Service Chaining for Multicast Traffic

by

Carlos Lee

B.Sc., University of Toronto, 2017

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

© Carlos Lee 2020
SIMON FRASER UNIVERSITY
Summer 2020

Copyright in this work rests with the author. Please ensure that any reproduction or re-use is done in accordance with the relevant national copyright legislation.

Approval

Name: **Carlos Lee**

Degree: **Master of Science (Computing Science)**

Title: **Service Chaining for Multicast Traffic**

Examining Committee:

Chair: Ouldooz Baghban Karimi
Lecturer

Mohamed Hefeeda
Senior Supervisor
Professor

Khaled Diab
Supervisor
University Research Associate

Keval Vora
Internal Examiner
Assistant Professor
School of Computing Science

Date Defended: **July 20, 2020**

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 thesis, 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. Oktopus consists of two steps. The first one generates a set of network segments for the ISP network, 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, which leads to significant improvements in the quality of the calculated graphs. 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 in simulations 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%.

Keywords: Multicast Traffic Engineering; Service Chaining; ISP Network

Table of Contents

Approval ii

Abstract iii

Table of Contents iv

List of Tables vi

List of Figures vii

1 Introduction 1

1.1 Problem Statement 2

1.2 Thesis Contributions 2

1.3 Thesis Organization 3

2 Background and Related Work 4

2.1 Background 4

2.1.1 Software-defined Networking 4

2.1.2 Network Function Virtualization 5

2.2 Related Work 7

2.2.1 Service Chaining 7

2.2.2 Service Deployment 8

2.2.3 Traffic Engineering (TE) 8

2.2.4 Control-plane Frameworks 8

3 System Model and Problem Definition 10

3.1 System Model 10

3.2 Problem Definition and Hardness 12

4 Proposed Oktopus Algorithm 14

4.1 Definitions 14

4.2 High-level Overview 14

4.3 Segment Generation 15

4.4 Calculating the Multicast Distribution Graph . . . 17
4.5 Illustrative Example . . . 22
4.6 Analysis of Oktopus . . . 23

5 Implementation and API **26**
5.1 Implementation . . . 26
5.2 Oktopus API . . . 26
5.2.1 Application API . . . 27
5.2.2 Session API . . . 29
5.2.3 Routing API . . . 30
5.2.4 Developer Benefits . . . 30

6 Evaluation **32**
6.1 Setup . . . 32
6.2 Oktopus versus OPT . . . 33
6.3 Oktopus versus the Closest Algorithm . . . 35

7 Conclusion and Future Work **48**
7.1 Conclusion . . . 48
7.2 Future Work . . . 48

Bibliography **50**

List of Tables

Table 5.1 The proposed Session interface. 28
Table 5.2 The proposed Routing interface. 29

Table 6.1 Results of Oktopus versus OPT for 10 sessions. 34

List of Figures

Figure 2.1 OpenFlow switch. The packet enters from the ingress port and matches to one or more flow tables. If the matching entry is found, then action associated is performed, such as forwarding to a particular egress port. To perform multicast forwarding, the packet is forward to the group table where it gets cloned. 5
Figure 2.2 An example of an SDN network with an optimization layer. 6
Figure 2.3 Example of a service chaining requirement. 7
Figure 3.1 A multicast session with a source node a, destination nodes f, g, h and i, and required services $1 \rightarrow 2$. Arrows are links of the multicast distribution graph. 11
Figure 4.1 Illustrative example of a multicast session. 22
Figure 5.1 Operators use the Oktopus API to express the network application. Then, the optimization engine runs the Oktopus algorithm to compute the multicast distribution graph. 27
Figure 6.1 Percentage of allocated multicast sessions for the Ion ISP topology across different scenarios. # multicast sessions: 4,000. 36
Figure 6.2 Percentage of allocated multicast sessions for the AttMpls ISP topology across different scenarios. # multicast sessions: 4,000. 37
Figure 6.3 Percentage of allocated multicast sessions for the Dfn ISP topology across different scenarios. # multicast sessions: 4,000. 38
Figure 6.4 Percentage of allocated multicast sessions for the Columbus ISP topology across different scenarios. # multicast sessions: 4,000. . . 39
Figure 6.5 Percentage of allocated multicast sessions for the Colt ISP topology across different scenarios. # multicast sessions: 4,000. 40
Figure 6.6 Percentage of allocated multicast sessions with different number of multicast sessions. 42
Figure 6.7 Average routing cost and graph size per session for the Ion ISP Topology. 43

Figure 6.8 Average routing cost and graph size per session for the AttMpls ISP Topology. 44
Figure 6.9 Average routing cost and graph size per session for the Dfn ISP Topology. 45
Figure 6.10 Average routing cost and graph size per session for the Columbus ISP Topology. 46
Figure 6.11 Average routing cost and graph size per session for the Colt ISP Topology. 47

Chapter 1

Introduction

Recently, network operators have adopted Network Function Virtualization (NFV) [55, 39] to reduce the cost of purchasing and managing middleboxes such as firewalls and IDSes. In NFV, the functionality of a hardware-based middlebox is implemented as a virtual network function (VNF). This led the research community to explore different aspects of NFV such as implementing VNFs efficiently and securely [30, 52, 40, 41, 44, 21], building network stack for VNFs [31, 38], managing their state [32, 58], and deploying them [37, 49, 56]. For simplicity, we refer to a VNF as a *service*.

ISP networks have observed various changes in terms of their architecture and the complexity of Internet applications. Specifically, the adoption of NFV allows large ISPs to deploy various services at different locations in their networks to support their customer needs, and the adoption of Software Defined Networking (SDN) allows complex Internet application design. 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 [57]), IPTV [23], webinars and video conferencing [11] and massive multiplayer games [13]. The scale of these applications is unprecedented. For instance, Facebook Live aims to stream millions of live sessions to millions of concurrent users [57, 45]. Many large ISPs use multicast to carry traffic of these applications through their networks efficiently. For example, AT&T has deployed AT&T TV, and BT has deployed YouView, where both use multicast.

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 require to pass through a firewall, IDS, and 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 nodes throughout the ISP network, 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 calculate multicast distribution graphs for multicast sessions efficiently.

1.1 Problem Statement

The problem we address in this thesis is designing an efficient multicast service chaining algorithm for large-scale ISPs. 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 satisfy all business requirements. 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.

1.2 Thesis Contributions

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 session in the network. Oktopus has two goals when it calculates the distribution graph: (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 about the ISP topology or its available resources.

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 propose several ideas in the design of Oktopus, 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 as well as the closest algorithm in the literature [46] in simulations 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 graphs in less than two minutes. In addition, 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 utilizes the available ISP resources to minimize the routing cost.

1.3 Thesis Organization

The thesis is organized as follows. Chapter 2 includes a brief background of the ISP architecture and summarizes related work. Chapter 3 presents the system model and the problem definition. Chapter 4 discusses the proposed Oktopus algorithm. Chapter 5 presents the implementation and API of the framework that runs the Oktopus algorithm. The proposed Oktopus algorithm is evaluated in Chapter 6. Chapter 7 concludes the thesis and describes potential future work.

Chapter 2

Background and Related Work

2.1 Background

In this chapter, we provide an overview of software-defined networking and network function virtualization. Then, we discuss the related works in Section 2.2.

2.1.1 Software-defined Networking

Traditionally, Internet service providers (ISPs) managed their networks by writing custom scripts and deploying them to different network devices such as routers and switches. This process is error-prone as it may result in configuration and logic errors, and time-consuming as the operator needs to develop and test every script in isolation and end-to-end. In addition, ISPs suffer from vendor and product lock-in, where ISPs could not easily deploy new hardware from other vendors or even upgrade to new hardware from the same vendor. Thus, managing a large-scale network with heterogeneous devices is a hard task.

To reduce the management costs, ISPs have recently adopted the software-defined networking (SDN) architecture [3]. SDN isolates the control plane that manages the routing logic from the data plane that manages the forwarding logic. It allows ISPs to define how packets are forwarded from a logically centralized controller, as shown in Figure 2.1, to engineer the network traffic.

SDN defines the traffic-engineered path by using match-action rules on the packets' headers. The rules are distributed to the switches by the SDN controller. On the other hand, the conventional protocol Multiprotocol Label Switching (MPLS) [22] defines the path by encapsulating the packet with labels. The labels are distributed by an additional protocol, such as Label Distribution Protocol (LDP) [53]. To enforce the traffic-engineered path in the ISP network using MPLS, ISPs must carefully design the routing logic by computing the labels and the forwarding logic by computing the LDP.

The OpenFlow protocol [34] is the defacto standard to implement SDN. Figure 2.1 shows the components of the OpenFlow switch. An SDN switch contains a set of flow tables, where these tables consist of flow entries that match action rules on the packet headers. The actions

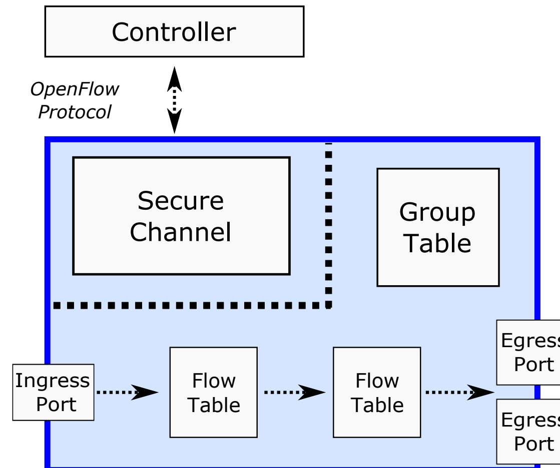


Figure 2.1: OpenFlow switch. The packet enters from the ingress port and matches to one or more flow tables. If the matching entry is found, then action associated is performed, such as forwarding to a particular egress port. To perform multicast forwarding, the packet is forward to the group table where it gets cloned.

include forwarding the packet to the next flow table, dropping the packet, or forwarding the packet to a specific egress port. An SDN switch also contains a group table that consists of group entries to support multicast forwarding [1]. A group entry contains a list of action buckets, where each bucket contains a set of actions. A packet matched to a group entry is cloned for each bucket, and the corresponding bucket actions are applied to the cloned packet.

The flexibility of SDN makes the process of developing new applications at the controller time-consuming. Specifically, network operators need to make custom optimizations suited for each application. Recently, the industry [4, 2, 6] and academia [27, 26, 25] have proposed adding an optimization layer at the control plane to facilitate the SDN application development process. The functionality of this optimization layer includes providing expressive programming interfaces, computing the traffic-engineered paths, and converting the paths to SDN rules. In particular, traffic-engineered paths are realized using match-action rules installed at SDN switches. An example is shown in Figure 2.2. The optimization layer receives the SDN applications request and computes a set of traffic-engineered paths. Then, the calculated rules are installed at the network switches via the SDN controller.

2.1.2 Network Function Virtualization

In addition to network connectivity, ISPs also provide network services to their customers. Specifically, customers of large-scale ISPs are well beyond end users, and they may include content and cloud providers, mobile network operators, and other (often smaller) ISPs. These customers require additional processing of their traffic inside the network through

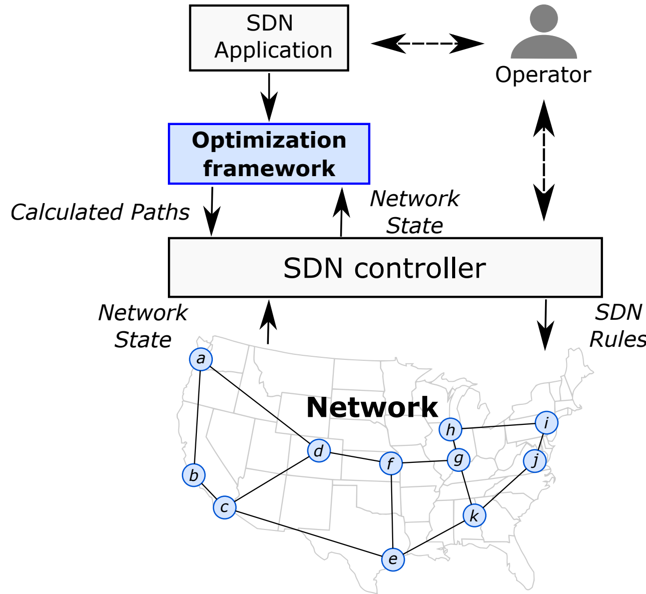


Figure 2.2: An example of an SDN network with an optimization layer.

services. Examples of these network services are deep packet inspection (DPI) [9, 50], firewall, carrier-grade NAT (CG-NAT), load balancers [19], analytics, logging, content caching, video transcoding, and monitoring [5].

