3-27-2018

Traffic-Aware Deployment of Interdependent NFV Middleboxes in Software-Defined Networks

Wenrui Ma
wma006@fiu.edu

DOI: 10.25148/etd.FIDC006528

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

TRAFFIC AWARE DEPLOYMENT OF INTERDEPENDENT NFV MIDDLEBOXES IN SOFTWARE-DEFINED NETWORKS

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Wenrui Ma

2018

To: Dean John Volakis
College of Engineering and Computing

This dissertation, written by Wenrui Ma, and entitled Traffic Aware Deployment of Interdependent NFV Middleboxes in Software-Defined Networks, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Sundaraja Sitharama Iyengar

Niki Pissinou

Jason Liu

Gang Quan

Deng Pan, Major Professor

Date of Defense: March 27, 2018

The dissertation of Wenrui Ma is approved.

Dean John Volakis
College of Engineering and Computing

Andres G. Gil
Vice President for Research and Economic Development and
Dean of the University Graduate School

Florida International University, 2018

DEDICATION

To my parents and husband.

ACKNOWLEDGMENTS

First and foremost, I would like to express my sincerest thanks and appreciation to my major advisor Dr. Deng Pan. Without his invaluable advice and consistent encouragement throughout my Ph.D. study, it would not have been possible for me to go through this doctoral journey and complete my dissertation. I am extremely grateful for his endless patience, continued guidance during my Ph.D. research.

Second, I want to extend my gratitude to all my dissertation committee members: Dr. S.S Iyengar, Dr. Pissinou, Dr. Jason Liu and Dr. Gang Quan, for taking their time to serve on the committee and provide insightful comments on my dissertation work.

Third, I am very grateful to all my co-authors, Jonathan Beltran and Zhenglin Pan, for our valuable discussion and helpful assistance to my research problem.

Additionally, I would like extend my thanks to our department staff for assisting me with the administrative taskes necessary during my doctoral study: Olga Carbonell, Carlos Cabrera, Vanessa Cornwall, etc.

Finally, and most importantly, I want to thank my husband and my parents for their unconditional love. Their company and understanding has been an immense motivation of my Ph.D. life.

ABSTRACT OF THE DISSERTATION

TRAFFIC AWARE DEPLOYMENT OF INTERDEPENDENT NFV MIDDLEBOXES IN SOFTWARE-DEFINED NETWORKS

by

Wenrui Ma

Florida International University, 2018

Miami, Florida

Professor Deng Pan, Major Professor

Middleboxes, such as firewalls, Network Address Translators (NATs), Wide Area Network (WAN) optimizers, or Deep Packet Inspectors (DPIs), are widely deployed in modern networks to improve network security and performance. Traditional middleboxes are typically hardware based, which are expensive and closed systems with little extensibility. Furthermore, they are developed by different vendors and deployed as standalone devices with little scalability. As the development of networks in scale, the limitations of traditional middleboxes bring great challenges in middlebox deployments.

Network Function Virtualization (NFV) technology provides a promising alternative, which enables flexible deployment of middleboxes, as virtual machines (VMs) running on standard servers. However, the flexibility also creates a challenge for efficiently placing such middleboxes, due to the availability of multiple hosting servers, capabilities of middleboxes to change traffic volumes, and dependency between middleboxes. In our first two work, we addressed the optimal placement challenge of NFV middleboxes by considering middlebox traffic changing effects and dependency relations. Since each VM has only a limited processing capacity restricted by its available resources, multiple instances of the same function are necessary in an NFV network. Thus, routing in an NFV network is also a challenge to determine not only

a path from the source to destination but also the service (middlebox) locations. Furthermore, the challenge is complicated by the traffic changing effects of NFV services and dependency relations between them. In our third work, we studied how to efficiently route a flow to receive services in an NFV network.

We conducted large-scale simulations to evaluate our proposed solutions, and also implemented an Software-Defined Networking (SDN) based prototype to validate the solutions in realistic environments. Extensive simulation and experiment results have been fully demonstrated the effectiveness of our design.

TABLE OF CONTENTS

CHAPTER PAGE

1. INTRODUCTION 1
1.1 Background 1
1.2 Motivation 2
1.3 Outline of Our Work 4
1.3.1 Traffic Aware Placement of NFV Middleboxes 4
1.3.2 Traffic Aware Placement of Interdependent NFV Middleboxes 5
1.3.3 Service Aware Flow routing 6

2. RELATED WORK 7
2.1 Network Function Virtualization 7
2.1.1 NFV Framework 7
2.1.2 NFV Software Architecture 8
2.1.3 NFV Hardware Architecture 9
2.1.4 Middlebox Management Interfaces 10
2.1.5 General Purpose Network Elements 10
2.2 Software Defined Networking 11
2.2.1 SDN Architecture 11
2.2.2 SDN Switch Memory Management 12
2.3 SDN based Middlebox Policy Enforcement 13
2.4 NFV VM Deployment 14
2.5 Network Service Chaining 15

3. TRAFFIC AWARE NFV MIDDLEBOX PLACEMENT 17
3.1 Introduction 17
3.2 Problem Formulation 20
3.3 Traffic Aware Middlebox Placement with Predetermined Paths 24
3.3.1 Middlebox Placement for Single Flow 24
3.3.2 Middlebox Placement for Multiple Flows 27
3.4 Traffic Aware Middlebox Placement without Predetermined Paths 31
3.4.1 NP-Hardness Proof 31
3.4.2 LFGL based MinMax Routing for Single Flow 34
3.4.3 Optimization of Multiple Flows 37
3.5 Implementation 39
3.5.1 Flow Arrival Notification 40
3.5.2 Flow Path and Middlebox Placement Calculation 40
3.5.3 Midllebox VMs Startup 41
3.5.4 Flow Table Update 41
3.5.5 Middlebox Emulator Development 42
3.5.6 Link Load Monitoring 42
3.5.7 Post-processing 43

3.6 Experiment and Simulation Results . . . 43
3.6.1 Benchmark Solutions . . . 43
3.6.2 Experiment Results with Prototype . . . 44
3.6.3 Simulation Results . . . 49
3.6.4 Comparison between Experimental and Simulation Results . . . 54
3.7 Summary . . . 55

4. TRAFFIC AWARE PLACEMENT OF INTERDEPENDENT NFV MIDDLEBOXES . . . 57
4.1 Introduction . . . 57
4.2 Problem Statement . . . 60
4.3 Middlebox Placement with Predetermined Paths . . . 64
4.3.1 Non-Ordered Middlebox Set . . . 65
4.3.2 Totally-Ordered Middlebox Set . . . 68
4.3.3 Partially-Ordered Middlebox Set . . . 71
4.4 Middlebox Placement without Predetermined Path . . . 77
4.4.1 NP-Hardness . . . 77
4.4.2 Traffic and Space Aware Routing . . . 79
4.5 Experiment and Simulation Results . . . 81
4.5.1 Simulation Results . . . 81
4.5.2 Experiment Results with Prototype . . . 86
4.6 Summary . . . 91

5. SERVICE AWARE FLOW ROUTING . . . 92
5.1 Introduction . . . 92
5.2 Problem Formulation . . . 96
5.3 Algorithm Design . . . 100
5.3.1 NP-Hardness Proof . . . 100
5.3.2 Optimal Routing for Totally-Ordered Service Set . . . 102
5.3.3 Converting Partially-Ordered Set to Totally-Ordered Set . . . 105
5.3.4 Greedy Routing for Partially-Ordered Service Set . . . 105
5.3.5 Mix of Existing and New Instances . . . 106
5.4 Experimental And Simulation Results . . . 107
5.4.1 Simulation Results . . . 107
5.4.2 Experimental Results with Prototype . . . 112
5.5 Summary . . . 114

6. CONCLUSIONS AND FUTURE DIRECTIONS . . . 116

BIBLIOGRAPHY . . . 119

VITA . . . 130

LIST OF FIGURES

FIGURE PAGE

2.1 NFV Infrastructure. 8

2.2 SDN Architecture. 12

3.1 Traffic changing effects of middleboxes. 19

3.2 Proof of Theorem 1. 25

3.3 Reduction from 3-Satisfiability to TAMP for multiple flows with predetermined paths. 27

3.4 Reduction from Hamiltonian cycle to TAMP. 31

3.5 Middlebox placement workflow. 39

3.6 Tree topology for experiments. 45

3.7 LFGL experiment results. 46

3.8 Multipath topology for experiments. 47

3.9 LFGL based MinMax routing experiment results. 48

3.10 LFGL simulation results. 50

3.11 Improvement of multi-flow optimization with predetermined paths. . . . 51

3.12 LFGL based MinMax routing simulation results. 52

3.13 Improvement of multi-path optimization without predetermined paths. . 53

3.14 LFGL based MinMax routing simulation results (1024 hosts 1 Mbps link capacity). 54

3.15 Improvement of multi-path optimization without predetermined paths (1024 hosts 1 Mbps link capacity). 55

4.1 Traffic changing effects of middleboxes. 59

4.2 Examples of non-ordered, totally-ordered, and partially-ordered middlebox set. 65

4.3 Reduction from Clique to TAPIM with predetermined path. 72

4.4 Reduction from Hamiltonian Cycle to TAPIM. 77

4.5 NOSP simulation results. 83

4.6 TOSP simulation results. 84
4.7 Partial to total order conversion simulation results. 85
4.8 TASAR simulation results. 86
4.9 Abilene backbone network topology. 87
4.10 Prototype experiment results. 87
4.11 NSF network topology. 89
4.12 Prototype experiment results. 90
5.1 Service aware routing challenge in NFV networks. 94
5.2 Reduction from Hamiltonian Cycle to SAR. 100
5.3 Totally-ordered set routing example. 103
5.4 Simulation results for totally-ordered service set. 109
5.5 Simulation results for partially-ordered service set. 111
5.6 Partially-ordered service sets for experiments. 112
5.7 Prototype experiment results. 114

CHAPTER 1

INTRODUCTION

1.1 Background

Middleboxes are special network functions that offer valuable benefits, such as ensuring security (e.g., firewalls and intrusion detection systems), improving performance (e.g., proxies) and reducing bandwidth costs (e.g., WAN optimizers) [SHS+12, WQX+11]. Unlike networking equipment (e.g., switches, routers) focusing on network Layer 2/3 functions (forwarding and routing functions), middleboxes focus on examining and modifying traffic [SRR+11]. Nowadays, middleboxes are widely deployed in data centers, clouds and enterprise networks to achieve the aforementioned benefits [SHS+12, BASS11, LC15].

Traditional hardware-based middleboxes suffer from a number of drawbacks [MSG+16, PSS15], including high cost, short lifetime, function inflexibility, and difficulty to scale up. Virtualization technology provides a promising alternative. In computing, Virtualization refers to the act of creating a software-based (or virtual) representation of something rather than a physical one. Virtualization can apply to applications, networks, servers, and storage, and is an efficient way to boost efficiency and agility [CB05]. Network Function Virtualization (NFV) involves implementing network functions in software that can run in a variety of forms: as virtual machines (VMs), in hypervisors, on commodity servers, and as a collection of processes [GPGA12]. In NFV terminology, software middleboxes are referred to as Virtualized Network Functions (VNFs). ETSI [etsa] defines the NFV architecture enabling virtualized network functions (VNF) to be executed on commodity servers wrapped with a hypervisor [BDF+03, KKL+07]. Above the hypervisor layer, a VNF is typically mapped to one VM [LC15]. Utilizing benefits of the underlying

virtualization technology, NFV middleboxes enjoy many advantages not available in traditional hardware-based middleboxes [etsb], such as fast deployment, reduced energy consumption, and real-time optimization. In the following, we use middleboxes, network functions, and services interchangeably.

Even though NFV technology makes middleboxes provision flexibly, it also brings several challenges, such as the guarantee of performance for NFV middleboxes, dynamic instantiation and efficient placement of NFV middleboxes. In most cases, a traffic is required to pass through multiple middleboxes in a particular order, e.g., a traffic may be required to go through an IDS, then a proxy and finally a firewall [Z. 13]. In traditional networks, it requires lots of manual efforts to configure and update routing policies to steer traffic. With the development of Software Defined Networking (SDN), the trend of integrating SDN with NFV to achieve various network control and management goals has seen noticeable growth [YTG13]. SDN can be applied to assist NFV in addressing the challenges of dynamic resource management and intelligent service orchestration [Rao14].

1.2 Motivation

Since middleboxes focus on traffic inspecting and modifying, they have the potential to change the volume of processed traffic and may do it in different ways. For example, the Citrix CloudBridge WAN optimizer compresses traffic before sending it to the next hop, and may reduce the traffic volume by up to 80% [wan]. On the other hand, the BCH (63,48) encoder, used for satellite communications signaling messages, increases the traffic volume by 31% due to the checksum overhead [MVB93]. Finally, a firewall will keep the traffic rates of allowed flows unchanged and reduce the rates of denied flows to zero.

The placement of middleboxes is also constrained by the dependency relation that may or may not exist between middleboxes [S. 14]. For instance, an IPSec decryptor is usually placed before a NAT gateway [cis], while a VPN proxy can be placed either before or after a firewall [ms-].

Due to the flexible VM implementation of NFV, a number of challenges must be addressed to fully utilize its advantages. In the first place, *the possibility of multiple NFV servers to host a middlebox makes a strategic deployment plan necessary.* Unlike the traditional hardware appliance installed at a fixed location, a middlebox VM can be hosted by NFV servers at different locations. The middlebox may also change the volume of processed traffic. Thus, inappropriate deployment of middleboxes will cause flows to traverse lengthy paths and create congested links. Next, *the possibility of multiple middlebox instances of the same type necessitates an efficient routing algorithm.* While a single hardware appliance of a type is usually sufficient in a traditional network, multiple middleboxes of the same type may be necessary due to the limited processing capability of a single VM. When a flow needs to be processed by a sequence of middleboxes, it is challenging to find an efficient routing path that passes one of each type of required middleboxes, since there may exist numerous such combinations in the network.

Previous research on middleboxes has focused on middlebox virtualization on commodity servers [J. 12, V. 12, J. 15], virtualized software middlebox platforms [J. 14, GJVP+14], and placement and chaining of middleboxes in SDN networks [Z. 13, FSYM13, FCS+14]. To the best of our knowledge, traffic aware deployment of NFV middleboxes, and in particular the the traffic changing effects, have not been well investigated. In our research, we focus on studying traffic aware deployment of NFV middleboxes and service aware flow routing in software defined networks.

1.3 Outline of Our Work

The aim of our work is to design a suit of NFV middleboxes placement and flow routing algorithms based on different optimization objectives. In addition, we leverage SDN technology [opea, sdnb, sdna, Rao14] to build a prototype system for NFV middleboxes placement and flow routing to demonstrate our design. NFV enables the flexible and dynamic deployment of network functions. SDN separates the networks control and data plane. SDN controller has a global view of networks. NFV middleboxes can be initiated in real time at proper locations, and then SDN controllers can automatically enforce forwarding rules to route traffic to the desired middleboxes.

In networks, an elephant flow is long-lived and extremely large (in total bytes) [MUK+04, CMT+11]. For elephant flows, throughput is far more important than latency. A mouse flow [GM01] is often associated with bursty, latency-sensitive applications. Based on different features, we optimize traffic of elephant flows by carefully planning their middlebox locations and routing paths, and calculating efficient paths for mice flows that traverse existing middleboxes in the desired priority order. I summarize the contributions of our work in the following.

1.3.1 Traffic Aware Placement of NFV Middleboxes

In Chapter 3, we study how to efficiently deploy NFV middleboxes without dependencies to achieve load balance using an SDN approach, and considered in particular the traffic changing effects of different middleboxes. We formulate the Traffic Aware Middlebox Placement (TAMP) problem as a graph-based optimization problem, and solve it in two steps. First, we solve the special case of TAMP when flow paths are predetermined. For a single flow, we propose the Least-First-Greatest-Last (LFGL)

rule, and prove its optimality; for multiple flows, we prove NP-hardness by reduction from the 3-Satisfiability problem and propose an efficient heuristic. Next, we solve the general version of TAMP without predetermined flow paths. We prove that the general TAMP problem is NP-hard by reduction from the Hamiltonian problem, and propose the LFGL based MinMax routing algorithm by integrating LFGL with MinMax routing. To validate our design, we have implemented the proposed algorithms in a prototype system with the open-source SDN controller Floodlight [Flo] and emulation platform Mininet [min]. In addition, we conducted simulations in ns-3 for performance evaluation in large-scale networks. Extensive experiment and simulation results are presented to demonstrate the superiority of our algorithms over competing solutions.

1.3.2 Traffic Aware Placement of Interdependent NFV Middleboxes

In Chapter 4, we have studied the optimal placement of NFV middleboxes by considering different middlebox traffic changing effects and dependency relations. We first formulate the Traffic Aware Placement of Interdependent Middleboxes problem as a graph optimization problem with the objective to load-balance the network. Next, we solve the problem when the flow path is predetermined, and propose optimal algorithms for a non-ordered or totally-ordered middlebox set. For the general scenario of a partially-ordered middlebox set, we show that the problem is NP-hard by reduction from the Clique problem, and propose an efficient heuristic to convert a partially-ordered set to a totally-ordered one. On the other hand, when the flow path is not predetermined, we prove that the studied problem is NP-hard even for a non-ordered middlebox set by reduction from the Hamiltonian Cycle problem, and propose the Traffic And Space Aware Routing heuristic. We have conducted large scale simulations to evaluate the proposed solutions, and have also implemented an

SDN based prototype to validate them in realistic environments. Extensive simulation and experiment results are presented to show the effectiveness of our design.

1.3.3 Service Aware Flow routing

The limited processing capability of a VM makes it necessary to deploy multiple NFV instances of the same service. Routing in NFV networks is thus a challenge to not only find a path from the source to destination, but also determine the optimal service locations. In Chapter 5, we have studied the service aware routing problem in NFV networks, and consider in particular the traffic changing effects of NFV services and dependency relations between them. First, we formulate the service aware routing problem as a graph optimization problem, and prove that it is NP-hard by reduction from the Hamiltonian Cycle problem. Next, for the special scenario of a totally-ordered service set, we propose an efficient polynomial-time algorithm and prove its optimality. On the other hand, for the NP-hard general scenario of a partially-ordered service set, we propose two practical heuristics with low time complexity, one by converting the partially-ordered set to a totally-ordered one, and the other using a greedy approach. We have validated the design in an SDN based small-scale prototype, and also implemented the algorithms in the ns-3 simulator for large-scale performance evaluation. Extensive simulation and experimental results are presented to demonstrate the effectiveness of the proposed algorithms.

CHAPTER 2

RELATED WORK

In this chapter, we would highlight the research efforts that are related to our work. In particular, Section 2.1 reviews the development of Network Function Virtualization (NFV) technology. Section 2.2 presents a network architecture: Software-Defined Networking (SDN). Section 2.3 review the existing work, which adopt SDN as a solution to route traffic through middleboxes. Section 2.4 reviews the existing work on the deployment of NFV virtual machines (VMs). Section 2.5 describes the existing approaches for network service [1]chaining with different optimization objectives.

2.1 Network Function Virtualization

2.1.1 NFV Framework

The European Telecommunications Standards Institute (ETSI) defines the NFV architectural framework as shown in Fig 2.1 enabling virtualized network functions (VNF) to be deployed and executed on a Network Functions Virtualisation Infrastructure (NFVI) [etsb], which consists of commodity servers wrapped with a software layer that abstracts and logically partitions them [MSG+16]. Above the hypervisor layer, a VNF is typically mapped to one VM in the NFVI. The deployment, execution and operation of VNFs on the NFVI are steered by a Management and Orchestration (MANO) system [man, GKJ+13], whose behaviour is driven by a set of metadata describing the characteristics of the network services and their constituent VNFs. The MANO system includes an NFV Orchestrator in charge of

[1]In this dissertation, the three terms "service", "middlebox" and "network functions" have the same meaning.

the lifecycle of network services, a set of VNF managers in charge of the lifecycle of the VNFs and a virtualized infrastructure manager, which can be viewed as an extended cloud management system responsible for controlling and managing NFVI resources.

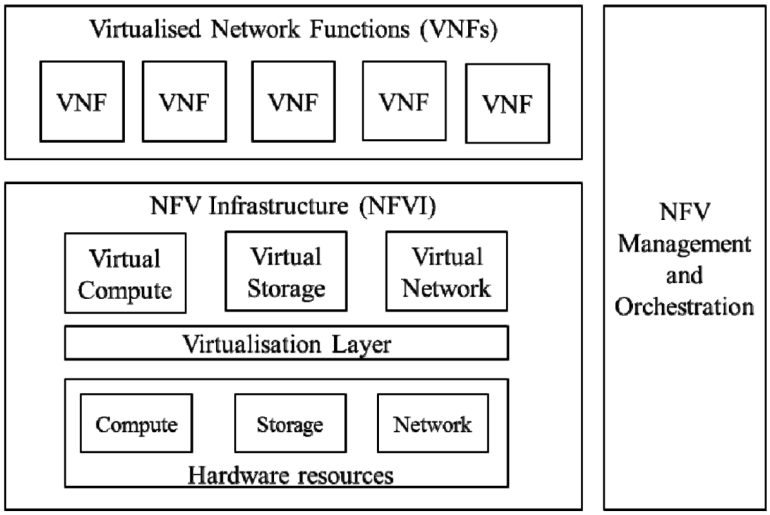


Figure 2.1: NFV Infrastructure.

2.1.2 NFV Software Architecture

NFV has been proposed to shift middlebox processing from closed network appliances to software running on commodity hardware. Hwang *et al.* [J. 15] propose the NetVM a software platform for running diversity network functionality at line-speed based on the general commodity hardware. It takes advantage of KVM [KKL+07] and DPDKs [dpd] high throughput packet processing capabilities, and further enables flexible traffic steering and overcomes the performance limitations of hardware switching. Thus, It provides the capability to support network functions chains by flexible, high-performance network elements. ClickOS [J. 14] is a high-performance, virtualized software middlebox platform. It provides small, booting quickly, and

little delay virtual machines, and over one hundred of them can be concurrently run while guaranteeing ine-rate pipe on the general commodity server. To achieve high performance, ClickOS relies an extensive overhaul of Xens I/O subsystem [STJP08] to speed up the networking process in middleboxes. ClickOS is proof that software solutions alone are enough to significantly speed up virtual machine processing, to the point where the remaining overheads are dwarfed by the ability to safely consolidate heterogeneous middlebox processing onto the same hardware. The results of NetVM and ClickOS shows that the software middleboxes can be hosted on virtual machines and migrated to other locations easily. Anat et al. propose a logically centralized framework named OpenBox [A. 15] that decouples the control plane of NFV services from their data plane. Jamshed et al. present mOs [M. 17a], a reusable networking stack, to provide a well-defined set of APIs for NFV applications to interact with the system. Our work can benefit from those research advances by implementing our solutions based on the above NFV architectures.

2.1.3 NFV Hardware Architecture

Multiple efforts have been focusing on designing efficient NFV hardware architectures. xOMB (Extensible Open MiddleBox) [J. 12] provides programmable, flexible and scalable middleboxes on the platform of general hardware like servers and operating systems to achieve high efficiency flow controlling. It utilize general programmable processing approaches with user-defined modules for network packet parsing, data transforming, and flow forwarding. By these design, xOMB shows how middleboxes can be utilized to support different services. To address the important resource management and controlling problems that arise in exploiting the benefits of middlebox deployment, CoMb [V. 12] is proposed by consolidating individual middleboxes through decoupling the software and hardware, which enables

software-based implementations of middlebox to deploy and run on a the general and consolidated hardware platform. On the other hand, CoMb consolidates the management of different middlebox into a single centralized controller, which takes a unified and network wide configurations and controlling for policy requirements across the overall traffic and applications. This is in contrast to todays approach where the middleboxs is controlled and managed separately. CoMb addressed these important resource control and management challenges, which results in reducing network provisioning cost and overhead in the deployment and operation of middlebox devices. Our work can benefit from these advances as well. We can use the consolidated servers to host middlebox services and openflow switches.

2.1.4 Middlebox Management Interfaces

There are some efforts to standardize middlebox control interfaces such as MIDCOM [SQT08] and SIMCO [sim]. Aaron et al. propose API extensions to expose middlebox internal state to an SDN controller [GPGA12]. They advocated for the design of a software-defined middlebox networking framework capable of supporting scenarios like middlebox scaling and live network migration. They believe that continued innovation in middlebox functionality and operation hinges on the development of SDN like frameworks for middlebox management. Their work offers insights we can manage the deployment of middleboxes by utilizing SDN technology.

2.1.5 General Purpose Network Elements

Many efforts have been made aiming to build commodity network elements using x86 CPUs [DEA+09, EGH+08, GHH+09], GPUs [HJPM11], and merchant switch silicon [LGL+11]. RouterBricks [DEA+09] propose a software router architecture

that parallelizes router functionality both across multiple servers and across multiple cores within a single server. Adam et al. [GHH+09] introduce a new class of system architectures for building network flow processing platforms. These architectures are built on the commoditization of x86 servers, switches and the availability of powerful open virtualization solutions. Egi et al. [EGH+08] identify principles for constructing high-performance software router systems on commodity hardware, and show that the solutions based on current and near-future commodity hardware are flexible, practical, and inexpensive. PacketShader [HJPM11] is a software router framework with Graphics Processing Unit (GPU) acceleration that brings significantly higher throughput over previous CPU-only implementation. ServerSwitch [LGL+11] integrats a powerful multi-core commodity server with a programmable switching chip. This design eliminate CPU overhead and processing latency, while also supporting programmability. These work show that the commoditization of network hardware and the potential to rewrite their control software. Our work can benefit from the above research by using the enhanced NFV servers embodying the design ideas of the proposed hardware architectures.

2.2 Software Defined Networking

2.2.1 SDN Architecture

Software Defined Networking (SDN) is an important networking architecture as shown in Fig. 2.2. It separates the control plane from the data plane (forwarding plane) [sdnb]. It follows in the spirit of efforts showing the benefits of centralization in routing, access control, and monitoring [GHM+05, CCF+05, MAB+08, CFP+07].

Control plane (a logically centralized controller) has a global view of networks, takes requests from the application layer and manages the network devices (data

plane) via standard protocols. Data plane just forwards packets based on decisions of the control plane. The benefits of SDN are dynamically traffic steering, greater agility and implementing network automation. There are several standard com-

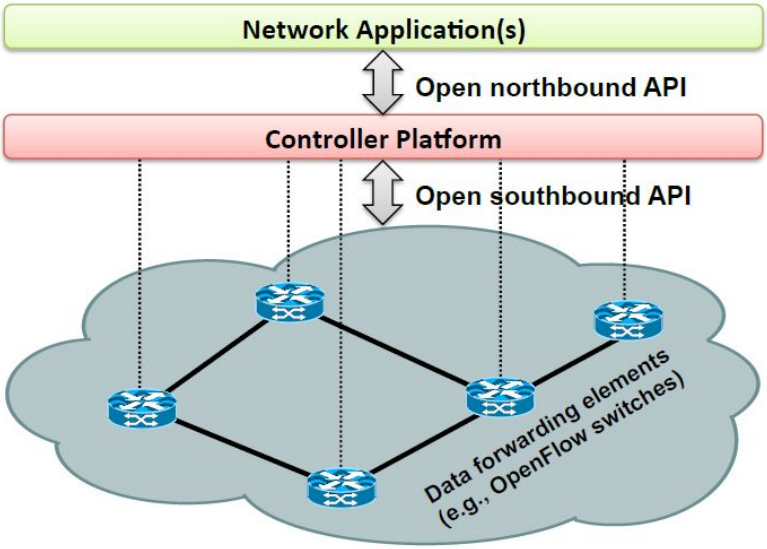


Figure 2.2: SDN Architecture.

munication protocols are defined between the control and data plane. OpenFlow protocol [opeb, opea, MAB+08] is the one earliest defined. In our work, the testbed is OpenFlow enabled. An OpenFlow switch has one or more flow tables. Each flow entry (a rule) contains a set of packet fields to match, and an action. Once some traffic matches a flow entry, the associated actions (drop, forward, modify, etc.) will be performed on the traffic. Depending on the flow entry enforced by a controller, an OpenFlow switch can behave like a router, switch or firewall.

2.2.2 SDN Switch Memory Management

To efficiently manage the expensive and power-hungry ternary content-addressable memory (TCAM) in switches, multiple works [ZLWZ10, GHM16, RHC+15] have studied shrinking the routing table size by aggregating rules. Uzmi et al. [UNT+11]

further propose a practical and near-optimal aggregation scheme to minimize the switch table size. Katta et al. propose the CacheFlow system [KARW14] for SDN to cache the most popular rules in a small TCAM, while relying on software to handle the cache miss traffic. Kang et al. [KLRW13] propose a rule placement algorithm to distribute forwarding policies across general SDN networks while managing rule-space constraints. Our work relates to the above ones by also considering the limit of middleboxes that can be hosted at a node due to resource constraints, such as switch TCAM or NFV server memory.

2.3 SDN based Middlebox Policy Enforcement

In traditional networks, middleboxes are installed at chokepoints and the network operators rely on error-prone and complex low-level configuration to steer traffic through a chain of middleboxes. SDN offers a promising alternative for middlebox policy enforcement by programmatically configuring forwarding rules. Sridhar [Rao14] presented a thorough study of SDN and how SDN technology can complement the network virtualization and network functions virtualization.

SIMPLE [Z. 13] presents a SDN-based policy enforcement layer for efficient middlebox-specific "traffic steering", which is built on SDN and existing legacy middleboxes. It can also easily fit into the context with software middleboxes running on commodity hardware, which can be instantiated in various locations dynamically. It enables the network managers and operators to specify a high-level abstractions of logical middlebox routing policy, and it then further automatically translates the policy into control rules with the knowledge of the physical network topology, forwarding device capacities, and resource constraints of the whole networks. Without mandating any placement or implementation constraints on middleboxes and

changing current SDN standards, SYMPLE offers efficient SDN-style control for middlebox-specific traffic steering, which is more modest compared to ongoing and parallel work developing new visions for SDN or middleboxes. SIMPLE focuses on balancing the middlebox load, but we focus on balancing the link load.

StEERING [ZBB+13a] presented a scalable framework for dynamically routing traffic through any sequence of middleboxes. Built on top of SDN, StEERING can support efficient forwarding at the granularity of subscribers and applications. The authors further propose an algorithm to select the best locations for placing services, such that the the performance is optimized.

