Software-Driven and Virtualized Architectures for Scalable 5G Networks

by

Mehrdad Moradi

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2018

Doctoral Committee:

Professor Z. Morley Mao, Chair
Assistant Professor N M Mosharaf Kabir Chowdhury
Associate Professor Harsha Madhyastha
Assistant Professor Neda Masoud

Mehrdad Moradi

moradi@umich.edu

ORCID iD: 0000-0001-8158-2759

This dissertation is dedicated to my family

for their endless love, support, and encouragement.

ACKNOWLEDGEMENTS

I could not complete this dissertation without the encouragement and support of many great people. First, I would like to thank my research advisor, Professor Z. Morley Mao. Morley has taught me more than I could ever give her credit here. She helped me to improve my research, writing, presentation, and critical thinking skills. More importantly, she created a unique opportunity and environment for me to collaborate with six different network operators and vendors (AT&T, NEC, Ericsson, Nokia, Huawei, and China Mobile) and to solve some real problems in managing hyper-scale cellular and datacenter networks. Second, I want to greatly thank my lovely wife, Maryam, for being able to accompany me for six years during the Ph.D. program. Thank you for your constant support through the ups and downs. While it has been bumpy at times, your confidence in me has enhanced my ability to get through it all and succeed in the end.

I would also like to thank my dissertation committee, Professor Harsha Madhyastha, Professor Mosharaf Chowdhury and Professor Neda Masoud, for their time and effort to help improve and refine my work. I will never forget Neda's help in the last minutes before my thesis proposal. Moreover, I appreciate Dr. Shubho Sen and Dr. Oliver Spatscheck from AT&T Labs for their great technical discussions and guidance throughout the SoftBox work, where we designed a radical low-latency 5G architecture from the ground up. I truly enjoyed working with Shubho in the past two years. Words cannot express my gratitude enough when I think about his kindness and expertise. I also appreciate Dr. Karthik Sundaresan's and Dr. Eugene Chai's mentorship and support throughout my 8-month internship at NEC Labs in Princeton. The SkyCore project, where we built real UAV-based 5G cellular networks,

could not be a successful work without their continuous and valuable help and support. Besides being experts in wireless networking, Karthik and Eugene are fantastic people in other aspects of life.

At the early stage of my graduate studies, Dr. Li Erran Li was the first person who taught me how to think out-of-the-box by guiding me in the design and development of the SoftMoW project aimed at conceptualizing the first hierarchical and recursive software-defined architecture for globally controlling continent-wide cellular networks. I am also deeply thankful to Dr. Ying Zhang who offered me to do my first internship in the US at Ericsson research and truly supported me before and during the internship. Ying waited four months until Ericsson got necessary permissions from the US government to hire me. Finally, I thank Professor Mike Reiter and Professor Feng Qian who have been the most influential people in improving my academic writing skills. I always remember their great annotations all over my first paper draft in the Caesar project where I had to realize a complex memory-efficient white box switch and router design.

The six-year life in graduate school has become wonderful because of many colleagues and friends. I would like to thank all the students in our group: Yikai Lin, Ashkan Nikravesh, Shichang Shawn Xu, Sanae Rosen, Yihua Guo, Alfred Qi Chen, Jeremy Erickson, David Ke Hong, Yuru Shao, Xiao Zhu, and Yunhan Jia. I would also like to thank other friends in the department, particularly for valuable discussions and suggestions on my works: Dr. Hamed Yousefi, Amir Rahmati, Ofir Weisse, Morteza Sheikhsofla, and Javad Bagherzadeh.

Finally, but not least, my thanks go to my parents, Shahrbanoo Farahzadi and Esmaeil Moradi, who have been a true source of inspiration and love throughout my life. My parents always supported me to be an independent thinker and have confidence in my abilities to follow new things that inspire me. Finally, my most heartfelt thanks to my brothers, Dr. Mohammad Moradi and Dr. Alireza Moradi, and my sister Mozhgan. This dissertation is heartily dedicated to them.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

ABSTRACT

In this dissertation, we argue that it is essential to rearchitect 4G cellular core networks–sitting between the Internet and the radio access network–to meet the scalability, performance, and flexibility requirements of 5G networks. Today, there is a growing consensus among operators and research community that software-defined networking (SDN), network function virtualization (NFV), and mobile edge computing (MEC) paradigms will be the key ingredients of the next-generation cellular networks. Motivated by these trends, we design and optimize three core network architectures, SoftMoW, SoftBox, and SkyCore, for different network scales, objectives, and conditions. SoftMoW provides global control over nationwide core networks with the ultimate goal of enabling new routing and mobility optimizations. SoftBox attempts to enhance policy enforcement in statewide core networks to enable low-latency, signaling-efficient, and customized services for mobile devices. SkyCore is aimed at realizing a compact core network for citywide UAV-based radio networks that are going to serve first responders in the future. Network slicing techniques make it possible to deploy these solutions on the same infrastructure in parallel. To better support mobility and provide verifiable security, these architectures can use an addressing scheme that separates network locations and identities with self-certifying, flat and non-aggregatable address components. To benefit the proposed architectures, we designed a high-speed and memory-efficient router, called Caesar, for this type of addressing scheme.

CHAPTER I

Introduction

1.1 Background and Thesis Statement

Cellular networks have become an integral part of our society. We use them to make phone calls, check news, watch videos, and make transactions. They are distributed throughout large geographical areas (e.g., a country) and consist of tens of thousands of packet processing elements (e.g., gateways, base stations).

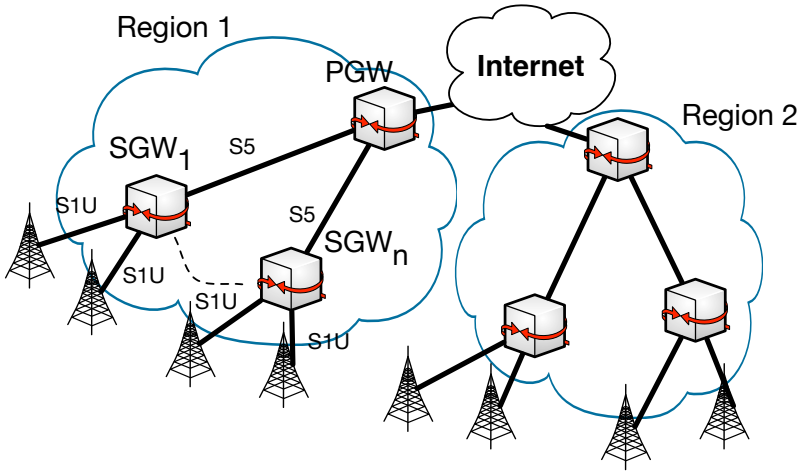


Figure 1.1: 4G cellular wide area network (WAN) with two regions

Today's 4G wide area networks (WANs) are organized into very large regions (Figure 1.1), each having an evolved packet core (EPC) network and a radio access network (RAN). Each EPC (Figure 1.2) contains an Internet edge comprised of packet data network gateways (PGWs) and a radio edge connecting to RAN. RAN consists of only base stations and provides LTE radio coverage for user equipments (UEs). EPC is responsible for processing

UEs' signaling and data traffic; it enforces network policies, provides always-on Internet connectivity, and offers seamless mobility support.

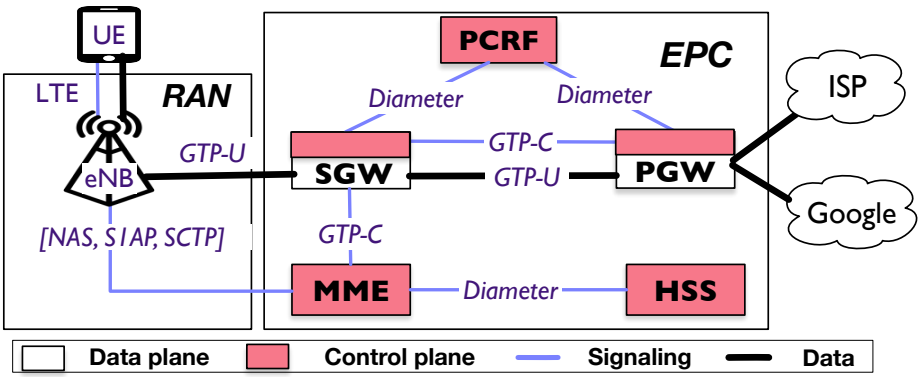


Figure 1.2: A cellular core region-RAN+EPC network

EPC has a hierarchical structure partitioning its functions among a group of dedicated nodes. At the Internet edge, the PGW connects the core to Internet/content providers and enforces most of the data plane policies (*e.g.,* NAT, DPI). At the RAN edge, enhanced node Bs (eNodeBs) are grouped into logical serving areas and connect to serving gateways (SGWs). Each SGW acts as a mobility anchor point for its eNodeBs. It also forwards each UE's data traffic between the eNodeB and PGW using a separate GTP-U (GPRS tunneling protocol) tunnel. To connect to the network, UEs must register with the mobility management entity (MME) through eNodeBs. MME continuously exchanges signaling traffic with UEs and eNodeBs to perform security and mobility functions (*e.g.,* authentication, handover). To handle these tasks, MME accesses home subscriber server (HSS) that is a centralized database containing UE-related information (*e.g.,* SIM card key). For connected UEs, policy and charging rule function (PCRF) authorizes the treatment that UEs' data flows receive by supplying QoS rules to PGW/SGW in real time.

Unfortunately, today's EPC networks suffer from an increasing pressure on their scalability, performance, and flexibility. Thus, operators are actively exploring different designs for 5G core networks to overcome the EPC challenges and meet emerging 5G use cases.

- **Scalability challenges.** First, the number of global LTE subscribers is around 1.2 billion with a peak daily addition of 2 million devices since 2016 [26]. On the one hand, this

trend and the continued exponential growth of mobile traffic put tremendous pressure on the EPC's data plane scalability. Soon, mobile traffic will represent around 20% of total IP traffic on the Internet. On the other hand, the fast growth of signaling traffic from mobile UEs poses a major challenge to scalability of EPC's control plane [27]. In response to each UE signaling messages, EPC generates a huge amount of internal control plane overheads or signaling storms [14, 27] due to its complex nodes and distributed protocols.

- **Flexibility challenges.** Second, diversity of devices supporting LTE is going beyond cell phones and is reaching to domestic robots, sensors, and cars. Unfortunately, the EPC networks control and data planes lack fine-grained customizability and programmability. EPC cannot easily realize diverse and customized 5G services for new use cases (e.g., public safety, tactile Internet, autonomous cars) [90].

- **Performance challenges.** Third, while many 5G use cases require ultra-low latency and gigabit bandwidth, recent studies show that mobile application performance is seriously degraded by EPC's inefficient policy enforcement and routing. EPC routes traffic of UEs destined to the Internet or other nearby UEs on long suboptimal paths between RAN and PGW [130]. Moreover, there is no control plane interaction between EPC instances located in different regions. Thus, UEs crossing region boundaries experience significant service disruption.

Thesis statement. *To address these challenges and realize emerging 5G use cases, my thesis statement is that cellular networks, particularly their core, must be rearchitected to provide scalability, flexibility, and performance as their first-order properties.*

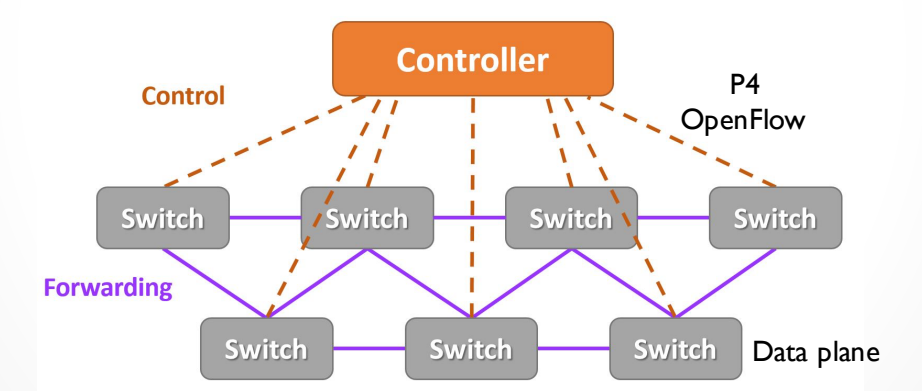


Figure 1.3: Software-defined networking (SDN)

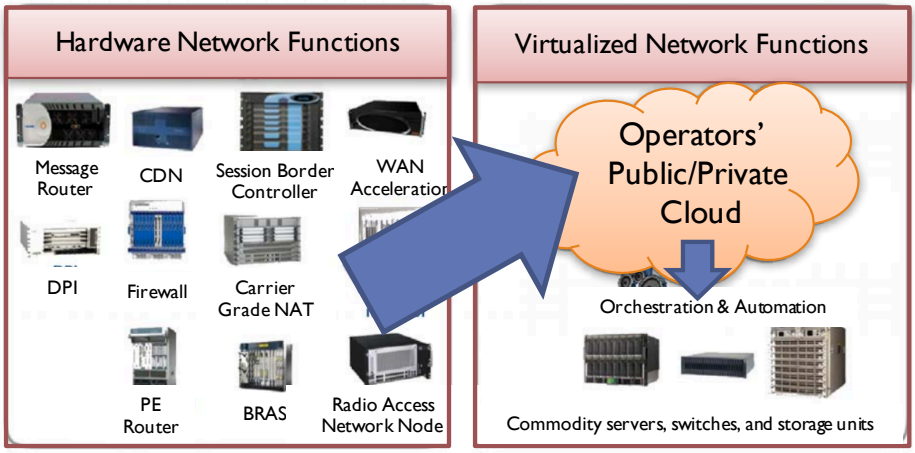


Figure 1.4: Network function virtualization (NFV)

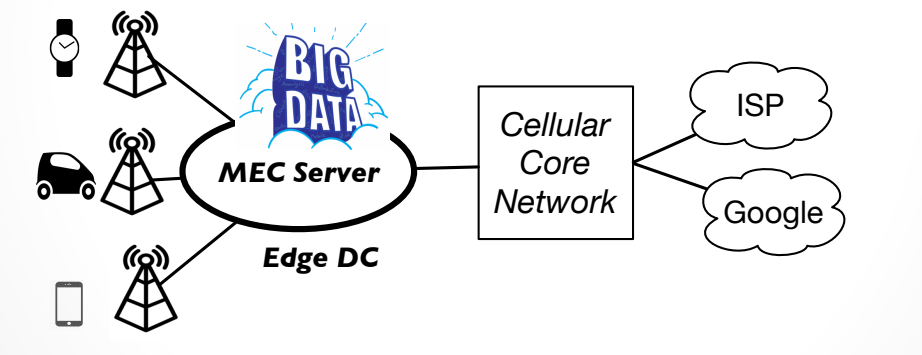


Figure 1.5: Mobile edge computing (MEC)

1.2 Overview and High-level Approach

Today, there is a growing consensus that software-defined networking (SDN), network function virtualization (NFV), and mobile edge computing (MEC) will be the dominating ingredients of 5G networks. SDN is a network design paradigm that advocates for programmability and automation in design, reconfiguration, and managing networks. A typical

software-defined network consists of a logically centralized controller and programmable switches (Figure 1.3). NFV is a network architecture concept arguing for migrating network functions (NFs) that traditionally run on complex dedicated hardware to commodity x86 servers (Figure 1.4). Finally, MEC (Figure 1.5) is a conceptual proposal that attempts to provide cloud computing capabilities close to radio edges of cellular networks to run low-latency applications (e.g., big data and machine learning). We combine the benefits of the SDN, NFV, and MEC paradigms in four different projects (as summarized in Figure 1.6) to support the thesis statement.

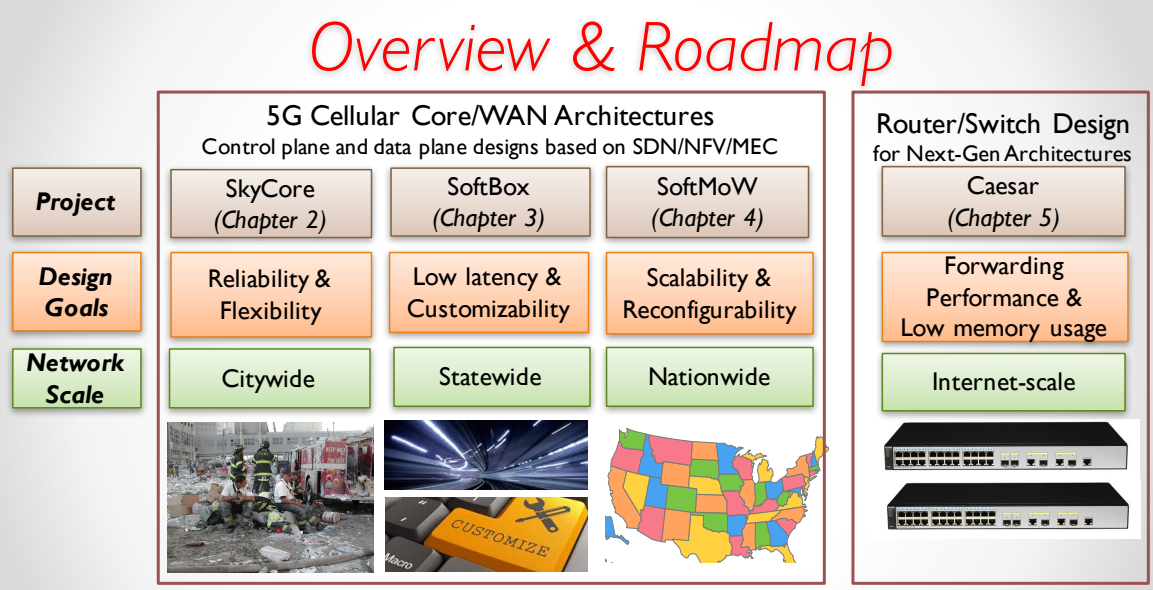


Figure 1.6: Summary of projects supporting this dissertation

In particular, we first design and optimize three 5G core network architectures for different network conditions, objectives and scales: (1) SkyCore that is an efficient core network solution for UAV-based LTE RANs that are going to provide on-demand LTE coverage for first responders and general public in citywide challenging environments, (2) SoftBox that is a low-latency, signaling-efficient, and customizable core network covering larger scale statewide LTE RANs, and (3) SoftMoW that is a scalable and dynamic cellular WAN architecture for managing nationwide LTE RANs. Finally, we present Caesar that is a high-speed and memory-efficient router architecture and can be deployed as a complementary solution in SkyCore, SoftBox, and SoftMoW or other future Internet architectures to improve mobility

support and verifiable security. In the following, we provide a more detailed overview of each of these projects.

In chapter 2, we discuss *SkyCore: moving core to the edge for untethered and reliable UAV-based LTE networks*. The advances in unmanned aerial vehicle (UAV) technology has empowered mobile operators to deploy LTE base stations (BSs) on UAVs, and provide on-demand, adaptive connectivity to hot-spot venues as well as emergency scenarios. However, EPC that orchestrates the LTE RAN faces fundamental limitations in catering to such a challenging, wireless and mobile UAV environment, particularly in the presence of multiple BSs (UAVs). In this work, we argue for and propose an alternate, radical *edge* EPC design, called SkyCore that pushes the EPC functionality to the extreme edge of the core network – collapses the EPC into a single, light-weight, self-contained entity that is co-located with each of the UAV BS. SkyCore incorporates elements that are designed to address the uniques challenges facing such a distributed design in this UAV environment, namely the resource-constraints of UAV platforms, and the distributed management of pronounced UAV and UE mobility. We build and deploy a fully functional version of SkyCore on a two (rotary-wing) UAV LTE network and showcase its (i) ability to inter-operate with commercial LTE BS as well as smartphones, (ii) support both hot-spot and stand-alone multi-UAV deployments, and (iii) superior control and data plane performance compared to other EPC variants in this environment.

In chapter 3, we present *SoftBox: a customizable, low-latency, and scalable 5G core network architecture*.SoftBox combines SDN, NFV, and MEC to enable the creation of customized, low latency, and signaling-efficient services on a per user equipment (UE) basis. SoftBox consolidates network policies needed for processing each UE's data and signaling traffic into a light-weight, in-network, per-UE agent. We designed a number of mobility-aware techniques to further *optimize*: resource usage of agents, forwarding rules and

updates needed for steering a UE's traffic through its agent, migration costs of agents needed to ensure their proximity to mobile UEs, and complexity of distributing the LTE mobility function on agents. In this project, we demonstrate that basic SoftBox has by 86%, 51%, and 83%-87% lower signaling overheads, data plane delay, and CPU core usage, respectively, than open source EPC systems. Moreover, our optimizations efficiently cut the peak load in SoftBox networks by 51%-78%. These results point to the feasibility and potential of the SoftBox concepts.

In chapter 4, we explain *SoftMoW: a scalable and reconfigurable cellular WAN architecture*. SoftMoW supports seamlessly inter-connected core networks distributed over a large geographical area (e.g., country or continent) by providing reconfigurable control plane and global optimization. To scale the control plane nation-wide, SoftMoW recursively builds up a hierarchical control plane with novel abstractions of both control plane and data plane entities. SoftMoW supports new network-wide optimization functions such as optimal routing and inter-region handover minimization. In this project, we demonstrate SoftMoW improves the performance, flexibility and scalability of cellular WAN using real LTE network traces with thousands of base stations and millions of subscribers. Our evaluation shows that path inflation and inter-region handovers can be reduced by up to 60% and 44% respectively.

In chapter 5, we focus on *Caesar: a high-speed and memory-efficient forwarding engine for next-generation Internet and cellular core architectures*. Many next-generation network architectures depart from using IP addresses. Instead, they use an addressing scheme that separates network locations and identities with self-certifying, flat and non-aggregatable address components. This addressing scheme has been successful in improving seamless mobility support and guarantees verifiable security. We can easily deploy this addressing scheme in our software-defined SkyCore, SoftBox, and SoftMoW architectures to further optimize them. However, the main challenge with this addressing scheme is that each of the

address components is often long, reaching a few kilobits, and would consume an amount of fast memory in data plane devices that is far beyond existing capacities. To address this challenge, we develop *Caesar*, a high-speed and length-agnostic forwarding engine for future border routers, performing most of the lookups within three fast memory accesses. To compress forwarding states, Caesar constructs scalable and reliable Bloom filters in Ternary Content Addressable Memory (TCAM). Our evaluation shows that Caesar is more energy-efficient and less expensive (in terms of total material cost) compared to optimized IPv6 TCAM-based solutions by up to 67% and 43% respectively. In addition, the total cost of our design is approximately the same for various address lengths.

CHAPTER II

SkyCore: Moving Core to the Edge for Untethered and Reliable UAV-based 5G Cellular Networks

2.1 Introduction

LTE networks that are ubiquitous today are deployed after sufficient RF planning in a region. However, the static nature of LTE base station (BS) deployments limits their ability to cater to certain key 5G use cases – surging traffic demands in hot spots (e.g. stadiums, event centers), as well as their availability in emergency situations (e.g. natural disasters), where the infrastructure could itself be compromised. Providing an additional degree of freedom for base stations, namely *mobility*, allows them to break away from such limitations.

UAV-driven LTE networks. In this regard, recent advances in unmanned aerial vehicle (UAV) technology has empowered operators to take on-demand, outdoor connectivity to another level, by allowing their base stations to be deployed aerially on UAVs (Figure 2.1), thereby offering complete flexibility in their deployment and optimization. Mobile operators like AT&T and Verizon have both conducted trials with LTE base stations mounted on UAVs [8, 10] (helicopter and fixed-wing aircraft respectively, Figure 2.3). AT&T also provided connectivity service from its UAV in the aftermath of hurricane Maria in Puerto Rico last year [9]. Further, with the availability of shared access spectrum like CBRS [7] in 3.5 GHz, this also opens the door for smaller, green field operators to deploy and provide on-

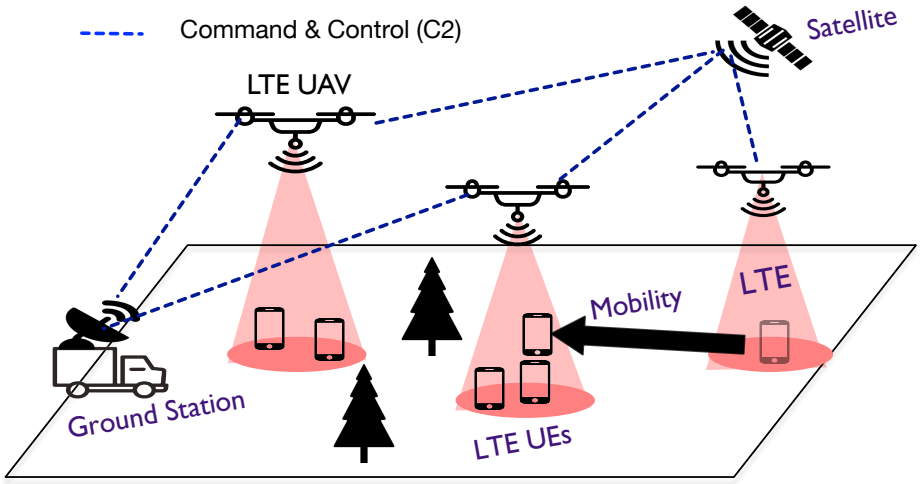


Figure 2.1: LTE UAV networks.

demand, private LTE connectivity service without the heavy cost associated with spectrum and deployment.

Limitations of legacy EPC. A typical LTE network requires the deployment of two essential components: a radio access network (RAN) consisting of multiple base stations (BSs) that provide wide-area wireless connectivity to clients (UEs), and a high-speed, wired core network of gateways (evolved packet core, EPC) that sits behind the RAN and is responsible for all the mobility, management and control functions, as well as routing user traffic to/from the Internet. Realizing a multi-UAV-driven RAN (BSs deployed on UAVs) with an *EPC on the ground* is one way to directly apply today's EPC architecture to the UAV environment (as shown in Figure 2.5) – this has been the case with current operator-driven UAV efforts. However, this faces significant limitations in delivering real value to this challenging environment. Specifically, while a **tethered set-up** (EPC-UAV link being wired, Figures 2.3a, 2.5a) significantly *limits the UAV's mobility and ability to scale to multiple UAVs*, a **wireless set-up** (EPC-UAV link being wireless/mobile, Figures 2.3b, 2.5b) incurs all the *vagaries of the wireless channel*. For the latter, the choice of the wireless technology becomes critical given that the EPC is responsible for setting-up, routing, and tearing down all voice/data bearers. It is essential for the EPC to *reliably reach all the UAVs* wirelessly, including those that are potentially far away in the presence of non-line-of-sight conditions (e..g buildings, foliage, etc.). Further it must deliver sufficient capacity to support

the traffic demands in the RAN. It is extremely challenging for a wireless technology, be it lower frequency (sub-6 GHz like LTE, WiFi, etc.) or higher frequency (mmWave), to simultaneously satisfy the *needs of range, reliability/robustness, and capacity* that the UAV environment demands from the critical EPC-RAN link.

Core at the Edge. Given the fundamental limitations in deploying an EPC on the ground to support a multi-UAV RAN, we advocate for a radical, yet standards-compliant re-design of the EPC, namely the *edge-EPC* architecture, to suit the UAV environment. As the name suggests, we aim to push the *entire* EPC functionality to the extreme edge of the core network, by collapsing and locating the EPC as a single, light-weight, self-contained entity on each of the UAVs (BSs) as shown in Figure 2.7. Being completely distributed at the very edge of the network, such an architecture completely eliminates wireless on the critical EPC-RAN path and hence the crippling drawbacks faced by the legacy architecture in this environment.

While definitely promising at the outset, realizing this radical design is not without its own set of challenges that are unique to the UAV environment. In particular, (i) *Resource-challenged environment:* The compute resources consumed by the numerous network functions in EPC is appreciable and becomes a concern when all the EPC functionality is collapsed into a single node, and deployed directly on a UAV platform – the latter being highly resource-challenged to begin with. This could significantly affect both the UAV's operational lifetime as well as the processing (control and data plane) latency of its traffic, thereby resulting in a reduced traffic capacity. (ii) *Mobility management:* The hierarchical nature of the legacy EPC architecture, gives a single network gateway (like mobility management entity, MME) a consolidated view of multiple BSs, thereby allowing it to efficiently manage handoffs during mobility of active UEs as well as tracking/paging mobile UEs that are in idle mode. Mobility of both active (handoffs) and idle UEs (paging) becomes a critical challenge, when the entire EPC is located at each of the UAVs, thereby restricting their view of events to only those that are local to the UAV.

Our proposal – SkyCore. Towards our vision of building *an untethered yet reliable*

UAV-based LTE networks, we present our novel EPC design, SkyCore. SkyCore embodies the edge-EPC architecture, while introducing two key pillars in its design to address the associated challenges – a complete *software refactoring of the EPC* for compute-efficient deployment on a UAV, and a new *inter-EPC communication interface* to enable fully functional operation in a multi-UAV environment. Through *software-refactoring*, SkyCore eliminates the distributed EPC interfaces and collapses all distributed functionalities into a single logical entity (agent) by transforming the latter into a series of switching flow tables and associated switching actions. It also reduces control plane signaling and latency by pre-computing and storing (in-memory data store) several key attributes (security keys, QoS profile, etc.) for the UEs that can be accessed quickly in real-time without any computation. To ensure complete EPC functionality, SkyCore manages mobility right at the edge of the network – it enables a new control/data interface to realize efficient *inter-EPC signaling and communication* directly between UAVs. This allows the SkyCore agents on each UAV to *proactively* synchronize state with each other, thereby avoiding the real-time impact of wireless (UAV-UAV) links on critical control functions – results in fast and seamless handoff of active mode UEs as well as tracking of idle mode UEs across multiple UAVs.

Real-world prototype. We have built a complete version of SkyCore on a single board server with a small compute and energy footprint; and deployed it on Matrix 600 Pro rotary-wing drones to create a two-UAV LTE network. To the best of our knowledge, this is the first realization of a self-contained edge-EPC solution that can support a multi-UAV network and is a direct affirmation of SkyCore's design. SkyCore's feasibility and functionality is validated by seamless integration and operation with a commercial LTE RAN (BS) from ip.access and off-the-shelf UEs (Moto G and Nexus smartphones). We demonstrate SkyCore UAVs to operate both as hot-spots that allow for better UE connectivity to the Internet, as well as for stand-alone connectivity of geographically separated UEs through two different UAVs (e.g. first responders in emergency scenarios), while also allowing for handoffs. Our real world evaluations of SkyCore and its comparison with a state-of-the-art software EPC

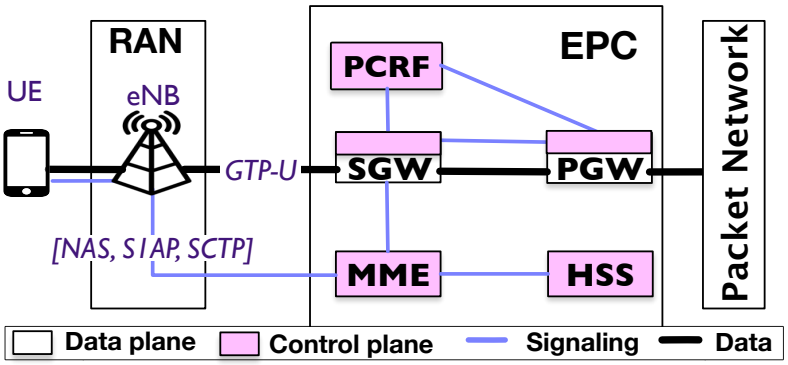


Figure 2.2: Legacy EPC architecture.

(OpenEPC [34]) on UAV clearly showcases SkyCore's superior performance and scalability – SkyCore provides an order of magnitude lower control plane latencies, incurs 5× lower CPU utilization, and provides data plane rates that currently scale up to a Gbps.

2.1.1 Summary of Contributions and Broader Implications

Our two key contributions in this chapter include,

- A novel edge-EPC solution, SkyCore that can *reliably and scalably* support a stand-alone, multi-UAV LTE network deployment that was not possible earlier.

- A real-world implementation and evaluation that showcases both its feasibility and its superior performance.

SkyCore's underlying design is driven by the observation that when connectivity between core network functions, which are on the critical path, is unreliable (wireless and mobile), the merits of pushing functionality to the edge of the network significantly outweighs the associated drawbacks. Hence, although designed for a multi-UAV environment, SkyCore's design can also benefit other deployments, where distributed critical network functions have to communicate over unreliable links (e.g. distributed enterprise RANs).

(a) *(b)*

Figure 2.3: (a) AT&T's and (b) Verizon's Cell on Wings

2.2 Motivation

2.2.1 UAV-based LTE Networks

We consider low-altitude UAV networks, such as those considered by mobile operators [8, 10] for on-demand, LTE network deployments. These are envisioned to serve as dynamic small cells that add capacity to macrocell networks in hot-spot venues, as well as provide stand-alone connectivity (without macrocells) for local communication in disaster scenarios. Relevance of our work to high-altitude, long endurance platforms like Google's Loon [19] and Facebook's Aquila [16] is discussed in Section 4.6. In a UAV-based LTE network, an LTE BS (eNB) is directly deployed on each UAV, and multiple of them together provide wireless connectivity to UEs over a desired wide area as shown in Figure 2.1. However, not much thought has been paid towards the deployment of an EPC to support such a RAN. Deploying and managing a traditional LTE EPC is a challenge in its own right. Designing one to support an LTE RAN on UAVs, which are highly restrictive in their compute capabilities, endurance and payload capacity, further amplifies the associated challenges.

To foster a better understanding, we first reiterate a short primer on EPC's key functionality, followed by the limitations of today's EPC for our target environment, and the benefits and drawbacks of an "alternate" edge EPC architecture.

2.2.2 EPC Primer

Figure 2.2 shows the network architecture of EPC, which is a distributed system of different nodes or network functions (NFs) that are required to manage the LTE network. The EPC

consists of data and control data planes: the data plane enforces operator policies (e.g., DPI, QoS classes, accounting) on data traffic to/from user equipment (UE), while the control plane provides key control and management functions such as access control, mobility and security management. eNodeBs (RANs) are grouped into logical serving areas and connected to serving gateways (SGW). The SGW is connected to an external packet network (e.g. the internet) via a packet data network gateway (PGW). PGW enforces most of data plane policies (e.g., NAT, DPI) and may connect the core to other IP network services (e.g., video server). The mobility management entity (MME) is responsible for access control, security and mobility functions (e.g., attach/detach, paging/handoff) in conjunction with the home subscriber server (HSS) database.

2.2.3 Limitations of Legacy EPC Architecture

The straight-forward way to apply EPC to our UAV network would be to collapse all the EPC network functions into a single node (EPC-in-a-box) and deploy this EPC node on a resource-capable node on the ground that can support multiple UAV BSs. This is the approach adopted by operators like AT&T and Verizon in their recent trials (Figure 2.3) [8, 10].

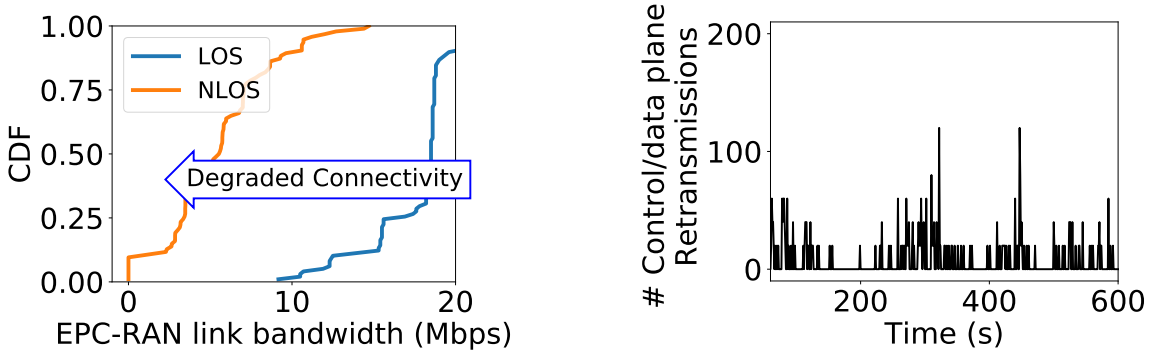


Figure 2.4: (a) Degraded throughput on EPC-RAN link (10 MHz LTE link) when UAV flies in LOS and NLOS (over a building) trajectory. (b) # of SCTP/TCP (user data) retransmissions in NLOS

2.2.3.1 Tethered Deployment (Wired EPC-UAV link)

In today's traditional LTE networks, the connectivity between EPC and eNBs (RAN) is a reliable, wired network provisioned with sufficient bandwidth for catering to the UE traffic demands in both downlink and uplink. A similar approach can be adopted in our UAV network, where the RAN runs on the UAV, which is tethered by a wire to a ground station running the EPC (Figure 2.3a, 2.5a). However, such an approach significantly limits the potential and flexibility of the UAV to fly and re-position itself to cater to network traffic requirements, not to mention the associated safety concerns and the infeasibility of scaling such a set-up to support a network of UAVs. With UAV technology advancing at a rapid pace to provide longer operational times [4], such a tethered EPC-on-ground does not offer a viable, future-proof solution.

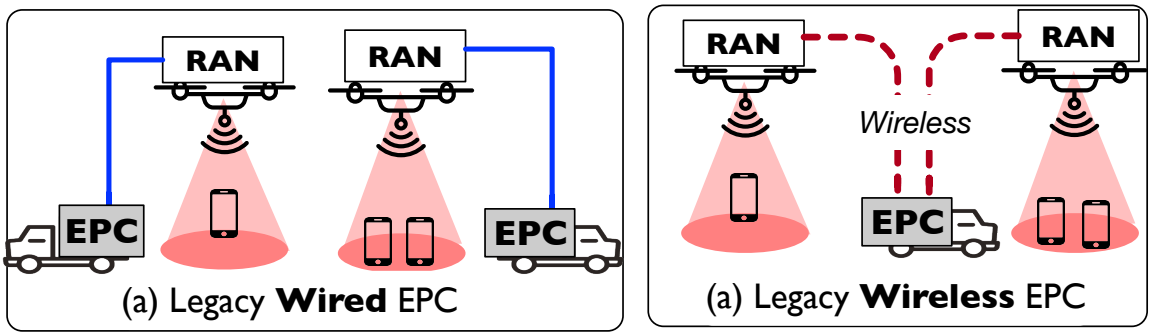


Figure 2.5: Legacy EPC variants for UAV networks

2.2.3.2 Un-tethered Deployment (Wireless EPC-UAV link)

The other alternative is where the connectivity between EPC on the ground and eNBs (UAVs) is wireless (Figure 2.3b, 2.5b).

Reliability vs. range vs. capacity: The wireless channel is inherently an unreliable medium, and is subject to wireless artifacts such as shadowing (building, trees, obstacles), multi path fading, etc. that can significantly degrade signal quality (by as much as 70% in our experiments, Figure 2.4a) and cause high packet retransmissions (more than 100 SCTP/TCP retransmissions, Figure 2.4b) and potentially cause disconnections. The choice of the wireless technology also plays an important role. Using lower frequencies like 700MHz, 1

GHz, etc. allows for better penetration and hence longer communication ranges and better reliability but significantly lesser bandwidth (capacity of few tens of MHz). In contrast, higher frequencies like mmWave (28 gHz, 60 GHz, etc.) offer significantly more bandwidth (hundreds of MHz to a GHz) but suffer from higher attenuation and hence lower range. While the latter can employ beamforming to cope with attenuation, they are limited by line-of-sight requirements and the need to constantly track the beam direction with respect to each UAV as they move – impediment for reliable operation in low altitude deployments. Thus, it is extremely challenging to identify a wireless modality for the critical EPC-RAN (ground to UAV) link that can offer the simultaneous features of reliable connectivity, increased communication range, and capacity, that is warranted by this EPC architecture.

Single point bottleneck: The EPC node on the ground becomes the routing focal point that ferries traffic not only between the UEs and the Internet but also between UEs within the UAV network. Hence, even if the UAV backhaul (connectivity between UAVs) is well-provisioned, having a small set of ground EPC nodes, concentrates all the traffic on the UAV backhaul towards these ground nodes, which in turn become the bottleneck. This would significantly degrade the capacity of the network as a whole. For a low altitude UAV network deployed to provide on-demand connectivity to a small geographic region, bulk of the traffic might be local – e.g. between users and content servers in events, or between first responders and/or affected people in emergencies. In such scenarios, incurring the wireless capacity bottleneck due to EPC on the ground is un-warranted.

A simple illustration in Figure 2.6 shows that the capacity offered by an EPC-on-ground architecture (capacity of x) even for a small 4-UAV network can be 6 times lower than if the local traffic were to be served directly between the UAVs (capacity of 6x). In addition, UAV and UE mobility are highly pronounced in these networks, which also leads to increased control signaling and associated latency over multiple wireless hops between the ground EPC node and the UAVs.

One option is to deploy multiple EPC nodes on the ground to allow for more reliable

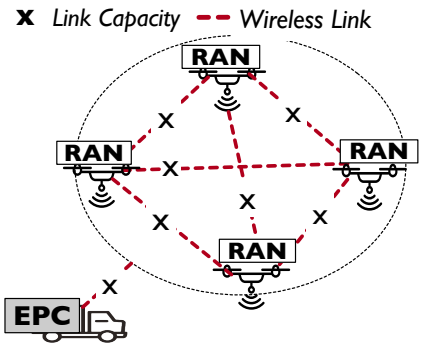


Figure 2.6: Capacity bottleneck

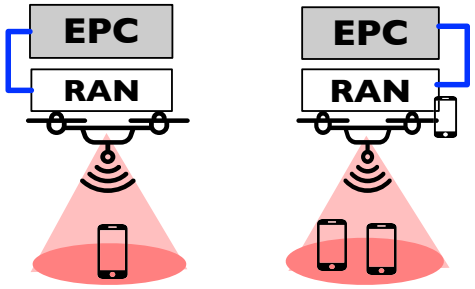
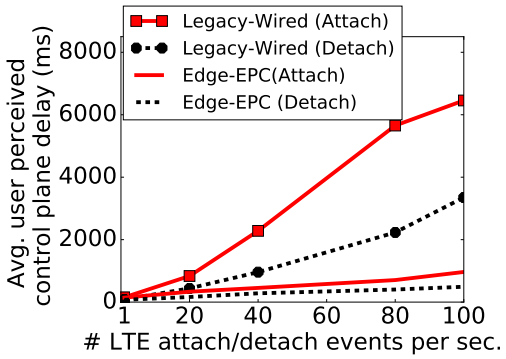


Figure 2.7: Edge EPC for LTE UAVs

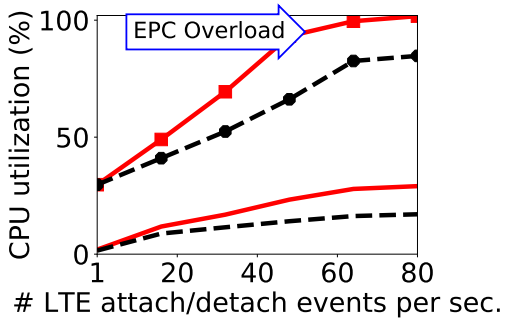
connectivity to all UAVs and to add capacity (akin to provisioning multiple gateways in wireless mesh networks [55]). However, this adds to both the cost as well as reliance on ground deployments, working against the flexibility offered by UAVs in the first place.

