Hierarchical Control Plane Designs to Scale SDN Applications

A Thesis
Submitted in partial fulfillment of
the requirements for the degree of
Doctor of Philosophy
by

Rinku Mahendrakumar Shah
(Roll No. 134053001)

Supervisors:
Prof. Mythili Vutukuru
and
Prof. Purushottam Kulkarni

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400076 (India)

9 February 2021

In memory of my loving brother, Videet

Acceptance Certificate

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay

The thesis entitled “Hierarchical Control Plane Designs to Scale SDN Applications” submitted by Rinku Mahendrakumar Shah (Roll No. 134053001) may be accepted for being evaluated.

Date: 9 February 2021

Prof. Mythili Vutukuru

Approval Sheet

This thesis entitled “Hierarchical Control Plane Designs to Scale SDN Applications” by Rinku Shah is approved for the degree of Doctor of Philosophy.

VSN

Prof. Vinayak Naik, BITS Pilani, Goa

Digital Signature
Varsha Apte (i02020)
09-Feb-21 03:16:20 PM

Examiners

Digital Signature
Mythili Vutukuru (i13124)
09-Feb-21 01:31:03 PM

Digital Signature
Purushottam Kulkarni (i06166)
09-Feb-21 04:44:43 PM

Supervisor (s)

Madhu N. Belur, EE Dept, IITB

Chairman

Date: 8th Feb 2021

Place: Mumbai

Declaration

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I declare that I have properly and accurately acknowledged all sources used in the production of this report. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Rinku Mahendrakumar Shah

Date: 9 February 2021

(Roll No. 134053001)

Abstract

Software-defined networking (SDN) proposes the decomposition of the control and data planes in network elements, with the control plane running in a logically centralized controller and the data plane running on simple SDN switches. While the SDN-based design of networks confers many benefits like increased flexibility, traditional centralized SDN controllers are known to suffer from scalability issues. There are two general approaches to scaling: (1) a horizontal SDN scaling architecture uses distributed systems principles to maintain consistency amongst the various controller replicas, and (2) a hierarchical SDN scaling architecture offloads a subset of control plane computations to local controllers located close to (or on) the data plane switches (switch-local CPU). The horizontal architecture scales sub-linearly since synchronization overhead increases with more replication, and can significantly increase control plane latencies. In hierarchical architecture, the control traffic that depends on global network state is handled by the centralized SDN controller, while the control plane traffic interacting only with the switch-local state is handled by the local controllers. These architectures have different tradeoffs. Horizontal scaling works well when the majority of control plane traffic requires computations on the global network state. In contrast, the hierarchical approach works well when most control plane traffic requires only access to switch-local state. Also, while the former design can offload any control plane computation to any replica (after proper state synchronization), the latter can only offload a subset of control plane computations that local controllers can process correctly. These two approaches are complementary and can independently scale the control traffic that requires the global network state and switch-local state. This thesis proposes new techniques to improve performance and scalability for a hierarchical SDN architecture.

Prior work on the hierarchical scaling of SDN controllers was limited to offloading computations based on the local switch-specific state alone. We develop hierarchical scaling solutions that offload computations to local controllers or programmable hardware switches based on a subset of global application state as well, thereby improving the state-of-the-art in hierarchically scaled SDN controller design. The key idea of our work is

that a subset of the global network-wide state that is accessed from only one network location at any time can be offloaded to (and accessed/updated at) local controllers or switches, with suitable synchronization from time to time. By offloading such state (and the computation that depends on this state) away from the centralized SDN controller and close to the end-user, the control plane capacity of SDN applications can be significantly improved and response times can be significantly reduced.

There are several challenges to realizing this idea. One such challenge is the identification of application computations that can be offloaded to the local controllers and programmable hardware switches. Also, offloading of state and the subsequent synchronization of the state between the local and central controllers must be performed in a manner that ensures application correctness under all modes of failures and other network events. Further, offloading must only be performed when the relative gains of offload outweigh the cost of synchronizing this state across controllers. Finally, offloading must be done in a manner that is transparent to application developers. Our work addresses all these challenges comprehensively.

In this thesis, we have built two systems. The first system, the Cuttlefish SDN controller framework, helps SDN applications adaptively offload state and computation to local controllers running on switch CPU, based on whether the offload improves application performance or not. CuttleFish is a generic framework that can be used by any SDN control plane application via the CuttleFish APIs. The second system, TurboEPC, is a specific implementation that demonstrates that it is possible to accelerate a mobile packet core SDN control plane application by offloading both state and computation to the hardware programmable dataplane. With programmable switches and languages such as P4 that can program such switches gaining popularity, we observe that offloading SDN application computation to the switches can lead to better performance gains than those seen with offloading computation to local software controllers alone.

We have prototyped and evaluated our ideas for both systems using several real-world applications, such as the SDN-decomposed mobile packet core, and demonstrated significant performance improvements over the status quo. The observed throughput was up to 2× and 12× the traditionally centralized SDN design for Cuttlefish and TurboEPC, respectively. Besides, the response latency was observed to be up to 80% and 97% lower than the traditionally centralized SDN design for Cuttlefish and TurboEPC, respectively.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Traditional network architectures are unable to meet the requirements of today's data center networks [7]. Network requirements are evolving, and therefore there is a need to re-evaluate traditional network architectures. Let us look at the evolving network use-cases and their impact on the network load and behavior.

- **Increase in traffic demand.** Modern network applications like *big data* process large amounts of data. *Internet-of-Things (IoT)* is yet another technology that adds enormous traffic to the network [8]. For example, huge amounts of video traffic is generated by surveillance cameras deployed widely for security purposes. With mobility support, users can request network services practically from anywhere, i.e., services are available from the office, home, or in transit. The ease of access to services adds to the increase in traffic trends. The network traffic dynamically increases, so there is a requirement for on-demand addition of network resources [7].

- **Unpredictable load on network devices and links.** With the advent of virtualization, a traditional physical server hosts multiple virtual machine (VM) instances across the data center network [9]. The users exchange traffic with any of these servers, and the servers exchange traffic with each other to maintain a consistent application state [9]. The physical locations of these VMs can change due to VM migrations that data centers pursue to optimize resources [10]. It is incredibly challenging to predict the amount of link traffic within the network, and hence static network resource provisioning fails.

- **Vendor-specific device interfaces.** To cope with the dynamic network demands, network designers continuously evolve the hardware capacities, network capabilities, and network protocols. They produce a variety of hardware and protocols. All of these devices have closed, vendor-specific interfaces for access, management,

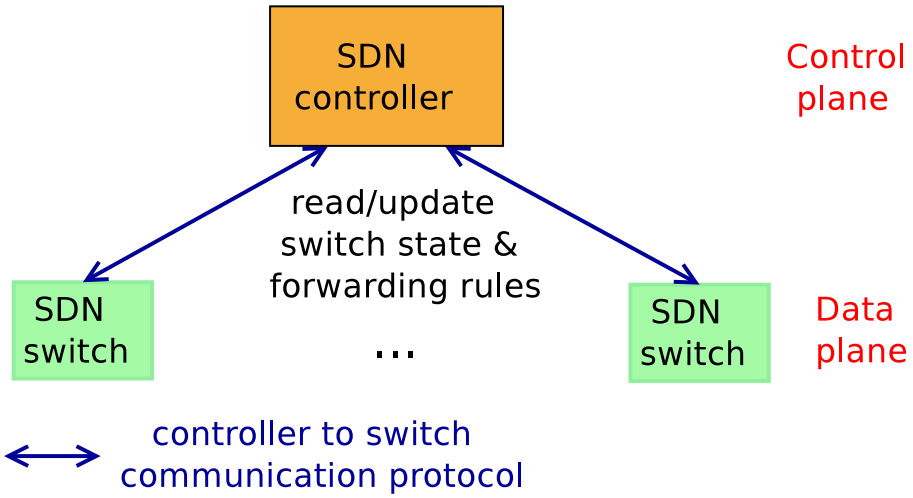


Figure 1.1: A software-defined network.

and configuration [11]. The only people who can innovate such equipment are the corresponding vendors, which introduces substantial delays for the addition of new features.

- **Manual network management.** If a network administrator wants to add or move any device, she must reconfigure multiple devices like the switches, routers, firewalls, and authentication servers. The network has to adhere to the security, privacy, and policy norms; therefore, the ACLs, VLANs, and other policies also have to be updated. Not to forget that the underlying devices have different interfaces; hence the device management commands are different. There is massive growth in the physical network size due to the increase in traffic demands, and manual network management is challenging [7].

The conventional network architectures are unable to respond to the rapid changes in the network demands. There is a need to provide network abstractions for devices and protocols to make networks programmable. Network programmability aids automation of complex network tasks like device configuration, forwarding, and monitoring. It eases the job of network administrators and promotes innovation.

Software-Defined Networking (SDN) is a step towards making the networks manageable. Software-defined networking [12] is a design paradigm of separating the control and data planes of network elements (see Figure 1.1). A software-defined network consists of a software-managed, logically centralized controller, and light-weight switches that are programmed with forwarding rules by the controller. Any data plane traffic for which the rules do not exist at the switching device, or signaling messages that require control plane processing, are directed to the controller by the switches. SDN applications

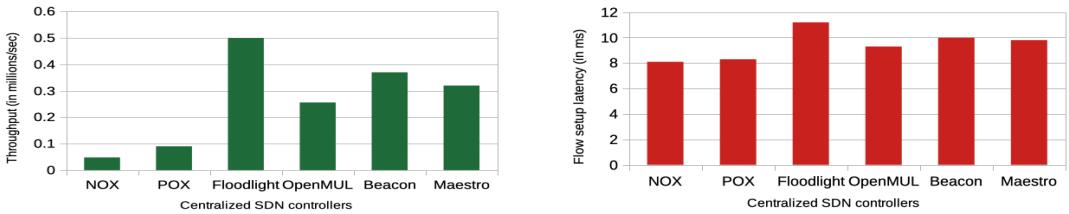
running at the controller process these messages and install corresponding forwarding rules on the data plane switches.

The SDN-based design of networks confers many benefits [11]. The SDN paradigm advocates the use of standard abstractions and interfaces for communication between the SDN controller and the vendor-specific devices. Standard abstractions and interfaces help the network administrator automate the configuration and management of the entire network remotely via the application running at the logically centralized SDN controller. Since all the network devices offer standard interfaces, addition or modification of vendor-specific network hardware does not require changes to the network management applications running at the controller. SDN supports the management of physical and virtual switches from a single centralized controller. The centralized view of the entire network offered by the SDN design helps in better management of network traffic. The controller monitors the network traffic by periodically fetching the load counters at the switch using the controller-device standard interfaces. The traffic engineering application running at the SDN controller decides the route modifications to balance the network traffic. As per these decisions, the controller application dynamically updates the route entries at the network switches. The amount of manual intervention for tasks like network configuration, debugging, and management reduces, thereby resulting in lower operating expenses (OPEX). With the changes in network requirements, we do not have to replace the existing network hardware. We can re-purpose the current network hardware to follow the instructions of the SDN controller. The reuse of existing equipment helps to reduce capital expenditures (CAPEX).

1.1 Limitations of traditional SDN controller design

With increasing traffic demands, the centralized controller that runs as a software component can become a scalability bottleneck [13–26]. The control plane is split and pushed away from the data plane, which leads to delayed control plane decisions and affects the control plane application performance in terms of throughput and latency. The centralized controller becomes a single point of failure and introduces reliability and security challenges.

To understand the SDN control plane scalability problem, let us observe the performance of the traditionally centralized SDN controllers. Figure 1.2 (a) and Figure 1.2 (b) show the control plane throughput and flow setup latency for centralized SDN controllers, respectively [1–3]. Note that the evaluation metrics are based on the flow setup throughput and latency. In addition, these numbers heavily depend on the evaluation environment pa-



(a) Control plane throughput.

(b) Flow setup latency.

Figure 1.2: Performance of traditional SDN controllers [1–3].

rameters such as workload and network topology considered during the individual experiment. Keeping this context in mind, we observe the maximum control plane throughput is *0.5M* control plane messages per second, and the minimum flow setup latency is *8ms*. *Are these centralized (single machine) SDN controllers fast enough? If not, how faster do they need to get?* Following are the requirements of some of the existing datacenter application deployments.

- Key-value (KV) stores play an important role in enterprises such as social networks, online retail, and risk analysis. Atikoglu et. al [27] have studied Facebook data center's Memcached (key-value store) application. The analysis of traffic characteristics demonstrated a peak load of more than *140M* requests per sec.

- There is enormous growth in the number of mobile subscribers over the years. Studies [28] have shown that India had *1 billion* mobile subscribers in 2016 and is expected to reach *1.7 billion* by the end of 2021. Using the traffic characterization studies [5, 6], we derive that the mobile control plane traffic will reach up to *460M* requests/sec by the end of 2021. Further, the latency targets for these control plane messages in future 5G networks [29–31] is as low as *1ms*. The high control plane load at the mobile backbone network with the centralized control plane makes it challenging to satisfy application SLAs.

- The increase in data generation has raised the demand for data centers globally. Kachris et. al [32] state that data center ingress traffic (user to the data center) was around 1200 Exabytes per year in the year 2016. Datacenter traffic measurement studies [33] suggest that 80% of the flows are smaller than 10KB in size. Using these measurements and traffic characterization studies, we infer the ingress control traffic rate at typical datacenters will reach up to *3.2 Million packets per sec*. Note that the datacenter traffic doubles every 12–15 months [34]. Datacenter applications such as load balancers, security applications (for example, DDOS attack detection

and firewall), network traffic monitoring, traffic engineering, and failure detection require to process such high rate incoming and outgoing traffic.

Traditional SDN controllers fail to process such high-intensity control plane traffic. They also fail to satisfy application SLAs because of high response latencies. The control packet processing involves packet traversal from the switch to the SDN controller, which adds few milliseconds to the response latency. The SDN controller scalability problem can result in serious repercussions. For example, on a network link failure if the failure detection application does not react quickly (slow convergence), there could be more than *60K retransmissions per second* [35]. We address the SDN control plane scalability problem in this thesis.

1.2 SDN scalability solutions

Traditional centralized SDN controllers are known to be not scalable [13–26]. Prior research has identified several scalability problems with centralized SDN controllers and the communication path between the data plane switches and controllers. There are proposed solutions to fix the same.

One set of solutions [13, 14, 18–20] develop *horizontally* scalable SDN controllers that scale the centralized SDN controller by instantiating multiple homogeneous instances of the centralized controller and distributing the control load with techniques like network topology partitioning or state partitioning. The controller instances use standard synchronization techniques to distribute application state between themselves to maintain a logically centralized view. For example, we can divide the entire network topology into smaller subsets and assign a subset to each controller instance for routing applications. Each controller instance takes the routing decision for the packets that arrive in its topology subset. The controller instance can also determine the route for a destination outside the topology subset since each replica is synchronized to maintain a consistent network-wide view.

Other solutions [15–17, 21–26] propose *hierarchical SDN controllers* which offload computation that does not require network-wide view to *local* controllers. A local controller is a replica of the centralized SDN controller, and it resides at (or close to) the switch. For example, some traffic engineering applications detect flows that comprise a large number of packets or huge packet sizes (elephant flows) before calculating optimal routes. These applications can offload the task of detecting large flows to local controllers. The local controllers maintain the local state of switch flow statistics, while the centralized *root* controller only runs route computations that require a network-wide view (global

state). This decoupled computation setup results in a lower computation load at the root controller and reduces the network traffic between the switches and the root controller. Hierarchical SDN controller scalability solutions like Difane [21] offload management tasks like access control, measurement, and routing to the intelligent switches. The controller dynamically generates a set of rules to satisfy network policies and offloads them to the switches. The data plane switches processes the incoming packets by applying the offloaded rules, and the load at the centralized root controller reduces.

The two design options — horizontally distributed controllers and hierarchical controllers — are complementary ideas, with their strengths and drawbacks. While the former design can offload any control plane computation to any replica (after proper state synchronization), the latter can offload only a subset of control plane computation that local controllers can process correctly. However, horizontally distributed controller frameworks incur a performance overhead due to the synchronization of network-wide state across replicas, while hierarchical controller designs have no such associated costs because the local switch-specific state does not require synchronization.

Our proposal to scale the SDN control plane extends the concept of hierarchical controller design. Existing hierarchical scaling solutions offload computations based on local switch-specific state. Our key insight is that we can improve the performance gains by an additional offload of computations. We discuss the details of our proposal in the next section.

1.3 Key ideas

Prior work implicitly classifies an SDN application's state into *global network-wide state* (that pertains to, or is concurrently accessed by, multiple switches/entities in the network) and *local switch-specific state*. The local state can be maintained at local controllers, and control plane messages that depend on such state can be offloaded to local controllers. The global state must be maintained at the root controller (or with tight synchronization across distributed controllers). The control plane messages that access the global state must necessarily be processed at the centralized root controller (or its synchronized instances).

1. **New taxonomy of application state**

   The key observation of our work is that, beyond the dichotomy of the local and global state, there is a third type of state that we refer to as *offloadable state*. We classify the application states as offloadable and non-offloadable.

   The global network-wide state that can be accessed concurrently from multiple network locations is called the *non-offloadable state*. SDN applications such as routing

protocols and failure handling algorithms use the network topology state. Such a global state should be consistent over the network. Therefore, the network topology state is an excellent example of a non-offloadable state.

We define *offloadable state* as the state that is accessed from a single network location, i.e., all control plane traffic that accesses this state should traverse through the common network edge switch. Offloadable state includes switch-local state and some types of session-specific application state. The traffic engineering application uses switch counters that maintain per-flow length, and such a state is a good example of a switch-local offloadable state. The per-flow (or per-session) tunnel identifier state used for encapsulation of data packets is an excellent example of a session-specific offloadable state.

As the non-offloadable state can be concurrently accessed from multiple network locations, we maintain this state at the centralized root controller and assure a consistent view across network locations. Any computation that depends on such states should be processed at the centralized controller and are called *non-offloadable* messages. The control plane messages that access the offloadable state and do not require concurrent access to any other non-offloadable state can be processed locally (switch-CPU or local controller) are called *offloadable* messages. The updates to the cached offloadable state at the local nodes are lazily synchronized with the state's master copy at the centralized root controller; that is, the cached state is synchronized only when the non-offloadable message requests access to the offloadable state. We describe the detailed state taxonomy with the help of real-life examples and guide classification of application messages in §3.2.1, §3.2.2, and §3.2.3.

2. **Adaptive offload of subset of control plane computations to local controllers**
   The key idea of our first work, *Cuttlefish*, is that, by synchronizing the offloadable state from the centralized root controller to specific local controllers, the messages that access the offloadable state (offloadable messages) can be offloaded to local controllers (close to the user). The offload to local controllers can lower the computation overhead at the centralized root controller, resulting in higher control plane capacity and lower latency for the SDN application.

   Performing computation based on offloadable state at local controllers is beneficial only if the state cached at the local controller needs to be synchronized with the master copy at the centralized root controller infrequently. If the traffic characteristics entail frequent updates to the offloadable state, there is frequent synchronization between the root and local controllers. This synchronization cost may outweigh the

benefit of computation offload, and a traditional design that does not offload such a state might work better. With traffic characteristics being dynamic, our SDN controller framework, Cuttlefish supports offloading of offloadable state, and associated computation, adaptively between the traditional centralized design and the Cuttlefish offload design based on the cost of synchronization, to optimize system performance.

3. **Offload subset of control plane computations to programmable hardware switches**

   We observe significant scalability and latency benefits when the subset of control plane computations are offloaded from the centralized root controller to the local controllers (close to the user). To increase the benefits further, we take inspiration from the recent advances in data plane technologies. The data plane switches are evolving from fixed-function hardware towards programmable components that can forward traffic at line rate while being highly customizable [36, 37].

   We can significantly increase the throughput and latency gains compared to our first work, Cuttlefish, if the control plane computations can be programmed and offloaded to the programmable edge switches (close to the end-user). We see performance improvements because the packet processing at the switch eliminates the traversal of the local controller's network stack and application layer stack. Programming the offloadable computations is made easy by the protocol independent, target-independent programming language, *P4* [4].

   The key idea of our second work, *TurboEPC*, is to demonstrate that we can accelerate the control plane of the mobile packet core SDN application by offloading the processing of offloadable messages at the programmable hardware switches, close to the end-user. The offloadable message processing also requires caching of offloadable state at the programmable hardware switches and appropriate state synchronization with the master copy of the offloadable state stored at the centralized controller, to avoid access to the stale state at the root controller by non-offloadable messages.

We implement a variety of use-cases like the mobile packet core, stateful load balancer, and the simple key-value cache to demonstrate the effectiveness of our proposals.

1.4 Challenges

There are several challenges in realizing our Cuttlefish and TurboEPC ideas. We list the challenges and describe the overview of the approaches taken to handle them.

- **Classification of application state and computations**
  An application designer who wishes to use our computation offload approach has to classify the application state. Our proposed state taxonomy classifies the application state as offloadable or non-offloadable (§3.1.2). If the control message processing requires access to the offloadable states alone, then we can process the message locally, at local controllers or programmable hardware switches. Otherwise, if the control message processing requires access to some non-offloadable state, we must process the message at the centralized root controller. We describe the detailed state taxonomy with real-world examples and guide the classification of application messages.

- **Inconsistency of offloadable state**
  Our proposed offload design comprises of two copies of the offloadable state. The centralized root controller has a master copy of the offloadable state. This state is cached locally, at the local controllers or programmable hardware switches. Offloadable messages are processed locally, and they modify the cached copy of the offloadable state that is lazily synchronized with the master copy. Offloadable message processing causes the offloadable state to diverge from the master copy resulting in stale offloadable state at the centralized root controller. The non-offloadable messages are processed at the centralized root controller and can sometimes access the stale offloadable state, leading to incorrect application behavior. Cuttlefish implements an automated state synchronization framework to manage offloadable state consistency. Cuttlefish uses batching mechanism to reduce the state synchronization costs. In TurboEPC, the programmable switch piggybacks the cached offloadable state values with the non-offloadable message and forwards it to the centralized root controller (on-demand state synchronization).