The dynamic, traffic-dependent, and hidden actions of middleboxes make it difficult to reason about the correctness of network-wide policy enforcement, analysis and troubleshoot networks. To address this issue, FlowTag [FSYM13] further add tags to trace outgoing packets, and deal with the dynamic changes imposed by middleboxes. It is a complement for SDN based service chaining approaches. SDN controllers are able to configure the operations of tag generation and consumption by the FlowTags APIs. This approach requires minimal extensions from middleboxes vendors and demands no new capabilities from switch vendors.

Our work differs from the above ones by not only implementing the correct policies through SDN, but also considering the middlebox traffic changing effects and different dependency relations.

2.4 NFV VM Deployment

Deployment of NFV VMs is a challenge, especially under resource constraints, and solutions have been proposed for different objectives, such as minimizing the operation cost [V. 17] or the number of instances [Y. 17]. Furthermore, NFV VM

deployment is jointly considered with flow routing to optimize the overall network performance. Kuo et al. [T. 16] study the joint problem of VM deployment and path selection by considering the correlation between the link and server usages. Luizelli et al. [M. 17b] model virtual function deployment and chaining as an optimization problem, and propose a fix-and-optimize based heuristic solution. Zhang et al. [Q. 17] present a hierarchical two-phase solution for joint optimization of deploying chained functions and scheduling requests. Dwaraki and Wolf [DW16] present a method of solving the node-constrained service chain routing problem by transforming the network representation to a layered graph.

2.5 Network Service Chaining

Network service Chaining refers to an ordered sequence of network functions that a specific flow must go through [sfc]. Specifically, a chain defines the required processing or functions and the corresponding order that should be applied to the data flow. These chains require integration of service policy and the above applications to achieve optimal resource utilization. Traditional network service functions include, e.g., firewalls, TCP optimizers, web proxies, or higher layer applications [BRL+14]. These network services are usually deployed manually as hardware appliances that are physically integrated in the network by cables. The traditional approach is tedious, error prone and clumsy.

SDN can steer traffic dynamically based on user requirements [JPA+13, ZBB+13b, SGP+15]. However, hardware-based middleboxes limit the benefit of SDN due to their fixed functionalities and deployment. NFV is a good enabler for SDN. With the ability of dynamic function provisioning offered by NFV and the centralized con-

trol of SDN, new opportunities emerge in service chaining. Better performance and resource utilization can be achieved with the software-defined NFV architecture.

Even in NFV networks, it is still recognized as a challenge to implement a chain of network functions (services) under certain dependency constraints. Multiple solutions [MDT14, S. 14, Y. 16] have been proposed to implement service function chains, mostly using a linear programming based approach. In particular, Moens and Turck propose an Integer Linear Program model named VNF-P [MDT14] that allocates resources for service chains in an NFV network. Mehraghdam et al. formalize the chaining of network functions using a context-free language, and allocate resources by solving a Mixed Integer Quadratically Constrained Program (MIQCP) [S. 14]. Mehraghdam, Dräxler, and Karl also present a YANG model [MK16, DK17] for flexible specification of complex service structures. Li et al. propose the NFV-RT [Y. 16] resource provisioning system in which a linear programming approach is developed to maximize the number of requests for each service chain. Different from the above works, the formal model defined in our work considers not only a service function chain, but also a more general partial order between services and the traffic changing effects of services. Furthermore, this project presents realistic algorithms and prototype implementation besides formal models.

CHAPTER 3

TRAFFIC AWARE NFV MIDDLEBOX PLACEMENT

Network Function Virtualization (NFV) enables flexible deployment of middleboxes as virtual machines (VMs) running on general hardware. Since different middleboxes may change the volume of processed traffic in different ways, improper deployment of NFV middleboxes will result in hot spots and congestion. In this chapter, we study the traffic changing effects of middleboxes, and propose Software-Defined Networking (SDN) based middlebox placement solutions to achieve optimal load balancing. We formulate the Traffic Aware Middlebox Placement (TAMP) problem as a graph optimization problem with the objective to minimize the maximum link load ratio. First, we solve the TAMP problem when the flow paths are predetermined, such as the case in a tree. For a single flow, we propose the Least-First-Greatest-Last (LFGL) rule and prove its optimality; for multiple flows, we first show the NP-hardness of the problem, and then propose an efficient heuristic. Next, for the general TAMP problem without predetermined flow paths, we prove that it is NP-hard even for a single flow, and propose the LFGL based MinMax routing algorithm by integrating LFGL with MinMax routing. We use a joint emulation and simulation approach to evaluate the proposed solutions, and present extensive experimental and simulation results to demonstrate the effectiveness of our design.

3.1 Introduction

The advancement of virtualization technology [BBE+13] has made NFV [HGJL15] a promising architecture for middleboxes. Middleboxes are traffic processing appliances that are widely deployed in data centers, enterprise networks, and telecommunications networks [DGK+13]. Traditional middleboxes are proprietary hard-

ware devices that implement specialized functions such as firewalls, VPN proxies, and WAN optimizers [GMPR15, CKP15]. Such hardware middleboxes suffer from a number of drawbacks [MSG+16, WRH+15], including high cost, short life time, function inflexibility, and difficulty to scale up.

NFV decouples network functions from physical equipment, and implements middleboxes by running network function software on virtualized general hardware [BRXM15]. An NFV server is an industry standard server that hosts multiple virtual machines (VMs), each implementing a middlebox function with specialized software programs. Software middleboxes can be instantiated at, or moved to, general servers at various locations of the network, without the need to install new hardware. Benefiting from the underlying virtualization technology, NFV enjoys many advantages not available in traditional hardware middleboxes [etsb], such as fast deployment, reduced energy consumption, and real-time optimization.

Unlike switches or routers that are only forwarding traffic, most middleboxes are traffic processing devices, and may change the volume of processed traffic and may do it in different ways. For example, the Citrix CloudBridge WAN optimizer [wan] may compress traffic to 20% of its original volume before sends it to the next hop. On the other hand, a Stateless Transport Tunneling (STT) proxy [stt] adds 76 bytes to each processed packet due to the encapsulation overhead. Finally, a firewall will keep the traffic rates of allowed flows unchanged and reduce the rates of denied flows to zero.

The following toy example in Fig. 3.1 illustrates the traffic changing effects of middleboxes. Consider a network consisting of three nodes v_1, v_2 and v_3, and two links (v_1, v_2) and (v_2, v_3). Each node has an attached NFV server, and each server can host a single middlebox. A flow f starts at v_1 and ends at v_3, whose initial traffic rate is 1. Two middleboxes m_1 and m_2 need to be applied to f. m_1 will

double the traffic rate, while m_2 will cut the traffic rate in half. If install m_1 on v_1 and m_2 on v_3, the load of links (v_1, v_2) and (v_2, v_3) will be $1 \times 2 = 2$, as shown in Fig. 3.1(a). However, by installing m_1 on v_3 and m_2 on v_1, we can reduce the load of both links to $1 \times 0.5 = 0.5$, as shown in Fig. 3.1(b).

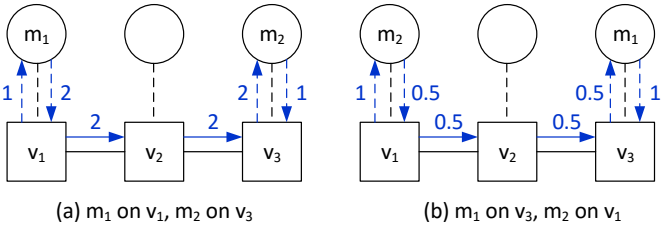


Figure 3.1: Traffic changing effects of middleboxes.

As can be seen, the flexibility of VMs brings a couple of challenges for efficient NFV implementation. First, since there may exist multiple candidate NFV servers, a strategic deployment plan is necessary to determine the optimal location for a middlebox. Next, due to traffic changing effects, the order to deploy different types of middleboxes is critical for balancing traffic load in the network.

In this work, we study optimal deployment of NFV middleboxes with the objective to achieve load balancing, and will focus on the persistent and large-sized elephant flows [A. 11] but not the transient and small-sized mice flows [A. 11], for the following three reasons. First, since elephant flows constitute a majority of the network traffic [MUK+04], optimizing elephant flows will efficiently help balance the entire network. Second, mice flows are transient, and may leave the network before the calculated optimization scheme takes effect, making it difficult to achieve the optimization objective. Third, the number of mice flows is much greater than that of elephant flows [A. 11], and the computation cost to manage so many dynamic flows is prohibitive. Therefore, our design is to deploy independent middleboxes for each elephant flow to avoid resource contention and congestion, but instead let mice

flows utilize the leftover processing capacity of middleboxes that have been deployed for elephant flows. The solution for mice flows will not be the focus of this chapter. The solution proposed in this work leverages the emerging SDN architecture [sdnb], which enables efficient optimization by decoupling the network control plane and data plane. An SDN based prototype has been implemented to demonstrate the practicality of our design.

Our main contributions are summarized as follows. First, we formulate the Traffic Aware Middlebox Deployment (TAMP) problem as a graph optimization problem with the objective to minimize the maximum link load ratio in the network. Second, when flow paths are predetermined, such as the case in the tree topology, we propose the Least-First-Greatest-Last (LFGL) rule to place middleboxes for a single flow, and prove its optimality. For multiple flows, we show that the TAMP problem is NP-hard by reduction from the 3-Satisfiability problem, and propose an efficient heuristic. Third, for the general scenario without predetermined flow paths, we prove that the TAMP problem is NP-hard even for a single flow by reduction from the Hamiltonian Cycle problem, and propose the LFGL based MinMax routing algorithm that integrates LFGL with MinMax routing. Fourth, we have implemented the proposed algorithms in a prototype with the open-source SDN controller Floodlight and network emulator Mininet. Finally, we present extensive experimental and simulation results to demonstrate the effectiveness of the proposed algorithms.

3.2 Problem Formulation

In this section, we formulate the Traffic-Aware Middlebox Placement (TAMP) problem.

Consider a network represented by a directed graph $G = (V, E)$. Each node $v \in V$ may have an attached NFV server, and its space capacity is denoted as $sc_u \geq 0$, i.e., the maximum number of middleboxes to host. For simplicity, we assume that each middlebox needs one space, and more processing power can be achieved by additional middlebox instances. A link $(u, v) \in E$ has a bandwidth capacity $bc_{u,v} \geq 0$, i.e., the available bandwidth. Its current link load is denoted as $l_{u,v}$.

Use M to denote the complete set of middlebox types. Each middlebox type $m \in M$ has an associate traffic changing factor $alter_m$, where $1 + alter_m$ is the ratio of the traffic rate of a flow before and after being processed by m.

Let F denote the set of flows. Each flow $f \in F$ is represented as a 4-tuple (s_f, d_f, t_f, M_f), in which $s_f \in V$ is the source node, $d_f \in V$ is the destination node, t_f is the initial traffic rate at the ingress point, and $M_f \subseteq M$ is the set of required middleboxes.

When a flow f enters the network, a path $route_f$ will be assigned for the flow, which is a decision variable defined as

$$route_f(u, v) = \begin{cases} 1, & \text{if flow } f \text{ traverses link } (u, v). \\ 0, & \text{otherwise.} \end{cases} \tag{3.1}$$

To avoid performance degradation for TCP flows, a flow is not allowed to be split among multiple paths [B. 10].

Use t_f^{v-} and t_f^{v+} to denote the traffic rate of flow f before entering and after leaving node v, respectively. If f is processed by a middlebox of type m, or middlebox m for short, at v, then $t_f^{v+} = t_f^{v-}(1 + alter_m)$. Note that $t_f^{s_f-} = t_f$. For convenience, use $t_f^{u,v} = t_f^{u+} = t_f^{v-}$ to represent the traffic rate of f on link (u, v).

In addition, a placement scheme $place_f$ will determine the locations to install each middlebox $m \in M_f$, which is a decision variable defined as

$$place_f(m,v) = \begin{cases} 1, & \text{if middlebox } m \text{ is installed at node } v. \\ 0, & \text{otherwise.} \end{cases} \tag{3.2}$$

Define the ratio between the aggregate load and capacity of a link to be the link load ratio, which is also called traffic intensity in queuing theory [Jai90] and determines the queuing delay. To achieve load balancing, our objective is to minimize the maximum link load ratio in the network by optimizing $route_f$ and $place_f$ for each flow $f \in F$, as shown Equation (3.3). Our solutions can also easily adapt to other optimization objectives [BSL06, GK06], such as minimizing the path cost or maximizing the residual capacity.

Our solutions can also easily adapt to other optimization objectives [BSL06, GK06], such as minimizing the path cost or maximizing the residual capacity.

$$\textbf{minimize } maxRatio \tag{3.3}$$

subject to the following constraints:

$$\forall (u,v) \in E : \\ \frac{l_{u,v} + \sum_{f \in F} route_f(u,v) t_f^{u,v}}{bc_{u,v}} \leq maxRatio \tag{3.4}$$

$$\forall f \in F : \\ \sum_{u \in V} route_f(s_f, u) = \sum_{u \in V} route_f(u, s_f) + 1, \tag{3.5}$$

$$\sum_{u \in V} route_f(u, d_f) = \sum_{u \in V} route_f(d_f, u) + 1 \tag{3.6}$$

$$\forall f \in F, \forall v \in V - \{s_f, d_f\} : \\ \sum_{u \in V} route_f(u,v) = \sum_{u \in V} route_f(v,u) \tag{3.7}$$

$$\forall v \in V : \\ \sum_{f \in F} \sum_{m \in M_f} place_f(m,v) \leq sc_v \tag{3.8}$$

$$\forall f \in F, \forall m \in M_f, \forall v \in V :$$

$$place_f(m, v) \leq \sum_{u \in V} (route_f(u, v) + route_f(v, u)) \tag{3.9}$$

$$\forall f \in F, \forall m \in M_f :$$

$$\sum_{u \in V} place_f(m, u) = 1 \tag{3.10}$$

$$\forall f \in F, \forall (u, v) \in E :$$

$$t_f^{u,v} =$$

$$t_f^{u-} route_f(u, v) \prod_{m \in M_f} (1 + place_f(m, u) alter_m) \tag{3.11}$$

Equation (3.4) states that, for a link (u, v), its load ratio $(l_{u,v} + \sum_{f \in F} route_f(u, v)$ $t_f^{u,v})/bc_{u,v}$ should be less than or equal to the optimization objective $maxRatio$. Equations (3.5) and (3.6) enforce each flow f to start and end at its source s_f and destination d_f, respectively. Equation (3.7) guarantees flow conservation at each intermediate node on the path, i.e., no flow generation or termination at an intermediate node. Equation (3.8) states that, for a node v, the total space demand of hosted middleboxes $\sum_{f \in F} \sum_{m \in M_f} place_f(m, v)$ should not exceed its space capacity sc_v. Equation (3.9) states that a middlebox m can be installed only on a node v that the flow path traverses. Equation (3.10) states that a middlebox m should be installed once and only once. Equation (3.11) states that, for a flow f, its traffic rate on a link (u, v) of its path, i.e., $route_f(u, v) = 1$, or its traffic rate when leaving u, or its traffic rate when entering v, is equal to its traffic rate when entering u, i.e., t_f^{u-}, multiplying the traffic changing ratios $1 + place_f(m, u) alter_m$ of all the middleboxes m placed at node u.

It can be seen that, optimal performance can be achieved by minimizing the maximum link load ratio on the routing paths of the flows in F, although the objective $maxRatio$ is the maximum link load ratio of the entire network. Proofs

are omitted. Thus, the following proposed solutions will focus on minimizing the maximum link load ratio on flow paths.

3.3 Traffic Aware Middlebox Placement with Predetermined Paths

In this section, we solve the TAMP problem when the flow paths, i.e., *route*, have been determined, or are unique in certain network topologies, such as the popular tree topology. We start with a single flow, i.e., $|F| = 1$, and propose the Least-First-Greatest-Last (LFGL) rule to achieve optimal performance. When there are multiple flows, i.e., $|F| > 1$, we prove that the TAMP problem is NP-hard by reduction from the 3-Satifiability problem, and propose an efficient heuristic.

3.3.1 Middlebox Placement for Single Flow

In reality, flows tend not to arrive at exactly the same time. Even if multiple flows arrive simultaneously in a software-defined network (SDN), the central controller will have to process them one by one. Thus, solutions for a single flow are of practical importance, especially for SDNs. Assume that the flow set F has only a single flow f, and the path $route_f$ has been determined. Obviously, a valid placement solution $place_f$ exists, if and only if the number of available NFV spaces on the routing path is greater than or equal to the number of required middleboxes, i.e., $|M_f|$.

The basic idea of our solution is to push heavy traffic out of the network core by decreasing the traffic rate at the beginning of the path, and increasing at the end of the path. Based on this observation, we propose an efficient rule called *Least-First-Greatest-Last* (LFGL) to optimally place middleboxes. The rule starts by sorting

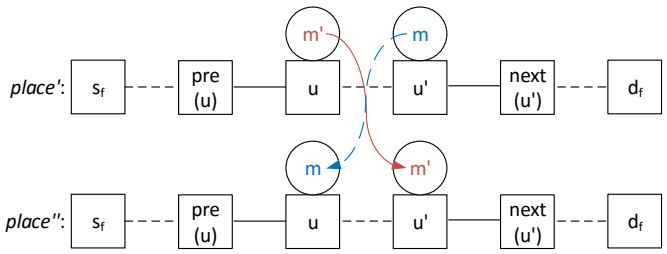


Figure 3.2: Proof of Theorem 1.

all the middleboxes $m \in M_f$ based on their traffic changing factors $alter_m$. It then places the middleboxes with non-positive factors, or shrinking middleboxes, from the head of the path in an increasing order. When a node has no space left, LFGL continues with the next node on the path. After finishing placing shrinking middleboxes, the rule switches to middleboxes with positive traffic changing factors, or expanding middleboxes, and place them from the path tail in the decreasing order of their factors. The deployment succeeds if all middleboxes are placed, and fails otherwise. As can be seen, LFGL processes each middlebox in M_f only once after sorting, and therefore its time complexity is $O(|M_f| \log |M_f|)$, i.e., the time complexity to sort M_f.

Theorem 3.3.1 *The Least-First-Greatest-Last rule minimizes the maximum link load ratio on the flow path.*

Proof. For the purpose of contradiction, assume that a different placement scheme $place'_f$ achieves maximum link load $maxRatio'$ lower than that of LFGL, i.e., $maxRatio'$ $< maxRatio$.

Without loss of generality, assume that the differences between the two placement schemes include shrinking middleboxes, and m is the one with the least traffic changing factor. By the LFGL rule, m is installed in the first available node u at its placement time, i.e., $place_f(m, u) = 1$. By comparison, the other placement scheme installed m on a different node u', i.e., $place'_f(m, u') = 1$, which must be

after u on the flow path, and instead a different middlebox m' is placed on u, i.e., $place'_f(m', u) = 1$. Apparently, the placement of m' is also different in $place_f$ and $place'_f$. Since among the differences between $place_f$ and $place'_f$, m has the least traffic changing factor, we know that $alter_m \leq alter_{m'}$.

Next, as shown in Fig. 3.2, we create a new placement scheme $place''_f$ by switching the locations of m and m' in $place'_f$, i.e.,

$$place''_f(n, v) = \begin{cases} place'_f(n, v), & \text{if } n \neq m, n \neq m' \\ place'_f(m, u')(= 1), & \text{if } n = m, v = u \\ place'_f(m', u)(= 1), & \text{if } n = m', v = u' \end{cases} \tag{3.12}$$

Then, the maximum link load ratio $maxRatio''$ of the new placement $place''_f$ will be less than or equal to that of $place'_f$, i.e., $maxRatio'' \leq maxRatio'$. Denote the traffic rates of f on link $(v, w) \in E$ under $place'_f$ and $place''_f$ as $t'^{v,w}_f$ and $t''^{v,w}_f$, respectively. Analyze the following three types of links.

1. For a link (v, w) between s_f and u, since the middleboxes placed before u are the same under $place'_f$ and $place''_f$, the traffic rates of f on such a link are also the same under both schemes, i.e., $t''^{v,w}_f = t'^{v,w}_f$.

2. For a link (v, w) between u and $next_f(u')$, as the locations of m and m' are exchanged in $place''_f$, $t''_f(v, w) = t'_f(v, w)(1 + alter_m)/(1 + alter_{m'})$. Since $alter_m \leq alter_{m'}$ as shown above, we know $t''^{v,w}_f \leq t'^{v,w}_f$.

3. For a link (v, w) between $next_f(u')$ and d_f, since the middleboxes placed after u' are the same under $place'_f$ and $place''_f$, the traffic rates of f on such a link are the same, i.e., $t''^{v,w}_f = t'^{v,w}_f$.

To sum up, for each link on the flow path of f, $place''_f$ achieves a lower or equal traffic rate for the flow than $place'_f$, resulting in a lower or equal maximum link load ratio, and it has one less difference with $place_f$ generated by LFGL. Continuing

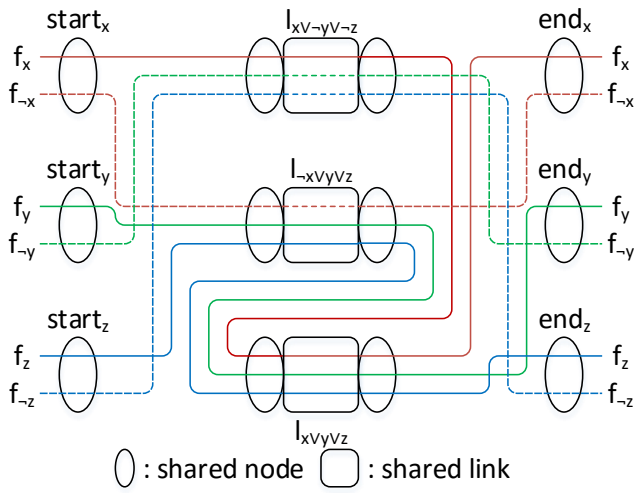


Figure 3.3: Reduction from 3-Satisfiability to TAMP for multiple flows with predetermined paths.

this process and eliminating all the differences between $place'_f$ and $place_f$, it can be shown by induction that $place_f$ achieves no higher maximum link load ratio than that of $place'_f$, i.e., $maxRatio \leq maxRatio'$, which contradicts the assumption. □

3.3.2 Middlebox Placement for Multiple Flows

We now solve the problem to place middleboxes for multiple flows with predetermined paths. As explained in the introduction, we do not let different elephant flows share the same middlebox to avoid hot spots, but instead the leftover processing capacity of installed middleboxes will be utilized by mice flows.

Theorem 3.3.2 *The Traffic Aware Middlebox Placement placement problem for multiple flows with predetermined paths is NP-hard.*

Proof. We prove by reduction from the 3-Satisfiability problem. The 3-Satisfiability problem decides whether a boolean formula in 3-CNF, i.e. the conjunction normal form with three boolean variables per clause, is satisfiable. An example is $(x \vee \neg y \vee \neg z) \wedge (\neg x \vee y \vee z) \wedge (x \vee y \vee z)$.

The reduction process is as follows. For a pair of boolean variables x and $\neg x$, we create two corresponding flows f_x and $f_{\neg x}$, and two nodes $start_x$ and end_x, each with one available middelbox space. Each flow f has an initial traffic rate of $t_f = 1$, and needs a single middlebox $M_f = \{m\}$ with $alter_m = -1 + \epsilon$, where ϵ is a small positive quantity less than one. In other words, the middlebox m will change the traffic rate of the processed flow to $1 \times (1 + alter_m) = \epsilon$. The two flows f_x and $f_{\neg x}$ both start at the shared node $start_x$, i.e., $s_{f_x} = s_{f_{\neg x}} = start_x$, and end at end_x, i.e., $d_{f_x} = d_{f_{\neg x}} = end_x$. For each clause $C = x_1 \vee x_2 \vee x_3$, we create a shared link l_C with a bandwidth capacity of 10, which will be traversed by the three flows corresponding to x_1, x_2, and x_3. When a boolean variable is included in multiple clauses, its corresponding flow will traverse multiple links one by one. Except the shared links, different flows have separate links for the remaining sections of their path. The reduction result for the above example CNF formula is illustrated in Fig. 3.3. It can be seen that the reduction can be done in polynomial time.

Next, we show that if a 3-CNF formula has a satisfiable assignment, then the constructed TAMP problem has a maximum link load ratio of no more than $(2 + \epsilon)/10$, i.e., $maxRatio \leq (2 + \epsilon)/10$. Given a satisfiable assignment, if a variable x (or $\neg x$) is assigned the true value, we let the corresponding flow f_x (or $f_{\neg x}$) place its middlebox on $start_x$, and the negation flow $f_{\neg x}$ (or f_x) on end_x. Note that a 3-CNF formula has a satisfiable assignment if and only if each clause C has at least one variable x assigned the true value, whose corresponding flow f_x will put its middlebox on node $start_x$. Therefore, when f_x arrives at the shared link l_C, its traffic rate is ϵ, and thus the load of l_C is at most $2 + \epsilon$. Since each shared link has a load of no more than $2 + \epsilon$, and any other link has a load of no more than 1, the maximum link load ratio of the entire network is no more than $(2 + \epsilon)/10$.

On the other hand, if the constructed TAMP problem instance has a maximum

link load ratio of no more than $(2 + \epsilon)/10$, it indicates that each shared link l_C has at least a flow f_x with its traffic rate being less than one, which means that its middlebox is placed at $start_x$. For each such flow f_x, by assigning the corresponding boolean value x a true value, we obtain a satisfying assignment for the 3-CNF formula. □

The hardness of TAMP for multiple flows with predetermined paths lies in the factorial number of possible sequences to process the multiple flows. Since different flows are competing for middlebox spaces, flows processed earlier may consume spaces and make them unavailable for flows processed later. We did not find an efficient way to simultaneously process multiple flows except for very limited scenarios with special topologies and traffic changing factors. Alternatively, we will present below a practical heuristic.

The basic idea of the heuristic is to place middleboxes for multiple flows by first processing each individual flow using the LFGL rule and then optimizing middlebox placement between flow pairs. The optimization of multiple flow middlebox placement extends the idea from a single flow to multiple flows. When multiple flows share a common sub-path, we apply the LFGL rule to the middleboxes of those flows on the common path, by placing the middlebox that decreases the maximum amount of traffic at the head of the sub-path, and the middlebox that increases the maximum amount of traffic at the end. However, one difference with LFGL for a single flow is that, since different flows may have different initial traffic rates, we need to consider the amount of traffic change caused by each middlebox, i.e., the product of the traffic rate before entering the middlebox and its traffic changing factor, instead of just the traffic changing factor as in the case for a single flow. Fortunately, by Theorem 3.3.1, the middleboxes of a single flow should still be placed in the increasing order of their traffic changing factors, so we know the traffic rate of a

flow before entering a middlebox, and thus can obtain the traffic change amount by multiplying its traffic changing factor. For example, if a middlebox of flow f has the i-th least traffic changing factor (ties broken arbitrarily), i.e., $M_f[i]$ in the sorted list, then the flow rate before entering the middlebox is $t_f \prod_{x=1}^{i-1}(1 + alter_{M_f[x]})$, and the its traffic change amount is $\delta_{M_f[i]} = alter_{M_f[i]} t_f \prod_{x=1}^{i-1}(1 + alter_{M_f[x]})$. The

Algorithm 1 Middlebox Placement for Multiple Flows with Predetermined Paths

Require: $G, F, route$
Ensure: *place*
1: sort $f \in F$ in decreasing order of initial traffic rate t_f
2: **for** each flow f in F **do**
3: apply LFGL to place middleboxes $m \in M_f$ for f
4: calculate traffic change amount δ_m for each $m \in M_f$
5: **end for**
6: **for** each pair of flows f and f' **do**
7: P = set of common sub-paths between f and f'
8: **for** each common sub-path $u \sim> v \in P$ **do**
9: $M[1..n]$ = extract all middleboxes of f and f' on common sub-path $u \sim> v$
10: sort $m \in M[1..n]$ in increasing order of traffic change amount δ_m and insert back in order
11: **end for**
12: **end for**

pseudo code to place middleboxes for multiple flows with predetermined paths is shown in Algorithm 1. Brief explanation is as follows. Line 1 sorts all the flows in the decreasing order of their initial traffic rates, inspired by the First-Fit Decreasing Bin Packing algorithm [CLRS09] to give priority to large flows. Lines 2 to 5 apply LFGL to each flow, and calculate the traffic change amount of each middlebox of the flow. Line 6 prepares a pair of flows for optimization. Line 7 finds the common sub-paths of the two flows, which may be multiple. Line 8 processes one of such common sub-paths, and line 9 extracts all the middleboxes of the two flows on the common sub-path. Line 10 sorts the extracted middleboxes based on their traffic changing amounts and insert them back to the nodes on the sub-path.

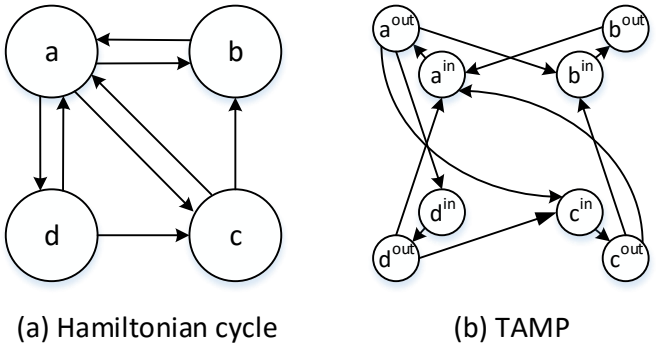


Figure 3.4: Reduction from Hamiltonian cycle to TAMP.

3.4 Traffic Aware Middlebox Placement without Predetermined Paths

In this section, we study the general TAMP problem where the flow paths *route* are not predetermined. We first show that the general TAMP problem is NP-hard even for a single flow by reduction from the Hamiltonian cycle problem, and then propose an efficient heuristic by integrating LFGL and MinMax routing that minimizes the maximum link load on the flow path. We also discuss the processing of multiple flows without predetermined paths.

3.4.1 NP-Hardness Proof

When flow paths are not determined, the TAMP problem is NP-hard, even for a single flow.

Theorem 3.4.1 *The general Traffic-Aware Middlebox Placement problem is NP-hard.*

Proof. We prove by reduction from the Hamiltonian cycle problem, which determines for a directed graph $G = (V, E)$ whether there exists a simple cycle that

contains each vertex in V. Note that a Hamiltonian cycle must be a simple cycle without repeated nodes.

Given an instance of the Hamiltonian Cycle problem with a graph G , we construct an instance of the TAMP problem with a graph $G' = (V', E')$ as follows.

