TOWARDS A TRAFFIC-AWARE CLOUD-NATIVE CELLULAR CORE

by

Amit Kumar Sheoran

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

Department of Computer Science

West Lafayette, Indiana

August 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Sonia Fahmy, Chair
Department of Computer Science

Dr. Ninghui Li
Department of Computer Science

Dr. Xiangyu Zhang
Department of Computer Science

Dr. Voicu Popescu
Department of Computer Science

Dr. Puneet Sharma
Hewlett Packard Labs

Approved by:
Dr. Kihong Park

To my parents

ACKNOWLEDGMENTS

First and foremost, I would like to express my profound appreciation and gratitude to my advisor Prof. Sonia Fahmy for her unrelenting guidance, encouragement, and support during the course of my PhD. She not only motivated me to start the PhD program, but, her patience and insights - from start to finish - have been instrumental in both, shaping my research and making it presentable. I hope I will have the ability and patience to mentor and collaborate with others like she has mentored me, with an open mind and a gentle hand.

I am extremely grateful to Prof. Bruno Ribeiro and Prof. Chunyi Peng for helping me navigate various technical challenges during the course of my work. I benefited immensely from their technical expertise and their unyielding attitude towards problem solving. I am grateful to Prof. Voicu Popescu, not only for serving on my thesis committee, but for also being a wonderful collaborator. I also want to thank my thesis committee members Prof. Ninghui Li and Prof. Xiangyu Zhang for their insights and timely feedback.

I am deeply indebted to Dr. Puneet Sharma, who has not just been a long-term collaborator and thesis committee member, but also was pivotal in inspiring me to join the PhD program. I thank him for his mentorship and the inspiring discussions - often tempered with real-world pragmatism. I owe a debt of gratitude to my mentor and collaborator at AT&T Labs, Dr. Jia Wang for her insights and guidance during our work, and for inspiring me into taking my first steps towards a research career. I also want to thank Matthew Osinski for his guidance and support during (and after) our work at AT&T Labs.

I would like to thank my colleagues and coauthors, both past and present - Lianjie Cao, Navin Modi, Danushka Menikkumbura, Mijanur Palash, Xiangyu Bu, Jianfei Gao, Mohamed Zahran, Jiayi Meng, and Umakant Kulkarni - for their support and thought-provoking discussions which were often the highlight of my day. A special note of thanks to my friends - Rakshit, Jawad, Shivaram, Nidhi, Raj, Ananya, and Kalyani - for all the discussions and hikes which made my experience at Purdue a memorable one.

On a personal note, I would like to thank my parents and my sister for their love and unwavering belief in me. My wife, Divya, deserves a special thanks for the innumerable sacrifices that she made over the years. I will eternally be grateful for her love, encouragement, and for the selflessness with which she allowed me to pursue this degree. Finally, I would also like to extend my gratitude to our parents for their constant support and encouragement.

TABLE OF CONTENTS

LIST OF TABLES 10

LIST OF FIGURES 11

ABBREVIATIONS 13

ABSTRACT 14

1 INTRODUCTION 15
  1.1 Empirical Analysis of Container-driven VNFs 16
  1.2 Latency-driven Deployment of VNFs 16
  1.3 Container-driven Deployment of VoLTE Caller-ID Validation 17
  1.4 Thesis Statement 17

2 BACKGROUND 18
  2.1 Introduction to Cellular Networks 18
  2.2 Introduction to Network Functions Virtualization 20
  2.3 Related Work 22

3 A CASE FOR CONTAINER-DRIVEN FINE-GRAINED VNF RESOURCE FLEXING 26
  3.1 Introduction 26
  3.2 Virtual Network Functions 27
    3.2.1 EPC Control Plane 28
    3.2.2 Intrusion Detection Systems 28
  3.3 Experimental Setup 28
    3.3.1 Evolved Packet Core Setup 29
    3.3.2 IDS Setup 30
  3.4 MME Experiments 31
    3.4.1 Methodology 31
    3.4.2 Results 32

3.5 Suricata Experiments 35
3.5.1 Methodology 35
3.5.2 Results 36
3.5.3 Results from the Second Trace File 38
3.5.4 Root Cause Analysis for VM Results 39
3.6 Snort Experiments 39
3.7 Suricata and Snort with Multiple Instances 40
3.8 Chapter Summary 43

4 CONTAIN-ED: AN NFV MICROSERVICE SYSTEM FOR CONTAINING E2E LATENCY 44
4.1 Introduction 44
4.2 Contain-ed Architecture 46
4.3 Contain-ed in Action 51
4.3.1 Project Clearwater: IMS in the Cloud 52
4.3.2 Mapping with Contain-ed 53
4.4 Experimental Evaluation 54
4.4.1 Experimental Setup 54
4.4.2 Experimental Results 56
4.5 Chapter Summary 59

5 INVENIO: PROVENANCE-DRIVEN MICROSERVICE DEPLOYMENT IN THE CELLULAR CORE 60
5.1 Introduction 60
5.2 Motivation 63
5.3 Challenges 65
5.4 Problem Definition 68
5.5 Invenio Design 70
5.5.1 Inputs 71
5.5.2 Header Inference 72
5.5.3 Noise Filtering 74

5.5.4 Attribute Extraction 74
5.5.5 Session Slicing 74
5.5.6 Transaction Slicing 74
5.5.7 Placement 75
5.6 Evaluation 75
5.6.1 Clearwater Case Study 77
Architecture 77
Affinity Analysis 79
Performance 81
5.6.2 Microservice-based VoLTE 81
Affinity Analysis 83
Placement 84
Performance 87
5.7 Other Applications 89
5.8 Chapter Summary 90

6 NASCENT: CONTAINER-DRIVEN DEPLOYMENT OF VOLTE CALLER-ID VERIFICATION 91
6.1 Introduction 91
6.2 Background and Related Work 93
6.2.1 VoLTE Architecture and Call Setup. 93
6.2.2 Caller-ID Spoofing. 95
6.3 Design Goals and Lessons Learned 96
6.3.1 Goals 96
6.3.2 Comparison of Existing Proposals and Lessons Learned 97
6.4 Nascent Design 98
6.4.1 Overview 98
6.4.2 Variants 100
6.4.3 Meeting Design Goals 103
6.5 Experimental Evaluation 103

6.5.1 Implementation and Experimental Setup 104
6.5.2 Experimental Results 106
Traditional Deployment Model 106
NFV Deployment Model 109
Selective Validation 111
Tradeoffs among the Three Variants 111
6.6 Discussion 113
6.7 Chapter Summary 114

7 CONCLUSIONS AND FUTURE WORK 115
7.1 Provisioning in the 5G Core 116
7.2 ML-based Provisioning of Cellular Networks 117

REFERENCES 119

LIST OF TABLES

3.1 Comparison of Activation time. 35

3.2 Memory Usage of Suricata at 1X workload 36

3.3 CPU Usage of Suricata at 1X workload. 37

3.4 CPU Usage of Suricata at 2X workload. 37

3.5 CPU Usage of Snort at 4X workload 39

3.6 Memory Usage of Snort at 4X workload 40

4.1 IMS and EPC Affinity Aggregates (AAs) 50

5.1 Example user and session headers 66

5.2 *Invenio*-generated candidate header names 73

5.3 Testbed configuration 76

5.4 NF affinity for Clearwater 79

5.5 Functionality of NFs in PCRF 82

5.6 Manual analysis of communication between VNFCs in PCRF μservice designs . 84

5.7 NF affinity for VoLTE 85

6.1 Comparison of network-assisted caller-ID spoofing detection solutions. 95

LIST OF FIGURES

2.1 LTE Network Architecture 18
2.2 Difference in Monolithic and Microservice architecture 22
3.1 Setup for EPC Tests 29
3.2 Setup for IDS Tests 30
3.3 Time taken to handle registration requests. 32
3.4 Time taken to handle registration requests on bare metal with varying the request rate. 34
3.5 Time taken to handle registration requests by bare metal and two instances of VMs or containers. 34
3.6 Cumulative packets decoded and dropped by Suricata in four setups at all loads. 38
3.7 Cumulative packets decoded and dropped by Snort in four setups at all loads. . 40
3.8 Performance of Snort with multiple VM and containers instances. 41
3.9 Performance of Suricata with multiple VM and container instances. 42
4.1 Contain-ed Architecture 48
4.2 Clearwater Architecture 52
4.3 Clearwater IMS Call Flow 53
4.4 Contain-ed setup with REGISTER AA 56
4.5 Successful REGISTER calls 57
4.6 Contain-ed setup with REGISTER/SUBSCRIBE AAs 58
4.7 Successful SUBSCRIBE calls 58
5.1 A microservice-based cellular network 63
5.2 Impact of transaction type distribution on number of messages exchanged between NF pairs in microservice-based VoLTE implementation 64
5.3 *Invenio* architecture 70
5.4 VoLTE workloads 78
5.5 Clearwater architecture 79
5.6 Impact of latency and affinity on Clearwater 80
5.7 VoLTE system architecture 82
5.8 PCRF architectures 83

5.9 VoLTE placement with *Invenio* 86
5.10 Impact of latency and affinity on VoLTE performance 88
6.1 LTE network architecture and VoLTE call setup flow. 94
6.2 NASCENT design: 1a and 3a are used to access local memory (i.e., no messages are exchanged). 99
6.3 NASCENT variants. 102
6.4 Experimental setup. 104
6.5 Percentage of successful calls with 0% spoofed calls. 106
6.6 CPU utilization of IMS Server with 0% spoofed calls. 107
6.7 Average latency in VoLTE call setup with 0% spoofed calls. 107
6.8 Percentage of successful calls with 5% spoofed calls. 109
6.9 Percentage of successful calls with 10% spoofed calls. 109
6.10 Percentage of successful calls with 0% spoofed calls with NFV deployment. . . . 110
6.11 CPU utilization with 0% spoofed calls with NFV deployment. 110
6.12 Average latency in VoLTE call setup with 0% spoofed calls with NFV deployment. 111
6.13 Percentage of successful calls with varying spoofing percentage for the Pre-Notification-Rx+ variant. The number within parentheses indicates the % of calls validated. 112

ABBREVIATIONS

CPU	Central Processing Unit
CSP	Communication Service Providers
EPC	Evolved Packet Core
IMS	IP Multimedia Subsystem
LTE	Long Term Evolution
NE	Network Element
NF	Network Function
NFV	Network Functions Virtualization
SFC	Service Function Chain
SLA	Service Level Agreement
SLO	Service Level Objective
REST	Representational State Transfer
TCP	Transmission Control Protocol
VM	Virtual Machine
VNF	Virtual Network Function
VNFC	Virtual Network Function Component
VoLTE	Voice over LTE

ABSTRACT

Advances in virtualization technologies have revolutionized the design of the core of cellular networks. However, the adoption of microservice design patterns and migration of services from purpose-built hardware to virtualized hardware has adversely affected the delivery of latency-sensitive services.

In this dissertation, we make a case for cloud-native (microservice container packaged) network functions in the cellular core by proposing domain knowledge-driven, traffic-aware, orchestration frameworks to make network placement decisions. We begin by evaluating the suitability of virtualization technologies for the cellular core, and demonstrating that container-driven deployments can significantly outperform other virtualization technologies such as Virtual Machines for control and data plane applications.

To support the deployment of latency-sensitive applications on virtualized hardware, we propose using Virtual Network Function (VNF) bundles (aggregates) to handle transactions. Specifically, we design *Invenio* to leverage a combination of network traces and domain knowledge to identify VNFs involved in processing a specific transaction, which are then collocated by a traffic-aware orchestrator. By ensuring that a user request is processed by a single aggregate of collocated VNFs, Invenio can significantly reduce end-to-end latencies and improve user experience.

Finally, to understand the challenges in using container-driven deployments in real-world applications, we develop and evaluate a novel caller-ID spoofing detection solution in Voice over LTE (VoLTE) calls. Our proposed solution, NASCENT, cross validates the caller-ID used during voice-call signaling with a previously authenticated caller-ID to detect caller-ID spoofing. Our evaluation with traditional and container-driven deployments shows that container-driven deployment can not only support complex cellular services but also outperform traditional deployments.

1. INTRODUCTION

Advances in virtualization technologies and widespread availability of cloud infrastructure have revolutionized modern cellular networks. Cellular providers are increasingly adopting Network Functions Virtualization (NFV) to leverage the cloud infrastructure and support low cost, flexible, and on-demand service delivery. While the migration of monolithic implementations of Network Functions (NFs) to virtualized cloud-based infrastructure has been successful, the transition of hardware-based NFs to cloud-native NFs is far from complete. To maximize the use of cloud resources and allow fine-grained resource flexing, telecommunication NFs must transition from traditionally used monolithic designs to cloud-native design patterns such as microservices architectures, containerized services and distributed management and orchestration.

A key challenge in deployment of cloud-native telecommunication NFs is the inherent design philosophy of traditional telecommunications NFs. In order to ensure deterministic performance and support low latency services, NFs in the cellular core adopt monolithic designs that result in Virtual Network Functions (VNFs) of large resource footprints which do not readily lend themselves to cloud-native designs. Large footprint VNFs packaged in Virtual Machines (VMs) severely limit the ability of the network orchestrator to place these VNFs in close proximity, which can lead to high end-to-end (e2e) latency and Service Level Agreement (SLA) violations. Additionally, owing to complexities in cellular protocol designs and long Service Function Chains (SFCs) used in service delivery, SLA violations can quickly cascade to severely degrade the quality of service.

In this work, we make a case for a cloud-native cellular core by tackling the problems faced by network providers in migration of hardware-based NFs to cloud-native VNFs. Specifically, we make the following contributions: (1) We analyze the efficacy of lightweight virtualization technologies in supporting cellular VNFs, (2) We propose the use of functional decomposition and VNF bundles (aggregates) to support deployment of low-latency applications, and (3) We analyze the benefits of container-driven deployments in solving real-world problems by developing a novel called-ID spoofing detection solution and comparing its performance in traditional and container-driven deployments.

We briefly describe our proposed solutions in the remainder of this chapter.

1.1 Empirical Analysis of Container-driven VNFs

The primary factors driving NFV adoption are efficient resource usage and agility in terms of elastic resource allocation. Multiple virtualization technologies can be used to deploy VNFs in the cloud, and selecting the right virtualization technology is a vital step towards deploying cloud-native cellular cores.

We compare the performance and resource usage of three VNFs with bare metal (BM), container, and VM instantiations. Our experiments indicate that compared to VM-driven deployments, container-driven deployments have shorter instantiation times, incur significantly lower performance overhead, and result in smaller overall footprints. These attributes make containers an ideal choice for cellular VNFs where booting delays directly affect system performance, and higher elasticity is desired to reduce the overall operational cost.

1.2 Latency-driven Deployment of VNFs

Current NFV deployments have merely focused on migration of traditional monolithic NFs to virtualized cloud-based infrastructure. However, owing to the high memory and resource footprints of monolithic applications, dynamic instantiation of these VNFs can frequently lead to conditions where VNFs in an SFC are placed on different racks or even different data centers (DCs). Distributed instantiations of SFCs and inherent latency variations within data centers can result in significant performance degradation since current applications and network protocol stacks are designed for traditional deployments and therefore react poorly to such “unbounded” latencies.

In Contain-ed, we propose functional decomposition of monolithic VNFs to support deployment of latency-sensitive applications on DCs with unpredictable latency. We use traffic-type based VNF aggregation to create affinity aggregates (AA) which are lightweight microservice-based VNF bundles that handle messages of a single traffic type, and then use traffic-aware orchestration to co-locate VNFs in AAs. To enable orchestrators to develop an effective placement strategy for microservice-based SFCs without prior knowledge of service functionality, we also propose a system, *Invenio*, that uses the knowledge of user actions and provenance information extracted from network traces to compute transactional affinity between VNFs in an SFC for each traffic type. The transactional affinity values are then used to make placement decisions to meet latency constraints of a given traffic type.

1.3 Container-driven Deployment of VoLTE Caller-ID Validation

An important consideration in adaption of new technologies in cellular core is the ease with which they can integrated with existing cellular core. Since services in cellular core are provided by long SFCs, VNFs deployed using cloud-native designs must often interact with existing monolithic functions to support new services. It is therefore essential to understand the efficacy of containers in supporting real-world applications with long SFCs.

We use Voice over LTE (VoLTE) caller-ID spoofing detection service as a case study to understand the impact of container-driven cellular core. In this case study, we propose NASCENT, a novel Network-assisted caller ID authentication solution to validate the caller-ID used during call setup. We prototype and experimentally evaluate the performance of NASCENT on traditional (bare metal) and container-driven deployments to understand the efficacy of containers in supporting new real-world services. Our experiments confirm that container-driven deployments are not only capable of supporting complex services with long SFCs, but can also outperform existing bare metal service deployments.

1.4 Thesis Statement

The thesis of this work is as follows: *Deploying latency-sensitive cellular services designed using monolithic architectures on shared virtualized hardware can significantly reduce quality of service. Functionality-driven decomposition of monolithic applications into microservice components, coupled with traffic-aware orchestration frameworks, can increase quality of service.*

2. BACKGROUND

In this chapter, we give some background on cellular networks and network functions virtualization, and discuss some related research.

2.1 Introduction to Cellular Networks

The advent of mobile broadband radio access and the convergence of Internet and mobile services have revolutionized cellular networks. Adoption of all Internet Protocol (IP) network architecture by cellular networks has enabled them to support high-bandwidth services promised by mobile broadband. Successive generations of cellular networks (3G, 4G, 5G) promise to support increasingly complex services driven by the evolution of high speed Radio Access Networks (RANs) and faster access to internet servers. The part that links these RANs and internet together is the core network. Core network combines the power of high-speed radio access technologies with the power of innovative application enabled by the internet to realize the promise of mobile broadband. Figure 2.1 presents the architecture of the 4th generation of cellular networks (4G) called Long-Term Evolution (LTE) [1]. LTE network is the first generation of cellular core to integrate an all IP core with high speed RAN.

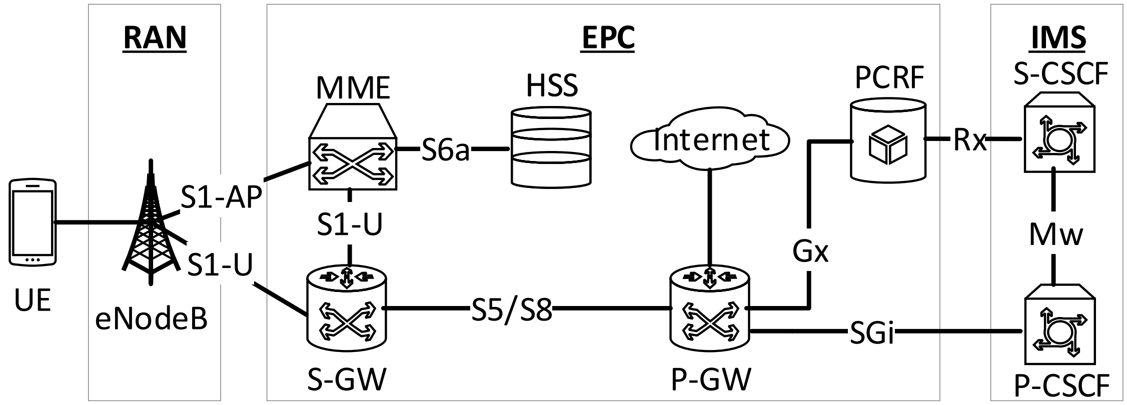


Figure 2.1. LTE Network Architecture

The basic LTE architecture is shown in Figure 2.1. The LTE network can be divided in two parts: RAN and Evolved Packet Core (EPC). The RAN comprises of the eNodeB, which is responsible for providing radio connectivity to the users. The EPC network consists of the Mobility Management Entity (MME), Home Subscriber Server (HSS), Serving Gateway (S-GW), Packet Data Network Gateway (P-GW), and the Policy Rules and Charging Function (PCRF). The MME and HSS are control plane entities responsible for signaling, mobility, and security functions for User Equipment (UE) attaching over the RAN. S-GW and P-GW are data path entities responsible for data transfer to and from the UEs. The P-GW along with the PCRF is also responsible for Quality of Service (QoS) and gating control functions for each mobile subscriber.

To uniquely identify a cellular subscriber, each UE's SIM card is associated with two globally unique identifiers (a) an International Mobile Subscriber Identity (IMSI) and (b) a Mobile Station International Subscriber Directory Number (MSISDN) (telephone number). The SIM card also contains a shared secret which is used to authenticate the UE to the network with the help of a user database called HSS. The MME acts as the initial attach point to the network. The MME relies on the subscription information stored in the HSS to handle network attachment and creation of default/dedicated bearers that provide basic IP connectivity to the UE. The MME supports several standard interfaces such as S1-AP, S6a/S13, and S11 to interact with eNodeB, HSS, and S/PGW respectively. During the initial attach, the UE generates an attach request to the eNodeB. Upon receipt of the attach request from the eNodeB over S1-MME interface, the MME authenticates the user by retrieving the authentication information stored in the user profile in the HSS using Diameter [2] based S6a interface.

After the authentication step is completed, the MME initiates creation of default bearer towards the S-GW over the S11 interface using the GTP-C protocol. The S-GW creates a default bearer, and requests the P-GW to allocate an IP address to the attaching UE. After the context setup at the eNodeB, the default bearer is activated and basic data connectivity is established with the UE. After the default bearer procedure is completed, the P-GW initiates the creation of a dedicated EPC bearer to support the Voice over LTE (VoLTE) service. Unlike previous generation of cellular networks (2G/3G), VoLTE carries voice traffic and its signaling, in IP packets and therefore a dedicated bearer is created to tunnel VoLTE signaling packets.

The VoLTE and SMS service in EPC is provided with the support of an *external* subsystem called IP Multimedia Subsystem (IMS) [3]. The IMS offers voice and multimedia services over IP via Call Session Control Functions (CSCFs) which use the Session Initiation Protocol (SIP) [4] for call setup signaling. During the setup of a VoLTE call, P-GW acts as the interface between the EPC and IMS networks. The P-GW typically includes the control function commonly known as the Policy and Charging Enforcement Function (PCEF), which forwards signaling packets to the Proxy-CSCF (P-CSCF) using the SGi interface. The S-CSCF after decoding the SIP message, triggers network bandwidth allocation at the PCEF. The communication between the S-CSCF and the PCEF is mediated by the PCRF which communicates the the S-CSCF and the PCEF using Diameter-based [2] Rx and Gx interface, respectively.

It is important to note that IMS is an independent subsystem and therefore it does not use EPC user identifiers such as the IMSI and MSISDN for identification and authentication of the callers. Instead, IMS uses SIP protocol based IP Multimedia Private Identity (IMPI) and IP Multimedia Public Identity (IMPU) for user addressing and authentication. Usage of different identifiers the EPC and IMS subsystem make the VoLTE network susceptible to caller-ID spoofing attacks described in chapter 6.1.

2.2 Introduction to Network Functions Virtualization

Network Functions Virtualization (NFV) is the decoupling of Network Functions (NFs) from proprietary hardware appliances and running them as software applications using virtualized technologies such as Virtual machines (VM) or containers. Network functions such as firewalls, load balancers, and EPC elements (Chapter 2.1) deployed using NFV (referred to as Virtual Network Functions (VNFs)) can then be instantiated on commercial off-the-shelf (COTS) such as x86 servers.

Network services such as the VoLTE service in Chapter 2.1 are delivered using a combination of network functions. A Service Function Chain (SFC) defines an ordered set of abstract NFs and ordering constraints that must be applied to packets and/or frames in the process of service delivery. As an example, VoLTE call setup request must be processed by the P-GW, P-CSCF, S-CSCF, and the PCRF (Figure 2.1), in that order, before call setup can be completed. Network orchestrators dynamically instantiate VNFs involved in delivery of a service and then use programmable network technologies such as Software Defined Networking (SDN) to switch together VNFs into SFCs.

The benefit of NFV is twofold: reduced capital/operational expenses and increased network agility. Unlike traditional deployments where network functions and services are deployed using purpose build hardware, network services in NFV are deployed on COTS servers rented from public cloud platforms such as Amzaon EC2 [5], Google cloud [6], etc which allows service providers to reduce the capital expense required to deploy new services. Additionally, since resources in NFV not pre-allocated, service providers can allocate additional resources on demand, further reducing operational expenses. On demand resources can be added by a) Scale-up: allocating additional resources to existing VNF instances, or b) Scale-out: creating new instances of VNFs. New service offerings can now be created by adding one or more VNFs to an SFC and an existing services can be extended by adding software modules, whereas unpopular features can be removed by deleting service modules.

A new trend in the NFV is the use of microservice based architectures. Unlike traditional deployments and VM-based NFV deployments where cellular services were provided by complex monolithic applications, microservice-based designs advocate the use of small, loosely-coupled, Representational state transfer (REST)-based reusable components for service delivery. Each microservice component provides a single functionality (such as key-value store, timer service, etc) and communicates to other microservices using well defined REST-based messages. Consequently, microservice-based architectures facilitate independent scaling and update of individual microservices, avoiding long development cycles and improving elasticity. Figure 2.2 shows the difference between an application deployed using monolithic architecture and microservice architecture.

Despite their advantages, NFV and microservices represent a significant departure from the traditional cellular deployments and have broad implications ranging from software design to orchestration. Unlike traditional deployments where proprietary hardware and cus-

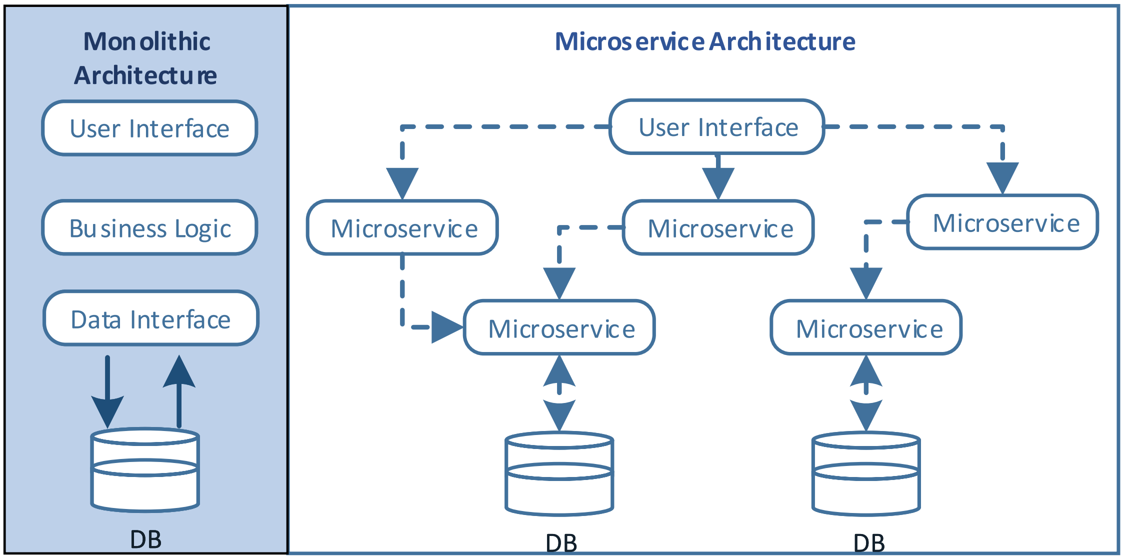


Figure 2.2. Difference in Monolithic and Microservice architecture

tom build low latency data centers were used to guarantee QoS, supporting low latency services is becoming more challenging. Functional decomposition of monolithic applications into smaller microservices is resulting in increasingly longer SFCs and the latency budgets available to individual microservices are shrinking. Deployment of latency-sensitive services while achieving conflicting goals of maximizing infrastructure usage and avoiding SLA violation(s) therefore requires traffic-aware orchestration frameworks which can make e2e placement and resource allocation decisions.

2.3 Related Work

In this section, we review the literature associated with the adoption of cloud native designs in cellular cores. Specifically, we discuss the choice of virtualization technology, research related to developing low latency cellular core, and automated placement solutions. Further, we present a brief overview of existing caller-ID validation solutions, to put in context the need for a network-assisted caller-ID validation solution and the advantages of container-driven deployments in supporting such functional extensions in the cellular core.

Virtualization Performance: Network Function Virtualization [7] has gained significant momentum over the past few years [8]. Gember *et al.* [9] investigated application deployment in the cloud from various perspectives, including elasticity, network flow distribution, and virtual machine placement. Banerjee *et al.* [10] discussed how NFV can help scale EPC systems. Elasticity in intrusion detection systems has been investigated in [11], [12]. None of the above studies specifically compares containers to virtual machines.

More recently, the use of containers has been studied by Anderson *et al.* [13] and Kamarainen *et al.* [14]. Anderson *et al.* [13] examined the impact of network technologies like OVS, bridging and macvlan on the throughput of Docker containers. Kamarainen *et al.* [14] explored the impact of virtualization techniques including containers and VMs on cloud gaming systems. Their study examines the impact of virtualization on video encoding and hardware resource sharing and does not consider transaction-based systems and network layer entities as we do.

Service Decomposition: Placement problems and network latency have been widely studied in the context of NFV. A generally accepted direction for scalable cloud-based infrastructure is decoupling user state storage from VNF processing logic. Kablan *et al.* [15] investigate a stateless design that leverages technologies like RAMCloud over InfiniBand to demonstrate how a Network Address Translation (NAT) function can be decomposed into packet processing and data. The focus of their work is, however, on demonstrating how a stateless design improves the elasticity of NFV deployments. They do not consider the impact of stateless design principles on limiting e2e latency. Decoupled control and user plane network functions are also described in [16]. This work aims at minimizing communication latency between the control plane elements while simultaneously bringing the user data processing elements close to the network edge, improving the overall user experience.

Basta *et al.* [17], Hawilo *et al.* [18] and Katsalis *et al.* [19] propose redesign of existing networks to reduce latency. There is, however, little work that uses container-driven backward-compatible solutions. Basta *et al.* [17] explore several implementation models in which v-EPC can be deployed. However, the implementation models proposed can result in extensive refactoring of existing implementations. The scope of their work is limited to the placement of the user and control planes in EPC gateways, and does not include common principles that can be applied to any SFCs. Hawilo *et al.* [18] discuss a scheme for bundling EPC components, guided by the principles of a flat architecture and the decoupling of the control and user planes. While this architecture proposes bundling an NF and provides an analysis of the benefits of the proposed architecture, it does not investigate how these changes can be implemented in current architectures. The work closest to ours was conducted by Katsalis *et al.* [19]. This work analyzes a stateless 5G design pattern. They propose a micro service-driven stateless RAN architecture which uses shared control plane contexts for data storage. The work is limited to the analysis of RAN and does not delve into the application of this design to general SFCs.

Service placement and monitoring: Functionality-based decomposition has been proposed to reduce latency and increase throughput for cellular network control planes [19]–[22]. That work uses manual analysis of network architecture and traffic to find the functional elements that can be aggregated. Stratos [23] avoids traversing oversubscribed inter-rack links during function placement, and Selimi et al. [24] explore placement to maximize bandwidth utilization. None of these studies consider workload types and transactions. Other work [25]–[27] formulates the placement problem as a graph partitioning problem or an optimization problem. This is orthogonal to our work, as our notion of transactional affinity is a new factor to consider as an input to the placement problem.

Recent studies [28]–[32] have highlighted challenges in integrating, deploying, and managing microservice-based applications. For instance, `ucheck` [28] uses runtime verification and enforcement of invariants to help service providers manage microservice-based applications. Probius [29] finds performance bottlenecks by correlating VNF, hypervisor, and system metrics. NFVPerf [30] uses network traces collected from NFs to compute per-hop message processing latency which is then used to infer performance bottlenecks. Our work proposes transaction-driven microservice deployment, and is thus complementary to this line of work.

Protocol inference and traffic enrichment: Extracting protocol state information from network traces, or *reverse engineering* a protocol, has been widely studied in the literature. Prior work extracts specifications of unknown protocols [33]–[37], and uses inferred message formats to detect malware signatures [34], [38]. *Invenio* also exploits protocol header information, but utilizes protocol analyzer tools to extract user-identifying headers. Other categories of work in this area use xml or json formats exported by tools such as wireshark to derive protocol state machines [39], [40]. Application-specific information is used to group available messages into sessions [39]. Network traces identify dialogs in HTTP and SIP traffic [40]. While *Invenio* shares finite-state machine extraction techniques with these papers, it differs in one important aspect: we use the extracted state information to compute affinity between NFs for an entire SFC. In contrast, prior work extracts the state machine for a single NF and does not merge state machines from multiple NFs to derive transaction information for an SFC.

Several efforts [41], [42] enrich network messages with “Metadata” [42] using the Network Service Header (NSH [41]) which can be used by NFs to steer traffic and maintain per-user state. *Invenio* can benefit from the presence of the NSH header in network messages. Metadata carried in these headers can be utilized to identify the user associated with a message.

Caller-ID validation: Several solutions for caller-ID validation have been proposed in the literature. These solutions can be categorized as endpoint-only or network-assisted. Some endpoint-only solutions [43], [44] use challenge-and-response between the caller and callee, which requires the caller to respond to an SMS [44] or a call [43]. This requires the caller’s cooperation, and mandates updates on all phones (i.e., all possible callers), which is unlikely in the foreseen future. Most network-assisted solutions either deploy an additional global authority (e.g., a public certification service [45]–[48]) or a Public Key Infrastructure (PKI) [49] to authenticate each party before call setup. An easier-to-deploy approach is to authenticate callers at the gateway during call setup [50], [51] by cross validating the forwarded ID with the authenticated one. This approach is effective in principle but has not been deployed in practice, partly because all existing solutions would incur an unacceptable performance penalty. Our work adopts this general approach but designs a practical solution compatible with current infrastructure at a much lower overhead.

3. A CASE FOR CONTAINER-DRIVEN FINE-GRAINED VNF RESOURCE FLEXING

In this chapter, we make a case for using lightweight containers for fine-grained resource flexing for Virtual Network Functions (VNFs) to meet the demands of varying workloads. We quantitatively compare the VNF performance and infrastructure resource usage of three different instantiations (bare metal, virtual machine, and container) of three selected VNFs. The three VNFs we use for our experiments are the Mobility Management Entity (MME) of the Evolved packet core (EPC) architecture for cellular networks, the Suricata multi-threaded Intrusion Detection System (IDS), and the Snort single-threaded IDS. Our results show that container-based instantiations not only incur low resource usage but also have shorter boot time. In order to understand the efficacy of containers in supporting real-world application with long SFCs, we also evaluate a container based Voice over LTE (VoLTE) application. Our results indicate that NFV based VoLTE deployments significantly outperforms existing bare metal based service deployments, making containers an ideal choice for real-world NFV deployments.