These network services are provided as middleboxes [60, 50] deployed inside the ISP network. A middlebox is a specialized hardware designed to execute a dedicated network service. To reduce the cost of purchasing and managing middleboxes, ISPs virtualize these services and deploy them to general-purpose CPUs. This is referred to as Network Function Virtualization (NFV) [55, 39]. Specifically, the functionality of a hardware-based middlebox is implemented as a virtual network function (VNF) and is deployed to servers [49]. NFV offers the same network service capability at a lower cost with the tradeoff of increased latency.

The implementation of NFV consists of three components: flow classifier, service function forwarder (SFF), and the VNF. The classifier examines the incoming flow from the ingress port, and based on the flow type, the required services and the ISP-specified rules, the classifier attaches Network Service Header (NSH) protocol headers [43] between Layer 2 and Layer 3 headers. The NSH header specifies for every packet the service type and the location of the VNF to process the packet. When the packet traverses the network, the SFF parses the NSH header and forwards it to the corresponding server. The last component, the VNF, is the virtualized network service that processes the packet. For example, a firewall VNF has a set of match rules to drop packets, or a virtualized NAT maps IP addresses to/from public and private address space.

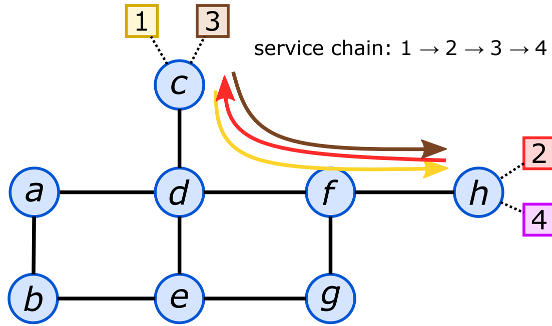


Figure 2.3: Example of a service chaining requirement.

ISP offers the ability for a network application to specify an ordered set of network services that the traffic must pass through [24]. The ordered set of network services is referred to as service chaining. The network services in the service chaining must be strictly processed in the order the application had specified. Service chaining may contain a cycle, or the resulting traffic may traverse the same link multiple times to satisfy the service chaining requirement. Figure 2.3 shows a service chain that is mapped to network paths. The network services numbered 1 and 3 are deployed on node c, and 2 and 4 on node h. The network paths are colored according to the network service to differentiate the traffic passing the links multiple times.

2.2 Related Work

We summarize the relevant works in the following.

2.2.1 Service Chaining

Prior works, e.g., [47, 20, 36], addressed the unicast service chaining problem. For example, the work in [47] proposed to transform the original ISP network to another graph and then to find a service-chained path using the Dijkstra algorithm. The work in [20] considered the joint service chain routing and service deployment problem in unicast. These methods are not applicable to multicast because they may introduce network loops or allocate more network resources as they do not consider the branching nature of multicast.

Several prior works, e.g., [59, 35, 46], addressed the problem of service chaining for multicast traffic. Xu et al. [59] proposed an algorithm to search for the best Steiner tree that contains the required number of services. This work assumed the services are deployed at all ISP locations, which is not practical in many scenarios. Kuo et al. [35] 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).

A recent work [46] 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 work in certain situations where the number of multicast sessions is small. However, as the number of sessions increases, MSA may not allocate many sessions, or it may congest the network. In contrast to MSA, Oktopus is general as it calculates multicast distribution graphs for large numbers of sessions without making assumptions about the ISP resources.

2.2.2 Service Deployment

The problem of service deployment focuses on optimizing service placement with different objectives. In [29, 48], the authors considered the problem of jointly optimizing the number and location of deployed services to minimize their deployment cost and maximize their usage. The authors in [42] considered the problem of the service placement and request scheduling to maximize the utilization of services and minimizes the response latency of the request. In [36], the authors considered the joint problem of service placement and path selection. Here, we assume that the services are already deployed at locations within the ISP, and we do not address the deployment problem.

2.2.3 Traffic Engineering (TE)

Multicast TE refers to the process of computing links of multicast trees that achieve a specific objective, e.g., minimizing the maximum link usage. The current de-facto standard, mLDP [54], builds a shortest-path multicast tree from the source to all destinations in the multicast session. Thus, it may not be able to achieve the required TE objectives. Prior works, e.g., [12, 28], addressed the multicast TE problem while optimizing the usage of link and forwarding table capacities. MTRSA [28] is a heuristic algorithm that runs when the forwarding table of a router is overloaded. This algorithm reroutes the paths that lead to the overloaded router to other paths that reduce the total routing cost. OBST [12] addressed the online multicast TE problem while considering the rerouting cost of multicast trees. Unlike these works, Oktopus efficiently calculates the (complex) multicast distribution graphs while considering the available link, processing, and forwarding table capacities.

2.2.4 Control-plane Frameworks

There are several control-plane frameworks that provide a high-level programming interface for the network operator. For instance, frameworks such as ONOS Intents [4], Boulder [2], and NIC [6] provide a simplified way to create and manage forwarding rules at the controller.

These frameworks, however, do not calculate the optimized network paths and assume that these paths are given to them.

Moreover, frameworks such as SOL [27], Chopin [26] and DEFO [25] compute optimized unicast paths to satisfy a specific routing objective. SOL [27] and Chopin [26] pre-generate network paths and pass them to an existing optimization solver (e.g., CPLEX) to calculate the optimized traffic flows. DEFO [25] calculates optimized paths by searching for a sequence of middle-points in the network instead of engineering every link in the path. However, DEFO cannot efficiently support service chaining as middle-points have limited control over the chosen links. In contrast to these works, Oktopus supports multicast distribution graphs while engineering every link in the calculated graphs.

Chapter 3

System Model and Problem Definition

In this chapter, we state the system model in Section 3.1. In Section 3.2, we provide the problem definition and hardness.

3.1 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. Another common use-case for multicast services happens when large-scale content providers, such as Facebook and Twitch, partner with ISPs to deliver live streams to millions of users. Our abstract system model supports these and other use cases of 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 [34]) 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., point of a presence (PoP). We refer to ISP locations as nodes for simplicity. Specifically, 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, the node n has servers that host a set of services. The ISP

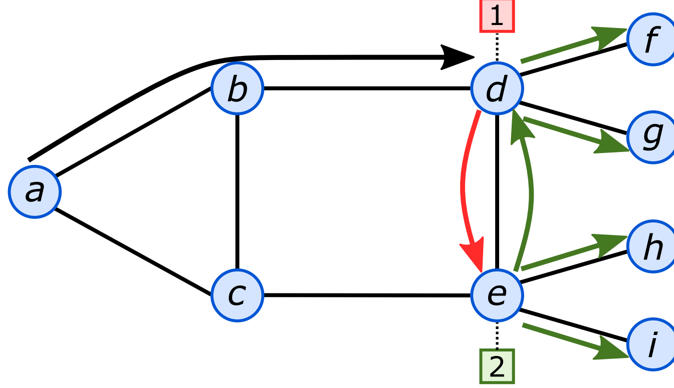


Figure 3.1: A multicast session with a source node a, destination nodes f, g, h and i, and required services $1 \rightarrow 2$. Arrows are links of the multicast distribution graph.

allocates processing resources to process $p_{v,n}$ bits/sec for a 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 $\langle src, dsts, \mathbb{V}, \text{b} \rangle$, where src and $dsts$ are its source and destinations, $\mathbb{V}$ is the sequence of required services, and b is the session bandwidth demand bits/sec, respectively. 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 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 3.1 shows an example of an input multicast session to Oktopus defined by $\langle a, \{f, g, h, i\}, \{1 \rightarrow 2\} \rangle$. Services 1 and 2 are deployed at nodes d and e, respectively. Arrows in the figure represent the multicast distribution graph calculated by the proposed algorithm.

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 the 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 $b \times \sum_{l \in \mathbb{L}} \gamma_l$. In addition, the total routing cost of $\mathbb{S}$ is $\sum_{s \in \mathbb{S}} b_s \sum_{l \in \mathbb{L}_s} \gamma_l$, where $\mathbb{S}$ is the set of requested sessions.

The ISP controller performs the following three operations to calculate the multicast distribution graphs. First, it periodically receives requests from ISP customers to allocate resources to their multicast sessions. Second, for each incoming multicast session request, the ISP controller runs the Oktopus algorithm to calculate a multicast distribution graph $\mathcal{D}$. Finally, upon the calculation of $\mathcal{D}$, the controller maps the graph to match-action rules and send them to corresponding routers. Note that the second step is a continuous online process. The Oktopus algorithm is triggered as soon a multicast session is requested instead of triggering once for multiple multicast sessions. The focus of this thesis is on the design and optimization of the second step, while the design of multicast applications and match-action rules mapping are beyond the scope of this thesis.

3.2 Problem Definition and Hardness

The problem addressed in this thesis 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 (in the system-level) at each ISP location to the session as well as engineering edges of the distribution graph (in the network-level) to pass through required services while (i) minimizing the routing cost of the session without exceeding the available processing resources at nodes, link capacities, and forwarding table sizes at core routers, and (ii) maximizing the number of allocated multicast sessions in the ISP network.

The considered problem introduces many hard constraints and requirements that make solving the problem very challenging. We show the hardness of the considered problem in Theorem 1. Previous works, e.g., [8], 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 instance, the Steiner Tree problem is known to be NP-Hard [14], but it can be solved with an approximation factor of 1.386 [10], meaning that the total cost of the Steiner tree can be found within 1.386 times of the optimal cost Steiner tree. The multicast problem with link and node constraints referred to as Scalable Multicast Traffic Engineering (SMTE) problem is proven to be NP-Hard to solve and NP-Hard to approximate within a factor $\alpha(n)$ of the optimum solution [28], where α is any polynomial-time computable function, and n is the number of nodes in the network. Since SMTE is a subset of the considered problem without the service chaining requirement, the considered problem is at least as hard as SMTE to solve and approximate.

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

Proof. We show a polynomial-time reduction from the Steiner tree problem to a special instance of the considered problem, which happens when the multicast session does not require any services. Specifically, given a graph $(\mathbb{N}', \mathbb{E}')$, a source a', a set of destinations $\mathbb{D}'$, the Steiner tree problem is to calculate a minimum-cost tree in $(\mathbb{N}', \mathbb{E}')$ that connects the destinations $\mathbb{D}'$. The reduction algorithm takes as input an instance $(\mathbb{N}', \mathbb{E}', a', \mathbb{D}')$ of the Steiner tree problem. It outputs an instance $(\mathbb{N}, \mathbb{E}, \mathbb{S})$ of the considered problem, where $\mathbb{N} = \mathbb{N}'$, $\mathbb{E} = \mathbb{E}'$, and $\mathbb{S} = \{\langle a', \mathbb{D}', \phi \rangle\}$. Since the Steiner tree problem is NP-hard, thus the considered is NP-hard. □

In summary, because of its huge search space, the considered multicast service chaining problem 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 thesis, we propose a principled, heuristic, approach to efficiently find near-optimal solutions for the multicast service chaining problem.

Chapter 4

Proposed Oktopus Algorithm

This chapter discusses the proposed algorithm (Oktopus). First, we establish the terms used by Oktopus in Section 4.1. Then, we provide an overview of Oktopus in Section 4.2. Afterward, we discuss in detail the two main steps of the algorithm in Section 4.3 and Section 4.4. An illustrative example and the analysis of Oktopus are provided in Section 4.5 and Section 4.6, respectively.