1. For each node $v \in V$, create two nodes $v^{in}, v^{out} \in V'$, where v^{in} has a space capacity of zero, i.e., $sc_{v^{in}} = 0$, and v^{out} of one, i.e., $sc_{v^{out}} = 1$. Connect the two nodes with an edge $(v^{in}, v^{out}) \in E'$, and set its bandwidth capacity to ten, i.e., $bc_{v^{in},v^{out}} = 10$.

2. For each edge $(u, v) \in E$, create an edge $(u^{out}, v^{in}) \in E'$, and set its bandwidth capacity to ten, i.e, $bc_{u^{out},v^{in}} = 10$. An example to create G' from G is shown in Fig. 3.4.

3. Create a flow f, which is the only flow in F, i.e., $F = \{f\}$. The flow source and destination are both s^{in}, i.e., $s_f = d_f = s^{in}$, where s is an arbitrary node in V. The initial traffic rate is one, i.e., $t_f = 1$. The number of required middleboxes of f is the same as the number of nodes in V, i.e., $|M_f| = |V|$, and each middlebox does not change the volume of processed traffic, i.e., $\forall m \in M_f, alter_m = 0$.

Clearly, the above reduction process can be done in polynomial time.

Next, we show that if G has a Hamiltonian cycle, then the TAMP instance with G' and F has a maximum link load ratio of no more than 0.1, i.e., $maxRatio \leq 0.1$. Given the Hamiltonian cycle of G we construct a similar path $route_f$ in G' as follows. Assuming that the Hamiltonian cycle in G starts with s, for each node v and edge (u, v) in the Hamiltonian cycle, add edge (v^{in}, v^{out}) and (u^{out}, v^{in}) to $route_f$, respectively. Since the Hamiltonian cycle traverses each node in G exactly once, and each node v in G maps to a pair of nodes v^{in} and v^{out} in G', $route_f$

traverses each node in G' exactly once as well. Thus, we can see that $route_f$ has $|V|$ available spaces on the path, sufficient to host all the required middleboxes in M_f. Further, $route_f$ traverses any link in G' at most once, resulting in a maximum link load ratio of 0.1, given that the traffic rate of f is always one.

Reversely, if the TAMP instance with G' and F has a solution $route_f$ and $place_f$ with a maximum link load ratio of 0.1, G will have a Hamiltonian cycle. Given $route_f$ in G', we construct a Hamiltonian cycle in G as follows. Starting with $s_f = s^{in}$, sequentially add the corresponding node $v \in V$ of each incoming node $v^{in} \in V'$ on $route_f$ to the cycle in G. Since $route_f$ traverses all outgoing nodes v^{out} to obtain sufficient middlebox spaces, the constructed cycle traverses all the nodes in G. Further, since the maximum link load ratio in G' is 0.1, $route_f$ traverses each link including (v^{in}, v^{out}) for any v at most once. Thus, the constructed cycle in G traverses each node exactly once, and is a Hamiltonian cycle. □

Corollary 3.4.2 *There is no polynomial-time approximation algorithm with an approximation ratio less than two for the general Traffic Aware Middlebox Placement problem unless P=NP.*

Proof. By contradiction, assume that there exists a polynomial-time approximation algorithm that achieves an approximation ratio of $C < 2$.

Consider an instance of the Hamiltonian Cycle problem with a graph $G = (V, E)$. Construct an instance of the TAMP problem with a graph $G' = (V', E')$ and a flow set F as in the proof of Theorem 3.4.1.

If G has a Hamiltonian cycle, then the constructed TAMP problem has an optimal solution with the maximum link load ratio of 0.1. Thus, the approximation algorithm should return a solution with the maximum link load ratio less than or equal to $0.1 \times C = C < 0.2$.

Otherwise, if G does not have a Hamiltonian cycle, the constructed TAMP problem either does not have a solution or has a solution with the maximum link load ratio of at least 0.2 due to multiple passes of a link.

To sum up, by simply checking whether the maximum link load ratio returned by the approximation algorithm is less than 0.2, we can determine whether the original Hamiltonian cycle problem has a solution within polynomial time, which is a contradiction to the NP-hardness of the Hamiltonian Cycle problem. □

The general TAMP problem actually resembles the NP-hard Traveling Salesman Path (TSP) problem [FS07], which is a generalized version of the Hamiltonian Cycle problem, because TSP allows the source and destination to be different, instead of being the same node, and further the TSP solution needs to minimize the path cost in addition to traversing each node in the network. However, while the set of nodes to traverse in TSP is known, i.e., all nodes, the set of nodes to traverse in TAMP may be a subset of nodes, and there are a combinatorial number of such subsets in the network. Furthermore, even the subset of nodes is known in TAMP, it is the harder non-metric version of TSP, because computer networks generally do not satisfy the triangle inequality [LBSB09]. Due to the hardness of the general TAMP problem, we will focus on design efficient and practical heuristics.

3.4.2 LFGL based MinMax Routing for Single Flow

Next, we propose the LFGL based MinMax routing algorithm to calculate the routing path and middlebox placement for a single flow f. The basic idea is to integrate the LFGL rule with the MinMax routing algorithm that minimizes the maximum link load ratio on the flow path.

Based on the LFGL rule, the algorithm also works in two stages. In the first stage, the algorithm traverses the network from the flow source s_f, and iteratively calculates the MinMax path to each node as in Dijkstra's algorithm. When the MinMax path to a node v is determined, the algorithm will attempt to place shrinking middleboxes on v until there is no more space, as if the node is on the selected final path. In addition, the relaxation process will be applied to update the MinMax paths to the neighbors of v. The search will follow multiple candidate paths, and thus the algorithm will attempt placing the same middlebox at different nodes with different candidate paths. The search along a candidate path will terminate if the last shrinking middlebox has been placed on a node, which we call a termination node. When there is no more candidate path to search, the algorithm switches to the second stage to process expanding middleboxes.

In the second stage, the algorithm traverses the network, in a similar way as in the first stage, but backward from the flow destination d_f, and places expanding middleboxes when the MinMax path to a node is found. Note that if all middleboxes are successfully placed, the departure traffic rate $t_f^{d_f+}$ at the destination d_f will be $t_f \prod_{m \in M_f}(1 + alter_m)$, which will be used as the "initial" traffic rate in the second stage. After all expanding middleboxes have been placed along a candidate path, the second stage continues searching the MinMax paths to the remaining nodes, until it reaches a termination node u of the first stage. This means that a path for flow f has been found with two sections: from s_f to u and from u to d_f. For this reason, we call u a junction node. Similar as the first stage, the second stage stops when there is no more path to search. After the second stage finishes, the algorithm collects all the junction nodes, each corresponding to a different path. The algorithm compares the maximum link load ratio of each path, and selects the path with the minimum maximum link load ratio.

Algorithm 2 Place Middlebox on Node

Require: sorted $M_f[1..n], u, start, flag$
Ensure: $place_f, index(u)$ or $index'(u), t_f^{u+}$ or t_f^{u-}

1: $i = start$
2: **if** $flag < 0$ **then**
3: **while** $sc_u > 0$ **and** $i \leq n$ **and** $alter_{M_f[i]} \leq 0$ **do**
4: $place_f(M_f[i], u) = 1; sc_u - -; i + +$
5: **end while**
6: $index(u) = i - 1$
7: $t_f^{u+} = t_f^{pre(u)+} \prod_{m \in M_f, place_f(m,u)=1} (1 + alter_m)$
8: **else**
9: **while** $sc_u > 0$ **and** $i \geq 1$ **and** $alter_{M_f[i]} > 0$ **do**
10: $place_f(M_f[i], u) = 1; sc_u - -; i - -$
11: **end while**
12: $index'(u) = i + 1$
13: $t_f^{u-} = t_f^{next(u)-} / \prod_{m \in M_f, place_f(m,u)=1} (1 + alter_m)$
14: **end if**

Algorithm 3 shows the pseudo code of the LFGL based MinMax routing algorithm, and Algorithm 2 shows the pseudo code of the *placeMiddlebox* function. The latter places shrinking ($flag = -1$) or expanding ($flag = 1$) middleboxes on node u from the *start*-th one of the sorted list. For easy description, we use $pred_f(u)$ and $next_f(u)$ to represent the preceding and succeeding node of u on the path of flow f, i.e., $route_f(pred_f(u), u) = 1$ and $route_f(u, next_f(u)) = 1$.

Brief explanation of Algorithm 3 is as follows. Line 1 sorts the middleboxes of flow f in the increasing order of their traffic changing factors. Lines 2 to 9 initialize the first stage, where Saw is the set of nodes whose MinMax paths from the source s_f have been determined. Line 2 also places shrinking middleboxes on the source s_f until there is no more space or no more shrinking middleboxes. In lines 3 to 9, for each neighbor u of the source s_f, if the link (s_f, u) has more available bandwidth than $t_f^{s_f+}$, then there is a candidate path from s_f to u. $mllr(u)$ records the maximum link load ratio of the candidate path till u. Lines 10 to 18 are the loop to find the

MinMax path to a node at a time. At the beginning of each loop, the node $u \notin Saw$ with the minimum maximum link load ratio $mllr(u)$ will be selected and added to Saw. Line 12 places shrinking middleboxes on u. Lines 13 to 17 apply the relaxation process, i.e., checking each neighbor v of u to see whether there is a new path to v via u with a lower maximum link load ratio, and update if yes. Lines 19 to 42 run the second stage in a similar manner, but starting from the flow destination d_f and placing expanding middleboxes. In lines 33 to 36, if the search reaches a node u till which all the shrinking and expanding middleboxes have been placed in the first and second stage, respectively, u will be added to the junction node set J. When the second stage finishes, lines 43 to 47 select from J the node u with the minimum maximum link load, and the final MinMax path can be constructed by tracing $pred_f(u)$ from u to s_f and $next_f(u)$ to d_f. Otherwise, if J is empty, the algorithm fails to find a path.

Since the LFGL based MinMax routing algorithm integrates the LFGL rule and MinMax routing, which is similar to Dijkstra's algorithm, its complexity is the product of the two, i.e., $O((|V| \log |V| + |E|) \times |M_f|)$.

3.4.3 Optimization of Multiple Flows

Since the general TAMP problem is NP-hard even for a single flow, the problem becomes more challenging for multiple flows. We propose a solution similar to that in Section 3.3.2, i.e., processing individual flows followed by optimizing middlebox placement between flow pairs. In detail, we first sort all the flows in the decreasing order of their initial traffic rates, to give priority to large flows. Then, we apply the LFGL based MinMax routing algorithm to calculate the routing path and middlebox locations for each individual flow. Finally, we check each pair of flows, identify their common sub-paths, and optimize by swapping middleboxes.

Algorithm 3 LFGL based MinMax Routing

Require: $G, f, M_f[1..n]$
Ensure: $route_f, place_f$
sort $m \in M_f[1..n]$ in increasing order of $alter_m$
Stage 1: $Saw = \{s_f\}; placeMiddlebox(s_f, M_f[1..n], 1, -1)$
for each neighbor u of s_f **do**
    if $bc_{s_f,u} - l_{s_f,u} \geq t_f^{s_f+}$ **then**
        $mllr(u) = (l_{s_f,u} + t_f)/bc_{s_f,u}; pred_f(u) = s_f$
    else
        $mllr(u) = \infty$
    end if
end for
while $\exists u \notin Saw, index(pred_f(u)) < n$ **and** $alter_{M_f[index(pred_f(u))+1]} \leq 0$ **do**
    select such u with min $mllr(u); Saw = Saw \cup \{u\}$
    $placeMiddlebox(u, M_f[1..n], index(pred_f(u)) + 1, -1)$
    for each neighbor v of u **do**
        if $bc_{u,v} - l_{u,v} \geq t_f^{u+}$ **and** $mllr(v) > \max(mllr(u), (l_{u,v} + t_f^{u+})/bc_{u,v})$ **then**
            $mllr(v) = \max(mllr(u), (l_{u,v} + t_f^{u+})/bc_{u,v}); pred_f(v) = u$
        end if
    end for
end while
Stage 2: $Saw' = \{d_f\}; placeMiddlebox(d_f, M_f[1..n], n, 1);\ J = \emptyset$
if $d_f \in Saw$ **and** $index(d_f) + 1 = index'(d_f)$ **then**
    $J = J \cup \{d_f\}; mllr(d_f) = \max(mllr(d_f), mllr'(d_f))$
end if
for each neighbor u of d_f **do**
    if $bc(u, d_f) \geq t_f^{d_f-}$ **then**
        $mllr'(u) = (l_{u,d_f} + t_f^{d_f-})/bc_{u,d_f}; next_f(u) = d_f$
    else
        $mllr'(u) = \infty$
    end if
end for
while $\exists u \notin Saw'$ **do**
    select such u with min $mll'[u]; Saw' = Saw' \cup \{u\}$
    $placeMiddlebox(u, M_f[1..n], index'(next_f(u)) - 1, 1)$
    if $u \in Saw$ **and** $index(u) + 1 = index'(u)$ **then**
        $J = J \cup \{u\}; mllr(u) = \max(mllr(u), mllr(u'))$
        continue
    end if
    for each neighbor v of u **do**

if $bc_{v,u} - l_{v,u} \geq t_f^{u-}$ **and** $mllr'(v) > \max(mllr'(u), (l_{v,u} + t_f^{u-})/bc_{v,u})$ **then**
    $mllr'(v) = \max(mllr'(u), (l_{v,u} + t_f^{u-})/bc_{v,u}); next_f(v) = u$
end if
end for
end while
if $J \neq \emptyset$ **then**
    select $u \in J$ with min $mllr(u)$; **exit** with success
else
    exit with failure
end if

3.5 Implementation

Due to its centralized control logic, the emerging Software Defined Networking (SDN) architecture is an ideal platform to implement the proposed algorithms. We have implemented a prototype system using the open-source SDN controller

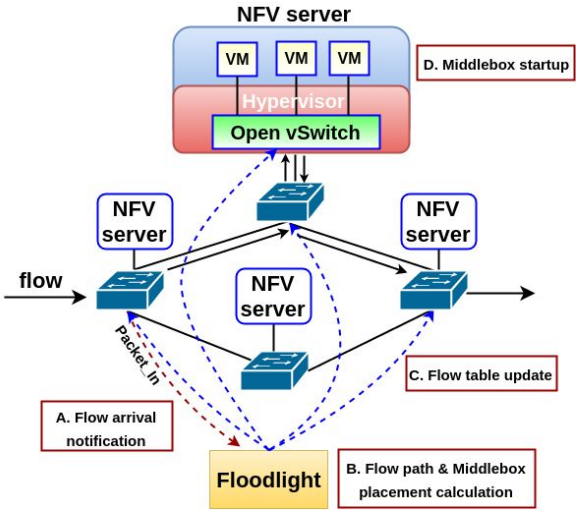


Figure 3.5: Middlebox placement workflow.

Floodlight [Flo] and network emulator Mininet [min] to demonstrate our design. In this section, we describe the prototype implementation and discuss real deployment issues.

As explained in Section 3.1, an SDN network is decoupled into the control plane and data plane. For the control plane, we use the modular Floodlight controller, and implement the proposed algorithms as a new module to calculate flow path and middlebox placement. For the data plane, we pick the Mininet network emulator. Mininet can conveniently create a network testbed of hosts and SDN-enabled switches, each as a virtual machine (VM). For each switch in our prototype, we also create an attached NFV server, and connect it with the switch via a high speed link.

Fig. 3.5 summarizes the workflow of our prototype to process a new incoming flow, and each step is explained in detail below.

3.5.1 Flow Arrival Notification

Elephant flows can be statically determined based on the application types, such as data backup or VM migration, or dynamically detected using existing techniques in the literature [A. 11, MUK+04]. When the ingress switch detects or learns an elephant flow, it tries to find a matching entry for the flow in its flow table. If there is no matching entry, the switch wraps a packet of the flow within an *OFPT_PACKET_IN* message, and sends it to the controller. Upon receiving the forwarded packet, the controller learns the arrival of the new flow, and will try to calculate a path where each link has a sufficient bandwidth capacity for the flow.

3.5.2 Flow Path and Middlebox Placement Calculation

Our module to calculate the flow path and middlebox placement is triggered by the *OFPT_PACKET_IN* message received by the controller. The module determines the application type based on the packet header information, such as IP addresses and port numbers, and determines the set of required middleboxes for the new flow

according to predefined profiles. Next, the module applies the proposed algorithms to calculate a flow path and middlebox placement locations.

3.5.3 Midllebox VMs Startup

Once the middlebox locations have been calculated, the controller will remotely wake up or start VMs on the selected NFV servers, which can be done through the communication between the VM control software and server hypervisor. For example, for the Kernel-based Virtual Machine (KVM) [kvm] hypervisor, the command line tool Virsh [vir] can be used to remotely start, shutdown, suspend, or resume VMs; for the VMware ESXi hypervisor [esx], the VMware vCenter server [vce] can be used to remotely control VMs. Predefined VM images for different middleboxes will thus be loaded to perform the desired network functions. Our Mininet prototype focuses on evaluating the network performance, and starts the middlebox VMs in advance.

3.5.4 Flow Table Update

After the controller obtains the flow path and middlebox locations, it will accordingly update the switch flow tables to ensure correct packet forwarding. A path is specified in Floodlight as a list of Node-Port tuples in the form of (*DatapathId*, *OFPort*), where *DatapathId* is the Datapath Identity of an OpenFlow instance on a switch and *OFPort* represents a port number of the instance. For the routing path, the controller sends an *OFPT_FLOW_MOD* message to each switch on the routing path. The messages contains the matching fields to define the flow, rule priority, and actions for the flow. In this case, the action is to forward packets of the flow to the calculated output port. On the other hand, for the middleboxes, the controller sends

two types of *OFPT_FLOW_MOD* messages to ensure placement locations. The first is to the associated switch, which tells the switch to forward packets to the attached NFV server. The second is to the Open vSwitch (OVS) running in the hypervisor of the NFV server, which instructs the hypervisor to send matching packets to one of the hosted middlebox VMs.

3.5.5 Middlebox Emulator Development

To evaluate the effects of middleboxes with different traffic changing factors, we have developed a middlebox emulator program using the *libpcap* library [lib]. While a normal TCP/UDP socket will only process packets destined to it, the emulator intercepts all packets forwarded by OVS in the hypervisor using the *libpcap* APIs. To emulate a shrinking middlebox m, the emulator discards intercepted packets with a probability of $-alter_m$. On the other hand, if m is an expanding middlebox, the emulator duplicates intercepted packets with a probability of $alter_m$. In this way, the emulator will accurately change the traffic volume as indicated by the traffic changing factor. After processing, the emulator continues to forward the packets to their actual destinations.

3.5.6 Link Load Monitoring

To obtain the link load information necessary for the LFGL based MinMax routing algorithm, we have developed a link load monitoring module in Floodlight. The module periodically sends the *OFPT_STATS_REQUEST* message to every switch to request traffic statistics. Upon receiving the request, a switch will send the *OFPT_STATS_REPLY* response, which contains the transmitted byte count for each port of the switch along with other statistics data. By collecting the information

from all switches, the module estimates the load of each link by calculating the exponentially weighted moving average.

3.5.7 Post-processing

After a flow finishes, the flow table entries will be automatically removed after idle or hard timeout [opeb] by the switch, and the middelbox VMs can be shut down or hibernated after a period of inactivity or manually by the controller.

3.6 Experiment and Simulation Results

We use a combination of experiments and simulations to evaluate the proposed algorithms. Experiments in the implemented prototype generate performance data in realistic environments, and simulations in the ns-3 simulator enable us to conduct evaluations in large scale networks with hundreds of hosts. In this section, we present extensive experiment and simulation data to demonstrate the effectiveness of our design.

3.6.1 Benchmark Solutions

Since there are no existing algorithms for the studied TAMP problem in the literature, we designed the following benchmark solutions. From the proof of Theorem 3.3.1, it can be seen that the middleboxes of a flow placed in the increasing order of their traffic changing factors always result in better performance, and thus all the benchmark solutions sort the middleboxes before placement.

In case that the flow paths have been predetermined, there are three benchmark solutions as follows.

- *First-fit:* The first-fit rule starts with the head of the flow path, and places the sorted middleboxes one by one, each at the first available NFV server.
- *Last-fit:* The last-fit rule starts from the end of the flow path, and places the middleboxes sorted in the decreasing order one by one, each at the first available NFV server from the tail, or the last from the head.
- *Random-fit:* The random-fit rule places the middleboxes at random available nodes along the path.

In case that the flow paths are not predetermined, four benchmark solutions are used in the simulations. Each first applies load-balanced (ECMP) shortest path routing to determine the flow path, and then uses LFGL, First-fit, Last-fit, or Random-fit to place middleboxes on the shortest path. Two benchmark solutions are used in the experiments. One is the optimal solution based on the formulation, the other is shortest path routing with LFGL. We did implement First-fit, Last-fit, and Random-fit in this part of the experiments because LFGL has shown superior performance in the previous experiments with predetermined paths.

3.6.2 Experiment Results with Prototype

In the prototype experiments, we use the following performance metrics to compare our design and benchmark solutions.

- *End-to-end delay:* The end-to-end delay is the delay between the time points that a packet leaves the source and arrives at the destination. Congestion will result in a longer end-to-end delay.
- *Maximum link load:* The maximum link load is the upper bound of the link load of all the links during the entire experiment run. It is a direct indicator of the congestion level.

For traffic generation, we run Iperf [ipe] on hosts to generate real constant bit rate (CBR) UDP traffic. We also patched Iperf to be able to measure the end-to-end delay.

Effectiveness of LFGL Rule: We first show the effectiveness of the LFGL rule for the TAMP problem with predetermined paths by comparing it with benchmark solutions.

We pick the tree topology, because it is a popular choice for institutional networks, and there is only a predetermined single path between any pair of nodes. We set up a four-layer binary tree with seven switches and eight hosts, as depicted in Fig. 3.6. Each link has 10 Mbps bandwidth. Each switch u has an attached NFV server with two middlebox spaces, i.e., $sc_u = 2$. We sequentially create four flows from $h1, h2, h3$, and $h4$ to $h5, h6, h7$, and $h8$, respectively. The initial traffic rate of each flow is adjusted from 0.5 Mbps to 4 Mbps with a stride of 0.5 Mbps. Each flow f requires two middleboxes $M_f = \{m_1, m_2\}$ with traffic changing factors of $alter_{m_1} = -0.2$ and $alter_{m_2} = 0.2$. As a result, the combined traffic changing effect of both middleboxes is $\prod_{m \in M_f}(1 + alter_m) = (1 - 0.2)(1 + 0.2) = 0.96$. When a flow starts, it has only one path, and the locations of its middleboxes will be calculated by the above rules.

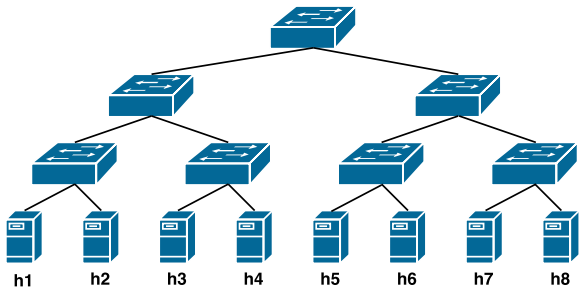
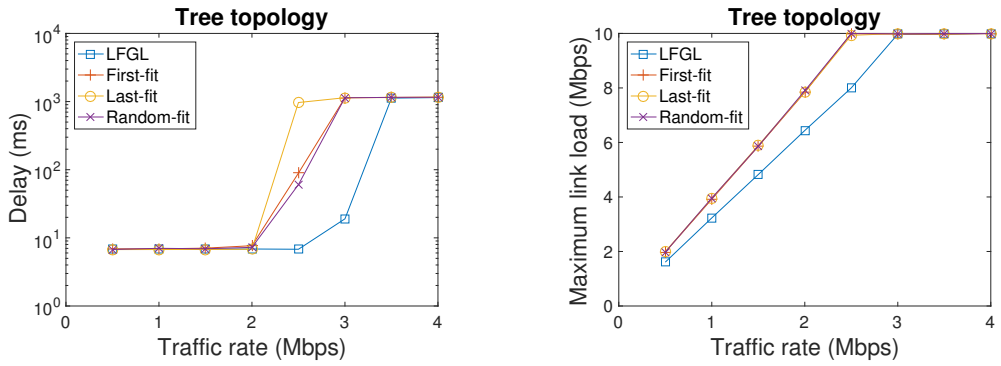


Figure 3.6: Tree topology for experiments.



(a) End-to-end delay. (b) Maximum link load.

Figure 3.7: LFGL experiment results.

Fig. 3.7(a) shows the average end-to-end delays (in logarithm) of LFGL, First-fit, Last-fit, and Random-fit. We can see that with its optimal placement strategy, LFGL consistently beats the other three rules with shorter delay and postponed congestion. In detail, initially, when the flow rate is small and there is no congestion, all the rules have a small constant delay at about 7 ms. When the flow rate increases to 2.5 Mbps, the delay of LFGL keeps stable, but that of other rules starts increasing due to congestion. This can be explained by the fact that the root is the bottleneck of a tree, and $2.5 = 10/4$ Mbps is the bandwidth available at the root for each of the four generated flows. Specifically, Last-fit has the longest delay at 968 ms, since it puts all middleboxes at the end of flow path, and therefore the flow rate is $1\times$ in most traversed links. On the other hand, First-fit has shorter delay at 89 ms, because it puts middleboxes at the beginning, and thus the flow rate is $0.96\times$ in most traversed links. Using a random strategy, random-fit achieves a short delay of 60 ms. Further, even when the flow rate increases to 3 Mbps, LFGL has a moderate delay of 20 ms. Finally, when the flow rate increases to 3.5 Mbps, all rules have a saturated network with an end-to-end delay of about 1150 ms.

Fig. 3.7(b) shows the maximum link load ratios of the four rules. Again, LFGL consistently achieves the best performance among the benchmark solutions. For

First-fit, Last-fit, and Random fit, their maximum link load ratio is approximately 4 times of the ratio between the flow rate and link capacity, and reaches one when the flow rate is 2.5 Mbps. This is consistent with the observation in Fig. 3.7(a) that congestion happens at 2.5 Mbps for the three rules. On the other hand, the maximum link load ratio of LFGL is approximately 3.2 times of the ratio between the flow rate and link capacity, and reaches one when the flow rate is 3 Mbps. The reason is that LFGL decreases flow rates as early as possible and increase as late as possible, and thus minimizes the traffic volumes in the network core.

Effectiveness of LFGL based MinMax Routing: Next, we demonstrate the effectiveness of the LFGL based MinMax routing algorithm. We pick the butterfly

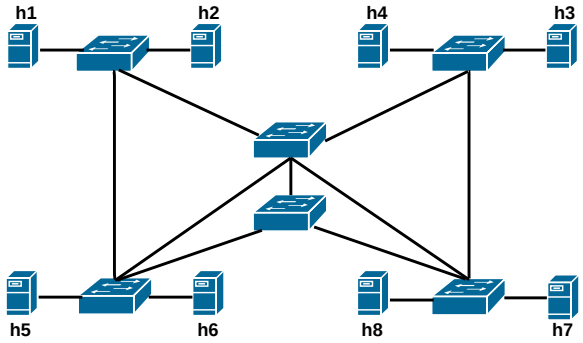
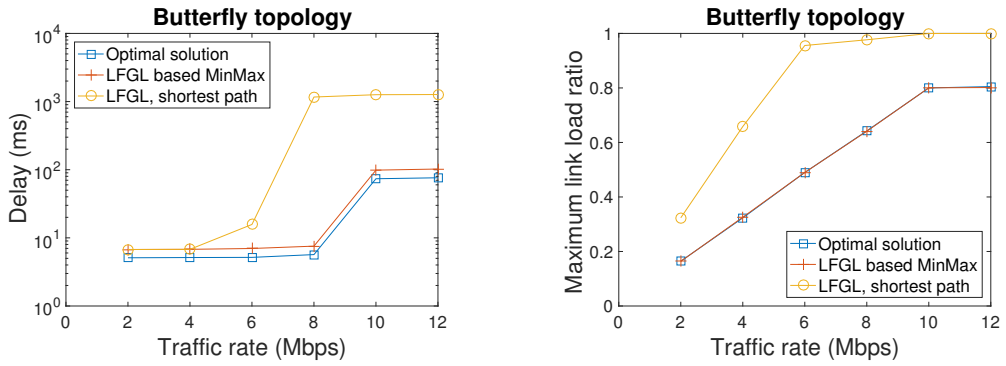


Figure 3.8: Multipath topology for experiments.

topology with multiple available paths as depicted in Fig. 3.8. The network contains six switches and eight hosts. Each switch has two middlebox spaces, and each link has 10 Mbps bandwidth. Four flows are sequentially started from h_1, h_2, h_3, and h_4 to h_7, h_8, h_5, and h_6, respectively. Each flow needs two middleboxes with traffic changing factors of -0.2 and $+0.2$, respectively. We adjust the flow traffic rate from 2 Mbps to 12 Mbps with a stride of 2 Mbps. Fig. 3.9(a) shows the average end-to-end delay. We can see that LFGL based MinMax routing outperforms LFGL with shortest path routing due to its traffic awareness in path selection, and achieves



(a) End-to-end delay. (b) Maximum link load.

Figure 3.9: LFGL based MinMax routing experiment results.

performance close to that of the optimal solution calculated from the formulation in Section 3.2. While LFGL based MinMax routing finds disjoint flows to minimize the maximum link load, LFGL with shortest path routing chooses the same shortest path for multiple flows, resulting in earlier congestions and longer delays. The optimal solution produces the shortest delay because it selects the path with the smallest maximum link load ratio as well as least number of hops. In detail, the delay of LFGL with shortest path routing is initially stable at about 7 ms, increases to 16 ms when the flow rate is 6 Mbps, and exceeds 1 second once the flow rate becomes 8 Mbps. On the other hand, the delays of LFGL based MinMax routing and the optimal solution are stable until the flow rate increases to 10 Mbps, and grow to about 102 ms and 96 ms, respectively, when the flow rate reaches 12 Mbps. Fig. 3.9(b) shows the maximum link load ratio. Although the optimal solution has shorter delays than LFGL based MinMax routing due to its shorter paths, the maximum link load ratio performance of the latter is on a par with that of the former thanks to traffic awareness in path selection. On the other hand, LFGL with shortest path routing saturates much earlier because of overlapping flow paths.

