Stony Brook University

OFFICIAL COPY

The official electronic file of this thesis or dissertation is maintained by the University Libraries on behalf of The Graduate School at Stony Brook University.

© All Rights Reserved by Author.

Minimally Disruptive Management Frameworks for Network Functions

A Dissertation presented

by

Zafar Ayyub Qazi

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

December 2015

Stony Brook University

The Graduate School

Zafar Ayyub Qazi

We, the dissertation committee for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation

Samir Das - Dissertation Co-Advisor
Professor, Department of Computer Science

Vyas Sekar - Dissertation Co-Advisor
Research Assistant Professor, Department of Computer Science

Aruna Balasubramanian - Chairperson of Defense
Assistant Professor, Department of Computer Science

Phillipa Gill
Assistant Professor, Department of Computer Science

Vijay Gopalakrishnan
Director Inventive Science, AT&T Research
This dissertation is accepted by the Graduate School

Charles Taber
Dean of the Graduate School

Abstract of the Dissertation

Minimally Disruptive Management Frameworks for Network Functions

by

Zafar Ayyub Qazi

Doctor of Philosophy

in

Computer Science

Stony Brook University

2015

Networks today rely on network functions or middleboxes (e.g., firewalls, WAN optimizers) to provide critical performance, security, and policy compliance capabilities. However, today the management of these middleboxes is hard. First, these middleboxes are implemented as dedicated hardware appliances, making it difficult to dynamically scale resources. Second, operators need to carefully plan the network topology, manually set up rules to route traffic through the desired sequence of middleboxes, and implement safeguards for correct operation in the presence of failures and overload.

We can overhaul today's network infrastructure to address these problems by introducing flexibility in routing and the implementation of these middleboxes. However, from network operator's perspective a key question is whether we can address these problems in a minimally disruptive manner, e.g., which require minimal changes to existing middlebox implementations and routing mechanisms.

In this thesis, I describe two case studies for introducing more flexibility in middlebox management with minimal changes to existing middlebox implementations and routing mechanisms. In the first part of the thesis, I describe SIMPLE, a Software-Defined Networking (SDN) based ef-

ficient middlebox traffic steering solution which works with existing middlebox implementations and uses existing SDN APIs. In SIMPLE, I address algorithmic and system design challenges to demonstrate the feasibility of using SDN to simplify middlebox traffic steering. In the second part of the thesis, I describe KLEIN a cellular core re-design that uses Network Function Virtualization (NFV) and smart resource management, stays within the confines of current cellular standards and uses legacy routing in the core network. I address key challenges w.r.t. scalability, responsiveness and in realizing KLEIN via backwards-compatible orchestration mechanisms.

Dedicated to my parents and my siblings, Ihsan and Ali, for their constant support and unwavering belief in my abilities.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Thesis Approach and Contributions 3

1.3 Outline 5

2 Background **6**

2.1 Middleboxes 6

2.2 Software Defined Networking 7

2.3 Network Function Virtualization 9

2.4 Cellular Network Background 10

3 SIMPLE: Simplifying Middlebox Policy Enforcement Using SDN **13**

3.1 Motivation and Contributions 13

3.2 Related Work 15

3.3 Opportunities and Challenges 18

3.3.1 Middlebox composition 18

3.3.2 Middlebox resource management 19

3.3.3 Dynamic traffic transformation 21

3.4 SIMPLE System Overview 22

3.5 SIMPLE Data Plane Design 25

3.5.1 Unambiguous forwarding 25

3.5.2 Compact forwarding tables 26

3.6 SIMPLE Dynamics Handler . . . 28
3.6.1 Design constraints . . . 28
3.6.2 Idea: Flow correlation . . . 29
3.6.3 Similarity-based correlation . . . 31
3.7 Resource Management . . . 33
3.7.1 Offline-Online Decomposition . . . 34
3.7.2 Offline ILP-based pruning . . . 34
3.7.3 Online load balancing with LP . . . 37
3.7.4 Extensions . . . 37
3.8 Implementation . . . 38
3.9 Evaluation . . . 40
3.9.1 Benefits of SIMPLE . . . 41
3.9.2 Scalability and optimality . . . 43
3.9.3 Accuracy of the DynHandler . . . 45
3.10 Summary . . . 48
4 A Framework to Evaluate the NFV Design Space 49
4.1 Motivation . . . 51
4.1.1 Design Space of NFV . . . 51
4.1.2 Motivating Scenarios . . . 53
4.2 Inputs and Requirements . . . 55
4.3 Provisioning Model . . . 56
4.3.1 Control Variables . . . 56
4.3.2 Formulation . . . 57
4.4 Example Use Cases . . . 60
4.5 Summary . . . 63
5 KLEIN: A Minimally Disruptive Design for an Elastic Cellular Core 65
5.1 Motivation and Contributions . . . 65

5.2 Related Work 68
5.3 Limitations of Current Practises 72
5.3.1 Data Set 72
5.3.2 Load Balancing 73
5.3.3 Impact on Applications 74
5.3.4 Resource Provisioning 75
5.3.5 Provisioning Cost vs. Wider Deployment 76
5.3.6 Summary 76
5.4 Design Space Exploration 76
5.4.1 Design space 77
5.4.2 Methodology 78
5.4.3 Results 80
5.4.4 Summary 81
5.5 System Overview and Challenges 81
5.5.1 Overview 82
5.5.2 Challenges 83
5.6 Resource Manager 84
5.6.1 Problem Formulation 85
5.6.2 Key Ideas 86
5.6.3 Our Approach 87
5.7 Network Orchestration 92
5.7.1 Wide-area orchestration 92
5.7.2 Intra-datacenter orchestration 93
5.7.3 KLEIN's reconfigurations 94
5.8 Implementation 95
5.9 Evaluation 96
5.9.1 Scalability and Optimality 98
5.9.2 End-to-End System Validation 99

5.9.3 New Opportunities . . . 100
5.9.4 UE Migrations . . . 101
5.10 Summary . . . 103

6 Conclusions and Future Work 105
6.1 Contributions . . . 105
6.2 Future Work . . . 106
6.2.1 Customization and Modularization of EPC Network Functions . . . 106
6.2.2 Efficient State Management . . . 109

Bibliography 110

List of Figures

2.1 Typical LTE elements and architecture. 10

3.1 Example to illustrate the requirements that middlebox deployments place on SDN. The table shows the different physical sequences of switches and middleboxes used to implement the two logical policy chains: Firewall-IDS and Firewall-IDS-Proxy. 19

3.2 Example of potential data plane ambiguity to implement the policy chain Firewall-IDS-Proxy in our example topology. We annotate different instances of the same packet arriving at the different switches on the arrows. 20

3.3 Overview of the SIMPLE approach for using SDN to manage middlebox deployments. 22

3.4 Example of SIMPLE data plane configurations. The other cases: hop-by-hop with loop and SwitchTunnels with no loop are similar and are not shown for brevity. . . 27

3.5 Similarity based correlation of incoming and outgoing flows through a middlebox. . 32

3.6 High-level overview of the offline-online decomposition in the ResMgr. 35

3.7 Integer Linear Program (ILP) formulation for pruning the set of physical sequences to guarantee coverage for each logical chain while respecting switch TCAM constraints. 36

3.8 Linear Program (LP) formulation for balancing load across middleboxes given a pruned set. 38

3.9 Load on all middleboxes for Internet2 topology. 42

3.10 Maximum middlebox load comparison across topologies with SIMPLE, CoMb, today's Ingress-based deployments relative to the optimal ILP-based configuration. 43

3.11 Response time in the case of a middlebox failure and traffic overload. 44
3.12 Fraction of sequences with loops. 45
3.13 Coverage vs. available switch capacity for selected topologies. We use 3 policy chains per ingress-egress pair. 46
3.14 Accuracy of the SIMPLE DynHandler for two types of proxy-specific policies. . . 47

4.1 The current fixed and proprietary implementation of network functions *vs* the NFV vision of elastic, cost effective, mix-match and potentially hybrid deployment of network functions. 50
4.2 Example to motivate the different design tradeoffs in provisioning. 52
4.3 Example to illustrate the different design tradeoffs in functional placement and routing. 53
4.4 Example to illustrate how flow conservation is modeled. 59
4.5 Total provisioning cost for different NFV models. 60
4.6 Impact on total provisioning cost wit varying cloud cost. 61
4.7 Impact on total provisioning cost with varying setup+OPEX cost. 62
4.8 Impact on provisioning cost with varying resources. 63

5.1 Load across different data centers over the course of a day. 72
5.2 Load distribution across data centers for each time interval. 73
5.3 Impact of EPC load on file download time. 74
5.4 Provisioning cost vs. number of data centers with static provisioning and routing. . 75
5.5 Linear Program (LP) formulation for CLEANSLATE. 77
5.6 The load balancing optimality gap between INTERMEDIATE and CLEANSLATE. . . 78
5.7 Reduction in provisioning cost with INTERMEDIATE and CLEANSLATE. 79
5.8 KLEIN system overview. 82
5.9 Decomposition and decoupling in resource management. 87
5.10 Global controller formulation for distributing load across regions. 89
5.11 Regional controller formulation for control traffic placement (CP). 90
5.12 Regional controller formulation for data traffic placement (DP). 91

5.13 Network orchestration mechansims in a KLEIN based cellular core. 92
5.14 KLEIN's responsiveness 96
5.15 KLEIN's optimality 97
5.16 Varying reconfiguration period 98
5.17 KLEIN's failure handling 99
5.18 KLEIN's failure response 100
5.19 Impact on end-application performance 101
5.20 Handling traffic overload on an EPC instance by instantiating a new EPC instance. 102
5.21 Validation of KLEIN on EPC testbed 103
5.22 Varing UE migration threshold, and observing the impact on optimality gap. 104

6.1 Examples of functional customization. 107

List of Tables

3.1 A taxonomy of the dynamic actions performed by different middleboxes that are commonly used today [124] and the corresponding information that we need to infer at the SDN controller. 30

3.2 End-to-end metrics for the topology in Figure 3.1 on Emulab and Mininet. Having confirmed that the results are similar, we use Mininet for larger-scale experiments. . 40

3.3 Time and control traffic overhead to install forwarding rules in switches. 40

3.4 Time to generate load balanced configurations subject to switch constraints. 47

5.1 Scalability with a 2-level resource management decomposition. 83

List of Abbreviations

SDN	Software Defined Networking
NFV	Network Function Virtualization
3GPP	Third Generation Partnership Program
S-GW	Serving Gateway
P-GW	Packet Gateway
MME	Mobile Management Entity
HSS	Home Subscriber Server
LTE	Long Term Evolution

Acknowledgements

As I reflect back on my graduate school journey, I feel my most important learning has been to understand myself a little better. Failures, successes, periods of loneliness and contemplation, have provided me an opportunity to discover myself. I am indebted to a large number of people for providing me support, care and help during this period. Their support has been instrumental in making this journey fruitful and fun.

First, I am grateful to my advisors Samir Das and Vyas Sekar for their constant support, help and guidance. Samir has been incredibly supportive and patient. Apart from five years of financial support, he allowed me the freedom to pursue interesting research directions. I am deeply indebted to Vyas Sekar for his constant support and for always being a message away. I have been and continue to be amazed by Vyas's incredible conscientiousness and attention to technical detail. I have been even more amazed by his sense of concern for his students and willingness to help; he has always been ready to give feedback on a technical draft, look at a mathematical formulation or an algorithm. I am also grateful to Vijay Gopalakrishnan and Kaustubh Joshi for educating me about cellular networks and for all their help. I would like to thank Minlan Yu, Seungjoon Lee and JK Lee, all of whom have been great mentors. I am also grateful to my other thesis committee members Aruna Balasubramanian and Phillipa Gill for being part of the committee and for giving me comments on my thesis. I would also like to thank my fellow student collaborators for all their help: Pralhad Deshpande, Navid Azimi, Luis Chang, Rui Mao, William Tu, Phani Krishna, Himanshu Shah, Zhibin Zhou, Seyed Fayazbakhsh, Ashish Tanswer.

This thesis would not have been possible without the support of my parents. Being the youngest of three kids, it hasn't been easy on my mother while I have been away all these years. However, she has always motivated me to do what I enjoy and always believed in my abilities. She has constantly reminded me not to forget the bigger picture in life – my health, family and friends. My

father has been a rock of solid support, his kind words have always inspired me. He has been a constant inspiration. I feel fortunate that my parents have visited me four times in the US. They attended my orientation and would this year be attending my convocation ceremony.

One of the inspirations for me in pursuing a PhD was my elder brother Ihsan Qazi. He has been an inspiration, a great mentor, some one who has always reminded me of the joys of learning and creating new things. During my entire graduate studies, he has always been available to discuss anything, from reviewing my paper to giving me advice on life. I am also grateful to him for believing in me far more than I have believed in my self and for always finding the best in me. Surely, this thesis would not have been possible without him. I also deeply grateful to my eldest brother Raza Ali for his constant support through the years. He has always motivated me to attack important problems in my research, to strike a balance in life and most importantly to enjoy life. I would also like to thank my sister-in-law, Saleha Ali for her amazing support. I would also to thank Uncle Rana and their family for all their support while I have been in US.

I have been very fortunate to have my best friend Saad Nadeem here at Stony Brook University. We have been friends for almost two decades. I am really grateful to him for believing in me for all these years, for providing me amazing support, for reminding me to keep contributing and helping other people, and for all the intellectually invigorating discussions. I am also indebted to my friend Chitra Mohan. She has been extremely supportive, believed in me more than I do, and on innumerable occasions given me perspective on life. She has motivated me to stay fit, been my gym partner, and also helped me learn many amazing food recipes.I would like to thank my friend and apartment mate Rami. Rami has been a great help and true friend. I have enjoyed listening to his strong opinions on a variety of topics which have been both educating and entertaining. I would also like to thank my other friends and collegues who made my stay in US memorable: Salman Mahmood, Uzair Shujaa, Wasif Shabbir, Summit Bajaj, Moussa Ehsan, Sindhuja Thirumalai, Taimoor Shahid, Dawood Ibrahim, Sana Dawood, Ayon Chakraborty, Fatima Zarinni, Zahaib Akther and Ruwaifa Anwar.

Chapter 1

Introduction

1.1 Motivation

Networks today no longer just perform routing and forwarding responsibilities, they in reality perform a lot of complex in-network processing. They rely on specialized network functions or middleboxes to provide critical performance, security, and policy compliance capabilities. These middleboxes such as WAN optimizers, proxies, intrusion detection and prevention systems, network and application-level firewalls, caches and load-balancers have found widespread adoption in modern networks. Surveys show that these middleboxes play a critical role in many network settings [74, 92, 125, 128, 137]. For instance, a survey across 57 network operators [128] showed the number of these middleboxes is comparable to the number of routers and switches for many enterprise networks of various deployment sizes. Similarly, cellular networks host a large number of middleboxes [137].

Today the infrastructure for these middleboxes has developed in a largely uncoordinated manner. New form of middleboxes typically emerge as one-off solutions addressing a specific need, and are patched into the infrastructure through adhoc and often manual techniques. For instance, operators manually set up rules to route the traffic through the desired sequence of middleboxes. This process of deploying middleboxes in today's networks is inflexible and prone to misconfiguration. According to [11], 78% of data center downtime is caused by misconfiguration. A middlebox survey [128] across 57 network operators showed that there are large number of middlebox failures,

and network operators spent an estimated time of between one and five hours per week dealing with middlebox failures. More than 50% of middlebox failures are due to misconfiguration and about 15% are due to middlebox overload scenarios [128].

A key challenge in supporting middleboxes in today's networks is that there are no available protocols and mechanisms to explicitly insert these middleboxes on the path between end-points. As a result, network operators deploy middleboxes implicitly by placing them on the physical path. Network operators need to carefully plan the network topology, manually set up rules to route traffic through the desired sequence of middleboxes, and implement safeguards for correct operation in the presence of failures and overload [74]. As the complexity and scale of networks increase, it is becoming harder and harder to rely on these ad-hoc mechanisms [92]. In addition, middlebox applications are typically resource intensive. They are implemented as dedicated hardware appliances, and each middlebox is provisioned resources to handle peak traffic load. Hence these middlebox resources cannot be amortized across space and applications even if workloads offer natural opportunities to do so. These problem make it difficult to achieve the following desired objectives:

- **Policy Composition:** Network policies typically require packets to go through a sequence of middleboxes (e.g., firewall+intrusion detection system+proxy). Today, network operator's have to manually plan middlebox placements and configure routes to enforce such policies. A desired goal would be to ensure that given a set of middlebox-specific policies, these policies can be correctly and dynamically implemented in the network.

- **Resource Management:** Middleboxes perform complex packet processing (e.g., deep packet inspection). A key requirement from network operators is to be able to control how the load is distributed across different middlebox instances, and to avoid overload scenarios [128]. Additionally given the diversity of middleboxes, network operator's would also want to control how these network functions are provisioned and placed. Unfortunately, today the network operators manually set up routing paths to balance load across different middlebox instances and they have to pre-provision middleboxes to handle peak load scenarios [124].

- **Packet Modification** Middleboxes modify packet headers (e.g, NATs) and even change

session-level behaviors (e.g., WAN optimizers and proxies use persistent connections). Today, network operators have to account for these effects via careful placement or manually reason about the impact of these modifications on routing configurations. The desired goal would be to detect these modifications automatically and dynamically account for these modifications.

On approach to achieve these objectives is to overhaul today's network infrastructure. We can consider new architectures for middlebox deployments that change both how individual middleboxes are implemented and how a network of middleboxes are managed [124, 50, 71]. However, such an approach will not work with existing middlebox deployments. Middleboxes are today deployed in huge numbers in all types of networks and the middlebox market is worth billions of dollars [43], with many vendors selling different types of middleboxes with proprietary implementations. From a network operator's perspective a key question is whether we can address these problems in a minimally disruptive manner, e.g., which require minimal changes to existing middlebox implementations and routing mechanisms. In thesis, we try to address this question and investigate how we can address middlebox management challenges in a minimally disruptive manner.

1.2 Thesis Approach and Contributions

The key contribution of this thesis is to investigate whether we can address middlebox managment challenges in a minimally disruptive manner, and the design of orchestration mechanisms to address the above middlebox management challenges with minimal changes to existing middlebox implementations and routing interfaces. In this thesis I will describe two case studies for introducing more flexibility in middlebox management, in a minimally disruptive manner.

1. In my first study, I propose SIMPLE [116], a Software-Defined Networking-based orchestration layer for efficient middlebox-specific traffic steering. Software-Defined Networking (SDN) offers a promising alternative for control and coordination over traffic forwarding, by using logically centralized management, decoupling the data and control planes, and providing the ability to programmatically configure forwarding rules [55]. Middleboxes, however,

introduce new aspects (e.g., policy composition, resource management, packet modifications) that fall outside the purvey of traditional L2/L3 functions that SDN supports (e.g., access control or routing). [146]. In SIMPLE, I addresses these challenges and propose an efficient middlebox-specific policy enforcement layer. In designing SIMPLE, I take an explicit stance to work within the constraints of existing SDN interfaces and middleboxes. To this end, I address algorithmic and system design challenges to demonstrate the feasibility of using SDN to simplify middlebox traffic steering.

2. In my second study, I will describe KLEIN a cellular core re-design that uses Network Function Virtualization (NFV) and smart resource management, stays within the confines of current cellular standards and uses legacy routing in the core network. Network Function Virtualization (NFV) aims to leverage standard virtualization technologies to consolidate many network appliance and equipment types onto industry standard high volume servers and storage, which could be located in data centers and network nodes [25]. However, realizing the benefits of NFV introduces several challenges: 1) The need for dynamic resource management for managing the placement and deployment of virtualized NFs and 2) Network orchestration mechanisms to dynamically route traffic to required network functions. Addressing these challenges in minimally disruptive manner in the cellular core, means designing a resource management layer that can be scalable and responsive while using backwards-compatible orchestration mechanisms, and staying within the confines of existing cellular standards. In KLEIN, I address key challenges w.r.t. scalability, responsiveness and in using backwards-compatible orchestration mechanisms to realize the benefits of dynamic resource management layer.

These approaches, besides providing practical and minimally disruptive solutions, also give insights on, to what extent these problems can be solved by constraining the flexibility and control on a specific design dimension, such as routing or middlebox implementation. For instance, SIMPLE provides more control over routing while working with unmodified middlebox implementations. KLEIN on the other hand, for a wide area context, introduces more flexibility in middlebox implementations, while working with legacy wide-area protocols and being within the confines of

cellular protocols. In a sense, any additional flexibility will only help these solutions to add more freedom on the constrained dimensions.

A general contribution of this thesis is to formulate these middlebox management problems as optimization problems, and to design scalable and efficient heuristics for solving these otherwise hard problems. With the emergence of Software Defined Networking (SDN), about which we provide background in Chapter 2, a number of management problems are now amenable to being formulated as centralized optimization problems. This thesis provides scalable and efficient heuristics for middlebox resource management problems.

1.3 Outline

This thesis is organized as follows:

- Chapter 2 provides background on middleboxes, SDN, NFV and cellular networks.
- Chapter 3 presents SIMPLE, a system for efficient middlebox policy enforcement.
- Chapter 4 presents a framework to reason about designs in the NFV design space.
- Chapter 5 describes KLEIN, a NFV-based EPC design for cellular networks.
- I summarize the key contributions and the implications of the work presented here before highlighting some potential avenues for future work in Chapter 5.

Chapter 2

Background

In this chapter, we give an overview and background of middleboxes. We also provide background on two key paradigms that we use in this thesis: Software Defined Networking (SDN) and Network Function Virtualization (NFV). We also provide background on cellular networks, as we consider cellular networks as a use case in some of our studies.

2.1 Middleboxes

Network deployments evolve in response to changing application, workload, and policy requirements. In current networks, the de-facto approach to introduce new functionality is often through the deployment of specialized network appliances or "middlebox". A middlebox, also called a *network appliance* or a *network function* is defined as:

"*A middlebox is defined as any intermediary device performing functions other than the normal, standard functions of an IP router on the datagram path between a source host and destination host.*" [20]

These middleboxes are deployed primarily for security and performance benefits. They are also used for other purposes such as billing, asset tracking, usage monitoring, network address translation, protocol conversion, etc. Some examples are firewalls, application firewalls, intrusion detection systems, intrusion prevention systems, proxies, caches, WAN optimizers, protocol accelerators, application gateways and SSL offloaders.

The key difference between these middleboxes and traditional L3 routers/L2 switches is that these middleboxes perform complex and varied operations on packets such as deep packet inspection. There are new categories of middleboxes on the market every year. These middleboxes are often stateful (e.g., some maintain per-flow or per-session state [72, 118]. They remember fine-grained data that is updated as frequently as every packet or every connection.

Today, these middleboxes are deployed in all types of networks, enterprise networks [128, 124, 97] as well as ISP [140, 136] networks. A survey [124] from a large enterprise network showed that number of middleboxes were comparable to the number of routers. Another survey [128] across 57 network operators showed that the number of middleboxes were roughly the same as the number of L3 routers and L2 switches, and this was true for network deployments of different sizes.

Several studies report on the rapid growth of middlebox market; the market for network security appliances alone was estimated to be 6 billion dollars in 2010 and expected to rise to 10 billion in 2016 [43]. In other words, middleboxes are a critical part of today's networks and it is reasonable to expect that they will remain so for the foreseeable future.

2.2 Software Defined Networking

The Internet architecture today tightly couples the control logic and packet handling inside the individual router and switches. As a result, each router and switch participates in distributed protocols that define the control logic. For example, in IP networks, the path-computation logic is governed by distributed protocols such as OSPF and EIGRP. The routing protocols dictate not only how the routers learn about the topology, but also how they select paths. Similarly, in Ethernet networks, the control logic to compute paths is embedded in the Spanning Tree protocol.

However, today's networks are growing in size and there is increasing demand for supporting diverse and rich services. There is a need for satisfying network-level objectives and capabilities far more sophisticated than best-effort packet delivery. These ever-evolving requirements have led to incremental changes in the control-plane protocols, as well as complex management-plane software that tries to coax the control plane into satisfying the network objectives. The resulting

complexity is responsible for the increasing fragility of IP networks and the tremendous difficulties facing people trying to understand and manage their networks [75].

This has motivated the idea to make the networks programmable and to decouple the control logic from packet handling mechanisms to satisfy network-wide objectives. This has culminated in what we now refer to as "Software Defined Networking (SDN)". SDN simplifies the network management by decoupling the control plane (e.g., an intended routing policy) from the data plane (e.g., packet forwarding). The control logic is then realized through a logically centralized control plane which has a network-wide view [104]. This logically centralized control plane is amenable for realizing network-wide objectives such as traffic engineering, policy enforcement etc. In SDN, the network elements such as router and switches are programmable, and the logically centralized control plane can program these elements dynamically based on operator objectives and policies.

In SDN, there is a south bound API, which refers to how the control plane will interact with the router and switches. OpenFlow [55] is the most well know protocol used for the controllers to communicate with routers and switches. There are a large number of SDN controllers proposed [33, 14, 16, 96, 77]. In SDN there is also a notion of north bound APIs, which define how network applications are going to interact with the SDN control plane. Network applications can be written on top of the SDN controllers. Over the last few years, they have been a large and diverse set of SDN-based network applications proposed, including from monitoring and measurement [144, 107, 54, 143], middlebox management [116, 66, 72], security applications [65, 130, 103, 85, 114], virtualization [72, 118, 89, 129], flow scheduling and load balancing [135, 48, 82, 47], wide area management [87, 84, 80], managing wireless network [79, 78], cloud management [117, 109], big data handling [62, 134] and optical networks [56, 61].

Many major industry players including (Google, AT&T, Verizon) have completely embraced SDN [26, 41, 87]. For instance, AT&T, recently announced its target that by 2020, 75% its network will be controlled by software, and in this effort it has reorganized nearly 130,000 of its organization, including IT, network and operations staff, to focus on this new effort [10].

This is very interesting because the intellectual ideas proposed by SDN are not new. There has been a lot of work in the last 20-25 years related to active networks and centralized control, where

some what similar ideas have been proposed [68]. However it seems now most of these ideas have found compelling use cases which have driven the adoption of these technologies [68].

2.3 Network Function Virtualization

Network functions or middleboxes (e.g., firewalls, intrusion detection systems, application gateways) have been traditionally implemented using specialized and proprietary hardware. While this was necessary for performance in the past, it also leads to high cost and inflexibility as the intended function is physically tied with the hardware platform that implements it. Today networks are populated with a large and increasing variety of proprietary hardware appliances. To launch a new network service or application often requires yet another variety of a middlebox, which means finding space and power for the physical box; in addition managing these middleboxes is highly complex [128, 92] and requires a great deal of expertise and labor [128]. Moreover, hardware-based appliances rapidly reach end of life, which results in deploying these devices in repeated cycles. Worse, hardware life cycles are becoming shorter as technology and services innovation accelerates, making it costly and complicated to evolve the network. [25]. All these limitations and inefficiences make both the capital and operating costs of networks high and also makes it complex to manage these networks [128].

Network Functions Virtualization (NFV) aims to address these problems by decoupling the software from the hardware, virtualizing and consolidating many network appliances onto industry standard high volume servers. Network Functions Virtualisation is potentially applicable to any data plane packet processing and control plane function in fixed and mobile network infrastructures [25].

There have been recently many efforts to decouple the middlebox software from the hardware. These include designing software middleboxes as x86 middleboxes implemented in software [35, 111], new designs and pipelines for software middleboxes [124, 50, 95], as well as middleboxes running inside VMs in data centers [101, 30, 86].

Inspired from these trends, leading vendors are responding by announcing new software appliance products [7, 13, 42]. Given these benefits, major service providers have deployed (or are

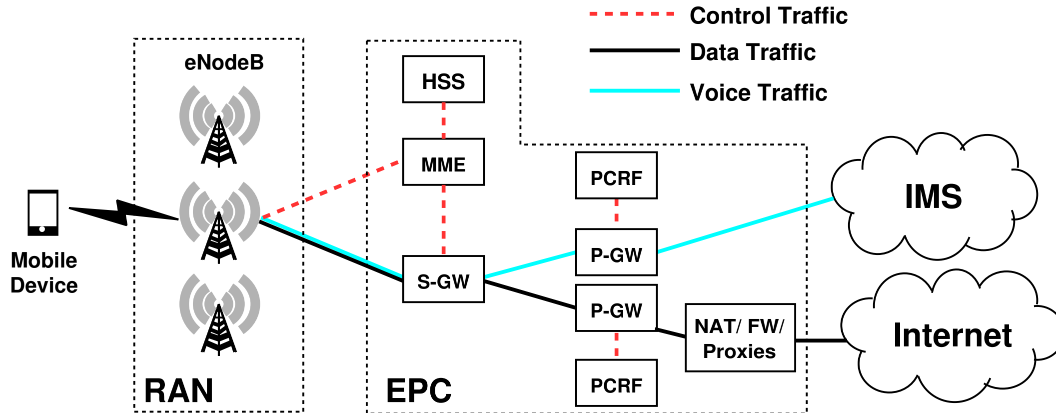


Figure 2.1: Typical LTE elements and architecture.

planning to deploy) data centers to have a distributed pool of hardware resources to offer services to where and when the capacity is needed [26, 41]. One potential concern with NFV is data plane performance; fortunately, several recent advances in networking, virtualization, and operating systems (e.g., SR/IOV, kernel bypass) have demonstrated the viability of line-rate packet processing in software [121, 120, 113]. Thus, such performance concerns are increasingly becoming less of a concern and with recent hardware support these concerns diminish further [7, 36].

2.4 Cellular Network Background

We describe the architecture of the cellular network in this section. Third Generation Partnership Program (3GPP) is the program that standardizes systems, architectures and protocols related to a large number of cellular technologies. We focus on 3GPP Long Term Evolution (LTE) for ease of exposition; however, 3G and 2G also have very similar architectures, though the specific components and their functions vary.

As depicted in Figure 2.1, the LTE cellular network consists of two main components: the LTE Radio Access Network (RAN), and the evolved packet core (EPC). LTE RAN consists of the eNodeB (enhanced NodeB), which communicates with mobile devices via the radio link and then forwards packets to the eventual destination via the EPC. The eNodeB also performs radio resource control and cooperates with the Mobility Management Entity (MME) for mobility management (e.g., handover). To cover a large geographic footprint and to provide high quality service, a

typical cellular service provider employs tens of thousands of eNodeBs.

The main elements of the EPC consist of the MME, the serving gateway (S-GW), and the Packet Data Network Gateway (P-GW). The MME is responsible for all control plane messaging including user authentication via the Home Subscriber Server (HSS), session establishment and release, and mobility management. The S-GW and P-GW are on the data path, and their main function is packet routing/forwarding, traffic management and accounting, and policy enforcement. The S/P-GW also act as anchor points in the cellular network with the S-GW being the anchor for inter-eNodeB handover, and the P-GW acting as a gateway/anchor to external networks (e.g., the Internet). The LTE standard allows for the specification and enforcement of dynamic policies (e.g., changing priority for a flow) within the cellular network. The Policy and Charging rules function (PCRF) is the repository of such policies. Whenever a new flow starts, the PCRF is consulted to identify policies that apply to the flow. The policy and charging enforcement function (PCEF), which is typically built into the P-GW, is responsible for the enforcement of cellular policies. Finally, most cellular EPCs also include middleboxes like NATs, firewalls and proxies that are traversed before a packet reaches the Internet.

A typical cellular network has a few hundred of these EPC components. The data plane elements are typically deployed in a small number of pre-provisioned data centers [141] while the control plane elements are deployed closer to eNodeBs for efficiently handling latency sensitive control plane traffic. When building out these data centers, the EPC is typically provisioned in distinct units we call as 'zones'. A zone typically consists of P-GWs, possibly S-GWs, and other associated middleboxes (e.g., NAT, firewall) and network elements. When the traffic in existing zones reaches a capacity threshold, a new zone is added.

The EPC is typically partitioned to handle different types (e.g., LTE VoIP or VoLTE, Internet data, M2M, corporate VPN) of traffic. This partitioning is achieved through the use of access point names (APN). A cellular provider can associate different traffic types to one or more APNs. A set of APNs – depending on their traffic volume – is mapped to a zone. As a result, the P-GW and other middleboxes and network elements in the zone are configured to serve a set of APNs. Roughly, a zone serves as a basic provisioning unit in the data center while an APN serves as

a traffic classifier where its traffic is load-balanced across multiple zones (e.g., a zone serving a metropolitan area).