3.1 Introduction

Communication Service Providers (CSPs) are increasingly adopting Network Functions Virtualtization (NFV) in their infrastructure. Two primary factors driving NFV adoption are efficient resource usage, and agility in terms of elastic resource allocation [7]. However, CSPs face key challenges in this NFV transformation. While virtualization allows service and resource allocation agility, virtualization of network functions needs to be implemented with minimal overhead for efficient resource usage. *Virtual machines* (VMs) and *containers* are the two most widely deployed virtualization mechanisms in the cloud. Containers such as LXC [52] and Docker [53] are becoming popular for tenant and application isolation in cloud ecosystems. Compared to virtual machines, containers exhibit lower overhead and higher performance.

In this chapter, we compare the performance and resource usage of three Virtual Network Functions (VNFs) with bare metal (BM), container, and Virtual Machine (VM) instantiations, at a variety of load levels and resource allocation configurations. The three VNFs we benchmark are: (1) The Mobility Management Entity (MME) of the Evolved packet core (EPC) architecture for cellular networks, (2) the Suricata multi-threaded Intrusion Detection System (IDS), and (3) the Snort single-threaded IDS. To the best of our knowledge, ours is the first extensive empirical study that compares resource usage efficiency and elastic resource flexing of VMs and containers for VNF implementation.

We demonstrate that container-based deployments incur significantly lower performance overhead, compared to VM-based deployments, while reducing the initialization time of the system. This makes containers an ideal choice for systems where booting delays directly affect how quickly additional resources can be allocated to VNFs (agility), and higher elasticity is desired to reduce the overall operational cost. We experiment with deploying multiple instances on the same hardware platform, and find that such an elastic deployment model provides high resource utilization without the need for re-architecting unoptimized VNF implementations such as single-threaded applications.

Thus, the contribution of this chapter is threefold:

1. We benchmark the performance of the Mobility Management Entity, Suricata, and Snort VNFs on bare metal, containers and VMs under varying workloads and with different numbers of instances.
2. We analyze the time required by the VNFs to start in VMs and containers.
3. Based on our results, we make several recommendations for resource flexing of VNFs, including VNFs based on legacy single-threaded software.

3.2 Virtual Network Functions

We consider two types of VNFs for our empirical study: (1) an evolved packet core (EPC) control plane VNF, MME, that is primarily a control plane network function, and (2) two

IDS VNFs (Snort and Suricata) that are primarily data plane network functions and stress the forwarding plane of the VNF ecosystem.

3.2.1 EPC Control Plane

We evaluate the performance of the Mobility Management Entity (MME) from the EPC control plane. Figure 2.1 shows the position of the MME in the LTE network. MME is a control plane entity which acts as the inital attach point to the LTE network and supports user attach and authentication procedure. We use MME in our evaluations to understand the impact of virtualization on control plane elements.

3.2.2 Intrusion Detection Systems

Suricata http://suricata-ids.org/ and Snort https://www.snort.org are popular intrusion detection systems. The reason we experiment with both of them is that Snort is single-threaded whereas Suricata is multi-threaded. This difference allows us to evaluate the impact of the implementation on total system throughput when NFV orchestration systems scale-out by creating additional instances of the same VNF.

3.3 Experimental Setup

The testbed we use in our experiments includes two host machines that are directly connected by Ethernet. One host acts as the sender running a traffic generator, and the other as a receiver that runs our VNF. Both machines have the following hardware configuration: **CPU** Intel Xeon X3430 @ 2.40 GHz; Nehalem; EIST disabled; VT-x and VT-d enabled; HT not supported, **RAM** 2 x 2GB DDR3-1333, **HDD** 500GB Seagate 3.5" 7200RPM + 2 x 1 TB Seagate 3.5" 7200RPM, and **NIC** 2 x Broadcom 1 Gbps.

The sender host runs Ubuntu 14.04.4 64-bit LTS with **gcc** 4.8.4. The IDS experiments use **tcpreplay** 4.1.1. The receiver host runs Ubuntu 15.10 64-bit with the following software: **gcc** 5.2.1, **docker** 1.11.0, **libvirt** 1.2.16, and **qemu** 2.3.

3.3.1 Evolved Packet Core Setup

In the MME experiments, we run the MME, SGW and PGW nodes on physical machines, VMs, or Docker containers. The HSS and eNode applications are hosted on two physical machines connected via a switch (Figure 3.1).

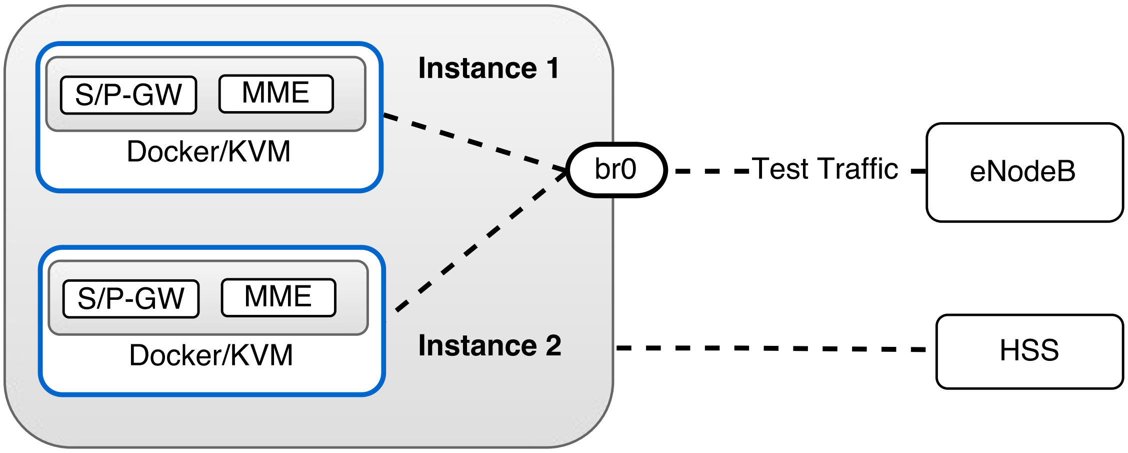


Figure 3.1. Setup for EPC Tests

The EPC experiments use the following setups: (1) **Bare metal (BM):** The MME runs directly on the hardware and reads the incoming data directly from the NIC. (2) **Container (Docker):** In case of multiple instances of the MME running within Docker, a bridge is used to direct traffic from the NIC to the appropriate MME container instance. (3) **Virtual Machine (VM):** We use QEMU/KVM [54] hypervisor to run VM instantiation of MME. Similar to the container case, when there are multiple instances of MME, a bridge is used to direct traffic from the NIC to appropriate MME instance. The VMs run the same version of the operating system and software as their bare metal counterpart to ensure a consistent runtime environment.

In all MME experiments, the memory limit it set as 2 GB for VM and containers. When multiple VM instances are used, the CPU cores are proportionality divided among the MME instances.

3.3.2 IDS Setup

In the IDS experiments, the sender uses multiple instances of TCP-replay to generate test traffic by replaying trace files (Figure 3.2).

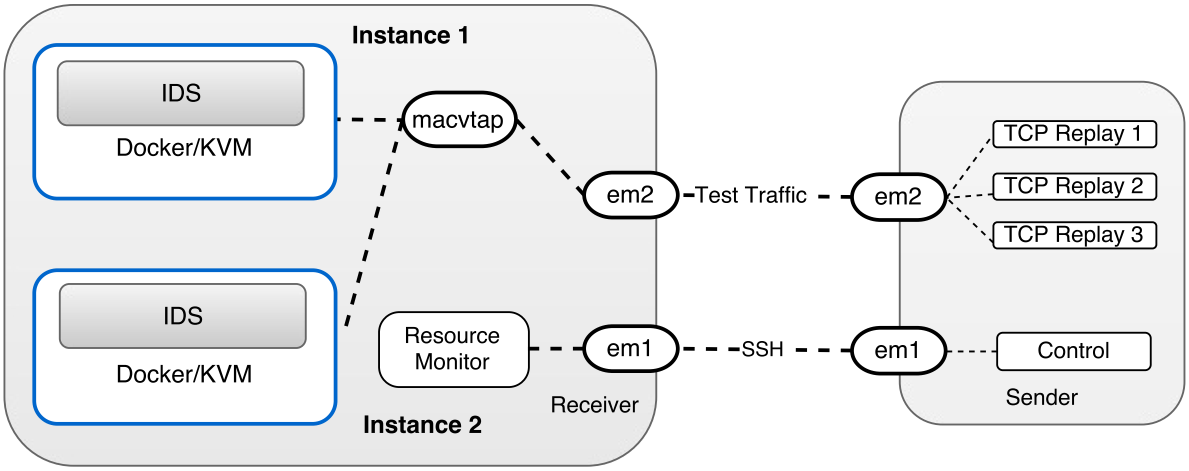


Figure 3.2. Setup for IDS Tests

We use **suricata** 3.0.1 or **snort** 2.9.8.2 with **EmergingThreat Rules** 20160414. The docker and VM image for Suricata and Snort resemble the software of the host, running Ubuntu Server 15.10 64-bit and having the same gcc and Suricata/Snort (including configuration and rule set) versions.

The IDS experiments use setups similar to the MME experiments described earlier. Since IDS VMs are primarily data-plane applications we also evaluated how their performance is impacted by various virtualized network technologies being deployed by NFV users. In particular, we experimented with three mechanisms for directing incoming IDS workload to network function instances: (1) **Direct NIC access:** Direct access to the NIC is available for bare metal and container instantiations. This direct access to NIC is not available for the VM instantiation. With multiple instances, additional de-muxing support is required for redirecting traffic appropriately. (2) **macvtap:** Instead of exposing host NICs to containers as above, we create a macvtap device. After experimenting with different means of forwarding traffic, including bridging and Virtual Ethernet Port Aggregator (VEPA), we found that macvtap of mode “passthrough” (requiring VT-d and SR-IOV) and model “virtio” incurs the least overhead, and therefore we use it for our IDS experiments. This is consistent with the findings of Anderson *et al* [13] for macvlan. We will use "DockerV" to refer to this setup. (3) **Linux Bridge:** In this setup, a Linux bridge is used to direct traffic from the host NIC to the VM or container. This setup is only used when multiple IDS instances run on the same host.

3.4 MME Experiments

3.4.1 Methodology

We use openair-cn [55] to benchmark the performance of the EPC control plane. Openair-cn is a 3GPP-compliant implementation of EPC components including MME, HSS, S-GW and P-GW that can be executed on general purpose hardware. Our test setup includes a MME node connected to the HSS and a client emulator framework based upon the openair-interface oaisim [56] application. MME is co-located with the S-GW, and all communication between them is handled via internal queues. The oaisim application connects to the MME over the S1 interface, and generates attach requests emulating multiple UEs at a constant rate per second.

The eNodeB emulator is used to generate registration requests which result in the exchange of several control plane messages between eNodeB, MME, HSS and the S-GW. During the experiments, we measure the time taken by the MME to successfully process UE registration requests. The experiments are run multiple times to obtain a number of samples ($>= 20$), and the average run time of all samples is used for comparison.

The emulator generates registration requests at a constant rate of 400 registration requests per second and the total time taken by MME to respond to a total of 4000, 6000 and 8000 registration requests is measured. We measure the performance of the system with 1, 2 and 4 CPU cores enabled. Since the MME implementation handles most of the processing in a single thread, enabling multiple cores allows us to analyze the scale-out capability provided by VM and container deployments.

3.4.2 Results

We analyze the time taken by each setup to handle the registration requests. The results in Figure 3.3 show that VMs incur significantly higher overhead than the bare metal setup and Docker containers. The VM can result in 10-21% overhead, whereas the overhead of Docker is 0-3%. The traffic generation rate for these experiments is 400 registrations/second. It takes 10, 15 and 20 seconds to generate the required 4000, 6000 and 8000 requests, respectively. We note that time taken by the MME to handle these sessions increases as the number of concurrent sessions handled increase. More sessions timeout when the system load is higher.

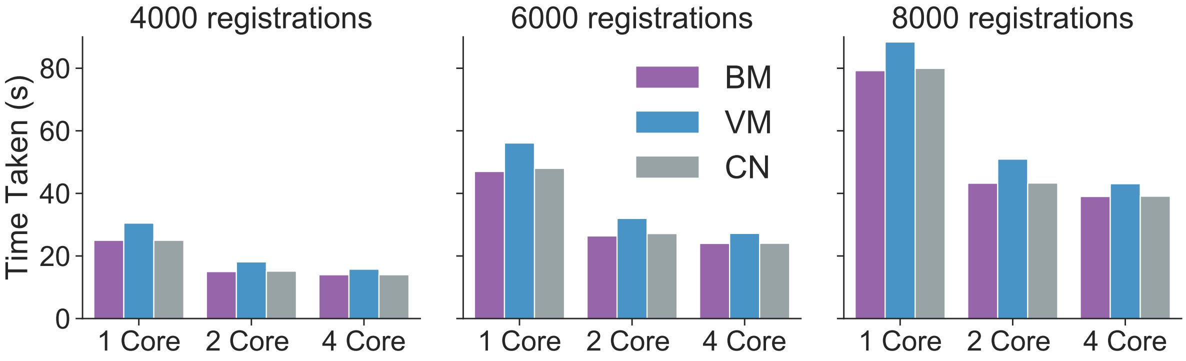


Figure 3.3. Time taken to handle registration requests.

As MME is a transaction-based system and stores all the active sessions in memory, the maintenance overhead increases with number of active sessions. This includes indexing time required to fetch and store the session related data in the internal data-structures, and timers maintained to handle events like timeout and heartbeats. The current implementation of MME uses a hash-based indexing mechanism to store the UE information. The likelihood of collisions and chaining increases when the number of active sessions being handled by an instance increases.

As noted earlier, the MME is co-located with the SGW and PGW. Consequently, when only a single CPU core is available, performance of the system is constrained by the available processing power. When 2 cores are available to the system, performance of the MME significantly increases as the MME application processing thread utilizes one of the CPU cores and other features can use the second available core. However, this performance benefit is not observed when 4 cores are available, augmenting the claim that the system is single-threaded. This implementation limits the ability of MME application to utilize all the available cores in the system as only a single instance of the application can be instantiated. However, VM and container based systems can allow creation of multiple instances of the MME application to allow better resource utilization. To analyze this capability, we replace the single instance of the VM and container based MME application with two instances that share the available resources.

In this test setup, we generate the same number of registration requests as the earlier setup, but the number of registration requests handled by each instance is halved. Consequently, the number of active sessions handled at each instance is reduced to 2000, 3000 and 4000. We also double the request generation rate from the client to 800 registrations/second so that each MME instance receives the requests at 400 registrations/second. Both instances of the MME are configured at the eNodeB, and the emulator sends the request to each MME in a round-robin fashion. To establish a baseline, we first show the results when a single instance of MME handles traffic at the rate of 800 registrations/second on a bare metal machine. These results are presented in Figure 3.4. We find that increasing the traffic generation rate does not have a significant impact on the performance of the MME. This is because the bottleneck when using a single instance of the system is the MME application thread and not the transport receive thread.

Figure 3.5 shows that the time taken to handle the registration requests is considerably reduced when traffic is split across two instances with similar processing resources. Additionally, as the number of CPU cores increase from 2 to 4, we find that the time taken by the MME to handle registration requests decreases which indicates better utilization of available processing resources, compared to the case when only a single instance of the MME was used. While these results reflect the behavior of our MME implementation and may not be directly applicable to other commercial deployments, they can be used to infer the benefits of the microservices architecture in transaction-based systems.

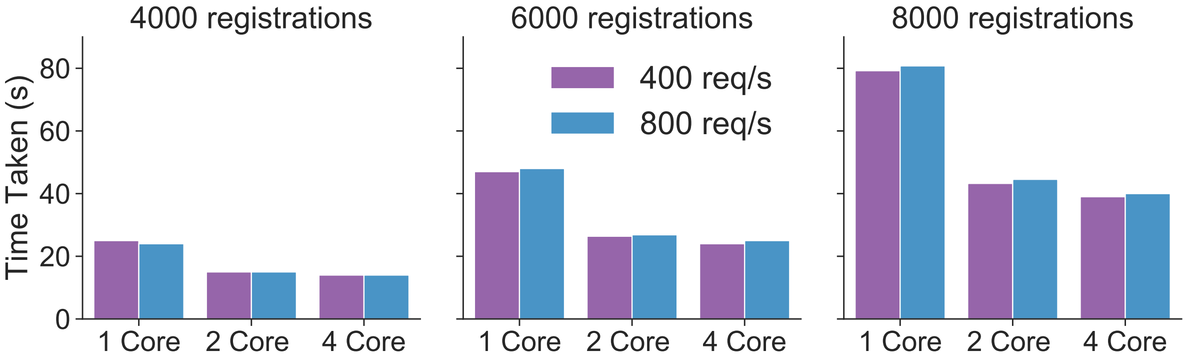


Figure 3.4. Time taken to handle registration requests on bare metal with varying the request rate.

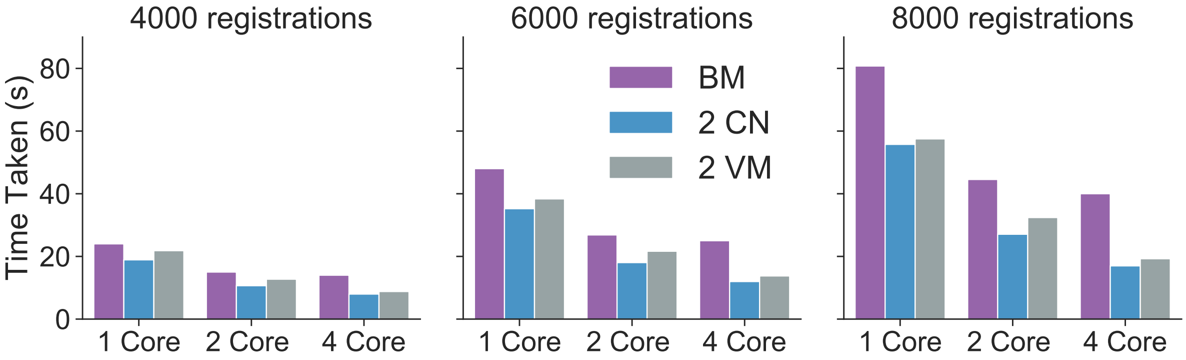


Figure 3.5. Time taken to handle registration requests by bare metal and two instances of VMs or containers.

We also study the time required for the MME to become operational. In cases where virtual implementations are used to maximize the resource utilization, it should be possible to provision a new instance of the MME without incurring significant delay. The boot time of MME is the time taken to initialize internal data structure, create transport connections and time taken to establish the diameter application-level connection. In the current implementation, MME and HSS must exchange a diameter capability exchange request (CER) and capability exchange response (CEA) before the MME can begin accepting connections from the eNodeB.

Table 3.1. Comparison of Activation time.

Metric	Bare Metal	Docker	VM
Time(s)	4.77	4.81	12.01

As shown in Table 3.1, the time taken by the Docker container to start is much closer to the time taken by the bare metal, whereas the time taken by VM is considerably higher, due to the overhead involved in loading and booting the guest-OS kernel and hypervisor.

3.5 Suricata Experiments

3.5.1 Methodology

We benchmark Suricata based on statistics it reports and resource usage of the entire host machine. The reason why we examine resource usage of the entire system is that both Docker and QEMU have overhead not reported by their API. For example, Docker's `stat` API only reports resource usage inside the container and excludes Docker itself. The CPU and RAM overhead of forwarding traffic is not included in the Suricata, Docker, or QEMU processes. Therefore, comparing resource usage of the entire system is more comprehensive. **Trace Files:** We use two trace files: (1) `bigFlows.pcap` provided by TCPreplay. According to the TCPreplay site, it captures "real network traffic on a busy private network's access point to the Internet" and contains 40686 flows and 132 network protocols [57]. It sends 359,457 KB of data in 791,615 packets in 5 minutes, and (2) `snort.log.1425823194` – one of the ISTS '12 trace files [58]. It generates 155,823 KB of data in 22 seconds. **Increasing Load:** To increase load, more TCPreplay processes may run in parallel. We will use "load level" to denote how many concurrent processes are used. For example, "4X load" means that a test has four TCPreplay processes concurrently replaying the trace file. **Aggregating the Results:** We run each test configuration multiple times to obtain a number of samples (>= 30). We then take the median to generate a representation of the test. Before generating a sample, the receiver host is rebooted to restore system state back to the original. We measure host memory and CPU usage, and the number of packets captured, analyzed (decoded), and dropped by the IDS. We do not examine the number of alerts triggered because it is highly affected by the packet drops.

3.5.2 Results

We first analyze resource usage by comparing memory and CPU utilization of the Suricata host running different setups at various load levels, then compare the performance of Suricata in different setups.

Unless otherwise noted, the memory limit is set to 2 GB for Docker, DockerV, and VM setups, and all 4 CPU cores are accessible.

Memory Usage: Our results indicate that the memory overhead of Docker is trivial compared to bare metal, while VM setup consumes substantially more memory. Table 3.2 shows the memory usage (average and standard deviation σ) of Suricata in VM, Docker and bare metal setups at 1X workload. Docker has a small memory footprint since the host and Docker container have shared libraries of the same version, eliminating the need to load more libraries. Although the VM runs the same software setup, a full-fledged guest operating system must be maintained, which results in high memory overhead.

Table 3.2. Memory Usage of Suricata at 1X workload

Metric	Bare Metal	Docker	DockerV	VM
Average	9.85	10.20	10.22	23.83
σ	0.33	0.33	0.34	2.29

We also tested Docker, DockerV, and VM with 4X `bigFlows.pcap` and a smaller 512 MB memory limit. While the first two worked without problems, memory thrashing occurred to the Suricata VM before CPU became the bottleneck.

CPU Usage: Table 3.3 shows the CPU usage of Suricata with Docker, VM and bare metal with 1X workload. As seen from Table 3.3, Docker does not impose significant CPU overhead, while the CPU usage of the VM setup is considerably higher compared to bare metal.

Comparing Table 3.3 to Table 3.4, we find that the CPU usage of the bare metal, Docker and DockerV setups multiplies corresponding to load level, but the VM setup saturates the

Table 3.3. CPU Usage of Suricata at 1X workload.

Metric	Bare Metal	Docker	DockerV	VM
Average	19.65	20.92	21.81	269.39
σ	4.61	4.66	4.90	49.66

Table 3.4. CPU Usage of Suricata at 2X workload.

Metric	Bare Metal	Docker	DockerV	VM
Average	40.75	42.11	44.00	399.90
σ	9.87	9.90	10.44	0.70

CPU at 2X workload. This results in the CPU becoming a bottleneck at workloads greater than 2X. We also observe that the Docker setup incurs an overhead of 1% to 4% depending upon the load level. There is a roughly 0% to 5% increase in CPU usage associated with macvtap depending on traffic throughput.

Packets Received: In all four load levels we use, packet capture is about the same for all four setups. However, the VM setup tends to receive less packets in the end. After the TCPreplay ends, we send SIGTERM signal to Suricata and wait for it to exit gracefully. It is likely that for VM setup it still has packets yet to capture when exiting, resulting in the discrepancy exhibited.

Packets Dropped: Dropping packets is a sign that Suricata cannot process the workload given the resource constraints imposed. As seen from Figure 3.6, only VM setup observes packet dropping starting from 2X load, and almost all increased load above 2X is dropped. This result is consistent with the CPU usage information in Table 3.4. CPU is nearly saturated by Suricata at 2X load level and therefore almost all additional traffic generated by 4X load is dropped.

Packets Decoded: Comparing the numbers of packets decoded in Figure 3.6, we find that Docker and DockerV setups are on par with bare metal setup in terms of speed of packet analysis (decoding). However, only at 1X load when CPU has not become a bottleneck, is speed of packet decoding in VM setup comparable with bare metal, and the speed decreases as load increases, which reveals severe performance degradation.

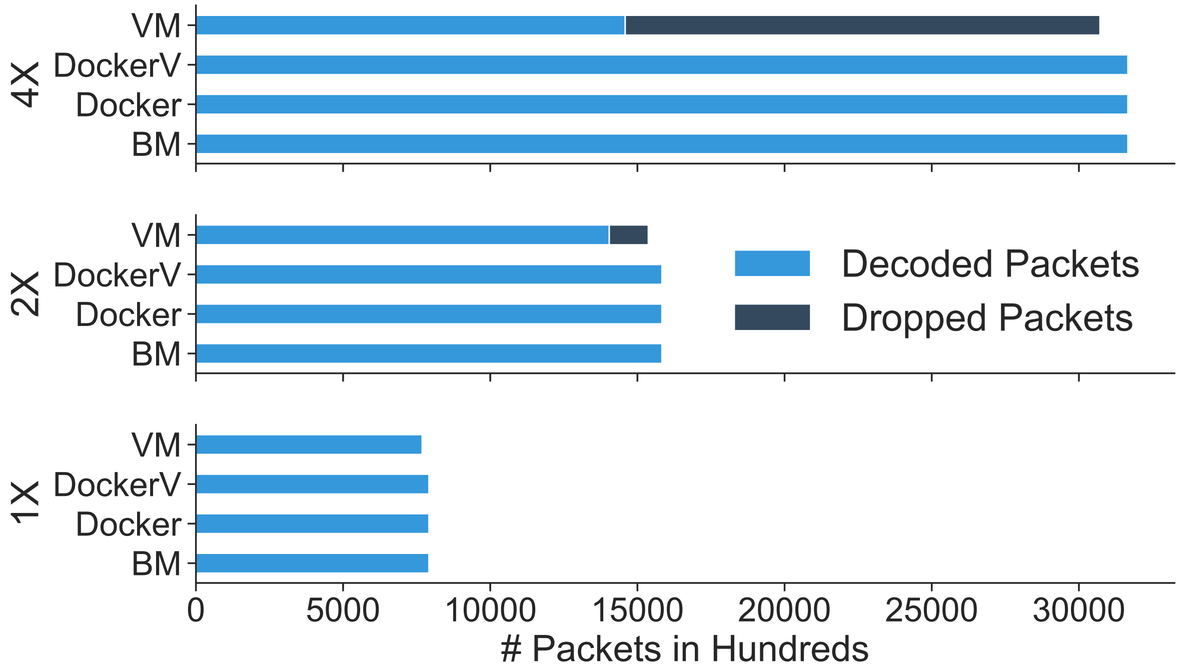


Figure 3.6. Cumulative packets decoded and dropped by Suricata in four setups at all loads.

3.5.3 Results from the Second Trace File

The trace file, `snort.log.1425823194`, requires the receiver host to use more CPU to handle the high-throughput traffic. Similar results were observed, but we saw bare metal saturated at 4X load. At 4X load, Suricata drops 67,126 packets in bare metal, 67,064 packets in Docker, 81,282 packets in Docker with macvtap, and 282,453 packets in VM. In fact, Suricata in VM runs so slowly that even the kernel drops 205,095 packets to reclaim buffer space before Suricata is able to read them. This confirms that (1) Docker has comparable performance with bare metal, (2) the overhead of macvtap can be nontrivial, and (3) the VM setup runs more slowly than others.

3.5.4 Root Cause Analysis for VM Results

To determine what caused the dramatic performance degradation, we profiled Suricata running in bare metal and in VM. The culprit we found is the frequently called function 'UtilCpuGetTicks()' which flushes the instruction pipeline and reads the x86 Timestamp Counter (TSC). The x86 instruction 'rdtsc' may cause VM exit [59], and by checking the msr register we verified that it indeed causes VM exit in our VM setup. This makes the instruction expensive in the VM, and results in a massive performance penalty. To further isolate the cause, we ran this sole function 100 million times in bare metal and in VM, and it takes 6.6 seconds (on average) to finish in bare metal, but 126.1 seconds in VM.

3.6 Snort Experiments

For the Snort experiments, we use the same setup as Suricata except that Suricata is replaced by Snort. Again, we ran each test configuration at least 30 times, and used the median of each metric to form a representation of the result. We discuss the outcome at 4X load with `bigFlows.pcap`.

Snort exports statistics only on exit. Although a stop signal was sent to Snort process 20 seconds after TCPreplay finished, Snort did not stop immediately after receiving the signal.

Figure 3.7 confirms that Snort works best in bare metal, followed by Docker. Surprisingly, DockerV and VM give similar results. There are two factors to take into account: (1) Snort ran slightly longer than DockerV after receiving the exit signal, and (2) the total number of packets to receive is supposed to be 3,166,460 – similar to what happened to Suricata in VM at 4X load, Snort in VM failed to capture all packets. Although Snort is single-threaded, the VM overhead caused the VM setup to use more than 100% CPU combined (Table 3.5).

Table 3.5. CPU Usage of Snort at 4X workload

Metric	Bare Metal	Docker	DockerV	VM
Average	73.62	75.17	79.42	137.23
σ	13.28	13.80	14.02	18.48

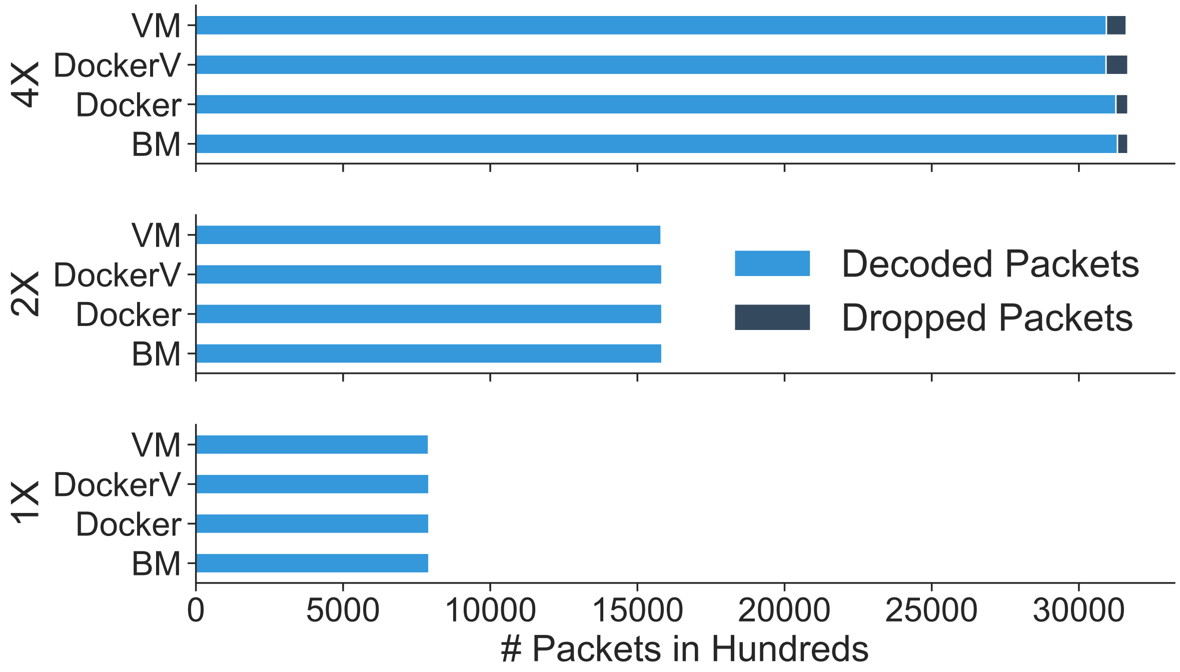


Figure 3.7. Cumulative packets decoded and dropped by Snort in four setups at all loads.

In terms of memory usage, the VM setup uses more than twice the memory of the Docker setup. Docker and DockerV setups use slightly more RAM (∼1%, or ∼40 MB) than that of bare metal (Table 3.6).

Table 3.6. Memory Usage of Snort at 4X workload

Metric	Bare Metal	Docker	DockerV	VM
Average	12.84	13.25	13.26	27.45
σ	0.40	0.32	0.33	0.36

3.7 Suricata and Snort with Multiple Instances

We now investigate Suricata and Snort when multiple instances of these IDSes are deployed within the same host. Based upon our findings in previous experiments, we know that Snort does not effectively utilize all available CPU cores in the system due to its single-threaded design. While this design of Snort limits its scalability on the bare metal, we can deploy multiple instances of containers and VMs to utilize the available CPU cores more effectively.

This setup uses a Linux bridge to transfer the incoming traffic to the VM and container instances. Since both Snort and Suricata bare metal setups handle the 4X workload without significant packet drops, we use 4X and 8X workload to evaluate the scalability of these systems with multiple instances. The values presented in this section represent the median value of (>=5) runs of each experiment.

In Figure 3.8 and 3.9, CN is used to refer to a Docker setup with a Linux bridge, and VM is used to indicate a VM setup with a Linux bridge. In case of multiple instances of containers or VMs, the value presented is the sum of the number of packets processed by each instance independently. 2-CN and 4-CN are used to indicate the number of packets processed by two and four container instances, respectively. 2-VM represents the number of packets processed by the two VM instances collectively.

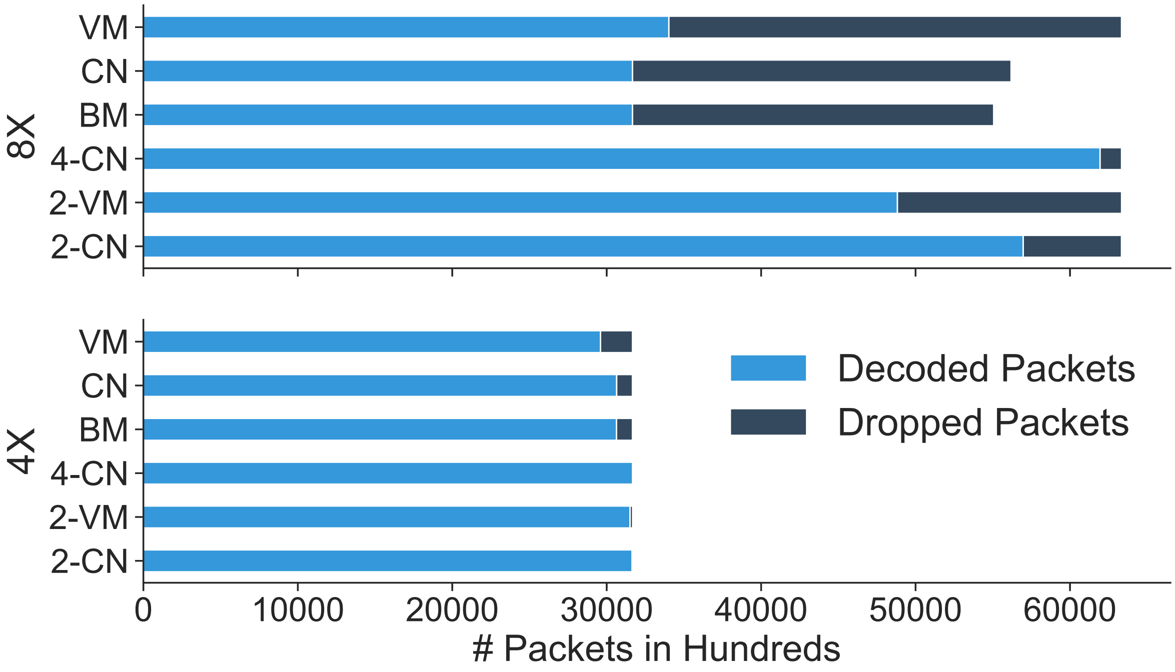


Figure 3.8. Performance of Snort with multiple VM and containers instances.

