Yeti: Stateless and Generalized Multicast Forwarding

Khaled Diab and Mohamed Hefeeda, *Simon Fraser University*

https://www.usenix.org/conference/nsdi22/presentation/diab-yeti

This paper is included in the Proceedings of the
19th USENIX Symposium on Networked Systems
Design and Implementation.

April 4–6, 2022 • Renton, WA, USA

978-1-939133-27-4

Yeti: Stateless and Generalized Multicast Forwarding

Khaled Diab Mohamed Hefeeda
School of Computing Science
Simon Fraser University
Burnaby, BC, Canada

Abstract

Current multicast forwarding systems suffer from large state requirements at routers and high communication overheads. In addition, these systems do not support generalized multicast forwarding, where traffic needs to pass through traffic-engineered paths or requires service chaining. We propose a new system, called Yeti, to efficiently implement generalized multicast forwarding inside ISP networks and supports various forwarding requirements. Yeti completely eliminates the state at routers. Yeti consists of two components: centralized controller and packet processing algorithm. We propose an algorithm for the controller to create labels that represent generalized multicast graphs. The controller instructs an ingress router to attach the created labels to packets in the multicast session. We propose an efficient packet processing algorithm at routers to process labels of incoming packets and forwards them accordingly. We prove the correctness and efficiency of Yeti. In addition, we assess the performance of Yeti in a hardware testbed and using simulations. Our experimental results show that Yeti can efficiently support high speed links. Furthermore, we compare Yeti using real ISP topologies in simulations against the closest systems in the literature: a rule-based approach (built on top of OpenFlow) and two label-based systems. Our simulation results show substantial improvements compared to these systems. For example, Yeti reduces the label overhead by 65.3%, on average, compared to the closest label-based multicast approach in the literature.

1 Introduction

Recent large-scale Internet applications have introduced a renewed interest in scalable multicast services. Examples of such applications include live Internet broadcast (e.g., Facebook Live), IPTV [27], webinars and video conferencing [22], and massive multiplayer games [26]. The scale of these applications is unprecedented. For instance, due to the COVID-19 pandemic, a recent study [2] reported an increase by one order of magnitude within two months in video conferencing traffic passing though a major European ISP. Moreover, Facebook Live aims to stream millions of live sessions to millions of concurrent users [8, 41]. To reduce the network load of such applications, ISPs can use multicast to efficiently carry the traffic through their networks. Examples of commercial systems using multicast include AT&T UVerse [40] and BT YouView [37]. Beyond multimedia systems, multicast is also useful for various applications such as real-time stock market updates, cloud applications [33], and publish-subscribe systems [24, 36, 39]. For instance, the CIO of the Japan Exchange Group highlighted the importance of multicast for their stock trading operations [32].

Large ISPs need to support *generalized* multicast forwarding to handle various business requirements. Specifically, providers of large-scale live applications require ISPs carrying their traffic to meet target quality metrics or SLAs (service level agreements), especially for popular/paid live multicast sessions. To meet the SLAs for various customers, ISPs may need to direct the traffic to network paths different from the minimum-cost ones computed by the routing protocols deployed in the ISP network. This is usually referred to as *traffic engineering*. Prior works, e.g., [7, 11, 12, 16], have proposed algorithms to support various traffic engineering objectives.

In addition, ISP customers may require their multicast traffic to pass though an *ordered* sequence of network services such as firewall, intrusion detection, and video transcoding before reaching the destinations. This is referred to as *service chaining*. Network services are usually deployed as virtual functions running on servers attached to *some* of the core routers in the ISP network. Previous works, e.g., [1, 5], presented algorithms for calculating optimal network paths to satisfy service chaining requirements.

Given the service chaining and traffic engineering requirements of recent applications, multicast sessions can no longer be represented as simple spanning trees. Rather, they need to be represented as general graphs. Efficiently forwarding traffic of multicast sessions represented as arbitrary graphs is, however, a challenging problem. One of the main concerns is the *state* that needs to be maintained at routers, which grows linearly with the number of multicast sessions. This state also needs to be frequently updated to handle session changes and network dynamics, which imposes substantial communication and processing overheads, especially on core routers that need to support high-speed links carrying numerous sessions.

In this paper, we address the lack of scalable and generalized multicast forwarding systems for large-scale ISPs. In particular, we propose a *fully stateless* approach, called Yeti, to implement generalized multicast graphs. Yeti supports fast adaptation to network dynamics such as routers joining/leaving sessions and link failures, and it does not impose significant communication overheads. To the best our knowledge, Yeti is the first multicast forwarding system that supports multicast sessions with traffic engineering and service chaining requirements. A high-level overview of Yeti is illustrated in Figure 1.

The main idea of Yeti is to completely move the forwarding information for each graph to the packets themselves as labels. Designing and processing such labels, however, pose key challenges that need to be addressed. First, we need to efficiently encode the graph forwarding information in as few labels as possible. Second, the processing overheads and hardware usage at routers need to be minimized. This is to support many concurrent multicast sessions, and to ensure the scalability of the data plane. Third, forwarding packets should not introduce *ambiguity* at routers. That is, while minimizing label redundancy and overheads, we must guarantee that routers will forward packets *on and only on* the links of the multicast graph. Yeti addresses these challenges.

This paper makes the following contributions:

- We propose a generalized multicast forwarding system that completely eliminates the state at routers; a long-standing problem for multicast. The proposed system supports service chaining and traffic engineering requirements.
- We design a control-plane algorithm to calculate an optimized label for each generalized multicast graph.
- We design an efficient packet processing algorithm for routers to handle labels attached to packets. The algorithm forwards packets only on links of the multicast graph. And it does not introduce any redundant traffic or create loops in the network.
- We present proofs to show the correctness of Yeti.
- We implement Yeti in a hardware testbed using a programmable router (NetFPGA) to demonstrate its practicality. Our results show that Yeti can support high-speed links carrying thousands of multicast sessions.
- We compare Yeti against a rule-based approach implemented using OpenFlow and the closest label-based approaches, LIPSIN [25] and BIER-TE [3], in simulations using real ISP topologies with different sizes. Our simulation results show that unlike Yeti which does not maintain state at any core router, the rule-based approach requires maintaining state at every router in the session and LIPSIN maintains state at about 20% of the routers. In addition, Yeti reduces the label overhead by 65.3%, on average, compared to BIER-TE.

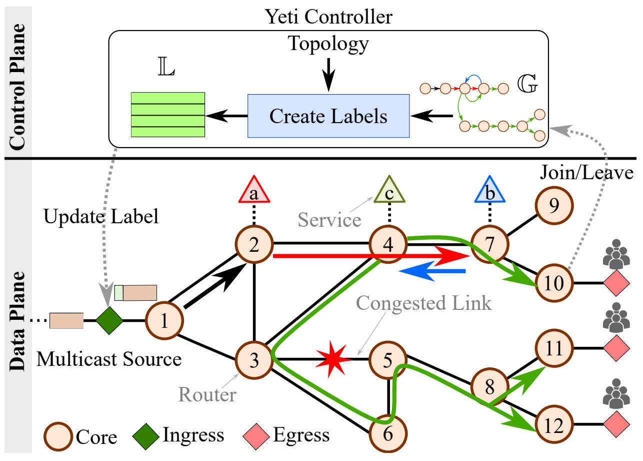


Figure 1: High-level overview of Yeti.

2 Related Work

We divide the related multicast forwarding works in the literature into stateful, stateless, and hybrid approaches.

Stateful Multicast Approaches. These multicast approaches require storing forwarding state about sessions at routers. The traditional IP multicast [31] is an example of such approaches. IP multicast is implemented in core routers produced by most vendors. However, it suffers from scalability issues in real deployments [29]. In particular, the group management and tree construction protocols, e.g., IGMP [28] and PIM [13], require maintaining state at routers for each session, and they generate control messages among routers to refresh and update this state. Thus, in practice, router manufacturers tend to limit the number of multicast sessions [38]. In addition, IP multicast uses shortest paths and cannot implement generalized graphs.

Recent SDN-based protocols, e.g., OpenFlow [17], can implement rule-based approaches, where a controller installs header-matching rules and actions to forward/duplicate packets. Since OpenFlow stores state at every router along the multicast trees, the total forwarding state at routers grows with the number of sessions.

Stateless Multicast Approaches. There are a few recent proposals for designing stateless multicast forwarding systems. For example, BIER [10] encodes global router IDs of tree receivers in the label as a bitmap. BIER supports only shortest paths and cannot realize traffic-engineered ones. A recent amendment to BIER, called BIER-TE [3], supports traffic-engineered trees. BIER-TE maps each bit position in the label to one of the links attached to routers in the network. It encodes the links of a multicast tree as corresponding bit positions in the calculated label. Upon receiving a packet, a BIER-TE router checks the bit positions in the label. If the router matches one of its links in the label, it clears its position in the label and forwards/duplicates the packet on that link. The bitmap structure used in BIER-TE allows it to only implement multicast sessions represented as trees, and it cannot implement general multicast distribution graphs, as such graphs could have cycles to allow the multicast traffic to be processed by the specified set of network services. This is illustrated in the example multicast distribution graph in Figure 1, where packets of the session traverse the link between routers 4 and 7 three times to be processed by the services a→b→c.

Hybrid Multicast Approaches. Yeti is not the first system that moves forwarding information as labels attached to packets. However, prior systems did not support generalized forwarding, and they needed to maintain state at some or all routers belonging to the multicast tree. We refer to these systems as hybrid approaches. For example, the early work by Chen et al. [30] proposed a label-based system that attaches link IDs to every packet in a multicast session, and removes unwanted portions of the label as the packet traverses the network. The processing algorithm in [30] requires maintaining state at every router belonging to a multicast session in order to remove the labels, which is not scalable. Later works, e.g., mLDP [23], enable multicast in label-switched paths (LSPs). mLDP forwards traffic on the shortest paths and thus cannot support traffic-engineered trees. It also requires an additional protocol to distribute labels among routers.

LIPSIN [25] uses a Bloom filter as label to encode global link IDs of a tree. LIPSIN may result in redundant traffic or forwarding loops, because of the probabilistic nature of Bloom filters. Thus, LIPSIN requires an additional protocol where downstream routers notify upstream ones if they falsely receive a packet. This protocol imposes additional state and communication overheads on routers.

Segment routing (SR) [4] is a recent proposal to support traffic-engineered unicast flows. It was later extended to support multicast by considering every tree as a segment in the network. It attaches one label containing the tree ID to packets of a session. Routers along the tree maintain a mapping between that label and the output interfaces. That is, the SR multicast extensions require maintaining state at routers for every tree.

In §5, we compare Yeti versus a rule-based approach implemented in OpenFlow, LIPSIN, and BIER-TE as they are the closest stateful, hybrid, and stateless multicast systems, respectively, that can support traffic engineering requirements.

3 Problem Definition and Solution

We start this section by specifying the considered problem and its challenges. We next describe an overview of Yeti and its main components. This is followed by the details of each component. In the **Appendix** §C, we present an illustrative example of Yeti to demonstrate its details.

3.1 Problem Definition and Challenges

The problem considered in this paper is how to efficiently forward the traffic of a *generalized* multicast session that may need to be processed by an ordered set of network services and/or directed through a specific set of network paths within the ISP network.

For illustration, consider the ISP network in the lower part of Figure 1, which contains ingress, core, and egress routers marked by different shapes and colors. Some of the core routers are connected to servers offering various (virtualized) network services such as intrusion detection and video transcoding. There is a multicast source connected to the ingress router and multiple receivers reachable through the three egress routers. The creator of the multicast session requires the traffic to be processed by the three network services a→b→c in order. In addition, the ISP implements traffic engineering mechanisms for various objectives, e.g., to minimize the maximum link utilization (MLU), which requires the traffic to avoid the link $3 \rightarrow 5$ in this example. The colored arrows in the figure show the different paths taken by the traffic of the multicast session to reach all receivers. These paths form a graph (not tree), which we refer to as the *multicast distribution graph* $\mathbb{G}$. Note that nodes in the distribution graph represent routers, not end users. The top right part of Figure 1 shows the graph for the multicast session marked in the lower part of the figure.