4.1 Definitions

To enable control over the selection of graph links, the proposed algorithm uses a *path* as the building block for calculating the distribution graph. We define two terms that we use extensively when describing the proposed algorithm: *segment* and *service-chained path*.

A network *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. Moreover, a segment does not fulfill service chaining requirements. For example, the sequence of nodes $\{a, b, d, g\}$ in Figure 3.1 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 *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 $\{\langle a, \phi\rangle, \langle b, \phi\rangle, \langle d, \{1\}\rangle\ \langle e, \{2\}\rangle, \langle d, \phi\rangle, \langle f, \phi\rangle\}$ form a valid path to reach f to satisfy the services $\{1, 2\}$, where ϕ means this node does not provide any services to the session.

4.2 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 $\mathcal{D}$ to satisfy the service chaining requirements.

We focus on designing an algorithm that can produce the distribution graph efficiently and can be implemented robustly. At the same time, it balances the network resource utilization and routing cost optimization. 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 populates the segment store with the generated network segments and the second step uses the segments to calculate the multicast distribution graph.

The GenerateSegments function, described in Section 4.3, 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 observations from TE 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 Section 4.4. 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 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 and it should forward them on different interfaces. Thus, the proposed algorithm should produce graphs that satisfy the service chaining requirements while not resulting in forwarding ambiguity at routers. To calculate efficient a graph without forwarding ambiguity, we propose a lightweight data structure to maintain and track information about each path in the calculated graph.

4.3 Segment Generation

The GenerateSegments algorithm computes segments between each pair of nodes in the ISP network. The algorithm takes as inputs the ISP network topology $(\mathbb{N}, \mathbb{E})$, the maximum number of segments between every node pair K, and the ratio between the maximum segment length and network diameter, referred to as ω. 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

Algorithm 1 Generate the set of network segments, and stores them in segment store.

Input: $\mathbb{N}$: ISP locations (or nodes)
Input: $\mathbb{E}$: ISP network links
Input: $\mathcal{S}$: Segment store

```
1: function GenerateSegments(ℕ, 𝔼, 𝒮)
2:     //Iterate for each node pair
3:     for n ∈ ℕ do
4:         for m ∈ ℕ do
5:             if [n, m] ∉ 𝒮 then
6:                 //Generate Segment for node pair
7:                 𝒮[n, m].add(BreadthFirst(K, ω, n, m,))
8:                 𝒮[n, m].add(EdmondsKarp(n, m, 𝒮))
```

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 GenerateSegments algorithm shown in Algorithm 1, has two nested loops indexed by n and m, respectively, each of which traverses the ISP nodes $\mathbb{N}$. The algorithm creates a new entry $\mathcal{S}[n, m]$ if such entry does not exist. The algorithm then generates segments and inserts them to $\mathcal{S}[n, m]$ as follows. For each node pair n and m, the algorithm traverses the ISP network between the node pair using breadth-first, and computes the first K loop-free segments whose lengths do not exceed the value $\omega \times D$, where D is the network diameter. The algorithm then calculates a set of edge-disjoint segments between the node pairs n and m, and adds them to $\mathcal{S}[n, m]$. At a high level, this is done by calculating the maximum flow between the node pair using existing algorithms (e.g., Edmonds–Karp algorithm [18, 14]) to calculate 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., [25, 26]. 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.

4.4 Calculating the Multicast Distribution Graph

For the 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 weighted segment which consists of a sequence of nodes $\mathbb{N}_p$ and a weight value w_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 n 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. Furthermore, 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 At high level, the CalculateGraph algorithm calculates the distribution graph by merging existing paths, and generates new segments when it detects that the network would be overloaded.

The CalculateGraph algorithm, shown in Algorithm 2, 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 CalculateGraph algorithm iterates over the destinations and invokes the FindPaths function for every destination, which we will describe in details later.

The CalculateGraph 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 CalculateGraph algorithm calculates a set of minimum-cost paths that satisfy the service chaining requirements $\mathbb{V}$ as follows. To pick a destination (Lines 4–6), 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). This sorting ensures that the algorithm does not create unnecessary or longer paths that may lead to higher routing costs.

The algorithm then iterates over all nodes in the calculated paths so far. Each traversed node n 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 11), and calculates the set of paths by calling the FindPaths function (Line 12), 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 (Lines 18–19).

Algorithm 2 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 = {}
4:      dsts = SortDestinations(s)
5:      while len(dsts) > 0 do
6:          Pop a destination dst from dsts
7:          paths = null; cost = ∞;
8:          // Search from previous solution to dst
9:          for p ∈ 𝒟.get_paths() do
10:             for n ∈ ℕ_p do
11:                 srv = {𝕍_s \ out(o_n, n, s).processed_services}
12:                 sol =FindPaths(ℕ, 𝒮, s, p, n, dst, srv)
13:                 if sol.cost < cost then
14:                     paths = sol.paths
15:                     cost = sol.cost

16:         // Paths were found to reach dst
17:         if paths != null then
18:             update_resources(ℕ, 𝔼, paths)
19:             𝒟.update_graph(paths)
20:         else
21:             dst.traversal_count += 1
22:             Push dst to the end of dsts
23:         // Trigger dynamic segment generation
24:         if dst.traversal_count == 2 then
25:             Generate new segments and add them to 𝒮
26:         else if dst.traversal_count > 2 then
27:             Pop dst from dsts
28:             Push dst to unallocated
29:     return 𝒟
```

The proposed CalculateGraph 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 at the end of the loop (Line 22). 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 23–25). 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 3, 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 or a destination of the session, respectively. The calculated paths and their costs are maintained in a solution object referred to as *sol*.

The main idea of the algorithm is to break down a segment in $\mathcal{S}$ from *src* to *dst* into smaller paths when *srv* cannot be satisfied directly using that segments. Each path is calculated to minimize the routing cost while satisfying *srv*. This path breakdown ensures that the algorithm uses 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 3, respectively.

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 (Line 7) as:

$$W_1 \sum_{l} link_weight(l, s) + W_2 \sum_{v} \sum_{n} node_weight(v, n, s),$$

where $0 \leq w \leq 1$, W_1 and W_2 are normalization factors, l and n are links and nodes belonging to the path, 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:

$$link_weight(l, s) = \begin{cases} \gamma_l \alpha_{l,s}^{1+f_l}, & \alpha_{l,s} > 0.5 \\ \gamma_l \alpha_{l,s}, & \text{otherwise}, \end{cases} \tag{4.1}$$

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 (4.1) grows exponentially as the link usage increases and exceeds half

Algorithm 3 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

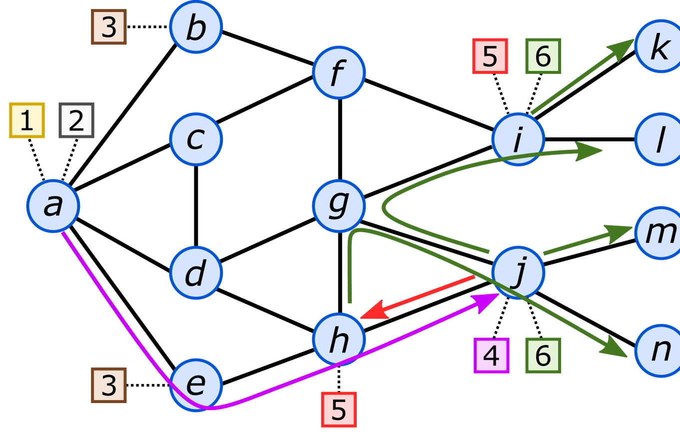
```
1:  function FindPaths(ℕ, 𝒮, s, p, src, dst, srv)
2:      sol = new_solution(); cand_paths = {}
3:      [A] Find a path from src to dst that satisfies srv
4:      for seg ∈ 𝒮[src, dst] do
5:          // Locate services and calculate weights
6:          srv_map = ServicesAlongSegment(s, seg, srv)
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(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 = ℕ.get_nodes_by_srv(v)
24:     for m ∈ nodes do
25:         tmp_sol = FindPaths(ℕ, 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(ℕ, 𝒮, s, p, src, dst, srv)
34:     sol.sc_paths.add(next_sol.sc_paths)
35:     sol.cost += next_sol.cost
36:     return sol
```

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 CPU resources for the service v at n as $\Big(b_s + u_{v,n}\Big)/p_{v,n}$.

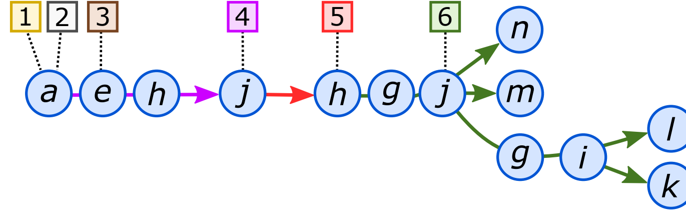
The algorithm then creates a candidate path scp given the calculated satisfied services and path weight (Line 9). The algorithm sets the scp parent to p and updates the packet classes for the incoming interface i_n for every node $n \in \mathbb{N}_{scp}$ as $inc(i_n, n, s) = out(o_{prev(n)}, prev(n), s)$, 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 validates if the candidate path scp would introduce forwarding ambiguity at routers. This happens 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 breaks 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 provides 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. Specifically, 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.



(a) Illustrative process of the proposed Oktopus algorithm.



(b) The output multicast distribution graph.

Figure 4.1: Illustrative example of a multicast session.

4.5 Illustrative Example

We describe an illustrative example of the proposed algorithm. Figure 4.1a 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). Note that not all services are deployed at all ISP nodes, and a single node may host multiple services.

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) → Firewall (2) → Encoder (3) → Content Manager (4) → Ad Insertion (5) → Transcoder (6). For simplicity, we assign to each service an ID from 1 to 6.

We 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 to node n that can satisfy the service chaining requirements. Thus, 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 3. 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 4.1a.

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 3). Notice that although service 3 is deployed at b, our algorithm does not include it in the sub-path because it results in 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 3). In this example, both nodes h and i provide this service. So, the algorithm sets the source to j and recursively calls itself twice, each of which with a different destination (Lines 24–29 in Algorithm 3). 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 this path results in lower routing cost from j.

Finally, the algorithm 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 3). 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 routing cost. Notice that the algorithm does not use the link (h, j) more than once as it only has a capacity of one unit of bandwidth. Prior algorithms, e.g., [46], do not have control over the selection of individual links, and thus, they may overload some network links. Moreover, although nodes h and j appear in the three calculated sub-paths, Oktopus ensures that the type of ingress traffic to each node is unique. Hence, core routers are able to process ingress traffic without ambiguity.

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 4.1a, node j is used to calculate paths to destinations m and l, and node i is used to reach destination k. Figure 4.1b depicts the final output multicast distribution graph.

4.6 Analysis of Oktopus

The following lemma shows the time and space complexities of the proposed algorithm.

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. The proposed algorithm consists of two steps: segment generation and graph calculation. The time complexity of the GenerateSegments algorithm is $\mathcal{O}(N^3E^2)$. Specifically, for each node pair, the algorithm generates K segments using breadth-first in $\mathcal{O}(K(N+E))$ and a set of edge-disjoint segments using Edmonds–Karp algorithm in $\mathcal{O}(NE^2)$, where K

is the number of segments per node pair generated. Since there are N^2 node pairs, the time complexity is $\mathcal{O}(N^2(K(N+E)+NE^2))$. However, K is a constant value smaller than N and E. Thus, the GenerateSegments algorithm runs in $\mathcal{O}(N^3E^2)$.

The time complexity of the CalculateGraph algorithm to calculate the distribution graph is $\mathcal{O}(N^4E^2)$. Specifically, the algorithm iterates over the destinations of the multicast session, where the maximum number of destinations is $\mathcal{O}(N)$. For each destination, it performs two operations. First, it searches for the best node to connect a new path to by calling the FindPaths. Second, it calls the GenerateSegments algorithm when needed. So, the running time of the CalculateGraph is $\mathcal{O}(N(NF+N^3E^2))$, where $\mathcal{O}(F)$ is the time complexity of the FindPaths algorithm that we calculate as follows.

The FindPaths algorithm runs in $\mathcal{O}(KVN^2)$, where K is the number of generated segments per every node pair, and V is the maximum length of a service chain. Specifically, it iterates over K generated segments, and for each segment, it calculates the available services in that segments and its link weights. Since the maximum length of a segment is $\mathcal{O}(N)$, the running time of this step is $\mathcal{O}(KN)$. This happens $\mathcal{O}(VN)$ time in the worst-case scenario, when the generated segments cannot support the required services without breaking down these paths. Specifically, the algorithm recursively calls itself $\mathcal{O}(N)$ times up to $\mathcal{O}(V)$ times (Lines 24–29 in Algorithm 3). This, however, does not result in exponential running time for two reasons. First, each recursive call does not result in additional recursive calls as the algorithm chooses these nodes because they provide the required service. Second, the algorithm chooses only one node to search for the remaining services (Line 33 in Algorithm 3).

Putting it together, the time complexity of the CalculateGraph algorithm is $\mathcal{O}(N(KVN^3+N^3E^2)) = \mathcal{O}(N^4(KV+E^2))$. Since K and V are small values in the range of 4 to 20 (i.e., $KV \ll E^2$), the running time of the CalculateGraph algorithm is $\mathcal{O}(N^4E^2)$.

The space complexity of the proposed algorithm is $\mathcal{O}(N^3E)$. The GenerateSegments algorithm takes $\mathcal{O}(N^3(K+E))$ space because for every node pair, it generates K segments using breadth-first traversal and $\mathcal{O}(E)$ edge-disjoint segments, each of size $\mathcal{O}(N)$. Moreover, the CalculateGraph uses $\mathcal{O}(N)$ space for the session. Therefore, the total space complexity of the proposed algorithm is $\mathcal{O}(N^3E)$. □

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 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 [25, 51, 26, 27]. And most ISP networks are sparse with the number of links E ranges from 500 to around 2,000. Our experiments in Chapter 6 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. Second, 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 3 can run in parallel.

Finally and most importantly, the time complexity analysis is for the *worst-case* scenario, which assumes that the dynamic segment generation and recursive composition of services happen for every destination per multicast session, which is very unlikely for most practical situations. We note that these two steps are optional in our algorithm and they are used 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. Disabling the dynamic segment generation reduces the time complexity to $\mathcal{O}(N^3)$ and further disabling the recursive composition of services brings the worst-case time complexity to $\mathcal{O}(N^2)$.

Chapter 5

Implementation and API

In this chapter, we discuss the implementation of our control-plane framework and list the API provided by the framework.

5.1 Implementation

We implemented a full control-plane framework in Python, which we release its source code for the community [16]. The framework has two main components: Oktopus APIs and an optimization engine. The Figure 5.1 shows the components of the control-plane framework.

The Oktopus APIs provide a set of high-level interfaces to developers and network operators to define their applications. Specifically, the Application APIs define the ISP topology and available resources, the Session APIs determine the multicast session parameters including their service chaining requirements, the Routing APIs set the routing objectives (e.g., min. routing cost) and link constraints. We describe the details in section 5.2.

The optimization engine receives these inputs from the APIs, and runs the Oktopus algorithm to compute the multicast distribution graphs.

5.2 Oktopus API

The Oktopus API enables operators to describe *what* goals the network application needs to achieve instead of *how* these goals are realized. For example, these APIs do not consider multicast distribution graphs because, from the operator perspective, multicast distribution graphs are the mechanisms to carry traffic. Listing 5.1 shows an example of a typical usage of the Oktopus APIs.

In the network application, the operator specifies per-session requirements and network-wide constraints and objectives by utilizing three sets of APIs: Application, Session, and Routing.

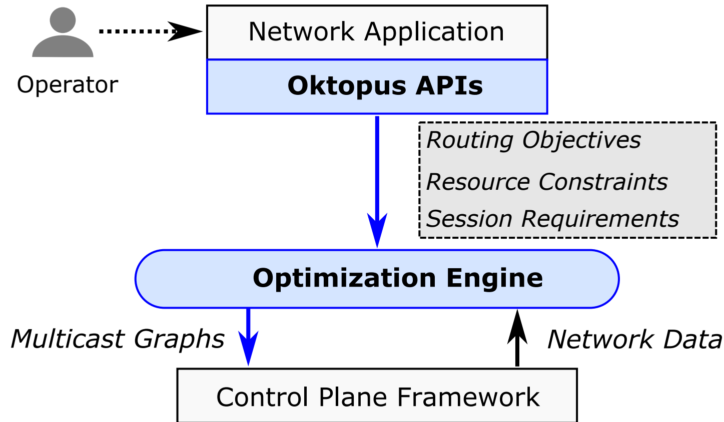


Figure 5.1: Operators use the Oktopus API to express the network application. Then, the optimization engine runs the Oktopus algorithm to compute the multicast distribution graph.