Figure 3.8 shows that the performance of Snort improves significantly when the incoming traffic is split between multiple instances. The performance difference is more pronounced at 8X workload as a single Snort instance is unable to handle the incoming traffic. We find that the number of packets dropped significantly decreases when two container instances are deployed and continues to decrease with four instances. This behavior is consistent with our earlier finding when multiple instances of MME were deployed on the same host. Further, we note that while two VM instances provide significantly higher performance compared to a single instance of VM and bare metal, the performance benefit is not on par with a multiple container deployment.

We also analyze the performance of Suricata by deploying multiple instances of VMs or containers on the same host machine. As noted earlier, Suricata is multi-threaded and is capable of utilizing all available CPU cores even with a single instance deployment. Table 3.4 showed that Suricata saturates the available CPU cores at 2X traffic when deployed as a single instance.

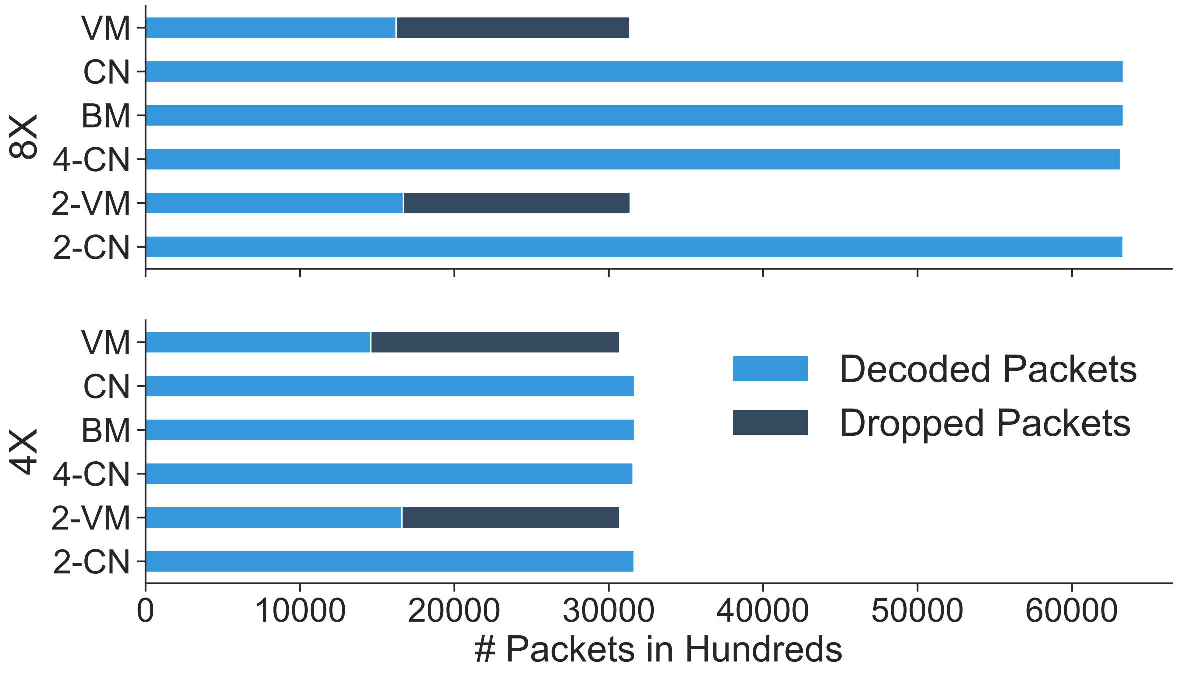


Figure 3.9. Performance of Suricata with multiple VM and container instances.

From Figure 3.9, we find that there is no observable performance difference between the single and multiple instances deployment. While we observe some performance gain (∼1%) in the VM setup when two VM instances are deployed, we find that the factor limiting system performance is the available CPU and it cannot be circumvented by deploying multiple instances.

The observations from both the MME and Snort experiments validate the efficacy of a container-based microservice architecture for legacy software that is architecturally constrained from providing scalability on modern hardware platforms.

3.8 Chapter Summary

In this chapter we empirically compared the performance of various control and data plane NFs on bare metal, VM and container based deployments. Our resuls show that containerized instantiations of both control-plane (MME) as well as data-plane (Snort and Suricata) VNFs consumed less CPU and memory resources in comparison to traditional VM-based deployments. Smaller resource footprint and low performance overheads of containerized VNFs coupled with their smaller instantiation time, make containers an ideal choice for the deployment of next generation cloud-native architecture required to support low latency cellular services without compromising SLAs.

4. CONTAIN-ED: AN NFV MICROSERVICE SYSTEM FOR CONTAINING E2E LATENCY

Network Functions Virtualization (NFV) has enabled operators to dynamically place and allocate resources for network services to match workload requirements. However, unbounded end-to-end (e2e) latency of Service Function Chains (SFCs) resulting from distributed Virtualized Network Function (VNF) deployments can severely degrade performance. In particular, SFC instantiations with inter-data center links can incur high e2e latencies and Service Level Agreement (SLA) violations. These latencies can trigger timeouts and protocol errors with latency-sensitive operations.

Traditional solutions to reduce e2e latency involve physical deployment of service elements in close proximity. These solutions are, however, no longer viable in the NFV era. In this chapter, we present our solution that bounds the e2e latency in SFCs and inter-VNF control message exchanges by creating *microservice aggregates* based on the affinity between VNFs. Our system, *C*ontain-ed, dynamically creates and manages affinity aggregates using light-weight virtualization technologies like containers, allowing them to be placed in close proximity and hence bounding the e2e latency. We have applied Contain-ed to the Clearwater [60] IP Multimedia Subsystem and built a proof-of-concept. Our results demonstrate that, by utilizing application and protocol specific knowledge, affinity aggregates can effectively bound SFC delays and significantly reduce protocol errors and service disruptions.

4.1 Introduction

In traditional deployments of large carrier-grade systems, network service elements (functions) execute on hardware with dedicated CPU, memory and storage resources. The hardware boxes are connected via high speed links in operator data centers (DCs). Since the network is purpose-built to handle predefined network elements (NEs) and workload, the deployment is optimized to meet service requirements [61]. This includes allocating adequate resources and carefully placing NEs to meet latency requirements. NEs that constitute a Service Function Chain (SFC) or, more generally, a forwarding graph, are deployed in the same data center, and are carefully configured to meet Service Level Agreements (SLAs) or Quality of Service (QoS) requirements.

Network Functions Virtualization (NFV) leverages Commercial off-the-shelf (COTS) hardware to dynamically deploy network services. New network service instances are created by adding NEs to existing SFCs using virtualization and programmable networking technologies such as Software Defined Networking (SDN). NFV orchestration frameworks can instantiate these Virtualized Network Functions (VNFs) on-demand. The eventual placement of these VNFs is a balancing act by the orchestrator to meet both the QoS requirements of the deployed service and the need for cloud providers to maximize the utilization of the underlying infrastructure. Owing to the operational polices of the orchestrator and the physical locations of the data centers that it manages, new NE instances may be located on different racks or even different data centers. This, coupled with the unpredictable latency variations due to the sharing of the underlying physical infrastructure among services, can cause violations of end-to-end (e2e) latency requirements of SFCs [62]. Distributed instantiations of SFCs and latency variations can cause significant performance degradation since current applications and network protocol stacks are designed for traditional deployments and therefore react poorly to such “unbounded” latencies.

In systems such as Evolved Packet Core (EPC) and IP Multimedia Subsystems (IMS) where multiple NEs participate in service delivery, congestion on any interconnecting link triggers message drops or retransmissions. Constituent NEs often aggressively retransmit latency-sensitive messages to ensure timely execution of the protocol call flows [1]. Such re-transmissions aggravate network conditions, leading to further QoS deterioration. Since a single event/action can produce multiple message exchanges among the constituent elements in an SFC, an orchestrator must consider, when placing the SFC elements, the type and frequency of message exchanges among the SFC elements – a factor which is not considered by current orchestration frameworks.

A natural solution to control latency with NFV is to instantiate service elements within an SFC onto physical machines that are in close proximity. This ensures that congestion in other parts of the DC, as well as latency due to inter-DC communication, can be avoided. However, such a placement policy will force cloud provides to preallocate VNF resources in designated sections of the DC, ultimately undermining their ability to maximize infrastructure resource utilization. Even if such a policy can be implemented, the footprints of current VM-based VNFs are far too large to guarantee close proximity allocation. VNFs also support mobile users: even if a user is assigned to an NE where all SFC elements meet latency demands, user mobility makes it impossible to sustain such assignments. The mobile user can move across geographic regions, and this generally entails handover of the user session to NEs physically closer to the user location, which will inevitability result in user traffic traversing multiple data centers.

In this chapter, we explore the design of a small-footprint, stateless and portable VNF solution based on aggregating microservices. Our solution, Contain-ed, meets latency demands while simultaneously supporting user mobility and elastic resource allocation. Contain-ed aims to:

1. Bound e2e service latency by creating collocated aggregates of NEs.
2. Develop a service-aware, latency-sensitive orchestration and deployment framework at a low cost to the provider.

4.2 Contain-ed Architecture

Our design is guided by two key observations: **(1) VNF Affinity:** The number of messages among VNFs depends on the standards being used and the SFC structure. For example, in a virtualized EPC system [63], 41% of the signaling messages that are incident on the Mobility Management Entity (MME) are propagated to the Serving Gateway (SGW), but only 18% of the MME signaling load is propagated to the Packet Data Network Gateway (PGW). Protocol message exchanges and/or SFC dependencies enable us to identify affinity between VNFs or VNF components (VNFCs). **(2) Transactional Atomicity:** Transactions are sequences of messages that are exchanged among VNFs/VNFCs to handle a network event. Table 4.1 enumerates common network events in the IMS and EPC systems. These network events are often a result of user actions and each independent action (e.g., REGISTER) can trigger a sequence of messages. VNFs involved in processing a user's messages generally allocate/update state information, and this state information is used to process future messages of this user. The state information is either stored locally (in traditional network designs) or in shared storage (in NFV based designs). We observe that, due to state dependencies, user messages that are part of a specific transaction are, in general, processed by a specific VNF instance. However, once the transaction is completed, this state information can be shared with other VNF instances to handle future transactions.

Contain-ed leverages these observations as follows. Affinity dictates that certain VNFs in an SFC or VNFCs in a complex VNF be placed in close proximity to meet e2e latency requirements. The smaller resource footprint of virtualization technologies like containers enables microservice bundles of VNFs with high affinity to be placed near each other. Contain-ed creates network microservice bundles called *Affinity Aggregates (AAs)*. AAs are bundles of network services comprising VNFs that have message exchange affinity towards each other. AAs are instantiated as a single logical entity of microservices using lightweight virtualization technologies (containers). Each AA is configured to handle a predetermined transaction type and only consists of VNFs/VNFCs involved in processing this transaction type. Contain-ed includes components for managing and orchestrating AA instances, with the goal of distributing load across active AA instances and resource flexing according to workload variations. Figure 4.1 illustrates the Contain-ed architecture, whose components we now describe.

Affinity Analytics Engine (AAE): The AAE is an offline module that analyzes SFC dependencies and message exchange sequences to determine the required AA types. The AAE uses the VNF affinity information derived from analyzing the VNF messages exchanges to decide which VNFs should be bundled together as an AA. The AAE also determines transactional boundaries so that the same AA instance is used to handle a transaction's message exchange sequence in an atomic manner. Additionally, the AAE determines what to store in the shared state store across all AA instances.

A single transaction can generate significant amounts of intermediate state information, based on the structure of the SFC and the protocols involved. While it is possible to publish all intermediate state information generated by an AA to the shared state store, such a design would lead to significant performance degradation due to increased message exchanges between the AAs and the state store. Furthermore, all intermediate state information is not

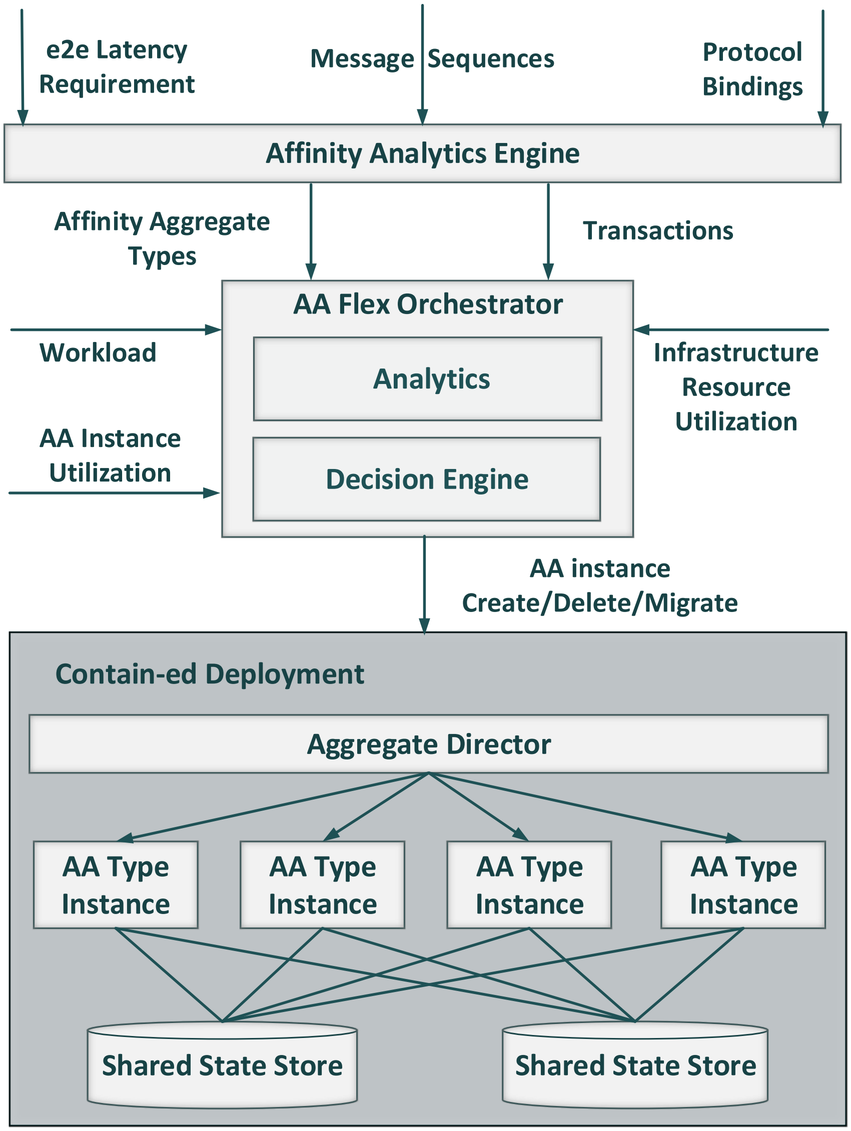


Figure 4.1. Contain-ed Architecture

required by the VNFs/VNFCs to handle independent transactions. As an example, the MME processes 10 of the 18 messages generated during the EPC Attach procedure [63], and each of these message exchanges is capable of generating intermediate state information. However, if the AA instance that handles the Attach request does not change during Attach procedure, there is little merit in publishing intermediate state information to the state store. The AAE therefore leverages transactional boundaries to determine the minimum state information that must be shared across AA instances. Only state information that persists across transactions is published in the state store.

Contain-ed transactions are specific to an SFC, and the messages that constitute a transaction are driven by protocol bindings within the SFC. Example message exchanges for IMS and their transaction boundaries are shown in Figure 4.3. Table 4.1 lists the AA types from our analysis of IMS and EPC protocol message exchanges and latency requirements. When AAs have the same VNFs, the same AA type can be used to handle different kinds of transactions/network events. For services such as Home Subscriber Server (HSS), Policy and Charging Rules Function (PCRF) and Online Charging System (OCS) that need database lookups, the Front End (FE) component [64] can be instantiated with the AA. The REGISTER AA can also contain the Application Server (AS) if specified in the Initial Filter Criteria (iFC) [3], [65].

AA Flex Orchestrator (AFO): The AFO manages the life-cycle of AA instances. It continuously monitors their resource usage and workload. If the latency requirements of a specific request type are not being met, the AFO deploys new AA instances with appropriate resources and at appropriate locations to meet the latency requirements. Conversely, if the workload decreases, the AFO removes unneeded AA instances after migrating active user sessions to other active AAs. Contain-ed transactions are short-lived compared to user sessions, which enables the AFO to elastically manage the resource allocation for the incoming workload. AA instance information is communicated to the AA Director for forwarding transactions. For example, all VNFs that participate in user registration can be bundled into a REGISTER AA type. Depending on the allocated resources (hence capacity of the AA type) and expected peak load, the AFO determines the number of instances of this AA type to deploy and how/when to add/remove instances to match workload dynamics.

AA Director (AD): The AD is an online module that directs incoming traffic to different active AA instances based on transaction types. The AD maintains a list of all active AA instances and their capabilities, and directs traffic (along transaction boundaries) accordingly. Multiple instances of a particular AA type can coexist with different resource allocations. When a new AA instance is spawned, the AFO updates the AD with its AA type and resource allocation. This enables the AD to intelligently load-balance the incom-

Table 4.1. IMS and EPC Affinity Aggregates (AAs)

Network Event	VNFs in AA	AA Type
IP Multimedia Subsystem (IMS)		
REGISTER	P/I/S-CSCF,HSS	REGISTER
INVITE	P/I/S-CSCF, AS, OCS	INVITE
NOTIFY SUBSCRIBE	P/I/S-CSCF	SUBSCRIBE
Evolved Packet Core (EPC)		
ATTACH	MME, HSS, SGW PGW, PCRF	ATTACH-DETACH
DETACH	MME, HSS, SGW PGW, PCRF	ATTACH-DETACH
HANDOVER	MME, SGW	HANDOVER-SR
BEARER SETUP	MME, SGW PGW, PCRF	BEARER-CRT
SERVICE REQUEST (SR)	MME, SGW	HANDOVER-SR

ing workload on available AA instances. The AD analyzes each incoming packet to classify it according to the AA types and transaction boundaries. All messages associated with a particular transaction (*e.g.*, messages that are part of a single user registration request) that were handled by a specific AA instance will continue to be directed to the same instance until the transaction is completed. This implies that AAs can only be deleted when there are no active transactions pending. When the AFO decides to scale-in an AA instance, the AD removes it from the active list and stops sending new transactions to it.

A single AD instance is capable of handling incoming traffic for multiple AA types. In cases where these AAs are part of different systems, such as the EPC and the IMS, which use different signaling protocols, the AD has to support multiple protocols. The AD is not, however, required to understand all the protocols that are used within the SFCs. For example, an analysis of the AAs in Table 4.1 reveals that all *inbound messages* for the IMS AAs use the Session Initiation Protocol (SIP) [4]. Similarly, AAs in the EPC system use the GPRS Tunneling Protocol (GTP-C) [66] for all *inbound messages*. Therefore, an AD that handles both EPC and IMS traffic using the AAs described in Table 4.1 is only required to support the SIP and GTP-C protocols.

Shared State Store (SSS): The shared state store is used by AA instances to store persistent state information across transaction boundaries. This allows incident workload to be distributed across multiple AA instances. The SSS is implemented as a key-value store and is agnostic to the actual structure/definition of state elements as specified by VNFs. The AAs use a simple Representational State Transfer (REST) based interface to store/fetch the sate information. Several VNFs (including Clearwater which we use in our evaluation) already support persistent state information management for horizontal scaling of individual components. The SSS can be deployed as a geographically redundant cluster when a single instance cannot handle the workload. The AFO can create multiple instances of the SSS in case the data store/fetch latency exceeds a predetermined threshold.

Contain-ed determines VNFs/VNFCs that handle messages of a specific transaction type and deploys these VNFs or VNFCs as a single AA. For example, Contain-ed can create an AA that handles only REGISTER messages (REGISTER-AA) and another that only handles messages of type INVITE (INVITE-AA) as described in Table 4.1. Such a decomposition of the functionality of an SFC into AAs offers the following key advantages: **(1)** Since not all elements of an SFC are involved in processing all transaction types, the resource requirements of an AA can be significantly lower than that of the original SFC. AAs with lower resource footprints are more likely to be instantiated in close proximity as compared to the entire SFC. **(2)** AAs enable granular resource allocation by coupling resource allocation with traffic composition. The AFO can scale-out AAs handling a specific transaction type as the percentage of messages of that transaction type increases. This allows Contain-ed to react, in real time, to incoming traffic composition.

4.3 Contain-ed in Action

In this section, we illustrate how Contain-ed can be leveraged for deploying an IMS instance to increase the utilization of the underlying NFV infrastructure.

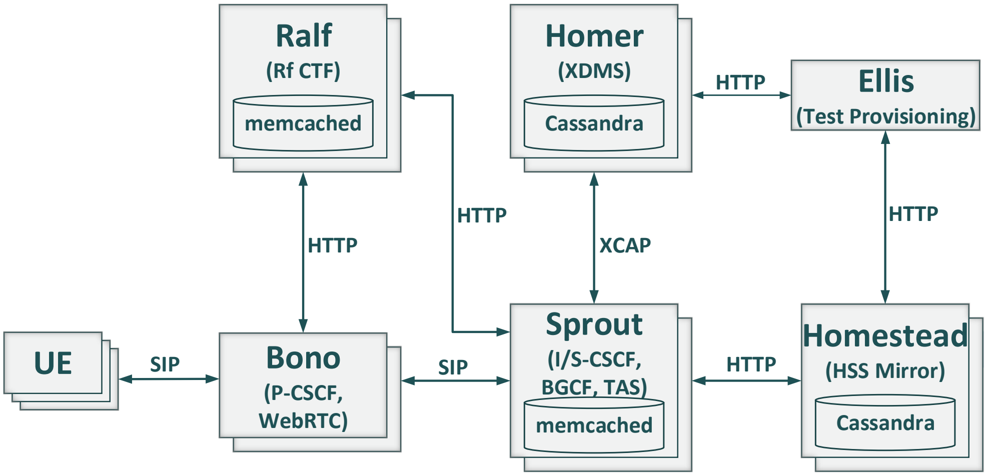


Figure 4.2. Clearwater Architecture

4.3.1 Project Clearwater: IMS in the Cloud

We choose Clearwater, an open-source IMS implementation. The availability of a containerized implementation of Clearwater enabled us to better compare performance of different IMS deployment options. While Clearwater provides a horizontally scalable clustered IMS implementation, the VNF components in Clearwater do not strictly match standard IMS functional elements. Clearwater utilizes web-optimized technologies like Cassandra and memcached to store long-lived state, provide redundancy, and eliminate the need for state replication during scale-in and scale-out.

The architecture of Clearwater is illustrated in Figure 4.2 (adapted from [60]). For brevity, only the components used in our experiments are depicted. We briefly explain the Clearwater components that can be deployed individually and horizontally scaled. **Bono** is the edge proxy component that implements the P-CSCF (Proxy Call Session Control Function) in the 3GPP IMS architecture [3]. SIP clients communicate with Bono over UDP/TCP connections and are anchored at a Bono instance for the lifetime of the registration. **Sprout** implements the Registrar, I/S-CSCF (Interrogating/Serving CSCF) and Application Server components. Sprout nodes store the client registration data and other session and event state in a memcached cluster. There are no long-lived associations between a user session and a Sprout instance. **Homestead** provides a REST interface to Sprout for retrieving the

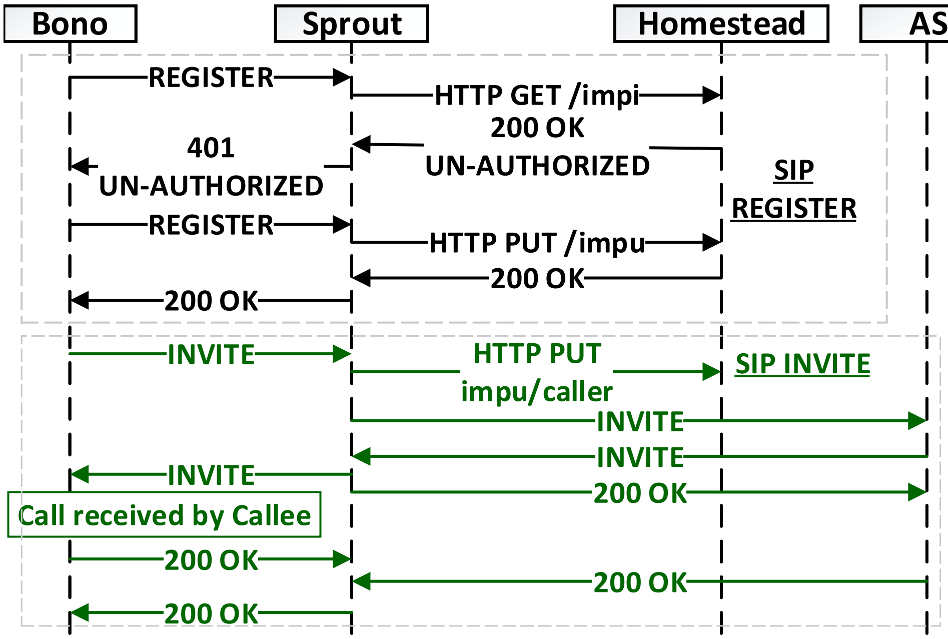


Figure 4.3. Clearwater IMS Call Flow

authentication vectors and user profiles. Homestead can host this data locally or retrieve it from the HSS using the Diameter Cx interface. **Homer** acts an XML Document Management Server that stores the service profiles. **Ralf** implements the Off-line Charging Trigger Function (CTF). Bono and Sprout report chargeable events to Ralf. Figure 4.3 illustrates this call flow without a third-party REGISTER in the iFC.

4.3.2 Mapping with Contain-ed

We determine the AAs by applying the principles in §4.2: **(1) VNF Affinity**: Analyzing the 3GPP IMS architecture [3], we find that there is high affinity between the P-CSCF and S-CSCF components. Therefore, we can aggregate the Bono and Sprout nodes in Clearwater to create an AA. **(2) Transactional Atomicity**: We demonstrate the application of this principle by analyzing user registration in IMS, which generates two messages by the user device. Since both messages must be handled by the same instance of Bono and Sprout, we consider user registration as a transactional boundary.

We thus create an AA of type "REGISTER" corresponding to the SIP REGISTER call flow. A similar reasoning allows us to create AAs of types "SUBSCRIBE" and "INVITE" for the IMS user SUBSCRIBE/NOTIFY and INVITE call flows, respectively. The AAs for "REGISTER" and "SUBSCRIBE" consist of an instance of Bono and Sprout, while the AA for "INVITE" additionally contains an instance of Ralf due to the CTF interaction described in Table 4.1. We do not use an Application Server (AS) in our testing, so it is not included in the AAs.

In Clearwater, Bono and Sprout operate in a transaction-stateful manner. Transactions in the same SIP dialog can be handled by a different Sprout instance since the Sprout instances share long-lived user state using memcached. Clearwater therefore supports the transactional atomicity property of Contain-ed. Contain-ed leverages the Clearwater memcached as the shared state store.

We develop the AD component based on the OpenSIPS [67] dispatcher. In this implementation, the AD anchors all the incoming and outgoing calls from Clearwater and acts a stateless inbound proxy. It uses a hash on the message "Call-ID" to direct incoming request messages. This mechanism ensures that the messages for the same user session are directed to the same AA instance. For outgoing messages, the AD inserts appropriate SIP headers to ensure that messages take appropriate paths.

4.4 Experimental Evaluation

We developed a prototype implementation of the Contain-ed deployment component (the dark shaded box in Figure 4.1). We use the information in Table 4.1 to bundle the Clearwater components into AA types. The AAs are instantiated at startup to match the workload requirements (the AFO dynamic scaling/instantiation functionality is not yet implemented).

4.4.1 Experimental Setup

We use Docker version 17.03.0-ce and Docker-compose (v1.11.2) for microservice container lifecycle management. The Clearwater VNF components run within a container on the same physical host. A private subnet created by Docker is used for communication between these containers, thereby minimizing the communication latency among the Clearwater VNF microservices. The physical resources of the server are shared by all containers and there are no resource constraints on an individual container. The Contain-ed AD component is deployed on the same physical machine as the Clearwater VNF. The AD runs on the physical machine directly, and therefore shares the resources with the Clearwater VNF components.

Workload generation: We use SIPp [68] as a workload generator. SIPp runs on a dedicated physical machine, and generates two types of requests: REGISTER and SUBSCRIBE. As shown in Figure 4.3, REGISTER requests are used to register the user device in the network and result in the generation of two messages (initial request and challenge response) from the user device. SUBSCRIBE requests are used to subscribe to the the state of a user already registered with Clearwater. A SUBSCRIBE request from the requesting client is followed by a NOTIFY request from the server to update the client with the user subscription status. We measure the number of failures by the observing the result code in the SIP response message. Per the SIP specification, for register, "200 OK" indicates success and "401 Unauthorized" is used to challenge. All other 3XX and 4XX codes are considered failures. We observe the error codes received by SIPp for each message type and use them to infer the number of failures.

We generate a workload of 300 requests/s to 1800 requests/s in steps of 300 requests/s, and measure the total number of failed calls for each workload type. As described earlier, aggressive retransmission of requests by the client or middleboxes can exacerbate performance problems, so we disable this to increase the overall throughput. In order to circumvent the impact of retransmissions on our experiments, we configure SIPp to not retransmit requests that failed due to timeouts. Each experiment runs for 60 seconds. The results presented below represent the mean of at least 10 samples for each call rate and delay value.

The performance of a complex VNF like Clearwater is impacted by the control interplay among its functional components. Previous studies [8] have revealed that disproportionate resource utilization by Clearwater components can influence system performance, and the overall throughput depends on the resources allocated to individual components. Clearwater employs token buckets and timeout-based peer blacklisting mechanisms for fault-tolerant overload control. This can also influence the overall throughput. Furthermore, individual components may timeout and discard incoming requests. As an example, Sprout uses a timer to wait for the response messages from Homestead, and if no response is received before a timeout, a failure response (response code timeout 408) is issued to the client. To minimize the impact of disproportionate resource utilization, we do not allocate dedicated resources to any container and all Clearwater components share the available system resources. However, overall performance is limited by the token bucket rate and timeout(s) at individual components, resource utilization notwithstanding.

4.4.2 Experimental Results

Our experiments are designed to investigate the impact of network latency on Clearwater, and to quantify the performance benefits of Contain-ed. We begin by benchmarking Clearwater in "ideal" conditions on our testbed. In this case, all communicating VNF components are instantiated on the same physical machine. A single instance of Clearwater is created and both REGISTER and SUBSCRIBE messages are handled by this instance. This setup is labeled "ideal" in our plots. We measure the performance of this setup with both REGISTER and SUBSCRIBE workloads.

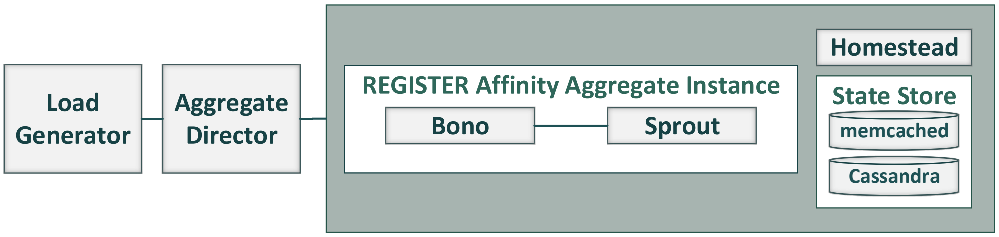


Figure 4.4. Contain-ed setup with REGISTER AA

We also measure the performance of Clearwater when the VNF components are not located on the same physical machine and therefore the communication latencies are higher than the ideal case. We simulate a scenario where the Sprout node is located in a different DC by adding delays on the Sprout-bound links. As described earlier, the SIP REGISTER request generates two register messages from Bono to Sprout and two database lookup re-

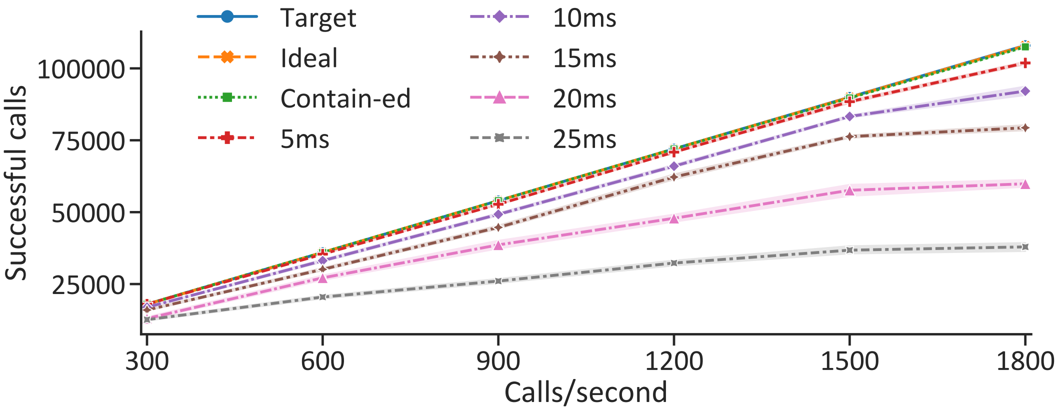


Figure 4.5. Successful REGISTER calls

quests from Sprout to Homestead, and therefore Sprout placement is vital to the performance of Clearwater. We use "tc" to introduce delays on the links from Bono to Sprout and Sprout to Homestead. We use delays of 5 ms, 10 ms, 15 ms, 20 ms and 25 ms and compare the performance of this setup with the "ideal" case. Figures 4.5 and 4.7 present the results. In both figures, the error bars represent the minimum and maximum values observed among all samples for a data point. The label "Target" in the figures indicates the maximum number of calls that can be successfully processed at a given call rate.