Our problem then becomes how to get routers in the ISP to forward the traffic of a multicast session represented by an arbitrary multicast distribution graph $\mathbb{G}$. Existing algorithms in the literature, e.g., [1, 12], can be used to calculate $\mathbb{G}$ to satisfy various service chaining and traffic engineering requirements; our proposed (forwarding) solution is orthogonal to the calculation of $\mathbb{G}$ and can work with any of them.

The arbitrary nature of the distribution graph makes designing scalable multicast forwarding systems a challenging problem. A possible approach to address this problem is to maintain state (e.g., in the form of match-action rules) at routers. However, as mentioned in §2, maintaining state at routers is not scalable even for traditional multicast forwarding without service chaining and traffic engineering requirements. This is because the state, which grows linearly with the number of multicast sessions, not only consumes the scarce SRAM resources of routers, but it also needs to be frequently updated to handle network failures and session dynamics (e.g., router joining/leaving). The cost of updating the state consists of two factors. First, routers need to process many update (control) messages while processing data packets, which may result in slowing down the data plane operations. Second, frequent state updates negatively impact the network agility and consistency, because the control plane has to *schedule* the updates to corresponding routers to ensure consistency [20], since greedy state updates may result in violating the SLA objectives [6].

To reduce state, a part or all of the forwarding information can be moved to labels which are attached to the packets of multicast sessions, where routers use these labels in the forwarding decisions. However, efficiently representing multicast graphs in compact labels is difficult, especially for multicast sessions that have service chaining and traffic engineering requirements. If not carefully designed, labels representing a multicast graph can grow large in size and hence impose significant communication overheads, and more critically they could introduce *ambiguity* at routers, i.e., routers may not be able to decide which interface(s) to forward the packets on. This may introduce duplicate packets and forwarding loops, which substantially increases the load on the ISP network and wastes its bandwidth and processing resources.

In summary, forwarding generalized multicast graphs presents multiple challenges that Yeti addresses. Specifically, stateful approaches *reduce scalability* as they impose substantial memory and processing overheads on switches. On the hand, current stateless approaches significantly *increases packet sizes* and may *introduce forwarding ambiguity*. This would defeat the main purpose of deploying multicast in the first place. Yeti breaks this long-standing trade-off between scalability, efficiency, and correctness by completely moving the forwarding state into compact labels, and carefully processing them in the data plane.

3.2 Solution Overview

Yeti is a *stateless* multicast forwarding system that efficiently implements general multicast graphs inside a single ISP network; extending Yeti to support multiple ISPs is described in §3.6. As shown in Figure 1, the ISP network has data and control planes. The data plane is composed of routers. Every router is assigned a unique ID, and it maintains two data structures: Forwarding Information Base (FIB) and interface list. FIB provides reachability information between routers using shortest paths. The interface list maintains the IDs of all local interfaces. The control plane (or the *controller*) learns the ISP topology, shortest paths between routers, and interface IDs for every router, which is done using common intra-domain routing and monitoring protocols.

Yeti consists of a centralized controller and a packet processing algorithm. The controller calculates the distribution graph for a multicast session using existing algorithms, e.g., [1, 12], and it creates, using our algorithm in §3.4, an optimized set of labels $\mathbb{L}$ to realize this graph in the data plane. As detailed in §3.3, Yeti defines four types of labels; each serves a specific purpose in encoding the graph efficiently. The controller sends the set of labels $\mathbb{L}$ to the ingress router of the session, which in turn attaches them to all packets of the session. When a graph $\mathbb{G}$ changes (due to link failure or egress router joining/leaving the session), the controller creates a new set of labels and sends them *only* to the ingress router, no other routers need to be updated.

Name	Type	Content	Content Size (bits)
FSP	00	Global router ID	$1 + \lceil \log_2 N \rceil$
FTE	01	Local interface ID	$\lceil \log_2 I \rceil$
MCT	10	Interface bitmap	I
CPY	11	Label range (in bits)	$\lceil \log_2 (N \times \text{size(FTE)}) \rceil$

Table 1: Label types in Yeti. N is the number of routers, and I is the maximum number of interfaces per router.

The packet processing algorithm, described in §3.5, is deployed at core routers. It processes the labels attached to packets and forwards/duplicates packets based on these labels. It also determines the subset of labels to attach to the forwarded packets such that the subsequent routers can realize the distribution graph without any ambiguity, forwarding loops, or redundant traffic. We present a theorem proving the correctness of Yeti in §3.6.

We present an illustrative example in the **Appendix**. This example implements the distribution tree in Figure 1, and it shows the details of creating labels at the controller and processing packets at routers.

3.3 Label Types in Yeti

Yeti is a label-based system. Thus, one of the most important issues is to define the types and structures of labels in order to minimize the communication and processing overheads, while still being able to represent generalized multicast graphs. We propose the four label types shown in Table 1. The first two label types are called *Forward Shortest Path* (FSP) and *Forward Traffic Engineered* (FTE). They are used by routers to forward packets on paths that have no branches. The other two label types are *Multicast* (MCT) and *Copy* (CPY). The MCT label instructs routers to duplicate a packet on multiple interfaces, and the CPY label copies a subset of the labels. Every label consists of two fields: *type* and *content*. The *type* field is used by routers to distinguish between labels during parsing, and the *content* field contains the information that this label carries. The size of a label depends on the size of the *content* field.

We use the example topology in Figure 2 to illustrate the rationale used in defining Yeti labels. In the figure, solid lines denote tree links and the dotted line denotes a link on the shortest path to some destinations. The ISP avoids it because it is congested in this example.

We divide a multicast graph into *path segments* and *branching points*. A path segment is a contiguous sequence of routers without branches. If a path satisfies any sub-sequence of the service chaining requirements of a session, the path segment

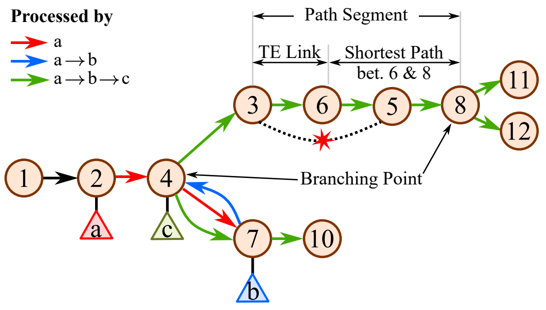


Figure 2: Illustration of path segments and branching points in Yeti. Segment 3 → 8 does not follow the shortest path.

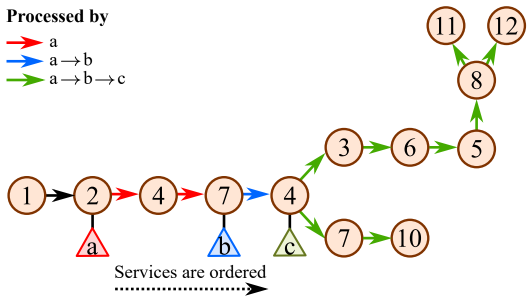


Figure 3: The resulting tree $\mathbb{T}$ for the graph $\mathbb{G}$ in Figure 2.

ends when there is a router with at least one service. A branching point refers to a router that duplicates packets on multiple interfaces. For the example in Figure 2, there are five path segments: {1→2}, {2→4→7}, {7→4}, {7→10}, and {3→6→5→8}. Routers 4 and 8 are branching points.

The basic label in Yeti is FTE, where a router is instructed to forward the packet carrying the label on a specific interface. In many situations, however, packets follow a sequence of routers on the shortest path. For these situations, we define the FSP label, which replaces multiple FTE labels with just one FSP label. An FSP label contains a global router ID, which instructs routers to forward incoming packets on the shortest path to that router. In addition, the first bit in an FSP label indicates whether the packet will be processed at the corresponding router. For example, in Figure 2, instead of using two FTE labels for the links $\{6 \rightarrow 5\}$ and $\{5 \rightarrow 8\}$, Yeti uses one FSP label with destination ID set to node 8. In large topologies, FSP labels significantly reduces label overheads.

FTE and FSP labels can represent path segments, but they cannot handle branching points where packets need to be duplicated on multiple interfaces. Notice that, after a branching point, each branch needs a different set of labels because packets will traverse different routers. To handle branching points, we introduce the MCT and CPY labels. The MCT label instructs routers to duplicate packets on multiple interfaces using a bitmap of size I bits, where I is the maximum interface count per router. The bitmap represents local interface IDs, where the bit locations of the interfaces that the packet should be forwarded on are set to one. The CPY label does not represent a forwarding rule. Instead, it instructs a router to copy a subset of labels when duplicating packets to a branch without copying all labels. Specifically, consider a router that duplicates packets to n branches. The MCT label is followed by n sets of labels to steer traffic in these branches, where every label set starts with a CPY label. The CPY label of one branch contains an offset of label sizes (in bits) to be duplicated to that branch. For example, in Figure 2, router 4 will process an MCT label followed by two CPY labels for the traffic represented with a green arrow, one for each of the two branches. The CPY label content size in Table 1 uses the worst-case scenario. This happens when the graph has the largest diameter, which is $O(N)$ and every link is traffic-engineered, where N is the number of core routers.

3.4 Creating Yeti Labels at the Controller

The ENCODEGRAPH algorithm, shown in Algorithm 1, runs at the controller to create labels. We omit the pseudo code for some functions that the algorithm calls due to space limitations. When the distribution graph $\mathbb{G}$ changes, the algorithm calls the BUILDTREE function to create a tree $\mathbb{T}$ that reflects the order of network services needed before reaching the destinations. Then, the ENCODEGRAPH algorithm calls the CREATELABELS function with the created tree to calculate a new set of labels $\mathbb{L}$ to encode the graph paths and sends them to the ingress router, which attaches them to every packet in the session. The details of our algorithms are as follow.

Building the Tree. The BUILDTREE function traverses the multicast graph and creates a list of tuples for every path and provided services on that path. For example, in Figure 2, the tuple $\langle\{7 \rightarrow 4\}, \{a \rightarrow b\}\rangle$ represents packets traversing the path $\{7 \rightarrow 4\}$ after processed by the services a and b.

The BUILDTREE function then traverses these tuples from the tuple starting with the source node. The function keeps track of the current parent and visited nodes in the tree, and builds the tree $\mathbb{T}$ as follows. For every tuple, the function creates nodes with every router ID and the provided services in that tuple. For the tuple mentioned earlier, the function creates the nodes: $(7, \{a \rightarrow b\})$ and $(4, \{a \rightarrow b\})$. The function adds a node to $\mathbb{T}$ if it did not exist before. Then, the function adds that node to the children of current parent, and sets the new node as the current parent. If a node exists in the tree, the function sets it as the current parent. Figure 3 depicts the resulting tree of the graph in Figure 2.

Creating Labels. The CREATELABELS algorithm divides $\mathbb{T}$ into segments and branching points. The algorithm calculates FSP and FTE labels for every segment, and MCT and CPY labels at branching points. The label order is important because it reflects which routers process what labels. The algorithm traverses the core routers of $\mathbb{T}$ in a depth-first search order starting from the core router connected to the ingress router.