  There is one more reason for the inconsistency of the offloadable state. The Cuttlefish framework automatically switches from the offload design to the traditional centralized design when the offloadable state synchronization costs increase. The incoming packets are processed even during the migration between the SDN designs. We need to ensure that the state accessed by these packets is consistent and also ensure the correctness of the application. Cuttlefish framework implements a migration protocol to ensure state consistency during SDN design migration.

- **High state synchronization costs**
  The updates to the offloadable state at the centralized root controller are immediately synchronized with the local cache copy to maintain strong consistency. The state synchronization cost increases if the offloadable state is updated frequently at the centralized root controller, and the benefits of offload are lost. The Cuttlefish framework determines the state synchronization cost and dynamically switches between the proposed offload design and the traditional centralized SDN design to reduce the state synchronization costs. TurboEPC implements the offload concepts for the mobile packet core application, where the current traffic distribution trend is such that the synchronization costs are within limits. In case the traffic distribution changes in the future, we should not offload computations of the mobile users whose state requires frequent state synchronization (e.g., users with high mobility rate) and thereby control the state synchronization cost.

- **State losses due to offload node failure**
  Failure of local nodes where the offloadable state is cached can lead to loss of the latest version of the offloadable state. Such losses can result in state inconsistencies and incorrect application behavior. TurboEPC overcomes this challenge by replicating the offloadable state across the programmable switches at the data plane, and the SDN controller implements a failover mechanism to tackle switch failures. Note that, Cuttlefish does not implement any unique failure management technique since SDN controller frameworks have inbuilt mechanisms for state replication and fault management. Cuttlefish relies on these mechanisms instead of reinventing the wheel.

- **Limited memory to store offloadable state**
  Programmable hardware switches have limited memory (few 10's of MBs) for storing application states. The offloadable state size can be large enough to not fit into the limited switch memory—for example, the mobile packet core application manages millions of active users ([5, 38]), and it requires 100's of MBs of memory. It is not possible to accommodate the context of so many users within a single switch. To overcome this challenge, TurboEPC partitions the offloadable state across multiple switches, which increases the probability of storing the offloadable state for all users at the data plane. TurboEPC also implements traffic steering mechanisms to forward the incoming control messages to the switch where the required state is stored. This challenge does not apply to Cuttlefish since the offloadable state is stored at the local controllers, and they run in software and have enough memory.

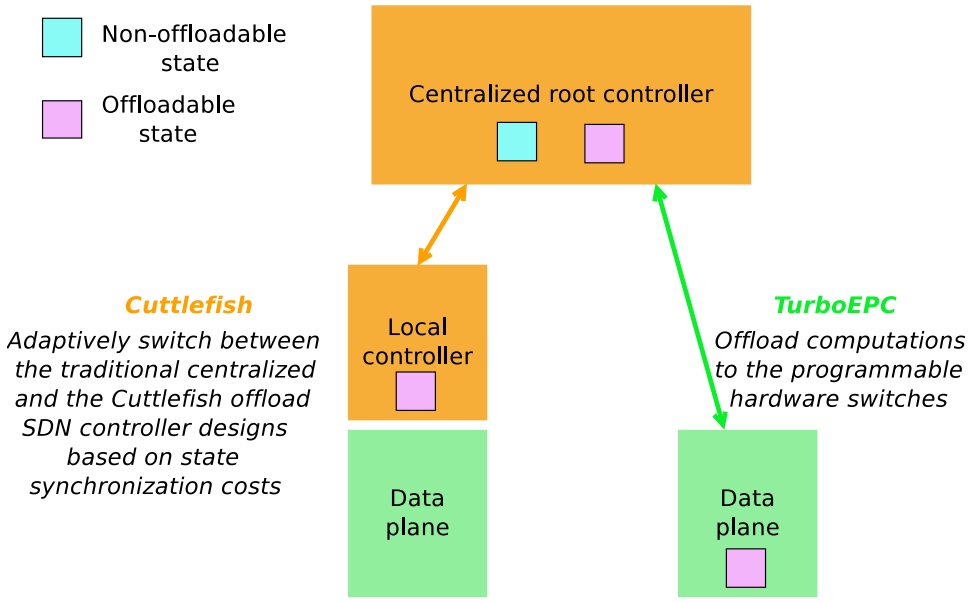


Figure 1.3: Contributions.

Note that, given the fixed and limited amount of storage in the programmable hardware dataplanes, the goals of scalability and fault tolerance are conflicting. TurboEPC prioritizes fault tolerance over scalability since we require the user context to be consistent and available. We address the question of how best to partition user contexts across multiple programmable switches in §5.3.2.

1.5 Thesis contributions

We now describe our contributions (illustrated in Figure 1.3).

1. **Classification of application state**
   We introduce a new taxonomy of state for SDN applications beyond the existing notions of global network-wide and local switch-specific states. We classify the application state either as an offloadable state or a non-offloadable state. The application programmers who wish to use our scalability framework have to classify the messages (or packets) as offloadable and non-offloadable. We describe the detailed state taxonomy with the help of complex real-world applications like the mobile packet core and guide the classification of application messages in Chapter 3.

2. **Cuttlefish: Adaptive computation offload to local controllers**
   Cuttlefish offloads a subset of control plane computations (offloadable messages) to the local controllers, close to the user. This framework manages the synchronization of the offloadable state across the centralized root and local controllers. Further, our

framework continuously monitors the cost of state synchronization across the centralized root and local controllers. It dynamically switches between the traditional centralized and our proposed offload SDN designs to maximize the application performance, in a manner that is transparent to the application. The implementation and evaluation of the benefits of our design are discussed in Chapter 4.

3. **TurboEPC: Computation offload to programmable hardware switches**
   TurboEPC redesigns the SDN-based mobile packet core, and offloads a significant fraction of signaling procedures from the control plane to the programmable data plane (hardware switches), thereby improving the performance significantly. We implement TurboEPC over P4-based programmable software and hardware switches, to demonstrate the feasibility of our design. Further details about the TurboEPC design and implementation are discussed in Chapter 5.

4. **Comparison of proposed controller scalability designs with the status quo**
   We present the quantification of performance gains of our offload frameworks Cuttlefish and TurboEPC over traditional centralized design (single-core and multi-core) and the horizontal scaling SDN design. We also provide a guide for the choice of the SDN design based on the application and traffic characteristics. The detailed performance comparison of all SDN designs is presented in Chapter 6. The codebase of our work is open-source [39, 40] and is available for innovation.

1.6 Summary

In this chapter, we discussed the evolution of traditional networks to software-defined networks and introduced the SDN concept along with the benefits and limitations. We explained the reason for the existence of the SDN control plane scalability problem. We discussed the existing solution approaches and their limitations. We introduced our ideas and proposals for a scalable SDN control plane, along with the challenges that we have addressed, and listed our contributions towards solving the SDN control plane scalability problem.

The organization of the rest of the thesis is as follows — Chapter 2 provides a comprehensive discussion on the SDN control plane scalability background and related work. Chapter 3 describes the proposed state taxonomy for SDN application, presents a guide to classify SDN messages, and applies the guide to classify the state of some real-world applications like the mobile packet core. Chapter 4 presents the design, implementation, and evaluation of the Cuttlefish framework. Chapter 5 presents the design, implemen-

tation, and evaluation of the TurboEPC framework. Chapter 6 presents the empirical performance comparison of the SDN controller designs—the traditional centralized design, horizontal scaling design, Cuttlefish design (offload to local controllers), and the TurboEPC design (mobile packet core SDN application offloaded to edge programmable hardware switches). We also provide insights into the choice of an appropriate SDN design based on the application and traffic characteristics. Chapter 7 provides insights on how this work applies to the future 5G mobile packet core and other real-world applications, and Chapter 8 concludes this thesis.

Chapter 2

Background and Related Work

This chapter covers all the concepts, technologies, and use cases that are the fundamentals of this thesis. We discuss software-defined networking, which is a new paradigm that introduces control plane programmability. We also discuss the recent advances that enable data plane hardware programmability. We describe the mobile packet core use-case since we will use it to demonstrate the benefits of our proposed ideas. While discussing the concepts, we also provide related work and differentiate our work from the existing works.

2.1 Software-defined networking

The traditional networking model advocates a tightly coupled control plane and data plane. The term *data plane* refers to the commodity network switches that perform the function of packet forwarding, and the term *control plane* refers to the program that configures the forwarding rules at the switch tables. The routing protocols like Routing Information Protocol (RIP), Open Shortest Path First (OSPF), and Border Gateway Protocol (BGP) are examples of control plane programs. Due to the tightly coupled design, traditional networks are unable to satisfy the new generation network requirements. Following are some of the challenges faced by the traditional networks-

1. **Add/upgrade network services.** With traditional networks, it is challenging to write code for new network services or upgrade the existing ones. Along with the implementation of the network service functionality, the programmer has to manually ensure the compliance of the new service with the existing network policies. With the increase in the number of services in the network, network application programming in the traditional networking model becomes very complicated. It requires enormous time and effort leading to poor revenue models [7].

2. **Network device configuration for large-scale networks.** Network management involves configuration and monitoring of vendor-specific network devices that are physically distributed across the network. Traditional networks either manually configure the network devices or use vendor-specific interfaces. But, in order to use vendor-specific interface alone, all the devices should be manufactured by the same vendor, which may not be true for large networks [7].

3. **Real-time network monitoring and control.** The traditional networks are not well-designed to automatically perform network operations like adapting to network load changes or solve network faults in large-scale networks [41]. Traditional network applications are distributed, and hence they are slow and fail to make real-time decisions.

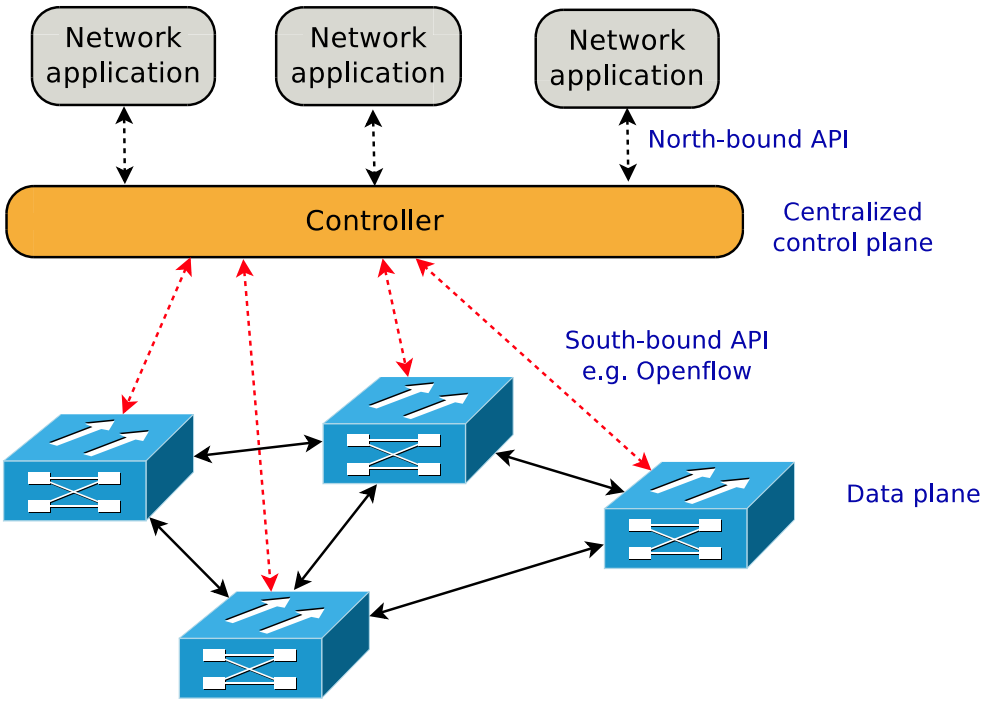


Figure 2.1: Traditional SDN architecture

Software-Defined Networking (SDN) is a new networking paradigm that fixes the challenges of traditional networks. With SDN, the traditional fully distributed networking control plane model moves towards a centralized model (shown in Figure 2.1). The SDN control plane is a software capable of running on commodity servers, which makes the control plane programmable. This software control plane component is known as an *SDN controller*. The controller installs the forwarding rules on switches using a South-bound API like *OpenFlow* [42], to enable appropriate packet forwarding. All the incoming data

packets for which the forwarding rules do not exist at the switch or the control messages that require control plane processing, are directed to the SDN controller by the switches. The communication between the SDN controller and the switches helps in maintaining consistent network visibility at the controller. The applications are written on top of the SDN controller, and they could utilize the global network view, using the North-bound APIs.

SDN paradigm proposes a logically centralized control plane that maintains the global network view, and advocates the use of network abstractions and the standardization of network interfaces. The network-wide view comprises of statistics about the flows, switches, and the network links for the entire topology. SDN design confers many benefits [11].

- **Centralized network provisioning.** SDN applications are offered a unified perspective of the entire network. Such abstractions help the programmers to write centralized network applications that provide services like enterprise management and resource provisioning.

- **Abstraction of networking infrastructure.** Network elements like the network devices, virtual networks (an organizational network provisioned in the cloud with the hosts as VMs), and network service chains (logical chain of VMs) use the standard device interfaces for centralized and dynamic configuration and management.

- **Granular security.** Modern data centers have replaced the typical physical server machines by virtual machines that run over commodity servers. The use of virtual machines poses an additional challenge for firewalls and content filters. SDN provides a central control point for regulating enterprise security and privacy policies.

- **Low operational costs.** SDN benefits like centralized network administration and management help cut operating costs. Most services like network configuration, debugging can be automated using the global network view at the SDN controller. The amount of manual intervention reduces, thereby resulting in lower operating expenses (OPEX).

- **Low capital infrastructure costs.** SDN is implemented using open standards like OpenFlow, and the SDN controller provides an abstraction for network devices from multiple vendors, so we do not have to be constrained to a specific vendor (vendor neutrality). With the increase in network capacity demands, we do not have to replace the existing network hardware. Existing hardware can be repurposed to

follow the instructions of the SDN controller. The reuse of existing equipment helps to reduce the capital expenditures (CAPEX).

- **Consistent and timely content delivery.** One of the important benefits of SDN is the ability to quickly, automatically, and dynamically configure the routes for data traffic; based on the load on network devices and links. This SDN benefit helps improved user experience and quality of service (QoS) for real-time applications like the Voice over Internet Protocol (VoIP).

Along with all the above benefits, SDN-based design has its limitations. With the increasing traffic demands, the centralized controller that runs as a software component can become a scalability bottleneck [13–20, 22–25]. The control plane is split and pushed away from the data plane, which leads to delayed control plane decisions and affects the control plane application performance in terms of throughput and latency. The centralized controller becomes a single point of failure and introduces reliability and security challenges. We address the SDN control plane scalability problem in this thesis.

2.2 SDN control plane scalability

The SDN-based networking design can be easily adopted if it can handle the scalability, throughput, and latency demands of SDN applications. There are three kind of bottlenecks in an SDN-based network —

- **Data plane.** The network switches that forward the data packets could become the bottleneck under high network load. The solution is to increase the number of switching hardware devices or replace the current switches with a higher capacity switch.

- **SDN controller.** The SDN controller is a piece of software running on commodity servers. The amount of control traffic that the traditional SDN controller can process is limited. The control packet that arrives at the controller after saturation is either dropped or the packet processing latency increases. Researchers have proposed a variety of controller scalability designs [13–20, 22–25] to solve the SDN controller capacity problem.

- **Switch to controller communication.** The control traffic between the network switch and the controller can fill the network pipe during heavy control traffic. Researchers have proposed solutions that avoid frequent traffic to the centralized controller by processing a subset of the control plane requests at the local controllers

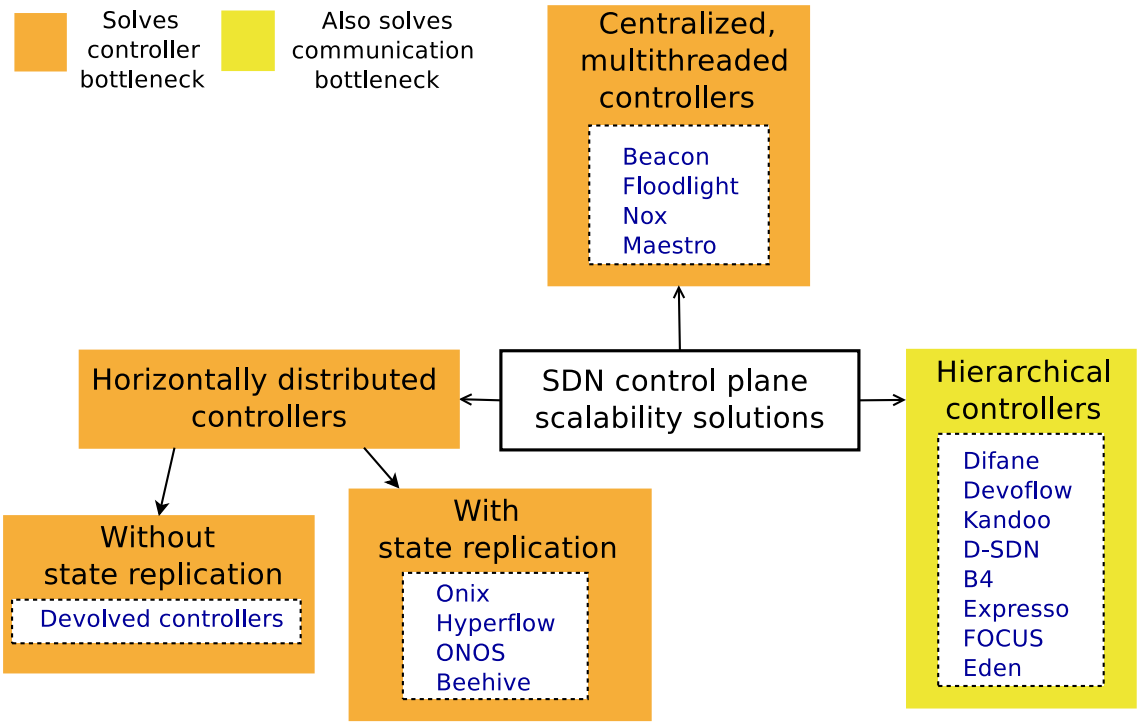


Figure 2.2: Classification of SDN control plane scalability approaches.

that run at (or close-to) the switches [15–17, 22–25], which reduces the communication bottleneck.

The focus of our work is to scale the control plane of software-defined networks, so we discuss the solutions that deal with — (1) SDN controller bottleneck and the (2) switch to controller communication bottleneck. The existing literature on SDN control plane scalability can be classified as follows (refer Figure 2.2) —

1. **Centralized, multithreaded controller design.** Traditionally, SDN controllers were single-threaded. The most intuitive step towards scalability is to design the controller as a multithreaded program, to parallelize control traffic processing (see Figure 2.3). Beacon [43], Floodlight [44], NOX [45], Maestro [46] are some of the popular multithreaded SDN controllers. They improve the flow processing capabilities using multiple thread pipelines and shared queues.

   Even with the thread parallelism, there is a hard limit up to which a single physically centralized controller can scale. For large networks, the centralized controller could be far from the ingress switches—switches through which the traffic enters the network. The distance between the ingress switch and the controller is directly proportional to the control plane response latency.

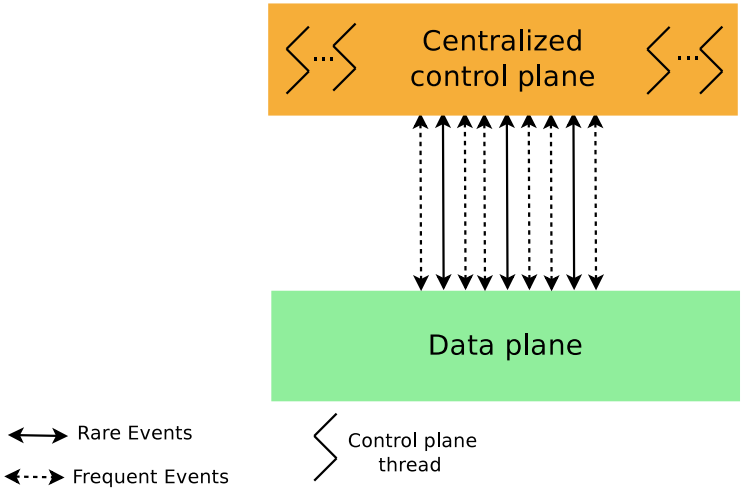


Figure 2.3: Centralized, multithreaded controller design.

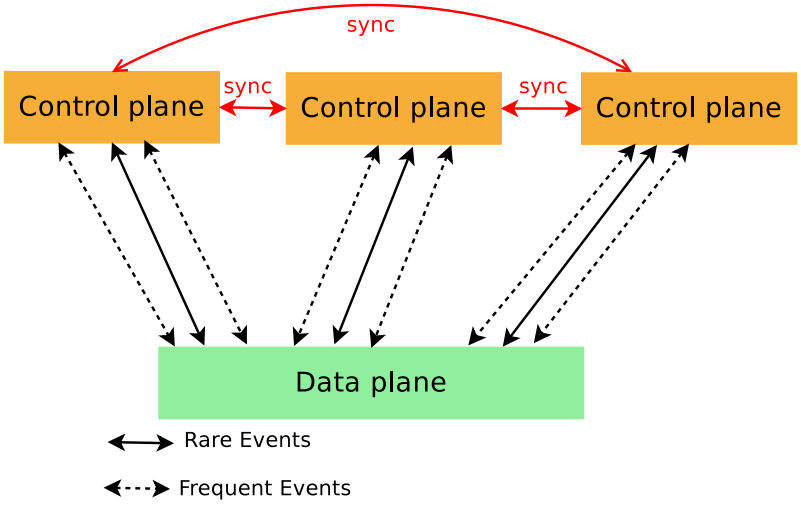


Figure 2.4: Horizontally distributed controller design

2. **Horizontally distributed controller design.** The capacity of the centralized controller can exhaust with high-frequency control plane traffic. One of the solutions is to horizontally distribute the control plane load over multiple homogeneous controller replicas running at commodity servers (see Figure 2.4). Each controller replica manages the control traffic that arrives at a subset of network topology switches. The SDN control plane becomes scalable and provides better control plane throughput as compared to the centralized controller design. The controller replicas should implement strict synchronization mechanisms to maintain the consistent network-wide state, which increases the computation overheads at the replicas. The horizontally distributed controllers can be further classified based on how the network state is managed, as follows —

Horizontal distribution without state replication. Devolved Controllers [18] follow a horizontal distribution where each controller replica manages the subset of the network, but none of the controllers have a complete network-wide view. This class of controllers is useful when none of the control applications require a network-wide view.