2.2.4 Challenges in Edge EPC Architecture

To counteract the challenges in deploying a legacy EPC architecture, we focus our attention to a radically different "edge" EPC architecture. Here, the *entire* EPC is collapsed and located as a single, self-contained entity on each of the UAVs as shown in Figure 2.7. Being completely distributed at the edge of the network, such an architecture would completely eliminate the crippling drawbacks of faced by the previous architecture resulting from wireless connectivity between EPC and eNBs. While definitely promising at the outset, it does encounter a different set of challenges in its realization.



(a)



(b)

Figure 2.8: Edge-EPC has high overheads on UAVs and results in performance bottlenecks and degraded user experience

2.2.4.1 Resource-challenged

An EPC consists of multiple network functions along with the interfaces and tunneling protocols between them. Further, most of these are stateful network functions and consist of both control and data plane functionality. These network functions, which used to be deployed by operators on specialized hardware, are now slowly migrating to a virtualization environment with the recent advances in NFV (network function virtualization [28, 109, 57]. Nevertheless, the compute resources consumed by these network functions is appreciable and becomes a concern when all the EPC functionality is collapsed onto a single node. Deploying an EPC node on the UAV could significantly affect both its operational lifetime as well as the processing (control and data plane) latency of its traffic, thereby resulting in a highly reduced traffic capacity. This can be observed in Figures 2.8a, 2.8b, where the latency and CPU utilization of control plane functions can be an *order higher* in Edge-EPC, when the platform (such as that on a UAV) is resource-constrained (experimental details in Section 2.7).

2.2.4.2 Handling Mobility at the Edge

Conventional EPC has a hierarchical structure, where a single PGW spans multiple SGWs, and a single SGW spans multiple eNBs. As the UE (in active mode) moves from one cell to another (handoff), this is handled locally by its SGW. Further, every UE has a tracking area (TA, set of neighboring eNBs) associated with it, which the EPC will use to page (all eNBs in its TA) to locate it when in idle mode. When the UE moves out of its current TA, it notifies the EPC of its updated TA. Thus, UE mobility is handled seamlessly in legacy EPC.

Active-mode mobility (Handoffs). Network dynamics in the form of UE and/or UAV mobility forms a significant part of our operating environment. However, with the collapse of the hierarchical architecture in Edge-EPC, one needs to now enable communication between the EPC entities on individual UAVs to enable seamless handoff across UAVs. In today's mobile networks, a UE hardly moves across different PGWs within the same operator's

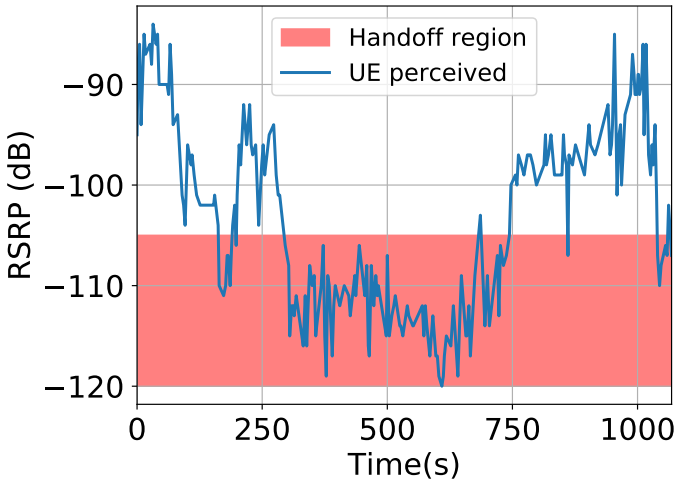


Figure 2.9: EdgeEPC fails in seamlessly handling increased handoffs in our LTE UAV environment

network (a single PGW spans a significantly large area - hundreds of miles). When such an event does happen, the connection is terminated with the existing PGW and re-established with the new PGW causing service disruption. However, such events are the norm rather than an exception in our environment. Figure 2.9 illustrates the number of potential handoff events that can be triggered due to appreciable signal variations even during a short UAV flight (less than 50m) in our experiments. Hence, it becomes critical to enable seamless EPC-EPC communication for handling mobility in the edge EPC architecture. This is needed to also handle UAV mobility, i.e. when one UAV goes down for a re-charge and is replaced by another UAV – a migration of state from one UAV (EPC) to another is imperative.

Idle-mode mobility (Tracking/Paging). With the ability to page idle UEs over large tracking areas (spanning several BSs), it is fairly straight-forward to locate any UE in the network in legacy EPC. This is however, a challenge for the edge-EPC architecture, where there is no single PGW that spans all the UAVs (eNBs). Further, since the notion of tracking area disappears (due to collapsed EPC), locating a UE when in idle mode appears to be infeasible, prompting the need for new or adapted mobility mechanisms.

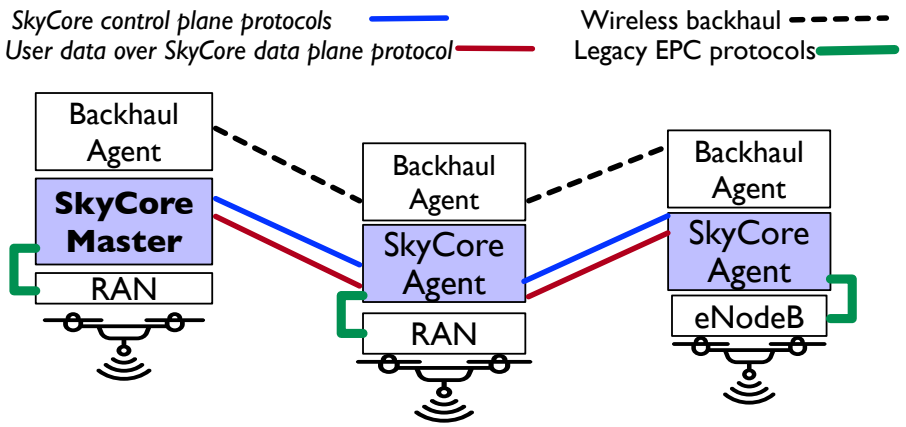


Figure 2.10: SkyCore network architecture

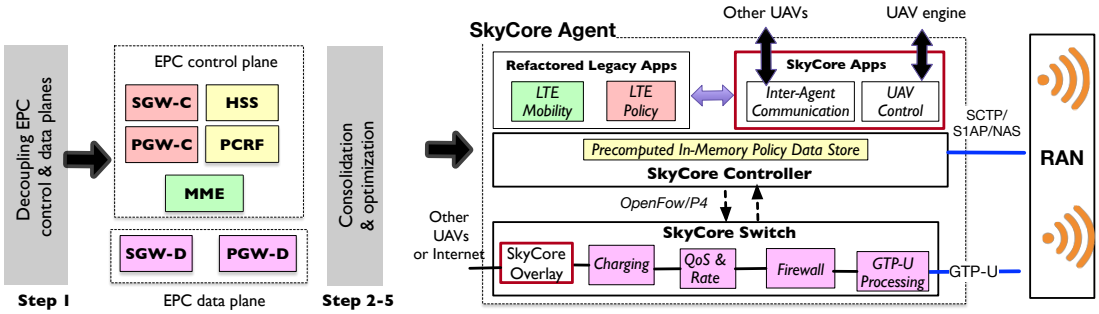


Figure 2.11: SkyCore refactors the EPC functionality into a lightweight agent having new interfaces for interaction with the local UAV and other UAVs.

2.3 SkyCore: Design Overview

SkyCore adopts the edge EPC architecture as shown in Figure 2.10. SkyCore collapses the entire EPC and pushes it to the edge of our network, namely at each of the UAVs themselves, where it is colocated with the RAN. While this completely eliminates wireless from the critical path between EPC and RAN, to address the challenges associated with the edge architecture, SkyCore introduces two novel design components. Briefly,

Software-refactoring of EPC functionality. To reduce its compute footprint on the UAV, SkyCore adopts a software refactoring approach to eliminate distributed EPC interfaces and collapse all distributed functionalities into a single logical entity. It realizes this by transforming the distributed data plane functions into a series of switching flow tables and associated switching actions (corresponding to functions like GTP encapsulation/decapsulation, charging, etc.). It also reduces control plane signaling and latency by pre-computing and storing (in-memory) several key attributes relating to security keys, QoS profile, etc. for the

UEs that can be accessed locally in real-time without any computation.

Efficient Inter-EPC communication. With every UAV now running its own EPC agent, even a simple eNB-eNB handoff of an active UE across two UAVs now becomes a inter-MME (MME-MME) handoff, which needs to be accomplished across two different EPC agents. SkyCore enables a new control/data interface that allows agents on different UAVs to *proactively* (in background) synchronize the state of UEs. This bypasses the real-time impact of wireless (UAV-UAV links) on critical control path functions, allowing for seamless handoffs and tracking of idle mode UEs right at the edge. The HSS equivalent in each SkyCore agent maintains the location (anchoring SkyCore agent) of all UEs in the network. Hence, when an agent sends a UE location update, the agents in other UAVs update their HSS accordingly. Thus, whenever traffic needs to be sent from a SkyCore agent to a specific UE located at another UAV, the HSS will reveal the destination SkyCore agent at which the UE is anchored and to whom the traffic has to be routed. The actual routing path to be taken by the traffic on the mesh backhaul is then determined by SkyCore, with the underlying backhaul topology information made available by a backhaul agent that resides on the UAV[1].

We now explain each of these design components in detail.

2.4 Software Refactoring of EPC

2.4.1 Minimalistic SkyCore Agent Architecture

Each SkyCore agent has a minimalist and UAV-aware SDN-based architecture (Figure 2.11), consisting of a controller that executes the control functions to process UEs' signaling traffic and to coordinate with other agents, and a switch that processes user data traffic. In the following, we describe five high-level steps that we take to refactor and extend the EPC functionality onto our agent architecture.

Step 1. Decoupling the EPC control and data plane pipelines. One of the main

[1]The design of the backhaul agents responsible for maintaining a well-provisioned, connected wireless mesh topology is outside the scope of this work.

reasons behind high complexity and overhead of EPC is its nodes performing mixed control and data plane functions. To make the EPC functionality suitable for UAVs, we first decouple the EPC control and data planes. Among the EPC nodes, MME, PCRF, and HSS are pure control nodes. Hence, our decoupling does not affect these elements, and only affects SGW and PGW. The resulting control components from the decoupling are PGW-C, SGW-C, MME, PCRF, and HSS, and the data elements include SGW-D and PGW-D (C stands for control and D for data). While the benefits of decoupling control and data planes have been articulated before [98, 86], we apply it in the context of UAV networks, and enhance it substantially with the following mechanisms.

Step 2. Categorizing the functionality of the EPC control plane. Next, we categorize the EPC control nodes based on their high-level functionality. In our decoupled EPC, there are three types of control nodes. SGW-C and PGW-C are responsible for managing QoS policy enforcement and routing on user data traffic. MME exchanges signaling traffic with UEs and BSes. PCRF and HSS dynamically generate network security and QoS policies for other nodes. To compress the EPC functionality, we try to consolidate the nodes in each category on top of our agent controller and remove the EPC distributed protocols as follows.

Step 3. Collapsing SGW-C, PGW-C, and MME into light-weight applications. We extract the internal functions in the SGW-C and PGW-C nodes and refactor them into a single SDN application, called Policy Application, on top of the controller. We do the same process for MME and transform it into a Mobility Application. One notable aspect of this consolidation is that we naturally eliminate the complex GTP-C protocol, its six interfaces, and continuous control messages from the core (Figure 2.2). This makes the corresponding SDN applications extremely lightweight and extensible without hurting their original functionality. Note that these applications still exchange information with each other but through simple local publish-subscribe mechanisms.

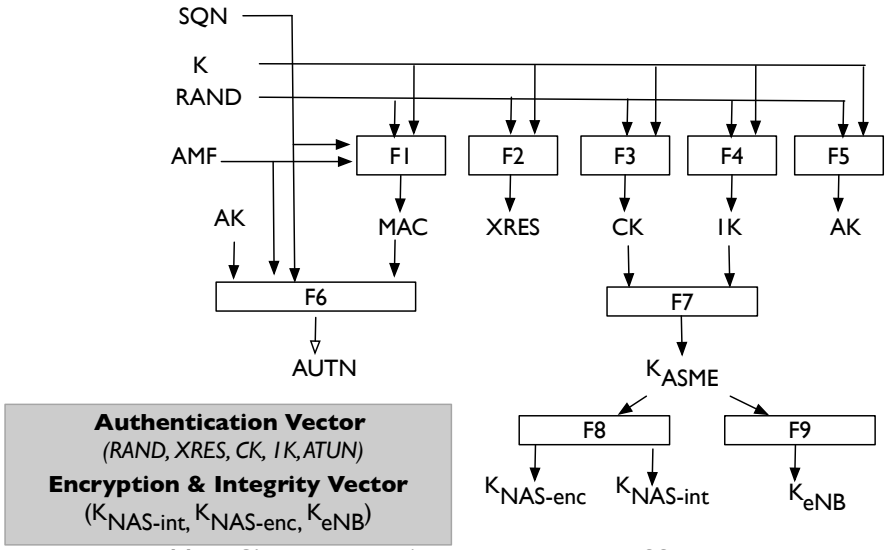
Step 3. Eliminating HSS and PCRF from the LTE UAV core and replacing them with a precomputed policy data store. Next, we focus on HSS and PCRF that are known

to be the source of today's signaling storms in cellular networks [27, 35]. HSS stores hundreds of database tables containing different UEs' states often on disk. Moreover, it acts as a proxy between MME and these tables and performs different types of complex security and location tracking computations. Similarly, PCRF often accesses a logical database (sometimes implemented in the HSS) and dynamically generates different QoS and charging policies of UEs. In SkyCore, we completely eliminate these two nodes from our agents. We show that dynamic policy generation can be carefully replaced with a precomputed in-memory policy data store. Precomputation combined with in-memory transactions substantially minimizes the overhead of the core on resource-challenged UAVs (elaborated in Section 2.4.2). This also removes the complex Diamater protocol from the core.

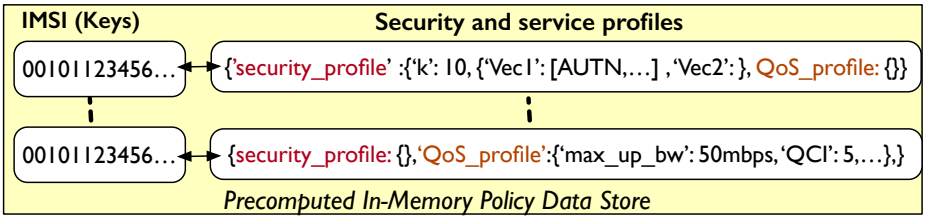
Step 4. Adding UAV-specific applications to the core. One of the key differences between SkyCore and traditional EPC is in its continuous interaction with the UAV hardware and its APIs. In particular, we advocate for two new applications on top of our agents. Each SkyCore agent runs UAV Control Application that listens to flight change events from UAV and remaining battery resources on the UAV. This is necessary for our agents to properly handoff UEs to each other, e.g., when a UAV needs to immediately leave the network for recharging. Such use cases clearly show the potential of our SDN-based UAV-aware architecture. In addition, we design an Inter-UAV Communication Application that exchanges control plane messages with its neighbor agents to synchronize states *proactively*, thereby enabling seamless mobility (active and idle). Other legacy EPC applications and new SkyCore core applications that need to exchange information with each other, do so through our local publish-subscribe protocols (discussed in Section 2.5).

Step 5. Replacing the hierarchical data plane gateways with an SDN switch. Since SkyCore is a flat architecture, it eliminates the need for hierarchical gateways on each UAV. To further make our agents compact, we refactor the functionality of SGW-D and PGW-D into a single software switch. Each data plane function in S/PGW-D is implemented as a separate

Match+Action table in this software switch. Each table performs a lookup on a subset of user's data traffic fields and applies the actions corresponding to the first match. Users' traffic travel through these tables before leaving or entering the UAV. In particular, our software switch performs: (1) UL/DL data rates enforcement, (2) firewall operations, and (3) QoS control by transport-level mechanisms (e.g., setting DiffServ) based on QoS class identifier (QCI) associated with each UE. While legacy EPC tunnels each UE's traffic into two tunnel segments across the RAN, PGW-D, and SGW-D, SkyCore departs from this approach and terminates GTP-U tunnels inside our the agent switch (decapsulates GTP-header from uplink packets sent by the BS and encapsulates a proper GTP-U header in downlink packets to the BS) for two reasons. First, per-UE tunnels do not scale in LTE UAV networks as UEs are mobile and these tunnels are subject to frequent changes. Second, our consolidation already eliminates the need for GTP-U tunnels between the SGW-D and PGW-D functionality.



(a) EPC' heavy security procedures at HSS



(b) Skycore's precomputation of network policies

Figure 2.12: SkyCore's precomptation of network policies not only makes the core resource-efficient but also minimizes network access delay

2.4.2 SkyCore Precomputed Policy Data Store

We now describe how HSS/PCRF can be replaced with precomputed network policies on SkyCore agents. As shown Figure 2.12-(b), the SkyCore data store associates each UE IMSI information with its precomputed policies, which can be quickly accessed by different applications on the agent. Our pre-computation approach not only reduces the user perceived network access delay but it also makes the core extremely resource-efficient. In Section 2.5, we will discuss how policy changes by an agent are propagated to other agents in the network to ensure their state is consistent.

2.4.2.1 Precomputation of Security Policies

Network Access Security in LTE networks relies on the shared user-specific key, K, that is stored in HSS and UE sim cards. The LTE security processes assume that cloned UEs and spoof networks do not know the correct value of K. From K, the HSS dynamically computes (shown in Figure 2.12(a)) an authentication vector (AV) and an encryption vector (EV) as part of a larger LTE attach process, when UEs switch on or enter an area with LTE coverage. The EPC and the UE confirm each other's identities using the AV. The signaling traffic between the a UE and the network is encrypted using the EV to ensure intruders cannot read and modify them. The computation of these vectors involves resource-intensive cryptographic operations (F1-F9) on 256-bit long strings, thereby wasting valuable clock cycles on UAVs.

Offline computation of security vectors. When there are a few UEs, the overhead of computing such security vectors on UAVs is manageable. However, when UAVs are providing on-demand LTE connectivity over a large geographical region, many UEs are likely to send LTE attach requests to the network at the same time. Such realistic workloads can quickly use the available compute resources (for core) on UAVs and substantially degrade the QoE experienced by users. To resolve these issues, our key idea is to depart from real-time security vector computations on UAVs. We precompute and store a reasonable number

of security vectors for each UE and store them on the SkyCore agent. All these vectors are computed with the same *K*, and *RAND* (a random number generated by HSS) but with different consecutive sequence numbers (*SQN*). Different *SQN* numbers ensures signaling messages cannot be replayed by intruders.

Since each pair of vectors is computed with a different *SQN* number, it can be used only once by SkyCore during the LTE attach procedure. If the same pair is reused, the UE will reject the network assuming it is a spoof network that is trying to replay old authentication messages. Thus, each of our agents locally removes a *used* pair of security vectors and invalidates it at other agents through our inter-UAV communication application (see section 2.5). Note that the number of attach requests generated by a legitimate UE, when it switches on or comes back into network coverage, is limited. Hence, SkyCore precomputes a small number of such vectors for a UE. In rare cases, when a UE uses all its precomputed vectors (e.g., due to frequent restarts), SkyCore agents fall back to computing new vectors for such UEs in real-time, and propagate them to other agents in the network.

2.4.2.2 Precomputation of Service Policies

In LTE networks, PCRF dynamically generates quality of service (QoS) and charging rules for a UE. PCRF continuously feeds the PGW and SGW with real time QoS rules. Rather than generating these rules in real-time by accessing many different tables, we precompute the entire rule set that must be applied to UEs' traffic, and consolidate and store them onto our agents. In particular, SkyCore consolidates three types of rules that deal with (i) *QoS* (bit rate, loss rate, etc.), (ii) *priority* (flow handling during congestion), and (iii) *charging* (offline, online and time-dependent).

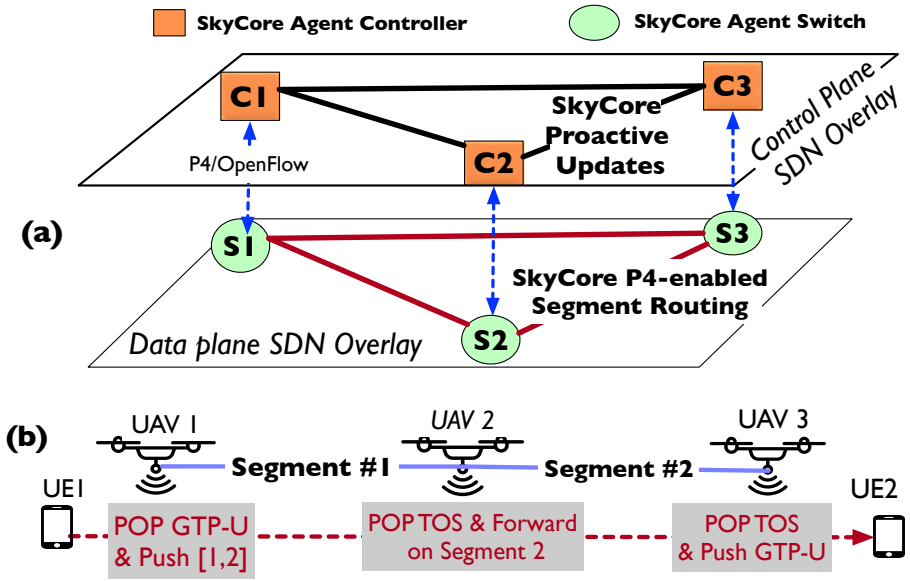


Figure 2.13: (a) SkyCore's network-wide control and data plane connectivity for LTE UAV networks. (b) Example of our segment routing

2.5 Efficient Inter-Agent Communication

2.5.1 Scalable SDN Control and Data Overlays

SkyCore agents seamlessly exchange control and data traffic with each other, a functionality that is lacking in today's EPC instances. We leverage SDN overlays to create two virtualized network layers (slices) on top of the physical UAV network (Figure 2.13). One of these network slices is used for control plane traffic between SkyCore agents and the other is for data traffic. Our separation of the control and data traffic ensures time-critical control plane traffic is not affected when the network is saturated. To form the overlays, we use traffic tunneling technologies but depart from existing approaches used in EPC and datacenter (DC) networking [85, 83] since they require frequent changes to the network configuration (discussed shortly). We adopt a variant of segment-based routing in SkyCore and propose a design for its optimization based on the P4 language [60], thereby allowing operators to define new packet headers for SDN switches.

Segment-based overlays equipped with global source routing. *Tunneling:* We interconnect each pair of *neighboring* (geographic proximity) agents using a tunnel defined with a label. Whenever, an agent decides to send control or data plane traffic to any other agent in the

network, it pushes a stack of labels onto the packets. The top-of-the-stack label corresponds to the next tunnel segment the packet must traverse. Whenever an agent receives a packet from its neighbor, it checks the TOS label from the packet and forwards the packet based on the inner label to its neighbor. There is a master that is responsible for computing the label stack that each agent must use to communicate with the other agents. Instead of adding separate MPLS packet headers for each label, SkyCore designs a new packet header based on the P4 language to contain *all* the labels in the stack to reduce overhead. It equips switches with new actions to read the labels at different positions. *Routing:* In SkyCore, one of the UAVs is selected (periodically) to double up as a master agent that is responsible for global route computation. It periodically collects information from other agents, related to average loss rate and bandwidth on wireless links between different agents (UAVs), remaining battery capacity on UAVs, and the amount of traffic demand between different UAVs. The master agent uses this information to compute and disseminate forwarding rules for routing traffic over the UAV mesh backhaul in the sky. *Proximity-based segments enable scalability:* Note that UAV and UE mobility are common in our environment. Hence, a conventional EPC approach of establishing per-UE tunnels (GTP-U tunnels) will require frequent tunnel updates (tear down, modification, or set up). Similarly, employing a tunnel between every pair of UAV agents (akin to remote DC-DC tunnels) will require updates to a large fraction of the tunnels, even when only a small number of UAVs move. In contrast, most of SkyCore's tunnel segments do not change in such scenarios as they are designed to carry aggregate traffic only between nearby pair of UAVs.

2.5.2 Proactive Stateless Mobility Support

SkyCore replaces the notion of centralized HSS and PCRF with precomputed policy data store replicated at different agents. Hence, it is essential that the UE states and policies are consistent across different agents, particularly during UE mobility. Reactive approaches to consistency management e.g., Distributed hash table (DHT) [120], put wireless (inter-UAV

links) on the critical path of control functions. SkyCore avoids this real-time dependence by adopting a *proactive synchronization* of state between agents – each agent proactively broadcasts its changes to UE policies and states to other agents in the network. Such an approach (i) minimizes the control plane delay between agents, particularly in mobility scenarios as the destination agent already knows the latest information about the mobile UE; (ii) enables seamless handoff of active UEs to a neighboring UAV, when the current UAV goes down for a recharge; and (iii) is scalable because the amount of control plane traffic that is broadcasted is negligible compared to user data plane traffic among agents. A SkyCore agent needs to send only three types of broadcast update messages in the network to build up a consistent network-wide view: (1) security update to notify other agents that it has used one of the security vectors precomputed for a UE and to request other agents to invalidate the vectors. (2) location update to inform other agents that a particular UE has attached to its UAV. (3) policy update to communicate its local changes to the precomputed QoS and charging profile of a UE.

2.5.2.1 Idle-Mode and Connected-Mode Mobility

SkyCore's proactive state synchronization scheme accelerates the handling of increased mobility events in multi-UAV LTE networks.

Idle-mode mobility (Paging). The underlying edge-EPC design in SkyCore limits each UAV (BS) to its own tracking area. Hence, when our target UE When an idle mode UE moves from one UAV to another, it realizes a change in its TA on waking up (prompted by a periodic timer), and sends a TA update request to the SkyCore agent on the new UAV. Since the agent at the new UAV already has the UE's latest policies and states from SkyCore's proactive updates, it knows which security vectors to use for communication with the UE. Hence, it immediately sends a TA update response back to the UE, which can then quickly switch back to its idle mode to continue saving power. It also broadcasts the updated location of the UE to all other SkyCore agents in the network, eliminating the need for explicit UE

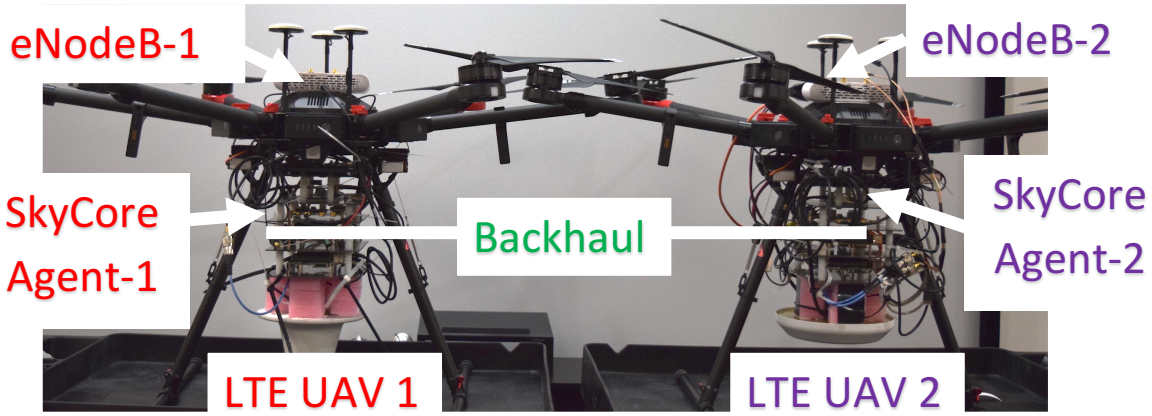


Figure 2.14: Multi-UAV SkyCore prototype

paging. This also ensures that the other agents can push the correct label stack on the packets destined for this UE.

Active-mode mobility (Handoffs). Now, consider the UE to instead be in connected (active) mode during the move. Based on the LTE protocol, it performs continuous signal strength measurements and sends them to the first UAV's BS. If the BS detects the signal strength of the neighboring second UAV to be stronger, it sends a handoff request message to its SkyCore agent. This agent then notifies the agent on the second UAV of the incoming UE (without having to transfer/update any state on the destination agent) and then confirms the handoff with its own BS, which then informs the UE. Then the UE connects to the BS on the second UAV, whereupon its SkyCore agent notifies all other agents in the network with a location update for this UE. Finally, our agent on the first UAV pushes the updated label stack corresponding to the UE onto its pending downlink packets and forwards them to the second UAV.

2.6 Implementation

SkyCore prototype. We prototyped a complete version of SkyCore that involved extensive engineering effort. Our prototype has four notable features: (1) seamlessly works with commercial LTE RANs and off-the-shelf UEs (sim-cards are programmed to connect to SkyCore) by exchanging signaling and data traffic with them; (2) is fully virtualized and can manage multiple LTE UAVs out of the box by forming a wireless network of SkyCore agents; and (3) fully adheres to our proposed designs both for a single agent (Figure 2.11) and across

agents (inter-agent communication) (Figure 2.10 and Figure 2.13). Each SkyCore agent consists of a controller enforcing control plane policies and a switch processing user data traffic. We developed a high-performance multi-threaded controller in C++ and built our SkyCore switch on top of OVS [105] software switch in the kernel space. We substantially instrumented and optimized OVS as it does not support our custom flow tables and switch actions (e.g., our P4-enabled tunneling scheme and GTP-U tunnel encapsulation/decapsulation operations). Since our baseline (Edge-EPC based on OpenEPC [34]) operates in the user space, we developed another variant of the SkyCore switch in the user space on top of Lagopus software switch [23]. This ensures that our comparisons are at the architecture level and independent of a particular packet forwarding technology.

UAV experiments. We conduct three kinds of experiments. *(1) Outdoor Small-scale: 2 UAV, few UEs.* We deploy the SkyCore prototype on two DJI Matrice 600 Pro drones (an advanced off-the-shelf drone). We securely install two machines on each of drone. One of the machines (platform P1) is a low-end single-board 4-core server with 8 GB of RAMs and 1.9GHz CPU that executes SkyCore and Edge-EPC. It is also equipped with a wireless network card to support our inter-agent communication. The other machine supports a commercial LTE small cell (ip.access S60 eNB) supporting LTE UEs (50Mbps downlink rate per UE) and connects through an Ethernet cable to platform P1. *(2) Outdoor Large-scale: 2 UAV, tens of UEs.* To stress test SkyCore's control plane in the presence of a large number of UEs, we replace the eNB on the drone with another single-board server that runs a unified RAN emulator (emulates both eNB and activity of a large number of UEs). The emulator interacts with the LTE core similar to real UEs. *(3) Emulating Powerful UAV platforms.* To understand SkyCore's performance with more powerful UAVs, we emulate the latter by replacing platform P1 with a high-end server (platform P2) – an Intel Xeon E5-2687W processor operating at 3.0 GHz with 12 CPU cores and 128 GBs of RAM. Since it is not possible to fly our current drone with such a server, only these experiments are conducted in the lab.

Metrics. We study four performance metrics: (1) UE-perceived control delay in network

access (LTE attach/detach), (2) UE-perceived service disruption time in LTE active/idle-mode mobility, (3) CPU usage on our resource-constrained UAVs, and (4) supported data plane rate for user traffic.

2.7 Evaluation

We show the basic functionality and potential of SkyCore in realizing hotspot and stand-alone LTE UAV networks. We then demonstrate that SkyCore is more efficient and lightweight than Edge-EPC on different platforms both in small and large-scale experimental settings, thanks to SkyCore's software refactoring and efficient inter-agent communication scheme.

2.7.1 Small-Scale On-Drone Evaluation

We form a two-drone LTE network, each in partial line of sight (affected by one building) of a single mobile UE on the ground. Each drone covers a region with the diameter of 650 feet. The drones operate in a small overlapping area for our mobility experiments.

2.7.1.1 Basic Functionality: LTE Hotspots Use Case

Forming on-demand hotspots is an important use case for LTE UAV as well as 5G networks [67]. In a single-drone experiment, we show this functionality by connecting one of our drones to the Internet through a wireless network not accessible to our UEs on the ground. Next, we turn on a Moto G phone on the ground, which sends an LTE attach request to the SkyCore agent through the on-drone eNB. SkyCore agent successfully completes the LTE attach process by quickly accessing its precomputed policy data store. Then, we visit CNN.com and watch a 4K Youtube video on the phone. Finally, we take Moto G into the airplane mode, causing the UE to properly detach from our agent. Figure 2.15 shows this basic functionality by depicting the data traffic exchanged between the UE and the Internet.

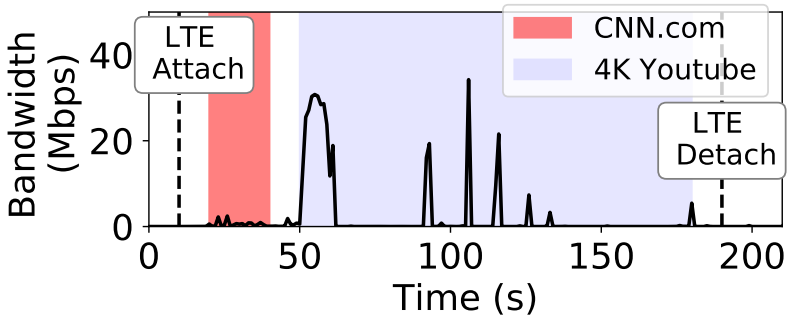


Figure 2.15: LTE hotspot use case–exchanged data traffic over time

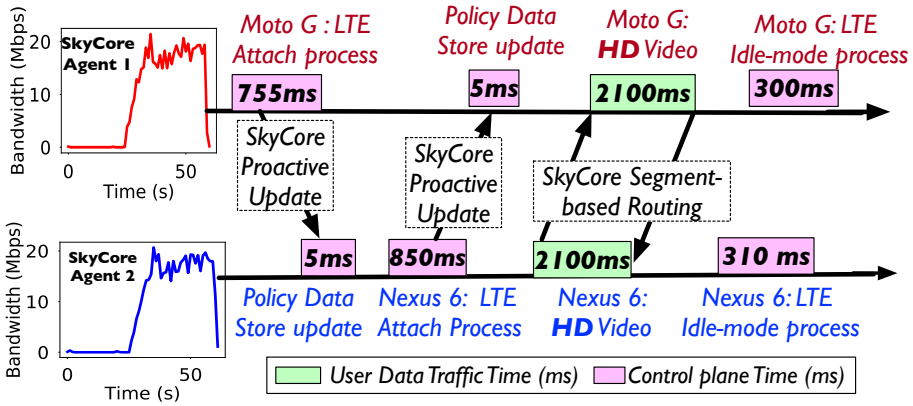


Figure 2.16: Standalone LTE network use case–control plane timeline

2.7.1.2 Basic Functionality: Standalone LTE Use Case

Next, we show SkyCore's ability to create standalone LTE networks (e.g., between first responders across an impassable mountain). To emulate such a scenario, we establish a direct video call between our two UEs across a building, each connected to a separate drone, through our inter-agent data plane overlay. Figure 2.16 shows the timeline of control and data plane traffic exchanges between the two drones. We again turn on a Moto G phone in the area covered by the first drone. Its SkyCore agent handles the LTE attach process and sends a background SkyCore update message to the other donor's agent. The update message consists of UE's location and security update messages as described in Section 2.3. After the second agent processes this update, we turn on a Nexus 6 phone in the area covered by the second drone, triggering a similar SkyCore update message to the first agent in the background. Finally, we establish a 35-sec HD video call from Nexus 6 to Moto Go. Owing to SkyCore's proactive background updates, the agent corresponding to Nexus 6 does not have to wait to

discover the location of other UE. Based on our segment-based tunneling scheme, it immediately pushes correct label stacks on its egress user data traffic and forwards it to the other agent. A similar process manifests in the reverse direction. In this two-UAV enabled video call, 7.5K video packets were successfully exchanged between the two UEs and delivered good quality.

Table 1: Benefits of refactoring on UE-perceived QoS

	Avg. Data plane Bandwidth (Mbps)		**Avg. UE-perceived Control delay (ms)**	
	Downlink	Uplink	Attach	Detach
SkyCore	48.2	17.8	921	300
Edge-EPC	21.7	10.9	1545	750

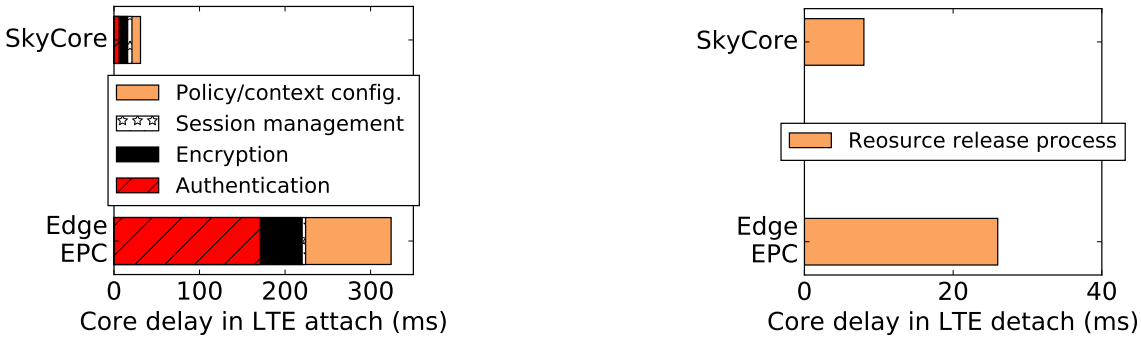


Figure 2.17: Breakdown of network access delay

2.7.1.3 Performance Benefits of Refactoring

Using the same setting, we show that SkyCore is significantly more lightweight than Edge-EPC. For a fair comparison with Edge-EPC, we employ SkyCore's user space version here. We sample and average the LTE attach/detach delay and uplink/downlink bandwidth for Moto G in the area covered by the first drone at 40 locations. As Figure 2.17 and Table 1 show, SkyCore on average reduces the net control plane delay spent in the core by 69%-90% and the UE-perceived control plane delay by 40%-60%. In addition, it doubles the uplink/downlink rates measured for the UE. Further, SkyCore lowers the CPU usage on the LTE core machine by 25% in the LTE attach/detach events. These savings come from our precomputation of network policies and consolidation of the EPC functionality into a compact SDN design.

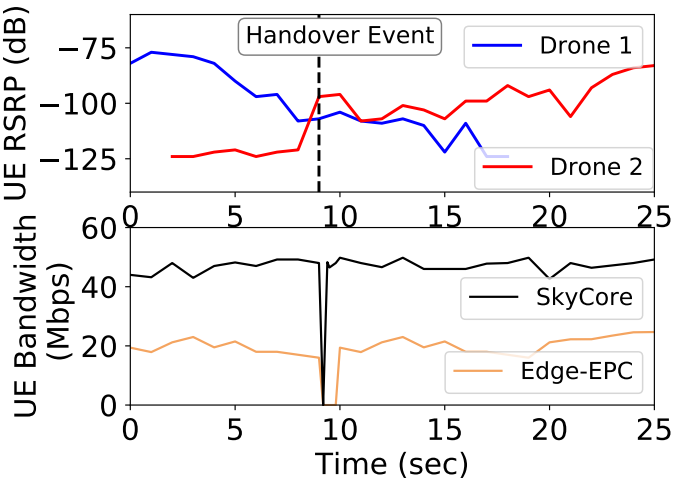


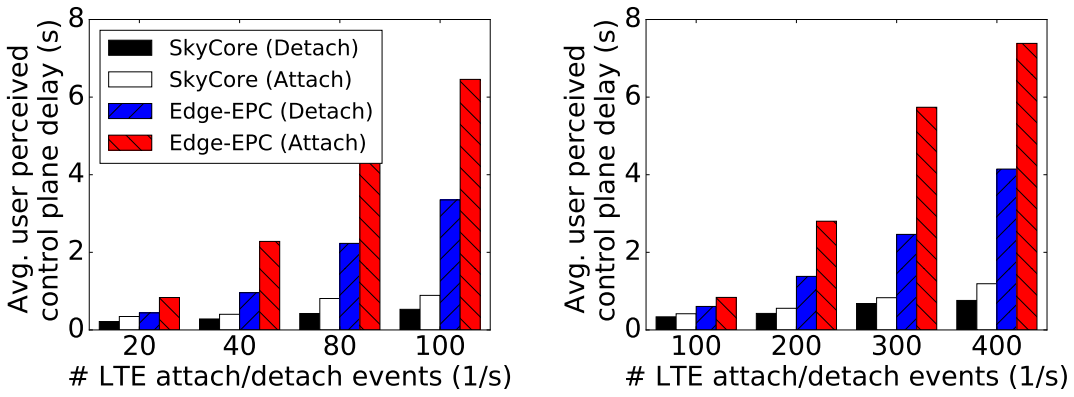
Figure 2.18: SkyCore provides seamless active-mode mobility while Edge-EPC causing severe connection drops

2.7.1.4 Efficient Inter-Agent Communication: Handoff

Unlike Edge-EPC, SkyCore supports seamless UE mobility. In this experiment, we measure the service disruption experienced by a mobile UE moving between regions covered by our two drones. Figure 2.18 depicts the signal strength received from the two drones on the UE and its continuous bandwidth measurements using `iPerf`. Based on the measurements received from the UE, the RAN on the first drone sends a Handoff Required message to the core. In SkyCore, since the agents are already synced, the UE gets migrated to the second drone within a minimal 140 ms (incurred in inter-agent coordination). In contrast, Edge-EPC does not handle mobility of the UE and thus forces the UE to go through the detach process with the first drone, followed by the heavy attach process with the second drone. The entire process results in 2 seconds of disconnection time, significantly impacting mobile application performance.

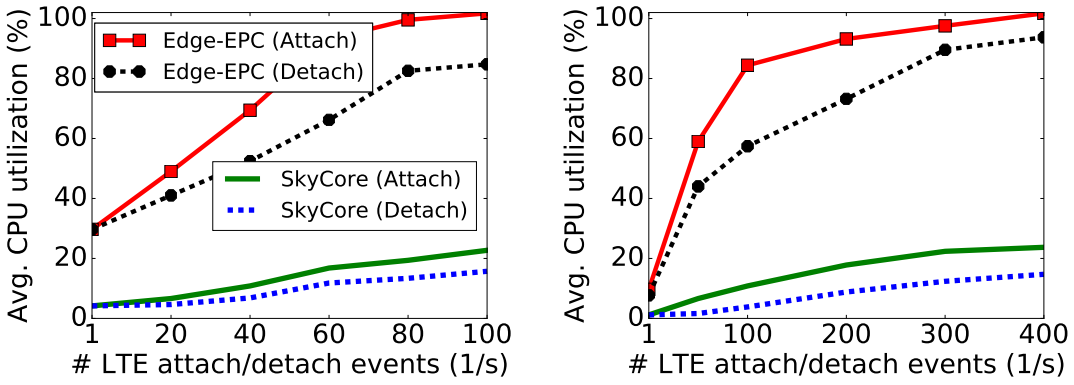
2.7.2 Large-Scale On-Drone Evaluation

Using the same two-drone experimental setting, we replace the ip.access eNB with a RAN/UE simulator on each drone to test SkyCore and Edge-EPC under large-scale network access and mobility workloads.



