Oktopus: Service Chaining for Multicast Traffic

Khaled Diab, Carlos Lee, and Mohamed Hefeeda
School of Computing Science, Simon Fraser University, Canada

***Abstract*—Multicast service chaining refers to the orchestration of network services for multicast traffic. Paths of a multicast session that span the source, destinations and required services form a complex structure that we refer to as the multicast distribution graph. In this paper, we propose a new path-based algorithm, called Oktopus, that runs at the control plane of the ISP network to calculate the multicast distribution graph for a given session. Oktopus aims at minimizing the routing cost for each multicast session while satisfying all service chaining requirements. Oktopus consists of two steps. The first one generates a set of segments from the given ISP network topology, and the second step uses these segments to efficiently calculate the multicast distribution graph. Oktopus has a fine-grained control over the selection of links in the distribution graphs that leads to significant improvements. Specifically, Oktopus increases the number of allocated sessions because it can reach ISP locations that have the required services, and thus includes them in the calculated graph. Moreover, Oktopus can reduce the routing cost per session as it carefully chooses links belonging to the graph. We compared Oktopus against the optimal and closest algorithms using real ISP topologies. Our results show that Oktopus has an optimality gap of 5% on average, and it computes the distribution graphs multiple orders of magnitude faster than the optimal algorithm. Moreover, Oktopus outperforms the closest algorithm in the literature in terms of the number of allocated multicast sessions by up to 37%.**

I. Introduction

Recently, network operators have started to adopt Network Function Virtualization (NFV) [1], [2] to reduce the cost of purchasing and managing middleboxes such as firewalls and intrusion detection systems (IDSes). In NFV, the functionality of a hardware-based middlebox is implemented as a virtual network function (VNF). This has led the research community to explore various aspects such as implementing VNFs efficiently and securely [3], [4], [5], [6], [7], [8], building network stacks for them [9], [10], managing their state [11], [12], and deploying them [13], [14], [15]. For brevity, we refer to a VNF as a *service*.

ISP networks have observed various changes in terms of their architectures and the complexity of Internet applications they carry. Specifically, large ISPs tend to deploy various services at different locations in their networks to support their customers' needs. Moreover, many recent Internet applications allow their users to produce and consume content anytime at high rates. Examples of such applications include live Internet broadcast, e.g., Facebook Live [16], IPTV [17], webinars and video conferencing [18] and massive multiplayer games [19]. For instance, Facebook Live aims to stream millions of live sessions to millions of concurrent users [16], [20]. Multicast can be used to efficiently support these applications. Many large ISPs use multicast to efficiently carry traffic through their networks. For example, AT&T has deployed UVerse and BT has deployed YouView, where both use multicast.

978-1-7281-6992-7/20/$31.00 ©2020 IEEE

As these Internet applications become complex, providers of these applications require their multicast traffic to pass through ordered sequences of network services. For example, traffic of a live video stream may be required to pass through a firewall, IDS and video transcoder. The orchestration of ordered services in a multicast session is referred to as *multicast service chaining*. A crucial requirement for service chaining is that packets of a session need to be processed by the required sequence of services before reaching their destinations. Since services are typically deployed at different ISP locations, packets of a multicast session may need to visit a node or link multiple times. Therefore, the paths of a multicast session that requires service chaining may not necessarily form a tree. Instead, paths that span the source, destinations and required services form a more complex structure that we refer to as a *multicast distribution graph*. To realize service chaining, the ISP needs to efficiently calculate multicast distribution graphs for multicast sessions.

Calculating multicast distribution graphs that fulfill service chaining is, however, a challenging task. First, the ISP needs to jointly allocate resources at the network layer (i.e., link capacities) and system layer (i.e., processing capacities of network services), while minimizing the routing cost per session. Second, the ISP should maximize the number of allocated multicast sessions in order to maximize the utilization of the available network resources. This can be a hard task to achieve especially when the number of sessions increases. Third, since an ISP does not necessarily deploy all service instances at all of its locations, the calculated graphs may include loops in the network. Forwarding loops may waste significant network resources especially for bandwidth-demanding applications such as live video streaming. In addition, they may introduce forwarding ambiguity at routers. Finally, the search space of multicast service chaining is much larger than its unicast counterpart. As a result, exhaustive search algorithms may not calculate the distribution graphs in a reasonable time.

In this paper, we address the complex problem of multicast service chaining for large-scale ISPs. We propose a new algorithm, called Oktopus, that runs at the control plane of the ISP network to calculate a multicast distribution graph for every multicast session. Oktopus has two goals when it calculates distribution graphs: (i) maximizing the number of allocated multicast sessions in the ISP network, and (ii) minimizing the average routing cost per session. Oktopus is *efficient* as it achieves these goals without exceeding the

ISP network and processing resources, and it does not create forwarding loops in the ISP network. Oktopus is also *general* as it does not make assumptions on the ISP topology or where the services are deployed.

The key idea of Oktopus is the design of a new *path-based approach* to calculate the multicast distribution graph. Specifically, for a given multicast session, Oktopus calculates a set of valid network paths that satisfy the service chaining requirements from the source to each destination. Then, it combines the calculated paths to all destinations to form the final distribution graph. We introduce multiple new ideas in the design of Oktopus to address the complexity of the multicast service chaining problem and its huge search space, such as efficient offline and on-demand path generation, path weight calculation and lightweight tracking of path direction.

We evaluate and compare Oktopus against the optimal solution and the closest algorithm in the literature [21] using real ISP topologies with different sizes. Compared to the optimal solution, Oktopus produces multicast distribution graphs with a routing cost of about 5% more than the ones produced by the optimal algorithm, while it computes the distribution graphs multiple orders of magnitude faster than the optimal algorithm. Moreover, when increasing the number of multicast sessions, the optimal algorithm fails to calculate a solution within 24 hours for large ISP topologies, while Oktopus calculates the distribution graph of each multicast session in a few seconds. Furthermore, our results show that Oktopus outperforms the closest algorithm in the literature in terms of the number of allocated multicast sessions by up to 37%, and it efficiently uses the available ISP resources to minimize the routing cost.

II. Related Work

Prior works, e.g., [22], [23], [24], addressed the unicast service chaining problem. For example, Sallam et al. [22] proposed to transform the ISP network to another graph and then find a service-chained path using Dijkstra's algorithm. These methods are not applicable to multicast because they may introduce loops and/or allocate more resources as they do not consider the multicast branching nature.

Recent works, e.g., [25], [26], [21], addressed the problem of service chaining for multicast traffic. Xu et al. [25] proposed an algorithm to search for the best Steiner tree that contains the required number of services. This work assumed services are deployed at all ISP locations, which is not practical in many scenarios. Kuo et al. [26] addressed the multicast service chaining problem in cloud environments by building network overlays, while Oktopus focuses on multicast service chaining at the network level (i.e., handling link and forwarding table capacities). Ren et al. [21] proposed an algorithm, called MSA, to solve the multicast service chaining problem. This algorithm builds an adjacency graph where each node represents a service and each edge represents the shortest path between the service nodes. MSA then finds a path from the source to the last service function from the adjacency graph. Finally, it calculates a Steiner tree that connects the last service nodes to the destinations. MSA, however, assumes infinite link capacities, and thus it may only work in certain situations where the number of multicast sessions is small. However, as the number of sessions increases, MSA may not be able to allocate many sessions, or it could congest the network. In contrast to MSA, Oktopus is general as it calculates multicast distribution graphs for large numbers of sessions without assuming infinite link capacities, which is the practical case.