Horizontal distribution with state replication. SDN controller designs like Onix [13], Hyperflow [14], ONOS (Open Networking Operating System) [19], and Beehive [20] replicate the controller state. All ONOS controller replicas maintain a consistent network-wide view so that any controller replica can serve any control plane request. In contrast, Onix, Hyperflow, and Beehive maintain the state that pertains to the assigned topology subset and implement replication to ensure failure recovery. Onix is a robust and scalable distributed control platform. It provides a programmable data structure (Network Information Base (NIB)) for application programmers to store controller state. Onix offers two kinds of data stores, replicated transactional database, and distributed hash tables (DHT), to support horizontal distribution and hierarchical distribution, respectively. Hyperflow uses multiple physically distributed NOX controllers, and a subset of data plane requests is assigned to each NOX controller. The NOX controllers use Hyperflow's publish/subscribe messaging system for inter-controller communication. ONOS has evolved from centralized Floodlight [44] SDN controllers. ONOS supports multiple physically distributed SDN controller replicas that are logically centralized. The ONOS framework implements state replication services and consensus techniques to ensure a consistent network-wide view and solve the single point of failure problem.

Beehive comes closest to our work, Cuttlefish. Beehive transforms a centralized controller application into a distributed system. In the case of Beehive, the application state is stored at any of the distributed controllers. Every controller runs an expensive synchronization protocol to maintain a consistent map for the application state. Beehive applications must query a globally synchronized index to determine the location of the state required for a particular computation. Locating the application state and packet migration requires multiple network stack and application stack traversals, leading to an increase in packet response times. In contrast, Cuttlefish routes messages by identifying the designated controller (where the application state is available) at the data plane switch itself, which reduces routing delays.

3. **Hierarchical controller design.** In horizontal distribution, the control plane load is distributed amongst multiple homogenous, physically distributed, and logically

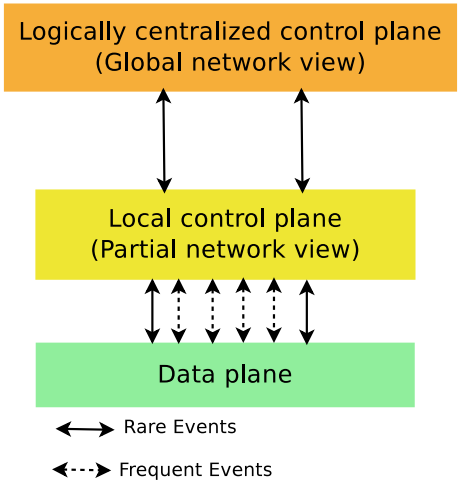


Figure 2.5: Hierarchical controller design

centralized controller replicas. However, the control plane requests have to travel from the switch to the controller, resulting in higher control plane response times.

Hierarchical controller techniques like [15–17, 21–26] scale controllers by offloading certain computations from the centralized root controller to local controllers that run at (or close to) the ingress switch. Researchers have observed that not all control plane messages require network-wide state for processing; certain messages only need local switch-specific state. Applications such as traffic engineering require network-wide statistics like flow type, cumulative queue length for each network path, and link utilization. These statistics are obtained from local switch counters like the number of active flows, per-flow sizes, average queue length, and average time spent by the packet at each queue; these statistics change frequently. In a hierarchical scaling approach (see Figure 2.5), we assign the tasks that require the network-wide state to the centralized root controller and the tasks that require the local switch-specific state to the local controllers (close to the switch). This results in reduced response latencies for offloaded computations and switch-controller network bandwidth savings.

Devoflow [15] and Kandoo [16] propose scalable flow management using a hierarchical distribution approach. The key idea is to offload some computations to local controllers or switches. These proposals expose the APIs for sampling, invoking triggers, and collection of approximate counters, which can be used by SDN programmers for applications like elephant-flow detection, multi-path forwarding, and fault-detection running at the centralized root controller. We now explain the approach using the traffic engineering application example from Kandoo. This appli-

cation implements elephant-flow detection. A flow with a large number of packets or many huge-sized packets is called an elephant flow. Such flows can lead to starvation of the other flows that follow the same path and require special treatment. The traffic engineering application comprises two components—(1) Detection of elephant flows, which requires flow-specific state like average per-flow packet size and flow length. This state is available at the switch (local state), (2) Reroute the flows, which requires the complete network topology state (global state). Hence, the traffic engineering application can offload the task of elephant flow detection to local controllers (close to the switch). The centralized root controller computes network routes when the local controller identifies an elephant flow and triggers the centralized controller. This decoupled computation setup results in a lower computation load at the root controller and reduces the network traffic between the switches and the root controller.

FOCUS [17] offloads a subset of local functions to the switch instead of local controllers. SDN application is written as a set of FOCUS rules that uses FOCUS APIs. A FOCUS rule comprises of a tuple <trigger, action-list> which is similar to Openflow's <match, action> paradigm. The FOCUS agent sits in between the switch OS and the Openflow agent and executes actions when the corresponding trigger is invoked. Like FOCUS, Eden [25] distributes the processing of control packets partially at the switch and the rest at the end host. Eden tags the packets with the message type, and the data plane switch decides whether the message is processed using the switch match+action tables, or the message should be forwarded to the host application for processing. Difane [21] controller dynamically generates a set of rules to satisfy network policies, and caches pre-computed forwarding rules across a subset of local switches, to avoid expensive communication with the controller when new flows arrive. The pre-installed rules comprise management tasks like access control, measurement, and routing at the intelligent switches. The data plane switch processes the incoming packets by applying the offloaded rules, reducing the load at the centralized root controller, and reducing response latency.

Hierarchical controller proposals like D-SDN [22] and B4 [23] provide additional services at the local controllers. D-SDN implements mechanisms for security and fault-tolerance for local controllers. B4 is a hierarchical controller for the traffic engineering (TE) application, where the ONOS-based local controllers collect local network information and pass this information to the global centralized controller. B4 implements fault tolerance at both the local and centralized controllers and applies Paxos for failure detection and recovery. Expresso is Google's hierarchical

controller framework that allows Google to dynamically choose the server location from where the content for individual users must be served. This decision is based on real-time measurements of end-to-end network connections.

Hybrid design. The hierarchical scaling approach is useful only when the control plane application performs frequent computations based on switch-specific local state. If applications require the network-wide state alone, then the horizontal scaling approach should be used. Proposals like Orion [47], SPARC [48], and MARS [49] implement hybrid scaling techniques to experience the best of both horizontal and hierarchical scaling techniques. Orion scales the routing application by offloading the static component to the local controllers. SPARC defines a high-level language for dynamic offload of policy processing between the horizontal and hierarchical controllers. MARS implements adaptive network management. It uses machine learning to allocate the load to the horizontal and hierarchical controllers.

Our proposals, Cuttlefish and TurboEPC, advocate a modified version of the hierarchical scaling framework. Apart from offloading the computations that require local switch-specific state, we also offload computations based on a subset of the global state, which we call offloadable state (defined at §1.3) to local controllers (Cuttlefish) or hardware programmable switches (TurboEPC). We call such computations as offloadable. The computations that are processed at the centralized root controller are called non-offloadable. The amount of computation offload is a lot more than the existing hierarchical controller proposals; therefore, many computations are taken away from the centralized controller, which leads to control plane scaling and a significant reduction in response latencies. Since our design offloads computations based on a subset of the global state to the local controller, we require synchronization of offloadable state between the root and local controllers.

State distribution frameworks. Some of the control plane scalability designs that we have discussed in this section require to maintain a consistent network-wide view. To do so, we require a framework for state distribution and management. The techniques used in our proposal (Cuttlefish) to manage distributed state across root and local controllers are similar to ideas used in frameworks [50–53] that manage the distributed state in networking applications. Split/merge [50] provides a state management API to applications for managing scale-up and scale-down operations. The state is transparently split between the middlebox (a network function VM) replicas for scale-up and merged to one replica for scale-down. OpenNF [51] improves split/merge by providing options for loss-free, and ordered state updates between the middlebox replicas. On the other hand, the goal of Pico replication [52] is to provide a low overhead, high availability framework for mid-

Table 2.1: Fields recognized by the OpenFlow standard [4].

Version	Date	Header fields
OF 1.0	Dec 2009	12 fields (Ethernet, TCP/IPv4)
OF 1.1	Feb 2011	15 fields (MPLS, inter-table metadata)
OF 1.2	Dec 2011	36 fields (ARP, ICMP, IPv6, etc.)
OF 1.3	Jun 2012	40 fields
OF 1.4	Oct 2013	41 fields

dleboxes. To dynamically grow or shrink the number of SDN controllers, Elasticon [53] proposes a switch migration protocol and enables load shifting between controllers. Some ideas of Cuttlefish, including the state management API and the protocol to guarantee ordered message delivery when migrating between controller modes, have been inspired by this body of literature.

2.3 Dataplane programming

The data plane is a forwarding element that processes packets. It takes a packet as an input, matches the packet header fields with the forwarding table rules to derive the corresponding action (forward/modify/drop), and decides where to send the packet. A traditional data plane hardware device (e.g., fixed-function switches) comprise of a dedicated ASIC (Application-Specific Integrated Circuit) with the packet forwarding logic fixed at design time and has configurable forwarding tables. A typical ASIC is designed to process standard packet headers like Ethernet, IP, VLAN, and GRE. The packet header structure and the stages of the packet processing pipeline are also fixed at design time. The action to be taken on a rule match is chosen from the fixed set of actions defined at design time.

The data plane programmer may want to define a custom wire protocol, or a custom packet header structure, or a custom encapsulation technique, or add more packet processing actions to the fixed-function devices. For such customizations, the data plane programmer must approach the hardware vendor who would take a few months to provide you with the new device. Such upgrades to fixed-function devices lead to wastage of both the time and resources, and lacks flexibility. We want the data plane programmer to have the ability to define the packet processing logic independent of the underlying hardware. The programmer should also have the ability to reprogram with a different logic on the same device.

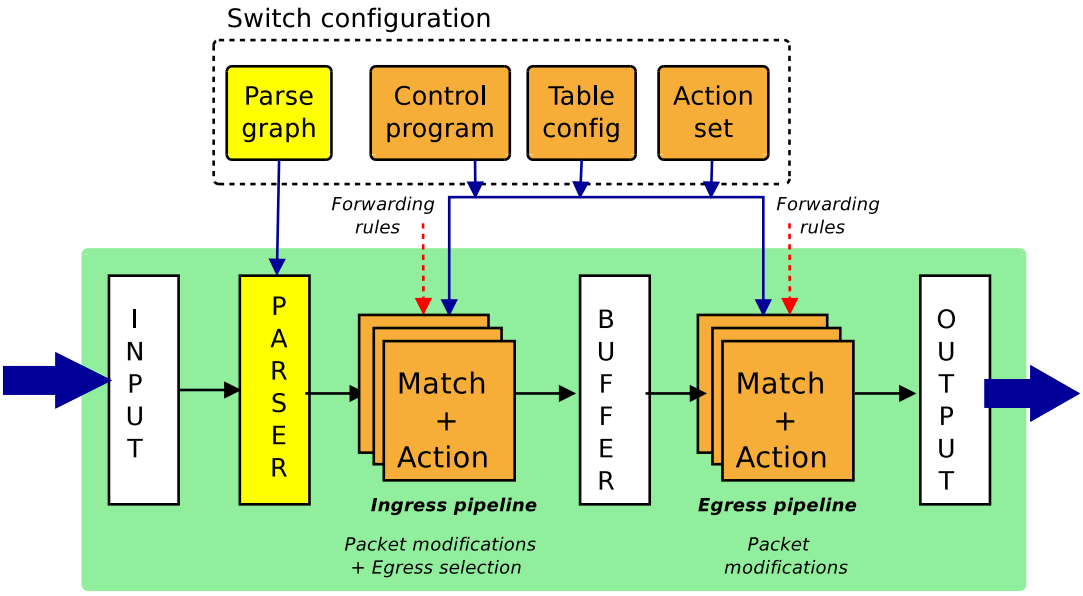


Figure 2.6: The abstract forwarding model.

2.3.1 A step towards dataplane generalization

Software-defined networks introduce network programmability with the use of the OpenFlow [42] API to configure and program the data plane devices. The OpenFlow standard explicitly specifies the set of protocol headers that can be configured for the data plane device. Table 2.1 depicts that the set of header fields defined by OpenFlow standard has been growing over the years (from 12 to 41 fields), thereby increasing the complexity of the OpenFlow specification. The OpenFlow standard improves the flexibility as compared to fixed-function devices, but does not provide the flexibility to the programmer to add new fields to the OpenFlow specification. The requirement of new headers and header fields is increasing. For example, the data centers continuously evolve their packet encapsulation techniques (e.g. NVGRE [54], VXLAN [55], and STT [56]), and these techniques may not be supported by the existing data plane switches. Therefore, data centers fall back to the software switches like Open vSwitch [57], OVS-DPDK [58], BESS [59], VPP [60], and VALE [61]. These switches are unable to run at line-rate of 100s/1000s of Gbps hence become a performance bottleneck for simple data plane forwarding.

The networking research community has proposed programmable switches [4], and Figure 2.6 depicts the abstract model for such switches. Now, we describe the workflow of the abstract forwarding model. The packet incident at the switch ingress is first handled at the parser. The packet body is assumed to be stored in the device buffers and is unavailable for matching. The parser extracts the fields from the packet headers

as defined by the programmable parser. The extracted header fields are then passed to the ingress and egress match+action tables. The match+action tables perform matching with the rules in the table, and the corresponding action is processed. Both the ingress and egress match+action modify the packet header, but the ingress determines the egress (output) port and the output queue for the packet. The match+action tables can be arranged in series, parallel, or combination of both. The next stage of the packet is determined by the output of the previous match+action table. Based on the ingress processing, the packet may be forwarded, dropped, replicated, or recirculated. Packets can carry additional information between stages, called metadata, which is treated identically to packet header fields. Some examples of metadata include the ingress port, timestamp that can be used for packet scheduling, and user-defined data that the user wishes to pass between the tables.

Two types of operations control the forwarding model: Configure and Populate. Configure operations include programming the device parser, setting the order of match+action stages, and specifying the header fields processed by each stage. Configuration determines which protocols are supported and how the switch may process packets. Populate operations add (and remove) the entries to the match+action tables that were specified during the configuration. Match+action table entries determine the policy applied to packets at any given time.

In contrast with OpenFlow, the abstract forwarding model generalizes the following— (1) OpenFlow assumes a fixed parser, whereas this abstract model supports a programmable parser that allows new header/protocol definitions. (2) OpenFlow assumes match+action tables in series, whereas this abstract model provides flexibility. (3) The abstract model allows the definition of new actions that are protocol-independent and supported by the switch. The abstract model generalizes how packets are processed in different forwarding devices like Ethernet switches and routers. These devices can be either fixed-function switch ASICs, Network Processor Unit (NPU), reconfigurable switches, Field-Programmable Gate Array (FPGA), or the software switches.

The abstract forwarding model forms the basis for recent programmable hardware switches [36, 37] that can forward traffic at terabit speeds while being highly customizable. Programming the hardware is not easy since each hardware executes the low-level machine language. But, the generalization provided by the abstract forwarding model is useful in designing a high-level language for programming the data plane devices, *P4 (Programming Protocol-independent Packet Processors) [4]*.

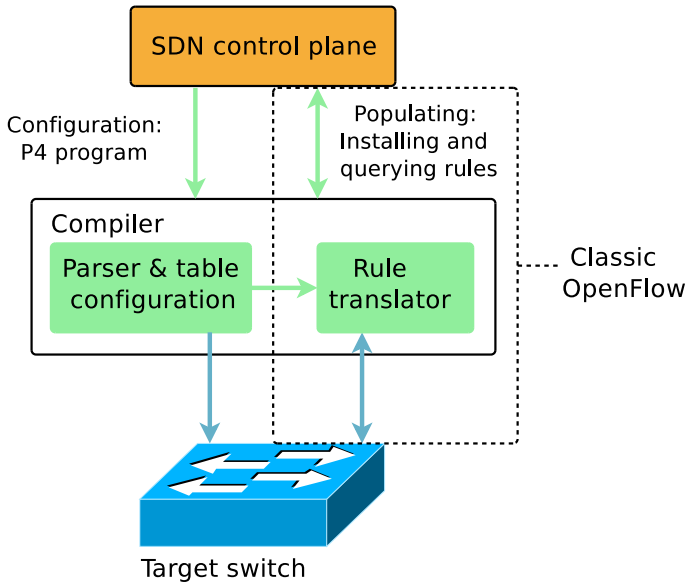


Figure 2.7: P4 is a language to configure switches.

2.3.2 Dataplane programming tools — P4 and P4Runtime

P4 is a domain-specific language that defines the data plane pipeline formally. P4 can be used to program network devices like the programmable hardware ASIC (Intel Flexpipe [62], Cisco's Doppler [63], Cavium's Xpliant [64], Barefoot Tofino [36]), NPU (Netronome Agilio CX [37], EZchip [65]), FPGA (Xilinx [66], Altera [67]), and CPU-based software switches (Open Vswitch [57], eBPF [68], DPDK [69], VPP [60]). P4 can describe fast pipelines for hardware data plane targets (or devices) and slow pipelines for software data plane targets. It can be used to program both programmable devices and fixed-function devices. Figure 2.7 shows how P4 and Openflow can program the data plane targets [4]. A user writes the data plane program for the target in P4 language that specifies both packet processing and the initial match+action table configurations. In contrast, the vendor-specific or open APIs like OpenFlow are designed to populate the forwarding tables alone, since the packet processing logic is fixed at design time. The data plane components that can be customized using P4 are as follows —

- **The packet processing pipeline.** The programmer can define the lifecycle of any packet by specifying a custom set of match+action stages that a packet has to pass through.

- **The packet parser.** The programmer can define a new packet header format and define a packet parser for the same.

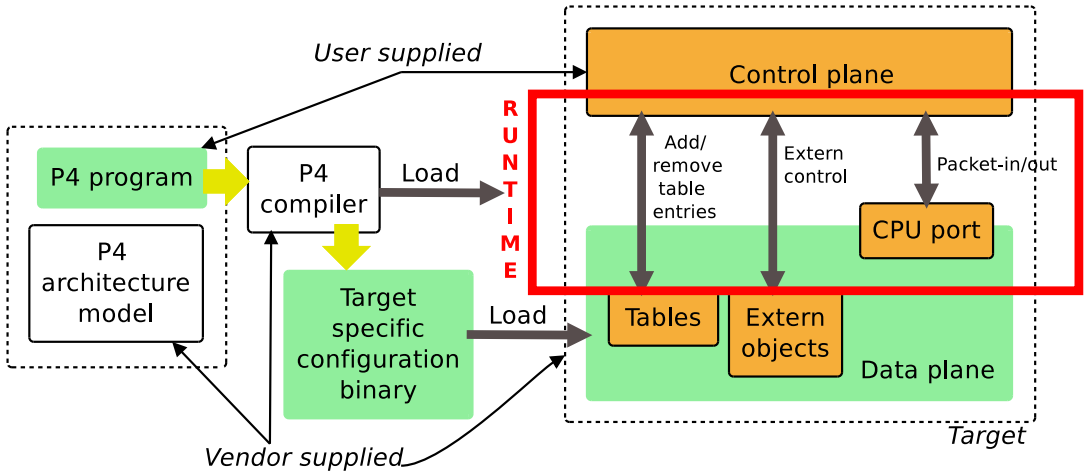


Figure 2.8: Programming a target with P4.

- **The match+action tables.** A traditional switch has configurable forwarding tables. But with the programmable switch hardware, the programmer can define custom tables that are used to match packet metadata.
- **The actions.** The programmer can define custom actions beyond typical 'forward' and 'drop'.

P4 has three main goals—

- **Reconfigurability.** The data plane programmer should be able to reconfigure the data plane packet parsing and processing via the controller.
- **Protocol independence.** The switch should not restrict itself to a fixed set of protocols and packet formats. The programmer should be able to define custom packet formats, custom actions, and custom match+action tables via the controller.
- **Target independence.** The data plane P4 program should be agnostic to the underlying programmable hardware target (or device), i.e., the same P4 program can be compiled for different data plane targets. The vendor-defined compiler should consider the capabilities of the target switch, and convert a target-independent description (written in P4) into a target-dependent program (used to configure the switch).

Figure 2.8 shows a typical workflow when programming a device using P4. Devices vendors provide the software runtime framework for dynamic communication between the control and data plane (example, p4Runtime), the architecture definition, and a P4 compiler that translates the P4 code to the target-specific binary code and configuration.

P4 programmers write the data plane programs (with the target capabilities and limitations in mind) that describe the working of the P4-programmable device components, and their external data plane interfaces.

The P4 compiler compiles a set of P4 programs and generates two artifacts [70] — (1) A data plane configuration like the packet parser and match+action table configuration. (2) An API to manage the data plane state from the control plane. The API is used to insert, update, or delete table entries, provide serialized packet transfer between control and data plane (packet-in/packet-out), and modify vendor-specific extern objects.