```
app = App(topo_path)
# Session APIs: session requirements
s = Session(addr, src, dsts, bw)
s.traverse(['fw', 'ids']) # service chaining
# Routing APIs: routing obj. and link constraints
r = Routing()
for link in app.getLinks():
    r.addLinkCapConstraint(link, 'bps', 1e9)
r.addObjective('minRoutingCost')
# App APIs: available resources and services
app.addSessions(s); app.setRoutes(r)
app.setAlgorithm('oktopus') # or other algorithms
app.solve() # produces a solution
```

Listing 5.1: An example of using Oktopus APIs.

5.2.1 Application API

This API allows the operator to control different aspects of an application such as topology, services, session, routing, and solution.

The operator can manipulate the topology using `Link`, `Node` and `Service` objects. The operator adds, deletes and gets a link or node using `add`*, `del`* and `get`* functions, respectively. In addition, the operator gets access to services at every node, and sets the available resource using `setResource` function. A list of services in the network can be retrieved by type using `getServices(type)` function (e.g., `app.getServices('fw')`). Similar to links, nodes and services, sessions can be added and deleted as well. In addition,

API	Semantics

`addr`, `src`, `dsts`
`bw`
`res`
	address, source and destinations
	bandwidth demands
	required resource map

`addConstraint`(*name*, *val*)	when routing, the max. *name* must not exceed *val* in the resulting tree. *name*='delay'
`avoid`(*nodes*) `avoid`(*links*) `avoid`(*sessions*)	the session should avoid a set of *nodes*, *links* or other *sessions*
`traverse`(*services*)	the session should pass through a set of *services*. *services* is ordered to preserve chaining requirements

Table 5.1: The proposed Session interface.

Oktopus allows the operator to retrieve a group of sessions by their source. For example, `app.getSessionsBy('src', 'a')` will return all sessions with source node of a. The operator uses Session APIs in Table 5.1 to manipulate sessions.

To define routing costs, constraints and objectives, the operator creates and sets a routing object using `setRoutes` function. Table 5.2 shows the available Routing APIs in Oktopus. Since the framework is algorithm-agnostic, using `setAlgorithm` allows users to set the engine algorithm of the application. Finally, the framework produces the application solution when the operator invokes `solve()`. Pushing OpenFlow rules is independent of the Oktopus APIs, allowing the operator to use multiple controller technologies.

Listing 5.2 shows how the Application APIs are used. This application loads the network topology from a file, modifies link capacity and delay between nodes 1 and 2, sets CPU capacity of a firewall at node 3, adds a new session, sets a routing object, sets the engine algorithm to use Oktopus algorithm and then calculates the solution.

```
app = App(topo_path)
app.getLink(1,2).cap = Gb(1)
app.getLink(1,2).delay = msec(10)
fw = app.getNode(3).getService('fw')
fw.setResource('cpu', 50) # set service fw to 50 cpu capacity
app.addSessions(Session(addr, src, dsts, bw)) # add session
app.setAlgorithm('oktopus')
app.solve() # produces a solution
```

Listing 5.2: General structure of Oktopus applications.

Group	API	Semantics
Cost and Usage	addLinkCostFn($link, name, linkCostFn$)	Calls $linkCostFn$ to calculate the link cost when a session tree passes $link$
	addLinkUsageFn($link, name, linkUsageFn$)	Calculates how link resources are consumed using $linkUsageFn$
	linkCostFn($link, session$) linkUsageFn($link, session$)	Definitions of link cost and usage functions. They both take link and session objects. The former returns the cost, and the latter modify the link object.
	addNodeCostFn($node, srv, name, nodeCostFn$) addNodeUsageFn($node, srv, name, nodeUsageFn$) nodeCostFn($service, session$), nodeUsageFn($service, session$)	Similar to link functions. The main difference is that node functions expect a service object since a node may have multiple services.
Constraints	addLinkCapConstraint($link, name, val$)	For a given $link$, limit the max. value of $name$ to val. Currently, $name$='bps' ($link$.bps $\leq val$)
	addNodeCapConstraint($node, srv, name, val$)	For a given $node$ and srv, limit the max. value of $name$ to val ($node$.getService(srv).getResource($name$) $\leq val$)
Objectives	addObjective($name$)	Adds an objective to be optimized by Oktopus. Currently, Oktopus supports the following: minMaxLinkLoad, minRoutingCost

Table 5.2: The proposed Routing interface.

5.2.2 Session API

This API, shown in Table 5.1, creates sessions and adds constraints and requirements. To create a session, the operator inputs its IP address, source, destinations and bandwidth demands. If service chaining is required for the session, the operator has to create a resource requirement map. This map has two keys: service name and resource name, and its value is the required resource value. For example, Listing 5.3 depicts a session with two required services: firewall (`fw`) and video transcoder (`vt`) with required CPU of 10 and 20 units per second, respectively. If the operator defines service chaining requirement without a resource map, it assumes that the session does not consume service resources, and it will satisfy the order only.

```
res_dict = {'fw':{'cpu':10}, 'vt':{'cpu':20}} # define resource consumption
s = Session(addr, src, dsts, bw, res=res_dict)
# define session requirements
s.addConstraint('delay', msec(30)) # s.delay <= 30 msec
s.avoid(n1)
s.traverse(['fw', 'vt']) # service chaining
```

Listing 5.3: An example of using the Session API.

Oktopus API provides additional three APIs to define constraints and requirements. The function `addConstraint(name, val)` limits the maximum `name` value of the tree to `val`. Currently, Oktopus API supports limiting the `delay` per session, which is useful to control per-session routing constraints. For example, the network operator may choose to limit the maximum allowed delay to 30 msec for the session in Listing 5.3. For per-session

traffic isolation and steering and service chaining, Oktopus API exposes the functions `avoid` and `traverse`, respectively. In Listing 5.3, the session should avoid node `n1` and traverse two services: firewall and video transcoder.

5.2.3 Routing API

The operator uses this API by creating and modifying a routing object. Unlike the Session API, the Routing API (shown in Table 5.2) defines node and link constraints and network-wide objectives. Moreover, it allows the operator to change how Oktopus calculates costs and consumes network resources by implementing cost and usage functions.

Listing 5.4 illustrates a typical usage for the Routing API. First, the operator defines a custom functions: `cpuCostFn` and `cpuUsageFn` to modify the cost and consumption behavior of CPU resources at all services. Note that Oktopus algorithm allows different functions for different services, nodes and links. Then, the application adds a constraint for each link to limit the maximum bandwidth to 10^9 bit per second (bps). It also adds a constraint for every firewall and video transcoder to limit their CPU load by 90% and 70%, respectively. The network objective in this example is to minimize the routing cost.

5.2.4 Developer Benefits

Using the Oktopus API, the operator uses 71% fewer lines of code (LOC) [15] to write the service chaining multicast application compare to the specialized algorithm, MSA [46]. Although LOC does not directly show the developer benefit, we believe that using an intent-based API such as Oktopus API is far simpler than writing the complex formulation.