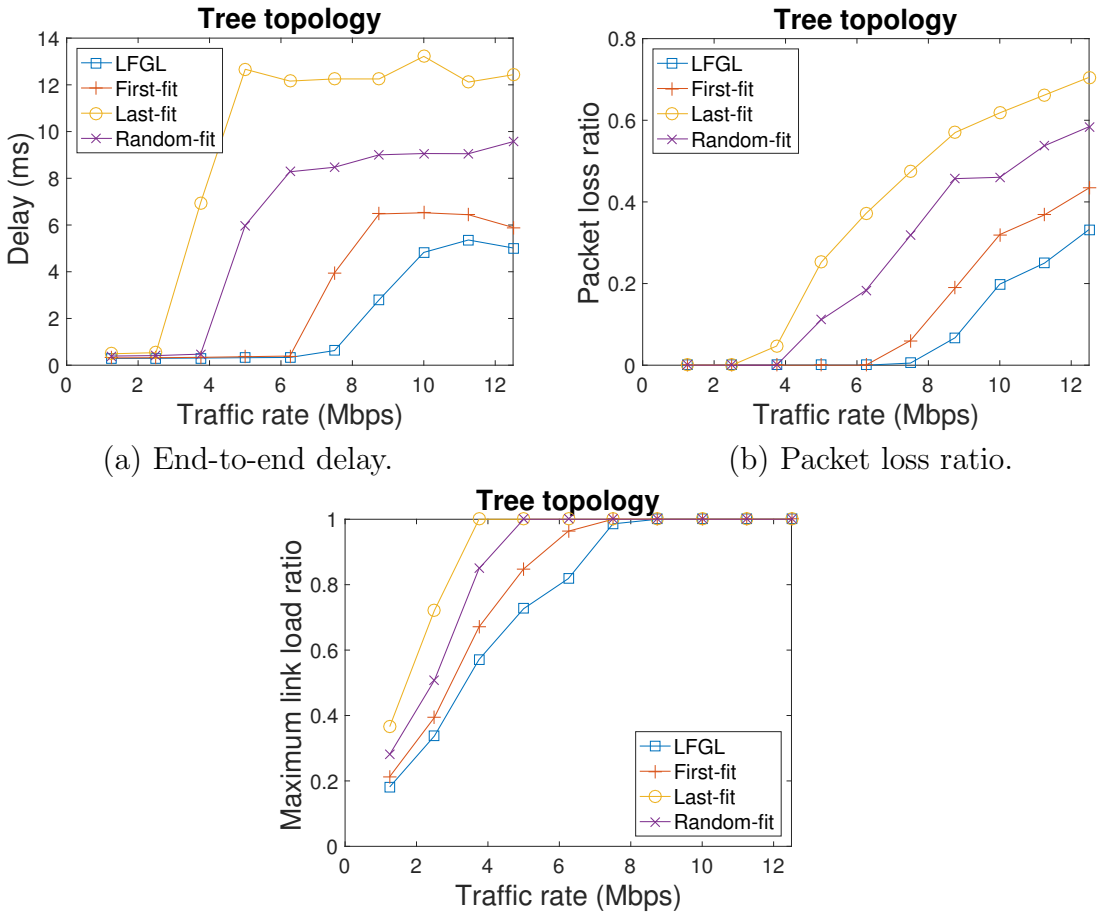
3.6.3 Simulation Results

In this subsection, we present simulation results in ns-3 to evaluate the proposed algorithms in large scale networks. To better reflect realistic traffic characteristics, instead of using CBR traffic as generated by Iperf, we use the ns-3 On-Off burst traffic model. A flow is in the Off state with no traffic for 50% of the time, and in the On state with continuous CBR traffic for the remaining 50% of the time. The traffic rate of a flow in the On state is the product of a baseline traffic rate and a random number between 0.5 to 1.5. Each flow f requires three middleboxes $M_f = \{m_1, m_2, m_3\}$ with the traffic changing factors of $alter_{m_1} = -0.5$, $alter_{m_2} = -0.2$, and $alter_{m_3} = 0.2$. Each link in the network has a bandwidth capacity of 100 Mbps and propagation delay of 2 ms. Each simulation run lasts 300 seconds, and the presented data are the average of four simulation runs.

In addition to the end-to-end delay and maximum link load, we also collect the following additional performance data.

- *Packet loss ratio:* The packet loss ratio is the ratio of the number of lost packets to the number of sent packets. Heavier congestion will result in a larger packet loss ratio.

Topology with Predetermined Paths: We first conduct simulations for topologies with predetermined paths. As in Section 3.6.2, we pick the tree topology, and set up a four-layer quad tree with 21 switches and 64 hosts. Each switch has 10 middlebox spaces. Each host generates a flow to a random destination, and we adjust the average traffic rate of each flow from 1.25 Mbps to 12.5 Mbps with a stride of 1.25 Mbps. Fig. 3.10(a) compares the average end-to-end delay of LFGL, First-fit, Last-fit, and Random-fit. We can observe a similar trend as in the prototype experiment data that LFGL consistently achieves the shortest delay due to its optimal



(a) End-to-end delay.

(b) Packet loss ratio.

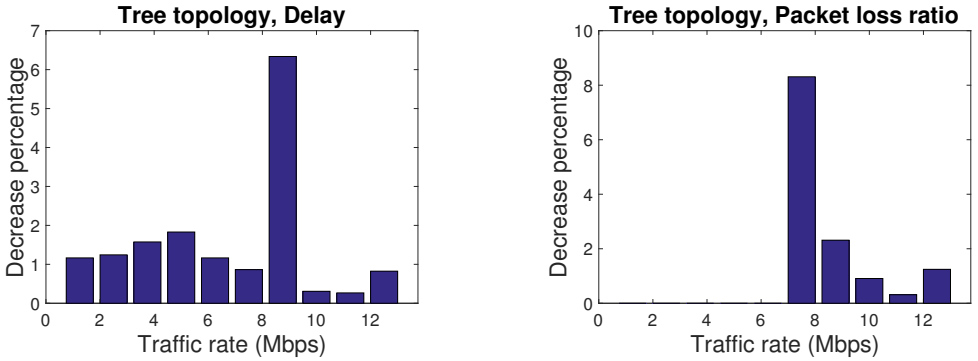
(c) Maximum link load ratio.

Figure 3.10: LFGL simulation results.

middlebox placement. By contrast, Last-fit has the longest delay, because it places middleboxes at the end of flow path, and the flow rate is $1\times$ in most links on the path. First-fit achieves shorter delay, since it places middleboxes at the beginning, and thus the flow rate is $(1-0.5)(1-0.2)(1+0.2) = 0.48\times$ in most links. Random-fit has a delay between that of First-fit and Last-fit with a random strategy.

Fig. 3.10(b) and (c) show the packet lost ratio and maximum link load ratio, respectively, of the four rules. The conclusion is consistent with that in Fig. 3.10(a) that, in the order of LFGL, First-fit, Random-fit, and Last-fit, each rule achieves a lower packet loss ratio as well as a lower maximum link load ratio than the subse-

quent ones.



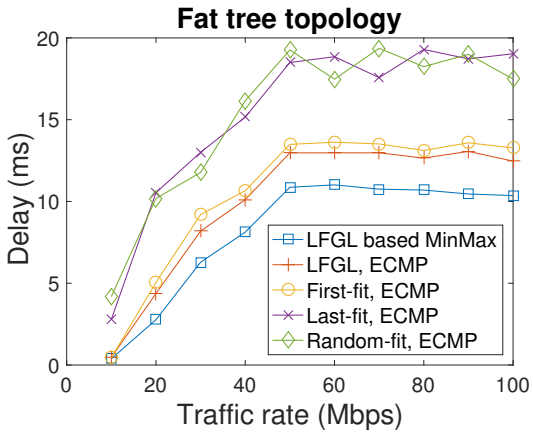
(a) End-to-end delay. (b) Packet loss ratio.

Figure 3.11: Improvement of multi-flow optimization with predetermined paths.

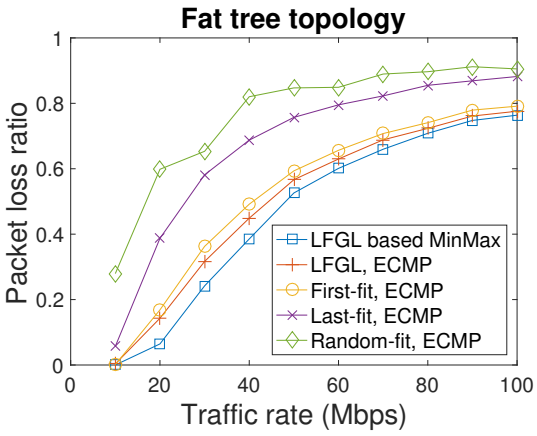
We have also evaluated the optimization algorithm for multiple flows presented in Algorithm 1. As comparison, LFGL processes the multiple flows individually in a random order. Fig. 3.11(a) and (b) show the improvements of the optimization algorithm on the end-to-end delay and packet loss ratio, respectively. We can see that it reduces the end-to-end delay and packet loss ratio by up to 6% and 8%, respectively. The improvement is not significant in some cases, because the optimization algorithm cannot find many middlebox pairs to swap. Note that when the flow rate is less than or equal to 6 Mbps, both solutions have a zero packet loss ratio, and thus there is no improvement. Since the maximum link load ratio measures the instantaneous worst case performance, we do not see significant improvements by the optimization algorithm.

Topology without Predetermined Paths: Next, we consider multi-path topologies, and set up a 8-pod fat tree [AFLV08] with 80 switches and 128 hosts. To utilize the multiple paths of the fat tree, we apply equal-cost multi-path (ECMP) [ecm] load balancing for the shortest-path routing algorithm, by randomly dispatching a flow to one of the available next hops. The traffic rate of each flow is adjusted

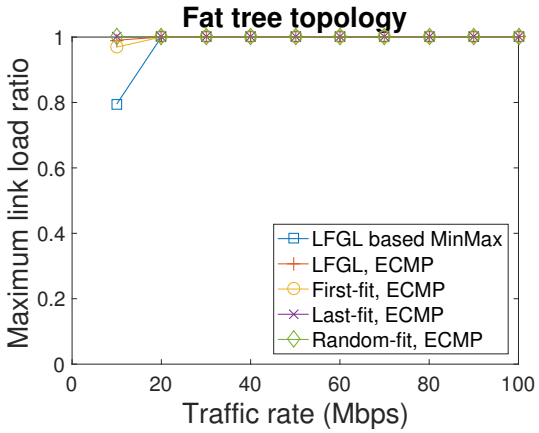
from 10 Mbps to 100 Mbps with a stride of 10 Mbps. Other settings are similar as in the tree simulations in Section 3.6.3. Fig. 3.12(a) compares the end-to-end delay



(a) End-to-end delay.



(b) Packet loss ratio.

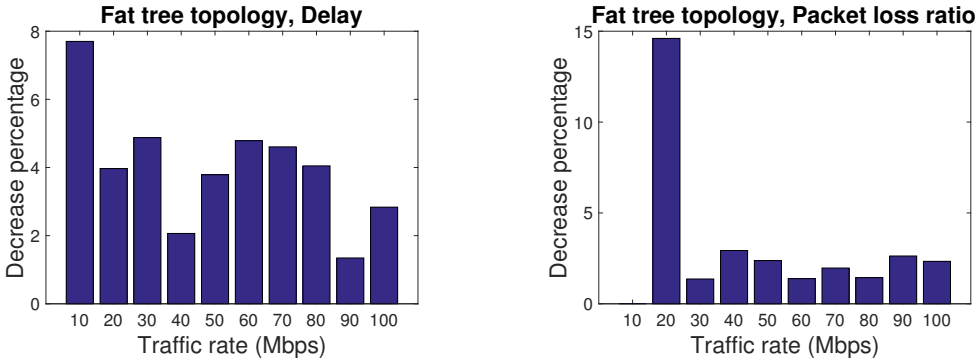


(c) Maximum link load ratio.

Figure 3.12: LFGL based MinMax routing simulation results.

of LFGL based MinMax routing with four benchmark solutions. We can see that LFGL based MinMax routing consistently achieves the shortest end-to-end delay. Among the other four benchmark solutions, LFGL achieves a shorter delay than the remaining. Fig. 3.12(b) shows the packet loss ratio. We can clearly see that in the order of LFGL based MinMax routing, LFGL, First-fit, Last-fit, and Random-fit, each achieves a lower packet loss ratio than the subsequent ones. Finally, Fig. 3.12(c) illustrates the maximum link load ratio. When the flow rate is 10 Mbps, we see that

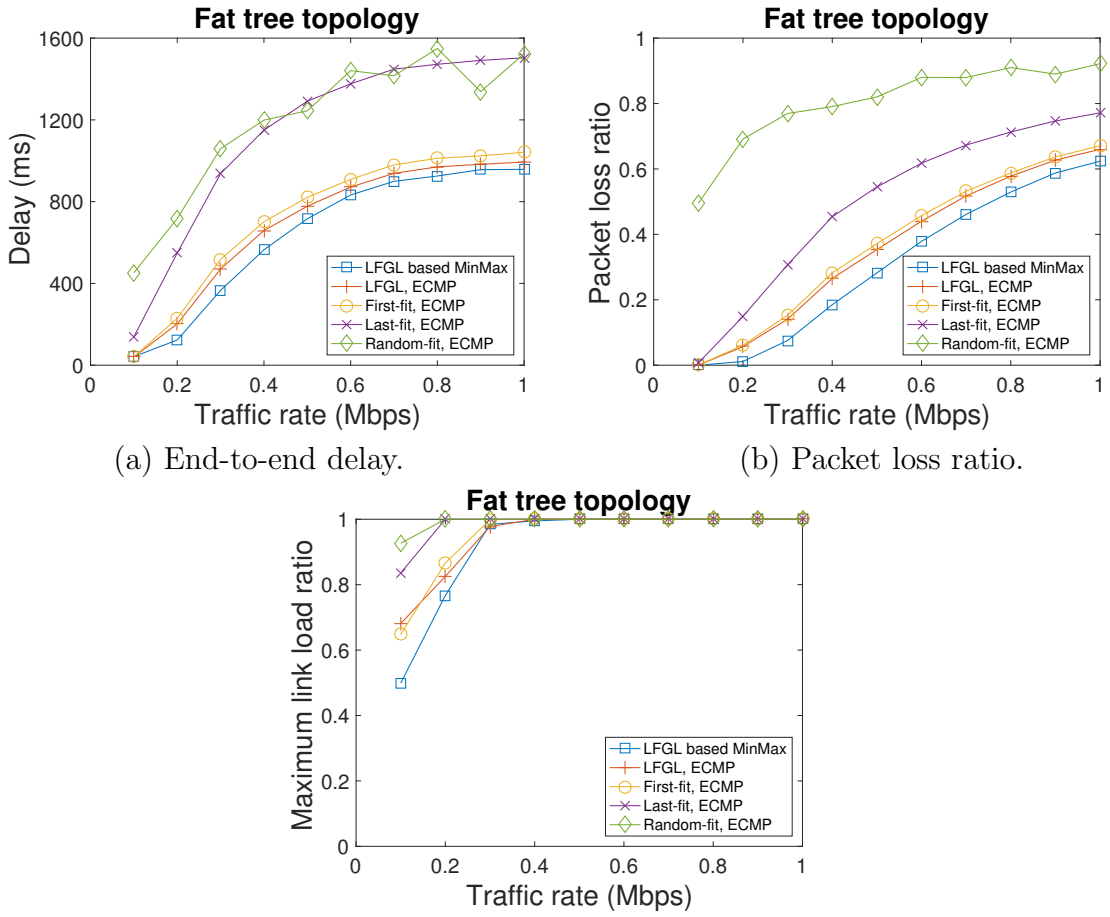
LFGL based MinMax routing performs the best, but when the flow rate increases to 20 Mbps and above, all algorithms have a saturated instantaneous maximum link load ratio of one. Similarly, we have also conducted simulations to evaluate the opti-



(a) End-to-end delay. (b) Packet loss ratio.

Figure 3.13: Improvement of multi-path optimization without predetermined paths.

mization algorithm for multiple flows as explained in Section 3.4.3. As comparison, LFGL based MinMax routing processes the multiple flows individually in a random order. Fig. 3.13(a) and (b) show the improvements of the optimization algorithm on the end-to-end delay and packet loss ratio. We can see that it reduces the end-to-end delay and packet loss ratio by up to 7% and 14%, but is not always effective due to the random order taken by LFGL based MinMax routing to process the flows. Again, when the flow rate is 10 Mbps, both algorithms have a zero packet loss ratio, and thus no improvement is seen in the figure. For performance evaluation in large scale networks, we have conducted simulations in a fat-tree network with 1024 hosts. To reduce the simulation convergence time, we decrease the link capacity to 1 Mbps. The simulation results for LFGL based MinMax routing are presented in Fig. 3.14. We can see that the algorithm also works in large scale networks, beating other benchmark solutions. However, the small link capacity leads to a longer transmission delay and consequently a longer end-to-end delay in Fig. 3.14(a). The



(a) End-to-end delay.

(b) Packet loss ratio.

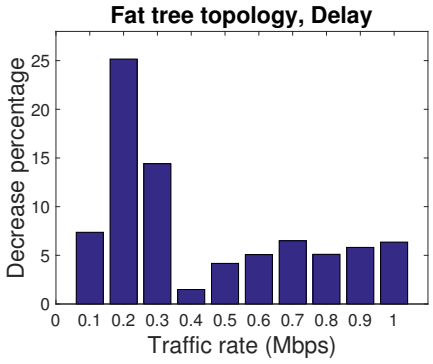
(c) Maximum link load ratio.

Figure 3.14: LFGL based MinMax routing simulation results (1024 hosts 1 Mbps link capacity).

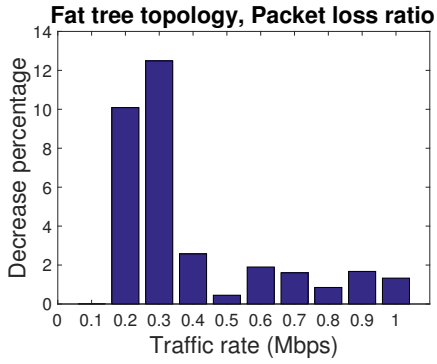
data in Fig. 3.15 also show that our optimization algorithm significantly improves the performance for multiple flows.

3.6.4 Comparison between Experimental and Simulation Results

Comparing the above experimental and simulation results, we can see that they are consistent. In the case with predetermined flow paths, the proposed LFGL rule outperforms other solutions, which rank in the order of First-fit, Random-fit, and Last-fit when the traffic changing ratio product of all middleboxes of a flow is less



(a) End-to-end delay.



(b) Packet loss ratio.

Figure 3.15: Improvement of multi-path optimization without predetermined paths (1024 hosts 1 Mbps link capacity).

than one. In the case without predetermined flow paths, LFGL based MinMax routing beats other solutions with shortest path routing.

However, there are also differences due to different link capacities and network sizes. Since the link capacity in the experiments is smaller than that in the simulations, the end-to-end delay in the experiments is long than that in the simulations. Also, the smaller network size and flow number in the experiments result in smoother and more predictable curves due to less variation.

3.7 Summary

With the development of virtualization technology, Network Function Virtualization enables flexible deployment of middleboxes as VMs running on commodity server hardware. In this chapter, we have studied how to efficiently deploy such middleboxes to achieve load balancing using a Software-Defined Networking approach, and considered in particular the traffic changing effects of different middleboxes. We formulate the Traffic Aware Middlebox Placement (TAMP) problem as a graph based optimization problem, and solve it in two steps. First, we solve the special

case of TAMP when flow paths are predetermined. For a single flow, we propose the Least-First-Greatest-Last (LFGL) rule, and prove its optimality; for multiple flows, we prove NP-hardness by reduction from the 3-Satisfiability problem, and propose an efficient heuristic. Next, we solve the general version of TAMP without predetermined flow paths. We prove that the general TAMP problem is NP-hard by reduction from the Hamiltonian problem, and propose the LFGL based Min-Max routing algorithm by integrating LFGL with MinMax routing. To validate our design, we have implemented the proposed algorithms in a prototype system with the open-source SDN controller Floodlight and emulation platform Mininet. In addition, we conducted simulations in ns-3 for performance evaluation in large scale networks. Extensive experiment and simulation results are presented to demonstrate the superiority of our algorithms over competing solutions.

CHAPTER 4

TRAFFIC AWARE PLACEMENT OF INTERDEPENDENT NFV MIDDLEBOXES

Network function virtualization (NFV) enables flexible implementation of network functions, or middleboxes, as virtual machines running on standard servers. However, the flexibility also creates a challenge for efficiently placing such middleboxes, due to the availability of multiple hosting servers, capability of middleboxes to change traffic volumes, and dependency between middleboxes. In this chapter, we address the optimal placement challenge of NFV middleboxes, and propose solutions for middleboxes of different traffic changing effects and with different dependency relations. First, we formulate the Traffic Aware Placement of Interdependent Middleboxes problem as a graph optimization problem. When the flow path is predetermined, we design optimal algorithms to place a non-ordered or totally-ordered middlebox set, and propose an efficient heuristic for the general scenario of a partially-ordered middlebox set after proving its NP-hardness. When the flow path is not predetermined, we show that the problem is NP-hard even for a non-ordered middlebox set, and propose a traffic and space aware routing heuristic. We have evaluated the proposed algorithms using large scale simulations and prototype experiments, and present extensive evaluation results to demonstrate the effectiveness of our design.

4.1 Introduction

Network function virtualization (NFV) transforms the implementation of network functions, also called middleboxes, from proprietary hardware appliances to virtual machines (VMs) running on industry standard servers [V. 12]. Leveraging the

underlying virtualization technology, VM-based software middleboxes bring many benefits that were not previously available, such as accelerated time-to-market, reduced hardware and operation cost, improved security, and elastic scalability [etsc].

The flexibility of VMs also poses challenges for efficient NFV implementation. In particular, traditional hardware middleboxes are deployed at fixed locations in the network, and leave no choice of service locations. In an NFV network, each switch may have one or more attached NFV servers with standard hardware that can host VMs of arbitrary network functions [V. 12]. It is thus possible to optimize the network performance by carefully selecting the location to place a software middlebox among multiple candidate servers. Improper placement decisions may cause inefficient flow paths and traffic jam.

Furthermore, the NFV service location challenge is complicated by the traffic changing effects of middleboxes. Unlike switches and routers that forward traffic without changing its volume, middleboxes may change the traffic volumes of processed flows, and may do it in different ways. For example, the Citrix CloudBridge WAN optimizer compresses traffic before sending it to the next hop, and may reduce the traffic volume by up to 80% [wan]. On the other hand, the BCH(63,48) encoder, used for satellite communications signaling messages, increases the traffic volume by 31% due to the checksum overhead [MVB93].

We use the following example to illustrate the traffic changing effects of middleboxes. Consider a network of three nodes v_1, v_2 and v_3, and two links (v_1, v_2) and (v_2, v_3). Each node has an attached NFV server (denoted as a circle in Fig. 4.1), and each server can host a single middlebox. A flow f starts at v_1 and ends at v_3, whose initial traffic rate is 1. Two middleboxes m_1 and m_2 need to be applied to f. m_1 will double the traffic rate, while m_2 will decrease it by 50%. By placing m_1 on v_1 and m_2 on v_3, the loads of links (v_1, v_2) and (v_2, v_3) will be $1 \times 2 = 2$, as shown

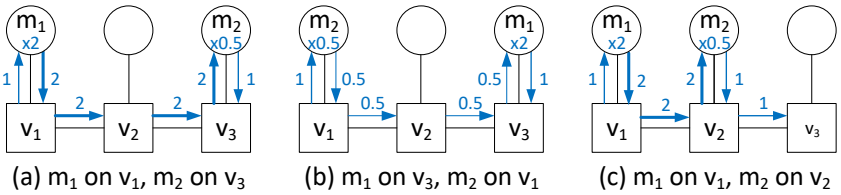


Figure 4.1: Traffic changing effects of middleboxes.

in Fig. 4.1(a). However, by placing m_1 on v_3 and m_2 on v_1, the loads of both links are reduced to $1 \times 0.5 = 0.5$, as shown in Fig. 4.1(b).

The placement of middleboxes is also constrained by the dependency relation that may or may not exist between middleboxes [S. 14]. For instance, an IPSec decryptor is usually placed before a NAT gateway [cis], while a VPN proxy can be placed either before or after a firewall [ms-]. In the above example, if there is a constraint for m_1 to be applied before m_2, then the placement scheme in Fig. 4.1(b) would violate the constraint. However, by placing m_1 on v_1 and m_2 on v_2, we can still reduce the load of link (v_2, v_3) from 2 to 1 as in Fig. 4.1(c), in contrast to the case in Fig. 4.1(a).

In this work, we study the optimal placement of NFV middleboxes. We propose comprehensive solutions that address the traffic changing effects of middleboxes, and consider different types of middlebox dependency relations. The proposed solutions will focus on elephant flows [A. 11], since it is more effective to optimize those large-size and long-duration flows [B. 10]. Our design utilizes the emerging Software-Defined Networking (SDN) technology as the implementation platform, because it enables efficient optimization by decoupling the network control plane and data plane.

Our main contributions in this chapter are summarized as follows.

1. We formulate the Traffic Aware Placement of Interdependent Middleboxes (TAPIM) problem, considering in particular a generalized partial order for

the middlebox dependency relation, as a graph optimization problem with the objective to load-balance the network.

2. For topologies with predetermined paths, such as the tree, we design optimal algorithms for the special case when the middlebox set is a non-ordered or totally-ordered one. For the general case when the dependency relation is a partial order, we show that the TAPIM problem is NP-hard by reduction from the Clique problem, and propose an efficient heuristic to convert a partially-ordered set to a totally-ordered one.

3. For topologies without predetermined paths, we prove that the TAPIM problem is NP-hard even for a non-ordered middlebox set by reduction from the Hamiltonian Cycle problem. Our proposed solution then works in two steps: first finding a path with enough resources to host all the middleboxes, and then placing the middleboxes on the given path.

4. We have implemented the proposed algorithms in the ns-3 simulator and a SDN based prototype, and present extensive simulation and experiment results to demonstrate the effectiveness of our design.

4.2 Problem Statement

In this section, we formulate the Traffic Aware Placement of Interdependent Middleboxes (TAPIM) problem.

Consider a network represented by a directed graph $G = (V, E)$. Each node $v \in V$ may have one or more attached NFV servers, and its space capacity is denoted as $sc[u]$[1]≥ 0, i.e., the maximum number of middleboxes to host. For simplicity, we

[1]We use square brackets [] to denote properties or known values, and round brackets () denote to functions or variables.

assume that each middlebox needs one space, and additional processing power can be achieved by multiple load-balanced middlebox instances [A. 08]. A link $(u, v) \in E$ has a bandwidth capacity $bc[u, v] \geq 0$, i.e., the amount of available bandwidth. For easy representation, we define connectivity by $connect[u, v] = 1$ if $bc[u, v] > 0$. The existing load of the link is denoted as $load[u, v]$.

For route calculation, each link $(u, v) \in E$ is assigned a weight, denoted as $weight(u, v, l)$, which is a non-decreasing function of the link load l, i.e., $\forall l \leq l', weight(u, v, l) \leq weight(u, v, l')$. A broad category of weight functions satisfy the non-decreasing requirement, such as the ones used by the popular Cisco EIGRP [eig] and OSPF [osp] protocols. The non-decreasing link weight function helps load-balance network traffic when the routing protocol aims to minimizes the path cost, which is defined as the weight sum of all the path links.

An elephant flow f is defined as a 4-tuple (src, dst, t, M), in which $src \in V$ is the source node, $dst \in V$ is the destination node, t is the initial traffic rate when f arrives at the ingress switch of the network, and M is the set of required middleboxes. (M may include multiple instances of the same middlebox type for increased processing power if necessary.)

Each middlebox $m \in M$ has an associated traffic changing factor $ratio[m] \geq 0$, which is the ratio of the traffic rate of a flow after and before being processed by m. The *dependency* relation $\leftarrow$ is defined as a strict partial order on M that is

1. Irreflexive: $m \nleftarrow m$,
2. Transitive: $m \leftarrow m'$ and $m' \leftarrow m''$ then $m \leftarrow m''$, and
3. Asymmetric: $m \leftarrow m'$ then $m' \nleftarrow m$.

Intuitively, $m \leftarrow m'$ means that the middlebox m should be applied before m'. For easy representation, define $depend[m, m'] = 1$ if $m \leftarrow m'$, and 0 otherwise.

When the flow f enters the network, a multi-hop path, denoted as *route*, will be assigned for the flow, which is a decision variable defined as

$$route(v,i) = \begin{cases} 1, & \text{if } v \in V \text{ is the } i^{th} \text{ hop on the path.} \\ 0, & \text{otherwise.} \end{cases} \tag{4.1}$$

We define i to start from one, and denote the last hop number as n for convenience. Note that repeating nodes are allowed on the path to enable more general solutions, i.e., $\exists v, i \neq i' : route(v,i) = 1$ and $route(v,i') = 1$. To avoid performance degradation for TCP flows, a flow is not allowed to be split among two paths [B. 10].

In addition, a placement scheme, denoted as *place*, will determine the location for each middlebox $m \in M$, which is a decision variable defined as

$$place(m,i) = \begin{cases} 1, & \text{if } m \in M \text{ is placed on the } i^{th} \text{ hop.} \\ 0, & \text{otherwise.} \end{cases} \tag{4.2}$$

To achieve load balance, we do not consider sharing of a middlebox by multiple elephant flows, but instead leave the remaining capacity of a placed middlebox to mice flows.

Use $t_{in}(i)$ and $t_{out}(i)$ to denote the incoming and outgoing traffic rate of flow f at the i^{th} hop on the path, respectively. If f is processed by a single middlebox m at the i^{th} hop, then $t_{out}(i) = t_{in}(i) ratio[m]$. Note that the incoming traffic rate at the flow source is the initial traffic rate, i.e., $t_{in}(1) = t$. For convenience, use $t(u,v) = \sum_{i \in [1,n-1]} route(u,i) route(v,i+1) t_{out}(i)$ to represent the traffic rate of f on link (u,v).