III. System Model and Problem Definition

A. System Model

Multicast can be used in various scenarios. A common use-case is when a major ISP, e.g., AT&T, manages multicast sessions for its own clients. Clients in this case can be end users in applications such as IPTV and live streaming. Clients could also be caches for content providers such as Netflix, where the contents of such caches are periodically updated using multicast.

We consider a multi-region ISP network that has data and control planes. The data plane is composed of core routers deployed in multiple geographical regions. The control plane (referred to as the *controller*) learns the ISP network topology. This is simple to achieve using common intra-domain routing and monitoring protocols. The controller sends match-action rules (e.g., using the OpenFlow protocol [27]) to core routers to inform them how to forward packets of multicast sessions.

The ISP network is modeled as a graph ($\mathbb{N}$, $\mathbb{E}$), where $\mathbb{N}$ represents ISP locations and $\mathbb{E}$ represents links between the locations. Each link $l \in \mathbb{E}$ has a capacity of c_l bits/sec. The ISP sets a cost γ_l of forwarding a unit of traffic on link l. An ISP location refers to a physical entity that contains routers and servers, e.g., a point of presence (PoP). We refer to ISP locations as nodes for simplicity. Each node $n \in \mathbb{N}$ contains a core router to forward traffic to/from other nodes. That core router has a forwarding table of size f_n entries, which is used to maintain match-action rules sent by the controller to forward multicast packets. In addition, node n has servers that host a set of virtualized services. The ISP allocates computing resources to process $p_{v,n}$ bits/sec for each service v deployed at node n. The deployment and allocation of services are beyond the scope of this work.

A multicast session s is defined by the tuple $\langle src, dsts, \mathbb{V} \rangle$, where src and $dsts$ are its source and destinations, and $\mathbb{V}$ is the sequence of required services. The session has a bandwidth of b bits/sec. Packets of s need to be processed by the sequence of required services $\mathbb{V}$ before reaching its destinations.

The ISP controller uses Oktopus to calculate for the multicast session s a distribution graph $\mathcal{D}$, which is defined by paths spanning src and $dsts$ while satisfying the required sequence of services $\mathbb{V}$. The controller then maps the graph to match-action rules and sends these rules to corresponding routers.

Figure 1 shows an example of a network that provides two services. Service 1 is deployed at nodes b and d, while service 2 is deployed at node e. Each link has a capacity of 2 bandwidth units, and each service has a capacity of 1 processing unit. There are two concurrent multicast sessions defined by $\langle a, \{f, g, h, i\}, \{1 \rightarrow 2\} \rangle$ and $\langle a, \{g, i\}, \{1\} \rangle$, each has 1

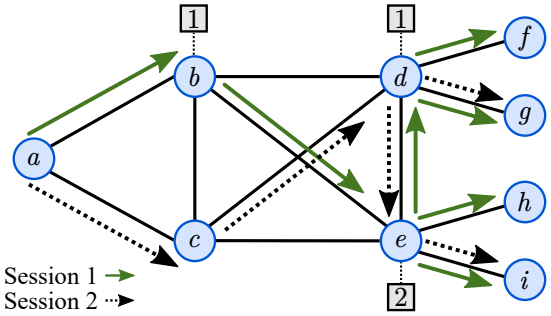


Fig. 1: Illustrative example showing two multicast sessions with service chaining.

bandwidth unit. Solid and dotted arrows represent the multicast distribution graphs calculated by the proposed algorithm for the two multicast sessions. Notice that the algorithm does not exceed the processing capacity of services, and it balances the load across the links.

Each graph node n belonging to $\mathcal{D}$ represents a core router and a set of services if this ISP location processes packets of s. Moreover, packets of a multicast session maintain in their headers what services these packets have passed through so far. These headers are updated by every service that processes the packets. We define the packet class as the set of written services in its headers. Specifically, the classes of incoming or outgoing packets on interface i at node n of session s are referred to as $inc(i, n, s)$ or $out(i, n, s)$, respectively.

Edges of $\mathcal{D}$, denoted by $\mathbb{L}$, are calculated to satisfy the service chaining requirements, and thus, they may not follow shortest paths computed by intra-domain routing protocols. We define the routing cost of forwarding packets of a multicast session s on links of $\mathcal{D}$ as $b \times \sum_{l \in \mathbb{L}} \gamma_l$.

B. Problem Definition and Hardness

The problem addressed in this paper is to calculate a multicast distribution graph $\mathcal{D}$ for an input multicast session s with required services $\mathbb{V}$ by allocating the processing resources (at the system-level) of the services deployed at various ISP locations to the session as well as engineering edges of the distribution graph (at the network-level) to pass through the required services while minimizing the routing cost of the session without exceeding the available processing resources at nodes, link capacities, and forwarding table sizes at core routers.

Theorem 1. *Determining the multicast distribution graph $\mathcal{D}$ that minimizes the routing cost of a multicast session with a given sequence of services $\mathbb{V}$ is NP-hard.*

Proof Sketch: We prove the theorem by showing a polynomial-time reduction from the NP-hard Steiner tree problem to a special instance of our problem, which happens when the network resources are infinite and the multicast session does not require any services. ■

We shed some lights on the difficulty of solving the multicast service chaining problem, beyond being NP-hard. Previous works, e.g., [28], showed that finding a *unicast* path that satisfies service chaining requirements is computationally intractable even when the available processing resources are infinite. This is because the search space of this family of problems is prohibitively large. For the multicast case, the search space is even larger as a multicast distribution graph needs to be composed of multiple unicast paths to reach all destinations of the session. Even without service chaining, the multicast traffic engineering problem is very challenging to solve with any analytical bounds on the performance. For example, Huang et al. [29] showed that solving the multicast traffic engineering problem without service chaining cannot be approximated within any ratio.

In summary, because of its huge search space, the multicast service chaining problem considered in this paper cannot be solved optimally for practical topology sizes, nor can it be solved with any constant-factor approximation algorithms as shown in previous works for even simpler instances of it. Yet, it is a practical and important problem. In this paper, we propose a principled, heuristic, approach to efficiently find near-optimal solutions for the multicast service chaining problem.

IV. Proposed Solution

To aid the reader in understanding our solution, we structure this section as follows. We start by defining the basic terms used in the solution and summarizing the basic design principles employed to develop it. Then, we present a high-level overview showing how the solution works and its main components. This is followed by a detailed description of each component. Then, we analyze the time and space complexities of our solution and describe how it can be deployed in practical settings. Finally, we present a complete example showing all steps of our solution on a small topology.

A. Definitions and Design Principles

Definitions. We define two terms that we use extensively when describing our algorithm: *segment* and *service-chained path*. A path *segment* from a node n to a node m is a sequence of nodes from n to m without a loop. That is, no node appears more than once in a segment. For example, the sequence of nodes $\{a, b, d, g\}$ in Figure 1 forms a segment from a to g. However, the node sequence $\{a, b, c, a, b, d, f\}$ is not a valid segment from a to f as it contains the loop $\{a, b, c, a\}$. A segment may not fulfill service chaining requirements. Note that our use of the term "segment" is different from the use of the same term in the recent Segment Routing (SR) initiative [30]; while a segment in both SR and Oktopus refers to a section of consecutive links between two endpoints, SR requires a segment to be the shortest path between these endpoints whereas Oktopus does not impose this requirement.

A *service-chained path* is a sequence of pairs of nodes and services that satisfies the service chaining requirements in $\mathbb{V}$ either partially or fully. For brevity, we refer to such a sequence as a **path**. For example, the sequence of nodes $\{\langle a, \phi\rangle, \langle b, \phi\rangle, \langle d, \{1\}\rangle \langle e, \{2\}\rangle, \langle d, \phi\rangle, \langle f, \phi\rangle\}$ forms a valid

path to reach f and satisfies services $\{1, 2\}$, where ϕ means this node does not provide any services to the session.

