language, its ecosystem, and related literature can be found in [29].

A. Overview

Programming protocol-independent packet processors (P4) [28] is a programming language used to describe the processing behavior of the data plane of compatible network devices, so-called targets. A P4 target follows an architecture that defines the processing pipeline and P4 primitives that are supported. Further, architectures can define so-called externs that extend the capabilities of the processing pipeline with target specific functions, e.g., support for cryptography. Figure 10 illustrates a simplified P4 pipeline of the Intel Tofino™ defined through the Tofino Native Architecture (TNA).

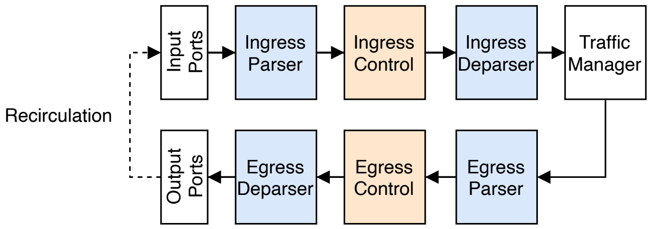


FIGURE 10. Visualization of a simplified Tofino Native Architecture (TNA) [30] P4 pipeline. The pipeline consists of an ingress parser, ingress control, ingress deparser, traffic manager, egress parser, egress control, and egress deparser.

The processing pipeline of the Intel Tofino™ is divided into `ingress` processing, i.e., when a packet is received, and `egress` processing, i.e., when a packet is transmitted. When a packet is received, it is first parsed by the ingress parser, and relevant packet headers are extracted. Afterward, the received packet is processed according to the implemented processing logic in the ingress control. This may involve changing header fields, storing information in registers, and deciding which port the packet should be forwarded to. This is typically done by matching the previously extracted header fields against user-defined match+action tables (MATs). After ingress processing, the packet is serialized through the ingress deparser and passed on to the traffic manager. The traffic manager is responsible for passing the packet to the correct egress port and performing packet replication, e.g., when a packet is cloned. Egress parser, egress control, and egress deparser have similar functionality as their ingress counterparts. After the egress deparser, the packet is physically transmitted through the corresponding egress port.

B. Packet Recirculation

P4 does not support the concept of loops. Therefore, iterative packet processing cannot be done within a single pipeline iteration. Iterative packet processing can be achieved through two mechanisms, called `resubmission` and `recirculation`. A resubmitted packet is immediately placed at the beginning of the ingress pipeline, i.e., at the ingress parser, after the initial ingress processing has finished. Therefore, resubmission allows to repeat the ingress processing on a packet. Resubmission can only be invoked during ingress processing. Further, the resubmitted packet corresponds to the initially received packet, i.e., all changes to the packet during the ingress processing are not applied. In contrast, recirculation takes place after egress processing. The packet is placed in the ingress parser as soon as the egress processing has stopped and all changes during the ingress and egress processing are applied. Recirculation on the Intel Tofino™ is a passive mechanism, which means that there is no P4 primitive that actively invokes recirculation. Ports can be configured to operate in a recirculation mode, i.e., packets transmitted through such a port are immediately placed in its ingress path again. This behavior is comparable to a physical loop. A packet that should be recirculated is forwarded through a port that operates in recirculation mode. We refer to such ports as recirculation ports.

C. Packet Parser

The packet parser extracts the relevant header fields used during ingress and egress processing. Both ingress and egress parsers are modeled as finite-state machine (FSM) and have a limited number of bytes that can be extracted depending on the capabilities of the ASIC. The parser divides the packet into extracted headers and its payload. The payload is not available during packet processing. Headers that are not extracted are considered to be part of the packet payload.

The parser extracts pre-defined headers according to the implemented FSM and the deparser emits previously extracted, possibly modified, (and still valid) headers. Figure 11 shows the definition of a custom header `example_header` and how it is extracted and emitted during parsing.

If a header is `invalidated` during ingress/egress processing, the header is not emitted in the deparser and, consequently, removed from the packet.

In addition to header extraction, P4 also supports to `advance` a packet during parsing. Thereby, the advanced bytes are removed from the packet. Figure 12 shows an example of a parsing state that advances the packet by 10 B.

Finally, the Intel Tofino™ provides a `ParserCounter` extern [30] that can be used to implement simple loops during parsing. We leverage the capabilities of the `ParserCounter` extern, `extracting` (and keeping) header fields and `advancing` (and removing) bytes in our SEET implementation (see Section VI) to dynamically keep

```
// header definition
header example_header {
    bit<8> type;
    bit<16> identifier;
}

// ingress parser
state parse_example_header {
    pkt.extract(hdr.example_header);
    // transit to the next state
    transition select(hdr.example_header.type) {
        ...
    }
}

// ingress deparser
// "add" previously extracted header to the packet
pkt.emit(hdr.example_header)
```

FIGURE 11. Example of a custom header that is extracted in the ingress parser and emitted in the ingress deparser.

```
// ingress parser
state remove_10_bytes {
    pkt.advance(8 * 10);
    // transit to the next state
}
```

FIGURE 12. Bytes can be removed by advancing the packet.

a certain number of segments and remove the remaining segments in a SEET header.

VI. P4 Implementation of SEET for Tofino

In this section, we give a brief overview of the P4 implementation of SEET for the Intel Tofino™. First, we give a high-level overview of the implementation. Then, we discuss its parsing logic in detail. The source code of SEET is available on GitHub[8].