Before a device can send or receive data, it has to first establish a GTP (GPRS Tunneling Protocol) tunnel. The GTP tunnel, established between the eNodeB and the P-GW, provides logical point-to-point connectivity per device as it moves around in the network. The GTP tunnel comprises of two halves; one between eNodeB and S-GW and one between S-GW and P-GW. While the latter is retained as long as the device is registered in the network, the former is torn down whenever the device goes idle, and re-created whenever data is exchanged. When the device moves from eNodeB to another, the tunnel between with eNodeB and the S-GW also moves. To setup the tunnel, the device first identifies the APN to use and then the associated P-GW. It then initiates establishment of a GTP tunnel. Similarly, it initiates tunnel creation when it wakes up and has data to send. However, the network has to "page" the device whenever there is data for the device and the device is idle. The device, when it receives a page, wakes up and reestablishes the tunnel between the eNodeB and the S-GW.

To summarize, there are multiple services and devices that run inside a cellular core network supporting not only LTE, but also 3G and 2G networks. Today, the platforms running these services comprise of fixed hardware appliances that are statically provisioned and configured. Different traffic types, however, may have different load patterns and peaks. Similarly, traffic at different locations may behave differently. Finally, the traffic for one service, e.g., 3G, may reduce over time and be replaced with another, e.g., LTE. Virtualizing the cellular network elements allows us to consolidate these functions and dynamically scale and place these functions based on demands across specific dimensions.

Chapter 3

SIMPLE: Simplifying Middlebox Policy Enforcement Using SDN

3.1 Motivation and Contributions

Surveys show that middleboxes (e.g., firewalls, VPN gateways, proxies, intrusion detection and prevention systems, WAN optimizers) play a critical role in many network settings [74, 92, 125, 128, 137]. Achieving the performance and security benefits that middleboxes offer, however, is highly complex. This complexity stems from the need to carefully plan the network topology, manually set up rules to route traffic through the desired sequence of middleboxes, and implement safeguards for correct operation in the presence of failures and overload [74].

Software-Defined Networking (SDN) offers a promising alternative for *middlebox policy enforcement* by using logically centralized management, decoupling the data and control planes, and providing the ability to programmatically configure forwarding rules [55]. Middleboxes, however, introduce new dimensions for SDN that fall outside the purvey of traditional Layer 2/3 (L2/L3) functions that SDN tackles today. This creates new opportunities as well as challenges for SDN that we highlight next.

- **Composition:** Network policies typically require packets to go through a sequence of middleboxes (e.g., firewall+IDS+proxy). SDN can eliminate the need to manually plan mid-

dlebox placements or configure routes to enforce such policies. At the same time, using flow-based forwarding rules that suffice for L2/L3 applications atop SDN can lead to inefficient use of the available switch TCAM (e.g., we might need several thousands of rules) and also lead to incorrect forwarding decisions (e.g., when multiple middleboxes need to process the same packet).

- **Load balancing:** Due to the complex packet processing that middleboxes run (e.g., deep packet inspection), a key factor in middlebox deployments is to balance the processing load to avoid overload [128]. SDN provides the flexibility to implement load balancing algorithms in the network and avoids the need for operators to manually install traffic splitting rules or use custom load balancing solutions [135]. Unfortunately, the limited TCAM space in SDN switches makes the problem of generating such rules to balance middlebox load both theoretically and practically intractable.

- **Packet modifications:** Middleboxes modify packet headers (e.g, NATs) and even change session-level behaviors (e.g., WAN optimizers and proxies use persistent connections). Today, operators have to account for these effects via careful placement or manually reason about the impact of these modifications on routing configurations. By taking a network-wide view, SDN can eliminate errors from this tedious process. Due to the proprietary nature of middleboxes, however, a SDN controller may have limited visibility to set up forwarding rules that account for such transformations.

This work presents the design and implementation of SIMPLE,[1] a SDN-based policy enforcement layer for middlebox-specific traffic steering [92]. SIMPLE allows network operators to specify a logical middlebox routing policy and automatically translates this into forwarding rules that take into account the physical topology, switch capacities, and middlebox resource constraints. In designing SIMPLE, we take an explicit stance to work within the confines of existing SDN capabilities (e.g., OpenFlow) and without modifying middlebox implementations.

Corresponding to the above challenges, there are three key components in SIMPLE's design:

[1] SIMPLE =Software-defIned Middlebox PoLicy Enforcement

- **Efficient data plane support for composition (§3.5):** We use two key ideas: tunnels between switches and leverage SDN capabilities to add tags to packet headers that annotate each packet with its processing state.

- **Practical unified resource management (§3.7):** We decompose the intractable optimization into a hard offline component that accounts for the integer constraints introduced by switch capacities and an efficient online component that balances middlebox load in response to traffic changes.

- **Learning middlebox modifications (§3.6):** We exploit the reporting capabilities of SDN switches to design lightweight flow correlation mechanisms that account for most common middlebox-induced packet transformations.

We implement a proof-of-concept SIMPLE controller that extends `POX` [33] (§3.8). Using a combination of live experiments on Emulab [138], large-scale emulations using Mininet [21], and trace-driven simulations, we show that SIMPLE (§3.9):

- improves middlebox load balancing 6× compared to today's deployments and achieves near-optimal performance w.r.t. new middlebox architectures [124];

- takes ≈100 ms to bootstrap a network and to respond to network dynamics in a 11-node topology;

- takes ≈1.3 s to rebalance the middlebox load and is 4 orders of magnitude faster than strawman optimization schemes.

3.2 Related Work

Middlebox policy enforcement: The work closest to SIMPLE is pLayer [92] which provides a Layer-2 solution to route traffic through middleboxes. In [92], the authors propose a policy-aware switching layer, a new layer-2 for data centers consisting of inter-connected policy-aware switches, called pswitches. Unmodified middleboxes are placed off the network path by plugging them into

pswitches. Based on policies specified by administrators, pswitches explicitly forward different types of traffic through different sequences of middleboxes. pLayer, however, does not address the following issues that SIMPLE tackles: load balancing or routing with switch constraints, the impact of middleboxes modifying headers, and possible routing loops. Another early effort Flow-stream [76] envisions "virtual middleboxes" with an OpenFlow frontend for routing. It proposes the implementation of network functionalities in virtualized machines/servers/routers run on top of commodity PCs. The flow of traffic among these virtual network entities is controlled by a programmable network switch implementing Openflow. In some sense, FlowStream and pLayer were ahead of their time; they preceded SDN/OpenFlow adoption and do not consider the constraints or capabilities that they offer.

Concurrent effort by Jin et al., also highlights challenges related to routing loops and switch constraints for middlebox steering in cellular networks [90]. While they employ a similar tag-based solution for the loop problem, their solution to address switch constraints involves a separation of edge vs. core functionality and the use of aggregation operators. SIMPLE focuses on balancing the middlebox load and uses the offline-online decomposition to address the switch constraints.

Other works consider the problem of routing traffic to specific monitoring nodes [119] and considers middlebox placement in conjunction with cloud applications [52, 98]. These do not consider middlebox composition, switch constraints, or dynamic packet transformations.

Middleboxes + SDN: Recent work has proposed new software-based programmable middleboxes [50, 124]. The work in [50] presents xOMB , a programmable software middlebox architecture based on commodity servers and operating systems. xOMB employs a general programmable pipeline for network processing, composed of xOMB-provided and user defined C++ modules responsible for arbitrary parsing, transforming, and forwarding messages and streams. Modules can store state and dynamically choose different processing paths within a pipeline based on message content. CoMb [124], presents a new middlebox infrastructure design, where instead of specialized middleboxes, the hardware and software is decoupled, and middlebox applications are run on top of a consolidated hardware platform. CoMb [124] and xOMB [50] argue for extensible middleboxes that use commodity hardware similar to prior work on software routers [64, 95]. SIMPLE

does not attempt to provide these benefits. Because SIMPLE is agnostic to how middleboxes are implemented, it can easily extend to such deployments. In fact, these may offer new dimensions of flexibility to dynamically initiate new middlebox capabilities at desired locations.

Recent work has also suggested SDN-based new interfaces for manipulating middlebox state [71]. In [71], the authors propose a framework to realize software-defined middlebox networking. Similar to the management of routers and switches in SDN, they propose APIs and interfaces for managing middleboxes in SDN. These APIs consist of mechanisms whereby a centralized SDN controller can manage middlebox state. SIMPLE can benefit from these, especially in the context of dynamic transformations. Given the nature of the middlebox market, however, it is less likely that these efforts will be adopted in the near term and SIMPLE offers a practical alternative in the interim. There are also some efforts to standardize middlebox control interfaces such as MIDCOM [132] and SIMCO [24].

Recent work has also considered offloading middlebox functions to service providers [73, 128]. The work in [73, 128] proposes mechanisms through which an entire enterprise network's middlebox processing can be moved to a cloud infrastructure.

Given the size of the middlebox market [44], the *diversity* of functions [125, 128]), the *proprietary* nature of middlebox implementations (e.g., specialized DPI hardware [31]), the above efforts likely face significant barriers to adoption. Furthermore, there are large legacy deployments that are unlikely to go away. Thus, while these forward-looking research efforts are valuable, they are not immediately realizable. SIMPLE takes an explicit stance to work within the confines of existing middlebox implementations and SDN capabilities. Furthermore, the ideas in SIMPLE will apply to service providers who provide the outsourced middlebox services [73, 128].

Policy management in SDN: SDN has traditionally focused on L2/L3 policies such as access control, rate limiting, and routing [55, 96]. Recent work provides abstractions to compose different policy modules [105]. Pyretic [105] introduces new programming abstractions for building applications out of multiple, independent modules that jointly manage network traffic. Complementary to these works, SIMPLE supports middlebox policies that defines the traversal of middlebox chains.

In the data plane, prior work suggests methods to reduce the switch memory usage for flow-based rules [108, 145]. vCRIB [108] proposes a virtualized Cloud Rule Information Base (vCRIB) that provides operators or a network management system with the abstraction of an unbounded prioritized list of rules. They jointly consider the resource, cost, and performance constraints in the hypervisors and switches, and automatically installs rules at the right location to optimize the packet processing performance while minimizing the resource usage and cost. DiFANE [145] considers the problem of implementing a large number of high level policies inside a SDN network in a scalable way. Instead of sending the first packet of new flow to an SDN controller, DiFANE processes all the packets in the data-plane, by directing missed packets to intermediate switches. While SIMPLE uses some of these ideas, it takes a unified view of both switch resource and middlebox constraints.

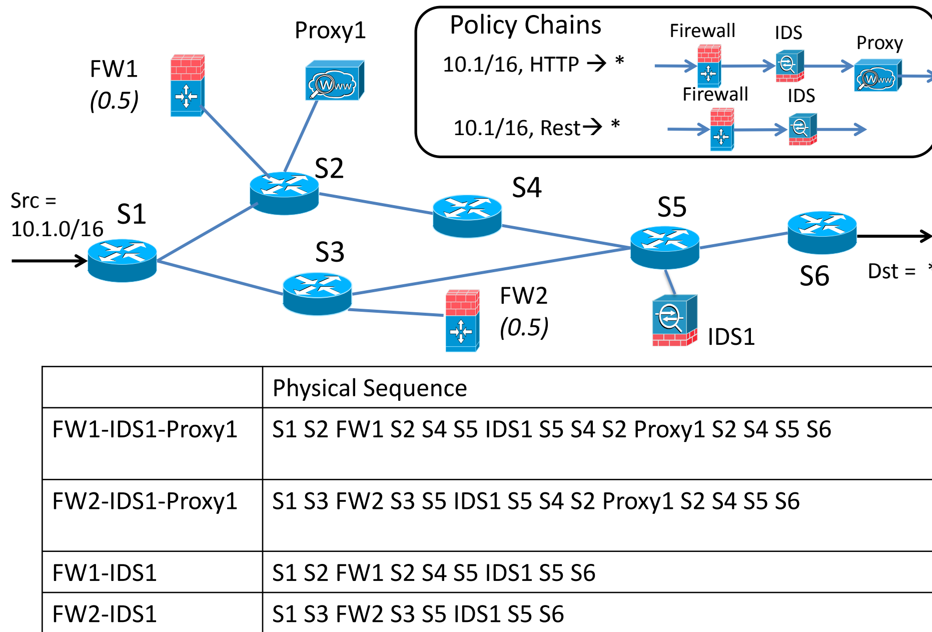
3.3 Opportunities and Challenges

We begin by identifying key challenges in using SDN for middlebox-specific policy enforcement. To make this discussion concrete, we use the example network in Figure 3.1 with 6 switches S1–S6, 2 firewalls FW1 and FW2, 1 IDS, and 1 Proxy.

3.3.1 Middlebox composition

Typical middlebox policies require a packet (or session) to traverse a sequence of middleboxes. (This is an instance of the broader concept of "service chaining".) In our example, the administrator wants to route all HTTP traffic through the *policy chain* Firewall-IDS-Proxy and the remaining traffic through the chain Firewall-IDS. Note that many middleboxes are stateful and need to process both directions of a session for correctness.

Opportunity: Today, middleboxes are placed at manually induced chokepoints and the routing is carefully crafted to ensure stateful traversal. In contrast to this semi-manual and error-prone process, SDN can programmatically ensure correctness of middlebox traversal. Furthermore, SDN allows administrators to focus on *what* policy they need to realize without worrying about *where* this is enforced. Consequently, SDN allows more flexibility to route around failures and middlebox overload and incorporate off-path middlebox capabilities [73].



	Physical Sequence
FW1-IDS1-Proxy1	S1 S2 FW1 S2 S4 S5 IDS1 S5 S4 S2 Proxy1 S2 S4 S5 S6
FW2-IDS1-Proxy1	S1 S3 FW2 S3 S5 IDS1 S5 S4 S2 Proxy1 S2 S4 S5 S6
FW1-IDS1	S1 S2 FW1 S2 S4 S5 IDS1 S5 S6
FW2-IDS1	S1 S3 FW2 S3 S5 IDS1 S5 S6

Figure 3.1: Example to illustrate the requirements that middlebox deployments place on SDN. The table shows the different physical sequences of switches and middleboxes used to implement the two logical policy chains: Firewall-IDS and Firewall-IDS-Proxy.

Challenge = Data plane mapping: Consider the *physical sequence* of middleboxes FW1-IDS1-Proxy1 for HTTP traffic in the example. Let us zoom in on the three switches S2, S4, and S5 in Figure 3.2. Here, S5 sees the same packet thrice and needs to decide between three actions: forward it to IDS1 (post-firewall), forward it back to S2 for Proxy1 (post-IDS), or send it to the destination (post-proxy). It cannot, however, make this decision based only on the packet header fields. The challenge here is that even though we have a valid composition of middlebox actions, this may not be realizable because S5 will have an ambiguous forwarding decision. This suggests that the use of simple flow-based rules (i.e., the IP 5-tuple) traditionally used for L2/L3 functions will no longer suffice.

3.3.2 Middlebox resource management

Middleboxes involve complex processing to capture application-level semantics and/or use deep packet inspection. Studies show that middlebox overload is a common cause of failures [74, 128], and thus an important consideration is to balance the load across middleboxes. For example, in Figure 3.1, we may want to divide the processing load equally between the two firewalls.

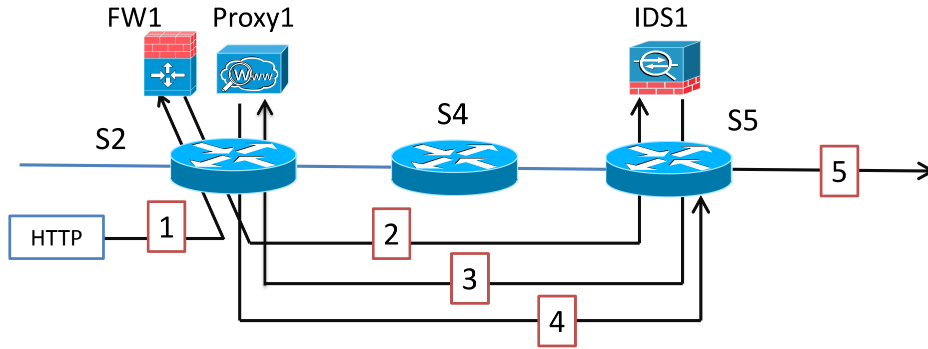


Figure 3.2: Example of potential data plane ambiguity to implement the policy chain Firewall-IDS-Proxy in our example topology. We annotate different instances of the same packet arriving at the different switches on the arrows.

Opportunity: Today, operators need to *statically* set up traffic splitting rules or employ custom load balancing solutions.[2] In contrast, a SDN controller can use data plane forwarding rules to flexibly implement load balancing policies and route traffic through specific *physical sequences* of switches and middleboxes in response to network dynamics [135].

Challenge = Data plane constraints: SDN switches are limited by the number of forwarding rules they can support; these rules are in TCAM and a switch can support a few thousand rules (e.g., 1500 TCAM entries in 5406zl switch [60]). In a large enterprise network with $O(100)$ firewalls and $O(100)$ IDSes [124], there are $O(100 \times 100)$ possible combinations of the Firewall-IDS sequence. Imagine a load balancing algorithm that splits the traffic uniformly across all such combinations. Now, each such split needs to have forwarding rules to route the traffic to the correct physical middleboxes. Thus, in the worst case, a switch in the middle of the network that lies on paths between these firewalls and IDSes may need $O(100 \times 100)$ forwarding rules. This an order of magnitude larger than today's switch capabilities [60]. In practice, the problem can be even worse—we will have several policy chains each with multiple middleboxes, e.g., each ingress-egress pair may have a policy chain per application port (e.g., HTTP, NFS). This implies that we cannot directly use existing middlebox load balancing algorithms as these do not take into account switch constraints [124].

[2]Our conversations with network operators reveals that they often purchase a customized load balancer for each type of middlebox!

3.3.3 Dynamic traffic transformation

Many middleboxes actively modify traffic headers and contents. For example, NATs rewrite the IP addresses of individual packets to map internal and public IPs. Other middleboxes such as WAN optimizers may spawn new connections and tunnel traffic over persistent connections.

In Figure 3.1, suppose there are two user groups accessing websites through Proxy1 in an enterprise: The employee user group from source subnet 10.1.1.0/24 should follow middlebox policy Proxy-Firewall; while the guest user group from subnet 10.1.2.0/24 should follow middlebox policy Proxy-IDS. The proxy delivers the traffic from different websites to users in the two user groups. Unfortunately, the traffic exiting the proxy may have different packet headers, sessions, and payloads compared to the traffic entering it. Thus, it is challenging for the controller to install rules at S2 to steer the appropriate traffic to the Firewall or IDS (depending on the original user group).

Opportunity: In order to account for such dynamic packet transformations, operators today have to resort to ad hoc measures: (1) placing middleboxes carefully (e.g., placing Firewall and IDS after the proxy to ensure all traffic traverses all middleboxes); or (2) manually reason about the correctness based on coarse models of middlebox behaviors. While these stop-gap measures may work, they make the network brittle as it needlessly constrains legitimate traffic (e.g., if the chokepoint fails) and may also allow unwanted traffic to pass through (e.g., if we use wildcard rules). Using a network-wide view, SDN can address these concerns by taking into account such dynamic packet transformations.

Challenge = Controller visibility: Ideally, the SDN controller needs to be aware of the internal processing logic of middleboxes in order to account for traffic modifications before installing forwarding rules. This logic, however, may be proprietary to the middlebox vendors. Furthermore, these transformations may occur on fine-grained timescales and depend on the specific packets flowing through the middlebox. This entails the need to automatically adapt to such middlebox-induced packet transformations.

In summary, we see that middleboxes introduce new opportunities for SDN to reduce the complexity involved in carefully planning middlebox placements and semi-manually setting up

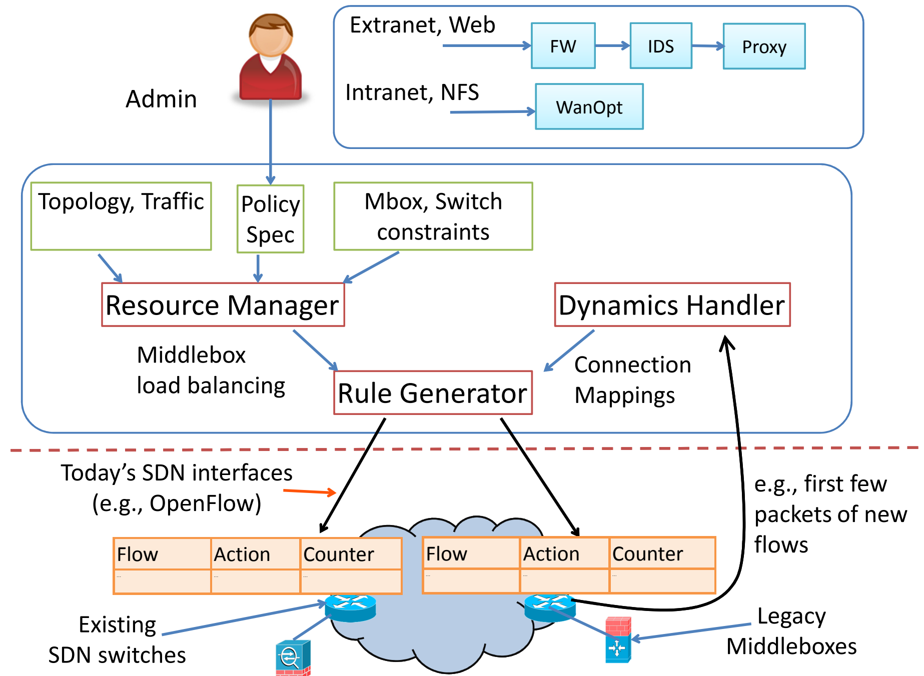


Figure 3.3: Overview of the SIMPLE approach for using SDN to manage middlebox deployments.

forwarding rules to implement the middlebox policies in an efficient load-balanced manner. At the same time, however, there are new challenges for SDN—data plane support for composition, managing both switch and middlebox resources efficiently, and incorporating middlebox-induced dynamic transformations.

3.4 SIMPLE System Overview

Our goal in this work is to address the challenges from the previous section without modifying middleboxes and working within the constraints of the existing SDN switches and today's SDN standards (i.e., OpenFlow). Our solution, called SIMPLE, is an SDN-based policy enforcement layer that translates a high-level middlebox policy into an efficient and load balanced data plane configuration that steers traffic through the desired sequence of middleboxes.

Figure 3.3 gives an overview of the SIMPLE architecture showing the inputs needed for various components, the interactions between the modules, and the interfaces to the data plane. Note that SIMPLE only needs to configure SDN-enabled switches; middleboxes do not need to be extended

to support new SDN-like capabilities. We begin by describing the high-level inputs to SIMPLE:

1. **Processing policy:** Building on the SDN philosophy of direct control, we want network administrators to specify *what* processing logic needs to be implemented and not worry about *where* this processing occurs or how the traffic needs to be routed. Building on previous middlebox research [91, 92, 124], this policy is best expressed via a *dataflow* abstraction as shown. Here, the operator specifies different *policy classes* (e.g., external web traffic or internal NFS traffic) and the sequence of middlebox processing needed per class.

2. **Topology and traffic:** SIMPLE must ultimately translate the logical policy specification to the physical topology. Thus, it needs a network map indicating where middleboxes are located, the links between switches, and the link capacities. We also need an expected volume of traffic T_c traversing each policy class. Such inputs are typically already collected in network management systems [69].

   For simplifying our presentation, we assume that each middlebox is connected to the network via an SDN-enabled switch as shown in Figure 3.1; our techniques also apply to deployments where middleboxes act as a "bump-in-the-wire". We use M_j and S_k to denote a specific middlebox and switch respectively.

3. **Resource constraints:** There are two types of constrained resources: (1) packet processing resources (e.g., CPU, memory, accelerators) for different middleboxes and (2) amount of TCAM available for installing forwarding rules in the SDN switches. We associate each switch S_k with flow table capacity $TCAM_k$ (number of rules) and each middlebox M_j with a packet processing capacity $ProcCap_j$.[3]

   In addition, we need the per-packet processing cost across middleboxes and classes. For generality, we assume that these costs vary across middlebox instances (e.g., they may have specialized accelerators) and policy classes (e.g., HTTP vs NFS). Let $Footprint_{c,j}$ denote the per-packet processing cost for a packet belonging to class c at the middlebox M_j.

[3]We can extend this to model each type of resource (CPU, memory) separately, but avoid doing so for brevity.

Corresponding to the three high-level challenges outlined in the previous section, we envision three key modules in the SIMPLE controller as shown in Figure 3.3.

1. The **ResMgr** module takes as input the network's traffic matrix, topology, and policy requirements and outputs a set of middlebox processing assignments that implement the policy requirements. This module takes into account both middlebox and switch constraints in order to optimally balance the load across middleboxes.

2. The **DynHandler** module automatically infers mappings between the incoming and outgoing connections of middleboxes that can modify packet/session headers. To this end, it receives packets (from previously unseen connections) from switches that are directly attached to the middleboxes. It uses a lightweight *payload similarity* algorithm to correlate the incoming and outgoing connections and provides these mappings to the RuleGen module described next.

3. The **RuleGen** module takes the output of the ResMgr (i.e., the processing responsibilities of different middleboxes) and the connection mappings from the DynHandler and generates data plane configurations to route the traffic through the appropriate sequence of middleboxes to their eventual destination. In addition, the RuleGen also ensures that middleboxes with stateful session semantics receive both the forward and reverse directions of the session. As we discussed, these configurations must make efficient use of the available TCAM space and avoid the ambiguity that arises due to composition that we saw in Section 3.3.1 . Thus, we need an efficient data plane design that supports these two key properties.

Conceptually, we envision the ResMgr and DynHandler running as controller applications while the RuleGen can be viewed as an extension to the network operating system [77]. We envision SIMPLE as a proactive controller for the common case of middleboxes that do not modify packet headers to avoid the extra latency of per-flow setup. By construction, the DynHandler is a reactive component as it needs to infer the connection mappings on the fly.

3.5 SIMPLE Data Plane Design

There are two high-level requirements for the SIMPLE data plane. First, as we saw in Figure 3.2, a switch cannot rely on the flow 5-tuple for forwarding. Second, we need to ensure that the rules can fit within the limited TCAM which will be especially critical for larger networks with middleboxes distributed throughout the network. To address these problems, we present a data plane solution that uses a combination of tags and tunnels. While the use of tagging or tunneling in a general networking or SDN context is not new, our specific contribution here is in using these ideas in the context of middlebox policy enforcement.

To simplify our discussion in this section, we start by assuming that middleboxes do not change the IP 5-tuple. They may, however, arbitrarily change payloads and other fields (e.g., VLAN ids, MPLS, ToS fields etc.). We relax this assumption in §3.6.

3.5.1 Unambiguous forwarding

Referring back to Figure 3.2, S5 needs to know if a packet has traversed the Firewall (send to IDS), or traversed both Firewall and IDS (send to S2), or all three middleboxes (send to dst) to know the next hop. That is, we need switches to identify the *segment* in the middlebox processing chain that the packet is currently in; a segment is a sequence of switches starting at a middlebox (or an ingress gateway) and terminating at the next middlebox in the logical chain. Intuitively, we can track the segment by keeping per-packet state in the controller or in the switch. As neither option is practical, we use a combination of topological context and packet tagging to encode this processing state.

- *Based on input port when there are no loops:* The easy case is when the sequence of switches is *loop free*; i.e., each directional link appears at most once in the sequence. In this case, a switch can use the incoming interface to identify the logical segment. Consider the sequence FW1-IDS1 in Figure 3.4a, where the packet needs to traverse *In–S2-FW1-S2-S4-S5-IDS1-S5–Out*. In this case, S2 forwards packets arriving on "In" to FW1 and packets arriving on the FW1 port to S4.
- *Based on ProcState tags when there are loops:* If there is a loop in the physical sequence, then the combination of input interface and packet header fields cannot identify the middlebox segment.

To address this, we introduce a ProcState tag that encodes the packet's processing state; ProcState tags are embedded inside the packet header using either VLAN tags, MPLS labels, or unused fields in the IP header depending on the fields supported in the SDN switches. The controller installs *tag addition* rules at the first switch of each segment based on packet header fields and input ports. Downstream switches use these tags in their forwarding action.

Figure 3.2 shows tag addition rules at S2: $\{$HTTP, from FW1$\}$ $\rightarrow$ ProcState =FW; $\{$HTTP, from Proxy1$\}$ $\rightarrow$ ProcState =Proxy. The forwarding rules at S5 are: $\{$HTTP, ProcState =FW$\}$ $\rightarrow$ forward to IDS1; and $\{$HTTP, ProcState =Proxy$\}$ $\rightarrow$ forward to destination. The key idea here is that S5 can use the ProcState tags to differentiate between the first instance of the packet arriving in the second segment (send to IDS) and the fourth segment (send to destination).

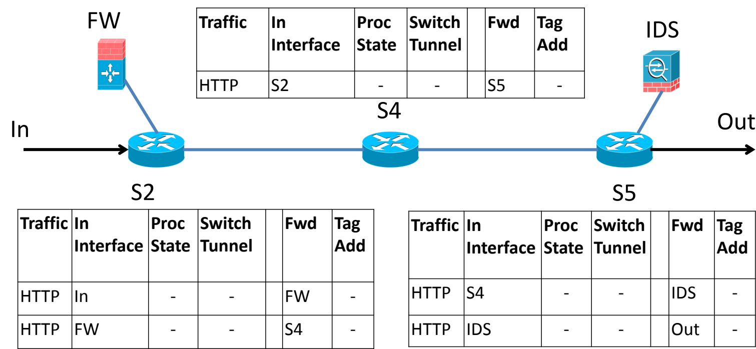
3.5.2 Compact forwarding tables

In the simplest case, we use *hop-by-hop* forwarding rules at every switch along a physical sequence as shown in Figure 3.4a. While this works for small topologies, it does not scale to large topologies with many switches, multiple middlebox policy chains, and many possible physical instantiations of a specific policy chain. To reduce the number of forwarding entries, we leverage the observation that switches in the middle of each segment of a physical sequence do not need fine-grained forwarding rules. The only role they serve is to route the packet toward the switch connected to the next middlebox in the sequence.

Building on this insight, we use *inter-switch tunnels* or SwitchTunnels between all pairs of switches. Here, each switch maintains two forwarding tables: (1) a FwdTable specifying fine-grained per-flow rules for middlebox traversal and (2) a TunnelTable indicating how to reach every other switch in the network, similar to DiFane [145]. The TunnelTable is computed using traditional routing metrics by the SDN controller. The TunnelTable can be implemented in TCAM using OpenFlow rules or in SRAM [145].[4]

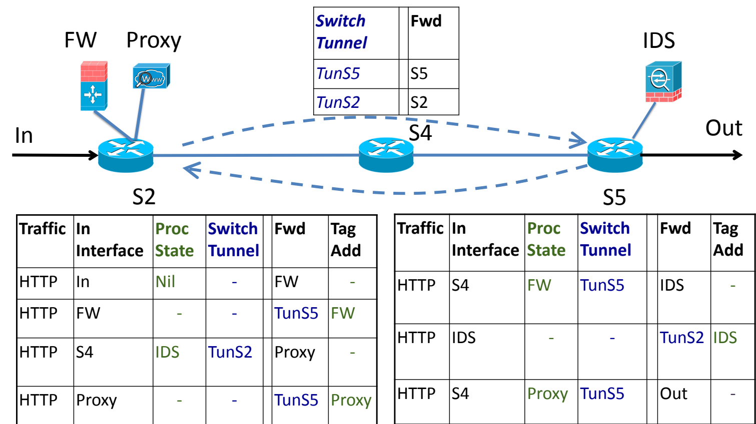
With this in place, the ingress switch tunnels packets to the switch connected to the first middlebox in the sequence. A switch in the middle of a segment uses its TunnelTable to forward packets

[4]DiFane maintains tunnel entries to each egress. SIMPLE needs entries to each egress and switches connected to middleboxes.



Policy = Rest: FW→IDS

(a) Hop-by-hop, No loop



Policy = HTTP: FW →IDS →Proxy

(b) Tunnel, Loop

Figure 3.4: Example of SIMPLE data plane configurations. The other cases: hop-by-hop with loop and SwitchTunnels with no loop are similar and are not shown for brevity.

through the SwitchTunnel toward the next middlebox. Switches directly connected to middleboxes are responsible for forwarding packets to the middlebox and marking packets with the next SwitchTunnel entry. Note that switches terminating a middlebox segment need fully descriptive rules (similar to the hop-by-hop case) to forward traffic to/from the middlebox. We demonstrate the practical benefits of using SwitchTunnels in §3.9.1.

Example: To see how this works, we revisit the example from §2 in Figure 3.4b. This scenario uses SwitchTunnels in conjunction with ProcState because the sequence has a loop. We focus first on the SwitchTunnels aspect. The key idea is that instead of rules specifying the next hop, switches connected to middleboxes tunnel traffic to the switch attached to the next middlebox.