Consistent with most popular routing protocols, such as Cisco EIGRP and OSPF, our optimization objective is to determine *route* and *place* to minimize the

path cost of flow f as shown in Equation (4.3). It should be noted that the proposed solutions can easily adapt to other optimization objectives, such as minimizing the maximum link load in the network.

$$\textbf{minimize} \sum_{i=1}^{n-1} \sum_{u \in V} \sum_{v \in V} route(u,i) route(v,i+1) weight(u,v,t_{out}(i) + load[u,v]) \tag{4.3}$$

subject to the following constraints:

$$\forall i > n, \forall v \in V : route(v,i) = 0 \tag{4.4}$$

$$route(src,1) = 1, route(dst,n) = 1 \tag{4.5}$$

$$\forall v \in V : \sum_{i \in [1,n]} \sum_{m \in M} place(m,i) route(v,i) \leq sc[v] \tag{4.6}$$

$$\forall (u,v) \in E : t(u,v) + load[u,v] \leq bc[u,v] \tag{4.7}$$

$$\forall m \in M : \sum_{i \in [1,n]} place(m,i) = 1 \tag{4.8}$$

$$\forall m, m' \in M : \left(\sum_{i \in [1,n]} place(m',i) i - \sum_{i \in [1,n]} place(m,i) i \right) \times depend[m,m'] \geq 0 \tag{4.9}$$

$$\forall i \in [1,n] : t_{out}(i) = t_{in}(i) \prod_{m \in M} place_f(m,i) ratio[m] \tag{4.10}$$

$$\forall i \in [1,n-1], \forall u,v \in V : route(u,i) route(v,i+1) \leq connect[u,v] \tag{4.11}$$

Brief explanation of the model is as follows. Equation (4.3) defines the optimization objective. To discourage repeated links on the flow path, when the flow traverses

the same link multiple times, the link weight of each traverse will be counted separately in the path cost. Equation (4.4) states that the n^{th} hop is the last hop of the flow path. Equation (4.5) enforces the first and last hops of the flow path to be the source src and destination dst, respectively. Equation (4.6) states that, for a node v, the total space demand of hosted middleboxes $\sum_{i\in[1,n]}\sum_{m\in M} place(m,i)route(v,i)$ should not exceed its space capacity $sc[v]$. Equation (4.7) states that, for a link (u,v), the aggregate load of all flows traversing it $t(u,v) + load[u,v]$ should not exceed its bandwidth capacity $bc[u,v]$. Equation (4.8) states that a middlebox m should be installed once and only once. Equation (4.9) enforces the dependency relation between middleboxes, or in other words m must be placed no later than m' if the former is depended on by the latter. Equation (4.10) states that, the outgoing flow traffic rate at a hop, i.e., $t_{out}(i)$, is equal to the incoming rate, i.e., $t_{in}(i)$, multiplying the traffic changing ratios $ratio[m]$ of all the middleboxes m placed at this hop, i.e., $\prod_{m\in M} place_f(m,i)ratio[m]$. It also ensures flow conservation, in the sense that no flow traffic can be generated or terminated at an intermediate node, except the effects of middleboxes. Equations (4.11) enforces that consecutive hops of the flow path must be connected in the network.

4.3 Middlebox Placement with Predetermined Paths

In this section, we propose solutions for the TAPIM problem when the flow path, i.e., $route$, is predetermined. For example, in the tree topology, there is a unique path between any pair of leaves. We look at three cases of the problem. First, for the special case when there is no dependency between any middleboxes, we propose the Non-Ordered Set Placement algorithm that uses the least-first-greatest-last rule. Next, for the special case when there is a total dependency order on the middlebox

set, we propose the dynamic programming based Totally-Ordered Set Placement algorithm. Finally, for the general scenario of a partial dependency order on the middlebox set, we prove that it is NP-hard by reduction from the Clique problem, and propose an efficient heuristic to convert a partially-ordered set to a totally-ordered set.

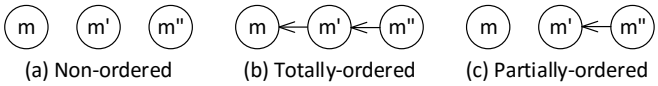


Figure 4.2: Examples of non-ordered, totally-ordered, and partially-ordered middlebox set.

4.3.1 Non-Ordered Middlebox Set

We start with the special case that the middlebox set M is a *non-ordered set*, i.e., $\forall m, m' \in M, m \nleftarrow m'$ and $m' \nleftarrow m$. Thus, different middleboxes can be placed in an arbitrary order. An example is shown in Fig. 4.2(a), where no dependency exists between middleboxes. We propose the *Non-Ordered Set Placement* (NOSP) algorithm, and show its optimality.

The basic idea is to shrink the flow as early as possible by installing the middleboxes that decrease the traffic rate from the path head, and expand the flow as late as possible by installing the middleboxes that increase the traffic rate from the path tail.

Apparently, the placement will succeed if the number of available spaces on the path is greater than or equal to the number of required middleboxes, i.e.,

$$\sum_{i\in[1,n]}\sum_{v\in V}\left(route[v,i]sc[v]/\sum_{i'\in[1,n]}route[v,i']\right)\geq|M|$$

If there are enough spaces, the NOSP algorithm places the middleboxes as follows.

1. Sort all the middleboxes $m \in M$ based on their traffic changing ratios $ratio[m]$.
2. Place the middleboxes m with $ratio[m] < 1$ from the path head in an increasing order of their traffic changing ratios. When a node has no more space, continue with the succeeding node on the path.
3. Place the middleboxes m with $ratio[m] \geq 1$ from the path end in an decreasing order. When a node has no more space, continue with the proceeding node.

Algorithm 4 Non-Ordered Set Placement

Require: $G, f, route, M[1..|M|]$
Ensure: $place$

```
1: sort m ∈ M[1..|M|] in increasing order of ratio[m]
2: i = 1, j = 1
3: while j ≤ |M| and ratio[M[j]] < 0 do
4:     while sc[v_i] > 0 and i ≤ n and ratio[M[j]] < 0 do
5:         place(M[j], i) = 1; sc[v_i] − −; j + +
6:     end while
7:     if sc[v_i] = 0 and j ≤ |M| and ratio[M[j]] < 0 then
8:         if i = n then
9:             exit with no-space error
10:        else
11:            i + +
12:        end if
13:    end if
14: end while
15: i = n, j = |M|
16: while j ≥ 1 and ratio[M[j]] ≥ 0 do
17:    while sc[v_i] > 0 and j ≥ 1 and ratio[M[j]] ≥ 0 do
18:        place(M[j], i) = 1; sc[v_i] − −; j − −
19:    end while
20:    if sc[v_i] = 0 and i ≥ 1 and ratio[M[j]] ≥ 0 then
21:        if i = 1 then
22:            exit with no-space error
23:        else
24:            i − −
25:        end if
26:    end if
27: end while
```

The pseudo code of LFGL is shown in Algorithm 4. For easy description, denote the i^{th} hop node on the flow path as v_i. Line 1 sorts all the middleboxes $m \in M[1..|M|]$ in the increasing order of their traffic changing factors $ratio[m]$. Lines 2 to 14 install the middleboxes with traffic changing ratios less than one from the head of the flow path. In detail, line 2 initializes the first stage by starting with the first hop and the middlebox with the least traffic changing ratio, i.e., $M[1]$ after sorting. Line 3 is the loop to process middleboxes with ratios less than one. Line 4 checks if the current hop v_i has available spaces. If yes, Line 5 installs the middlebox $M[j]$ on hop i, decrements the number of available spaces $sc[v_i]$ at v_i, and increments the middlebox index j. Lines 7 and 8 check whether the current hop v_i is already the last hop and there are still middleboxes with ratios less than one to install. If yes, line 9 exits since there is no more space on the flow path. Otherwise, if the current hop v_i is not the last hop, line 11 continues with the next hop on the path. Lines 15 to 27 install the middleboxes with ratios greater than or equal to one from the tail of the flow path in a similar manner as above.

As can be seen, NOSP processes each middlebox in M only once after sorting, and therefore its time complexity is $O(|M| \log |M|)$, i.e., the time complexity to sort M.

Lemma 4.3.1 *NOSP minimizes the flow rate on each link of the path.*

The proof of Lemma 4.3.1 is omitted, which is similar to the proof of Theorem 3.3.1 in Chapter 3.

Theorem 4.3.2 *The Non-Ordered Set Placement algorithm achieves the minimum path cost.*

Proof. By Lemmas 4.3.1, NOSP minimizes the flow rate on each path link. Note that for the link traversed multiple times by the path, NOSP minimizes the flow rate

for each pass. Therefore, the total load of each link as the sum of the existing load and flow rate is also minimized. Given that the link weight function $weight(u, v, l)$ is a non-decreasing function of the total link load l, NOSP minimizes the weight of each path link and subsequent the path cost. □

4.3.2 Totally-Ordered Middlebox Set

Next, we solve the other special case of TAPIM when the middlebox set is a totally-ordered set, i.e., $\forall m, m' \in M$, either $m \leftarrow m'$ or $m' \leftarrow m$, or in other words the middleboxes form a dependency chain. An example is shown in Fig. 4.2(b), in which m must be placed m' and m' before m''. Although the placement order of the middleboxes has been determined, it is still necessary to determine the optimal placement location for each middlebox, because there may be an excessive number of available spaces on the flow path. For easy description, we use m_j to denote the j^{th} middlebox from the head of the dependency chain, and v_i to denote the i^{th} hop node on the flow path, where i and j start from 1.

We propose a dynamic programming based algorithm called *Totally-Ordered Set Placement* (TOSP) based on the following observation. Use TOSP(i, j) to denote the minimum weight sum of the first i links when place the first j middleboxes, i.e., m_1 to m_j, on the first i hops, i.e., v_1 to v_i, of the flow path. The optimal substructure gives the following recursive formula.

$$\text{TOSP}(i,j) = \begin{cases} w(1;1,j), \text{ if } i = 1. \\ \min_{x\in[1,j+1]} \Big(\text{TOSP}(i-1,x-1) + \quad w(i;x,j)\Big), \text{ otherwise.} \end{cases} \tag{4.12}$$

where $w(i; x, j)$ is the weight of link (v_i, v_{i+1}) when placing middleboxes m_x to m_j on node v_i, i.e.,

Equation (4.12) states that if $i = 1$, TOSP(i, j) is simply the weight of the first path link when placing all the first j middleboxes on the first hop v_1. Otherwise, the optimal result TOSP(i, j) to place the first j middleboxes on the first i hops is to select the minimum link weight sum among $j + 1$ possible solutions, in which the x^{th} solution places the first $x - 1$ middleboxes on the first $i - 1$ hops, i.e., TOSP$(i - 1, x - 1)$, and places the remaining middleboxes m_x to m_j on the i^{th} hop v_i, i.e., $w(i; x, j)$.

$$w(i;x,j) = \begin{cases} 0, \text{ if } x > j. \\ weight\Big(v_i, v_{i+1}, t\prod_{y\in[1,j]} ratio[m_y] + \quad load[v_i, v_{i+1}]\Big), \\ \text{ if } sc[v_i] \geq j - x + 1. \\ \infty, \text{ otherwise.} \end{cases} \tag{4.13}$$

Equation (4.13) calculates the weight of the i^{th} path link, i.e., (v_i, v_{i+1}), when placing middleboxes m_x to m_j on the i^{th} hop v_i, and sets it to zero if $x > j$ or infinity if v_i has less than $j - x + 1$ available spaces. Note that if $x \leq j$ and v_i has sufficient spaces, $w(i; x, j)$ does not depend on x. In other words, as long as v_i has no less than $j - x + 1$ spaces to host middleboxes m_x to m_j, the weight of link (v_i, v_{i+1}) is the same, which simplifies the calculation of TOSP(i, j) as the sum of the minimum sub-solution $\min_{x\in[j-sc[v_i]+1,j+1]}\big\{\text{TOSP}(i-1, x-1)\big\}$ and a constant. Thus, we can rewrite the recursive relationship as:

$$\begin{aligned} & \text{TOSP}(i,j) \\ = & \min_{x\in[1,j+1]}\big\{\text{TOSP}(i-1,x-1) + w(i;x,j)\big\} \\ = & \min_{x\in[j-sc[v_i]+1,j+1]}\big\{\text{TOSP}(i-1,x-1) + w(i;x,j)\big\} \end{aligned} \tag{4.14}$$

$$= \min_{x \in [j - sc[v_i] + 1, j+1]} \{\text{TOSP}(i-1, x-1)\} + weight\Big(v_i, v_{i+1}, load[v_i, v_{i+1}] + t \prod_{y \in [1,j]} ratio[m_y]\Big)$$

The pseudo code of TOSP is shown in Algorithm 5. Lines 1 to 7 initialize the first row of the dynamic programing matrix. Line 1 checks if the first hop v_1 has j spaces. If yes, TOSP$(1, j)$ is assigned the weight of the first link in line 3, and otherwise infinity in line 4. Lines 8 and 9 start the iteration to calculate the remaining entries in the matrix. Based on the previous results, lines 10 to 15 finds among the viable schemes the one with the minimum link weight sum to place a portion of middleboxes in the first $i-1$ hops. Finally, line 16 calculates the optimal TOSP(i, j) by adding the minimum link weight sum of the first $i-1$ hops and the link weight of the last hop.

Algorithm 5 Totally-Ordered Set Placement

Require: $G, f, route, M[1..|M|], depend$
Ensure: $place$
1: **for** $j = 1$ **to** $|M|$ **do**
2: **if** $sc[v_1] \geq j$ **then**
3: TOSP$(1, j) = weight(v_1, v_2, t \prod_{y=1}^{j} ratio[m_y] + load[v_1, v_2])$
4: **else**
5: TOSP$(1, j) = \infty$
6: **end if**
7: **end for**
8: **for** $i = 2$ **to** n **do**
9: **for** $j = 1$ **to** $|M|$ **do**
10: $min = \infty$
11: **for** $x = j - sc[v_i] + 1$ **to** $j + 1$ **do**
12: **if** TOSP$(i-1, x-1) < min$ **then**
13: $min =$TOSP$(i-1, x-1)$
14: **end if**
15: **end for**
16: TOSP$(i, j) = min + weight(v_i, v_{i+1}, t \prod_{y=1}^{j} ratio[m_y] + load[v_i, v_{i+1}])$
17: **end for**
18: **end for**

When the flow path *route* is not efficient and contains repeating nodes, the above algorithm may obtain a sub-optimal result. The reason is that, different hops of a repeating node share middlebox spaces, but the above algorithm processes those hops always from the path head, and thus assigns earlier hops higher priority. A simple solution is to first enumerate all the possibilities to divide the shared spaces among different hops of a repeating node, and then apply TOSP to each possible division. For example, if the flow path contains a repeating node with s spaces that appears twice at the i_1^{th} hop v_{i_1} and the i_2^{th} hop v_{i_2}, we view v_{i_1} and v_{i_2} as two independent nodes by allocating $x \in [0, s]$ spaces to v_{i_1} and $s - x$ spaces to v_{i_2}. TOSP is then applied to each different x value, and the minimum path cost among all the cases is the optimal solution.

When there is no repeating node on the flow path, the time complexity of the TOSP algorithm is $O(n|M|^2)$, because the dynamic programming table has n rows and $|M|$ columns, and it takes up to $O(|M|)$ time to calculate each table entry. When there are r repeating nodes and each node has up to s spaces and appears in up to h hops, the time complexity is $O(s^{(h-1)r}n|M|^2)$, because there are $s^{(h-1)r}$ possible divisions of shared spaces, and the time complexity to apply TOSP to each division is $O(n|M|^2)$. Fortunately, efficient routing paths should have small or zero r and h values.

4.3.3 Partially-Ordered Middlebox Set

We now solve the general scenario where the dependency relation is a partial order. The following theorem shows the NP-hardness of the problem.

Theorem 4.3.3 *The TAPIM problem with a predetermined path for a partially ordered middlebox set is NP-hard.*

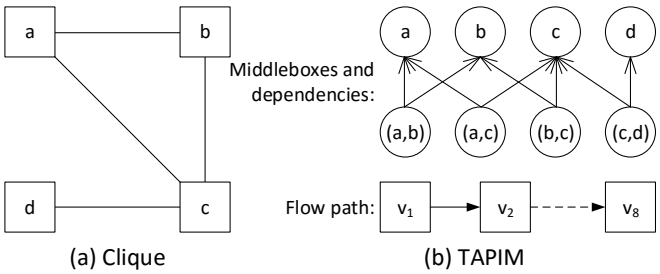


Figure 4.3: Reduction from Clique to TAPIM with predetermined path.

Proof. We prove by reduction from the Clique problem [CLRS09]. The clique problem decides whether an undirected graph $G = (V, E)$ has a clique of size k, which is a complete sub-graph with k vertices and $\binom{k}{2}$ edges. For example, the graph in Fig. 4.3(a) has a clique of size 3: $(\{a, b, c\}, \{(a, b), (a, c), (b, c)\})$.

Given an instance of the Clique problem with a graph $G = (V, E)$, an instance of the TAPIM problem can be constructed in polynomial time as follows.

1. For each vertex $p \in V$, create a vertex middlebox m_p with $ratio[m_p] = 2$.
2. For each edge $(p, q) \in E$, create an edge middlebox $m_{(p,q)}$ with $ratio[m_{(p,q)}] = 2^{-k/\binom{k}{2}}$.
3. The middlebox corresponding to an edge (p, q) depends on the two middleboxes corresponding to its two incident vertices p and q, i.e., $m_p \leftarrow m_{(p,q)}$ and $m_q \leftarrow m_{(p,q)}$.
4. There is a single flow f with the initial traffic rate of one, i.e., $t = 1$. The path has $|V| + |E|$ nodes. Use v_i to denote the i^{th} node on the path. Each node has a space capacity of one, i.e., $sc[v_i] = 1$.
5. Each link on the path has a bandwidth capacity of infinity, i.e., $bc[v_i, v_{i+1}] = \infty$. The link $(v_{k+\binom{k}{2}}, v_{k+\binom{k}{2}+1})$ is called the critical link, with its weight being one if the link load is no more than one and infinity otherwise, i.e.,

$$weight((v_{k+\binom{k}{2}}, v_{k+\binom{k}{2}+1}), l) = \begin{cases} 1, \text{ if } l \leq 1. \\ \infty, \text{ if } l > 1. \end{cases} \tag{4.15}$$

The weight of any other link is always zero.

Next, we show that that if the graph $G = (V, E)$ has a clique of size k, then the constructed TAPIM instance has a minimum path weight of one. Assume the solution clique of size k is $G' = (V', E')$, the solution for TAPIM is constructed as follows.

1. For each vertex $p \in V'$, place the corresponding middlebox m_p one by one starting from the path head v_1.
2. For each edge $(p, q) \in E'$, continue placing the corresponding middlebox $m_{(p,q)}$ along the path.
3. For each remaining vertex $p \in V \setminus V'$, continue placing the corresponding middlebox m_p.
4. For remaining edges in $(p, q) \in E \setminus E'$, continue placing the corresponding middlebox $m_{(p,q)}$.

Since $G' = (V', E')$ is a complete sub-graph, for each edge middlebox $m_{(p,q)}$ placed in Step 2, its two predecessor vertex middleboxes m_p and m_q must have been placed in Step 1. Furthermore, since Step 3 places all the remaining vertex middleboxes, the predecessors of all edge middleboxes placed in Step 4 are satisfied. Therefore, there is no dependency violation. Also, when the flow arrives at the critical link $(v_{k+\binom{k}{2}}, v_{k+\binom{k}{2}+1})$, it has traversed $|V'| = k$ vertex middleboxes and $|E'| = \binom{k}{2}$ edge middleboxes, its flow rate is $1 \times 2^k \times (2^{-k/\binom{k}{2}})^{\binom{k}{2}} = 1$. As a result, the weight of the critical link is one, and the entire path cost is also one.

Conversely, if the constructed TAPIM instance has a minimum path cost of one, then the graph $G = (V, E)$ has a clique of size k denoted as $G' = (V', E')$. We show by contradiction that the first $k + \binom{k}{2}$ middleboxes placed on the path must be k vertex middleboxes and $\binom{k}{2}$ edge middleboxes.

1. For contradiction, assume that there are more than k vertex middleboxes and fewer than $\binom{k}{2}$ edge middleboxes. Since the traffic changing ratio of a vertex middlebox is 2 and that of an edge middlebox is $2^{-k/\binom{k}{2}}$, the flow rate will be greater than one when it comes to the critical link, and the weight of the critical link would be infinity instead of one.

2. For contradiction, assume that there are fewer than k vertex middleboxes and more than $\binom{k}{2}$ edge middleboxes. With fewer than k vertex middleboxes, the number of edges generated by those vertices in G must be fewer than $\binom{k}{2}$, and thus there must exist an edge middlebox whose predecessors have not been satisfied.

Thus, the first $k + \binom{k}{2}$ middleboxes placed on the path are exactly k vertex middleboxes and $\binom{k}{2}$ edge middleboxes, and the predecessor vertex middleboxes of all edge middleboxes are included in this set. Therefore, the sub-graph G' corresponding to those vertex and edge middleboxes form a clique of size k. □

After proving the NP-hardness, our solution to place a partially-ordered middelbox set is to first convert it to a totally-ordered middlebox set and then apply TOSP.

Following the least-first-greatest-last rule to place in NOSP, the objective of the conversion algorithm is to arrange the middleboxes in the resulting total order chain in the increasing order of their traffic changing ratios. The intuitive solution is thus to iteratively find the middleboxes without dependencies, remove among them the

one with the least traffic changing ratio, and add it to the end of the total order chain. For example, given four middleboxes with the following traffic changing ratios and dependencies: $1.4 \leftarrow 1.5$ and $1.6 \leftarrow 0.1$, the conversion result will be the following total order chain: $1.4 \leftarrow 1.5 \leftarrow 1.6 \leftarrow 0.1$.

To increase the solution search space, we also add lookahead information by searching further beyond just the middleboxes without dependencies. Define a self-dependent middlebox tree of size k to be a tree of k middleboxes that are rooted from a single middlebox and depend on only the middleboxes in the tree. The traffic changing ratio of the tree is the product of the traffic changing ratios of all the middleboxes in the tree. In the above example, $1.6 \leftarrow 0.1$ is a self-dependent tree of size 2, and its traffic changing ratio is $1.6 \cdot 0.1 = 0.16$.

Algorithm 6 Converting Partially-Ordered Set to Totally-Ordered Set with lookahead of k

Require: $k, M, depend$
Ensure: M'
1: **for** each middlebox $m \in M$ **do**
2: **if** m has no dependency **then**
3: $minratio[m] = \min_{x=1}^{k}\{$ratio of size x self-independent tree with root $m\}$
4: **else**
5: $minratio[m] = \infty$
6: **end if**
7: **end for**
8: **for** $j = 1$ **to** $|M|$ **do**
9: select in M middlebox m with least $minratio[m]$
10: $M'[j++] = m$
11: $M = M \setminus \{m\}$
12: **for** each middlebox m' directly dpendening on m **do**
13: $minratio[m'] = \min_{x=1}^{k}\{$ratio of size x self-independent tree with root $m'\}$
14: **end for**
15: **end for**

The conversion algorithm with a lookahead value of k works in iterations as follows. In each iteration, the algorithm first finds all the middleboxes with no

dependency. Using each of such middleboxes as the root, the algorithm calculates the self-independent tree of size up to k that has the minimum traffic changing ratio. Among all the calculated trees with different root middleboxes, the algorithm selects the one with the minimum traffic changing ratio, removes its root middleboxes, and adds it to the total order chain. For the above example, the first iteration generates two trees of size up to 2: 1.4 of size 1 with 1.4 being the root, and $1.6 \leftarrow 0.1$ of size 2 with 1.6 being the root. Since the traffic changing ratio of the latter 0.16 is less than that of the former 1.4, the root of the latter will be removed. The resulting total order chain after the algorithm converges is thus: $1.6 \leftarrow 0.1 \leftarrow 1.4 \leftarrow 1.5$.

The pseudo code of the conversion algorithm is shown in Algorithm 6. Lines 1 to 7 conduct the initialization to calculate for each middlebox without dependencies the minimum ratio self-independent tree. Lines 8 to 15 are the iterations to build the result total order chain. Line 9 finds among the middleboxes without dependencies the one with the minimum ratio self-independent tree, line 10 adds it to the total order chain, and line 11 removes it from the original middlebox set. Lines 12 to 14 calculate for each child of the removed middlebox its minimum ratio self-independent tree.

When the lookahead parameter $k = 1$ or 2, the time complexity of the conversion algorithm is $O(|M| \log |M|)$, because there are up to $|M|$ iterations, and the time complexity to select the middelebox with the minimum traffic changing ratio is $O(\log |M|)$ using a heap. When $k = 2$, the optimal self-dependent trees of size up to 2 with each middlebox being the root can be pre-calculated in $O(|M|)$ time. Since $|M|$ is usually small, and $k = 2$ will be sufficient in most cases.

4.4 Middlebox Placement without Predetermined Path

In this section, we solve the TAPIM problem when the flow path, i.e., *route*, is not predetermined. We start by proving that the TAPIM problem without a predetermined path is NP-hard even for a non-ordered set by reduction from the Hamiltonian Cycle problem. We then propose a two-step solution by first finding a flow path with sufficient spaces and then applying the algorithms in Section 4.3 to place middleboxes on the given path.

4.4.1 NP-Hardness

The following theorem shows the hardness of the TAPIM problem without a predetermined flow path.

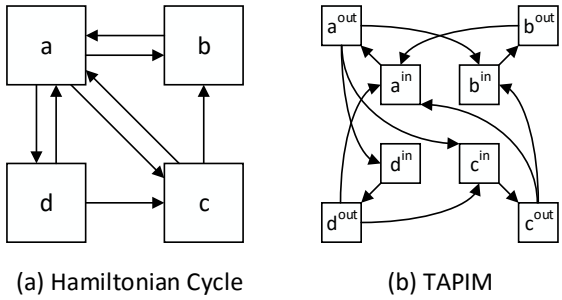


Figure 4.4: Reduction from Hamiltonian Cycle to TAPIM.

Theorem 4.4.1 *Without a predetermined flow path, the TAPIM problem is NP-hard even for a non-ordered middlebox set.*

Proof. The proof is similar to the proof of Theorem 3.4.1 in Chapter 3. For ease of reading, we still give the detailed proof as following. The proof is by reduction from the Hamiltonian Cycle problem, which determines for a directed graph $G = (V, E)$ whether there exists a simple cycle that contains each vertex in V. Note that a Hamiltonian cycle must be a simple cycle without repeating nodes.

For a Hamiltonian Cycle instance with a graph $G = (V, E)$, a TAPIM instance with a graph $G' = (V', E')$ can be constructed in polynomial time as follows:

1. For each vertex $v \in V$, create two nodes $v^{in}, v^{out} \in V'$, where v^{in} has a space capacity of zero, i.e., $sc[v^{in}] = 0$, and v^{out} of one, i.e., $sc[v^{out}] = 1$. Connect the two nodes with a link $(v^{in}, v^{out}) \in E'$, and set its bandwidth capacity to infinity, i.e., $bc[v^{in}, v^{out}] = \infty$. Its weight is one if the load is no more than one, i.e., $\forall l \leq 1, weight(v^{in}, v^{out}, l) = 1$, and infinity otherwise.

2. For each edge $(u, v) \in E$, create a link $(u^{out}, v^{in}) \in E'$, and set its bandwidth capacity to one, i.e, $bc[u^{out}, v^{in}] = 1$, and its weight to be zero, i.e., $\forall l, weight(u^{out}, v^{in}, l) = 0$. An example to create G' from G is shown in Fig. 4.4.

3. Create a flow f with the source and destination both being a^{in}, i.e., $src = dst = a^{in}$, where a is an arbitrary vertex in V. The initial traffic rate is one, i.e., $t = 1$. The middlebox set M is a non-ordered one with $|V|$ middleboxes, i.e., $|M| = |V|$, and each middlebox has a traffic changing ratio of one, i.e., $\forall m \in M, ratio[m] = 1$.

Next, we show that if G has a Hamiltonian cycle, then the TAPIM instance with G' and f has a minimum path cost of $|V|$. Since the Hamiltonian cycle of G traverses each vertex in V exactly once, we can construct a similar path *route* in G' that traverses each node in V' exactly once as follows. Without loss of generality, assume that the Hamiltonian cycle in G starts at a, traverses all the other vertices, and ends at a.

1. For each edge (u, v) in the Hamiltonian cycle, add link (v^{out}, v^{in}) to the path.

2. For each vertex v in the Hamiltonian cycle, add link (v^{in}, v^{out}) to the path.

For the example in Fig. 4.4, G has a Hamiltonian cycle $a \Rightarrow d \Rightarrow c \Rightarrow b \Rightarrow a$, and the constructed path *route* in G' is $a^{in} \Rightarrow a^{out} \Rightarrow d^{in} \Rightarrow d^{out} \Rightarrow c^{in} \Rightarrow c^{out} \Rightarrow b^{in} \Rightarrow b^{out} \Rightarrow a^{in}$. Since the Hamiltonian cycle traverses each vertex in G exactly once, and each vertex v in G maps to a pair of nodes v^{in} and v^{out} in G', *route* also traverses each node, including each outgoing node v^{out} that has a space, in G' exactly once. Thus, the path *route* has $|V|$ available spaces, sufficient to host all the middleboxes in M. Also, *route* traverses each (v^{in}, v^{out}) edge exactly once, and thus the weight of each link is one and the path cost is $|V|$.

In the other direction, if the constructed TAPIM instance with G' and f has a solution *route* and *place* with a minimum path cost of of $|V|$, then G will have a Hamiltonian cycle. Given *route* in G', we construct a simple cycle in G as follows. Starting with $src = a^{in}$, sequentially add the corresponding vertex $v \in V$ of each incoming node $v^{in} \in V'$ on *route* to the cycle in G. Next, we show that the constructed cycle in G is a Hamiltonian cycle. Remember that f needs to traverse all outgoing nodes v^{out} in order to obtain sufficient middlebox spaces, and the only way to reach v^{out} is through v^{in}. Therefore, *route* traverses all the nodes in G', and the constructed cycle traverses all the vertices in G. In addition, since the path cost is $|V|$, each (v^{in}, v^{out}) link for any v is traversed at most once. Therefore, the constructed cycle in G traverses each vertex $v \in V$ exactly once, and it is thus a Hamiltonian cycle. □

4.4.2 Traffic and Space Aware Routing

Our solution to TAPIM without a predetermined path works in two steps by first finding a viable path for the flow and then applying the algorithms in Section 4.3 to place the middleboxes on the determined path.

From the proof of Theorem 4.4.1, it can be seen that it is NP-hard to find the minimum cost path with sufficient spaces, and thus we propose the Traffic And Space Aware Routing (TASAR) heuristic. The basic idea is to originate from the source, iteratively route to a nearby node with spaces until sufficient spaces have been accumulated, and finally go to the destination.