(a) Platform P1 *(b) Platform P2*

Figure 2.19: SkyCore substantially reduces network access time in LTE UAV networks within the limits of their compute resources



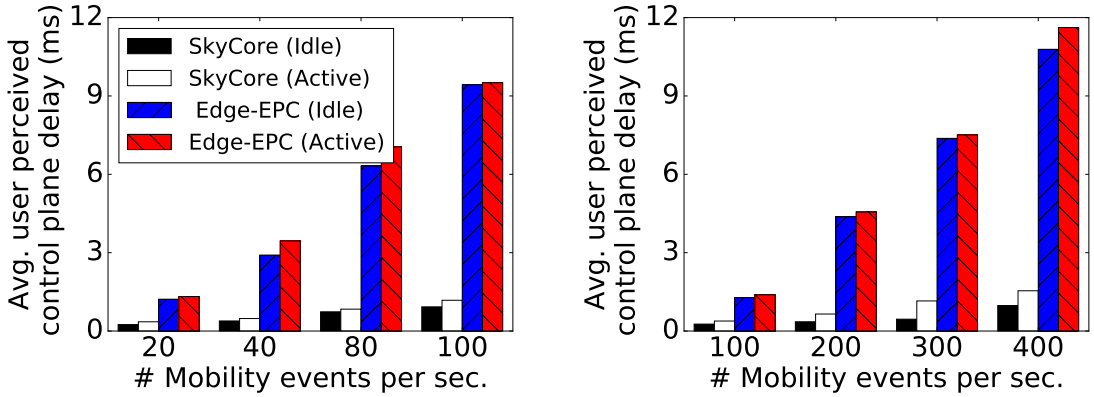
(a) Platform 1 *(b) Platform 2*

Figure 2.20: SkyCore uses minimal CPU resource to handle large-scale network access requests

2.7.2.1 Attach/detach Storm (Flash Crowd)

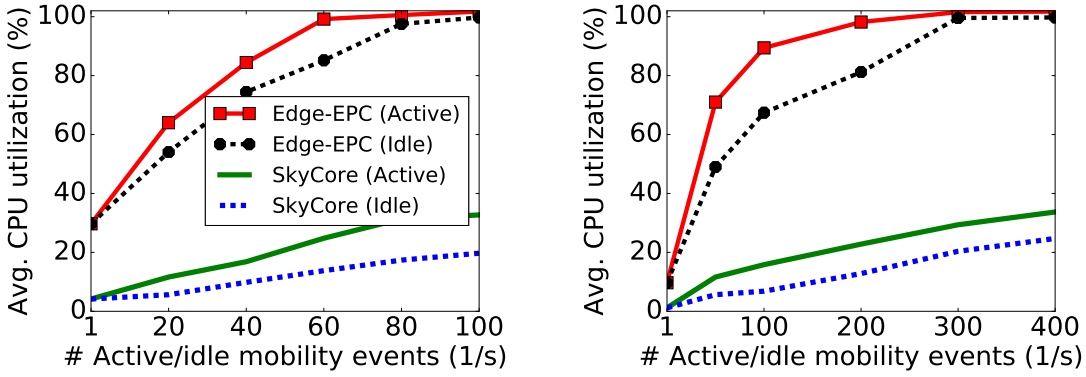
This experiment demonstrate SkyCore's operating potential in highly resource-constrained UAV environments. Our RAN/UE emulator on the first drone emulates a flash crowd event with a large number of users entering the region covered by the drone. Similarly, the emulator creates LTE detach storms having many users gracefully disconnecting from the drone. During this process, we sample the CPU utilization of the LTE core machine and measure the average control plane delay perceived by UEs. In Figure 2.19a we observe that users experience exponentially larger delays when the attach/detach load on Edge-EPC increases. In particular, when the number of attach requests reaches 100, UEs must wait by up to 6 seconds before connecting to the network, thereby degrading QoE. In contrast, we notice that the network access delay is below 1s when the drone employs SkyCore. To

better understand the reason, we look at Figure 2.20a showing the CPU utilization of the core machine. Since EPC is a complex system, we observe that Edge-EPC quickly uses available CPU resources on the drone and thus faces performance bottlenecks. Although user perceived control plane delay in the detach process is usually less critical in practice, the same trend can be observed for both SkyCore and Edge-EPC.



(a) Platform P1 *(b) Platform P2*

Figure 2.21: SkyCore efficiently and seamlessly supports large-scale idle-mode and connected-mode UE mobility between UAVs.



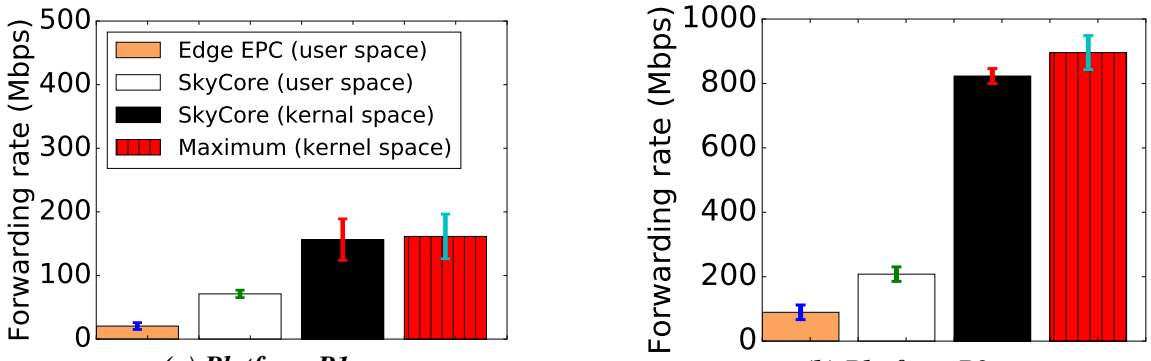
(a) Platform 1 *(b) Platform 2*

Figure 2.22: SkyCore supports large-scale idle-mode and connected-mode user mobility among UAVs in a resource-efficient manner

2.7.2.2 Mobility-Intensive LTE UAV

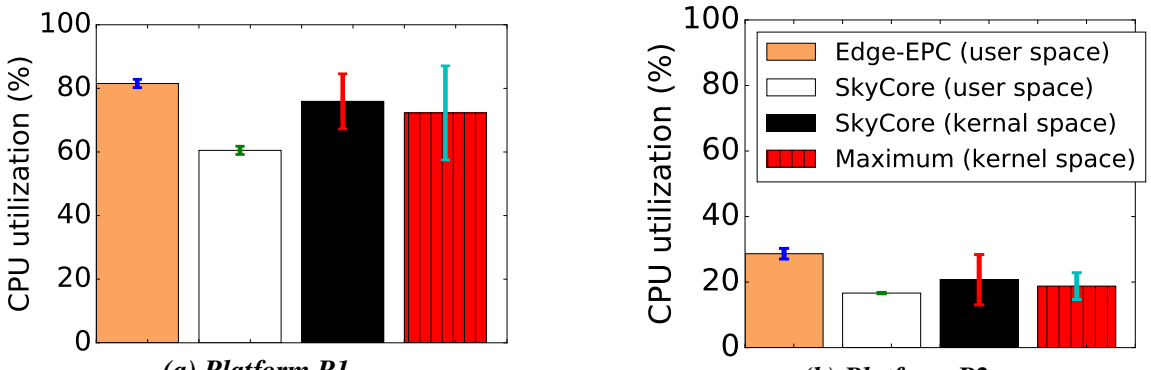
This experiment demonstrates SkyCore's capability in handling increased mobility events in LTE UAV networks. We add the second drone to our experiment. Our RAN/UE simulator on the first drone and second drone simulate scenarios where a large number of connected and idle UEs move between the areas covered by the two drones. We increase the number of

mobility events until either Edge-EPC or SkyCore face performance bottlenecks. Focusing on the active-mode mobility, the RAN/UE simulator on the first UAV sends a variable number of LTE Handoff Required Messages and TA Update Requests to the core network to trigger active-mode and idle-mode mobility events. We measure the service disruption experienced by the UEs when Edge-EPC and SkyCore are in place as well as the CPU utilization of the LTE core machine. Figure 2.21a shows that UEs experience a large control plane delay and service disruption in the Edge-EPC deployment. Due to lack of control plane communication between Edge EPC instances, UEs have to undergo a complete detach (first drone) and attach (second drone) process both during connected-mode and idle-mode mobility. In Edge-EPC, when 100 mobility events occur per second, users on average experience by up to 10 second of disruption, which is very significant. More importantly, by transforming each mobility event to a pair of LTE attach-detach events, we observe in Figure 2.22a that Edge-EPC creates severe bottlenecks on the drone platform. In contrast, the SkyCore agents sitting on the two drones quickly and seamlessly execute the handoff and TA update operation, owing to proactive synchronize of network policies associated with different UEs in the background. Thus, they incur minimal computation during mobility workloads.



(a) Platform P1 *(b) Platform P2*

Figure 2.23: SkyCore's refactoring of the EPC increases the data rate support on resource-challenged UAVs.



(a) Platform P1 *(b) Platform P2*

Figure 2.24: SkyCore's refactoring of the EPC minimizes the CPU resource needed on UAVs for achieving a specific data rate

2.7.2.3 Stress-testing Data Plane

In a single-drone experiment, we instruct our RAN/UE simulator to generate data traffic for variable number of UEs in the network in parallel. It encapsulates the traffic of each UE into a separate GTP tunnel similar to real RANs. We run `iPerf3` bandwidth tests for the simulated UEs in parallel. Figure 2.23a shows the aggregate, steady forwarding rate supported by SkyCore and Edge-EPC. When using the same packet forwarding technology, we observe SkyCore (user space) supports 2× more packet forwarding rate compared to Edge-EPC on the drone. Our software refactoring and data plane consolidation substantially removes the I/O costs and processing delays from the LTE core data plane. We were able to further improve the throughput by 2× (close to a Gbps) by moving our software switch to the kernel space. Maximum is the ideal version of our OVS switch that processes user data traffic without applying any network policies.

2.7.3 Scaling to Powerful UAV Platforms

We replace the core machine (platform P1) with a high-end server (platform P2) to emulate more power UAV platforms in the future. Figures 2.19b, 2.20b, 2.21b, 2.22b demonstrate our evaluation results with platform P2 for the previous three experiments. We observe that SkyCore is substantially more resource-efficient than Edge-EPC even on high-end servers. SkyCore is able to scale and provide almost line-rate forwarding rate while using a fraction

of the drone's CPU resources. We plan to move SkyCore's implementation to OVS-DPDK to more efficiently leverage the available CPU cores.

2.8 Related Work

Recently, the wireless networking community have proposed several software-defined EPC solutions. SoftCell [86] and MCORD [28] enhance the programmability of EPC by decoupling its control and data planes. KLEIN [108] optimizes the placement of EPC components on geo-distributed DCs. ECHO [101] deals with EPC-node failure in unreliable public clouds. PEPC [106] scales the EPC data plane by creating a per-UE EPC-in-box. While there are some similarities between SkyCore and these proposals, the differences are significant. These works make minimal or no change to the 3GPP EPC architecture (protocols, nodes), thereby inheriting most of its complexities. In contrast, SkyCore significantly rearchitects EPC for resource-constrained LTE UAV networks. In addition, the prior designs are customized for highly-reliable often hierarchical DC infrastructure, where over provisioning and reactive network updates are inexpensive. In contrast, SkyCore operates in an un-reliable wireless environment, where such approaches are not scalable, and thus optimizes the core in this regard.

There is a rich literature in distributed SDN control planes designs with hierarchical and flat structures (e.g., ONOS [58], [87, 81]). Most of the schemes are designed for DC networks and operate based on a centralized data store or complex consensus algorithms, which are ill-suited for our unreliable multi-UAV environment.

DroneNet [67] extends the coverage of existing LTE cells by creating WiFi on-drone hotspots. Some recent works [91, 125] investigate the optimization of a UAV trajectory for certain mobile users on the ground (e.g., maximize the min average rate among all user). These RAN efforts are predominantly for a single UAV and complementary to SkyCore that focuses on the EPC design for multi-UAV networks.

CHAPTER III

SoftBox: A Customizable and Low-Latency, and Signaling-Efficient 5G Core Network Architecture

3.1 Introduction

In the previous chapter, we identified limitations of the EPC architecture in the city-scale LTE UAV networks. In this chapter, we focus on statewide terrestrial EPC networks that suffer from three critical issues: (i) lacks fine-grained customizability and programmability in both its control and data planes [86], (ii) exhibits large control and data plane delays because of routing UEs' signaling and data traffic through long paths [130], and (iii) consists of complex nodes and protocols generating huge control plane overheads or signaling storms [36]. *EPC's inefficient network policy management* lies at the root of these issues; EPC partitions and distributes policies for a mobile UE on different nodes. These nodes often must be placed in geo-distributed data centers to maximize EPC's efficiency in mobility support [24]. Thus, large forwarding delays, high signaling overheads, and bottlenecks in massive service customization are inherent in EPC (Section 3.2.2).

Motivated by these issues, operators are exploring 5G core network designs [20, 2]. Although the requirements and use cases are not yet finalized, these networks are expected to have three properties [47, 3, 121, 122]: (i) build optimized and customized services on a per-UE basis to support the proliferation of heterogeneous devices (*e.g.,* domestic robots), (ii)

realize ultra-low latency (*e.g.,* sub-5ms) with gigabit experience for UEs that run real-time applications (*e.g.,* AR devices, self-driving cars), and (iii) have minimal signaling overheads that cause severe performance degradation. Today, there is growing consensus among operators that SDN and NFV are among the key technologies for achieving these properties [5]. However, existing SDN/NFV solutions cannot easily realize these properties because they often build on the EPC architecture and its inefficient policy management scheme, and thus inherit main weaknesses of today's EPC networks. On the one hand, virtual EPC designs (*e.g.,* SCALE [57], KLEIN [108], PEPC [106]) are focused on network automation and make no major enhancement to the EPC architecture. On the other hand, SDN EPC designs (*e.g.,* SoftCell [86], SoftMoW [99, 98], MCord [28]), which decouple the EPC control and data planes to independently scale each, make matters worse. Their decoupling further distributes policies associated with mobile UEs on more nodes.

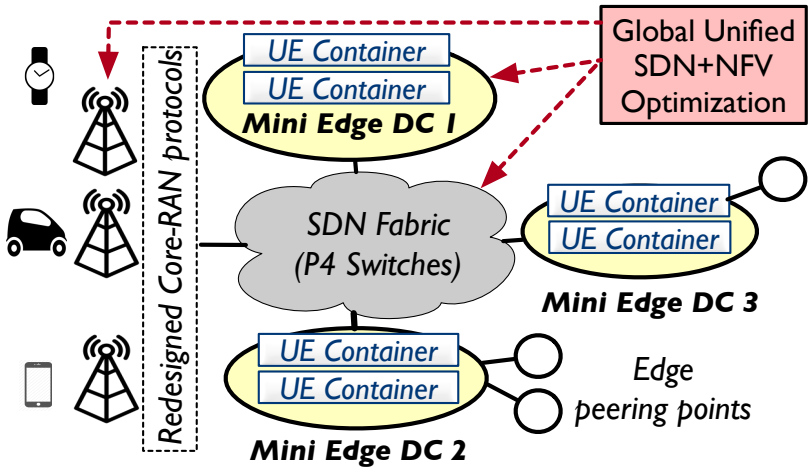


Figure 3.1: SoftBox consolidates the policies associated with each UE into a UE container in its proximity.

To address the EPC issues and realize the 5G core properties, we explore a different point in the SDN/NFV design space. We design SoftBox (Figure 3.1), a radical rethink of the EPC architecture, that replaces the network policies scattered over the EPC nodes far from users with a scalable, flat, and modular architecture where a per-UE agent close to RANs enforces many complex policies (*e.g.,* mobility management). Using NFV, SoftBox creates a light-weight, programmable, logical box for each mobile UE in its proximity. The box consolidates almost all control and data network functions (NFs) needed for processing the

UE's signaling and data traffic. Leveraging SDN, SoftBox programs the data plane to steer each UE's traffic through the corresponding logical box. We argue that SoftBox is a *flexible, scalable, and novel architecture:*

First, the SoftBox architecture meets the requirements of 5G core networks by enabling operators to build customized, low latency, and signaling-efficient services on a per-UE basis. No existing solution supports all these simultaneously (see Table 1). SoftBox can flexibly select and optimize a different set of control functions (*e.g.,* security management) and data functions (*e.g.,* DPI) for each UE box based on the UE's needs and capabilities (*e.g.,* battery life). In addition, SoftBox supports each mobile UE with a consistent low latency experience; it migrates UE boxes independently and ensures each of them always is in the UE's proximity. Finally, our consolidation of policies into UE boxes eradicates EPC's distributed protocols that generate east-west signaling (control plane) overheads.

Second, the SoftBox architecture is scalable. We are not simply proposing a per-UE EPC-in-a-box design [13]. In fact, we rearchitect and optimize the EPC functions for the UE box environment. In addition, SoftBox realizes each UE box using a container that is a *lightweight, isolated Linux process*. Containers have near zero virtualization overheads compared with virtual machines (VMs) [73]. Finally, our UE containers are compact as they only carry the binaries of optimized EPC functions. The combined effect is that *SoftBox systems support substantially more UEs on the same number of CPU cores than EPC systems (6.2-8.3× more).*

Third, SoftBox is a novel solution that redesigns the core from ground up. It goes beyond being a UE container cluster management system (*e.g.,* Google Kubernetes [18]) and carefully addresses five network design and optimization questions: Which and how cellular core functions should be rearchitected for UE containers? How does a SoftBox core interact with LTE RANs? Can we leverage UEs' mobility patterns to more efficiently place and migrate UE containers in the core? Can we leverage UEs' radio state to reduce resource usage of UE containers? And how should we steer each mobile UE's traffic through

its container?

3.1.1 Summary of Contributions and Roadmap

In summary, we make the following *three contributions* in this chapter:

- We propose SoftBox, a scalable and novel architecture for the cellular core that fixes EPC's policy management issue and meets the customization, latency, and signaling requirements of 5G core networks (Section 3.2). We explore the idea of slicing the core into UE containers, flesh out different components of SoftBox, refactor the core functionality into them, and design the lifecycle of UE containers and define their interactions with LTE RANs (Section 3.3).

- We develop novel solutions to further optimize SoftBox by identifying challenges of slicing the core into many UE containers (Section 3.3.6). We design efficient mobility-aware mechanisms to *optimize* resource usage of UE containers (Section 3.4), SDN forwarding rules and updates needed to steer UEs' traffic through UE containers (Section 3.5), control and data plane costs of UE container migrations (Section 3.6), and performance of control plane (signaling) communication between UE containers and RANs (Section 3.7).

- We build a detailed proof-of-concept prototype of SoftBox using open-source software (*e.g.,* Docker container [12], RYU SDN controller [41], OAI EPC [33]). We evaluate SoftBox by combining real LTE traces collected from 200 PhoneLab testbed UEs [38], synthesized LTE traces for 20M UEs, prototype experiments, and RAN+EPC testbed experiments on PhantomNet [34] (Section 3.8).

Summary of results. We show that basic SoftBox has by 86%, 51%, and 83%-87% lower signaling overheads, data plane delay, and CPU core usage, respectively, than two EPC systems (*i.e.,* OAI EPC, OpenEPC). The improvements are independent of the number of

UEs and packet processing technology. Moreover, our optimizations efficiently cut the data and control plane loads in the basic SoftBox by 51%-98% (Section 3.8). These results point to the feasibility and potential of the SoftBox concepts.

3.2 Motivation and Context

In addition to providing the EPC basic functionality (connect user equipments (UEs) to the Internet, handle their mobility in the connected and idle modes, and enforce network policies on their signaling and data traffic), SoftBox has three goals that will guide our design decisions. These goals will be derived based on EPC architecture challenges (Section 3.2.2) and emerging 5G use cases [47, 3].

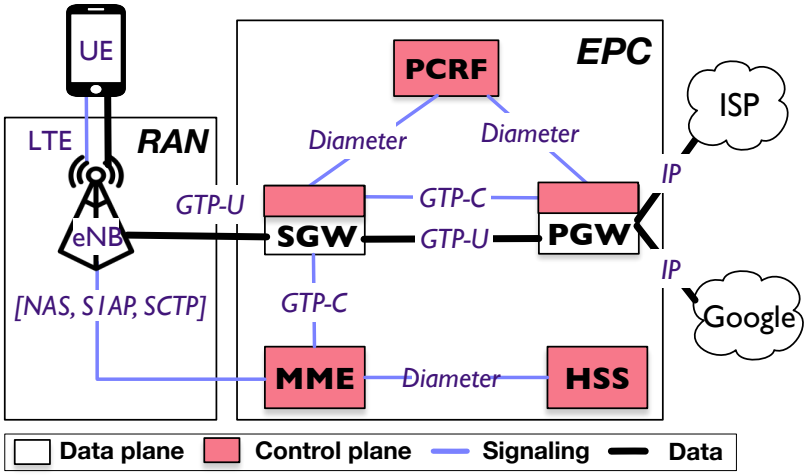


Figure 3.2: EPC network architecture

3.2.1 Design Goals for SoftBox

Goal 1: per-UE customization of control and data functions. With the advent of IoT, UEs connecting to 5G networks will be highly heterogeneous. Without creating performance bottlenecks and network management complexities, SoftBox must enable operators to compose a unique service for each UE or a class of UEs based on their needs and capabilities (*e.g.,* hardware, plan, mobility). The per-UE service customization and optimization must go beyond data plane functions (*e.g.,* DPI) that manipulate UEs' data traffic in the core. It

should also encompass control plane functions (*e.g.,* paging and handover management) that process UEs' signaling traffic.

Goal 2: ultra-low control and data plane latency. Today's well-designed EPC deployments typically have 10-30ms data plane delay and 10-60ms control plane delay [24]. To improve user experience and support a wide range of 5G use-cases (*e.g.,* high-speed mobility, device-to-device communication), SoftBox must be able to create services with ultra-low control and data plane latency (*e.g.,* sub-5ms [20, 92]) for UEs. Achieving this goal is challenging particularly due to diversity of latency requirements across different UEs and their unplanned mobility.

Goal 3: minimal signaling overhead. The problem of cellular network congestion is not much about UEs' data traffic, but in the EPC control plane generating tremendous overheads in response to UEs' signaling traffic [14]. The increasing number of UEs is further escalating the signaling overhead and pushing EPC networks to their limit. Previous studies report the global signaling overhead in EPC networks has increased from 30M to 200M messages per sec in the past three years [36]. Optimizing sources of the signaling overheads is of crucial importance for SoftBox.

3.2.2 EPC Architecture Challenges

Basing the design of SoftBox on the EPC architecture fundamentally limits us in efficiently meeting the above design goals.

EPC *partitions* network policies or service associated with a UE and *scatters* the partial policies on its different nodes (*e.g.,* PGW, MME). To maximize the EPC efficiency, these nodes must be deployed in geo-distributed DCs often far from users [24]. This distributed network policy management has three known consequences. First, large control and data plane delays are unavoidable due to increased propagation delay between distant EPC nodes, extra I/O delays at each node, and sub-optimal routing protocol among EPC nodes [130]. Second, this design requires to frequently synchronize the partial network policies/states

Table 1: SoftBox in the SDN/NFV design space

	Distributed vEPC	SDN EPC	Per-UE vEPC	SoftBox
Examples	[57, 108]	[86, 99]	[106]	-
Realizations of				
+Ultra low latency	○	◔	○	●
+Per UE customization	○	○	●	●
+Low signaling overhead	○	◒	◒	●
Optimizations of				
+Idle UE containers	N/A	N/A	○	●
+Per-UE traffic steering	N/A	N/A	○	●
+UE container migration	N/A	N/A	○	●
+Core-RAN communication	N/A	N/A	○	●
Incrementally deployable	●	●	●	●

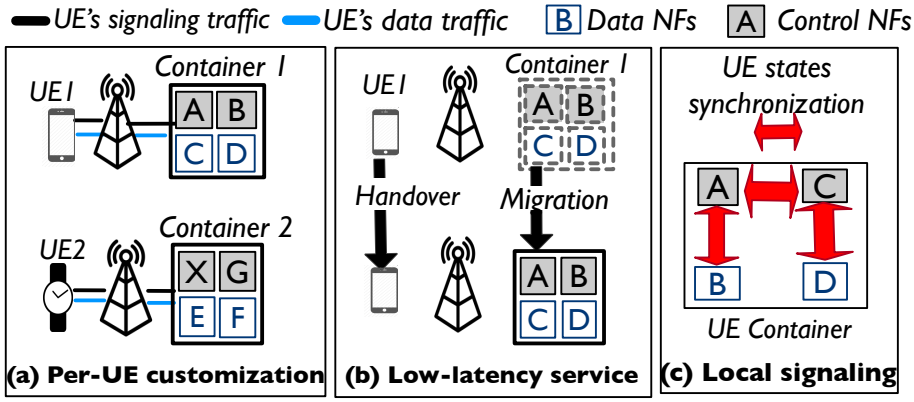


Figure 3.3: Three conceptual benefits of SoftBox core networks

scattered on the EPC nodes (*e.g.,* QoS policies on PCRF and PGW, UEs' locations on MME and SGW) after each UE's control/data packet or flow [14]. The synchronizations, which often happen using complex distributed protocols (*e.g.,* GTP-C, Diameter), incur significant east-west signaling overheads [36]. As the third consequence, per-UE customization of control functions (*e.g.,* paging at MME) and data functions (*e.g.,* DPI at PGW) do not scale. As such a customization enlarges the EPC internal states, together with the delay and signaling overhead issues, performance bottlenecks quickly appear in EPC systems [110, 57]. Existing NFV/SDN EPC systems virtualizing EPC on commodity servers or decoupling its control and data planes often build on the EPC architecture (nodes, protocol, interfaces) and thus, as documented, they have most of its weaknesses (detailed in Section 3.9 and summarized in Table 1).

Table 2: Summary of design decisions in the basic version of SoftBox.

Components	Motivation/Reason	Benefits
UE containers consolidating network policies close to RANs	*EPC challenges are caused by its approach of partitioning policies for a mobile UE on geo-distributed nodes*	• *Enables scalable per-UE customization* • *Realizes ultra-low control and data plane delays* • *Minimizes east-west signaling overheads*
Mini DCs and egress points close to RANs	*EPC is distributed in a few DCs far from RANs and this infrastructure contributes to many inefficiencies in EPC*	• *Ensures proximity of UE containers to mobile UEs* • *Allows quickly exiting/entering UEs' Internet traffic from/to the core*
Programmable SDN fabric interconnecting RANs and mini DCs	*EPC inefficiently routes UEs' traffic, e.g., direct traffic of a UE to a nearby UE is sent to the the Internet*	• *Allows properly steering mobile UEs' traffic through their UE container* • *Enables UE-to-UE traffic over optimal paths*
Unified SDN/NFV controller with distributed agents	*EPC is not designed to perform global network optimizations*	• *Enables global optimization of the core resources and performance* • *Enables scalable execution of optimization results through agents*
Upgrading protocols between LTE RAN and SoftBox	*Legacy EPC protocols (e.g., S1AP/SCTP, GTP-U) do not scale well with the increased number of nodes in SoftBox Changing LTE RANs is impractical*	• *Designed a minimally disruptive plan for their new protocol deployments* • *By placing a proxy inside eNodeBs translating the legacy EPC protocols to efficient SoftBox protocols and vice versa*
Cellular-specific protocols for UE container orchestrations	*Effective orchestration of UE containers without continuous interaction with LTE RANs/UEs is impossible*	• *Efficiently manage UE containers based on LTE events generated by RANs/UEs*

3.3 SoftBox Core Architecture

This section describes the basic design of the SoftBox architecture that overcomes the EPC architecture challenges and realizes our design goals motivated by evolving 5G use cases. Our key design decisions in the basic SoftBox are summarized in Table 2.

3.3.1 Need for the SoftBox Architecture

SoftBox is founded on a simple change to the EPC architecture. SoftBox consolidates each mobile UE's network policies, which are partitioned and placed on geo-distributed nodes in EPC, into a UE container in its proximity. Containers use lightweight OS-level virtualization technologies (*e.g.,* Linux cgroups and namespaces) that allow us to flexibly package a logical service with its entire runtime environment into a single Linux process. A container image can be instantly executed on different servers while retaining its full functionality. Qualitative performance benefits of containers over VMs already presented in Section 3.1. Our approach of vertical slicing of the core functionality into UE containers at the radio edge equips SoftBox with four validated properties (Section 3.8) that no EPC supports simultaneously (see Table 1). Before going into detail, we first present these properties to motivate the need for adoption of SoftBox.

Property 1: SoftBox scalably supports per-UE customization of the core control

and data planes. As shown in Fig 3.3-a, the isolation of UE services from each other enables operators to select and optimize a different set of control and data NFs for each UE, without making other UE services complex or degrading their performance. Another aspect of per-UE customization support lacking in EPC is that SoftBox can flexibly allocate RAM/CPU resources on servers to UE containers based on service requirements. Conceptually, as long as an NF meets the following two criteria, one can place it into UE containers with no modification to its logic: (a) always creates a separate packet processing pipeline for each UE and maintains no shared state for different UEs and their traffic and (b) always performs local computation and does not need global network state to make efficient decisions. Through our careful analysis of the 3GPP-defined EPC architecture [1] and systems [34, 33], we have found all the EPC NFs already possess these properties and thus are amenable to the SoftBox approach of refactoring into UE containers (more detail in Section 3.3.3). Note that to improve the performance of EPC, sometimes operators attach generic middleboxes to PGW (*e.g.,* video optimizer, traffic compression). Most of these NFs meet the above features as well so an operator can flexibly place them in UE containers.

Property 2: SoftBox realizes ultra-low control and data plane delays for mobile UEs by minimizing different factors contributing to large delays in today's EPC networks. By placing the container for each UE in its proximity, SoftBox minimizes propagation delay in the core. Also, SoftBox cuts extra I/O and processing delays, which EPC's redirection of a UE's traffic through multiple nodes incurs. Moreover, by decoupling UE services from each other, SoftBox independently migrates each mobile UE's service to its proximity to ensure a consistent delay experience (see Fig 3.3-b). Finally, by co-locating control and data NFs into UE containers, SoftBox minimizes EPC's large synchronization delay between its control and data plane nodes.

Property 3: SoftBox can minimize signaling overheads in the core. EPC runs complex protocols (e.g., Diameter, GTP-C) to synchronize mobile UE states on its different nodes, causing east-west signaling overheads. SoftBox eradicates the need for these protocols by

centralizing and isolating the core NFs for each UE in a UE container. Because different NFs inside our UE containers synchronize UE states through local message exchanges often on top of inexpensive publish-subscribe mechanisms (Fig 3.3-c), SoftBox significantly reduces the east-west signaling overheads in the core.

Property 4: SoftBox can be deployed at large scale. Later, we show that SoftBox supports substantially more UEs on the same number of CPU cores than today's EPC (6.2-8.3× more) for three reasons. First, we go beyond containerizing EPC and instantiating EPC on a per-UE basis, and optimize the EPC NFs for the UE container environment. Second, containers are lightweight Linux processes with near zero overheads (*e.g.,* virtualization, startup delay) compared to VMs [73]. Third, the binaries of NFs are very small in size and per-UE instantiation of them in UE containers has insignificant overheads.

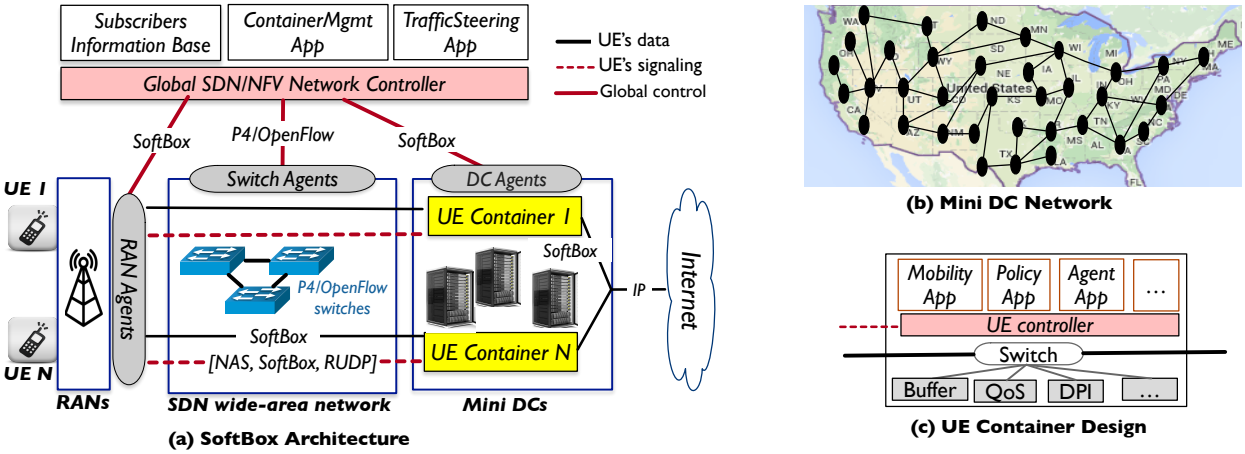


Figure 3.4: SoftBox redesigns the cellular core to build customized, signaling-efficient, and low latency services.

3.3.2 Overview: Transforming EPC into SoftBox

We first provide an overview of five high-level steps that we take to systematically transform EPC into the basic SoftBox. Each step is associated with proper forward references to our detailed technical discussion.

Step 1: Rearchitecting the EPC functionality for UE containers. While each basic UE container is expected to have the EPC functionality at the very least, a naive per-UE containerized EPC is far from our vision for SoftBox. EPC is originally designed for distributed

deployment scenarios and thus is too complex and inefficient for single-box deployments in UE containers. Moreover, EPC is not programmable to support fine-grained per-UE customization as described in Section 3.2.2. To develop SoftBox, we will refactor and optimize the EPC functionality for the UE container environment (Section 3.3.2).

Step 2: Deriving SDN/NFV components of SoftBox. The most naive and incremental realization of SoftBox is to replace EPC with UE containers in today's EPC infrastructure (Figure 3.2). One can put UE containers behind a logical load balancer inside existing EPC DCs and connect LTE RANs to the load balancer instead of EPC. As it turns out, while this naive SoftBox design still makes the core network more scalable and flexible, it substantially limits us in meeting our three design goals (e.g., ultra-low latency for mobile UEs in Section 3.2.1), largely because today's infrastructure is not designed for SoftBox-like solutions. Therefore, we derive a minimal set of necessary software and infrastructure components for SoftBox (Figure 3.4) to ensure it can satisfy our expectations (Section 3.3.3).

Step 3: Connecting SoftBox to existing RANs/UEs. A practical core network design must be incrementally deployable. SoftBox must not force operators to access or modify the network stack of their subscribers, eNodeBs, or routers in peering ISPs. Hence we expand the design of SoftBox to ensure it can efficiently and seamlessly interact with these players (Section 3.3.4). In particular, the same as EPC, SoftBox continue to exchange IP traffic with ISPs and signaling traffic over NAS (Non-Access Stratum) protocol with UEs. For different performance and scalability reasons, we will develop new protocols between the SoftBox core and LTE RANs but we deploy them in a minimally disruptive way without touching the source code of eNodeBs.

Step 4: Orchestrating UE containers. Automatically managing the lifecycle of UE containers (*i.e.,* their creation, placement, migration, and termination) is an essential function in SoftBox. It is impossible for SoftBox to properly perform these tasks without a direct and continuous interaction with RANs and UEs. Existing NFV orchestration schemes (*e.g.,* [107]) have serious functional limitations in this regard as they are originally designed for virtualiz-

ing wired networks functions. Consequently, we will design orchestration mechanisms and protocols in SoftBox to effectively manage UE containers based on LTE events generated by RANs/UEs. In Section 3.3.5, we largely focus on UE container creation and termination and let our more advanced design for other operations to evolve in later sections.

Step 5: Global SDN/NFV optimization of SoftBox. The above steps result in a basic version of the SoftBox architecture, which meets our three design goals in a scalable and efficient fashion (Section 3.8). Our basic SDN/NFV architecture provides unique opportunities for global optimization of network resources and performance. We provide an overview of the process of developing an optimized SoftBox in Section 3.3.6 and Table 2, while presenting our technical solutions in the next four sections.

3.3.3 Software & Infrastructure Components of SoftBox

SoftBox replaces the EPC architecture with a fully SDN/NFV solution that globally controls UE containers in a programmable and elastic environment via open protocols and interfaces. Existing network infrastructures (deployed for EPC) make it either hard or impossible for SoftBox to fully achieve its properties (discussed in Section 3.3.1). Therefore, SoftBox, as a next-generation core network architecture, makes a set of critical changes to the existing infrastructures. A SoftBox network consists of four main components: (1) mini data centers (DCs), (2) SDN switching fabric, (3) UE containers, and (4) global SDN/NFV control plane. We describe the rational behind these components and their functionality.

Component 1: Mini DCs and egress points close to RANs. Today's EPC is distributed in a few DCs far from RANs [24]. The EPC connects to ISPs at a single point where the PGW is located (Fig 3.2). This DC infrastructure inherently contributes to a lot of the inefficiencies in EPC (*e.g.,* large Internet access delay). To ensure UE containers are always in the proximity of mobile UEs, SoftBox envisions a sufficient quantity of mini DCs close to RANs (*e.g.,* a DC per state as shown in Figure 3.4-b). Moreover, for quickly processing UEs'

Internet traffic, a reasonable fraction of mini DCs are connected to the Internet. In general, the number, capacity, and location of mini DCs depend on many factors (*e.g.,* latency requirement). Note that the industry push towards multi access edge computing (MEC) [104] already created the need for such DCs and more egress points so SoftBox can benefit from such trends.

Component 2: Programmable switches interconnecting DCs and RANs. To enforce network policies, SoftBox steers each mobile UE's traffic through its corresponding UE container. A UE may simultaneously communicate with different end points, *e.g.,* other UEs connected to nearby eNodeBs in the case of IoT Apps or external Internet hosts. To quickly set up, modify, and tear down paths in the data plane, SoftBox contains a flat switching fabric of programmable SDN switches that seamlessly interconnect RANs and mini DCs (Figure3.4-a). The inter-DC SDN network (Figure3.4-b) ensures our latency-sensitive DC-to-DC traffic can be routed over efficient paths *inside* the SoftBox core. This type of traffic is generated when UE containers migrate among mini DCs to ensure their proximity to mobile UEs, or when two or more UEs (*e.g.,* autonomous cars) with their UE container in different mini DCs directly communicate. Today's EPC architecture and infrastructure are very inefficient in handling UE-to-UE traffic. First, EPC is designed to always route traffic of a UE to the Internet, even if the destination is a UE connected to a nearby eNodeB. Second, EPC instances (e.g., in different regions) cannot directly communicate with each other as EPC DCs are not connected to each other through direct links.

Component 3: Customizable and optimized UE containers. In SoftBox, UE containers are the smallest unit of service, each embedding a customized bundle of data and control NFs for processing a UE's data and signaling traffic. Our basic yet extensible UE containers provide the EPC functionality. However, they are extremely more efficient, lightweight and programmable compared to an EPC-in-box design. Our basic UE containers are derived from EPC in two steps as follows:

1. *Optimizing the EPC functionality*: We first extract the functionality distributed on the EPC nodes (*i.e.,* S/PGW, MME, PCRF, HSS in Figure 3.2) and refactor them into a set of programmable NFs. Each container is a single process, where communication between its internal threads occur through local message passing mechanisms (*e.g.,* pub-sub). Since we consolidate the NFs into UE containers, we naturally remove the EPC distributed protocols (*i.e.,* Diameter, GTP-C) from their implementation. We also upgrade the protocols that run between EPC and RANs (e.g., GTP-U, S1AP/SCTP) with more scalable ones (will be explained in Section 3.3). Our approach of removing/upgrading complex EPC protocols make the NFs and UE containers fast and lightweight at the end. In particular, through the refactoring process, our basic UE containers are equipped with the EPC data functions (DPI, NAT, firewall, buffers, accounting/QoS) and control functions (LTE mobility, security, and policy management NFs).

2. *Developing a minimalist SDN architecture:* Next, we design a programmable SDN architecture for UE containers by decoupling their control and data planes. As shown in Figure3.4-c, SoftBox isolates each of the data NFs into a Linux network namespace and interconnects them using a software switch. Then, it places the control NFs on a platform called **UE controller** running at least two applications: *a) MobilityApp* executing the LTE mobility management operations (*e.g.,* handover, paging) and security management operations (*e.g.,* authentication, encryption) by exchanging signaling traffic with the UE through RANs. *b) PolicyApp* enforcing data plane policies (QoS, monitoring, charging) by forwarding the UE's data traffic through different in-container data NFs.

Component 4: Unified SDN/NFV control plane. Finally, SoftBox core networks are

equipped with a global SDN/NFV controller (Figure 3.4-a) that optimizes their performance and resources (*e.g.,* minimizes migrations of UE containers and flow rules in the SDN fabric). For scalability and performance reasons, the global controller places an agent in each DC, SDN switch, and eNodeB to execute its commands. Our global controller and SDN switch agents communicate over the standard OpenFlow/P4 protocol. However, due to lack of protocols for configuring eNodeB/DC agents, we develop custom protocols between our global controller and them (will be presented in Section 3.3.5, Section 3.5-Section 3.7) .

3.3.4 Connecting SoftBox to LTE RANs and UEs

SoftBox is an incrementally deployable solution as elaborated in Section 3.3. Here, we explain how SoftBox connects to existing LTE RANs and UEs without any direct modification to their network stack. In RAN+EPC networks, each LTE eNodeB exchanges two types of traffic with EPC (see Figure 3.2). Without loss of generality, consider the traffic in the uplink direction. An eNodeB encapsulates each UE's data traffic into a separate GTP-U tunnel and forwards it to EPC. GTP-U packets are further encapsulated into a UDP/IP header by the eNodeB since commodity devices (e.g., routers) between EPC and RANs do not process this protocol. Moreover, an eNodeB maintains a persistent SCTP (Stream Control Transmission Protocol) connection to MME. Over this connection, it acts as a proxy and multiplexes different UEs' signaling or NAS (Non-Access Stratum) messages onto a single 4G S1AP (S1 Application Protocol) session and transports them to MME.

Each UE's NAS signaling and data traffic is processed into a different UE container in SoftBox. To properly forward different UEs' traffic to their corresponding UE containers, LTE RANs must communicate with the SoftBox control plane (UE controllers and global controller) and exchange different control messages and events. Since modifying the network stack of eNodeBs for implementing new protocols is impractical, we propose a minimally disruptive design to facilitate communication of SoftBox with LTE RANs: We instruct our agents at eNodeBs (introduced in Section 3.3.3) to implement a *translation layer* and map the

EPC protocols to custom SoftBox protocols and vice versa. At a high level, the translation layer performs two operations (Figure 3.4). First, since some of the SDN switches (e.g., OpenFlow switches) do not support match+action rules on GTP-U packets and GTP-U is a complex protocol, it replaces each UE's GTP-U tunnel with an MPLS tunnel. In Section 3.5, for minimizing the SDN rules and updates, we will propose a highly scalable SoftBox protocol over MPLS. Second, our translation layer reverses the eNodeB's multiplexing function, which puts different UE's signaling traffic onto a single S1AP/SCTP connection with MME. For each UE connected to the eNodeB, it establishes a separate transport connection (e.g., TCP, reliable UDP (RUDP)) with the corresponding UE controller, and properly forwards the UE's signaling traffic to it over our custom application layer protocol (will be elaborated in Section 3.5).