This is indicated by the TunS5 entries in the Fwd actions at S2 for traffic incoming from FW and Proxy and the TunS2 entry at S5 for traffic incoming from IDS. Note that S4, a switch with no middleboxes attached, does not need any fine-grained forwarding rules; it uses the SwitchTunnel to look up its TunnelTable (shown in italics). S2 (and similarly S5) checks whether there are terminals for the SwitchTunnel to see if they need to forward the packet to a locally attached middlebox.

The figure also shows the corresponding ProcState to distinguish different instances of the same packet arriving at the same switch. Note that SwitchTunnels alone do not solve the ambiguity problem caused by loops; we may have packets traversing the same tunnel twice and thus we will still need ProcState tags. Again, the switches connected to the middleboxes (S2, S5) are responsible for adding the ProcState and for checking these while making forwarding decisions to the next middlebox in sequence.

3.6 SIMPLE Dynamics Handler

The key remaining issue in installing forwarding rules is that middleboxes may dynamically modify the incoming traffic—when middleboxes modify flows' packet headers, the forwarding rules on downstream switches must account for the new header fields. For example, when a NAT translates the external address to the internal one, the controller must be aware of such translations and install correct forwarding rules to direct traffic to the next middlebox or egress switch.

3.6.1 Design constraints

Table 3.1 summarizes the different types of middleboxes commonly used in enterprises today and annotates them with key attributes: the type of traffic input they operate on, their actions, and the timescales at which the dynamic traffic modifications occur. For example, an IP firewall checks both the packet header information, and makes a decision on whether to drop the packet or forward it, while a NAT checks the source and destination IP and port fields in the packet headers and rewrites these fields. Note that vendors may differ in their logic for the same class of middlebox. For example, different NAT implementations may either randomly or sequentially increase the port number when a new host connects to it. In summary, we see that middleboxes operate at different timescales, modify different packet headers, and operate at diverse granularities (e.g., packet vs.

flow vs. session).

Ideally, we would like fine-grained visibility into the processing logic and internal state of each middlebox to account for such transformations. The longer-term option is standardized APIs for middleboxes to export such information [66, 71]. Given the vast array of middleboxes [125], large number of middlebox vendors [44], and the proprietary nature of these functions, achieving standardized APIs and requiring vendors to expose internal states does not appear to be a viable near-term solution.

Given the diverse and proprietary nature of this ecosystem and our explicit stance to avoid modifying middleboxes, we follow the following driving principle. Rather than model middleboxes or ask network operators to specify the dynamic behaviors of middleboxes, we treat middleboxes as blackboxes and try to automatically learn their relevant input-output behaviors. In this work, we take a *protocol-agnostic* approach to see how much accuracy we can achieve with a general framework. As we show later (§3.9.3), we get close to 95% matching accuracy with only a few packets overhead. By adding protocol-specific state (e.g., HTTP state machines) or incorporating middlebox-specific information, we can further improve this accuracy.

3.6.2 Idea: Flow correlation

The natural question is why do we think this is feasible? Note that we do not need visibility into the internal proprietary logic of the middlebox. We only need to reason about the middlebox behaviors pertinent to forwarding and policy enforcement. That is, we only need to identify how the incoming and outgoing flows (or sessions) at the middlebox are correlated.[5]

Consider the following physical path traversed by a packet: $S_k \rightarrow M_j \rightarrow S_k$. With respect to the middlebox, we have an incoming set of flows, $Incoming(S_k \rightarrow M_j)$ and an outgoing set, $Outgoing(M_j \rightarrow S_k)$. Our goal is to identify which flow(s), $F \in Incoming$ is (are) causally related to some flow(s) in $Outgoing$.

In the simplest case, middleboxes (e.g., Firewall) do not change the packet headers and do not multiplex/spawn flows. In this case, we can directly map the incoming and outgoing flows. (This

[5]We were inspired in part by the success of flow correlation techniques used in the security literature to detect stepping stones and information leakage [147]. Our problem is arguably simpler than the security setting: the middlebox is a blackbox, not adversarial.

Middlebox	Input	Actions	Timescale	Info needed	Approach
FlowMon	Header	No change	–	None	–
IDS	Header, Payload	No change	–	None	–
IP Firewall	Header	Drop?	–	None	–
IPS	Header, Payload	Drop?	–	None	–
Redundancy eliminator	Payload	Rewrite payload	Per-packet	None	–
NAT	Flow	Rewrite header	Per-flow	Header mapping	Payload Match
Load balancer	Flow	Rewrite headers & reroute	Per-flow	Session mappings	Payload Match
Proxy	Session	Map sessions	Per-session	Session mappings	Similarity Detector
WAN-Opt	Session	Map sessions	Per-session	Session mappings	Similarity Detector

Table 3.1: A taxonomy of the dynamic actions performed by different middleboxes that are commonly used today [124] and the corresponding information that we need to infer at the SDN controller.

is marked as *None* in the information needed column in Table 3.1).

Oher middleboxes (e.g., NAT) may change packet header fields, but do not change the packet payloads and are also flow preserving. Consider a NAT that simply rewrites headers. In this case, there is a one-to-one correspondence between the incoming and outgoing packets. Moreover, the payloads of the packets are unmodified. Thus, we can simply do an *exact payload match* between the incoming and outgoing packets to detect the flow correlations (labeled as *payload match* in the table).

The more challenging case is when the middleboxes may create new sessions or merge existing sessions (e.g., proxy, WAN optimizer). For these middleboxes, we cannot directly match the payloads of individual packets because one flow into a middlebox can be mapped to multiple flows going out of the middlebox, and vice versa. In other words, we do not have a bijection between $Incoming$ and $Outgoing$ any more. For example, the proxy may merge multiple users' requests to the same website into a single request, change the HTTP fields in a request header (e.g., using HTTP protocol 1.1 instead of 1.0), prefetch contents, and serve requests from cached responses for popular websites. We discuss our solution for this case next.

3.6.3 Similarity-based correlation

To make this discussion concrete, we focus on the proxy scenario as it is the most challenging case—it changes headers, modifies payloads, and does not maintain a one-to-one correspondence between incoming and outgoing flows.

In this case, we observe that even though the traffic is not identical after it traverses the middlebox, the payloads will still have a significant amount of *partial overlap*. For example, in the case of web content delivered through the proxy to the user, even though the initial HTTP preambles may differ between the incoming and outgoing flows, the web page content will still match. Thus, we leverage *Rabin fingerprints* [58, 115] to calculate the (partial) similarities across flows. Because middleboxes are typically session-oriented and only keep a limited amount of state on each incoming flow, we only need to correlate this flow to the outgoing flows that appear within a small *time window*. To this end, we leverage the switches to forward packets that do not match the flow table rules to the SDN controller for further inspection.

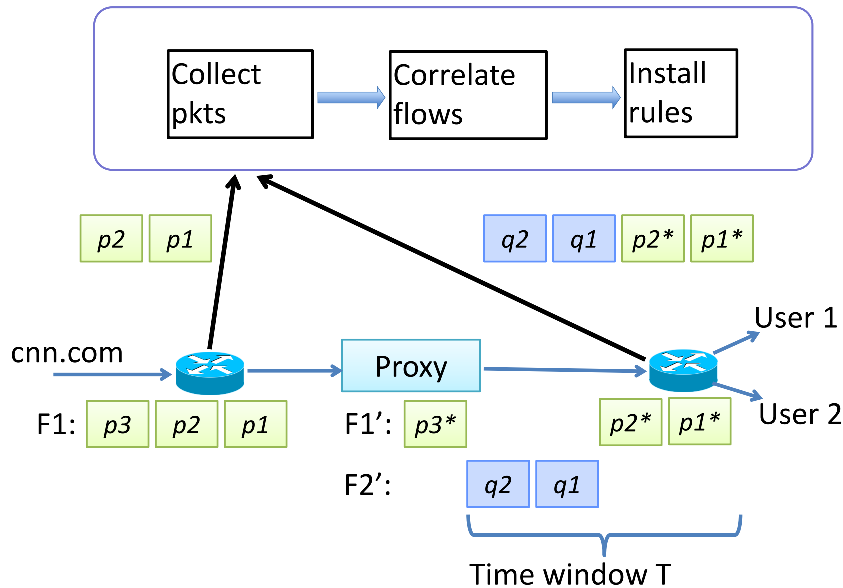


Figure 3.5: Similarity based correlation of incoming and outgoing flows through a middlebox.

Given these insights, the SIMPLE DynHandler runs a similarity-based correlation algorithm in three steps (Figure 3.5):

(1) Collect packets: When a new flow (e.g., $F1$) arrives from the Internet to the middlebox, the switch sends the first P packets of the new flow to the controller (e.g., $p1$ and $p2$ in Figure 3.5). Similarly, we collect the first P packets for all the flows going out of the middlebox within a time window W (e.g., the packets $p1^*$ and $p2^*$ for flow $F1'$ and packets $q1$ and $q2$ for flow $F2$). The controller reconstructs the payload stream from the P packets collected for each flow [111]. W here controls the search scope of flows that may be correlated and P controls the bandwidth and processing overhead of the controller.

(2) Calculate payload similarity: As discussed earlier, the middlebox may modify or reorder part of the stream, and thus we cannot directly compare payloads. We compute a similarity score which calculates the amount of overlap between every pair of flows. Because dividing the data stream into fixed size chunks is not robust (e.g., a middlebox may shift the content by adding or removing some data), we leverage Rabin fingerprints [58] to divide the stream into shift-tolerant chunks. Let

the number of chunks from the two payload streams with the same hash value be N^{common}. Then, the *similarity score* for the pair of streams is $N^{common} / \min(N_1, N_2)$, where N_1, N_2 are the number of chunks for the two streams.

(3) Identify the most similar flows: We identify the flow going out of the middlebox that has the highest similarity score with the new incoming flow. If there are multiple outgoing flows with the same highest similarity, we identify all these flows as correlated with the incoming flow. For example in Figure 3.5, we may find that $F1$ has higher similarity with $F1'$ than $F2'$.

Policy-specific optimizations: The two parameters W and P together determine the bandwidth and computation overhead of the controller to run the correlation step. We can tune the bandwidth and processing overhead of the DynHandler based on the middlebox policies the operators want to enforce. For instance, we may want to achieve higher accuracy even at the expense of higher overhead for security-sensitive policies. This is because different policies may require different granularities of correlation accuracy. Let us consider two specific policies in our proxy example: *(1) Stateful access control:* The operators may only allow incoming traffic from websites for which users have initiated the visits and *(2) User-specific policies:* The operators may want traffic to/from a subset of hosts to go through a IDS after the proxy. In case (2), we need to correlate the incoming flow with the actual user, while in case (1), we only need to correlate the incoming flow with the flows to *any* of the users. As a result, we need lower correlation accuracy for case (1), and thus can reduce both the time window W and the number of packets P sent to the controller.

3.7 Resource Management

The key challenge in the ResMgr is the need to account for both the middlebox constraints and the flow table capacity of SDN switches. This makes the problem significantly more challenging compared to prior optimization models for middlebox load balancing (e.g., [83, 124]). Unfortunately, this optimization problem is NP-hard and is practically inefficient to solve for realistic scenarios (§3.9.2). Due to space constraints, we do not show the formal hardness reduction; at a high-level the intractability is due to the integer constraints necessary to model the switch table sizes.

3.7.1 Offline-Online Decomposition

We address this challenge by decomposing the optimization into two parts: (1) an offline stage where we tackle the switch constraints and (2) an online linear program formulation that only deals with load balancing (see Figure 3.6). The offline pruning stage only needs to run when the network topology, switches, middlebox placements, or the network policy changes. The online load balancing stage runs more frequently when traffic patterns change on shorter timescales.

The intuition here is that the physical topology and middlebox placement are unlikely to change on short timescales. Based on this, we run an offline *pruning* stage where given a set of logical chains, we select a subset of the available physical sequences that will not violate the switch capacity constraints. In other words, there is sufficient switch capacity to install forwarding rules to route traffic through all of these sequences simultaneously. In this step, we ensure that we have sufficient degrees of freedom; e.g., each $PolicyChain_c$ will have a guaranteed minimum number of distinct physical sequences and that no middlebox becomes a hotspot.

Given this *pruned set*, we formulate the load balancing problem as a simpler linear program. While we do not prove the optimality of our decomposition, we can intuitively reason about the effectiveness—with high Cov we can achieve a close-to-optimal solution as it yields sufficient flexibility for load balancing. Our results (§3.9.2) show that we find near-optimal solutions ($\geq$ 99% of optimal) for realistic network topologies and configurations.

3.7.2 Offline ILP-based pruning

Modeling switch resource usage: For each chain $PolicyChain_c$, we do a brute-force enumeration of all possible *physical middlebox sequences* implementing it. In Figure 3.1, the set of all middlebox sequences for the chain Firewall-IDS is {FW1-IDS1, FW2-IDS1}. Let $PhysSeq_c$ denote the set of all physical sequences for $PolicyChain_c$; $PhysSeq_{c,q}$ denotes one instance from this set. We use $M_j \in PhysSeq_{c,q}$ to denote that the middlebox is part of this physical sequence.

The main idea here is that in order to route traffic through this sequence, we need to install forwarding rules on switches on that route. Let $Route_{c,q}$ denote the switch-level route for $PhysSeq_{c,q}$ and let $Rules_{k,c,q}$ denote the number of rules that will be required on switch S_k to route traffic

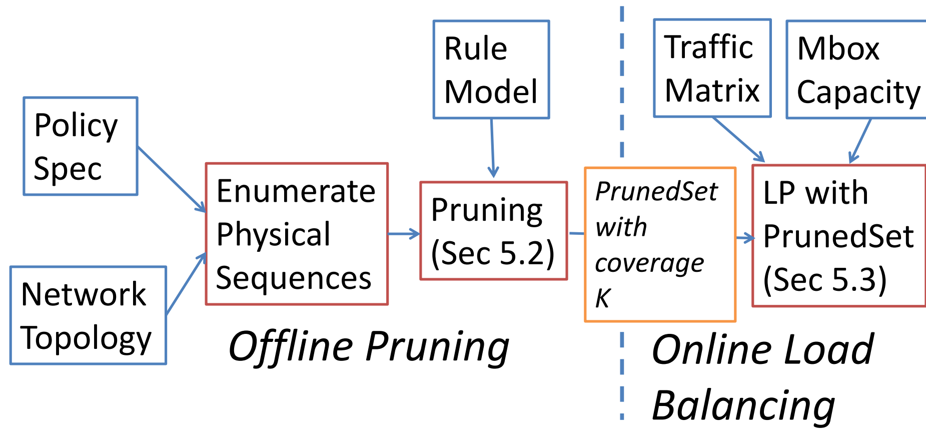


Figure 3.6: High-level overview of the offline-online decomposition in the ResMgr.

through $Route_{c,q}$. Now, the value of the $Rules_{k,c,q}$ depends on the type of forwarding scheme we use. To see why, let us revisit the data plane solutions from §3.5.

1. *Hop-by-hop:* Here, $Rules_{k,c,q}$ is simply the *number of times* a switch appears in the physical sequence, i.e., each switch needs a forwarding rule corresponding to every incoming interface on this path.

2. *Tunnel-based:* In this case, switches in the middle of a tunnel segment do not need rules specific to $PhysSeq_{c,q}$; they use the TunnelTable independent of $PhysSeq_{c,q}$. On the other hand, switches attached to a middlebox need two non-tunnel rules to forward traffic to and from that middlebox.[6] Consider the physical sequence S1-S2-FW1-S2-S4-S5-IDS1-S5-S6. Here, S2 and S5 need two rules to steer traffic in/out of the middleboxes but the remaining switches do not need new rules.

Integer linear program (ILP) Formulation: There are two natural requirements: (1) The switch constraints should not be violated given the pruned set of sequences, and (2) Each logical chain should have enough physical sequences assigned to it, so that we retain sufficient freedom to

[6]As a special case, the ingress and egress switches will also need a non-tunnel rule to map the 5-tuple to a tunnel.

$$Minimize \;\; MaxMboxOccurs, subject\ to \tag{3.1}$$

$$\forall c: \sum_{q} d_{c,q} \geq Cov \tag{3.2}$$

$$\forall k: \sum_{\substack{c,q\ s.t. \\ S_k \in PhysSeq_{c,q}}} Rules_{k,c,q} \times d_{c,q} \leq TCAM_k \tag{3.3}$$

$$\forall j: \; MboxUsed_j = \sum_{c,q\ s.t. M_j \in PhysSeq_{c,q}} d_{c,q} \tag{3.4}$$

$$\forall j: \; MaxMboxOccurs \geq MboxUsed_j \tag{3.5}$$

$$\forall c, q: d_{c,q} \in \{0, 1\} \tag{3.6}$$

Figure 3.7: Integer Linear Program (ILP) formulation for pruning the set of physical sequences to guarantee coverage for each logical chain while respecting switch TCAM constraints.

achieve near-optimal load balancing subsequently.

We model this problem as an ILP shown in Figure 3.7. We use binary indicator variables $d_{c,q}$ (Eq (3.6)) to denote if a particular physical sequence has been chosen. To ensure we have enough freedom to distribute the load for each chain, we define a target *coverage level* Cov such that each $PolicyChain_c$ will have at least Cov distinct $PhysSeq_{c,q}$ assigned to it in Eq (3.2). We constrain the total switch capacity used in Eq (3.3) to be less than the available TCAM space. Here, the number of rules depends on whether a given sequence is "active" or not. (Note that this conservatively assumes that there will be some traffic routed through this sequence and thus we will need a forwarding rule.)

At the same time, we want to make sure that no middlebox becomes a hotspot; i.e., many sequences rely on a specific middlebox. Thus, we model the number of chosen sequences in which a middlebox occurs and also the maximum occurrences across all middleboxes in Eq (3.4) and Eq (3.5) respectively. Our objective is to minimize the value of $MaxMboxOccurs$ to avoid hotspots. Since we do not know the optimal value of Cov, we use binary search to identify the largest feasible value for Cov.

By construction, formulating and solving this problem as an exact ILP guarantees that if there is a feasible solution, then we will find it. While solving an ILP might take a long time for a large network, we note that this is an infrequent operation that only needs to be run when the topology changes. Furthermore, we find that the time for pruning is only $\approx 1800\,\text{s}$ even for a 250-node topology (§3.9.2).

3.7.3 Online load balancing with LP

Having selected a set of feasible sequences in the pruning stage, we formulate the middlebox load balancing problem as a *linear program* shown in Figure 3.8. The main control variable here is $f_{c,q}$, the *fraction of traffic* for $PolicyChain_c$ that is assigned to each (pruned) physical sequence $PhysSeq_{c,q}$.

First, we need to ensure that all traffic on all chains is assigned to some physical sequence; i.e., these fractions add up to 1 for each c (Eq (3.8)). Next, we model the load on each middlebox in terms of the total volume of traffic and the per-class footprint across all physical sequences it is a part of (Eq (3.9)). Note that we only consider the physical sequences that are part of the pruned set generated from the previous section. Also note that the f variables are continuous variables in $[0, 1]$ unlike the d variables which were binary variables. We pick a specific load balancing objective to minimize the maximum middlebox load across the network (Eq (3.10)). That said, this framework is general enough to accommodate other load balancing goals as well. The ResMgr solves the LP to obtain the optimal $f_{c,q}$ values and outputs these to RuleGen.

3.7.4 Extensions

Handling node and link failures: While we expect the topology to be largely stable, we may have transient node and link failures. In such cases, the pruned set may no longer satisfy the coverage requirement for each $PolicyChain_c$. Fortunately, we can address this by precomputing pruned sequences for different switch, middlebox, and link failure scenarios.

Handling policy changes: We also expect middlebox policy changes to occur at relatively coarse timescales. The flexibility that SIMPLE enables, however, may introduce dynamic policy invocation scenarios; e.g., route through a packet scrubber if we observe high load on a web server.

$$\text{Minimize } MaxMboxLoad \tag{3.7}$$

$$\forall c: \sum_{q: PhysSeq_{c,q} \in Pruned} f_{c,q} = 1 \tag{3.8}$$

$$\forall j: \ Load_j = \frac{\sum_{\substack{c,q \ s.t. M_j \in PhysSeq_{c,q} \\ PhysSeq_{c,q} \in Pruned}} f_{c,q} \times T_c \times Footprint_{c,j}}{ProcCap_j} \tag{3.9}$$

$$\forall j: MaxMboxLoad \geq Load_j \tag{3.10}$$

$$\forall c, q: f_{c,q} \in [0, 1] \tag{3.11}$$

Figure 3.8: Linear Program (LP) formulation for balancing load across middleboxes given a pruned set.

Given that there are only a finite number of middlebox types and a few practical combinations, we can precompute pruned sets for dynamic policy scenarios as well.

Other traffic engineering goals: The load balancing LP can be extended to incorporate other traffic engineering goals as well. For example, given the traffic assignments, we can model the load on each link and constrain it such that no link is more than 30% congested. We do not show these extensions due to space constraints.

3.8 Implementation

In this section, we describe our SIMPLE prototype (using POX [33]) following the structure in Figure 3.3.

RuleGen: For each class c, RuleGen identifies the ingress-egress prefixes and partitions the traffic into smaller sub-prefix pairs in the ratio of the $f_{c,q}$ values [135]. It initially assumes that the traffic is split uniformly across sub-prefixes; it uses the rule match counts from the switches to rebalance the load if the traffic is skewed. To generate the rules, it makes two decisions. First, it chooses a SwitchTunnel or hop-by-hop scheme based on network size. Second, for each sequence $PhysSeq_{c,q}$, it checks for loops to add ProcState tags. We currently use VLAN or ToS fields. While we describe our design in the context of uni-directional flows for clarity, RuleGen ensures correctness for stateful middleboxes by setting up forwarding rules for the reverse path as well.

Rule checking: We implement *verification scripts* that take the rules generated by the RuleGen module to check for two properties: (1) Every packet that requires $PolicyChain_i$ goes through some sequence that implements this chain; and (2) A packet should not traverse a middlebox if the policy does not mandate it. For middleboxes that do not change packet header fields, our data plane mapping guarantees the above two properties by construction. When middleboxes change packet header fields, the controller can verify these properties by combining the header space analysis [93] and the similarity-based correlation in the DynHandler. First, we understand how an incoming flow F_1 to a middlebox M_1 maps to outgoing flow(s) F_1^*. Next, we leverage the header space analysis of rules at switches to understand the reachability of flows F_1^* between two middleboxes along the physical chain (say, between M_1 and M_2). By iterating across all the middleboxes, we can understand the end-to-end reachability for different flows and verify if it matches operator's policies.

ResMgr: The ResMgr uses `CPLEX` for LP-based load balancing and the ILP-based pruning step. We currently support all single link, switch, and middlebox failure scenarios. We also implement an optimization to *reuse* the previously computed solution to bootstrap the solver instead of starting from scratch.

DynHandler: We use existing SDN capabilities for the DynHandler. The SIMPLE controller installs rules at switches connecting to the middleboxes to retrieve the first few packets for each new flow. We use a custom implementation of the Rabin fingerprinting algorithm configured with an expected chunk size of 16 bits. (We found that this offers the best tradeoff between overhead and accuracy.) The DynHandler runs the correlation algorithm as described in §3.6 and provides the mappings to the RuleGen. The new inferred rules that account for the packet transformations are more specific than proactively installed rules (which use prefix aggregation). Note that these rules are on-demand and transient (i.e., they expire) in that they only need to last for the duration of a flow. We currently assume there is sufficient space to hold these dynamic rules.[7]

[7]For example, we can run the optimization step with an input parameter $TCAM'_k$ that is a small constant less than the actual $TCAM_k$ to accommodate these dynamic rules.

3.9 Evaluation

We use a combination of emulation-based evaluation in Emulab and Mininet, and trace-driven simulations. We do so to progressively increase the scale of our experiments to larger topologies given the resource constraints (e.g., node availability, VM scalability) that arises in each setup. Due to the lack of publicly available information on network topologies and middlebox-related policy, we use network topologies from past work [131, 124] as starting points to create augmented topologies with different middlebox placements. We assume a gravity-model traffic matrix for the topologies except Figure 3.1. We use OpenvSwitch (v 1.7.1) [27] as the SDN switch and use custom Click modules to act as middleboxes [95].

System Benchmarks

Setup: In each topology, every switch has a "host" connected to it and every switch has at most one middlebox. Every pair of hosts has a policy chain of three (distinct) middleboxes. We use `iperf` running on the hosts to emulate different traffic matrices and use different port number/host addresses to distinguish traffic across chains. Each link has an emulated bandwidth of 100 Mbps.

Platform, Config	Time to Install Rules(s)	Overhead (B)	Max MB Load (KB/s)	Max Link Utilization (KB/s)
Emulab, SIMPLE	0.041	5112	25.2	25.2
Mininet, SIMPLE	0.039	5112	25.2	25.2

Table 3.2: End-to-end metrics for the topology in Figure 3.1 on Emulab and Mininet. Having confirmed that the results are similar, we use Mininet for larger-scale experiments.

Topology	#Switches, #Hosts, #Mboxes	#Rules	Time (s)	Overhead (KB)
Figure1	6, 2, 4	36	0.04	5
Internet2	11, 11, 10	1699	0.09	180
Geant	22, 20, 20	6964	0.19	820
Enterprise	23, 23, 20	6689	0.31	710

Table 3.3: Time and control traffic overhead to install forwarding rules in switches.

We focus on three key metrics here: the time to install rules, the total communication overhead at the controller, and the maximum load on any middlebox or link in the network relative to the optimal solution. We begin by running the topology from Figure 3.1 on different physical machines on the Emulab testbed. We run the same setup on Mininet and check that the results are quantitatively consistent between the two setups in Table 3.2. We also check on a per-node and per-link basis that the loads observed are consistent between the two setups (not shown). Having confirmed this, we run larger topologies such as Internet2, Geant, and Enterprise using Mininet.

Time to install rules: Table 3.3 shows the time taken by SIMPLE to proactively install the forwarding rules for the four topologies in Mininet. The time to install is around 300 ms for the 23-node topology. The main bottleneck here is that the controller sends the rule tables to each switch in sequence. We can reduce this to 20 ms overall with multiple parallel connections. These are consistent with reported numbers in the literature [53, 122].

Controller's communication overhead: The table also shows the controller's communication overhead in terms of Kilobytes of control traffic to/from the controller to install rules. Note that there is no other control traffic (except for the DynHandler inference) during normal operation. These numbers are consistent with the total number of rules that we need to install.

3.9.1 Benefits of SIMPLE

Next, we use Mininet-based emulations with larger topologies to highlight the benefits that SIMPLE enables for middlebox deployments. As a point of comparison, we use a hypothetical *Optimal* system that uses the same logic as SIMPLE. The main difference is that instead of the optimization, it uses an exact ILP to solve a joint optimization with both switch and middlebox constraints without the pruning step (not shown).

Flexiblity in middlebox placement: We compare SIMPLE with today's *Ingress*-based middlebox deployments, where for each ingress-egress pair, the middleboxes closest to the ingress are selected. Here, we assume that there are two types of middleboxes Firewall and IDS and that each switch is attached to one instance of a Firewall and an IDS. As a point of reference, we consider a emulated *CoMb* setup with "consolidated" middleboxes [124]. Specifically, we emulate a unified

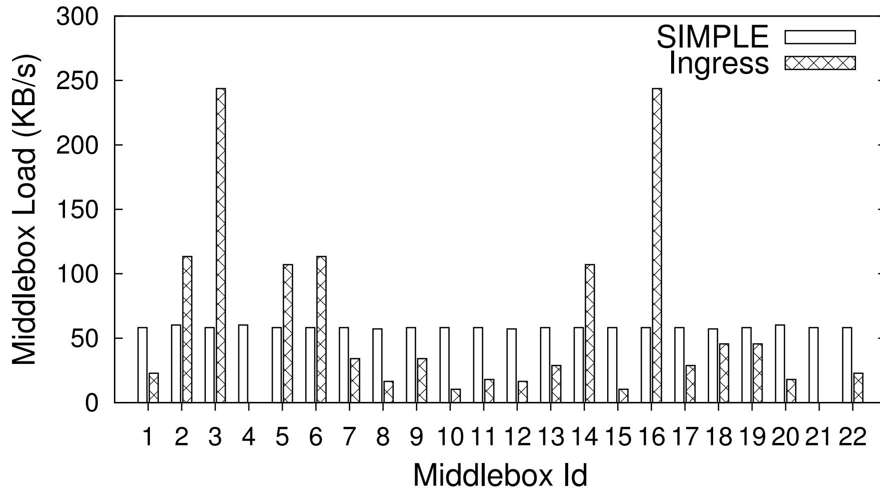


Figure 3.9: Load on all middleboxes for Internet2 topology.

Firewall+IDS middlebox with 2× capacity.

First, we look at the Internet2 topology and look at the per-middlebox loads in Figure 3.9. We see that SIMPLE distributes the load more evenly and can reduce the maximum load by almost 5×. Figure 3.10 shows the (normalized) maximum load across middleboxes with different configurations. First, SIMPLE is 3–6× better than today's ad hoc Ingress setup. Second, the performance gap between CoMb and SIMPLE is negligible—SIMPLE can achieve the same load balancing benefits as CoMb with unmodified middlebox deployments. It is worth noting that CoMb offers other benefits via module reuse and hardware multiplexing that SIMPLE does not seek to provide. The result here shows that the spatial distribution capabilities of SIMPLE and CoMb are similar.

Reacting to middlebox failure and traffic overload: We consider two dynamic scenarios in the Internet2 topology: (1) one of the middleboxes fails and (2) there is traffic overload on some of the chains. In both cases, we need to rebalance the load and we are interested in the time to reconfigure the network. Figure 3.11 shows a breakdown of the time it takes to rerun the SIMPLE LP,[8] generate new rules, and install them. We see that the overall time to react is low (<150 ms) and the overhead of the SIMPLE-specific logic is negligible compared to the time to install rules.

[8]We precompute pruned sets for single node failure scenarios.

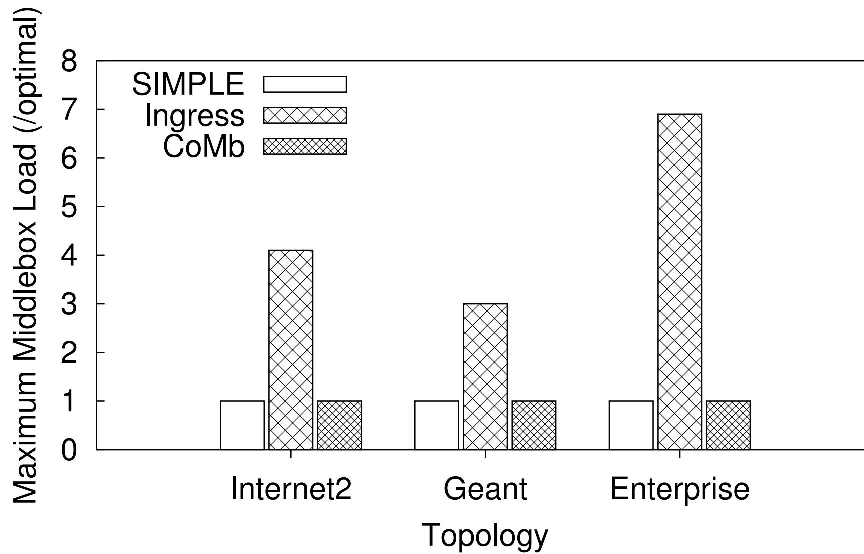


Figure 3.10: Maximum middlebox load comparison across topologies with SIMPLE, CoMb, today's Ingress-based deployments relative to the optimal ILP-based configuration.

Need for SIMPLE dataplane: One natural question is whether the ProcState tags are actually being used. Figure 3.12 shows that a non-trivial fraction of sequences selected by Optimal and SIMPLE do require ProcState tags. While one could argue that more careful placement could potentially eliminate the need for ProcState tags, we believe that we should not place the onus of such manual planning on operators. Moreover, under failure or overload scenarios, it might be necessary to use sequences with loops for correct policy traversal even with planned placements.

3.9.2 Scalability and optimality

Next, we focus on the scalability and optimality of the ResMgr using simulations on larger topologies. For brevity, we only show results assuming that each policy chain is of length 3. We vary two key parameters: (1) the available TCAM size in the switches and (2) the number of policy chains per ingress-egress pair.