A. Overview

Most of the packet processing logic in general P4 pipelines is done within the so-called ingress and egress parts of the pipeline. However, with SEET, we need to be able to split the SEET header of a packet at an arbitrary byte position, which is not possible during regular ingress/egress processing. Therefore, most of the SEET processing logic is done within the parser. We leverage the capabilities of the parser to `extract` (and keep) header fields and to `advance` (and remove) bytes from a packet to remove parts of the SEET header dynamically during parsing. Figure 13 illustrates the concept of the implementation where a SEET packet should be split into two SEET packets.

When a SEET packet is received for the first time, it is parsed up to the first two segments[9]. Then, the packet is copied and both the original and the packet copy are equipped with a bridge header[10]. The bridge header contains two values K_{bytes} and R_{bytes}. The first value (K_{bytes}) specifies how many bytes should be extracted and kept from the SEET header. The second value (R_{bytes}) specifies how many bytes should be advanced and removed from the SEET header. Then, both packet versions are recirculated. After recirculation, both packets are parsed before they enter the ingress section of the pipeline. Thereby, the parser extracts the first K_{bytes} B from the SEET header and removes the next R_{bytes} B. In the example of Figure 13, a SEET header with length $L = X + Y$ should be split after X B. Therefore, the first packet copy extracts the first X B and removes the next Y B of the SEET header, and the second packet copy extracts zero B and removes the next X B. Finally, the first packet copy is forwarded to its intended neighbor, and the second packet is treated as a new SEET packet for further processing. This procedure is repeated until the whole SEET header has been processed[11].

If the first segment has the bitstring indicator (B) set, the contained IPMC packet is replicated according to the local bitstring to all relevant neighbors without SEET header. Forwarding logic for bitstring-based replication is similar to [20] [21]. If the first segment has the deliver bit (D) set, an additional packet copy is passed to the upper layers of the device.

B. Parsing Logic

Packets that are recirculated are received on special ports and always carry a bridge header that contains the number of bytes that should be extracted (K_{bytes}) and the number of bytes that should be removed (R_{bytes}). Thereby, K_{bytes} is split into two fields: `tens` and `units`.

`Tens` represents the number of 10 B that should be extracted, and `units` represents the number of single bytes that should be extracted. Therefore, if 85 B should be extracted, `tens` equals eight and `units` equals five. We defined seven different headers with sizes of {100, 50, 20, 10, 5, 2, 1} byte(s). The 20-byte and 2-byte headers are implemented as header stacks of size 2, i.e., up to two 20-byte and up to two 2-byte headers can be extracted within the same header stack. Further, we defined 14 different parsing states P_i^K, $i \in [1, 14]$ that combine the different headers to extract $i \cdot 10$ B depending on the value of `tens`. For

[8]https://github.com/uni-tue-kn/seet

[9]If a bitstring follows the first segment, the bitstring is parsed instead.

[10]A bridge header is a header that is temporarily prepended to the Ethernet header for local processing. The header is removed when the packet is forwarded to its final destination.

[11]The length field of the original first segment is used to determine whether the whole SEET header has been processed.

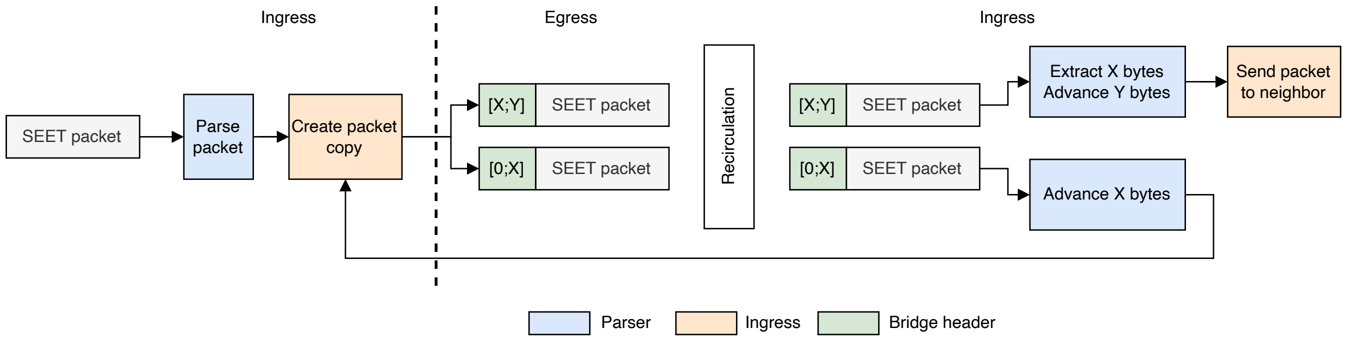


FIGURE 13. **High-level implementation overview of the SEET forwarding logic. Most of the processing logic is done within the parser.**

example, the parsing state P_{13}^{K} extracts one 100-byte header, one 20-byte header, and one 10-byte header. Similarly, nine additional parsing states extract up to 9 B depending on the number of `units`. This approach can extract between 0 and 149 B with at most two parse state transitions.

Afterward, the `ParserCounter` extern is used to advance the packet R_{bytes} times by one byte. The combination of extracted headers and the `ParserCounter` extern ensures that the maximal parse depth is not exceeded.

VII. Fragmentation Algorithm

We introduced an encoding for Segment-Encoded Explicit Tree (SEET) in Section IV. It represents the forwarding tree of a packet in the packet's header. However, so far we ignored that the maximum header size that can be processed by forwarding nodes is limited due to technical restrictions. Therefore, multiple packets may be required to deliver a message to all of its receivers. We present a simple yet efficient algorithm that runs in the control plane to fragment a message into multiple packets such that traffic overhead is minimized. The algorithm is executed when multicast groups change, and its output is used to configure the ingress node of a SEET domain. First, we give an overview of the idea of the algorithm. Then, we present its details. Afterwards, we discuss its runtime. Finally, we illustrate the algorithm by a brief example.