Compared to the traditional fixed-function packet processing systems, P4 provides the following benefits [70] —

- **Flexibility.** P4 makes many packet forwarding policies expressible as programs.
- **Expressiveness.** P4 can express sophisticated, hardware-independent packet processing algorithms using basic operations and table look-ups. Such programs are portable across hardware targets that implement the same architectures (assuming sufficient resources are available).
- **Resource mapping and management.** P4 programs define the storage resources abstractly as variables (e.g., source MAC address). The compilers map such user-defined fields to available hardware resources and manage low-level details such as allocation and scheduling.
- **Software engineering.** P4 programs provide important benefits such as type checking, information hiding, and software reuse.
- **Component libraries.** Component libraries supplied by manufacturers can be used to wrap hardware-specific functions into portable high-level P4 constructs.
- **Decoupling hardware and software evolution.** Programmable device manufacturers may use abstract architectures to decouple further the evolution of low-level architectural details from high-level processing.
- **Debugging.** Manufacturers can provide software models of the architecture to aid in the development and debugging of P4 programs.

P4Runtime: a control plane API to configure the data plane.

We have discussed the utility of the P4 programming language. It helps in describing the working of the data plane components at a higher level of abstraction. We now describe *P4Runtime* [71], a target-independent and architecture-independent control plane

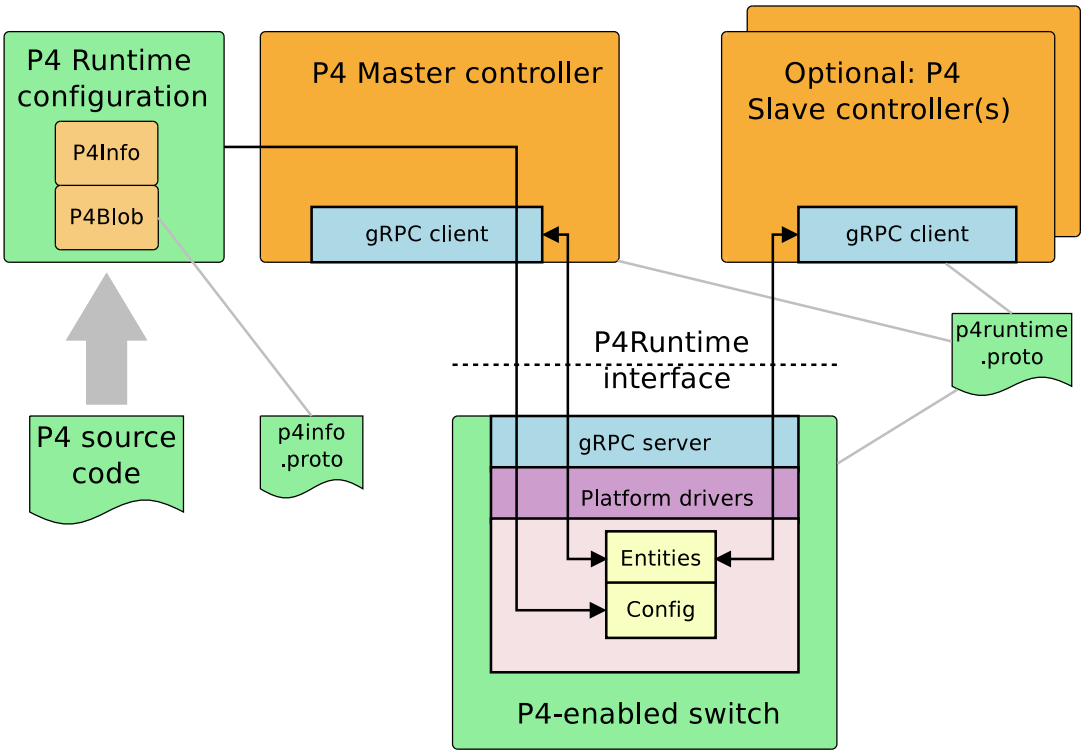


Figure 2.9: P4Runtime reference architecture.

API that enables the communication between the control plane and the data plane. The P4Runtime API is used to configure the packet processing pipeline of the data plane, initialize the data plane entities like the match+action tables, and update the data plane entities at runtime from the controller. We now discuss the components of the P4Runtime reference architecture, as shown in Figure 2.9.

The P4-enabled (programmable) switch can be controlled by one or more controllers, using the multi-controller protocol implemented by P4Runtime. This multi-controller feature helps to build distributed and high-availability P4Runtime controller designs. To avoid the race conditions, we can have only one controller authorized with write access to any data plane entity (e.g., tables, counters, and meters), and the switch pipeline configuration. This controller is called the *master controller*. The rest of the controllers have read-only access and are called *slave controllers*. If the master controller fails, one of the slave controllers is elected as the master. A role-based arbitration scheme is implemented by P4Runtime to manage controller roles.

The P4Runtime API defines the messages and semantics of the interface between the client(s) (controller) and the server (switch target). The client-server communication uses the *General Purpose RPC (gRPC)* [72] protocol. The P4Runtime API is specified by the Protobuf file, *p4runtime.proto*, which is available on GitHub as part of the standard [73]. The p4runtime.proto file is compiled using the Protobuf compiler to produce both client

and server implementation stubs for a variety of languages. It is the responsibility of target implementers to instrument the server. Server implementations for some of the P4 target devices that support P4Runtime are available at the p4lang/PI GitHub repository [74].

We now discuss the necessary components to configure a P4-enabled switch. P4 compiler backends are developed for each unique target by the device vendors. The P4 compiler compiles the user-defined set of P4 programs. It ensures that the code is compatible with the specified target and rejects the incompatible code. The P4 compiler generates as output, the P4 device configuration (device-specific), and the metadata (target-independent as well as architecture-independent). The metadata describes the overall program as well as all entity instances (tables and extern instances) derived from the P4 program. The P4 compiler assigns each entity instance unique numeric ID. This identifier is used as a "handle" by the P4Runtime API calls to access and manage the entity instances. The map of the entity IDs with the P4Runtime entity messages is referred to as the "P4Blob".

The ForwardingPipelineConfig captures data needed to realize a P4 forwarding-pipeline and map various IDs passed in P4Runtime entity messages.

P4Runtime controller takes the output of the P4 compiler for the initial configuration of the data plane target. A P4Runtime controller chooses a configuration appropriate to a particular target device and installs it via a *SetForwardingPipelineConfig* RPC. A controller can also query the device configuration from the target via the *GetForwarding-PipelineRequest* RPC. The pipeline configuration obtained from a running device helps to synchronize the controller to its current state. We can also use P4Runtime API with fixed-function devices. The controller does not program the target with the device configuration but uses the P4info file that describes the P4Runtime API messages to configure the device. As part of the business requirement, some device vendors may want to keep the P4 source code private. In such cases, the controller only needs a P4info file to render the correct P4Runtime API, and remotely configure the target device.

We have seen that the P4Runtime API provides a flexible architecture for communication between the control and data planes. P4Runtime is currently in its adolescent phase and is likely to provide many more exciting features in the future.

2.3.3 In-network application computation

The focus of in-network computing is to compute tasks within the network, using the existing network forwarding devices. The recent research on programmable network hardware (ASICs, NPUs, and FPGAs) allows the programmer to implement and accelerate the user-space applications by offloading application computations to the hardware.

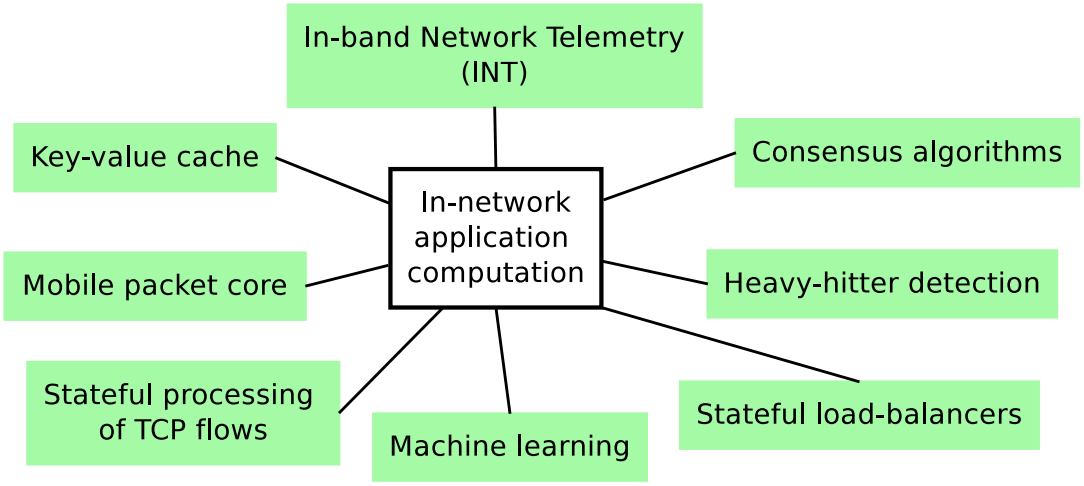


Figure 2.10: Classification of literature on in-network application computation.

High-level languages like P4 can be used to write the application code that can be compiled for the programmable forwarding hardware. The availability of high-level programming language and compilers for the target hardware acts as a catalyst to attract programmers to explore in-network computing. Next, we discuss the existing body of work (see Figure 2.10) where applications achieve significant performance benefits by offloading the computations to programmable data plane hardware.

One of the popular applications is the *In-band Network Telemetry (INT)* [75]. INT is an abstraction where the data packets at the switch query the switch state, such as queue size, link utilization, and queuing latency. This switch state is appended to the packet by every switch on the network route. When the packet reaches the designated destination, the network switch state appended to the packet is forwarded back to the monitoring node. Such real-time state monitoring can help the evolution of use-cases like network congestion control, traffic engineering, link failure detection, and verification of network flows. Marple [76], an alternative to INT, presents a query language with familiar constructs like map, filter, group-by, and zip for network performance monitoring. The query is compiled and run at the data plane, and the statistics are sent back to the controller. Another popular use-case implemented at the data plane is the heavy-hitter flow detection [77–79]. Flows with large traffic volumes are called as heavy-hitters. Identification of heavy-hitters is essential for several applications like flow-size aware routing, DoS detection, and traffic engineering.

Data centers typically employ hundreds or thousands of servers to load-balance incoming traffic over application servers. Some researchers implement the stateful load-balancers [80–82] within the data plane, which reduces the latency and saves server resources. Some researchers [83, 84] have implemented the slow consensus algorithms

like Paxos to the data plane to accelerate them. Solutions like Netcache [85] and KV-Direct [86] accelerate the access to the key-value store by caching the hottest key-value entries at the data plane switches. The high-speed switches process the high frequency read queries for these entries at line-rate, and the load on the key-value servers reduces significantly. Solutions like Blink [35] and Wharf [87] detect link failures at the data plane to quickly detect major traffic disruptions and early failure recovery. AccelTCP [88] implements a portion of the TCP stack on the data plane switches to accelerate the stateful processing of TCP flows. SwitchML [89] saves the network bandwidth by offloading the task of workflow aggregation to the data plane switches for machine learning applications. Few proposals [90, 91] offload the GTP header encapsulation and decapsulation processing to the data plane edge switch of the mobile packet core. Molero.E [92] demonstrates the possibility of accelerating the control plane functions like failure detection/notification via offload to the programmable data plane. We observe that data plane programming research has provided acceleration benefits to a wide range of applications.

Our work TurboEPC takes this line of work one step further and proposes the offload of frequent and offloadable signaling mobile packet core procedures to the programmable switches. We provide a guide that helps identify the offloadable computations in Chapter 3.

2.3.4 Can we offload any application to programmable hardware dataplane?

After observing the benefits obtained by the existing literature, you may want to ask the question — Can I offload my application computations to the data plane hardware? To answer this question, we present Table 2.2 that lists the limitations (based on the hardware constraints) [70, 77] on the type of computations that can be processed on the programmable data plane device. If we can program (and compile) the application considering these limitations, we can offload it.

To get the sense of what application computations can be offloaded, let us understand the constraints faced by some of the existing offloaded applications (Figure 2.10).

- Applications that offload the key-value store operations (Netcache [85] and KV-direct [86]) observe limited memory constraints. The data center comprises billions of key-value items, but we have on-switch storage for about 64K entries [85]. Also, the system is limited by the number of write operations due to low switch table update rates (point 9, Table 2.2). These systems use intelligent algorithms to utilize the limited resources optimally.

Programmable dataplane hardware constraints

1. The programmable dataplane hardware has memory limitations.
   (a) There is limited amount on-switch memory (10s of MB [36, 37, 93]).
   (b) There a limit on number of per-packet accesses to memory storing state at each pipeline stage (typically just one read-modify-write).
   (c) There is a limit on amount of memory available per pipeline stage.
   (d) Register memory in one stage cannot be accessed by a packet at other stages, to avoid hazards caused by concurrent accesses by packets at different stages [94].
2. There is a limit on the number of packet processing pipeline stages.
3. There is a limit on the time budget (1 ns [95]) to manipulate state and process packets at each stage. This is to ensure line-rate performance.
4. There can be no stalls during packet processing since the data plane switch commits to line-rate performance, i.e., the packet cannot wait at any stage for data or completion of other tasks, it has to move from one stage to the other at every clock tick.
5. We can move most packets just once through the each pipeline to avoid stalls and reduced throughput.
6. The code should not have any loops, and it cannot be recursive. This constraint ensures a deterministic number of pipeline stages and hence adheres to the line-rate performance commitment.
7. The match field items within the parsed packet should not be encrypted.
8. The application state can be stored at the switch tables or registers, but they have a limit on the maximum width.
9. The update rate at the commodity switches is much lower compared to the commodity servers. For example, Noviflow switches [93] support table update rates of 10K entries per second.
10. The target may not support all possible arithmetic operations. For example, operations such as multiplication, division, polynomials or logarithms may not be supported. Some architectures may only support multiplication with small constants, or shifts with small values due to operand constraints.
11. Floating point arithmetic is not supported.
12. Arithmetic operations may be supported on an integer number of bytes due to alignment and padding constraints.

Table 2.2: Programmable dataplane hardware limitations.

- Offload of heavy-hitter detection requires the switch to maintain state over multiple packets at line-rates of 10-100 Gbps. The application needs to maintain the state for millions of flows, and the state manipulations are pipelined over multiple stages (points 1–4, Table 2.2). Hashpipe [77] proposes new algorithms and data structures to offload the heavy-hitter application with an acceptable compromise for accuracy.

- Machine learning computations are difficult to offload as they require complex arithmetic such as multiplication, polynomials, and logarithms; they operate over floating-point data. An example machine learning component, feature extraction, requires the packet to iterate through the pipeline. These constraints refer to points 1, 5,10, and 11 of Table 2.2). Some machine learning solutions [89, 96] design intelligent techniques and offload selective computations to the dataplane. For ex-

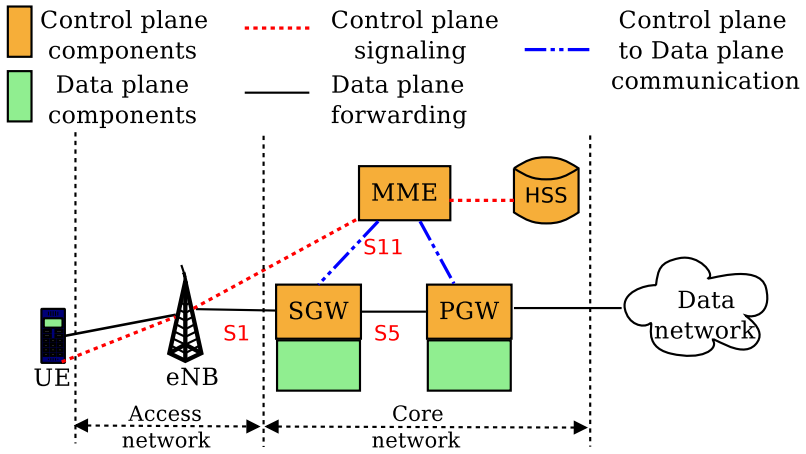


Figure 2.11: Traditional EPC architecture with unified control and data planes.

ample, switches do not support logarithm operations, but we can easily compute log(x * y) if log(x) and log(y) are known.

- TCP flow processing functions can be broadly classified as follows, based on the complexity. (1) Complex functions such as reliable data transfer (handling ACKs), packet retransmission on inferring losses, tracking received data for flow reassembly, enforcing congestion/flow control, and detecting errors. (2) Functions such as checksum offload, connection setup/teardown, and blind relaying of packets between two connections requiring no application-level intervention. The second set of functions are either stateless with a fixed processing cost or somewhat stateful. AccelTCP [88] selectively offloads the latter.

2.4 The mobile packet core

This section describes the use-case, the 4G LTE EPC (Long Term Evolution Evolved Packet Core) [97], which we have used to demonstrate the benefits of our control plane scaling designs, Cuttlefish and TurboEPC. We have chosen EPC because it is an example of a popular and complex SDN application covering all the state and compute patterns found in other applications too.

2.4.1 The mobile packet core architecture

Figure 2.11 shows the architecture of the traditional 4G mobile packet core. The "access network" is the part of a telecommunications network that gives the mobile user access to the telecommunications services. Multiple such access networks are connected together via the backbone which is known as the "core network". The core network also provides the gateway to the other networks. The mobile network core connects the radio

access network, consisting of user equipments (UEs) and the base stations (eNBs) with other packet data networks, including the Internet. The main components of the mobile network core and their basic functions are described below —

Home Subscriber Server (HSS). HSS is a global database that contains subscriber related information like the subscriber identifiers, the security keys for confidentiality and integrity, and the current subscriber location. HSS also provides support for mobility management, call and session setup, user authentication, and access authorization.

Mobility Management Entity (MME). MME is the control plane component of the EPC architecture. It is responsible for signaling between the eNBs and the core network. MME authenticates UEs using the subscriber security state stored with the HSS. MME keeps track of the UE's location and state, which helps handover of UEs between the eNodeBs. MME is also involved in bearer (user plane connection) activation and its deactivation procedures. It also chooses the SGW for a UE during UE registration, or relocation (handover). MME generates and allocates temporary UE identities, GUTI (Globally Unique Temporary Identifier), authorizes the UE, and enforces UE roaming restrictions if there are any. MME can also support Lawful Interception (LI) of user signaling.

Serving Gateway (SGW). SGW connects the radio (wireless) mobile network with the core network. It deals with the data plane function of forwarding the IP data traffic between the UE and the packet data network. It is also the anchor point in case of handover between the eNodeBs. The IP data traffic is encapsulated within the core network using the GPRS Tunneling Protocol (GTP). Since SGW is the component on the packet forwarding path within the core network, it has to implement GTP-based packet forwarding. For GTP implementation, SGW has to assign a unique identifier for each UE flow —Tunnel End-point Identifier (TEID), and also configure the forwarding, encapsulation, and decapsulation rules for the UE tunnels at the network switches. The TEID assignment and rule configuration operations require the SGW to intercept and process the signaling messages sent by the UE during the session establishment (or termination) process. Therefore, SGW implements both the control plane as well as the data plane functions.

Packet data network Gateway (PGW). PGW is the point of interconnection between the mobile core network and the external IP networks (a.k.a, Packet Data Network (PDN)). The functions of PGW are the same as that of the SGW. Additionally, the PGW also performs the functions of allocation of IP address and IP prefix, policy control, and charging. Similar to SGW, PGW also implements both the control plane as well as the data plane functions.

To enable independent scaling of the control and data plane logic in the S/PGWs, the later releases of 4G LTE espoused the Control and User Plane Separation (CUPS)

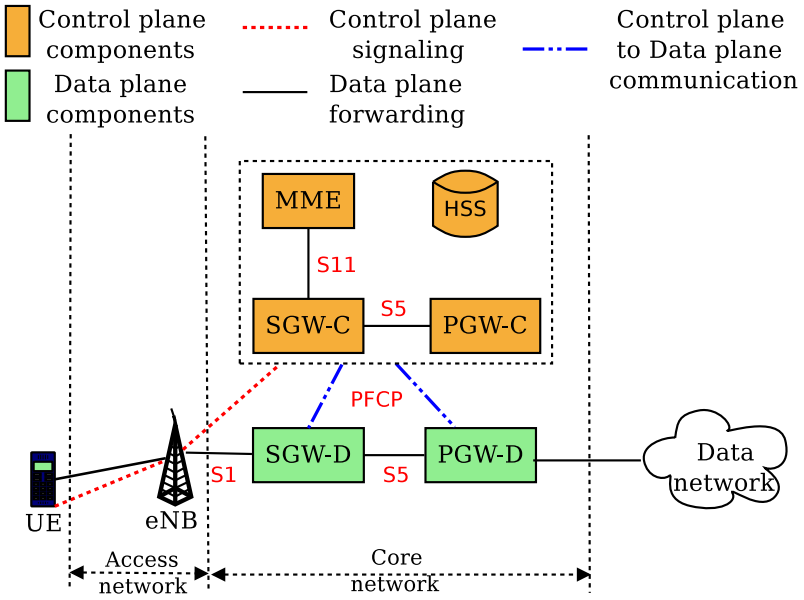


Figure 2.12: Traditional CUPS-based EPC architecture.

principle. The concept of the split of the control plane components from the data plane hardware switches is termed as software-defined networking in IP networks, whereas the telecom domain terms it as CUPS. Therefore, the CUPS-based EPC design benefits from the advantages of SDN design that we have seen in §2.1. Figure 2.12 shows the LTE EPC architecture with CUPS; the S/PGWs are separated into control and data plane entities, which communicate using a standardized protocol called PFCP (Packet Forwarding Control Protocol [98]). The eNB communicates with the SGW over the S1 interface in the user (or data) plane. The MME and the SGW communicate over S11 interface in the control plane, whereas the SGW and the PGW communicate over S5 interface in the user and control planes using GTP-U and GTP-C protocol respectively.

The upcoming 5G standard fully embraces the CUPS principle, as shown in Figure 2.13. In the 5G core, the Access and Mobility Management Function (AMF), Session Management Function (SMF), and other components handle signaling traffic in the control plane. In contrast, the User Plane Function (UPF) forwards traffic in the data plane. The control and data plane components once again communicate via PFCP.

In this thesis, we base our discussion of Cuttlefish, and TurboEPC on the CUPS-based EPC architecture shown in Figure 2.12. We assume that the MME and the control plane components of the S/PGWs are implemented atop an SDN controller, and the data plane of the S/PGWs is implemented in SDN switches. Our ideas easily generalize to the 5G architecture, as well as other CUPS-based EPC implementations, e.g., if the control plane components were to be standalone applications.