Compute Time: Table 3.4 compares the time to generate the configurations along two dimensions: the type of optimization (i.e., Optimal vs. SIMPLE) and the forwarding scheme (i.e., with or without SwitchTunnels). SIMPLE lowers rule generation time by four orders of magnitude for

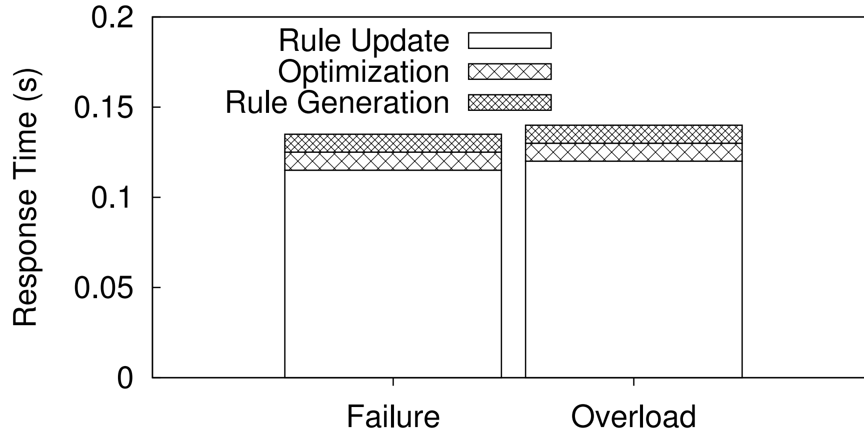


Figure 3.11: Response time in the case of a middlebox failure and traffic overload.

larger topologies. As a point to evaluate the scalability to very large topologies, we consider an augmented AS3356 graph (labeled as AS3356-aug) where we add 4 more "access" switches to every switch from the PoP-level topology. Even for this case, SIMPLE only takes ≈1 second. This is well within the typical timescales of traffic engineering decisions [69]. (The Optimal columns are empty because we gave up after a day.)

Optimality gap: We evaluate the optimality gap for all topologies and observe that across diverse configurations of switch capacity and the number of policy chains, SIMPLE is very close (99%) to the optimal in terms of the middlebox load (not shown).

Benefit of SwitchTunnels: Figure 3.13 shows that with SwitchTunnels, the *coverage* for each logical chain increases substantially. A coverage of 0 implies that there was no feasible solution. For some configurations, we see that we find feasible solutions only with SwitchTunnels (e.g., AS1221 and AS3356). In addition, we observe a gain of up to 3× with SwitchTunnels. This confirms the value of SwitchTunnels to better utilize the available switch capacity and to provide more degrees of freedom for load balancing.

Scalability of pruning: While pruning does involve solving a large ILP, using `CPLEX` it only takes ≈800 s and ≈1800 s to compute the pruned set for the two largest topologies AS3356 and

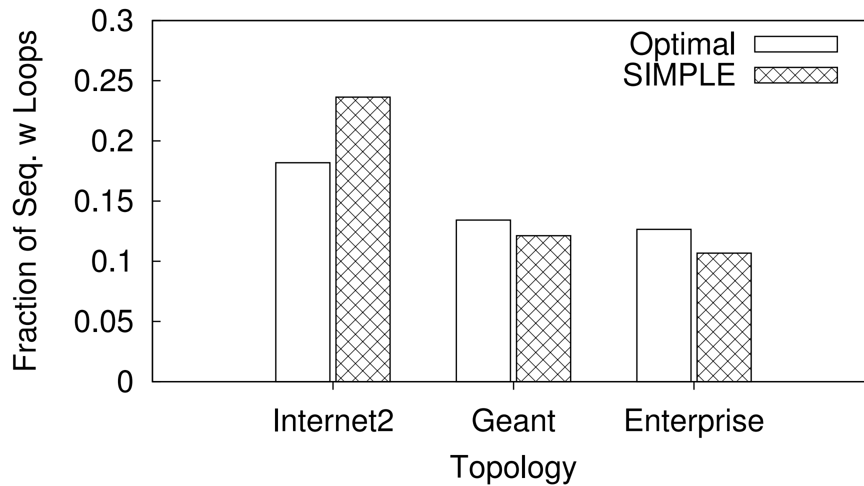


Figure 3.12: Fraction of sequences with loops.

AS3356-aug respectively. Since this is an offline (and infrequent) step, this overhead is quite acceptable. As we discussed, we reduce this by bootstrapping the solver to use solutions from previous iterations. Using this optimization reduces the pruning time substantially from 1800 s to 110 s for AS3356-aug.

3.9.3 Accuracy of the DynHandler

Proxies create the most number of challenges in terms of dynamic behaviors—they create/multiplex sessions and change packet contents. Thus, we focus on the accuracy of the DynHandler in inferring correlations between responses from the web servers to a Squid proxy and from the Squid instance to the individual users. To make the evaluation concrete, we consider two types of policies: *user-specific policies* (i.e., identify the specific user responsible for an incoming connection); and *stateful policies* (i.e., check if there is some user who initiated the traffic).

We introduce two error metrics: (1) *Missed policy rate:* The fraction of Internet→Squid sessions that we should apply a policy but we do not. In the stateful policy, it means that the session is initiated by a user but we cannot find any user to match the session. The user-specific policy is more complex because the proxy can multiplex sessions and thus an Internet→Squid session can

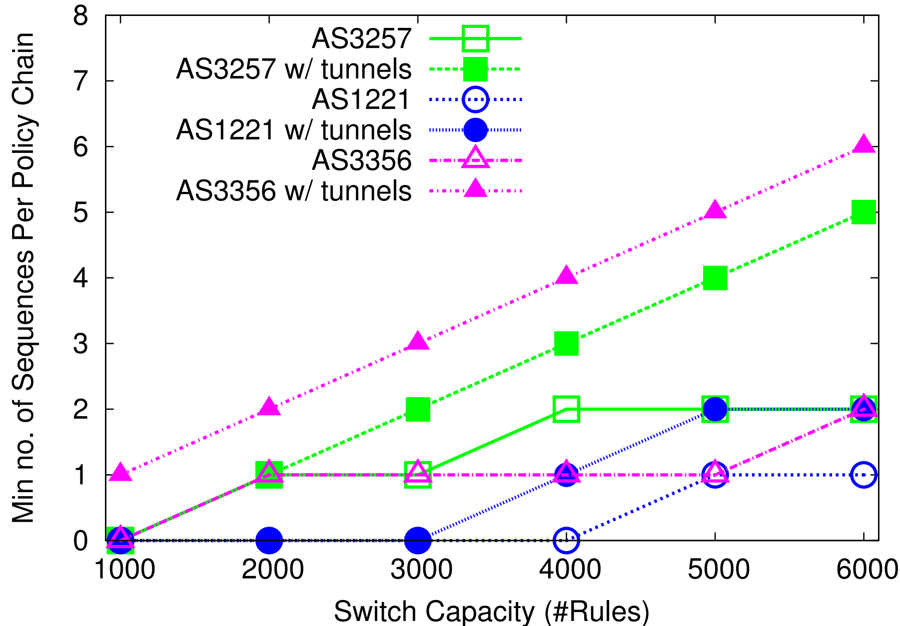


Figure 3.13: Coverage vs. available switch capacity for selected topologies. We use 3 policy chains per ingress-egress pair.

map to multiple users. Therefore, we define it as missed policy when we fail to find all the users that match a session. (2) *False policy rate:* The fraction of Internet→Squid sessions that we should not apply a policy but we incorrectly do. In the user-specific policy, it means that we identify the wrong users that match a session (although we may identify some right users at the same time).

We consider 20 simultaneous user web browsing sessions to access popular top 100 US websites [39]. To accurately emulate web page effects (e.g., Javascript, multiple connections etc), we use Chrome configured with the Squid as an explicit proxy. In our experiment, we observe and collect 394 sessions from Internet→Squid and 1328 sessions Squid→Users.

Obtaining the ground truth of mappings is itself a challenging problem given the complexity of Squid actions. This becomes especially hard as many websites use third-party content (e.g., analytics javascripts or Facebook widgets). As a heuristic approximation, we instrument each browser instance with unique (but fake) UserAgent strings to allow us to correlate the sessions. Unfortunately, even this turns out to be insufficient because Squid may request the website for multiple users and may prefetch a website and cache the content to serve future users. As such, we view the

Topology	#Switches	Time(s)			
		Opt	Opt w/ tunnel	SIMPLE	SIMPLE w/ tunnel
Internet2	11	0.3	0.3	0.01	0.01
Geant	22	2.29	1.99	0.09	0.14
Enterprise	23	1.76	2.46	0.01	0.01
AS1221	44	23394	91.7	0.04	0.29
AS1239	52	722.7	218.1	0.06	0.2
AS3356	63	122246	3239	0.22	0.48
AS3356-aug	252	-	-	0.92	1.22

Table 3.4: Time to generate load balanced configurations subject to switch constraints.

error rates we report as conservative upperbounds on the true error rates of the DynHandler since our ground truth is itself incomplete.

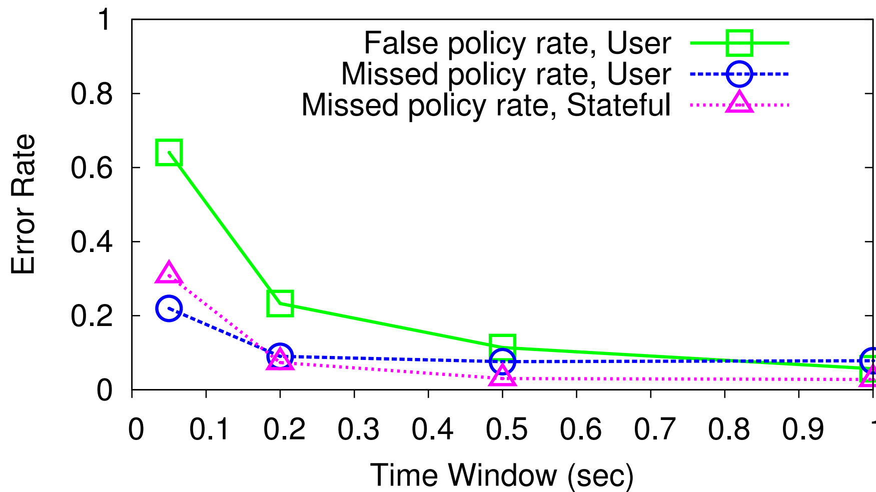


Figure 3.14: Accuracy of the SIMPLE DynHandler for two types of proxy-specific policies.

Figure 3.14 shows the error metrics for user-specific and stateful policies as a function of the correlation window and using the first 5 packets. (The false policy rates for stateful policies are zero and thus we do not show it.) We see that for the user-specific policy, at 500 ms the false policy rate is 11.4% and the missed policy rate is 7.6%. If we only need to realize the stateful policy, then we can use a smaller time window (e.g., W=200 ms) to achieve similar error rate. In both

cases, the bandwidth overhead from the switch to the controller is small; with a window of 500 ms the overhead is 65KB on average (not shown). The processing overhead at the controller is also relatively small, taking 150 ms for 1000 correlations.

3.10 Summary

Middleboxes represent, at the same time, an opportunity, a necessity, and a challenge for SDN. They are an opportunity for SDN to demonstrate a practical use-case for L4–L7 functions that the market views as important; they are a necessity given the industry concerns surrounding the ability of SDN to integrate with existing network infrastructure; and they are a challenge as they introduce aspects that fall outside the scope of traditional L2/L3 functions that motivated SDN.

This work was driven by the goal of realizing the benefits of SDN-style control for middlebox-specific traffic steering without mandating any placement or implementation constraints on middleboxes and without changing current SDN standards. To this end, we address key system design and algorithmic challenges that stem from the new requirements that middleboxes imposed—efficient data plane support for composition, unified switch and middlebox resource management, and automatically dealing with dynamic packet modifications. While our goal is admittedly more modest compared to ongoing and parallel work developing new visions for SDN or middleboxes, it is arguably more timely, practical, and immediately deployable.

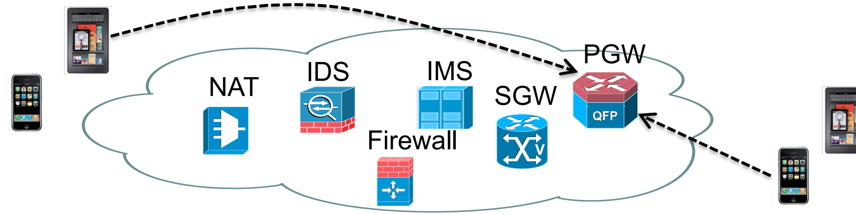
Chapter 4

A Framework to Evaluate the NFV Design Space

Networks are today populated with a large and increasing variety of proprietary hardware appliances. These include middlebox applications such as firewalls, intrusion detection systems, application gateways, transcoders. Networks today incur high capital costs in deploying a broad range of these expensive and inflexible hardware appliances. In addition, both volume and diversity of traffic is increasing sharply. Networks also need to host a variety of new in-network services to satisfy customer demands. This has motivated the case for *network functions virtualization* (NFV) [25]. The motivation in NFV is accelerating the pace of innovation by reducing the cycles to deploy new equipment, economies of scale provided by commodity hardware, resource multiplexing via virtualization, dynamic provisioning and elastic scaling, and the ability to experiment with new services without significant upfront costs [25](see also Figure 4.1). The high-level idea is that the various *network functions* (NFs) can be replaced by virtualized or software applications on commodity hardware platforms.

As highlighted in an early whitepaper [25], this vision builds upon, and is complementary, to existing work in the software-defined networking (SDN). The main differences are that: (a) it broadens the scope of the data plane functions beyond OpenFlow-enabled switches; and (b) it focuses more on the carriers and the services they would like to offer to their customers.

Today: Fixed function, proprietary hardware, at "chokepoints"



NFV: Commodity hardware, virtualized, flexible

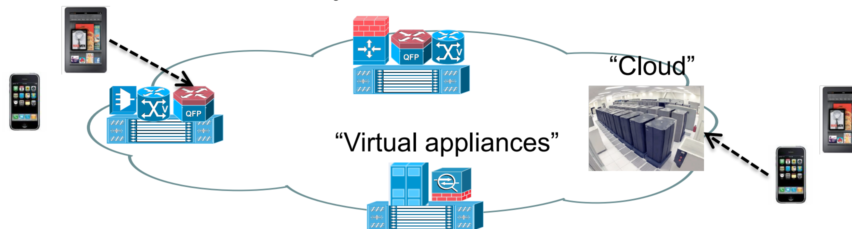


Figure 4.1: The current fixed and proprietary implementation of network functions *vs* the NFV vision of elastic, cost effective, mix-match and potentially hybrid deployment of network functions.

While the above long-term vision is appealing, it also leaves open several aspects of the design space that operators would need to address in practice w.r.t.:

- the type of *platforms* (e.g., pure cloud, or fixed-but-flexible hardware infrastructure);
- the type of *provisioning* (i.e., where to place new flexible hardware or datacenters); and
- *demand distribution* (i.e., how to route user demands to different function instances to meet logical policy requirements).

Given this broad design space, there are several possible NFV instantiations by combining choices across the individual platform, provisioning, and distribution dimensions. For instance, we can imagine a consolidated deployment where the network provider has a small number of datacenters at which the network functions (NFs) can be run in a dynamic, elastic manner. At the other extreme, we can imagine current deployments with specialized NF hardware. We can also envision "nano-datacenter" like models with flexible NF hardware distributed throughout the network [133]. These deployments will naturally have different provisioning, operational, and performance characteristics.

Given this diverse and broad design space, network operators will need systematic decision systems to help them evaluate the cost-benefit tradeoffs of different points in the design space. We highlight some motivating scenarios in §4.1. We use cellular network as illustrative examples. Furthermore, even before embarking on rearchitecting their network infrastructure [8], operators need to first *quantify* the potential CAPEX and OPEX benefits that specific NFV strategies might offer.

In this work attempt to shed light on these issues. To this end, we cast the NFV deployment problem as a systematic optimization framework (§4.3). Our framework is general enough to capture different points in the design space and also consider hybrid NFV different deployments (e.g., some combination of pure cloud and fixed hardware). In order to estimate the potential benefits of different NFV strategies, the operators provide as input historical traffic demands and policy based service chaining requirements for different traffic patterns (§4.2). Our framework will then output guidelines on the optimal provisioning strategy and the cost benefits it offers. Operators can use such a framework for "what-if" analysis to evaluate the cost-benefit tradeoffs of different deployment strategies.

We use cellular networks as a use case study to illustrate some of our findings. As illustrative examples we show how operators can use our framework to evaluate the benefits of different NFV designs (§4.4). For instance, we observe that using flexible hardware minimizes the deployment cost in many scenarios. We also conduct a sensitivity analysis to evaluate the effects of changing different input parameters on the optimal deployment strategy.

4.1 Motivation

We begin by outlining the NFV design space and then use motivating scenarios to highlight how the optimal NFV strategy depends on the workload and cost factors.

4.1.1 Design Space of NFV

We identify three key dimensions for the design space:

- *Platform type:* Network functions (NFs) can be realized in many ways. Today, each NF is a dedicated appliance (*Single*) providing a specialized capability. Going forward, one can imagine

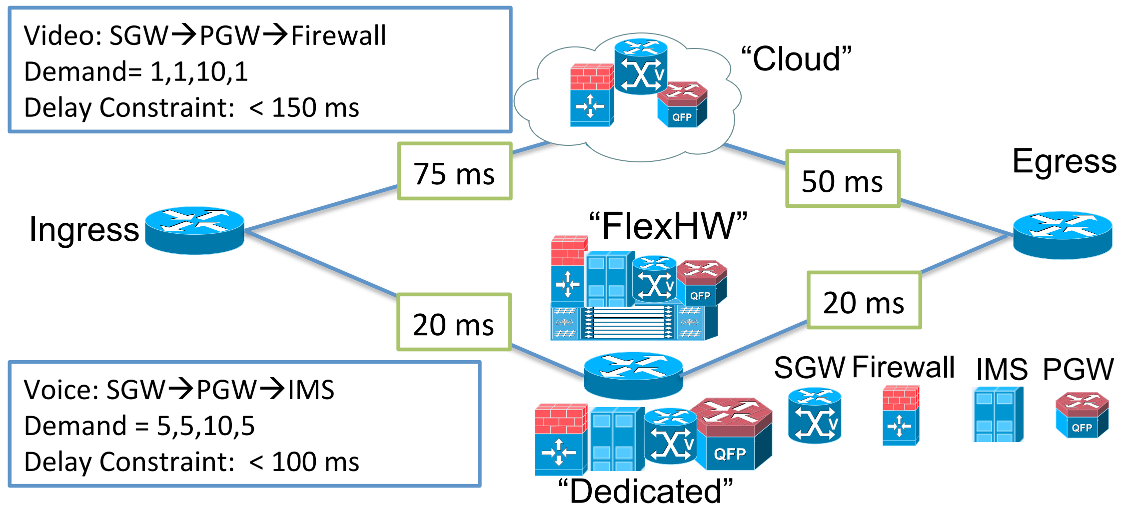


Figure 4.2: Example to motivate the different design tradeoffs in provisioning.

a flexible commodity hardware (*FlexHW*) that can be repurposed to run different types of NFs on demand [101, 124, 50]. Going one step further, we can imagine that the functions are themselves outsourced to a *Cloud* service that can elastically scale resources for different NFs.[1]

- *Provisioning and placement:* A key operational decision is deciding how and where to provision NF platforms. At one extreme, the provider can choose a single *Cloud* location. At the other extreme we can envision a nano-datacenter model where every cell base station has an associated mini-*Cloud* (e.g., [133]). We can also consider simple hybrids where each location has pre-provisioned *Single* and *FlexHW* boxes and we have a few *Cloud* locations.

- *Scaling and distribution strategies:* Given a specific provisioning/placement strategy, another aspect of the design space is how the available hardware resources (possibly elastic) are used to serve the (varying) offered load. Again, we can envision several possible strategies here: optimal load balancing, or routing to nearest available instance of a specific NF, or offloading to the cloud beyond a threshold value of load.

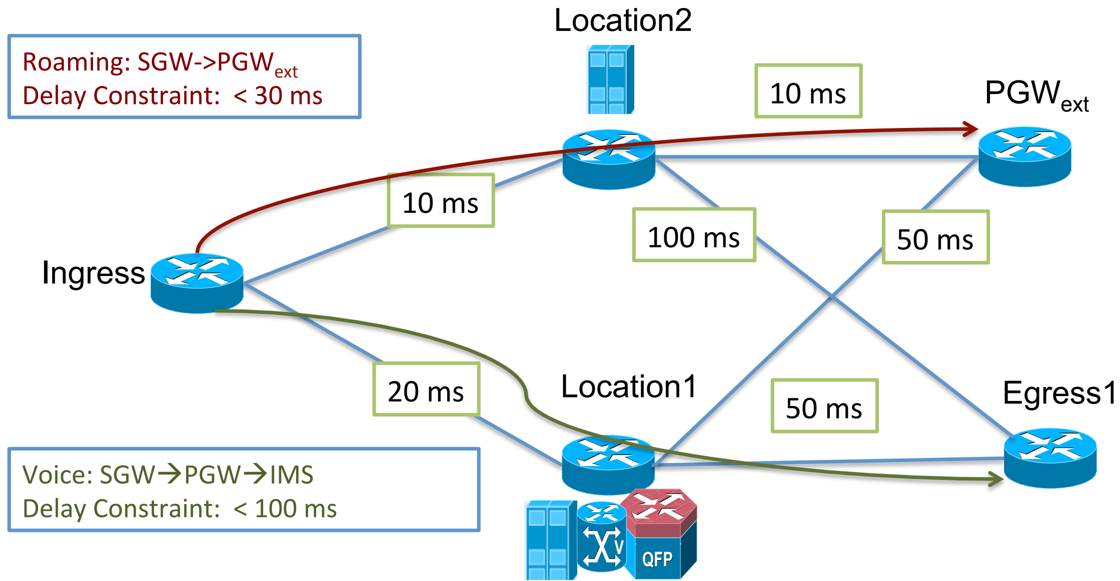


Figure 4.3: Example to illustrate the different design tradeoffs in functional placement and routing.

4.1.2 Motivating Scenarios

We use the simple scenarios in Figure 4.2 and Figure 4.3 to illustrate how different points in the NFV design space may be optimal depending on the performance constraints, traffic patterns, policy chains and CAPEX and OPEX.

Optimizing provisioning cost: Consider the single traffic class, *Video*, with traffic entering at "Ingress" and exiting at "Egress" with the *service chain* SGW-PGW-FIREWALL. Suppose the traffic volume for this traffic class across four time epochs is $\{1,1,10,1\}$. Assume that the (fixed) cost per unit capacity for fixed in-network provisioning is 20. On the other hand, if we use *Cloud* to dynamically provision the needed resource, assume the amortized cost per unit capacity provisioned is 10. With fixed pre-provisioning we need to allocate for peak traffic and thus a capacity of 10 units costing 200. But when outsourcing the processing to the cloud, the cost is only 130. Thus, we decide to provision the entire service chain for *Video* in the *Cloud*.

Performance constraint: Now, assume that we add another traffic class, *Voice*, with the *service*

[1]Industry reports use the terms *FlexHW* and *Cloud* rather loosely. They are however quite distinct in their cost-performance tradeoffs and thus one of our goals in formalizing the NFV problem space is to crystallize these loose characterizations of NFV instantiations.

chain SGW-PGW-IMS. Assume that *Voice* has a performance constraint that the average latency should be less than 100 ms. Then, we cannot use the *Cloud* for processing traffic belonging to this class. Assume also that the traffic volume for *Voice* across the four time epochs is $\{5,5,10,5\}$. Now, a hybrid solution where we provision a capacity of 10 units on fixed hardware and 10 units on the *Cloud* dynamically only in epoch 3 will be the most cost effective. This solution has cost 300, while provisioning entirely on the *Cloud* costs 380 (if there are no performance constraints) and provisioning entirely on a fixed hardware costs 400.

Functional placement and routing: To illustrate placement/routing issues consider the network in Figure 4.3. We have two traffic classes *Roaming* and *Voice*. The traffic belonging to *Roaming* requires processing SGW-PGW_{ext}, where PGW_{ext} is the PGW of the external network to which the traffic belongs. On the other hand, *Voice* needs processing using the service chain SGW-PGW-IMS. Assume that the carrier normally prefers to provision all resources in "Location 1" due to cost or management issues. However, when the *Roaming* traffic enters the network, it must move the related SGW processing to "Location 2" as going via "Location 1" violates the delay constraint for *Roaming*. Assume also after a network upgrade the delay on the Location 1-PGW_{ext} link falls to 5 ms. Then, the SGW processing can move back to Location 1. Moving the SGW processing back and forth depending on traffic and network changes could be facilitated via use of *FlexHW*.

The above scenarios highlights the tradeoffs and considerations operators need to make. It is important for operators to be able to analyze these tradeoffs before rolling out new NFV deployments. However, given the size and complexity of modern cellular deployments, it may be impractical for operators to manually evaluate the entire space of design options. Thus, our goal is to develop a decision support framework that allows cellular operators to systematically explore different design options before deploying new hardware. For instance, they would like to specify policy requirements (e.g., service chains for different user classes) and performance constraints (e.g., load, congestion, or latency) and use such a decision support system to choose the right mix of platform, provisioning, and distribution strategies from the broader NFV design space highlighted above.

4.2 Inputs and Requirements

We begin by describing the requirements and inputs that operators need to provide to our framework. This data can be obtained from their network logs, policy configurations, and vendor-specific benchmarks.

Traffic patterns: Cellular traffic is divided into different logical *classes* based on different user/customer demands. For example, the classes may capture *regular users* vs. *roaming customers* vs. *machine-to-machine* (M2M) traffic, with different requirements. We assume that the operator has historical demand patterns, with $|T_{c,e}|$ representing the volume of traffic for class c observed in epoch e.

Processing requirements: Each class c is associated with a *policy service chain* or a sequence of *network functions* (NF) that process traffic in c. Let $SC_c = NF_1 \prec NF_2 \prec \ldots$ denote the service chain for class c. These NFs could span cellular-, IP-, and application-level processing. Let $FP_{c,m,t}$ denote the processing cost (e.g., CPU usage) per-packet in class c for running a NF NF_m on a specific *type* of NF platform t. For example, the per-packet CPU usage may differ across virtualized vs. non-virtualized deployments.

Performance constraints: Performance constraints specify that traffic in class c should have some pre-specified performance PC_c [18]. As a simple starting point, we consider the end-to-end latency for each class.

Cost factors: The provisioning costs associated with rolling out the NF platforms may depend on the platform t (i.e., *FlexHW* or *Cloud*) and the location l (e.g., power, cooling costs). We capture these costs as follows. First, we assume that there is a *fixed* cost of deploying an instance of type t at location l, $Fixed_{l,t}$; e.g., this captures administrative and labor costs in rolling out new deployments and the operational costs. For deployments like *Cloud* this may be zero as they may use a pay-as-you-go model. Second, there is a *hardware* cost, $Var_{l,t}$, depending on the amount of resource provisioned for this instance; e.g., based on the number of CPU cores or memory on the hardware. Third, we have a *elastic* factor depending on the actual resources used, $Elas_{l,t}$; e.g., this can be linear in the amount of resources used for *Cloud* and zero for the others.

4.3 Provisioning Model

In this section, we describe a formal optimization framework that captures the cost of provisioning the network to meet the time-varying processing and performance requirements, given the policy constraints, platform costs, traffic demands, and network specifications. Network operators can use this framework to (a) systematically quantify the potential benefits of NFV in their deployments and (b) estimate the relative benefits offered by different points in the NFV design space.

4.3.1 Control Variables

As a starting point, we present a model where the operator is considering a set of possible platform instances p to deploy. As discussed in the previous section, each p can be instantiated in many ways (e.g., *Single* vs. *Cloud*) with varying cost-performance characteristics. We use $t(p)$ and $l(p)$ to denote the type (e.g., *Cloud* or *Single*) and location of a specific NF platform instance p. Now, there are two main types of control variables that we need to capture:

- *Provisioning:* The first decision is a binary decision if we want to deploy an instance at a specific location. This is captured by a $\{0{,}1\}$ variable $Active_p$. If we choose to deploy, then we also need to decide how much hardware resource to provision, captured by the variable Res_p. For some platform types, we also have an elastic option (e.g., *FlexHW* or *Cloud*), in this case we also use dynamic provisioning decision variables, $Res_{p,e}$.
- *Load distribution:* Given a provisioning strategy, we need to meet the processing requirements and distribute the load across the various p instances. In general, each class may have different chains and the required NFs can be instantiated at *any* set of instances capable of running these NFs. We can also flexibly route traffic to balance the network and platform loads [116]. To capture these considerations, we introduce *flow variables*, $f_{c,e,p_m,p'_{m'}}$, that represents the fraction of traffic in class c in epoch e, routed from a NF instance p_m to another NF instance $p'_{m'}$ (similar to Figure 4.4). Note that we can flexibly capture different routing strategies by scoping these flow distribution variables differently. (Some of these variables will not appear if m or n do not appear in SC_c.)

Objective function: Our objective is to minimize the total provisioning cost. This has three components: (1) fixed costs of instantiating platforms; (2) hardware costs for each "active" platform; and (3) the dynamically provisioned compute resources per epoch. Eq (4.1) shows this total cost in terms of the $Active_p$, Res_p, and $Res_{p,e}$ control variables:

$$\sum_{p} Fixed_{l(p),t(p)} \times Active_p + Var_{l(p),t(p)} \times Res_p + \sum_{p,e} Elas_{l(p),t(p)} \times Res_{p,e} \tag{4.1}$$

As discussed earlier, these factors depend on the type of platform; e.g., *Cloud* may have $Fixed_{l(p),t(p)} = Var_{l(p),t(p)} = 0$, but have $Elas_{l(p),t(p)} > 0$, while other models have $Elas_{l(p),t(p)} = 0$.

4.3.2 Formulation

Next, we describe how we capture the various processing and provisioning constraints.

Resources provisioned: First, we need to capture the amount of resources provisioned. We begin by capturing the total compute load on a NF instance p_m on the platform p during epoch e in Eq (4.2):

$$\forall e, p, m : LoadPerNF_{p_m,e} = \sum_{c} \sum_{\substack{m' \text{ s.t } m' \in SC_c \\ \text{and } m' \in p'}} FP_{c,m,t} \times f_{c,e,p_m,p'_{m'}} \times |T_{c,e}| \tag{4.2}$$

Then, in Eq (4.3), the total load on a platform p in an epoch e is simply the sum over all NF functions that p can support:

$$\forall e, p : Load_{p,e} = \sum_{m \in p} LoadPerNF_{p_m,e} \tag{4.3}$$

Now, our provisioning strategy must ensure that each NF platform has sufficient resources to cover the processing requirements per epoch and across all epochs. Thus, we have Eq (4.4) and

Eq (4.5):

$$\forall e, p : Load_{p,e} = Res_{p,e} \tag{4.4}$$

$$\forall e, p : Load_{p,e} \leq Res_p \tag{4.5}$$

In addition, and depending on other capacity constraints (e.g., space, power, or available hardware configurations), we may also have upper bounds on the total resources per-platform at each location:

$$\forall p : Res_p \leq Cap_p \tag{4.6}$$

Fixed costs: Next, we need to model the fixed costs associated with the above provisioning strategy. These fixed costs are incurred if the platform is being used in *at least* one of the epochs with non-zero resources. Thus, we have the following relationship between the binary $Active_p$ variables and the Res_p variables:

$$\forall p : Res_p \leq Cap_p \times Active_p \tag{4.7}$$

Modeling traffic distribution: The above equations model the provisioning aspects, but do not capture how the traffic processing is *distributed* across the platforms. In other words, we need to model how the $f_{c,e,p_m,p'_{m'}}$ variables are quantified. There are two key things we need to capture here. First, we need to model the *coverage* constraint that for each c, the desired service chain SC_c has been assigned to some set of platform instances. Second, we also need to ensure that the service chain is correctly applied in the intended *sequence*. Let $SC_c[j]$ denote the jth NF in the chain SC_c. Note that each $SC_c[j]$ may be realized using several candidate platform instances.

We model this using two sets of constraints. First, in Eq (4.8) we ensure that the entire fraction of traffic is routed to the first hops.

$$\forall c, e : \sum_{p_m : m = SC_c[1]} f_{c,e,p_m} = 1 \tag{4.8}$$

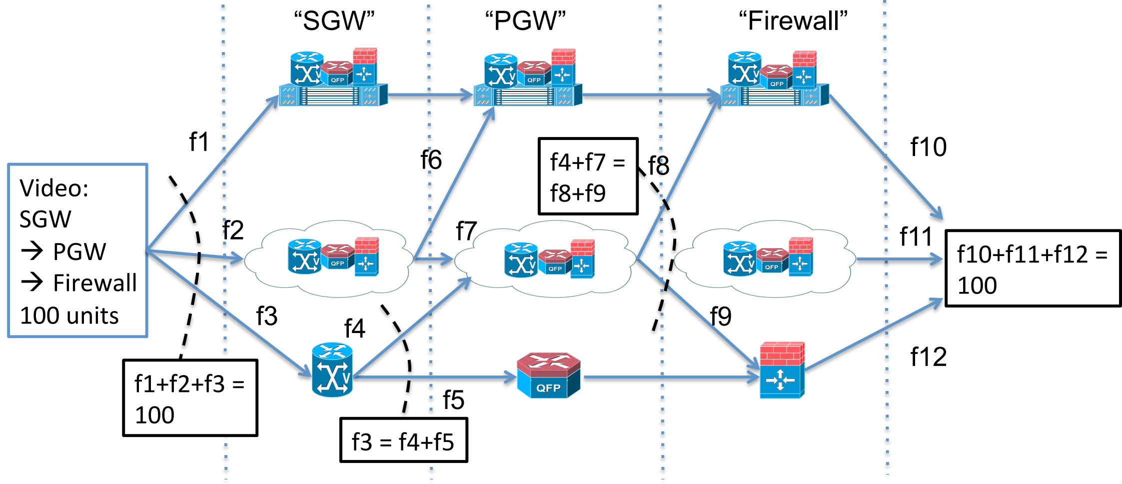


Figure 4.4: Example to illustrate how flow conservation is modeled.