A. Overview

The presented encoding features two major ideas for minimizing the representation of a multicast tree. First, a long subpath of the path to a receiver can be bridged by a single SEET segment. Therefore, it is reasonable to address receivers that share long subpaths with the same packet. However, every replication in the multicast tree requires an additional SEET header. Thus, the number of replications in a multicast tree should be small. Second, multiple receivers with a common penultimate hop can efficiently be addressed by a local bitstring. We conclude that receivers should be grouped such that the multicast trees of the resulting packets contain few replication nodes except for replications at the penultimate hop.

B. The Algorithm

We propose a simple yet efficient algorithm which groups receivers according to the above observation. Given are a network topology, a source node, a set of receivers, and the desired paths for all source-receiver pairs. A forwarding tree is constructed by merging paths with common subpaths at the last node present in both paths. Likewise, a tree and a path are merged by adding a new branch to the tree at the last node present on the path. The algorithm starts with an empty packet header. A depth-first search in the forwarding tree is started at the source node of the message. Every time a receiver r of the message is discovered, it is added to the packet header according to exactly one of the following cases:

1) If the header is empty, a SEET header to r is introduced.
2) If the header does not contain a SEET header with a common subpath to r, a SEET header to r is introduced.
3) If the header contains a SEET header s to the penultimate hop of r, r is included in the local bitstring of s.
4) If a SEET header s addresses a node r' with the same penultimate hop p as r, s is removed from the header, a SEET header to p is introduced, and r and r' are added to the local bitstring of the new SEET header.
5) If the header contains some SEET header s with a common subpath to r and none of the other cases applies, a SEET header to the last possible replication node of s and r is inserted before s and a SEET header to r is introduced.

If the resulting packet header exceeds the maximum header length, the discovered receiver is not added to the header. Instead, the current packet is finished and the receiver is added to a new packet header. The algorithm terminates when all receivers are added to a packet.

C. Runtime

Let V and E be the sets of vertices and edges in the network topology. Depth-first search has a runtime of $\mathcal{O}(|V| + |E|)$. In the worst case, every node of the topology is a receiver.

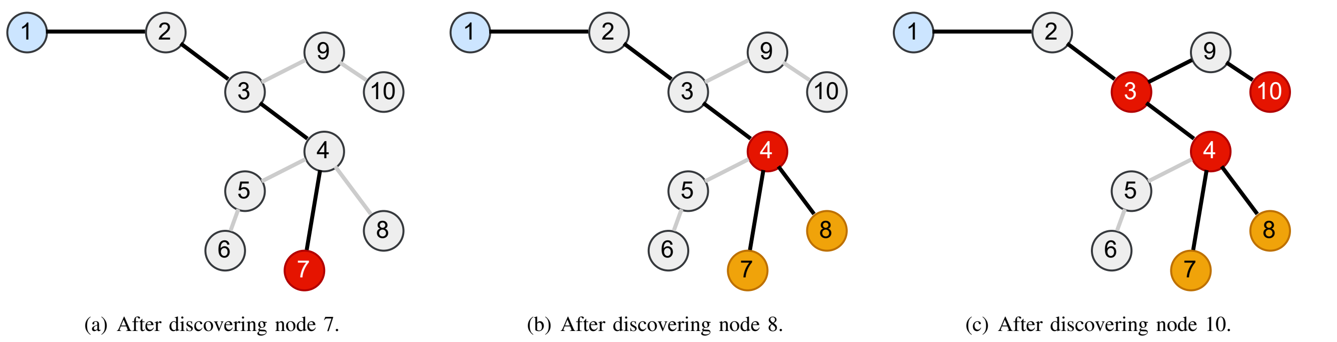

(a) After discovering node 7.
(b) After discovering node 8.
(c) After discovering node 10.

FIGURE 14. **Depiction of the algorithm for a message sent from node 1 to the nodes 7, 8, and 10. Red nodes are addressed by a SEET header while orange nodes are addressed by a local bitstring.**

Deciding which of the above cases applies and finding the replication point in case 5) can be done in $\mathcal{O}(|E|)$ if the multicast tree of the current packet is stored as list of edges in topological order. Thus, the runtime of the presented algorithm is $\mathcal{O}(|V| \cdot |E|)$ in the worst case. Typically, this is an heavy overestimation as the multicast tree of the current packets contains significantly less than $|E|$ edges.

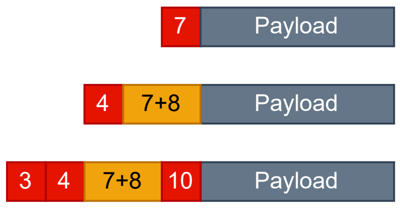


FIGURE 15. **Headers aftet discovering nodes 7, 8, and 10 in the example. SEET headers are depicted red while local bitstrings are depicted orange. Numbers indicate the destinations of the respective header.**

D. Illustrating Example

We explain the algorithm by a brief example. Figures 14(a)–14(c) depict a network topology and three steps of the algorithm. A message should be sent from node 1 to the nodes 7, 8, and 10. The nodes are discovered in ascending order. The headers after the steps of the example are shown in Figure 15. Initially, no receiver is covered by the packet's header. When node 7 is discovered in Figure 14(a), case 1) from the algorithm's description applies. Thus, a SEET header with node 7 as destination is introduced. Then, node 8 is discovered in Figure 14(b). The current header contains a SEET header to a node with the same penultimate hop as node 8. Thus, case 4) of the algorithm's description applies. The SEET header to node 7 is removed and a SEET header to the penultimate hop, node 4, is introduced instead. Nodes 7 and 8 are addressed by the local bitstring of node 4. Finally, node 10 is discovered in Figure 14(c). The path to node 10 shares a common subpath with the path of the already existing SEET header to node 4. Thus, case 5) of the algorithm's description applies. The latest possible replication node to reach node 4 and node 10 is node 3. Thus, a SEET header to node 3 is introduced which contains SEET headers to node 4 and node 10 recursively.