Design Principles. We summarize below the main principles and intuitions that we used to develop an efficient solution for the quite-challenging multicast service chaining problem.

- **Reducing the search space.** As described in §III-B, the multicast service chaining problem has a prohibitively-large search space. To address this, we conceptually divide the solution into two phases. The first phase computes intermediary data that depends only on the underlying ISP topology without considering the network services and their dynamic capacities. This intermediary data is a subset of possible segments that is carefully chosen to cover different parts of the network. We *pre-compute* this data offline to save time, since it does not depend on the requirements of multicast sessions which are known only when the sessions arrive. The second phase utilizes the precomputed segments and considers the services needed by individual multicast sessions. In case that the initial path segments were not sufficient to find a solution for a given multicast session, we incrementally add more segments on-demand in real time.
- **Recursive composition of network services.** For a given session and a node pair, the precomputed data may not necessarily contain segments that satisfy the service chaining requirements between that node pair. To address this, we expand the search space on demand, by dividing each segment into smaller ones, and recursively composing paths that satisfy the service chaining requirements from shorter segments.
- **Balancing the link load.** In practice, network operators prefer balancing the load across different links to avoid congestion. To consider this, we design link and service weights to direct our algorithm to chose paths in a way that leads to balancing the load across links and services.

B. High-level Overview

Oktopus runs at the ISP controller to calculate the multicast distribution graph. It takes as input the ISP network topology $(\mathbb{N}, \mathbb{E})$ and the parameters of the multicast session s, which are its source src, destinations dst, required services $\mathbb{V}$, and bandwidth b. The algorithm then produces the distribution graph $\mathcal{D}$ that satisfies the service chaining requirements. The proposed algorithm is path-based. That is, the key insight is that links of the calculated graph are the union of unicast paths from the source to all destinations. To efficiently realize this path-based approach, Oktopus runs two steps to calculate the distribution graph: *segment generation* and *graph calculation*. The first step generates a set of network segments for the ISP network, and the second step uses the segments to calculate the multicast distribution graph.

The GENERATESEGMENTS function, described in §IV-C, builds the set of network segments, and it stores them in what we call segment store. Segment generation happens offline (i.e., before the graph calculation starts) as well as on-demand (i.e., during the graph calculation). As the number of potential segments grows exponentially with the network size, the main challenge of building such a segment store is the balance between the space overhead and diversity of generated segments. We address this challenge by utilizing some practical traffic engineering observations to reduce the number of generated segments in the initial segment store, while progressively generating new segments as needed.

Next, for a given multicast session, the CALCULATEGRAPH function computes its distribution graph as detailed in §IV-D. Specifically, for every destination in the multicast session, CALCULATEGRAPH examines segments in the segment store which can reach that destination, creates service-chained paths, calculates a weight value for each path, and picks the paths that satisfy the multicast service chaining objectives. Then, the algorithm merges these paths to build the final graph. However, simple path merging may result in forwarding ambiguity at routers. This ambiguity happens when a router receives packets of a multicast session on the same interface but it should forward them on different interfaces based on the services that each packet has gone through already. Thus, the proposed algorithm should produce graphs that satisfy the service chaining requirements while not resulting in forwarding ambiguity at routers. To calculate an efficient graph without forwarding ambiguity, we propose a lightweight data structure to maintain and track information about each path in the calculated graph.

C. Generating Segments

The GENERATESEGMENTS algorithm computes segments between each pair of nodes in the ISP network. The algorithm takes as inputs the ISP network topology $(\mathbb{N}, \mathbb{E})$ and the maximum number of segments to generate between every node pair K. In addition, we employ another parameter, which we call ρ, that is defined as the ratio between the maximum segment length and network diameter. This parameter is typically in the range of 1–2 and it controls the latency of generated segments. The algorithm then returns the segment store (denoted by $\mathcal{S}$) that maintains the generated segments. The segment store is a key-value data structure, where the key is a node pair (n, m), and the value is the list of generated segments between n and m. The segment store provides fast access to valid network segments, which is needed by the next step in the algorithm.

The pseudo code of the GENERATESEGMENTS algorithm is omitted due to space limitations. At high-level, the algorithm has two nested loops indexed by n and m, each iterates over the ISP nodes $\mathbb{N}$. It creates a new entry $\mathcal{S}[n, m]$ if such entry does not exist. It then generates segments and inserts them into $\mathcal{S}[n, m]$ as follows. For each node pair n and m, it traverses the ISP network between the node pair using breadth-first search, and computes the first K loop-free segments whose lengths do not exceed $\rho \times D$ hops, where D is the network diameter. It then calculates a set of edge-disjoint segments between the node pairs n and m, and adds them to $\mathcal{S}[n, m]$. It does so by calculating the maximum flow between the node pair using existing algorithms (e.g., Edmonds–Karp algorithm [31], [32]) to create the flow and residual networks. Saturated edges in

the residual network (i.e., links that can still forward traffic) correspond to the edge-disjoint segments.

There are three observations behind the design of the GENERATESEGMENTS algorithm. These observations guide the generation of segments that balance link usage while calculating the graph. First, the breadth-first traversal produces segments with diverse links compared to depth-first traversal. Second, edge-disjoint segments add more degrees of freedom to the generated segments as indicated by prior works, e.g., [33], [34]. Third, segments that are much longer than the network diameter would impose significant packet delays, and they should not be used to calculate the distribution graph.

Finally, we design the GENERATESEGMENTS algorithm to generate segments without depending on the deployed services at ISP locations or multicast sessions. Instead, the generated segments rely only on the ISP network topology. This is because the ISP network topology does not change frequently, while the ISP may deploy or remove services at smaller time scales to adapt to different network loads (i.e., the number of multicast sessions). This enables Oktopus to maintain and reuse the segment store across different runs of the algorithm.

D. Calculating the Multicast Distribution Graph

For a given multicast session, the graph calculation algorithm calculates a set of service-chained paths (or paths for short) and merges them to build the distribution graph. Unlike segments maintained in the segment store, every path is created for a specific multicast session, and it maintains information about the satisfied services by that path. This information is calculated and used by the graph calculation algorithm.

A path p is a sequence of nodes $\mathbb{N}_p$. Each node $n \in \mathbb{N}_p$ maintains the previous node $prev(n)$, and the used incoming and outgoing interfaces i_n, o_n, respectively. In addition, the last node in p that provides any service to the multicast session s is referred to as l_p. The path p also has a parent path $par(p)$ which is used to build the final graph and validate forwarding decisions. The graph calculation algorithm calculates a weight value w_p for path p, which determines the cost of forwarding and processing traffic on links and nodes of p, respectively.

The CALCULATEGRAPH Algorithm. The CALCULATEGRAPH algorithm, shown in Algorithm 1, takes as inputs the ISP network topology ($\mathbb{N}$, $\mathbb{E}$), the generated segment store $\mathcal{S}$, and the multicast session s. The algorithm then calculates and returns the distribution graph $\mathcal{D}$. The algorithm iterates over the destinations and invokes the FINDPATHS function for every destination, which we will describe in details later.

The algorithm initializes a new graph $\mathcal{D}$ with a single path. This first path has only one node, which is the source of the multicast session. For every destination in the multicast session, the algorithm calculates a set of minimum-cost paths that satisfy the service chaining requirements $\mathbb{V}$ as follows. To pick a destination (Lines 3–5), the algorithm builds a weighted graph $(\mathbb{N}, \mathbb{E})$, assigns for each link $l \in \mathbb{E}$ the cost value γ_l, and sorts destinations according to the shortest paths from the source (i.e., a destination with the shortest path is picked first).