Second, we model *flow conservation* constraints that ensures that the traffic incoming into one "stage" in the service chain is routed to the next "stage" in the desired sequence(e.g, see Figure 4.4).[2] Then, we have:

$$\forall p_m, c, e \text{ s.t } m = SC_c[j] \;\&\; j > 1: \\ \sum_{p'_{m'}: m' = SC_c[j-1]} f_{c,e,p_m,p'_{m'}} = \sum_{p'_{m'} = SC_c[j+1]} f_{c,e,p_m,p'_{m'}} \tag{4.9}$$

Performance bounds: Now, given the flow distribution variables $f_{c,e,p_m,p'_{m'}}$, we can also model the network-level performance that traffic for each class perceives. As a simple starting point, we model the *average latency* and ensure that this is less the given threshold PC_c. If $Lat_{p_m,p'_{m'}}$ is the typical network latency on the path from p_m to $p'_{m'}$, then we can capture the performance bound as shown below:

$$\forall c, e: \sum_{j=1}^{|SC_c|-1} \sum_{\substack{p_m, p'_{m'} \text{ s.t} \\ m = SC_c[j] \\ m' = SC_c[j+1]}} f_{c,e,p_m,p'_{m'}} \times Lat_{p_m,p'_{m'}} \leq PC_c \tag{4.10}$$

[2]In this work, we are not we are not mandating a specific data plane implementation. We could use wildcard rules [145] or tunnels [116].

4.4 Example Use Cases

Next, we highlight some illustrative use cases to validate how operators can use our framework to evaluate NFV design tradeoffs.

Setup: Due to lack of publicly available information on cellular network topologies, we use the PoP-level Internet2/Abilene topology.[3] We currently use four traffic classes, each with three different NFs. We assume there are a total of 8 different NFs [18]. We use a gravity model based on city populations as a baseline traffic demand and simple randomized variability models. We model the different provisioning problems as an integer linear program (ILP) and use CPLEX to solve the problem.

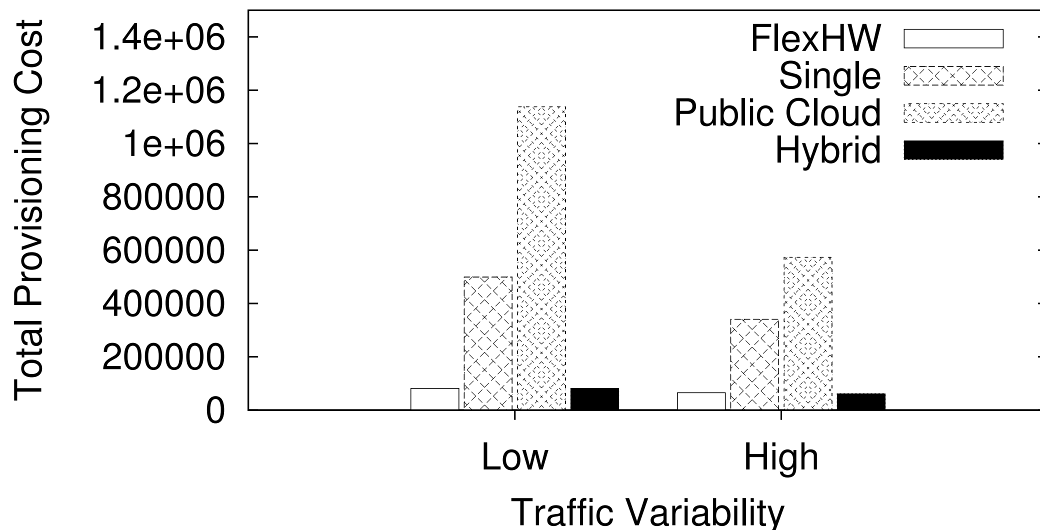


Figure 4.5: Total provisioning cost for different NFV models.

We instantiate the different platform types as follows:

- *FlexHW* assumes a commodity server is being used with standard virtualization technologies to run different functions on a single platform;

[3]We have also experimented with other ISP topologies from RocketFuel.

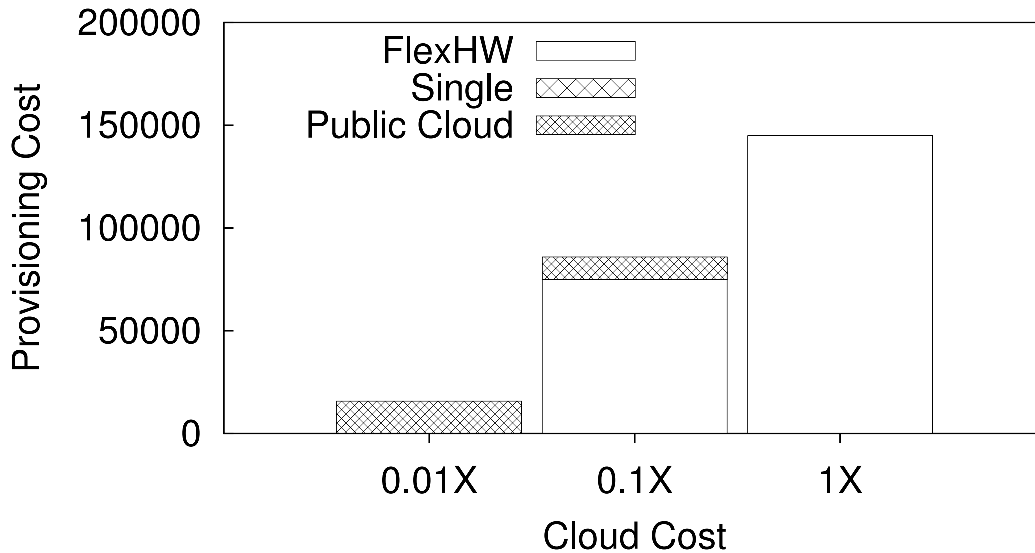


Figure 4.6: Impact on total provisioning cost wit varying cloud cost.

- *Single* assumes specialized hardware for running single functions;
- *Public Cloud* outsource the processing to a public cloud provider; and
- A *Hybrid* deployment model which allows full flexibility to use any combination of the above three deployment models.

Again, given the absence of accurate cost numbers for NFV platforms, we obtain ballpark numbers for the different costs by the following strategy. Since the different platforms have fundamentally different cost/service models, we normalize the costs by computing the dollar cost in provisioning/running the platform for unit traffic; e.g., dollars-per-Mbps. First, for *Public Cloud*, we consider bandwidth costs in Amazon EC2[4], and we compute the normalized cost assuming a monthly transfer volume of 500 TB. Second, for the *FlexHW* hardware we assume a typical commodity server of price $2,500 as representative for *FlexHW*. Third, for the *Single* devices, we use numbers from published work and assume a specialized device at 20 Gbps capacity costs roughly $80,000 [110] as representative for *Single* devices. Finally, for the setup and operational cost, we

[4]The CPU, memory costs were much lower than bandwidth costs

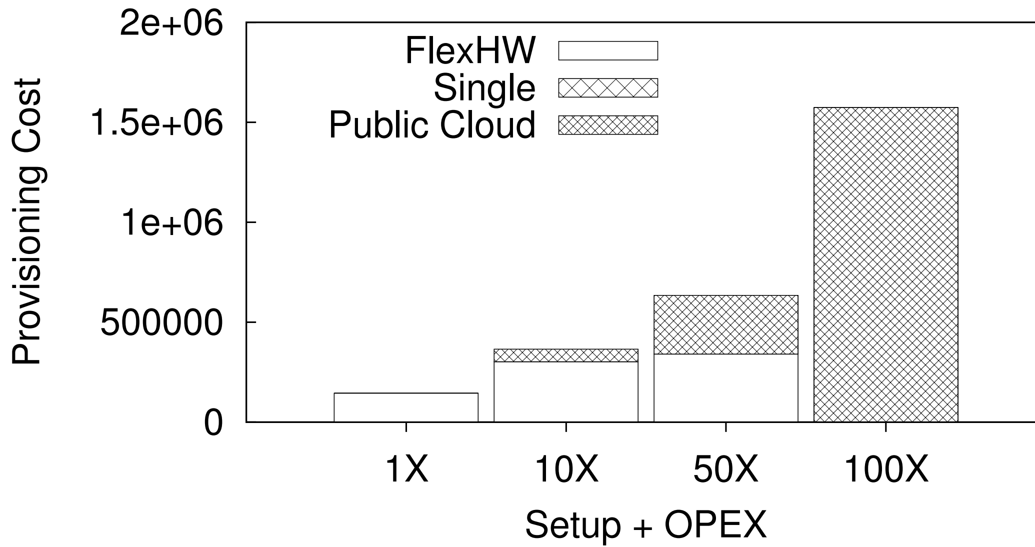


Figure 4.7: Impact on total provisioning cost with varying setup+OPEX cost.

use a common industry rule of thumb and model it to be twice the equipment cost [2].

Comparing different design options: To illustrate this point, Figure 4.5 shows that with the given costs a deployment strategy using only *FlexHW* has the same provisioning cost as the *Hybrid* when the traffic variability is low and very close to *Hybrid* when the variability is high.

Next, we show the impact of varying different input parameters like cloud cost, setup and operational cost, and performance of each platform type. We consider a *Hybrid* deployment model for these experiments with a random traffic matrix. Figure 4.6 shows that as the *Cloud* cost decreases, *Cloud* becomes a more viable option for processing of network functions, with a *Cloud* cost of *0.01X*, resulting in a *Cloud* only optimal strategy. Figure 4.7 shows that as the setup and operational cost increases, there is less incentive in pre-provisioning resources inside the network. Figure 4.8 shows that as the performance gap between virtual appliances and specialized hardware increases, it maybe cost effective to use a combination of these two types of platforms. Note, for Figure 4.8 we only consider a *Hybrid* model consisting of *FlexHW* and *Single* platforms.

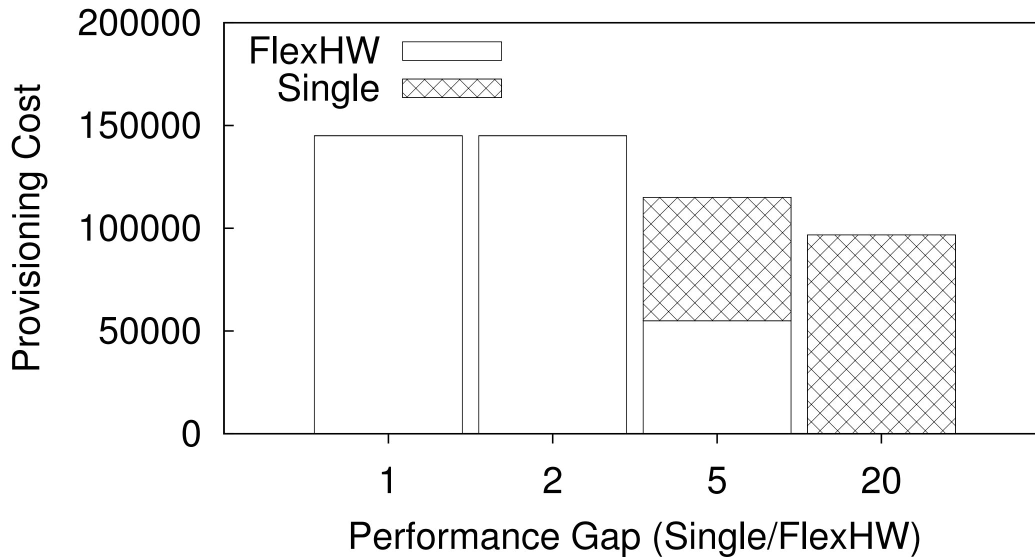


Figure 4.8: Impact on provisioning cost with varying resources.

4.5 Summary

This is a general framework that can be used by ISPs to make design decisions. It allows operators to capture a variety of design dimensions such as type of platform, load distribution, resource provisioning and placement. It also captures different processing and performance constraints that the network operators may have to catpture. In addition to such a framework, any design option will need real-time control capabilities to efficiently leverage the virtualization opportunities. To realize these benefits a *scalable* control plane will be needed to manage virtualized functions across all provisioned locations.[5] In particular, prior control plane architectures have required significant changes to the network; e.g., using SDN [116] or changes to middleboxes [72, 67, 124, 118], or have argued for clean-slate cellular architectures [90]. A natural open question in this context is whether we can design a practical and incrementally deployable control plane for cellular core that can achieve NFV benefits working within the constraints of existing cellular data plane (e.g.,

[5]This is different from the control plane of the cellular protocols (Section 2.4); this pertains to the control of the NFV functions themselves.

PMIP, GTP tunnels) and signaling (e.g., 3GPP) protocols, or if we need significant changes to today's cellular core architectures. We investigate this in the next chapter.

Chapter 5

KLEIN: A Minimally Disruptive Design for an Elastic Cellular Core

5.1 Motivation and Contributions

Over the last few years, we have observed explosive growth in mobile Internet-connected devices, spurred by the commoditization of smartphones, tablets, and other devices [45]. Reports suggest that mobile traffic volumes are poised to surpass traditional fixed-line Internet usage for many applications [23, 37, 32]. Furthermore, with the onset of Internet-of-Things deployments, analysts predict orders of magnitude more devices connected via cellular networks with diverse application demands.

This dramatic growth in volume and application diversity creates significant stresses on the *cellular core*—the operator's network between the radio access and the egress to the global Internet. The cellular core is a critical piece of the infrastructure which provides key cellular-specific data plane functions such as the Serving Gateway (S-GW) and Packet Data Network Gateway (P-GW) and various IP- and application-layer middlebox services (e.g., firewalls, proxies, and transcoders). Today, such functions are deployed using expensive and fixed function "big-iron" appliances [137]. These appliances are typically concentrated in a small number of datacenter sites in the operator's backbone and user traffic is routed to the nearest datacenter using standard cellular procedures;

e.g., using 3GPP standards [141].

Unfortunately, this current architecture results in fundamental sources of *inelasticity*, which in turn hurts costs, application performance, and evolvability [126]. (We elaborate in §5.3). For instance, the fixed capacity of the hardware forces operators to make provisioning decisions that lead to both significant underutilization, and an inability to handle unanticipated changes in the workload such as flash crowds, failures, and signaling storms (e.g., [46, 19]). This architecture also creates inefficient tradeoffs between provisioning cost and latency considerations; i.e., consolidation lowers cost via statistical multiplexing but inflates paths vs. disaggregation to reduce path lengths escalates costs as each site needs to be provisioned for peak loads.

Now, it is possible to address these sources of inelasticity using a clean-slate approach that fundamentally refactors how the cellular core is designed, provisioned, and managed. Indeed, several recent efforts (e.g., [90, 106]) have demonstrated the promise of such clean-slate architectures that argue for ubiquitous deployment of core functions and suggest that we need per-flow SDN-like mechanisms, using new "smart" switches at every base station.

The driving question behind our work is to ask if a clean-slate redesign is fundamentally necessary or if we can address these aforementioned limitations of cellular core networks in a *minimally disruptive* manner.

To address this question, we use data from a large cellular carrier to quantitatively evaluate three candidate cellular core designs: (1) TODAY's fixed hardware approach using 3GPP compliant routing; (2) A hypothetical INTERMEDIATE design that uses network functions virtualization (NFV), requires no changes to existing cellular signaling and core routing, and performs dynamic load distribution; and (3) A CLEANSLATE solution that uses NFV but is not constrained to be 3GPP compatible and can use fine-grained per-flow routing. Our analysis shows (perhaps surprisingly) that the INTERMEDIATE design achieves close-to-optimal provisioning tradeoffs and load balancing objectives relative to the CLEANSLATE approach.

We thus argue that this INTERMEDIATE design can serve as the basis for a *minimally disruptive* design for future cellular core architectures that can address today's cellular core limitations. In particular, NFV is already a reality for carriers [9, 40, 8, 126], and there are many open-source and

commercial efforts to virtualize cellular core functions [17, 29, 28, 6, 3, 4].

Building on these insights, we design KLEIN,[1] which provides a practical realization of this above INTERMEDIATE design. Specifically, KLEIN extends existing cellular core in two minimally disruptive ways: (1) use of virtualized EPC functions together with (standard) SDN mechanisms for service chaining inside the datacenters and (2) a global resource management scheme for mapping devices' traffic to different datacenter locations. KLEIN is 3GPP-compliant and requires no changes to existing cellular signaling and core routing.

This work addresses two key challenges to translate the hypothetical INTERMEDIATE design into reality. First, we design a responsive resource management layer that can handle billions of devices and thousands of data centers. Second, we engineer backwards-compatible network orchestration mechanisms to realize these dynamic resource management decisions.

We prototype KLEIN using the open source `OpenAirInterface` [28] platform. We use Floodlight and custom controllers for the KLEIN control plane to manage the core network. We validate KLEIN using a range of trace-driven and real testbed experiments. We find that: (a) KLEIN is scalable, it takes less then 20s to reconfigure the load with 2000 data centers and 5 billion devices; (b) KLEIN is close to optimal, within 10% of an ideal CLEANSLATE for different traffic mix and latency budgets; (c) KLEIN can improve end-application performance by a factor of 5; and (d) SIMPLE can handle data center failures both rapidly and efficiently, taking less than 2.3s, and reducing the maximum data center load by a factor of 2.

Contributions and Roadmap: In summary, this work makes the following contributions:

- An empirical demonstration of key limitations of today's cellular core with respect to cost-performance tradeoffs (§5.3).
- A data-driven design space exploration (§5.4) that shows that it is indeed possible to address these limitations with a minimally disruptive design.
- A practical architecture (§5.5), with a responsive and scalable resource management algo-

[1]The name is inspired by Yves Klein, a pioneering artist in the Minimal art movement, `https://en.wikipedia.org/wiki/Yves_Klein` The name also means "small" or little in German which is indicative of the change we mandate.

rithm that can handle billion of devices and thousand of sites (§5.6) and backwards compatible orchestration mechanisms (§5.7).

- A proof-of-concept implementation (§5.8), to show the benefits of elastically scaling and balancing load on virtualized EPC functions.

5.2 Related Work

SDN and NFV in Cellular Networks: Recent proposals on redesigning the cellular core using SDN and NFV argue for a clean-slate approach [90, 106], with new cellular signaling protocols and SDN-like routing.

SoftCell [90] proposes a clean-slate SDN based cellular core that can support fined-grained policies for mobile devices in cellular core networks. To achieve this goal they propose a new cellular core architecture, requiring smart access switches at each base station, SDN-based routing over the core network, while using commodity switches and middleboxes in the core network. The access switches perform fine-grained packet classification on traffic from UEs. SoftCell proposes a logically centralized SDN controller responsible for managing the routing over the core network. The main contribution is to propose a new data plane that can support a large number of policies given this new architecture. SoftCell achieves this by compressing the switch forwarding tables by aggregating them along different dimensions (e.g., service policy, base station, mobile device). While such a proposal provides flexibility in service policy enforcement, it requires overhaul of today's cellular core infrastructure and is not compatible with existing cellular standards.

SoftMoW [106] also proposes a new architecture for the cellular core. The motivation is to enable a highly distributed cellular core with distributed middleboxes and programmable switches. They propose a hierarchical and reconfigurable control plane design. They also propose a new cellular signaling control plane for the core network. They design a logically centralized controller which manages routing and mobility of devices across the cellular network. Similar to SoftCell, they require access switches at each base station that perform fine grained packet classification as well as a new programmable data plane. Similar to SoftCell, SoftMoW requires a major overhaul of today's cellular core infrastructure as well as existing cellular signaling protocols mandated by

3GPP.

In this work, our focus is to explore practical cellular core designs that are minimally disruptive, and can be deployed in the near future. In this respect, we designed KLEIN, a cellular core that does not require any changes to routing over the core network, uses existing cellular signaling protocols, NFV/SDN inside data centers and has a smart resource management layer. The core contribution of our work is to design a scalable and responsive resource management layer that uses backwards compatible network orchestration mechanisms.

Other proposals [51, 123, 112, 94, 59] consider virtualizing and decomposing core EPC functions like S-GW and P-GW. The work in [51] analyze the EPC nodes and classify their functions according to their impact on data-plane and control-plane processing. They investigate the current OpenFlow implementation's capability to realize basic core operations such as QoS, data classification, tunneling and charging. Their analysis shows that functions, which involve high data packet processing such as tunneling, have more potential to be kept on the data-plane network element. The work in [123] proposes decomposing the control and data-plane functionality inside EPC functions. In their design, the S-GW and P-GW data-plane functionality is implemented in OpenFlow switches, whereas the control plane functionality is move to a SDN controller. In [94], the authors present a study on the evolution of cloud-based EPC, where all the control functions of the SGW, PGW and MME are moved in the cloud. The data-plane is again shifted into the OpenFlow switches, and these switches are extended to support GTP. The authors of [112] propose the MobileFlow architecture for future carrier networks. In this architecture again, the EPC data and control plane is split such that the data-plane can be programmed, and the control plane is centralized. The eNodeB also participates in this functional split, and the entire EPC and eNodeB control plane is centralized. Such a proposal requires changes in the eNodeB, to transform the existing deployed eNodeBs in the entire network. The work in [59] argues for overhaul of today's cellular core network such that all the EPC functionality is moved in the cloud and the backhaul consists of only programmable SDN switches.

These proposals mainly focus on architecture and high level designs for the EPC. They do not focus on the practical mechanisms which will be needed to decide the placement and load

distribution of virtualized core functions over a highly distributed cellular core network. This is a challenging problem, and one of the main contributions of this work is a scalable and responsive resource manager.

There have been other problems in the mobile networks arena that have been addressed using SDN and NFV, such as heterogeneous wireless networks [49], SDN-based radio access network designs SDN [79, 78].

Middlebox Management: There has been prior work which has focused on "middlebox" service chaining and load balancing [116, 124, 148]. SIMPLE [116], discussed in the first half of this thesis addressed the problem of efficient policy enforcement using SDN. SIMPLE solves the problem of load balancing subject to switch and middlebox capacity constraints. SIMPLE assumes that the middlebox placements are known a priori. CoMb [124] proposes a new architecture for implementing and managing middleboxes. In contrast to standalone, specialized middleboxes, CoMb decouples the hardware and software, and enables software-based implementations of middlebox applications to run on a consolidated hardware platform. CoMb consolidates the management of different middlebox applications/devices into a single (logically) centralized controller that takes a unified, network-wide view generating configurations and accounting for policy requirements across all traffic, all applications, and all network locations. StEERING [148] also addresses the problem service chaining given a large number of subscriber and applications. It uses multiple tables inside a switch to implement a large number of service policies. In this work, we consider the problem of load distribution and network function placement over the entire core network consisting of potentially hundreds of sites and mobile devices, using backwards compatible mechanisms. This introduces unique challenges of scalability and practicality that we address in this work. Proposal such as SIMPLE can be used to implement middlebox-specific policies inside a data center. Other works [128, 73] provides mechanisms offloading traditional middlebox processing to the cloud.

Middlebox and EPC State Management: Other proposals have also suggested NFV-like ideas for traditional middleboxes [72, 118]. SplitMerge [118] characterizes the state of traditional middlebox such as load balancers, NATs etc. They propose mechanisms for transparently spliting state

between many replicas of middlebox function or merging it back into one, while ensuring the flows are routed to the correct middlebox replica. OpenNF [72] proposes a control plane architecture that provides efficient, coordinated control of both internal NF state and network forwarding state to allow quick, safe, and fine-grained reallocation of flows across NF instances in data center/enterprise network. In [99], the authors describe the state maintained by different network functions in LTE architecture, how this state is accessed, how different network functions could be refactored and modularized. In KLEIN, we primarily use legacy cellular signalling protocols to migrate state of EPC functions, and use OpenNF to migrate state of traditional middleboxes.

SDN Control Plane Design: There have been prior proposals for scalable control plane design in different contexts [106, 90, 100, 65, 63, 142, 81]. The authors of SoftMoW [106] propose a hierarchical and reconfigurable SDN control plane for managing the cellular core network. In SoftCell [90], the authors propose a centralized controller, with local controller agents installed at every access switch on each base station. The work in [100] also proposes a hierarchical SDN control plane for wide area networks. In Bohatei [65] the authors propose a control plane for managing DDoS attacks. The control plane consists of a global controller responsible for managing load across data centers and local controller within each data center, responsible for orchestration within a data center. ElastiCon [63] proposes mechanisms for distributing the data-plane load among different SDN controllers. Kandoo also addresses the problem of scalability in SDN controllers. In Kandoo there two layers of controllers: (i) the bottom layer is a group of controllers with no interconnection, and no knowledge of the network-wide state, and (ii) the top layer is a logically centralized controller that maintains the network-wide state. Controllers at the bottom layer run only local control applications (i.e., applications that can function using the state of a single switch) near datapaths. These controllers handle most of the frequent events and effectively shield the top layer. Beehive [142] provides a programming abstraction for writing distributed control applications in SDN. In KLEIN, our goal is to investigate how we can design a scalable and responsive resource management layer for handling thousands of sites and billion of mobile devices. We observe that simple two layer control plane designs do not scale. We design a three-level control-plane hierarchy, motivated by the workload characteristics and deployments of cellular net-

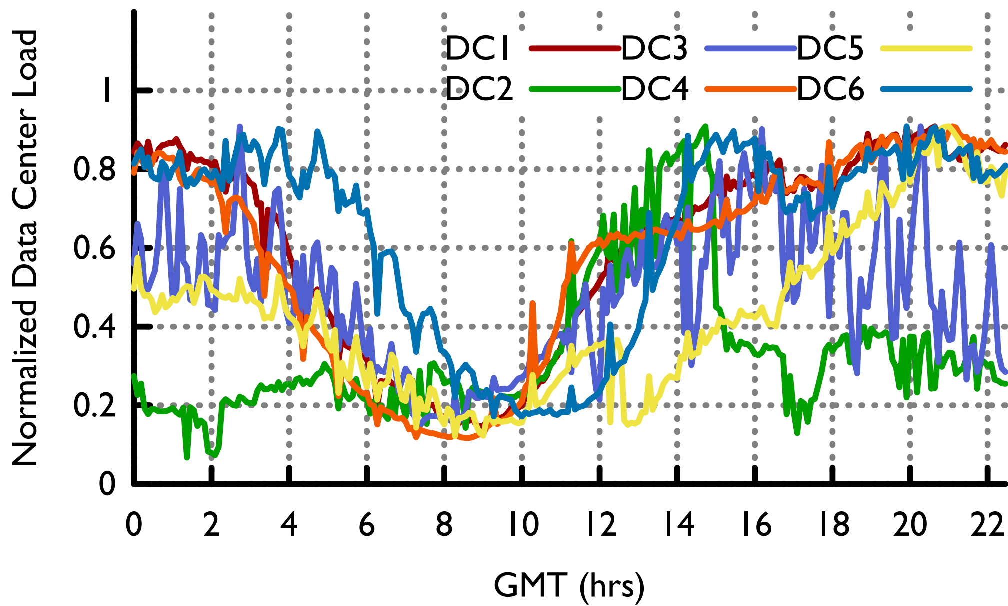


Figure 5.1: Load across different data centers over the course of a day.

works.

5.3 Limitations of Current Practises

In this section we analyze the practices of a nation-wide cellular carrier to manage it cellular core network, which hosts a large number of middleboxes. Specifically, we analyze the impact of the way network functions are provisioned and how traffic is routed to them. The motivation is to understand the limitations of current practises. We first briefly describe the dataset used for the analysis and then use it to highlight three key limitations with today's cellular core network.

5.3.1 Data Set

Dataset description: We use load traces collected for several months during 2014-2015 at tens of thousands of base stations of a large cellular provider in the US. The dataset gives a time series of data traffic volumes at 5 minute intervals at each base station, for each 'APN', and 'GW device'. APN refers to one or more collection of services (e.g., Voice, Data, M2M). GW device refers to the actual hardware appliance in the datacenter that runs an EPC data-path element (S-GW or P-GW) that processes the corresponding data. For each device, we have the information about the corre-

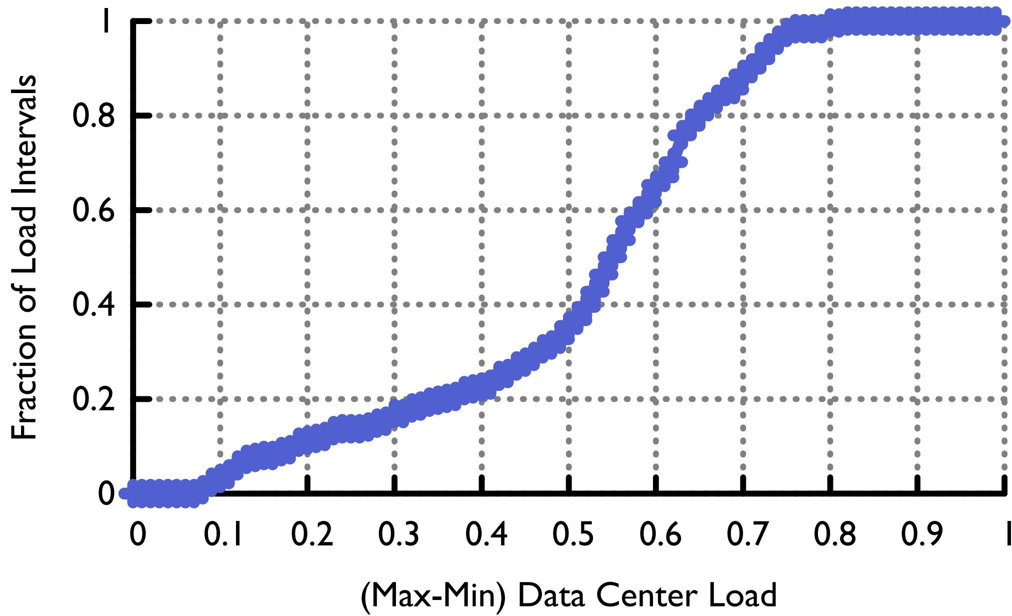


Figure 5.2: Load distribution across data centers for each time interval.

sponding data center the GW is located in. The data set covers tens of data centers and hundreds of APNs. No user data or any personal information that identifies individual users are collected. Next, we focus on specific set of analysis on this data set to highlight specific limitations.

5.3.2 Load Balancing

The fixed nature of provisioning and static routing in the cellular core causes load imbalance, which in turn could impact user-perceived application performance. Figure 5.1(a) shows the imbalance in S/P-GW loads across data centers for one entire day. Since the data centers capacities vary, the load of each data center is normalized by the peak load seen in that data center in the entire data set. This quantity serves as proxy for the capacity.

For any given time interval, we observe difference in the utilization of the most utilized data center and the least utilized data center. We observe this difference could be as high as 80%. Such load imbalance is not an anomaly. Figure 5.2(b) shows the distribution of the difference between maximum and minimum normalized data center loads. We observe that the difference is more than 50% for more than 60% of the times. Also, for about 15% of the time, load imbalance is more than 70%. This means that one data center is maximally loaded while another is a lot of spare

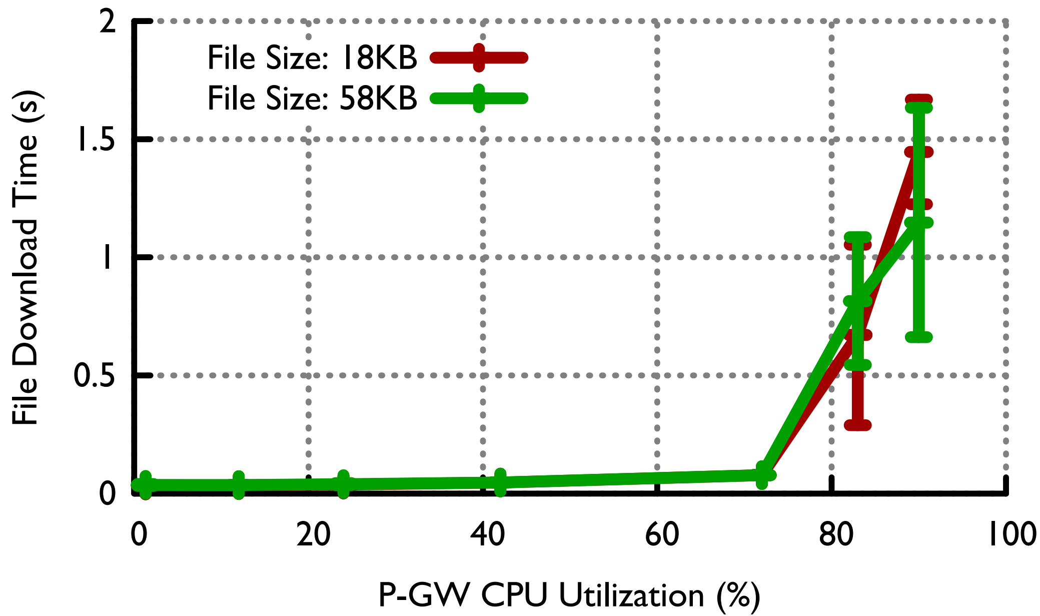


Figure 5.3: Impact of EPC load on file download time.

capacity. Load imbalance could potentially increase overload and failure scenarios [128], affecting the performance end-user applications. The ability to dynamically balance the load across data centers can both potentially improve application performance as well as provide resource savings for the network operators.