Algorithm 1 Encode a multicast graph into labels.

```
Input:  $\mathbb{G}$: multicast graph
Input:  $S$: ordered list of services in the session
Input:  $\mathbb{P}$: shortest path map
Output: $\mathbb{L}$: labels to be sent to the ingress router
1:  function EncodeGraph($\mathbb{G}$, $S$, $\mathbb{P}$)
2:      $\mathbb{T}$ = BuildTree($\mathbb{G}$, $S$)
3:      return CreateLabels($\mathbb{G}.src$, $\mathbb{T}$, $S$, $\mathbb{P}$)
4:  function CreateLabels($source$, $\mathbb{T}$, $S$, $\mathbb{P}$)
5:      $\mathbb{L} = \{\}$, $pth\_seg = \{\}$, $stack = \{source\}$
6:      while ($stack$.size() $> 0$) do
7:          $r = stack$.pop() // a router in $\mathbb{T}$
8:          // Services provided by $r$ for the session
9:          $srv = S$.at($r$)
10:         $core\_children = \mathbb{T}$.core_children($r$)//core routers
11:         $children = \mathbb{T}$.children($r$) //core and egress routers
12:         // Build a path segment $pth\_seg$
13:         if ($core\_children$.size() $== 1$) then
14:             $pth\_seg$.append($r$); $stack$.push($children[0]$)
15:         // Handle a path segment (create FSP & FTE)
16:         // $S[0]$ is the next expected service
17:         if ($core\_children$.size() $== 0$ or $S[0] \in srv$) then
18:             $pth\_seg$.append($r$)
19:             $lbls$ = CalcSegmentLbl($\mathbb{T}$, $pth\_seg$, $\mathbb{P}$)
20:             $\mathbb{L}$.push($lbls$); $pth\_seg = \{\}$
21:             $S$.remove($srv$)
22:         // Handle branching point (create MCT & CPY)
23:         else if ($children$.size() $> 1$) then
24:             if ($pth\_seg$.size() $> 0$) then
25:                 $pth\_seg$.append($r$)
26:                 $lbls$=CalcSegmentLbl($\mathbb{T}$, $pth\_seg$, $\mathbb{P}$)
27:                 $\mathbb{L}$.push($lbls$); $pth\_seg = \{\}$
28:             $\langle mct\_lbl, cpy\rangle$ = CreateMCT($children$)
29:             $\mathbb{L}$.push($mct\_lbl$)
30:             if ($cpy$) then // Creating CPY labels
31:                 for ($c \in children$) do
32:                     // A recursive call for each branch
33:                     $br\_lbls$ = CreateLabels($c$, $\mathbb{T}$, $S$, $\mathbb{P}$)
34:                     $offset$ = CalcLabelSize($br\_lbls$)
35:                     $\mathbb{L}$.push(CPY($offset$));$\mathbb{L}$.push($br\_lbls$)
36:     return $\mathbb{L}$
```

It keeps track of the router r that is being visited, and one path segment (pth_seg). Once a router r is visited, if r has only one core child (Line 13 in Algorithm 1), this means that r belongs to the current segment. The algorithm then appends r to pth_seg, and pushes its child to the stack to be traversed later. For example, the algorithm pushes routers 3, 6, 5 and 8 in Figure 3 to pth_seg because each of them has only one child. If r has no core children or it provides some services (has a path segment), or r has more than one child (has a branching point), the algorithm calculates labels as follows.

Handling Path Segments. The CreateLabels algorithm creates a label for a path segment when pth_seg ends. This happens in three cases. First, when r is connected to an egress router (e.g., router 10 in Figure 3). Second, when r is a branching point and pth_seg is not empty (e.g., router 8 in Figure 3). Third, when r provides at least on service (e.g., router 2 in Figure 3). In all cases, the algorithm appends r to pth_seg and calculates FSP and FTE labels using CalcSegmentLbl.

CalcSegmentLbl takes as inputs a tree $\mathbb{T}$, a path segment pth_seg and the shortest path map $\mathbb{P}$, and calculates the FSP and FTE labels of the given pth_seg. It divides pth_seg into two sub-paths: one that follows the shortest path, and one that does not. It then recursively applies the same to the latter sub-path. Specifically, CalcSegmentLbl concurrently traverses pth_seg and the shortest path between source and destination. It stops when the traversal reaches a router in pth_seg that does not exist in the shortest path. This means that this router does not follow the shortest path, hence, it adds an FSP label for the previous router. If pth_seg has routers that do not follow the shortest path, CalcSegmentLbl adds an FTE label and recursively calls itself using a subset of pth_seg that is not traversed so far. CalcSegmentLbl does not generate two consecutive FSP labels. When it calculates an FSP label, it either terminates, or creates an FTE label followed by a recursive call.

For the example in Figure 3, the CalcSegmentLbl algorithm processes the segment $\{3 \rightarrow 6 \rightarrow 5 \rightarrow 8\}$ as follows. It finds that the link (3, 6) is not on the shortest path from 3 to 8. It calculates an FTE label for this link, and recursively calls itself with the sub-path $\{6 \rightarrow 5 \rightarrow 8\}$ as an input, for which the algorithm creates an FSP label with router ID 8.

Handling Branching Points. The CreateLabels algorithm calculates MCT and CPY labels at branching points. The algorithm calls CreateMCT that returns MCT label and a boolean value cpy indicating whether CPY labels are required. To create an MCT label, CreateMCT initializes an empty bitmap of width $I+1$ (I is the maximum interface count per router). For every child c of r, it sets the bit location in this bitmap that represents the interface ID between r and c. It checks if CPY labels are needed as follows. If any child c has at least one core child, this means that this core child needs labels to forward/duplicate packets. Otherwise, if all children have no other core children, the router r is either directly connected to an egress router, or its children are connected to egress routers. Thus, these routers do not need more labels and Yeti does not create CPY labels for these branches. For example, at router 4 in Figure 3, core children 3 and 7 have other core children which are 6 and 10, respectively. Hence, two CPY labels are created for the two branches at 4. The algorithm does not create CPY labels at router 8, because its core children 11 and 12 have no other core children.

Recall that a CPY label copies a subset of labels at a specific branch. If CPY labels are needed at the branching point and r has n children/branches, the MCT label is followed by

n CPY labels, and every CPY label is followed by labels to forward packets on the corresponding branch. Specifically, the algorithm iterates over the children of r. For every child c, the algorithm adds an FSP label if the child provides services. Then, the algorithm recursively calls CREATELABELS to create labels of the corresponding branch (Line 33). The created CPY label for a branch contains the size of this branch labels in bits to be copied. We calculate this size by accumulating the size of every label type in *br_lbls* (Line 34).

Time and Space Complexities. In the worst-case scenario, when every router is a branching point, the ENCODEGRAPH algorithm needs to create labels for each branch. Thus, the time complexity of the ENCODEGRAPH algorithm is $O(K^2N^2 + M)$, where N is the number of routers, M is the number of links, and K is the maximum service chain length. We note that the values of N, M and K are not large for realistic ISP networks. The number of ISP routers N is in the range of 10's–100's [9,18], most ISP networks are sparse with number of links M ranging from 500 to around 2,000, and the length of service chains K ranges from 2–10 [1]. Given these practical values, the ENCODEGRAPH algorithm can easily run on a commodity server. Notice that the CALCSEGMENTLBL algorithm processes the segment after all routers of that segment is traversed. Thus, it only adds linear overhead to the first term of the time complexity. The space complexity of the ENCODEGRAPH algorithm is $O(N^2D)$, where D is the diameter of the network.

3.5 Processing Yeti Packets

The proposed packet processing algorithm is to be deployed at core routers, and it processes Yeti packets. This is done by setting a different Ethernet type for Yeti packets at ingress routers. A core router checks the Ethernet type of incoming packets, and invokes the processing algorithm if they are Yeti packets. The algorithm forwards/duplicates packets and it removes labels that are not needed by next routers. It also copies a subset of labels at branching points.

The packet processing algorithm works as follows. If the packet has no labels, this means the packet reached a core router that is attached to an egress router. So, the packet is forwarded to that egress router. Otherwise, the algorithm processes labels according to the following cases:

(1) FSP Label. If the FSP label content is not the receiving router ID, it means that this router belongs to a path segment. The algorithm then forwards the packet along the shortest path based on the underlying intra-domain routing protocol *without* removing the label. If the FSP content equals the router ID, this means that the path segment ends at this router. The algorithm first checks whether the packet needs to be processed by services connected to that router. If the first bit is set, then the packet is forwarded to the datacenter. Otherwise, the algorithm removes the current label and calls the packet processing algorithm again to process next labels. This is because the packet may have other labels.

(2) FTE Label. The algorithm removes the label, extracts the local interface ID, and forwards the packet on that interface.

(3) MCT Label. The algorithm first copies the original labels, and removes the labels from the packet. It then extracts the MCT content into *mct*. The MCT label contains the interface ID bitmap (*mct*.intfs) and whether it is followed by CPY labels (*mct*.has_cpy). The algorithm iterates over the router interfaces in ascending order of their IDs. It locates the interfaces to duplicate the packet on. For every interface included in the MCT label, the algorithm clones the packet. If the MCT label is followed by CPY labels, the algorithm removes the corresponding CPY label, extracts the following labels based on the offset value, and forwards the cloned packet on the corresponding interface.

Time and Space Complexities. The time complexity of the algorithm is $O(I)$, where I is the maximum interface count per router. The algorithm does not require additional space at routers.

3.6 Analysis and Practical Considerations

Analysis. The following theorem proves the correctness of Yeti. That is, Yeti forwards packets on and only on links belonging to the multicast graph. This is a critical objective for large-scale multicast sessions, as redundant traffic wastes network resources and overloads routers. Due to space limitation, we show the full proof in the Appendix §A.

Theorem 1 (Correctness). *Yeti forwards packets on and only on links that belong to the multicast graph.*

Proof Sketch. Yeti guarantees correctness by creating an ordered set of labels to realize the given multicast distribution graph. We analyze the order and type of the created labels and prove that they do not result in forwarding loops or redundant traffic while delivering the traffic to all receivers in the multicast session. □

Practical Considerations. The description of Yeti has focused on offering a scalable multicast service within a *single* ISP using programmable routers. In the Appendix §B, we describe simple techniques to enable Yeti across multiple ISPs and its incremental deployment.

4 Evaluation in a Testbed

We present a *proof-of-concept* implementation of the proposed multicast forwarding system, and we conduct experiments in a testbed with a NetFPGA programmable router.

In addition, since switches that support the P4 programming language [21], e.g., Tofino, are getting popular in practice, we also implemented Yeti in P4. We obtained a license for the Intel P4 software development environment (SDE) version 9.5.0, which contains various tools, including a P4

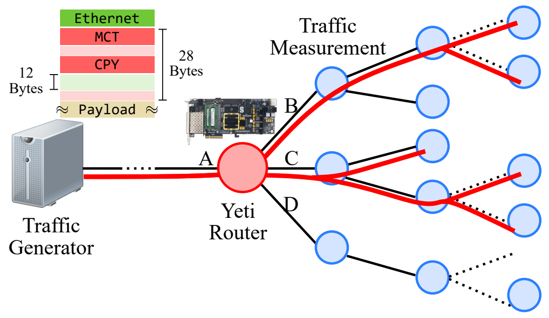


Figure 4: Setup of our testbed.

Latency (μsec)	**50th %**	**95th %**	**99.9th %**
Yeti	960.4	973	1,042
MAC forwarding	960.3	972.9	1,040
Difference	0.1	0.1	2

Table 2: Packet latency (in μsec) measured in our testbed.

compiler (`bf-p4c`) and a switch model. The compiler produces code that runs on Tofino switches. We validated our implementation using the switch model included in the SDE. The details of the P4 implementation of Yeti are presented in Appendix §D.

4.1 Testbed Setup

The testbed, shown in Figure 4, has a Yeti Router representing a core router in an ISP topology that receives and processes packets of concurrent multicast sessions. We implemented the Yeti Router in a programmable processing pipeline using NetFPGA SUME [19], which has four 10GbE ports. The testbed also has a 40-core server with an Intel X520-DA2 2x10GbE NIC, which is used to generate traffic of multicast sessions at high rate using MoonGen [14].

Our router implementation is based on the open source project in [34]. This project contains three main Verilog modules: `input_arbiter`, `output_port_lookup` and `output_queues`. Our implementation modifies the last two modules in the router as follows. The `output_port_lookup` module is modified to read the first label to decide which ports to forward/duplicate the packet on. For each duplicated packet, it decides the labels to be detached and maintains this information in the packet metadata. We also modified the `output_queues` module to runs at every output queue of the router, and detach labels that are not needed in the outgoing packets.