Algorithm 1 Calculate a multicast distribution graph.

Input: $\mathbb{N}$: ISP locations (or nodes)
Input: $\mathbb{E}$: ISP network links
Input: $\mathcal{S}$: Segment store
Input: s: multicast session
Output: $\mathcal{D}$: the calculated distribution graph

```
1:  function CalculateGraph(N, E, P, s)
2:      D = new_graph(s) // Initialize a graph with one path
3:      unallocated = {}; dsts = SortDestinations(s)
4:      while len(dsts) > 0 do
5:          Pop a destination dst from dsts
6:          paths = null; cost = ∞;
7:          // Search from previous solution to dst
8:          for p ∈ D.get_paths() do
9:              for n ∈ N_p do
10:                 srv = {V_s \ out(o_n, n, s)}
11:                 sol =FindPaths(N, S, s, p, n, dst, srv)
12:                 if sol.cost < cost then
13:                     paths = sol.paths
14:                     cost = sol.cost
15:         // Paths were found to reach dst
16:         if paths != null then
17:             update_resources(N, E, paths)
18:             D.update_graph(paths)
19:         else
20:             dst.traversal_count += 1
21:             Push dst to the end of dsts
22:         // Start dynamic segment generation
23:         if dst.traversal_count == 2 then
24:             Generate new segments and add them to S
25:         else if dst.traversal_count > 2 then
26:             Pop dst from dsts
27:             Push dst to unallocated
28:     return D
```

This sorting ensures that the algorithm does not use long paths that may lead to higher routing costs.

The algorithm then iterates over all nodes in the calculated paths so far. Each traversed node represents a candidate branching in the graph to reach the destination. For every traversed node n, the algorithm computes the set of remaining services to be satisfied if the graph would branch at n (Line 10), and calculates the set of paths by calling the FINDPATHS function (Line 11), which returns a set of paths and their costs. The CALCULATEGRAPH algorithm uses the minimum-cost paths, and updates the available resources and $\mathcal{D}$. The algorithm also updates the packet classes for every interface in the nodes belonging to the calculated paths.

The algorithm implements two fall-back strategies to improve its decisions. First, if the algorithm cannot calculate paths to a destination, it pushes that destination to the end of destination list to be revisited again (Line 21). This is because the calculated graph would grow as the algorithm allocates more paths, and thus, the algorithm would have better

search space for that destination. The second strategy is to dynamically generate new segments and add them to $\mathcal{S}$, which happens if the algorithm could not calculate paths twice for the same destination (Lines 22–24). After updating the segment store, the algorithm traverses the destination again.

The dynamic segment generation updates the segment store $\mathcal{P}$ to reflect the available ISP resources after allocating preceding multicast sessions. It removes links with full capacity from the network, and then triggers the GENERATESEGMENTS algorithm. However, instead of generating segments for all node pairs in the network, it generates segments for overloaded node pairs only. An overloaded node pair is a node pair whose all segments in $\mathcal{P}$ have reached full link capacity.

The FINDPATHS Algorithm. The proposed FINDPATHS algorithm, shown in Algorithm 2, computes the service-chained paths that satisfy the set of services srv of a multicast session s from a source src to a destination dst. Notice that src and dst may not be necessarily the source and a destination of the multicast session. The calculated paths and their costs are maintained in a solution object referred to as sol.

The main idea of the algorithm is to recursively break down a segment in $\mathcal{S}$ from src to dst into smaller fragments when srv cannot be satisfied directly using that segment. Each path is calculated to minimize the routing cost while satisfying srv. This segment breakdown enables the algorithm to explore a larger search space when it cannot find an immediate segment in the segment store. The recursive algorithm has two steps to realize this idea, which are denoted by A and B in Algorithm 2.

The first step, denoted by A, examines segments from src to dst in $\mathcal{S}$, calculates candidate paths, and returns the minimum-cost path. To calculate a path scp from a segment seg in $\mathcal{S}$, the algorithm first calculates the sequence of services from srv that can be satisfied using seg (Line 6). The algorithm maintains the calculated sequence as a map srv_map, where the key is the node, and the value is the list of services provided by that node. Given s and srv_map, the algorithm calculates the path weight w between 0 and 1 (Line 7) as:

$$W_1 \sum_{l} link\_weight(l,s) + W_2 \sum_{v} \sum_{n} node\_weight(v,n,s),$$

where W_1 and W_2 are normalization factors, l and n are links and nodes belonging to the path, and v represents all services in srv. We calculate the individual link weight $link\_weight(l,s)$ to balance the traffic across network paths, by considering both the network condition (i.e., link usage) and network structure (i.e., link importance). Thus, we calculate the link weight based on its cost γ_l, usage u_l, and importance factor f_l as follows:

$$link\_weight(l,s) = \begin{cases} \gamma_l \alpha_{l,s}^{1+f_l}, & \alpha_{l,s} > 0.5 \\ \gamma_l \alpha_{l,s}, & \text{otherwise,} \end{cases} \quad (1)$$

where $\alpha_{l,s} = \big(u_l + b_s\big)/c_l$. We calculate the link importance factor f_l as its *betweenness score*, which is the sum of the fraction of all-pairs shortest paths that pass through this link. It assigns higher weights to links that have higher probability to carry traffic. The link weight in Equation (1) grows exponentially as the link usage increases and exceeds half of its capacity. The exponential growth rate is proportional to the link betweenness score. In other words, a link that is frequently used and located on a critical path costs more to allocate. We calculate the node weight $node\_weight(v,n,s)$ based on the bandwidth b_s and the total and used processing capacities (in bits/s) for the service v at n as $\big(u_{v,n} + b_s\big)/p_{v,n}$.

The algorithm then creates a candidate path scp given the

Algorithm 2 Find service-chained paths to a destination.

Input: $\mathbb{N}$: ISP locations (or nodes)
Input: $\mathcal{D}$: Segment store
Input: s: multicast session
Input: p: parent path
Input: src, dst: source and destination
Input: srv: set of ordered services
Output: sol: service-chained paths and their costs

1: **function** FINDPATHS($\mathbb{N}, \mathcal{S}, s, p, src, dst, srv$)
2: sol = new_solution(); $cand\_paths$ = {}
3: A Find a path from src to dst that satisfies srv
4: **for** $seg \in \mathcal{S}[src, dst]$ **do**
5: // Locate services and calculate weights
6: srv_map = SERVICESALONGSEGMENT(s, seg, srv)
7: w = CALCULATEWEIGHTS(sol, s, seg, srv_map)
8: // Build a service-chained path
9: scp = CREATEPATH(s, p, seg, w, srv_map)
10: // Check link capacities and packet classes
11: **if not** ISVALIDPATH($\mathbb{N}, scp$) **then**
12: **continue**
13: $cand\_paths$.add(scp)
14: **Pick** the path scp with min. cost and max. # services
15: $sol.paths$ = {scp}; $sol.cost$ = $scp.cost$
16: **if** ISCOMPLETEPATH(scp, srv) **then**
17: **return** sol
18: B Find paths to satisfy remaining services
19: **Adjust** scp and its cost
20: $sol.paths$ = {scp}; $sol.cost$ = $cost$
21: src = l_{scp}; $next\_cost$ = ∞; $next\_src$ = $null$
22: v = $scp.next\_srv$ // First unsatisfied service
23: $nodes$ = $\mathbb{N}.get\_nodes\_by\_srv(v)$
24: **for** $m \in nodes$ **do**
25: tmp_sol = FINDPATHS($\mathbb{N}, \mathcal{S}, s, scp, src, m, \{v\}$)
26: **if** tmp_sol.cost $<$ $next\_cost$ **then**
27: $next\_src$ = m; $next\_sol$ = tmp_sol
28: $next\_cost$ = $tmp\_sol.cost$
29: $sol.paths$.add($next\_sol.paths$)
30: $sol.cost$ += $next\_sol.cost$; src = $next\_src$
31: **Update** p to be the last service-chained path in sol
32: **Update** srv to be the set of remaining services
33: $next\_sol$ = FINDPATHS($\mathbb{N}, \mathcal{S}, s, p, src, dst, srv$)
34: $sol.sc\_paths$.add($next\_sol.sc\_paths$)
35: $sol.cost$ += $next\_sol.cost$
36: **return** sol