In detail, the TASAR heuristic works as follows. It starts by calculating the number of spaces needed on the path besides those in the source and destination, i.e., $|M| - sc[src] - sc[dst]$. Next, the heuristic enters iterative loops to accumulate the necessary number of spaces. In the x^{th} iteration, the heuristic runs Dijkstra's algorithm [CLRS09] from v_x, with $v_1 = src$, to find the nearest (in terms of the path cost) node with spaces, denoted as v_{x+1}, and add the path from v_x to v_{x+1} to the flow path *route*. If sufficient spaces have been accumulated, i.e., $|M| - sc[src] - sc[dst] - \sum_{i=1}^{x+1} sc[v_i] \leq 0$, the iteration stops, and the heuristic runs Dijkstra's algorithm for the last time to find the minimum cost path from the current node v_{x+1} to the destination *dst*. Otherwise, if more spaces are needed, the iteration continues. The

Algorithm 7 Traffic and Space Aware Routing

Require: $G, src, dst, |M|$
Ensure: *route*
1: $missing = |M| - sc[src] - sc[dst]$
2: $v_1 = src; i = 1$
3: **while** $missing > 0$ **do**
4: $\quad v_{i+1}$ = nearest node from v_i with spaces
5: $\quad$ append to flow path *route* the section from v_i to v_{i+1}
6: $\quad missing = missing - sc[v_{++i}]$
7: **end while**
8: append to flow path *route* the section from v_i to *dst*

pseudo code of the TASAR heuristic is shown in Algorithm 7. Line 1 calculates the number of missing spaces. Line 2 initializes the loop between line 3 and 7, which uses Dijkstra's algorithm to find the nearest node with spaces and appends the path

to it to the flow path. Finally, line 8 applies Dijkstra's algorithm again to find the minimum cost path to the destination.

The time complexity of the heuristic is $O(|M|(|E| + |V| \log |V|))$, because the there will be up to $O(|M|)$ iterations, and the time complexity of each iteration is that of Dijkstra's algorithm $O(|E| + |V| \log |V|)$.

4.5 Experiment and Simulation Results

We use a combination of simulations and experiments for performance evaluation. We have conducted simulations to obtain performance data in large scale networks, and have also built a prototype to validate the solutions in a realistic environment. In this section, we present extensive simulation and experiment results to show the effectiveness of our design.

4.5.1 Simulation Results

We have implemented the proposed algorithms in the ns-3 simulator, and used the same performance metrics mentioned in Section 3.6.2 for benchmark comparisons.

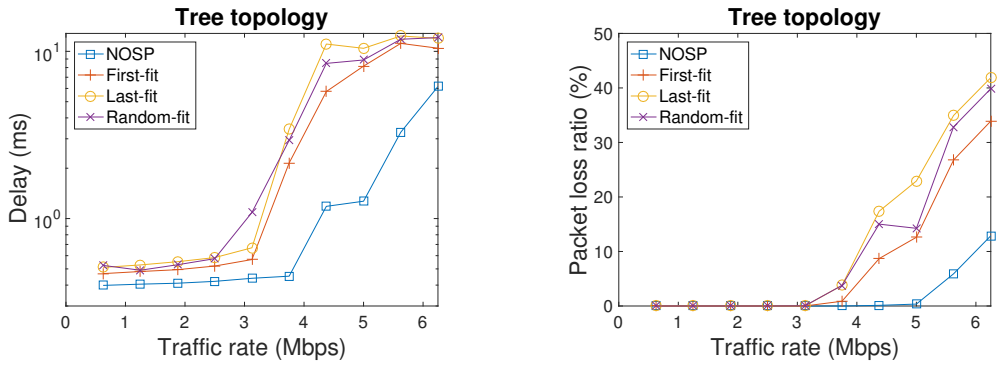
To reflect the burst of realistic traffic, we adopt the on-off traffic model. When a flow f is in the on state, its initial traffic rate t is the product of a baseline rate and a random number between 0.5 to 1.5; when in the off state, its initial traffic rate is zero. A flow is in each of the two states for 50% of the time. There are two middlebox sets with different traffic changing ratios and dependency relations, and each flow will randomly choose one of them. Each link in the network has a bandwidth capacity of 100 Mbps and a propagation delay of 2 μs. Every simulation run lasts five minutes, and the presented result is the average of four simulation runs.

Placing Non-Ordered Set with Predetermined Path: For the NOSP algorithm to place a non-ordered middlebox set on a predetermined path, since there are no existing solutions for the studied problem, we designed the following three benchmark algorithms. Same as NOSP, all the benchmark algorithms sort the middleboxes based on their traffic changing ratios before placing them.

1. *First-fit* continuously places the sorted middleboxes in the increasing order from the head of the flow path.
2. *Last-fit* continuously places the sorted middleboxex in the decreasing order from the tail of the flow path.
3. *Random-fit* randomly places the sorted middleboxes on random nodes on the path that have spaces.

We pick the tree topology, since is a popular choice among institutional networks, and there is only a single path between any pair of nodes. We set up a four-layer quad-tree with 21 switches and 64 hosts. Each switch has 13 spaces to ensure sufficient spaces for all flows. The link weight is set consistent with CISCO EIGRP [eig] as $bandwidth/(256-load)$. Each host generates a flow to a random destination. The two candidate sets of middleboxes are: {0.7, 0.8, 1.1, 1.2} and {0.8, 0.9, 1.1, 1.3}. The baseline traffic rate of each flow ranges from 0.625 to 6.25 Mbps with a stride of 0.625 Mbps.

Fig. 4.5(a) shows the average end-to-end delays of the four algorithms. We can see that NOSP consistently achieves the shortest delay due to its optimal middlebox placement scheme. On the other hand, Last-fit has the worst performance, because it places middleboxes at the path end, and the flow rate is $1\times$ on most links of the path. By contrast, First-fit achieves relatively shorter delay by placing middleboxes at the beginning of the path. The reason is that half of the flows picked the first set



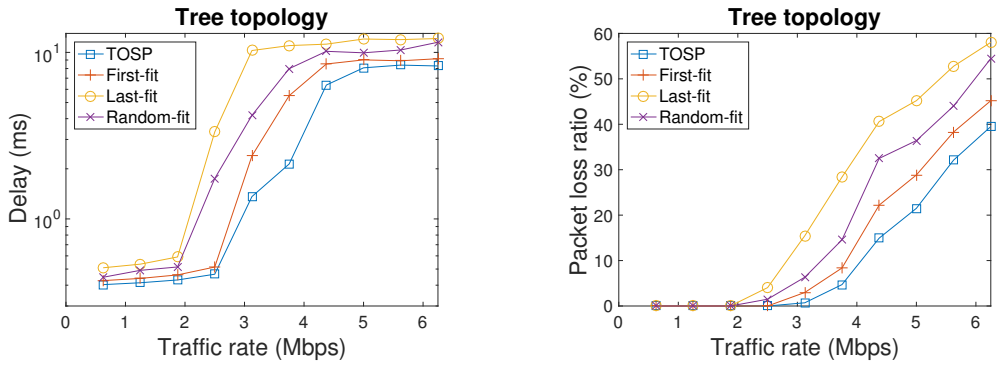
(a) End-to-end delay. (b) Packet loss ratio.

Figure 4.5: NOSP simulation results.

of middleboxes with an aggregate traffic changing ratio of $0.7 \cdot 0.8 \cdot 1.1 \cdot 1.2 = 0.74$, and the other half picked the second set with a ratio of $0.8 \cdot 0.9 \cdot 1.1 \cdot 1.3 = 1.03$, so on average the middleboxes placed at the path head reduce the traffic rate of a flow to $(0.74 + 1.03)/2 = 0.885\times$, which is the traffic rate on most links of the path. Finally, the delay of Random-fit is between that of Last-fit and First-fit due to its random strategy.

Fig. 4.5(b) plots data for the packet loss ratio. We can observe a similar trend that NOSP always achieves the lowest packet loss ratio. When the flow traffic rate is small, all the algorithms have zero packet loss ratios. Compared with NOSP, other algorithms have packet loss happened much earlier, and their ratios increase much faster.

Placing Totally-Ordered Set with Predetermined Path: Next, we evaluate the TOSP algorithm with similar benchmark algorithms as above, in which First-fit, Last-fit, and Random-fit place the middleboxes based on the given total order chain from the path head, tail, and randomly, respectively. The traffic changing ratios and dependency chains of the two candidate sets of middleboxes are: $\{0.8 \leftarrow 1.1 \leftarrow 0.7 \leftarrow 1.2\}$ and $\{1.2 \leftarrow 0.7 \leftarrow 1.1 \leftarrow 0.8\}$. Other simulation settings are the same



(a) End-to-end delay. (b) Packet loss ratio.

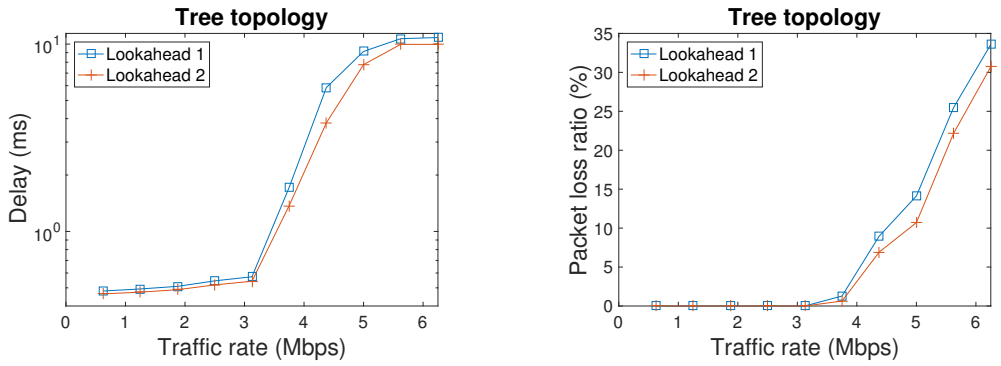
Figure 4.6: TOSP simulation results.

as above.

As shown in Fig. 4.6(a), TOSP achieves the shortest end-to-end delay because of its dynamic programming based optimal middlebox placement scheme. Similar as above, the delays of the other three algorithms increase in the sequence of First-fit, Random-fit, and Last-fit. The packet loss ratio data in Fig. 4.6(a) are consistent, and TOSP consistently outperforms others.

Placing Partially-Ordered Set with Predetermined Path: To evaluate the placement of partially-ordered middlebox sets on predetermined paths, we use the proposed heuristic to convert the partially-ordered sets to fully-ordered sets, and then apply TOSP. We adjust the lookahead parameter from one to two and compare their performances. The traffic changing ratios and the dependencies of the two candidate sets of middleboxes are: $\{1.1 \leftarrow 0.8, 1.2 \leftarrow 0.7\}$ and $\{1.1 \leftarrow 1.2, 1.3 \leftarrow 0.7\}$. Note that different total order chains will be generated when using the two different lookahead values.

Fig. 4.7(a) compares the end-to-end delay of the two different lookahead values. We can see that the lookahead value of two achieves shorter delays with a deeper



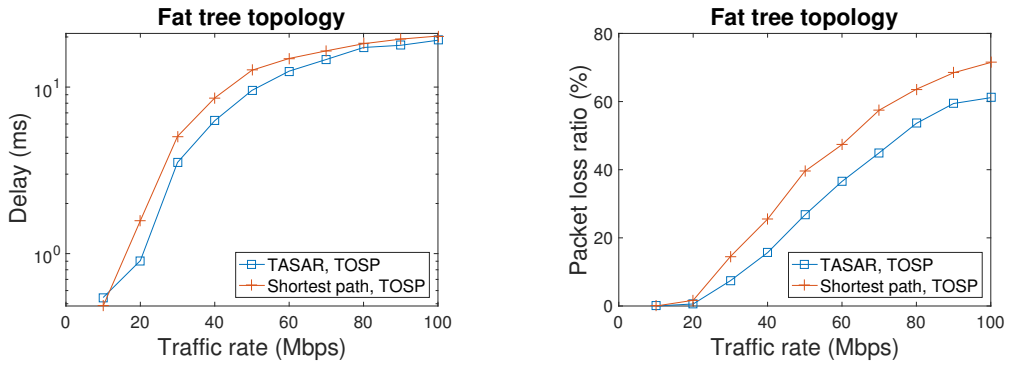
(a) End-to-end delay. (b) Packet loss ratio.

Figure 4.7: Partial to total order conversion simulation results.

search into the solution space. Similarly, Fig.4.7(b) shows that the lookahead value of two achieves lower packet loss ratios.

Placing Middleboxes without Predetermined Path: Finally, we evaluate the Traffic And Space Aware Routing (TASAR) heuristic by comparing it with a hop count based and ECMP (i.e., load-balance) enabled shortest path routing algorithm. For the multi-path topology, we choose an 8-pod fat tree with 80 switches and 128 hosts. Each host generates a flow to a random destination out of its own pod. The baseline traffic rate of each flow ranges from 10 to 100 Mbps with a stride of 10 Mpbs. The two sets of candidate middleboxes after conversion are: $\{1.2 \leftarrow 0.7 \leftarrow 1.1 \leftarrow 0.8\}$ and $\{1.3 \leftarrow 0.7 \leftarrow 1.1 \leftarrow 1.2\}$.

We first compare the routing success ratios of the two algorithms. We adjust the number of spaces per switch from 6 to 8, and calculate the percentage of flows that can successfully find paths with sufficient middlebox spaces. As shown in Table 4.1, when the space number per switch is 6, the routing success ratio of TASAR is 5% higher than that of shortest path routing. When the number increases 7, TASAR achieves a 100% routing success ratio, while shortest path routing cannot find path for 3.75% of flows. Finally, when it increase to 8, both algorithms achieve 100%



(a) End-to-end delay. (b) Packet loss ratio.

Figure 4.8: TASAR simulation results.

routing success ratios.

Spaces per switch	TASAR	Shortest path routing
6	93.75%	88.91%
7	100%	96.25%
8	100%	100%

Table 4.1: Flow routing success ratio.

Next, we fix the space number per switch to 8, and compare the end-to-end delay and packet loss ratio of the two algorithms. Fig. 4.8(a) shows that, when the baseline flow rate is 10 Mbps, TASAR has a slight longer delay, because its paths are not as shortest as those of shortest path routing. However, once the flow rate increases beyond 10 Mbps, TASAR consistently delivers shorter delays due to its traffic awareness in path calculation. Fig. 4.8(b) also shows that TASAR consistently achieves lower packet loss ratios.

4.5.2 Experiment Results with Prototype

To validate our design, we first implement the algorithms TOSP, TASAR and shortest path routing in a module running on Floodlight. Then we conduct experiments by running the module on the top of the prototype system described in Section 3.5.

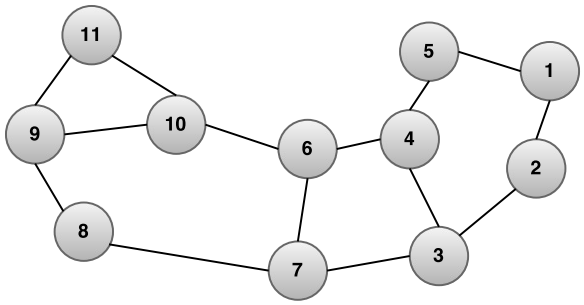
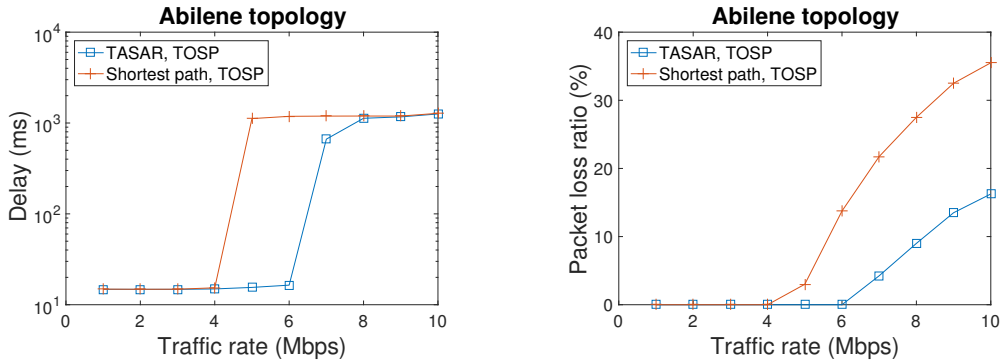


Figure 4.9: Abilene backbone network topology.



(a) End-to-end delay. (b) Packet loss ratio.

Figure 4.10: Prototype experiment results.

We pick the Abilene backbone topology [abi], as shown in Fig 4.9. Each node has a space capacity of three, and each link has a bandwidth capacity of 10 Mbps. For traffic generation, we use Iperf to create constant bit rate UDP traffic flows. Four flows are generated: two from node 1 to 8, and two from 11 to 2. The initial traffic rate of each flow ranges from 1 to 10 Mbps with a stride of 1 Mbps. Each flow needs four middleboxes with the following traffic changing ratios and total order chain after conversion: $\{1.2 \leftarrow 0.7 \leftarrow 1.1 \leftarrow 0.8\}$.

As can be seen in Fig. 4.10(a), the experiment data is consistent with the simulation results, and TASAR achieves shorter end-to-end delays than the shortest path routing. Also, Fig. 4.10 (b) shows that the former achieves much lower packet loss ratios.

To validate our design with some real middlebox functions, we further developed

three types of middlebox emulators: the first type performing compression function, the second type performing encryption function, and the third type keeping the data unchanged. For convenience, we name them as *compression middlebox emulators*, *encryption middlebox emulators*, and *transparent middlebox emulators* respectively. The implementation details of the three types of middleboxes are described below.

Compression Middlebox Emulator: We developed the emulator using the *libpcap* and *zlib* [lzi] libraries. The emulator uses *libpcap* APIs to capture traffic on specified interfaces. It can capture specific traffic (e.g., only UDP packets, only packets going to port 80, etc) by applying a rule set. Every time the emulator gets a new packet, it calls the *compress* method provided by the *zlib* library to compress the payload of the packet. In order to accurately locate the payload, we defined the structure of the packet header. After the payload is compressed, the emulator calls the *libpcap* APIs to send the processed packet back to the interface, where the original packet is captured. Since the content of each packet may be different, the compression ratio changes over time. For the ratio setting of a compression middlebox emulator in our algorithms, we take the average value. For example, if we send an X-MByte file through a compression middlebox emulator, and then the size of the processed file changes to Y MBytes, then the ratio of the middlebox is Y/X (with a precision of one decimal place).

Encryption Middlebox Emulator: For the implementation of the encryption middlebox emulator, some of the features in the compression middlebox emulator (e.g., packet capturing) can be reused. We omit the same features, and only highlight differences. Unlike the previous emulator, once a packet is received, the emulator calls the *AES_encrypt* method provided by the library *OpenSSL* [opec] to encrypt the payload of the packet. The emulator performs Advanced Encryption standard (AES) encryption on data. In detail, we use AES-128 bit encryption, where 128 bit

is AES key length. The length of the output of AES encryption algorithm is related to the length of the input and a static variable *AES_BLOCK_SIZE*, whose value is 16. Since the ratio of output length to the input length is a variable, we take an average as the emulator's ratio. The method of calculating the average value is the same as the one for compression middlebox emulators.

Transparent Middlebox Emulator: For some middleboxes with a ratio of 1, they do not change the size of the data that flows through them. As we mentioned above, we can set a *libpcap* rule set. Packets matching the set of rules will be forwarded directly. The emulator is similar to a firewall, which can allow the specified flows and keep them unchanged.

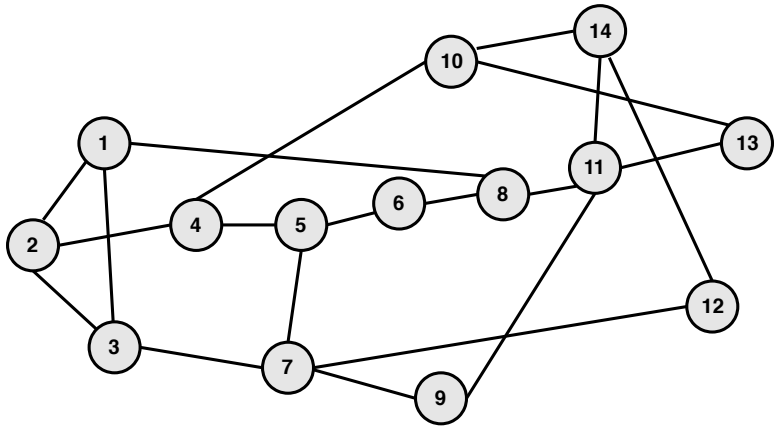
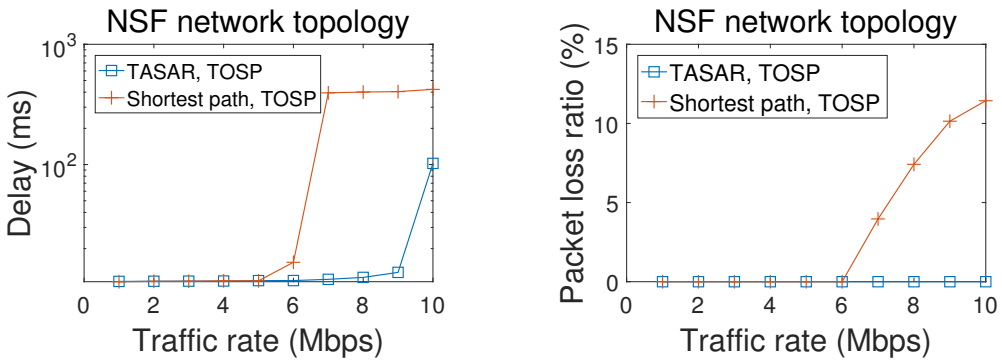


Figure 4.11: NSF network topology.

Flow #	**Node Pair**	**Routes (calculated by shortest routing)**
1	$(1, 2)$	1-2
2	$(1, 8)$	1-8
3	$(2, 7)$	2-3-7
4	$(7, 12)$	7-12
5	$(8, 14)$	8-11-14
6	$(13, 14)$	13-11-14

Table 4.2: Flow and Routes



(a) End-to-end delay. (b) Packet loss ratio.

Figure 4.12: Prototype experiment results.

For experiments with the above emulators, we select the NSF network topology cited from the paper "Demand-aware network function placement" [LZTM16], as shown in Fig.4.11 which has 14 switches and 21 links. Each switch has an attached NFV server with three middlebox spaces, and each link has a bandwidth capacity of 10 Mbps. For traffic generation, we create six constant bit rate UDP traffic flows, by sending a 400-MByte *.txt* file using Iperf. The initial traffic rate of each flow ranges from 1 to 10 Mbps with a stride of 1Mbps. Each flow needs to go through three middleboxes with corresponding middlebox chain: $\{0.8 \leftarrow 1.0 \leftarrow 1.2\}$. As a result, the combined traffic changing effects of both middleboxes is $0.8 \cdot 1.0 \cdot 1.2 = 0.96$. The source and destination node pair of each flow is shown in Table 4.2, cited from the same paper [LZTM16]. The flow paths calculated by shortest routing are also shown in Table 4.2.

As shown in Fig. 4.12(a), the experiment results are similar to the previous ones. TASAR achieves shorter end-to-end delays than shortest path routing. Fig. 4.12(b) shows that the former has no packet loss. In detail, any two paths calculated by TASAR routing does not include the same link. That means the flow rate is less than or equal to $0.96\times$ on each link of the flow path, and then there is no network congestion. In addition, for middlebox placement, TOSP may have more

alternative nodes on the paths calculated by TASAR to achieve better results. By contrast, shortest path routing has higher delays and packet loss ratios, due to network congestion and different middlebox placement.

4.6 Summary

The advancement of virtualization technology has made NFV a promising platform for network function provisioning. However, the flexibility to run an NFV middlebox on any available standard server also creates a challenge for efficient NFV implementation. In this chapter, we have studied the optimal placement of NFV middleboxes by considering different middlebox traffic changing effects and dependency relations. We first formulate the Traffic Aware Placement of Interdependent Middleboxes problem as a graph optimization problem with the objective to load-balance the network. Next, we solve the problem when the flow path is predetermined, and propose optimal algorithms for a non-ordered or totally-ordered middlebox set. For the general scenario of a partially-ordered middlebox set, we show that the problem is NP-hard by reduction from the Clique problem, and propose an efficient heuristic to convert a partially-ordered set to a totally-ordered one. On the other hand, when the flow path is not predetermined, we prove that the studied problem is NP-hard even for a non-ordered middlebox set by reduction from the Hamiltonian Cycle problem, and propose the Traffic And Space Aware Routing heuristic. We have conducted large scale simulations to evaluate the proposed solutions, and have also implemented an SDN based prototype to validate them in realistic environments. Extensive simulation and experiment results are presented to show the effectiveness of our design.

CHAPTER 5

SERVICE AWARE FLOW ROUTING

Network Function Virtualization (NFV) implements network functions as Virtual Machines (VMs) running on standard commodity servers, and enjoys many benefits thanks to the underlying virtualization technology. However, since each VM has only a limited processing capacity restricted by its available resources, multiple instances of the same function are necessary in an NFV network. Thus, routing in an NFV network is a challenge to determine not only a path from the source to destination but also the service locations. Furthermore, this challenge is complicated by the traffic changing effects of NFV services and dependency relations between them. In this chapter, we study how to efficiently route a flow to receive services [1] in an NFV network. First, we formulate the Service-Aware Routing (SAR) problem as a graph optimization problem, and prove that it is NP-hard. Next, for the special scenario when the required set of services is a totally-ordered set, we propose a polynomial-time algorithm and prove its optimality. For the NP-hard general scenario of a partially-ordered service set, we propose two practical heuristics with low time complexity, one by converting the partially-ordered set to a totally-ordered one, and the other using a greedy approach. We have evaluated our design using prototype based experiments and large scale simulations. Extensive results are presented to demonstrate the effectiveness of the proposed algorithms.

5.1 Introduction

Network functions, such as firewalls and proxies, are widely deployed in data centers, and are traditionally implemented as proprietary hardware appliances [etsb].

[1] In this chapter, the term "service" has the same meaning as "middlebox".

Instead, NFV implements such network functions as VMs running on standard commodity servers [V. 12]. Compared with traditional hardware appliances, NFV brings many advantages, including accelerated time-to-market, reduced hardware and operation cost, and elastic scalability [etsc], thanks to the underlying virtualization technology.

Unlike hardware network appliances that increase processing capacities by adding more hardware resources, an NFV network adds multiple VM instances to raise the processing capacity of a function [Y. 17], as each VM has only a limited processing capacity restricted by its available resources. For instance, while the Palo Alto Networks's PA-7080 hardware firewall delivers a throughput of 200 Gbps [pa-a], its VM-series virtual firewalls achieve a throughput ranging from 200 Mbps to 16 Gbps [pa-b].

Because multiple instances of the same function may be hosted by different physical servers, routing a flow in an NFV network needs to not only find a path from the source to destination, but also determine the service locations, i.e., where a flow will receive its desired services. Bad decisions may cause inefficient flow paths and performance degradation. In Fig. 5.1(a), a flow goes from the source src to destination dst, and requires the service s. Among the three candidate paths, the middle one is the optimal, since it traverses an instance of s and has only two hops.

Furthermore, the NFV routing challenge is complicated by the traffic changing effects of different services and dependency relations between them. To start with, a network service may increase or decrease the traffic volume of a processed flow. For instance, the Citrix CloudBridge WAN optimizer compresses traffic to 20% of its original volume before sending it to the next hop [wan], while a Cisco IPSec VPN proxy adds up to 73 bytes of overhead for each processed packet [ips]. In Fig. 5.1(b), assume that a flow has an initial traffic rate of 1, and requires two services

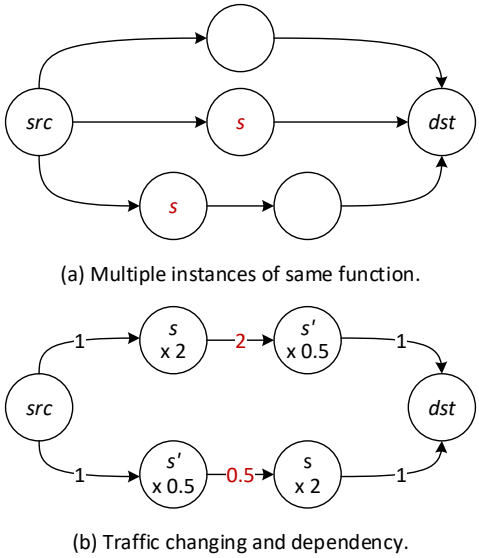


Figure 5.1: Service aware routing challenge in NFV networks.

s and s'. The former will double the traffic volume and the latter will cut it to half. Comparing the two candidate paths, the bottom one is more efficient, because the load of its second path link is 0.5 instead of 2 as in the top path.

The second constraint faced by routing in an NFV network is the dependency relation that may or may not exist between a pair of services. For instance, an IPSec VPN proxy is usually placed before a NAT gateway to terminate tunneling [cis], while it can be placed either before or after a firewall [ms-]. For the example in Fig. 5.1(b), if there is a constraint that service s must be placed before s', then the bottom path that was more efficient is now invalid.

In this work, we study the Service-Aware Routing problem in NFV networks that finds the optimal path for a flow to efficiently traverse a deployed instance for each of its required services. Different from existing works [Y. 17, V. 17, T. 16, M. 17b, Q. 17, DW16, MMP15, MSB+17, MBP+17] that study the deployment of NFV VM instances, this work optimizes routing among the already deployed NFV instances, which is critical for delay-sensitive flows that cannot tolerate the VM

instantiation delay. Furthermore, this work considers the traffic changing effects of different services and dependency relations between them, and also proposes practical solutions with an implemented prototype. To the best of our knowledge, no existing work has studied the traffic changing effects for routing flows among deployed NFV instances.

Our solutions leverage the emerging Software-Defined Networking (SDN) technology as the implementation platform, as it enables efficient network optimization by decoupling the network control plane from the data plane. We have developed an SDN based prototype to demonstrate the practicality of our design and validate it in realistic environments.

The main contributions in this work are summarized as follows.

1. We formulate the Service Aware Routing (SAR) problem for NFV networks as a graph optimization problem with the objective to load-balance the network, and prove that it is NP-hard by reduction from the Hamiltonian Cycle problem.
2. For the special scenario when the set of required services is a totally-ordered set, we propose an efficient polynomial-time algorithm, and show its optimality.
3. For the NP-hard general scenario of a partially-ordered service set, we propose two practical heuristics: one by converting the partially-ordered set to a totally-ordered one, and the other using a greedy approach.
4. We have evaluated our design using prototype based experiments and large scale simulations. Extensive experimental and simulation results are presented to demonstrate the effectiveness of the proposed algorithms.