4.2 Experiments and Results

We transmit labelled packets of concurrent multicast sessions at the maximum link speed in our testbed (10 Gbps) from the traffic-generating server to the Yeti Router. We stress Yeti by transmitting traffic that requires copying and rearranging labels for each packet. In every experiment, we attach labels with different sizes to each packet. These labels contain MCT and CPY labels. The MCT label instructs the Yeti Router to duplicate packets on two ports B and C in Figure 4. We report the results for a sample label size of 28 bytes. This is because, as we show in §5, most of the packets have this label size for traffic engineering and service chaining scenarios in different ISP networks. The CPY labels instruct Yeti to copy 12 and 16 bytes to ports B and C, respectively. We measure the outgoing traffic on port B. The main parameter that we control is the packet size, which we vary from 64 to 1024 bytes. We report three important metrics for the design of high-end routers: packet latency, resource usage, and throughput.

Latency. We report the packet processing latency at port B in Figure 4 when the Yeti Router processes CPY labels (i.e., the worst-case scenario in terms of processing). We measure the latency by timestamping each packet at the traffic generator, and taking the difference between that timestamp and the time the packet is received at port B. We use the Berkeley Extensible Software Switch [15] to timestamp packets. Since it may add overheads while timestamping and transmitting packets, we compare the latency of Yeti processing against the basic forwarding in the same testbed, which is done by matching the fixed-length MAC address. Table 2 shows multiple statistics of the packet latency for both Yeti and unicast forwarding when the packet size is 1,024 bytes. The table shows that the latency of Yeti processing under stress is close to the simple unicast forwarding. For example, the difference of the 95th percentiles of packet latency is only 0.1 μsec when the packet size is 1,024 bytes.

Resource Usage. We measure the resource usage of the packet processing algorithm, in terms of the number of used look-up tables (LUTs) and registers in the Yeti Router, which are generated by the Xilinx Vivado tool after synthesizing and implementing the project. Our implementation uses 12,677 slice LUTs and 1,701 slice registers per port. Relative to the available resources, the used resources are only 3% and 0.2% of the available LUTs and registers, respectively. Thus, Yeti requires small amount of resources while it can forward traffic of many concurrent multicast sessions.

Throughput. In Figure 5, we compare the rate of incoming packets to the Yeti Router versus the rate of packets observed at port B. The figure shows that the numbers of transmitted and received packets per second are the same (i.e., no packet losses). The figure also shows that our algorithm can sustain the required 10 Gbps throughput for all packet sizes.

We realize that core routers have large port density and high speeds. We believe that Yeti can achieve line-rate performance in these routers, because Yeti processes each incoming packet independently and adds small processing latency per packet as shown in Table 2.

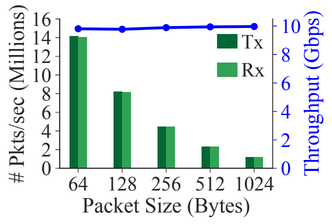


Figure 5: Throughput of received traffic from our testbed.

5 Evaluation using Simulation

We analyze the performance of Yeti and compare it against the closest multicast approaches using simulation.

5.1 Simulation Setup

Simulator. We implemented a Python-based simulator to compare the performance of different multicast systems in large setups using realistic ISP topologies. The simulator has two components. The first acts as the Yeti controller in Figure 1. When this component receives an egress router event, it updates the corresponding multicast graph, and then generates labels using Algorithm 1. The second component simulates the packet processing algorithm in §3.5. The simulator also implements prior systems for comparisons.

ISP Topologies. We use 14 topologies in the Internet Topology Zoo dataset [35]. This dataset represents a wide range of actual ISPs, where the number of routers ranges from 36 to 197, and the number of links ranges from 152 to 486.

Multicast Sessions. We simulate dynamic and diverse multicast sessions. The source of each session is randomly selected from one of the ISP routers. The session bandwidth is randomly chosen from the set $\{0.5, 1, 2, 5, 10\}$ Mbps, which represents the bandwidth values of different types of applications. The session duration is randomly assigned to a value from $\{10, 20, 40, 60, 80, 100, 120\}$ minutes. These values reflect a wide range of short to long multicast sessions. In addition, while the session is active, we make its receivers join and leave according to random events generated from a Poisson distribution, where 60% of these are join events and 40% are leave events. We make the receiver join rate 50% higher than the leave rate to incrementally stress the system with more multicast receivers as the time passes.

Each multicast session requires a set of network services, or service chain. We vary the length of the service chain from 3 to 5 as these lengths represent common service usage patterns [1, 42]. To represent practical deployment of services in ISPs, we divide services to essential and supplementary according to their popularity [42]. Essential services such as firewalls are deployed at all ISP locations, whereas supplementary services such as video encoders are only deployed at some of the ISP locations. To stress our system, we set the percentage of ISP locations that provide supplementary services to only 25%. Each multicast session includes two randomly chosen essential services and the rest of the services are supplementary ones, also randomly chosen.

Since Yeti does not dictate how multicast graphs are computed, we use the algorithms in [12] and [1] to calculate the graphs based on the traffic engineering and service chaining requirements, respectively.

Experiments and Statistics. We simulate the operation of an ISP managing concurrent and dynamic multicast sessions over an extended period of time (24 hours), where about 200,000 sessions are created over the simulation period. Specifically, we first choose one of the 14 ISP topologies and generate the multicast sessions using the characteristics described above. Then, we repeat the experiment for the same ISP topology five times, starting from different seeds for the random distributions. Thus, for each ISP topology, we collect and analyze statistics from about 1M randomly generated, diverse, and dynamic multicast sessions. Then, the whole process is repeated for each of the 14 ISP topologies.

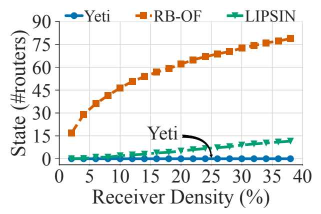
We report the 95-percentile of various performance metrics in the following subsections, as it reflects the performance over extended number of sessions. We present representative samples of our figures, using the ISP topologies with the largest and median numbers of routers. We also present averages and normalized averages (per router) across all ISP topologies, to infer the performance in general settings. When we present the (normalized) averages, we report the standard deviation in each case preceded by $\pm$.

Yeti is the first multicast forwarding system to support service chaining and traffic engineering. Thus, we first compare a simpler version of Yeti against the state-of-art systems for multicast sessions with traffic engineering requirements as these cannot support service chaining. Then, we analyze the performance of Yeti for multicast sessions with traffic engineering and service chaining requirements.

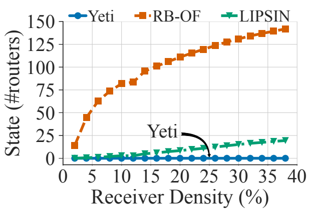
5.2 Yeti vs Stateful and Hybrid Approaches

We compare Yeti versus the closest stateful and hybrid multicast forwarding systems, which are OpenFlow [17] and LIPSIN [25]. We implemented a rule-based multicast forwarding system using OpenFlow [17] (referred to as RB-OF), because rule-based is a general packet processing model that is supported in many networks. The rule-based system is stateful as it installs match-action rules in routers.

LIPSIN is a hybrid approach that encodes the tree link IDs of a session using a Bloom filter. For every link, the LIPSIN controller maintains D link ID tables with different hash values. LIPSIN creates the final label by selecting the table that results in the minimum false positive rate. Since LIPSIN may result in false positives, each router maintains state about incoming links and the label that may result in loops for every session passing through this router. We set the filter size of LIPSIN to the 99th-percentile of the label



(a) #Routers=110



(b) #Routers=197

Figure 6: State size.

size of Yeti. This enables LIPSIN to encode a large number of links per tree in its labels. We use the same parameters proposed for LIPSIN: we set D to 8 tables and use five hash functions per link. We use Bloom filters and Murmurhash3 hashing functions.

State Size. Figure 6 shows the 95-percentile of the state size per multicast session as the density of receivers varies, which corresponds to the number of core routers needed to maintain state. The results are shown for the median and largest topologies with sizes 110 and 197 routers, respectively. The results for other topologies are similar (shown in Appendix §E).

First, notice that Yeti does not require any state at any core router. In contrast, RB-OF needs to maintain state at each router and that state increases with the topology size as well as the density of receivers in each multicast session. For example, the state size increases from 80 to 130 rules when the receiver density increases from 10% to 30%. LIPSIN, on the other hand, needs to maintain state at up to 20 routers when the receiver density is 40%. In this case, multicast graphs span 50% of the routers. That is, LIPSIN needs to maintain state at up to 20% of the routers in the multicast graph.

State Update Rate. Maintaining state at routers does not only consume their limited SRAM, but also increases the overheads of updating this state at routers when the distribution graph changes [6, 20].

We assess the average number and percentage of routers to be updated when a single multicast tree changes, and show the results for sample ISP topologies with various sizes in Table 3. Recall that the Yeti controller needs to update *one and only one* (ingress) router when a session changes, which is independent of the topology size. The state for RB-OF and LIPSIN, on the other hand, grows with the topology size and number of receivers in the multicast sessions. Thus, as Table 3 shows, RB-OF and LIPSIN controllers need to update up to 103.3 and 19.6 core routers per each distribution graph change, respectively. That is, Yeti reduces the number of routers to be updated by up to 103X and 20X compared to RB-OF and LIPSIN, respectively.

To demonstrate the generality of Yeti performance, we calculate the average percentage of routers in the ISP topology to be updated for each change in the multicast distribution graph, which is taken over all 14 ISP topologies. We present the results in the last row in Table 3, which show that Yeti reduces the average state update rate by 97.5% and 87.1% compared to RB-OF and LIPSIN, respectively.

ISP Size	RB-OF	LIPSIN	Yeti	Saving (%)
49	18.9	4.9	1.0	94.7 / 79.7
84	55.9	12.3	1.0	98.2 / 92.0
125	76.8	19.6	1.0	98.7 / 95.0
158	91.2	11.8	1.0	98.9 / 91.5
197	103.3	18.7	1.0	99.0 / 94.7
Norm. Avg. (%)	48±10	9±3	1±0.6	**97.5 / 87.1**

Table 3: Number of routers that need to be updated for each change in the multicast distribution graph. The shown savings in the right most column are relative to RB-OF and LIPSIN, respectively. The averages in the bottom row are computed across all 14 ISP topologies and normalized by the number of routers in each topology, and they represent the average percentage of routers to be updated for each change.

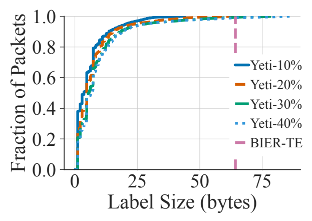
In summary, compared to stateful and hybrid approaches, Yeti scales well and can handle dynamic multicast sessions, as it does not require maintaining state at any router, and it significantly reduces the need for frequent state updates.

5.3 Yeti vs A Stateless Approach

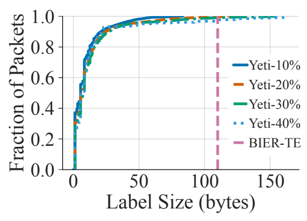
We compare Yeti against BIER-TE [3], which is a recent label-based multicast forwarding system. We implemented the basic features of BIER-TE as described in [3], which are the bit positions for the forward-connected, forward-routed and local-decap actions. This means that we *conservatively* report the minimum size of BIER-TE labels for every ISP topology. Since Yeti and BIER-TE are stateless and both use labels, we only analyze the label size and its imposed total overhead.

We first asses the label size per packet for multiple receiver densities. We present the CDF of the label size for the median and largest topologies in Figure 7; the results for other topologies are given in Appendix §E. Figure 7 indicates that the label size for Yeti is much smaller than that of BIER-TE in practical scenarios. For example, for the topology with the median number of routers (110 routers), Figure 7a, and receiver density of 30%, Yeti reduces the label size for 50% of the packets by 91.6% compared to BIER-TE. Moreover, the label size in Yeti for 90% of the packets is less than 19 bytes, while the label size of BIER-TE is 64 bytes. For the largest topology in our dataset (197 routers) and for receiver density of 30%, Figure 7b shows that the label size in Yeti is 15X smaller than BIER-TE for 50% of the packets.