3.3.5 Putting all together: Orchestration of UE containers

SoftBox dynamically and quickly provisions UE containers over the network by continuously and effectively interacting with LTE RANs. When a switched-on UE sends an attach request, we instruct the eNodeB agent to send a UE container creation request to the global controller that performs three operations: (1) fetches the UE's profile (*e.g.,* plan, type) from the subscribers (Figure 3.4-a), (2) instantiates a customized UE container for the UE in a DC through its DC agent, and (3) programs the eNodeB and SDN switches through its agents to direct the UE's traffic to the UE container. Next, the eNodeB establishes a session with the in-container UE controller and starts forwarding the UE's signaling traffic to it. In the meantime, the UE controller installs rules into the in-container software switch to enforce different NFs on the UE's data traffic. When the UE moves, other procedures happen in SoftBox (*e.g.,* updating data plane tunnels, container migration) that are discussed in the next four sections. For scalability, the global controller communicates with UE containers through its DC-level agents based on a publish-subscribe model. Through agents, it subscribes to certain events in containers (e.g., low QoE) or reconfigures them with new policies in runtime. When a UE turns off, the eNodeB agent notifies the global controller who fetches the UE states from

the container, backs them up in the subscribers database, and destroys the container.

3.3.6 Optimized SoftBox: Design & Optimization Challenges

The above basic version of SoftBox meets our design goals (Section 3.8). To make SoftBox more effective and efficient for deployments at scale, we further optimize it along four dimensions. Our optimization challenges and solutions to them are unique to and novel in the context of SoftBox. We provide an overview on the optimized SoftBox in the below and Table 3. For brevity, we delay detailed discussion of related work to later sections.

Challenge 1: Scalable optimization of idle UEs' containers. In cellular networks, UEs spend most of their time in the idle mode, so a large fraction of UE containers in SoftBox can be underutilized at any point in time. Each of them incurs small CPU/RAM overheads without receiving/sending any traffic from/to the UE. Adopting the traditional NFV approach [132] suggests that SoftBox's global controller must monitor, stop and resume UE containers in response to UEs' transitions between the idle and active modes. Unfortunately, this approach does not meet our unique design requirements. First, the strategy of stopping idle UEs' container does not work as some in-container NFs (*e.g.,* paging) must *always* run in the core. Second, the global optimization is not scalable with many UEs continuously changing their state in the network.

Solution: Self-optimizing UE container design (Section 3.4). To optimize idle UEs' containers, we propose a *self-optimizing* UE container design. Using the latest capabilities in Linux, we develop a control logic inside each UE container that monitors the UE's connection state and quickly minimizes the container resource usage when the UE becomes idle. Our distributed in-container approach is flexible in customizing the optimization in each UE container depending on its set of always-on NFs and the UE's state changes pattern. In addition, it is more scalable and faster than the centralized optimization method that is expected to place the burden of optimization on SoftBox's global controller.

Challenge 2: Minimizing the SDN fabric rules and updates. SoftBox steers traffic of UEs through their respective containers for processing. Given there are millions of UE containers in the network, scalable traffic steering is challenging. On the one hand, encapsulating each UE's flows into a separate tunnel, similar to EPC, leads to huge forwarding states in SDN switches equipped with very small flow tables. Also, the mobility of UEs necessitates frequently updating the per-UE tunnels, degrading the controller throughput and traffic performance. On the other hand, end-to-end traffic aggregation (*e.g.,* between each eNodeB and gateways) does not work as each UE's traffic must be processed by a separate middlebox (container).

Solution: Enhanced segment routing scheme (Section 3.5). To perform scalable traffic steering, we design a segment routing (SR) [43]-based scheme for SoftBox. By itself, SR is only a source routing mechanism on path segments, but not a ready solution for our problem, *e.g.,* SR does not specify where and how path segments must be established, so it does not necessarily reduce the number and update rate of flow rules in our SDN data plane. The key insight in our SR-based scheme is to recursively derive a minimal set of path segments (by forming a novel abstraction over the SoftBox's SDN data plane), pre-establish the segments in the data plane permanently, reuse them for steering different UEs' traffic as much as possible, and carefully perform SR-based source routing at RANs to relieve our SDN switches of the task of switching UEs' traffic between path segments.

Challenge 3: Minimizing UE container migrations costs. SoftBox initially places the container of each UE in its proximity to realize ultra-low latency. Since UEs are mobile, their latency to their container can increase so SoftBox needs to migrate each of such UE containers to a DC that is closer to the corresponding UE. Cloud operators can often plan VM migration in advance, and only a small fraction of their VMs need migration [95]. In contrast, UE container migrations occur in real time and at a large scale as UE mobility

is the norm and unplanned in cellular networks. Such migrations pose two issues: incur a control load on SoftBox's global controller, thus needing us to scale it up, and result in an imbalance of load distribution on mini DCs, thus forcing us to over-provision their capacity.

Solution: Distributed mobility-aware migration scheme (Section 3.6). SoftBox introduces two online algorithms that leverage UEs' mobility pattern to simultaneously minimize the two container migrations overheads. For scalability and performance, SoftBox offloads most of the logic and mechanisms of these algorithms onto UE containers rather than centralizing them on its global controller: each UE container independently captures and processes the UE's mobility traces over time, determines the timing of its migration, and finally notifies the global controller. The controller makes a real-time decision on the destination mini DC based on the inputs from the container and its global network view.

Challenge 4: Scalable signaling sessions between the SoftBox core and LTE RANs. Today's protocols between LTE RANs and EPC are designed with an assumption that a single fixed node in EPC (i.e., MME) exchanges signaling traffic with UEs through eNodeBs (Figure 3.2). Thus, SoftBox's approach of processing each UE's signaling traffic on a UE controller (Figure 3.4) poses two challenges. First, it rapidly increases the number of connections from the core to RANs, which becomes unmanageable as the network grows. Second, it needs LTE RANs to be able to determine the network location of UE containers corresponding to arbitrary UEs on the fly as they are dynamically created and can migrate between different DCs. This is challenging because this functionality is not available in existing LTE RANs and deploying a central off-path registry and discovery service [114] is not scalable or efficient.

Solution: Scalable and fast transport and discovery protocols (Section 3.7). To overcome the challenges with signaling sessions between the SoftBox and LTE RANs, we depart from connection-oriented transport protocol that runs between today's RANs and EPC (S1AP over SCTP in Figure 3.2). We replace it with a reliable

lightweight connectionless transport protocol (our SoftBox protocol over Reliable UDP (RUDP) in Figure 3.4). This simple change enables an arbitrary number of UE controllers with dynamic network locations to exchange signaling traffic with their mobile UE through different eNodeBs. In addition, we design a fast service discovery protocol between SoftBox and LTE RANs, enabling eNodeBs to locate the UE container mapping to an arbitrary UE in zero-round trip time (0-RTT) as opposed to the central off-path approach (discussed in the above) incurring high delays. Our protocol is novel because it effectively uses existing LTE signaling messages that mobile UEs exchange with the network.

Roadmap. Next, we go into the details of each challenge to realize the optimized version of SoftBox.

Table 3: Summary of design decisions in the optimized version of SoftBox.

Minimization of	Motivation	Summary of Technical Solutions
Resource usage of idle UE containers	*A large fraction of UEs are idle most of the time*	*Developed a self-optimizing UE container design* • *Faster than the centralized optimization method on the global controller* • *Seamlessly operates by using advanced Linux technologies*
SDN fabric rules and updates	*Small flow tables in SDN switches and Performance disruptions in flow rule updates*	*Designed a scalable segment routing (SR)-based scheme* • *Recursively derive and pre-establish path segments in the data plane* • *Reuse them for steering different UEs' traffic*
UE container migrations costs	*Increased latency between mobile UEs and UE containers*	*Built algorithms and mechanisms leveraging UEs' mobility patterns* • *Minimize control loads and DC load imbalances caused by migrations* • *Scalablely measure latencies between UEs and UE containers*
Signaling communication costs with RANs	*Increased number of connections from SoftBox (UE containers) to LTE RANs* *Dynamic network locations of UE containers*	*Realized scalable and deployable protocols between SoftBox and LTE RANs* •*Replace legacy S1AP/SCTP transport protocol with SoftBox/RUDP* • *Enable LTE RANs to determine the network location of an arbitrary UE container*

3.4 Scalable and Flexible Optimization of Idle UE Containers

Optimizing resource usage of UE containers on servers is crucial to minimize power consumption of mini DCs in SoftBox. Each UE container runs some NFs, each spawning multiple threads to process the UE's traffic while building some UE-specific states in memory. Thus, a UE container uses both RAM and CPU resources. Storing the states typically requires an insignificant amount of RAM (Section 3.8.3). Therefore, we incorporate unique characteristics of cellular networks to minimize CPU usage of UE containers. In LTE networks, UEs

switch between the idle and active modes to save battery power, while being idle most of the time. Although an idle UE does not send or receive any traffic through LTE RANs, its corresponding UE container still has a small CPU overhead on its server in the SoftBox core. The overhead comes from the fact that in-container NFs actively access their ports and queues via polling and manipulate their threads and internal states, even if they process no traffic. Such tiny CPU overheads quickly add up since there are many UE containers in the network.

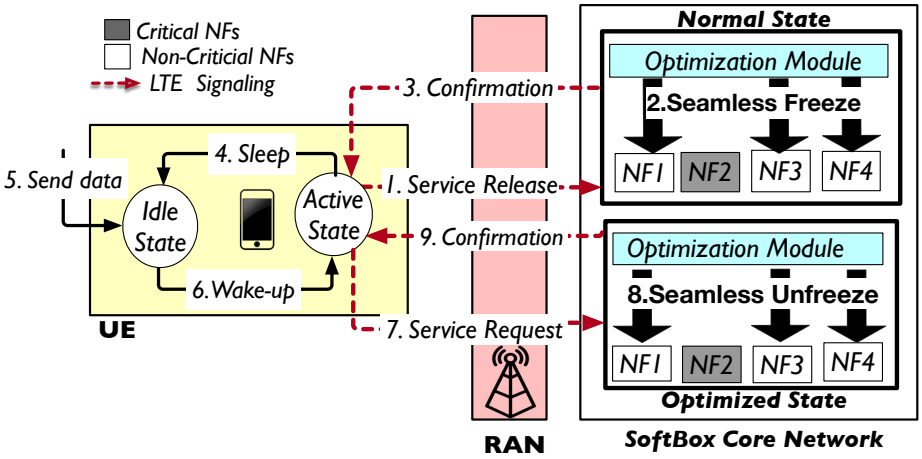


Figure 3.5: Our scalable and flexible optimization of idle UEs' container

A naive approach: centralized optimization. We believe there is a good opportunity to optimize such UE containers. One might think we should borrow the global optimization approach adopted in many existing NFV platforms (*e.g.,* Picocenter [132]): have our global controller (Figure 3.4) continuously track each UE container, gracefully terminate UE containers corresponding to idle UEs, and instantiate new UE containers for those UEs once returning to the active mode. Although these procedures can be easily implemented in SoftBox by sending asynchronous notifications on top the POSIX API [68] from the global controller to UE containers, such global optimization approach does not meet our design requirements. First, stopping idle UEs' container ceases all in-container NFs while some of them must always run, even though they are in the idle mode. For example, MobilityApp refactoring MME in UE containers (Figure 3.4) needs to locate UEs particularly when they are idle through the paging procedure. In theory, the set of "always-on" NFs can be different for each UE. Second, it is not scalable to have our global controller monitor millions of UE

containers and stop/resume them in large-scale networks.

Our approach: self-optimizing UE container design. To overcome the above shortcomings, we propose to carefully design UE containers in SoftBox to optimize themselves *autonomously* when their UE becomes idle. In other words, we distribute the logic and mechanisms of resource usage optimization into UE containers. Our design of self-optimizing UE containers has two properties: (i) scalable and fast as it does not involve our global controller in the optimization loop, and (ii) flexible as it allows us to customize the implementation of optimization in each UE container depending on its potentially unique set of *always-on* NFs. In more detail, we enhance our basic UE containers by developing a process management module based on Supervisor [44] in them, which uses advanced capabilities in Linux to do the optimization task. Our module listens to the UE state changes between idle and active (see Figure 3.5). When the UE requests the UE controller to switch to the idle mode, it quickly freezes all noncritical NFs (*e.g.,* DPI) in the UE container and keeps only critical ones (*e.g.,* MobilityApp) running (**Steps 1-4**). When the UE returns to the active mode and connects to the network, it instantly resumes the frozen NFs (**Steps 5-9**). To *seamlessly* freeze/unfeeze the selected NFs, we leverage cgroupfreezer API [46] rather than the POSIX API inside UE containers. The cgroupfreezer library uses the Linux kernel freezer code to prevent the freeze/unfreeze cycle from becoming visible to the NFs being frozen so they can keep their memory states. The POSIX signals are observable within the NFs so their threads may select how to respond to them (*e.g.,* block) that can cause them to break.

While our optimization method is not complex, it is novel and effective in the context of SoftBox networks. On the one hand, it meets our unique design requirements. On the other hand, it significantly cuts the peak CPU usage of UE containers as a large fraction of UEs are always idle (3.8).

3.5 Traffic Steering With Minimal and Stable Forwarding Rules

In SoftBox, the global controller must program tunnels in the SDN fabric to properly steer each mobile UE's traffic through the corresponding UE container that can migrate among mini DCs (Figure 3.4). Existing SDN switches have limited flow table entries and updating them frequently (when UE and UE containers move) can cause traffic forwarding disruptions (*e.g.,* packet drops). To cope with these limitations, our key insight is to *pre-compute* and *pre-establish* a minimal set of *permanent* tunnel segments into the data plane and *reuse* them for steering different UEs' traffic as much as possible. This approach naturally reduces the numbers and update rates of flow rules in our SDN switches. We already discussed shortcomings of other design options (*e.g.,* establishing separate tunnel for each UE) in Section 3.3.6, and thus focus on realizing our novel traffic steering scheme in four steps in this section. Our scheme can be implemented on top of existing SR (segment routing) mechanisms [43]. At a high level, we have SoftBox's global controller: (**Steps 1-2:**) recursively break down the problem of finding a minimal and reusable set of permanent tunnel segments (endpoints and paths) into smaller subproblems by forming our novel abstraction, called "recursive middleboxes", over SoftBox' SDN switches. (**Step 3:**) then leverage source routing mechanism at RAN to relieve our SDN switches of the task of switching UEs' traffic between different tunnels (**Step 4:**) finally offload a part of traffic steering task onto UE containers for scalability.

Step 1. Forming the recursive middleboxes abstraction. We first explain how the global controller forms the "recursive middleboxes" abstraction to break down our large-scale traffic steering problem. The controller views each DC, rack, host, and container in SoftBox's data plane as an abstract middlebox. These middleboxes are recursively nested into each other, where mini DCs and UE containers are the outermost and innermost abstract middleboxes respectively. Similar to hardware middleboxes, each abstract middlebox has two logical ports to separate its ingress and egress traffic. Conceptually, each abstract middlebox

embeds a group of SDN switches in the network. Figure 3.6 depicts child middleboxes recursively embedded into a DC middlebox.

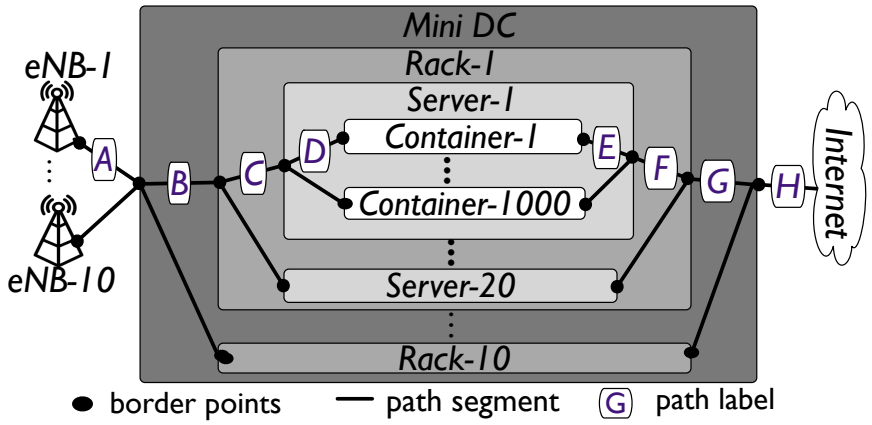


Figure 3.6: Our "recursive middleboxes" abstraction to scalably steer UEs' traffic through containers

Step 2. Recursive tunnel segment computation. Over our abstraction, the global controller recursively pre-computes and pre-establishes a minimal set of permanent tunnel segments in the data plane and later reuses them for different UEs. We now describe our tunnel segment computation for UEs' uplink traffic to the Internet and shortly discuss other traffic types. Our tunnel segment computation procedure is as follows: (i) the controller first addresses a small problem of steering aggregate uplink traffic of each eNodeB through DC middleboxes: it sets up a label-based tunnel segment between each eNodeB and its nearby DC middleboxes (*e.g.,* tunnel `A` in Figure 3.6), and a segment between each DC middlebox and close Internet gateways (*e.g.,* tunnel `H`). The Internet gateways and DCs can be determined based on operators' requirements for closeness of UE containers to UEs (Section 3.6), (ii) moving inside each DC middlebox, the controller solves the problem of steering the DC's aggregate ingress traffic through different rack middleboxes and exiting it from the DC. In this case, it sets up a segment between the DC ingress point and each rack middlebox (*e.g.,* tunnel `B`) and a segment between each rack middlebox and the DC egress point (*e.g.,* tunnel `G`), and (iii) recursively, the same procedure continues in each rack and then in each server. Conceptually, the controller sets up a segment between the ingress port of each child middlebox and that of its parent (*e.g.,* tunnels `B,C,D`), and one between the

egress port of each child middlebox and that of its parent (*e.g.,* tunnels `E,F,G`). On servers, the controller provisions tunnel segments for the maximum expected UE containers. When it dynamically creates UE containers on servers, it reuses pre-established tunnel segments (*e.g.,* tunnels `D,E`) and only connects UE containers to them. Except UE specific tunnels on servers (*e.g.,* tunnels `D,E`), the rest of tunnels in the network will carry aggregate traffic of different UEs as follows.

Step 3. Source routing on abstract middleboxes Next, we explain how the global controller steers each UE's uplink traffic through its UE container. To minimize the number of rules needed in SDN switches for tunnel switching, it instructs agents in eNodeBs (see Figure 3.4) to perform SR's source routing. Assume a UE has sent a packet to an eNodeB (*e.g.,* eNB-1 in Figure 3.6). To redirect the packet through the nested middleboxes containing the corresponding UE container (*e.g.,* Container-1), our eNodeB agent encodes a stack of labels into the packet (the stack can be compressed using P4 [60, 97]). The top half of the stack consists of the labels of segments from the eNodeB down to the UE container (*e.g.,* `[A,B,C,D]`). The bottom half of them are labels from the UE container up to the Internet egress point (*e.g.,* `[E,F,G,H]`). After encoding the stack, the eNodeB agent sends out the packet. Our data plane switches always forward the packet based on the outermost label sitting on the top of the stack (TOS). When the packet arrives at the ingress port of each abstract middlebox (*i.e.,* DC ingress switch, TOR switch, server NIC/soft switch), it pops the TOS label from the packet. Then, the packet is directed to the proper inner middlebox based on the new TOS label. This continues until the packet reaches an ingress port of the UE container. After the UE container completes its processing, the remaining lower half of the stack is used to forward the packet from the container to the Internet gateway.

Step 4. Label stack distribution and handling other traffic types. First, our source routing requires proper distribution of label stacks to eNodeB agents in the network. When

a UE sends LTE attach request through an eNodeB, the global controller creates the UE container and instructs its eNodeB agent with the proper label stack (Section 3.3.3). For scalability, the global controller preloads each UE container with the set of label stacks in the network and instructs it to proactively inform different eNodeB agents with proper label stacks when its UE moves or it migrates among mini DCs. Second, our traffic steering scheme handles downlink and UE-to-UE traffic. For downlink traffic from the Internet to UEs, the controller instructs Internet ingress switches to do source routing similar to eNodeBs. For scalable classification of UEs' downlink traffic needed pushing proper label stacks onto each UE's flows at the ingress switches, the controller uses the approach described in SoftCell [86]. For UE-to-UE traffic, it pre-installs direct tunnel segments among immediate child middleboxes of a middlebox (*e.g.,* racks inside the DC in Figure 3.6) to steer such traffic through multiple UE containers

We will show the above traffic steering scheme significantly reduces SDN rules and updates in large-scale SoftBox (Section 3.8.3).

3.6 Scalable & Mobility-Aware UE Container Migration Scheme

In SoftBox, the UE container placement is crucial to build ultra-low latency services on a per-UE basis. Since UEs are mobile, the latency between them and their container increases so SoftBox's global controller needs to transfer the UE containers between DCs. There are diverse tools for seamless container migrations [30, 132]. Thus, we focus on two challenges that are unique to SoftBox. Container migrations can (i) incur a large control load to the global controller, and (ii) lead to an imbalanced distribution of containers among mini DCs. Traditional VM/container migration schemes do not meet our performance and scalability requirements (Section 3.3.6, Section 3.9) since our environment deals with unplanned UE mobility, strict deadlines, and many UE containers. Thus, we develop a container migration scheme that is *novel in two aspects*: it consists of distributed and scalable components

to schedule containers needing migration, and leverages UE mobility patterns to choose migration destinations efficiently.

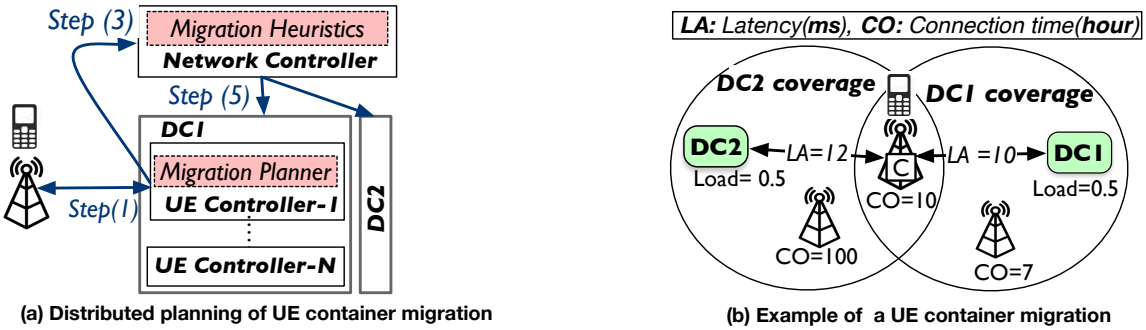


Figure 3.7: Our container migration scheme with the distributed planning & mobility-aware heuristics

3.6.1 Distributed Planning of UE Container Migrations

SoftBox allows operators to flexibly determine the maximum latency between each UE and its container (*e.g.,* 5ms). Assuming UE containers are already placed inside DCs, a migration event occurs when the UE-UE container latency becomes more than the *migration threshold*. Since there are many containers in the network, our design principle is to reduce the role of the global controller in migrations. It is not scalable to have the controller (i) continuously track the latency between millions of UEs and containers to determine which containers must be migrated, and (ii) collect and process fine-grained mobility patterns of many UEs (*e.g.,* handover and connection history) to be used in its migration decisions. To address the issues, we distribute the load of scheduling migrations and collecting mobility history of UEs on containers in five steps. We design the UE controller's MobilityApp in each container to: (**Step 1**.) measure its latency to the UE autonomously and continuously (see Fig 3.7-a), (**Step 2**.) record and prune the mobility information of the UE, and (**Step 3**.) make a decision locally regarding its migration timing and issue a migration event to the global controller when it is necessary. Upon being notified, the global controller only: (**Step 4.**) runs our migration algorithms (in Section 3.6.2) to select a mini DC for the container based on its global view of DC capacities and the mobility pattern sent by the UE controller, and (**Step 5**.) coordinates the old and new servers to handle the migration (Section 3.8.3).

3.6.2 Mobility-Aware Heuristics for UE Container Migrations

In response to each migration event, the global controller selects a mini DC from the pool of eligible DCs satisfying the migration threshold; it also tries to achieve two goals in its real-time decision-making process (migration events are not known a priori thus offline optimal algorithms cannot be used). First, it minimizes the total number of needed migrations over time to reduce the load on itself. Second, it balances the load of containers among DCs (or minimizes the max load of DCs in terms of the number of containers). In each migration event, our heuristic is to have the controller migrate the UE container to a DC from the eligible DCs pool, that maximizes the normalized term $ContainerDurability(DC) + AvailableCapacity(DC)$. The durability function computes how long a DC can host the UE container without forcing it to issue a migration event. The capacity function returns the remaining capacity of the input DC in terms of the number of containers. Intuitively, continuous sum of the product of these two normalized functions in migration events reduces the needed container migrations and DC load imbalances. The capacity function definition is easy but the durability one can be realized in multiple ways. In this chapter, we suggest two algorithms that both use the same capacity function but each having a different durability function. To simplify the description of the algorithms, we use Figure 3.7-b where the attachment of a UE to eNodeB C causes its container to issue a migration event. The problem is the controller needs to migrate the container either to DC1 or DC2 to meet the latency requirement.

- **Our Least-loaded-proximity (LLP)** algorithm assumes the chance of next migrations reduces if the UE container is moved closer to the UE. Thus, it has the durability function return reciprocal of the normalized latency between an input DC and the UE. In the example, LLP picks DC1 because it is closer to the UE and the two DCs are equally loaded.

- **Our Least-loaded-mobility (LLM)** algorithm assumes the mobility/connection patterns of the UE in past time windows (*e.g.,* weeks) can determine the DC which can run

the container for the longest time. Thus, it computes the durability function output for each eligible DC as follows: it (i) first identifies eNodeBs that their latency to the DC is below the migration threshold, and (ii) normalizes and returns the total connection time of the UE to those eNodeBs in the past time window as the function output. In the example, LLM chooses DC2 as the UE has spent most of its time in the past with the eNodeBs in the range of DC2 and the DCs are equally loaded.

Our distributed online container migration scheme is not complex, but it effectively cuts peak loads on our global controller and DCs, has a high performance in diverse settings compared to the optimal offline solutions, and meets our design constraints (Section 3.8).

3.7 Scalable Interaction of SoftBox Core and LTE RAN

For performance reasons, we distributed the MME on UE controllers inside UE containers. The protocols carrying UEs' signaling traffic between EPC MME and LTE eNodeBs are not scalable and sufficient for the SoftBox core containing millions of UE controllers. They were originally designed for an environment where a centralized fixed MME handles all UEs' signaling traffic. Our UE containers are in a far larger quantity and migrate between different DCs so their network location continuously changes. To handle the signaling traffic between SoftBox (UE controllers) and LTE RANs, we (1) design a fast and mobility-aware service discovery protocol that enables the RANs to locate the UE containers and (2) carry the signaling traffic over a more scalable transport protocol.

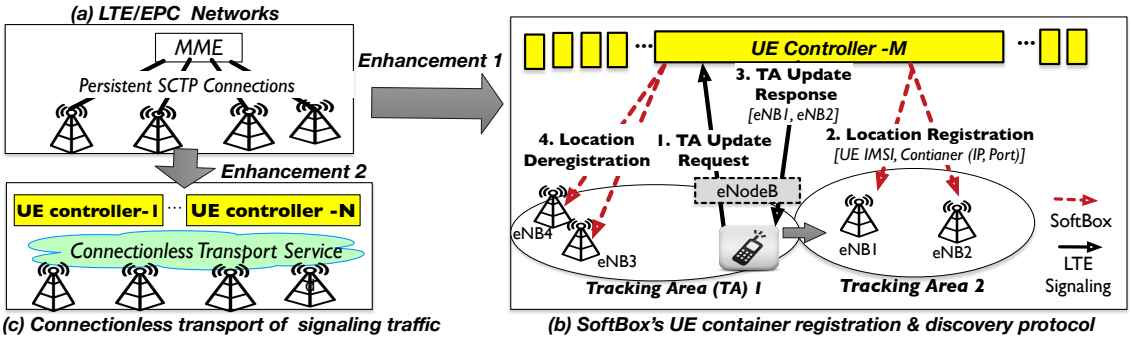


Figure 3.8: Connectionless per-UE mobility management equipped with mobility-aware service discovery protocol.

3.7.1 Fast and Mobility-Aware UE Container Discovery

In a 4G RAN+EPC network, the MME node often is deployed at a fixed and static network location (IP address and port). MME acts as a server and eNodeBs as clients (see Figure3.8-a). Each eNodeB reads the network location of MME from a configuration file and then permanently connects to it. In SoftBox, UE controllers in UE containers replace MME. Unlike MME, both their numbers and network locations dynamically changes as a result of UE mobility events causing UE container migration, creation, and termination procedures. Clearly, the approach of storing configuration files in eNodeBs no longer is effective or efficient as it would require tracking the location changes and frequently updating the configuration file in every eNodeB in the network. Therefore, we propose that the we enable eNodeBs to dynamically discover the latest location of UE controllers/containers corresponding to UEs. This approach makes the network management easier compared to the the traditional config file approach. Using the latest locations, the eNodeBs can establish connections with UE controllers to properly forward UEs' signaling traffic to them. One generic model to realize this discovery is to place a global service registry in SoftBox networks (*e.g.,* [114]), register and update the location of each UE controller with it and have different eNodeBs query the registry service. The clear drawbacks of this centralized model is (1) scalability as the global registry service can become a hotspot and (2) increased delay in accessing the LTE network as the global service can be far from eNodeBs and mobile UEs must wait for the discovery process to complete. Rather than relying on such generic mechanisms with their associated shortcomings, we believe a better approach to the discovery process is to incorporate characteristics of mobility protocols in cellular networks.

We design a *fast and mobility-aware UE container registry and discovery service*. Rather than having a global registry service, we place a local container registry service on each eNodeB (inside our agent/shim described in Section 3.3.3-5). Each eNodeB uses its local registry to quickly resolve its queries in zero-round trip time (0-RTT). We design the UE controllers to dynamically and proactively register/deregister themselves with the registry instances on

nearby eNodeBs before those eNodeBs need to discover them. To realize this, we build on a key aspect of mobility support in cellular networks. In LTE networks, MME registers each UE to a tracking area (TA). Each TA is a logical group of eNodeBs. At any time, MME limits the mobility of a UE to its registered TA to ensure the broadcast paging procedure and its associated radio wake-ups will not affect all UEs. This limitation has two consequences. First, the UE runs a timer whose periodic expiration causes the UE to report back and confirm its current TA to MME. Second, when a UE wants to exit its current TA, it must explicitly request MME to assign it to a different TA. Based on this concept, our UE container registry protocol works as follows (see Figure3.8-b): ***(Step 1.)*** When a UE controller receives a TA update request from the UE intending to exit its current TA, it computes a new TA for the UE (*e.g.,* TA2 in the example). ***(Step 2.)*** Then, it registers its location (*e.g.,* [192.168.4.82, 2153]) with the registries on eNodeBs in the new TA (*e.g.,* eNB1, eNB2). ***(Step 3.)*** Next, it sends a TA update to the UE consisting of the new eNodeBs information. ***(Step 4.)*** Finally, the UE controller deregisters itself from the registry instances on the eNodeBs in the old TA (*e.g.,* eNB3, eNB4).

3.7.2 Connectionless RAN-Core Signaling Traffic

In 4G/LTE networks, each eNodeB establishes a S1AP/SCTP connection with MME Then, MME and eNodeBs create logical NAS signaling sessions over these S1AP/SCTP connections for different UEs (detailed in 3.3.3). As a UE moves around, MME quickly migrate the UE's signaling session on the SCTP connections with different eNodeBs in the handover procedure. In SoftBox, because of distributing MME on UE controllers, multiplexing signaling sessions of different UEs into the same set of fixed S1AP/SCTP connections is no longer an option as each UE must communicate with a different end point (UE controller) in the SoftBox core. On the other hand, establishing dedicated connections for each UE controller with different eNodeBs rapidly escalates the core-RAN connections by a factor equal to number of UEs. Managing these connections is very costly both in terms of the network performance and the overhead on RANs. As UE containers migrate between different DCs and UEs move,

RANs needs to continuously set up and tear down many connections with the SoftBox core. To address these problems, we transport the signaling traffic between SoftBox (UE controllers) and RAN (eNodeBs) through our custom application layer protocol over a reliable connectionless transport protocol (see Figure 3.4 and Figure 3.8-c). This approach eliminates the handshake cost/complexity while allowing an arbitrary number of UE controllers and eNodeBs to have low-cost communication with the guaranteed-order packet delivery. Our current prototype employs Reliable UDP (RUDP) [39] as a lightweight connectionless protocol. It is worth mentioning that we continue using TCP between the global controller and other entities (*e.g.,* switches, eNodeBs) as these sessions are static and permanent.

3.8 Evaluation

Methodology. Using prototype and large-scale trace-driven evaluation, we first show that the basic (unoptimized) SoftBox architecture is more scalable and efficient than EPC. Our metrics are: (1) the number of CPU cores needed for a large-scale network, (2) the signaling overhead in the core, (3) the performance for the device-to-device (D2D) communication. For comparison, we use widely-used EPC systems, OpenEPC [34] and OAI EPC [33]. We carefully ensure that our comparisons with EPC are fair as our prototyped SoftBox has similar or better functionality compared to the EPC systems, uses the same packet processing technology. In addition, we study them under similar deployment conditions. We then demonstrate that the enhanced version of SoftBox equipped with our four schemes optimizing UE container migrations (Section 3.6), idle UEs' containers (Section 3.4), traffic steering through UE containers (Section 3.5), and UE container discovery (Section 3.7) is even more effective and efficient for large-scale deployments due to having lower data and control loads and higher performance. In the optimized SoftBox evaluation, we validate that each of our optimization techniques is efficient in diverse settings with hundreds of mini DCs and tiny UE container migration thresholds. Next, we describe the details of the SoftBox prototype and our LTE dataset followed by our evaluation results. In evaluating SoftBox, we used tens of servers with 16 CPU

*Table 4: EPC & SoftBox signaling overheads–**:common*

	EPC			SoftBox		
	Protocol	*Purpose*	*#Msg*	*Protocol*	*Purpose*	*#Msg*
Signaling	*NAS***	*Mobility/security management*	*8*	*NAS***	*Mobility/security management*	*8*
	S1AP/SCTP	*UE's signaling transport*	*22*	*SoftBox/RUDP*	*UE's signaling transport*	*2*
	Diameter	*Credit control Accounting Location update Authentication*	*94*	*Docker*	*Container creation & destroy*	*2*
				JSON	*Container configuration*	*2*
	GTP-C	*GTP tunnel setup*	*10*	*OpenFlow*	*SDN switch configuration*	*4*
DB	*SQL***	*UE state persist & read*	*385*	*SQL***	*UE state persist & read*	*30*

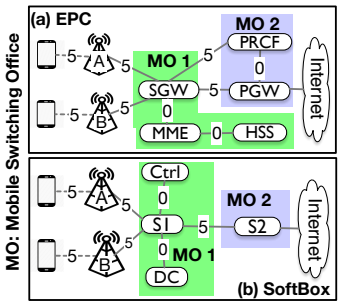


Figure 3.9: Setup in the D2D experiment

cores, 64GB RAM, and four 10GbE NICs on the Emulab-PhantomNet EPC/LTE testbed [37].

3.8.1 Prototype and LTE Dataset

Prototype. We developed a detailed prototype of SoftBox that can work with LTE eNodeBs equipped with our protocols. We emulate each mini DC using 10 OVS switches organized into a two-level leaf-spine topology and a variable number of UE containers. In our large-scale experiments, we interconnect mini DCs and simulated RANs based on a realistic flat topology containing 1K OVS switches [118]. Our global controller is implemented on top of RYU [41] with its two apps (Figure 3.4-a). The first one instantiates and migrates containers using Docker [12] and Flocker [30] respectively. The second one runs our recursive traffic steering scheme by configuring switches using OpenFlow. Inside DCs, our prototyped UE containers run a minimal Linux and fully realize our design (Figure 3.4-b). Inside individual UE containers, we (i) refactor the EPC data plane by building high-performance DPI, firewall, NAT, buffers and charging/QoS NFs using the Netfilter [32] and nDPI [31] libraries to process UEs' data traffic and (ii) refactor the EPC control plane through developing the UE controller with its three apps on top of RYU to process UEs' signaling traffic (MobilityApp is a modified MME [33]).

Large-scale LTE traces. We also collected LTE traces from 200 real UEs by deploying a mobile app on the PhoneLab testbed [38]. Over 2 months, we captured the UEs' radio and data messages with LTE networks (accessed their Qualcomm Diagnostic Interface) and their GPS locations. We then developed simple heuristics (*e.g.,* time-shifting, mixing) to synthesize

traces for 20M UEs based on the real ones. We built a RAN simulator to feed our prototype and EPC systems with the data. For 20M UEs, our final dataset has 30 billion idle-active state changes, 10B handovers, 40K cells, and UEs' real-time location. Since our dataset does not have the cells' location needed for our container migration study and public databases have little coverages for them, we used multiple clustering algorithms [70] to estimate the location of cells based on different UEs' GPS samples. For brevity, we do not explain them here.

Table 5: Average RTTs for the D2D traffic

	Propagation delay (ms)	Tunnel mgmt. delay (ms)	RTT (ms)
SoftBox	10	0.78	21.56
EPC	20	2.33	44.67

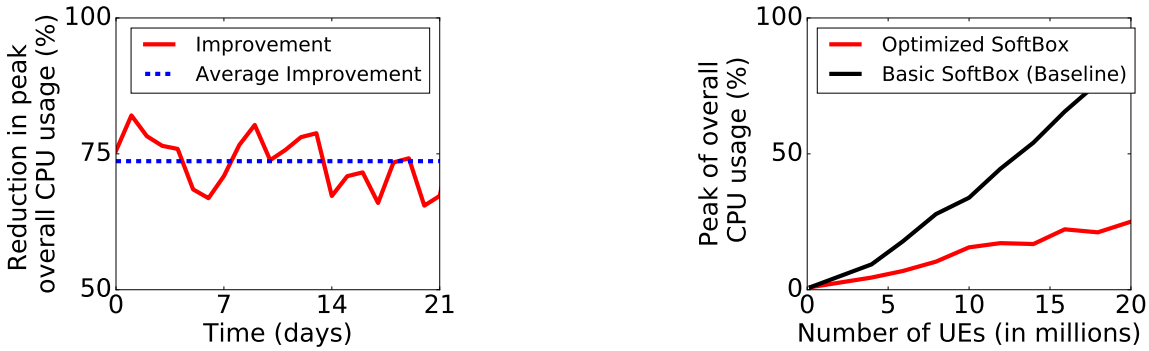


Figure 3.10: Effects of optimizing idle UEs' container

3.8.2 Evaluation of Basic SoftBox Architecture

We show that even the basic SoftBox is more scalable, flexible, and efficient than EPC. Additional optimizations for SoftBox as detailed in Section 3.4, Section 3.6, Section 3.5, and Section 3.7 are disabled in the following experiments.

8.2.1. Scalability w.r.t. CPU Cores. We measure the CPU cores needed by the EPC systems and our prototyped SoftBox (the global controller and UE containers) to handle a 20M UE network.

Finding 1: provisioning and maintenance costs of UE containers is minimal. We characterize the number of CPU cores that SoftBox need to run the UE controller for 20M UEs. measure the throughput of our global controller running on one server. We generate

synthetic attach requests from turned-off UEs and UE container migration events for registered mobile UEs. Our multi-threaded controller can simultaneously process at least 400K attach and 300K migration requests per second on the server. In attach events, it instantiates UE containers and similar to EPC, programs one tunnel per-UE in its data plane (as we do not show our traffic steering optimization discussed in Section 3.5). In migration events, it randomly picks an eligible sink mini DC and server (as we do not show our migration optimization in Section 3.6) and coordinates the old servers and new servers to do the container migration. As per prior large-scale studies [86], the peak numbers of simultaneous attach and handover events for 20M UEs are significantly below 400K and 300K (our dataset shows a similar pattern). Thus, we were able to provide low latency services for 20M UEs (with any migration latency threshold) by using 16 CPU cores for the global controller, which is a minimal cost.

Finding 2. Our UE containers are lightweight and SoftBox is more scalable than EPC systems. We now characterize the number of CPU cores that SoftBox needs for 20M UE containers. Each of our container images is light-weight and 40MB on disk. Each running container is assigned to a single core in a server. Each UE container has a high performance and can forward packets at 9-10Gbps when it is highly customized. By generating synthetic traffic based on rates captured from real UEs in our LTE dataset, we observe each container consumes at most 1.2% of the CPU core's processing capacity. Also, it occupies less than 0.5% of the total RAM on the server. As a result, we were able to simultaneously execute 80 UE containers on each core and around 1250 of them on each of the servers. We have observed that OpenEPC and OAI EPC support at most 200 and 150 simultaneous UEs respectively. Compared to them, SoftBox requires 6.2-8.3$\times$ fewer CPU cores to support any given number of UEs. For 20M UEs, we require 1.35M-1.88M fewer cores than these EPC systems. Vertical slicing of the core, eliminating most the EPC complex protocols, engineering lightweight UE containers are the main reasons for this efficiency.

8.2.2. Scalability w.r.t. Signaling Overhead. Using the same setup, we now show SoftBox is more scalable than EPC systems in terms of signaling overhead. We connect a simulated eNodeB and a UE to each of SoftBox, OpenEPC and OAI EPC. We measure the number of signaling messages exchanged in them for the UE when it attaches, downloads a 10MB file, and gracefully detaches. In this specific experiment, we do not study the handover operation due to the limitations of OpenEPC and OAI EPC but this only underestimates the capability of SoftBox in this regard.

Finding 3. SoftBox generates 86% fewer signaling messages in the core compared to EPC. Table 4 shows the breakdown of signaling messages exchanged in OpenEPC (OAI EPC is similar) and SoftBox. OpenEPC creates 134 signaling messages and issues 385 database queries to handle the attach/detach procedure. This high overhead is because the EPC architecture distributes the policies associated with the UE among different nodes, continuously synchronizing them using complex protocols. SoftBox cuts these overheads by consolidating the control and data functions for the UE into a container and eliminating EPC's complex protocols (*e.g.,* GTP-C/U, Diameter, SCTP). For backward compatibility purposes, SoftBox does not change the EPC's signaling sessions (NAS) with the UE so the NAS message type is common among SoftBox and OpenEPC. As shown in Table 4, SoftBox generates 86% and 92% fewer control and DB messages, respectively, for the UE than EPC. This is significant in large networks, *e.g.,* SoftBox produces 23.2M fewer signaling messages for 20M UEs.

8.2.3. Performance w.r.t. Data Plane Delay. The D2D traffic that is generated through communication between nearby devices (*e.g.,* autonomous cars) is expected to surge in 5G. Most of D2D applications need small end-to-end delays. We show that SoftBox is more efficient than EPC in supporting D2D low-latency communication due to consolidating network policies close to RANs. In a simple experiment, we have two UEs attached to

different eNodeBs and ping each other through a SoftBox and OpenEPC (Figure 3.9). We measure the data plane delay experienced by the UEs in terms of the propagation and tunnel management delays inside the two core networks. For a meaningful comparison, we assume an operator has two physical mobile switching offices (MOs) and the components of both networks are similarly distributed among them: we place OpenEPC's PCRF-PGW in office MO2 and its MME-SGW-HSS in office MO1 similar to real deployments [24]. SoftBox has an SDN switch, a mini DC, and a global controller in MO1, and another switch in MO2.