calculated satisfied services and path weight (Line 9). The algorithm sets the scp parent to p and updates the packet classes for the incoming interface i_n for every node $n \in \mathbb{N}_{scp}$ as $inc(i_n, n, s) = out(o_{prev(n)}, prev(n), s)$, where $prev(n)$ is the preceding node to n in $\mathbb{N}_{scp}$. If n is the first node, then $prev(n) = src$. Similarly, the algorithm sets the packet classes for the outgoing interface o_n for every node $n \in \mathbb{N}_{scp}$ as $out(o_n, n, s) = inc(i_n, n, s) \cup srv\_map[n]$.

The algorithm then validates the calculated paths as follows. It first checks that all link, processing and forwarding table capacities are not exceeded. Then, it rejects the path scp if it would introduce forwarding ambiguity at routers. Forwarding ambiguity occurs when the same packet class appears more than once for the same session in any incoming or outgoing interface in $\mathbb{N}$ if scp would be returned. Duplicate packet classes on an incoming interface means that a router cannot tell which outgoing interface these packets should be forwarded to, while duplicate packet classes on an outgoing interface means that an ISP location produces more traffic than expected.

After validating all paths, the algorithm picks the minimum-cost path with maximum number of satisfied services. The algorithm then returns this path if it satisfies all services srv. Otherwise, the algorithm recursively composes a path satisfying the required services by breaking down the path to dst by merging sub-paths that do not belong to $\mathcal{S}[src, dst]$ in the second step of the algorithm, denoted by B, as follows. The algorithm recursively calls itself to calculate sub-paths from the last node in scp that provides a service to a node m that supports the first unsatisfied service as well as sub-paths from m to dst. This ensures that the algorithm controls each link in $\mathcal{D}$. First, in Lines 21–29, the algorithm sets src to be the last node in scp that provides any services (denoted by l_{scp}), and sets v to the first unsatisfied service in srv by scp. The algorithm then retrieves all nodes that provide the service v without exceeding their CPU resources. The algorithm calls itself with the new source, destination and set of services to be satisfied, and chooses the minimum-cost paths. Notice that for each of these recursive calls, the parent path is the sub-path scp calculated from step A. Second, the algorithm then finds sub-paths from m to dst (Lines 31–34). This happens by recursively calling itself as well with different source and services. The algorithm sets the source to be m, the services to be the remaining services after finding the node m, and the parent to be the path calculated by the previous recursive call.

E. Illustrative Example

We describe an illustrative example of the proposed algorithm. Figure 2 depicts an ISP network consisting of 14 nodes. The capacity of every link in the network is a single unit of bandwidth, and the cost of forwarding one unit of bandwidth on every link is 1. The figure shows the services that are deployed at every node (in squares).

There is a multicast session of a live video to be streamed from node a to nodes k, l, m and n. Packets of this session are transmitted at one unit of bandwidth, and they need to be processed by a chain of the following services: IDS (1) $\rightarrow$ Firewall (2) $\rightarrow$ Encoder (3) $\rightarrow$ Content Manager (4) $\rightarrow$ Ad Insertion (5) $\rightarrow$ Transcoder (6). For simplicity, we assign to each service an ID from 1 to 6.

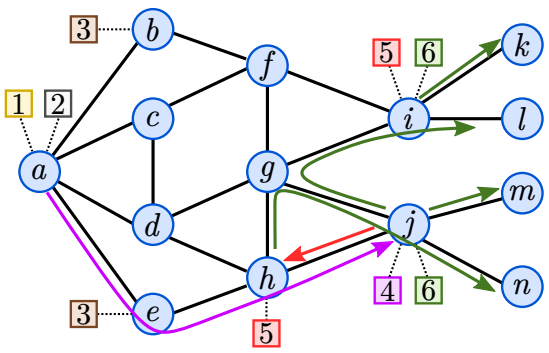


Fig. 2: Illustrative example of the proposed Oktopus algorithm.

We first show how the proposed algorithm allocates ISP resources from the source at node a to one of the destinations at node n. Notice that there is no single segment, i.e., a sequence of nodes from a to n, that can satisfy the service chaining requirements. To address this, Oktopus breaks down the path to node n into multiple paths, each of which satisfies a sub-sequence of the required services. This is done by step B in Algorithm 2. In this example, the proposed algorithm calculates three paths from node a to node n to satisfy the service chaining requirements as shown in Figure 2.

The first calculated path is $\{\langle a, \{1, 2\}\rangle, \langle e, \{3\}\rangle, \langle h, \phi\rangle, \langle j, \{4\}\rangle\}$ to satisfy the $\{1, 2, 3, 4\}$ service requirements (Line 19 in Algorithm 2). Notice that although service 3 is deployed at b, our algorithm does not include it in the sub-path because it results in a larger routing cost.

Next, the algorithm searches from node j for all nodes that provide service 5, since node j is the last node that supports the maximum number of required services (Line 21 in Algorithm 2). Since both nodes h and i provide this service, the algorithm sets the source to j and recursively calls itself twice, each of which with a different destination (Lines 24–29 in Algorithm 2). The two recursive calls return $\{\langle j, \phi\rangle, \langle h, \{5\}\rangle\}$ and $\{\langle j, \phi\rangle, \langle g, \phi\rangle, \langle i, \{5\}\rangle\}$, respectively. Our algorithm chooses the path $\{\langle j, \phi\rangle, \langle h, \{5\}\rangle\}$ to satisfy the service $\{5\}$ as it results in lower routing cost from j.

The algorithm then calculates a path from node h to node n while satisfying service $\{6\}$. The algorithm sets the source to h and the destination to n and recursively calls itself (Line 33 in Algorithm 2). This recursive call returns the path $\{\langle h, \phi\rangle, \langle g, \phi\rangle, \langle j, \{6\}\rangle, \langle n, \phi\rangle\}$ as it has the lowest cost.

Finally, to calculate paths to the remaining destinations, the algorithm finds a node from which the calculated paths result in the lowest routing cost. As shown in Figure 2, node j is used to calculate the paths to destinations m and l, and node i is used to reach destination k.

F. Analysis and Practical Considerations

Time and Space Complexities. The following lemma shows the worst-case time and space complexities of Oktopus.

Lemma 1. *The proposed algorithm terminates in polynomial time in the order of $\mathcal{O}(N^4E^2)$ per session, where N and E are the numbers of ISP locations and links, respectively. The space complexity of the algorithm is $\mathcal{O}(N^3E)$.*

Proof Sketch: The proposed algorithm consists of two steps: segment generation and graph calculation. The time complexity of the first step is $\mathcal{O}(N^3E^2)$ as it iterates over each node pair and runs breadth-first traversal and Edmonds–Karp algorithms. The time complexity of the second step to calculate the distribution graph is $\mathcal{O}(N^4E^2)$, which searches for the best node to connect a new path to and dynamically generates new segments when needed.