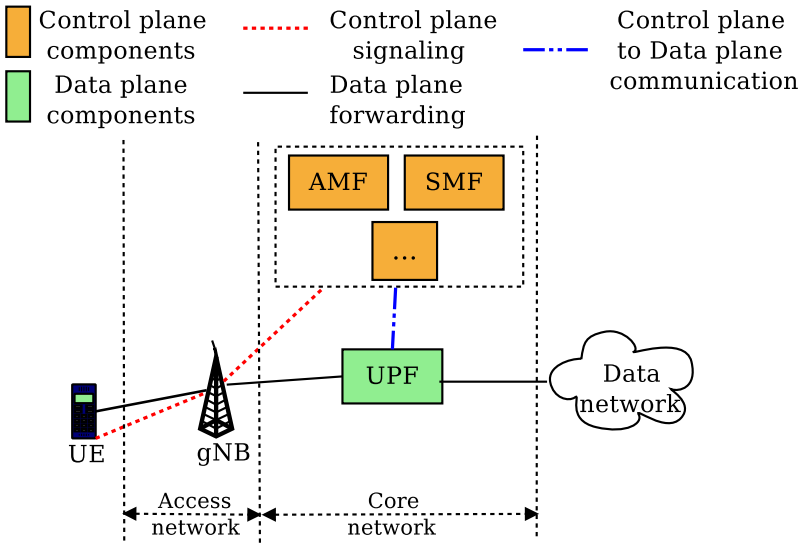


Figure 2.13: The 5G mobile packet core architecture.

2.4.2 The LTE EPC procedures

We first describe the process of forwarding the IP data packets between the UE and the packet data networks (Internet) in a mobile network. The packet routing from the UE up to the packet data network is the responsibility of the components that implement the data plane of the cellular network, viz., eNB, SGW, and PGW. The mobile network uses the GTP protocol that encapsulate the user data packets between the mobile core entities when passing through the mobile packet core.

GTP over traditional IP-based routing. GTP is used over traditional IP-based routing since it provides several benefits —

- It is hard to handle user mobility using IP-based routing since the IP address changes with the location, and the data packets on the route are dropped. In contrast, in the case of GTP, when the UE is mobile, its IP address remains the same as it is not used for forwarding. The packets are forwarded using GTP tunnel identifiers provided between the PGW and eNB via the SGW. Hence, GTP provides mobility.

- A single UE can use multiple tunnels to obtain different network QoS.

- UE's IP address remains hidden, so tunneling provides security.

GTP-based tunneling in mobile networks. GTP packets can be of three types, GTP-C, GTP-U, and GTP'. GTP-C is the control part of the GTP, and it is used for core network signaling like bearer activation, deletion, or modification. GTP-U is used in the user

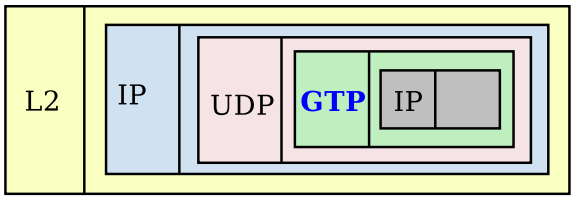


Figure 2.14: Encapsulated GTP packet.

plane to carry user traffic in mobile networks. GTP' has the same structure as GTP-C and GTP-U, but it is used to carry charging (billing) information within the mobile network.

We describe the GTP packet (refer Figure 2.14) to further understand the process of tunneling data packets within the mobile network. The GTP header consists of Tunnel Endpoint Identifiers (TEIDs) that uniquely identify the path of a user's traffic through the core. The S/PGWs in the core network route the data plane traffic based on the TEID values. Separate TEIDs are generated for both the datapath links (eNB-SGW and SGW-PGW), and for both the directions of traffic (uplink and downlink). When a user's IP data packet arrives from the wireless network at the eNB (uplink), it is encapsulated into a GTP packet, which is then transmitted over UDP/IP, first between the eNB and the SGW uplink tunnel, and then between the SGW and PGW uplink tunnel. The egress, PGW, decapsulates the GTP header before forwarding the user's data to external networks. The downlink packets destined for the UE that arrive at the PGW follow the reverse process.

User (UE) control and user plane connections. Note that we have simplified certain EPC-specific terms, for easier understanding of concepts. Figure 2.15 shows the connections established in user/control planes and the states maintained in the corresponding planes. To support traffic transmission between the user and the network (UE through PGW), the bearer (user plane) and signaling (control plane) connections are established. The radio bearer is the user plane tunnel (GTP-U) between the UE and the eNB, which is identified using bearer identifiers for both uplink and downlink communications. The S1 bearer is the user plane tunnel between the eNB and SGW that identified by the tunnel identifiers S1 SGW-TEID (uplink) and S1 eNB-TEID (downlink). The S5 bearer is the user plane tunnel between the SGW and PGW that is identified by the tunnel identifiers S5 PGW-TEID (uplink) and S5 SGW-TEID (downlink). The mobile network establishes the signaling connections for control plane communication between; (1) UE and MME using radio+S1 connection, (2) SGW and MME using S11 connection, and (3) SGW and PGW using S5 connection.

Connection states at the UE and MME. We now discuss the connection states at the UE and the MME under the active and idle UE conditions. When the UE is registered with

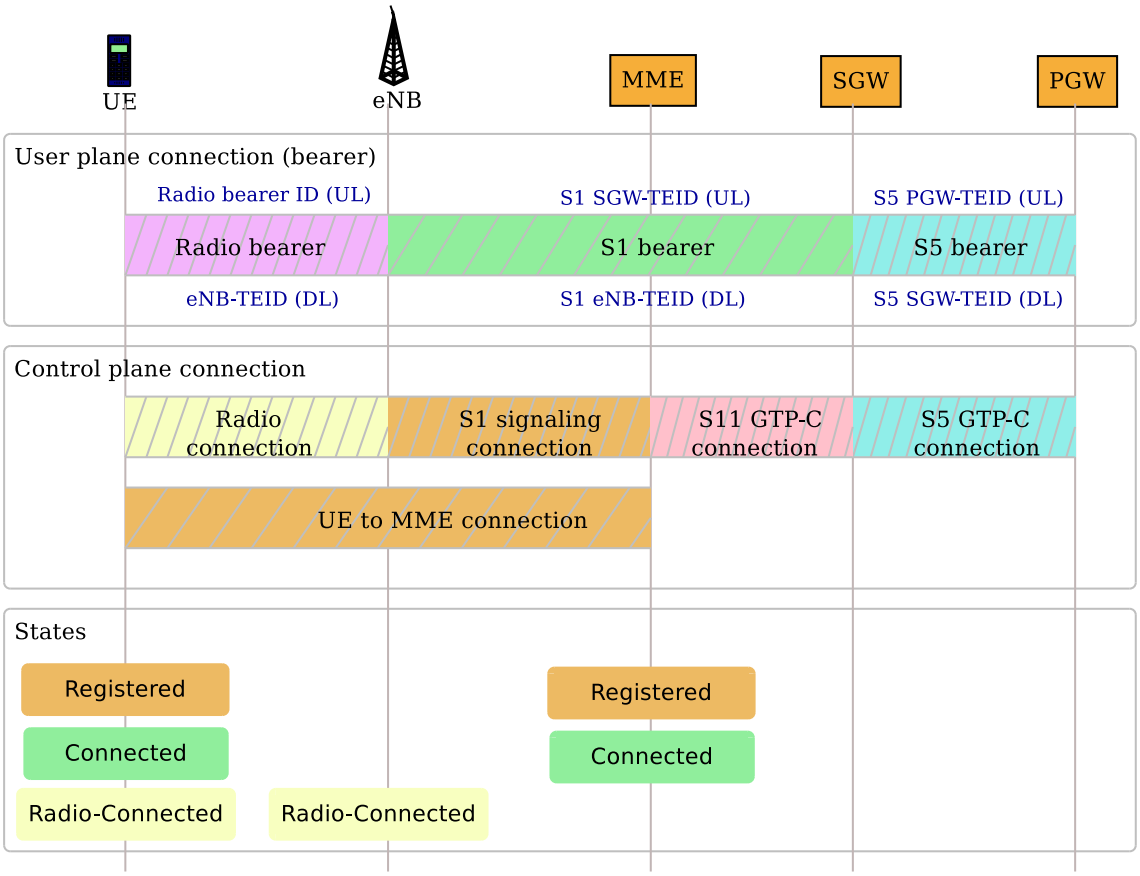


Figure 2.15: UE's connections and states in the EPC network.

the network and is active, the UE and the MME are in *Registered* and *Connected* state, while the UE and eNB are in *Radio-Connected* state. If the UE does not send any data for a few seconds (inactive), the UE is considered to be idle. Under idle conditions, the radio and S1 bearer resources are released. The UE and the MME are now in *Registered*, but *Idle* state. The UE and eNB are in the *Radio-Idle* state.

LTE EPC control plane procedures. A mobile user (UE) requires multiple services from the mobile network like network accessibility and seamless mobility. A *procedure* is a logical task or a service that the user needs from the EPC network. For example, after the UE is switched ON, it has to register with the network to access network services, and this is done using the "attach" procedure. In this section, we briefly discuss the LTE EPC control plane procedures listed in Table 2.3. We have not described the policy control functions carried out by the PCRF (Policy and Charging Rules Function), since we do not implement the PCRF function in our prototype. Note that the core network performs several other procedures beyond those discussed here; however, this description suffices to understand our work.

Attach procedure. When a UE connects to an LTE network for the first time, the initial message sent from the UE via the eNB triggers the *attach* procedure in the mobile packet

Table 2.3: LTE EPC control plane procedures.

Procedure	**Description**	**Workflow**
Attach	UE registers with the mobile network	Figure 2.16
S1 release	Deactivates the data channel (S1) when UE is idle	Figure 2.17
Service req	Activates the data channel (S1) when UE becomes active	Figure 2.19
Detach	UE is de-registered from the network	Figure 2.20
Handover	Manage the UE's connection when UE changes its location	Figure 2.21

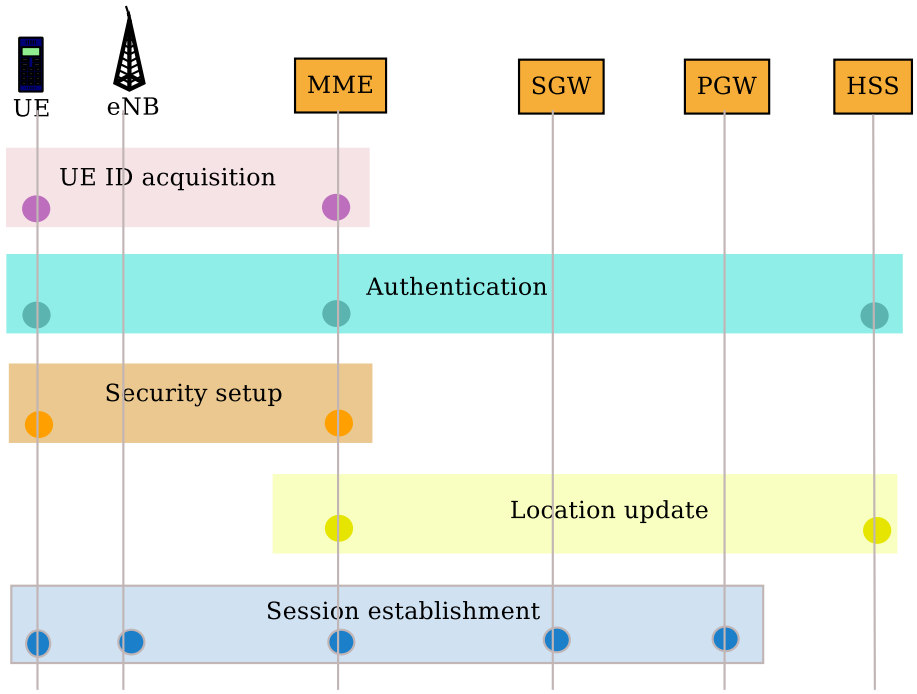


Figure 2.16: The attach procedure.

core. Figure 2.16 shows the components of the attach procedure, and the solid circles at each component indicate the EPC nodes that are involved in the processing.

1. The MME identifies the UE using the global identifier, IMSI (International Mobile Subscriber Identity), and learns about the security algorithms supported by the UE, and proceeds to the next step, authentication.

2. The UE and the network mutually authenticate each other using the user state stored in the HSS. The authentication procedure consists of the following two steps: (1) Authentication Vector (AV) acquisition (2) mutual authentication between the MME and the UE. AV comprises of (a) RAND, a random number used as a seed to the security algorithms, (b) AUTN (Authentication Token) used by the UE to authenticate the network, (c) XRES (eXpected Response) used by the mobile network to authenticate the UE, and (d) K_{ASME} (ASME: Access Security Management

Entity) is an intermediate master key used to derive the rest of the security keys. The HSS generates an AV by using the LTE master key in the IMSI, and sends it to the MME. The MME sends RAND and AUTN to the UE, but keeps XRES and K_{ASME} for user authentication and security key derivation, respectively. The UE generates the AV using the RAND, authenticates the network by comparing the generated AUTN with the AUTN sent by MME, and sends the generated response (RES) to the MME. The MME compares the XRES value with the RES sent by the UE to authenticate the user. The UE and the mobile network are now mutually authenticated.

3. In the NAS security setup phase, the MME selects ciphering and integrity algorithms and informs the UE about the choice of algorithms. Both the UE and MME independently derive the integrity key and the encryption key from K_{ASME}.

4. The MME sends UE's IMSI, and MME identifier to the HSS, to notify UE's successful registration and obtain UE's subscription information. The HSS registers the UE's location, and replies to MME with the message that includes: (1) The Access Point Name (APN) that the UE subscribes to, i.e., the data network name. (2) The subscribed PGW ID which determines the PGW through which the UE can access the subscribed APN. (3) Subscribed QoS profiles that contain UE's information like the minimum and maximum uplink/downlink bandwidth that the UE can have.

5. Finally, the MME sets up the state required to forward user traffic through the core at the eNB, SGW, and PGW that are on the path from the user to the external packet data network The MME allocates the network/radio resources such that the user's subscribed QoS is satisfied. The eNB, SGW, and the PGW set up the radio bearer, S1 bearer, and S5 bearer, uplink and downlink channels. PGW allocates an IP address to the UE to access the external network. All the connection states at the UE and MME are set to be active, i.e., *Registered*, *Connected*, and *Radio-Connected*. The UE is now successfully registered with the core network, and the UE's data channel is functional.

S1 release procedure. If the UE goes idle without sending data for a certain duration (usually 10-30 seconds [99]), a *S1 release* procedure is invoked to release the unused resources. Figure 2.17 shows the components of the S1 release procedure.

1. The SGW releases the S1 bearer, with all the uplink/downlink S1 bearer resources associated with the UE, but retains the uplink S1 SGW-TEID state. So, when the

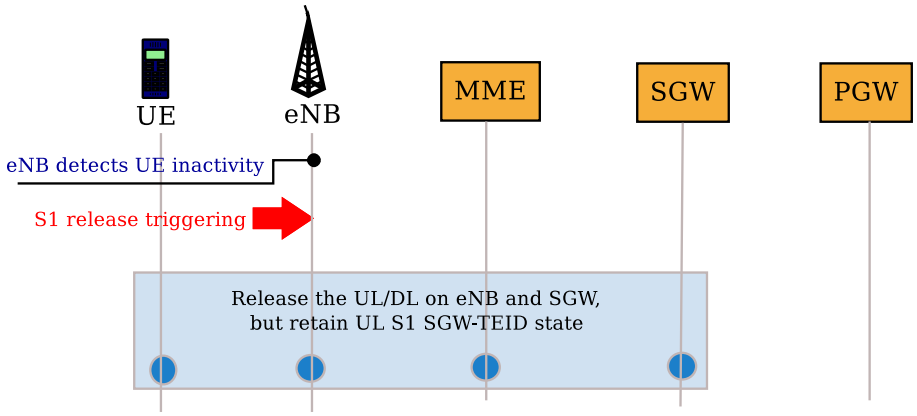


Figure 2.17: The S1 release procedure.

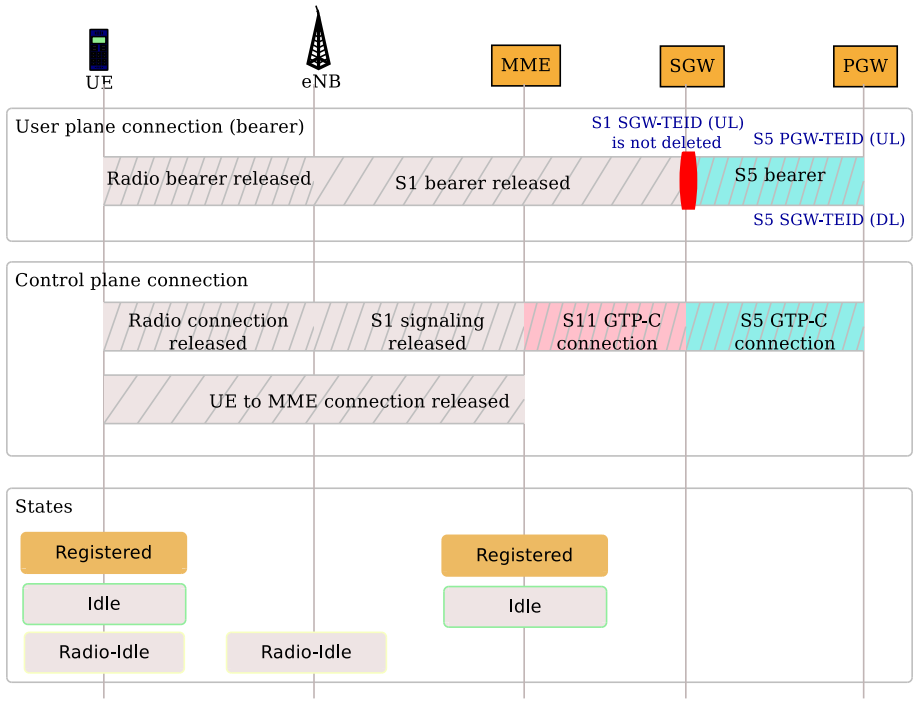


Figure 2.18: Connections and states in the EPC network after S1 release processing.

uplink packets arrive, the eNB can obtain the uplink S1 SGW-TEID from the MME (Service Request procedure), and deliver the packets through the S1 bearer without delay. After the release of eNB resources at the SGW, if downlink packets destined to the UE arrive, the SGW buffers them and delivers them only after the downlink S1 bearer is re-established.

2. The eNB deletes all UE context and releases the radio bearer channels.

3. The MME deletes all eNB related information (address and TEIDs) for the UE, but retains the rest of the UE's MME context, including the uplink S1 SGW-TEID. The MME updates the UE's connection state as *Idle*. With this, the S1 release procedure

is successfully completed. Figure 2.18 shows UE's state and its connection state after the processing of the S1 release procedure.

Service request procedure. During the S1 release procedure, the UE resources related

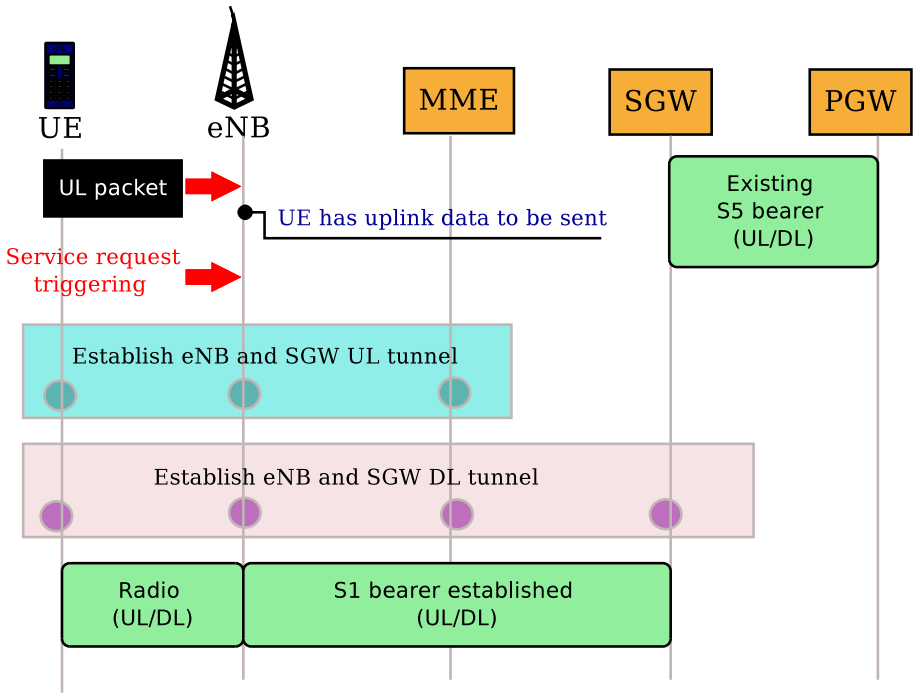


Figure 2.19: The service request procedure.

to data communication are released at the eNB and SGW of the core network. Later, when the UE becomes active (UE-initiated service request), or the network wants to send data to the UE (Network-initiated service request), we need to reassign the previously released bearer resources. In this thesis, we have implemented the UE-initiated service request. The details of network-initiated service request procedure are available at [97]. Figure 2.19 shows the components of UE-initiated service request procedure.

1. The MME and eNB perform the optional security setup procedures. An uplink path is set up from the UE up to the PGW, which includes the radio and S1 bearer.

2. eNB allocates downlink radio and S1 tunnel identifiers, and the downlink bearer is established. This allows delivery of downlink traffic from the PGW up to the UE.

3. The MME and the UE updates the UE's connection state as active (*Connected*). This marks the successful completion of the service request procedure.

Detach procedure. The detach procedure disconnects the UE from the network.