Eventually the header will exceed the size limit in larger topologies with more receivers than in the presented example. The header is considered full in this case and a new empty header is the new working header. The depth-first search proceeds with the last discovered node and the algorithm terminates when all receivers were discovered.

VIII. Evaluation

We evaluate the encoding and the message fragmentation algorithm. To that end, we compare the presented approach with traditional IPMC and BIER. First, we introduce the methodology of the evaluations. Then, we motivate the fragmentation algorithm by evaluating the header sizes imposed by SEET. Afterward, we evaluate the relative overhead of SEET and BIER with respect to packet transmissions compared to IPMC. Finally, we present results regarding the overall traffic transmitted in the network.

A. Methodology

We give the details of the evaluation setup such as algorithm, network topology, traffic model, and evaluation metrics.

1) Algorithm

We use the algorithm from Section VII to construct SEET packet headers. We employ the methodology from [22] to compute optimized BIER domains, i.e., optimized subsets of BFERs.

2) Network Topology

We sampled 20 graphs with 1024 nodes according to the Waxman model [31] such that the average node degree is 4. The nodes of these graphs represent the core nodes of a distribution network. For each core node, we added 16 end systems and connected them to the respective core node.

Thus, the resulting network topologies contain $(16+1)\cdot 1024 = 17408$ nodes and $(16+2)\cdot 1024 = 18432$ links.

3) Traffic Model

For every network topology n and number of receivers $r \in \{1, 2, 4, ..., 16384\}$, we sampled 20 sets of r receivers from the set of end systems of n. For every such set of receivers $\mathcal{R}$, a message is send from every end system to all end systems in $\mathcal{R}$. The same sets are used to evaluate SEET, BIER, and IPMC. We remark that randomized sets of receivers constitute a worst case for SEET as the local bitstrings cannot be leveraged for leafs of different core nodes.

4) Metrics

We calculate results with three metrics: maximum header size, relative packets, and relative traffic.

5) Implementation

All evaluation results are computed with calculations for static settings, i.e., no time-dependent simulation is required. That means that all metrics only depend on the sent packets and the topology. The evaluations were implemented in Rust. The execution was accelerated via data parallelism with Rayon. The network topologies and the evaluation code can be found on GitHub[12].

a: Maximum header size

The maximum header size is the number of bytes required to encode a set of receivers, excluding headers of lower layers and IP headers.

b: Source packets

The source packets metric captures the load imposed to source nodes due to packet construction. It is the number of packets sent per source node averaged over all end systems. We remark that IPMC requires exactly one source packet regardless the the set of receivers. Thus, all results can be considered to be relative to IPMC.

c: Relative packets

The relative packets metric represents the overhead of individual packet hops compared to IPMC. Let p_{IPMC} be the number of packet hops required to sent a message from some source node to some set of receivers via IPMC. If an alternative multicast approach $A \in \{\text{BIER}, \text{SEET}\}$ requires p_A packet transmissions for the same source node and set of receivers, the relative packets metric is formally defined as $\frac{p_A}{p_{\text{IPMC}}}$. For every number of receivers r, we report results averaged over all source nodes, sets of receivers with size r, and network topologies for the maximum header size and the relative packets metrics.

[12]https://github.com/uni-tue-kn/seet

d: Relative traffic

The relative traffic metric captures the overhead of data transmitted in the network for a given set of receivers. Thus, it is the sum of the sizes of all packet hops for all source end systems, including payload and IP headers, relative to IPMC. We assume a payload of 500 B as empirical studies suggest this is the average payload of IP packets in the Internet [32]. For every number of receivers r, we report results averaged over all network topologies and receiver sets of size r.

B. Header Size

The header size of IPMC and BIER packets is predefined and does not depend on the set of receivers. This is not the case for SEET due to its tree engineering capabilities. Figure 16 depicts the average initial header size resulting from sending a message to varying numbers of receivers.