As seen from Figures 4.5 and 4.7, increasing communication latency to a single Clearwater component (Sprout) can result in significant performance degradation. The impact of the introduced latency is not significant at low call rates. However, as the call rate reaches the system capacity, there is significant drop in system throughput. This is a consequence of the timeouts experienced at individual components. As load increases, the number of messages that are waiting for a response at each individual component becomes larger, and higher system capacity is utilized in sending timeout responses at each individual component.

We now describe our experimental setup using Contain-ed. Figure 4.4 shows an instantiation of Contain-ed to handle REGISTER messages. It consists of the REGISTER AA (Sprout and Bono), the shared state store, Homestead, and the AD. Figure 4.6 depicts the setup of Contain-ed for handling SUBSCRIBE. This setup consists of two AAs (REGISTER, SUBSCRIBE), since the users must be registered before SUBSCRIBE messages. Both the REGISTER and SUBSCRIBE AAs contain an instance of Bono and Sprout. All other VNF components like Homestead and the shared state store are shared by the AAs. For the SUBSCRIBE setup, all REGISTER messages are handled by the REGISTER AA, and SUBSCRIBE/NOTIFY messages are handled by the SUBSCRIBE AA. A single instance of AD is created in both the cases, which forwards the incoming traffic to the appropriate AA.

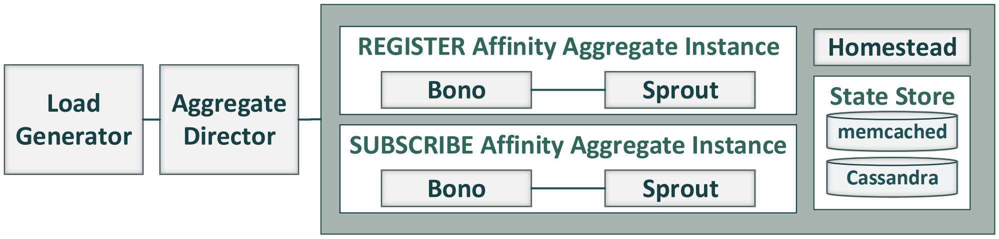


Figure 4.6. Contain-ed setup with REGISTER/SUBSCRIBE AAs

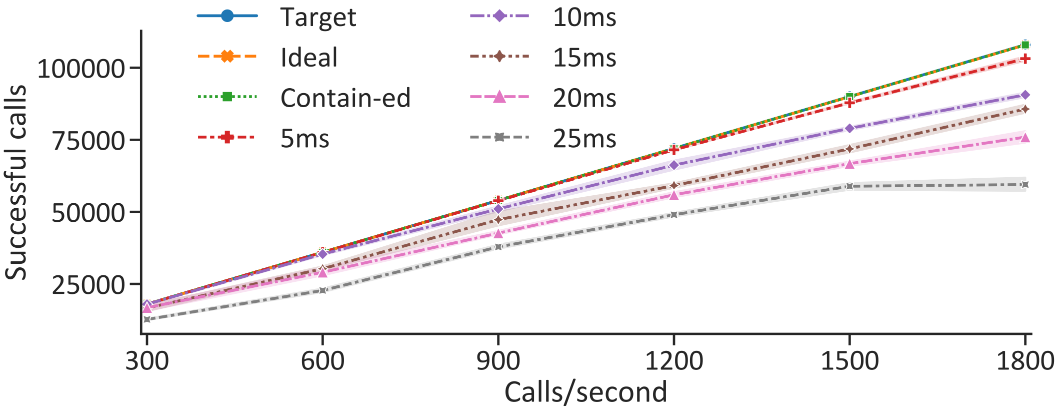


Figure 4.7. Successful SUBSCRIBE calls

Comparing the results of Contain-ed with "ideal" in Figures 4.5 and 4.7, we conclude that the AD does not result in significant call drop compared to the ideal setup, and the overhead due to the AD does not significantly impact overall performance. The DC setup with induced latency increasingly drops higher numbers of messages as the latency increases, but the Contain-ed setup continues to process messages without suffering from significant performance degradation. Even when multiple AA instances of different types are created, the performance impact of the AD and Contain-ed is minimal.

It is important to note that workloads react differently to increasing latency. This is due to the nature of communication between various components within the VNF. User actions that require memory lookup/update (authorization/billing events) will respond poorly to increased latency towards the memcached/cassandra components and workloads that require frequent communication with other components like SUBSCRIBE will respond poorly to increased latency towards state management components within the VNF. With traditional network placement, it is difficult to strike the right balance between workloads and their dependencies. In contrast, the Contain-ed setup can ensure co-location of VNF components for each workload type, and, as seen from the results above, will continue to process various workload types without suffering from significant performance degradation.

4.5 Chapter Summary

In this chapter we presented *C*ontain-ed, a VNF placement solution that bounds the e2e latency in SFCs and inter-VNF control message exchanges by creating *microservice aggregates* based on the affinity between VNFs. We showed that complex control plane VNFs can be decomposed into functionality based microservices components which can then be aggregated and deployed using light-weight virtualization technologies. Such *microservice aggregates* when placed in proximity can bound e2e latency while providing fine grained resource flexing. Our results demonstrate that, by utilizing application and protocol specific knowledge, orchestrators can effectively bound SFC delays and significantly reduce SLA violations.

5. INVENIO: PROVENANCE-DRIVEN MICROSERVICE DEPLOYMENT IN THE CELLULAR CORE

Cloud-native architectures enable rapid service deployment and scaling in the cellular core. However, integrating poorly understood microservice components into traditional Service Function Chains (SFCs) limits a provider's control over the end-to-end latency incurred in service delivery. Orchestration frameworks instantiate and place myriads of microservice components without fully understanding the impact of their placement decisions on user requirements.

In this chapter, we explore challenges faced by service providers in managing complex SFCs, and propose *Invenio* to enable the providers to develop an effective placement strategy for microservice-based SFCs without prior knowledge of service functionality. *Invenio* uses the knowledge of user actions and provenance information extracted from network traces to compute *transactional affinity* between the functions in an SFC for each user action. The transactional affinity values are then used to make placement decisions to meet latency constraints of individual user actions. Our experiments with two IP Multimedia Subsystem (IMS) implementations demonstrate that transactional affinity-based placement significantly reduces failures by limiting message processing latency within SFCs.

5.1 Introduction

Network Functions Virtualization (NFV) has enabled service providers to deploy virtualized instances of Network Functions (NFs) on demand [69]. New service offerings are now created by adding one or more NFs to a Service Function Chain (SFC), i.e., a graph of NFs. An existing service can be extended by adding software modules, whereas unpopular features can be removed by deleting service modules. Software architectures have evolved to support this rapid pace of service deployment, and disaggregated fine-grained microservice designs are now replacing monolithic designs [70], [71].

The power to rapidly add and delete new services comes at a cost, however. SFCs are becoming more complex, and the effort associated with service deployment is growing [25]–

[27], [32]. Service providers, in an attempt to cut costs, are increasingly using private or public clouds to deploy services that had traditionally been confined to a single data center and had used carefully-designed proprietary hardware. This *cloudification* poses unique challenges to orchestration frameworks, particularly in instantiating and placing Virtualized Network Functions (VNFs) in an SFC with strict Service Level Agreements (SLAs) [32].

Prior work [20]–[22], [72] has shown that network functions (NFs) in systems such as the cellular Evolved Packet Core (EPC) and IP Multimedia Subsystems (IMS) have stringent end-to-end latency requirements and react poorly to unpredictable latency variation. Fortunately, service providers can leverage their knowledge of NF functionality and meticulously define SFCs [1], [3]. Virtualization platforms such as Openstack [73], Kubernetes [74], and Docker [53] allow administrators to configure "affinity policies" in NF placement. The affinity policies specify which NFs should be co-located to meet SLA requirements. However, the increasing use of non-standard interfaces and the ongoing integration of 5G core (5GC) [75] into existing 4G network deployments is necessitating extensive manual re-analysis of communication patterns. The diversity of NFs in modern networks and the new 5GC interfaces make determining the SFCs involved in service delivery and the affinities between their constituent NFs a time-consuming and error-prone task.

NFs with microservice designs further complicate SFCs. Microservices advocate the use of fine-grained, independent components that can be deployed as autonomous entities communicating via REST-based proprietary interfaces [28], [31]. This results in disaggregation and decomposition of a VNF into multiple smaller VNF Components (VNFCs), and longer, more complex SFCs [71]. Further, lack of standardization in microservice architectures yields VNFCs that play roles that do not accurately map to an NF defined by standards. That is, a VNFC may take the role of several standard-defined NFs and support several network interfaces. Conversely, the functionality of a standard-defined NF may be collectively performed by multiple VNFCs. The ambiguity in the role of VNFCs implies that placement using domain knowledge is insufficient, and we need automated tools to infer communication patterns between microservice components to aid service providers. Our work attempts to fulfill this need.

We use the information exposed by microservices to optimize NF placement and meet SLAs [25]–[27]. However, merely co-locating NFs based on the number of messages they exchange [27] can yield unexpected results due to the diversity of workloads. Instead, we propose grouping events triggered by a user action into *transactions*, and computing *transactional affinity* between NFs. A provider can then make placement decisions based on transactional affinity values, together with policy and transaction type distribution. For example, a VNFC used during voice calls, but not for SMS (text-msg), can be placed based on the most common traffic type.

In this chapter, we propose *Invenio*, a system for aiding service providers in deploying control-plane NFs. *Invenio* maps user activity at the network edge to traffic in the network core, computes transactional affinity, and makes placement decisions. *Invenio* includes two subsystems that are executed after upgrades or policy and service changes: one in which a snapshot of traffic is analyzed to compute affinity values, and another in which an orchestrator uses computed transactional affinity values, in conjunction with policy rules and current transaction type distribution, to make placement decisions. *Invenio* empowers providers to optimize placement to meet SLA objectives even with upgrades in services and micorservices, and changing user QoE demands. For example, a provider may choose to optimize placement to (a) reduce latency of the currently dominant workload type, or (b) reduce latency of interactive workload types, such as voice calls which have a higher impact on user Quality of Experience (QoE) [76]. In summary,

1. We identify the challenges for a service provider to meet SLAs (§5.3) and introduce the notion of transactional affinity (§5.4).
2. We propose *Invenio* for service providers to identify transaction types for user actions (§5.5). This includes, to the best of our knowledge, the first session and transaction slicing algorithms to isolate messages of a given transaction type for a specific user. *Invenio* can also check NF interoperability and diagnose network problems (§5.7).
3. We experimentally demonstrate the benefits of placement based on affinity by applying *Invenio* to microservice-based cellular network implementations and evaluating the impact of placement on the performance of voice-call and text-msg workloads (§5.6).

We find that placement with *Invenio* results in up to 21% performance gain compared to message count-based placement algorithms, and up to 51% gain over default placement. While our evaluation uses the 4G control plane as a case study, the principles underlying *Invenio* are applicable to the service-based architecture of the 5GC and other microservice-based deployments.

5.2 Motivation

A control-plane NF can be instantiated on bare metal (as a Physical Network Function (PNF)) or on virtualized hardware (as a VNF), and a VNF can be deployed as a collection of VNFCs. In the rest of this chapter, we use the term NF to refer to all three types of instantiations (PNF/VNF/VNFC). The increasing use of private or public clouds to reduce operational costs has yielded scenarios where NFs in an SFC are deployed across multiple physical machines in one or more data centers. Consider Fig. 5.1 which shows an example microservice-based cellular network for Voice over LTE (VoLTE) that includes wireless access, session management, voice-call signaling, policy control (QoS), and billing. Latency-sensitive NFs (such as signaling and policy) may be connected by high and unpredictable latency links. An orchestrator that cannot instantiate the entire SFC in Fig. 5.1 on a single machine or rack can identify the NFs exchanging a large number of messages and place them in close proximity. Modern networks offer many services, however, and NFs exchange different types and numbers of messages to support each service.

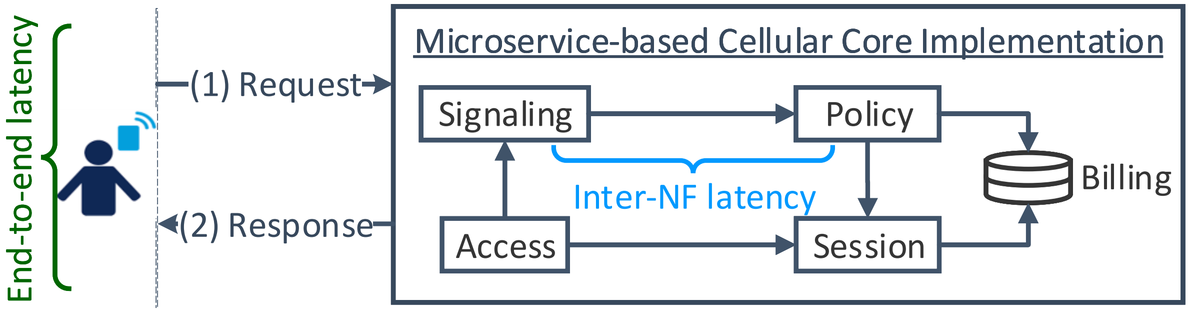


Figure 5.1. A microservice-based cellular network

Not all services have equal impact on user-perceived latency and QoE. For example, interactive services such as voice calls impact user QoE more than non-interactive services such as text-msg or presence services [76]. Orchestrators must reduce the end-to-end latency of interactive services by minimizing the inter-NF latency for NFs handling these services. Simple techniques such as counting total messages exchanged between NFs [27] are not always effective in making placement decisions as they do not explicitly consider the impact of inter-NF latency on user QoE.

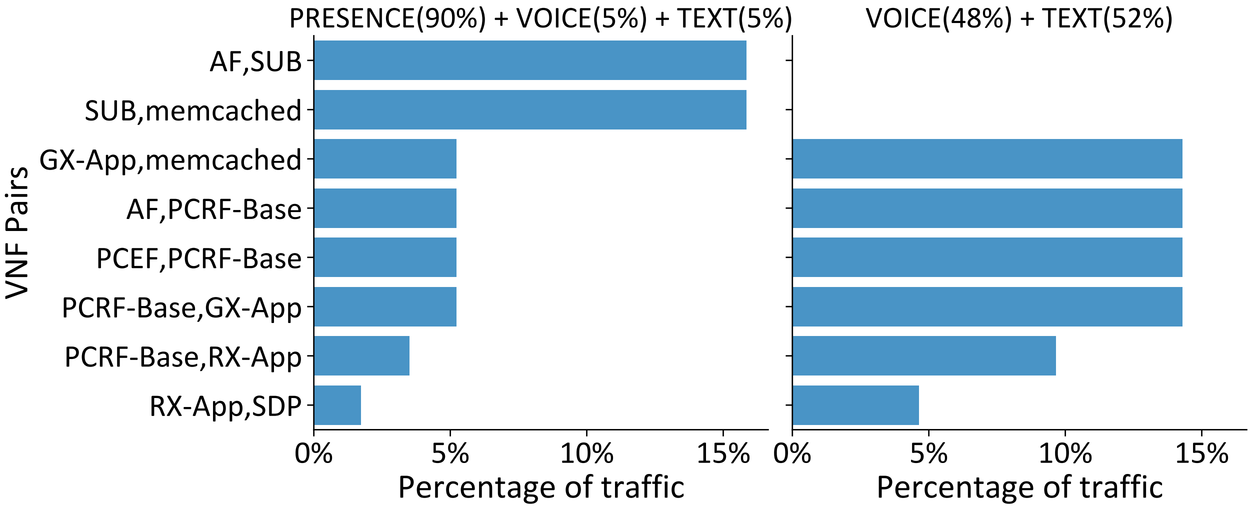


Figure 5.2. Impact of transaction type distribution on number of messages exchanged between NF pairs in microservice-based VoLTE implementation

Fig. 5.2 shows the percentage of traffic exchanged by NF pairs (in our implementation in §5.6.2) for two different transaction type distributions of voice-call, text-msg, and presence services. The plot on the left uses traffic proportions from typical busy-hour IMS traffic [77] in which presence is triggered ∼9x more frequently than voice. Clearly, exchanged messages depend on the incoming transaction type distribution and therefore merely using the number of messages for placement [27] may optimize non-interactive services such as presence and degrade user QoE. To meet SLAs for latency-sensitive services, service providers may (a) create dedicated NFs to optimize specific functionality [22], or (b) decompose existing monolithic applications into lightweight microservice components, that are then aggregated by functionality to create NF bundles, placed together with a higher probability [21]. Manually identifying and configuring bundles can be difficult and error-prone, however.

Our work empowers service providers to easily and automatically react to upgrades and changing user QoE demands. Based on prior research [20]–[22], [72], we observe that: (a) NFs typically exchange several messages to complete a seemingly simple user action such as turning on User Equipment (UE) or making a voice call, and (b) Network endpoints only perceive latency in the actions they trigger (i.e., end-to-end latency in Fig. 5.1) and are oblivious to message exchanges and inter-NF latency within an SFC. User QoE therefore only depends on user action/network response pairs, such as initiating a voice call (action) and hearing a dial tone (network response), or turning on an Internet connection (action) and being connected to a packet access network such as LTE (network response).

5.3 Challenges

The goal of *Invenio* is to facilitate NF placement by leveraging readily available knowledge of *endpoint actions*. We group events or messages triggered due to a single user action into *transactions*. We then use this transaction information to compute *transactional affinity* (§5.4) between NFs for each transaction type. The transactional affinity information is used for NF placement. Since NFs in modern networks exchange numerous messages, manually determining control messages that are triggered due to a specific endpoint (or associated user or subscriber) action can be tedious and error-prone. We propose to (a) automatically isolate SFC control messages related to a user, and (b) map each message to an action invoked by that user. We describe the challenges in accomplishing these tasks in the remainder of this section.

Scale and complexity: We need to understand the protocols and message formats exchanged by each NF. For example, consider a cellular network EPC (including NFs to inter-work with previous generation networks (2G, 3G) and WiFi). Such an EPC deployment can involve 60+ NFs communicating via 15+ protocols over 150+ interfaces using 500+ message types [1], [78]. While many of these NFs are logical, the sheer number of NFs, supported protocols, and message types makes isolating and understanding control-plane traffic a difficult task.

User and session identification: Networks such as cellular networks check user (subscriber) identifiers located in control messages to determine the user associated with a device or a network endpoint, and NFs use these identifiers to enforce policies and bill users.

A user is identified by: (a) **Subscriber-ID:** the key used by the network to authenticate a device, identify packets associated with it, and bill the user, and (b) **Session-ID:** the key allocated by an NF to group together messages triggered by a device. Unlike the subscriber-ID, the value of session-ID is not pre-allocated, *i.e.,* NFs allocate a value at run-time. Different protocols and interfaces use different terms to refer to the subscriber-ID and session-ID carrying headers. Table 5.1 lists example protocol headers used by cellular network protocols.

Since the session-ID is dynamically allocated, the relation between session-ID and subscriber-ID may vary based on the NFs involved in message processing. When a single device creates multiple connections at the same time (such as in EPC), multiple session-IDs may be allocated to the same subscriber-ID. Additionally, an EPC/IMS may create a mapping between the session-ID and the subscriber-ID, and then use the two values interchangeably. Fig. 5.4 depicts an example where the IMS network uses the User-Name in the *From* header of the SIP protocol [4] to determine the subscriber-ID while interacting with the user, but further messages generated due to this user interaction use other protocol headers, such as the *Public-Identity/Subscription-Id* [2] used when communicating with the Policy and Charging Rules Function (PCRF). After receiving the initial message ((2) CCR in Fig. 5.4a) from the Policy Charging and Enforcement Function (PCEF), the PCRF creates a mapping be-

Table 5.1. Example user and session headers

Protocol	**Interface Name**	**Subscriber-ID Header Name**	**Session-ID Header Name**
SIP	Gm [3]	To, From	Call-ID
Diameter	Cx [79]	User-Name, Public-Identity	Session-Id
	Gx [80]	Subscription-Id	Session-Id
S1AP [1]	S1-MME	IMSI, TMSI	eNB-UE-S1AP-ID
HTTP/2 (5G) [75]	N7/N11	SUPI, SUCI	pduSessionId
	Rx/N5	SUPI, SUCI	appSessionId

tween Subscription-Id and session-ID. This mapping identifies the user in all future message exchanges between the PCRF and PCEF (9, 10, 15 and 16 in Fig. 5.4b omit the subscriber-ID headers and only carry the session-ID header). Therefore, identifying all messages that are triggered due to a user action requires understanding the mapping between session-ID and subscriber-ID.

As in the 4G core, 5GC NFs [75] use headers such as Subscription Permanent Identifier (SUPI) and Subscription Concealed Identifier (SUCI) to identify, authorize, and bill traffic. A user can create multiple sessions with 5GC data networks and therefore the 5GC NFs use session headers such as the pudSessionId in conjunction with the user ID to uniquely identify user sessions. Example headers used in 5CG are shown in Table 5.1. This shows that *Invenio* principles are applicable to the 5GC. When 4G EPC and the 5GC coexist, the complexity of manual NF placement further increases.

Proprietary microservices: Microservice architectures use fine-grained autonomous components, fragmenting traditional control-plane NFs into multiple VNFCs [25], [26], [32]. The VNFCs are independently instantiated, and communicate with each other using proprietary message formats. This lack of standardization implies that the roles and functionalities of VNFCs are not well-understood and can change with new versions, altering their affinity. Consequently, service providers must (re)analyze affinity whenever NFs are upgraded or a service is added/removed. Microservices also result in longer, more complex SFCs, reducing the latency allowed for each VNFC component [71].

While the lack of standardization can complicate mapping a given message to a user action, microservices often reuse the subscriber-ID/session-ID in traditional signaling protocols [60], [81] to facilitate logging and reduce performance overhead. For example, the timer service (Chronos) in Clearwater [60], a popular microservice-based IMS implementation, uses the "Call-ID" header in Session Initiation Protocol (SIP) messages to manage timers. This behavior can be exploited to trace VNFC-generated messages to user actions.

Lessons learned: The above discussion highlights three consequences for *Invenio*. First, *Invenio* should automate message and event processing, which should be transformed into a protocol-agnostic format before further processing. Second, *Invenio* should understand the relation between different identifiers used by NFs to correlate messages related to the same user. This involves understanding the user-identifying headers used by standard protocols, and correlation of identifiers in proprietary message payloads used by VNFCs. Third, *Invenio* should understand user actions and their corresponding responses, and map each message to a specific user action. Since internal implementations of microservice-based systems change frequently, *Invenio* should only use endpoint messages which follow well-known protocols (such as messages (1) and (2) in Fig. 5.1) to map messages to user actions.

5.4 Problem Definition

Consider sets N, U, M, and R, where N represents NFs (PNFs, VNFs, or VNFCs) in a network, U represents user devices or end points that utilize the services provided by the network, M represents messages that can be sent or received by all NFs in N, and $R \subseteq M$ represents user request messages generated by a user $u \in U$ and responses sent back to users $\in U$.

To utilize network services, a user $u \in U$ sends a request r to an NF in N. A request sent by an endpoint u triggers the generation of several messages $m \in M$ between a subset of NFs. We denote this subset by N_r. NFs use a number of protocols to complete processing user requests. Typically, an NF will handle a part of the functionality, and forward messages to the next NF in an SFC. Additionally, an NF may utilize interfaces exposed by other NFs to acquire information needed for processing the message itself. Since we aim to model messages that are sent/received by each NF, the manner in which the messages are exchanged is irrelevant and we model messages of all protocols using the set M.

Set $M_r \subset M$ represents the messages triggered to handle a given user request $r \in R$. M_r does not include user-sent or received messages $\in R$. In addition to the messages M_r that are generated by an NF to handle a user-triggered request r, NFs in an SFC may generate messages that are *not* handling a request from u. Such messages include (but are not limited to) messages generated to synchronize state between NFs, and keep-alive or setup/teardown messages. These messages $\in M$ but $\notin M_r$ for any $r \in R$. For example, an online charging message that is generated by an NF to ensure the successful processing of a user-triggered message r is in M_r. An offline charging message that is generated by an NF to later bill the user for an already processed request is $\in M$ but $\notin M_r$.

We use the function Ω to define a mapping between the user-triggered messages and the NFs involved in processing these messages as well as the messages triggered by NFs to handle these messages; that is, $\Omega(r) = (N_r, M_r)$.

A service request from an endpoint may simply include a request message r_{start} from the endpoint and a response r_{end} from the network where $r_{start}, r_{end} \in R$. However, there are cases such as challenge-response procedures where the endpoint may have to respond to multiple requests from network to complete the initial request. That is, the sequence of messages processed by the endpoint to complete a service request is $< r_{start} \cdots r_{end} >$. We use the term "transaction" to refer to this set of messages, R_t, that are exchanged for delivering a specific service to the endpoints, where $R_t = \{r_{start}, \cdots, r_{end}\}$.

For a given transaction t, we define N_t and M_t as follows.

$N_t = \bigcup\limits_{r \in R_t} N_r$ represents all NFs involved in processing transaction messages R_t.

$M_t = \bigcup\limits_{r \in R_t} M_r$ represents all messages processed by all NFs in N_t to handle transaction messages R_t.

The NFs and messages involved in processing a transaction are typically the same for every instance of a certain transaction type tt.

Let the $c(tt, n_x, n_y)$ be the number of messages in M_{tt} exchanged between a pair of NFs $n_x, n_y \in N_{tt}$ for transaction type tt. The *transactional affinity* (referred to simply as "affinity" in the remainder of the chapter) between NFs n_x, n_y is defined as:

$$\texttt{Affinity}(tt, n_x, n_y) = c(tt, n_x, n_y). \tag{1}$$

Invenio affinity can easily incorporate additional metrics by updating Equation (1). For example, the function $c(tt, n_x, n_y)$ can incorporate the number of the hops traversed or latency incurred in communicating with a specific NF or it can avoid over-subscribed links. The function may also be updated to incorporate licensing or hardware constraints that limit the placement of network servers such as the Home Subscriber Server (HSS) and the Online Charging System (OCS).

In summary, *Invenio* computes N_r and M_r, determines the mapping $\Omega(r) = (N_r, M_r)$ for each request r, and identifies each transaction type tt and its associated N_{tt} and M_{tt} sets. This information is then used in Equation (1) to compute the affinity between NFs for every transaction type tt, in order to place NFs with relatively high affinity in close proximity. Unlike prior work, *Invenio* affinity considers the *complete transaction* instead of one or a few messages that are not accurate measures of the entire user experience.

5.5 Invenio Design

Fig. 5.3 shows the *Invenio* architecture. *Invenio* has two components: an affinity engine, executed after upgrades, and a placement engine, executed when a new NF is to be instantiated or after major changes in policies or transaction type distribution.

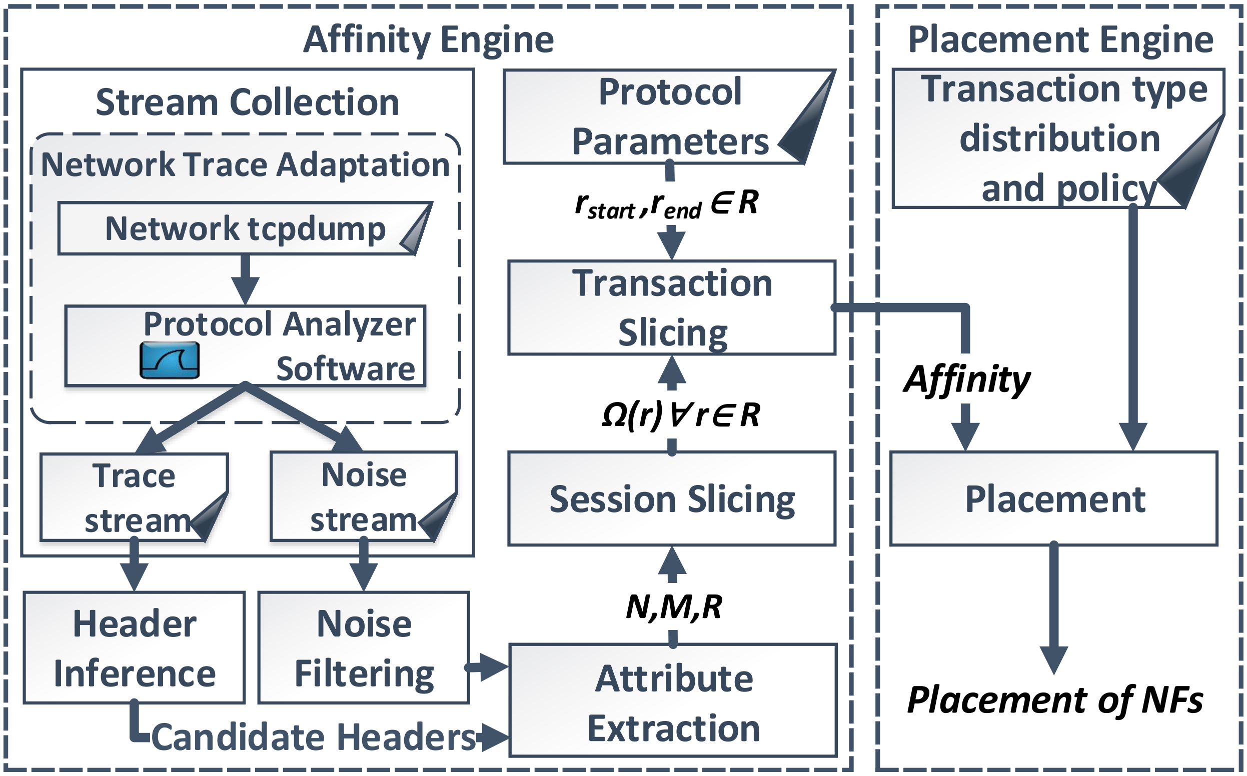


Figure 5.3. *Invenio* architecture

5.5.1 Inputs

NF message stream: *Invenio* uses messages exchanged between NFs in an SFC to identify transactions and their associated messages. Message sequences can be extracted from network traces or NF-provided information such as debug logs or a VNF Event Stream (VES) as specified in ONAP [82] and OPNFV [83]. In the absence of such structured data steams, *Invenio* uses traffic traces from running PNFs/VNFs/VNFCs. These traces can be collected at individual PNFs where physical infrastructure is used, or at Open vSwitch (OVS) or Docker bridge in case VNFs are deployed using virtualization platforms such as OpenStack [73] or Docker [53].

Invenio uses traffic snapshots collected after microservice upgrades. Traffic traces can also be collected during integration tests [84]. The affinity engine merges traffic traces in order to compute affinity values for each transaction type (Equation (**1**)). In the rest of this chapter, we use the term **trace stream** to refer to these input traffic snaphots.

Trace streams often contain extraneous messages that are not generated due to endpoint actions and are not part of any M_r. Such messages include setup messages, heartbeat messages, and synchronization messages. These messages are exchanged between NFs even in the absence of user-generated traffic. To eliminate such messages, *Invenio* uses a trace steam collected during an idle period, henceforth referred to as **noise stream**. An idle period is when NFs are running but no traffic is initiated by user devices.

Invenio utilizes output generated by open-source packet analyzer software such as Wireshark [85] to decode raw messages. We use Wireshark to export Packet Description Markup Language (PDML) and Portal Structure Markup Language (PSML) files, and use these files as inputs.

Protocol parameters: *Invenio* uses domain knowledge of the service provider, specified in a configuration file containing the following information in xml format: **(a) Transaction start/end messages:** are the message pairs $r_{start}, r_{end} \in R$ that are used by endpoints to start and terminate a service request. These values are only required for protocols that are used by endpoints. That is, for the example shown in Fig. 5.1, these values are only required for the SIP protocol used in the endpoint messages (1) and (2).

(b) Subscriber-ID and session-ID header names: are the names of the headers used by protocols to transmit subscriber-IDs and session-IDs. The headers may be used in conjunction with the output of the header inference module (§5.5.2) to extract the subscriber-ID and session-ID values from a message. These inputs are only necessary in cases such as the GPRS Tunneling Protocol (GTP), which uses integer identifiers instead of (the more common) text identifiers in the Uniform Resource Identifier format specified in RFC [86].

Transaction type distribution and policy: *Invenio* uses transaction type distribution and provider policy information to decide NF placement. The transaction type distribution can be obtained from an NF that processes endpoint messages. For example, the Proxy Call Session Control Function (P-CSCF) handles all inbound SIP traffic and therefore the P-CSCF NF instance(s) has the transaction type distribution information. Policy information allows providers to optimize NF placement based on their specific requirements (§5.5.7).

5.5.2 Header Inference

The first step in generating the set M_{tt} for computing affinity is to find user-identifying headers. A header inference module analyzes messages in the trace stream to identify possible headers that carry the subscriber/session-ID values.

Algorithm 1 computes candidate header names. The algorithm has two stages. Stage 1 (lines 1-12 of Algorithm 1) computes all header names that can carry the subscriber-ID values using the subscriber-ID formats described in [86]. This stage generates a list of candidate subscriber-ID header names ($candSubHdrN$) per protocol. Stage 2 (lines 13-20 of Algorithm 1) identifies all headers whose values repeat in messages exchanged between a pair of NFs using a specific protocol (Step 1, lines 13-16). Since the session-ID headers are used instead of subscriber-ID headers, message exchanges must carry the same value of session-ID in all messages. However, messages exchanged between NFs also carry routing or NF-identifying headers which repeat frequently.

These header names are eliminated from the $candSubHdrN$ (Step 2, lines 17-20). Table 5.2 lists sample results that confirm that correct header names were inferred by the algorithm from traces for SIP and Diameter.