```python
def cpuCostFn(srv, session):
    session_cpu = session.res[srv.name]['cpu']
    used_cpu = srv.getUsedResource('cpu')
    cpu_cap = srv.getResource('cpu')
    return (session_cpu+used_cpu)/cpu_cap

def cpuUsageFn(srv, session):
    srv.addUsedResource('cpu', session.res[srv.name]['cpu'])

r = Routing()
for link in app.getLinks():
    # limit link to 1e9 bps
    r.addLinkCapConstraint(link, 'bps', 1e9)
    # custom cost funtion
    r.addLinkCostFn(link, 'bps', costFn=lambda l,s:(s.bw+l.used_cap)/l.cap)
    # custom usage funtion
    r.addLinkUsageFn(link, 'bps', usageFn=lambda l,s: l.used_cap += s.bw)

for node in app.getNodes():
    # limit the cpu of fw on node to 0.9
    r.addNodeCapConstraint(node, 'fw', 'cpu', 0.9)
    r.addNodeCapConstraint(node, 'vt', 'cpu', 0.7)
    for srv in node.getServices():
        # custom cpu cost funtion of srv on node
        r.addNodeCostFn(node, srv, 'cpu', costFn=cpuCostFn)
        # custom cpu usage funtion of srv on node
        r.addNodeUsageFn(node, srv, 'cpu', usageFn=cpuUsageFn)