5.3.3 Impact on Applications

Load imbalance can potentially hurt application performance. To demonstrate this, we benchmarked the impact of EPC load on file download times on a software EPC testbed (Phantomnet [29]). Here, we increase the P-GW load by generating background traffic, and observe the impact on downloading files of two different sizes (16 KB and 58 KB) over a TCP connection. We observe in Figure 5.3(c) that when P-GW utilization is $\geq 80\%$, the file download time is more than an order of magnitude higher as compared to the case when P-GW utilization is $<$70%. High EPC load clearly hurts user-perceived performance. Extreme load imbalance demonstrated in Figure 5.1 means that such performance metrics would improve if flexible provisioning could be made available.

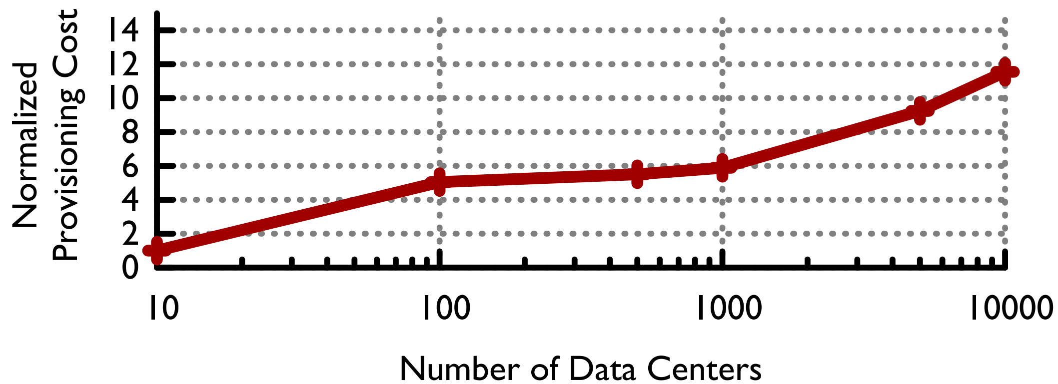


Figure 5.4: Provisioning cost vs. number of data centers with static provisioning and routing.

5.3.4 Resource Provisioning

At the same time, today's networks are over-provisioned. We compare *sum of peak* loads on individual S/P-GW devices with *peak of sum* loads on these devices. The ratio of sum of peak vs. peak of sum is a good indicator of resource over-provisioning. We observe that the sum of peak GW/DC load is about 1.7 times the corresponding peak aggregate load respectively. This means at most only 60% of provisioned resources are utilized at any time. As the network functions are statically pre-provisioned, the operators have to provision the network resources taking into account the peak load. They also over-provision the network to take into account future traffic growth and potential failure scenarios. However, a more flexible implementation of network functions (e.g., implementing them as virtual appliances) can provide the operators the ability to dynamically place and provision network functions. This can potentially provide resource savings, better application performance as well as ability to quickly incorporate diverse services inside the networks.

5.3.5 Provisioning Cost vs. Wider Deployment

Today's networks suffer from path inflation as all UE data traffic has to be routed to one of the few large centralized data centers [141]. Wider deployment of processing sites can improve latency,[2] but due to the fixed nature of mapping traffic to processing sites this can happen only at the expense of higher provisioning cost. Figure 5.4 demonstrates this. The plot here assumes various number of data centers. The locations of such data centers are assumed close to the eNodeBs they meant to serve. This is done by a nearest-neighbor clustering of eNodeBs in the 2D space and locating the data centers at the centroid of such clusters. We map the traffic to the nearest data center and compute the sum of peak loads on the data centers as the provisioning cost. The plot shows that in order to get an order of magnitude improvement in latency, for example, the cost will increase 5 fold (while from 10 to 500 data centers).

5.3.6 Summary

In summary, our data-driven analysis shows that:

- Load imbalance in cellular core network can be as high as 80%. This could impact application performance by as much as 7 fold.
- At most only 60% of the core network compute resources are utilized at any time.
- For wider cellular core deployments e.g., 500 data centers, provisioning cost is 5× higher than in the case of 10 data centers.

5.4 Design Space Exploration

The previous analysis shows the limitations in today's cellular core. These can potentially be addressed by a clean slate redesign - a generalization of recent work in [90, 106] - that distributes the processing resources widely and performs a fine grained (such as per-flow in the ideal case) dynamic mapping of traffic to such resources. While an approach like this could be optimal, this also requires a fundamental redesign of the cellular core. In this section we demonstrate – using a similar data-driven analysis as before – that such a disruptive redesign is unnecessary and an

[2] This can be done by expanding the use of small on path data centers e.g., central offices. (See Section 2.4).

$$\text{Minimize} \sum_{d} Prov_d, \text{ subject to} \quad (5.1)$$
$$\forall d, e : Load_{d,e} < Prov_d \quad (5.2)$$
$$\forall d, e : Load_{d,e} = \sum_{a,d,e,i} f_{a,d,i,e} \times T_{a,e,i} \times F_a \quad (5.3)$$
$$\forall a, d, e, i : f_{a,d,e,i} \in [0, 1] \quad (5.4)$$
$$\forall d : Prov_d < Cap_d \quad (5.5)$$
$$\forall a, e, i : T_{a,e,i} = \sum_{d,e} f_{a,d,e,i} \times T_{a,e,i} \quad (5.6)$$
$$\forall a, d, e, i : Latency_{d,i} \times f_{a,d,e,i} < Budget_a \quad (5.7)$$

Figure 5.5: Linear Program (LP) formulation for CLEANSLATE.

intermediate 3GPP compatible design can be used to address these limitations.

5.4.1 Design space

We consider a broad design space characterized by four dimensions: (1) *Implementation Platforms:* We could have each network function running on a fixed, hardware-based appliance or as a virtualized/software appliance; (2) *Routing granularity:* packets can be routed across the cellular core network e.g., per-flow (e.g., SDN-like) routing vs per-UE tunnels (conforming 3GPP); and (3) *Resource management:* How the traffic from different UEs and APNs are allocated to the available compute and network resources.

Given the dimensions of this design space, we consider three concrete instances.

- TODAY's network deployment, where (1) fixed, hardware based appliances are used to implement the EPC functions, (2) routing of flows is done at a per-UE granularity and (3) the resource management is static and each UE is routed to the nearest data center.

- At the opposite extreme, we consider a CLEANSLATE approach, where (1) network functions are virtualized, (2) fine-grained per-flow routing (which is in conflict with 3GPP constraints), and (3) dynamic resource management. This a logical extension of recent clean slate designs [90, 106]; the key addition is that we assume some form of optimal resource

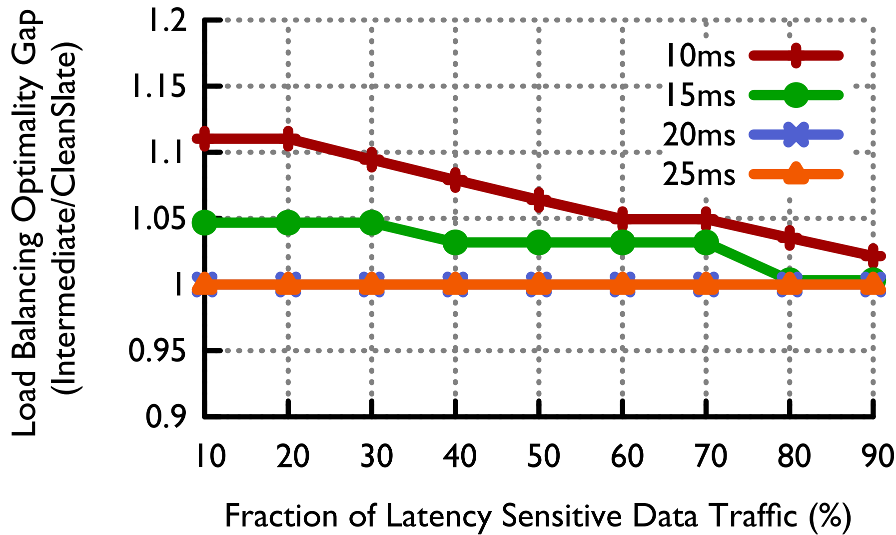


Figure 5.6: The load balancing optimality gap between INTERMEDIATE **and** CLEANSLATE**.**

management scheme which these prior efforts largely ignore.

- Finally, we consider an intermediate approach that we call INTERMEDIATE. INTERMEDIATE attempts to preserve the benefits of CLEANSLATE while also preserving full 3GPP compatibility. Here, (1) the network functions are still virtualized; but (2) routing decisions are at a per-UE granularity rather than the per-flow approach in CLEANSLATE; (3) dynamic resource management is used to re-balance the load as necessary.

5.4.2 Methodology

In order to compare the three design points, we conduct a data-driven study, using same data as in (Section 5.3). We devise linear programming based optimizations for modeling CLEANSLATE and INTERMEDIATE designs to evaluate the provisioning and load balancing benefits. Below we describe the simulation setup and optimizations.

Simulation Setup: We classify APNs into 3 different groups: Data, M2M, and Voice. For the first class, we assume that the traffic can be further divided into latency-sensitive and latency-tolerant applications. We vary the mix of latency sensitive data traffic and the delay budget for latency

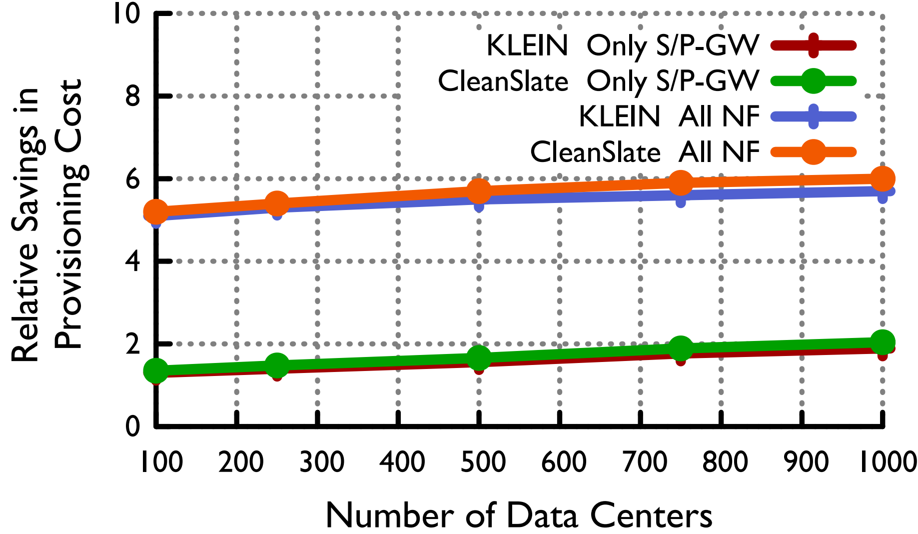


Figure 5.7: Reduction in provisioning cost with INTERMEDIATE **and** CLEANSLATE**.**

sensitive applications. The input to the simulations is the data traffic volume corresponding to different traffic classes from the data set.

Optimizations: In the CLEANSLATE design, we assume that the traffic can be split from each base station and application class, and routed to different data centers. We formulate CLEANSLATE as a linear program, with the objective of minimizing the total resources provisioned in the core network (Eq (1)). $Prov_d$ corresponds to the capacity to be provisioned in a data center d . The key decision variable in CLEANSLATE is a fractional variable $f_{a,d,e,i}$ which gives the fraction of traffic from base station i and application a, that should be routed to the data center d, in a time epoch, e. Figure 5.5 shows the LP. It includes the constraints that all the traffic should be served Eq (6), and that the latency budget for each traffic class should satisfied Eq (7). $Load_{d,e}$ is the load on a data center d in an epoch e, $T_{a,e,i}$ is the traffic volume from eNodeB i for traffic class a in epoch e, Cap_d is the maximum capacity that can be allocated to data center d, F_a is the total resource footprint of an application class a, $Latency_{d,i}$ is the latency from the eNodeB i to data center d, and $Budget_a$ is the latency budget for application class a.

For INTERMEDIATE, we map each base-station's load to a single data center. This is because of the 3GPP constraint that at any time we can only have a single SGW attached to a UE. We model

INTERMEDIATE as an ILP with same objective with the objective of minimizing the total resources provisioned in the core network (sum of resources provisioned in each data center. However, the key decision variable in this case is a binary variable $b_{a,d,e,i}$ which gives mapping of traffic from base station i, application a, that should be routed to the data center d, with the additional constraint that all the traffic from a base station i is routed to the same data center.

We also consider a load balancing exercise, where we minimize the maximum data center utilization for CLEANSLATE and INTERMEDIATE. The formulations are similar, except (1) the objective is minimizing $MaxDCUtil$, where $MaxDCUtil$ is the utilization of the most utilized data center in the core network, (2) provisioned capacity, $Prov_d$ at each data center d is fixed, and (3) the key decision variables are not per-epoch: $f_{a,d,i}$ and $b_{a,d,i}$.

5.4.3 Results

Load balancing: We first consider a load balancing objective, where we try to minimize the maximum utilization of any data center for CLEANSLATE and INTERMEDIATE, and evaluate the optimality gap: $\frac{Intermediate}{CleanSlate}$. We vary the mix of traffic (delay sensitive and delay tolerant) and the latency budgets of the applications. We consider 4 hours of peak load on a week day in November and reconfigure the traffic routing every 5mins. We take the maximum load observed for each data center over the 4 hour period for both CLEANSLATE and INTERMEDIATE. Figure 5.6 shows the optimality gap with different delay budgets and traffic mix. The key takeaway is that for all latency budgets, INTERMEDIATE performs close to the CLEANSLATE design. More concretely, we observe an optimality gap of about 5-10%, if the latency budget is 10ms and fraction of latency-sensitive traffic is less than 70%. For all other cases, the optimality gap is less than 5%, and it converges to close to 0 if the fraction of latency sensitive traffic is 90% or more. The reason the optimality gap is reducing as the fraction of latency-sensitive traffic increases is because then even clean slate has very few opportunities for balancing load across sites. Since most traffic is latency constrained, there are only a few sites where the traffic can be sent to.

Reduction in provisioning cost: Next, we consider a provisioning exercise to minimize the resources needed to handle the time varying traffic patterns generated across a week. The metric of interest here is the relative savings that INTERMEDIATE and CLEANSLATE provides vs. today's de-

ployment model where all traffic is usually routed to the nearest data center: $\frac{Cost_{Intermediate/CleanSlate}}{Cost_{Today}}$. We observe that with increasing number of data centers, we can achieve similar benefits in resource savings in Figure 5.7. We can save as much as 2 times more S/P-GW resources with INTERMEDIATE and CLEANSLATE as compared to Today's static architecture, and as 6 times more total resources, if we also consider other network functions.[3]. CLEANSLATE and INTERMEDIATE achieves these benefits while satisfying the latency budgets for data traffic.

5.4.4 Summary

The data-driven simulations show that:

- INTERMEDIATE is within 10% of CLEANSLATE in terms of load balancing for a variety of traffic mix and latency budgets.
- Both CLEANSLATE and INTERMEDIATE need 6× less resources as compared to Today for a cellular core with 1000 data centers.

We argue that this INTERMEDIATE approach can form the basis of a minimally disruptive cellular core. We show in the following sections how such an approach can be implemented using only minimal changes in cellular infrastructure while assuring complete legacy compliance. This makes it a very attractive option for the carriers.

5.5 System Overview and Challenges

The previous section showed that we can potentially address most of the limitations of the cellular core today through a hypothetical INTERMEDIATE design, which is 3GPP compliant. In this section, we give an overview of KLEIN, a practical instantiation of the INTERMEDIATE design. We discuss our envisioned core network deployment, the challenges in realizing SIMPLE and outline our key ideas to address these challenges.

[3]In the case of "ALL NF" in Figure 5.7, we also took into account placement of middleboxes other than S/P-GW (e.g., Firewalls, Transcoder etc.) by assuming hypothetical service chains for different traffic classes.

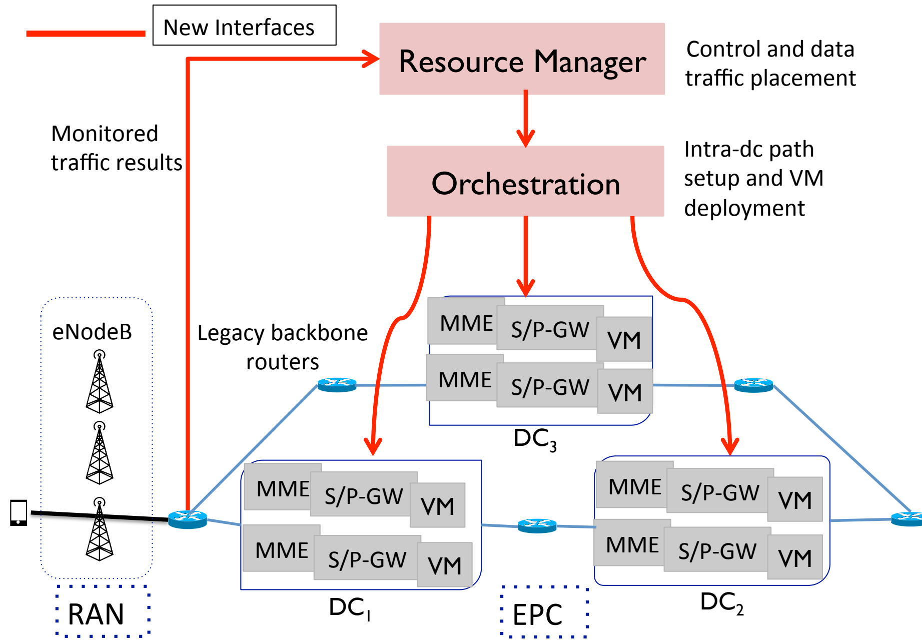


Figure 5.8: KLEIN system overview.

5.5.1 Overview

We envision that the cellular carrier has deployed many data centers. We assume that the cellular carrier can have both large data centers and small data centers. Large nation wide carriers already have a few thousand central offices [16]. We assume data centers are connected from the base station via traditional backbone routers and inter-connecting links. Each data center has commodity hardware servers and runs virtualized EPC functions (e.g., MME, S/P-GW) and other network functions (e.g., Firewall, NAT).

Our objective is to dynamically distribute the network load and provision virtualized network functions over the provider's available compute/network resources. This requires a resource manager which deals with load distribution and placement of virtualized network functions. For achieving these we want to construct a backwards compatible network orchestration layer, which is compatible with existing 3GPP mechanisms and requires minimal changes to the existing core network.

As seen in Figure 5.8, KLEIN extends the existing cellular core in three simple ways: (1) virtualized network functions instead of fixed hardware appliances, (2) resource manager, which performs dynamic resource management, and (3) a backwards-compatible network orchestration

Number of UEs	Number of data centers	Computation Time
∼50,000	6	∼20s
∼50,000	100	∼500s
∼50,000	1000	>1 day

Table 5.1: Scalability with a 2-level resource management decomposition.

layer for implementing the output of the resource manager. We argue that each of these changes is minimally disruptive. First, for (1), we observe that the use virtual functions is already on several carrier roadmaps [9, 40, 8, 126]. Second, (2) is a "bolt-on" control component that does not require any additional network infrastructure. Finally, for (3), we observe that in contrast with CLEANSLATE, KLEIN does not require changes to the existing 3GPP mechanisms or core network routing.

5.5.2 Challenges

(1) Responsive resource management: The two key challenges in designing a responsive and scalable resource manager are:

- First, the problem size makes it difficult to develop a responsive and scalable resource manager, for instance, a nation wide cellular carrier in the nearby future may have billions of devices and few thousand data centers. Specifically, this entails solving a large optimization problem, which can not scale to such input size even with state-of-the-art solvers. Even a 2-level decomposition does not scale, as shown in Table 5.1. It takes >1 day for it to reconfigure the load, even for a small input size of 50,000 UEs and 1000 data centers.

- Second, the resource manager has to instantiate both the cellular control and data plane functions, which have inter-dependencies. As we described in §2, MME is the cellular control function, and S/P-GW are data plane functions in EPC. The MME interacts with S/P-GW during different events, e.g., a UE's attachment to the network and eNodeB handover. 3GPP provides guidelines for delay budgets for MME and S/P-GW [18]. The S/P-GW delay budgets depends primarily on the requirements of the data traffic, whereas the delay budget for MME depends on the device characteristics (e.g., mobile smartphone device vs stationary

M2M device). MME delay budgets are more stringent because the MME deals with all the signaling traffic from the UE. If we assume that MME and S/P-GW can be placed in different data centers, we have three different types of delay budgets to consider, $Budget_{t,a}^{data}$, $Budget_{t}^{UE-MME}$ and $Budget_{t}^{MME-SGW}$, for device type t and application a, Modeling these three constraints, yields a quadratic constraint, as we show later in §5.6.

(2) Network orchestration: The second key challenge is to implement the output of the resource manager, i.e., map a UE's data and cellular control traffic to VMs. This has broadly three challenges: 1) Backwards compatible wide-area orchestration to get the UE to the selected data center, and 2) Intra-dc orchestration: to get the traffic through selected VMs, 3) Handling load reconfiguration, because unlike today's core network which is static, KLEIN derives its benefits from dynamically reconfiguring the network load, which may require moving a UE's traffic to a different data center to re-balance the load on the network. This raises the question if KLEIN can use existing orchestration and 3GPP mechanisms?

5.6 Resource Manager

The KLEIN resource management module distributes the network load across the datacenters, while ensuring that the latency budgets for different applications are satisfied. The key challenge here is achieving responsiveness at scale. In order to realize the benefits of the vision we outlined in §5.4, we need this module to rebalance the load and assign UEs to compute resources on fine-grained timescales (tens of seconds). However, this is challenging because the underlying distribution problem entails solving a large-scale and non-linear optimization. To address this, we use a combination of three key ideas: (1) solving the problem at an aggregate rather than UE granularity; (2) decoupling the problem of placing control and data-plane functions; and (3) decomposing the global optimization into a three-level hierarchy. As we will show this enables near optimal performance at scale.

We begin by setting up the key requirements of the optimization problem and then present our key ideas.

5.6.1 Problem Formulation

Provider Setup: We assume that the cellular core has been provisioned with a set of data centers $\{D_d\}_d$ and high-capacity backbone switches and links. Traffic from the base stations will be forwarded to one or more data centers. Each data center D_d has pre-provisioned capacity with fixed number of servers and each server s has fixed number of VM slots. We assume that the network is partitioned into *regions* – a collection of data centers in geograhpical proximity. This is similar to the way the core network is partitioned today [57].

The cellular operator has historical traffic patterns and has rough estimates of traffic volumes associated with end-user applications and cellular control traffic. Let, $Data_{u,a,e}$ represent the data traffic associated with a UE u, an application a, in an epoch e and let $Ctrl_{u,e}$, represents the control signalling traffic associated with the UE u, in an epoch e. This information may be collected using network monitorig data (e.g., NetFlow).

Processing requirements: Different applications and device types may require different processing requirements. For instance a M2M device may be required to go through a specific chain of service or NFs. Similarly video traffic may go through additional Transcoder middleboxes. For each traffic class, c, consisting of a combination of device-type t and application a, there is a associated policy service chain or a sequence of NFs and each device type t is also as associated with a set of cellular control functions it needs.

Objective: The goal is to decide the assignment of data-plane and cellular control-plane traffic to a data center, and provisioning of required EPC functions and middleboxes. There are a few considerations in this assignment. First, we want traffic load across servers in the core network to be balanced. Specifically, the utilization of data centers to be balanced. Second, we need to ensure that each UE u meets it processing (e.g., service chains) and latency bounds on data and control plane functions.

Formulation: We introduce three key decision variables: and (1) $n_i^{d,s}$ denoting how many VMs of type v_i (can be MME, S/P-GW and other network functions) of NF i to run on server s of data center d; (2) $DP_{u,s,d}$ which denotes whether data-traffic for UE u is processed in server s in data center d and (3) $CP_{u,s,d}$ which denotes whether cellular control traffic for UE u is processed at

server s in datacenter D.

Unfortunately, solving this problem is challenging on two key fronts. First, this is a large discrete optimization problem where the problem size (million of UEs and potentially thousands of data centers) makes it computationally intractable. Second, attempting to model constraints on the latency budgets between data-plane and control-plane functions inevitably yields quadratic constraints as shown below which make the problem even harder to solve with off-the-shelf solvers.[4] Specifically, Eq 5.8 highlights how modeling the latency between the SGW and MME introduces a non-linear interaction between the CP and DP variables. $L_{d,d'}$ refers to the latency between data center d and d'.

$$\forall u \in t : \sum CP_{u,s,d} \times DP_{u,s',d'} \times L_{d,d'} \leq Budget_t^{MME-SGW} \tag{5.8}$$

5.6.2 Key Ideas

As we saw above, solving this problem is challenging because of the *scale* and the *non-linear* dependencies across the decision variables. To address these issues, we introduce three key heuristcs:

- *Aggregation:* The first insight is that we do not need to solve the problem at a UE granularity. We may be able to achieve near-optimal results by aggregating UEs into groups of UEs, and make decisions at coarser aggregate granularities.

- *Hierarchical decomposition:* Secondly, we observe that in practice we do not always need to solve the global optimization problem every few seconds. We can decompose the optimization along the natural hierarchy of global, regional, and intra-datacenter local controllers. For instance, the global controller need not assign the precise server inside the datacenter or the specific datacenter and can instead delegate these to the regional and local controllers respectively. Thus, higher levels which need a more global view solve simpler problems at coarser timescales, while the lower levels which need to be more responsive to avoid performance issues can run more rapid reconfigurations.

[4]Note that in section §5.4, we only modeled the data-plane latencies, hence there were no quadratic constraints.

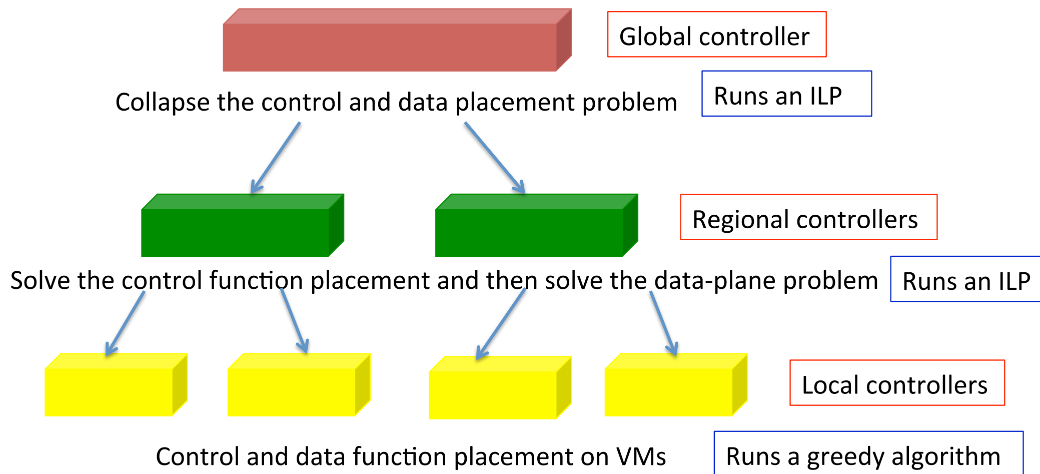


Figure 5.9: Decomposition and decoupling in resource management.

- *Decoupling control and data placement:* The key reason for the quadratic constraint in Eq 5.8 is that we were trying to solve the joint optimization of placing control and data functions. We can break this nonlinearity in one of two ways: (1) We can choose the control function placement and then solve the data-plane problem, with additional contstraint of $Budget_t^{MME-SGW}$ or (2) constrain the control and data plane functions to be in the same server (i.e., collapsing CP and DP). As we will see below, we find that approach (2) works well in the global controller and apporoach (1) in the regional controllers.

5.6.3 Our Approach

Next, we describe how we synthesize the above three heuristics into a practical and scalable resource management solution.

Three level hierarchy: We decompose the optimization problem into three logical subproblems following the natural structure of large cellular providers. Figure 5.9 shows this decompistion.

1. The global controller runs a *Region Selection Problem (RSP)* that assigns (aggregate) UE groups to specific regions;

Algorithm 1 Local controller heuristic for SSP

▷ **Input:** *IntraRackCost*, *InterRackCost*, $CP_{g,d}$, $DP_{g,d}$, CF_g, $SC_{g,a}$, $DF_{a,i}$, $Data_{g,a}$, $Ctrl_g$
▷ **Output:** $n_i^{d,s}$
#Assignment of *MME* VMs to servers
foreach UE group g assigned to data center d
    while all of $Ctrl_g$ not assigned
        #Consider control traffic footprint CF_g
        do Assign v_{mme} to the emptiest server
#Assignment of data plane NF VMs to servers
foreach UE group assigned to data center d
    foreach application a
        while all NF i in the service chain $SC_{g,a}$ not considered
            do $N \leftarrow v_i$
        $localize(N)$
▷ function $localize$ tries to assign all of its input VMs to the same server or rack
$localize(N)$ {
assign all VMs in N to emptiest server
    if failed
        then assign all VMs to emptiest rack
    if failed
        then split VMs in N across rack
    Update $n_i^{d,s}$
}

$$
\begin{aligned}
&Minimize \;\; MaxRegionUtil, subject\ to && (5.9)\\
&\forall r : Cap_r = \sum_{d \in r} Cap_d && (5.10)\\
&\forall r : Load_r < Cap_r && (5.11)\\
&\forall g : CF_g = \sum_{u} CF_u && (5.12)\\
&\forall r : Load_r = \sum_{g,r} CP_{g,r} \times Ctrl_g \times CF_g && (5.13)\\
&\forall g, r : CP_{g,r} \in \{0, 1\} && (5.14)\\
&\forall r : MaxRegionload > Load_r && (5.15)\\
&\forall r : FracRegionLoad_r = Load_r / Cap_r && (5.16)\\
&\forall r : MaxRegionUtil > FracRegionLoad_r && (5.17)\\
&\forall g : \sum_{d} CP_{g,r} \times Ctrl_g = Ctrl_g && (5.18)\\
&\forall g, r : L_{g,r} = \frac{\sum_{u \in g} \sum_{d \in r} L_{u,d}}{\sum_{u \in g} \sum_{d \in r} (1)} && (5.19)\\
&\forall g, r : L_{g,r} \times CP_{g,r} < Budget_g^{UE-MME} && (5.20)
\end{aligned}
$$

Figure 5.10: Global controller formulation for distributing load across regions.

2. The regional controller then runs the *Data center Selection Problem (DSP)* and further subdivides set of the (aggregate) UE groups assigned to it across the datacenters in its region;

3. The local or intra-datacenter controller runs a *Server Selection Problem (SSP)* which assigns specific servers inside each selected data center (as given by DSP) to run the required VMs.

This decomposition enables us to scale as the individual RSP, DSP, SSP problems can be solved respectively by the global, regional and local controllers. We also evaluated other degrees of decomposition and found that for the workload characteristics, this 3-level decomposition was a sweet spot between scalability and complexity; e.g., we tried a 2-level decomposition strategy and earlier showed in §5.5 that it does not scale (Table 5.1). We describe the specific optimization subproblem each tackles and the approach we use next.