Finding 4. SoftBox has lower RTTs for D2D traffic than similarly deployed EPC. First, SoftBox incurs 50% less propagation delay for the D2D traffic than EPC in the above deployment since it creates two UE containers in the DC close to them and selects a more direct path between the UEs through them (Table 5). In contrast, EPC inefficiently routes the D2D traffic to the PGW for the policy enforcement. Second, EPC's distributed tunneling protocol (GTP-C/U in Figure 3.2) is very complex as explained in Section 3.3.2. Thus, we observe EPC has 2.9× higher processing delay compared to SoftBox.

3.8.3 Evaluation of Optimized SoftBox Architecture

In this part, we evaluate the optimized version of SoftBox and show that our four optimization techniques presented in Section 3.4, Section 3.5, Section 3.6, and Section 3.7 lead to substantial improvements of SoftBox's performance.

8.3.1. Benefits of Idle-mode Optimization. We first show that our optimization of idle UEs' container (in Section 3.4) further improves SoftBox's scalability. Recall that we placed a module *inside* UE containers, which listens to UE state changes. When the UE becomes idle, it freezes all the container NFs using `cgroupfreezer` (except MobilityApp that always run for the paging operation). When the UE becomes active, it unfreezes them quickly. Our optimized SoftBox is compared to the basic SoftBox as the baseline, which keeps UE containers and their internal NFs active regardless of the UE state. Note that stopping idle

UEs' containers did not meet our requirements. Based on the methodology described earlier, our RAN simulator feeds our prototype with LTE traces (UEs' signaling/data traffic).

Finding 5. Our idle-mode optimization cuts the peak of overall CPU usage of UE containers by 51-73%. Figure 3.10-Right shows that instantaneous peak of CPU usage of UE containers for a variable number of UEs in the optimized and basic SoftBox networks. We observe that the optimization lowers the global peak of usage computed over 3 weeks by almost 51-73% regardless of the number of attached UEs. The reason is that a small fraction of UEs is simultaneously in the active mode at any point in time, which could be true for any cellular network of any size. Figure 3.10-Left provides a daily view of the CPU usage optimization of 20M UE containers over 3 weeks, showing that our optimization reduces local (daily) peaks of CPU usage by 65-82%. The dramatic savings suggest the potential for dynamically packing more containers on underutilized CPU cores, and reducing the overall needed CPU cores. We leave this to future work.

8.3.2. Benefits of Migration Optimization. Using the same LTE traces, we show that our migration optimization of UE containers simultaneously and efficiently (i) reduces the number of migrations and (ii) balances the load of UE containers on DCs. In a broader sense, it enables us to provision fewer resources for the global controller and UE containers for a given number of UEs. Our design of distributed migration scheme, online migration algorithms (least-loaded-mobility (LLM), least-loaded-proximity (LLP)), and our implementation of them discussed in Section 3.6 and earlier in this section. To show the potential for migration optimization, we compare the LLM/LLP-equipped SoftBox with the unoptimized SoftBox (baseline) that chooses an eligible DC in migration events randomly. To evaluate the performance of our algorithms, we conduct *Empirical Competitive Analysis*: we measure the performance of the LLM/LLP algorithm *over* optimal offline algorithms. In our experiments, we change different parameters (*e.g.,* # DCs, # UEs, migration threshold).

Finding 6: Our optimization significantly cuts the container migration costs in low

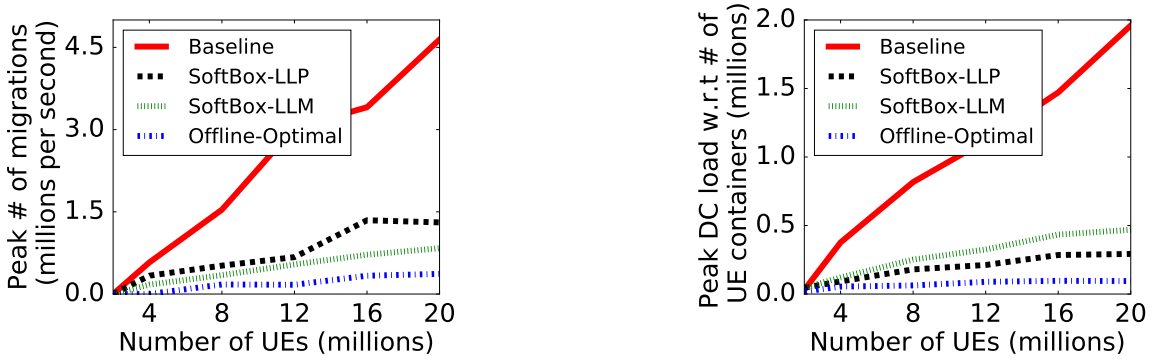


Figure 3.11: Effects of optimizing UE container migrations

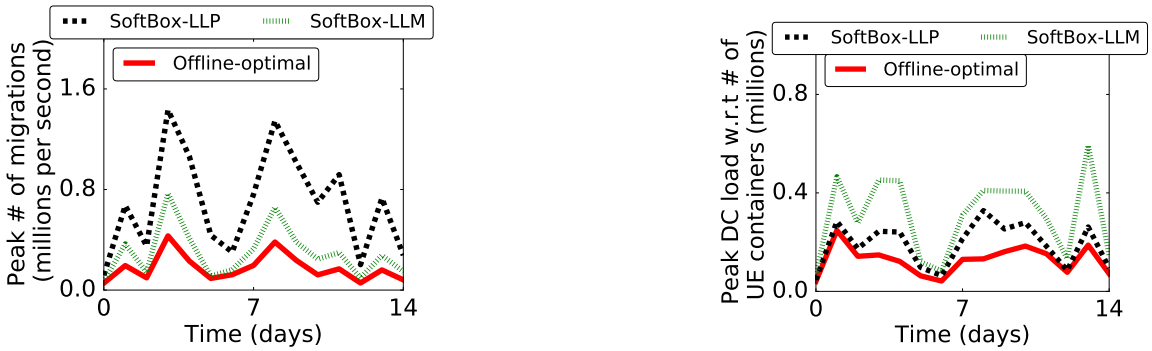


Figure 3.12: Efficiency of our migration algorithms

latency SoftBox networks. We study an extreme case where the goal is to provide ultra-low latency services for all UEs in the network. Thus, we set the migration threshold (max UE-UE container latency) to 0.02ms. This extremely low value puts SoftBox under significant pressure and thus better illustrates the implications of our optimization. Moreover, we evenly distribute 100 DCs among cells and vary the number of UEs. Figure 3.11 shows the peak number of migrations per second and the peak DC load for different systems over 2 weeks with respect to the UE count. Compared to the baseline, we observe the online LLP and LLM algorithms on average reduce the peak number of migrations by up to 66% and 77%. Also, LLP and LLM on average cause by about 78% and 71% better DC load balancing respectively. Figure 3.12 shows the daily (local) peaks of UE container migrations and DC loads. We observe our LLM/LLP algorithm is close to the optimal offline algorithm for each goal. LLM is more efficient than LLP in reducing migrations due to considering UEs' mobility pattern in the past time windows (set to 7 days) in its decisions (Figure 3.12-Left). LLP shows a higher performance in the DC load balancing than LLM (Figure 3.12-Right) as it causes more migrations and thus balances the DCs load more frequently.

Finding 7: Our online algorithms for the migration optimization are efficient and reasonably close to optimal offline algorithms. We show the efficiency of our algorithms is not limited to one case by conducting *empirical competitive analysis* (defined earlier). We repeat the above 2-week long experiment for every possible point in the space where the migration threshold varies from 0.005 to 10ms, and the number of DCs varies from 100 to 1K. In this large set of experiments, (i) LLM and LLP are at most 2.99× and 8.53× worse than the optimal offline migration optimizer, and (ii) LLM and LLP are at most 3.83× and 2.91× worse than the optimal offline DC load balancer. Overall, LLM is more efficient than LLP. *Note that LLM and LLP are multi-objective and online while the optimal algorithms are single-objective and offline, so we believe these are promising results.*

8.3.3. Benefit of Traffic Steering Optimization. We now validate if our enhanced segment routing scheme steering UEs traffic through UE containers substantially minimizes (i) flow rules in SoftBox's SDN data plane and (2) flow rule updates when UEs and UE containers move in the network. Our baseline is to establish a separate tunnel for each UE similar to EPC. Recall that our large-scale steering problem is unique to SoftBox and there is no other prior solution for it to the best of our knowledge. We continue using the previous setup with the container migration threshold of 0.02 ms and 100 DCs.

Finding 8: Our enhanced segment routing significantly reduces tunnels and flow rules in SoftBox. As the number of mobile UEs increases from 5M to 20M, Figure3.13-Left depicts the maximum number of flow rules across SDN switches in the network (When both downlink and uplink traffic are steered through UE containers). The max value stays below 1K for the optimized SoftBox since it aggregates different UEs' traffic into small set of pre-established tunnel segments (derived through our recursive middlebox abstraction) and employs source routing at RANs and Internet gateways to minimize rules needed for switching tunnels in the core. Meanwhile, the max value rapidly grows to 447K for the baseline due to its installation of two end-to-end tunnels per UE into the data plane. It is

Table 6: Effects of our container discovery optimization

Number of UEs (M)		5	10	15	20
Discovery Time (ms)	SoftBox	0.051	0.054	0.058	0.06
	Baseline	2.22	3.21	4.54	8.72

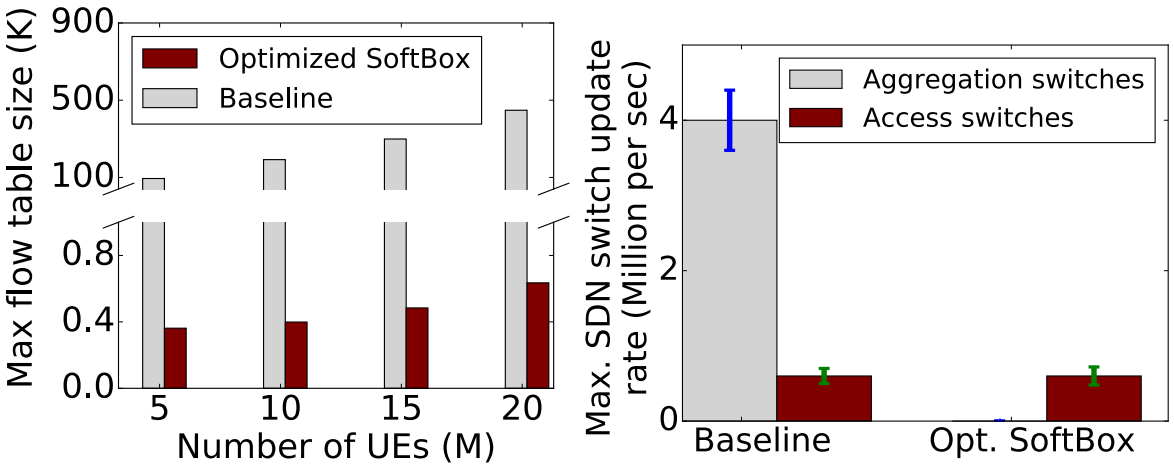


Figure 3.13: Effects of our traffic steering optimization.

worthwhile that the optimized SoftBox demonstrates on average 0.5% and 0.3% smaller packet drops and forwarding delays compared to the baseline as it precomputes and reuses minimal tunnels in the data plane.

Finding 9: Our enhanced segment routing minimizes the data plane reconfiguration rate in peak mobility. Because UE containers and UEs move, established tunnels for UEs must be modified to ensure correct traffic steering. For 20M mobile UEs, Figure3.13-Right depicts the flow rule modification rates in the optimized SoftBox and baseline. A smaller update rate means a lower risk of transient packet forwarding delays. The baseline issues by up to 4.7M rule updates per sec to the SDN switches in peak mobility hours. Due to establishing a separate tunnel for each UE, it is more sensitive to mobility events and modifies rules in many switches in response to them. A large fraction of the updates by the baseline maps to aggregation layer switches in the WAN and mini DCs. Updating this type of switches is riskier than edge layer software switches on servers. In contrast, because the optimized SoftBox reuses pre-established tunnel segments in the data plane, it does not need to update flow rules in its SDN switches. It only issues at most 700K per sec commands (not flow rules) to access layer switches on servers to connect UE containers to existing tunnel segments (see Section 3.5).

8.3.4. Benefits of UE container Discovery Optimization. In SoftBox, a UE container discovery request is triggered by an eNodeB when it starts serving a UE but does not know the network location of its corresponding UE controller. We designed a distributed mobility-aware protocol between eNodeBs and UE containers to optimize the performance and scalability of SoftBox in handling such discovery requests (Section 3.7.1). Our key idea was to locally resolve them by placing a minimal container registry service at eNodeBs and having UE controllers proactively and dynamically register with them as UEs move in the network. Here, we compare our protocol against the centralized approach as the baseline, where a global registry service co-located with our global controller handles discovery requests from RANs.

Finding 10: Our optimized mobility-aware container discovery protocol has near zero resolution time. For a variable number of mobile UEs from 5M to 20M in the network, Table 6 shows the average of UE container discovery times aggregated across different eNodeBs. We observe that our protocol offers near-zero UE container discovery times because it enables eNodeBs to resolve their queries locally. In contrast, the centralized approach has large discovery times of 2.2-8.72ms, increasing the control plane delay in SoftBox and degrading QoE experienced by mobile UEs. There are two reasons for this trend. First, the queuing delay at the central registry increases with respect to the number of UEs. Second, the propagation delay caused by round trips between RANs and the central registry significantly contributes to the discovery time.

3.9 Related Work

Software EPC systems. Many virtual EPC systems (*e.g.,* [108, 57]) port each of the EPC nodes to a VM to provision and scale the nodes based on time-varying traffic load. They offer almost no changes to the EPC architecture, build on EPC's inefficient policy management scheme, and thus inherit most of its weaknesses. Existing SDN EPC systems in the literature

(*e.g.,* [86, 99, 28]) are targeted on providing independent scaling of control and data planes and global routing of traffic in EPC. These systems decouple the EPC control and data planes, centralize its control plane (*i.e.,* MME, PCRF) on a logical SDN controller and disaggregate its data plane (*e.g.,* in S/PGW-Data) into single-function middleboxes (*e.g.,* DPI). The decoupling/disaggregation further scatters the EPC policies around the network and reproduces the known EPC issues in the SDN environment [79]. In a parallel work, PEPC [106] suggests a per-UE EPC-in-a-box design to increase the EPC packet forwarding rate. While there are some similarities between the SoftBox and PEPC proposals, differences between them in terms of architecture and functionality are major. SoftBox is a clean-slate architecture that completely redesigns and optimizes the core while PEPC is a different form of vEPC deployment. Unlike SoftBox, PEPC (i) does not provide mechanisms for realizing ultra-low delays in the core, (ii) its signaling overhead is similar to EPC due to running EPC's complex nodes (*e.g.,* HSS) and protocols (*e.g.,* Diameter, GTP-C, SCTP), and (iii) more importantly, lacks global SDN/NFV control over the core that SoftBox provides to identify and address four critical optimization problems.

Multi access edge computing (MEC) [104] is a conceptual proposal for deploying *general* cloud services close to users in cellular networks. ACACIA [64] is an MEC realization for the AR application. Unlike SoftBox, MEC is not a cellular core architecture and almost does not touch the EPC stack. CloudLet [113] and MobiScud [124] create per-user boxes similar to SoftBox but for a different problem. They view the core as a "blackbox" and accelerate UE apps by offloading their execution to clouds. In contrast, SoftBox is an architecture redesigning the core from ground up.

NFV research. Simple [107] uses OpenFlow to steer traffic through distributed middlebox chains. Such systems are designed for the traditional NFV. SoftBox consolidates each UE's NFs into a box, thus eliminating such complexities. Moreover, a wide range of packet processing platforms (*e.g.,* DPDK), high-performance NFV libraries (*e.g.,* ClickOS [93]) improve the efficiency of software packet processing. SoftBox can benefit from them. Our

prototype uses Netfilter while achieving a high throughput.

Container networking/migration. In the ISP context, CORD [48] leverages containers and virtualizes the single-point functionality on a "residential gateway" into a single container in the central office. SoftBox is conceptually different from CORD as it consolidates the distributed EPC into UE containers close to RANs and solves challenges specific to cellular networks (Section 3.1, Section 3.3.2). There are works in VM placement algorithms both for intra and inter DCs [95, 53]. These algorithms are not well-suited for our container migration problem dealing with unplanned UE mobility and orders of magnitude more boxes.

CHAPTER IV

SoftMoW: A Scalable and Reconfigurable 5G WAN Architecture

4.1 Introduction

In the previous two chapters, we focused on designing and optimizing citywide and statewide 5G core networks. In this chapter, we focus on the challenges associated with managing hyper-scale nationwide cellular wide area networks (WANs).

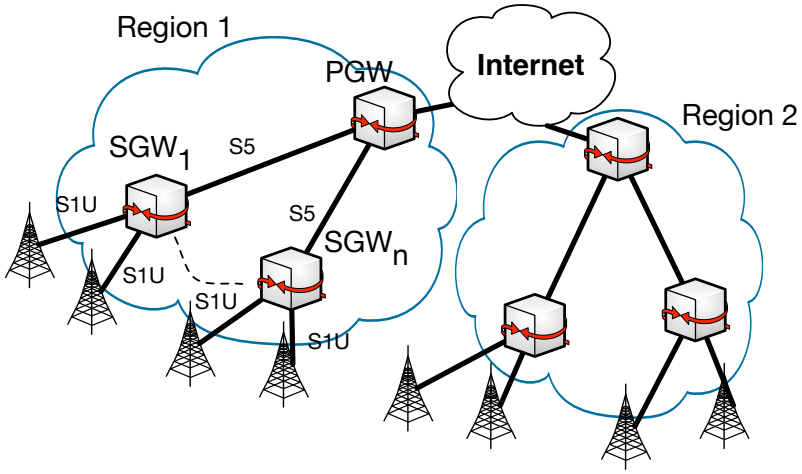


Figure 4.1: An LTE WAN with two regions

The current nationwide LTE WAN architecture is organized into very large and rigid regions (Figure 4.1). Each large region has a core network and a radio access network. The core network contains an Internet edge comprised of packet data network gateways (PGWs). The radio network consists of only base stations. In this architecture, there are

minimal control plane and data plane interactions among regions other than distributed interference management at radio access networks and limited coordination for mobility (e.g. no inter-PGW mobility [135]). All users' outgoing traffic must traverse a PGW and possibly go through the Internet.

This rigid WAN architecture is becoming harder and harder to support new trends of mobile traffic. First, mobile application performance is seriously impacted by the lack of Internet egress points per region. Specifically, as shown by a recent study [131], the lack of sufficiently close Internet egress points is a major cause of path inflation, suboptimal routing, and QoS degradation in large operators. Second, the continued exponential growth of mobile traffic puts tremendous pressure on the scalability of PGWs. Third, the fast growth of signaling traffic known as the signaling storm problem [29] poses a major challenge to the scalability of the control plane.

Rather than organizing mobile WANs as rigid regions with no direct traffic transit, we argue that cellular networks should have a seamlessly *inter-connected core network* with a logically centralized control plane. The inter-connected core network should consist of a fabric of simple core switches and a distributed set of middleboxes (software or hardware). The control plane directs traffic through efficient network paths that might cross region boundaries rather than exiting to the Internet directly from the origin region. The control plane should also globally support seamless UE mobility and optimize the performance of mobile traffic. For example, mobile traffic routing should be globally optimized; regions should be reconfigured to adapt to its workload.

Such an architecture raises unique challenges in scalability in comparison with data-center networks [76, 102] and inter-data center WANs [85, 83] since the cellular WAN has its own unique properties and challenges. First, the logically centralized control plane needs to control tens of thousands of switches and middleboxes, and hundreds of thousands of base stations in the data plane. A control plane with many controller instances in one data center cannot effectively handle the signaling load (*e.g.,* connection setups and handovers) from

hundreds of millions of subscribers distributed throughout a continent. Second, without global network states and a single controller exerting control, it will be hard to perform network wide routing optimization and inter-region handover minimization.

4.1.1 Summary of Contributions

To address these problems, we present SoftMoW, a scalable network-wide control plane that supports global optimization and control plane reconfiguration. SoftMoW makes the following contributions.

- First, SoftMoW *recursively* builds up the hierarchical control plane with novel abstractions consisting of both control plane and data plane entities. Key to our SoftMoW architecture is the controller. It is designed to be *modular* which consists of the network operating system (NOS), operator applications and the recursive abstraction application (RecA). NOS provides core services such as routing and path implementation. NOS does not handle cellular specific functions. Operator specific functions (e.g. mobility management) are implemented as applications on top of NOS. All recursive abstraction functions are implemented in RecA.

- Second, to enable scalable end-to-end path setup, SoftMoW presents a *novel label swapping mechanism* such that each controller only operates on its logical topology and each switch along a flow's path only sees at most one label. This new mechanism reduces the states in the switches.

- Third, SoftMoW designs new network-wide optimization functions such as optimal routing and region optimization to minimize inter-region handover.

- Fourth, we demonstrate that SoftMoW improves the performance, flexibility and scalability of cellular WAN using real LTE network traces with thousands of base stations and millions of subscribers. Our evaluation shows that path inflation and inter-region handovers can be reduced by up to 60% and 44% respectively.

4.2 SoftMoW Design Overview

SoftMoW's goal is to design a scalable cellular WAN architecture (both the control plane and data plane) to enable network-wide optimizations. We introduce the components in a SoftMoW network, the design challenges and our solutions.

4.2.1 SoftMoW Components

SoftMoW does not require expensive, inflexible and specialized devices (*e.g.,* PGWs and SGWs) that integrate control and data plane operations with middlebox functions. SoftMoW does not change the LTE protocols used in the user equipment (UE) and the protocols between UE and base stations. SoftMoW has the following high-level architectural components.

Nation-wide inter-connected core networks. SoftMoW distributes and inter-connects programmable switches nation-wide. The network in one region should have enough egress points through a subset of the switches. An egress point can connect to other regions of the same carrier, other carriers' mobile networks, Internet service providers or content providers at peering points to exchange traffic. This eliminates the internal path inflation problem caused by the lack of sufficiently close egress points and enhances end-to-end QoS metrics by offering better diversity of external paths.

Radio access networks. Radio access networks consist of base stations which are organized and inter-connected into base station group (BS group) with different topologies (*e.g.,* ring, mesh, and spoke-hub) to ensure intra-BS-group fast-path communications. BS groups are connected to core network switches locally. We assume each base station has an *access switch* performing fine-grained packet classifications on traffic from UEs.

Middleboxes and service policies. SoftMoW departs from the centralized policy enforcement at PGWs and utilizes middleboxes which can be flexibly placed throughout the cellular WAN. For scalability, middlebox functions will be mostly limited to edge networks of the cellular WAN. Middlebox instances can potentially implement any sophisticated network functions. The functions can be specific to application types (*e.g.,* noise cancellation func-

tion and video transcoding function) and operators (*e.g.,* charging and billing), and security (*e.g.,* firewall, and IDS). A service policy is then met by directing traffic through a *partially ordered set* (also known as poset) of middlebox types. Given the location and utilization of middlebox instances, the controller can implement a poset using various combinations of physical instances.

Controller. The controller enforces a rich set of service policies on subscribers' network access through new global network applications. These applications are based on a global view of the inter-connected core networks, which are not available in current LTE networks or recently proposed cellular architectures such as SoftCell [126]. Specifically, the controller sets up end-to-end optimal paths for aggregate flows and minimizes the number of inter region handovers.

4.2.2 Design Challenges and Solutions

Challenge 1: scalable control plane. The logically centralized control plane needs to control tens of thousands of switches and middleboxes, hundreds of thousands of base stations in the data plane. A control plane with many controller instances in one data center (*e.g.,* [87, 58]) will not effectively handle the signaling loads (*e.g.,* connection setups and handover events) from hundreds of millions of subscribers distributed throughout a continent. Also, a flat decentralized architecture where local controllers only communicate with their neighbors (*e.g.,* [123]) is not scalable enough to support fast and global optimizations. It requires distributed algorithms that involve many rounds of message exchanges.

Solution: recursively build up a hierarchical and reconfigurable control plane. SoftMoW hierarchically constructs a *network-wide control plane* that is *reconfigured* in response to the signaling loads and traffic patterns. The control plane consists of geographically distributed controllers that are organized into a tree structure. Recursively from the leaf level, each controller (except the root) exposes a small number of logical and reconfigurable data plane entities to its immediate parent. These entities aggregate many switches, middleboxes

and base stations. To enable global optimization such as routing optimization by ancestor controllers, the exposed logical switches and their interconnections are described as a *virtual fabric* with annotated bandwidth, latency, and hop count information.

Challenge 2: scalable end-to-end path implementation. Our cellular WAN provides connections between millions of UEs and thousands of Internet egress points, the number of routing states in the core network switches is tremendous. One way to implement the routes is to aggregate flows traversing the same path, assign them one label and route on labels (e.g. MPLS). In a decentralized flat control plane, implementing a label-switched path involves all controllers and switches on the path. To do this, each controller has to know the global state. Keeping entire data plane states consistent at each controller or storing them into a central data base is not scalable. In SoftMoW, each controller has a limited summarized view over a set of logical entities to improve scalability, but this makes the state management and path implementation more challenging.

Solution: scalable recursive label swapping. SoftMoW leverages its tree structured control plane architecture. Using a novel recursive label swapping approach, SoftMoW implements end-to-end paths while keeping per packet overhead minimal. An ancestor controller pushes labels onto packets of matching flows traversing its logical and reconfigurable switches. Recursively, these labels will be replaced with local labels by each lower level controllers. At the physical data plane switches, only a local label is pushed onto packets of matching flows, which each represents a local regional path segment. When packets leaving a region, the local label is popped off and an ancestor's label is pushed.

Challenge 3: scalable topology discovery and maintenance. Topology discovery is easy in flat multi-controller settings. Each switch is controlled by one controller instance. A controller sends discovery messages from all ports of registered switches. When a switch receives a discovery message, it forwards the message to the controller. The controller then maintains the link between the source and destination switches and stores link-specific information (*e.g.,* port name, link capacity). In SoftMoW, detecting links is more challeng-

ing because each cross-region link is visible to only one non-leaf controller; the non-leaf controller needs to discover it without breaking the abstraction.

Solution: recursive discovery protocol. We design and introduce a novel global discovery protocol allowing recursive discovery of topologies by each controller. Each leaf controller first discovers its own physical topology. Then the parent controller is exposed with a logical topology and can discover the cross-region links it controls. This process continues until the root controller discovers its topology. Controllers at the same level can perform discovery at the same time in parallel. The sequential process only applies to the bootstrapping phase. During normal operations, periodical discovery messages will be carried out concurrently.

Challenge 4: network-wide optimization. SoftMoW's goal is to enable global optimizations for control plane and data plane functions such as optimal routing and inter-region handover minimization. Maintaining and performing optimization with global network states for a country-wide network is not scalable.

Solution: design algorithms on abstract topologies of hierarchical controllers. SoftMoW supports global network optimization without a global network state at each controller. We demonstrate this feature using two important network functions. First, application traffic may have its own requirements on the path (e.g. low-latency path for delay-sensitive VoIP). In SoftMoW, the path is computed by controllers from the leaf to the root. If a local optimal path meeting the application requirements is found, it is used without further delegating to ancestor controllers. We show that the root controller is guaranteed to find an optimal path in terms of performance metrics (*e.g.,* latency and hop count). Second, an inter-region handover requires the involvement of an ancestor controller, the source controller and destination controller. In this procedure, new paths have to be implemented and in-flight packets have to be diverted to the target base station. To minimize control plane load, SoftMoW performs inter region handover optimization. The optimization is done from the root controller to leaf controllers. We show that the process converges if handover traffic pattern does not change

during the optimization.

4.3 SoftMoW Control Plane

We first give an overview of how we recursively construct the control plane and the logical data plane, and present the design of the controller architecture.

4.3.1 Recursive Constructions

As shown in Figure 4.2, SoftMoW hierarchically builds a reconfigurable network-wide control plane. The control plane consists of geographically distributed controllers that are organized into a tree structure, each controller associated with a level number and a globally unique ID. The topmost node is the *root* controller which can make coarse-grained decisions for the entire network, and level 1 nodes are *leaf* controllers close to the physical data plane. The number of levels, the number of children per node, and the geographical location of each node can be determined based on fine-grained latency budgets of control functions [25] as well as the density and size of the physical topology.

SoftMoW partitions the physical data plane network into logical regions whose borders can change over time based on traffic and failure patterns. Each leaf region is managed by a leaf controller. In Figure 4.2, leaf controllers (level-1) discover their physical switches and build the level-1 data plane, and they also abstract some entities for level-2 controllers. The level-2 controllers obtain logical network entities from the leaf controllers, discover their logical level-2 data plane and also make logical network entities. Finally, the root controller (level-3) obtains logical network entities from the level-2 controllers and builds the level-3 data plane. When building these data planes recursively from the leaf level, each controller simplifies its topology and exposes the following three types of *logical* and *reconfigurable* data plane entities to its parent.

- **Gigantic Switch (G-switch)** aggregates a number of physical or gigantic switches and the controller. A G-switch is programmable and characterized by an ID, ports, and a

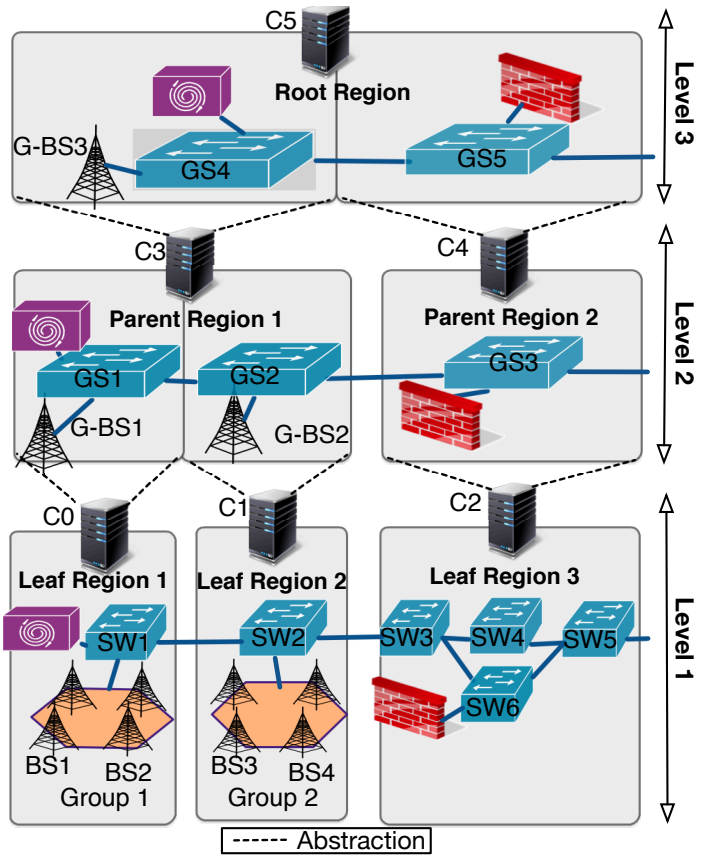


Figure 4.2: A 3-level SoftMoW architecture

virtual fabric (will be clear in Section 4.3.2) and flow table. Each port of a G-switch corresponds to border ports of its constituent switches, i.e. is connected to either Internet domains (*e.g.,* ISP) or neighboring regions.

- **Gigantic Middlebox (G-middlebox)** hides physical or G-middlebox instances of the same type and function (*e.g.,* light weight DPI) and their controller. A G-middlebox can be attached to G-switches, and is identified with the sum of the processing capacities and utilization of constituent instances.

- **Gigantic Base station (G-BS)** summarizes one or more adjacent BS groups or G-BSes, and their controller. A G-BS inherits the union of the radio coverage of underlying base stations and connects to ports of a G-switch.

Abstracting the logical region for the parent. To build the first logical data plane (level 2), each leaf controller builds and exposes a single G-switch for all switches, a G-middlebox

for all middlebox instances of the same type, a G-BS for one or more adjacent BS groups (will be clear in Section 4.5.2). Intuitively, a parent's logical region is the union of regions exposed from its children in the tree. Recursively, non-leaf controllers except the root (*e.g.,* level-2 controllers in Figure 4.2) perform the same procedure on G-middleboxes, G-switches, and G-BSes located in their logical region.

Reconfiguration of logical data plane devices. Each non-leaf controller can reconfigure logical entities exposed from its children. This gives each controller the ability to optimize its descendants' control plane hierarchy and data plane operations without a global state, solely based on its partial view and abstract topology. Any non-leaf controller can initiate a reconfiguration that indirectly causes controllers in its subtree to level-by-level from bottom-to-top interact with each other to modify the exposed logical entities. This new feature enables interesting global applications such as minimizing "east-west" control load in the cross-controller handovers (see Section 4.5).

4.3.2 G-Switch Virtual Fabric

To enable global optimization, *e.g.,* traffic engineering and optimal routing, each controller in the tree hierarchy should know a few pieces information about the internal inter-connections behind its G-switches. SoftMoW exposes a virtual switch fabric for each G-switch. A virtual switch fabric (vFabric) is a succinct representation allowing the parent controller to have three pieces of information per G-switch port pair:*latency*, *hop count*, and *available bandwidth*.

Using standard shortest path algorithms, each child controller constructs these metrics by computing multiple shortest paths for each port pair in its topology. Note that, for the bandwidth metric, different port pairs of the G-switch can share bottleneck links. In this case, if the available bandwidth exposed for a port pair in the child controller's data plane changes more than a predetermined threshold, the child controller will recompute new bandwidths, update the vFabric and notify the parent controller.

4.3.3 Controller Architecture

We design a modular controller architecture as shown in Figure 4.3. A SoftMoW controller consists of a network operating system (NOS), operator applications and an application called RecA that implements the recursive abstraction.

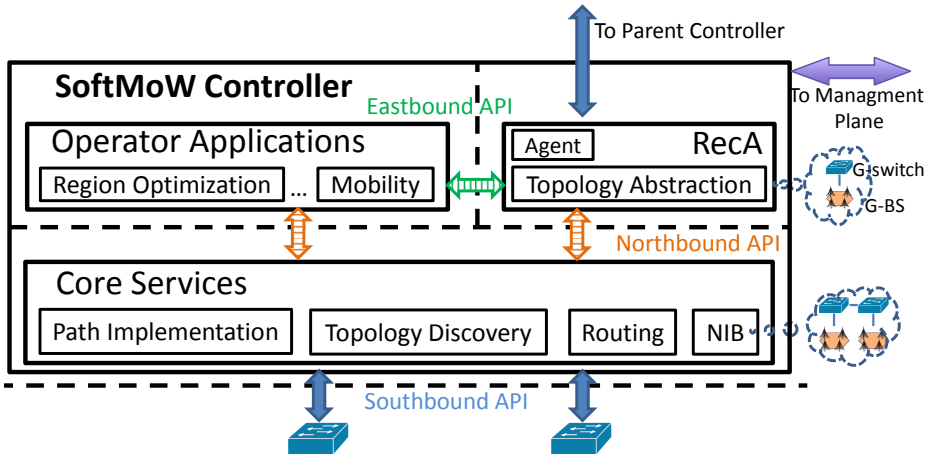


Figure 4.3: SoftMoW controller architecture

Network operating system. SoftMoW expects a number of core services: *path implementation*, *topology discovery*, *routing* and *network information base (NIB) query*. SoftMoW NOS can reuse any existing controller platforms that expose these services through a northbound API. SoftMoW NOS is agnostic of cellular specific functions and other controllers in the hierarchy. NOS communicates with switches (logical or physical) using a southbound API, e.g. OpenFlow API extended to support our virtual fabric feature.

Operator applications. SoftMoW cellular specific functions are implemented as operator applications on top of the NOS, e.g. functions similar to LTE such as home subscriber server (HSS), policy charging and rule functions (PCRF), mobility and new functions such as region optimization and routing optimization. Applications can use the northbound API to get network information (e.g. topology) and set up their configurations (e.g. path setup, sending messages).

Recursive abstraction application (RecA). To implement the recursive abstraction, we design a NOS application called RecA. RecA encapsulates all functions related to the recursive abstraction and provides an eastbound API for operator applications. RecA has two

basic modules: *agent* and *topology abstraction*. RecA's topology abstraction module queries the NIB using the NOS northbound API. It abstracts a network topology (including switches, base stations and middleboxes) as one G-switch, a number of G-BSes (border BS groups need to be exposed in a specific way, will be clear in Section 4.5) and one G-middlebox of each type. The RecA agent communicates with a parent controller (if any). For each logical device, the agent establishes a channel to its parent controller. This way logical devices act as physical ones (*e.g.,* a G-switch acts as a physical switch).

RecA provides the eastbound API to other operator applications. An operator application can register its message type in RecA, and give messages that it cannot handle to RecA; then RecA will send the message to its parent controller as a Packet-In event. The agent also handles messages from the parent. If a message is about path implementation, the agent sends it to the topology abstraction module, which translates the message to multiple messages using the current network view of NIB; if the message is of a type registered by an operator application, it is sent to the application. RecA and operator applications use the northbound API to send messages to logical (child controllers) and physical data plane entities.

Management plane. The management plane bootstraps the recursive control plane. It configures all controllers in the hierarchy via dedicated channels (e.g. assigns IP addresses, and region identifier, and configures the tree structure). The RecA at each controller exports its topology to the management plane. The region optimization applications communicate with the management plane to reconfigure local or physical network devices. The management plane also coordinates UE state transfer during region optimization.

4.4 Core Services

SoftMoW core services provided by the network operating system includes the NIB, topology discovery, routing and path implementation. Similar to the NIB in other controller designs [87], SoftMoW's NIB consists of network devices, device type (e.g. base station,

middlebox, switch), links and their metrics. We assume standard mechanisms (e.g. those in [87]) to gather NIB and maintain NIB's consistency. The NOS has visibility of its own local network topology (physical or logical), does not maintain UE state, is not aware of any ancestor or descendant controllers (may communicate with peer controllers). Now we proceed to present the other three core services.

4.4.1 Recursive Topology Discovery

SoftMoW presents the first topology discovery protocol in a recursively built control plane architecture. Topology discovery in SoftMoW is much more challenging than in flat architectures. This is because only leaf controllers have direct control over physical switches. Yet each inter G-switch link is physical and is only visible to the ancestor controller of both endpoints of the link.

In SoftMoW, each controller discovers a subset of total links of the physical topology. Data plane switches and links (logical and physical) are discovered sequentially from bottom to top; controllers at each level can discover their (inter G-switch) links in parallel. We now proceed to describe the procedures of topology discovery: *G-switch discovery*, *inter G-switch link discovery* and *Computation of G-switch abstraction*. These procedures are performed by RecA and the topology discovery module. Base stations, middleboxes and links with them as endpoints can also be discovered similarly. If base stations and middleboxes do not implement our discovery protocol, they can also be configured by the management plane.

4.4.1.1 G-switch Discovery

Similar to physical switches, the RecA agent of each non-root controller connects to the parent controller. After a controller starts, its topology discovery module first discovers all switches (G-switches or physical switches) in its region. If the switch type is G-switch, the controller also performs a feature request to obtain the virtual fabric information. The G-switch device information is stored in NIB. The controllers use the southbound API (*e.g.,*

Openflow) to get the G-switch information.

4.4.1.2 Inter G-switch Link Discovery

Link discovery message. After G-switch discovery, a controller uses inter G-switch Link Discovery Protocol to find the links between its G-switches. For each G-switch port, it initiates a link discovery message, which has a meta data field and a stack field. The link discovery message traverses through the controller hierarchy down to the physical data plane, goes through a physical link, and is reported from the receiving switch back to its origin along the controller hierarchy. The meta data field carries the properties of the traversed physical link (*e.g.,* latency and loss rate), which is filled by the leaf controller on its path. The stack stores the traversed path in the controller hierarchy with the format of (Controller ID, G-switch ID, G-switch port).

Origination path. In more detail, when the topology discovery module in a controller discovers inter G-switch links, link discovery messages are sent out from each port of G-switches (which is actually received by the corresponding child controller). Intuitively, the link discovery message recursively is passed to lower-level child controllers and finally sent out of a port of a physical switch. The initiating controller pushes its ID, the G-switch ID and the port onto the stack. When the RecA agent of a child controller receives the message from its parent, the message is forwarded to the RecA topology abstraction module. This module extracts the G-switch and port from the top of the stack, and maps them to one of its G-switches and its port. Then, RecA pushes its ID, the G-switch ID and port onto the stack. If the controller is a leaf controller, it also encodes meta data of the physical link into the meta data field. RecA calls the northbound API SendMsg(switch, port, msg) to send the message.

Return path. Both topology discovery module and RecA register the link discovery Packet-In message from the lower-level. When a controller receives a link discovery message from one of its G-switches with an incoming port. It pops the stack to get the (Controller ID,

G-switch ID, G-switch port). In the topology discovery module, if the popped controller ID is its ID, the link discovery message has been originated by itself, so a new inter G-switch link is discovered. This inter G-switch link is added to the NIB of the current controller. In RecA, if the controller ID is not its ID and the stack is not empty, the link discovery message is reported to the parent by the RecA agent; if the stack is empty, the link discovery message is dropped indicating the link discovery message can not return to the initiating controller and there is no inter G-switch link on the path.

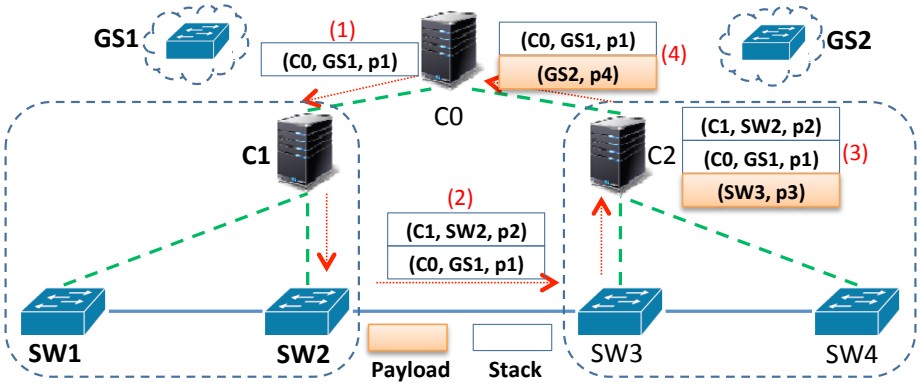


Figure 4.4: A link discovery example in SoftMoW

Example. Figure 4.4 shows an example of inter G-switch link discovery. The root controller intends to discover the link between G-switch GS1 and GS2 on its logical data plane. The link discovery protocol finishes in 4 steps. (i) The root controller C0 initiates a link discovery message. It populates the stack with its own ID C0 and the G-switch ID GS1 and port number p1. (ii) The child controller C1 receives the link discovery message. It translates the G-switch ID and port number into the physical switch ID SW2 and port number p2. Then, C1 pushes (C1, SW2, p2) onto the stack. (iii) Physical switch SW3 receives the message at port p3 and passes it to its controller C2. C2 encodes the receiving (SW3, p3) into link discovery message. C2 pops the stack and find the controller ID at the top of the stack is C1, which is not its ID. So it translates the (SW3, p3) to corresponding ID and port number of its abstract G-switch, which is (GS2, p4), and passes the link discovery message to its parent C0. (iv) C0 pops the stack and find the controller ID at the top of the stack is its ID. In this way, it finds the inter G-switch link between its G-switches (i.e. GS1 and GS2).