r.addObjective('minRoutingCost')
app.setRoutes(r)
```

Listing 5.4: An example of using the Routing API.

Chapter 6

Evaluation

In this chapter, we compare Oktopus versus the optimal and closest algorithms in simulations using real ISP topologies.

6.1 Setup

Algorithms Compared Against. We implemented a Python-based simulator to evaluate Oktopus's performance across various scenarios and compare Oktopus against prior work. The simulator creates network applications with the control-plane framework describe in Chapter 5. The optimization engine receives input from the Oktopus APIs and runs an algorithm to compute the multicast distribution graphs. In the current version of our framework, we implemented and integrated the Oktopus algorithm, optimal (OPT), and MSA [46] algorithms in the optimization engine. We implemented MSA because it is the closest algorithm in the literature solving the considered multicast service chaining problem. We implemented OPT using CPLEX 12.8 and using the problem constraints from [46].
Performance Metrics. We consider four important performance metrics that measure the quality of the allocated multicast sessions:

- **Percentage of allocated multicast sessions:** The ratio between the allocated and total numbers of multicast sessions. The larger the percentage the better the fulfillment of the ISP customer requirements.

- **Average routing cost:** The cost of forwarding packets of a multicast session on links of its distribution graph, and it is defined by $b \sum_{l \in \mathbb{L}} \gamma_l$.

- **Average graph size:** The average number of links of the calculated distribution graph per session. A small graph indicates lower delays to reach all destinations.

- **Average running time:** The average elapsed time to calculate a distribution graph per session.

Control Parameters. We use the following five real ISP topologies from the Internet Topology Zoo [33]: 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 this recent study [17] to generate the service chain requirements.

We control five parameters for every experiment as follows:

- **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.

- **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 [7].

- **Service chain length**: We vary the length of the service chain from 3 to 6.

- **Service deployment**: Since different services have different usage patterns [17], we categorize services into two types, essential and auxiliary type. Essential services are popular ones such as firewalls and IDS, and they are deployed to all ISP nodes. Auxiliary services, on the other hand, include services that are least 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%. The service function capacity per node is set equal to the number of sessions in the multicast application.

- **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. More specifically, the first and second services of the service chain are randomly chosen from the essential services, and the rest of the services are chosen from auxiliary services. In random ordering, services in the service chain are randomly ordered.

6.2 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. As we will show, running

ISP	Routing Cost	Avg. Graph Size		Avg. Running Time	
	Opt. Gap	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 6.1: Results of Oktopus versus OPT for 10 sessions.

OPT for numbers of sessions larger than 10 results in out-of-memory exceptions and/or takes too long to terminate.

Results for 10 Sessions. We summarize the results in Table 6.1. We note that both algorithms allocated all the multicast sessions, so we focus on the other performance metrics. First, the optimality gap of Oktopus ranges from 1.75% to 5.9% for the considered topologies except for Columbus topology. The optimality gap is higher for the Columbus topology, with a gap of 8.5% due to the complex topology structure. Overall the optimality gap is small, considering that the optimality gap is measured between the total routing cost to allocate all the sessions. Second, we compare the average graph size to study the quality of the distribution graph. Oktopus calculates distribution graphs with similar sizes to the OPT counterpart. For instance, for the Colt topology, Oktopus calculates graphs with only an additional five links per session on average. The size of the multicast distribution graph for other topologies shows a similar result of around 5.5% larger than the ones produced by the optimal algorithm and with similar standard deviation. Finally, Oktopus computes the graphs 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 only 3.5 s 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 only.

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 only while using less than 1 GB of memory.

6.3 Oktopus versus the Closest Algorithm

Recall that the closest algorithm in the literature, called MSA, assumes infinite link capacities and thus it cannot allocate large number of sessions. We show in the following that Oktopus outperforms MSA across all performance metrics.

Percentage of Allocated Sessions. We first show in Figure 6.1 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 6.1a 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 6.1b 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 deployed to more nodes. This is because Oktopus considers the CPU utilization across the available services.

Figure 6.1c 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. This means that Oktopus allocated 808 more multicast sessions using the same network resources.

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 6.1d 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.

We can also see similar results from other ISP topologies across different scenarios in Figures 6.2–6.5. For example, Figure 6.3 shows the percentage of allocated multicast sessions in a smaller ISP topology (Dfn, 57 nodes). Oktopus increases the percentage of allocated multicast sessions up to 28%, 42%, 37%, and 24% for different service chain lengths, deployed auxiliary services percentages, receiver densities, and service chain orderings, respectively.

Next, we present the percentage of allocated multicast sessions for ISP topologies with different sizes in Figure 6.6. 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%, 28% and 20% for the three topologies, respectively.

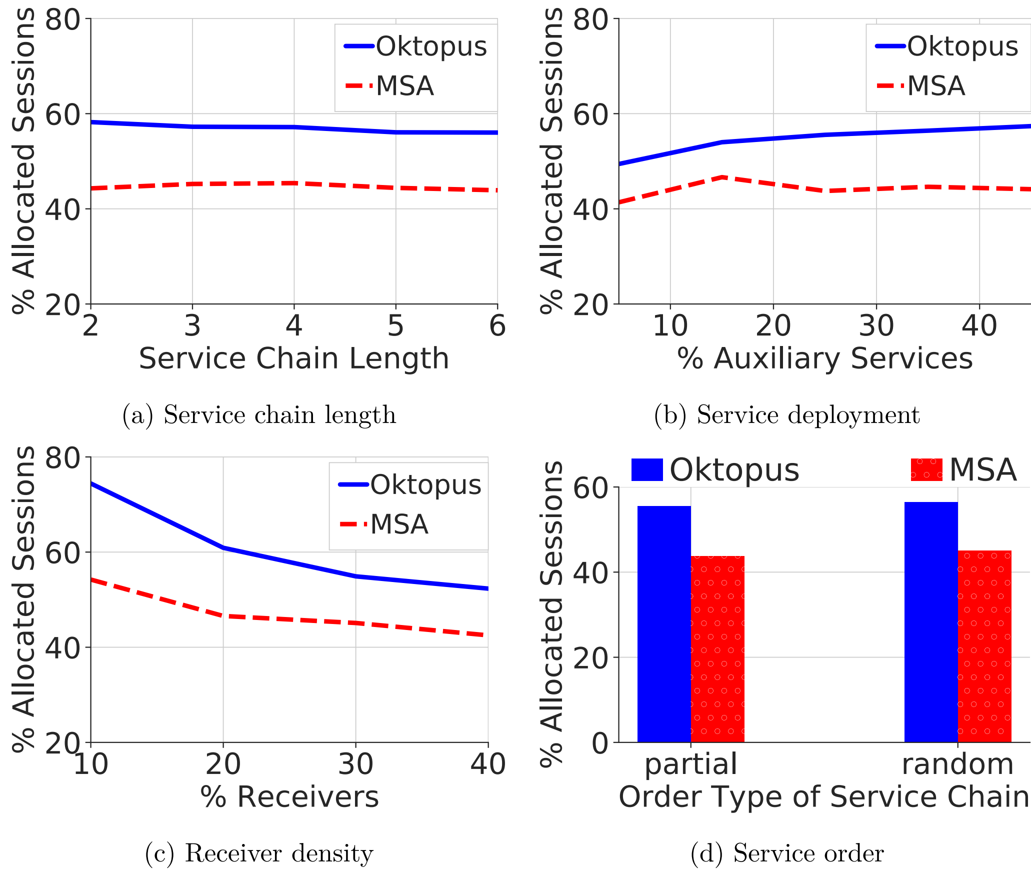


Figure 6.1: Percentage of allocated multicast sessions for the Ion ISP topology across different scenarios. # multicast sessions: 4,000.

Figure 6.2: Percentage of allocated multicast sessions for the AttMpls ISP topology across different scenarios. # multicast sessions: 4,000.

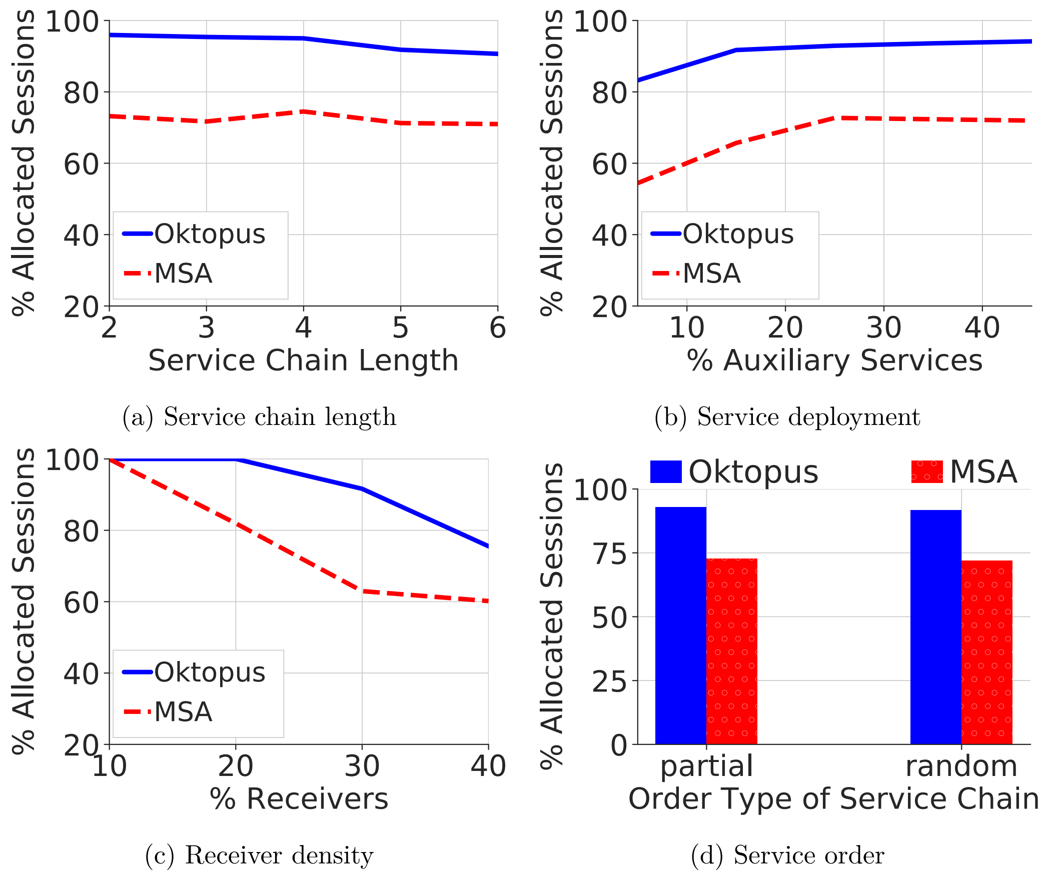


Figure 6.3: Percentage of allocated multicast sessions for the Dfn ISP topology across different scenarios. # multicast sessions: 4,000.

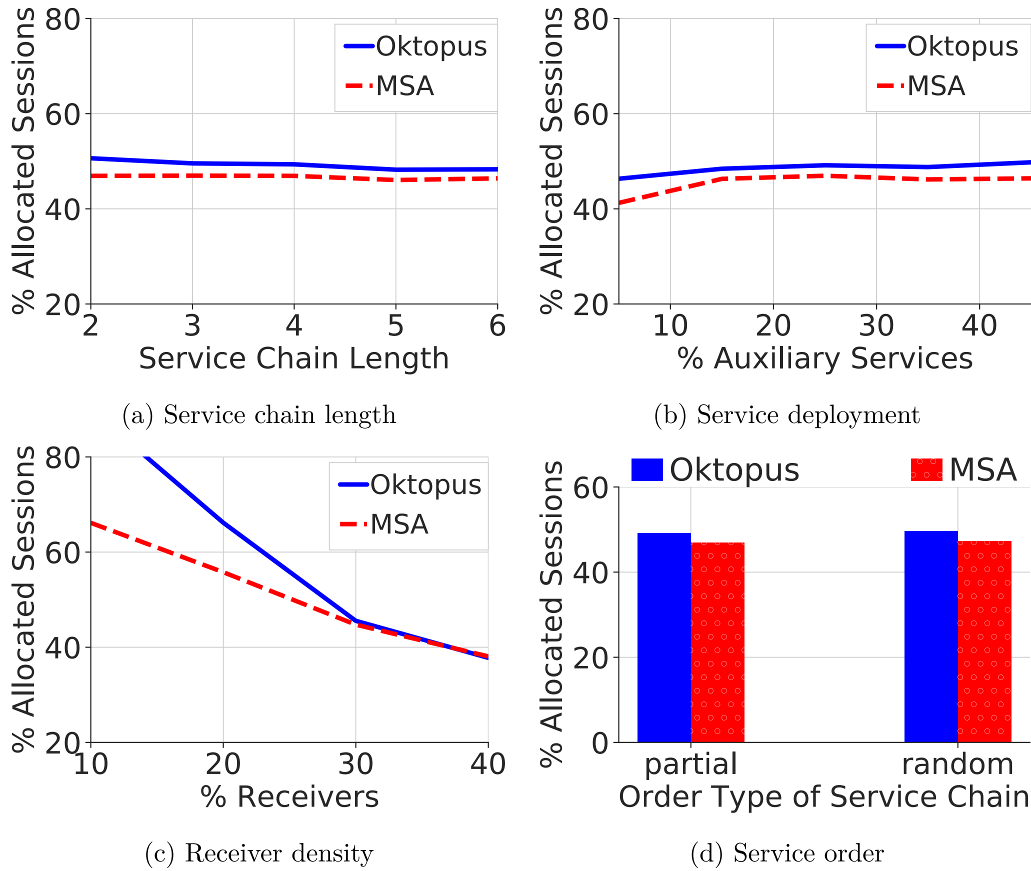


Figure 6.4: Percentage of allocated multicast sessions for the Columbus ISP topology across different scenarios. # multicast sessions: 4,000.

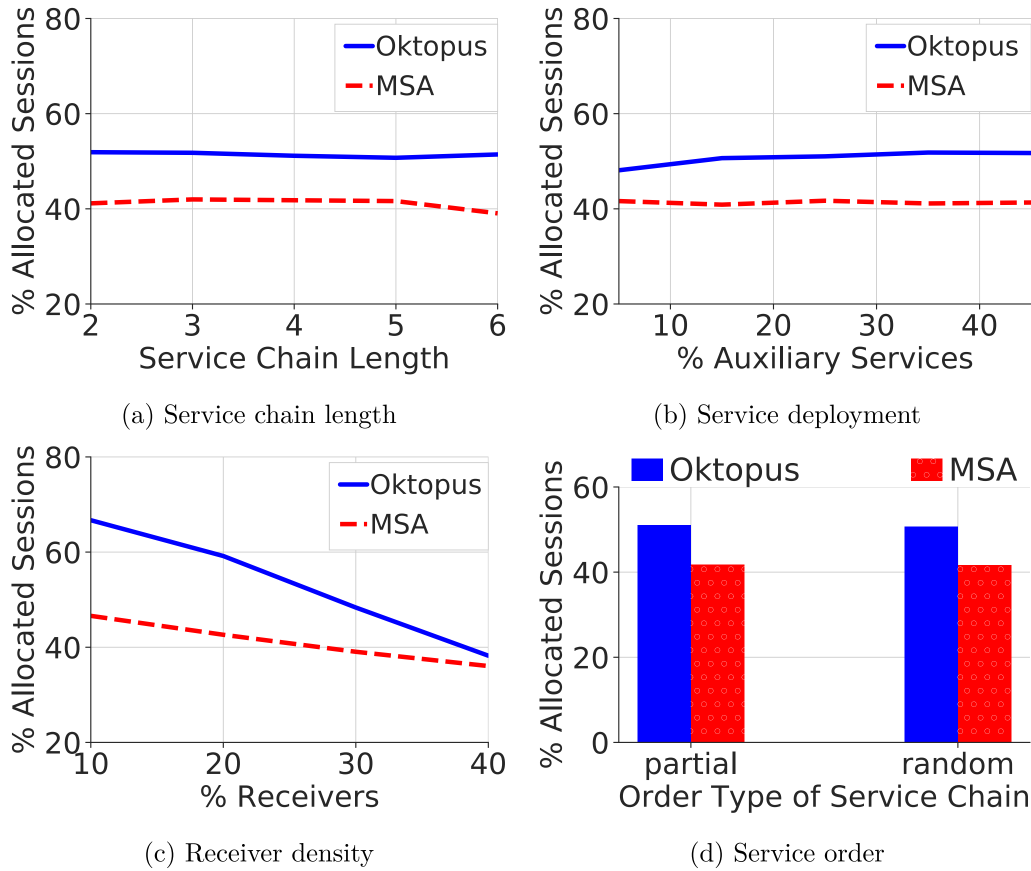


Figure 6.5: Percentage of allocated multicast sessions for the Colt ISP topology across different scenarios. # multicast sessions: 4,000.

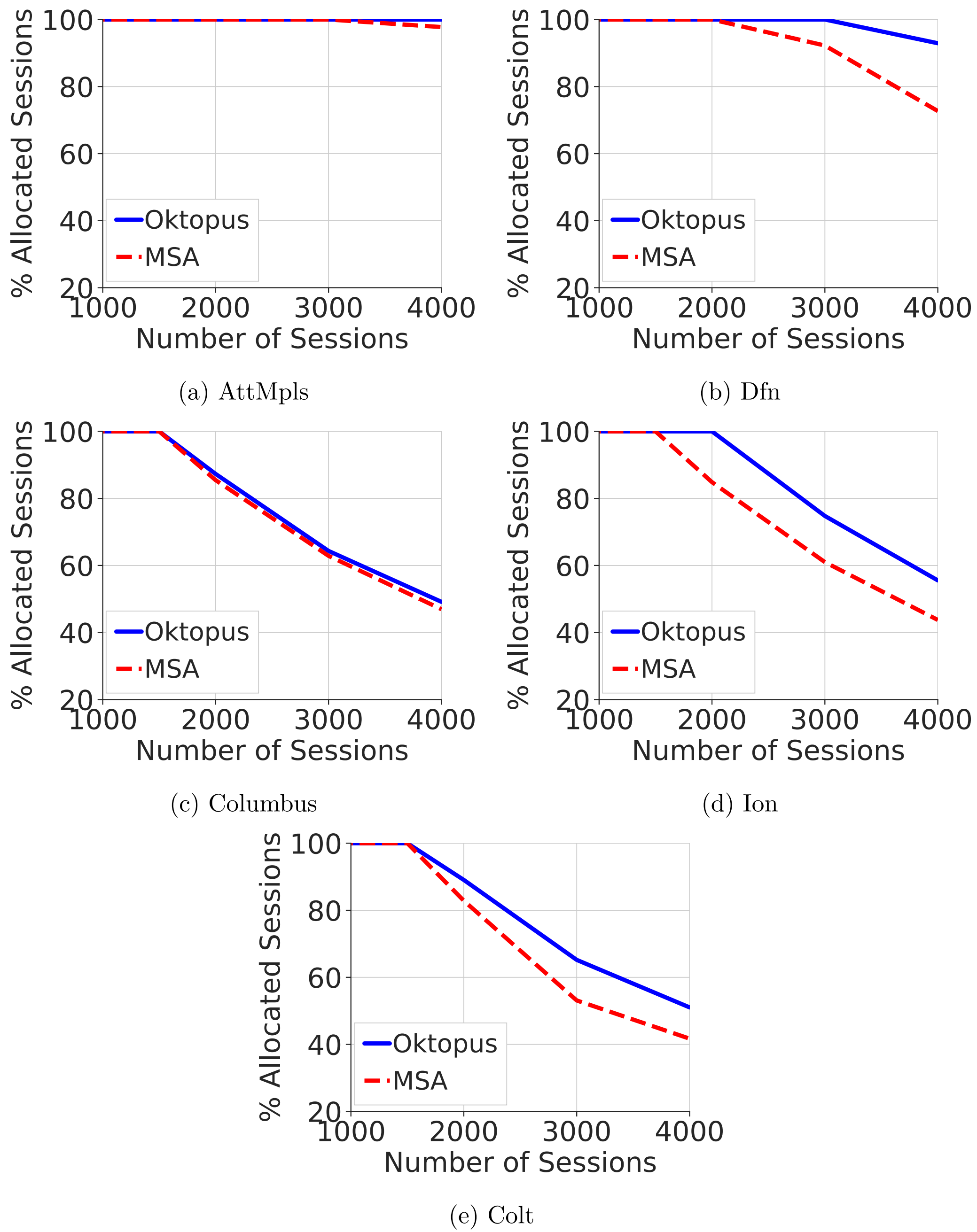
Oktopus utilizes network resources more efficiently than MSA because it calculates better paths while satisfying the required services by carefully engineers each link in the calculated graphs while balancing the load across different links.