The space complexity of the algorithm is $\mathcal{O}(N^3E)$ because it generates K segments for every node pair using breadth-first traversal and $\mathcal{O}(E)$ edge-disjoint segments each of size N. ■

Practicality. Although the time complexity of the proposed algorithm may appear large, we believe the algorithm is practical and can run for real ISP topologies for the following reasons. First, the time complexity analysis is for the *worst-case* scenario, which assumes that the dynamic segment generation and recursive composition of services steps happen for every destination in the multicast session, which is very unlikely for most practical situations. If the algorithm does not invoke the dynamic segment generation, the time complexity reduces to $\mathcal{O}(N^3)$, which is further reduced to $\mathcal{O}(N^2)$ if the recursive compositioning of services is not invoked. We note that these two steps are *optional* in our algorithm and they are invoked only in case of not finding a solution based on the precomputed segment store. That is, the network operator may disable either or both of these steps for fast computations, albeit at the cost of increasing the possibility of not finding a solution in some rare cases. Second, the values of N and E are not large for realistic ISP networks. The number of ISP locations N is in the range of 10's–100's [33], [35], [34], [36], and most ISP networks are sparse with number of links E ranging from 500 to around 2,000. Our experiments in §V using real ISP topologies show that the unoptimized, sequential, version of Oktopus takes, on average, a few seconds to compute the distribution graph for a given multicast session on a commodity workstation.

Finally, we note that many steps of the proposed algorithm can run in parallel to reduce the running time. For example, there are parallel variants of the breadth-first and edge-disjoint traversal algorithms used by the segment generation algorithm. Moreover, the body of the first loop in Algorithm 2 can run in parallel.

Handling Link Failures. Oktopus handles failures as follows. For a failed link (a, b), the algorithm sets its capacity to zero. Then, for every distribution graph that includes the failed link, the algorithm calculates new paths from a to all direct children of b in the graph. Specifically, for every calculated path $p = \{\langle i, S_i\rangle, \ldots, \langle a, S_a\rangle, \langle b, S_b\rangle, \langle j, S_j\rangle, \ldots\}$ that includes (a, b), we reset this path to $\{\langle i, S_i\rangle, \ldots, \langle a, S_a\rangle\}$. Then, we re-run Algorithm 1 with a destination set to be the direct child of b (i.e., j) and srv to be S_b.

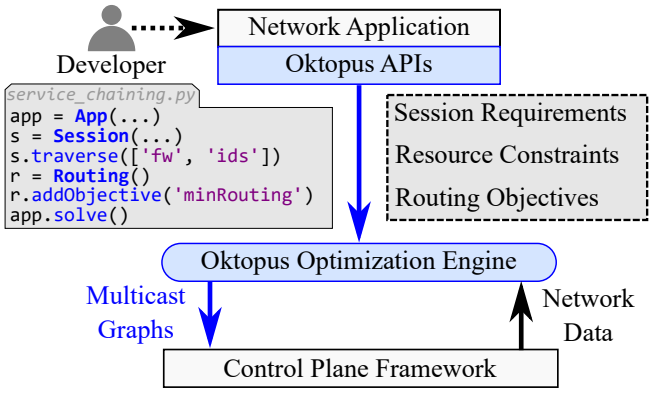


Fig. 3: Overview of the open-source Oktopus framework.

Deployment. Oktopus can be easily integrated with SDN-managed ISP networks. For ISP networks that rely on distributed routing protocols, they can support the centralized control needed by Oktopus using ideas such as Fibbing [37].

V. Implementation and Evaluation

A. Implementation

We have implemented a complete framework to support managing multicast service chaining in ISP networks. The framework is implemented in Python and it is open-source [38]. Figure 3 shows this framework, where the left part of the figure illustrates an example of using the framework.

As shown in Figure 3, the framework has two main components: Optimization Engine and APIs. The Optimization Engine implements the algorithm presented in §IV to compute a multicast distribution graph for each given multicast session. The APIs provide easy interfaces for developers and operators to create and manage multicast sessions. They are divided into three subsets: App, Session, and Routing. The App APIs are used to define the ISP topology and available resources. The Session APIs are used to specify the multicast session parameters including their service chaining requirements. The Routing APIs are used to set the routing objectives (e.g., minimum routing cost) and link constraints.

B. Evaluation Setup

To evaluate Oktopus and compare it against others, we implemented a simulator that calls our framework with various parameters. We implemented the Oktopus, optimal (OPT), and MSA [21] algorithms in the current version of our framework. We implemented OPT using CPLEX 12.8. We implemented MSA because it is the closest algorithm in the literature solving the considered multicast service chaining problem.

We use the following five real ISP topologies from the Internet Topology Zoo [39]: AttMpls (25 nodes), Dfn (57 nodes), Columbus (70 nodes), Ion (125 nodes) and Colt (153 nodes). We chose these topologies as they constitute representative samples of different network sizes. We set the

link capacity to 10Gbps, and link cost to 1 per bandwidth unit. Moreover, due to the lack of publicly available data on service chaining, we use the results of a recent study [40] to generate the service chain requirements.

We control five parameters for every experiment as follows. (i) Session characteristics: for every multicast session, its source node is chosen randomly. In addition, the percentage of nodes to be destinations is chosen randomly to be 10%, 20%, 30%, or 40% of the total nodes. (ii) Number of sessions: we vary the number of multicast sessions from 1,000 to 4,000. For comparison versus OPT, we set the number of multicast sessions to 10 to ensure that OPT produces a solution within 24 hours. In each generated dataset, 21%, 57% and 22% of the multicast sessions have bandwidth of 2 Mbps, 7.2 Mbps and 15 Mbps, respectively [41]. (iii) Service chain length: we vary the length of the service chain from 3 to 6. (iv) Service deployment: since different services have different usage patterns [40], we categorize services into two types, essential and auxiliary type. Essential services are popular ones such as firewalls and IDS, and they are deployed at all ISP nodes. Auxiliary services include services that are not commonly used such as video encoders and traffic monitors. For each dataset, we set the percentage of nodes that provide auxiliary services to be one of 5%, 15%, 25%, 35% or 45%. (v) Service chain ordering: we generate service chains with different ordering of essential and auxiliary services. Specifically, we have two orderings: partial and random. In partial ordering, the essential services are ordered before the auxiliary services. The first and second services of the service chain are randomly chosen from essential services, and the rest of the services are chosen from auxiliary services. In random ordering, services in the service chain are randomly ordered.

We consider four important performance metrics that measure the quality of the allocated multicast sessions: (i) Percentage of allocated multicast sessions, which is the ratio between the allocated and total numbers of multicast sessions. The larger the percentage the better the fulfillment of the ISP customer requirements. (ii) Average routing cost, which is the cost of forwarding packets of a multicast session on links of its distribution graph, and it is defined by $b \sum_{l \in \mathbb{L}} \gamma_l$. (iii) Average graph size, which is the average number of links of the calculated distribution graph per session. A small graph indicates lower delays to reach all destinations. (iv) Average running time, which is the average elapsed time to calculate a distribution graph per session. For the routing cost and graph size, we calculate the standard deviation of the randomly generated multicast sessions.

In the following, we show representative samples of our results due to space limitations; other results are similar.