Algorithm 1: Candidate header analysis

```
Input  : $(traceFile, endpointProtocol)$
Output: $candSubHdrN, candSnHdrN$
/* Find candidate user header names for protocol */
for each packet $p$ in $endpointProtocol$ do
    for each header $h$ in $p$ do
        if $h.value$ conforms to subscriber_id format then
            $candSubHdrN[p.protocol] \longleftarrow candSubHdrN[p.protocol] \cup h.name$
            $p.candSubV \longleftarrow p.candSubV \cup h.value$
            $subHdrV = subHdrV \cup (h.value, p.candSubV)$
/* Find candidate user header names for all protocols */
for each value $subV$ in $subHdrV$ do
    for each packet $p$ in $traceFile$ do
        if $p.protocol \neq endpointProtocol$ then
            for each header $h$ in $p$ do
                if $h.value \approx subV$ then
                    $candSubHdrN[p.protocol] \longleftarrow candSubHdrN[p.protocol] \cup h.name$
/* Find candidate session header names */
/* Step 1.  Find header values that repeat between a pair of NFs */
for each protocol $protocol$ in all protocols do
    for each packet $p$ in $protocol$ do
        for each header $h$ in $p$ do
            $uniqueSnV[h.value] \longleftarrow uniqueSnV[h.value] \cup (h, p)$
/* Step 2.  Eliminate headers whose value repeats in messages of multiple
   users */
for each entry e in $uniqueSnV$ do
    if $|e.p| > 1$ then
        if $|e.p.candSubV| == 1$ then
            $candSnHdrN[p.protocol] \longleftarrow candSnHdrN[p.protocol] \cup e.h.name$
```

Table 5.2. *Invenio*-generated candidate header names

Protocol	Subscriber-ID		Session-ID		
	Predicted	Actual	Predicted		Actual
			Step-1	Step-2	
SIP	FROM.user, TO.user, contact.user	FROM.user TO.user	R.URI, VIA, Call-ID, CSEQ, ALLOW	Call-ID	Call-ID
Diameter	Service-Ctx-Id User-Name Sub-Id-Data, e1164.msisdn	User-Name, Sub-Id-Data	Origin-Host, Origin-Realm Ssn-ID Auth-App-Id	Ssn-ID	Ssn-ID

5.5.3 Noise Filtering

The noise filtering module eliminates messages not generated due to user actions that should not be part of M_{tt} for any transaction type. This includes (a) messages exchanged by protocols that are not specified in any transaction start message in the configuration file, (b) messages with type/name matching a message in the noise file, and (c) messages that do not carry any subscriber-ID or session-ID header.

5.5.4 Attribute Extraction

Since protocols and interfaces use different encoding formats (binary or text) to exchange information, we convert the input message stream to a protocol-agnostic intermediate format: event objects. Each event object is associated with (a) Transport-layer information (source and destination IP address and ports) to identify the NFs in the SFC, and (b) Subscriber-ID and session-ID headers from each packet extracted using the output of header inference module or configuration input.

5.5.5 Session Slicing

The session slicing module operates on the event objects generated by the attribute extraction module, and uses event information to identify all messages associated with a single user. That is, it computes $\Omega(r) = (N_r, M_r)$ for each user request. This involves correlation of session and subscriber-ID headers from event objects, and identifying all session-IDs that correspond to a single subscriber-ID. Since the input stream may contain messages and events from multiple users, this module analyzes message sequences to find the longest sequence of successful message exchanges, simultaneously merging the shorter sequences due to multiple session/subscriber-IDs.

5.5.6 Transaction Slicing

The transaction slicing module determines the NF set N_{tt} and message set M_{tt} associated with each transaction type tt. The session slicing module gives possible message sequences in $\Omega(r)$. This set can include messages from multiple users, as a single user may not generate all possible transaction types. Since messages in the trace stream are chronologically ordered, the packets in the set M_t corresponding to a certain transaction t have monotonically increasing identifiers. For example, in Fig. 5.4b, all messages from (7) INVITE to (12) 200 OK are part of the same transaction. Using the r_{start}, r_{end} messages input by service providers in the configuration file, *Invenio* slices messages of each transaction.

Each transaction type tt independently provides a service to a user u. In practice, multiple transaction types may have a strict dependence, and a service may involve invoking multiple transaction types. For example, a voice-call service requires INVITE and BYE as shown in Fig. 5.4b (messages 7-18). Such transactions are merged in *Invenio*. Sets N_{tt} and M_{tt} are then used to compute the affinity between NFs for every transaction type tt using Equation (**1**).

5.5.7 Placement

The placement engine uses affinity information generated by the affinity engine, together with input transaction type distribution and provider policies, to make the final NF placement decisions. NFs with the highest affinity values (Equation (**1**)) are co-located or placed in close proximity. In our experiments, we focus on identifying NFs that must be co-located to minimize the impact of inter-NF latency on interactive services. We have not implemented complete multi-criteria placement algorithms [25]–[27], and leave the integration of *Invenio* with a multi-criteria placement algorithm to future work.

5.6 Evaluation

We evaluate *Invenio* with two systems: (a) Clearwater: an open-source microservice-based implementation of IMS, (b) VoLTE: a prototype Voice over LTE (VoLTE) implementation in which NFs are functionally decomposed into microservice-based VNFCs (Fig. 5.7). We collect network traces from all NFs and use *Invenio* to compute affinity between NFs. We use these affinity values to decide the NF placement and evaluate the performance of

Table 5.3. Testbed configuration

Server	CPU	Cores	RAM	NFs Deployed
R430	2x Intel Xeon E5-2620 v4	16	64 GB	Clearwater
DL120	1x Intel Xeon X3430	4	8 GB	Swarm Workers, Load-Generator

two workload types in deployments where NFs are connected by high latency links. Our goal is to answer the following questions:

1. How effective is *Invenio* in computing affinity with multiple protocols? (§5.6.1, §5.6.2)
2. What influence do *Invenio*-generated affinity values have on NF placement? (§5.6.1, §5.6.2)
3. What is the impact of inter-NF latency on performance under different workloads? (§5.6.1, §5.6.2)

Implementation: *Invenio* includes ∼2600 lines of Python code. We developed a prototype microservice-based VoLTE system using Kamailio [87] version 5.0.4 as the SIP server for evaluation. We added a REST message interface to Kamailio to communicate with the PCRF. We also developed prototype implementations of the PCRF and PCEF for the VoLTE system. All REST-based components are developed as application extensions to the KORE library [88] (version 2.0.0). The PCRF and PCEF are developed as application extensions in the FreeDiameter library [89] version 1.2.1 (∼3700 lines of new C code).

Experimental testbed: Our testbed includes one Dell PowerEdge R430 and 5 HP ProLiant DL120 G6 (Table 5.3) connected by a Gigabit Dell N2024 Switch. We use Docker [53] version 17.03.0-ce and Docker-compose (v1.11.2) to deploy NFs for Clearwater (Fig. 5.5) and VoLTE (Fig. 5.7). Each NF runs within a container and all containers are deployed on the same physical host.

Workloads: We use two primary network services: (a) **Voice-call**: This service involves two transaction types (INVITE and BYE). (b) **Short Message Service (text-msg)**: This service utilizes a single transaction of type MESSAGE. We also use SUBSCRIBE (which supports the Presence service) to illustrate the impact of message-count based placement on system performance. However, SUBSCRIBE messages are not generated during performance evaluation and system performance is only evaluated for interactive workloads (voice-call and text-msg).

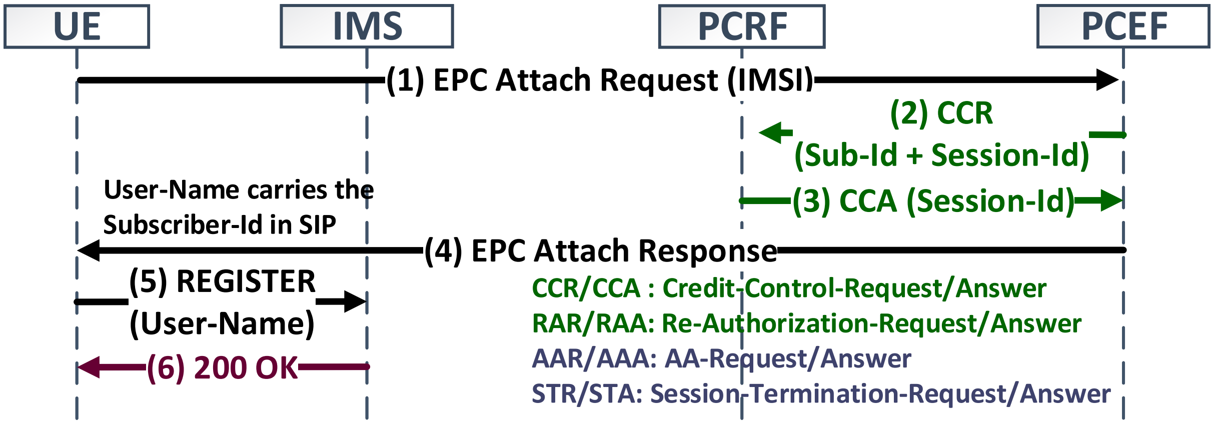
Following SIP standards, every SIP endpoint registers itself with the IMS network using a REGISTER message (Fig. 5.4a) before utilizing the voice-call or text-msg service, as depicted in Fig. 5.4b and 5.4c. In VoLTE, where the SIP messages are tunneled over the EPC network, a SIP endpoint must additionally attach itself to the EPC network before generating the REGISTER message (steps $1-4$ in Fig. 5.4a). For brevity, we only depict the communication between the IMS and EPC, and omit messages exchanged during the EPC attach [1].

Methodology: SIPp [68] is used to generate two types of workloads: voice-call and text-msg, sending four types of messages: REGISTER, INVITE, BYE and MESSAGE. Each SIPp instance runs on a dedicated physical machine and saturates available system resources. We measure failures by the observing the result code in the SIP response messages. We record the total number of successful calls or messages for each workload type. For the voice-call workload, where multiple transaction types are required to complete a call, we only count the number of calls that were successfully completed; i.e., partially completed calls are ignored. Each experiment runs for 30 seconds. Each experiment is repeated at least 5 times and results are shown with 95% confidence intervals.

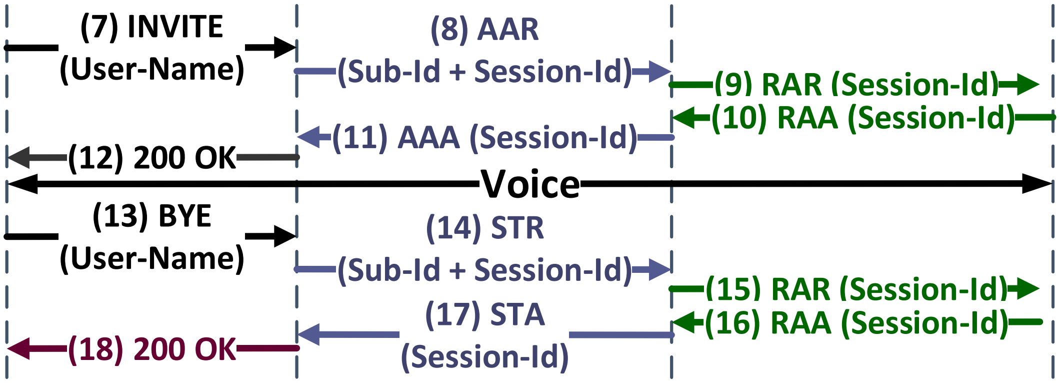
5.6.1 Clearwater Case Study

Architecture

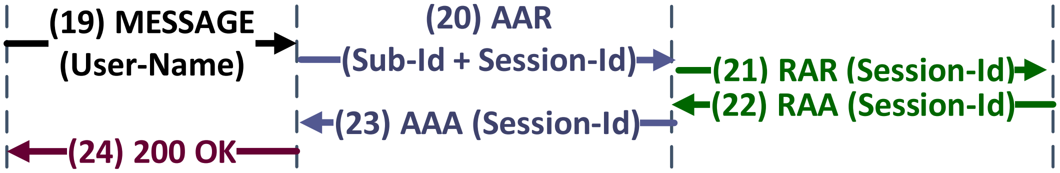
Clearwater [60] is an open-source platform for a microservice-based containerized implementation of an IMS. Clearwater uses REST-based communication to retrieve authentication vectors, manage timers and handle state synchronization, which makes it ideal for a case study. The architecture is illustrated in Fig. 5.5 (adapted from [60]). Only the components used in our experiments are depicted. We use Clearwater version 1.0 (clearwater-docker release-120). **Bono** an edge proxy that implements the P-CSCF (Proxy Call Session Con-


UE
IMS
PCRF
PCEF
(1) EPC Attach Request (IMSI)
(2) CCR
(Sub-Id + Session-Id)
(3) CCA (Session-Id)
User-Name carries the
Subscriber-Id in SIP
(4) EPC Attach Response
(5) REGISTER
(User-Name)
(6) 200 OK
CCR/CCA : Credit-Control-Request/Answer
RAR/RAA: Re-Authorization-Request/Answer
AAR/AAA: AA-Request/Answer
STR/STA: Session-Termination-Request/Answer

(a) EPC and IMS registration in VoLTE



(b) Successful voice-call in VoLTE



(c) Successful text-msg in VoLTE

Figure 5.4. VoLTE workloads

trol Function (CSCF)) in the 3GPP IMS architecture [3]. SIP clients communicate with Bono over UDP/TCP connections. **Sprout** implements the Registrar, I/S-CSCF (Interrogating/Serving CSCF) and Application Server components. **Homestead** provides a REST interface to Sprout for retrieving authentication vectors and user profiles. **Chronos** is a distributed, redundant, reliable timer service. Bono and Sprout report chargeable events to the Charging Trigger Function **Ralf**.

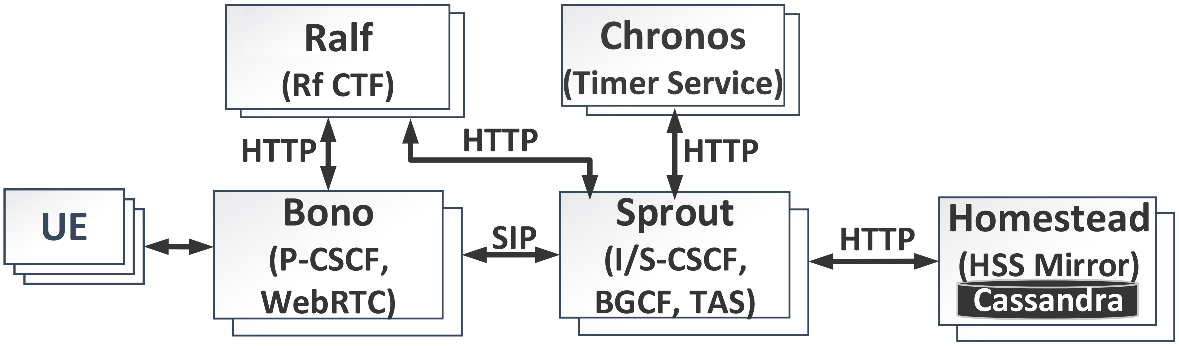


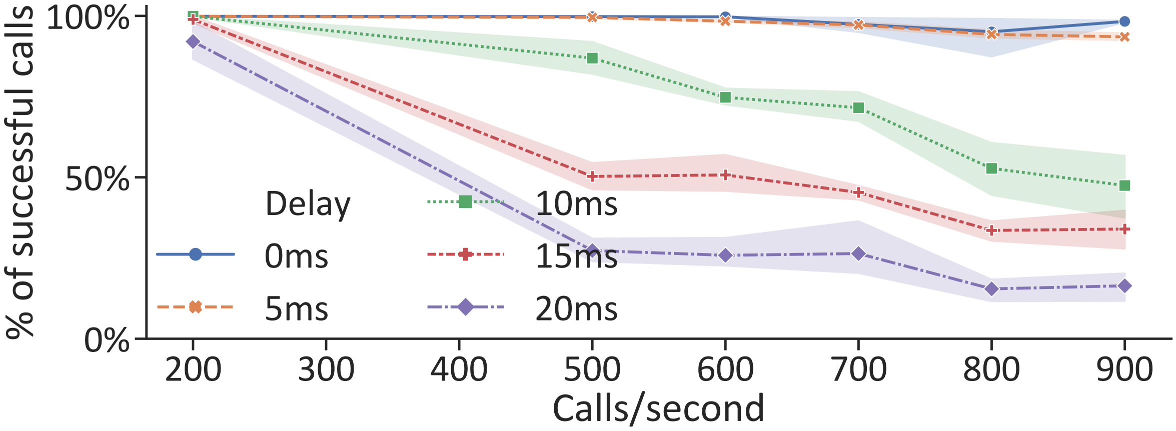
Figure 5.5. Clearwater architecture

Table 5.4. NF affinity for Clearwater

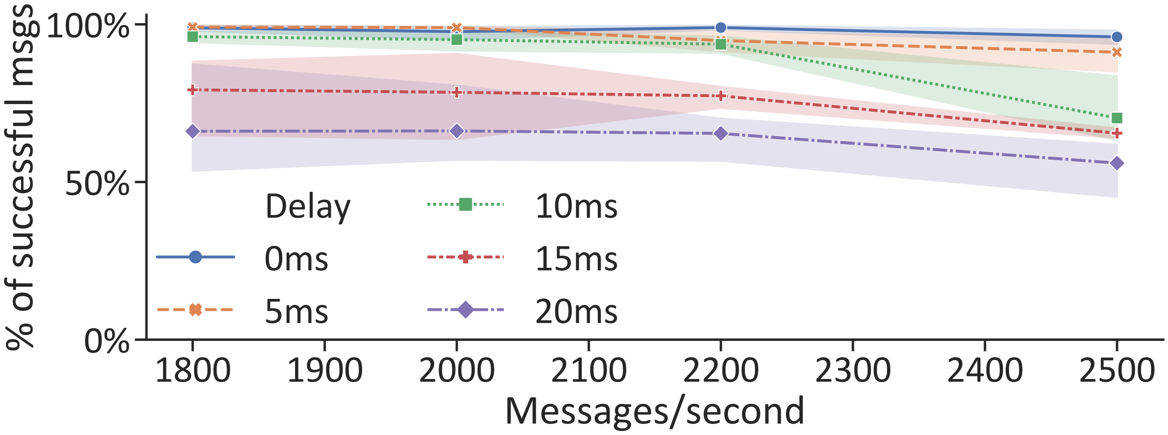
Transaction Type (tt)	NF Pair (n_x, n_y)	Affinity c(tt, n_x, n_y)
NFs: Bono, Sprout, Ralf		
Voice-call	Bono, Sprout	10
	Bono, Ralf	8
	Sprout, Ralf	4
Text-msg	Bono, Sprout	4
	Bono, Ralf	2
	Sprout, Ralf	0

Affinity Analysis

We use *Invenio* to compute affinity between NFs in Clearwater for both voice-call and text-msg workloads. The results are presented in Table 5.4. We observe that the affinity between Clearwater NFs is different for voice-call and text-msg traffic. For instance, for voice-call traffic, there is high affinity between Bono, Sprout and Ralf, whereas for text-msg traffic, Bono and Ralf only exchange two messages, and no messages are exchanged between Sprout and Ralf. Ralf therefore has a higher affinity with Bono and Sprout for voice-call workload compared to text-msg workload. The placement of Ralf w.r.t. to Bono and Sprout thus has a higher impact on the performance of voice calls compared to the text-msg workload.



(a) Voice-call performance



(b) Text-msg performance

Figure 5.6. Impact of latency and affinity on Clearwater

Performance

We first benchmark the performance of the voice-call and text-msg workloads with negligible delay. These results serve as baselines and are labeled "ideal" in our plots. We then use "tc" to introduce latency on links connecting two NF pairs (a) Ralf to Sprout and (b) Ralf to Bono, to validate the impact of placement of Ralf on performance. We experiment with delays of 5 ms, 10 ms, 15 ms, and 20 ms and compare to the "ideal" case. Fig. 5.6a and 5.6b present the results of voice-call and text-msg workloads, respectively. We make the following observations from the figures: (a) Even a single high-latency link can result in significant performance degradation for both voice-call and text-msg workloads, and (b) Performance degradation for the voice-call workload (in which Ralf has higher affinity) is more than for the text-msg workload (in which Ralf has lower affinity). These observations underscore the need for careful VNFC placement. While manual analysis shows that Sprout and Bono (which collectively implement the functionality of the CSCF) must always be co-located, analysis of IMS standards does not suffice for proprietary IMS implementations such as Clearwater in which internal implementation determines the affinity values between VNFCs (Bono/Sprout and Ralf).

5.6.2 Microservice-based VoLTE

Fig. 5.7 shows the architecture of a VoLTE system which includes (a) a SIP server that handles SIP/IMS signaling from the endpoints, referred to as the Application Function (AF), (b) a PCRF that allocates QoS rules to a user, (c) a PCEF that enforces QoS rules per user, and (d) a SUB module that provides Presence functionality. The messages exchanged for voice calls and text messages are presented in Fig. 5.4b and Fig. 5.4c, respectively.

Our VoLTE implementation can be deployed in multiple configurations and is used to study the impact of microservice decomposition and placement on system performance. In the experiments in the rest of the chapter, we treat the VoLTE implementation as a whitebox system. That is, we analyze the call flows manually to validate the *Invenio* output. In contrast, the Clearwater IMS implementation was treated as a blackbox where we did not

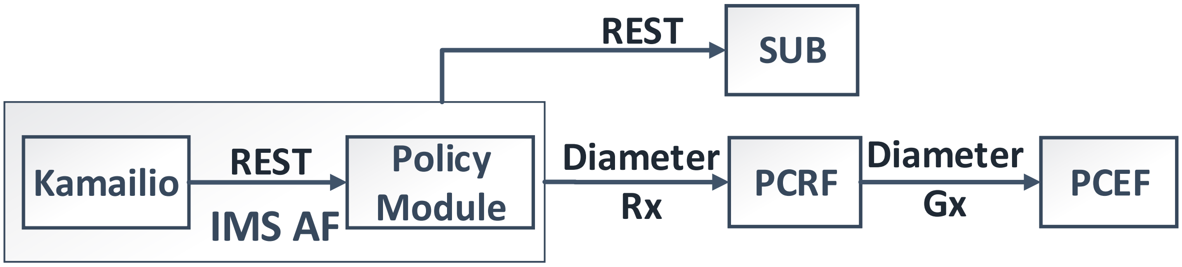


Figure 5.7. VoLTE system architecture

analyze the call flows but used *Invenio* affinity to study the impact of latency on network services.

Microservice decomposition: We decompose the PCRF into independent microservice components and deploy each microservice as an independent VNFC. Fig. 5.8 shows the architectures we use to deploy the EPC PCRF. In the legacy PCRF model (monolithic design) depicted in Fig. 5.8(a) (top left), all components are deployed in a single process and communicate via API calls. This communication is not externally observable, and the entire PCRF is deployed as a single NF. In contrast, in the microservice designs depicted in Fig. 5.8(b,c,d), the PCRF functionality is collectively provided by the VNFCs described in Table 5.5. All VNFCs expose a synchronous REST interface for external communication, so communication between the VNFCs is externally observable.

Table 5.5. Functionality of NFs in PCRF

VNF	VNFC	Interface	Functionality
PCRF	PCRF-Base	REST, Rx [90], Gx	Diameter [2] protocol functionality and interface with other VNFCs
	Gx-App	REST	Process Gx Interface messages
	Rx-App	REST	Process Rx Interface messages
	SDP	REST	Process Session Description Protocol (SDP) [91] payload
	PCRF-Gx-Base	Gx, REST	Diameter functionality and interface with other Gx VNFCs
	PCRF-Rx-Base	Rx, REST	Diameter functionality and interface with other Rx VNFCs
	memcached	REST	Gx-App and Rx-App synchronization

Table 5.6 compares PCRF μService Design-1, 2 and 3. We observe that in μService Design-1 (Fig. 5.8(b)), PCRF-Base communicates with all other VNFCs and, consequently, has affinity with all other VNFCs. In μService Design-2 (Fig. 5.8(c)), the PCRF-Base only has affinity with Rx-App and Gx-App. The SDP VNFC only has affinity with the Rx-App, and the Gx-App has affinity with memcached. In μService Design-3 (Fig. 5.8(d)), the

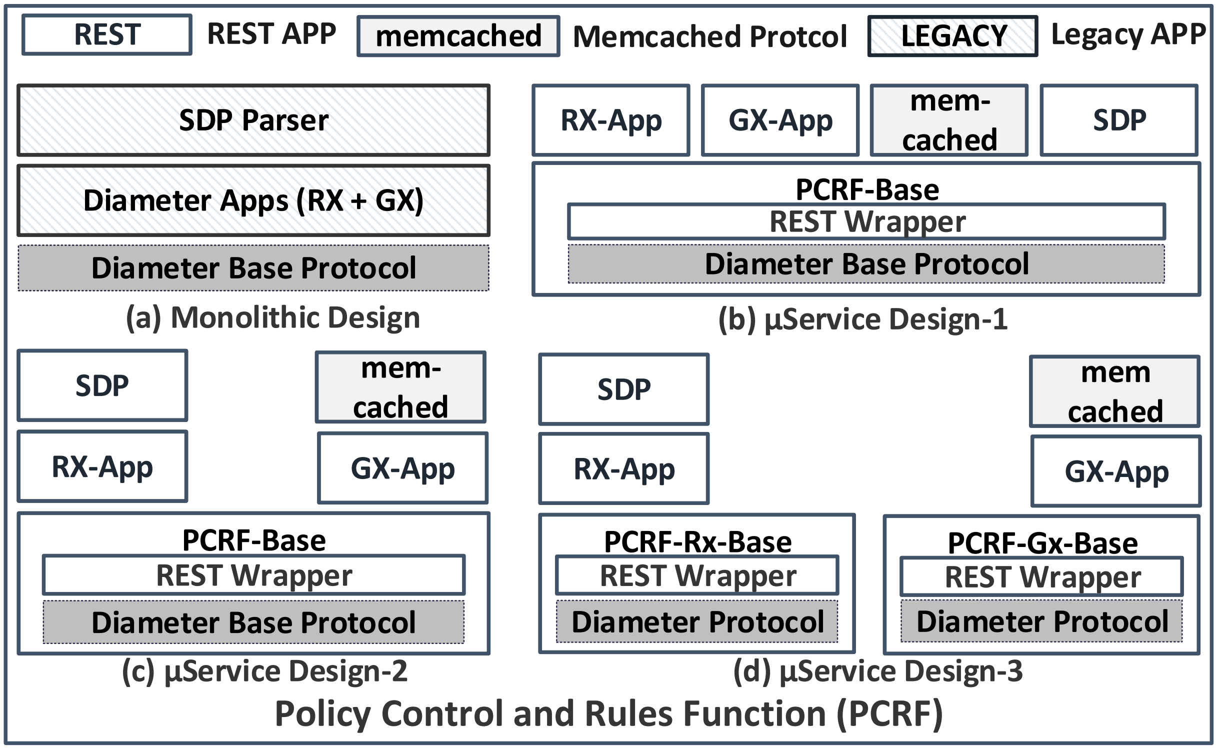


Figure 5.8. PCRF architectures

PCRF-Base VNFC is decomposed into PCRF-Rx-Base and PCRF-Gx-Base, which only have affinity with Rx-App and Gx-App VNFCs, respectively.

Affinity Analysis

We collect traffic traces (tcpdump) of voice-call and text-msg traffic with PCRF deployed in three configurations (μService Design-1, μService Design-2, and μService Design-3). Table 5.7 gives the results.

We make two observations from the results: (1) Affinity differs for the voice-call and text-msg traffic. For instance, in μService Design-1, PCRF-Base exchanges two messages with the SDP VNFC in case of voice-call traffic, but the SDP VNFC is not involved in the processing of text-msg traffic, and (2) Affinity differs in the three designs. For instance, for voice-call traffic, in μService Design-2 there is affinity between the PCRF-Base and SDP VNFC. In contrast, in μService Design-2, PCRF-Base only communicates with Rx-App and

Table 5.6. Manual analysis of communication between VNFCs in PCRF μservice designs

PCRF Decomposition	**PCRF Microservices (VNFCs)**				
		Gx-App	Rx-App	SDP	memcached
μS Design-1	PCRF-Base	✓	✓	✓	✓
μS Design-2	PCRF-Base	✓	✓	✗	✗
	Gx-App	NA	✗	✓	✓
	Rx-App	✗	NA	✓	✗
μS Design-3	PCRF-Gx-Base	✓	✓	✗	✗
	PCRF-Rx-Base	✗	✓	✗	✗
	Gx-App	NA	✗	✓	✓
	Rx-App	✗	NA	✓	✗

there is no affinity between the PCRF-Base and SDP VNFC. Affinity is only listed for the monolithic and μService Design-3 for the text-msg workload since others are similar. The SUB VNFC does not involve PCRF, and is omitted from Table 5.7. We compare *Invenio* results with the results of our manual analysis in Table 5.5 and verify that *Invenio* accurately identifies the transactions and affinity values for all transaction types.

Placement

To study the impact of *Invenio*-generated affinity values on placement, we deploy VoLTE with PCRF μService Design-3 on a Docker Swarm [92] cluster with three worker nodes. Each worker node is allocated a maximum of 4 VNFCs by the orchestrator. Fig. 5.9(a) (top) shows an ideal placement on this cluster. AF is deployed as two VNFCs (Kamailio and Policy-Module), which are always co-located, so we only show it as AF. Fig. 5.9(b) shows placement with *Invenio*-generated affinity values. The affinity values and resulting constraints are given to the Swarm orchestrator by the "affinity_group" [93] parameter in the Docker-compose configuration file. Fig. 5.9(c) shows the result of an instantiation in which the number of messages exchanged between VNFCs is used make the placement decision, as discussed by Sampaio et al. [27]. This results in a placement where VNFCs that exchange highest number of messages – (AF, SUB, and memcached) as seen from the left side of Fig. 5.2 – are co-located. Any transaction type distribution which has at least 75% presence traffic (lower than the 90% in busy-hour IMS traffic in [77]) will result in the same placement. Fig. 5.9(d)

Table 5.7. NF affinity for VoLTE

Transaction Type (tt)	NF Pair (n_x, n_y)	Affinity c(tt, n_x, n_y)
Monolithic Design, NFs: AF,PCRF,PCEF		
Voice-call	AF, PCRF	4
	PCRF, PCEF	4
Text-msg	AF, PCRF	2
	PCRF, PCEF	2
μService Design-1, AF, PCRF-Base, Gx-App, Rx-App, SDP, memcached, PCEF		
Voice-call	AF, PCRF-Base	4
	PCRF-Base, Gx-App	4
	PCRF-Base, Rx-App	4
	PCRF-Base, SDP	2
	PCRF-Base, memcached	4
	PCRF-Base, PCEF	4
μService Design-2, AF, PCRF-Base, Gx-App, Rx-App, SDP, memcached, PCEF		
Voice-call	AF, PCRF-Base	4
	PCRF-Base, Gx-App	4
	PCRF-Base, Rx-App	4
	Gx-App, memcached	4
	Rx-App, SDP	2
	PCRF-Base, PCEF	4
μService Design-3, AF, PCRF-Rx-BASE, PCRF-Gx-Base, Gx-App, Rx-App, SDP, memcached, PCEF		
Voice-call	AF, PCRF-Rx-Base	4
	PCRF-Rx-Base, Rx-App	2
	PCRF-Rx-Base, PCRF-Gx-Base	2
	Rx-App, PCRF-Gx-Base	2
	Rx-App, SDP	2
	PCRF-Gx-Base, Gx-App	4
	Gx-App, memcached	4
	PCRF-Gx-Base, PCEF	4
Text-msg	AF, PCRF-Rx-Base	2
	Rx-App, PCRF-Gx-Base	2
	PCRF-Rx-Base, Rx-App	2
	PCRF-Gx-Base, Gx-App	2
	Gx-App, memcached	2
	Rx-App, SDP	0
	PCRF-Gx-Base, PCEF	2

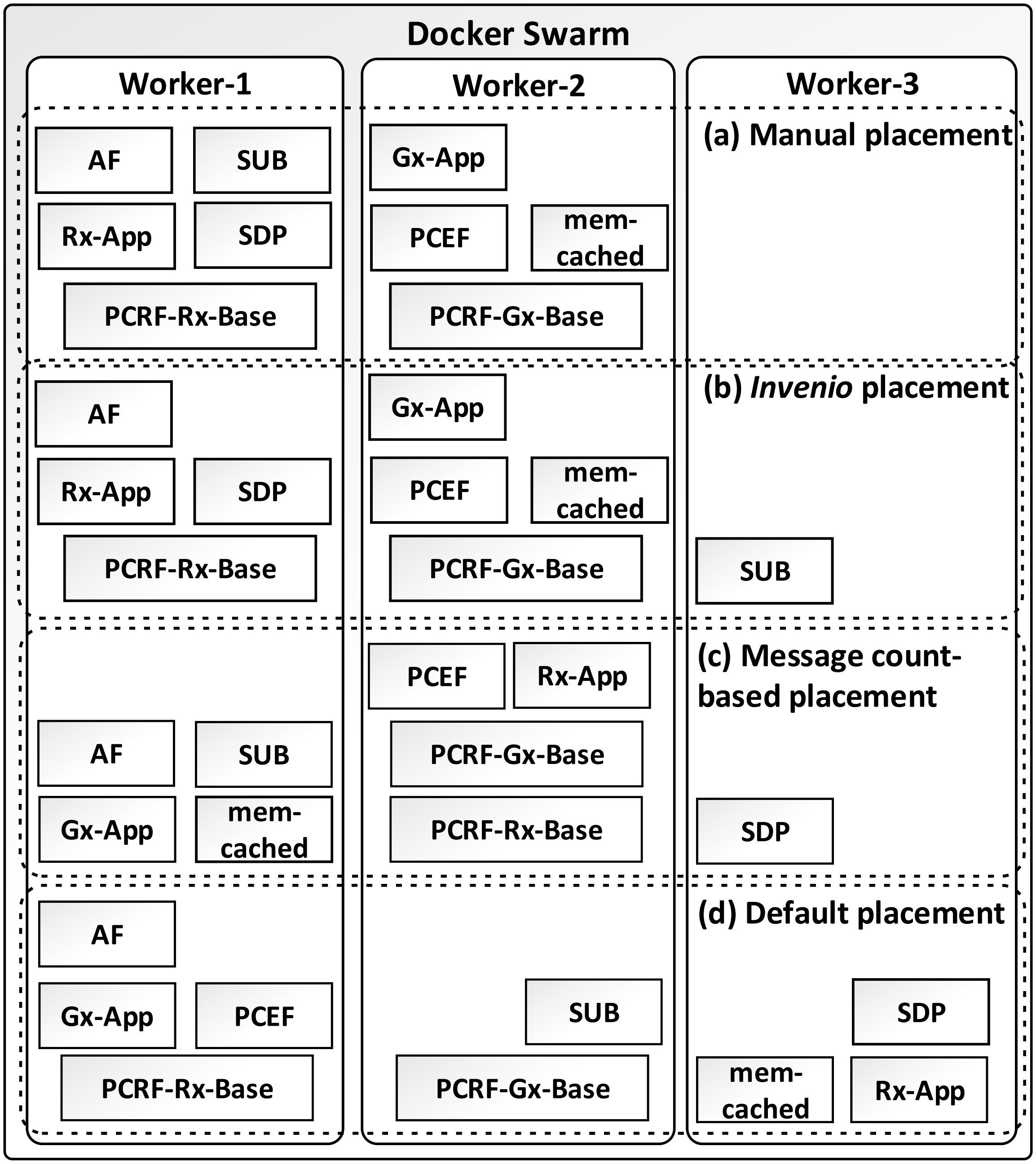


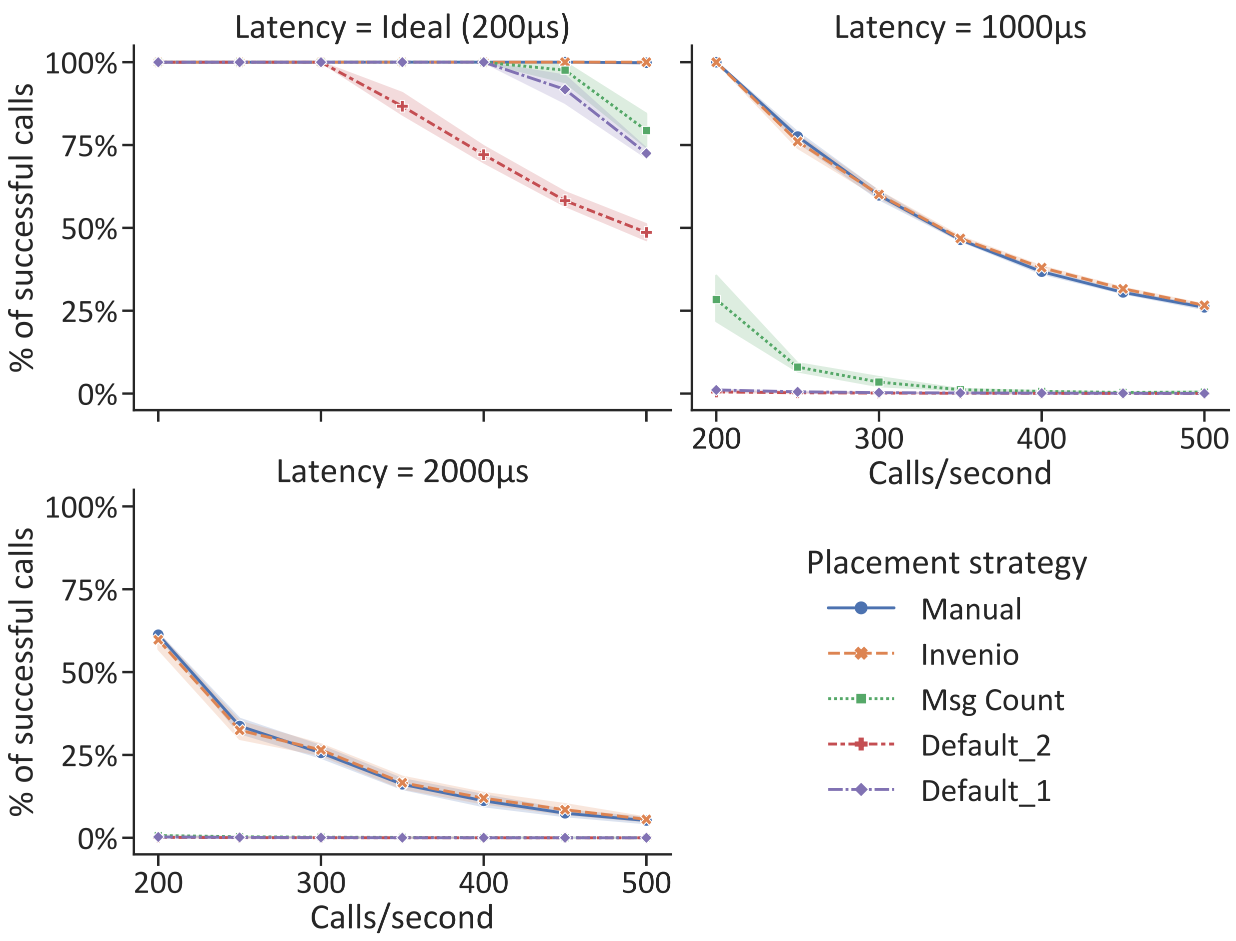
Figure 5.9. VoLTE placement with *Invenio*

depicts the results of an instantiation with no constraints given to the Swarm orchestrator. This results in a random placement of NFs (labeled "Default" in Fig. 5.9).