Region Selection Problem: (RSP): In the first step, KLEIN global controller distributes traffic

$$\begin{aligned}
&Minimize\ \ MaxDCUtil, subject\ to && (5.21)\\
&\forall d : Load_d < Cap_d && (5.22)\\
&\forall d : Load_d = \sum_{g,d} CP_{g,d} \times Ctrl_g \times CF_d && (5.23)\\
&\forall g, d : CP_{g,d} \in \{0, 1\} && (5.24)\\
&\forall d : MaxDCload > Load_d && (5.25)\\
&\forall d : FracDCLoad_d = Load_d / Cap_d && (5.26)\\
&\forall d : MaxDCUtil > FracDCLoad_d && (5.27)\\
&\forall g : \sum_{d} CP_{g,d} \times Ctrl_g = Ctrl_g && (5.28)\\
&\forall g, d : L_{g,d} \times CP_{g,d} < Budget_g^{UE\text{-}MME} && (5.29)
\end{aligned}$$

Figure 5.11: Regional controller formulation for control traffic placement (CP).

across region such that the load is balanced at a region-level granularity. The global controller takes as input the $Ctrl_g$ and $Data_g$ values, as well the aggregate capacities of individual regions and assigns each aggregate UE group, g to a specific region based. To break the non-linear/quadratic dependency between the control and data-plane functions, the RSP simply assigns both to be in the same region, collapsing the CP and DP. We can formulate the RSP as an ILP, and rerun this ILP every after every 60 minutes periodically. We discuss the choice of this reconfiguration period in §5.9.1 In case any region is overloaded, they can make a `upcall` to KLEIN's global controller to reconfigure the load. The RSP formulation in essence is similar to the ILP formulation described in §5.4 for INTERMEDIATE, except that reconfiguration decisions are made for UE groups and load is distributed across regions. We describe this formulation in detail in Figure 5.10. Note, CF_u refers to the control traffic footprint of a device, u, i.e., processing cost per packet (e.g., CPU usage). $L_{u,d}$ refers to the latency from a mobile device u to data center d.

Data center Selection Problem (DSP): Each regional controller then has the responsibility for selecting the data center for every UE group g that has been assigned to it by the RSP. Specifically, it has to choose a data-center for the control functions and another (possibly different) datacenter for the data-functions. At this stage, the regional controllers seek to distribute load at an aggregate datacenter granularity within the region. Now, the DSP runs a two-step procedure to break the

$$
\begin{aligned}
&Minimize\ \ MaxDCUtil, subject\ to && (5.30)\\
&\forall d : Load_d < Cap_d && (5.31)\\
&\forall d : Load_d = \sum_g \sum_a DP_{g,d} \times Data_{g,a} \times DF_{g,a} && (5.32)\\
&\forall g, d : DP_{g,d} \in \{0, 1\} && (5.33)\\
&\forall d : MaxDCload > Load_d && (5.34)\\
&\forall d : FracDCLoad_d = Load_d / Cap_d && (5.35)\\
&\forall d : MaxDCUtil > FracDCLoad_d && (5.36)\\
&\forall g : \sum_d DP_{g,d} \times Data_{g,a} = Data_{g,a} && (5.37)\\
&\forall g : \sum_d DP_{g,d} = 1 && (5.38)\\
&\forall g, d : L_{g,d} \times DP_{g,d} < Budget_g^{Data} && (5.39)\\
&\forall g, d : L_{d,d'} \times DP_{g,d} \times CP_{g,d'} < Budget_g^{MME-SGW} && (5.40)
\end{aligned}
$$

Figure 5.12: Regional controller formulation for data traffic placement (DP).

quadratic dependency. First, it places the control functions and then runs a separate ILP for data plane functions. The ILP is similar to the DSP problem, except that the load is distributed across data centers, instead of regions. Figure 5.11 shows the control problem formulation and Figure 5.12 shows the data-plane formulation in detail. Note, $Data_{g,a}$ refers to the data traffic footprint of an application, a, i.e., processing cost per packet (e.g., CPU usage). This procedure runs roughly every 5 minutes. We discuss the choice of this reconfiguration period in §5.9.1. In case a data center is overloaded, the local controller can make a `upcall` to KLEIN's regional controller.

Server Selection Problem (SSP): Finally, each local controller within each datacenter has to distribute load across servers and instantiate VMs. We use a simple greedy heuristc here, choosing nodes with higher capacities and ensuring that NFs in the same chain are assigned to the same server or the same rack similar to prior work [70, 65]. We describe this in detail in Algorithm 1. Note, *IntraRackCost* refers to the latency between servers within the same rack, and *InterRackCost* refers to the latency between two servers in different racks. $SC_{g,a}$ refers to the service chain corresponding to a group of devices, g and an application, a.

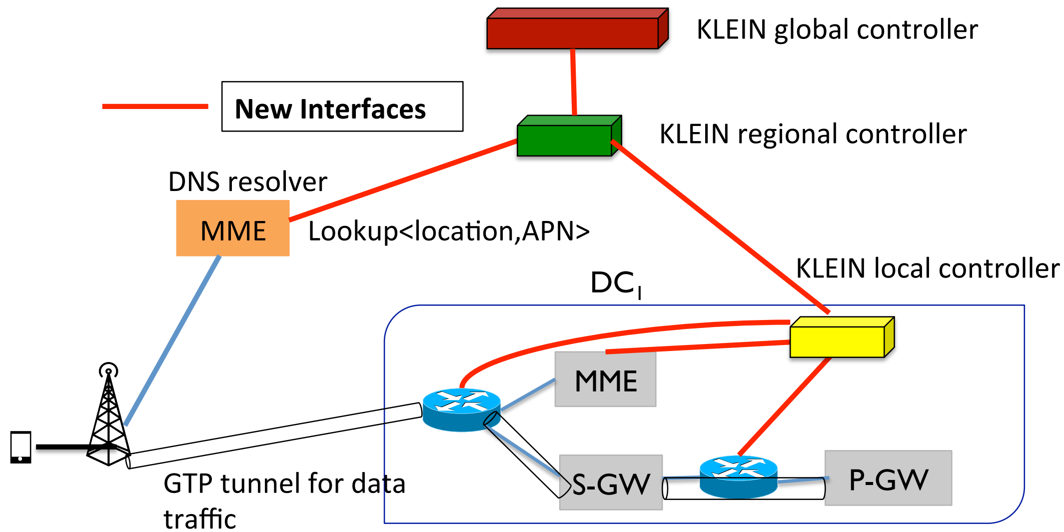


Figure 5.13: Network orchestration mechansims in a KLEIN based cellular core.

5.7 Network Orchestration

Given the output of KLEIN's resource manager, UE's data and cellular control traffic need to be assigned to VMs. We discuss how this is done in two phases: (1) wide-area orchestration to get the UE to the correct DC, (2) intra-DC orchestration to get the traffic through correct VMs. Figure 5.13 provides an overview of these orchestration mechanisms. Below we describe these mechanisms, and how KLEIN handles the reassignment of UEs.

5.7.1 Wide-area orchestration

In KLEIN, the cellular carrier's wide area network– the backbone network connecting the base stations and the data centers remains unmodified. We assume legacy routing, with carriers using existing tunneling mechanisms such as GTP to connect base stations to data centers. Today, when a UE attaches to an eNodeB, the eNodeB performs a DNS lookup to identify a MME to forward the attach request to. In response to the attach request, the MME acts as a DNS resolver to help select the S/P-GWs and set up the GTP tunnels.

In KLEIN, we enhance the attachment process slightly. In the initial MME selection DNS query, the eNodeB now includes the device and subscriber identifier of the UE in addition to the

location information it sends today. This DNS query is serviced by the nearest MME using its DNS resolver capabilities. It uses the device and subscriber identifier to map the UE into a KLEIN UE group, and subsequently uses the mapping provided by the KLEIN controller to select an MME for servicing the UE's attach request and as its future *home* for control plane traffic. An attach request is now sent by the eNodeB to the chosen home MME, which similarly identifies the UE's group from the device and subscriber information in request and selects a S/P-GW based on the UE group to GW mapping provided by the KLEIN controllers.

This enhanced registration procedure requires the MME to maintain a connection with the KLEIN's controllers to obtain an up-to-date UE group to MME and S/P-GW mapping. Fortunately, since MMEs use a standard DNS interface for server selection, such a connection requires minimal integration on the KLEIN controller's part.

5.7.2 Intra-datacenter orchestration

While the wide-area orchestration is responsible for choosing an MME, S-GW, and P-GW to route the UE's traffic. intra-DC orchestration is needed to implement the NF (middlebox) service chains that traffic passes through after it leaves the P-GW. There are two main considerations here:

1. After it leaves the P-GW, the next VM a packet needs to be sent to depends on the service chain associated with the traffic. We need to implement service chains corresponding to different traffic classes (UE devices and applications). These NFs may include elements such as NAT, firewalls, intrusion detection systems, TCP-termination proxies for improving latency and throughput, content-caches, or media transcoders.

2. With elastic scaling, each service chain NF may be implemented as a collection of load-balanced VMs and the number of such VMs may grow or shrink based on demand. This requires a load balancing mechanism at the level of each server.

We borrow from prior work [65, 116, 102, 12] to solve (1) and (2). We use SDN inside data centers to enforce service-chain policies by dynamically routing traffic to the desired sequence of VMs. We apply service chain policies based on APN values. We use a tag-based approach similar to [65, 116] to ensure that service chains can be correctly implemented. Each VM has a tag

value, and forwarding is done based on these tags. A service chain we assume is based on the UE device-type and application.

To balance load across multiple instances of a network function, we cannot use a dedicated load balancer, because it would itself become a bottleneck, since its on the path of every VM. We use a distributed load balancer at the level of each server, similar to [65, 102] to balance load among multiple instances of the same network functions in a server while handling scale-in and scale-out of the NF VMs.

5.7.3 KLEIN's reconfigurations

As KLEIN's resource manager reconfigures traffic load, a UE's processing may need to be migrated to new EPC instances. 3GPP protocol's mandate that MME selects the S/P-GW for a device. To ensure backwards-compatibility KLEIN maintains an interface with MME instances and dynamically updates in these MME instances mappings from UE to S/P-GW instances. We use standard 3GPP handover mechanisms to migrate a UE to a different MME/SGW instance [1, 15] and existing (but non-standardized) mechanisms for P-GW handover [5, 22]. To ensure that migrating the P-GW and NAT associated with a UE to a new site does not change the IP address associated with the UE's sessions we assume, as is common practice in carriers today, that the data center sites are connected over a layer 2 MPLS-based VPN. As has been demonstrated in previous work [139, 88], IP session migration using L2VPNs can accommodate even demanding applications like gaming.

Below we consider all the three possible types of reconfigurations by KLEIN and how they will be handled by the network orchestration layer:

- **Intra-DC:** In KLEIN, a local controller may reconfigure the load to a different VM, inside the same data center. If a UE's traffic is placed to a different MME instance, the local controller triggers a MME handover from the old MME instance using standard 3GPP handover procedures [1]. If the UE's data traffic needs to be moved to another S/P-GW instance within the same data center, the local controller initiates a standard 3GPP S-GW handover that moves the UE session context from the old S/P-GW instance to the new S-GW instance. These handovers are standard operating procedure for supporting mobility in existing 3GPP

networks, and are well supported by existing implementations. For an intra-DC handover, no changes to the service chain are needed, because the distributed load balancer ensures session affinity such that the new P-GW will continue to send a UE's packets to the same next hop VM in the chain. Typically, session affinity is implemented using a common flow-table across the SDN switches implementing the load balancer [34].

- **Intra-Region:** In case the UE's data or signalling traffic needs to be moved to another data center, the same standard mechanisms for MME, S/P-GW handover as in the intra-DC case are used. However, there are several important differences. First, in case of intra-DC handovers, the new mapping is provided by the regional controller to the local controllers. Second, when a P-GW or external facing NAT moves from one DC to another, we need to ensure that the UE's IP address does not change. This is achieved through the L2 MPLS-VPNs as described above. And finally, any middlebox state also needs to be migrated from the old DC to the new DC. This can be achieved by ensuring that the middleboxes support a state-migration protocol such as OpenNF [72].

- **Inter-Region:** In case a UE's traffic is moved to a data center in a different region, the global controller passes UE remappings to the regional controller (old-new data center), which then in turn passes it to the local controllers. The local controllers then trigger MME handover or S-GW handover. In case of P-GW, it moves the instance to the new data center. Similar considerations of middlebox state migration as in the intra-region case are involved in inter-region transfers.

5.8 Implementation

In this section, we describe an implementation of KLEIN that we will use in the following section for performance evaluation. The implementation consists of EPC, resource management and orchestration layers and uses emulated UEs and eNodeBs.

EPC implementation: We use `OpenAirInterface` (OAI) version 0.1, an open source software implementation of EPC functions (MME, S-GW, P-GW, HSS), and eNodeB and

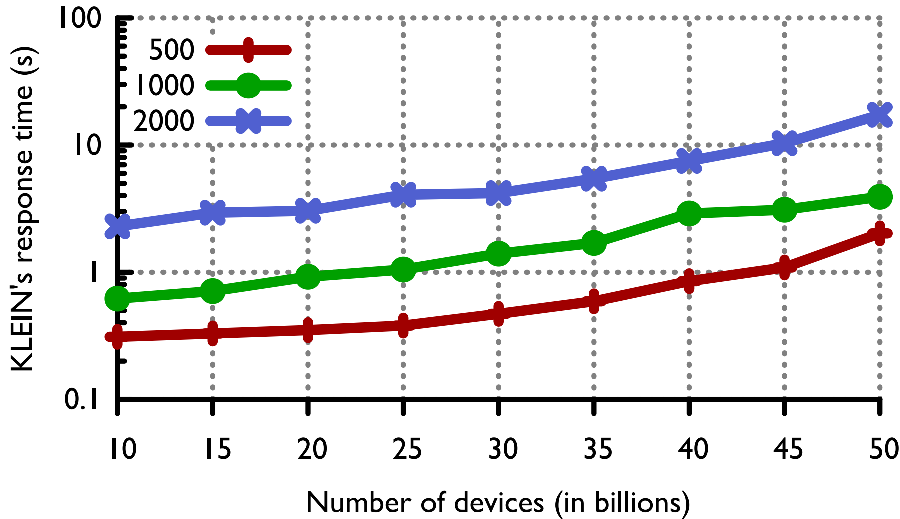


Figure 5.14: KLEIN**'s responsiveness**

UEs [28]. The UE and eNodeB behaviors are emulated; they are virtualized and run inside VMs. The EPC functions are also run inside a VM. In OAI, the EPC functions, viz., MME, S-GW, and P-GW, are tightly integrated and run inside the same VM. A key limitation of OAI is that the binding of UE to EPC is static. We extended OAI to enable dynamic remapping of the UE to a different EPC instance. We made some simplifying assumptions to do this, such as copying all UE contexts to all MMEs at the beginning. It is possible to do this for a small testbed and does not impact the broad performance measures we are interested in here.

Resource management and orchestration layers: Each EPC instance is realized as a VM. Emulated UEs and eNodeBs are also run inside VMs. In our testbed, we have a single UE per eNodeB because of OAI constraints. We use OpenvSwitch [27] to emulate switches inside the DCs. We developed custom implementations of the global and regional controllers using `CPLEX` to run these algorithms. We use the `Floodlight` [14] SDN controller which installs rules inside SDN switches to dynamically route traffic among the VMs.

5.9 Evaluation

In this section, we use a combination of real testbed and trace-driven simulations to demonstrate the following benefits of KLEIN:

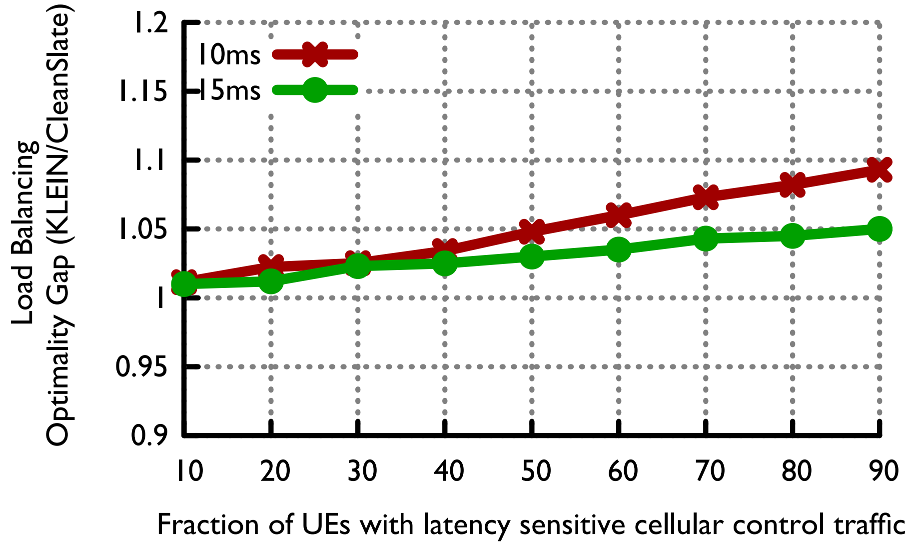


Figure 5.15: KLEIN**'s optimality**

1. KLEIN is scalable and near-optimal. Our system takes less than 20s to reconfigure a network with 2000 data centers and 5 billion devices and is within 10% of an ideal CLEANSLATE for a range of workloads. (§ 5.9.1)

2. The KLEIN end-to-end prototype implementation delivers the promised benefits and accurately mirrors the intended load distribution. (§ 5.9.2)

3. KLEIN offers new dimensions of elasticity and fault tolerance. It can handle data center failures both rapidly and efficiently, taking less than 2.3s, and reducing the maximum load by a factor of 2. (§ 5.9.3)

Setup and methodology: Before going into the results, we describe the testbed and simulation setups used for the experiments.

- **Testbed:** The testbed runs on Emulab [138]. The testbed uses upto 24 machine with 2.4 GHz 64-bit Quad Core and 12GB of RAM. On each machine, we assigned equal ammount of resources to each VM: 1 vCPU (virtual CPU) and 3GB of memory. Each VM runs Ubuntu 12.04 (Linux kernel version 3.13.0-32-generic).
- **Simulation setup:** The large scale simulations using data set in §5.3.1 are run on a machine with 80 CPU cores and 500 GB of RAM, with each core being a Xeon E74850 running at 2 GHz.

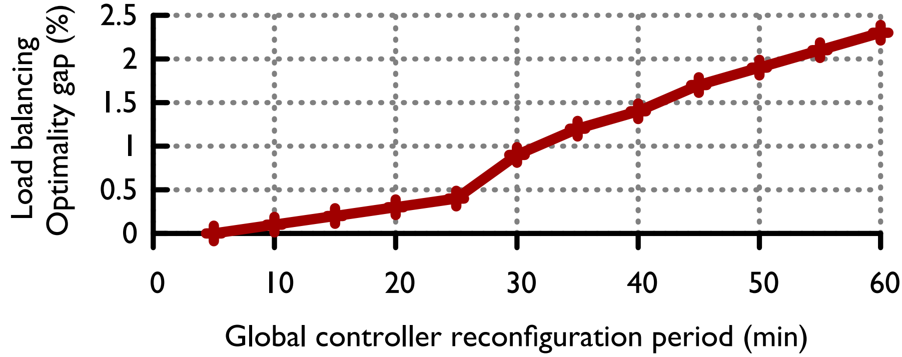


Figure 5.16: Varying reconfiguration period

- **Deployments:** We consider different core network sizes, from 500 data centers to 2000 data centers. The locations of such data centers are assumed close to the eNodeBs they meant to serve. This is done by a nearest-neighbor clustering of eNodeBs in the 2D space and locating the data centers at the centroid of such clusters.
- **Traffic demands:** We use the data set in §5.3.1 to get the traffic demands. In addition, we vary the mix of traffic (latency sensitive and latency tolerant) and the delay budgets from different traffic classes to consider a variety of traffic scenarios.

5.9.1 Scalability and Optimality

Scalability and responsiveness: Figure 5.14 shows the run time of KLEIN for different cellular core network sizes and number of devices. A configuration (A,B) corresponds to a deployment of *B* data centers, where we assumed *A* regions. The y-axis shows the total response time when both the global and regional controllers have to reconfigure load. We observe that run time is less than 20s for a network with 2000 sites and 50 billion devices.[5] In contrast with a 2-level decomposition (as shown in Table 5.1) even with aggregation, it takes more than a day to reconfigure the load.

Optimality: Figure 5.15 shows for different traffic mix and delay budgets, how KLEIN's heuristics perform. The y-axis shows the load balancing optimality gap, (KLEIN/CleanSlate). The CleanSlate solution optimally load balances the data and control plane functions without consider-

[5]Note for 50 billion devices, we use a total 50,000 aggregates of UEs.

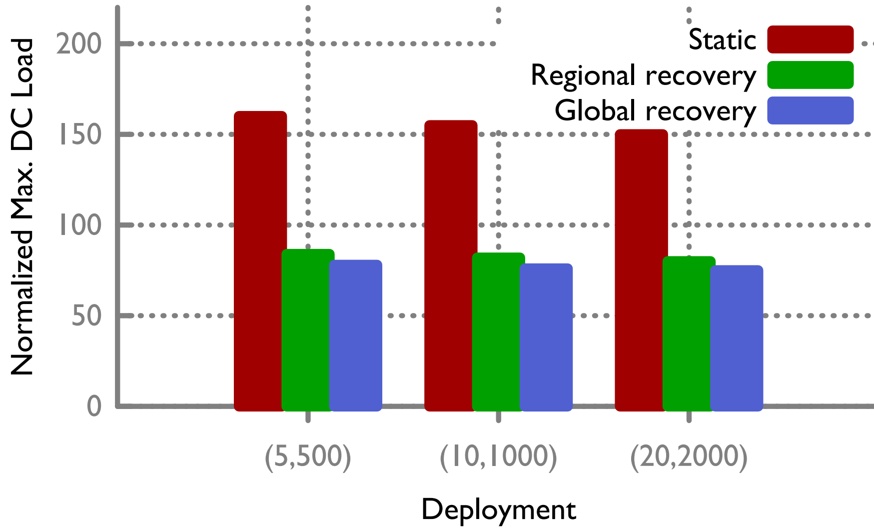


Figure 5.17: KLEIN**'s failure handling**

ing any

$Budget_t^{MME-SGW}$ constraints. We observe KLEIN's remains within 10% of the optimal solution, even for very stringent delay budget of 10 ms.

Varying reconfiguration period: Figure 5.16 shows the impact of reconfiguration period on the effectiveness of KLEIN's load balancing. The y-axis shows the load balancing gap, where we compare against a base line where every 5 mins both the regional and global controller reconfigure the load. We vary the global controller reconfiguration period, from 5 mins to 60 mins, while the regional controllers constantly reconfigure the load every 5 mins. Even with global reconfiguration at 60 min intervals, the gap is only 2.5%. We find that one effective strategy is where global controller reconfigures the load periodically every 60 mins, while the regional controller performs reconfigurations every 5mins.

5.9.2 End-to-End System Validation

We have validated our testbed demonstrating expected load balanced operation. Here, we consider 16 UEs attached to 16 base stations, and a total of 8 EPC instances. We use traffic data from 16 different base stations from the real data set, and scale their load to adjust to the capabilities of the testbed. We then dynamically map a UE's traffic to a specific EPC instance, based on the

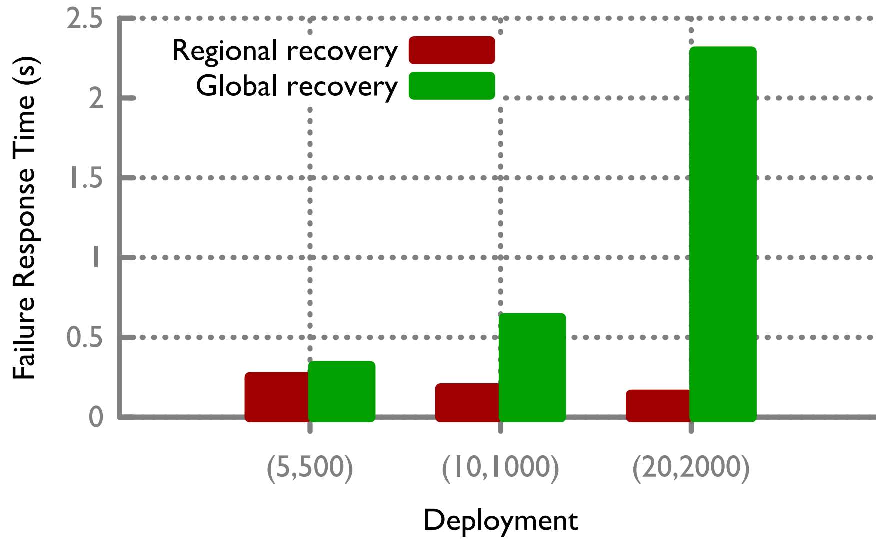


Figure 5.18: KLEIN**'s failure response**

load on the network and using SDN Floodlight controller, to dynamically install forwarding rules the switches. We consider three different traffic distributions which represent loads hour apart. Figure 5.21 shows the gap between the observed load and the output from KLEIN. Figure 5.21 shows the gap between the observed load and the output from KLEIN optimization (Expected). For all EPC instances, the observed load is within 5% of the expected load.

We also noted the number of control bytes exchanged when moving a UE to a different MME and S-GW instance. This is roughly 6 KB for S-GW and 2 KB for MME. These are modest numbers even when millions of UEs are moved.

5.9.3 New Opportunities

Elastic scaling: One of the new opportunities that KLEIN offers is the ability to dynamically scale EPC functions. This is because in KLEIN, the EPC functions are virtualized and can be instantiated based on demand. To illustrate this, we consider an elastic scaling experiment in our testbed, where we assume three UEs attached to an EPC instance which experiences a spike in load. The EPC instance is overloaded and we instantiate another EPC instance, and move some traffic to this new instance. The Figure 5.20 shows the time series. The load is evenly distributed and KLEIN avoids a potential overload scenario. We observe it takes in the order of a few ms to

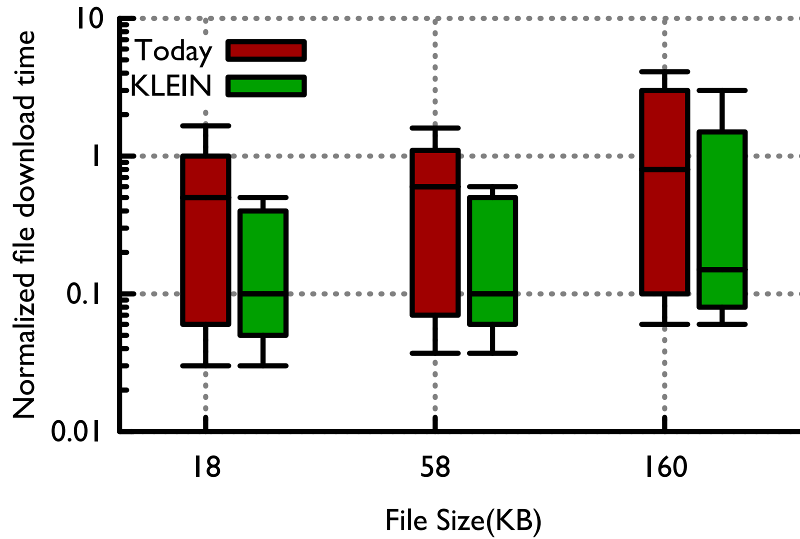


Figure 5.19: Impact on end-application performance

exchange UE state between different EPC instances and be able to route data traffic to the new EPC instance.

Failure handling: We consider a scenario where we fail individual data centers, and use KLEIN's dynamic remapping to reconfigure the load on the network. We compare it against a *Static* failure management strategy, where in the case of a failure, the load is statically mapped to the nearest data center. Figure 5.17 shows KLEIN avoids potential overload scenarios in the face of failure. We consider 3 different core network deployments, consisting of 500, 1000 and 2000 data centers. KLEIN can handle data center failures at two levels. It can perform (1) *Regional recovery*, where the regional controller tries to redistribute the load within the same region and (2) *Global recovery*, where the global controller redistributes the load across the cellular core. KLEIN can reduce maximum load on any data center in the cellular core by upto 100% as compared to a *Static* strategy. As shown in Figure 5.18, a *Global recovery* takes KLEIN upto 2.3s and a *Regional recovery* upto 0.3s.

5.9.4 UE Migrations

When we rebalance the load across sites, it requires us to either move some already active UEs to another site or assign new UEs to a site. In the former case, the following problem is created: every

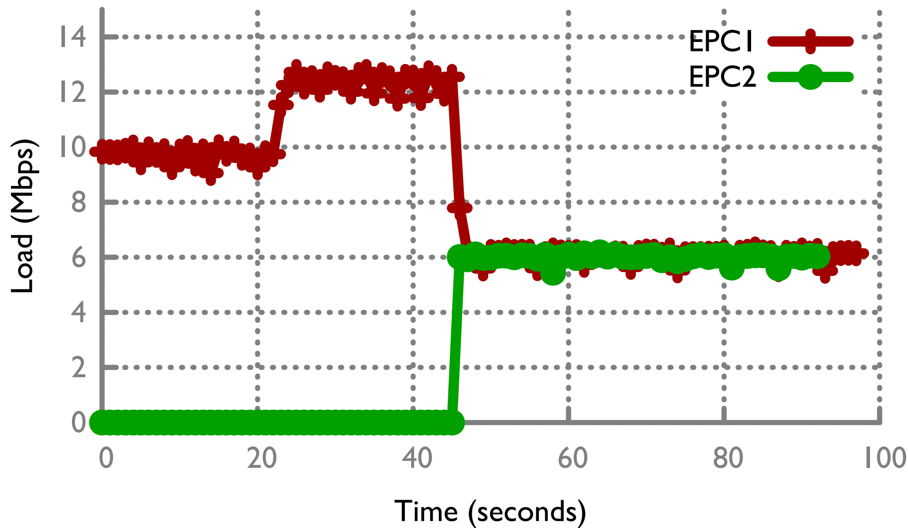


Figure 5.20: Handling traffic overload on an EPC instance by instantiating a new EPC instance.

active UE that moves to a new site, needs its relevant state to be moved from the old site to the new site, to ensure correct processing. We saw that using existing 3GPP protocols this state could be migrated. We also observed through microbenchmarking experiments that the overhead of moving this state is small. However, in case there are too many UEs that need to move simultaneously, we can constrain the number of UE migrations. In our resource manager, we also consider constraining the number of simultaneous UE migrations from one site to another site. We assume all the UEs are active, and consider the following UE migration threshold: we constrain the percentage of UEs that can move across consecutive load distribution configurations, e.g., only 10% UEs can be moved in any load configuration. These constraints can be implemented by both the global as well as the regional controllers. We also investigate how adding these UE migration constrains, affects the optimality of our solution. Here the optimality gap, refers to the gap between a constrained and unconstrained solution.

Figure 5.22 shows how these UE migration constrains impact the optimality of the load balancing solution with different number of UE groups: *Group100*, *Group1000* and *Group10000*. We observe in Figure 5.22 that if we have a UE migration threshold of 30%, the optimality gap (the difference between the optimal and the considered solution) is close to 0, where if the UE

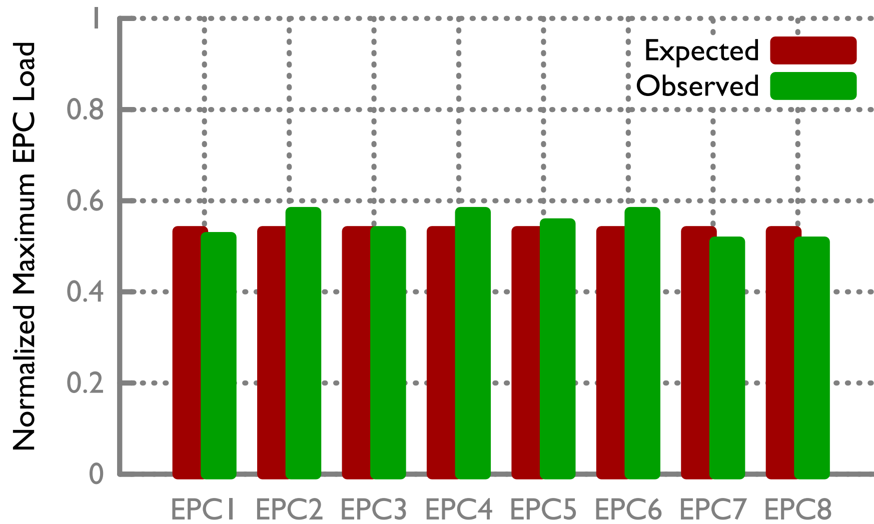


Figure 5.21: Validation of KLEIN on EPC testbed

migration threshold is close to 0, the optimality gap can be as high as 20%.

5.10 Summary

Today the cellular core suffers from a range of limitations and inefficiencies. As operators are rapidly improving the access bandwidth in (4G/LTE) networks, the core network remains the bottleneck. As carriers are looking to redesign their networks, an important question is whether we need to completely redesign the cellular core. Using a data-driven analysis we find that we can near-optimally achieve the benefits of an optimal clean slate approach using SIMPLE, a minimally disruptive redesign. The key observation is that by combining virtualized network functions with a smart resource management layer and a more distributed cellular core, we can achieve almost all of the promised elasticity benefits while working within the operational constraints of existing 3GPP standards.