Average Routing Cost and Graph Size. We next present the average routing cost and calculated graph size for the Ion ISP topology while varying the number of multicast sessions in Figure 6.7. Figure 6.7a shows that Oktopus results in similar routing cost per multicast session as MSA, while Oktopus allocates more multicast sessions as previously shown in Figure 6.6d.

To show the quality of the distribution graph, we also plot the average size of the calculated multicast distribution graphs in Figure 6.7b. The figure shows that both Oktopus and MSA computes similar graph sizes as well. Moreover, Figure 6.7c shows the standard deviation of the graph size. Oktopus is slightly larger than its counterpart of MSA (by up to two links) as Oktopus balances the traffic across the links. Also, it shows the average size of the multicast distribution graphs is close to the true average size. For instance, based on the 2000 allocated sessions, the average size of a distribution graph calculated by Oktopus is 79 links with a standard deviation of 15 links. This means that with 99.9% confidence the true average size of the multicast distribution graph is just $\pm$ 1 link from 79.

These results indicate that Oktopus can balance between maximizing the number of allocated sessions while reducing the average routing cost per session for the same network conditions. We observed similar results for other topologies and scenarios shown in Figures 6.8–6.11.

Average 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. For example, the average running time on the Ion topology is 8.625s and 8.5s per session for Oktopus and MSA algorithms, respectively. This slight overhead is used towards calculating more efficient distribution graphs, as demonstrated in the previous experiments. Moreover, many steps in the proposed algorithm can be run in parallel to reduce the average running time. We leave these optimizations for future work.



(a) AttMpls

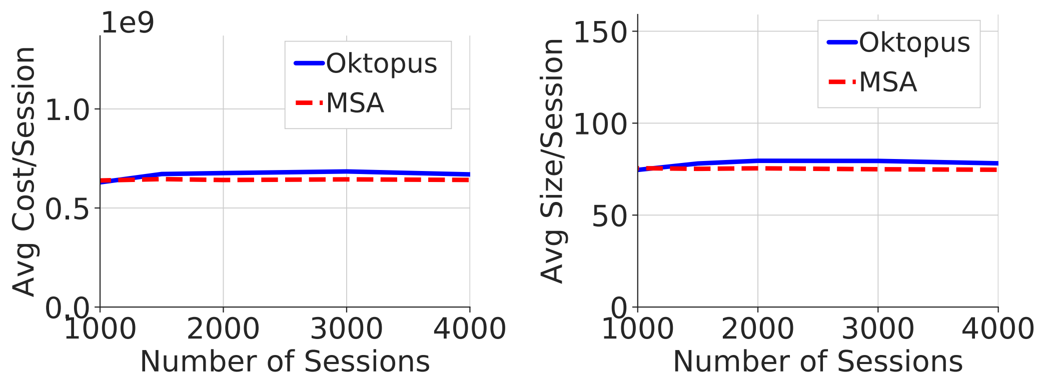
(b) Dfn

(c) Columbus

(d) Ion

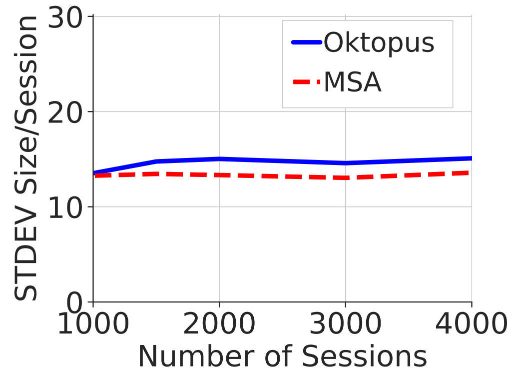
(e) Colt

Figure 6.6: Percentage of allocated multicast sessions with different number of multicast sessions.



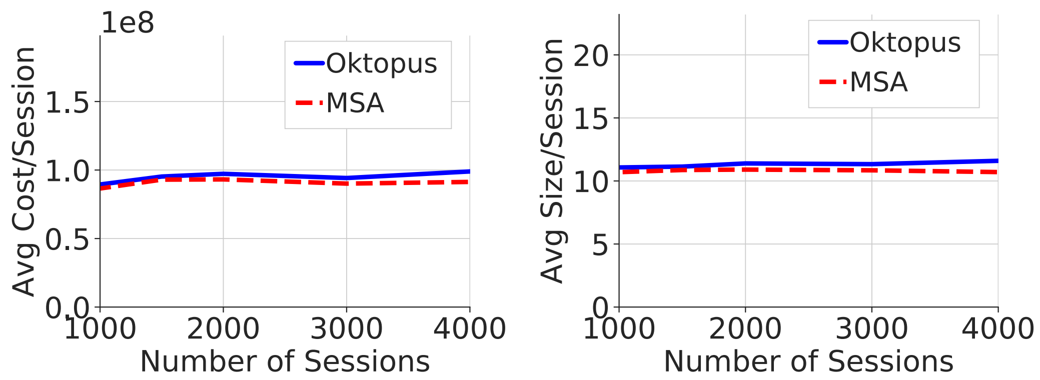
(a) Avg. routing cost

(b) Avg. graph size



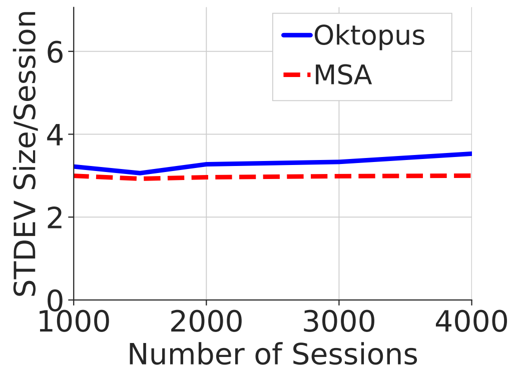
(c) Standard Deviation of graph size

Figure 6.7: Average routing cost and graph size per session for the Ion ISP Topology.



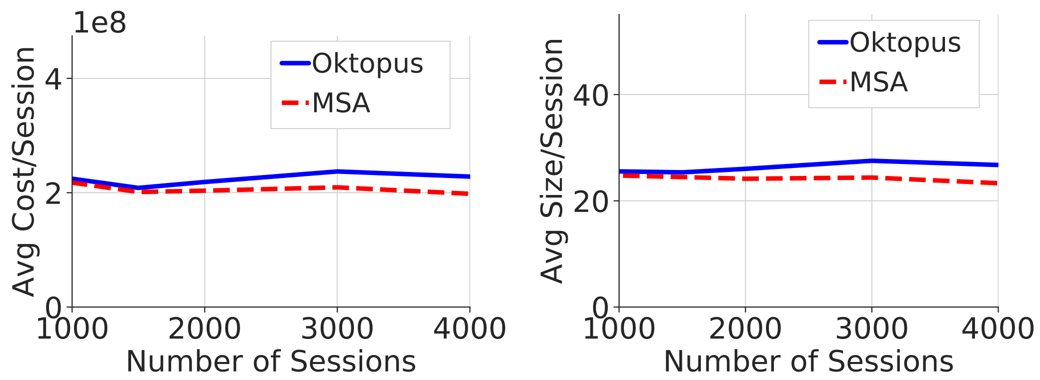
(a) Avg. routing cost

(b) Avg. graph size



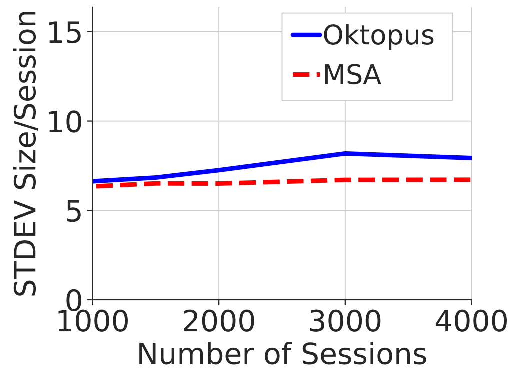
(c) Standard Deviation of graph size

Figure 6.8: Average routing cost and graph size per session for the AttMpls ISP Topology.



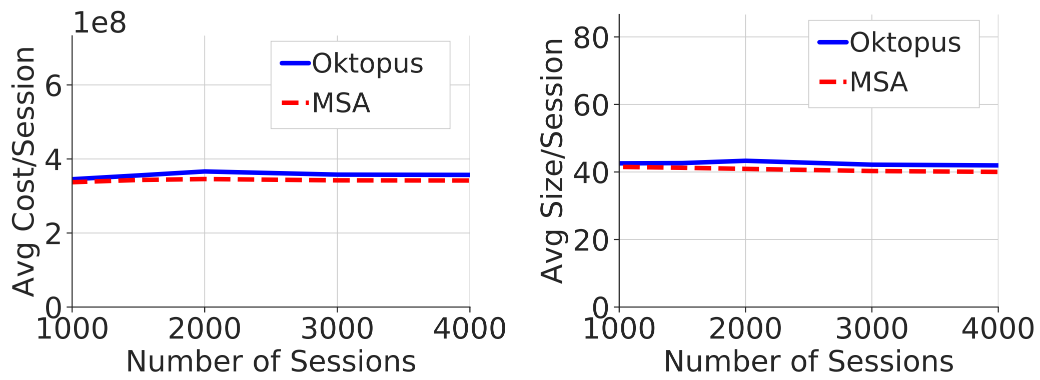
(a) Avg. routing cost

(b) Avg. graph size



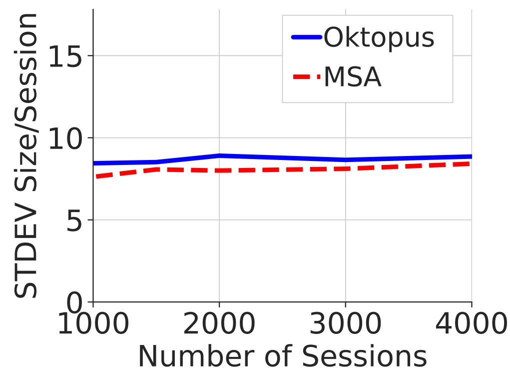
(c) Standard Deviation of graph size

Figure 6.9: Average routing cost and graph size per session for the Dfn ISP Topology.



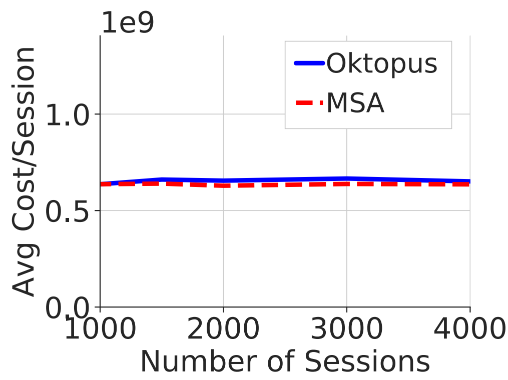
(a) Avg. routing cost

(b) Avg. graph size

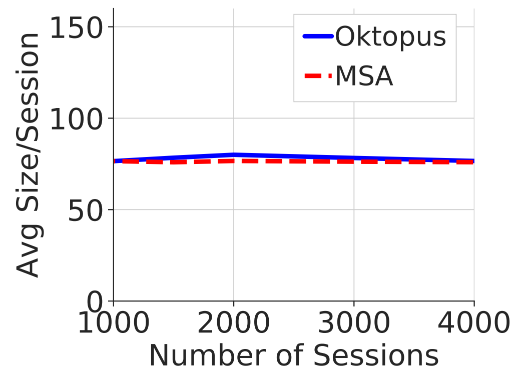


(c) Standard Deviation of graph size

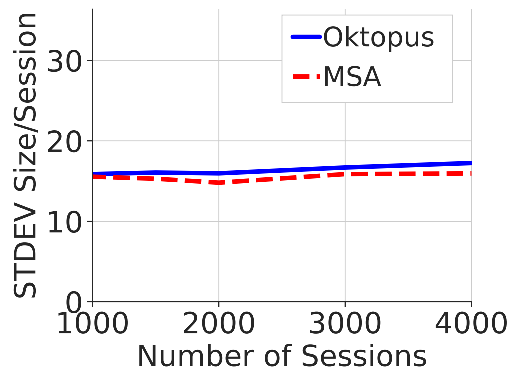
Figure 6.10: Average routing cost and graph size per session for the Columbus ISP Topology.



(a) Avg. routing cost



(b) Avg. graph size



(c) Standard Deviation of graph size

Figure 6.11: Average routing cost and graph size per session for the Colt ISP Topology.

Chapter 7

Conclusion and Future Work

In this chapter, we conclude the thesis and discuss some potential future works.

7.1 Conclusion

In this thesis, we considered the problem of multicast service chaining. 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 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 Oktopus on an intent-driven control plane framework. We evaluated Oktopus in simulations using real ISP topologies, and compared it versus the optimal solution and closest algorithm in the literature. Our experiments showed substantial gains compared to these algorithms. Specifically, 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 in the literature. Thus, Oktopus can optimized multicast service chaining in the ISP environment.

7.2 Future Work

Future work could examine the dynamic aspect of service chaining. The dynamic aspect of service chaining is referred to as *branching* [24], where a network service in the chain affects the requirement of the rest of the chain. For example, the initial service chain is $DPI \rightarrow Load_Balancer$. The DPI network service executes the traffic, and attack traffic is detected. Instead of going to the $Load_Balancer$, the traffic is altered to include a firewall, $DPI \rightarrow firewall$.

In this thesis, we assume that the network services are deployed, and multicast sessions are requested one at the time. Future work could include network service placement into

the optimization and explore optimizing concurrent sessions. Furthermore, it could extend Oktopus to support multiple applications and incorporate fair allocation of resources among applications in the optimization objectives.

Another future work worth examining is the extension to a multiple ISPs domain setting. In this case, the solution of the multicast service chaining problem will need to work with limited information, since separate administrative ISPs may not disclose all the network topology information.

Bibliography

[1] OpenFlow Switch Specification. Technical report, Open Networking Foundation, October 2013.

[2] Boulder Intent Framework. https://bit.ly/2WM8f7C, June 2015. [Online; accessed June 2020].

[3] Framework and Architecture for the Application of SDN to Carrier networks White Paper. https://bit.ly/2MeeBXg, July 2016. [Online; accessed June 2020].

[4] ONOS Intent Framework. https://bit.ly/2Tn6NGt, May 2016. [Online; accessed June 2020].

[5] Network Functions Virtualisation (NFV); Use Cases. Technical report, ETSI, May 2017.

[6] OpenDayLight Network Intent Composition (NIC). https://bit.ly/2XqPYMj, March 2018. [Online; accessed June 2020].

[7] White Paper: Cisco Annual Internet Report (2018–2023). https://bit.ly/2LK6q4M, March 2020. [Online; accessed June 2020].

[8] Saeed Akhoondian Amiri, Klaus-Tycho Foerster, Riko Jacob, and Stefan Schmid. Charting the Algorithmic Complexity of Waypoint Routing. *ACM SIGCOMM Comput. Commun. Rev.*, 48(1):42–48, April 2018.

[9] Anat Bremler-Barr, Yotam Harchol, David Hay, and Yaron Koral. Deep packet inspection as a service. In *Proc. of ACM CoNEXT'14*, pages 271–282, Sydney, Australia, December 2014.

[10] Jaroslaw Byrka, Fabrizio Grandoni, Thomas Rothvoß, and Laura Sanità. An Improved LP-Based Approximation for Steiner Tree. In *Proc. of STOC '10*, pages 583—592, Cambridge, Massachusetts, June 2010.

[11] Xiangwen Chen, Minghua Chen, Baochun Li, Yao Zhao, Yunnan Wu, and Jin Li. Celerity: A low-delay multi-party conferencing solution. *IEEE Journal on Selected Areas in Communications*, 31(9):155–164, September 2013.

[12] Sheng-Hao Chiang, Jian-Jhih Kuo, Shan-Hsiang Shen, De-Nian Yang, and Wen-Tsuen Chen. Online Multicast Traffic Engineering for Software-Defined Networks. In *Proc. of IEEE INFOCOM'18*, pages 414–422, Honolulu, HI, April 2018.

[13] Tae Won Cho, Michael Rabinovich, K.K. Ramakrishnan, Divesh Srivastava, and Yin Zhang. Enabling Content Dissemination Using Efficient and Scalable Multicast. In *Proc. of IEEE INFOCOM'09*, pages 1980–1988, Rio de Janeiro, Brazil, April 2009.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, July 2009.

[15] Al Danial. cloc, count lines of code. `https://github.com/AlDanial/cloc`. [Online; accessed June 2020].

[16] Khaled Diab and Carlos Lee. Oktopus framework. `https://github.com/Oktopus-Multicast/oktopus_framework`. [Online; accessed August 2020].