4.4.1.3 Computation of G-switch Abstraction

The RecA application in a controller uses the northbound API topo=GetTopology() to get its G-switches and inter G-switch links, and then it computes one abstract G-switch. In an abstract G-switch, all internal ports (i.e. ports between G-switches) are hidden, and all border ports are exposed. SoftMoW also computes other properties between G-switch port pairs, such as latency, bandwidth and hop count as discussed in Section 4.3.2. The parent controller requests the G-switch features (e.g. virtual fabric) from the RecA agent in the child controller via the southbound API. G-BS and G-middleboxes can also be computed similarly. We do not go into the details in this dissertation.

4.4.2 Route Computation

SoftMoW must provide UEs with Internet access. The routing service computes end-to-end optimal paths through the northbound API (path, match fields)=Routing(request, service policy). The inputs are a routing request and a service policy. The outputs are a computed path and match fields to classify the flow. The computed paths are implemented using the path implementation service.

Interdomain routes. To perform routing, SoftMoW interacts with ISPs and content providers through an interdomain routing protocol (*e.g.,* BGP) at egress points. Similar to a RCP server [61], leaf controllers run the route selection procedure on behalf of their gateway switches, each keeping a session with an eBGP speaking router in a neighbor ISP. For each gateway switch, leaf controllers select interdomain routes for all prefixes. In addition, the network performance of each selected route is measured (*e.g.,* hops, latency) [131]. Leaf controllers forward the selected routes to their parent as Packet-In messages, each is associated with performance metrics. The routing module in each controller registers for interdomain routing messages, and puts them into NIB. Recursively, the RecA agent reads the interdomain routes from NIB and sends it to the parent (with translation to the G-switch). This procedure finishes once the root receives interdomain routes from its G-switches.

Recursive routing. When a controller has a routing request from one of its operator's applications (*e.g.,* bearer request), it first checks if its logical region has an interdomain route to the destination on the Internet and the end-to-end internal path satisfies the performance constraints if specified in the request (*e.g.,* latency). In addition, it checks whether the middlebox poset can be met in its logical region if specified in the service policy field. If so, the routing module returns the path and match fields, and then the application implements the path. If no path is found, the operator application delegates the request to RecA agent, which creates a routing request and sends to the parent, where the application in the parent controller registers in the core to get the message and process it. The application also registers for the response in RecA (*e.g.,* to store in local caches). The delegation procedure increases the chance of satisfying the request since the parent has a better global view due to having a larger logical region. We will explain the routing service usage in handling bearer requests (Section 4.5.1).

Optimality discussion. Each controller might need to compute internal paths to interdomain routes through its own egress points. Using the virtual fabric of G-switches, the routing service can find a shortest path between the logical or physical gateway switches and base stations. We can guarantee a shortest path computed by a controller is the shortest is in the controller's region and its corresponding physical topology. We call such paths **locally optimal**. However, the shortest path in a controller's region may not be the global shortest path in the entire topology. We define shortest paths computed by the root controller in its global abstract topology as **globally optimal**. In general, a controller at a higher level is able to compute more optimal paths compared to any controller in its subtree.

Example. Using the interdomain routing messages, we know egress points E1 and E2 are 10 hops away from the address prefix A in Figure 4.5. The leaf controller C2 receives routing request (BS group= Group 2, destination=A) with the constraint of the maximum end-to-end hop count of 14. C2 computes the shortest path (SW2, SW3, SW4) going through $E2$ since it satisfies the performance requirement. This path is a local optimal path in C2's

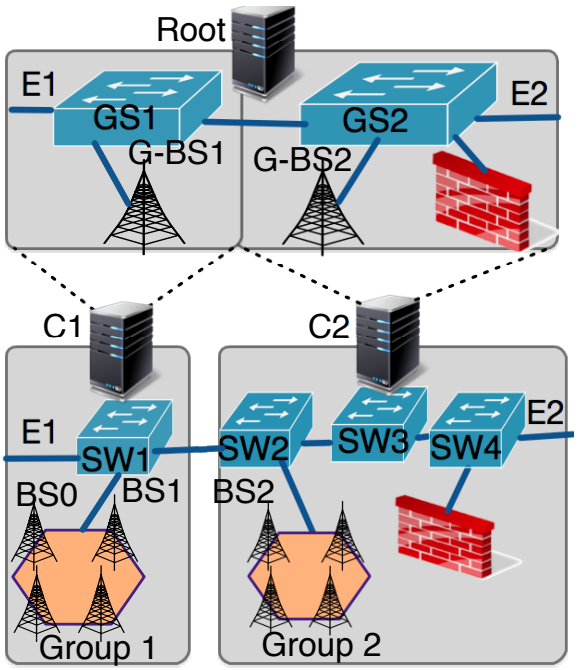


Figure 4.5: Local optimal v.s. global optimal

region. With the global network view, path (SW2, SW1) is one hop closer to the destination. The virtual fabric of a G-switch contains performance metrics for all port pairs. The root has the virtual fabric of G-switches GS1 and GS2, so it can easily compute the globally optimal path exiting from GS1.

4.4.3 Global Path Implementation

In SDN architectures where a controller has full visibility of its physical data plane topology [85, 83], path setup is straight-forward. The controller installs a match-action rule on each switch along the path. The match-action rule can match IP prefixes, VLAN tags, MPLS labels or some combinations of them. In SoftMoW, a controller aggregates flows on the same path, assigns them the same label and sets up routing on labels. So the states in switches can be significantly reduced. However, non-leaf controllers do not have full visibility of the physical data plane topology. We present a scalable mechanism that enables non-leaf controllers to implement paths in their abstract topology onto the underlying physical data plane. A northbound API PathSetup(match fields, path) is provided to applications to set up an input path with certain match fields.

In SoftMoW, a leaf controller can simply implement any intra-region paths. Similar to SoftCell [126], the access switch of base stations can perform fine-grained packet classification and push labels onto packets matching flow rules. Then, switches along the path are programmed to forward traffic based on specified labels. A non-leaf controller does not have control over physical switches, and multiple descendant controllers make partial forwarding decisions; so its path setup is more challenging. Similar to leaf controllers, a non-leaf controller should be able to instruct the access G-switch attached to each G-BS to classify packets and push virtual labels into the traffic, and program its G-switches along any desired path to operate based on pushed virtual labels.

To implement this operation, intuitively, when RecA agent in each child controller receives virtual label switching or packet classification rules, it translates them using its own topology. Each virtual label switching rule is mapped onto internal paths between the egress and ingress ports of the child controller's logical region, and the path computation is performed by the routing module. During the recursive translations, descendant controllers can establish any desired number of internal shortest paths between the ingress and egress points as long as the performance metrics of computed paths comply with the parent's virtual fabric. A descendant controller should be able to push a separate local label on top of the parent's label to establish each local path. Accordingly, the classification rule should be updated for each local path and installed into constituent access switches, each attached to a component G-BS.

High-overhead label stacking. To implement the recursive translations of virtual rules onto physical switches in underlying topology, a simple approach is to recursively stack k labels in the packets where k is the level of the controller initiating the path setup. Label stacking allows a label specified by an ancestor controller to be visible and available in the packets traversing across physical inter-G-switch links detected by the controller itself. Label stacking approach gives the illusion of packets traversing through the region of controllers at different level. When traffic enters a logical region at any level, the controller reads the label

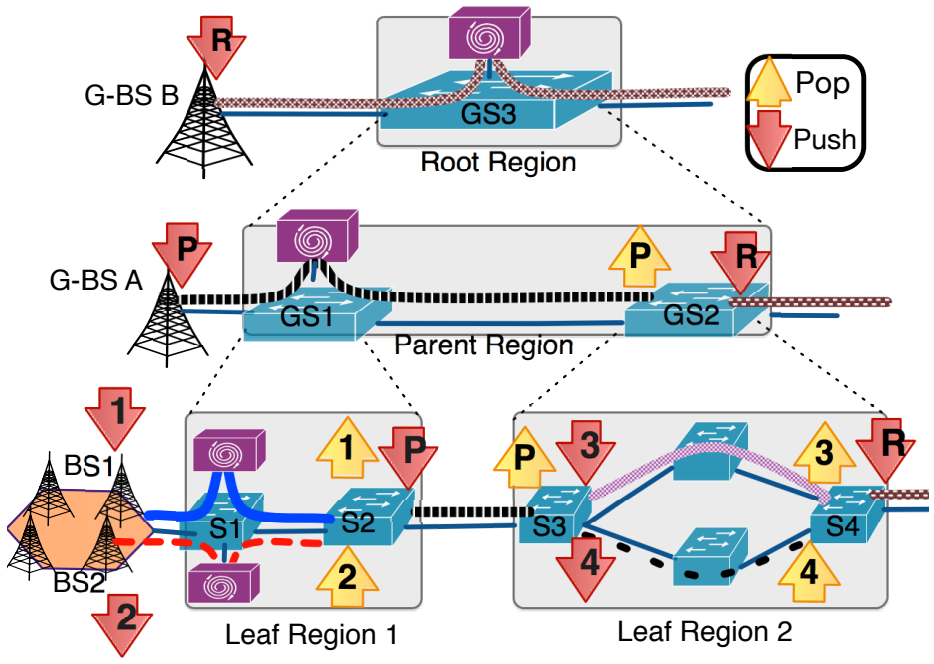


Figure 4.6: Recursive label swapping

in the stack at the same level. This approach is not scalable in nation-wide mobile networks since it increases per-packet overhead due to encapsulating k labels in each packet, which exacerbates the bandwidth consumption as the number of levels in the SoftMoW architecture increases.

Label stacking example. Figure 4.6 shows logical regions of two leaf controllers, their parent, and the root controller (controllers are excluded for simplicity). The root has a single-path service policy for rate-limiting bidirectional traffic between G-BS B and a destination address prefix. To satisfy this policy, the root pushes label R at access switch of G-BS B and then installs the corresponding virtual rule into G-switch $GS3$ to forward traffic specified by label R. At the level below, the parent controller receives the rules. Based on its local view, it decides to stack label P on R (*i.e.,* pushes [R P]) onto the packets. It programs the G-switches GS1 and GS2 to process incoming traffic with label P. In this approach, leaf controller 1 should at *least* push the stack [R P] onto each packet at the base stations. This allows leaf region 2 to read P from the stack and perform the forwarding. Then the rest of the network reads label R of the egress traffic from region 2. Intuitively, this gives the illusion of packets traversing up to the parent region at S2, and traversing down at S3. Also, the packet

traverses up to the root level at S4. It is easy to imagine an increase in the packet header space and network bandwidth consumption, as SoftMoW levels increases, due to stacking multiple labels in packets.

Scalable recursive label swapping. We propose a novel *recursive label swapping mechanism* eliminating the high bandwidth overhead per-packet. In our approach, each packet has only one label at any given time. We have observed that a label specified by a non-leaf controller only needs to be visible across physical inter G-switch links detected by the controller itself. Thus we instruct controllers to perform *label pop* and *label push* operations. Each controller at the ingress switch (physical or gigantic) of its logical region pops the label (specified by an ancestor who controls the just traversed link) of the traffic. It then pushes an internal label corresponding to each internal path. Finally, it programs switches along each path. At the egress switch of its logical region, the controller aggregates the internal paths by popping their label. It then pushes back the ancestor's label onto packets of the flow. This mechanism guarantees the global coordination between the controller by having the necessary label at each switch while it minimizes the bandwidth overhead.

Recursive label swapping example. In Figure 4.6, the root adds label *R* to the traffic group at access switch of G-BS A similar to the previous example. It then programs G-switch GS3 to forward traffic based on label *R* to the rest of network. In this step, the controller of parent region receives the classification and forwarding rules. Using the push operation, it only pushes its local label *P* due to the local preference and does not mark the traffic with label *R*. Using the pop operation, it pops *P* and pushes backs the root's label *R* at G-switch GS2 where it loses its control on the egress traffic.

In the leaf region 1, the leaf controller decides to load balance the packets between two rate limiters, so it implements two local paths with label 1 and 2. With the push operation, it pushes label 1 and 2 at access switches of BS1 and BS2 respectively. With the pop operation, these two labels are replaced with the parent's label *P* at egress switch S2, so the next leaf region can process the traffic. In the leaf region 2, switch S3 is programmed to perform load

balancing on ingress traffic from region 1. The leaf controller implements two separate paths by pushing local labels 3 and 4, and popping *P* at switch S3. These paths are aggregated at egress switch S4. The local labels are popped off and the root's label *R* is pushed back onto the packets. As shown in the physical data plane, packets always carry a single label denoted with different patterns while many controllers make partial decisions.

4.5 Operator Applications

A key cellular network function is mobility management which includes setting up bearers (a bearer provides network connectivity service to the UE) and handovers. Mobility management is performed by the Mobility Management Entity (MME) in LTE whereas it is done by the mobility application in SoftMoW. The key differences are: (1) mobility application is simpler because of the use of the controller's northbound API which is not available in LTE; (2) it supports mobility better (*e.g.,* LTE does not support inter-PGW handovers [135]). LTE mobility management has many procedures, due to the lack of space, we only discuss main functions that highlight the differences. Besides the mobility management, we present a new application, the region optimization application, to reduce the handover load of the controllers.

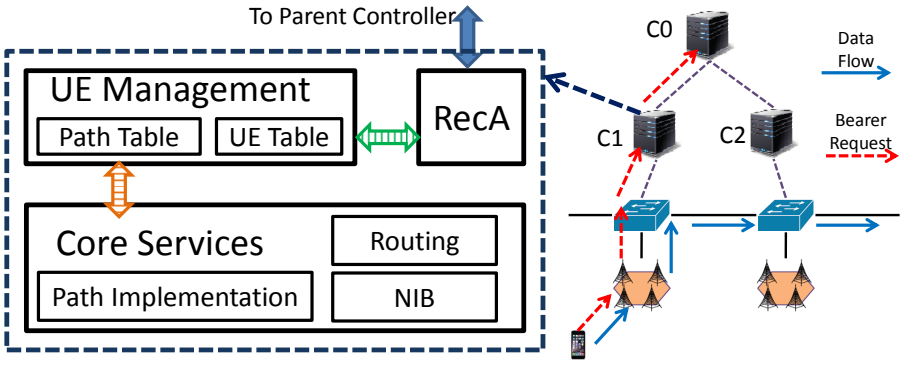


Figure 4.7: UE management application

4.5.1 UE Bearer Management

In each SoftMoW controller, the mobility application registers for the bearer request message type in the core. It also registers for the bearer response from the parent in RecA. The mobility application maintains two tables (Figure 4.7): (i) UE table where each row contains a bearer request and a local path ID. (ii) Path table that maps path IDs to the their details. A bearer request can be in the format of (UE ID, BS ID, SRC IP, DST IP, REQ) where the "DST IP" is the destination address on the Internet and "REQ" contains QoS. For example, some UE applications can request for better QoS on the end-to-end latency.

When a UE sends a bearer request to the base station, the request is forwarded to the leaf controller as a Packet-In message. The mobility application receives the request from the core and associates a service policy (*i.e.,* a middlebox chain) with it if necessary. If there is no precomputed path in the path table, the mobility application calls the routing service using the northbound API (path, match fields)=Routing(request, service policy). Then it calls the northbound API (pathID, pathInfo)=PathSetup(path, matching fields) provided by the path implementation service. Finally, the path information is cached in the path table and the mobility application asks the base station to allocate the resources. As discussed in Section 4.4.2, if the routing service cannot find an end-to-end path that satisfies the bearer request and service policy, the mobility application sends the bearer request to RecA, which is forwarded to the parent controller.

Example. In Figure 4.7, the UE requests for a path with a larger bandwidth, which cannot be found by the routing service of C1 in its region; the request is sent to the root controller C0. C0 computes the path, stores the UE and path information, and sends the UE bearer response to C1's RecA. The C1's RecA implements the local path in it region, and C1's mobility application registers in its RecA to get the path information. Also, C2's RecA implements the rest of path in its region once it receives the virtual rule from C0.

The bearer state is synchronized between the UE and the mobility application. If the UE becomes idle, its bearers will be deactivated. We add two more fields to the UE table

indicating whether a UE is active or idle, and whether the UE request has been handled locally or by the parent. When the mobility application deactivates a bearer, it updates the tables and also asks the path implementation service with the northbound API deactivatePath(pathID, pathInfo) to deactivate its path. If the UE bearer has been handled by the parent controller, the mobility application continues to request bearer deactivation from its parent via RecA.

4.5.2 UE Mobility

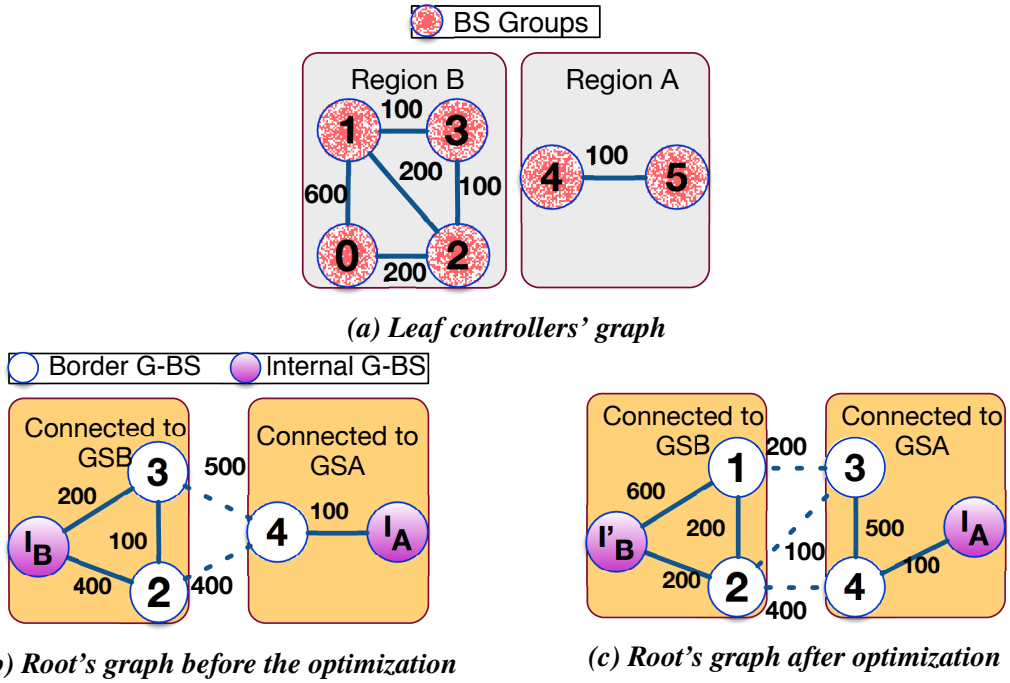
LTE has many handover procedures depending whether the source base station and target base station has a direct connection or not and whether the UE's current associated MME, or the serving gateway needs to be changed or not, etc. Similarly, there are many handover procedures in SoftMoW. We only discuss two main types of handovers: intra region and inter region. The handovers are performed through the coordination of the mobility application, RecA, the routing service and the path implementation service.

The Intra region type is used to handover a UE between a source base station and a target base station when both of them are in the same leaf region. This type of handover is easy, so we focus on complex inter region handovers. In inter region handovers, the source and target base stations are located in different leaf regions. Thus each corresponding border G-BS is exposed by a separate leaf controller. To simplify the inter region handover procedure and allow fine-grained region optimizations, we assume controllers do not aggregate gigantic stations and physical BS groups sitting at the border of their logical region with others in the recursive abstraction procedure. A leaf controller abstracts each border BS group as a single G-BS for its parent, and non-leaf controllers expose a single G-BS for each G-BS located at their region's boundaries. However, controllers can group, abstract, and expose their internal G-BSes and BS groups in different ways.

To handover a UE from the source base station to the target base station in inter region handover, SoftMoW only requires base stations abstracted as a border G-BS to advertise the corresponding *G-BS ID* along with other information periodically through the physical

broadcast channel. When the source leaf controller and the UE agree on the handover target, the source leaf controller sends a handover request to its parent. The request contains at least source and target *G-BS IDs* and *BS IDs*. The mobility application registers for the handover request in the core. If the current controller is the ancestor of both the source and target leaf controllers, it starts a procedure to handle the request; otherwise the request is sent to RecA and forwarded to the parent controller recursively. For simplicity, we explain the inter region handover procedure through an example.

Example. To handover a UE from BS1 to BS2 in Figure 4.5, C1 sends a handover request from (G-BS1, BS1) to (G-BS2, BS2) to the root. The root requests G-BS2 to allocate the resources at the BS2 to the UE. Then, it implements a new path between G-BS1 and G-BS2 to transfer in-flight packets and establishes some paths E2 and G-BS2 for new flows. Once the handover finishes, the root asks G-BS1 to release the resources. It then removes old paths between G-BS1 and E1 as well as between G-BS1 and G-BS2.



(a) Leaf controllers' graph

(b) Root's graph before the optimization

(c) Root's graph after optimization

Figure 4.8: Inter-region handover optimization

4.5.3 Region Optimization and Reconfiguration

Inter region handovers increase "east-west" control plane load because they require the intervention of at least three controllers: the source and target leaf controllers, and the ancestor controller. Allocating more resources to busy nodes in the controller hierarchy is difficult due to the geographical distribution and also increases the intra-node coordination costs. Thus the regions should be refined to reduce this type of load; each non-leaf controller should reconfigure its own logical region to minimize the inter region handover load it handles. To achieve this goal, the region optimization application changes borders between sub-regions, each exposed by an immediate child controller, based on handover patterns. Handover patterns vary across time-of-day. Thus it is difficult to find static borders using an offline and static approach, so each controller should be able to perform optimizations periodically and on a slow time-scale. In particular, we are interested in minimizing inter region handovers at the root (level L) first because a handover request processed and handled by the root goes through more controllers. Similarly, the controllers at the level $n-1$ have a higher priority compared to the controllers at the level $n-2$. Hence we should run the handover optimization algorithm first at the root. Once the root is done, all controllers at level $n-1$ can run the optimization in parallel, and similarly for the levels below.

4.5.3.1 Region Optimization Algorithm

We now discuss the optimization algorithm for a non-leaf controller which we call the initiator controller.

Handover graph input. When the mobility application processes handover requests, it can log these processing. Then a *handover graph* can be computed, in which each node of the graph is a G-BS and an edge shows the number of handover in the past time window (*e.g.,* several hours) between two nodes. The region optimization application can fetches all handover graphs from the mobility application. The two applications can communicate through mechanisms such as inter-process communication. We do not provide any further

details for lack of space.

Example. For a two-level SoftMoW architecture, Figure 4.8b represents a global handover graph built by the root through aggregating histories. Figure 4.8a shows the leaf regions' BS group-level handover graph. As discussed earlier, to allow the root to run fine-grained optimization at the site-group level, leaf controllers have abstracted each border BS group (*e.g.,* BS groups 3 and 2) as a single G-BS (*e.g.,* G-BS 3 and 2) and have exposed to the root. However, they have abstracted adjacent internal BS groups all together. A similar rule applies to any other non-leaf controllers.

Greedy algorithm. Using the handover graph, the region optimization application in the initiator controller computes the reconfiguration of its logical data plane by refining sub-regions, each exposed from a child controller. The region optimization informs the management plane about the changes. The management plane performs the actual reconfiguration. In handover-specific reconfiguration, the initiator detaches a border G-BS connected to a source G-switch and then re-associates it with a destination G-switch. The source and destination G-switches are connected through an inter G-switch links (discovered by the initiator). This operation transfers the control of the border G-BS to new descendant controllers in the initiator's subtree. We propose a simple greedy local search algorithm to decide which border G-BS should be reconfigured by the initiator. In our algorithm, the initiator at each step selects a border G-BS connected to a G-switch, which yields the maximum gain. The gain is defined as the reduction in the amount of inter region handovers requiring the intervention of the initiator.

Example. Figure 4.8b shows the root level handover graph before the optimization showing the root handles 900 inter region handovers between G-switches A and B or the corresponding leaf regions shown in Figure 4.8a. Based on the gain function, the controller selects border G-BS 3 for the reconfiguration since it gives the maximum gain 200 (=500-200-100). The root associates the G-BS with G-switch GSA.

Constraints. We assume we have the lower bound LB_i and the upper bound UB_i on the

amount of control plane loads (*e.g.,* UE arrival) that each G-switch (or actual child controller) can handle. When the initiator picks the maximum gain border G-BS, it avoids reducing the load of a G-switch GS_i to below LB_i or increasing it to above UB_i, assuming the load of each type of control plane events (*e.g.,* bearer arrival) incurred by a G-BS is given.

Termination and Convergence After the above steps, the initiator controller can enter into a new iteration of reconfiguration computation by selecting the next G-BS. The algorithm terminates when there is no more positive gain. *The sequential-parallel approach converges* because the handover optimization at an initiator controller, which is done by refining its logical sub-regions under its control, neither produces nor removes any gains for ancestor controllers except for the initiator itself, and controllers in its subtree. This is because a controller cannot affect inter region handovers seen at ancestor controllers.

4.5.3.2 Reconfiguration Protocol

Region optimization application computes the reconfiguration and sends reconfiguration messages to the management plane.

Finding leaf controllers. The management plane subscribers to topologies changes from NIB and abstraction changes from RecA. Using the topology information and configuration information, the management plane finds the source and destination leaf controllers, and instructs them to fulfill the G-BS re-association request from the region optimization application.

Reconfiguration. At this step, the source leaf controller finds a cut containing switches that are necessary to transfer the border BS group (abstracted as a single G-BS to allow fine-grained optimization) to the target leaf's region. It then communicates with the switches and component base stations to seamlessly add the target leaf controller as their new controller. In this procedure, the source leaf controller sets the role of the target leaf controller to the equal role (*e.g.,* OpenFlow "OFPCR_ROLE_EQUAL"). This role means both the source and target leaf controllers receive all events generated by data plane devices (*i.e.,* BS group,

switches, and middleboxes). The management plane instructs: (i) the source leaf controller to handle events generated by existing rules and avoid installing new rules. (ii) the target leaf controller to process all new requests (*e.g.,* handover, routing, UE arrival, and path implementation). To make states consistent, the source controller transfers existing UE states and path information to the target controller in advance. When old communications finish, the source controller disconnects itself from the data plane devices and the new controller gets the master role.

Updating logical data planes. After a successful control transfer at the leaf level, the logical regions are updated from bottom to top in a recursive fashion to reflect new abstract topologies. Recursively, each RecA agent along the path modifies the G-switch ports and the virtual fabric for its parent. Next, the parent automatically discovers new inter G-switch links. Also, the RecA agents need to update, register, or deregister G-BSes. This is because some internal BS groups in the source leaf region become border BS groups, which should be reflected recursively. Figure 4.8c shows the root's handover graph after reconfiguring G-BS 3. The procedure transfers the control of BS group 3 from the source region B to the target region A. As a result, the new border BS group 1 is separated from I_B, abstracted as border G-BS 1 and exposed to the root. This leads to updating the internal G-BS I_B to I'_B which has lost BS group 1. Also, the target leaf controller might need to treat previous border BS groups as internal BS groups due to an expansion of its region.

4.6 Discussion

We discuss how a basic SoftMoW can handle the controller, switch, and link failures, and implement consistent paths.

Controller failure recovery. To guarantee the reliability of the control plane, each logical node in the tree structure contains master and hot standby instances. For each node, NIB is decoupled from the controller logic and stored in a reliable storage system (e.g. Zookeeper [84]). The NIB is shared between the master and standby. The standby uses

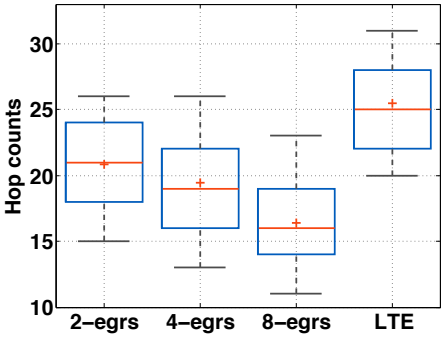


Figure 4.9: End-to-end hop count

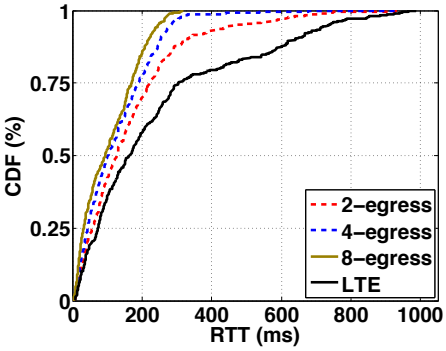


Figure 4.10: End-to-End latency

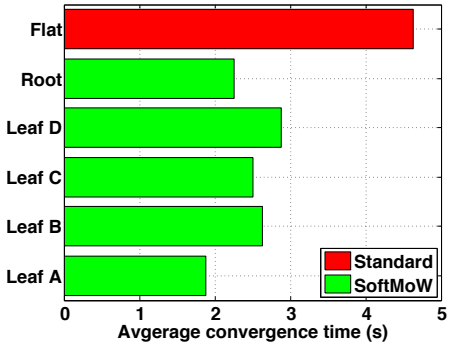


Figure 4.11: Convergence time

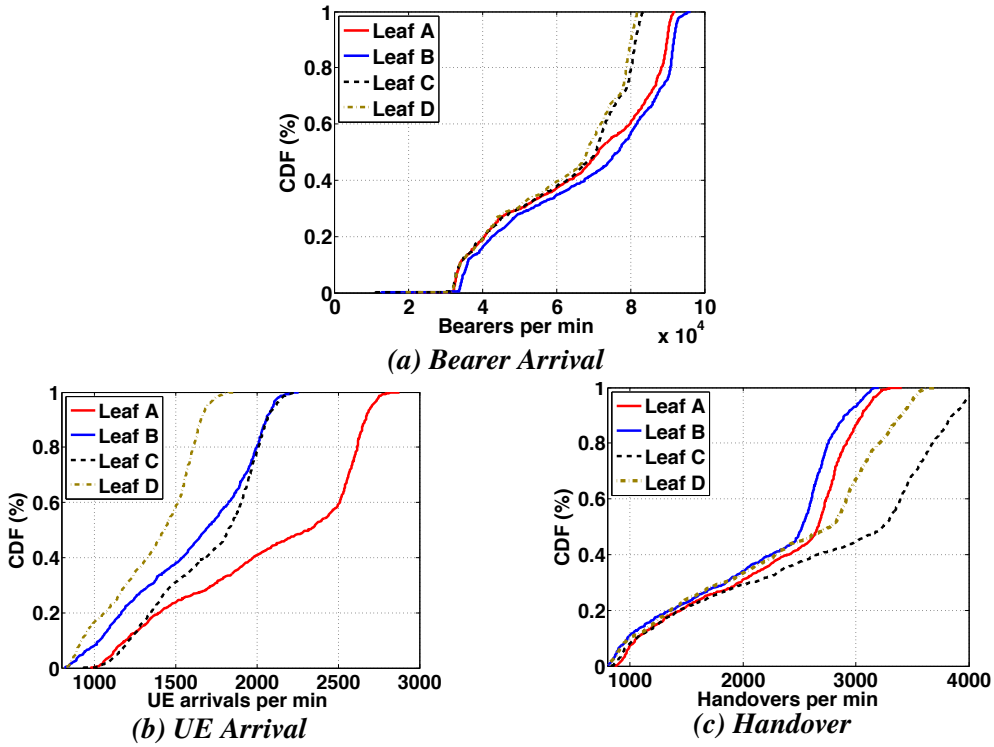
a heartbeat protocol to detect the failure of its master. Also, each physical or logical (*i.e.,* master and standby) switch connects to both master and standby instances. All messages from a physical or a gigantic switch are duplicated and delivered to both instances. If a master is alive, the standby does not do anything. Otherwise, it takes over the master's work immediately. When the master controller receives an event, it first logs the event arrival in the NIB, and then processes it. When the master fails, the hot standby detects this and immediately checks the event logs and redo unfinished events.

Switch and link failure recovery. When a link failure occurs, the leaf or ancestor controller, which discovered the link, is notified through our recursive discovery protocol. If the failure affects the exposed G-switch and virtual fabric in a way that cannot be masked from the ancestor controllers, changes are reflected bottom up which may cause upper-level controllers to recompute new paths. Otherwise the controller finds affected local paths and implements alternative shortest paths with the same performance.

Consistent path setup. In SoftMoW, path implementations by a controller are pushed top-down. However, the topology updates propagate bottom-up. If we want to provide

strong consistency between controllers in neighboring levels, messages needs to be ordered (*e.g.,* paxos, locks) which impacts the agility of path implementations. SoftMoW guarantees eventual consistency. If a failure happens due to inconsistency (*e.g.,* path implementation during topology changes), SoftMoW's controllers recomputes new paths. To guarantee a packet goes through a consistent path during path updates, the new path and packets are assigned a new version number. The packets with the old version number can still use old rules to guarantee reachability.

4.7 Implementation and Evaluation



(a) Bearer Arrival

(b) UE Arrival

(c) Handover

Figure 4.12: Cellular loads on balanced regions

We prototype the architecture of SoftMoW to show the performance gains of SoftMoW compared with current rigid LTE architecture and evaluate the scalability of our topology discovery protocol. Finally we show the effectiveness of inter region handover optimization using trace-driven simulations.

4.7.1 Prototype and Methodology

Data plane. We prototype SoftMoW on top of the Floodlight [17] and Mininet [89]. Leaf controllers use the OpenFlow protocol to communicate with switches while other controllers interact with logical data plane elements through a custom API similar to OpenFlow. We build realistic data plane topologies using the RocketFuel dataset [119]. We present the results for a data plane containing 321 software switches. To attach radio access networks, we use our LTE data set. We connect each BS group to an access switch. Each BS group contains at most 6 inferred base stations organized in a ring topology. The minute-level uplink and downlink traffic rates of BS groups is obtained from the dataset. We set the delay and bandwidth of links to 5ms and 1Gbps respectively.

LTE dataset. We collected about 1TB traces from a large ISP's LTE network during one week in the summer of 2013. The dataset covers a large metropolitan area with more than 1000 base stations and 1 million mobile devices. The trace is *bearer*-level. A radio bearer is a communication channel between a UE and its associated base station with a defined Quality of Service (QoS) class. The trace includes various events such as radio bearer creation, UE arrival to the network, UE handover between base stations. From the trace, we compute the uplink and downlink traffic per minute per base station. When a flow arrives and there is an existing radio bearer with the same QoS class, the flow will use the existing radio bearer. Radio bearers time out in a few seconds, so a long flow may trigger several radio bearer creation and deletion events. Because the data set does not contain flow-level information, we use radio bearers to estimate flow activities.

BS group inference. Our LTE dataset does not contain BS-group level information, so we infer BS groups by a simple algorithm. We assume each group has at most 6 base stations organized based on the ring topology. Our algorithm aims to find groups maximizing the weight of intra-group edges in the global handover graph. The optimal solution is NP-hard, so we design a greedy algorithm. In each iteration, the edge with the lowest weight is removed and then strongly connected components with fewer than 6 base stations are computed. We

remove the components from the working graph and mark each as a new BS group. Finally, inferred BS groups are partitioned to form approximately equal-sized logical regions with similar cellular loads. We carefully assign a geographical location to each BS group to preserve the neighborhood relationship among them.

4.7.2 Routing Performance

We first focus on a two-level architecture with 4 leaf regions. We approximately place the leaf controllers in the center of their region. The root controller runs in the middle of the complete topology. SoftMoW's inter-connected core network increases the choices of Internet egress points so that the control plane can compute optimal end-to-end paths. We compare the two-level SoftMoW architecture with an existing rigid LTE region for the same number of base stations. To model egress points, we use iPlane [22] consisting of traceroute information from PlanetLab [65] nodes to Internet destinations. To consider routing changes, we replay the hop counts and latencies from multiple snapshots. The root implements internal shortest paths for traffic by taking into account both internal hop counts (from the G-BS to an egress point) and external hop counts (from an egress point to the destination).

Figure 4.9 illustrates the distribution of end-to-end hop counts as a function of the number of egress points for 11590 destinations on the Internet. We observe the average hop count decreases from 20.83 to 16 as the number of egress points increases from 2 to 8. This is because internal path inflation disappears since the traffic is directed through sufficiently close egress points, and also diversity of external paths improves the Internet access performance. In particular, SoftMoW with 8 egress points can reduce the average end-to-end hop count by 36% compared to LTE network. In addition, SoftMoW can also reduce end-to-end latencies by computing globally optimal paths at the root. Figure 4.10 depicts the CDF of RTT latency. We observe the 75th and 85th percentile RTT latencies reduce by 43% and 60% when we switch from the LTE network to the 8-egress point SoftMoW.

4.7.3 Discovery Protocol Performance

In the same setting, we now measure the convergence time of our recursive discovery protocol. The convergence time is measured per controller and starts from the beginning of a discovery period until all links and ports are discovered and become stable. We compare our results to the standard discovery protocol (*e.g.,* LLDP) when a single controller is placed at the root's location and discovers all the links and ports.

Figure 4.11 shows the average convergence time for different controllers in our architecture and the flat control plane. We observe SoftMoW's controllers detect their topology between 44% and 58% faster compared to the flat discovery by the single controller. We identified the queuing delay at controllers is the root cause of such differences and the propagation delays between the controllers and switches have insignificant effects. The queuing delay is in proportion to the number of ports and links in topology.

Table 1: SoftMoW Controller Abstractions

	Discovered			Exposed	Exposed
	SW	Ports	Links	Ports	Ports (%)
Leaf A	55	218	80	58	26
Leaf C	79	250	99	52	20
Leaf B	68	213	87	39	18
Leaf D	98	416	167	81	19
Root	4	230	115	-	-

Basically, SoftMoW is more scalable and can detect faults faster compared to flat single controller deployments because a large portion of links and ports are masked from each controller. Table 1 shows the leaf controllers on average have exposed 20.75% of total ports discovered in their logical region to the root controller. Also, 73% of total links are hidden at the root level.

4.7.4 Handover Optimization

We characterize the cellular load on the leaf controllers and the effectiveness of inter region handover optimization through network measurement and simulation. We simulate a SoftMoW with two levels. In the first level, we define four and eight roughly equal-sized logical

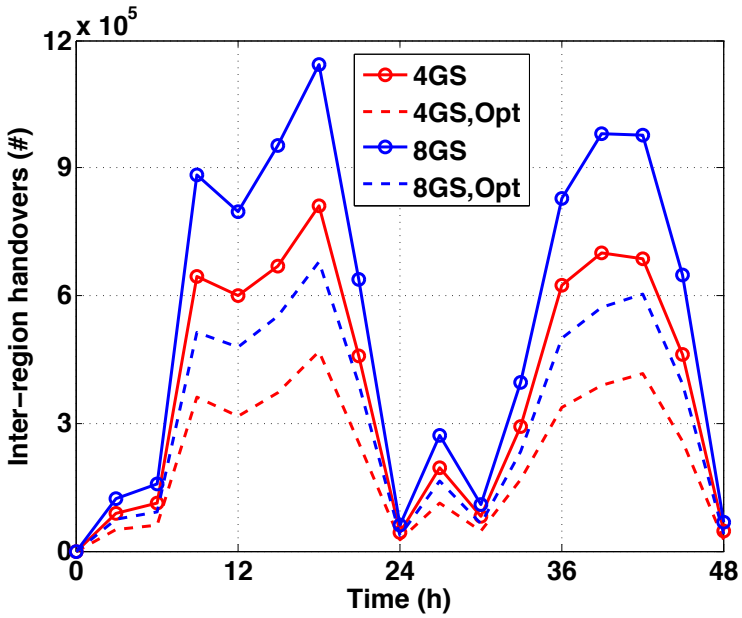


Figure 4.13: Handover optimization

regions, each assigned to a leaf controller. In the second level, the root controller manages the abstract topology.

Cellular loads. Each leaf controller should handle three types of cellular events in addition to exposing logical devices to the root: *bearer arrival*, *UE arrival*, *handover request*. In practice, each type of cellular event can triggers multiple rounds of message passing between the controller and the logical data plane. Figure 4.12a shows the CDF of bearer arrivals. We observe each leaf controller handles as high as 10^5 bearer arrivals per minute. We use the bearer arrivals as the estimate of the number of packet-in messages received by the leaf controllers. Figure 4.12b shows leaf controllers receive and process between 1000 and 3000 attachment requests from UEs connecting to a base station in their region, which are triggered when users turn on their device. Figure 4.12c depicts the aggregate intra-region and inter-region handover requests processed by leaf controllers that varies between 1000 and 4000 per minute.

Optimization results. Periodically, the root refines the abstract sub-regions exposed from the leaf controllers based on its global handover graph. It strives to reduce the load of inter region handovers, which also improves the handover performance. In the optimization, we avoid drastically unbalancing the three cellular loads on each leaf controllers. Figure 4.13

shows the number of inter region handovers handled by the root over 48 hours for 8-region and 4-region settings. We observe the number of handover requests increases (i) in peak hours and (ii) by doubling the number of logical regions. The root runs the reconfiguration algorithm every 3 hours by collecting local handover graphs. We assume each GS (*i.e.,* leaf controllers) should not handle more (less) than 30% of their maximum (minimum) initial cellular loads per minute. Given these constraints, Figure 4.13 depicts the root can reduce the load of inter region handovers by 38.08% to 44.61% using our iterative greedy reconfiguration algorithm.

4.8 Related Work

Scalable control planes. Maestro [62] utilizes parallelism to achieve high scalability on multi-core machines. SoftMoW can benefit from the proposed techniques to make logical and physical rule installations faster at each node. HyperFlow [123] and Onix [87] are multi-controller designs without any explicit hierarchical structure. Kandoo [81] improves HypeFlow by leveraging a two-level controller. Unlike SoftMoW, Kandoo cannot be extended to more than two levels and can run specific applications such as elephant flow detection. In contrast to SoftMoW, these systems do not offer sufficient scalability to support continent-wide global applications.

Scalable data planes. To scale the data plane, SoftMoW, PNNI [74], XBar [94] hierarchically abstract a given network as logical entities. To control their specific target network and satisfy requirements, each of them offers different abstractions. PNNI's abstractions is designed for ATM networks. SoftMoW is the first complete recursive and reconfigurable architecture with richer abstractions suitable for cellular WAN operators. Unlike XBar and PNNI, SoftMoW builds virtual fabrics for its G-switches to enable network-wide optimization such as routing. In addition, SoftMoW runs a novel recursive label swapping mechanism to minimize the bandwidth overhead and data plane states.

Inter-DC control plane. Control plane architectures for data center WANs such as

B4 [85] and SWAN [83] are specific to inter-DC traffic engineering. Inter-DC WAN topologies have several order of magnitudes fewer nodes and edges compared to the cellular WAN topologies [63]. SoftMoW's recursive and reconfigurable abstraction scales the network much better.

Cellular network control plane. Recently, researchers have also proposed flexible control plane architectures for cellular networks. SoftRAN [78] is a design specific to radio access networks. SoftRAN handles intelligent resource block allocation to optimize utilities. SoftCell [126] focuses on providing operators with fine-grained policies and compresses data plane rules. In contrast to prior work, SoftMoW handles inter-connected cellular core networks.