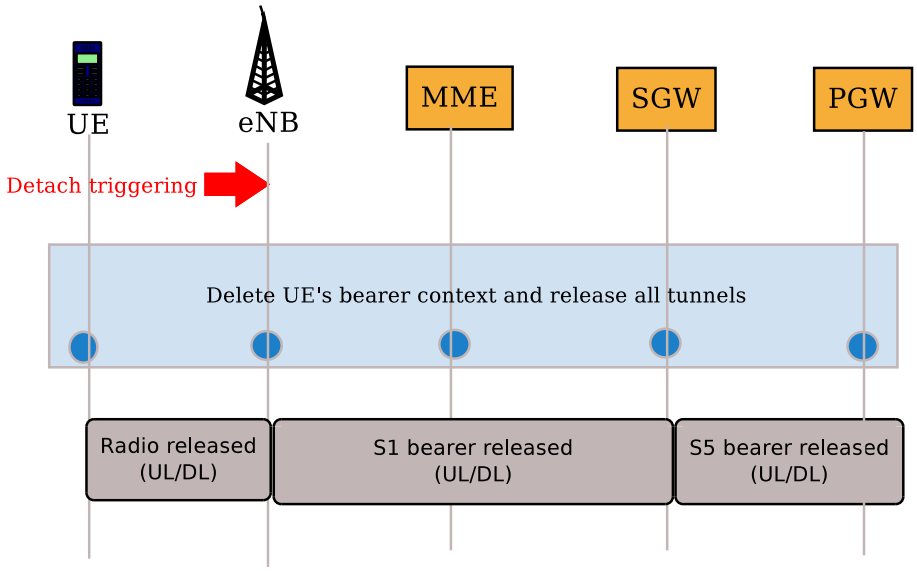


Figure 2.20: The detach procedure.

1. The detach procedure can be either initiated by the MME, or the HSS, or the UE. The MME initiates a detach procedure under the following conditions: (1) operator maintenance process, (2) authentication failure, (3) lack of resource availability, or (4) poor radio link quality. The HSS initiates a detach procedure if: (1) the user profile stored at the HSS has changed, so the profile at the MME has to be changed, (2) the operator is trying to restrict access to an illegal UE (stolen UE). The UE can initiate the detach procedure due to the following cases: (1) if UE is turned off, (2) if the USIM card is removed from UE, or (3) if UE is attempting to use a non-EPS service (e.g., SMS). Figure 2.20 shows the UE-initiated detach procedure.

2. The UE, MME, and the eNB release all the resources allocated to the UE, including the control and data connections and the UE context.

Handover procedure. When a UE moves from one network location to another, it triggers a *handover* procedure (refer Figure 2.21) in the core. The handover procedure is invoked under multiple cases: (1) eNB changes, but the SGW is the same (intra-SGW handover), (2) SGW changes, but the MME is the same (inter-SGW handover), and (3) MME changes (inter-city handover). We describe the inter-SGW handover callflow in Figure 2.21 as we have implemented this handover case. The other handover cases and their details are available at [97].

1. The UE measures the signal strength of its serving cell and neighbor cells. The measurement value is either periodically reported to the eNB, or when a measurement event is triggered. When the signal strength of a neighbor cell becomes higher than

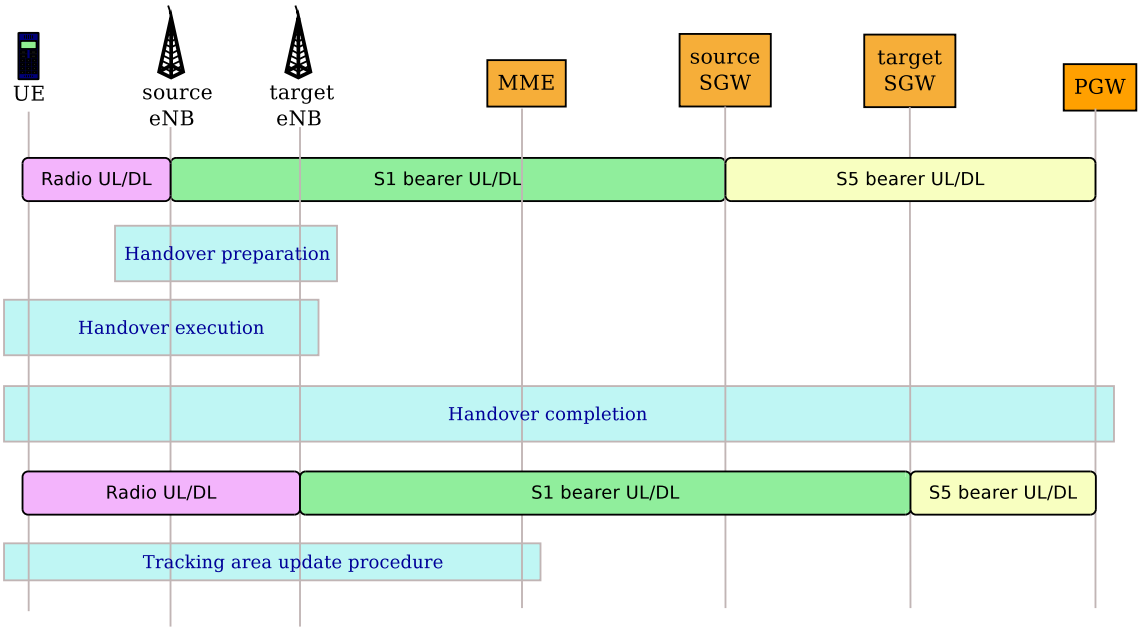


Figure 2.21: The inter-SGW handover procedure.

that of the UE's serving cell, and the difference is higher than the handover margin, the handover procedure is triggered by the eNB. The handover margin is used to avoid the ping-pong effect, and is calculated using hysteresis by the EPC network. The ping-pong effect refers to repeated handovers between the same two cells.

2. During the handover preparation phase, the source eNB and the target eNB prepare for a handover. The source eNB sends the UE's context to the target eNB. If the target eNB is capable of satisfying the service quality, it establishes a downlink packet forwarding bearer.

3. During the handover execution phase, the handover process is carried out. The UE disconnects the radio link from the source eNB and connects to the target eNB. The resources are allocated for packet forwarding between (1) the two eNBs, and (2) new resources for the UE are allocated at the target eNB for radio and downlink S1 bearer. The downlink packets for the UE are forwarded to the target eNB and are buffered there until UE successfully completes the handover.

4. The MME determines that the SGW is relocated and selects a new SGW. The target SGW allocates S1 SGW-TEID and S5 SGW-TEID, for the uplink and downlink traffic respectively. The MME confirms the path switch to the target eNB, and it starts using the new SGW's address for forwarding subsequent uplink packets. The target eNB informs the success of the handover to source eNB. The source eNB and MME release the resources associated with the old path.

5. The new UE location information is updated with the HSS. The UE is now successfully connected to the target location.

2.4.3 Scalability solutions for the mobile packet core

We have briefly introduced the mobile packet core application in §2.4.2. With the SDN (or CUPS) paradigm, a logically centralized software control plane of the mobile packet core can potentially become a performance bottleneck. We observe that the signaling traffic in the mobile network is growing rapidly [100, 101], fueled by smartphones, IoT devices, and other end-user equipment that frequently connect to the network in short bursts. In fact, the signaling load in LTE is 50% higher than that of 2G/3G networks [100], and would grow much more with the adoption of the 5G technology. This high signaling load puts undue pressure on the packet core, making it difficult for operators to meet the signaling traffic SLAs [102]. The research community has suggested several approaches to solve this control plane scalability challenge. We classify and explain the existing approaches and differentiate our work from them.

Horizontal scaling of mobile packet core. Some controllers like SCALE [103], MobileStream [104], and MMLite [105] use the technique of horizontal scaling, where the incoming control plane traffic is distributed amongst multiple homogeneous SDN controllers, which cooperate to maintain a consistent view of the shared global network-wide state amongst themselves using standard consensus protocols.

Optimizations to the EPC protocol. Mobile control plane scalability solutions like DPCM [106], CleanG [107], Pozza et al. [108], and Raza et al. [109] modify the EPC protocol, such that they reduce the number of messages exchanged between the UE and the core network, or some of the EPC messages that were processed sequentially, are now parallelized. Such optimizations reduce the overall turn-around time for EPC control plane message processing, and thereby improves EPC throughput and scalability. Solutions like the PEPC [110] and Heikki et al. [111] refactor the EPC state to reduce the interprocedural communication and the overhead of state transfer costs, to scale the mobile packet core.

Hierarchical scaling of the mobile packet core. All of the above scalability solutions run at the mobile core network, which is a few tens of milliseconds away from the user. All the mobile control plane messages have to travel to the core network, resulting in higher response time delays. To cope with this, DMME [112] and Balakrishnan et.al [113], use hierarchical scaling to offload the attach and handover control plane procedures to local SDN controllers that are located close to the eNB and the UE. Therefore, the response latencies for the offloaded functions are reduced by several orders of magnitude. Soft-

cell [114] proposes the solution to accelerate the 4G data plane forwarding via the offload of the packet route installation task to the edge switch. They further minimize the forwarding table size by aggregating the flow rules within the switch. While this work is primarily focused on optimizing the data plane processing, our work *TurboEPC* accelerates the control plane via the offload of signaling message processing to the edge switch.

Our proposal, TurboEPC is inspired by hierarchical SDN controllers. TurboEPC proposes refactoring of the mobile core intending to offload a subset of the control plane processing to programmable data plane switches closer to the end-user. However, it is quite different from them. First, we apply the idea of offloading computation from SDN controllers to data plane switches in the CUPS-based mobile packet core. Second, the traditional hierarchical SDN controllers offload the computations based on local switch-specific state, whereas TurboEPC also offloads computations that depend on specific type of global state. We classify the signaling procedures of the mobile packet core into two classes, based on the type of state accessed during the processing. For example, attaching a user to the network entails authenticating the user using a network-wide subscriber database, and setting up the forwarding path of the user under mobility requires access to the global network topology. The signaling procedures like the S1 release and service request access the user context of a single subscriber, and do not access any network-wide global state. The S1 release and service request procedures comprise ~63–90% of the total EPC traffic distribution. Hence, TurboEPC offloads the control plane processing of the frequent S1 release and service request procedures to the programmable switches, thereby providing significant throughput and latency benefits. This existing body of work is orthogonal and complementary to our work, and TurboEPC can leverage these control plane optimizations for the processing of non-offloadable messages at the root controller.

2.5 Summary

We have described the concepts of software-defined networking, data plane programming, and the mobile packet core. SDN concepts are necessary to understand all of our work, whereas the data plane programming concepts are necessary to understand our TurboEPC work. The mobile packet core application is the use-case that we use to demonstrate the effectiveness of both our ideas, Cuttlefish and TurboEPC.

While describing the basic concepts, we have also discussed the research problems and the solution approaches that prior work has considered. We have provided a clear differentiation of our work from the existing solution approaches in this chapter.

Chapter 3

State Taxonomy of SDN applications

We have discussed the SDN control plane scalability designs in Chapter 2. Our proposed designs are based on the hierarchical scaling approach, where we offload subset of the application computations and the corresponding state from the centralized root controller to the local controllers or switches. The programmer has to identify the application computations and states that can be offloaded to the local controllers, to achieve application scalability. The offload of the application state should not introduce state inconsistencies at the centralized root controller, and the application's correctness should not be compromised. So, it is necessary to have guidelines on what application states and computations can be offloaded locally.

In this chapter, we describe the application state classification proposed by the existing hierarchical scaling designs, but we believe that the classification is not complete. So, we propose a new state taxonomy for SDN applications and illustrate the state classification process for real-life examples. We also provide a comprehensive guide that helps the programmers to classify the application state.

3.1 Application state taxonomy

The hierarchical control plane scaling techniques like Devoflow [15], Difane [21], Kandoo [16], Eden [25], and FOCUS [17] offload the subset of the application computations from the centralized controller to the local controllers or switches, close to the user. Figure 3.1 shows the typical hierarchical scaling design where the control plane computations that require local state alone are processed locally, and the ones that depend on the global state are forwarded to the centralized controller for processing. This additional offload of control plane computations close to the user increases the control plane capacity and reduces the response latency.

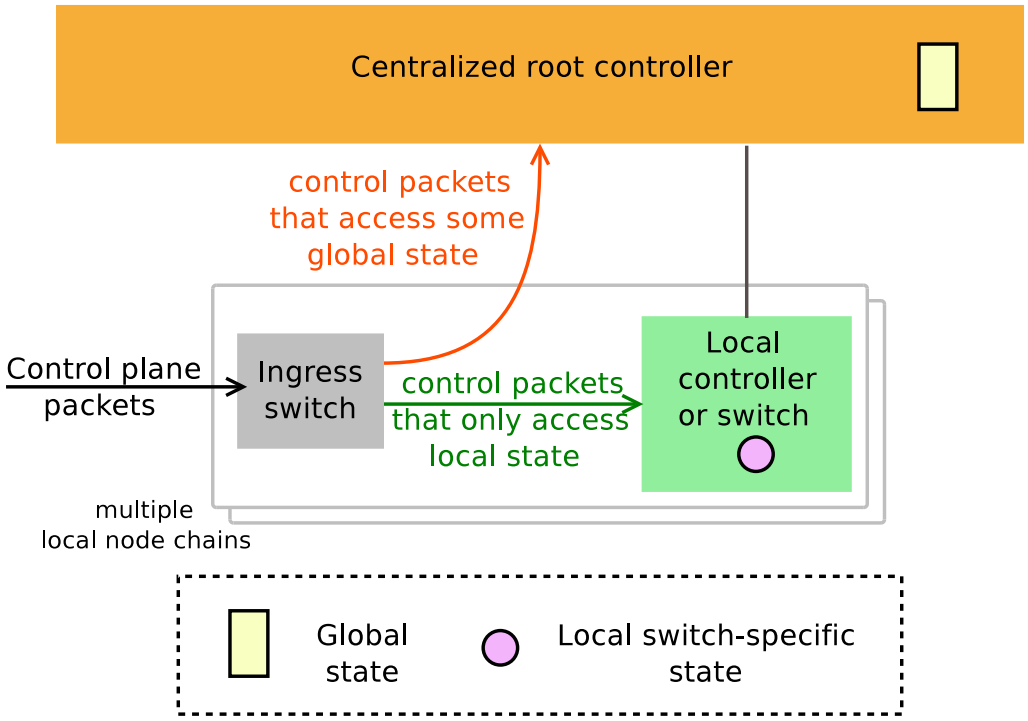


Figure 3.1: Hierarchical control plane scaling.

3.1.1 State taxonomy proposed by existing hierarchical solutions

To incorporate the hierarchical scaling approach in SDN control applications, the application programmer should identify the computations that can be offloaded to local controllers. For this purpose, the existing hierarchical solutions classify the application state as follows —

- **Local state.** The state that is either available at the network switches or can be derived from the switch state is called the local state. Let us take the example of the traffic engineering application that we discussed in §2.2. Among the other tasks, the traffic engineering application detects elephant flows. An elephant flow is a long-lived flow with a large number of packets or consists of many huge-sized packets. The network flows arrive into the network through the ingress switch, and the switch maintains the state for each flow. Some examples of the flow-specific state maintained at the switch include the flow length counters, average packet size, and counters for packets whose size is above a particular threshold (huge-sized packets). This flow-specific state is the local state for the traffic engineering application, and the application detects the elephant flows when one of the flow counters exceeds a threshold value. Therefore, the computations required to detect elephant flows require local state alone and can be offloaded to the local controllers or switches.

- **Global state.** The state that has a network-wide scope, and can be accessed concurrently from any network location is called the *global state*. For example, in case of the traffic engineering application discussed above, new routes are computed once an elephant flow is detected. The route computation process requires the global network topology information (global state). Such computations cannot be offloaded locally, so they run at the centralized controller. In case of LTE-EPC application discussed in §2.4, the application uses the global security key database, HSS, for mobile user authentication, confidentiality, anonymity, and integrity. The mobile user can be authenticated from any network location, and this state can be accessed concurrently from multiple network locations during the handover process. So, the HSS state is also an example of a global state, and the computations that depend on this state cannot be offloaded.

3.1.2 Our state taxonomy proposal

The control plane scalability designs proposed in this thesis, Cuttlefish and TurboEPC, are based on the hierarchical control plane scaling approach. Existing hierarchical scaling solutions offload computations based on the local state alone; while we take a step forward and offload computations based on some global state too. We propose a new state taxonomy for SDN applications as follows —

Non-offloadable state. The global network-wide state that can be accessed concurrently from multiple network locations is called the *non-offloadable state*. As the non-offloadable state can be concurrently accessed from multiple network locations (multiple edge switches), we maintain this state at the centralized root controller and assure the consistent view across locations. Any computation that depend on such state should be processed at the centralized controller. For application scalability, the non-offloadable state should be replicated consistently across horizontally scaled centralized controller replicas. The network topology state for the traffic engineering application and the HSS state of the EPC application that we discussed in the existing state taxonomy are examples of non-offloadable state.

Offloadable state. We observe that apart from the offload of computations that depend on the local switch-specific state, we can also offload certain computations that depend on the particular type of global state (e.g., session specific state), to the local controllers or switches. We define *offloadable state* as the state that is accessed from only a single network location (edge switch) That is, all the control plane messages access this state from the same network edge at a particular time. Examples of such state include switch-local state and some types of session-specific application state. The switch state like the

local counters (switch-specific) used by the traffic engineering application, is one of the examples of the offloadable state. We will now discuss more examples of offloadable state that are not just switch-local state.

The Open Networking Foundation (ONF) advocates deployment of the virtual (software) network functions controlled by a software-defined network [115]. We describe a few network functions that can benefit from SDN and also comprise of offloadable state. The Network Address Translation (NAT) application allows multiple connections of the private network to access the Internet through a small set of public addresses. The NAT application processes any new outgoing client connection (flow) from the private network. NAT application assigns a public address (IP+port) to this connection. The assigned public address replaces the private address (IP+port) of the outgoing packet, and the address map (private address, public address) is inserted into the NAT table. After this, when a response packet for this flow enters the private network, the public address is replaced by the private address by looking up into the NAT table. Similarly, when the subsequent flow packets leave the private network, the NAT table is used to replace the private address by the public address. The client NAT table entry is valid for the lifetime of the flow. The NAT application can use hierarchical scaling as follows. The centralized controller partitions the public IP addresses from the global address pool and offloads them to the local controllers or gateway switches. The local controller or the gateway switches run the NAT application to create and maintain the flow-specific NAT table state to perform address translation of network flows. The NAT table state cannot be accessed concurrently from multiple network locations (packets of a flow enter/leave via the same gateway). Hence, it is an example of an offloadable state.

Next, let us look at the virtual private network (VPN) tunneling protocols like OpenVPN, which build a secure tunnel for the users accessing the private network from the outside. The tunneling protocols use robust encryption techniques to prevent the user's data from being intercepted at the public network. During the connection setup process, the VPN application at the user and the network agrees upon the security keys and algorithms. After the connection is successful, all the packets communicated between the user and the private network are encrypted using the user's security state. Since all user packets enter the network through a single ingress gateway, we can correctly offload the per-user security state, and computations like tunnel encapsulation, decapsulation, encryption, and decryption to the local controllers/edge switches. The centralized controller maintains the global master keys and the policy state (non-offloadable state) for all the network users to create the user security state during the initial handshake. The connection-specific

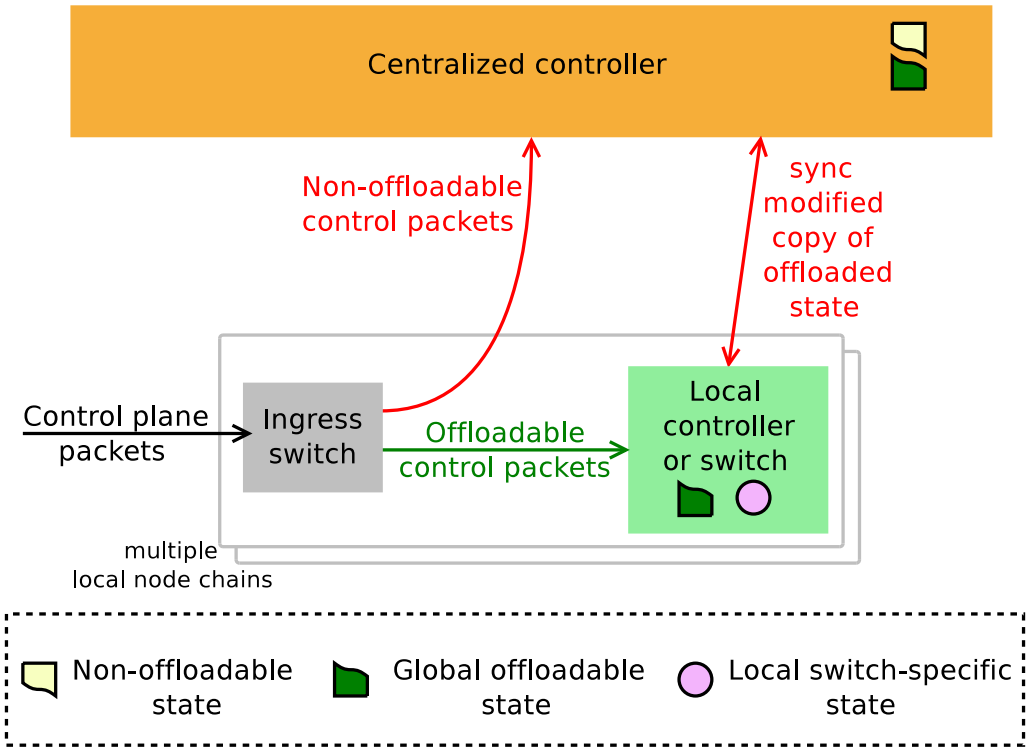


Figure 3.2: Proposed hierarchical control plane scaling.

security state is an example of an offloadable state as it cannot be accessed or updated concurrently from multiple locations.

In case of 4G LTE-EPC application (discussed in §2.4), the session-specific state of the UE like the forwarding state (tunnel identifiers like S1 SGW-TEID) and the connection state (*Connected* or *Idle*) are good examples of offloadable state. The UE forwarding state comprises of the tunnel identifiers (TEIDs) that are valid until a change in UE location or connection release. The UE connection state identifies if the UE is idle or connected to the network. The UE accesses and updates these states from a single network location; hence these states are offloadable. We discuss the detailed state taxonomy for the EPC application, and few other SDN applications in §3.2.2 and §3.2.3.