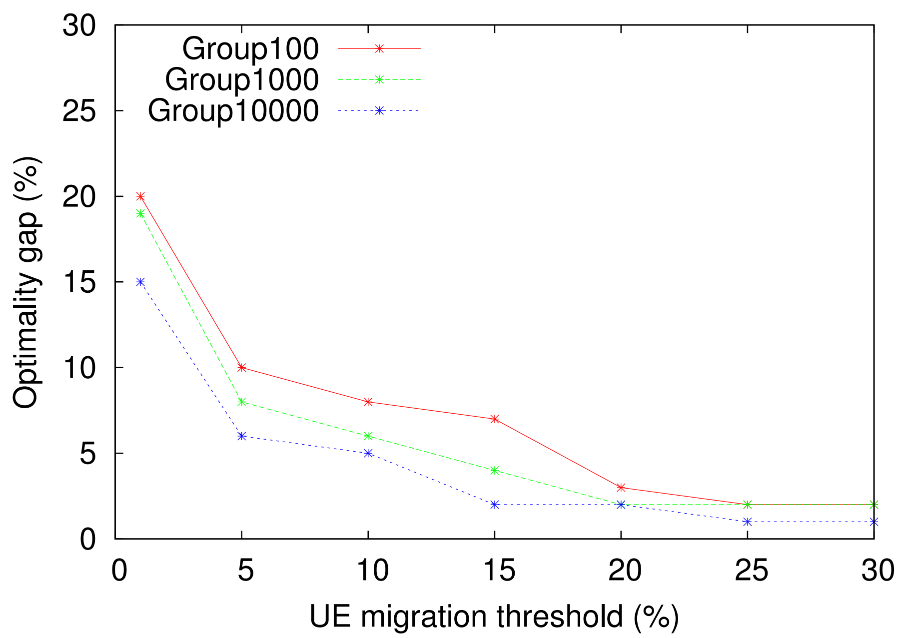


Figure 5.22: Varing UE migration threshold, and observing the impact on optimality gap.

Chapter 6

Conclusions and Future Work

The work in this dissertation was driven by finding a middle ground in designing management frameworks for middleboxes that can address middlebox-specific challenges while being minimally disruptive. Existing mechanisms for middlebox management are both inefficient and complex, while clean slate proposals require completely rearchitecting how middleboxes are implemented and managed. One of the main contributions of this dissertation is in showing in multiple contexts that most of the challenges in middlebox management can be addressed by practical management frameworks while requiring minimal changes in middlebox implementations and routing mechanisms. The key technical insights of this work are to show how several middlebox problems can be formulated as system-wide resource management problems, and to propose scalable and efficient heuristics for solving these otherwise hard optimization problems.

Next, I briefly summarize the main contributions of the work presented in this dissertation before highlighting some potential avenues for future work

6.1 Contributions

Efficient Middlebox-Specific Policy Enforcement:

I designed SIMPLE, an efficient system for enforcing middlebox-specific policies [116]. Chapter 3 described how SIMPLE addresses the challenges of policy composition, resource management and middlebox-induced dynamic modifications.

Efficient middlebox policy enforcement subject to SDN switch capacity constraints and middlebox capacity constraints is a hard problem. SIMPLE describes a scalable and efficient heuristic for solving this problem by decomposing the original hard optimization problem into a offline pruning stage and a fast online load balancing stage.

The other key challenge was to ensure correct policy enforcement. SIMPLE showed we can encode middlebox state in the packet to address this problem.

Lastly, we addressed the challenge of handling dynamic traffic transformations while assuming middleboxes to be black boxes. We designed a flow-correlation technique that can detect flow transformations by middleboxes.

SIMPLE did not require any modifications to middleboxes or any visibility inside middlebox implementations. It could readily work with existing middlebox deployments. At the same time, SIMPLE's load balancing algorithms are also enabler for NFV, to allow load distribution across virtual middlebox applications.

Designing a Flexible Cellular Core:
In Chapter 4, we proposed a re-design of the cellular core, KLEIN, that uses NFV and smart resource management. KLEIN confines largely to existing cellular signaling protocols and does not require any changes to the way of routing is done over the cellular backbone network.

In designing KLEIN, I addressed challenges of designing a scalable resource manager that could scale to thousands of sites and billions of devices. We leveraged the workload characteristics and nature of cellular deployments to design a hierarchical management plane that is highly responsive while offering a large number of benefits over the existing cellular core design.

6.2 Future Work

6.2.1 Customization and Modularization of EPC Network Functions

As the cellular architecture was originally designed to support voice communication seamlessly even for highly mobile users, we posit that todays cellular architecture is ill-equipped to carry a large number of end devices that carry increasingly heterogeneous forms of traffic. For example, the Internet of Things drive is pushing new forms of end devices into the network, which are often

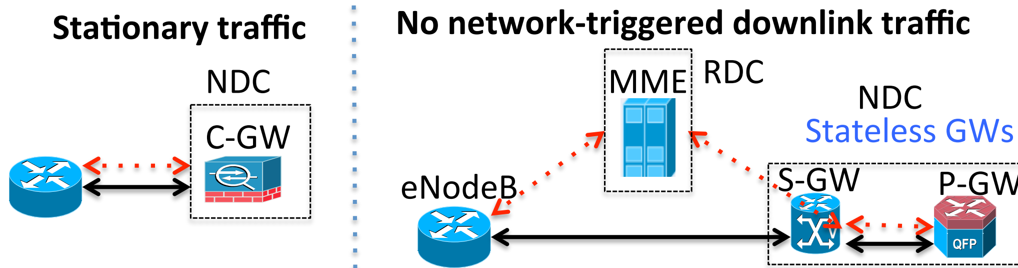


Figure 6.1: Examples of functional customization.

called M2M (Machine-to-Machine) devices. These M2M devices usually require little human intervention and show data communication characteristics different from smartphones (e.g., periodic short data sessions). Further, a significant portion of M2M devices are stationary and need minimal level of mobility/location management functions. However, even though they may need less functionality, the current EPC architecture carries the same amount of control state for an M2M device as a regular smartphone (e.g., GTP tunnel state, device location). While this renders the operators cost of supporting an M2M device on par with a smartphone, the user may expect a much cheaper charging model for M2M communications. In addition, a cellular provider will experience a huge scalability issue as billions of M2M devices are added to the cellular network. One approach to address this issue is to customize cellular core functions based on device/traffic types. Operators may develop customized functional modules at a smaller 'grain' than is done in today's network and then chain these modules differently for different traffic types that need different treatments in the control and data planes. This helps the network scale better by eliminating unnecessary system processing and saving resources. Also, simplified and optimally located service chains can result in performance gains. To illustrate potential benefits of functional customization, I here focus on two types of M2M devices. Figure 6.1 illustrates the two scenarios as well as the regular smartphone case for reference. Note, NDC refers to a National Data Center, and RDC refers to a Regional Data Center.

Stationary devices: They do not strictly need the entire MME or S-GW functionalities, as there are no needs for inter-eNodeB handovers and continuous location tracking and update. Here, the MME related functionalities strictly needed are: device registration/authentication and ability to lookup device location, providing much simpler MME design. Similarly, S-GW can also be

simplified as S-GW is involved in inter-eNodeB handovers. So instead of traditional full-fledged MME, S-GW and P-GW, we only need a customized GW module (that runs P-GW and a subset of MME and S-GW functions) that provides the minimum needed functionalities. Because the service chain can be simplified, a unit GW module can process more sessions, allowing them to scale much better.

Such a functional simplification can potentially improve the performance as well. To illustrate, we have performed a back-of-an-envelope calculation based on detailed LTE call flows [38] and propagation delay between eNodeB/MME/S-GW/P-GW of the cellular network under study. In the cases we consider, we find that bearer setup delay for initial device attachment procedure can be reduced by 70% and paging delay by 75%.

Devices with no network-triggered downlink traffic: Such devices always initiate communication (e.g., periodic transfer of update messages), and thus S-GW and P-GW do not strictly need to maintain GTP tunnel states for incoming messages to these devices all the time. Instead, depending on the traffic pattern, *stateless* S-GW and P-GW can clear particular GTP tunnel states and re-establish them when needed. Based on a recent study [127], compared to smartphones, M2M devices are less active, and the median active times are about 30%. This means that even though the devices are active for 30% time (or less), full-fledged S-GW and P-GW need to keep state for 100% of the time. We further perform a simple study using parameters such as the number of concurrent bearers S-GW/P-GW devices can support, session lengths and inter-arrival times of M2M traffic [127]. Applying Little's law, we estimate that each S-GW could support 0.5 million more devices (roughly 30 times more than currently could be supported), if the state was maintained only when the device was active. While these are back-of-the envelop calculations, they still illustrate the significance of the potential.

Here the key questions to explore are how EPC functions can be customized? How can we enable an architecture which can support highly customizable EPC stack? My work KLEIN in this dissertation, can be enabler for this, as it supports dynamic resource management with virtualized network functions in the EPC.

6.2.2 Efficient State Management

With dynamic resource management, a key challenge is we maybe required to move network function state. Some middleboxes maintain state (e.g., byte counters, socket context, device location), hence moving this state is necessary to ensure correct processing of packets. For instance, EPC functions such as MME maintain state related to device location and the device subscription attributes. Recent proposals, OpenNF [72] and SplitMerge[118] propose mechanisms for migrating middlebox state. However they focus on moving state within a data center and on traditional middleboxes like load balancer, IDS, NAT. In KLEIN, since we reconfigure the network load across data centers, network functions state maybe required to move across data centers. In KLEIN, we use existing cellular protocols for migrating state and the state is maintained inside individual EPC functions. Moving state frequently between highly distributed network functions may not be an efficient solution to address this problem.

A key question is what state should be maintained inside network functions and what state should be moved to controllers? Whether we need new APIs for managing state? How can we design more efficient mechanisms for moving state? One approach could be to investigate the state that is common across network functions such as device-specific state: device location and subscription state. To move this state to the data center's local controller, and keep the middlebox-specific state inside network functions. For moving state, we could imagine new APIs where device-specific state is moved between controllers of different sites while network-function specific state is moved between network functions and controllers.

Bibliography

[1] 3GPP Evolved Packet System (EPS); Evolved General Packet Radio Service (GPRS) Tunneling Protocol for Control Plane (GTPv2-C). http://www.3gpp.org/DynaReport/29274.htm/.

[2] A Simple Model for Determining True Total Cost of Ownership for Data Centers. http://tinyurl.com/kznlhn2.

[3] ADARA. http://www.adaranet.com/.

[4] Affirmed Networks. http://www.affirmednetworks.com/.

[5] Architectural EPC Extensions for Supporting Heterogeneous Mobility Schemes. Document by MEVICO, January 2013. http://www.mevico.org/D22.pdf.

[6] ARICENT. https://www.aricent.com/.

[7] Aryaka WAN Optimization. http://www.aryaka.com.

[8] AT&T Domain 2.0 Vision White Paper. http://tinyurl.com/p4uv3s3.

[9] AT&T launches virtualized packet core in Europe. http://www.fiercewireless.com/tech/story/att-launches-virtualized-packet-core-europe/2015-10-06.

[10] AT&T shifts 130K employees to focus on software networking transition. http://www.fiercetelecom.com/story/att-shifts-130k-employees-focus-software-networking-transition/2015-06-08.

[11] BladeLogic Sets Standard for Data Center Automation and Provides Foundation for Utility Computing with Operations Manager Version 5. Business Wire, Sept 15, 2003, .http://findarticles.com/p/articles/mi_m0EIN/is_2003_Sept_15/ai_107753392/pg_2.

[12] Contrail Architecture. http://www.juniper.net/us/en/local/pdf/whitepapers/2000535-en.pdf.

[13] Embrane. http://www.embrane.com/.

[14] Floodlight Controller. http://www.projectfloodlight.org/floodlight/.

[15] General Packet Radio Service (GPRS) enhancements for Evolved Universal Terrestrial Radio Access Network (E-UTRAN) access. http://www.3gpp.org/DynaReport/23401.htm/.

[16] Introducing ONOS - a SDN network operating system for Service Providers. http://onosproject.org/wp-content/uploads/2014/11/Whitepaper-ONOS-final.pdf.

[17] LTE Connectem Inc. http://www.connectem.net/.

[18] LTE Design and Deployment Strategies. http://tinyurl.com/lj2erpg.

[19] Managing the Signaling Storm. http://goo.gl/lkTyb1.

[20] Middleboxes: Taxonomy and Issues. https://tools.ietf.org/html/rfc3234.

[21] Mininet. http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet.

[22] Mobile Gateway Configuration Guide. Alcatel Lucent Technical Document, 2014. http://infoproducts.alcatel-lucent.com.

[23] Morgan Stanley Releases The Mobile Internet Report. http://www.morganstanley.com/.

[24] NEC's Simple Middlebox Configuration (SIMCO) Protocol (RFC 4540). https://tools.ietf.org/html/rfc4540.

[25] Network Functions Virtualisation. http://portal.etsi.org/nfv/nfv_white_paper.pdf.

[26] ONS2014 Keynote: John Donovan, Senior EVP, AT&T Technology & Network Operations. https://www.youtube.com/watch?v=tLshR-BkIas.

[27] Open vSwitch. http://openvswitch.org/.

[28] OpenAirInterface. http://www.openairinterface.org/.

[29] The OpenEPC Project. http://http://www.openepc.com/.

[30] OpenStack. https://www.openstack.org/.

[31] Palo Alto Networks. http://www.paloaltonetworks.com/.

[32] Percentage of all global web pages served to mobile phones from 2009 to 2015. http://www.statista.com/statistics.

[33] POX Controller. http://www.noxrepo.org/pox/about-pox/.

[34] Service Chain Load Balancing with OpenContrail. http://www.opencontrail.org/service-chain-load-balancing-with-opencontrail/.

[35] Snort. https://www.snort.org/.

[36] Solarflare solution overview. http://www.solarflare.com/.

[37] State of the News Media 2015. http://www.journalism.org/2015/04/29/state-of-the-news-media-2015/.

[38] The LTE Network Architecture. http://tinyurl.com/negszts.

[39] Top million US websites. http://ak.quantcast.com/quantcast-top-million.zip.

[40] vEPC in LTE networks: Time to move ahead. Blog, March 2015. https://techzine.alcatel-lucent.com/vepc-lte-networks-time-move-ahead?s_cid=smm15_tmc0481_bl.

[41] Verizon-Carrier Adoption of Software-defined Networking. https://www.youtube.com/watch?v=WVczl03edi4.

[42] Vyatta Software Middlebox. http://www.vyatta.com.

[43] World Enterprise Network and Data Security Markets. http://www.abiresearch.com/press/3591-Enterprise+Network+and+Data+Security+Spending+Shows+Remarkable+Resilience.

[44] World Enterprise Network Security Markets. http://www.abiresearch.com/research/product/1006059-world-enterprise-network-and-data-security/.

[45] The Zettabyte Era: Trends and Analysis. Cisco Technology White Paper, May 2015. http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/VNI_Hyperconnectivity_WP.pdf.

[46] Omer Abdelrahman and Erol Gelenbe. Signalling storms in 3G Mobile Networks. In *Proc. ICC*, 2014.

[47] S. Agarwal, M. Kodialam, and T.V. Lakshman. Traffic engineering in software defined networks. In *Proc. of INFOCOM*, 2013.

[48] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proc. of NSDI*, 2010.

[49] Hassan Ali-Ahmad, Claudio Cicconetti, Antonio de la Oliva, Martin Dräxler, Rohit Gupta, Vincenzo Mancuso, Laurent Roullet, and Vincenzo Sciancalepore. CROWD: An SDN Approach for DenseNets. In *Proc. of EWSDN*, 2013.

[50] James W. Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat. xOMB: Extensible Open Middleboxes with Commodity Servers. In *Proc. of ANCS*, 2012.

[51] Arsany Basta, Wolfgang Kellerer, Marco Hoffmann, Klaus Hoffmann, and Ernst-Dieter Schmidt. A virtual SDN-enabled LTE EPC architecture: a case study for S-/P-Gateways functions. In *Proc. of SDN4FNS*, 2013.

[52] Theophilus Benson, Aditya Akella, Anees Shaikh, and Sambit Sahu. CloudNaaS: A Cloud Networking Platform for Enterprise Applications. In *Proc. of SOCC*, 2011.

[53] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. The Case for Fine-Grained Traffic Engineering in Data Centers. In *Proc. of INM/WREN*, 2010.

[54] Rodrigo Braga, Braga, Edjard Mota, Mota, and Alexandre Passito, Passito. Lightweight DDoS Flooding Attack Detection Using NOX/OpenFlow. In *Proc. of LCN*, 2010.

[55] Martin Casado, Michael J. Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking Control of the Enterprise. In *Proc. of SIGCOMM*, 2007.

[56] M. Channegowda, R. Nejabati, and D. Simeonidou. Software-defined optical networks technology and infrastructure: Enabling software-defined optical network operations. *IEEE/OSA Journal of Optical Communications and Networking*, 5(10):A274–A282, 2013.

[57] Junguk Cho, Binh Nguyen, Arijit Banerjee, Robert Ricci, Jacobus Van der Merwe, and Kirk Webb. Smore: Software-defined networking mobile offloading architecture. In *Proc. of AllThingsCellular*, 2014.

[58] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. The Rabin–Karp algorithm. *Introduction to Algorithms*, 2001.

[59] J. Costa-Requena. SDN Intergation in LTE Mobile Backhaul Networks. In *Proc. of ICOIN*, 2014.

[60] Andrew R Curtis et al. DevoFlow: Scaling Flow Management for High-Performance Networks. In *Proc. of SIGCOMM*, 2011.

[61] N. Cvijetic, A. Tanaka, P.N. Ji, K. Sethuraman, S. Murakami, and Ting Wang. SDN and

OpenFlow for Dynamic Flex-Grid Optical Access and Aggregation Networks. *Journal of Lightwave Technology*, 32(4):864–870, 2014.

[62] Anupam Das, Cristian Lumezanu, Yueping Zhang, Vishal Singh, Guofei Jiang, and Curtis Yu. Transparent and Flexible Network Management for Big Data Processing in the Cloud. In *Proc. of HotCloud*, 2013.

[63] Advait Abhay Dixit, Fang Hao, Sarit Mukherjee, T.V. Lakshman, and Ramana Kompella. ElastiCon: An Elastic Distributed Sdn Controller. In *Proc. of ANCS*, 2014.

[64] 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 *Proc. of SOSP*, 2009.

[65] Seyed K. Fayaz, Yoshiaki Tobioka, and Vyas Sekar. Flexible and Elastic DDoS Defense Using Bohatei. In *Proc. USENIX Security*, 2015.

[66] Seyed Fayazbakhsh, Vyas Sekar, Minlan Yu, and Jeff Mogul. FlowTags: Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions. In *Proc. HotSDN*, 2013.

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

[68] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The road to sdn: An intellectual history of programmable networks. *SIGCOMM Computer Communication Review*, 44(2), April 2014.

[69] Anja Feldmann, Albert Greenberg, Carsten Lund, Nick Reingold, Jennifer Rexford, and Fred True. Deriving Traffic Demands for Operational IP Networks: Methodology and Experience. *IEEE/ACM Transactions on Networking*, 9(3):265–280, 2001.

[70] Aaron Gember, Anand Krishnamurthy, Saul St. John, Robert Grandl, Xiaoyang Gao, Ashok Anand, Thephilus Benson, Vyas Sekar, and Aditya Akella. Stratos: A Network-Aware Orchestration Layer for Middleboxes in the Cloud. In *Technical Report arXiv:1305.0209*, 2013.

[71] Aaron Gember, Prathmesh Prabhu, Zainab Ghadiyali, and Aditya Akella. Toward Software-defined Middlebox Networking. In *Proc. of HotNets*, 2012.

[72] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. OpenNF: Enabling Innovation in Network Function Control. In *Proc. of SIGCOMM*, 2014.

[73] Glen Gibb, Hongyi Zeng, and Nick McKeown. Outsourcing Network Functionality. In *Proc. of HotSDN*, 2012.

[74] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *Proc. SIGCOMM*, 2011.

[75] 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. *SIGCOMM Computer Communication Review*, 2005.

[76] Adam Greenhalgh, Felipe Huici, Mickael Hoerdt, Panagiotis Papadimitriou, Mark Handley, and Laurent Mathy. Flow Processing and the Rise of Commodity Network Hardware. In *Proc. of CCR*, 2009.

[77] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an Operating System for Networks. In *Proc. of CCR*, 2008.

[78] Aditya Gudipati, Li Erran Li, and Sachin Katti. RadioVisor: A Slicing Plane for Radio Access Networks. In *Proc. of HotSDN*, 2014.

[79] Aditya Gudipati, Daniel Perry, Li Erran Li, and Sachin Katti. SoftRAN: Software Defined Radio Access Network. In *Proc. of HotSDN*, 2013.

[80] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean P. Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. SDX: A Software Defined Internet Exchange. In *Proc. of SIGCOMM*, 2014.

[81] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. In *Proc. of HotSDN*, 2012.

[82] Brandon Heller, Srini Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. ElasticTree: Saving Energy in Data Center Networks. In *Proc. of NSDI*, 2010.

[83] Victor Heorhiadi, Michael K. Reiter, and Vyas Sekar. New Opportunities for Load Balancing in Network-wide Intrusion Detection Systems. In *Proc. of CoNEXT*, 2012.

[84] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *Proc. of SIGCOMM*, 2013.

[85] Hongxin Hu, Wonkyu Han, Gail-Joon Ahn, and Ziming Zhao. FLOWGUARD: Building Robust Firewalls for Software-defined Networks. In *Proc. of HotSDN*, 2014.

[86] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *Proc. of NSDI*, 2014.

[87] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-deployed Software Defined Wan. In *Proc. of SIGCOMM*, 2013.

[88] Virajith Jalaparti, Matthew Caesar, Seungjoon Lee, Jeffery Pang, and Jacobus Van der Merwe. SMOG: A Cloud Platform for Seamless Wide Area Migration of Online Games. In *Proc. of NetGames*, 2012.

[89] Hani Jamjoom, Dan Williams, and Upendra Sharma. Don't call them middleboxes, call them middlepipes. In *Proc. of HotSDN*, 2014.

[90] Xin Jin, Li Erran Li, Laurent Vanbever, and Jennifer Rexford. SoftCell: Scalable and Flexible Core Network Architecture. In *Proc. of CoNEXT*, 2013.

[91] D. Joseph and I. Stoica. Modeling middleboxes. *IEEE Network*, 2008.

[92] Dilip A. Joseph, Arsalan Tavakoli, and Ion Stoica. A Policy-aware Switching Layer for Data Centers. In *Proc.of SIGCOMM*, 2008.

[93] Peyman Kazemian, George Varghese, and Nick McKeown. Header Space Analysis: Static Checking for Networks. In *Proc. of NSDI*, 2012.

[94] James Kempf, Bengt Johansson, Sten Pettersson, Harald Luning, and Tord Nilsson. Moving the Mobile Evolved Packet Core to the Cloud. In *Proc. of WIMOB*, 2012.

[95] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, 2000.

[96] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A Distributed Control Platform for Large-scale Production Network. In *Proc. of OSDI*, 2010.

[97] Christian Kreibich, Nicholas Weaver, Boris Nechaev, and Vern Paxson. Netalyzr: Illuminating the Edge Network. In *Proc. of IMC*, 2010.

[98] L.E. Li, V. Liaghat, Hongze Zhao, M. Hajiaghay, Dan Li, G. Wilfong, Y.R. Yang, and Chuanxiong Guo. PACE: Policy-Aware Application Cloud Embedding. In *Proc. of INFOCOM*, 2013.

[99] Heikki Lindholm, Lirim Osmani, Hannu Flinck, Sasu Tarkoma, and Ashwin Rao. State Space Analysis to Refactor the Mobile Core. In *Proc. of AllThingsCellular*, 2015.

[100] James MacCauley, Aurojit Panda, Martin Casado, Teemu Koponen, and Scott Shenker. Extending SDN to Large-Scale Networks. *Open Network Summit, Research Track*, 2013.

[101] Joao Martins, Mohamed Ahmed, Costin Raiciu, and Felipe Huici. Enabling Fast, Dynamic Network Processing with clickOS. In *Proc. of HotSDN*, 2013.

[102] H. Matsuba, K. Joshi, M. Hiltunen, and R. Schlichting. Airfoil: A topology aware distributed load balancing service. In *Proc. of IEEE Cloud*, 2015.

[103] Stephanos Matsumoto, Samuel Hitz, and Adrian Perrig. Fleet: Defending SDNs from Malicious Administrators. In *Proc. of HotSDN*, 2014.

[104] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer

Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Computer Communication Review*, 2008.

[105] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing Software-defined Networks. In *Proc. of NSDI*, 2013.

[106] Mehrdad Moradi, Wenfei Wu, Li Erran Li, and Zhuoqing Morley Mao. SoftMoW: Recursive and Reconfigurable Cellular WAN Architecture. In *Proc. of CoNEXT*, 2014.

[107] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. DREAM: Dynamic Resource Allocation for Software-defined Measurement. In *Proc. of SIGCOMM*, 2014.

[108] Masoud Moshref, Minlan Yu, Abhishek Sharma, and Ramesh Govindan. vCRIB: Virtualized Rule Management in the Cloud. In *Proc. of HotCloud*, 2012.

[109] Yukihiro Nakagawa, Kazuki Hyoudou, and Takeshi Shimizu. A Management Method of IP Multicast in Overlay Networks Using Openflow. In *Proc. of HotSDN*, 2012.

[110] Parveen Patel, Deepak Bansal, Lihua Yuan, Ashwin Murthy, Albert Greenberg, David A. Maltz, Randy Kern, Hemant Kumar, Marios Zikos, Hongyu Wu, Changhoon Kim, and Naveen Karri. Ananta: Cloud Scale Load Balancing. In *Proc. of SIGCOMM*, 2013.

[111] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Computer Networks*, 1999.

[112] K. Pentikousis, Yan Wang, and Weihua Hu. Mobileflow: Toward software-defined mobile networks. In *Proc. of WIMOB*, 2013.

[113] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *Proc. of OSDI*, 2014.

[114] Philip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. A Security Enforcement Kernel for OpenFlow Networks. In *Proc. of HotSDN*, 2012.

[115] Himabindu Pucha, David G. Andersen, and Michael Kaminsky. Exploiting Similarity for Multi-Source Downloads using File Handprints. In *Proc. of NSDI*, 2007.

[116] Zafar Ayyub Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. Simple-fying middlebox policy enforcement using sdn. In *Proc. of SIGCOMM*, 2013.

[117] Ramya Raghavendra, Jorge Lobo, and Kang-Won Lee. Dynamic Graph Query Primitives for SDN-based Cloudnetwork Management. In *Proc. of HotSDN*, 2012.

[118] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *Proc. of NSDI*, 2013.

[119] Saqib Raza, Guanyao Huang, Chen-Nee Chuah, Srini Seetharaman, and Jatinder Pal Singh. MeasuRouting: A Framework for Routing Assisted Traffic Monitoring. *IEEE/ACM Transactions on Networking*, 20(1):45–56, 2012.

[120] L. Rizzo, M. Carbone, and G. Catalli. Transparent acceleration of software packet forwarding using netmap. In *Proc. of INFOCOM*, 2012.

[121] Luigi Rizzo and Giuseppe Lettieri. VALE, a Switched Ethernet for Virtual Machines. In *Proc. of CoNEXT*, 2012.

[122] Charalampos Rotsos, Nadi Sarrar, Steve Uhlig, Rob Sherwood, and Andrew W. Moore. OFLOPS: An Open Framework for Openflow Switch Evaluation. In *Proc. of PAM*, 2012.

[123] M.R. Sama, L.M. Contreras, J. Kaippallimalil, I. Akiyoshi, Haiyang Qian, and Hui Ni. Software-Defined Control of the Virtualized Mobile Packet Core. In *IEEE Communications Magazine*, 2015.

[124] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *Proc. of NSDI*, 2012.

[125] Vyas Sekar, Sylvia Ratnasamy, Michael K. Reiter, Norbert Egi, and Guangyu Shi. The middlebox manifesto: Enabling innovation in middlebox deployment. In *Proc. of HotNets*, 2011.

[126] S.Elby. Carrier Vision of SDN and future applications to achieve a more agile mobile businesss. Keynote at the OpenFlow World Congress, 2012.

[127] Muhammad Zubair Shafiq, Lusheng Ji, Alex X. Liu, Jeffrey Pang, and Jia Wang. A First

Look at Cellular Machine-to-machine Traffic: Large Scale Measurement and Characterization. In *Proc. of SIGMETRICS*, 2012.

[128] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making Middleboxes Someone else's Problem: Network Processing As a Cloud Service. In *Proc. of SIGCOMM*, 2012.

[129] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the Production Network Be the Testbed? In *Proc. of OSDI*, 2010.

[130] Seungwon Shin, Phillip A. Porras, Vinod Yegneswaran, Martin W. Fong, Guofei Gu, and Mabry Tyson. FRESCO: Modular Composable Security Services for Software-Defined Networks. In *Proc. of NDSS*, 2013.

[131] Neil Spring, Ratul Mahajan, and David Wetherall. Measuring isp topologies with rocketfuel. In *Proc. of SIGCOMM*, 2002.

[132] M. Stiemerling, J. Quittek, and T. Taylor. Middlebox Communication (MIDCOM) Protocol Semantics (RFC 5189). `https://tools.ietf.org/html/rfc3989`.

[133] Vytautas Valancius, Nikolaos Laoutaris, Laurent Massoulié, Christophe Diot, and Pablo Rodriguez. Greening the Internet with Nano Data Centers. In *Proc. of CoNext*, 2009.

[134] Guohui Wang, T.S. Eugene Ng, and Anees Shaikh. Programming Your Network at Run-time for Big Data Applications. In *Proc. of HotSDN*, 2012.

[135] R Wang, D Butnariu, and J Rexford. Openflow-Based Server Load Balancing Gone Wild. In *Proc. of Hot-ICE*, 2011.

[136] Zhaoguang Wang et al. An Untold Story of Middleboxes in Cellular Networks. In *Proc. SIGCOMM*, 2011.

[137] Zhaoguang Wang, Zhiyun Qian, Qiang Xu, Zhuoqing Mao, and Ming Zhang. An Untold Story of Middleboxes in Cellular Networks. In *Proc. of SIGCOMM*, 2011.

[138] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold,

Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. *SIGOPS Operating Systems Review*, 36(SI):255–270, 2002.

[139] Timothy Wood, K. K. Ramakrishnan, Prashant Shenoy, and Jacobus van der Merwe. CloudNet: Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines. In *Proc. VEE*, 2011.

[140] Tobias Flach Ethan Katz-Bassett David Choffnes Ramesh Govindan Xing Xu, Yurong Jiang. Investigating Transparent Web Proxies in Cellular Networks. In *Proc. of PAM*, 2015.

[141] Qiang Xu, Junxian Huang, Zhaoguang Wang, Feng Qian, Alexandre Gerber, and Zhuoqing Morley Mao. Cellular Data Network Infrastructure Characterization and Implication on Mobile Content Placement. In *Proc. of SIGMETRICS*, 2011.

[142] Soheil Hassas Yeganeh and Yashar Ganjali. Beehive: Towards a Simple Abstraction for Scalable Software-Defined Networking. In *Proc. of HotNets*, 2014.

[143] Curtis Yu, Cristian Lumezanu, Yueping Zhang, Vishal Singh, Guofei Jiang, and Harsha V. Madhyastha. FlowSense: Monitoring Network Utilization with Zero Measurement Cost. In *Proc. of PAM*, 2013.

[144] Minlan Yu, Lavanya Jose, and Rui Miao. Software Defined Traffic Measurement with OpenSketch. In *Proc. of NSDI*, 2013.

[145] Minlan Yu, Jennifer Rexford, Michael J. Freedman, and Jia Wang. Scalable Flow-based Networking with DIFANE. In *Proc. of SIGCOMM*, 2010.

[146] Zafar Qazi, Cheng-Chun Tu, Luis Chiang, Rui Miao, Vyas Sekar, and Minlan Yu. Practical and incremental convergence between SDN and Middleboxes. *Open Network Summit, Research Track*, 2013.

[147] Y. Zhang and V. Paxson. Detecting Stepping Stones. In *Proc. of SSYM*, 2000.

[148] Ying Zhang, N. Beheshti, L. Beliveau, G. Lefebvre, R. Manghirmalani, R. Mishra, R. Patneyt, M. Shirazipour, R. Subrahmaniam, C. Truchan, and M. Tatipamula. StEERING: A software-defined networking for inline service chaining. In *Proc. of ICNP*, 2013.