CHAPTER V

Caesar: A High-Speed and Memory-Efficient Forwarding Engine for Next-Generation Internet and Cellular Core Architectures

5.1 Introduction

Aiming at providing a more secure, robust, and flexible Internet and 5G cellular networks, the networking research community recently has focused on developing new architectures for these networks. For instance, our SoftBox, SkyCore, and SoftMoW proposals aim at developing an efficient and scalable 5G core architectures to realize emerging mobility services and use cases. AIP [54] introduces accountability at the IP layer, thus enabling simple solutions to prevent a wide range of attacks. XIA [80] supports an evolvable Internet by providing the capability to accommodate potentially unforeseen diverse protocols and services in the future.

Most of these proposals either switch to or benefit from a network addressing scheme instead of IP that has two important two features: Each address is decoupled from its owner's network location and permits its owner to cryptographically prove its ownership of the address. The separation feature enables improved mobility support and multi-homing. The cryptographic aspect facilitates authentication and authorization of control and data messages. However, on the down side, both features require addresses to be inherently *long* and thus take

up significant memory space due to a lack of hierarchical structure to support aggregation. For instance, in the design of MobilityFirst [50], each address component can be a few kilobits in size. Not surprisingly, it is expected to have forwarding tables on the order of gigabytes in future Internet and cellular architecture designs [80]. Such addressing schemes make the design and implementation of high-speed border routers challenging as detailed below.

First, memory provisioning becomes more difficult compared to existing network elements. The future Internet and 5G networks will experience a tremendous surge in the number of addressable end-points. Recent studies [49, 42] have predicted that the number of connecting devices and active address prefixes will jump to 50 billion and 1.3-2.3 million, respectively, by the end of 2020. On the other hand, the current rapid growth of the number of address prefixes (*i.e.,* about 17% per year) is the root of many existing problems for operators, who have to continuously shrink the routing and forwarding tables of their devices or upgrade to increasingly more expensive data planes [56].

Second, power consumption of border routers is expected to increase substantially. Most high-speed routers and switches utilize a specialized fast memory called Ternary Content Addressable Memory (TCAM) due to its speed and in particular its parallel lookup capabilities. TCAM is the most expensive and power-hungry component in routers and switches. It requires 2.7 times more transistors per bit [52] and consumes an order of magnitude more power [133] compared with the same size of SRAM. Therefore, increased address length imposes substantial cost and power consumption in particular on high-speed border routers with TCAM. Although software-based solutions might seem viable, their forwarding speed cannot compete with TCAM-based routers that can support up to 1.6 billion searches per second under typical operating conditions [15].

Third, the critical-path fast memory components of high-speed routers are small in size, and their capacity does not increase at a rate that would accommodate the large addresses of future Internet and 5G designs in the foreseeable future. Moore's law is only applicable to

slow memories (*i.e.,* DRAM) but not to fast memories [21]. As a matter of fact, we have observed that the TCAM capacity of the state-of-the-art high-speed routers has remained mostly unchanged for several years. As a result of limited memory, network operators still have difficulties in dividing the memory space between IPv4 and IPv6 addresses [45].

To address these challenges, recent research has offered scalable routing tables [115] and forwarding engines (*e.g.,* storage-based [100] and software-based [80]) for the new addressing schemes. Unfortunately, these solutions have limited performance due to the approach of storing addresses into slow memories. Also, due to a lack of address compression, efficiency and scalability of their proposed schemes are inversely proportional to the length of addresses. The same limitation also makes a large body of research in IP lookup [111], which optimizes longest prefix matching, ill-suited for flat and non-aggregatable addresses.

This chapter presents *Caesar*, a high-speed, memory-efficient, and cost-effective forwarding and routing architecture for border routers of next-generation network architectures. While we present Caesar for the *generalized future Internet architecture*, we can adopt the Caesar design and techniques in SkyCore, SoftBox, and SoftMoW to further optimize them. At a high level, Caesar leverages Bloom Filters [59], a probabilistic and compact data structure, to group and compress addresses into flexible and scalable filters. Filters[1] have been used in designing routers for both flat (*e.g.,* [128, 77]) and IP (*e.g.,* [66, 117]) addresses. These designs are optimized for small-scale networks (*e.g.,* layer two networks) and do not provide *guaranteed* forwarding speed and *full* correctness. Therefore, Caesar focuses on improving performance, memory footprint, energy usage, and scalability of routers deployed at future Internet domain borders.

5.1.1 Summary of Contributions

In particular, we make the following contributions:

- We propose a new method for grouping self-certifying addresses into fine-grained

[1] We use "filter" as a shorthand for Bloom Filter throughout this dissertation.

filters. The grouping scheme minimizes route update overhead and supports diverse forwarding policies. We also design the first high-speed forwarding engine that can handle thousands of filters and forward almost all incoming packets within three fast memory accesses.

- We design a backup forwarding path to ensure the correctness of forwarding. Our approach leverages the multi-match line of TCAM to detect false positives at high speed. We also introduce a blacklisting mechanism that efficiently caches RIB lookup results to minimize the frequency of accessing slow memory. In contrast, previous work either accesses slow memory several times per packet [66] or randomly forwards packets [128] when false positives occur.

- We strategically leverage counting filters [72] to support address removal while keeping the memory usage benefits of standard filters for high-speed forwarding. To achieve the best of both worlds, for each standard filter in TCAM, we construct a "shadow" counting filter in slow memory and *always* keep standard filters highly utilized in address removal and insertion procedures.

- Based on hash coding theory [134], we propose a hash computation scheme for filters to reduce the number of computations from k to $\log(k)$ per lookup (k is the number of hash functions for a filter). We show that the lookup processing overhead can be reduced by up to 70% compared to the flat scheme. Also, our scheme requires at most $1.16\log(k)$ hash computations for finding k different positions in a small filter while the flat scheme needs up to $1.5k$ computations.

- We perform analysis and extensive simulations using real routing and traffic traces to demonstrate the benefits of our design. Caesar is more energy-efficient and less expensive (in terms of total material cost) compared to optimized IPv6 TCAM-based solutions (*e.g.,* [129]) by up to 67% and 43% respectively. In addition, the cost of our

design remains constant for various address lengths.

5.2 Background and Motivation

Caesar's focus is on the *generalized future Internet architecture* but we can adopt the design and techniques presented in this chapter in the SkyCore, SoftBox, and SoftMoW architectures to further optimize them. As illustrated in Figure 5.2a, the generalized architecture is comprised of a set of *independent accountable domains (IADs)*. An IAD represents a single administration that owns and controls a number of small *accountable domains (ADs)*. For example, an AD can be a university or an enterprise network. In this model, each end host uses a global identifier (GID) to attach to ADs. In addition, a logically centralized name resolution service stores GID $\longleftrightarrow$ AD mappings to introduce new opportunities for seamless mobility and context-aware applications.

Packet forwarding at borders? The architecture has different routing and forwarding mechanisms compared to today's Internet. In particular, borders routers sitting at the edge of ADs build *forwarding states* or mappings between *destination ADs and next-hop ADs*. Formally, when a border router of AD_i receives a packet destined to $AD_d : GID_d$, it forwards the packet, through a physical port, to a next hop AD on the path to AD_d. The same procedure occurs until the packet reaches a border router of AD_d. Finally, based on GID_d, it is sent to an internal router where the destination end host is attached. In this procedure, AD addresses are cryptographically verifiable and thus they are long and non-aggregatable. The length of addresses is typically between 160 bits [54] and a few kilobits [50] leading to forwarding tables on the order of gigabytes [80]. In the future, larger address lengths are expected to counter cryptanalytic progress.

Why Bloom filters? Caesar employs filters to compress the forwarding states, *i.e.,* AD to next hop AD mappings, in the *border* routers. However, Caesar can be extended to support forwarding schemes with more components in its other pipelines (*e.g.,* XIA [80]). It also supports various standard forwarding policies (*e.g.,* multi-path and rate-limiting). A filter

is a bitmap that conceptually represents a group. It responds to membership test queries (i.e., "Is element e in set E?"). Compared to hash tables, filters are a better choice. First, they are length-agnostic, *i.e.,* both long and short addresses take up the same amount of memory space. Second, a filter uses multiple hash values per key or address, thus leading to fewer collisions. In the insertion procedure, a filter computes k different and uniform hash functions ($h_1, h_2, \ldots h_k$) on an input and then sets the bits corresponding to the hash values to 1. In a membership test, a similar procedure is followed; if all the bits corresponding to hash results have the value of 1, it reports the element exists otherwise the negative result is reported.

5.2.1 Caesar Design Goals and Challenges

Using filters for minimizing fast memory consumption poses several design challenges that are unique to the future Internet scale and Caesar's role as a high-speed border router, which make our work different from previous designs using similar techniques (*e.g.,* [77, 128, 117, 71]).

Challenge 1: Constructing scalable, reliable and flexible filters. Compared to the future Internet scale, a data center or enterprise network is very small in size with orders of magnitude fewer addresses. In such *single-domain*, *small-scale* networks, designing filters to compress forwarding states of flat addresses (*e.g.,* layer two (MAC) addresses) is straight-forward. One widely used approach is to construct multiple filters in each switch, each storing destination addresses reachable via the same next-hop port on the shortest path (*e.g.,* see [128, 77, 71]). Based on this approach, each switch generates and stores a few very *large* filters in terms of bit length and constituent members (addresses) since the number of ports on a switch is limited.

We argue this filter construction is very coarse-grained and thus not sufficiently scalable and flexible to be used in Caesar, because our target network consists of *multiple independent domains* and has a *higher scale*. First, there can be millions of AD addresses in the future

Internet, putting tremendous pressure on the forwarding plane. It is neither scalable nor reliable to store hundreds of thousands of AD addresses into each filter. This is because even a single bit failure in the filter bitmap can risk correctness by delivering a large portion of traffic to wrong next-hop ADs. Second, AD addresses are from various administrative domains, each of which can publish extensive routing updates. Because of storing many addresses into a few large filters, the above approach interrupts or "freezes" packets in the forwarding pipeline at a higher rate in response to each update. This is because modifying a filter requires inactivating the entire bitmap for consistency. For these reasons, the design of Caesar benefits from fine-grained filter construction with higher scalability and flexibility.

Challenge 2: Providing guaranteed high-speed forwarding. Caesar's goal is to achieve a forwarding rate similar to that of high-speed border routers (*e.g.,* 100s of millions of packets per second). However, compressing addresses into filters creates a bottleneck in the processing pipeline. To run a membership test on a filter, we need to compute k hash functions and access the memory k times in the worst case. Previous designs do not provide hash computation optimization and also access filters naively (*e.g.,* [66, 77]). Thus they have limited peak forwarding speeds, on the order of a few hundred kpps (*e.g.,* [128]), even for fewer than a hundred filters. This is orders of magnitude smaller than Caesar's objective. Also, instantiating more filters to support fine-grained policies makes existing designs more inefficient.

Challenge 3: Avoiding Internet-wide false positives. One key limitation of compression using filters is occasional false positives; that is, a filter incorrectly recognizes a non-existing address as its member due to hash collisions. In this case, all positions that correspond to hash values of the address have been set to 1 by insertions of other addresses. For a filter, there is an inherent tradeoff between the memory size and false positive rate. A filter naturally generates fewer false positives as memory footprint increases. For Caesar, false positives can result in Internet-wide black holes and loops, thus disrupting essential Internet services. To address this problem, multiple solutions have been proposed (*e.g.,* [96, 128]) that either

are very slow, incur domain-level path inflation or offer partial correctness. Caesar cannot borrow them because, as a border router, it must provide *deterministic correctness* at *high speed*.

Challenge 4: Updating filters and maximizing their utilization. Routing and forwarding tables might need to be updated. Supporting updates poses two challenges to Caesar. First, a routing message can lead to address withdrawal from filters. However, removing an address from a standard filter inevitably introduces false negatives. An address is mapped to k positions, and although setting any of the positions to zero is enough to remove the address, it also leads to removing any other addresses that use the same position. Second, even with supporting address removal, the total utilization of filters and the compression rate can be negatively impacted if many addresses are removed from a filter and distributed into other filters.

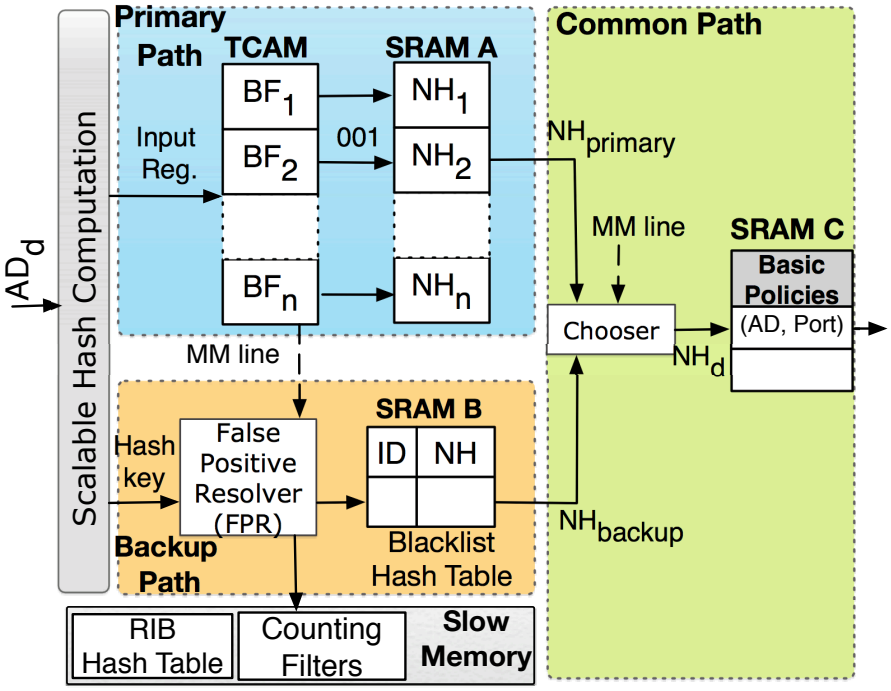


Figure 5.1: Caesar architecture. The backup path result is selected when MM (multi-match) flag is high.

5.2.2 Caesar Architecture Overview

Caesar benefits from two logical data structures: a routing information base (RIB) and a forwarding information base (FIB). The RIB maintains all paths to destinations ADs; the FIB is used to match ingress packets to outgoing links. Similar to modern hardware routers, Caesar implements the RIB and FIB using slow and fast memories respectively.

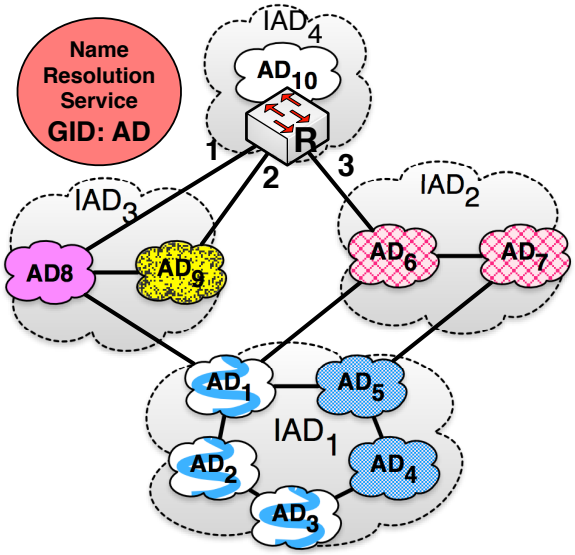
Caesar has a novel FIB design as illustrated in Figure 5.1, which consists of two *forwarding paths* or *pipelines*. Each pipeline performs a different series of actions on the input packet, but they both run in parallel. The vast majority of packets go through the *primary path* that leverages our scalable and flexible filters constructed in TCAM (Section 5.3). The *backup path* is built from the fast memory and handles uncommon cases where the primary path is not reliable due to false positives in the filters thus *rarely* is less efficient when it accesses the RIB (Section 5.4). In other words, the primary path ensures the common-case high-speed forwarding while the backup path guarantees the correctness.

Caesar minimally extends the RIB to support routing updates and keep filters of the primary path highly utilized in such events; it also optimizes the computational overhead of hash functions to remove a potential processing bottleneck (Section 5.5). Our design provides a practical solution that can be implemented by existing hardware (*e.g.,* SDN switches) with guaranteed performance. More importantly, our design can be replicated to support specific future forwarding schemes (*e.g.,* XIA [80] having more address components and the backward compatibility feature).

5.3 Primary Forwarding Path

We first describe our design of the primary forwarding path. A simple approach to compressing forwarding states is to group all destination addresses reachable through the same outgoing interface into a filter (*e.g.,* Buffalo [128]). In this section, we first discuss how our high-speed filters minimize data path interruptions, improve the reliability, and allow rapid

false positive handling compared to the simple method (Section 5.3.1). Then we describe how we dynamically instantiate filters and perform parallel membership tests (Section 5.3.3)



(a) Generalized future Internet architecture

Filter #	Members	Next Hop Pairs
IAD_{11}	$\{AD_1, AD_2, AD_3\}$	$(AD_8,1)$
IAD_{12}	$\{AD_4, AD_5\}$	$(AD_6,3), (AD_9,2)$
IAD_{21}	$\{AD_6, AD_7\}$	$(AD_6,3)$
IAD_{31}	$\{AD_8\}$	$(AD_8,1)$
IAD_{32}	$\{AD_9\}$	$(AD_9,2)$

(b) Caesar approach. Filter IAD_{ij} denotes j^{th} filter assigned to IAD_i

Filter #	Members	Next Hop Pair
1	$\{AD_1, AD_2, AD_3, AD_8\}$	$(AD_8,1)$
2	$\{AD_4, AD_5, AD_9\}$	$(AD_9,2)$
3	$\{AD_4, AD_5, AD_6, AD_7\}$	$(AD_6,3)$

(c) Simple approach (e.g., Buffalo [128]). Filter i is assigned to outgoing port i

Figure 5.2: Caesar's scalable and reliable filter construction in border router R.

5.3.1 Scalable and Reliable Filters

As shown in Figure 5.2b, Caesar's control logic stores forwarding states into multiple *fine-grained* filters in the data path, presenting a new abstraction. Each filter encompasses a group of destination AD addresses and is mapped to forwarding actions or instructions. To forward an incoming packet, the data path in parallel performs membership tests on filters and then learns how to deliver the packet to outgoing ports. At the control plane, Caesar introduces two *primary* properties to group and store destination AD addresses into filters. AD addresses that have the same properties at the same time get an identical group membership, and consequently are encoded into the same logical filter. Caesar's control plane is also flexible to define additional properties to form various groups. The primary properties are as follows (the design rationale will be clarified in subsections 5.3.1.1 and 5.3.1.2):

- **Location property** separates destination ADs that are advertised and owned by the same IAD from the others.

- **Policy property** separates destination ADs that are under the same forwarding policy, which is determined by Caesar's control plane, from the others.

Caesar's control logic continuously determines the FIB entries, and forms groups and constructs filters based on the local properties. Then, it couples each filter to *the forwarding policy* of the group. For simplicity, we focus on a basic forwarding policy below, even though Caesar supports more complex policies (*e.g.,* rate-limiting). For a destination AD address, Caesar's *basic policy or next-hop information* includes *all* (next-hop AD, outgoing port) pairs that are selected by the control plane for forwarding ingress traffic destined for the AD. For multi-path forwarding, the next-hop information simply consists of multiple such pairs.

Example. In a multi-path scenario, filters of border router *R* are shown in Figure 5.2b. Based on the Caesar's control logic outputs, destination ADs with the same policy and location properties are filled with the same pattern, each representing an address group

(Figure 5.2a). Then the groups are stored into five filters in Caesar's data path (Figure 5.2b). In this example, traffic to each of the addresses AD_4 and AD_5 is desired to be forwarded on multiple paths. For input packets, the data path runs parallel membership tests on the filters to retrieve the next-hop information at high speed (Section 5.3.3). We save memory from two aspects. First, we hash each long address into a few positions within a small filter. Doing so consumes significantly less memory than storing the original address does. Second, we reduce the memory usage of the next hop information by decreasing the number of FIB entries. Caesar further minimizes the overhead of maintaining next-hop information (Section 5.3.4).

5.3.1.1 Why Separation by Forwarding Policy?

At the high level, the *policy property* isolates destination AD addresses under the same forwarding actions from the others, and allows us to guarantee data path correctness. For any action or policy supported in the data path of Caesar routers (*e.g.,* rate limiting, ACLs, or next-hop information), the policy property ensures each address is only inserted into *one* group and thus leads to disjoint filters. This is a key design decision that allows our false positive detection procedure to work at high speed (will be detailed in Section 5.4.1).

Multi-match uncertainty problem. Existing address grouping approaches used in previous filter-based routers mostly store an address into multiple filters and inevitably make the reasoning about membership tests both hard and slow (*e.g.,* [77, 117, 66, 128]). For example, Figure 5.2c shows how Buffalo [128] establishes a simple approach to construct one filter per outgoing port, which is referred as "simple grouping method" in this chapter. Buffalo suffers from an uncertainty in its data path operations in multi-path forwarding scenarios as follows. Assume we are interested in splitting incoming traffic destined for an AD address into multiple outgoing links. The simple grouping method installs the AD address into multiple filters, each assigned to one of the egress links. For example, Buffalo inserts AD_4 and AD_5 into filters 2 and 3 in Figure 5.2c to perform load balancing. This

potentially equivocates the lookup operation output. If there are multiple matching filters, it is impossible to immediately distinguish between two states: 1) true multiple matches in the multi-path forwarding; and 2) multiple matches due to one or more false positives.

Current solutions. There are two solutions in the literature for mitigating the multi-match uncertainty problem in filter-based routers and switches. The first category of solutions accesses the RIB stored in slow memory and checks all candidates sequentially [66, 117] when multiple matches happen in a lookup. The other category of solutions forwards packets randomly without further checking or randomize filters [128, 112]. Because of insufficient performance and poor correctness, Caesar constructs disjoint filters, each of which is coupled and mapped to the *entire* forwarding actions of the group (*e.g.,* all specified next-hop pairs in multi-path scenarios) in its data path. For instance, in Figure 5.2b, Caesar stores AD_4 and AD_5 only into the filter IAD_{12} that is associated with both next-hop pairs as its forwarding actions. Therefore, Caesar expects exactly one matching filter from the lookup operation. Note if there are other policies or actions in addition to next-hop pairs, we can aggregate them in a similar way to build up disjoint filters.

5.3.1.2 Why Separation by Location?

As shown in Fig 5.2, the *location property* isolates destination AD addresses of different IADs into separate logical groups and makes constructed filters flexible and reliable. It minimizes processing interruption and performance degradation when the control plane updates a forwarding state in the FIB once it receives route updates or locally enforces new forwarding policies.

First, given there can be millions of AD addresses in the future Internet, the location-based isolation *systematically* ameliorates the reliability challenges by making defined groups small in size and shrinking filters in width substantially. Therefore a small portion of the Caesar's FIB becomes "frozen" when a desired filter is inactivated during its bitmap update, or when a bit failure occurs in a bitmap. However, existing designs that use the simple filter

construction method (*e.g.,* Buffalo [128]) can disrupt traffic forwarding to many destinations and are prone to more failure. This is because they store millions of addresses into a few very large filters.

One can use other properties to make groups more specific and smaller, but the location-based separation also limits side effects of *Route flapping* events, which have been identified by other work [69]. In the future Internet context, these events occur because of a hardware failure or misconfiguration in a border router of an *IAD*. In this case, the router advertises a stream of fluctuating routes for *AD*s in its owner *IAD* into the global routing system. However, Caesar's data plane keeps the majority of filters protected from any bitmap modification in response to such route updates, except those filters built for that problematic *IAD*.

Second, the location-based isolation allows Caesar to enforce business-specific policy efficiently. For example, Caesar's control plane can dynamically stop forwarding traffic to ADs in a specific IAD (*e.g.,* due to political reasons [127]) without interrupting traffic forwarding to other AD addresses.

5.3.2 Memory Technology for Filters

In practice, the number of filters generated based on the two primary properties can be high. This is because Caesar constructs more specific and fine-grained address groups. We approximate the worst case number of filters that might be constructed in our forwarding engine. To achieve our performance requirement, the approximation is used to find the best memory technology for filter implementation.

Let d denote the total number of IADs throughout the Internet. Also, let p be the total number of different forwarding policies that can be defined by the control plane of a Caesar router. Then the number of filters is $O(dp)$. For example, 2M filters are generated for $p = 20$ and $d = 10^5$ in the worst case. This poses performance challenges because Caesar must test filters very fast to achieve a high forwarding rate (*e.g.,* 100s of millions of packets per second (Mpps) [11]).

SRAM is the fastest memory technology in terms of access delay (about 4*ns* based on Table 1). However, it can provide high-speed forwarding rates only when it stores a small number of filters, and the performance dramatically degrades to a few kpps even when a few hundred filters are tested in a lookup [128]. This is because the memory bandwidth is limited (even for multi-port SRAMs) and there is a lack of parallelism in accessing multiple filters, each requiring k memory accesses in a membership test in the worst-case (k is the number of hash functions). Therefore serialization and contention become intensified as many fine-grained filters are instantiated.

To overcome the above limitations, we propose to realize filters using TCAM due to its three advantages over SRAM. First, it supports parallel search operation that can be used to lookup filters in one clock cycle (Section 5.3.3). Second, we can intelligently leverage one of its flags to handle false positives (Section 5.4). Third, it has less implementation complexity compared to the approach of using distributed SRAM blocks [88].

5.3.3 Parallel Lookup of Filters

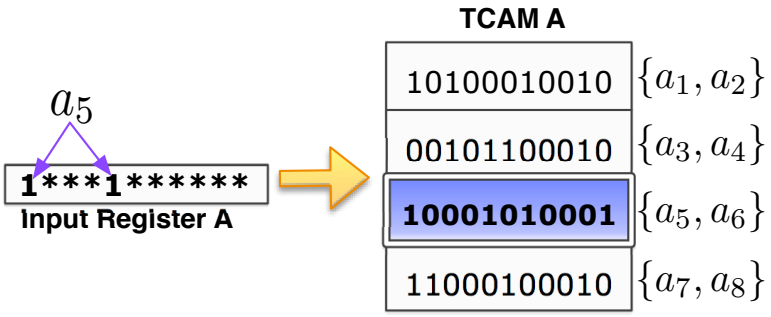
As shown in Figure 5.1, Caesar encodes filters that are heterogeneous in bit width and constituent members in TCAM data entries to attain its desired forwarding rate. TCAM is an associative memory that is made up of a number of entries. All data entries have an identical width, which is statically configurable by combining multiple entries. For example, a TCAM with a base width of 64 bits can be configured to various widths such as 128, 256, and 512 [51]. As shown in Figure 5.3, each bit of the memory and input register can be set to either 0, 1, or * (don't-care). To search a key, TCAM in parallel compares the content of the input register, which contains a search key, with all memory entries in one clock cycle. When there are multiple matches, it returns the index of the matching entry with the highest priority. Typically, an entry at a lower address has higher priority.

Heterogeneous filters. Since each *IAD* manages a different number of *AD*s, top-tier *IAD*s that form the core of the future Internet can own a lot more *AD*s compared to others.

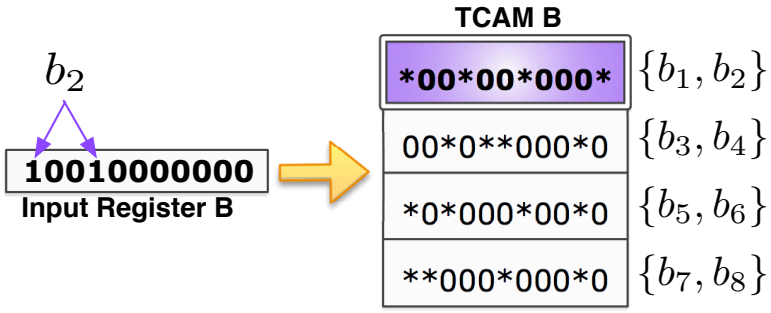
Hence the grouping properties can produce heterogeneous filters from three aspects: for a filter, the number of inserted addresses, the filter bit width, and the number of hash functions (in the insertion and test procedures) can be different from the others. We can show it is possible to store such heterogeneous filters in TCAM. For example, we can extend short-width filters by filling the don't-care values into different positions, but the memory space is wasted and the filter management becomes complex in terms of the insertion and membership test procedures.

Caesar memory allocation strategy. To avoid the low utilization and memory management overhead, we construct equal-sized filters that have identical bit width (w) and use the same k-hash functions. Caesar defines a global maximum capacity for filters by which it restricts the number of ADs in them. This maximum capacity, n_{max}, can be configured by the router's bootstrap program. By defining the maximum capacity, we can limit false positives in practice, and theoretically calculate an upper bound on their rate. Instead of storing all addresses of a group into a large filter, Caesar allocates, releases, and combines equal-sized filters depending on the size of a group that might change over time due to address insertion and removal into the data path (details in Section 5.5.2). In the evaluation, we conduct experiments to study how different n_{max} and w values affect the trade-off between the filter utilization and false positive rate (Section 5.6.2). Below, for simplicity, we focus on the lookup procedure in the primary path when n_{max} and w values are given.

Caesar's parallel filter lookup. Assume Caesar's TCAM contains a set of w-bit disjoint filters, each storing at most n_{max} destination AD addresses. Each equal-sized filter occupies a memory entry, as shown in Figure 5.3. We design two options in Caesar to perform parallel filter lookups. Suppose we would like to retrieve the basic forwarding policy or the next-hop pairs of an incoming packet destined for AD_{key}. First, k hash functions (H=$\{h_1, h_2, \ldots h_k\}$) are computed on AD_{key}. In the first option (Figure 5.3a), we set all the positions of the input register that do not correspond to H (*i.e.,* not set by any of the k hash functions) to the don't-care value, and set all other positions to 1. When the search is issued, the TCAM



(a) Option 1



(b) Option 2

Figure 5.3: Two options for a parallel membership test in TCAM when there is no false positive and k is 2

locates the target filter by matching 1s in one clock cycle. In the second option (Figure 5.3b), we set all the positions of the input register that correspond to H to 1, and set all other positions to 0. When we issue the search, the TCAM locates the filter whose positions that correspond to H have the don't-care value. Finally, we retrieve the next-hop information mapped to the matching filter to continue the packet processing.

Design implications. The first difference between the two options is how filters are represented in the TCAM. In the second option, all 1s within standard filters need to be changed to don't-cares. The second difference is about the number of writes to the input register bits. Assume the input register has the default value of 0, the second option requires setting only k positions in the input register while the first option needs to modify all w bits to 1s or *s. Although the power to toggle the memory and input register bits can be very small in practice, one can benefit from one of the options to perform hardware-specific optimizations for write-intensive workloads. Note that the encoding options do not change the false positive

rate or incur false negatives. Also, unlike IP routers that keep address prefixes sorted in decreasing prefix-length order to implement the longest prefix matching algorithm, the order of entries does not matter Caesar, except in uncommon cases where there are matches of multiple entries due to false positives. In this case, there is no unique ordering with which we can deterministically mask all false positives, so the backup forwarding path is triggered (Section 5.4).

5.3.4 Reducing Next-Hop Fast Memory

Because routers usually have a limited number of (next-hop AD, outgoing port) pairs, often many filters with different location properties are mapped to the same next-hop information (*e.g.,* AD_8 and AD_1 in Figure 5.2). We can eliminate the memory redundancy in storing next-hop information and make Caesar's data path more agnostic to the address length. At a cost of an extra fast memory access per lookup, we can store all different next-hop information into a separate fast memory space (*i.e.,* SRAM C in Figure 5.1), and then map each filter to a pointer (NH) pointing that memory. Each NH-pointer can be realized by using only one byte in most cases because of the limited number of ports on a router. This approach minimizes the fast memory overhead, in particular when TCAM contains a large number of filters. A similar technique can be applied when other forwarding actions or policies are supported in addition to (next-hop AD, outgoing port) pairs for filters.

Primary path performance implication So far, we have encoded AD addresses into the primary path such that the next-hop information can be retrieved in at most three memory access, each taking about $4ns$ delay based on Table 1. With using faster TCAMs [15], the primary path can support up to 1.6 billion filter searches per second under typical operating conditions

5.4 Backup Forwarding Path

We now describe the backup forwarding path and introduce a blacklisting mechanism to handle false positives as shown in Figure 5.1.

5.4.1 High-Speed False Positive Detection

The grouping properties lead to disjoint filters in the presence of different forwarding policies in a Caesar router. Therefore, we expect one matching filter in each parallel lookup and thus do not need to deal with the multi-match uncertainty problem (Section 5.3.1). Caesar deterministically interprets any multiple matching filters as an event that indicates the primary path is no longer reliable, and the processed packet might be forwarded to an incorrect next hop. Caesar intelligently detects such events using *Multi-Match (MM) line* of state of the art TCAMs, a flag indicating that there are multiple matching entries. In every lookup, the primary path is used if and only if the MM line output is low (see Figure 5.1). We describe the details as follows.

- The low MM line ensures that the index of the matching entry reported by the TCAM is the (only) correct filter. In other words, the destination address of the incoming packet is encoded in the TCAM without ambiguity. Therefore, the packet is processed based on the correct next-hop pointer ($NH_{primary}$) through the primary path.

- If the MM line is high, the true matching filter is at the i^{th} position and there is at least a filter at position $j \neq i$ that has returned false positive. If $j < i$, the reported index is not correct, otherwise the error is masked by the true matching filter that has higher priority (lower address). However, distinguishing between these two cases is not possible, so the backup forwarding path is triggered.

5.4.2 Blacklisting Mechanism

The backup forwarding path delays *at most* the first packet in a flow that is destined for an *AD* on which the primary path encounters multiple matches. There are two components in the backup path (see Figure 5.1):

- The *Blacklist Memory* is a very small, high-speed, and SRAM-based hash table that maps the hash value of an *AD* to its correct next-hop pointer (*NH*). In addition, each entry has an *expiration or idle* time that helps keep the hash table small and minimize potential collisions. An entry is deleted from the blacklist memory if it is not used for a predetermined period of time.

- The *False Positive Resolver (FPR)* is a component that creates entries in the Blacklist memory. It accesses the RIB that is stored in a hash table in slow memory, and retrieves the correct next-hop information, *i.e.,* all (next-hop *AD*, port) pairs, in constant time.

Backup path. Given the above components, the backup path works as follows (Figure 5.1). In parallel to the primary path, the backup path proactively retrieves the next-hop pointer NH_{backup} from the blacklist memory for every incoming address AD_d. If the primary path activates the MM line, *Chooser* picks NH_{backup} from the backup path, otherwise it selects $NH_{primary}$ from the primary path. If the MM line is high and NH_{backup} does not exist in Blacklist, the backup path delays forwarding and waits for *FPR* to retrieve the mapping from the RIB. *FPR* then updates *Blacklist* to avoid delaying subsequent packets belonging the same flow as well as future flows to the same destination AD. Finally, the NH_{backup} is sent to *Chooser* in this case.

Backup path performance implication. The backup path is as performant as the primary path almost always for three reasons. First, for optimally configured filters, the backup path result is rarely used because the multi-match rate is very small in theory and for actual workloads. We show this by analysis and evaluating two extreme cases of blacklisting (Section 5.6). Second, when a multi-match rarely occurs, the results of the backup and primary

paths are ready at the same time since the Blacklist memory mostly hits. Third, a Blacklist miss occurs for the first packet of a flow in the worst case (when the idle time is minimum). In such a case, the processing takes more time due to accessing slow memory. Caesar can minimize this by employing two known techniques. Given only next-hop pointers are needed to be retrieved, we can implement an efficient and summarized RIB to minimize the delay to one slow memory access (about 20*ns*). Also, we can minimize the miss rate by establishing a multi-level Blacklist approach similar to hierarchical caching schemes.

Security implication. From the security perspective, it is difficult for an attacker to trigger the backup path and to infer what ADs use this path for two reasons. The inference of k hash functions and filter organization, changing over time, is very hard. Second, the *observed delay* caused by the slow memory access is very small and happens infrequently.

5.5 Forwarding Optimizations

The parallel paths can process almost all packets within three fast memory accesses and offer deterministic correctness. However, the hash computation must be optimized to guarantee the entire performance. Although there are solutions for building uniform hashing (*e.g.,* [103]), the computation overhead is still an unsolved issue [82]. Related designs (*e.g.,* [128, 66, 77, 71]) have not taken into account this overhead. Our key idea is to exponentially minimize the number of hash computations, and then run them in parallel similar to state-of-the-art routers (Section 5.5.1). Caesar also handles route updates and optimizes the filter utilization (Section 5.5.2).

5.5.1 Scalable Hash Computation

We leverage a simple but effective technique to reduce the number of hash computations. Our approach is based on hash coding theory [134] with its basic property. If we have two different and uniformly distributed hash values $f(x)$ and $g(x)$ for input key x, we can construct hash value $h(x) = f(x) \oplus g(x)$ that is also from a uniform distribution. The main intuition is

that XOR is a uniform operation that generates both 0 and 1 with the same probability for random inputs (*i.e.,* 0 and 1 are equally likely to appear). This property is also applied to n-bit inputs in practice. For example, SSL computes the MD5 and SHA-1 of its inputs and combines them to avoid cryptanalytic attacks.

Hierarchical hash computation. Caesar recursively employs the property to faster generate hash values in the lookup and update procedure of *small* filters (*e.g.,* 288-bit), which have specific characteristics compared to large filters (will be clarified below). Given k_1 different hash values from uniform distributions, $H = \{h_1, h_2, h_3, \cdots h_{k_1}\}$, any non-empty subset of H is a candidate for constructing a new uniform hash value by performing the XOR operation among its members. Because H has $2^{k_1} - 1$ non-empty subsets, we can build $2^{k_1} - k_1 - 1$ new uniform hash values. This dramatically improves the hash computation performance. For instance, four different uniform hash values in $H = \{h_1, h_2, h_3, h_4\}$ give us $G = \{h_1 \oplus h_2, h_1 \oplus h_3, h_1 \oplus h_4, \ldots h_1 \oplus h_2 \oplus h_3 \oplus h_4\}$ that consists of 11 uniform hash values by performing only 11 XOR operations.

Internal correlation. Theoretically, the correlation between recursively-constructed hash values does not lead to more false positives as long as the seed set satisfies the uniformity and diversity requirements. In practice, even cryptographic hash functions might not completely satisfy the uniformity. In this case, we have observed negligible (positive and negative) difference values between the flat and hierarchical hash computation schemes in terms of false positive and multi-match rate, making this scheme practically useful.

Small filters and internal collision. Caesar's focus is on very small filters, which is different from a similar usage of the core property in previous work [117]. Particularly, our results show small filters require k hash values of an address to be different in contrast to large filters. However, k hash functions might generate fewer than k different hash values in practice. Therefore, we proactively compute sufficient extra hash values for each address, which we refer it as internal collision avoidance. In this case, we have observed the hierarchical scheme has substantially lower computational overhead compared to the standard flat scheme

(complete details in Section 5.6.2.4).

5.5.2 Optimized Route Update Support

To handle control plane messages that change forwarding states, Caesar should be able to remove any address from an old filter, and insert it into a new filter. However, a standard filter does not support graceful address removal.

We leverage counting filters [72] to realize this operation. In a counting filter, each position is extended from a single-bit (as in a standard filter) to an s-bit counter. To insert/remove an address, the value of each of the k positions, each corresponding to a hash value of the address, is incremented/decremented instead of being set/unset. Similarly, the membership test checks the positions to see if all of them have non-zero values or not. To integrate counting filters into Caesar, a trivial solution is to directly put them into TCAM, but this increases the TCAM memory usage as well as the complexity of the parallel lookups. Instead, Caesar keeps a counting filter in slow memory for each standard filter in TCAM. Caesar does not access the counting filters to perform forwarding, but only uses them to assist updating standard filters.

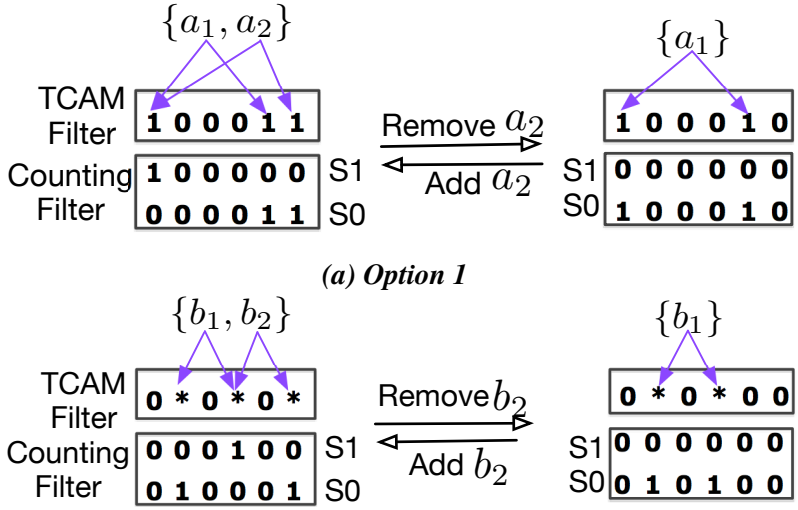


Figure 5.4: Address removal and insertion in filters when k ***and*** n_{max} ***are 2, and thus*** $s = \lfloor \log_2(n_{max}) \rfloor + 1 = 2$

Insertion and zero overflow. To store an address in the data path, Caesar first learns

its group based on the primary properties (Section 5.3.1). Then it determines the insertion position of the address in the TCAM based on its group. If the specified group is old, multiple filters already have been assigned to old members of the group in the TCAM. In this case, Caesar balances the load among existing filters of the group with a simple greedy approach. It selects an available position or filter with the minimum utilization ratio, n_i/n_{max} (n_i is the number addresses in the filter). Otherwise, Caesar assigns a new position to the group and the address. The specified position is also recorded in the RIB to support removing the address without false negatives later.

After determining the position, we first insert the address into the corresponding counting filter and then the standard filter in a simple procedure (as shown in Figure 5.4). When a counter changes from zero to non-zero, we change the corresponding bit in the standard filter from zero to one for option 1 (Figure 5.4a) and from zero to the don't care (*) for option 2 (Figure 5.4b). Unlike the previous usage of counting filters (*e.g.,* [72]), our counting filters can be configured to avoid overflows by setting $s = \lfloor \log_2(n_{max}) \rfloor + 1$ because each standard filter contains at most n_{max} addresses (Section 5.3.3).

Removal and high filter utilization. To remove an address, we first retrieve the its position (in TCAM) from the RIB, which is necessary to avoid generating false negatives. As shown in Figure 5.4, we then remove the address from the counting filter at the position by decrementing the counters, each of which maps to a hash value of the address. If any of the counters becomes zero, we set the corresponding bit in the standard filter to zero in the both options. In the address removal procedure, Caesar checks the utilization ratio of the affected standard filter. If the ratio is below a predetermined threshold, Caesar tries to combine the filter with other filters allocated to the same group. This is necessary to reduce the number of filters in the TCAM (We adjust the counting filters accordingly).

5.6 Evaluation

In this section, we first perform a cost-accuracy analysis (Section 5.6.1) and then do extensive simulations using multiple workloads (Section 5.6.2) to study Caesar from different aspects.

5.6.1 Cost-Accuracy Analysis

We provide a simple *approximation* showing the inherent trade-off between false positives of filters in the primary path and the total cost of Caesar. Note Caesar correctly processes all packets and false positives only affect the amount of traffic handled in the backup path. Similar to AIP and XIA [54], we assume each AD corresponds to an IP prefix in today's Internet but with *larger* size. The current number of active address prefixes is about 481k and we envision 1M *ADs* to accommodate a reasonable growth rate [42].