C. Oktopus versus OPT

We compare Oktopus versus OPT in terms of the routing cost, graph size and running time. We ran OPT on a server with 128 GB of memory and configured CPLEX to terminate in 24 hours. We set the numbers of multicast sessions to 10 and 50.

ISP	Routing Cost Opt. Gap	Avg. Graph Size		Avg. Running Time	
		OPT	Oktopus	OPT	Oktopus
AttMpls	4.75%	10.7	11.3	1.2 s	0.2 s
Dfn	1.75%	23.7	24.1	18 s	0.4 s
Columbus	8.5%	40.7	44.5	42 s	0.6 s
Ion	5.9%	71.9	75.1	1.8 hrs	3.5 s
Colt	5.5%	70.7	74.4	1.4 hrs	6 s

TABLE I: Results of Oktopus versus OPT for 10 sessions.

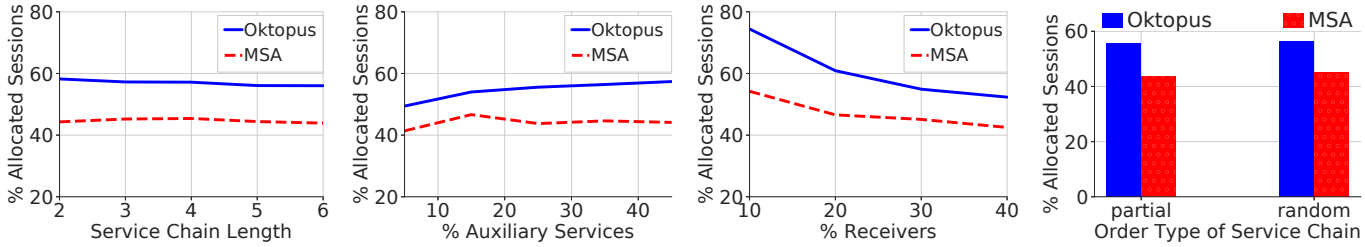
Results for 10 Sessions. We summarize the results in Table I. We note that both algorithms allocated all multicast sessions, so we focus on the other performance metrics. First, the largest optimality gap between Oktopus and OPT in the considered topologies is 8.5% for the Columbus topology. This gap ranges from 1.75% to 5.9% for the other topologies. On average, the routing cost optimality gap is 5.3%. Second, Oktopus calculates distribution graphs with similar sizes to the OPT counterparts. For instance, for the largest topology in our dataset, Oktopus calculates graphs with only additional five links per session on average. Moreover, Oktopus produces distribution graphs with sizes that are 5.5% larger than the ones produced by the optimal algorithm on average. Finally, Oktopus is much faster than OPT. As the size of the network increases, the running time of OPT explodes. For example, OPT requires 1.8 hrs per session for the Ion topology, while Oktopus calculates a graph in 3.5s per session. Moreover, running OPT requires a large amount of memory. For example, for the Ion ISP topology, OPT requires 55 GB of memory, while Oktopus requires less than 1 GB of memory.

Results for 50 Sessions. In this case, OPT could not produce a valid solution for large ISP topologies. For the Ion ISP topology, for example, OPT spent 24 hours and required about 100 GB of memory without calculating a final solution. Oktopus calculated valid distribution graphs for the 50 sessions in 72 seconds and used less than 1 GB of memory.

D. Oktopus versus the Closest Algorithm

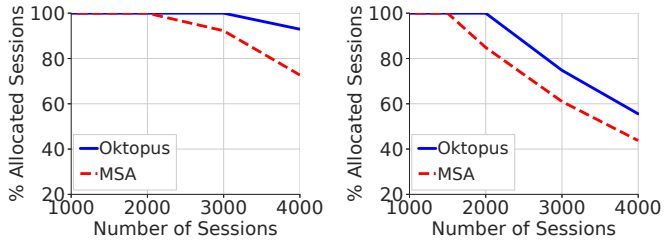
Percentage of Allocated Sessions. We first show in Figure 4 the percentage of allocated multicast sessions in the Ion ISP topology (of 125 nodes) for different service chain lengths, deployed auxiliary services percentages, receiver densities, and service chain orderings. We also set the number of multicast sessions to 4,000 in these experiments to stress Oktopus and MSA. The figure shows that Oktopus outperforms MSA across all the considered scenarios.

Figure 4a shows that Oktopus allocates more multicast sessions for all service chain lengths. Moreover, the figure shows that Oktopus performance is consistent even when increasing the service chain length. For instance, it increases the percentage of allocated multicast sessions by 31% and 30% for service chain length of 2 and 6, respectively. Figure 4b shows the allocation performance when increasing the percentage of deployed auxiliary services. Oktopus can utilize the added CPU resources more efficiently than MSA as it allocates 30% more multicast sessions when more auxiliary services are



(a) Service chain length (b) Service deployment (c) Receiver density (d) Service order

Fig. 4: Percentage of allocated multicast sessions for different scenarios. # multicast sessions is 4,000.



(a) Dfn (b) Ion

Fig. 5: Percentage of allocated multicast sessions for different ISP network topologies.

deployed to more nodes. This is because Oktopus considers the CPU utilization across the available services.

Figure 4c shows that Oktopus allocates more multicast sessions compared to MSA for different numbers of destinations. For instance, Oktopus increases the number of allocated multicast sessions by up to 37% when the receiver density is 10% of the nodes. In another experiment, we control the order of required services in the service chains; recall that Oktopus does not assume any knowledge of the service chain order. Figure 4d demonstrates that the order of services within service chains does not impact the allocation performance, and that the performance of Oktopus is robust even when the service order is random.

Next, we present the percentage of allocated multicast sessions for two ISP topologies with different sizes in Figure 5. The figure shows that Oktopus allocates more multicast sessions than MSA especially when the number of multicast sessions increases, and its performance is consistent across the different topologies. For example, when the number of multicast sessions is 4,000, Oktopus increases the percentage of allocated multicast sessions by 27% and 28% for the two topologies, respectively. This means that Oktopus calculates better paths than MSA while satisfying the required services, because the proposed path-based approach carefully engineers each link in the calculated graphs while balancing the load across different links.

Routing Cost and Graph Size. We next measure the average routing cost and calculated graph size for the Ion ISP topology while varying the number of multicast sessions. We observed similar results for other topologies and scenarios. The figure is not shown due to space limitations. Our results show that Oktopus has similar routing cost per multicast session as MSA, while Oktopus allocates more multicast sessions as previously shown in Figure 5b. In addition, Oktopus and MSA compute similar graph sizes. Moreover, the standard deviation of the graph size of Oktopus is slightly larger than its counterpart of MSA (by up to two links) as Oktopus balances the traffic across the links.

Running Time. We measure the average running time to calculate a distribution graph per session for both algorithms. Our measurements show that Oktopus adds a negligible overhead on average. Specifically, the average running time is 8.625s and 8.5s per session for Oktopus and MSA algorithms, respectively. This slight overhead is used towards calculating better distribution graphs as shown in the previous experiments.

VI. Conclusions

We considered the problem of multicast service chaining, which is NP-hard as we showed in the paper. We proposed a new algorithm, called Oktopus, to calculate multicast distribution graphs that minimize the routing cost per session. The main idea of Oktopus is to recursively calculate and merge paths that fulfill the required services from the source to all destinations. This path-based approach enables Oktopus to have control over calculated links in the distribution graphs, which improves the quality of the calculated graphs. We implemented and evaluated Oktopus using real ISP topologies. We compared Oktopus versus the optimal solution and the closest algorithm in the literature. Our experiments showed that Oktopus computes the distribution graphs with a small routing cost optimality gap while terminating multiple orders of magnitude faster than the optimal algorithm. Moreover, Oktopus increases the number of allocated multicast sessions by up to 37% compared to the closest algorithm.