5.2 Problem Formulation

In this section, we formulate the service aware routing (SAR) problem. Use S to denote the set of NFV services. Each service $s \in S$ has an associated traffic changing ratio $ratio[s]$[2], which is the (average) ratio of the traffic rate of a flow after and before being processed by s. The *dependency* relation $\leftarrow$ is defined as a strict partial order on S that is

1. Irreflexive: $s \nleftarrow s$,
2. Transitive: $s \leftarrow s'$ and $s' \leftarrow s''$ then $s \leftarrow s''$, and
3. Asymmetric: $s \leftarrow s'$ then $s' \nleftarrow s$.

If $s \leftarrow s'$, we say that the service s' depends on s, or intuitively s' must be applied after s. If s depends on no other service, i.e., $\forall s' \in S, s' \nleftarrow s$, we say that s has no dependency. For easy representation, define $depend[s, s'] = 1$ if $s \leftarrow s'$, and 0 otherwise.

Consider a network modeled as a directed graph $G = (V, E)$. A node $v \in V$ may have existing instances of various services. Use $pc[v, s]$ to denote the available processing capacity of service s at v, which may be the aggregate capacity of multiple instances located at v. For a link $(u, v) \in E$, use $bc[u, v]$ to denote its remaining bandwidth capacity. The existing load of the link is denoted as $load[u, v]$.

For route calculation, each link $(u, v) \in E$ is assigned a weight, denoted as $weight(u, v, l)$, which is a non-decreasing function of the link load l, i.e., $\forall l \leq l', weight(u, v, l) \leq weight(u, v, l')$. A broad category of weight functions satisfy the non-decreasing requirement, such as the ones used by the popular Cisco EIGRP [eig]

[2]We use square brackets [] to denote properties or known values, and round brackets () denote to functions or variables.

and OSPF [osp] protocols. The non-decreasing link weight function helps load-balance network traffic when the routing protocol aims to minimize the path cost, which is defined as the weight sum of all the path links.

A flow f is denoted as a 4-tuple (src, dst, t, S), in which $src \in V$ is the source node, $dst \in V$ is the destination node, t is the initial traffic rate at the ingress point of the network, and S is the set of required services.

When the flow f enters the network, a multi-hop path, denoted as $route$, will be assigned for the flow, which is a decision variable defined as

$$route(v, i) = \begin{cases} 1, & \text{if } v \in V \text{ is the } i^{th} \text{ hop on the path of } f. \\ 0, & \text{otherwise.} \end{cases} \tag{5.1}$$

We define i to start from one, and denote the last hop number as n for convenience. Note that repeating nodes are allowed on the path to enable more general solutions. To avoid performance degradation for TCP flows, a flow is not allowed to be split among multiple paths [B. 10].In addition, a service scheme, denoted as $service$, determines the service location for each required service $s \in S$ of flow f, which is a decision variable defined as

$$service(s, i, j) = \begin{cases} 1, & \text{if } s \in S \text{ is served at the } i^{th} \text{ hop} \\ & \text{as the } j^{th} \text{ service of that hop.} \\ 0, & \text{otherwise.} \end{cases} \tag{5.2}$$

Note $j \leq |S|$ since there are at most $|S|$ services.

Use $t_{in}(i)$ and $t_{out}(i)$ to denote the incoming and outgoing traffic rate of flow f at the i^{th} hop on the path, respectively. If f is processed by a single service s at the i^{th} hop, then $t_{out}(i) = t_{in}(i)ratio[s]$. Note that the incoming traffic rate at the flow source is the initial traffic rate, i.e., $t_{in}(1) = t$. For convenience, use

$t(u,v) = \sum_{i\in[1,n-1]} route(u,i)route(v,i+1)t_{out}(i)$ to represent the traffic rate of f on its path link (u,v).

Consistent with most popular routing protocols, such as Cisco EIGRP and OSPF, our optimization objective is to minimize the path cost of flow f as in Equation (5.3), by calculating *route* and *service*. The proposed solutions can easily adapt to other optimization objectives, such as minimizing the maximum link load in the network.

$$\begin{aligned}&\textbf{min}\sum_{i=1}^{n-1}\sum_{u\in V}\sum_{v\in V} route(u,i)route(v,i+1)\times\\ &weight(u,v,t(u,v)+load[u,v])\end{aligned} \tag{5.3}$$

subject to the following constraints:

$$\forall i > n, \forall v \in V : route(v,i) = 0 \tag{5.4}$$

$$route(src,1) = 1, route(dst,n) = 1 \tag{5.5}$$

$$\forall i < n : t_{in}(1) = t, t_{in}(i+1) = t_{out}(i) \tag{5.6}$$

$$\begin{aligned}&\forall i \le n, \forall j < |S| : \hat{t}(i,1) = t_{in}(i),\\ &\hat{t}(i,j+1) = \hat{t}(i,j)(1+\sum_{s\in S} service(s,i,j)(ratio[s]-1)),\\ &t_{out}(i) = \hat{t}(i,|S|+1)\end{aligned} \tag{5.7}$$

$$\begin{aligned}&\forall v \in V, \forall s \in S :\\ &\sum_{i\in[1,n]}\sum_{j\in[1,|S|]} route(v,i)service(s,i,j)\hat{t}(i,j) \le pc[s,v]\end{aligned} \tag{5.8}$$

$$\forall (u,v) \in E : t(u,v) \le bc[u,v] \tag{5.9}$$

$$\forall s \in S : \sum_{i\in[1,n]}\sum_{j\in[1,|S|]} service(s,i,j) = 1 \tag{5.10}$$

$$\forall s, s' \in S : \Big(\sum_{i'\in[1,n]}\sum_{j'\in[1,|S|]} service(s',i',j')(i'|S|+j') -$$

$$\sum_{i\in[1,n]}\sum_{j\in[1,|S|]} service(s,i,j)(i|S|+j)\Big) \times$$
$$depend[s,s'] > 0 \tag{5.11}$$

Equation (5.3) defines the optimization objective to be the path cost, which is the weight sum of all links on the path. Equation (5.4) defines the n^{th} hop to be the last hop on the flow path. Equation (5.5) enforces the first hop and last hop of the flow path to be the source src and destination dst, respectively. Equation (5.6) defines the incoming traffic rate $t_{in}(i)$ of a flow f at each hop i on its path. Equation (5.7) defines $\hat{t}(i,j)$ to be the traffic rate of a flow f before it is processed by the j^{th} service at the i^{th} hop on its path, and $t_{out}(i)$ to be the traffic rate after being processed by the last service at the i^{th} hop. Equation (5.8) states that, for the service s instances deployed at node v, the total amount of processed traffic should not exceed the available processing capacity $pc[v,s]$. Equation (5.9) states that, for a link (u,v), the traffic rate $t(u,v)$ of the flow on this link should not exceed the available bandwidth capacity $bc[u,v]$. Equation (5.10) states that the flow should be processed by a service s once and only once. Equation (5.11) enforces the dependency relation between services, or in other words s' must be placed after s if the former depends on the latter.

As can be seen, our formulation for the SAR problem considers only a single flow. The reason is that the SAR problem with a single flow is already NP-hard as will be seen in Section 5.3, and the SAR problem with multiple flows will be even harder and meaningless. Instead, the proposed algorithms for a single flow are of practical interest, because in reality flows tend not to arrive at the exactly same time, and even if multiple flows arrive simultaneously in an SDN network, the central controller will have to process them one by one. Furthermore, given the algorithms for a single flow, multiple flows can be processed one at a time.

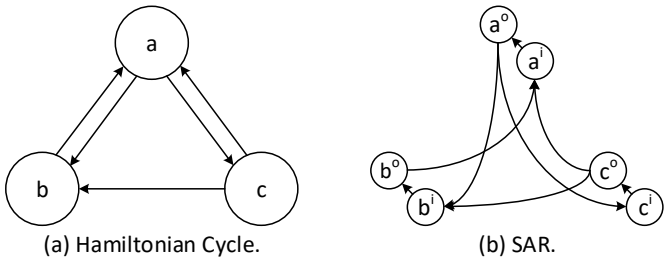


Figure 5.2: Reduction from Hamiltonian Cycle to SAR.

5.3 Algorithm Design

In this section, we first prove that the SAR problem formulated in Section 5.2 is NP-hard by reduction from the Hamiltonian Cycle problem. Then for the special scenario when the service set is a totally-ordered set, we propose an efficient polynomial-time algorithm and prove its optimality. On the other hand, for the NP-hard general scenario, we propose two practical heuristics with low time complexity, one by converting the partially-ordered service set to a totally-ordered one, the other using a greedy approach. Finally, we also discuss the scenario when the existing processing capacity of a service is insufficient.

5.3.1 NP-Hardness Proof

The following theorem shows the hardness of the SAR problem.

Theorem 5.3.1 *The Service Aware Routing problem is NP-hard.*

Proof. The proof of this theorem is similar to the proof of Theorem 4.4.1 in Chapter 4. For ease of reading, we give a detailed proof below.

Given an instance of the Hamiltonian Cycle problem with a graph G , we construct an instance of the SAR problem with a graph $G' = (V', E')$ and a flow f as follows.

- For each node $v \in V$, the service set S contains a service s_v corresponding to it. The traffic changing ratio of any service is set to one, i.e., $ratio[s_v] = 1$. There is no dependency between the services.
- For each node $v \in V$, create two nodes $v^i, v^o \in V'$, where v^o has an instance of service s_v with a processing capacity of one, i.e., $pc[v^o, s_v] = 1$. Connect the two nodes with a link $(v^i, v^o) \in E'$, and set its weight to one, i.e., $\forall l, w[v^i, v^o, l] = 1$, and bandwidth capacity to one, , $bc[v^i, v^o] = 1$.
- For each link $(u, v) \in E$, create a link $(u^o, v^i) \in E'$, and set its weight to zero, i.e., $\forall l, w[u^o, v^i, l] = 0$, and bandwidth capacity to one, i.e, $bc[u^o, v^i] = 1$. An example to create G' from G is shown in Fig. 5.2.
- For the flow f, its source and destination are both a^i, i.e., $src = dst = a^i$, where a is an arbitrary node in V. The initial traffic rate is one, i.e., $t = 1$. The set of required services is S.

Clearly, the above reduction process can be done in polynomial time. Next, we show that if G has a Hamiltonian cycle, then the SAR instance with G and f has a viable path with a cost of $|V|$. Given the Hamiltonian cycle of G we construct a similar path *route* in G' as follows. Assuming that the Hamiltonian cycle in G starts with $a \in V$, for each node v and link (u, v) in the Hamiltonian cycle, add links (v^i, v^o) and (u^o, v^i) to *route*, respectively. For the example in Fig. 5.2, G has a Hamiltonian cycle $a \Rightarrow c \Rightarrow b \Rightarrow a$, and the constructed *route* in G' is $a^i \Rightarrow a^o \Rightarrow c^i \Rightarrow c^o \Rightarrow b^i \Rightarrow b^o \Rightarrow a^i$. Since the Hamiltonian cycle traverses each node in G exactly once, and each node v in G maps to a pair of nodes v^i and v^o in G', *route* traverses each node in G' exactly once as well. Thus, we can see that *route* traverses all the $|V|$ required service instances. Further, *route* traverses any link in G' at most once, resulting in a path cost of $|V|$.

Reversely, given *route* in G', we construct a Hamiltonian cycle in G as follows. Starting with $src = a^i$, sequentially add the corresponding node $v \in V$ of each incoming node $v^i \in V'$ on *route* to the cycle in G. Since *route* traverses all outgoing nodes v^o to reach the instance of service s_v, the constructed cycle traverses all the nodes in G. Further, since the path cost is $|V|$, *route* traverses each link of (v^i, v^o) for any v at most once. Thus, the constructed cycle in G traverses each node exactly once, and is a Hamiltonian cycle. To ensure that a path of SAR can be converted to a path of the Hamiltonian Cycle problem, we add the link (v^i, v^o) to G' to detect whether a node is traversed multiple times. □

5.3.2 Optimal Routing for Totally-Ordered Service Set

Although the general SAR problem is NP-hard, we are able to design a polynomial-time optimal algorithm called *Totally-Ordered Set Routing* (TOSR), for the special scenario when the set of required services is a totally-ordered set. Specifically, the service set S of a flow f has a total order if $\forall s, s' \in S$, either $s \leftarrow s'$ or $s' \leftarrow s$, or in other words the services form a dependency chain. For convenience, we denote the k^{th} service in the chain as s_k. For instance, $s_1 \leftarrow s_2 \leftarrow s_3$ is a total order chain, in which s_1 must be applied before s_2, and s_2 before s_3.

Given the determined order of services, the flow has to visit an instance of each service one by one. Thus, the basic idea of TOSR is to use an iterative approach to find the least cost path to the instances of each service in the total order chain.

In detail, TOSR starts its first iteration from the flow source src, and searches for the least cost path to each instance of the first service s_1. The traffic rate of the flow in this portion of the path, i.e., before being processed by s_1, is $t_{in}(1) = t$. Thus, Dijkstra's algorithm is applied to find the least cost path with more than $t_{in}(1)$ bandwidth to each instance of s_1 with more than $t_{in}(1)$ processing capacity.

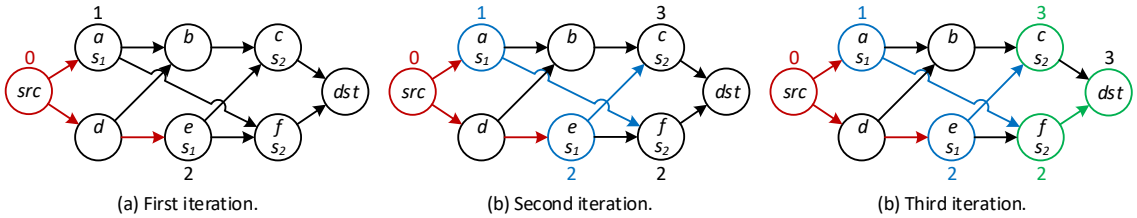


Figure 5.3: Totally-ordered set routing example.

An example is given in Fig. 5.3(a). For simplicity, assume that each link has the same weight of a unit and a sufficient bandwidth capacity, and each service instance has a sufficient processing capacity. In the example, the first iteration of TOSR finds the least cost paths from the source to the two instances of s_1 at nodes a and e, the former with a cost of 1 and latter of 2.

After finishing the $(k-1)^{th}$ iteration, TOSR has found the least cost path to each instance of the $(k-1)^{th}$ service s_{k-1} in the total order chain. In the k^{th} iteration, TOSR will continue from the instances of s_{k-1}, and search for the least cost path to each instance of the next service s_k. This is done by a revised version of Dijkstra's algorithm. For nodes with s_{k-1} instances, their initial costs will be those calculated from the $(k-1)^{th}$ iteration, i.e., the cost so far from the source src to each s_k instance; for the remaining nodes, their initial costs are set to infinity. In addition, the traffic rate in this portion of the path, i.e., after leaving an s_{k-1} service instance and before entering an s_k service instance, is $t_{in}(k)$. After initialization, TOSR uses a dynamic programming approach to iteratively find the least cost path with a sufficient bandwidth capacity to each of the remaining nodes and stop when the least cost paths to all the s_k instances have been found. Note that the k^{th} iteration of TOSR runs the revised Dijkstra's algorithm only once, because it calculates the least cost paths from the single source src to each s_k instance instead of all-pair least cost paths. In Fig. 5.3(b), the second iteration initializes the costs of nodes

a and e as 1 and 2, respectively, based on the results from the first iteration, and finds the least cost paths to the two instances of s_2 at nodes c and f, the former with a cost of 3 and latter of 2.

In a similar way, after finishing the $|S|^{th}$ iteration, TOSR has found the least cost path to each instance of the last service $s_{|S|}$. In the final iteration, TOSR calculates the least cost path from the instances of $s_{|S|}$ to the destination dst, and stops immediately once one such path is found. In Fig. 5.3(c), the final iteration initializes the costs of nodes c and f as 3 and 2, respectively, and finds the least cost path to dst with a cost of 3.

A shortcut can be taken by TOSR if the source src has an instance with a sufficient processing capacity for each of the first K services. Since the flow can receive the first K services at the source with a path cost of zero, TOSR can skip the first K iterations, and continue from the $(K+1)^{th}$ iteration.

Theorem 5.3.2 *The TOSR algorithm achieves the minimum path cost.*

Proof. The proof can be done by induction on the iteration number.

- Basis case: TOSR starts the first iteration from the source src with a cost of zero.
- Inductive case: If the $(k-1)^{th}$ iteration finds the least cost path to each instance of service s_{k-1}, Dijkstra's algorithm guarantees that the k^{th} iteration will find the least cost path to each instance of service s_k, and eventually to the destination dst.

□

The time complexity of the algorithm is $O(|S|(|E| + |V| \log |V|))$, because Dijkstra's algorithm complexity is $O(|E| + |V| \log |V|)$, and we need to run it for each of the $|S|$ services.

5.3.3 Converting Partially-Ordered Set to Totally-Ordered Set

Given the NP-hardness of the general scenario when the service set is a partially-ordered one, we propose two practical heuristic algorithms with low time complexity. Our first solution works in two steps by first converting the partially-ordered set to a totally-ordered one and then applying TOSR.

Based on the observation from the example in Fig. 5.1(b), the basic idea of the conversion algorithm is to arrange the services in the result total order chain in the increasing order of their traffic changing ratios, so that the traffic rate of the flow can be minimized on the path.

In detail, the conversion algorithm iteratively finds the services without dependency (e.g., 1.1 and 0.8 in the first iteration for the following example), removes among them the one with the minimum traffic changing ratio, and adds it to the end of the total order chain. For instance, given four services with the following traffic changing ratios and dependencies: $1.1 \leftarrow 0.7$ and $0.8 \leftarrow 1.2$, the conversion result will be the following total order chain: $0.8 \leftarrow 1.1 \leftarrow 0.7 \leftarrow 1.2$.

The time complexity of the conversion algorithm is $O(|S| \log |S|))$, because there are up to $|S|$ iterations, and the time complexity to select the service with the minimum traffic changing ratio is $O(\log |S|)$ using a heap. Once the totally-ordered service set has been obtained, the TOSR algorithm can be applied to calculate the flow path *route* and service locations *service*.

5.3.4 Greedy Routing for Partially-Ordered Service Set

Next, we propose a greedy heuristic for the general scenario of a partially-ordered service set as the benchmark for evaluation. The basic idea is to work in iterations to add the services without dependency one at a time. Note that the greedy heuristic

also applies to routing for a totally-ordered service set, since it is a special case of the general scenario.

In detail, the greedy algorithm starts by listing the candidates for the first service on the path, which are the services with no dependency. After finding the candidate services, the algorithm calculates the least cost path to each instance of those services. It then greedily picks the instance with the least cost, breaks a tie by selecting the service with the minimum traffic changing ratio, and removes the service from the service set S. Each iteration will find the path to the instance for one service and remove it from the set S. The iteration stops when S becomes empty. Finally, the algorithm finishes by finding the least cost path from the last service instance to the destination.

The time complexity of the algorithm is $O(|S|(|E| + |V| \log |V|))$, because there will be $O(|S|)$ iterations, each for a required service, and the time complexity of each iteration is $O(|E| + |V| \log |V|)$.

5.3.5 Mix of Existing and New Instances

In case none of the deployed instances of a service has a sufficient capacity to process the flow, a new instance needs to be started. In such a case, the flow path will traverse a mix of existing instances for some services and newly started instances for others. We extend the problem model in Section 5.2 with the following notations: $sc[v]$ to denote the space capacity of a node $v \in V$, $sr[s]$ to denote the space requirement of an instance of service $s \in S$, $pc[s]$ to denote the initial processing capacity added by a new instance of $s \in S$.

The first step of our solution to the mix problem is to apply the conversion algorithm to convert a partially-ordered service set, because the evaluation data in Section 5.4 show that the conversion algorithm working with TOSR consistently

outperforms the greedy heuristic. Next, if the services that need new instances are not consecutive in the total order chain, TOSR can be applied to obtain the optimal result by treating the function to start a new instance as a special service. In other words, when it is necessary to start a new instance for a service, TOSR finds the least cost paths from the previous iteration to the nodes with sufficient space capacities, and then from such nodes to the deployed instances of the next service in the total order chain. Otherwise, the problem is NP-hard, and the greedy heuristic can be applied. Whenever it is necessary to start an instance of a missing service, the greedy heuristic will search for the least cost path to a node with a sufficient space capacity.

5.4 Experimental And Simulation Results

In this section, we present extensive experimental and simulation results to demonstrate the effectiveness of the proposed algorithms.

We first implemented our algorithms on the top of the prototype system described in Section 3.5 of chapter 3, and then implemented the algorithms in the ns-3 simulator for performance evaluation in large scale networks. In the experiments and simulations, we use the same performance metrics as previous chapters. Since there is no existing algorithm for the SAR problem that considers various service traffic changing effects and dependency relations, our focus in this section will be the algorithms proposed in this chapter.

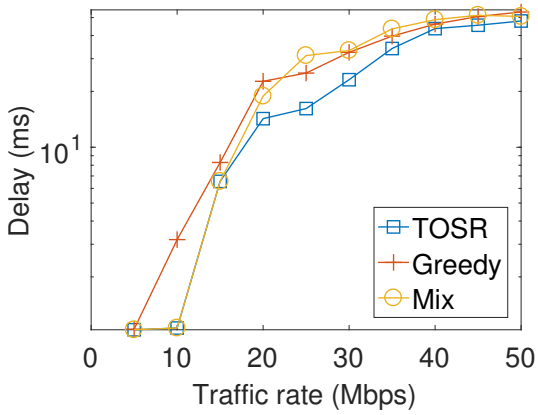
5.4.1 Simulation Results

The simulation settings in ns-3 are as follows. The fat tree topology is selected, since it is a popular choice among datacenter networks [AFLV08]. An 8-pod fat tree

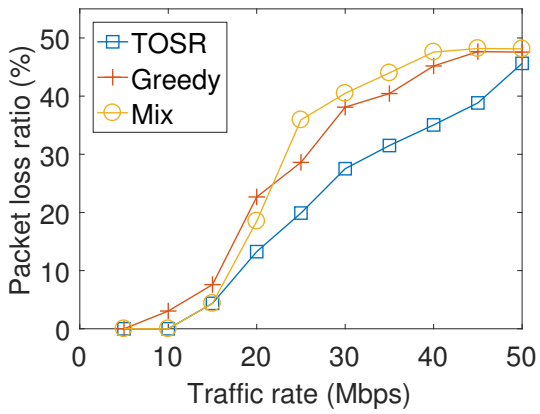
is set up with 80 switches and 128 hosts. Each link has a bandwidth capacity of 100 Mbps and a propagation delay of 2 μs. To mimic realistic networks, background traffic is generated that consumes 60% to 70% of the link capacity. We use the Cisco EIGRP link weight function [eig] by setting only K_2 to one. For traffic generation, we adopt the on-off model to reflect the burstiness of realistic traffic. When a flow f is in the on state its initial traffic rate t is the product of a baseline rate and a random number between 0.9 to 1.1; when in the off state, its traffic rate is zero. A flow is in each state for 50% of the time. The baseline traffic rate of each flow ranges from 5 Mbps to 50 Mbps with a stride of 5 Mbps. Each host sequentially generates a flow to a random host. Every simulation run lasts five minutes, and the result is the average of four runs.

Routing for Total-Ordered Service Set: First, for the routing of a totally-ordered service set, we compare the TOSR (Section 5.3.2), Greedy (Section 5.3.4), and Mix (Section 5.3.5) algorithms. The first two utilize only existing service instances, and the last one starts new instances if no existing instance has a sufficient processing capacity.

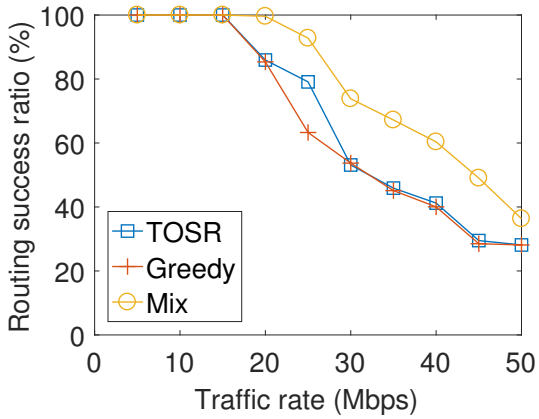
We consider four services with traffic changing ratios of 0.6, 0.9, 1.1, and 1.2, respectively, and a total of 4!=24 possible total order chains formed by the four services. Each flow randomly selects one of the 24 chains as its requested service set. NFV servers are connected to the switches, and each switch has a 50% probability to have an instance for each of four services, which has 60% to 90% of its 100 Mbps processing capacity remaining after processing the existing background traffic. In addition, each switch has a 25% probability to have a space capacity of one, which is sufficient to create a new instance of any service with an initial processing capacity of 100 Mbps.



(a) End-to-end delay.



(b) Packet loss ratio.



(c) Routing success ratio.

Figure 5.4: Simulation results for totally-ordered service set.

Fig. 5.4(a) compares the average end-to-end delays of the three algorithms. We can see that TOSR consistently outperforms Greedy because of its optimal route selection strategy as evidenced by Theorem 5.3.2. On the other hand, the delay of the Mix algorithm increases much faster as it accommodates more flows than the other two. In detail, when the traffic rate is less than or equal to 15 Mbps, TOSR and Mix achieve the same delay, which is shorter than that of Greedy. The reason for TOSR and Mix to have the same performance is that, when the traffic rate is small, the existing service instances have sufficient processing capacities to handle all flows, and thus no new instances are necessary. As the traffic rate increases, TOSR

still beats Greedy with a shorter delay, but the delay of Mix increases quickly and surpasses that of Greedy, because Mix is starting new instances to accommodate more flows and the increased amount of traffic causes a longer delay.

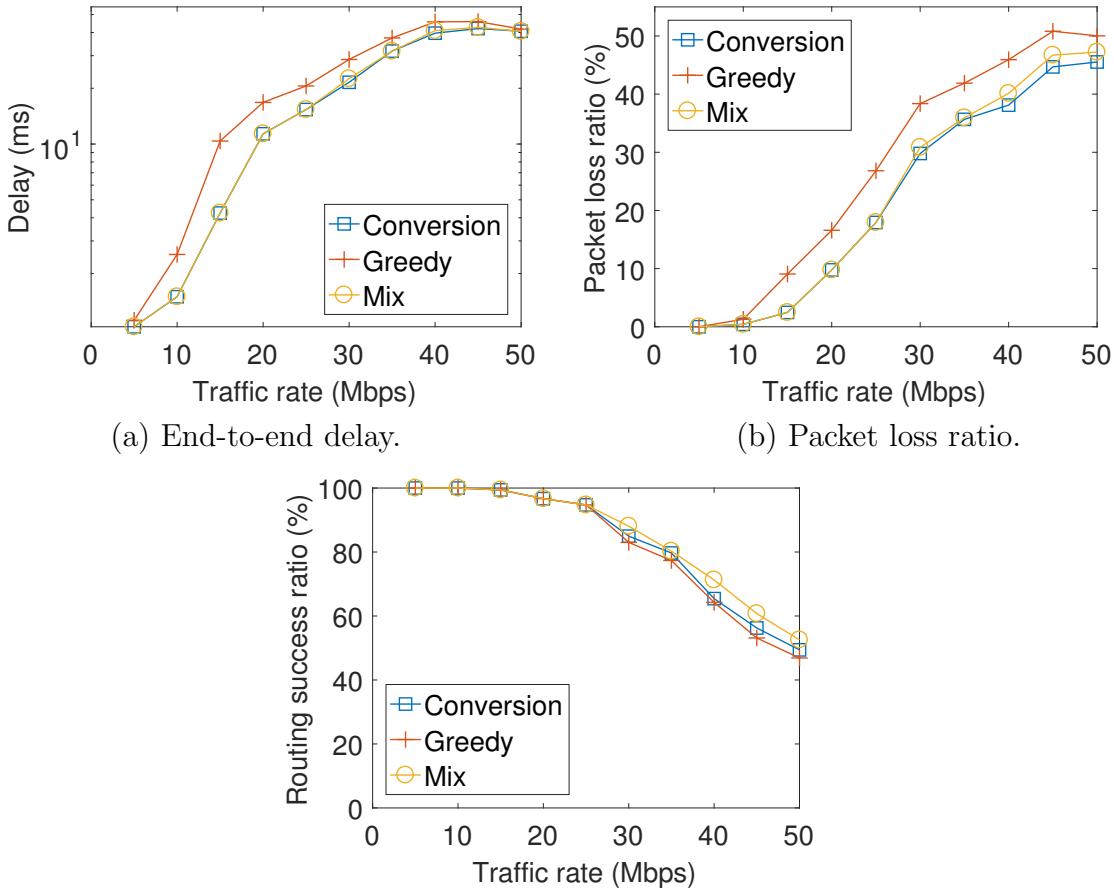
Fig. 5.4(b) illustrates the packet loss ratio, and a similar conclusion can be drawn. TOSR achieves a lower packet loss ratio than that of Greedy because of better balanced traffic. Mix initially achieves performance on par with that of TOSR, but has the highest packet loss ratio under high traffic rates due to a greater number of admitted flows.

Fig. 5.4(c) shows the routing success ratios of the three algorithm. Consistent with the observation in Fig. 5.4(a), when the flow traffic rate is no more than 15 Mbps, all the three algorithms have a 100% routing success ratio. When the traffic rate grows above 20 Mbps, Mix consistently achieves the highest routing success ratio because of the extra instances started, and on the other hand, TOSR achieves a similar but slightly better routing success ratio than that of Greedy given the same amount of available resource.

Routing for Partially-Ordered Service Set: Next, for the general scenario of a partially-ordered service set, we compare the Conversion (Section 5.3.3, working with TOSR), Greedy, and Mix algorithms.

We consider seven different services with traffic changing ratios of 0.6, 0.7, 0.8, 0.9, 1.1, 1.2, and, 1.3, respectively. For each flow, four random services are selected to create a random directed acyclic graph that defines the dependency relations between services.

Similar as above, each switch has a 50% probability to have an instance for each of the seven services, and a 25% probability to be able to start a new instance.



(a) End-to-end delay.

(b) Packet loss ratio.

(c) Routing success ratio.

Figure 5.5: Simulation results for partially-ordered service set.

Fig. 5.5(a) shows the average end-to-end delays of the three algorithms. We can see that the Conversion algorithm is superior to Greedy, because it considers the traffic changing ratios in the conversion process and then applies the optimal TOSR algorithm. Mix has an almost coincident curve with Conversion, since it works in a similar way by first converting the partially-ordered set to a totally-ordered one. Compared with Fig. 5.4(a), the delay of Mix does not increase as significantly when the traffic rate increases, because Mix does not admit as many additional flows as in the simulations for the totally-ordered service set, also evidenced in Fig. 5.5(c). The reasons are twofold. First, due to the increased number of services, the average

processing capacity per service demanded by each flow is decreased. The system bottleneck thus switches from the service processing capacity to the link bandwidth capacity, for which Mix cannot help.