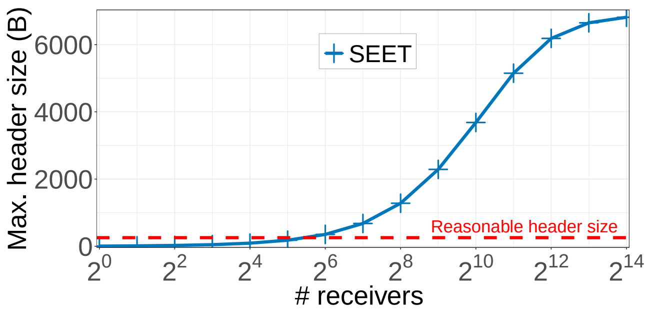
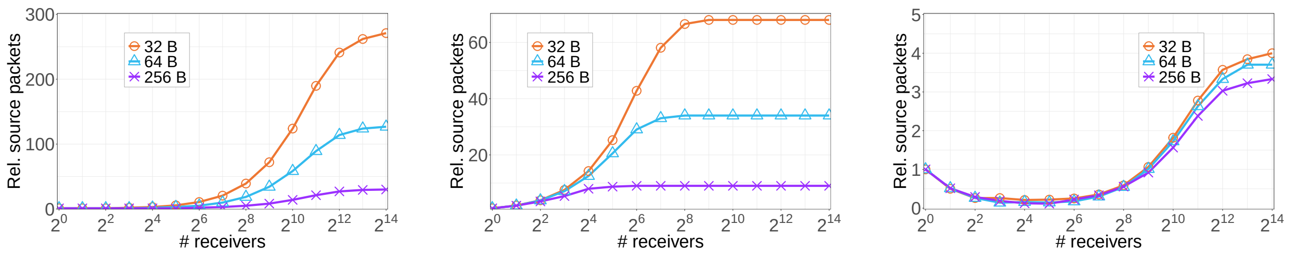


FIGURE 16. Average initial header size for varying numbers of receivers. The dashed line indicates a maximum header size of 256 B which can be processed by the presented implementation.

We observe a fast increase of the header size for moderate numbers of receivers ($r \geq 2^6$). With an increasing number of receivers, the header size reaches a plateau eventually. The reason for this behavior is the following. If two receivers are not leave of the same core node by chance, separate SEET headers are required to reach them. With an increasing number of receivers, chances are high that every core node is already included in the header. Thus, additional receivers can be added by simply flipping the corresponding bits in the local bitstrings of the SEET headers without increasing its sizes.

The dashed line indicates the maximum header size that can be processed by reasonable forwarding hardware (256 B). With such a header limit, only ~ 32 receivers can be addressed by a single packet in the case of uncorrelated receivers. We conclude that the message fragmentation algorithm from Section VII is necessary under realistic conditions. In contrast, BIER can encode 2048 receivers with the same header limit. Therefore, SEET is less efficient with respect to encoding denseness. However, SEET headers decrease in length on their path. Thus, no conclusions regarding the overall traffic volume can be drawn.



(a) SEET relative to IPMC. (b) BIER relative to IPMC. (c) SEET relative to BIER.

FIGURE 17. **Average number of source packets.**

C. Source Packets

We regard header sizes of more than 256 B as infeasible for practical applications due to hardware restrictions in forwarding devices. Thus, packets with more than 2^6 receivers must be split into multiple packets. However, the fragmentation of receivers into subsets matters with respect to the metrics from Section 4. We used the fragmentation algorithm of Section VII for this purpose. The fragmentation of receivers into subsets results in multiple packets per message send from an end system which imposes additional overhead.

Figures 18(a)–18(c) depict the average number of packets sent per end system. We observe that the number of source packets increases for larger sets of receivers (Figure 18(a)). This is consistent with the results from Section B. In the case of BIER (Figure 18(b)) the number of source packets saturates for rather small receiver sets and does not increase further. This is due to the design of BIER as only a single source packet per SD is required. Thus, the maximum number of source packets is sent when at least one receiver per SD is addressed.

Comparing SEET directly to BIER (Figure 18(c)), we see that SEET sends several times more packets than BIER in the case of many receivers. A BIER packet uses only a single bit in its header per receiver. While SEET also encodes some receivers with individual bits, replication nodes must be encoded with identifiers and subheaders. This in turn results in less receivers per header or more packets sent. However, in the case of small receiver subsets, SEET requires less source packets than BIER. This is due to BIER sending one packet per SD while SEET can address these receiver sets with a small number of packets.

D. Packet Overhead

We showed that the SEET encoding is less efficient than BIER with respect to the number of receivers addressable with a single packet. However, the forwarding tree of a packet with a small number of receivers contains less hops. Additionally, BIER and SEET require multiple packets for large sets of receivers which results in redundant packet transmissions compared to IPMC. Thus, it is not clear whether the remarks regarding encoding efficiency translate to the number of packet transmissions.

Figures 18(a)–18(b) depict the relative packet overheads of BIER and SEET for different maximum header sizes compared to IPMC. We observe that SEET (Figure 18(a)) and BIER (Figure 18(b)) benefit from larger headers as more receivers can be encoded within a single packet. Consequently, less additional packets need to be sent.

Further, the relative packet overhead of BIER and SEET compared to IPMC decreases for large sets of receivers. If a core node is already receiving a BIER or a SEET packet, forwarding it to an additional leaf of this core node requires only a single hop. The same does hold for IPMC which implies that the relative packet numbers decline.

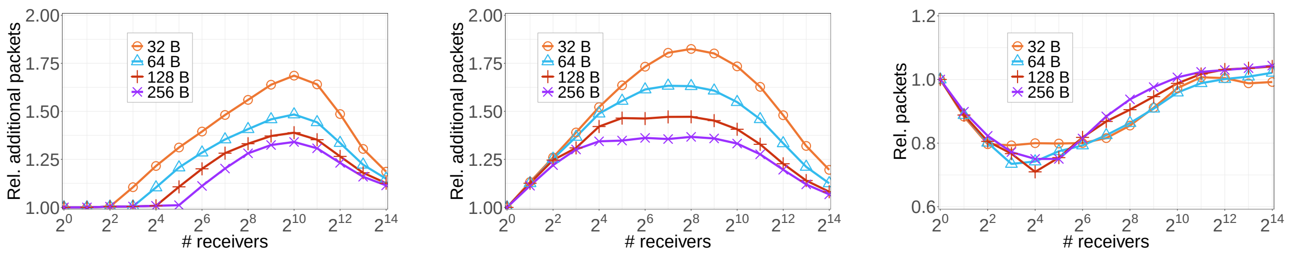
Figure 18(c) compares SEET directly to BIER. We see that SEET requires less or an equal number of packet transmissions than BIER. At first this result seems counterintuitive as BIER can address more receivers with a single packet. However, BIER subsets are statically configured and may be suboptimal from the perspective of some source nodes. In case of rather small sets of receivers BIER requires an individual packet per subset that contains at least one receiver. In contrast, SEET packets are individually optimized for every source node and set of receivers. Thus, a single packet is sufficient in many cases.