Acknowledgment

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

REFERENCES

[1] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker, "Resq: Enabling slos in network function virtualization," in *Proc. of USENIX NSDI'18*, Renton, WA, April 2018, pp. 283–297.

[2] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Proc. of USENIX NSDI'14*, Seattle, WA, April 2014, pp. 459–473.

[3] E. J. Jackson, M. Walls, A. Panda, J. Pettit, B. Pfaff, J. Rajahalme, T. Koponen, and S. Shenker, "Softflow: A middlebox architecture for open vswitch," in *Proc. of USENIX ATC'16*, Denver, CO, June 2016, pp. 15–28.

[4] C. Sun, J. Bi, Z. Zheng, H. Yu, and H. Hu, "Nfp: Enabling network function parallelism in nfv," in *Proc. of ACM SIGCOMM'17*, Los Angeles, CA, August 2017, pp. 43–56.

[5] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "Netbricks: Taking the v out of NFV," in *Proc. of USENIX OSDI'16*, Savannah, GA, November 2016, pp. 203–216.

[6] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy, "Safebricks: Shielding network functions in the cloud," in *Proc. of USENIX NSDI'18*, Renton, WA, April 2018, pp. 201–216.

[7] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," in *Proc. of USENIX NSDI'13*, Lombard, IL, April 2013, pp. 227–240.

[8] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: Enabling innovation in network function control," in *Proc. of ACM SIGCOMM'14*, Chicago, IL, August 2014, pp. 163–174.

[9] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park, "mos: A reusable networking stack for flow monitoring middleboxes," in *Proc. of USENIX NSDI'17*, Boston, MA, March 2017, pp. 113–129.

[10] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, "Clicknp: Highly flexible and high performance network processing with reconfigurable hardware," in *Proc. of ACM SIGCOMM'16*, Florianopolis, Brazil, August 2016, p. 1–14.

[11] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella, "Paving the way for NFV: Simplifying middlebox modifications using statealyzr," in *Proc. USENIX NSDI'16*, Santa Clara, CA, March 2016, pp. 239–253.

[12] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, "Elastic scaling of stateful network functions," in *Proc. USENIX NSDI'18*, Renton, WA, April 2018, pp. 299–312.

[13] C. Lan, J. Sherry, R. A. Popa, S. Ratnasamy, and Z. Liu, "Embark: Securely outsourcing middleboxes to the cloud," in *Proc. of USENIX NSDI'16*, Santa Clara, CA, March 2016, pp. 255–273.

[14] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making middleboxes someone else's problem: Network processing as a cloud service," in *Proc. of ACM SIGCOMM'12*, Helsinki, Finland, August 2012, p. 13–24.

[15] B. Trach, A. Krohmer, F. Gregor, S. Arnautov, P. Bhatotia, and C. Fetzer, "Shieldbox: Secure middleboxes using shielded execution," in *Proc. of ACM Symposium on SDN Research (SOSR'18)*, Los Angeles, CA, March 2018.

[16] "Zuckerberg really wants you to stream live video on Facebook," https://bit.ly/2v6uHqF, Wired, April 2016, [Online; accessed August 2020].

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

[18] 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*, vol. 31, no. 9, pp. 155–164, September 2013.

[21] 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*, July 2018, pp. 132–142.

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

[20] A. Raman, G. Tyson, and N. Sastry, "Facebook (A)Live?: Are Live Social Broadcasts Really Broadcasts?" in *Proc. of the International World Wide Web Conference (WWW'18)*, Lyon, France, April 2018.

[22] G. Sallam, G. R. Gupta, B. Li, and B. Ji, "Shortest path and maximum flow problems under service function chaining constraints," in *Proc. of IEEE INFOCOM'18*, 2018, pp. 2132–2140.

[23] H. Feng, J. Llorca, A. M. Tulino, D. Raz, and A. F. Molisch, "Approximation algorithms for the nfv service distribution problem," in *Proc. of IEEE INFOCOM'17*, 2017, pp. 1–9.

[24] T. Kuo, B. Liou, K. C. Lin, and M. Tsai, "Deploying chains of virtual network functions: On the relation between link and server usage," in *Proc. of IEEE INFOCOM'16*, 2016, pp. 1–9.

[25] Z. Xu, W. Liang, M. Huang, M. Jia, S. Guo, and A. Galis, "Approximation and online algorithms for nfv-enabled multicasting in sdns," in *Proc. of IEEE ICDCS'17*, June 2017, pp. 625–634.

[26] J. Kuo, S. Shen, M. Yang, D. Yang, M. Tsai, and W. Chen, "Service overlay forest embedding for software-defined cloud networks," in *Proc. of IEEE ICDCS'17*, June 2017, pp. 720–730.

[27] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, Jan 2015.

[28] S. A. Amiri, K.-T. Foerster, R. Jacob, and S. Schmid, "Charting the algorithmic complexity of waypoint routing," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 48, no. 1, p. 42–48, Apr. 2018.

[29] L. Huang, H. Hsu, S. Shen, D. Yang, and W. Chen, "Multicast traffic engineering for software-defined networks," in *Proc. of IEEE INFOCOM'16*, April 2016, pp. 1–9.

[30] C. Filsfils, S. Previdi, L. Ginsberg, B. Decraene, S. Litkowski, and R. Shakir, "Segment Routing Architecture," RFC 8402, Tech. Rep. 8402, July 2018.

[31] J. Edmonds and R. M. Karp, "Theoretical improvements in algorithmic efficiency for network flow problems," *Journal of the ACM*, vol. 19, no. 2, p. 248–264, Apr. 1972.

[32] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[33] R. Hartert, S. Vissicchio, P. Schaus, O. Bonaventure, C. Filsfils, T. Telkamp, and P. Francois, "A declarative and expressive approach to control forwarding paths in carrier-grade networks," in *Proc. of ACM SIGCOMM'15*, London, United Kingdom, 2015, pp. 15–28.

[34] V. Heorhiadi, S. Chandrasekaran, M. K. Reiter, and V. Sekar, "Intent-driven composition of resource-management sdn applications," in *Proc. of ACM CoNEXT'18*, New York, NY, USA, 2018, pp. 86–97.

[35] N. Spring, R. Mahajan, and D. Wetherall, "Measuring isp topologies with rocketfuel," *IEEE/ACM Transactions on Networking (TON)*, vol. 12, no. 1, pp. 2–16, February 2004.

[36] V. Heorhiadi, M. K. Reiter, and V. Sekar, "Simplifying software-defined network optimization using SOL," in *Proc. of USENIX NSDI'16*, Santa Clara, CA, March 2016, pp. 223–237.

[37] S. Vissicchio, O. Tilmans, L. Vanbever, and J. Rexford, "Central control over distributed routing," in *Proc. of ACM SIGCOMM'15*, London, United Kingdom, August 2015, p. 43–56.

[38] "Oktopus," https://oktopus-project.org, [Online; accessed August 2020].

[39] "The Internet Topology Zoo," http://www.topology-zoo.org/dataset.html, The University of Adelaide, July 2012, [Online; accessed August 2020].

[40] B. Donnet, K. Edeline, I. R. Learmonth, and A. Lutu, "Middlebox classification and initial model," https://bit.ly/3dZelXV, Middlebox Classification and Initial Model, [Online; accessed August 2020].

[41] "Cisco annual internet report (2018–2023) white paper," https://bit.ly/2LK6q4M, March 2020, [Online; accessed August 2020].