[17] Benoit Donnet, Korian Edeline, Iain R. Learmonth, and Andra Lutu. Middlebox classification and initial model. `https://bit.ly/3dZelXV`, July 2017. [Online; accessed June 2020].

[18] Jack Edmonds and Richard M. Karp. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *Journal of the ACM*, 19(2):248–264, April 1972.

[19] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proc. of USENIX NSDI'16*, pages 523–535, Santa Clara, CA, March 2016.

[20] Hao Feng, Jaime Llorca, Antonia M. Tulino, Danny Raz, and Andreas F. Molisch. Approximation algorithms for the NFV service distribution problem. In *Proc. of IEEE INFOCOM'17*, pages 1–9, Atlanta, GA, May 2017.

[21] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling Innovation in Network Function Control. In *Proc. of ACM SIGCOMM'14*, pages 163–174, Chicago, IL, August 2014.

[22] Luc De Ghein. *MPLS Fundamentals.* Cisco Press, 1st edition, November 2006.

[23] Vijay Gopalakrishnan, Bobby Bhattacharjee, K.K. Ramakrishnan, Rittwik Jana, and Divesh 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.

[24] Joel M. Halpern and Carlos Pignataro. Service Function Chaining (SFC) Architecture. RFC 7665, October 2015.

[25] 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, August 2015.

[26] 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, December 2018.

[27] Victor Heorhiadi, Michael Reiter, and Vyas Sekar. Simplifying Software-Defined Network Optimization Using SOL. In *Proc. of USENIX NSDI'16*, pages 223–237, Santa Clara, CA, March 2016.

[28] Liang-Hao Huang, Hsiang-Chun Hsu, Shan-Hsiang Shen, De-Nian Yang, and Wen-Tsuen Chen. Multicast Traffic Engineering for Software-Defined Networks. In *Proc. of IEEE INFOCOM'16*, pages 1–9, San Francisco, CA, April 2016.

[29] Nicolas Huin, Brigitte Jaumard, and Frederic Giroire. Optimal Network Service Chain Provisioning. *IEEE/ACM Transactions on Networking*, pages 1320–1333, May 2018.

[30] Ethan J. Jackson, Melvin Walls, Aurojit Panda, Justin Pettit, Ben Pfaff, Jarno Rajahalme, Teemu Koponen, and Scott Shenker. SoftFlow: A Middlebox Architecture for Open vSwitch. In *Proc. of USENIX ATC'16*, pages 15–28, Denver, CO, June 2016.

[31] Muhammad Jamshed, Younggyoun Moon, Donghwi Kim, Dongsu Han, and Kyoungsoo Park. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *Proc. of USENIX NSDI'17*, pages 113–129, Boston, MA, March 2017.

[32] Junaid Khalid, Aaron Gember-Jacobson, Roney Michael, Anubhavnidhi Abhashkumar, and Aditya Akella. Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr. In *Proc. of USENIX NSDI'16*, pages 239–253, Santa Clara, CA, March 2016.

[33] Simon Knight, Hung Nguyen, Nickolas Falkner, Rhys Bowden, and Matthew Roughan. The Internet Topology Zoo. *IEEE Journal on Selected Areas in Communications*, pages 1765–1775, September 2011.

[34] Diego Kreutz, Fernando Ramos, Paulo Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE*, 103(1):14–76, Jan 2015.

[35] Jian-Jhih Kuo, Shan-Hsiang Shen, Ming-Hong Yang, De-Nian Yang, Ming-Jer Tsai, and Wen-Tsuen Chen. Service Overlay Forest Embedding for Software-Defined Cloud Networks. In *Proc. of IEEE ICDCS'17*, pages 720–730, Atlanta, GA, June 2017.

[36] Tung-Wei Kuo, Bang-Heng Liou, Kate Ching-Ju Lin, and Ming-Jer Tsai. Deploying Chains of Virtual Network Functions: On the Relation Between Link and Server Usage. In *Proc. of IEEE INFOCOM'16*, pages 1–9, San Francisco, CA, April 2016.

[37] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: Securely Outsourcing Middleboxes to the Cloud. In *Proc. of USENIX NSDI'16*, pages 255–273, Santa Clara, CA, March 2016.

[38] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proc. of ACM SIGCOMM'16*, pages 1–14, Florianopolis, Brazil, August 2016.

[39] João Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *Proc. of USENIX NSDI'14*, pages 459–473, Seattle, WA, April 2014.

[40] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *Proc. of USENIX OSDI'16*, pages 203–216, Savannah, GA, November 2016.

[41] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. SafeBricks: Shielding Network Functions in the Cloud. In *Proc. of USENIX NSDI'18*, pages 201–216, Renton, WA, April 2018.

[42] Zhang Qixia, Yikai Xiao, Fangming Liu, John C.s Lui, Jian Guo, and Tao Wang. Joint Optimization of Chain Placement and Request Scheduling for Network Function Virtualization. In *Proc. of IEEE ICDCS'17*, pages 731–741, Atlanta, GA, June 2017.

[43] Paul Quinn, Uri Elzur, and Carlos Pignataro. Network Service Header (NSH). Internet-Draft draft-ietf-sfc-nsh-26, Internet Engineering Task Force, October 2017. Work in Progress.

[44] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proc. of USENIX NSDI'13*, pages 227–240, Lombard, IL, April 2013.

[45] Aravindh Raman, Gareth Tyson, and Nishanth Sastry. Facebook (A)Live?: Are Live Social Broadcasts Really Broadcasts? In *Proc. of the International World Wide Web Conference (WWW'18)*, pages 1491–1500, Lyon, France, April 2018.

[46] Bangbang Ren, Deke Guo, Guoming Tang, Xu Lin, and Yudong Qin. Optimal Service Function Tree Embedding for NFV Enabled Multicast. In *Proc. of IEEE ICDCS'18*, pages 132–142, Vienna, Austria, July 2018.

[47] Gamal Sallam, Gagan Gupta, Bin Li, and Bo Ji. Shortest Path and Maximum Flow Problems Under Service Function Chaining Constraints. In *Proc. of IEEE INFOCOM'18*, pages 2132–2140, Honolulu, HI, April 2018.

[48] Yu Sang, Bo Ji, Gagan Gupta, Xiaojiang Du, and Lin Ye. Provably efficient algorithms for joint placement and allocation of virtual network functions. In *Proc. of IEEE INFOCOM'17*, pages 1–9, Atlanta, GA, May 2017.

[49] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *Proc. of ACM SIGCOMM'12*, pages 13–24, Helsinki, Finland, August 2012.

[50] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. BlindBox: Deep Packet Inspection over Encrypted Traffic. In *Proc. of ACM SIGCOMM'15*, pages 213–226, London, United Kingdom, August 2015.

[51] Neil Spring, Ratul Mahajan, and David Wetherall. Measuring ISP topologies with Rocketfuel. *IEEE/ACM Transactions on Networking*, 12(1):2–16, February 2004.

[52] Chen Sun, Jun Bi, Zhilong Zheng, Heng Yu, and Hongxin Hu. NFP: Enabling Network Function Parallelism in NFV. In *Proc. of ACM SIGCOMM'17*, pages 43–56, Los Angeles, CA, August 2017.

[53] Bob Thomas, Loa Andersson, and Ina Minei. LDP Specification. RFC 5036, October 2007.

[54] Bob Thomas, IJsbrand Wijnands, Ina Minei, and Kireeti Kompella. Label Distribution Protocol Extensions for Point-to-Multipoint and Multipoint-to-Multipoint Label Switched Paths. RFC 6388, November 2011.

[55] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. ResQ: Enabling SLOs in Network Function Virtualization. In *Proc. of USENIX NSDI'18*, pages 283–297, Renton, WA, April 2018.

[56] Bohdan Trach, Alfred Krohmer, Franz Gregor, Sergei Arnautov, Pramod Bhatotia, and Christof Fetzer. ShieldBox: Secure Middleboxes Using Shielded Execution. In *Proc. of ACM Symposium on SDN Research (SOSR)'18*, pages 1–14, Los Angeles, CA, March 2018.

[57] Zuckerberg Really Wants You to Stream Live Video on Facebook. `https://bit.ly/2v6uHqF`, April 2016. [Online; accessed May 2020].

[58] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. Elastic Scaling of Stateful Network Functions. In *Proc. USENIX NSDI'18*, pages 299–312, Renton, WA, April 2018.

[59] Zichuan Xu, Weifa Liang, Meitian Huang, Mike Jia, Song Guo, and Alex Galis. Approximation and Online Algorithms for NFV-Enabled Multicasting in SDNs. In *Proc. of IEEE ICDCS'17*, pages 625–634, Atlanta, GA, June 2017.

[60] Xingliang Yuan, Xinyu Wang, Jianxiong Lin, and Cong Wang. Privacy-preserving deep packet inspection in outsourced middleboxes. In *Proc. of IEEE INFOCOM'16*, pages 1–9, San Francisco, CA, April 2016.