We further analyze the behavior of Yeti and the dynamics of its label size as packets traverse the network. In Figure 8,

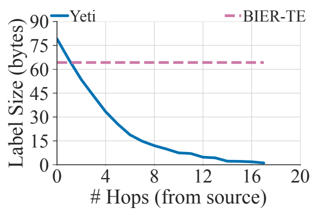


(a) #Routers=110

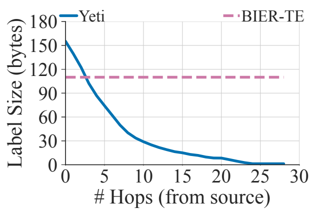


(b) #Routers=197

Figure 7: Label size CDF for different receiver densities.



(a) #Routers=110



(b) #Routers=197

Figure 8: Label size as packets traverse the network.

we plot the average label size for Yeti and BIER-TE versus the number of hops from source, for the median and largest topologies, more results for other topologies are given in Appendix §E. The figure shows that the label size for BIER-TE depends on the topology size, it is 64 and 110 bytes, for the median and largest topologies, respectively. In contrast, the label size in Yeti decreases quickly as the packet moves away from the source. For example, in Figure 8a, the label size of Yeti is reduced by 16.9% and 67.8% after traversing 1 and 5 hops, respectively. The label size of Yeti becomes smaller than that of BIER-TE after traversing 2 hops only, and Yeti reduces the label size by 87.5% after traversing the first 50% of the hops.

Finally, we assess the total end-to-end label overhead of Yeti and BIER-TE, which we define as the label size multiplied by the number of network hops the packet traverses; this is the area under the curves in Figure 8. Table 4 summarizes the label overhead in bytes for multiple sample topologies. The results show that Yeti achieves substantial savings, up to 70.2%, compared to BIER-TE. In addition, we report the average label overhead per router across all 14 ISP topologies. On average, Yeti needs only 4 bytes per router to forward packets of multicast sessions, whereas BIER-TE requires 11.4 bytes per router, that is Yeti achieves an average saving of 65.3% in the label overhead.

In summary, the label size in Yeti quickly decreases as packets move towards the multicast destinations, because routers copy only a subset of labels for every branch. This results in substantial savings in the label overheads compared to the closest label-based multicast forwarding system, BIER-TE. In addition, BIER-TE cannot satisfy the service chaining requirements for multicast traffic, while Yeti can.

5.4 Analysis of Yeti

We analyze the performance of Yeti in terms of effectiveness of FSP labels, label size, processing overheads and running time to satisfy various forwarding requirements.

Effectiveness of FSP Labels. We study the importance of the proposed FSP label type. Recall that a single FSP label represents multiple routers in a path segment if that segment is on the shortest path. To show the importance of FSP label across different topologies, we plot the average FSP saving for sample topologies in Figure 9a as well as the average over all ISP topologies. We observe consistent savings across the topologies which range from 4 to 9 routers per FSP label. That is, one FSP label saves 4–9 other labels, reducing the label overhead by up to 9X. The average FSP saving is 5.8±1.8 routers.

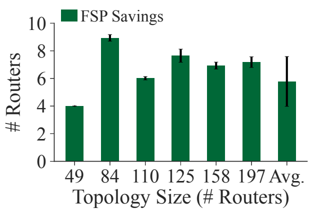
ISP Size	BIER-TE	Yeti	Saving (%)
49	299.3	89.3	70.2
84	1,283.3	508.5	60.4
125	1,761.5	588.5	66.6
158	3,123.0	1,176.2	62.3
197	3,190.0	1,062.6	66.7
Norm. Avg. (bytes/router)	11.4±4.8	4.0±1.8	**65.3**

Table 4: Label overhead in bytes for Yeti and BIER-TE.

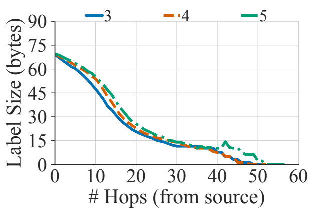
Next, we asses the FSP savings to satisfy service chaining requirements with different chain lengths in Table 5. As the results show, FSP labels efficiently encode path segments in the distribution graphs. For instance, an FSP label can encode about 6 routers on average for typical chain lengths.

Label Size. In Figure 9b, we plot the label size of Yeti versus the number of hops from source for service chaining requirements. The figure shows the results for the median topology of size 110. The results indicate that the label size of Yeti quickly decreases for all considered chain lengths as packets traverse towards the destinations. For instance, the label size decreases by 23% after traversing the first 10 hops when the chain length is 4. In addition, although the used routing policy [1] increases the number of hops to satisfy all service chaining requirements, Yeti is able to encode these large graphs in relatively small labels.

Processing Overhead. In Figure 10a, we plot the number of copy operations per packet versus the receiver density for multiple topology sizes to realize traffic engineering. The results for other ISP topologies are plotted in Appendix §E. The figure shows that the additional work per packet is small. The number of copy operations increases as the receiver density increases because the multicast distribution graphs have

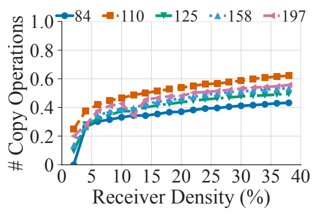


(a) FSP savings

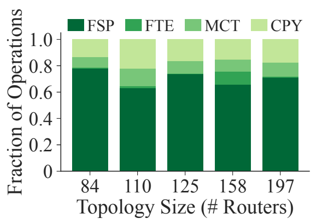


(b) Label size per hop

Figure 9: Analysis of FSP savings and label size of Yeti to satisfy traffic engineering and service chaining requirements.



(a) # copy operations



(b) Distribution of operations

Figure 10: Analysis of processing overheads of Yeti.

ISP	**Service Chain Length**		
Size	**3**	**4**	**5**
49	4.2±0.4	4.0±0.0	4.0±0.0
84	11.2±1.7	11.0±1.4	11.0±1.4
125	9.2±1.2	9.0±1.3	9.2±1.2
158	8.0±1.1	8.0±1.1	8.0±1.1
197	7.2±1.2	7.2±1.2	7.2±1.2
Avg.	6.4±2.2	6.3±2.2	6.4±2.2

Table 5: Number of traversed routers per FSP label to satisfy service chaining.

more branches in this case. However, the processing overhead increases slowly. For instance, in the 84-router topology, the average number of copy operations increases from 0.4 to 0.62 per packet when the receiver density increases from 5% to 38%. This pattern applies for other topologies as well. That is, Yeti routers scale in terms of processing as the network load increases.

We next present the distribution of FSP, FTE, MCT and CPY labels per router for the five topologies in Figure 10b. The figure shows that the fraction of processed CPY labels (the most expensive) per Yeti router across all sessions is small compared to other label types. For example, only 17% of the labels being processed at a Yeti router are CPY labels for the largest topology of 197 routers.

For service chaining, Yeti incurs a similar distribution of operations to Figure 10b. For instance, the fraction of processed CPY labels is 18.7% for the topology of size 110 when the chain length is 3. For a longer chain of length 5, the fraction of CPY labels increases slightly to 21% to satisfy all service chains. On average, the fractions of CPY labels per ISP topology are 19.2%±2.8%, 19.6%±2.8%, and 19.6%±2.3% for chain lengths of 3, 4 and 5, respectively. We present the average number of copy operations and distribution of all operations in Appendix §E, where the averages are taken over chain lengths.

Running Time of Yeti Controller. We ran the proposed CREATELABELS algorithm on a workstation with four 3.3 GHz cores and 32 GB of memory, and we measured the running time of creating labels per graph update at the controller. The running time varied from 4 msec to 10 msec based on the topology size. For the largest topology of size 197, the controller spends only about 10 msec per graph update to create the labels for the largest session. Thus, the proposed label creation algorithm is practical, can run on commodity servers, and it supports frequent graph updates and network dynamics.

In summary, Yeti imposes small label and processing overheads while satisfying service chaining and traffic engineering requirements.

6 Conclusions

We proposed an efficient, stateless, multicast forwarding system called Yeti that implements generalized multicast graphs in ISP networks. Unlike current rule-based multicast systems, Yeti does not require maintaining any state at routers. And unlike other label-based multicast systems, Yeti can direct traffic on arbitrary network paths to meet traffic engineering and service chaining requirements while reducing the label size significantly. The novel aspects of Yeti include (1) supporting general traffic forwarding requirements, (2) guaranteeing correctness, (3) composing small labels, and (4) processing labels efficiently at routers. We implemented Yeti in a programmable router and evaluated its performance. Our experiments show that Yeti can achieve line-rate performance while using a small amount of hardware resources. In addition, we conducted extensive simulations using real ISP topologies, and compared it versus the state-of-art approaches. Our results show that Yeti outperforms the other approaches by wide margins for all considered metrics.

Acknowledgments

We thank our shepherd, Behnaz Arzani, and the anonymous reviewers for their comments. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] K. Diab, C. Lee, and M. Hefeeda. Oktopus: Service chaining for multicast traffic. In *Proc. of IEEE ICNP'20*, pages 1–11, Madrid, Spain, October 2020.

[2] Anja Feldmann, Oliver Gasser, Franziska Lichtblau, Enric Pujol, Ingmar Poese, Christoph Dietzel, Daniel Wagner, Matthias Wichtlhuber, Juan Tapiador, Narseo Vallina-Rodriguez, Oliver Hohlfeld, and Georgios Smaragdakis. The lockdown effect: Implications of the covid-19 pandemic on internet traffic. In *Proc. of ACM IMC'20*, page 1–18, Virtual Event, October 2020.

[3] Toerless Eckert, Gregory Cauchie, and Michael Menth. Tree Engineering for Bit Index Explicit Replication (BIER-TE). Internet-Draft draft-ietf-bier-te-arch-08, Internet Engineering Task Force, July 2020. Work in Progress.

[4] Clarence Filsfils, Stefano Previdi, Les Ginsberg, Bruno Decraene, Stephane Litkowski, and Rob Shakir. Segment Routing Architecture. RFC 8402.

[5] B. Ren, D. Guo, G. Tang, X. Lin, and Y. Qin. Optimal service function tree embedding for nfv enabled multicast. In *Proc. of IEEE ICDCS'18*, pages 132–142, Vienna, Austria, July 2018.

[6] Jiaqi Zheng, Bo Li, Chen Tian, Klaus-Tycho Foerster, Stefan Schmid, Guihai Chen, and Jie Wu. Scheduling congestion-free updates of multiple flows with chronicle in timed sdns. In *Proc. of IEEE ICDCS'18*, pages 12–21, Vienna, Austria, July 2018.

[7] S. H. Chiang, J. J. Kuo, S. H. Shen, D. N. Yang, and W. T. Chen. Online multicast traffic engineering for software-defined networks. In *Proc. of IEEE INFOCOM'18*, pages 414–422, Honolulu, HI, April 2018.

[8] Aravindh Raman, Gareth Tyson, and Nishanth Sastry. Facebook (A)Live?: Are Live Social Broadcasts Really Broadcasts? In *Proc. of WWW'18*, page 1491–1500, Lyon, France, April 2018.

[9] Victor Heorhiadi, Sanjay Chandrasekaran, Michael K. Reiter, and Vyas Sekar. Intent-driven composition of resource-management sdn applications. In *Proc. of ACM CoNEXT'18*, pages 86–97, Heraklion, Greece, 2018.

[10] IJsbrand Wijnands, Eric C. Rosen, Andrew Dolganow, Tony Przygienda, and Sam Aldrin. Multicast Using Bit Index Explicit Replication (BIER). RFC 8279.

[11] L. H. Huang, H. C. Hsu, S. H. Shen, D. N. Yang, and W. T. Chen. Multicast traffic engineering for software-defined networks. In *Proc. of IEEE INFOCOM'16*, pages 1–9, San Francisco, CA, April 2016.

[12] M. Huang, W. Liang, Z. Xu, W. Xu, S. Guo, and Y. Xu. Dynamic routing for network throughput maximization in software-defined networks. In *Proc. of IEEE INFOCOM'16*, pages 1–9, San Francisco, CA, April 2016.

[13] Bill Fenner, Mark J. Handley, Hugh Holbrook, Isidor Kouvelas, Rishabh Parekh, Zhaohui (Jeffrey) Zhang, and Lianshu Zheng. Protocol Independent Multicast - Sparse Mode: Protocol Specification. RFC 7761.