We have to be careful when we classify some session-specific state as offloadable. Any state shared across sessions cannot be offloaded if that state can be accessed across multiple sessions. If such state is offloaded, it can be simultaneously updated by multiple sessions from different network locations, leading to race conditions and incorrect application behavior.

3.1.3 Proposed hierachical offload design

Figure 3.2 shows the design of our proposed hierarchical scaling design, where subset of the global state that is identified as offloadable, is cached at the local controller or the switch. After that, all the control plane computations that depend on offloadable state alone (offloadable computations), are processed at the local controller/switches (close to the user). The centralized controller processes the computations that depend on some non-offloadable state (non-offloadable computations). The amount of computation offload is much more than that of the traditional hierarchical scaling approach since the offloadable state also includes the subset of the global state. The additional offload of control plane computations to local controllers/switches significantly improves the control plane capacity and reduces the response latency compared to traditional hierarchical scaling techniques.

But, this additional state and computation offload comes with a side effect. The side effect of the proposed offload technique is that the copy of the global state at the local controller/switch should be synchronized with the centralized controller (and vice-versa) to ensure state consistency. Following are the challenges of our proposed offload approach—

1. The non-offloadable messages at the root controller modify the offloadable state, and the offloadable messages at the local controllers/switches access the stale state.

2. The offloadable state copy is modified at the local controllers/switches, and the non-offloadable messages (at root controller) might access the stale state.

3. The offloadable state is accessed from one location, but the location of the end-user may change.

4. The local controllers or switches might fail, taking the latest copy of the offloadable state along with them.

Therefore, it is necessary to keep the offloadable state at the centralized root controller synchronized with the local offloadable state. Our proposed systems, Cuttlefish and TurboEPC, solve these challenges in the following way. For case (1) mentioned above, we implement strict synchronization mechanisms so that the offloadable messages access the correct offloadable state (similar to write-through). For case (2), we implement lazy synchronization mechanisms (similar to write-back), i.e., the non-offloadable message that requires access to the offloadable state initiates on-demand synchronization operation. For case (3), the session's last control message is responsible for on-demand state

synchronization. For case (4), we must implement state replication techniques at the local controllers/switches. We should carefully choose the offloadable global state to avoid undesirably high synchronization costs. We provide guidelines for choosing the offloadable state and computations for an application, in §3.2.1.

3.2 What application computations can be offloaded?

We have proposed two hierarchical control plane scaling designs, Cuttlefish and TurboEPC. In case of Cuttlefish, we process the offloadable computations at the local controllers, whereas in the case of TurboEPC, we process them at the programmable hardware switches. We provide a guide for application programmers that help the identification of offloadable application messages for local controllers as well as programmable switches.

3.2.1 Guide to identify offloadable messages

The application messages that do not access any state (stateless) are offloadable. For every control message of the SDN application that accesses some state, test the following:
Essential conditions (to guarantee correctness)

1. All the states accessed by the message are offloadable. The state is said to be offloadable if the following conditions are satisfied.
   - The state is never updated (read-only).
   - The state is either switch-local or has session-wide scope.
   - The state is not accessed concurrently from multiple network locations.
2. In the case of offload to hardware programmable switch targets, we also need to ensure that the programmable target should support the computations required for the message processing. The programmable target could be an ASIC, an NPU, an FPGA, or a software switch. The detailed checklist to determine offload to hardware programmable targets is provided at §6.5.1.

Desirable qualities (to improve performance)

1. The message should span a significant fraction of total traffic, else the effort of offload implementation is wasted.
2. The offloadable state accessed by the message should not be frequently updated by the non-offloadable messages at the centralized root controller. Otherwise, the state synchronization cost will negate the benefits of offload.

If the control message satisfies the essential conditions, then the message is said to be offloadable. But, in order to achieve high performance, the control message should also have the desirable qualities. The first desirable quality measures the benefit of the offload, whereas the second quality measures the cost of the offload. An offload decision must be taken only if the benefits outweigh the costs.

To implement the hierarchical design, whenever the offloadable state is generated at the centralized controller, it should be cached at the local controllers or switches. After that, we should update the rules at the switching devices such that all the incoming offloadable messages are routed to the corresponding local controller/switch. We apply our proposed guidelines to a few popular SDN-based applications and classify their application state so that they can utilize the benefits of our proposed hierarchical design.

3.2.2 Identify offloadable messages for LTE EPC application

We have provided a guide on the conditions that a control message should satisfy to be the right candidate for offload. Now, we shall apply the rules illustrated in the guide to real-life applications. We have described the details of the CUPS-based LTE EPC application in §2.4 and its architecture is shown in Figure 2.12. In the traditional CUPS-based EPC model, all the signaling (control) messages are processed by the MME, SGW, and PGW control components that reside at the centralized SDN controller. To apply our hierarchical scaling technique, we classify the EPC state as offloadable and non-offloadable. The offloadable computations are processed at the SGW switch (close to the user) to reduce the load at the centralized controller.

Table 3.1 shows the various components of the per-user state, or *user context*, that is accessed by LTE procedures [97]. We identify the part of the user context that has *network-wide* scope (shaded rows in the table) as the non-offloadable state. A piece of user context has network-wide scope if it is derived from, or depends on, network-wide information.

The security keys of the user include the master key (K_{ASME}), the cipher key (CK), the integrity key (IK), the authentication key (AV), the NAS encryption key (K_{NASenc}), and the NAS integrity key (K_{NASint}). The IMSI (International Mobile Subscriber identification) and MSISDN (Mobile Subscriber ISDN Number) are the permanent identifiers that provide the unique international identification for the mobile subscriber. The security keys and the permanent identifiers are derived from information that is located in the centralized HSS database and hence have a network-wide scope. The IP address has network-wide scope as it is assigned from the global address pool. This address pool can be concurrently accessed by messages from multiple locations. The registration man-

EPC state	Description	Examples	network-wide OR per-user	Offloadable (Y/N)
Security keys	Used for user authentication, authorization, anonymity, confidentiality	K_{ASME}, CK, IK, AV, K_{NASenc}, K_{NASint}	network-wide	N
Permanent identifiers	Identifies the user globally	International Mobile Subscriber Identity (IMSI), Mobile Subscriber ISDN Number (MSISDN)	network-wide	N
Temporary identifiers	Temporary identity for security	Globally Unique Temporary ID (GUTI), Temporary Mobile Subscriber Identity (TMSI)	per-user	Y
IP address	Identifies the user	UE IP address	network-wide	N
Registration management state	Indicates if the user is registered to the network	REGISTERED, DEREGISTERED	network-wide	N
Connection management state	Indicates if the user is currently idle or connected	IDLE, CONNECTED	per-user	Y
User location	Tracks the current location of the user	Tracking Area (TA), TAI (TA identifier)	per-user	Y
Forwarding state	Used for routing data traffic within the packet core	Tunnel end-point identifiers (TEID)	per-user	Y
Policy/QoS state	Determines policies & QoS values	Guaranteed Bit Rate (GBR), Maximum Bit Rate (MBR)	per-user	Y

Table 3.1: Classification of LTE EPC state.

agement state of the mobile user tells whether the user is currently registered with the network. This state can be accessed concurrently by multiple messages, like: (1) the user registration request (attach) or user registration termination request (detach), (2) the network (MME/HSS) can terminate the connection if the user's policy is modified, and (3) messages for management tasks like network load calculation, that maintain count of registered users in an area, and so on. Therefore, the registration management state has network-wide scope.

On the other hand, the temporary identifiers GUTI (Globally Unique Temporary Identity) and the TMSI (Temporary Mobile Subscriber Identity) are assigned by the network when the user connects to a network location and this state changes when the user changes its location. The temporary identifier state is session-specific, and this state cannot be concurrently accessed from multiple locations, hence it is offloadable. The connection state of a user (whether connected or idle) is only changed based on local events at the eNB (whether radio link is active or not), and hence has local scope. The user location, the user policies, or the QoS state pertains to a specific user (per-user state). This per-user state can be safely offloaded to the network location where the user is connected.

We have classified the EPC application state as non-offloadable (network-wide) and offloadable (per-user, session-wide). Now, let us classify the EPC application messages. Table 3.2 shows the various user states that are accessed during the processing of each

Message	Security keys	Perm id	Temp id	IP address	Registration mgmt state	Connection mgmt state	User location	Forwarding state	Policy / QoS state	Freq (%) [5, 6]
Attach	r+w	r	r+w	r+w	r+w	r+w	r+w	r+w	r+w	0.5 – 1
Detach	—	r	r+w	r+w	r+w	r+w	r+w	r+w	—	0.5 – 1
Service request	—	—	r+w	r	—	r+w	—	r+w	—	30 – 46
S1 release	—	—	r+w	r	—	r+w	—	r+w	—	30 – 46
Handover	r+w	r	r+w	r	r+w	r+w	r+w	r+w	r+w	4 – 5

Table 3.2: Classification of LTE EPC control messages.

LTE EPC procedure, along with the relative frequencies of each procedure. A procedure consists of multiple messages exchanged in a sequence. The shaded cells represent the states with network-wide scope (non-offloadable) that are updated by the EPC procedures.

We see from this table that the set of messages in the S1 release and service request procedures modify only: (1) the connection management state (from CONNECTED to IDLE and vice versa), (2) the forwarding state (GTP tunnel identifiers), and (3) the temporary user identifiers, none of which have the network-wide scope. Note that a given user is only connected to one eNB at a time, and any changes in user location are notified to the core via suitable signaling messages (e.g., handover). If the user location changes, the offloadable state is synchronized with the state at the centralized controller to ensure consistent state access. Also, the offloaded state at the local controller/switch is deleted. Therefore, it is safe to offload some parts of the user context to the edge close to the current eNB without worrying about concurrent access to this state from other network locations. All states accessed by the S1 release and service request procedures are offloadable (non-shaded rows of the Table 3.2).

Our hierarchical design, Cuttlefish, implements the offload over local SDN controllers, but TurboEPC implements the offload over programmable hardware. We examined and found that the S1 release and service request procedures can be programmed using P4 language for the programmable data plane targets (tested for bmv2 software switch [116] and Netronome smartNIC target [117]).

Both S1 release and service request procedures span a considerable fraction of traffic, 30% to 46% each. Therefore, if we offload the per-user, offloadable state to local controllers or data plane switches closer to the eNB edge, the S1 release and service request procedures can be processed locally without being forwarded all the way to the centralized controller. The offload of the S1 release and the service request procedures

to the edge is particularly useful because of the high proportion of these messages in the already high LTE signaling traffic [5, 6, 100, 101].

The handover procedure results in a change in the location from where the offloadable state is accessed. The non-offloadable handover procedure requires access to the offloaded state, as well as the non-offloadable state. The handover request triggers the synchronization of the offloaded state with the centralized controller, and the local state is deleted. But the handover procedure spans a tiny fraction of the total traffic (4–5%). The attach and detach procedures create and delete the per-user offloadable state at the centralized root controller, that triggers the synchronization of the offloaded state with the local controller/switch. But these procedures span a fraction of less than 2% of the total traffic. Therefore, the offload costs are lower as compared to the offload benefits.

We have identified S1 release and service request procedures as offloadable procedures of the LTE EPC application.

3.2.3 Identify offloadable messages for stateful load balancer

In this section, we introduce the SDN-based stateful load balancer application and illustrate the state classification so that the application can achieve scalability benefits by using our proposed hierarchical design. Consider a simple stateful load balancer that balances the incoming connections among the pool of servers, based on the current load on the servers (measured by, say, the current number of ongoing connections at the servers). If this application were to be implemented within the SDN framework, the load balancer application running at the centralized root controller would perform the following computations— (1) maintain server load statistics, (2) assign a least loaded server to the client, upon the start of a new connection, (3) install forwarding rules to direct traffic to the assigned server for all subsequent packets of the connection after connection setup, and (4) add/remove servers from the pool to dynamically provision resources during server overload and underload conditions. Our description of the load balancer application is somewhat simplistic, but it captures the essence of real implementations.

The centralized SDN-based stateful load balancer application can be overwhelmed with the growth of the incoming connections, and the increase in the number of servers among which these connections are to be balanced. Now, if this application were to be designed for a hierarchical SDN controller framework like Cuttlefish or TurboEPC (see Figure 3.3), one possible way to offload the computations could be as follows:

- The centralized controller partitions the “global server pool” and assigns a subset of servers (local server pool) to each local node (controller/switch).

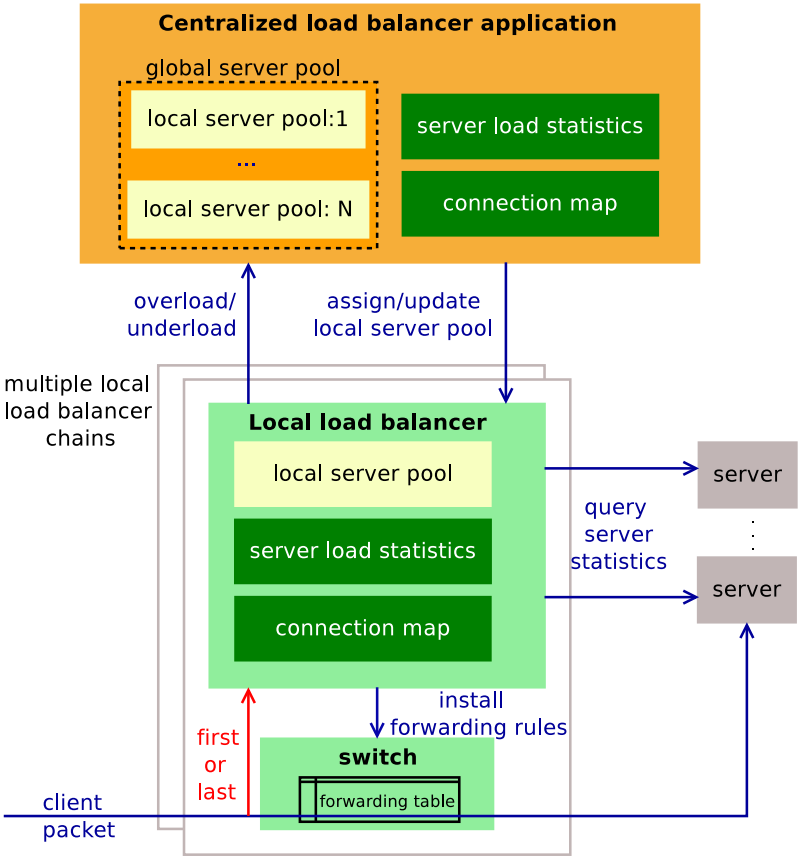


Figure 3.3: Hierarchical SDN-based stateful load balancer.

- The local nodes periodically queries each server assigned to the local server pool, and receives the number of active client connections at each server and the server utilization. This state is stored as "server load statistics". If any server is found overloaded or underloaded, the local node triggers the corresponding notification to the centralized controller.
  - The centralized controller queries the server load statistics from all the local nodes.
  - In case of an overload condition, if there are underutilized servers with some local server pool, the centralized controller moves the servers across local pools. Otherwise, the controller spawns a new server instance and assigns it to the overloaded local server pool.
  - In case of an underload condition, the centralized controller removes an underloaded server instance from the local and global server pools.
- The first and the last client packets from a client are processed by the local node. When the first packet arrives, the local node identifies the least loaded server from

State	Description	Example	Offloadable (Y/N)
Global server pool	List of active servers among which the load is balanced	{(server IP, server port), ... }	N
Local server pool	Subset of active servers assigned to local controller/switch	{(server IP, server port), ... }	N
Server load statistics	Current load level at each server	number of active connections, utilization	Y
Connection map	Stores mapping between client connection and assigned server	{(client (IP/port), server (IP/-port)), ...}	Y

Table 3.3: Classification of stateful load balancer state.

"server load statistics" and assigns it to the client. This client-server mapping is stored in the "connection map" state, and the forwarding rule is installed at the edge switch so that the switch forwards all consecutive client packets to the assigned server. The local node deletes the "connection map" entry when the last packet is received from the client.

Table 3.3 shows the state classification for the stateful load balancer application. All the states are stored at the controllers as key-value pairs. The shaded rows in the table denote non-offloadable states. The server pool state (global and local) is non-offloadable since provisioning and maintenance of servers require network topology information. The "server load statistics" state is offloadable since the server statistics are maintained for the assigned set of servers by the local node. The "connection map" state is valid for the lifetime of the client flow (session-wide), and this state cannot be accessed concurrently from multiple locations. So, the "connection map" state is offloadable.

Table 3.4 shows the stateful load balancer application layer messages and the corresponding states accessed by them. We refer to the proposed offload guide mentioned in §3.2.1 to identify the offloadable messages of the stateful load balancer application. The shaded cells of the Table 3.4 show the non-offloadable state that is updated by the load balancer messages. All the messages that do not update the non-offloadable state (shaded cells in the table) are offloadable. Therefore, "query server statistics", "install forwarding rules" messages, and all incoming client packets are offloadable. "Assign local server pool", "Update local/global server pool", and "Overload or underload trigger" messages update the non-offloadable state; therefore, these messages are non-offloadable and must be processed by the application that runs at centralized root controller. The frequency of client packet processing and periodic collection of local server statistics (offloadable

Message	Global server pool	Local server pool	Server load statistics	Connection map
Assign local server pool	r+w	r+w	—	—
Query server statistics	—	r	r+w	—
Overload or underload trigger	r+w	r+w	r	—
Update local/global server pool	r+w	r+w	—	—
First client packet	—	r	r	r+w
Last client packet	—	r	—	r+w
Install forwarding rules	—	—	—	r
Other client packets	—	—	—	—

Table 3.4: Classification of stateful load balancer messages.

computation) is much higher than that of overload/underload condition processing (non-offloadable computation). Therefore, the cost of synchronizing the local server pools (offload cost) is lower than the number of independent computations at the local nodes (benefit of offload).

We have not implemented the offload for the load balancer application over the hardware switches. We believe that it is possible to program the offloadable messages over programmable hardware since all the messages require simple computations and access key-value states. We have implemented the hierarchical stateful load balancer offload design for local controllers as one of the use cases of our proposal, Cuttlefish.

Note that several other network functions like stateful firewalls, stateful intrusion detection systems, NAT routers, and DNS can be decomposed into hierarchical SDN applications in this manner—a subset of application computations and corresponding state can be offloaded across local controllers, with each local controller handling part of the global state pertaining to its network location or traffic.

3.3 Summary

We have discussed the state taxonomy implemented by the traditional hierarchical solutions for application computation offloading and proposed a new application state taxonomy and an improved hierarchical offload design, to improve the application scalability over the status quo. We have provided a guide for the identification of offloadable state and computations. We apply this guide to classify the state of the real-life applications, the

SDN-based LTE EPC, and the stateful load balancer. Although we have only illustrated two examples, the offload guide is generic and can be applied to any SDN application.

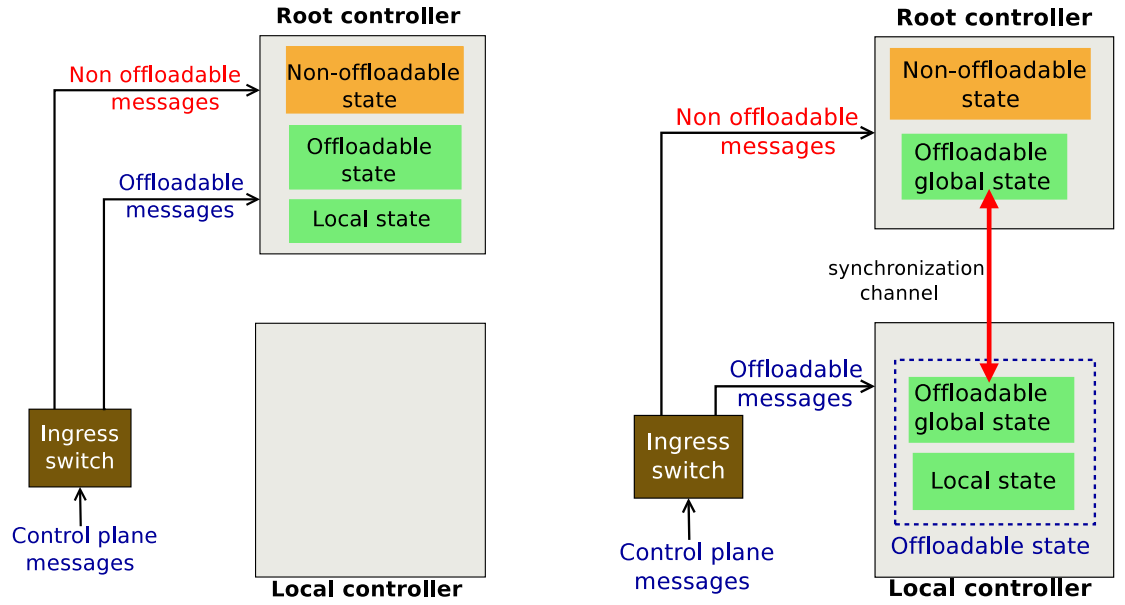
Chapter 4

Adaptive Offload of SDN Applications to Local Controllers

We have discussed the SDN control plane scalability problem and the existing literature on the scalability solutions in §2.2. In this chapter, we present the design and implementation of our proposed system, *Cuttlefish*, that advances the state-of-the-art hierarchical control plane scaling techniques.

4.1 Problem description

The existing literature on SDN control plane scalability has broadly classified the scalability solutions into horizontally distributed controllers and the hierarchical distributed controllers (§2.2). In the case of horizontally distributed SDN controllers, the incoming control plane traffic is distributed amongst multiple homogenous controllers that run concurrently over commodity servers. These controllers should be tightly synchronized to maintain the logically centralized network view. This design results in wastage of CPU cycles due to state synchronization. The control plane response latency is high since the control packets traverse all the way from the ingress switch up to the controller for processing. The existing hierarchical distributed controller design splits the control plane computations between the centralized root controller and multiple local controllers deployed close to the switch/user. This design offloads the application computations that only depend on the local switch-specific state to the local controllers, thereby scaling the SDN control plane and results in reduced lower response time latency for control plane traffic. Since the local SDN controllers store local state alone, there is no need for state synchronization with the centralized controller.