E. Traffic Overhead

The number of packets is an important metric for the processing load of forwarding hardware. Switching ASICs are limited by the number of packets that can be processed per second. However, network congestion and quality of service depend on the overall amount of traffic that must be transmitted. The total traffic amount depends on the number of individual packet hops and the size of a packet. Thus, there is a non-trivial tradeoff between reducing the number of packet hops or the header size. We compare SEET and BIER with respect to this tradeoff.

Figure 19 shows the traffic overhead of BIER and SEET relative to IPMC with small and large headers.

First, we observe a similar trend as in Figures 18(a)–18(b). With an increasing number of receivers, the relative overall



(a) SEET relative to IPMC. (b) BIER relative to IPMC. (c) SEET relative to BIER.

FIGURE 18. **Relative numbers of packet transmissions for varying numbers of receivers and different maximum header sizes.**

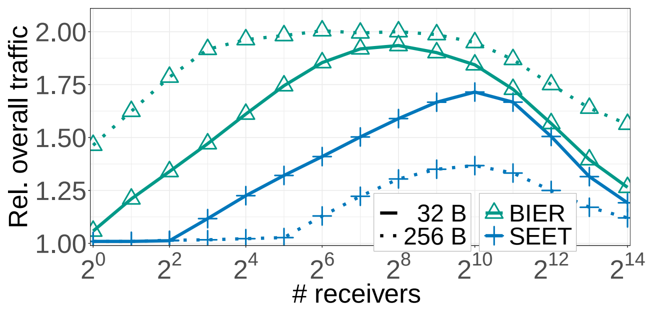


FIGURE 19. **Relative traffic of BIER and SEET for varying numbers of receivers and different maximum header sizes.**

traffic increases until almost all core nodes are already part of the distribution tree. Then, additional receivers can be addressed by simply flipping a bit in a local bitstring. Further, we observe that SEET results in less traffic overhead than BIER. While BIER is more efficient in encoding receivers into a packet's header, the size of a BIER header does not change along the packet's path. In addition, many bits of the header bits are set to 0, even for large sets of receivers. The topology under consideration consists of 16384 end systems. Thus, even with 8192 receivers, half of the header space is not efficiently used and only filled with zeros.

In contrast, these drawbacks of BIER do not apply for SEET. SEET can leverage the whole header limit at every packet to encode the distribution tree. Further, the size of the SEET header reduces at every replication node, and only the relevant parts are relayed to the respective subtree. Nodes that are not receivers of a packet are not represented in the header.

F. Performance

We measure the number of multicast groups that can be fragmented into packet headers per second. A multicast group only needs to be fragmented into multiple packets when multicast group memberships change. Once the multicast group has been fragmented, the ingress node of the SEET domain is configured, and packets are equipped with the appropriate SEET header in line rate, i.e., 100 Gbit/s per port. The measurement was performed on an AMD EPYC 7543 @ 2.8 GHz with 32 cores. The machine is equipped with 128 GB of RAM. However, the computation requires only 110 MB of RAM for all 32 threads combined. The algorithms and the evaluations were programmed in Rust. Figure 20 depicts the number of multicast groups that are processed per second. Even in the case of 16384 receivers, the presented approach constructed packets for more than 1000 multicast groups per second. Overall, runtimes increase linearly with the number of receivers in the evaluation scenario. Thus, we conclude that SEET and the fragmentation algorithm are suitable for applications with high multicast turnover rates.

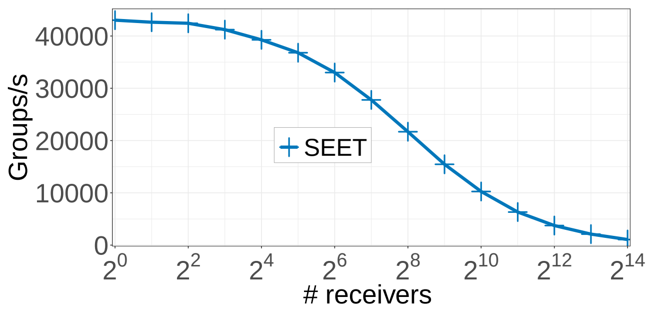


FIGURE 20. **Number of multicast groups that can be fragmented into packet headers per second.**

IX. Conclusion

In this work, we presented Segment-Encoded Explicit Tree (SEET), a novel stateless multicast protocol with tree engineering capabilities. SEET explicitly encodes a multicast distribution tree in the packet's header. SEET leverages principles from Segment Routing (SR) for tree engineering and BIER-like bitstrings for efficient packet replication towards leaf nodes. We presented a P4-based proof-of-concept implementation of SEET for the Intel Tofino™ ASIC that runs with 100 Gbit/s per port.