[14] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. Moongen: A scriptable high-speed packet generator. In *Proc. of ACM IMC'15*, pages 275–287, Tokyo, Japan, October 2015.

[15] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. Softnic: A software nic to augment hardware. Technical Report UCB/EECS-2015-155, EECS Department, University of California, Berkeley, May 2015.

[16] S. H. Shen, L. H. Huang, D. N. Yang, and W. T. Chen. Reliable multicast routing for software-defined networks. In *Proc. of IEEE INFOCOM'15*, pages 181–189, Hong Kong, China, April 2015.

[17] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proc. of the IEEE*, 103(1):14–76, Jan 2015.

[18] Renaud Hartert, Stefano Vissicchio, Pierre Schaus, Olivier Bonaventure, Clarence Filsfils, Thomas Telkamp, and Pierre Francois. A declarative and expressive approach to control forwarding paths in carrier-grade networks. In *Proc. of ACM SIGCOMM'15*, pages 15–28, London, United Kingdom, 2015.

[19] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore. Netfpga sume: Toward 100 gbps as research commodity. *IEEE Micro*, 34(5):32–41, September 2014.

[20] Xin Jin, Hongqiang Harry Liu, Rohan Gandhi, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Jennifer Rexford, and Roger Wattenhofer. Dynamic scheduling of network updates. In *Proc. of ACM SIGCOMM'14*, pages 539–550, Chicago, IL, August 2014.

[21] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, July 2014.

[22] X. Chen, M. Chen, B. Li, Y. Zhao, Y. Wu, and J. Li. Celerity: A low-delay multi-party conferencing solution. *IEEE Journal on Selected Areas in Communications*, 31(9):155–164, September 2013.

[23] Bob Thomas, IJsbrand Wijnands, Ina Minei, and Kireeti Kompella. LDP Extensions for Point-to-Multipoint and Multipoint-to-Multipoint Label Switched Paths. RFC 6388.

[24] Jay Kreps, Neha Narkhede, and Jun Rao. Kafka: a distributed messaging system for log processing. In *Proc. of ACM Workshop on Networking Meets Databases (NetDB'11)*, Athens, Greece, June 2011.

[25] Petri Jokela, András Zahemszky, Christian Esteve Rothenberg, Somaya Arianfar, and Pekka Nikander. Lipsin: Line speed publish/subscribe inter-networking. In *Proc. of ACM SIGCOMM'09*, pages 195–206, Barcelona, Spain, August 2009.

[26] T. W. Cho, M. Rabinovich, K. K. Ramakrishnan, D. Srivastava, and Y. Zhang. Enabling content dissemination using efficient and scalable multicast. In *Proc. of IEEE INFOCOM'09*, pages 1980–1988, Rio de Janeiro, Brazil, April 2009.

[27] V. Gopalakrishnan, B. Bhattacharjee, K. K. Ramakrishnan, R. Jana, and D. Srivastava. Cpm: Adaptive video-on-demand with cooperative peer assists and multicast. In *Proc. of IEEE INFOCOM'09*, pages 91–99, Rio de Janeiro, Brazil, April 2009.

[28] Bradley Cain, Dr. Steve E. Deering, Bill Fenner, Isidor Kouvelas, and Ajit Thyagarajan. Internet Group Management Protocol, Version 3. RFC 3376.

[29] C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefen. Deployment issues for the ip multicast service and architecture. *IEEE Network*, 14(1):78–88, January 2000.

[30] Wen-Tsuen Chen, Pi-Rong Sheu, and Yaw-Ren Chang. Efficient multicast source routing scheme. *Computer Communications*, 16(10):662–666, 1993.

[31] Stephen E. Deering and David R. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Transactions on Computer Systems*, 8(2):85–110, May 1990.

[32] AWS Announces Nine New Compute and Networking Innovations for Amazon EC2. https://bloom.bg/2t1N9py. [Online; accessed February 2022].

[33] Run IP Multicast Workloads in the Cloud Using AWS Transit Gateway. https://go.aws/2RT6stz. [Online; accessed February 2022].

[34] NetFPGA SUME Reference Learning Switch Lite. https://bit.ly/2UrUFlx. [Online; accessed February 2022].

[35] The Internet Topology Zoo. http://www.topology-zoo.org/dataset.html. [Online; accessed February 2022].

[36] Apache ActiveMQ. http://activemq.apache.org. [Online; accessed February 2022].

[37] Bt iptv (youview). https://bit.ly/3ssCvTz. [Online; accessed February 2022].

[38] Multicast Command Reference for Cisco ASR 9000 Series Routers. https://bit.ly/3AaVGDQ. [Online; accessed February 2022].

[39] RabbitMQ. http://www.rabbitmq.com. [Online; accessed February 2022].

[40] U-verse tv. https://bit.ly/3HLhfyL. [Online; accessed February 2022].

[41] Zuckerberg really wants you to stream live video on Facebook. https://bit.ly/2v6uHqF. [Online; accessed February 2022].

[42] Benoit Donnet, Korian Edeline, Iain R. Learmonth, and Andra Lutu. Middlebox classification and initial model. https://bit.ly/3dZelXV. [Online; accessed February 2022].

Appendix A Correctness of Yeti

Theorem 1 (Correctness)**.** *Yeti forwards packets on and only on links that belong to the multicast graph.*

Proof. Yeti guarantees correctness by creating an ordered set of Yeti labels for the given graph at the controller using the ENCODEGRAPH algorithm. Recall that the algorithm first creates a tree to represent the services needed before reaching the destinations. A node in that calculated tree consists of router ID v and sub-sequence of services $\mathbb{S}$. Every node appears only once in the tree (by construction). This means that the tree has no cycles.

The label creation algorithm traverses the tree to calculate the final labels. Within every path with provided services $\mathbb{S}$, the order and type of created labels represent how packets should be forwarded in the data plane. This is detailed as follows. FSP labels do not result in incorrect forwarding because: (1) an FSP label with ID v is only added when a tree node v is traversed by the CREATELABELS algorithm, (2) since every node with router ID v and services $\mathbb{S}$ is traversed once and only once by the algorithm, only a single FSP v can be added to the labels representing that node with services $\mathbb{S}$, (3) in the data plane, the router with ID v removes the FSP v label. Thus, no subsequent routers in along the path with same services can process that label and transmit the packet back to v, and (4) since the traversal starts from the source, if a node u precedes node v in the tree, the algorithm guarantees that u is traversed before v. Thus, there is no label to forward packets back to u.Similar properties are guaranteed for FTE labels in terms of links. Moreover, attaching multiple FTE labels does not result in incorrect forwarding. Otherwise, the given tree has loops, or routers do not remove FTE labels.

For MCT and CPY labels, since the CREATELABELS algorithm recursively creates the labels for each branch, the same guarantees apply within a single branch for branching points. In addition, routers in one branch do not forward packets to routers in other branches. This is because (1) the given tree is a proper one (i.e., has no cycles), and (2) every router in a given branch processes the subset of labels duplicated for that branch using the CPY labels. □

Appendix B Practical Considerations of Yeti

Multicast across ISPs. The description of Yeti thus far has focused on offering a scalable multicast service within a *single* ISP. Extending Yeti to multiple ISPs can be done in multiple ways. For example, a content provider could have separate agreements with different ISPs to serve clients within these ISPs, where each ISP runs its multicast service independently from the others. In this case, a separate feed of the multicast session traffic is provided from the content provider to each ISP. Agreements between major content providers, e.g., Facebook and Netflix, and large ISPs are not uncommon. Another way of extending Yeti to multiple ISPs is through tunneling, where a tunnel is established between an egress router of an ISP to an ingress router of another ISP. The ingress router of the second ISP would attach labels created by the controller of that ISP. While the tunneling approach does not reveal the internal network details of ISPs to each other, which is important in practice, it does require collaboration among ISPs to establish tunnels among some routers.

Incremental Deployment. An ISP may have some legacy routers that are not programmable and thus cannot run the packet processing algorithm of Yeti. There are multiple options that Yeti can still function in this situation, albeit with some workaround and minor overheads. First, Yeti is general and can support arbitrary multicast graphs. Thus, a possible solution is to modify the multicast graphs to avoid going through legacy routers. The multicast graphs is an *input* to our label creation algorithm, and thus the computed labels will not direct traffic through legacy routers. If a legacy router cannot be avoided, a tunnel can be created between the router immediately before the legacy router and each router following it has multicast destinations.

Appendix C Illustrative Example

We present a simple example to illustrate all steps of the proposed approach. Figure 11 shows the multicast tree of the session in Figure 1, where solid arrows indicate the graph links. The dotted line is the shortest path that the ISP avoids because it is over-utilized. Router IDs and used interface IDs are shown in the figure. The number of core routers and maximum interface count are 12 and 5, respectively. Thus, the label sizes (in bits) are 8, 8, 7 and 5 for MCT, CPY, FSP and FTE, respectively (Table 1).

The controller generates the shown labels using the ENCODEGRAPH algorithm as follows. First, the algorithm creates three FSP labels to encode path segments to routers 2, 7 and 4. Notice that the most significant bit in each of these labels is set to one as these nodes provide services a, b and c.

The algorithm then generates MCT 1-00110 to duplicate packets on interfaces 2 and 3 at router 4. Since the children 3 and 7 have core children, the algorithm sets the most significant bit in the MCT label to one, and creates two CPY labels for branches A and B. In branch B, the recursive call of Algorithm 1 creates labels for the path segment {3, 6, 5, 8} as follows. First, the algorithm appends routers 3, 6 and 5 to *pth_seg* because each has one core child (Line 13, Algorithm 1). When the algorithm reaches 8 (which has two core children), the algorithm appends it to *pth_seg* (Line 25, Algorithm 1) and creates labels for the path segment and the branching point at router 8. For the path segment, since link (3, 6) is not on the shortest path from 3 to 8, the algorithm creates FTE 011 to forward packet on interface 3 at router 3. In addition, the algorithm creates FSP 0-1000 to forward packets from 6 to 8, because the links (6, 5) and (5, 8) are on the shortest path between 3 and 8.

We describe the packet processing algorithm at representative routers. The dark labels in Figure 11 are the ones that are processed at given routers. When router 2 receives FSP 1-0010, it decides that the packet needs to be processed by service a. So, it removes the label and forwards the packet to the corresponding datacenter. When the processing is done, router 2 receives a packet which its first label is FSP 1-0111. Thus, it forwards the packet on the shortest path to router 7. Notice that router 4 does not remove FSP 1-0111 as it is not destined for it. Then, routers 7 and 4 receives packets where the contents of the FSP labels are their router IDs. After service c processes the packet, router 4 processes MCT 1-00110 by duplicating the packet on interfaces 2 and 3, and copying specific byte ranges using the CPY labels. Router 3 forwards the packet on interface 3. Router 8 removes FSP and MCT labels and duplicates the packet to routers 11 and 12. Since the packet has no labels at router 11 and 12, they transmit it to egress routers.

Appendix D Implementation of Yeti using P4

P4 is a data plane programming language that is getting popular due to its flexibility. Thus, we show that Yeti can be implemented in P4 switches. We implemented Yeti using the Intel P4 software development environment (SDE) version 9.5.0.