Second, with the increased number of services but the same number of available new instances, fewer instances can be started for each service, resulting in fewer additional admitted flows.

Fig. 5.5(b) shows the packet loss ratio. Again, Conversion outperforms Greedy with a lower packet loss ratio. Mix is almost on par with Conversion, except that it has a slightly higher ratio when the traffic rate is large due to the additional admitted flows.

Fig. 5.5(c) plots the routing success ratio. Due to the two reasons explained above, the ratios are overall higher than the results for the totally-ordered set, and the improvement made by Mix is not as significant. Still, the pattern is clear that Conversion performs better than Greedy because of its traffic awareness in route calculation, and worse than Mix because of the additional instances started by the latter.

5.4.2 Experimental Results with Prototype

To demonstrate the practicality of our solution, we have implemented the proposed algorithms as a module running on the Floodlight controller and conducted experiments on the top of the prototype system described in Section 3.4.3.

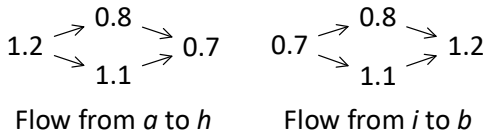


Figure 5.6: Partially-ordered service sets for experiments.

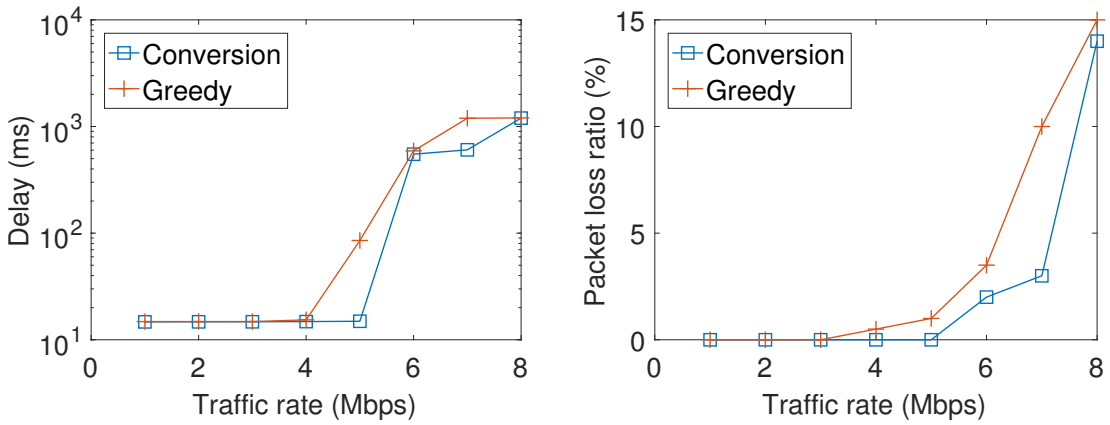
We consider four services with traffic changing ratios of 0.7, 0.8, 1.1, and 1.2, respectively, and pick the Abilene backbone topology [abi] with eleven nodes, as shown in Fig. 4.9. Eight nodes are randomly selected, each of which has the instances of the four services. Each instance has 60% to 90% of its 10 Mbps processing capacity remaining.

Each link has a bandwidth capacity of 10 Mbps. We use Iperf to generate constant bit rate UDP flows. Four flows without required services are generate as background traffic, with the information shown in Table 5.1. Two testing flows are generated: one from node a to h, and the other from i to b. Each flow has a partially-ordered service set with four services as shown in Fig. 5.6. The initial traffic rate of the two flows ranges from 1 Mbps to 8 Mbps with a stride of 1 Mpbs.

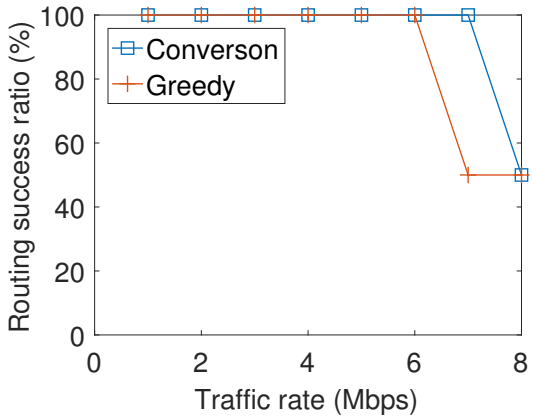
Flow #	Source	Destination	Traffic rate
1	e	b	4 Mbps
2	a	e	2 Mbps
3	d	c	6 Mpbs
4	f	g	6 Mbps

Table 5.1: Background flow information.

As can be seen in Fig. 5.7(a), the experiment results are consistent with the simulation ones, and Conversion achieves a shorter end-to-end delay than that of Greedy due to traffic awareness. Fig. 5.7(b) shows that the former also achieves a lower packet loss ratio. Finally, Fig. 5.7(c) shows that when the flow rate is no more than 6 Mbps, both algorithms can successfully route the two flows. When the traffic rate increases to 7 Mbps, Greedy can only successfully route one flow, and that happens to Conversion when the traffic rate increases to 8 Mbps.



(a) End-to-end delay. (b) Packet loss ratio.



(c) Routing success ratio.

Figure 5.7: Prototype experiment results.

5.5 Summary

The limited processing capability of a VM makes it necessary to deploy multiple NFV instances of the same service. Routing in NFV networks is thus a challenge to not only find a path from the source to destination, but also determine the optimal service locations. In this work, we have studied the service aware routing problem in NFV networks, and consider in particular the traffic changing effects of NFV services and dependency relations between them. First, we formulate the service aware routing problem as a graph optimization problem, and prove that it is NP-hard

by reduction from the Hamiltonian Cycle problem. Next, for the special scenario of a totally-ordered service set, we propose an efficient polynomial-time algorithm and prove its optimality. On the other hand, for the NP-hard general scenario of a partially-ordered service set, we propose two practical heuristics with low time complexity, one by converting the partially-ordered set to a totally-ordered one, and the other using a greedy approach. We have validated the design in an SDN based small-scale prototype, and also implemented the algorithms in the ns-3 simulator for large-scale performance evaluation. Extensive simulation and experimental results are presented to demonstrate the effectiveness of the proposed algorithms.

CHAPTER 6

CONCLUSIONS AND FUTURE DIRECTIONS

NFV enables flexible implementation of middleboxes, as VMs running on standard servers. However, the flexibility also creates a challenge for efficiently placing such middleboxes, due to the availability of multiple hosting servers, capabilities of middleboxes to change traffic volumes, and dependencies between middleboxes. Our proposed work focus on optimizing jointly the routing and the placement of NFV middleboxes, taking into account of capacity restrictions of NFV servers, to minimize the network congestion in software-defined networks.

With our first work (Chapter 3), we study the traffic changing effects of middleboxes, and propose SDN based non-ordered middlebox placement solutions to achieve optimal load balance. Our major contributions are: (1) we formulate the TAMP problem as a graph optimization problem with the objective to minimize the maximum link load, (2) we solve the TAMP problem when the flow paths are predetermined, such as in the tree topology, (3) we propose the LFGL rule and prove its optimality for a single flow, (4) for mutliple flows, we prove the problem is NP-hard by reduction from the 3-Satisfiability problem, and then propose an efficient heuristic, (5) for the general TAMP problem without predetermined flow paths, we prove that it is NP-hard even for a single flow by reduction from the Hamiltonian Cycle problem, and propose the LFGL based MinMax routing algorithm by integrating LFGL with MinMax routing. We build a prototype system to evaluate our proposed algorithms in a small scale network and also conduct extensive simulations in the ns-3 simulator. Both results fully demonstrate the effectiveness of our design.

With our second work (Chapter 4), we take account of dependency relations between middleboxes as a new constraint. The major contributions of this work are: (1) we formulate the TAPIM problem as a graph optimization problem, (2) when the

flow path is predetermined, we design optimal algorithms to place a totally-ordered middlebox set, (3) we propose an efficient heuristic for the general scenario of a partially-ordered middlebox set after proving its NP-hardness, (4) when the flow path is not predetermined, we show that the problem is NP-hard even for a non-ordered middlebox set, and propose a traffic and space aware routing heuristic. We have evaluated the proposed algorithms using large-scale simulations and prototype experiments, and present extensive results to show the effectiveness of our design.

Finally, with our third work (Chapter 5), we consider the scenario that some existed middleboxes still have capacities to process new flows. Therefore, some mice flows can take advantage of the remind process capacities of middleboxes. We initially formulate the problem and discuss its NP-hardness. When the set of required middleboxes of a new flow has a total priority order, we propose a polynomial time optimal solution. When the middlebox set has a partial order, we show that the problem is NP-hard by reduction from the Hamiltonian Cycle problem, and propose fast heuristics. We also will evaluate our proposed algorithms in ns-3 and our prototype.

To follow up with the work in my dissertation, some future work along the three directions are provided.

- Considering middleboxes may have traffic changing effects, we developed a middlebox emulator to emulate the effects. To further verify our algorithms, we developed several types of middleboxes with some real middlebox functions, and then apply them into our prototype test bed. The extra results in Chapter 4 show that our heuristic achieves better performance than the benchmark algorithms in small-scale experiments. But it is difficult for the heuristics to achieve optimal solutions even in small-scale experiments. In the future, we will try to explore whether we can convert our current problem

formulations into linear programming models. If they can be converted into linear programming models, we will pick some integer programming solvers to find optimal solutions in small-scale networks and compare our heuristics with the optimal solutions. If not, we will try to identify the approximation ratios of our heuristics and further improve them according to different scenarios.

- Hardware middleboxes are usually expensive, hard to operate and introduce significant energy consumption. The replacement of middleboxes by virtual network functions reduces energy consumption to some extent. In order to further optimize the energy consumption of the network, we will study how to schedule traffic and manage virtual network functions to maximize power efficiency.
- With the development of cloud computing and NFV technology, some enterprises outsource their middleboxes in the cloud to reduce costs and simplify middlebox management. We will study how to efficiently deploy and migrate middleboxes in clouds under resource constraints.

BIBLIOGRAPHY

[A. 08] A. Singh et al. Server-storage virtualization: integration and load balancing in data centers. In *ACM/IEEE Supercomputing*, 2008.

[A. 11] A. Curtis et al. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *IEEE INFOCOM*, 2011.

[A. 15] A. Bremler-Barr et al. Openbox: Enabling innovation in middlebox applications. In *ACM HotMiddlebox Workshop*, 2015.

[abi] Abilene backbone. `http://abilene.internet2.edu`.

[AFLV08] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.

[B. 10] B. Heller et al. Elastictree: saving energy in data center networks. In *USENIX NSDI*, 2010.

[BASS11] Theophilus Benson, Aditya Akella, Anees Shaikh, and Sambit Sahu. Cloudnaas: a cloud networking platform for enterprise applications. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 8. ACM, 2011.

[BBE+13] M Faizul Bari, Raouf Boutaba, Rafael Esteves, Lisandro Z Granville, Maxim Podlesny, Md Golam Rabbani, Qi Zhang, and Mohamed Faten Zhani. Data center network virtualization: A survey. *Communications Surveys & Tutorials, IEEE*, 15(2):909–928, 2013.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 164–177. ACM, 2003.

[BRL+14] Jeremias Blendin, Julius Rückert, Nicolai Leymann, Georg Schyguda, and David Hausheer. Position paper: Software-defined network service chaining. In *2014 Third European Workshop on Software Defined Networks*, pages 109–114. IEEE, 2014.

[BRXM15] Zvika Bronstein, Evelyne Roch, Jinwei Xia, and Adi Molkho. Uniform handling and abstraction of nfv hardware accelerators. *Network, IEEE*, 29(3):22–29, 2015.

[BSL06] Simon Balon, Fabian Skivée, and Guy Leduc. How well do traffic engineering objective functions meet TE requirements ? In *International Conference on Research in Networking*, pages 75–86. Springer, 2006.

[CB05] Susanta Nanda Tzi-cker Chiueh and Stony Brook. A survey on virtualization technologies. *RPE Report*, pages 1–42, 2005.

[CCF+05] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and implementation of a routing control platform. In *USENIX NSDI*, 2005.

[CFP+07] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. 37(4):1–12, 2007.

[cis] Cisco: NAT order of operation. `http://www.cisco.com/c/en/us/support/docs/ip/network-address-translation-nat/6209-5.html`.

[CKP15] David Coudert, Alvinice Kodjo, and Truong Khoa Phan. Robust energy-aware routing with redundancy elimination. *Computers & Operations Research*, 64:71–85, 2015.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3 edition, 2009.

[CMT+11] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. Devoflow: Scaling flow management for high-performance networks. In *ACM SIGCOMM*, 2011.

[DEA+09] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. Routebricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 15–28. ACM, 2009.

[DGK+13] Panagiotis Demestichas, Andreas Georgakopoulos, Dimitrios Karvounas, Kostas Tsagkaris, Vera Stavroulaki, Jianmin Lu, Chunshan

Xiong, and Jing Yao. 5g on the horizon: key challenges for the radio-access network. *Vehicular Technology Magazine, IEEE*, 8(3):47–53, 2013.

[DK17] Sevil Dräxler and Holger Karl. Specification, composition, and placement of network services with flexible structures. *International Journal of Network Management*, 27(2), 2017.

[dpd] Intel Data Plane Development Kit. `http://www.intel.com/content/www/us/en/communications/data-plane-development-kit.html`.

[DW16] Abhishek Dwaraki and Tilman Wolf. Adaptive service-chain routing for virtual network functions in software-defined networks. In *Proceedings of the 2016 workshop on Hot topics in Middleboxes and Network Function Virtualization*, pages 32–37. ACM, 2016.

[ecm] IP Multicast Load Splitting - Equal Cost Multipath (ECMP) Using S, G and Next Hop. `http://www.cisco.com/en/US/docs/ios/12_2sr/12_2srb/feature/guide/srbmpath.htmlg`.

[EGH+08] Norbert Egi, Adam Greenhalgh, Mark Handley, Mickael Hoerdt, Felipe Huici, and Laurent Mathy. Towards high performance virtual routers on commodity hardware. In *ACM CoNEXT*, 2008.

[eig] Cisco EIGRP. `http://www.cisco.com/c/en/us/support/docs/ip/enhanced-interior-gateway-routing-protocol-eigrp/16406-eigrp-toc.html`.

[esx] VMWare ESXi Hypervisor. `https://www.vmware.com/products/esxi-and-esx/overview`.

[etsa] ETSI. `http://www.etsi.org/standards`.

[etsb] Network functions virtualisation white paper. `https://portal.etsi.org/Portals/0/TBpages/NFV/Docs/NFV_White_Paper3.pdf`.

[etsc] Network functions virtualisation white paper 3. `https://portal.etsi.org/Portals/0/TBpages/NFV/Docs/NFV_White_Paper3.pdf`.

[FCS+14] Seyed Kaveh Fayazbakhsh, Luis Chiang, Vyas Sekar, Minlan Yu, and Jeffrey C Mogul. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In *USENIX NSDI*, 2014.

[Flo] Floodlight. `http://www.projectfloodlight.org/floodlight/`.

[FS07] Uriel Feige and Mohit Singh. Improved approximation ratios for traveling salesperson tours and paths in directed graphs. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 104–118. Springer, 2007.

[FSYM13] Seyed Kaveh Fayazbakhsh, Vyas Sekar, Minlan Yu, and Jeffrey C Mogul. Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions. In *ACM HotSDN*, 2013.

[GHH+09] Adam Greenhalgh, Felipe Huici, Mickael Hoerdt, Panagiotis Papadimitriou, Mark Handley, and Laurent Mathy. Flow processing and the rise of commodity network hardware. *ACM SIGCOMM Computer Communication Review*, 39(2):20–26, 2009.

[GHM+05] Albert Greenberg, Gisli Hjalmtysson, David A Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4D approach to network control and management. *ACM SIGCOMM Computer Communication Review*, 35(5):41–54, 2005.

[GHM16] Frédéric Giroire, Frédéric Havet, and Joanna Moulierac. Compressing two-dimensional routing tables with order. *Electronic Notes in Discrete Mathematics*, 52:351–358, 2016.

[GJVP+14] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. Opennf: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 163–174. ACM, 2014.

[GK06] Eric Gourdin and Olivier Klopfenstein. Comparison of different qos-oriented objectives for multicommodity flow routing optimization. In *Proceedings of the International Conference on Telecommunications (ICT 2006)*, 2006.

[GKJ+13] Aaron Gember, Anand Krishnamurthy, Saul St John, Robert Grandl, Xiaoyang Gao, Ashok Anand, Theophilus Benson, Vyas Sekar, and

Aditya Akella. Stratos: A network-aware orchestration layer for virtual middleboxes in clouds. *arXiv preprint arXiv:1305.0209*, 2013.

[GM01] Liang Guo and Ibrahim Matta. The war between mice and elephants. In *Network Protocols, 2001. Ninth International Conference on*, pages 180–188. IEEE, 2001.

[GMPR15] Frédéric Giroire, Joanna Moulierac, Truong Khoa Phan, and Frédéric Roudaut. Minimization of network power consumption with redundancy elimination. *Computer communications*, 59:98–105, 2015.

[GPGA12] Aaron Gember, Prathmesh Prabhu, Zainab Ghadiyali, and Aditya Akella. Toward software-defined middlebox networking. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, pages 7–12. ACM, 2012.

[HGJL15] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network function virtualization: Challenges and opportunities for innovations. *Communications Magazine, IEEE*, 53(2):90–97, 2015.

[HJPM11] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-accelerated software router. *ACM SIGCOMM Computer Communication Review*, 41(4):195–206, 2011.

[ipe] IPerf: the TCP/UDP bandwidth measurement tool. `http://sourceforge.net/projects/iperf/`.

[ips] Cisco VPN Services Port Adapter Configuration Guide. `http://www.cisco.com/c/en/us/td/docs/interfaces_modules/services_modules/vspa/configuration/guide/ivmsw_book/ivmvpnb.pdf`.

[J. 12] J. Anderson et al. xOMB: extensible open middleboxes with commodity servers. In *ACM/IEEE ANCS*, 2012.

[J. 14] J. Martins et al. ClickOS and the art of network function virtualization. In *USENIX NSDI*, 2014.

[J. 15] J. Hwang et al. NetVM: high performance and flexible networking using virtualization on commodity platforms. *IEEE Transactions on Network and Service Management*, 12(1):34–47, 2015.

[Jai90] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. John Wiley & Sons, 1990.

[JPA+13] Wolfgang John, Konstantinos Pentikousis, George Agapiou, Eduardo Jacob, Mario Kind, Antonio Manzalini, Fulvio Risso, Dimitri Staessens, Rebecca Steinert, and Catalin Meirosu. Research directions in network service chaining. In *Future Networks and Services (SDN4FNS), 2013 IEEE SDN for*, pages 1–7. IEEE, 2013.

[KARW14] Naga Katta, Omid Alipourfard, Jennifer Rexford, and David Walker. Rule-Caching Algorithms for Software-Defined Networks. Technical report, Princeton University, 2014.

[KKL+07] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[KLRW13] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the one big switch abstraction in software-defined networks. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 13–24. ACM, 2013.

[kvm] Kernel-based virtual machine. `http://www.linux-kvm.org/`.

[LBSB09] Cristian Lumezanu, Randy Baden, Neil Spring, and Bobby Bhattacharjee. Triangle inequality variations in the internet. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 177–183. ACM, 2009.

[LC15] Yong Li and Min Chen. Software-defined network function virtualization: a survey. *IEEE Access*, 3:2542–2553, 2015.

[LGL+11] Guohan Lu, Chuanxiong Guo, Yulong Li, Zhiqiang Zhou, Tong Yuan, Haitao Wu, Yongqiang Xiong, Rui Gao, and Yongguang Zhang. ServerSwitch: A Programmable and High Performance Platform for Data Center Networks. In *USENIX NSDI*, 2011.

[lib] TCPDUMP/LIBPCAP public repository. `http://www.tcpdump.org/`.

[lzi] zlib: A Massively Spiffy Yet Delicately Unobtrusive Compression Library. `https://zlib.net/`.

[LZTM16] Tachun Lin, Zhili Zhou, Massimo Tornatore, and Biswanath Mukherjee. Demand-aware network function placement. *Journal of Lightwave Technology*, 34(11):2590–2600, 2016.

[M. 17a] M. Jamshed et al. mos: A reusable networking stack for flow monitoring middleboxes. In *USNIX NSDI*, 2017.

[M. 17b] M. Luizelli et al. A fix-and-optimize approach for efficient and large scale virtual network function placement and chaining. *Computer Communications*, 102:67–77, 2017.

[MAB+08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[man] Network Functions Virtualisation Management and Orchestration. `http://www.etsi.org/deliver/etsi_gs/NFV-MAN/001_099/001/01.01.01_60/gs_nfv-man001v010101p.pdf`.

[MBP+17] Wenrui Ma, Jonathan Beltran, Zhenglin Pan, Deng Pan, and Niki Pissinou. Sdn-based traffic aware placement of nfv middleboxes. *IEEE Transactions on Network and Service Management*, 14(3):528–542, 2017.

[MDT14] Hendrik Moens and Filip De Turck. VNF-P: A model for efficient placement of virtualized network functions. In *IEEE CNSM*, 2014.

[min] Mininet. `http://mininet.org/`.

[MK16] Sevil Mehraghdam and Holger Karl. Placement of services with flexible structures specified by a YANG data model. In *IEEE NetSoft Conference and Workshops (NetSoft)*, 2016.

[MMP15] Wenrui Ma, Carlos Medina, and Deng Pan. Traffic-aware placement of nfv middleboxes. In *Global Communications Conference (GLOBECOM), 2015 IEEE*, pages 1–6. IEEE, 2015.

[ms-] Microsoft TechNet: VPNs and firewalls. `https://technet.microsoft.com/en-us/library/cc958037.aspx`.

[MSB+17] Wenrui Ma, Oscar Sandoval, Jonathan Beltran, Deng Pan, and Niki Pissinou. Traffic aware placement of interdependent nfv middleboxes. In *INFOCOM 2017-IEEE Conference on Computer Communications, IEEE*, pages 1–9. IEEE, 2017.

[MSG+16] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba. Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys & Tutorials*, 18(1):236–262, 2016.

[MUK+04] Tatsuya Mori, Masato Uchida, Ryoichi Kawahara, Jianping Pan, and Shigeki Goto. Identifying elephant flows through periodically sampled packets. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 115–120. ACM, 2004.

[MVB93] Michael J Miller, Branka Vucetic, and Les Berry. *Satellite communications: mobile and fixed services*. Springer Science & Business Media, 1993.

[opea] ONF, Openflow networking foundation. `http://archive.openflow.org/`.

[opeb] Openflow switch specification version 1.5.0. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf`.

[opec] OpenSSL: A Cryptography and SSL/TLS Toolkit. `https://www.openssl.org/`.

[osp] OSPF: frequently asked questions. `http://www.cisco.com/c/en/us/support/docs/ip/open-shortest-path-first-ospf/9237-9.html`.

[pa-a] Palo Alto Networks PA-7000 Series. `https://www.paloaltonetworks.com/campaigns/pa-7000-series`.

[pa-b] Palo Alto Networks VM Series. `https://www.paloaltonetworks.com/products/secure-the-network/virtualized-next-generation-firewall/vm-series`.

[PSS15] Nisha Panwar, Shantanu Sharma, and Awadhesh Kumar Singh. A survey on 5g: The next generation of mobile communication. *Physical Communication*, 2015.

[Q. 17] Q. Zhang et al. Joint optimization of chain placement and request scheduling for network function virtualization. In *Distributed Computing Systems (ICDCS)*, 2017.

[Rao14] Sridhar KN Rao. SDN and its use-cases-NV and NFV. *Network*, 2:H6, 2014.

[RHC+15] Myriana Rifai, Nicolas Huin, Christelle Caillouet, Frédéric Giroire, D Lopez-Pacheco, Joanna Moulierac, and Guillaume Urvoy-Keller. Too many sdn rules? compress them with MINNIE. In *Global Communications Conference (GLOBECOM), 2015 IEEE*, pages 1–7. IEEE, 2015.

[S. 14] S. Mehraghdam et al. Specifying and placing chains of virtual network functions. In *IEEE Cloud Networking*, 2014.

[sdna] SDN. `https://www.opennetworking.org/sdn-resources/sdn-definition`.

[sdnb] SDN White Paper. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf`.

[sfc] Service Function Chaining Problem Statement. `https://datatracker.ietf.org/doc/rfc7498/`.

[SGP+15] J. Soares, C. Gonalves, B. Parreira, P. Tavares, J. Carapinha, J. P. Barraca, R. L. Aguiar, and S. Sargento. Toward a telco cloud environment for service functions. *IEEE Communications Magazine*, 53(2):98–106, Feb 2015.

[SHS+12] 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. *ACM SIGCOMM Computer Communication Review*, 42(4):13–24, 2012.

[sim] Nec's simple middlebox configuration (simco) protocol version 3.0. `https://tools.ietf.org/html/rfc4540`.

[SQT08] Martin Stiemerling, Juergen Quittek, and Tom Taylor. Middlebox communication (midcom) protocol semantics. Technical report, 2008.

[SRR+11] Vyas Sekar, Sylvia Ratnasamy, Michael K Reiter, Norbert Egi, and Guangyu Shi. The middlebox manifesto: enabling innovation in middlebox deployment. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, page 21. ACM, 2011.

[STJP08] Jose Renato Santos, Yoshio Turner, G John Janakiraman, and Ian Pratt. Bridging the gap between software and hardware techniques for i/o virtualization. In *USENIX Annual Technical Conference*, pages 29–42, 2008.

[stt] Data center overlay technologies. `http://www.cisco.com/c/en/us/products/collateral/switches/nexus-9000-series-switches/white-paper-c11-730116.html`.

[T. 16] T. Kuo et al. Deploying chains of virtual network functions: On the relation between link and server usage. In *IEEE INFOCOM*, 2016.

[UNT+11] Zartash Afzal Uzmi, Markus Nebel, Ahsan Tariq, Sana Jawad, Ruichuan Chen, Aman Shaikh, Jia Wang, and Paul Francis. SMALTA: practical and near-optimal FIB aggregation. In *ACM CoNEXT*, 2011.

[V. 12] V. Sekar et al. Design and implementation of a consolidated middlebox architecture. In *USENIX NSDI*, 2012.

[V. 17] V. Eramo et al. An approach for service function chain routing and virtual function network instance migration in network function virtualization architectures. *IEEE/ACM Transactions on Networking*, PP(99):1–18, 2017.

[vce] VMware vCenter Server. `http://www.vmware.com/products/vcenter-server/overview.html`.

[vir] Virsh command reference. `http://libvirt.org/virshcmdref.html`.

[wan] Citrix cloudbridge technical overview. `https://www.citrix.com/content/dam/citrix/en_us/documents/products-solutions/cloudbridge-technical-overview.pdf`.

[WQX+11] Zhaoguang Wang, Zhiyun Qian, Qiang Xu, Zhuoqing Mao, and Ming Zhang. An untold story of middleboxes in cellular networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 374–385. ACM, 2011.

[WRH+15] Timothy Wood, KK Ramakrishnan, Jinho Hwang, Grace Liu, and Wei Zhang. Toward a software-based network: integrating software defined networking and network function virtualization. *Network, IEEE*, 29(3):36–41, 2015.

[Y. 16] Y. Li et al. Network functions virtualization with soft real-time guarantees. In *IEEE INFOCOM*, 2016.

[Y. 17] Y. Sang et al. Provably efficient algorithms for joint placement and allocation of virtual network functions. In *IEEE INFOCOM*, 2017.

[YTG13] Soheil Yeganeh, Amin Tootoonchian, and Yashar Ganjali. On scalability of software-defined networking. *IEEE Communications Magazine*, 2(51):136–141, 2013.

[Z. 13] Z. Qazi et al. SIMPLE-fying middlebox policy enforcement using SDN. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 27–38. ACM, 2013.

[ZBB+13a] Ying Zhang, Neda Beheshti, Ludovic Beliveau, Gregoire Lefebvre, Ravi Manghirmalani, Ravishankar Mishra, Ritun Patneyt, Meral Shirazipour, Ramesh Subrahmaniam, Catherine Truchan, et al. Steering: A software-defined networking for inline service chaining. In *Network Protocols (ICNP), 2013 21st IEEE International Conference on*, pages 1–10. IEEE, 2013.

[ZBB+13b] Ying Zhang, Neda Beheshti, Ludovic Beliveau, Gregoire Lefebvre, Ravi Manghirmalani, Ravishankar Mishra, Ritun Patneyt, Meral Shirazipour, Ramesh Subrahmaniam, Catherine Truchan, et al. Steering: A software-defined networking for inline service chaining. In *IEEE ICNP*, 2013.

[ZLWZ10] Xin Zhao, Yaoqing Liu, Lan Wang, and Beichuan Zhang. On the aggregatability of router forwarding tables. In *IEEE INFOCOM*, 2010.

VITA

WENRUI MA

2003-2007	B.S., Computer Science Zhengzhou University Zhengzhou, China
2007-2012	Network Engineer China Mobile Henan, China
2013-2018	Doctoral Candidate, Computer Science Florida International University Miami, Florida

PUBLICATIONS

W. Ma, C. Medina, and D. Pan, "Traffic-Aware Placement of NFV Middleboxes," *IEEE Global Communications Conference (GLOBECOM)*, San Diego, CA, Dec. 2015.

I. Vawter, D. Pan, and W. Ma, "Emulation Performance Study of Traffic-Aware Policy Enforcement in Software-Defined Networks," *National Workshop for REU Research in Networking and Systems (REUNS)*, Philadelphia, PA, Oct. 2014.

W. Ma, O. Sandoval, J. Beltran, D. Pan, and N. Pissinou, "Traffic-Aware Placement of Interdependent NFV Middleboxes," *IEEE International Conference on Computer Communications (INFOCOM)*, Atlanta, GA, May 2017.

W. Ma, J. Beltran, Z. Pan, D. Pan, and N. Pissinou, "SDN based Traffic Aware Placement of NFV Middleboxes," *IEEE Transactions on Network and Service Management (TNSM)*, vol. 14, no. 13, pp. 528-542, Sep. 2017.

W. Ma, J. Beltran, S. Sundaram, D. Pan, and N. Pissinou, "Service-Aware Routing in NFV Networks," under submission.