As very large multicast trees cannot be accommodated within a single packet header, we propose an efficient heuristic to provide mulitple traffic-efficient multicast trees that avoid additional traffic at best. We employed this algorithm to compare SEET and BIER in a quantitative study. The results showed that SEET mostly results in less source packets, packet transmissions, and overall traffic compared to BIER, especially for small and medium-size multicast groups. In that sense, SEET is a viable alternative for BIER.

Further, SEET supports tree engineering in contrast to BIER. It is clear that BIER's tree engineering variant BIER-TE scales worse than BIER with regard to sent packets as it also encodes links in addition to receivers in the bitstring. Therefore, SEET is in particular a strong alternative to BIER-TE. However, a direct comparison with BIER-TE was not possible as scaling BIER-TE towards large networks still suffers from management problems. Large BIER-TE domains need to be subdivided into connected subdomains, for which no solution has been presented so far. Future works may propose such an algorithm and compare the scalability of SEET and BIER-TE.

REFERENCES

[1] D. S. E. Deering, "Host extensions for IP multicasting," RFC 1112, Aug. 1989. [Online]. Available: https://www.rfc-editor.org/info/rfc1112

[2] N. K. Nainar, R. Asati, M. Chen, X. Xu, A. Dolganow, T. Przygienda, A. Gulko, D. Robinson, V. Arya, and C. Bestler, "BIER Use Cases," Internet Engineering Task Force, Internet-Draft, Sep. 2020, work in Progress. https://datatracker.ietf.org/doc/draft-ietf-bier-use-cases/12/.

[3] I. Wijnands, E. C. Rosen, A. Dolganow, T. Przygienda, and S. Aldrin, "RFC8279: Multicast Using Bit Index Explicit Replication (BIER)," Internet Engineering Task Force, Request for Comments, Nov. 2017, https://www.rfc-editor.org/info/rfc8279.

[4] T. Eckert, M. Menth, and G. Cauchie, "Tree Engineering for Bit Index Explicit Replication (BIER-TE)," RFC 9262, Oct. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9262

[5] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir, "Segment Routing Architecture," RFC 8402, Jul. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8402

[6] T. Eckert, M. Menth, X. Geng, X. Zheng, R. Meng, and F. Li, "Recursive BitString Structure (RBS) Addresses for BIER and MSR6," Internet Engineering Task Force, Internet-Draft draft-eckert-bier-rbs-00, Oct. 2022, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-eckert-bier-rbs/00/

[7] S. E. Deering, "Host extensions for IP multicasting," RFC 988, Jul. 1986. [Online]. Available: https://www.rfc-editor.org/info/rfc988

[8] B. Fenner, M. J. Handley, H. Holbrook, I. Kouvelas, R. Parekh, Z. J. Zhang, and L. Zheng, "Protocol Independent Multicast - Sparse Mode (PIM-SM): Protocol Specification (Revised)," RFC 7761, Mar. 2016. [Online]. Available: https://www.rfc-editor.org/info/rfc7761

[9] S. Islam, N. Muslim, and J. W. Atwood, "A Survey on Multicasting in Software-Defined Networking," *IEEE Communications Surveys & Tutorials*, vol. 20, pp. 355–387, 2018.

[10] Z. AlSaeed, I. Ahmad, and I. Hussain, "Multicasting in Software Defined Networks: A Comprehensive Survey," *Journal of Network and Computer Applications*, vol. 104, pp. 61–77, 2018.

[11] A. Iyer, P. Kumar, and V. Mann, "Avalanche: Data Center Multicast using Software Defined Networking," in *International Conference on COMmunication Systems and NETworks (COMSNETS)*, 2014.

[12] W. Cui and C. Qian, "Scalable and Load-Balanced Data Center Multicast," in *IEEE Globecom*, 2015.

[13] D. Voyer, C. Filsfils, R. Parekh, H. Bidgoli, and Z. J. Zhang, "SR Replication segment for Multi-point Service Delivery," Internet Engineering Task Force, Internet-Draft draft-ietf-spring-sr-replication-segment-15, Jun. 2023, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-spring-sr-replication-segment/15/

[14] D. R. H. Boivie and N. Feldman, "Small Group Multicast," Internet Engineering Task Force, Internet-Draft draft-boivie-sgm-02, Feb. 2001, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-boivie-sgm/02/

[15] M. Shahbaz, L. Suresh, J. Rexford, N. Feamster, O. Rottenstreich, and M. Hira, "Elmo: Source Routed Multicast for Public Clouds," in *ACM SIGCOMM*, 2019, p. 458–471.

[16] P. Jokela, A. Zahemszky, S. Arianfar, P. Nikander, and C. Esteve, "LIPSIN: Line speed Publish/Subscribe Inter-Networking," in *ACM SIGCOMM*, Barcelona, Spain, Aug. 2009.

[17] M. J. Reed, M. Al-Naday, N. Thomos, D. Trossen, G. Petropoulos, and S. Spirou, "Stateless Multicast Switching in Software Defined Networks," in *IEEE International Conference on Communications (ICC)*, May 2016, pp. 1–7.

[18] Z. Chen, J. Huang, Q. Wang, J. Liu, Z. Li, S. Zhou, and Z. He, "MEB: an Efficient and Accurate Multicast using Bloom Filter with Customized Hash Function," in *Asia-Pacific Workshop on Networking*, 2023, pp. 157—-163.

[19] D. Merling, S. Lindner, and M. Menth, "P4-Based Implementation of BIER and BIER-FRR for Scalable and Resilient Multicast," *Journal of Network and Computer Applications*, vol. 169, Nov. 2020.