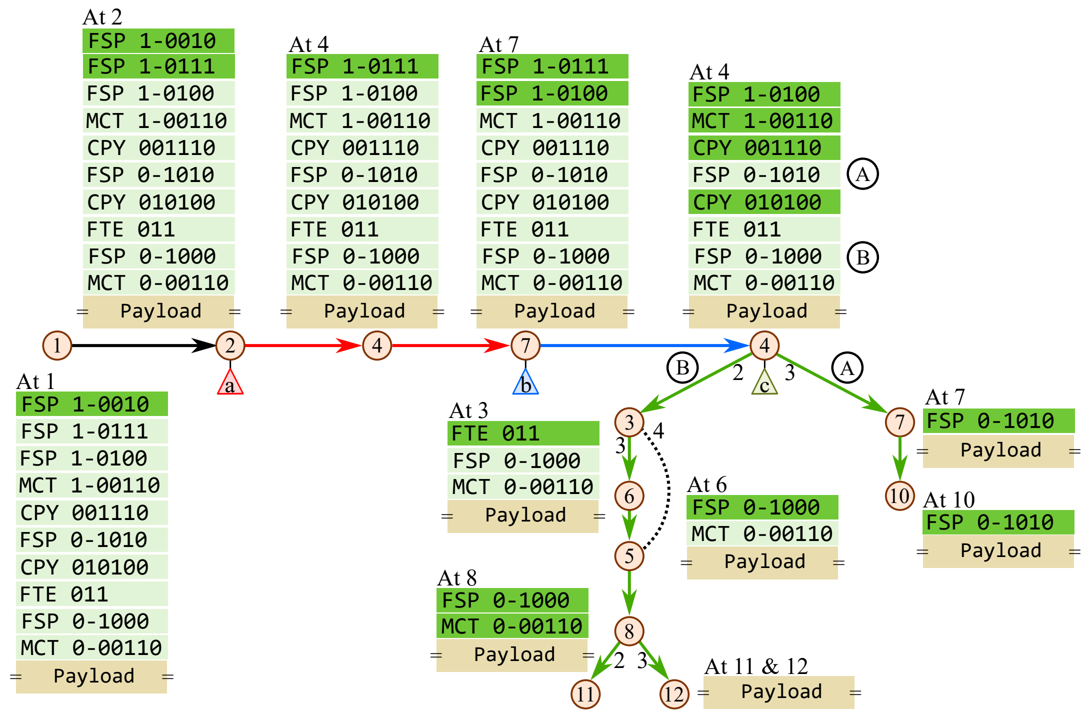


Figure 11: Illustrative example of how labels in Yeti represent the multicast tree in Figure 1.

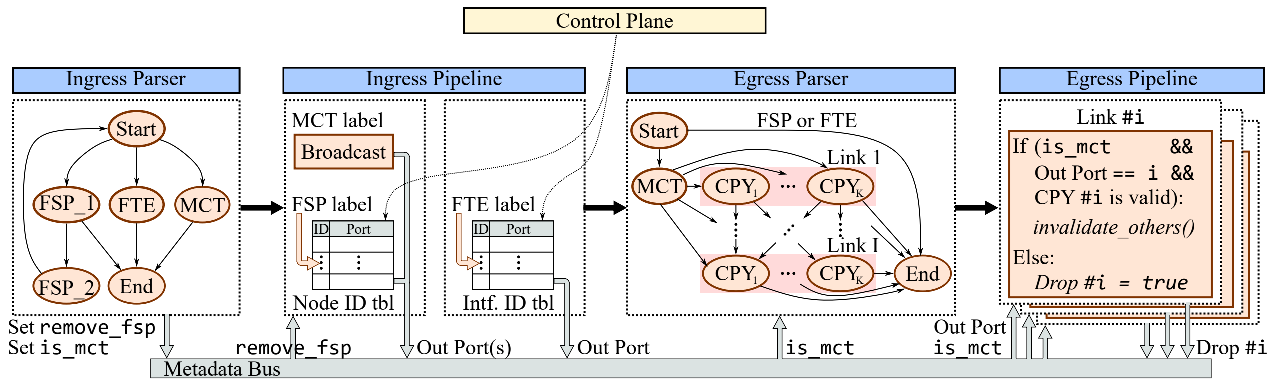


Figure 12: Design of Yeti in P4 switches.

We used the provided tools such as the P4 compiler (called `bf-p4c`) and switch model integrated with the SDE to realize our ideas and validate our implementation. We note that P4 programs implemented in open-source software switches, e.g., `bmv2`, may not necessarily work on actual hardware switches, because of the potential mismatch between the available physical resources in actual switches and the assumed resources in software switches. This is not the case for the Intel SDE, since its P4 compiler produces code for actual Tofino switches (which are also manufactured by Intel).

There are two main challenges in implementing Yeti using P4. First, current Tofino switches do not provide programmable primitives to implement new multicast systems. This is because the packet replication engine is implemented as a fixed-function block. Second, the current `bf-p4c` compiler does not support variable-length fields (i.e., `varbit`) needed to parse and process CPY labels. We made three design choices to address these challenges. We first divided the packet processing between ingress and egress pipelines to realize a programmable multicast primitive. Second, we relied on the available resources and programmable capabilities of the parsers to reduce the processing in ingress/egress stages. Finally, we explicitly unrolled the parsing and processing of a CPY label as an array of items.

Figure 12 illustrates the high-level design of our P4 implementation. Upon a packet arrival, the ingress parser processes FSP, FTE and MCT labels as follows. For an FSP label, the parser reads the included node ID, and parses the labels again if the parsed node ID is the same as the router ID. In this case, the parser sets a metadata field `remove_fsp` to be used by the ingress pipeline. Recall that the Yeti controller never creates two consecutive FSP labels (§3.4 and §3.6). Therefore, the parser always terminates. In the case of FTE and MCT labels, the parser reads their contents. In addition, it sets a metadata field `is_mct` when it parses an MCT label.

The ingress pipeline contains two tables to maintain node and interface IDs, and spans two stages. In the first stage, the algorithm processes an FSP label by reading an entry from the node ID table that matches the label content, and setting the outgoing port accordingly. The algorithm also removes the FSP label if the metadata field `remove_fsp` is set. The algorithm processes MCT labels, in the same stage, by duplicating the packet to all outgoing ports. The FTE label processing is done in the second stage, and it is similar to that of FSP. However, the algorithm always removes FTE labels.

The egress parser and pipeline are used only to handle MCT and CPY labels. For each duplicated packet, the egress parser extracts the contents of MCT label when the metadata field `is_mct` is set. The parser then reads CPY labels sequentially based on the content of MCT label and offsets in CPY labels. Specifically, for each CPY label, the parser extracts its offset and content. The content is read based on the offset value, and represented as an array of up to K multiples of B bits to emulate `varbit<K×B>`. When the parser is done, the egress pipeline identifies which outgoing port should transmit what CPY label based on the egress port number and validity of the CPY label. The MCT and remaining CPY labels are removed.

We did not have a physical Tofino switch in our lab to conduct measurement experiments at the time of conducting this research. We thus validated our implementation by writing and running multiple test cases, and sending and receiving packets to and from the Tofino switch model process. When we run a test case, we insert the required entries into node and link IDs tables. We send labeled packets using `scapy`, and verify the reception of outgoing packets based on the attached Yeti labels. A test case succeeds if all packets were received on and only on expected ports with expected headers.

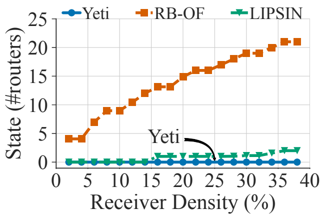
Appendix E Additional Simulation Results

This appendix includes more figures and results from our simulation.

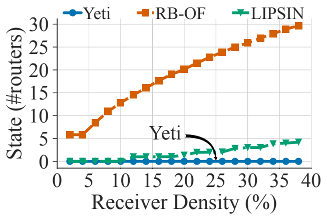
Figure 13 shows the state size for all 12 ISP topologies. The figure indicates that Yeti provides a scalable multicast service as it does not require any state at any router.

Figures 14–15 show the label size of Yeti versus BIER-TE. Compared to BIER-TE, the figures show that Yeti reduces the label size significantly across receiver densities and as packets traverse the network.

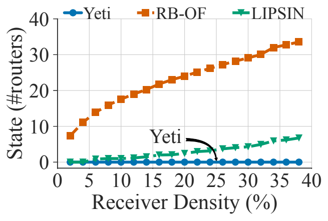
Figures 16–19 indicate that Yeti impose small label and processing overheads when supporting service chaining and traffic engineering requirements.



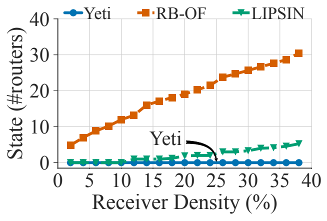
(a) #Routers=36



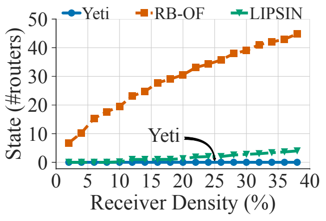
(b) #Routers=49



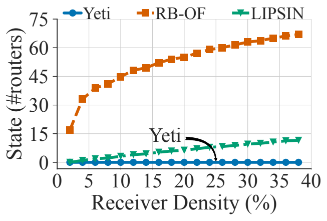
(c) #Routers=53



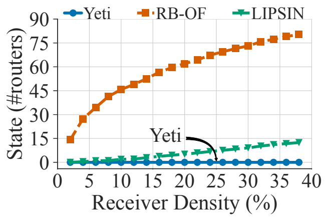
(d) #Routers=58



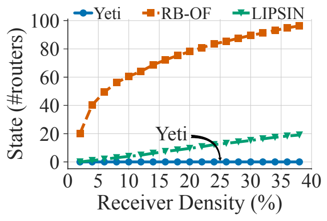
(e) #Routers=69



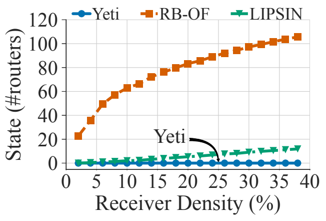
(f) #Routers=84



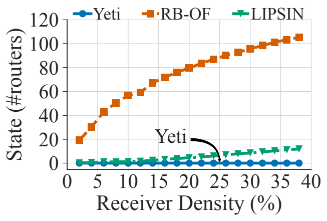
(g) #Routers=113



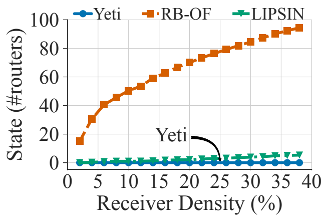
(h) #Routers=125



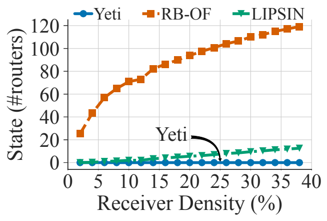
(i) #Routers=145



(j) #Routers=149

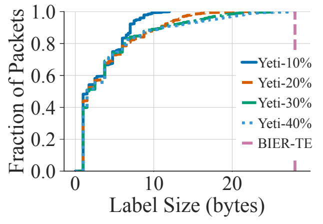


(k) #Routers=153

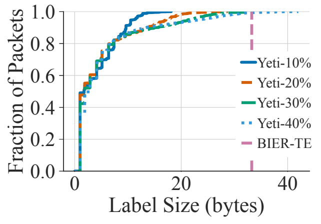


(l) #Routers=158

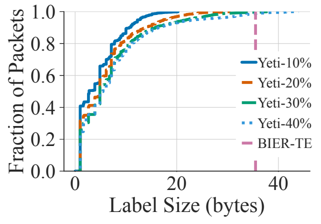
Figure 13: State size for the considered ISP topologies.