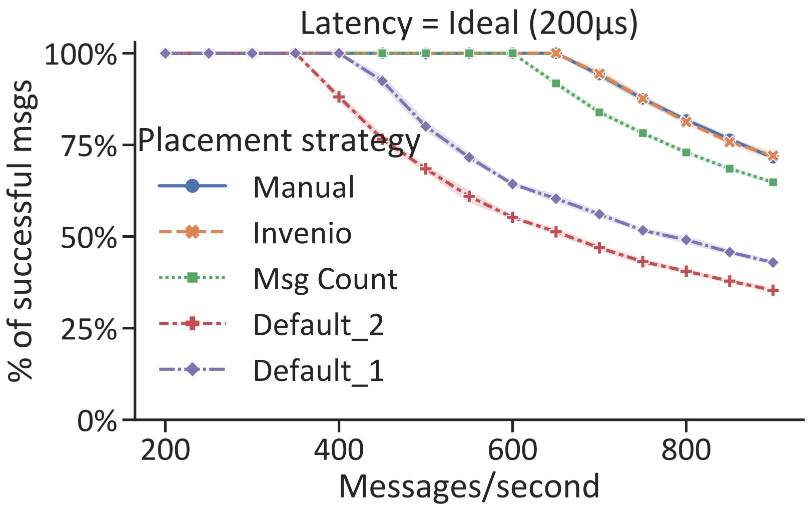
Performance

We evaluate the performance of the four placement strategies shown in Fig. 5.9 using the two workloads (voice-call and text-msg) and three different inter-NF latencies, where the Docker worker nodes are connected by 200 μs (ideal), 1000 μs, and 2000 μs links. Fig. 5.10 shows the results of two different outcomes of the default placement strategy – labeled Default 1 (which corresponds to Fig. 5.9(d)) and Default 2 with (PCEF, PCRF-RX) on Worker-1, (AF, SUB, SDP, memcached) on Worker-2, and (PCRF-GX, GX-App, Rx-App) on Worker-3. We choose PCRF μService Design-3 here as it completely decomposes the NFs into constituent microservices and therefore it is ideal for demonstrating the impact of placement.

Fig. 5.10a and 5.10b show that (1) *Invenio* closely matches the results of manual (handcrafted) placement for both voice-calls and text-msg workloads. The results of text-msg workload with 1000 μs, and 2000 μs follow similar trends as the voice-call workloads, and are omitted for brevity, (2) Both message count-based and default (random) placement strategies experience significant performance degradation as the inter-worker (inter-rack) latency increases. The impact of latency is insignificant at lower call rates, but there is significant drop in the overall system throughput as the call rate approaches system capacity, (3) The performance degradation for the voice-call workload is higher than the performance degradation for the text-msg workload. For example, comparing default placement under ideal conditions (inter-rack latency of 200 μs) to *Invenio* placement, for voice-call workload of 500 calls/second, nearly 52% calls are dropped but for text-msg workload of 900 calls/second, less than 36% of calls are dropped. This is a consequence of higher affinity between the NFs shown in Table 5.7. (4) Default placement – as seen from the results of Default 1 and Default 2 – can lead to significant variation in system performance, complicating capacity planning for microservice-based applications. *Invenio* placement clearly outperforms



(a) Voice-call performance with 200 μs, 1000 μs, and 2000 μs latency between workers



(b) Text-msg performance

Figure 5.10. Impact of latency and affinity on VoLTE performance

count-based and default placement while avoiding the non-deterministic outcomes inherent to count-based and default placement.

5.7 Other Applications

In addition to its use for affinity computation, *Invenio* can be extended for interoperability checking and fault diagnosis.

NF interoperability: The *Invenio* "Session Slicing" module (§5.5.5) outputs the sequence of messages processed by individual NFs, which can be used to construct a partial state machine. This state machine can be used to check interoperability between NFs from different vendors or of different versions. As an example, we used *Invenio* to check interoperability between the Home Subscriber Server (HSS) in Openair-cn [94] and the Mobility Management Entity (MME) in OpenEPC [95]. We configured *Invenio* to extract Diameter [2] messages exchanged between the MME and HSS in the network traces collected from OpenEPC and Openair-cn. By comparing these messages, we found that the OpenAir HSS expects a Session-Id Attribute Value Pair (AVP) in all messages, which is not provided by the OpenEPC MME. Proprietary message formats can also be supported. We developed wireshark dissectors for the EPC implementation in [96], which uses proprietary message formats, and successfully used *Invenio* to compare the state machine of this implementation with NFs in Openair-cn.

Fault diagnosis: *Invenio* can be used to diagnose scenarios when configuration or implementation problems at a specific NF result in service interruptions (failures or delays) for one or more users. In production environments where NFs process thousands of messages per second, identifying which NF in an SFC triggered a problem can be tedious and time-consuming, and requires a service provider to understand all protocols and message formats. An important step in identifying failures for a specific user is isolating messages (across an entire SFC) associated with a given user. This can be achieved by the "Session Slicing" module. *Invenio* can then report failed transactions and associated messages M_T.

5.8 Chapter Summary

In this chapter, we presented *Invenio*, a system that enables service providers to better manage the ever-growing complexity of microservice-based network functions (NFs). We showed that *Invenio* automatically computes the correct transactional affinity between NFs, and allows service providers to make and update NF placement decisions without the time-consuming and error-prone manual analysis currently used. Our experiments with IMS and VoLTE implementations confirm that, by using transactional affinity-based NF placement, service providers can effectively support services with stringent latency requirements, including 5G services.

6. NASCENT: CONTAINER-DRIVEN DEPLOYMENT OF VOLTE CALLER-ID VERIFICATION

Caller-ID spoofing deceives the callee into believing a call is originating from another user. Spoofing has been strategically used in the now-pervasive telephone fraud, causing substantial monetary loss and sensitive data leakage. Unfortunately, caller-ID spoofing is feasible even when user authentication is in place. State-of-the-art solutions either exhibit high overhead or require extensive upgrades, and thus are unlikely to be deployed in the near future. In this chapter, we seek an effective and efficient solution for 4G (and conceptually 5G) carrier networks to detect (and block) caller-ID spoofing. Specifically, we propose Nascent, Network-assisted caller ID authentication, to validate the caller-ID used during call setup which may not match the previously-authenticated ID. Nascent functionality is split between data-plane gateways and call control session functions. By leveraging existing communication interfaces between the two and authentication data already available at the gateways, Nascent only requires small, standard-compatible patches to the existing 4G infrastructure. We prototype and experimentally evaluate three variants of Nascent in traditional and Network Functions Virtualization (NFV) deployments. We demonstrate that Nascent significantly reduces overhead compared to the state-of-the-art, without sacrificing effectiveness.

6.1 Introduction

Vulnerabilities in widely-deployed packet-based telecommunications services have raised serious concerns about the security of current infrastructure [97]. A simple (and now pervasive) type of attack that exploits 4G Voice over LTE (VoLTE) vulnerabilities is the caller-ID spoofing attack [98], where an attacker impersonates another user by spoofing their telephone number or user name. An unsuspecting user may be deceived by the spoofed caller-ID displayed by their user equipment (UE) since this ID can correspond to a trusted organization such as a government agency [99]. Telemarketers also often use caller-ID spoofing to avoid detection by caller identification systems (e.g., Truecaller [100]), and trick users into receiving marketing calls. A recent phenomenon, neighbor spoofing [98], [101], uses a caller-ID that closely matches the receiver telephone number.

While caller-ID spoofing attacks were difficult to mount on traditional circuit-switched networks, the proliferation of SIP-based VoLTE services and easy access to caller-ID spoofing applications (e.g., SpoofCard [102] and SpoofTel [103]) have enabled an average telephony subscriber to mount such attacks, leading to losses in the billions of dollars [101].

Fundamentally, the caller-ID spoofing attack stems from a well-known vulnerability in the IP Multimedia Subsystem (IMS). Traditional IMS servers designed for Voice over IP (VoIP) do not validate the subscriber identifier in incoming call setup requests, which allows an attacker to impersonate other subscribers. Even if IMS servers can validate the caller-ID of incoming calls, the IMS network alone does not have sufficient information to validate the caller-ID [50]. In the case of VoLTE, a user is initially authenticated, but the identity indicated in the call setup requests arriving later is not validated by the IMS.

Several solutions have been proposed to tackle caller-ID spoofing. These include network-assisted authentication using shared secrets and cryptographic encryption [4], end-to-end certificate authentication [45], [65], [104], [105], challenge-response authentication (between caller and callee) [44], and call-back validation [43], [106]. Unfortunately, these solutions suffer from several drawbacks. Encryption-based solutions require additional message exchange with endpoints and expensive encryption. Certificate-based authentication requires additional infrastructure to manage and validate certificates. Call-back systems generate a validation call towards the caller-ID of an incoming call, effectively doubling the signaling workload. All endpoint-only approaches suffer from the problems that endpoints cannot always be trusted, and that a massive number of endpoints would need upgrade. These drawbacks ultimately make current solutions ineffective or infeasible to deploy. This leads us to focus our attention on designing *network-assisted solutions that are efficient and easy-to-deploy.*

We design a network-assisted approach to detect caller-ID spoofing, Nascent (Network-assisted caller ID authentication). By sharing intelligence between the Evolved Packet Core (EPC) and IMS networks, carriers can efficiently and effectively detect caller-ID spoofing at runtime, without requiring major infrastructure deployment or endpoint upgrades. We leverage subscriber data already available to EPC control-plane functions, but cross validate the caller-ID of an incoming voice call *at the IMS* to reduce the overhead on the EPC data-plane. We make the following contributions:

1. We propose NASCENT, a new lightweight spoofing detection approach that is easy-to-deploy in 4G and beyond.
2. We develop prototypes of three variants of NASCENT.
3. We experimentally evaluate the performance of NASCENT variants, and compare them to the RFC-defined proxy-to-user authentication [4] in both traditional and Network Functions Virtualization (NFV) deployments. We demonstrate that NASCENT is effective and exhibits low overhead.

The remainder of this chapter is organized as follows. In §6.2, we describe VoLTE, caller-ID spoofing, and related work. In §6.3, we compare prior network-assisted approaches to counter caller-ID spoofing. In §6.4, we discuss the design of our new approach, NASCENT, and in §6.5, we experimentally evaluate NASCENT. In §6.6, we discuss deployment of NASCENT, and §6.7 concludes the chapter.

6.2 Background and Related Work

4G LTE (and beyond) advance cellular networks to a packet-switched only infrastructure, migrating traditional circuited-switched voice support to VoLTE [107]. VoLTE carries voice traffic and its signaling in IP packets, akin to VoIP. In this section, we introduce necessary VoLTE background and explain why caller-ID spoofing is possible even with authentication in cellular networks. Finally, we summarize related work on countering caller-ID spoofing.

6.2.1 VoLTE Architecture and Call Setup.

Figure 6.1 depicts a simplified LTE network architecture and the VoLTE call setup flow. LTE provides voice service to user equipment (UEs, i.e., phones) in its core network, which consists of two main subsystems: Evolved Packet Core (EPC) and IP Multimedia Subsystem

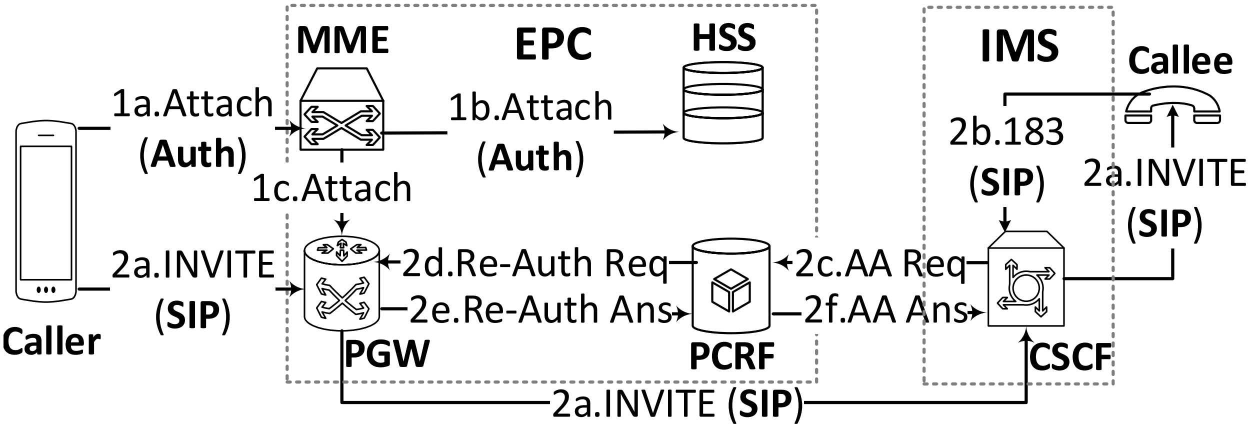


Figure 6.1. LTE network architecture and VoLTE call setup flow.

(IMS). EPC is responsible for data-plane packet delivery and its associated control functions such as the Policy and Charging Rules Function (PCRF), user authentication, and security. The Packet Data Network Gateway (PGW) is the EPC's critical network function which forwards packets and acts as the interface to other packet data networks like the Internet and IMS. The PGW typically includes the control function commonly known as the Policy and Charging Enforcement Function (PCEF), which communicates with the PCRF for quality and billing policy enforcement. The IMS offers voice and multimedia services over IP via Call Session Control Functions (CSCFs). IMS uses the Session Initiation Protocol (SIP) [4] for call setup signaling, which is the standard for VoIP.

A caller's UE must authenticate itself before making a call (step 1). User authentication is performed when the UE initially attaches to the network (e.g., powers on). Each UE's SIM card is associated with an International Mobile Subscriber Identity (IMSI) and a Mobile Station International Subscriber Directory Number (MSISDN) (telephone number), which are globally unique. A UE secret key is stored at the Home Subscriber Server (HSS), a user database. The Mobility Management Entity (MME) enforces user authentication towards the HSS, and updates authenticated UE information at the PGW. After that, the UE is authorized to make a call (step 2). To initiate a call, the UE sends a call setup request in a `SIP INVITE` message to the IMS which forwards the request to the callee. IMS later performs authentication and authorization (AA) with the PCRF (2d) and finally with the

PGW (2e) using the Diameter protocol [108]. This is needed for charging and QoS policy control. We show the signaling flow as a space-time diagram in Figure 6.3a.

6.2.2 Caller-ID Spoofing.

Caller-ID spoofing is feasible in VoLTE despite user authentication [97], [109]–[111]. The IMS and EPC use different addressing mechanisms to identify a UE. In IMS, the caller-ID is carried in the `From` header in the `INVITE` message. This header denotes the authentic caller's telephone number in the case of no spoofing. However, there is no guarantee that the forwarded caller-ID in the (`INVITE`) is exactly the same as the one which was authenticated in advance (IMSI and its true phone number) or associated with the derived one (e.g., temporary ID or the IP address allocated). In fact, real-world experiments have already validated that the current practice does not enforce any binding between SIP IDs and authenticated IDs, making users vulnerable to caller-ID spoofing [97], [109]–[111]. The root cause of caller-ID spoofing lies in the separation between user authentication and call setup signaling. Although authentication is initially executed (to authorize making a call), no mechanism prevents the caller from later altering the forwarded ID, thus hiding its authenticated ID during call setup.

Table 6.1. Comparison of network-assisted caller-ID spoofing detection solutions.

Solution	Effectiveness		Ease of deployment		Overhead			
	SIP Spoofed	SIP & IP Spoofed	Infra-structure	Standards-Compatibility	Network		Computation	Storage
					# Core	# UE		
[RFC] Proxy-to-user authentication [4]	✔	✔	None	Yes	6	6	High	Low
[RFC] TLS [4]	✔	✔	PKI	Yes	5	5	High	High
Passive validation [111]	✔	✘	Not applicable					
iVisher [50] •	✔	✔	None	No	21	0	Low	Low
Kim et al. [51]	✔	✔	None	No	0/(8 ◇)	0	High	High
NASCENT	✔	✔	None	Yes∗	0/4/6∗	0	Low	Low

• Works for VoLTE and VoIP; ◇ When stored in a remote key-value store; ∗ Depends on variant used

6.3 Design Goals and Lessons Learned

In this work, we aim to develop practical spoofing detection in carrier networks. We believe that detecting caller-ID spoofing *with network-assistance* is more effective and easier-to-deploy. This is because carrier networks are under the control of a few trustworthy service providers, which wish to protect users from ill-intended spoofing abuse, and enforce authentication and authorization, as commonly expected. In this section, we present our design goals and compare existing *network-assisted* approaches. Our objective is to understand the pros and cons of current solutions and gain insights for the design of NASCENT in §6.4.

6.3.1 Goals

An ideal network-assisted solution should be *effective*, *easy-to-deploy* and *efficient-to-run*.

(1) Effectiveness: An effective solution should detect both simplistic and sophisticated attacks. In the simplest case, the caller-ID in the `INVITE` From header is forged. An effective solution must work when the attacker spoofs other caller-IDs carried in the `From`, `To`, or `P-Asserted-Identity` fields, as well as the IP address. Note that when SIP messages are tunneled using other protocols, the source/destination IP address can be easily spoofed without impacting end-to-end packet delivery.

(2) Ease of deployment: An easy-to-deploy solution requires minimal hardware and software upgrades to the existing infrastructure. Solutions should not require (i) additional infrastructure such as PKI, or (ii) non-standard protocols or interfaces. A desirable solution should leverage existing, standard-compatible components and only require software upgrades.

(3) Efficiency: An efficient solution should exhibit low overhead in three aspects. **(i) Network overhead** refers to additional message exchanges required to support caller-ID spoofing detection. This includes: (a) Messages exchanged between network functions (NFs) within IMS or EPC, and (b) Messages exchanged between the UE and the EPC and IMS NFs. Since the EPC and IMS networks are often co-located or connected via high-speed links, message exchange between these NFs traverses fewer hops than message exchange between the UE and core network (IMS and EPC). Traversal of more hops, coupled with the latency introduced by last-mile radio links, makes message exchange with a UE more expensive. In the core, we count the logical number of messages exchanged between NFs. In practice, NFs may be connected via multiple hops, or the functionality of an NF may be collectively implemented by multiple nodes. **(ii) Computation overhead** refers to overhead of message processing, e.g., cryptographic calculations have higher overhead compared to trivial comparisons. **(iii) Storage overhead** refers to memory and disk usage. Since the precise computation and storage overhead depends on the implementation and deployment model, we only classify these overheads as high or low in Table 6.1, but they highly affect our results for both the PGW and IMS in §6.5.

6.3.2 Comparison of Existing Proposals and Lessons Learned

We compare existing network-assisted solutions in Table 6.1.

The standard (RFC 3261) [4] proposes two runtime caller-ID validation approaches: a challenge-response procedure (proxy-to-user authentication) and an encrypted channel in Transport Layer Security (TLS). Both are deemed effective but not efficient or easy-to-deploy because they require additional infrastructure, exchange additional messages with the endpoints, or involve expensive computations for decryption.

Passive validation [111] checks the caller-ID in the `INVITE` request only and thus is ineffective when the attacker spoofs both the IP address and the SIP header. For this reason, we do not consider it further. Some proposals utilize control-plane information available at network gateways to validate the caller-ID. iVisher [50] validates the caller-ID by tracing the call back to the originating gateway. While effective, iVisher requires several new messages which are not standard-compatible and thus require substantial upgrades at the gateways. An alternative solution [51] detects caller-ID spoofing by inspecting every SIP message received at the EPC gateway (e.g., PGW). This incurs high computation and storage overhead due to deep packet inspection, as the PGW is responsible for forwarding all IP packets, not just `SIP INVITE`. It is also expensive for the PGW to encode SIP protocol messages and terminate data-plane connections – operations typically performed by the CSCF – since the PGW is not SIP-aware.

Lessons learned: The above discussion sheds light on designing an effective, standard-compatible, low-overhead solution. First, the solution should leverage existing infrastructure and should purely be a software solution. Second, limiting the entire solution to a single data-path network function induces unacceptable overhead. The EPC gateway has the user authentication information needed for network-assisted validation but it lacks the context of VoLTE call setup. A gateway-only solution has a high computation cost (deep packet inspection) and resource waste (most packets are not VoLTE relevant). An IMS-only solution is infeasible since the IMS does not have authentication data to validate a caller-ID. Third, overhead of network communication with the endpoints is much higher communication within the core network, since messages to endpoints traverse lossy last-mile radio links and experience higher latency and more failures. Fourth, communication between network functions should exploit existing protocols and interfaces; otherwise, it is not standard-compatible and is more difficult to deploy (patch existing infrastructure).

6.4 Nascent Design

Based on the goals in §6.3.1, we need to design an effective, low-overhead and easy-to-deploy caller-ID spoofing detection solution that does not suffer from the drawbacks of the state-of-the-art network-assisted approaches discussed in §6.3.2.

6.4.1 Overview

Our solution, NASCENT, uses a *cross validation* approach. Unlike passive identifier validation solutions [111] that only utilize information available to the IMS servers, *cross validation* compares UE identifiers from *multiple networks*: the EPC and IMS networks in our case. The idea of cross validation stems from the availability of at least one authenticated network identifier that can be reliably used to identify a network endpoint.

We make the following key decision in designing NASCENT: *We split the caller-ID cross validation functionality among the IMS control plane and the PGW. We minimize expensive*

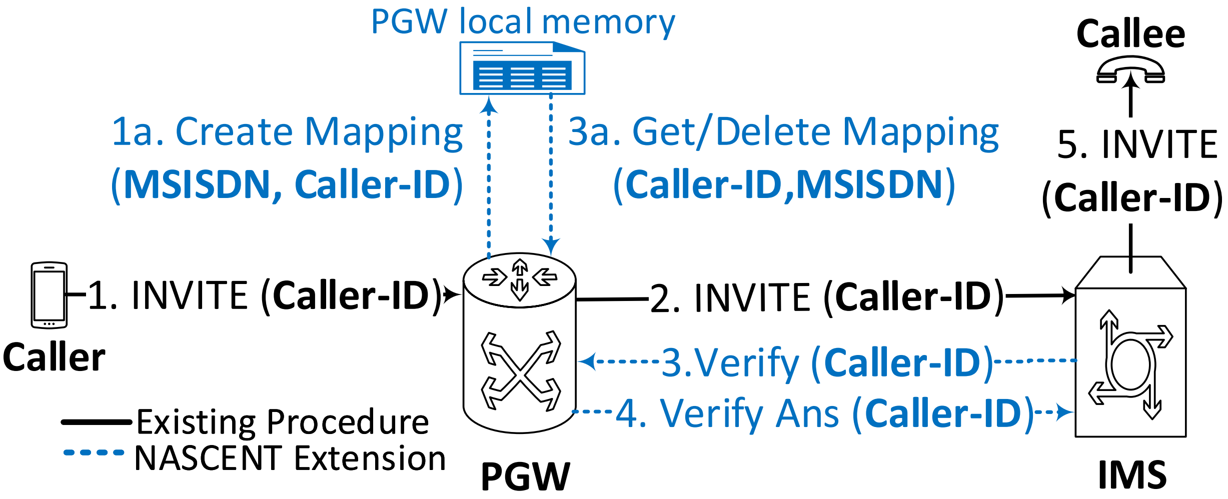


Figure 6.2. NASCENT design: 1a and 3a are used to access local memory (i.e., no messages are exchanged).

operations at the PGW, in order to reduce latency and overhead. Since IMS servers already manage and terminate SIP sessions, they require minimal changes to implement caller-ID validation. As shown in Figure 6.3a, the EPC network already supports communication between the IMS servers and EPC packet gateways [65], [80], [90]. Figure 6.2 depicts the basic idea of NASCENT. The PGW creates a mapping of the EPC identifiers (e.g., MSISDN) and IMS identifiers (e.g., SIP `Call-ID` [4], `From`) when it receives an `INVITE` message (step 1a). Before forwarding the `INVITE` request to the called UE, the IMS fetches the EPC identifier associated with the `INVITE` message (step 3 and 3a) and cross validates the caller-ID being forwarded against the MSISDN received from the EPC. Figure 6.2 depicts a simplified view of a traditional deployment. In practice, however, the EPC and IMS functions can be decomposed and deployed as multiple Virtualized Network Functions (VNFs) or can be aggregated and deployed as a single VNF, which does not impact our design.

NASCENT consists of following three components (new steps highlighted in blue in Figure 6.2):

(1) Mapping creation: The PGW monitors SIP messages generated by a UE and stores a mapping between the IMS and EPC identifiers when a SIP `INVITE` message is observed. The PGW already extracts the SIP payload from each tunneled packet and forwards this payload to the IMS servers. The PGW typically allocates a dedicated network interface

(Access Point Name (APN)) for IMS signaling messages and therefore SIP traffic can be efficiently monitored by observing traffic on this interface. The `Call-ID` header can be used by the PGW and IMS to uniquely identify a SIP message. (The actual headers/parameters used by the PGW and IMS to identify a SIP message depend on the implementation.)

The PGW will extract the SIP headers (`Call-ID`, `From`, `To`) and IP address, and save a mapping between these headers and the EPC identifiers (MSISDN, IMSI) associated with the tunnel. This is effective because the EPC network uses data tunnels to transport VoLTE signaling messages between the IMS and UE. We utilize the knowledge of tunnel identifiers associated with a UE to validate the UE identity in SIP signaling messages. The tunnel identifiers in EPC are used to transfer encrypted traffic between the PGW and UEs, and are unchanged for the duration a user session. This property of tunnel identifiers allows us to reliably associate each SIP request with a trusted identifier (MSISDN), using which runtime validation of caller-ID can be performed.

(2) Caller-ID validation: The IMS server CSCF queries the PGW for the EPC identifiers associated with a SIP `INVITE` message and validates the SIP headers (e.g., `From`, `To`) against the EPC identifiers. Since the PGW is configured to store the mapping of SIP headers and EPC identifiers, the CSCF uses the value extracted from the `INVITE` message to generate a validation request towards the PGW. The EPC network already provides well-defined, standard-compatible interfaces to communicate with IMS, and hence these interfaces can be leveraged for this operation.

(3) Mapping deletion: After replying to the CSCF, the PGW deletes the EPC and IMS identifier map for this caller-ID. Implicit deletion reduces memory requirements at the PGW since each mapping is only stored for a few milliseconds.

6.4.2 Variants

The current VoLTE architecture presents two main challenges to the design of NASCENT:

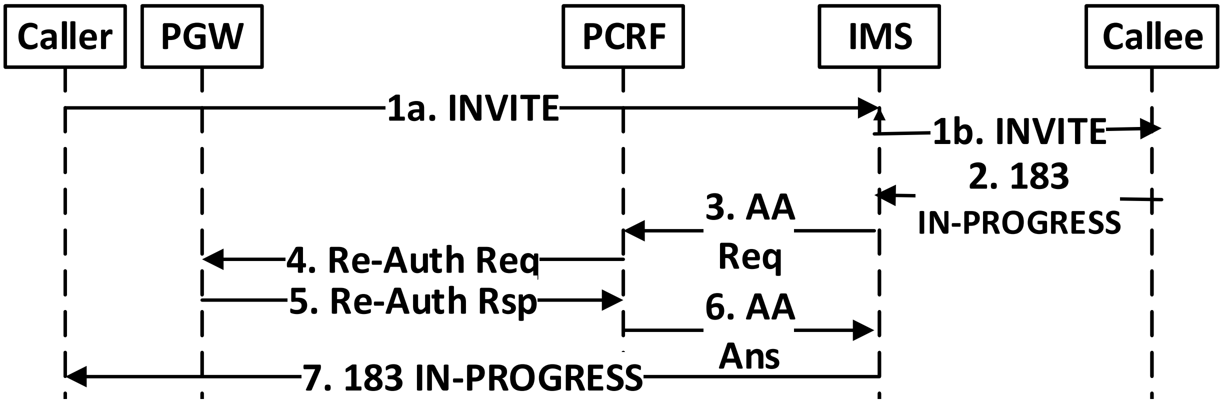
(1) The IMS AA procedure is performed after the callee is notified. As shown in Figure 6.3a, the IMS server only triggers rule generation after receiving media information from both caller and callee (from step 1a and step 2). Without additional signaling messages, the network can only detect a spoofed call after the user is notified of an incoming voice call (post-notification). Even if a spoofed call is detected and terminated by the network, the network has no means of conveying this information to the callee, and the user would still receive a "missed call." The network can convey the spoofed call notification to the user via a SIP CANCEL message used to terminate a spoofed call, and the UEs can be upgraded to support this spoofed call notification mechanism. Optionally, the operators can employ an external notification mechanism (such as SMS) to convey the spoofed call alert. If the percentage of spoofed calls in the network is relatively low, this may be acceptable.

(2) There is no direct communication between the IMS and PGW. If the network can validate the caller identity before the voice call is forwarded to the callee (pre-notification), spurious notifications can be avoided. In this case, the IMS network must query the PGW. IMS-to-PGW communication is mediated by the PCRF (Figure 6.3a). The IMS network uses the Diameter Rx interface [90] to exchange messages with the PCRF. The PCRF forwards messages to the PGW using the Diameter Gx [80] interface. A more efficient way to exchange EPC identifier information is to allow the IMS network to directly query the PGW by adding a new interface.