[20] ——, "Hardware-Based Evaluation of Scalable and Resilient Multicast With BIER in P4," *IEEE Access*, vol. 9, pp. 34 500–34 514, Feb. 2021.

[21] S. Lindner, D. Merling, and M. Menth, "Learning Multicast Patterns for Efficient BIER Forwarding with P4," *IEEE Transactions on Network and Service Management*, vol. 20, no. 2, pp. 1238–1253, Jun. 2023.

[22] D. Merling, T. Stüber, and M. Menth, "Efficiency of BIER Multicast in Large Networks," *IEEE Transactions on Network and Service Management*, 2023.

[23] M. Flüchter, F. Ihle, S. Lindner, T. Eckert, and M. Menth, "Extensions to BIER Tree Engineering (BIER-TE) for Large Multicast Domains and 1:1 Protection: Concept, Implementation and Performance," https://doi.org/10.48550/arXiv.2409.07082, Sep. 2024.

[24] L. Lu, Q. Li, D. Zhao, Y. Yang, Z. Luan, J. Zhou, Y. Jiang, and M. Xu, "Hawkeye: A Dynamic and Stateless Multicast Mechanism with Deep Reinforcement Learning," in *IEEE Infocom*, May 2023.

[25] Y. Liu, J. Xie, X. Geng, and M. Chen, "RGB (Replication through Global Bitstring) Segment for Multicast Source Routing over IPv6," Internet Engineering Task Force, Internet-Draft draft-lx-msr6-rgb-segment-04, Mar. 2023, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-lx-msr6-rgb-segment/04/

[26] C. Filsfils, P. Camarillo, J. Leddy, D. Voyer, S. Matsushima, and Z. Li, "Segment Routing over IPv6 (SRv6) Network Programming," RFC 8986, Feb. 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc8986

[27] K. Diab and M. Hefeeda, "Yeti: Stateless and Generalized Multicast Forwarding," in *USENIX Syposium on Networked Systems Design & Implementation (NSDI)*, Apr. 2022, pp. 1093–1114.

[28] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-Independent Packet Processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, 2014.

[29] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, "A survey on data plane programming with P4: Fundamentals, advances, and applied research," *Journal of Network and Computer Applications*, vol. 212, 2023.

[30] Intel, "P416 intel tofino native architecture - public version," https://github.com/barefootnetworks/Open-Tofino, 2021.

[31] B. Waxman, "Routing of multipoint connections," *IEEE Journal on Selected Areas in Communications*, vol. 6, pp. 1617–1622, 1988.

[32] F. Liu *et al.*, "The packet size distribution patterns of the typical internet applications," in *IEEE International Conference on Network Infrastructure and Digital Content*, 2012, pp. 325–332.

Steffen Lindner is a postdoctoral researcher specialized in software-defined networking (SDN), P4, time-sensitive networking (TSN), and congestion management. He studied, worked, and obtained his bachelor's (2017), master's (2019), and Ph.D. (2024) degrees at the University of Tuebingen.

Thomas Stüber received the Doctoral degree from the Chair of Communication Networks of Prof. Dr. habil. Michael Menth, Eberhard Karls University Tübingen, Germany, in 2024. He became part of the Communication Networks Research Group after writing his master's thesis there in 2018. His research interests include time-sensitive networking (TSN), scheduling, performance evaluation, and operations research. He currently works in the automotive industry in the context of time synchronization and other TSN features for HIL/SIL platforms.

Maximilian Bertsch is a software developer at a medtech startup located in Tuebingen, Germany. He obtained his master's degree in 2023 at the Eberhard Karls University Tuebingen. His interests are development with Python and running cloud infrastructure with Kubernetes.

Toerless Eckert is a Distinguished Engineer at Futurewei, California, USA where he works on innovations in architecture and standardization of the Internet and its protocols. His experiences includes planning, building and operating networks with new technologies, educating and supporting customers around the globe, researching, developing, standardizing and building network products, protocol and services and developing advanced, network integrated multimedia applications. Toerless is subject matter expert for routing, multicast, MPLS, QoS and secure network automation. He was a part of Cisco Systems IOS operating system development team where he worked on IP/IPv6/MPLS multicast and from the early 2000s, IP/IPv6 multicast standardization in DOCSIS 3.0 and integration of multicast with a variety of networked applications. He led the architectures for the Medianet and Autonomous Networking advanced development projects. Currently, Toerless is co-chair of the IETF ANIMA working group which is defining an IPv6-centric and fully autonomous and secure network communications infrastructure. He holds more than 45 patents, issued and pending, and is co-author of 13 IETF RFCs and various IETF drafts. Beside IETF and CableLabs, he has also worked for standardization in ETSI and ITU-T and has published research papers and research book chapters. Toerless holds a CS diploma from Friedrich Alexander Universität Erlangen Nürnberg, Germany.

Michael Menth, (Senior Member, IEEE) is professor at the Department of Computer Science at the University of Tuebingen/Germany and chairholder of Communication Networks since 2010. He studied, worked, and obtained diploma (1998), PhD (2004), and habilitation (2010) degrees at the universities of Austin/Texas, Ulm/Germany, and Wuerzburg/Germany. His special interests are performance analysis and optimization of communication networks, resilience and routing issues, as well as resource and congestion management. His recent research focus is on network softwarization, in particular P4-based data plane programming, Time-Sensitive Networking (TSN), Internet of Things, and Internet protocols. Dr. Menth contributes to standardization bodies, notably to the IETF.