False positive estimation. Intuitively, the false positive rate in the parallel filter lookup procedure for a destination *AD* address depends on two factors. First, the position of the true filter containing the address. Second, the fill factor (*i.e.,* the ratio of bits with the value 1) of all the filters above the true filter. The fill factor of filter i is a function of the number of inserted addresses in the filter (n_i), the width of entries (w), and the number of hash functions (k). Caesar does not insert more than n_{max} addresses in each filter, so we can derive a theoretical upper bound on the *maximum false positive rate (FP)* of filter i, given n_i and w are fixed, by $FP(i) \leqslant \left(1 - e^{\frac{-kn_{max}}{w}}\right)^k$. In this equation, k can be computed optimally for given n_{max} and w, $k_{opt} = 9w/13n_{max}$. We omit the detailed derivation for simplicity. Assuming all addresses are accommodated into E entries and a given packet matches any entry with the probability of $1/E$, *the maximum expected false positive rate* in parallel filter lookups can be calculated by Eq. 5.1.

$$\mathbb{E}[\text{false positive rate}] \leqslant \frac{(E-1)}{2}\left(1 - e^{\frac{-kn_{max}}{w}}\right)^k \tag{5.1}$$

The same approach can be used to derive *the maximum expected multi-match rate* in parallel

filter lookups, which results in Eq. 5.2.

$$\mathbb{E}[\text{multi-match rate}] \leqslant (E-1)\left(1-e^{\frac{-kn_{max}}{w}}\right)^{k} \tag{5.2}$$

Table 1: Fast memory reference price

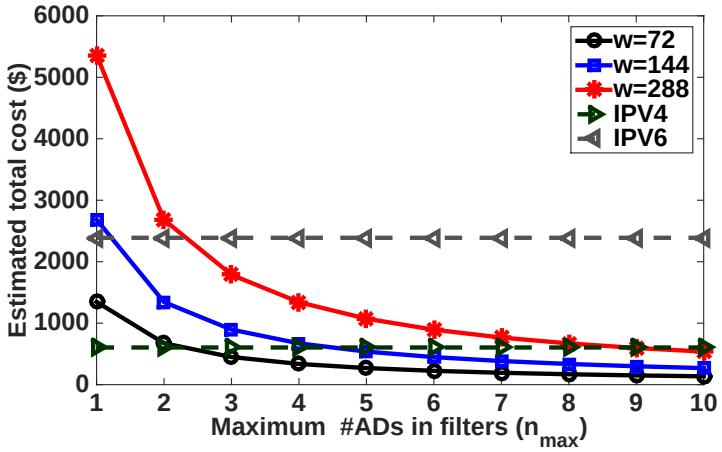
Memory	Capacity	Delay	Price	Company	Cost/ MB
SRAM	9MB	3-4ns	$90	Cypress	$10
TCAM	2.5MB	3-4ns	$390	Broadcom	$156

Cost estimation. Now, we turn into estimating the total material cost. Our prices have been quoted by Cypress and Broadcom for a TCAM and an SRAM working at 250 MHz as shown in Table 1. As expected, the TCAM is more expensive compared to the SRAM. Let C_{TCAM} and C_{SRAM} denote the cost-per-bit of TCAM and SRAM respectively. Assume Caesar has h next-hop pairs and addresses are q-bit long. Then, Eq. 5.3 gives an estimation of the total cost of a Caesar router excluding the blacklist memory which is expected to be small, while also ignoring the RIB and counting filters which use inexpensive DRAM. The first term is the TCAM cost, and the second term includes the cost of SRAM A and C (see Figure 5.1).

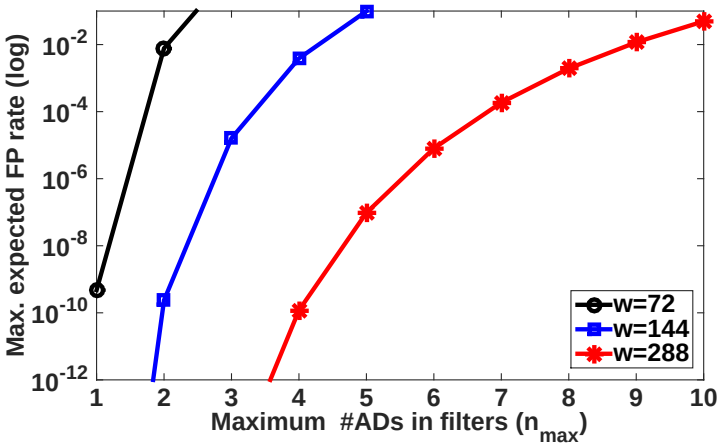
$$Total\,Cost = EwC_{TCAM} + (log(h)E + hq)\,C_{SRAM} \tag{5.3}$$

Based on Table 1, Figure 5.5a illustrates the total cost of Caesar for $h = 64$, $q = 1kb$, and variable n_{max} and w. We assume filters are fully utilized (*i.e.,* $E = \#ADs/n_{max}$). We also estimate the total costs of optimized TCAM-based IPv4 and IPv6 routers (*e.g.,* [129]) to be $604 and $2,389 respectively. For the same parameters and the optimal k values, Fig 5.5b depicts the maximum expected false positive rate in parallel filter lookups.

Finding 1. Caesar can be substantially less expensive compared to TCAM-based IPv6 routers. We observe there are several interesting options that provide a reasonable accuracy for the filters with a feasible total cost. For $n_{max} = 4$ and $w = 288$, the maximum expected false positive rate is around 10^{-10} and the total cost is $\$1,340$. In this case, Caesar is about 43% less expensive than the IPv6 router while our addresses are 8X longer than the



(a) Estimated cost for one million 1kb addresses; IP addresses are 32b and 128b.



(b) Expected false positive rate in parallel lookups; k is optimal.

Figure 5.5: Cost-accuracy analysis

IPv6 addresses.

Finding 2. The total cost of Caesar is constant for very long addresses. To analyze the sensitivity of our design to the AD address length, we compare the cost of Caesar router with TCAM-based IP router. Fig 5.6 shows that the total cost of our design is roughly constant even for very long addresses. In contrast, the total cost of IP routers increases linearly as addresses become longer (assuming future IP addresses can be longer to support new services).

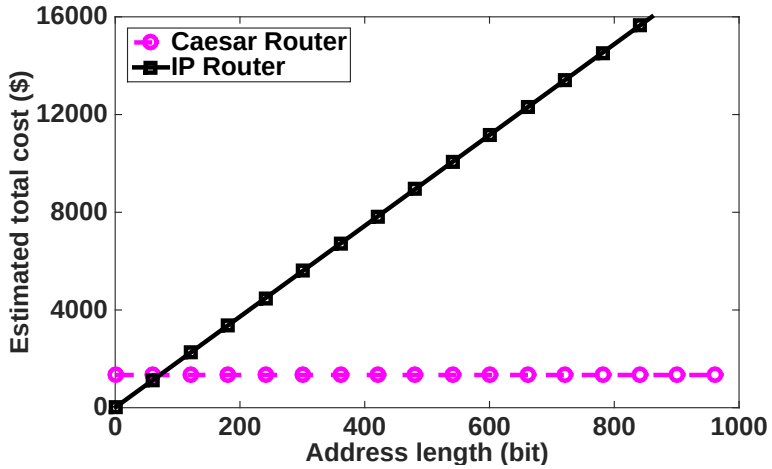


Figure 5.6: Address length vs. total cost of TCAM-based IP routers and Caesar routers with $w = 288$, $n_{max} = 4$

5.6.2 Extensive Trace-Driven Simulation

We now evaluate Caesar under real workloads. Because existing prototyping platforms such as NetFPGA lack sufficient TCAM, we implement an accurate packet-level simulation framework in C/C++ (with 2500 LoCs) and "mix and match" various datasets. We measure the multi-match rate, percentage of delayed flows, effects of grouping properties on the memory utilization, efficiency of the hierarchical hash computation scheme, and energy consumption.

Table 2: Experiment statistics

Experiment /Snapshot	Total # ADs	# forwarded packet	Monitoring duration(s)
Jan 1, 2008	149842	12481172	25
Jan 1, 2009	162102	12849066	31
Jan 1, 2010	180470	16516240	41
Jan 1, 2011	205361	25459705	52
Jan 1, 2012	234483	28460868	55
Jan 1, 2013	255424	16662799	42
Total		**112 M**	**246s**

5.6.2.1 Dataset and Methodology

We simulate the future Internet architecture using public datasets in six snapshots between 2008 and 2013 to consider the growth rate of IADs and ADs. In our simulation, IAD and

AD correspond to today's AS and IP prefix respectively. We select only address prefixes of length 24 to feed experiments with flat addresses. To consider high entropy of future addresses, we replace them with their corresponding hash value, which is computed using the SHA1 algorithm. To represent the business agreements among IADs, we utilize CAIDA inferred relations between ASes [6] and leverage RIBs and update traces of Route Views [40] to generate the FIBs based on the path length metric. We replay the packets in traffic traces collected from backbone links [6] and use a recent power model [51] to measure the dynamic power consumptions in the experiments. For each pair of n_{max} and w, we compute the optimal k and then construct filters based on the primary properties. For the space reason, we show the results for a single Caesar at the border of IAD 7726 when w is 144 and n_{max} is between 2 and 6. The other Caesar routers and other settings follow a similar trend. Table 2 lists the number of ADs and forwarded packets by this Caesar router, and the duration of traces in each snapshot. Although the monitoring duration is relatively short, the coverage of destination addresses is high and sufficient for our evaluation purpose. In total, the Caesar router forwards 112M packets, and upon receiving each route update message, it runs the best route selection procedure and updates filters if necessary. The average rate of route update messages varies between 88.2 and 277.1 across different snapshots.

Table 3: Multi-match rate and TCAM memory consumption for $w = 144$ ***and variable*** n_{max}**.**

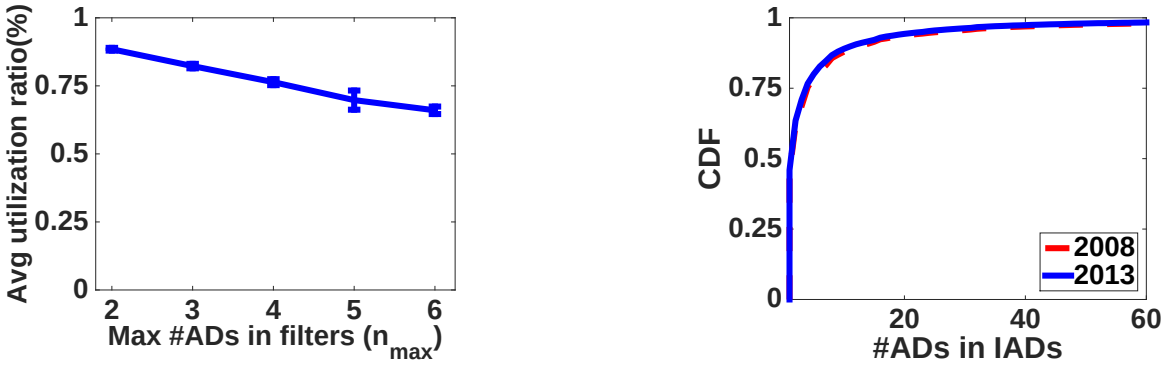
n_{max}	$k_{optimal}$	k_{Caesar}	Multi Match Rate(%)[w=144]						TCAM Memory Footprint(MB)[w=144]					
			2008	2009	2010	2011	2012	2013	2008	2009	2010	2011	2012	2013
2	49	6	0	0	0	0	0	0	1.46	1.58	1.75	1.98	2.25	2.46
3	33	6	0.0002	0	0.0001	0	0.0001	0.0003	1.05	1.13	1.26	1.42	1.6	1.76
4	24	5	0.033	0.05	0.032	0.035	0.032	0.001	0.85	0.92	1.02	1.14	1.29	1.41
5	19	5	1.84	1.35	4.47	6.06	6.78	9.34	0.73	0.79	0.88	0.98	1.1	1.2
6	16	5	14.69	14.22	18.71	25.68	29.94	38.82	0.66	0.71	0.79	0.88	0.98	1.08

5.6.2.2 Multi-Match Rate and Memory Consumption

We first measure the multi-match rate in the primary path. This rate indicates the amount of traffic that is forwarded by the backup path regardless of whether it is delayed by a slow memory access to the RIB due to the blacklist memory miss or it is delivered without any delay. Table 3 presents the multi-match results and the memory usage of filters in 30

configurations.

Finding 3. Caesar can forward most of the traffic through the primary path within three fast memory accesses. As expected, Table 3 shows the multi-match rate in each snapshot exponentially increases as n_{max} increases. Although predicting the exact multi-match rate is impossible, we observe different snapshots have the same order of magnitude of the multi-match rate for a fixed n_{max}. This indicates we can practically control the order of magnitude of the multi-match rate in Caesar routers. In the case that n_{max} is 2, we observe that the multi-match rate is zero thus the MM line never goes high. This means all the packets are forwarded through the primary path, mainly due to using more hash functions.



(a) *(b)*

Figure 5.7: Determining n_{max}. (a) Average filter utilization ratio of filters for $w = 144$ across all snapshots. (b) Distribution of ADs in IADs in the first and last snapshots.

Finding 4. To find a reasonable n_{max}, both the memory utilization and multi-match rate are determining factors. Based on Table 3, two important reasons suggest that we should keep n_{max} smaller than 5 for $w = 144$. First, we observe the multi-match rate increases several orders of magnitude, from -2 to 0, when n_{max} changes from 4 to 5. Second, the memory utilization rate (*i.e.,* the difference between the memory footprint of two consecutive n_{max} values) becomes smaller as n_{max} increases. Now the question is why the memory utilization rate decreases. Fig 5.7b shows the distribution of ADs in IADs between five years. Also, Fig 5.7a illustrates the average utilization of filters in TCAM, which is defined as $\sum_{i=1}^{E} n_i/(n_{max}E)$, across all snapshots for each n_{max} value. From these two figures, we observe the memory utilization rate reduces because the average utilization of filters goes

below 75% for n_{max} values larger than 4 that is because about 77% of IADs own fewer than 5 ADs.

5.6.2.3 Energy Consumption Breakdown

We now measure the total dynamic energy consumption of Caesar and compare it to optimized TCAM-based IPv4 and IPv6 solutions (*e.g.,* [129]). TCAM is the most power consuming component among different memory technologies in routers by several orders of magnitude. Therefore, we can ignore the energy consumptions in the other memory components of Caesar. The dynamic energy used in each search operation depends on many factors and parameters but is used in three high level architectural components [51]:

- **Match lines** that are charged in every lookup, and then except the lines that match input address, the others are discharged. The energy for this operation is proportional to the sum of match lines capacitance (*i.e.,* the ability to store charge).

- **Select lines** that are driven to allow the comparison between input address and entries to happen. The energy used to drive select lines usually increases as the size of TCAM increases.

- **Priority encoder** that needs some power to work and the required energy depends on the number of filters (E), and is independent of the width of filters (w).

Finding 5. Caesar consumes 67% less total energy compared to TCAM-based IPV6 routers while the addresses are substantially longer. Fig 5.8 illustrates the total energy consumption break down across the TCAM components for the snapshot 2012 (others have similar results) for $w = 144$, variable n_{max}, and $65nm$ CMOS technology. We repeat the similar experiments for the IP routers. We observe the total energy consumption of Caesar for $n_{max} = 4$ is only 1% higher than the IPV4 router and 67% less than the IPV6 router while addresses in Caesar are very longer.

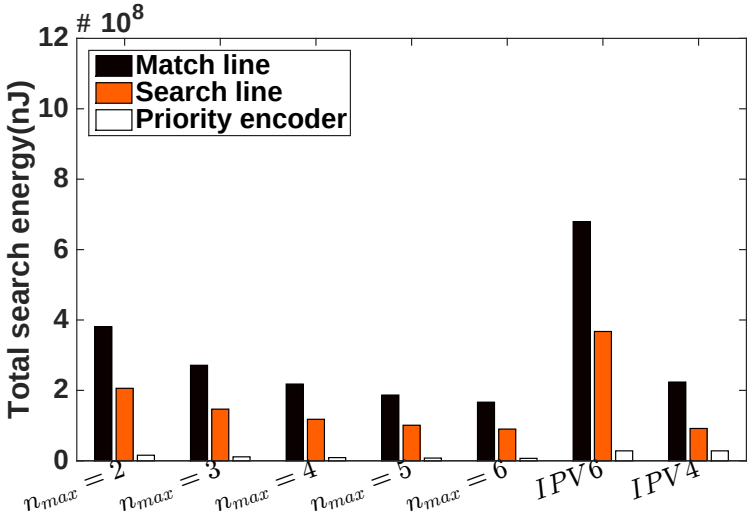


Figure 5.8: Total search energy breakdown for $w = 144$.

Table 4: Effects of permanent and per-flow blacklisting approaches

n_{max}	#Delayed Flows[Per-flow Blacklisting, w=144]						#Delayed Flows[Permanent Blacklisting, w=144]					
	2008	2009	2010	2011	2012	2013	2008	2009	2010	2011	2012	2013
2	0	0	0	0	0	0	0	0	0	0	0	0
3	3	0	23	0	2	1	1	0	2	0	1	1
4	923	903	764	1371	885	2067	141	30	145	261	245	115
5	32209	20282	25670	45888	110984	47815	3359	628	3096	6303	6040	2978
6	271902	175575	284606	656360	457918	308102	21366	4559	28789	47024	50039	21231

5.6.2.4 Hierarchical Hash Computation Scalability

We now compare the hierarchical and flat hash computation schemes from two aspects.

Finding 6. The hierarchical scheme needs smaller number of hash computations for handling internal collisions. Although filters use k different hash values to insert and test an address, k hash functions might generate fewer than k different hash values in practice, which we call it internal collisions. In small filters, we have observed such collisions can increase the multi-match rate by up 50%. This is because the number of buckets in small filters is limited, and the correlation among the k hash values computed for a given address is stronger in practice. To control internal collisions, we proactively compute extra hash values for each address in the insertion and test procedures. In this way, we obtain sufficient different hash values in both the flat and hierarchical schemes. We measure the computational overhead of the two schemes in terms of the number of hash computations and aggregate the results across all snapshots and configurations. For a given k, the hierarchical scheme

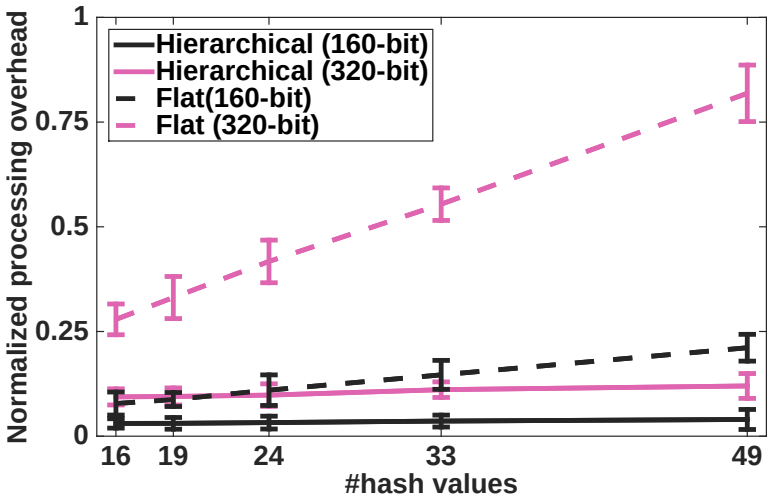


Figure 5.9: Normalized processing overhead of the hierarchical and flat schemes

requires at most $1.16\log(k)$ hash computations while the flat scheme needs at most $1.5k$ hash computations to generate k different values. This indicates our scheme performs well for small filters because an extra different hash value in the seed set can double the total number of different hash values.

Finding 7. The hierarchical scheme substantially reduces hash computation processing overhead. We also measure the computational overhead of the two schemes in terms of CPU time when multi-threading is enabled for the same number of threads. We plot the aggregate results in Fig 5.9. To study the effect of address length, we consider 160-bit and 320-bit AD addresses. For a fair comparison, we do not enable internal collision avoidance that generates extra hash values. For a fixed k, we observe our scheme in average incurs by up to 18% and 70% smaller processing overheads for 160-bit and 320-bit addresses compared to the flat scheme. Note the overall number of XOR operations in the hierarchical scheme is $2^{k_{Caesar}} - k_{Caesar} - 1$.

5.6.2.5 Blacklisting and backup path delay

Finally, we evaluate the blacklisting in two extreme cases: *per-flow* and *permanent*. Due to blacklisting (Section 5.4.2), only a small amount of packets activating multi-match line and going through the backup path are delayed. In the per-flow case, we store the destination *AD*

address of a flow that leads to a multi-match until the flow completes. In the permanent case, we permanently store *AD* addresses that lead to a multi-match, upon the first detection.

Finding 8. Very few flows are delayed by both the permanent and per-flow blacklisting schemes. Table 4 shows the difference between these two cases. The permanent case reduces the number of delayed flows by an order of magnitude compared with the per-flow case. For example, for $n_{max} = 4$, fewer than 261 flows are delayed by the permanent blacklisting while the per-flow blacklisting approach delays up to 17.93X more flows. Note when a flow is delayed in both approaches, except its first packet, the other packets are forwarded at high speed. In both approaches, the blacklist memory footprint is insignificant because each next-hop pointer is only one-byte.

5.7 Related Work

In the past few years, filters have been used in designing routing and forwarding engines. These efforts mostly have targeted improving the performance and simplifying traffic processing in enterprise switches and routers.

Much of the previous work focuses on memory-efficient IP routers [66, 117]. Unlike Caesar, these designs optimize the longest prefix matching algorithm to minimize the fast memory consumption. In particular, they store all address prefixes of the same length into a very wide filter. Given there can be multiple matches and the lookup uncertainty problem, these systems mostly test all candidates against a very large hash table located on slow memory to find the length of the longest match. Due to limited performance, these approaches cannot be used in high-speed border routers. Also, their coarse-grained filter construction is not reliable in practice.

Some other work focuses on designing low-cost and scalable switches handling flat addresses in small-scale and single-domain enterprise networks [77, 128]. The primary technique in such designs is constructing a very large filter per outgoing interface, each containing flat addresses reachable from the port. Upon facing the multi-match uncertainty

problem, these designs mostly randomly choose among matching candidate filters, and thus impose significant delays and path inflations. Also, they require many memory accesses per lookup, and therefore have limited peak performance.

In contrast to the above techniques, Caesar is designed for high-speed border routers in the future Internet. Caesar constructs fine-grained filters that are more reliable and scalable. Caesar does not need to compute separate hash values for accessing each filter. It tests all the filters in parallel in one clock cycle and does not waste many memory accesses to check all the matching filters as it can detect false positives at high speed using a hardware flag. Caesar minimizes hash computation overheads by recursively combining hash values.

Our idea of using filters in TCAM entries is similar to previous work [75]. However, the authors focus on the case that input register is filled by a set of elements instead of one to solve multiple string matching and virus detection problems. Although the authors propose a theoretical upper bound on the maximum false positive rate, still they do not provide any mechanism for detecting false positives at high speed. In contrast to this work, we reduce the hash computation overhead, design parallel forwarding paths, cleanly detect false positives, manage memory entries, and design an element removal procedure.

To optimize hash table operations in network processors, prior work [116] employs counting filter per table bucket. Instead, we use an expiration timer per address to minimize the size of the Blacklist memory and avoid occasional collisions. We can improve the robustness of the backup path by benefiting from such techniques.

Our idea of using small counting filters to support route changes is similar to some proposals (*e.g.,* [128, 66, 72]). In contrast to existing approaches, Caesar constructs equal-sized filters in terms of bit width and the maximum number of constituent members. We dynamically allocate counting and standard filters while maintaining them highly utilized. Also, our counting filters never experience overflow, and we do not modify the filters in the critical forwarding path during updates.

CHAPTER VI

Concluding Remarks

In this dissertation, we discussed three novel network architectures, SkyCore, SoftBox, and SoftMoW for 5G core networks to meet emerging 5G use cases and address key limitations of 4G core networks (EPC). These architectures provide scalability, performance, and flexibility as the first-order properties by leveraging recent paradigm shifts in networking (SDN, NFV, and MEC). Then, we architected Caesar that is a high-speed and memory-efficient router architecture to be deployed as a complementary solution in our 5G core proposals or other future Internet architectures to improve their mobility support and verifiable security. In the following, we conclude our contributions in each of these works and further highlight the connection among these works.

In the second chapter, we presented the design and real-world implementation of SkyCore that is a core network architecture for on-demand airborne 5G networks. These networks are expected to be deployed in challenging environments (e.g., natural disasters) to interconnect first responders and the general public. Given the fundamental limitations in deploying EPC (4G core) on the ground to support a multi-UAV RAN, we advocated for a radical, yet standards-compliant re-design of the EPC, namely the edge-EPC architecture, to suit the UAV environment. SkyCore embodied the edge-EPC architecture, while introducing two key pillars in its design to address the associated challenges – a complete software refactoring of the EPC for compute-efficient deployment on a UAV, and a new inter-EPC

communication interface to enable fully functional operation in a multi-UAV environment. SkyCore's design focused on the challenges unique to multi-UAV LTE networks that typically span from a few to at most tens of UAVs (city-scale). While our design decisions (e.g., inter-agent proactive updates, policy pre-computation) are efficient and scalable for our target environment, they are not designed to scale in nation-wide LTE networks with hundreds of millions of UEs. There is one major future work for SkyCore. We did not discuss the design of the backhaul agents forming the physical wireless mesh network among UAVs. The design of an efficient backhaul needs to be jointly optimized with the RAN as the position of the UAV simultaneously affects the performance of the backhaul as well as the access to the UEs.

In the third chapter, we proposed SoftBox, a novel architecture for 5G cellular core networks that enables customized, low latency, and signaling-efficient services on a per-UE basis. Compared to SkyCore that is an airborne citywide solution, SoftBox handles terrestrial RANs in larger geographical regions (e.g., multiple provinces or states). In particular, SoftBox consolidates the policies associated with each UE into a container in its proximity. SoftBox has been designed to be incrementally deployable and scale to a large number of UEs, with special attention to efficient schemes for further minimizing the resource usage of UE containers, the migration costs of UE containers, data plane forwarding states, and costs of signaling communications between the SoftBox core and LTE RANs. In particular, our optimized version of SoftBox is equipped with self-optimizing UE containers, enhanced segment routing protocol, mobility-aware container migration schemes, and fast protocols for communication with RANs. Our results are promising and point to the practical feasibility and potential of the SoftBox concept.

In the forth chapter, we discussed SoftMoW, a scalable cellular WAN architecture that is based on effective recursive and reconfigurable abstractions for both control plane and data plane. Compared to SoftBox and SkyCore that operate at the city-level and state-level scales, SoftMoW is optimized to for nationwide 4G WANs consisting of multiple core networks

(*e.g.,* SoftBox and SkyCore networks) and hundreds of millions of UEs. We designed a recursive link discovery protocol and virtual fabrics to allow automatic topology construction and support global resource management. SoftMoW optimizes network-wide objectives such as inter-region handover, path implementation, and routing. SoftMoW achieves these goals using novel algorithms benefiting from our scalable abstractions. Our evaluation results show that SoftMoW is very efficient and scalable. For future work, one may want to deploy SoftMoW in a large testbed.

Finally, we argued that many 5G core and future Internet architectures either advocate for or benefit from replacing the IP addressing scheme with one that has two features: (1) decouples each address from its owner's network location and (2) permits its owner to cryptographically prove its ownership of the address. The separation feature enables these architectures to improve mobility support and multi-homing at the network layer. However, such an addressing scheme requires addresses to be substantially long. To cope with this challenge, we proposed Caesar that is a practical solution for high-speed routers of next-generation networks using this modern addressing scheme. We designed scalable and reliable filters to compress long addresses. Due to the poor performance of storing filters into SRAMs, we designed a forwarding engine to search all the filters in parallel. To avoid forwarding loops and black holes, the engine uses two forwarding paths. We offered a novel blacklisting mechanism for accelerating the performance of the backup path. Caesar supports routing updates and performs intelligent memory management by utilizing counting filters in slow memory. For minimizing the computational overhead of hash functions, we proposed a hierarchical hash computation scheme for our small filters. Our evaluation results indicate our design is memory-efficient, energy-efficient, and high-performance in practice.

Bibliography

[1] 3GPP standard. http://www.3gpp.org/specifications.

[2] 5G systems. https://goo.gl/hrQec7.

[3] 5G vision. https://goo.gl/frAZxN.

[4] Amazon UAVs Charging. https://goo.gl/x66KXF.

[5] ATT-5G roadmap. http://goo.gl/eA0wSu.

[6] CAIDA Datasets. http://goo.gl/kcHlQf.

[7] CBRS Spectrum. https://goo.gl/3zbYyo.

[8] Cell on Wing (CoW): AT&T. http://about.att.com/innovationblog/cows_fly.

[9] Cell on Wing (CoW): Puerto Rico. http://about.att.com/inside_connections_blog/flying_cow_puertori.

[10] Cell on Wing (CoW): Verizon. https://goo.gl/q9YjNv.

[11] Cisco Routers. http://goo.gl/6CWpLA.

[12] Docker. https://goo.gl/hxsJZc.

[13] EPC-in-a-box. https://goo.gl/AdpKBU.

[14] Ericsson's Signaling Storm Analysis. https://goo.gl/XWyFWF.

[15] Esilicon TCAMs. http://goo.gl/O3rloQ.

[16] Facebook project aquila. https://goo.gl/gHYVa7.

[17] Floodlight. http://goo.gl/eXUprV.

[18] Google Kubernetes. https://kubernetes.io/.

[19] Google x: Project loon. https://goo.gl/skSz1z.

[20] Huawei-5G systems. https://goo.gl/dpYvW5.

[21] Internet Architecture Board. http://goo.gl/cnMyY9.

[22] iplane dataset. http://goo.gl/JZWdK2.

[23] Lagopus: SDN switch. http://www.lagopus.org/.

[24] LTE deployment and design strategies. https://goo.gl/wZXhov.

[25] LTE design and deployment. http://goo.gl/DMKymH.

[26] LTE growth study. https://www.ericsson.com/res/docs/2016/ericsson-mobility-report-2016.pdf.

[27] LTE signaling storm. http://goo.gl/qk6Bp9.

[28] M-Cord: Mobile Cord. http://goo.gl/pZgSvg.

[29] Managing the signaling storm. http://goo.gl/lkTyb1.

[30] Migrating stateful containers. https://goo.gl/wqpUD5.

[31] nDPI: DPI functions. http://goo.gl/1Nc3QC.

[32] netfilter. http://www.netfilter.org/.

[33] OpenAirInterface. http://openairinterface.org.

[34] OpenEPC. http://www.openepc.com/.

[35] Oracle Signaling Storm Index. `https://goo.gl/6BZ8Fo`.

[36] Oracle Signaling Storm Index. https://goo.gl/2k0niu.

[37] PhantomNet: LTE testbed. https://www.phantomnet.org/.

[38] PhoneLab: smartphone testbed. https://phone-lab.org/.

[39] Reliable UDP (RUDP). https://goo.gl/F5Z3ls.

[40] Route Views Dataset. http://goo.gl/cn8sT6.

[41] RYU controller. https://goo.gl/TK7TSS.

[42] Scaling Issues. http://goo.gl/SAnq28.

[43] Segment Routing. http://www.segment-routing.net/.

[44] Supervisor: A Process Control System. http://supervisord.org/.

[45] TCAM Memory Challange. http://goo.gl/OGYyKn.

[46] The cgroup freezer. https://goo.gl/lHhaeD.

[47] View on 5G Architecture. https://goo.gl/EqAeao.

[48] Virtual Subscriber Gateway (vSG). http://goo.gl/ySflpc.

[49] Predictions. http://goo.gl/qmnVDy, 2012.

[50] MobilityFirst Project. http://goo.gl/sD64f9, 2014.

[51] B. Agrawal and T. Sherwood. Modeling TCAM Power for Next Generation Network Devices. In *Proc. IEEE ISPASS*, 2006.

[52] M. Akhbarizadeh, M. Nourani, and D. Vijayasarathi. A Nonredundant Ternary CAM Circuit for Network Search Engines. *IEEE Trans. VLSI*, 2006.

[53] M. Alicherry et al. Network aware resource allocation in distributed clouds. In *Proc. IEEE INFOCOM*, 2012.

[54] D. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet Protocol. In *Proc. ACM SIGCOMM*, 2008.

[55] B. Aoun, R. Boutaba, Y. Iraqi, and G. Kenward. Gateway placement optimization in wireless mesh networks with qos constraints. *IEEE Journal on Selected Areas in Communications*, 2006.

[56] H. Ballani, P. Francis, T. Cao, and J. Wang. Making Routers Last Longer with ViAggre. In *Proc. USENIX NSDI*, 2009.

[57] A. Banerjee et al. Scaling the LTE Control-Plane for Future Mobile Access. In *Proc. ACM CoNEXT*, 2015.

[58] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, et al. Onos: towards an open, distributed sdn os. In *HotSDN*, 2014.

[59] B. H. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. In *ACM CCR*, 1970.

[60] P. Bosshart et al. P4: Programming protocol-independent packet processors. *ACM CCR*, 2014.

[61] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a routing control platform. In *Proc. USENIX NSDI*, 2005.

[62] Z. Cai and et al. The preliminary design and implementation of the Maestro network control platform, 2008.

[63] Y. Chen, S. Jain, V. K. Adhikari, Z.-L. Zhang, and K. Xu. A first look at inter-data center traffic characteristics via yahoo! datasets. In *Proc. IEEE INFOCOM*, 2011.

[64] J. Cho et al. ACACIA: Context-aware Edge Computing for Continuous Interactive Applications. In *Proc. ACM CoNEXT*, 2016.

[65] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an overlay testbed for broad-coverage services. In *ACM CCR*, 2003.

[66] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor. Longest Prefix Matching Using Bloom Filters. In *Proc. ACM SIGCOMM*, 2003.

[67] A. Dhekne et al. Extending cell tower coverage through drones. In *HotMobile*, 2017.

[68] U. Drepper and I. Molnar. The native POSIX thread library for Linux. *White Paper, Red Hat Inc*, 2003.

[69] A. Ermolinskiy and S. Shenker. Reducing transient disconnectivity using anomaly-cognizant forwarding. In *Proc. Workshop on Hot Topics in Networks*, 2008.

[70] M. Ester et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, 1996.

[71] C. Esteve, F. L. Verdi, and M. F. Magalhães. Towards a new generation of information-oriented internetworking architectures. In *Proc. ACM CoNEXT*, 2008.

[72] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. *IEEE Trans. Networking*, 2000.

[73] W. Felter et al. An updated performance comparison of virtual machines and linux containers. In *IEEE ISPASS*, 2015.

[74] T. A. Forum, M. Ahmed, and J. H. Rus. Private network-network interface specification version 1.0 (pnni 1.0), 1996.

[75] A. Goel and P. Gupta. Small Subset Queries and Bloom Filters Using Ternary Associative Memories, with Applications. In *Proc. ACM SIGMETRICS*, 2010.

[76] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. 2011.

[77] B. GrtinvaU. Scalable Multicast Forwarding. In *ACM CCR*, 2002.

[78] A. Gudipati, D. Perry, L. E. Li, and S. Katti. SoftRAN: Software defined radio access network. In *HotSDN*, 2013.

[79] B. Han et al. Network function virtualization: Challenges and opportunities for innovations. *IEEE Comm. Magazine*, 2015.

[80] D. Han, A. Anand, F. R. Dogar, B. Li, et al. XIA: Efficient Support for Evolvable Internetworking. In *Proc. USENIX NSDI*, 2012.

[81] S. Hassas Yeganeh and Y. Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In *HotSDN*, 2012.

[82] C. Henke, C. Schmoll, and T. Zseby. Empirical evaluation of hash functions for multipoint measurements. In *ACM CCR*, 2008.

[83] C.-Y. Hong, S. Kandula, R. Mahajan, et al. Achieving high utilization with software-driven wan. In *ACM CCR*, 2013.

[84] P. Hunt, M. Konar, et al. ZooKeeper: Wait-free coordination for internet-scale systems. In *USENIX ATC*, 2010.

[85] S. Jain, A. Kumar, S. Mandal, et al. B4: Experience with a globally-deployed software defined wan. 2013.

[86] X. Jin et al. Softcell: Scalable and flexible cellular core network architecture. In *Proc. ACM CoNEXT*, 2013.

[87] T. Koponen, M. Casado, N. Gude, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, 2010.

[88] C. E. LaForest and J. G. Steffan. Efficient multi-ported memories for fpgas. In *Proc. ACM/SIGDA*, 2010.

[89] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *HotSDN*, 2010.

[90] L. E. Li et al. Toward software-defined cellular networks. In *EWSDN*, 2012.

[91] X. Lin, V. Yajnanarayana, S. D. Muruganathan, et al. The sky is not the limit: Lte for unmanned aerial vehicles. *arXiv preprint arXiv:1707.07534*, 2017.

[92] Y. Lin, U. C. Kozat, et al. Pausing and Resuming Network Flows using Programmable Buffers. In *Proc. ACM SOSR*, 2018.

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

[94] J. McCauley, A. Panda, M. Casado, T. Koponen, and S. Shenker. Extending SDN to large-scale networks, 2013.

[95] X. Meng et al. Improving the scalability of DC networks with traffic-aware VM placement. In *Proc. IEEE INFOCOM*, 2010.

[96] M. Mitzenmacher. Compressed bloom filters. *IEEE/ACM Transactions on Networking (TON)*, 2002.

[97] M. Moradi et al. Caesar: High-speed and memory-efficient forwarding engine for future internet architecture. In *Proc. ACM/IEEE ANCS*, 2015.

[98] M. Moradi, L. E. Li, et al. SoftMoW: a dynamic and scalable software defined architecture for cellular WANs. In *HotSDN*, 2014.

[99] M. Moradi, W. Wu, et al. SoftMoW: Recursive and reconfigurable cellular WAN architecture. In *Proc. ACM CoNEXT*, 2014.

[100] S. C. Nelson and G. Bhanage. GSTAR: Generalized Storage-Aware Routing for MobilityFirst in the Future Mobile Internet. In *Proc. ACM MobiArch*, 2011.

[101] Nguyen, Binh and Zhang, Tian and Radunovic, Bozidar and others. A reliable distributed cellular core network for hyper-scale public clouds. Technical report, 2018.

[102] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A scalable fault-tolerant layer 2 data center network fabric. In *Proc. ACM SIGCOMM*, 2009.

[103] A. Ostlin and R. Pagh. Uniform Hashing in Constant Time and Linear Space. In *Proc. ACM STOC*, 2003.

[104] M. Patel et al. Mobile-edge computing introductory technical white paper. *White Paper, MEC Industry Initiative*, 2014.

[105] B. Pfaff, J. Pettit, T. Koponen, et al. The design and implementation of open vswitch. In *Proc. USENIX NSDI*, 2015.

[106] Z. Qazi et al. A High Performance Packet Core for Next Generation Cellular Networks. In *Proc. ACM SIGCOMM*, 2017.

[107] Z. A. Qazi et al. SIMPLE-fying middlebox policy enforcement using SDN. In *ACM CCR*, 2013.

[108] Z. A. Qazi et al. KLEIN: A Minimally Disruptive Design for an Elastic Cellular Core. In *Proc. ACM SOSR*, 2016.

[109] A. S. Rajan et al. Understanding the bottlenecks in virtualizing cellular core network functions. In *LANMAN*, 2015.

[110] A. S. Rajan et al. Understanding the bottlenecks in virtualizing cellular core network functions. In *LANMAN*, 2015.

[111] M. Á. Ruiz-Sánchez, E. W. Biersack, and W. Dabbous. Survey and taxonomy of ip address lookup algorithms. *Network, IEEE*, 2001.

[112] M. Sarela, C. E. Rothenberg, T. Aura, and Zahemszky. Forwarding anomalies in Bloom filter-based multicast. In *Proc. IEEE INFOCOM*, 2011.

[113] M. Satyanarayanan et al. The case for VM-based cloudlets in mobile computing. *Pervasive Computing, IEEE*, 2009.

[114] A. Sharma et al. A global name service for a highly mobile internetwork. In *Proc. ACM SIGCOMM*, 2014.

[115] A. Singla, P. Godfrey, K. Fall, G. Iannaccone, and S. Ratnasamy. Scalable Routing on Flat Names. In *Proc. ACM CoNEXT*, 2010.

[116] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. In *ACM CCR*, 2005.

[117] H. Song, F. Hao, M. Kodialam, and T. Lakshman. IPv6 lookups using distributed and load balanced bloom filters for 100gbps core router line cards. In *Proc. IEEE INFOCOM*, 2009.

[118] N. Spring et al. Measuring ISP topologies with Rocketfuel. *ACM CCR*, 2002.

[119] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with Rocketfuel. In *ACM CCR*, 2002.

[120] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM TON*, 2003.

[121] T. Taleb et al. PERMIT: Network slicing for personalized 5G mobile telecommunications. *IEEE Communications Magazine*, 2017.

[122] T. Taleb, A. Ksentini, et al. Lightweight mobile core networks for machine type communications. *IEEE Access*, 2014.

[123] A. Tootoonchian and Y. Ganjali. Hyperflow: A distributed control plane for openflow. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, 2010.

[124] K. Wang et al. MobiScud: A Fast Moving Personal Cloud in the Mobile Network. In *Proc. Workshop AllThingsCellular*, 2015.

[125] Q. Wu, Y. Zeng, and R. Zhang. Joint trajectory and communication design for multi-uav enabled wireless networks. 2018.

[126] Xin Jin and Li Erran Li and Laurent Vanbever and Jennifer Rexford. SoftCell: Scalable and flexible cellular core network architecture. In *Proc. ACM CoNEXT*, 2013.

[127] W. Xu and J. Rexford. MIRO: Multi-Path Interdomain ROuting. In *Proc. ACM SIGCOMM*, 2006.

[128] M. Yu, A. Fabrikant, and J. Rexford. BUFFALO: Bloom Filter Forwarding Architecture for Large Organizations. In *Proc. ACM CoNEXT*, 2009.

[129] F. Zane, G. Narlikar, and A. Basu. Coolcams: Power-efficient tcams for forwarding engines. In *Proc. IEEE INFOCOM*, 2003.

[130] K. Zarifis et al. Diagnosing path inflation of mobile client traffic. In *Proc. PAM*, 2014.

[131] K. Zarifis, T. Flach, S. Nori, D. Choffnes, R. Govindan, E. Katz-Bassett, M. Mao, and M. Welsh. Diagnosing path inflation of mobile client traffic. In *Proc. PAM*, 2014.

[132] L. Zhang et al. Picocenter: Supporting long-lived, mostly-idle applications in cloud. In *Proc. ACM EuroSys*, 2016.

[133] K. Zheng, C. Hu, H. Lu, and B. Liu. A TCAM-Based Distributed Parallel IP Lookup Scheme and Performance Analysis. *IEEE Trans. Networking*, 2006.

[134] A. L. Zobrist. A New Hashing Method with Application for Game Playing. Technical report, 1970.

[135] J. C. Zuniga, C. J. Bernardos, A. de la Oliva, T. Melia, R. Costa, and A. Reznik. Distributed mobility management: A standards landscape. *Communications Magazine, IEEE*, 2013.