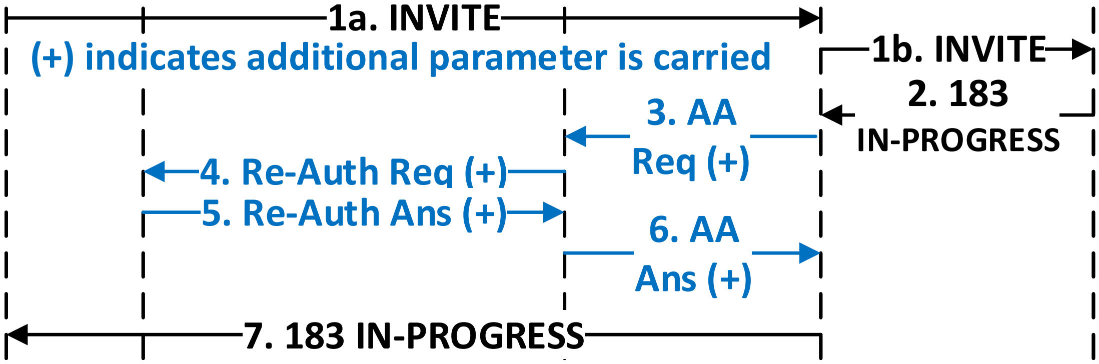
We therefore explore three alternative designs based on (a) whether the caller is validated before forwarding the voice call to the callee, and (b) if the EPC identifier information is queried using the existing Rx-Gx interface, or a new interface is added between the IMS and the PGW. These NASCENT variants are summarized as follows.

(1) Post-Notification No explicit messages are exchanged between the PGW and IMS to detect a spoofed call. The PGW provides the EPC identifiers to the IMS during the normal procedure after the user receives the voice call (Figure 6.3b). Rx and Gx messages can be modified to tunnel the additional parameters required to detect spoofing. The callee may receive a missed call notification when this variant is deployed.

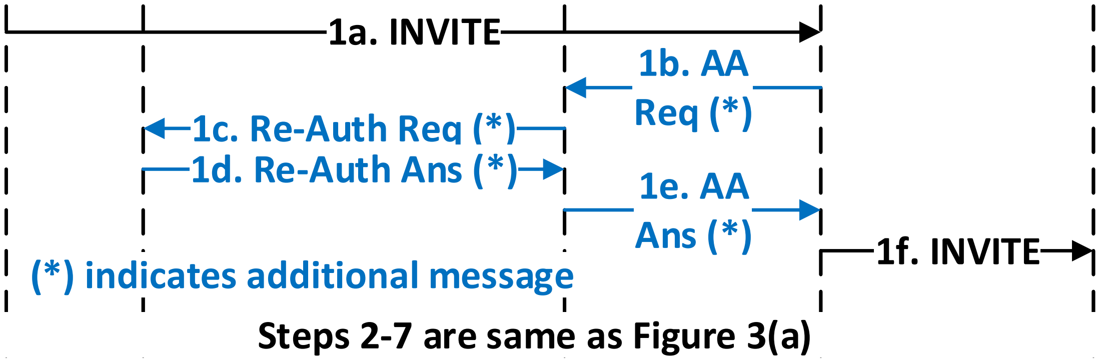
(2) Pre-Notification-Rx-Gx Caller-ID validation uses new signaling messages exchanged between the PGW and IMS prior to the `INVITE` message being forwarded to the callee. The PGW and IMS communicate using existing Rx and Gx interface messages and no new interfaces are required. Additional messages (Figure 6.3c) relayed via the PCRF incur



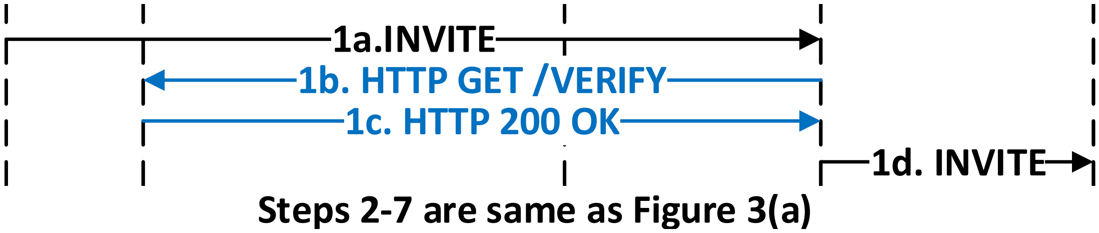
(a) Existing control plane interactions for a VoLTE call



(b) NASCENT Post-Notification



(c) NASCENT Pre-Notification-Rx-Gx



(d) NASCENT Pre-Notification-Rx+

Figure 6.3. NASCENT variants.

networking overhead but avoid maintaining additional configurations and connections at the PGW and IMS.

(3) Pre-Notification-Rx+ Caller-ID validation uses a new REST interface between the IMS and PGW. As shown in Figure 6.3d, the IMS uses this new interface to validate the caller identity before forwarding the message to the callee. This incurs configuration overhead as it requires the IMS to directly communicate with the PGW that is currently serving a user, and the IMS must therefore maintain a list of currently active PGW instances in the network.

6.4.3 Meeting Design Goals

NASCENT meets the goals of effectiveness, ease-of-deployment, and low overhead discussed in §6.3.1 as follows (see last row in Table 6.1): (a) NASCENT is effective with sophisticated spoofing attacks through its use of tunnel identifiers, (b) NASCENT does not use PKI, does not define new protocol messages and is compatible with the standards, (c) All NASCENT variants only require few additional messages, all between NFs in the core, thus exhibiting low network overhead, (d) NASCENT does not communicate with endpoints, reducing latency and overhead, (e) NASCENT only requires the PGW to provide the EPC identifiers associated with an `INVITE` message, and does not require the PGW to handle SIP request/response messages or terminate transport-layer connections initiated by the UE, thus incurring low computation overhead, and (f) NASCENT only requires the PGW to maintain each EPC and IMS identifier mapping for a brief period of time (until the call is accepted/rejected) and therefore does not require significant storage at the PGW.

6.5 Experimental Evaluation

In this section, we quantify the throughput, resource utilization, and latency incurred in VoLTE call setup.

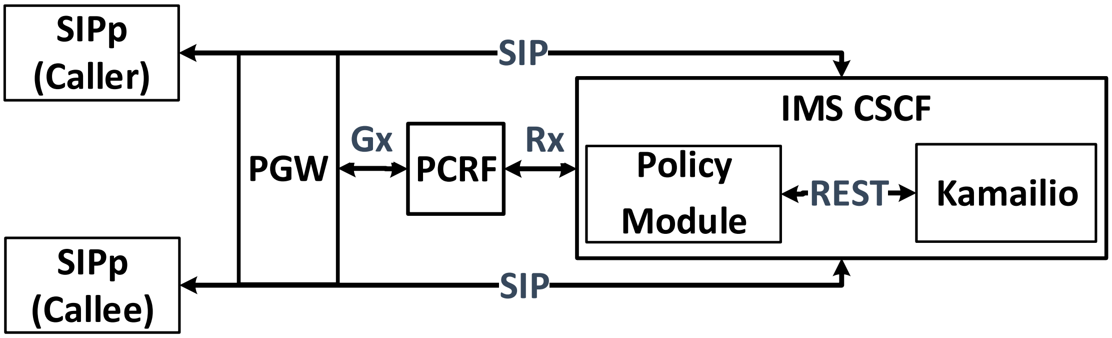


Figure 6.4. Experimental setup.

6.5.1 Implementation and Experimental Setup

We have developed a prototype of the IMS CSCF, PCRF and PGW to emulate VoLTE calls in our test environment as shown in Figure 6.4. The IMS consists of a SIP server that is used for handling SIP messages from the endpoints, and a policy module. We use Kamailio [87] version 5.0.4 as the SIP server. We extend the functionality of Kamailio to use a REST message interface to communicate with the policy module. The policy module supports REST interfaces using which Kamailio can trigger Diameter Rx Interface functions [90], [112] to communicate with the PCRF. The PCRF communicates with the PGW using the Diameter Gx [80] Interface. The policy module supports the REST interface using using the KORE library [88] (version 2.0.0). The policy module, PCRF, and PGW are developed as application extensions in the FreeDiameter library [89] version 1.2.1 using the C language (∼3700 lines of new code).

We compare proxy-to-user authentication (§6.3.2) and the proposed NASCENT variants with a baseline in which the caller-ID is not validated. We select proxy-to-user authentication as a representative network-assisted solution because (a) This approach is specified by the SIP RFC [68] and is already supported by existing implementations, and (b) Previous work [113] has found that its throughput is higher than TLS-based solutions.

In the baseline case, the PGW does not intercept SIP traffic from the UE, and Rx and Gx interface messages do not carry additional EPC identifiers. Proxy-to-user authentication is similar to the baseline case but uses additional messages to authenticate callers using the procedure defined in [4].

We use Docker version 17.03.0-ce and Docker-compose (v1.11.2) [53] to deploy and manage Virtualized Network Functions (VNFs) as shown in Figure 6.4. Each VNF runs within a container, and all containers are deployed on the same physical host: a Dell PowerEdge R430 (2x Intel Xeon E5-2620 v4) with 16 cores and 64 GB RAM.

Deployment models: We evaluate two deployment models: (a) Traditional deployment, and (b) Network Functions Virtualization (NFV) deployment. In traditional deployment, IMS and EPC are independent physical systems and no resources are shared among them. This setup emulates current deployments where IMS are EPC are deployed on separate physical machines. Kamailio is allocated a single core on CPU-1, while the policy module, PCRF, and PGW share the second CPU. This setup is used to measure the additional resources required to support the caller-ID spoofing solutions on the IMS servers. In NFV deployment, we instantiate all VNFs in Figure 6.4 on the same physical machine and configure them to share 4 cores on CPU-1. This is akin to expected 5G deployments.

Workload generation: We deploy two instances of SIPp [68], each on a separate physical machine. One SIPp instance is used as the caller and the other is used as the callee. Both caller and callee SIPp instances register the UEs with the IMS prior to the generation of `INVITE` messages. We observe the response codes received by SIPp and use them to infer the number of failures. Per the SIP specification, only `200 OK` messages indicate success and all other response codes are considered failures. The timeout for `INVITE` messages is set to 1 second; that is, an `INVITE` call is considered successful if a `200 OK` response is received within 1 second.

To generate workloads where the caller-ID is spoofed by the SIP caller, we configure SIPp to use a random value in the `From` header of the `INVITE` message. Rejection of a voice call with a spoofed caller-ID is considered a successful result. Therefore, in the plots in §6.5.2, we count `INVITE` calls rejected due to caller-ID mismatch as successful calls.

Our experiments aim at quantifying the overhead of caller-ID validation on the performance of VoLTE. In real deployment scenarios, EPC networks are over-provisioned to handle flash workloads, and therefore, rarely, if ever, reach actual capacity. We thus use simulated workloads to study NASCENT under a wide range of loads from light to heavy

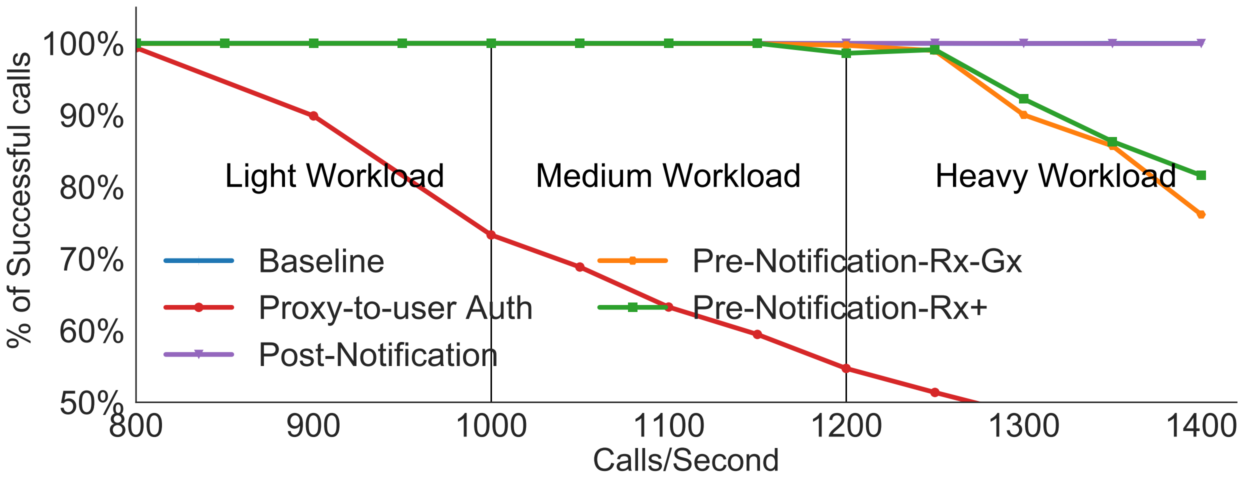


Figure 6.5. Percentage of successful calls with 0% spoofed calls.

(for stress testing). In both deployment models, we increase the workload until we saturate available system resources. We generate workloads between 800 calls/s to 2000 calls/s. We record the number of successful calls, CPU utilization, and time taken by the IMS to successfully process a voice call request, i.e., time taken to send a `200 OK` response code after an `INVITE` request is sent. We also measure the impact of the percentage of spoofed calls on performance. We generate workloads where 0-10% of `INVITE` message have a spoofed caller-ID. Each experiment runs for 30 seconds and the results represent the mean of at least 10 samples for each experiment. We also compute the standard deviation among the values. The standard deviation was within 1% of successful call percentage in the figures in §6.5.2. We will note the standard deviation for call setup latency where relevant. We use `docker stats` [53] to measure the CPU usage of VNFs. CPU usage is monitored every second and our plots represent the average CPU utilization over the experiment duration.

6.5.2 Experimental Results

Traditional Deployment Model

We begin by benchmarking the performance of the VoLTE calls in the baseline setup when no caller-IDs are spoofed. We compare the number of successful calls for each caller-ID validation solution to the baseline results.

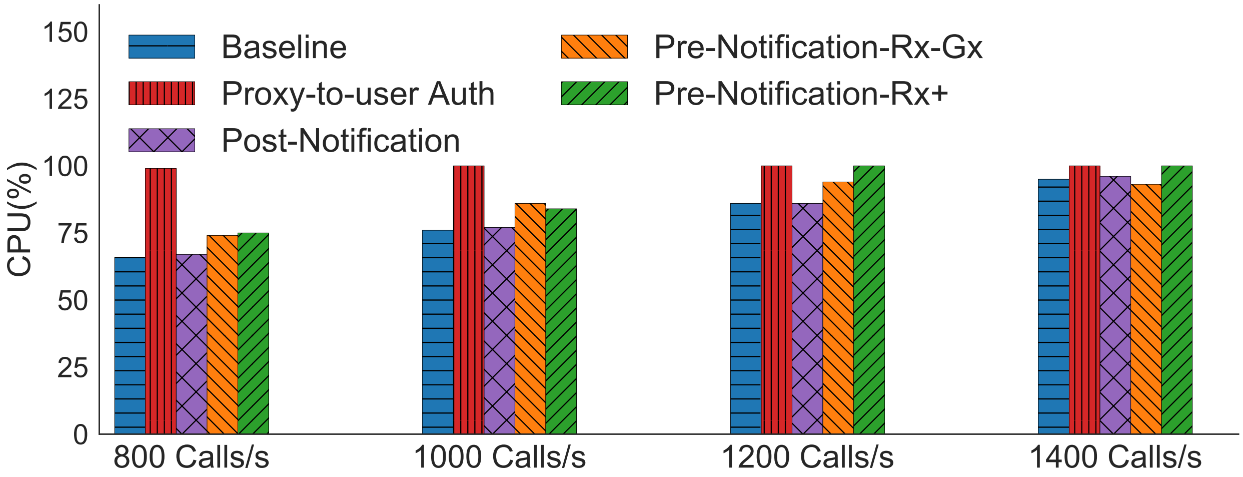


Figure 6.6. CPU utilization of IMS Server with 0% spoofed calls.

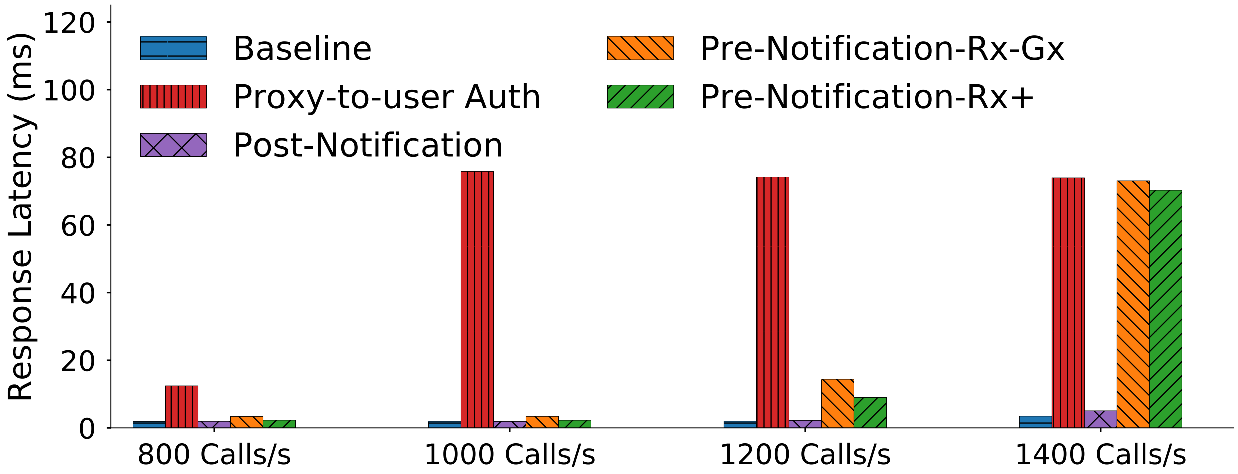


Figure 6.7. Average latency in VoLTE call setup with 0% spoofed calls.

Figure 6.5 presents the **percentage of successful calls** with 0% spoofed calls. Proxy-to-user authentication results in significant performance degradation even under light workload. The two Pre-Notification variants do not degrade performance under light and medium workloads, but increasingly degrade performance under higher workloads. Despite utilizing additional resources at the PGW, Post-Notification results in no significant performance degradation of the overall throughput at the IMS server, even under heavy workload. The performance degradation of proxy-to-user authentication (even at light workload) is a consequence of CPU saturation at the IMS server (Kamailio).

IMS CPU utilization: Figure 6.6 depicts the CPU utilization of the IMS server under four different workloads. The IMS server saturates the allocated CPU core at 800 calls/second with proxy-to-user authentication. This results in severe performance degradation as the workload increases. Nascent variants do not utilize significantly higher CPU compared to the baseline, and therefore do not result in performance degradation at light and medium workloads. At high workloads, the baseline saturates the available CPU core and therefore even Pre-Notification-Rx-Gx and Pre-Notification-Rx+ severely degrade performance.

PGW CPU utilization: Caller-ID validation solutions also require additional CPU resources at the PGW. We find that we need additional ∼15-29% CPU for the Post-Notification solution and ∼20-25% CPU for Pre-Notification-Rx+. At higher workloads, Post-Notification successfully handles a higher percentage of calls and therefore has higher CPU utilization.

Call setup latency: Since a VoLTE call is only established after a `200 OK` is received from the IMS server, any additional messages processed by IMS server will induce additional latency in the VoLTE call setup. Figure 6.7 presents these results. The standard deviation among the values representing each of the 10 individual runs is below 18 ms (below 4 ms for medium and light workloads) in this case. Proxy-to-user authentication incurs significant latency compared to the baseline. The three Nascent variants do not incur high latency at light and medium workloads. At higher workloads, as evident from Figure 6.6, CPU saturation leads to higher induced latency with Pre-Notification-Rx-Gx and Pre-Notification-Rx+. Since Post-Notification does not introduce additional messages compared to the baseline, the latency incurred is negligible.

Results with spoofed calls: Figure 6.8 and Figure 6.9 present the results at 5% and 10% spoofed calls. Comparing Figure 6.8 and Figure 6.9 with Figure 6.5, we observe that the performance of caller-ID validation improves as the percentage of spoofed calls increases. For example, Pre-Notification-Rx+ results in ∼18% call loss with 0% spoofing. However, at 5% and 10% spoofed calls Pre-Notification-Rx+ results in only ∼12% and ∼4% call drop, respectively. Since Pre-Notification rejects spoofed calls before forwarding the `INVITE` message to the caller, a higher percentage of CPU is available to legitimate calls in this case.

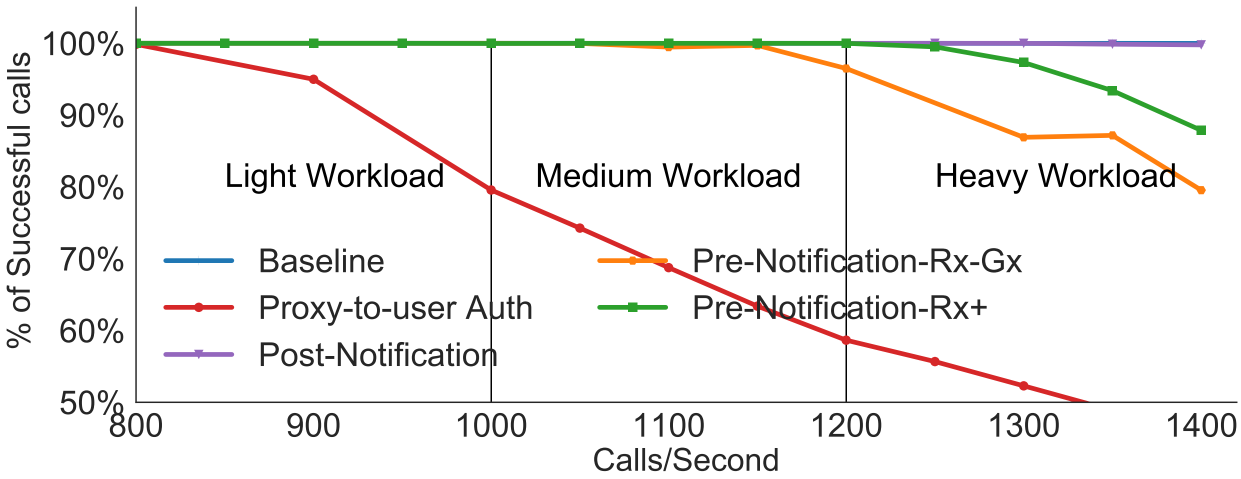


Figure 6.8. Percentage of successful calls with 5% spoofed calls.

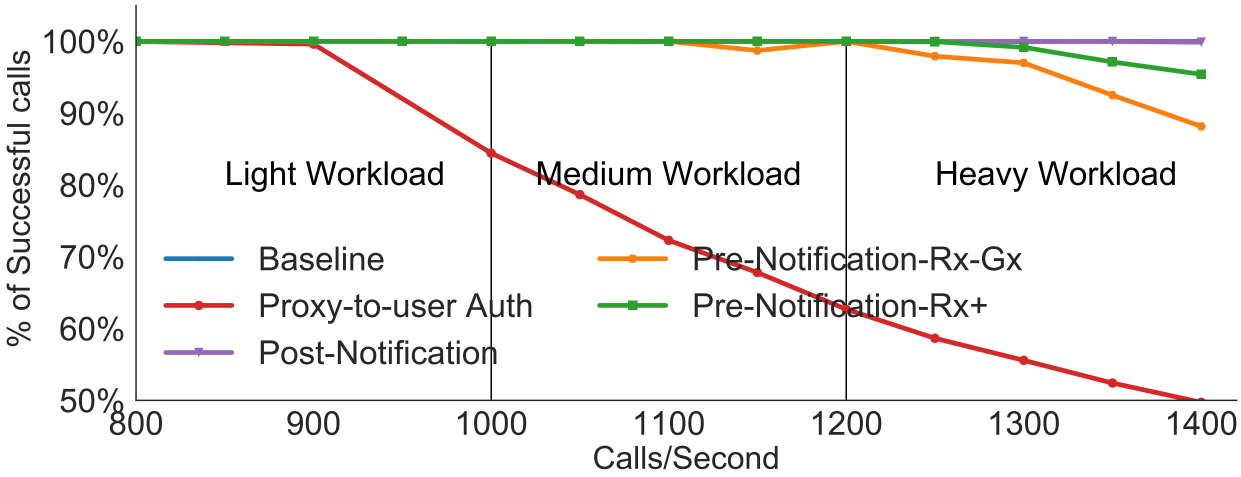


Figure 6.9. Percentage of successful calls with 10% spoofed calls.

NFV Deployment Model

We emulate a deployment where the EPC and IMS networks are instantiated on virtualized hardware platforms and are co-located to allow EPC and IMS VNFs to share system resources. This allows the IMS server to utilize more CPU resources and therefore we need higher workloads to saturate the IMS. Figure 6.10 presents the **percentage of successful calls** with 0% spoofed calls. Even with NFV deployment, proxy-to-user authentication results in significant performance degradation at light workloads. Pre-Notification-Rx-Gx

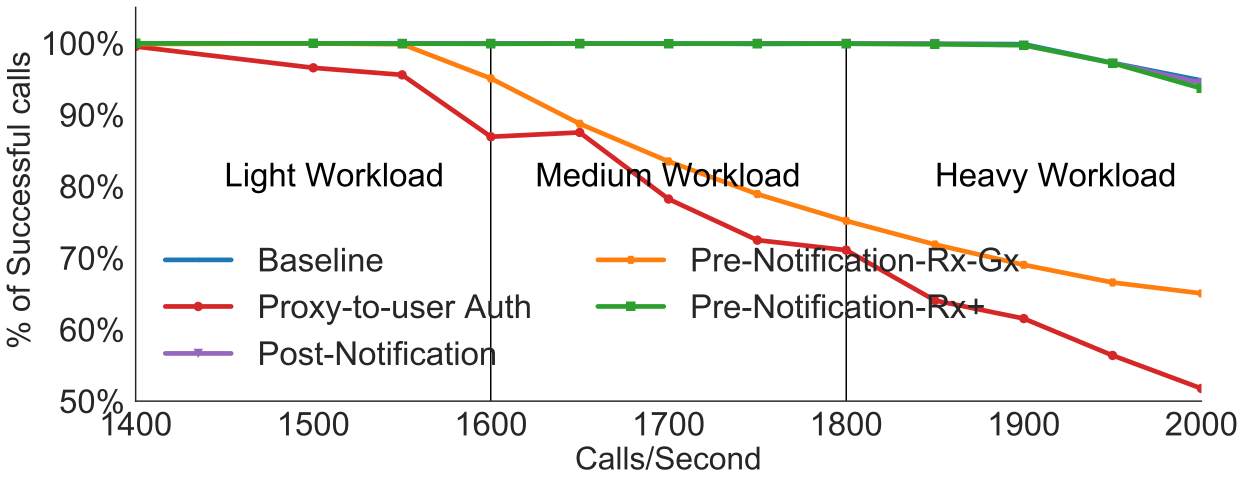


Figure 6.10. Percentage of successful calls with 0% spoofed calls with NFV deployment.

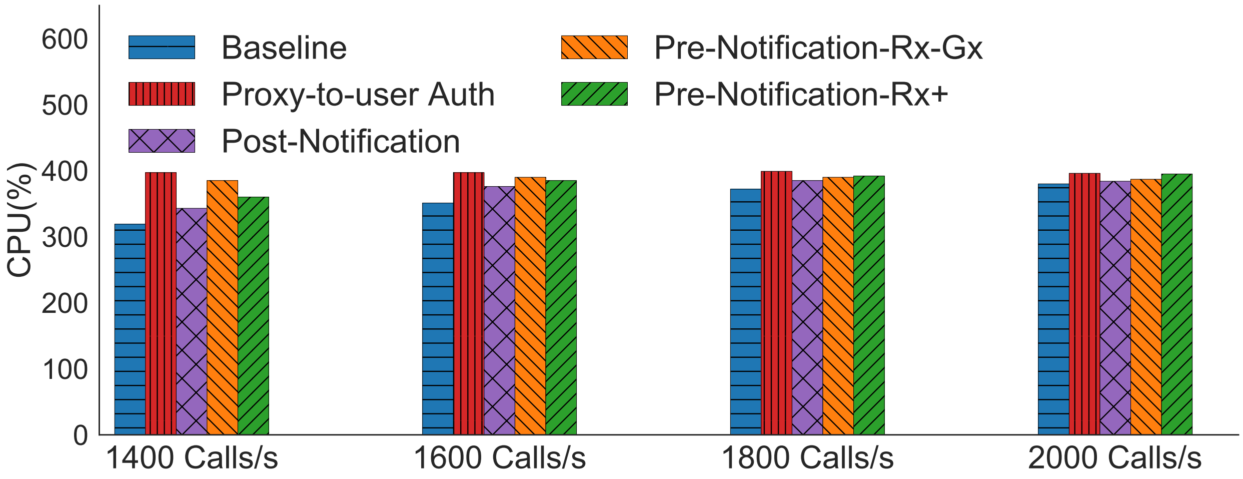


Figure 6.11. CPU utilization with 0% spoofed calls with NFV deployment.

also results in significant performance degradation at medium and high workloads. Since Pre-Notification-Rx-Gx relays the EPC identifiers via the PCRF, it exhibits higher CPU utilization than the other two variants due to the additional messages processed by the PCRF.

Figure 6.11 presents **total CPU utilization**. Proxy-to-user authentication uses the highest overall CPU even in the absence of the PGW SIP message interception overhead. Pre-Notification-Rx+ does not exhibit significantly higher CPU utilization or performance degradation than the baseline.

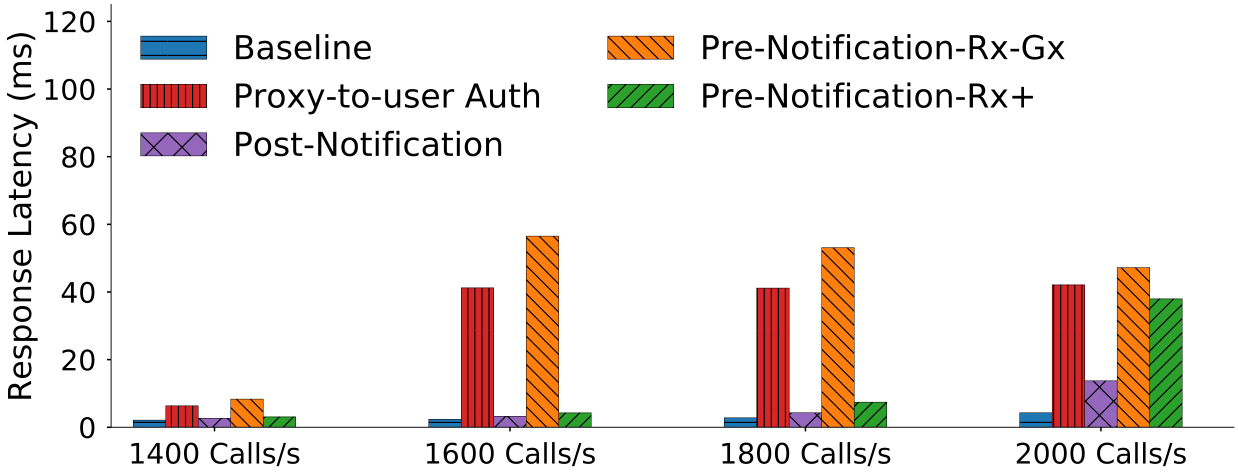


Figure 6.12. Average latency in VoLTE call setup with 0% spoofed calls with NFV deployment.

Figure 6.12 presents the **call setup latency** incurred in NFV deployment. The standard deviation among the values representing each of the 10 individual runs is below 9 ms (below 3 ms for medium and light workloads) in this case. Pre-Notification-Rx+ incurs only negligible latency compared to the baseline and Post-Notification at light and medium workloads.

Selective Validation

Selective validation at the IMS can be used in cases where the network is experiencing heavy workloads. For example, IMS servers can use historical data to determine which caller-IDs to be validated. Figure 6.13 shows the results of Pre-Notification-Rx+ when only a specific percentage of randomly selected calls are validated. As expected, the performance impact of caller-ID validation decreases as the percentage of calls that are validated decreases. When 10% of calls are validated, Pre-Notification-Rx+ has negligible overhead.

Tradeoffs among the Three Variants

The three Nascent variants offer service providers the flexibility to prioritize user experience, performance overhead, or deployment effort. Post-Notification has negligible overhead

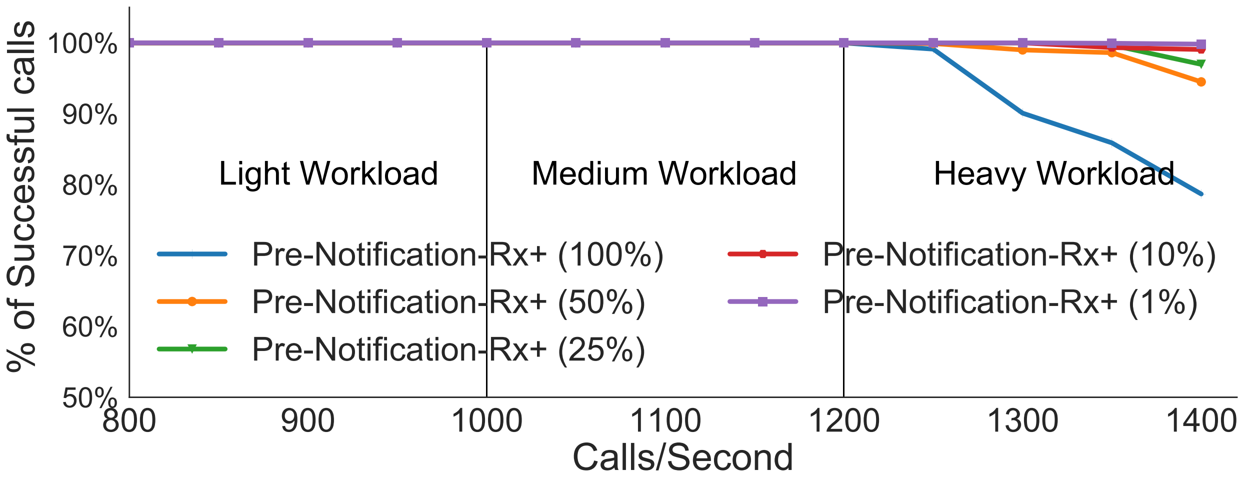


Figure 6.13. Percentage of successful calls with varying spoofing percentage for the Pre-Notification-Rx+ variant. The number within parentheses indicates the % of calls validated.

and requires no operational changes, but it may adversely impact user experience. Mobile subscribers may receive missed call notifications, and while this may be acceptable in the absence of a subscription-based (and possibly paid) caller-ID validation service, it is not ideal for end users.