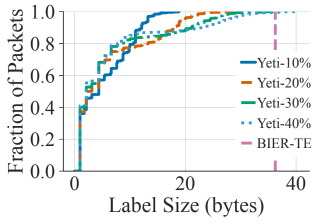
(a) #Routers=36



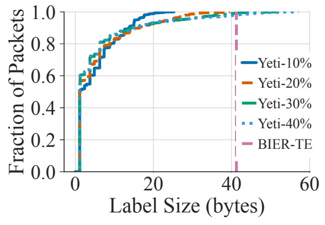
(b) #Routers=49



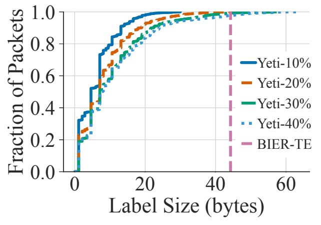
(c) #Routers=53



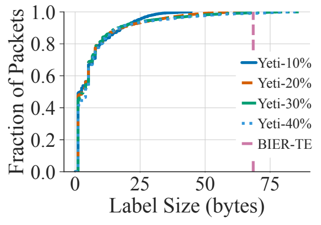
(d) #Routers=58



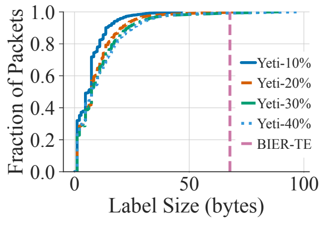
(e) #Routers=69



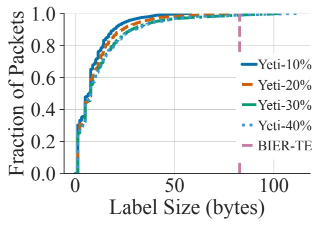
(f) #Routers=84



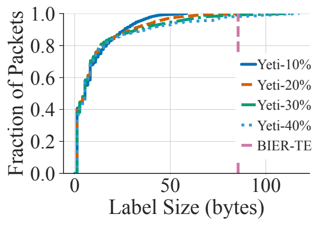
(g) #Routers=113



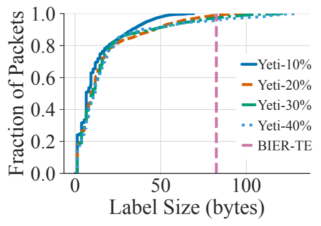
(h) #Routers=125



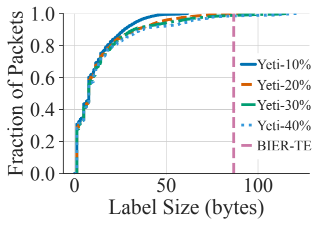
(i) #Routers=145



(j) #Routers=149

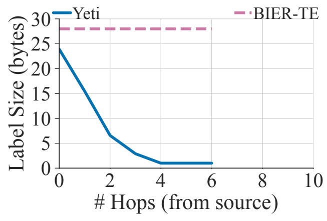


(k) #Routers=153

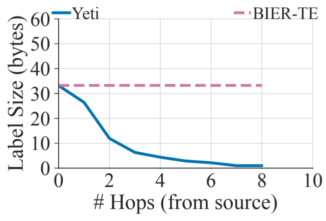


(l) #Routers=158

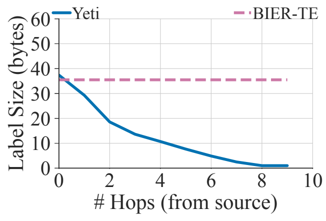
Figure 14: Label size CDF for different ISP topologies.



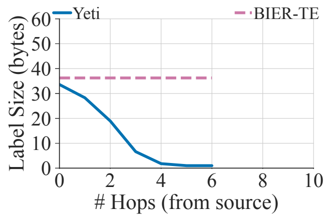
(a) #Routers=36



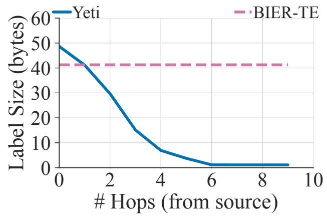
(b) #Routers=49



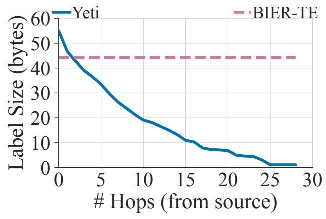
(c) #Routers=53



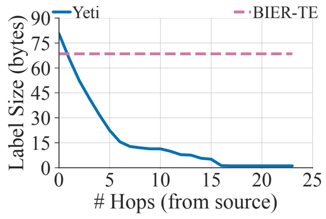
(d) #Routers=58



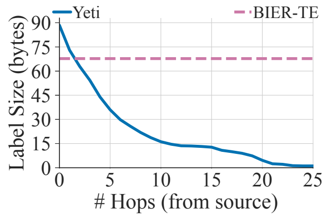
(e) #Routers=69



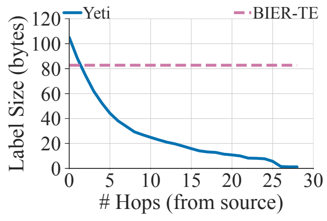
(f) #Routers=84



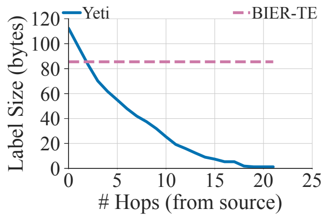
(g) #Routers=113



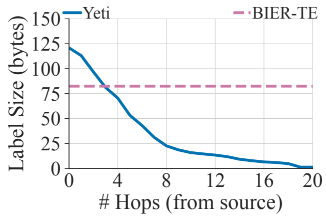
(h) #Routers=125



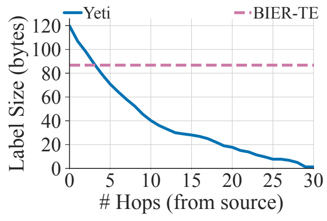
(i) #Routers=145



(j) #Routers=149

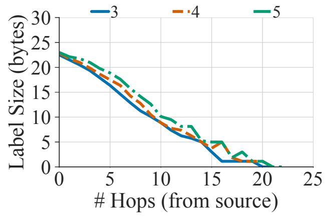


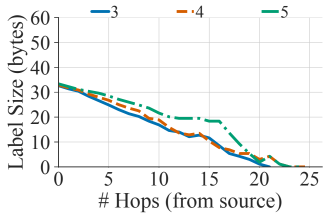
(k) #Routers=153

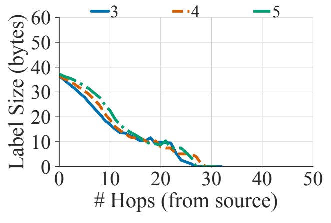


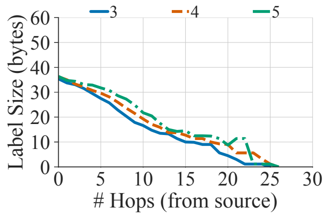
(l) #Routers=158

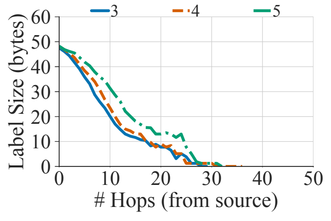
Figure 15: Label size of Yeti versus BIER-TE as packets traverse the network.

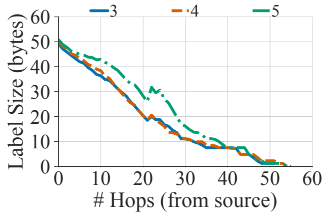

(a) #Routers=36

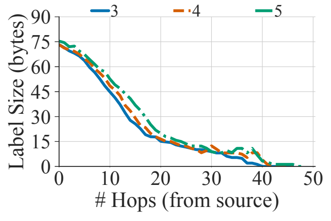

(b) #Routers=49

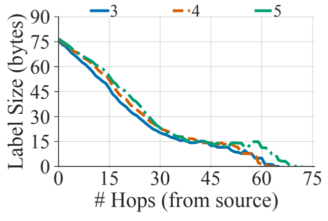

(c) #Routers=53

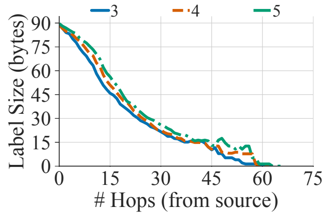

(d) #Routers=58

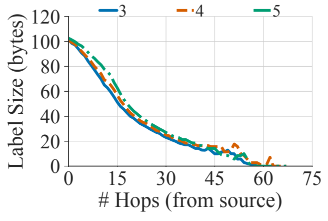

(e) #Routers=69

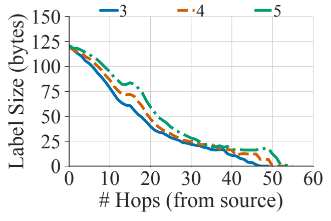

(f) #Routers=84

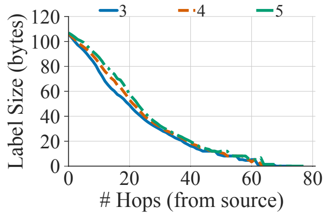

(g) #Routers=113

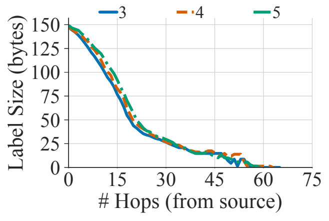

(h) #Routers=125


(i) #Routers=145


(j) #Routers=149

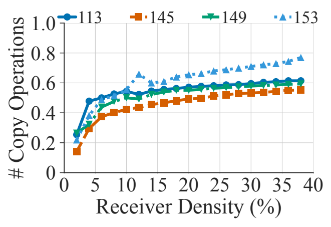

(k) #Routers=153


(l) #Routers=158

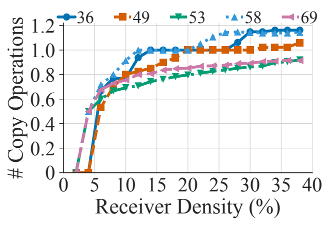


(m) #Routers=197

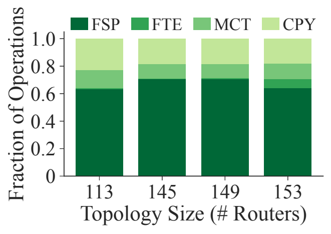
Figure 16: Analysis of label size of Yeti to satisfy service chaining requirements.



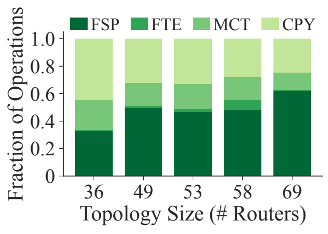
(a) # copy operations



(b) # copy operations



(c) Distribution of operations



(d) Distribution of operations

Figure 17: Analysis of processing overheads of Yeti for different ISP topologies.

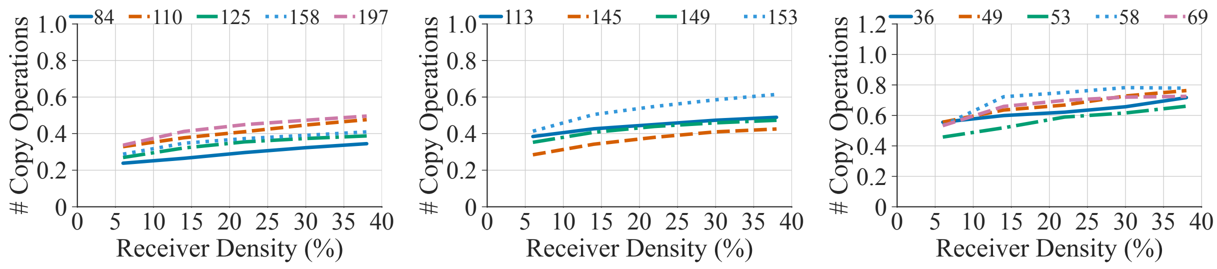


Figure 18: # copy operations for different ISP topologies to support service chaining.

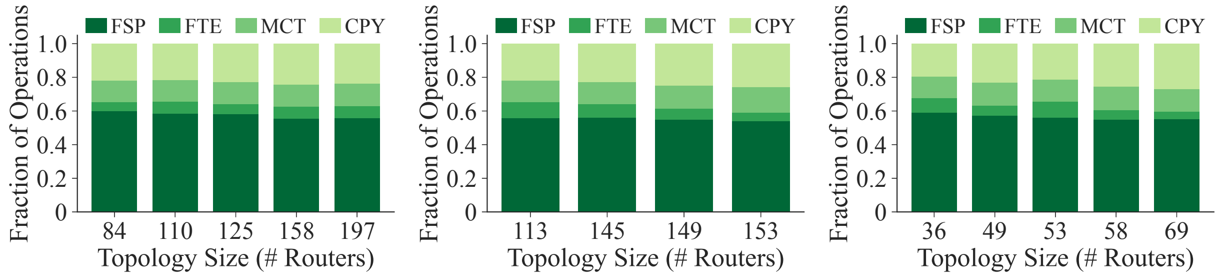


Figure 19: Distribution of FSP, FTE, MCT and CPY operations to support service chaining.