Pre-Notification-Rx-Gx drops spoofed calls before the user is notified and does not require high deployment effort (no new interfaces are added), but has the highest performance overhead among the three variants. Therefore, it may not be acceptable for operators or deployments which often encounter high flash workloads.

Pre-Notification-Rx+ overcomes the shortcomings of the other two variants at the cost of higher deployment and operational effort. This variant requires standardization of a new REST based interface and requires IMS servers to directly contact the PGW serving a user. The list of PGWs currently deployed (and serving a user) can be easily configured for traditional deployments, but this is more difficult for NFV deployments where PGW and CSCFs are dynamically instantiated to meet workload requirements.

6.6 Discussion

Microservice-based design: Extracting and storing subscriber identifiers can be implemented as a microservice module which can be independently deployed. Such a design has the following benefits: (a) It substantially reduces processing and storage requirements at the PGW, and (b) It allows the CSCF to directly retrieve the caller-ID from the microservice, thereby eliminating the need for PGW configuration (IP address and port) at the IMS. This design also benefits NFV-based deployments where multiple PGW instances are dynamically instantiated to handle incoming workload.

Legitimate use of caller-ID spoofing and service extensions: Caller-ID spoofing can be used in legitimate cases such as privacy protection or when a user has multiple subscriber identifiers, e.g., preferring to show a 1-800 number [98]. NASCENT may flag these legitimate cases as caller-ID spoofing. We leave freedom to the carriers to determine what action to take once caller-ID spoofing is detected.

For instance, only spoofed calls from subscribers who use multiple or private caller-IDs, or subscribe to a legitimate spoofing service, can be allowed through. Blocking caller-ID spoofing can also be an add-on service. In NASCENT, caller-ID validation is performed at the IMS and therefore its design can be easily extended to support additional functionality. Unlike the PGW, IMS servers have access to network databases (such as HSS), which store IMS subscription information and can be used to allow legitimate caller-ID spoofing. NASCENT's mapping tables can be exposed to more services, such as SMS, to enable them to validate users.

Effective and gradual deployment: NASCENT is effective when it is deployed in the caller's network, and does not need universal deployment. NASCENT may not be helpful if only deployed in the callee's network when the forwarded ID has been spoofed. In this case, other solutions may be necessary, such as endpoint-only caller-ID spoofing detection or additional infrastructure for end-to-end authentication (e.g., via PKI or global certification infrastructure). These solutions are orthogonal and can be simultaneously used.

Extension to non-VoLTE calling: While our work focuses on VoLTE, it is conceptually applicable to other voice services such as circuit-switched calls, WiFi calling, and Internet telephony. The key idea is to enforce cross validation between the caller-ID used in the call setup and the one authenticated by the carrier networks.

Applicability to 5G: NASCENT can be naturally extended to 5G, which still uses a VoLTE-like technique to support VoIP. The use of NFV in 5G makes it even easier to detect caller-ID spoofing, as long as the proposed changes are integrated into the VNFs at the IMS and PGW. During early stages of 5G deployment, it is easier to develop built-in defense against caller-ID spoofing than to patch 4G.

6.7 Chapter Summary

In this chapter, we have proposed an effective, efficient, and easy-to-deploy solution, NASCENT, for detecting caller-ID spoofing. NASCENT performs the main cross validation operations at the IMS, hence reducing the load on the EPC data-plane gateways, but leverages authentic identifier information supplied by the EPC network. We have implemented and experimented with three variants of NASCENT, and compared them to proxy-to-user authentication. We find that NASCENT achieves its goals of effectiveness and efficiency, and the three variants offer service providers flexibility to prioritize user experience, performance overhead, or deployment effort. Further, our evaluation of NASCENT on NFV (container-driven) deployment and traditional (bare metal) deployment, shows that container-driven deployments that allow IMS and EPC NFs to share resources can outperform traditional deployments.

7. CONCLUSIONS AND FUTURE WORK

In this dissertation, we analyzed the case of cloud-native (microservice container packaged) Virtual Network Functions (VNFs) in supporting latency-sensitive services in the cellular core. We proposed that by leveraging knowledge-driven, traffic-aware, orchestration frameworks, network providers can meet stringent Service Level Agreements (SLAs) of future network services while simultaneously incurring lower deployment costs due to elastic on-demand resource allocation on cloud infrastructure.

In Chapter 3, we empirically demonstrated that container-driven instantiation of control and data place VNFs incur significantly lower performance overheads compared to VM-driven instantiations. We showed that low performance overhead of containerized VNFs coupled with lower instantiation time and smaller footprints, make containers the logical choice for cloud-native cellular core.

In Chapter 4, we showed that functionality-based decomposition of monolithic cellular control plane functions can yield stateless microservice components which can be aggregated using domain knowledge and VNF affinity to create microservice aggregates. These microservice aggregates can then be instantiated in close proximity by a traffic aware orchestrator allowing network providers to bound end-to-end Service Function Chain (SFC) latency and significantly reduce SLA violations.

In Chapter 5, we developed *Invenio*, which automates identification of microservice aggregates described in Contain-ed (Chapter 4) and allows service providers to make VNF placement decisions without the time-consuming and error-prone manual analysis currently used. *Invenio* automatically computes the correct transactions and affinity between VNFs and makes automated affinity-driven placement decisions, enabling service providers to better manage the ever increasing complexity of microservice-based VNFs.

Finally, in Chapter 6, we evaluated the efficacy of container-driven deployments in supporting real-world applications with long SFCs by developing a novel caller-ID validation solution for VoLTE and comparing its performance on traditional and container-driven deployment. Our evaluation of caller-ID validation solution in an container-driven VoLTE system showed that virtualized deployments which allow SFC components to share resources can support new services without significantly degrading existing system performance.

We now discuss how lessons learned from this dissertation can be applied to the design of emerging 5G networks, and some future directions that can hasten the adoption of cloud-native designs in cellular service delivery.

7.1 Provisioning in the 5G Core

5G is the next generation of cellular networking technologies that promises ultra-fast, ubiquitous connectivity for billions of wireless devices to enable a new wave of services. Unlike previous generations of cellular networks (3G/4G) which were designed to support voice and basic data connectivity services, the 5G core (5GC) network is expected to support a wide variety of services with disparate bandwidth and latency requirements. Services in 5GC are therefore delivered using logically independent virtual networks called network slices.

The design of the new 5GC marks a significant departure from the well-defined, stateful, monolithic network functions used by earlier generations of cellular networks. 5GC adopts *stateless* network functions, deployed as a collection of loosely-coupled *microservice* components (Chapter 4) which are typically deployed on containers (Chapter 3). Consequently, microservice-based architectures facilitate independent scaling and update of individual microservices, avoiding long development cycles, increasing elasticity, and reducing operational costs (Chapter 6). This decomposition of NFs into multiple smaller microservice components, or VNF Components (VNFCs), however, leads to longer, more complex SFCs.

When NFs are decomposed into smaller VNFCs, latency budgets available to individual VNFCs shrink. Compared to monolithic implementations, microservice-based VNFCs exchange more messages to process a user request [71]. Consequently, even small communication delays between VNFCs can lead to significant performance degradation. Affinity driven solutions such as *Invenio* (Chapter 4) can automatically derive the transactional affinity values between VNFCs from the messages exchanged between VNFCs for each traffic type and optimize processing of pre-determined (typically, latency-sensitive) workload types.

Orchestrators monitor Key Performance Indicators (KPIs) of each VNF/VNFC instance, and dynamically make placement and resource allocation decisions. However, owing to stringent SLAs and dynamic instantiation of VNFs in 5GC, the performance of a 5G slice also depends on a) Scaling strategy, which determines if additional resources should be added via scale-up (add resources to existing VNF instances) or scale-out (create new instances of VNFs), and b) Scaling sequence, which determines the order in which VNFs in an SFC should instantiated and if multiple VNFs must be scaled-out/scaled-up at the same time. Therefore, orchestrators should understand not only the incoming workload type and affinity between VNFCs for each type, but also the communication patterns of multiple instances of VNFCs with multiple instances of state stores. Future orchestrators will therefore require solutions which make network placement decisions (cloud/edge) in tandem with resource allocation decisions to jointly optimize service performance [114]. One potential avenue for orchestrators, therefore, is to use Machine Learning (ML) algorithms to make joint placement decisions.

7.2 ML-based Provisioning of Cellular Networks

Recent research has explored the benefits of ML-driven resource allocation [115]–[118]. However, resource utilization patterns of traditional monolithic VNFs exhibit high diversity. Therefore, training ML models for monolithic VNFs requires extensive failure data which is often unavailable due to over-provisioned cellular networks [116]. While the need for training data remains a challenge, the simpler processing logic of microservice-based cloud-native VNFs (VNFCs) makes individual VNFs within the cellular core more amenable to ML-driven on-demand scaling. Future orchestrators can therefore employ ML-based solution to instantiate and place VNFs.

Another challenge in supporting on-demand scaling of VNFs in cellular core is *proactive scaling*. Provisioning latency-sensitive services requires anticipating overload and failures, in order to proactively allocate and initialize additional resources before customer experience degrades. One solution to proactive scaling is to use ML algorithms to predict future workloads [118]. Another possible solution is using low-cost cloud offerings for transparent scaling. For example, serverless functions can be used alongside traditional VM or container instances to handle excess workload [119] and mitigate SLA violations while using predictive scaling. We believe that the 5CG orchestrators will therefore use a combination of ML-based workload prediction coupled with serverless functions to mitigate the impact of microbursts in incoming workload (flash workloads) on user experience.

However, a key reason why current orchestrators fall short in preventing SLA violations for latency-sensitive services is the lack of end-to-end scaling strategies. Longer SFCs composed of VNFCs imply that only a fraction of the latency budget available to a monolithic application is available to each microservice. To address the challenges posed by latency-sensitive microservice-based applications, orchestrators must adopt end-to-end resource allocation plans that make batch scaling decisions for an entire SFC, alleviating the cascading QoS deterioration caused by reactive scaling of individual VNFs. An end-to-end cellular network orchestrator should consider multiple factors in making scaling and placement decisions, such as incident workload, workload type, position of a VNF in the SFC, and scaling strategy. ML-based algorithms can play an important role in developing end-to-end solutions for future cloud-native cellular cores.

REFERENCES

[1] *3GPP TS 23.401: General Packet Radio Service (GPRS) Enhancements for Evolved Universal Terrestrial Radio Access Network (E-UTRAN) Access*, http://www.3gpp.org/ftp/Specs/html-info/23401.htm, 3GPP.

[2] V. Fajardo, J. Arkko, J. Loughney, and G. Zorn, "Diameter Base Protocol," RFC Editor, RFC 6733, Oct. 2012, http://www.rfc-editor.org/rfc/rfc6733.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc6733.txt.

[3] *3GPP TS 23.228: IP Multimedia Subsystem (IMS)*, http://www.3gpp.org/DynaReport/23228.htm, 3GPP.

[4] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: Session initiation protocol," RFC Editor, RFC 3261, Jun. 2002, http://www.rfc-editor.org/rfc/rfc3261.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3261.txt.

[5] *Amazon EC2*, http://aws.amazon.com/ec2.

[6] Google Cloud, *Autoscaling Groups of Instances*, https://cloud.google.com/compute/docs/autoscaler/, 2019.

[7] *ETSI Network Functions Virtualisation (NFV) Architectural Framework*, http://www.etsi.org/deliver/etsi_gs/NFV/001_099/002/01.02.01_60/gs_NFV002v010201p.pdf.

[8] L. Cao, P. Sharma, S. Fahmy, and V. Saxena, "NFV-VITAL: A framework for characterizing the performance of virtual network functions," in *Proceedings of IEEE NFV-SDN*, 2015, p. 7.

[9] A. Gember, R. Grandl, A. Anand, T. Benson, and A. Akella, "Stratos: Virtual middleboxes as first-class entities," University of Wisconsin-Madison, Tech. Rep., 2012, Technical report TR1771.

[10] A. Banerjee, R. Mahindra, K. Sundaresan, S. Kasera, J. Van der Merwe, and S. Rangarajan, "Scaling the LTE control-plane for future mobile access," in *Proceedings of CoNEXT*, Dec. 2015.

[11] P. K. Shanmugam, N. D. Subramanyam, J. Breen, C. Roach, and J. V. der Merwe, "DEIDtect: Towards distributed elastic intrusion detection," in *Proceedings of DCC*, 2014.

[12] V. Heorhiadi, M. K. Reiter, and V. Sekar, “New opportunities for load balancing in network-wide intrusion detection systems,” in *Proceedings of CoNEXT*, 2012.

[13] J. Anderson, H. Hu, U. Agarwal, C. Lowery, H. Li, and A. Apon, “Performance considerations of network functions virtualization using containers,” in *Proc. of ICNC*, 2016.

[14] T. Kamarainen, Y. Shan, M. Siekkinen, and A. Yla-Jaaski, “Virtual machines vs. containers in cloud gaming systems,” in *Proc. of NetGames*, Dec. 2015, pp. 1–6. DOI: 10.1109/NetGames.2015.7382987.

[15] M. Kablan, B. Caldwell, R. Han, H. Jamjoom, and E. Keller, “Stateless network functions,” in *Proceedings of the 2015 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization*, 2015, pp. 49–54, ISBN: 978-1-4503-3540-9. DOI: 10.1145/2785989.2785993. [Online]. Available: http://doi.acm.org/10.1145/2785989.2785993.

[16] *3GPP TR 23.714: Study on control and user plane separation of EPC nodes*, http://www.3gpp.org/DynaReport/23714.htm, 3GPP.

[17] A. Basta, W. Kellerer, M. Hoffmann, H. J. Morper, and K. Hoffmann, “Applying NFV and SDN to LTE mobile core gateways, the functions placement problem,” in *Proceedings of the 4th Workshop on All Things Cellular: Operations, Applications, & Challenges*, 2014, pp. 33–38. DOI: 10.1145/2627585.2627592. [Online]. Available: http://doi.acm.org/10.1145/2627585.2627592.

[18] H. Hawilo, A. Shami, M. Mirahmadi, and R. Asal, “NFV: state of the art, challenges and implementation in next generation mobile networks (vepc),” *CoRR*, vol. abs/1409.4149, 2014. [Online]. Available: http://arxiv.org/abs/1409.4149.

[19] K. Katsalis, N. Nikaein, E. Schiller, R. Favraud, and T. I. Braun, “5G architectural design patterns,” in *2016 IEEE International Conference on Communications Workshops (ICC)*, May 2016, pp. 32–37. DOI: 10.1109/ICCW.2016.7503760.

[20] Z. A. Qazi, M. Walls, A. Panda, V. Sekar, S. Ratnasamy, and S. Shenker, “A high performance packet core for next generation cellular networks,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM ’17, Los Angeles, CA, USA: ACM, 2017, pp. 348–361, ISBN: 978-1-4503-4653-5. DOI: 10.1145/3098822.3098848. [Online]. Available: http://doi.acm.org/10.1145/3098822.3098848.

[21] A. Sheoran, P. Sharma, S. Fahmy, and V. Saxena, "Contain-ed: An nfv micro-service system for containing e2e latency," in *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems*, ser. HotConNet '17, Los Angeles, CA, USA: ACM, 2017, pp. 12–17, ISBN: 978-1-4503-5058-7. DOI: 10.1145/3094405.3094408. [Online]. Available: http://doi.acm.org/10.1145/3094405.3094408.

[22] M. T. Raza, D. Kim, K. H. Kim, S. Lu, and M. Gerla, "Rethinking LTE network functions virtualization," in *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, Oct. 2017, pp. 1–10. DOI: 10.1109/ICNP.2017.8117554.

[23] A. Gember, R. Grandl, A. Anand, T. Benson, and A. Akella, "Stratos: Virtual middleboxes as first-class entities," University of Wisconsin-Madison, Tech. Rep., 2012, Technical report TR1771.

[24] M. Selimi, L. Cerdà-Alabern, M. Sánchez-Artigas, F. Freitag, and L. Veiga, "Practical service placement approach for microservices architecture," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2017, pp. 401–410. DOI: 10.1109/CCGRID.2017.28.

[25] Y. Hu, C. de Laat, and Z. Zhao, "Optimizing service placement for microservice architecture in clouds," *Applied Sciences*, vol. 9, no. 21, 2019, ISSN: 2076-3417. DOI: 10.3390/app9214663. [Online]. Available: https://www.mdpi.com/2076-3417/9/21/4663.

[26] Y. Yu, J. Yang, C. Guo, H. Zheng, and J. He, "Joint optimization of service request routing and instance placement in the microservice system," *Journal of Network and Computer Applications*, vol. 147, p. 102 441, 2019, ISSN: 1084-8045. DOI: https://doi.org/10.1016/j.jnca.2019.102441. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804519303017.

[27] A. R. Sampaio, J. Rubin, I. Beschastnikh, and N. S. Rosa, "Improving microservice-based applications with runtime placement adaptation," *Journal of Internet Services and Applications*, vol. 10, 2019.

[28] A. Panda, M. Sagiv, and S. Shenker, "Verification in the age of microservices," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, ser. HotOS '17, Whistler, BC, Canada: ACM, 2017, pp. 30–36, ISBN: 978-1-4503-5068-6. DOI: 10.1145/3102980.3102986. [Online]. Available: http://doi.acm.org/10.1145/3102980.3102986.

[29] J. Nam, J. Seo, and S. Shin, "Probius: Automated approach for vnf and service chain analysis in software-defined nfv," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18, Los Angeles, CA, USA: ACM, 2018, 14:1–14:13, ISBN: 978-1-4503-5664-0. DOI: 10.1145/3185467.3185495. [Online]. Available: http://doi.acm.org/10.1145/3185467.3185495.

[30] P. Naik, D. K. Shaw, and M. Vutukuru, "Nfvperf: Online performance monitoring and bottleneck detection for nfv," in *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Nov. 2016, pp. 154–160. DOI: 10.1109/NFV-SDN.2016.7919491.

[31] T. Cerny, M. J. Donahoo, and M. Trnka, "Contextual understanding of microservice architecture: Current and future directions," *SIGAPP Appl. Comput. Rev.*, vol. 17, no. 4, pp. 29–45, Jan. 2018, ISSN: 1559-6915. DOI: 10.1145/3183628.3183631. [Online]. Available: http://doi.acm.org.ezproxy.lib.purdue.edu/10.1145/3183628.3183631.

[32] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, "Sinan: ML-based and QoS-aware resource management for cloud microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021, Virtual, USA: Association for Computing Machinery, 2021, pp. 167–181, ISBN: 9781450383172. DOI: 10.1145/3445814.3446693. [Online]. Available: https://doi.org/10.1145/3445814.3446693.

[33] T. Krueger, N. Krämer, and K. Rieck, "ASAP: Automatic semantics-aware analysis of network payloads," in *Proceedings of the International ECML/PKDD Conference on Privacy and Security Issues in Data Mining and Machine Learning*, ser. PSDML'10, Barcelona, Spain: Springer-Verlag, 2011, pp. 50–63, ISBN: 978-3-642-19895-3. [Online]. Available: http://dl.acm.org/citation.cfm?id=1997358.1997363.

[34] Y. Wang, Z. Zhang, D. D. Yao, B. Qu, and L. Guo, "Inferring protocol state machine from network traces: A probabilistic approach," in *Proceedings of the 9th International Conference on Applied Cryptography and Network Security*, ser. ACNS'11, Nerja, Spain: Springer-Verlag, 2011, pp. 1–18, ISBN: 978-3-642-21553-7. [Online]. Available: http://dl.acm.org/citation.cfm?id=2025968.2025970.

[35] G. Bossert, F. Guihéry, and G. Hiet, "Towards automated protocol reverse engineering using semantic information," in *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '14, Kyoto, Japan: ACM, 2014, pp. 51–62, ISBN: 978-1-4503-2800-5. DOI: 10.1145/2590296.2590346. [Online]. Available: http://doi.acm.org/10.1145/2590296.2590346.

[36] W. Cui, J. Kannan, and H. J. Wang, “Discoverer: Automatic protocol reverse engineering from network traces,” in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, ser. SS’07, Boston, MA: USENIX Association, 2007, 14:1–14:14, ISBN: 111-333-5555-77-9. [Online]. Available: http://dl.acm.org/citation.cfm?id=1362903.1362917.

[37] J. Antunes, N. Neves, and P. Verissimo, “Reverse engineering of protocols from network traces,” in *2011 18th Working Conference on Reverse Engineering*, Oct. 2011, pp. 169–178. DOI: 10.1109/WCRE.2011.28.

[38] C. Leita, K. Mermoud, and M. Dacier, “Scriptgen: An automated script generation tool for honeyd,” in *21st Annual Computer Security Applications Conference (ACSAC’05)*, Dec. 2005, 12 pp.–214. DOI: 10.1109/CSAC.2005.49.

[39] N. Griffeth, Y. Cantor, and C. Djouvas, “Testing a network by inferring representative state machines from network traces,” in *Software Engineering Advances, International Conference on*, Oct. 2006, pp. 31–31. DOI: 10.1109/ICSEA.2006.261287.

[40] M. Z. Rafique, J. Caballero, C. Huygens, and W. Joosen, “Network dialog minimization and network dialog diffing: Two novel primitives for network security applications,” in *Proceedings of the 30th Annual Computer Security Applications Conference*, ser. ACSAC ’14, New Orleans, Louisiana, USA: ACM, 2014, pp. 166–175, ISBN: 978-1-4503-3005-3. DOI: 10.1145/2664243.2664261. [Online]. Available: http://doi.acm.org/10.1145/2664243.2664261.

[41] P. Quinn, U. Elzur, and C. Pignataro, “Network Service Header (NSH),” RFC Editor, RFC 8300, Jan. 2018.

[42] J. Halpern and C. Pignataro, “Service function chaining (sfc) architecture,” RFC Editor, RFC 7665, Oct. 2015.

[43] H. Mustafa, W. Xu, A. R. Sadeghi, and S. Schulz, “You can call but you can’t hide: Detecting caller ID spoofing attacks,” in *Proc. of DSN*, 2014, pp. 168–179.

[44] H. Mustafa, W. Xu, A.-R. Sadeghi, and S. Schulz, “End-to-end detection of caller ID spoofing attacks,” *IEEE Transactions on Dependable and Secure Computing*, 2016.

[45] J. Li, F. Faria, J. Chen, and D. Liang, “A mechanism to authenticate caller ID,” in *World Conference on Information Systems and Technologies*, 2017, pp. 745–753.

[46] S. T. Chow, V. Choyi, and D. Vinokurov, *Caller name authentication to prevent caller identity spoofing*, US Patent 9,241,013, Jan. 2016.

[47] Y. Cai, *Validating caller id information to protect against caller id spoofing*, US Patent 8,254,541, Aug. 2012.

[48] S. A. Danis, *Systems and methods for caller id authentication, spoof detection and list based call handling*, US Patent 9,060,057, Jun. 2015.

[49] H. Tu, A. Doupé, Z. Zhao, and G.-J. Ahn, "Toward standardization of authenticated caller ID transmission," *IEEE Communications Standards Magazine*, vol. 1, no. 3, pp. 30–36, 2017.

[50] J. Song, H. Kim, and A. Gkelias, "Ivisher: Real-time detection of Caller ID spoofing," *ETRI Journal*, vol. 5, no. 5, Aug. 2014.

[51] S. Kim, B. Koo, and H. Kim, "Abnormal VoLTE call setup between UEs," in *Proc. of International Conference on Security and Management, SAM*, 2015.

[52] *Linux containers: Lxc*, https://linuxcontainers.org/.

[53] *Docker: An open platform for distributed applications for developers and sysadmins*, https://www.docker.com/.

[54] *Kernel Virtual Machine*, http://www.linux-kvm.org/page/Main_Page/.

[55] *Openair-cn: An implementation of the evolved packet core network*, https://gitlab.eurecom.fr/oai/openair-cn/.

[56] *Openairinterface*, https://gitlab.eurecom.fr/oai/openairinterface5g/.

[57] *Sample captures - tcpreplay*, http://tcpreplay.appneta.com/wiki/captures.html.

[58] *Pcap files from the the information security talent search (ists)*, http://www.netresec.com/?page=ISTS.

[59] *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2*, 2011.

[60] *Clearwater*, http://www.projectclearwater.org/.

[61] K. Pentikousis, Y. Wang, and W. Hu, "Mobileflow: Toward software-defined mobile networks," *IEEE Communications Magazine*, vol. 51, no. 7, pp. 44–53, Jul. 2013, ISSN: 0163-6804. DOI: 10.1109/MCOM.2013.6553677.

[62] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, “Network function virtualization: Challenges and opportunities for innovations,” *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, Feb. 2015, ISSN: 0163-6804. DOI: 10.1109/MCOM.2015.7045396.

[63] A. S. Rajan, S. Gobriel, C. Maciocco, K. B. Ramia, S. Kapury, A. Singhy, J. Ermanz, V. Gopalakrishnanz, and R. Janaz, “Understanding the bottlenecks in virtualizing cellular core network functions,” in *The 21st IEEE International Workshop on Local and Metropolitan Area Networks*, Apr. 2015, pp. 1–6. DOI: 10.1109/LANMAN.2015.7114735.

[64] *3GPP TS 23.335: User Data Convergence (UDC)*, http://www.3gpp.org/DynaReport/23335.htm, 3GPP.

[65] *3GPP TS 23.218: IP multimedia (IM) session handling*, http://www.3gpp.org/DynaReport/23218.htm, 3GPP.

[66] *3GPP TS 29.274, Evolved General Packet Radio Service (GPRS) Tunnelling Protocol for Control plane (GTPv2-C)*, http://www.3gpp.org/DynaReport/29274.htm, 3GPP.

[67] *Opensips*, http://www.opensips.org/.

[68] *SIPp*, http://sipp.sourceforge.net/.

[69] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, “Network function virtualization: Challenges and opportunities for innovations,” *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, 2015. DOI: 10.1109/MCOM.2015.7045396.

[70] S. Sharma, R. Miller, and A. Francini, “A cloud-native approach to 5G network slicing,” *IEEE Communications Magazine*, vol. 55, no. 8, pp. 120–127, 2017. DOI: 10.1109/MCOM.2017.1600942.

[71] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, Y. He, and C. Delimitrou, “An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems,” in *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Providence, RI, Apr. 2019.

[72] M. T. Raza and S. Lu, “Enabling low latency and high reliability for IMS-NFV,” in *2017 13th International Conference on Network and Service Management (CNSM)*, Nov. 2017, pp. 1–9. DOI: 10.23919/CNSM.2017.8256015.

[73] *OpenStack*, http://www.openstack.org/.

[74] Kubernetes, *Kubernetes*, https://kubernetes.io/.

[75] *3GPP TS 23.501: System Architecture for the 5G System*, http://www.3gpp.org/ftp/Specs/html-info/23501.htm, 3GPP.

[76] A. Sheoran, S. Fahmy, M. Osinski, C. Peng, B. Ribeiro, and J. Wang, "Experience: Towards automated customer issue resolution in cellular networks," in *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '20, London, United Kingdom: Association for Computing Machinery, 2020, ISBN: 9781450370851. DOI: 10.1145/3372224.3419203. [Online]. Available: https://doi.org/10.1145/3372224.3419203.

[77] V. S. Abhayawardhana and R. Babbage, "A traffic model for the IP multimedia subsystem (IMS)," in *2007 IEEE 65th Vehicular Technology Conference - VTC2007-Spring*, 2007, pp. 783–787. DOI: 10.1109/VETECS.2007.171.

[78] Dialogic, *IMS and VoLTE 3GPP Interfaces*, https://edit.dialogic.com/edu/interfaces.

[79] *3GPP TS 29.229: Cx and Dx interfaces based on the Diameter protocol*, http://www.3gpp.org/DynaReport/29229.htm, 3GPP.

[80] *3GPP TS 29.212: Policy and charging control over Gx reference point*, http://www.3gpp.org/DynaReport/29212.htm, 3GPP.

[81] *Designing microservices: Logging and monitoring*, https://docs.microsoft.com/en-us/azure/architecture/microservices/logging-monitoring, Microsoft.

[82] *ONAP - Open Networking Automation Platform*, https://wiki.onap.org/pages/viewpage.action?pageId=5734499, ONAP.

[83] *OPNFV - Open Platform for NFV*, https://www.opnfv.org/, OPNFV.

[84] R. Hao, D. Lee, R. K. Sinha, and N. Griffeth, "Integrated system interoperability testing with applications to voip," *IEEE/ACM Transactions on Networking (TON)*, vol. 12, no. 5, pp. 823–836, 2004.

[85] *Wireshark*, https://www.wireshark.org.

[86] T. Berners-Lee, R. T. Fielding, and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax," RFC Editor, STD 66, Jan. 2005, http://www.rfc-editor.org/rfc/rfc3986.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc3986.txt.

[87] *Kamailio SIP Server*, https://www.kamailio.org/w/.

[88] J. Vink, *KORE.io - An easy to use web platform for C*, https://kore.io/.

[89] *freeDiameter: Diameter Open Implementation*, http://www.freediameter.net/trac/.

[90] *3GPP TS 29.214: Policy and charging control over Rx reference point)*, http://www.3gpp.org/DynaReport/29214.htm, 3GPP.

[91] M. Handley, V. Jacobson, and C. Perkins, "Sdp: Session description protocol," RFC Editor, RFC 4566, Jul. 2006, http://www.rfc-editor.org/rfc/rfc4566.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc4566.txt.

[92] *Docker Swarm*, https://www.docker.com/products/docker-swarm.

[93] *Docker Compose*, https://docs.docker.com/compose/.

[94] *Openair-cn: Evolved Core Network Implementation of OpenAirInterface*, https://gitlab.eurecom.fr/oai/openair-cn.

[95] *OpenEPC - Evolved Packet Core (vEPC)*, https://www.openepc.com/.

[96] A. Jain, S. K. Lohani, and M. Vutukuru, "A comparison of SDN and NFV for redesigning the LTE Packet Core," in *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2016, pp. 74–80.

[97] C.-Y. Li, G.-H. Tu, C. Peng, Z. Yuan, Y. Li, S. Lu, and X. Wang, "Insecurity of voice solution VoLTE in LTE mobile networks," in *Proc. of CCS*, 2015.

[98] FCC, *Spoofing and caller ID*, https://www.fcc.gov/consumers/guides/spoofing-and-caller-id.

[99] *IRS urges public to stay alert for scam phone calls*, https://www.irs.gov/newsroom/irs-urges-public-to-stay-alert-for-scam-phone-calls.

[100] TrueCaller, https://www.truecaller.com/.

[101] *RoboKillerApp*, https://www.robokiller.com/blog/local-call.

[102] SpoofCard, https://www.spoofcard.com/.

[103] SpoofTel, https://www.spooftel.com/.

[104] B. Reaves, L. Blue, and P. Traynor, "AuthLoop: End-to-end cryptographic authentication for telephony over voice channels," in *USENIX Security*, 2016, ISBN: 978-1-931971-32-4.

[105] B. Reaves, L. Blue, H. Abdullah, L. Vargas, P. Traynor, and T. Shrimpton, “AuthentiCall: Efficient identity and content authentication for phone calls,” in *USENIX Security*, 2017.

[106] H. Deng, W. Wang, and C. Peng, “CEIVE: Combating Caller ID Spoofing on 4G Mobile Phones Via Callee-Only Inference and Verification,” in *Proc. of ACM MOBICOM*, 2018.

[107] *Voice over LTE*, http://www.gsma.com/technicalprojects/volte.

[108] V. Fajardo, J. Arkko, J. Loughney, and G. Zorn, *Diameter base protocol*, RFC 6733, Oct. 2012.

[109] H. Kim, D. Kim, M. Kwon, H. Han, Y. Jang, D. Han, T. Kim, and Y. Kim, “Breaking and fixing volte: Exploiting hidden data channels and mis-implementations,” in *Proc. of CCS*, 2015.

[110] P. Ventuzelo, O. Le Moal, and T. Coudray, “Subscribers remote geolocation and tracking using 4g volte enabled android phone,” in *Symp. on Information and Communications Security (SSTIC)*, 2017.

[111] *Metaswitch: Evaluating VoLTE security*, https://www.metaswitch.com/the-switch/evaluating-volte-security.

[112] *3GPP TS 29.213: Policy and charging control signalling flows and quality of service (QoS) parameter mapping*, http://www.3gpp.org/DynaReport/29213.htm, 3GPP.

[113] C. Shen, E. Nahum, H. Schulzrinne, and C. P. Wright, “The impact of TLS on SIP server performance: Measurement and modeling,” *IEEE/ACM Trans. Netw.*, vol. 20, no. 4, pp. 1217–1230, Aug. 2012, ISSN: 1063-6692.

[114] A. Sheoran, S. Fahmy, L. Cao, and P. Sharma, “Ai-driven provisioning in the 5g core,” *IEEE Internet Computing*, vol. 25, no. 02, pp. 18–25, Mar. 2021, ISSN: 1941-0131. DOI: 10.1109/MIC.2021.3056230.

[115] L. Cao, P. Sharma, S. Fahmy, and V. Saxena, “ENVI: Elastic resource flexing for network function virtualization,” in *Proc. of HotCloud*, 2017, pp. 1–11.

[116] J. Gao, M. Zahran, A. Sheoran, S. Fahmy, and B. Ribeiro, “Infinity learning: Learning Markov chains from aggregate steady-state observations,” in *Proc. of AAAI*, Feb. 2020.

[117] Y. Xiao, Q. Zhang, F. Liu, J. Wang, M. Zhao, Z. Zhang, and J. Zhang, “NFVdeep: Adaptive online service function chain deployment with deep reinforcement learning,” in *Proc. of IWQoS*, New York, NY, USA, 2019, ISBN: 9781450367783. DOI: 10.1145/3326285.3329056. [Online]. Available: https://doi.org/10.1145/3326285.3329056.

[118] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. Delimitrou, “Sinan: Ml-based and qos-aware resource management for cloud microservices,” in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS 2021, Virtual, USA: Association for Computing Machinery, 2021, pp. 167–181, ISBN: 9781450383172. DOI: 10.1145/3445814.3446693. [Online]. Available: https://doi.org/10.1145/3445814.3446693.

[119] J. H. Novak, S. K. Kasera, and R. Stutsman, “Cloud functions for fast and robust resource auto-scaling,” in *Proc. of COMSNETS*, 2019, pp. 133–140.