(a) Centralized mode. (b) Offload mode.

Figure 4.1: SDN operation modes.

The hierarchical distributed controller design is favorable compared to horizontally distributed controller design. However, not all control plane applications process a significant proportion of traffic that depend only on the local state; therefore, this design applies only to a small class of applications. The next section talks about the key idea of Cuttlefish, and it provides insights on how we can generalize the hierarchical scaling design and cater to a larger class of SDN applications.

4.2 Key idea, challenges, and contributions

We ask the key question: Can we increase the number of computations that can be offloaded to local controllers compared to the existing hierarchical control plane scalability solutions? To address this problem, we have proposed a new state taxonomy (described in §3.1.2) and a modified hierarchical offload design (described in §3.1.3). We define an additional class of SDN application messages that can be offloaded to the local controllers, i.e., the offloadable messages—messages that depend only on offloadable state (superset of local switch-specific state). The increase in the amount of offloadable messages lowers the computation overhead at the centralized root controller, resulting in higher control plane capacity. The local controllers reside close to the edge switch, thereby reduces the latency for the SDN applications.

To utilize the benefits of additional computation offload, we define two modes of operation for an SDN application, the *offload mode* (Figure 4.1(b)) and the *centralized*

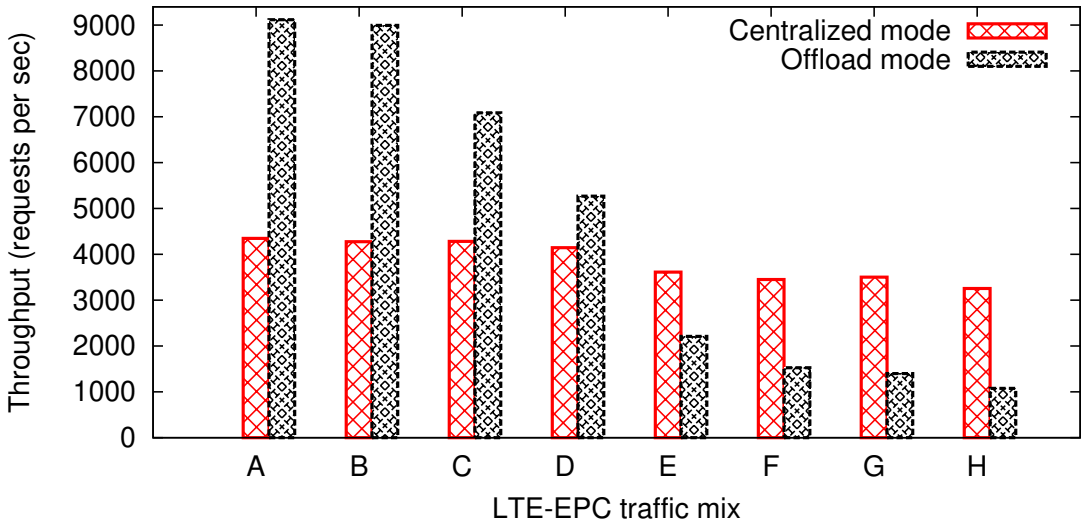


Figure 4.2: Performance with different controller modes

mode (Figure 4.1(a)). In offload mode, the copy of the offloadable state resides at the local controllers, and offloadable messages that access only offloadable state are processed locally. The non-offloadable messages are processed by the centralized root controller. The updates to the offloadable state at the root controller are synchronized consistently with the local controllers. But, the offloadable state updates at the local controllers are synchronized lazily (on-demand) with the centralized root controller. Therefore, the state synchronization cost is lower as compared to horizontal scaling design. In contrast, when operating in the default *centralized mode*, all application state resides at the centralized root controller, and all control plane messages (offloadable and otherwise) are processed at the centralized root (or one of its replicas in a distributed framework) controller. We have discussed several use cases in this thesis (§3.2) that show that several classes of SDN applications exhibit such offloadable state and offloadable messages.

Does the offload mode of operation always improve performance?

To answer the question, we show the experimental results in Figure 4.2. This figure shows the throughput of the 4G LTE packet core application, under various control plane traffic mixes, in both the centralized and offload modes of operation. The details for this experiment are discussed in §4.4. The performance and resource utilization metrics are discussed in Figure 4.11 and Figure 4.12. In this experiment, the proportion of registration messages (that update the offloadable state at the root controller, resulting in synchronization between the root and local controllers) monotonically increases from traffic mix A to mix H, and the proportion of offloadable messages decreases. We can observe that offload mode performs better than centralized mode for traffic mixes A to D, because a significant fraction of control plane messages is offloaded in offload mode,

thereby improving the capacity of the SDN controller. However, for the rest of the traffic mixes, the centralized mode performs better. The offload mode performance is poor for traffic mix E to H because the increase in the amount of non-offloadable messages increases the amount of state synchronization, resulting in high synchronization cost (in terms of CPU and network overhead) between the root and local controllers. That is, the cost of the offload is substantially higher than the offload benefits.

Let us look at the workload characteristics of real-world deployments. Atikoglu et al. [27] has presented the workload analysis from Facebook's Memcached (key-value store) deployment. Their observations include extreme variations in terms of read/write mix, request sizes and rates, and usage patterns. They have reported instances where the load varies by more than 2X within intervals as small as 16 minutes. Studies by Filiposka et al. [118] have reported temporal dynamicity for mobile control plane and data plane traffic for real-world deployments. Such temporal dynamic workload characteristics are behavioral in nature (for example, traffic peaks during the start of office hours) and apply to any application.

Given the evidence that the traffic characteristics are dynamic in nature, our key idea is that an SDN controller framework must support offloading of offloadable state and associated computation adaptively between the centralized mode and offload mode based on the cost of synchronization, to optimize system performance.

Key challenges and contributions

Once we have identified the offloadable application messages using our offload guide (§3.2.1), we can offload the computations of these messages to the local controllers, but we face a few challenges while designing the adaptive offload mechanism of Cuttlefish.

- **High state synchronization costs.** We need to synchronize the changes to the offloadable state between the centralized root and the local controllers for correct application behavior. But, we have discussed in §1.4 that high state synchronization costs can outweigh the offload benefits. Cuttlefish solves this problem in two ways.

    1. Cuttlefish implements optimization techniques like lazy (on-demand) state synchronization and batching to ensure that we do not waste synchronization cycles if the stale state is never accessed. For example, in the offload mode, the offloadable state is always accessed at the local controllers, so there is no need to synchronize state updates to the root controller (§4.3.3).
    2. The Cuttlefish framework dynamically determines the state synchronization cost. Despite the optimizations, if the synchronization cost is high such that

the application performance degrades, the framework automatically switches the SDN application processing from the proposed offload SDN mode to the traditional centralized SDN mode that does not require state synchronization (§4.3.4).

- **State consistency during mode migration.** The Cuttlefish root controller caches a copy of the offloadable state at the local controllers to process offloadable messages. There are two cases when this offloadable state can be inconsistent.

  1. When the Cuttlefish framework automatically switches between the centralized and offload SDN modes to ensure the best application performance, there is a short phase during which the application is not entirely in centralized mode or offload mode. During this phase, the application packets could arrive at both the controllers and should be processed correctly. We have designed a migration protocol to ensure that the offloadable state is accessed/updated at the correct location (either root or local controller). This latest state must be used once the mode migration phase completes. We describe the detailed design of the migration protocol for the switch between the offload and centralized modes in §4.3.6.
  2. When the non-offloadable state requires access to the offloadable state at the root controller, but the root controller has a stale copy due to lazy synchronization. Cuttlefish framework implements on-demand synchronization of the offloadable state as explained in §4.3.3.

- **Effort of code rewrite for existing SDN applications.** The application programmer must rewrite the application code to utilize the Cuttlefish framework. Our framework provides APIs (§4.3.2) to access offloadable state in an SDN application, and manages the synchronization of this state across the root and local controllers, to reduce the programming efforts. The programmer has to replace the state access procedures (get, put, delete) in the original application code by the Cuttlefish APIs, and provide the identification details about the offloadable and non-offloadable application messages.

  Using this limited input from the programmer, Cuttlefish can migrate the original SDN application to the one that can utilize the Cuttlefish framework. The Cuttlefish framework can dynamically select the appropriate SDN mode (centralized or offload) and automatically switch between them.

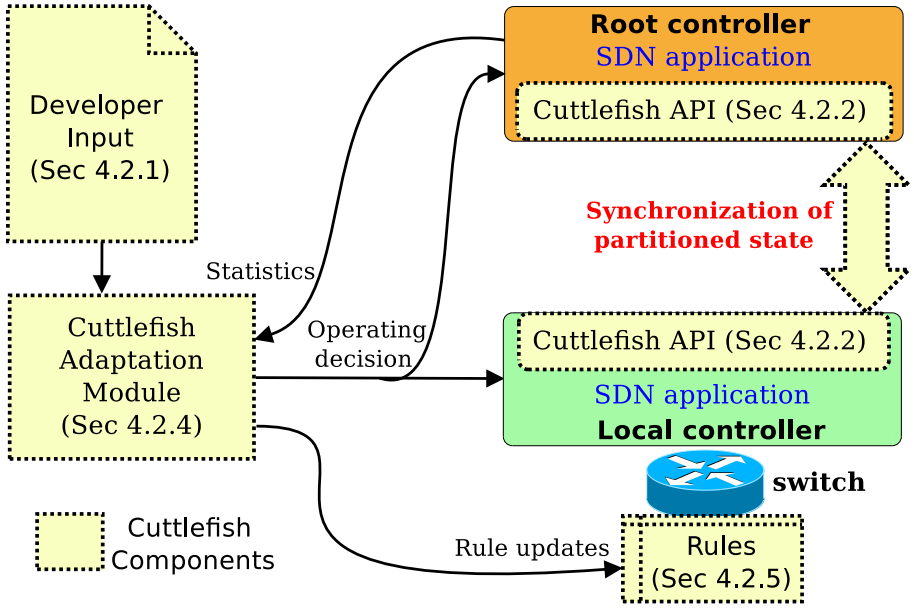


Figure 4.3: The Cuttlefish architecture.

4.3 Cuttlefish design and implementation

This section describes the design and implementation of the Cuttlefish hierarchical SDN controller framework. Figure 4.3 shows the architecture of Cuttlefish. Cuttlefish takes input from the application developer regarding the type of control plane messages, i.e., whether they are good candidates for offload (§4.3.1). SDN application developers write applications using Cuttlefish API (§4.3.2) functions to access the offloadable state. The framework takes care of transparently synchronizing this state across the root and local controllers based on the operating mode (§4.3.3). The heart of Cuttlefish is its adaptation module (§4.3.4) that dynamically measures the cost of synchronizing the offloadable state and the benefits due to offload, and decides on whether to operate the application in proposed offload mode or centralized mode. The framework enforces the offload decision by pushing suitable rules into the data plane SDN switches (§4.3.5). When the adaptation module decides to switch between controller modes, Cuttlefish ensures that the migration of offloadable state and redirection of control plane traffic happens correctly without any race conditions (§4.3.6). Finally, we describe our implementation of sample SDN applications in the Cuttlefish framework (§4.3.7).

4.3.1 Developer input

Cuttlefish requires the application developer to provide the following input: the types of messages in the control plane traffic, the control plane message identifier, and whether each of these messages is offloadable or not. We assume that the control plane

traffic to the application has a discrete, known number of message types, which can be identified by inspecting packets in the SDN switches. If the type of the control plane message cannot be identified by parsing standard L2-L4 headers alone, we assume that the switches are programmable using a language like P4 [4], to be able to parse application layer headers and identify the control plane message type. The application developer provides rules to identify incoming message types as part of the input specification. For each message type in the control plane traffic, the user specifies whether the message is offloadable or not.

Can Cuttlefish automatically classify application state? We have defined offloadable and non-offloadable state in §3.1.2. The classification of the application state is tightly tied with the application semantics. We cannot automate to infer if the given state is session-specific and would be accessed from the same network edge all the time. Therefore, we require the application developer to provide state classification as input.

Can Cuttlefish automatically classify control plane messages? Cuttlefish could cause application performance degradation if the application messages are incorrectly classified. It may be arguable that given the state classification as input, we can use compiler techniques such as lexical analysis and parsing to automatically identify the offloadable messages, i.e., the application messages that only access the offloadable state. Such automatically determined offloadable messages satisfy the essential conditions but may not satisfy the desirable qualities (§3.2.1) to declare an application message as offloadable. We require application developer intervention to determine if the message possesses desirable qualities to classify the application message as offloadable.

How does an application developer decide if a message can be offloaded to a local controller? Given our definitions of offloadable and non-offloadable state (§3.1.2) and guidelines on classification of application state at §3.2.1, the application messages can be classified as offloadable and non-offloadable. We have demonstrated the application state classification for real life application examples like the SDN-based LTE EPC (§3.2.2) and the stateful load balancer (§3.2.3). We expect application developers to have sufficient knowledge about application state semantics to be able to classify the application messages. This expectation from the developers is the standard practice that exists in prior work too. For example, Split/Merge [50] and OpenNF [51], provide APIs for moving state between distributed networking applications and require the developer input to have a similar understanding of the semantics of application state. Table 4.1 shows an example of developer input for the LTE EPC Cuttlefish application, listing the types of messages in the control plane traffic of the EPC application and whether they are offloadable. The shaded rows indicate the non-offloadable EPC messages.

Message type	**Message identifier**	**Offloadable? (true/false)**
Authentication Step 1	$switchRule_1$	false
Authentication Step 3	$switchRule_2$	false
NAS Step 2	$switchRule_3$	false
Send Access Point Name	$switchRule_4$	false
Send UE Tunnel id (teid)	$switchRule_5$	true
UE Context Release	$switchRule_6$	true
UE Service Request	$switchRule_7$	true
Context Setup Response	$switchRule_8$	true
Detach Request	$switchRule_9$	false

Table 4.1: Sample developer input for LTE EPC.

4.3.2 The Cuttlefish API

Application developers within the Cuttlefish framework do not need to write separate applications to run at the root and local controllers. Instead, developers must use the Cuttlefish state management API to access the offloadable state. The framework takes care of transparently synchronizing this state across the controllers, depending on the mode of operation. We assume all offloadable state can be stored as key-value pairs. Our API provides the following get/put/delete functions:

```
get(partition_id, map_name, key)
put(msg_id, partition_id, map_name, key, value)
delete(msg_id, partition_id, map_name, key)
```

The developer invokes Cuttlefish API functions when accessing the offloadable state in the application code, instead of invoking standard hashmap API functions.

The Cuttlefish API takes *map_name* as one of the parameters in the get/put/delete functions that identify the hashmap.

The offloadable state space is partitioned, and each local controller is assigned one partition, to optimize the synchronization overheads. The partition stores the offloadable state only for users who access the network via the switch associated with the corresponding local controller. Therefore, the offloadable state updates at the root controller for a particular user are synchronized with a single local controller. As per our definition of an offloadable state, the user accesses the offloadable state from a single location. The ingress switch or local controller identifiers are proxies for the user's (or user's offloadable state) location. By default, the packet sent by the ingress switch to the controller is

encapsulated with switch information (e.g., Openflow's packet-in header) like switch-id and output port. The programmer should use this packet header information and supply the ingress switch identifier as *partition_id* parameter, so that the Cuttlefish controller can determine the partition to be used for offloadable state access.

The parameter *msg_id* corresponds to the identifier of the message that generated the state update. This parameter is part of the *put* API, to let our framework attribute synchronization costs to control plane messages (more details in §4.3.4).

Listing 4.1: Code snippet using standard hashmap API.

```
1  import java.util.HashMap;
2  //Programmer's SDN application written for Floodlight SDN controller
3  public class TestMain {
4    public static void main(String[] args) {
5      // Create a HashMap objects for the KV stores
6      HashMap<String, String> ueTunnelMap = new HashMap<String, String>();
7      HashMap<String, String> freeTunnelMap = new HashMap<String, String>();
8      HashMap<String, String> ueStateMap = new HashMap<String, String>();
9      //Message identifiers for incoming application messages
10     ...
11     final static String DETACH_MESSAGE = "2";
12     ...
13      //Process control plane messages (PACKET_IN)
14     ...
15     //Processing for "DETACH_MESSAGE"
16     case DETACH_MESSAGE:
17       //Extract ueKey from packet header
18       ...
19       String tunId = ueTunnelMap.get(ueKey);
20       freeTunnelMap.put(ueKey, tunId);
21       ueStateMap.del(ueKey);
22       ...
23       break;
24     ...
25     }
26 }
```

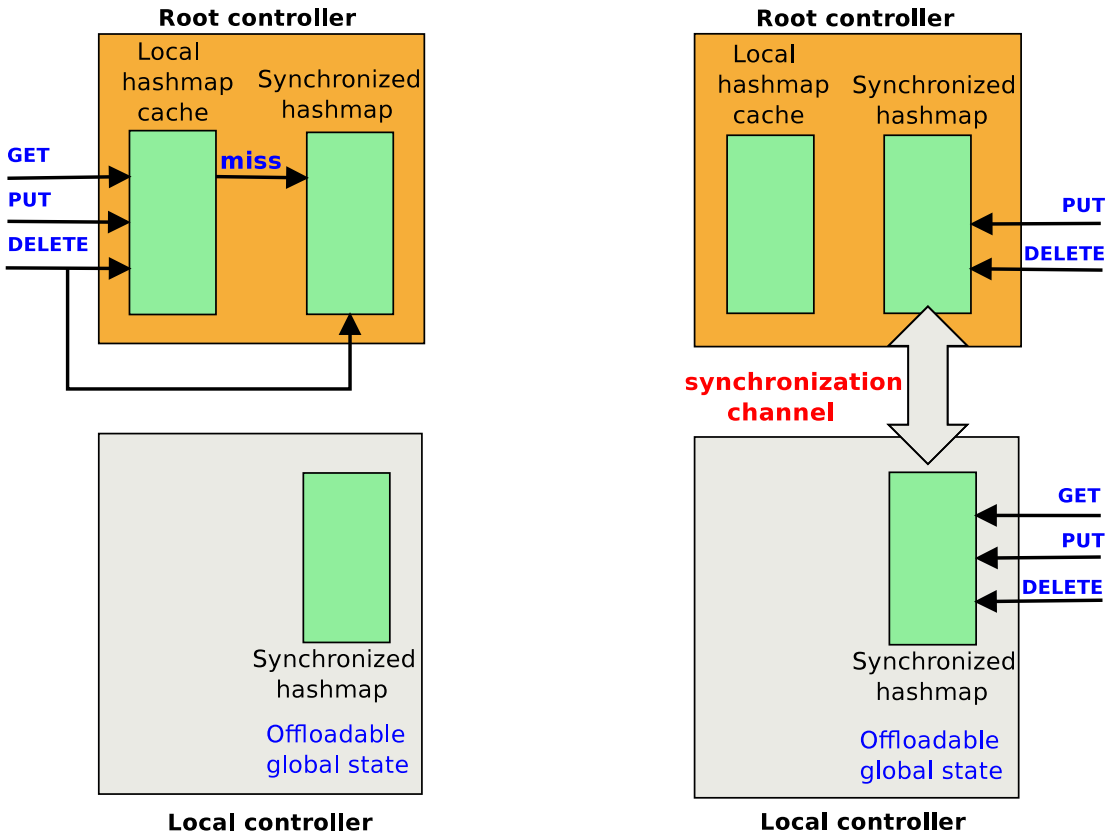
To help understand the usage of Cuttlefish API, we provide a code snippet that uses get/put/del API for key-value operations. Listing 4.1 shows the code snippet with a standard API call. Listing 4.2 shows the code snippet with the corresponding Cuttlefish API call. The sample code snippet shows the subset of the detach control plane request processing for the 4G mobile packet core. We retrieve the tunnel identifier assigned to the detaching user, add the tunnel identifier to the free list, and delete the user's state. This objective is implemented as Lines 19–21 of Listing 4.1 which correspond to lines 29–31 of Listing 4.2. Lines 26 and 28 of Listing 4.2 shows how Cuttlefish API could be used to retrieve `partition_id` and `msg_id`, respectively. Lines 10–14 of Listing 4.2 shows the `msg_id` declaration for application messages.

Listing 4.2: Code snippet using Cuttlefish API.

```
1  //Create ConcurrentHashMap objects for the KV stores in predefined Cuttlefish class
2  public class CF ... {
3  ...
4  public static ConcurrentHashMap<String, String> ueTunnelMap = new
       ConcurrentHashMap<String, String>();
5  public static ConcurrentHashMap<String, String> freeTunnelMap = new
       ConcurrentHashMap<String, String>();
6  public static ConcurrentHashMap<String, String> ueStateMap = new
       ConcurrentHashMap<String, String>();
7  ...
8  }
9  //Message identifiers for incoming application messages declared in predefined
       Cuttlefish class
10 public class CfConstants{
11 ...
12 final static String DETACH_MESSAGE = "2";
13 ...
14 }
15 //Programmer's SDN application written for Floodlight SDN controller
16 public class TestMain {
17   public static void main(String[] args) {
18       ...
19       //Process control plane messages (PACKET_IN)
20       ....
21       //Processing for "DETACH_MESSAGE"
22       case DETACH_MESSAGE:
23          //Extract ueKey and sourceIP from packet header
24          ...
25          // Obtain ingress switch identifier (partition_id) using Cuttlefish API
26          DatapathId testDpid = CfConstants.getDpid(sourceIP.toString());
27          //Obtain the message identifier (msg_id) for DETACH_MESSAGE
28          int msgId = Integer.parseInt(CfConstants.DETACH_MESSAGE);
29          String tunId = CF.get(testDpid, "ueTunnelMap", ueKey);
30          CF.put(msgId, testDpid, "freeTunnelMap", ueKey, tunId);
31          CF.del(msgId, testDpid, "ueStateMap", ueKey);
32          ...
33          break;
34          ...
35    }
36 }
```

4.3.3 Cuttlefish API implementation

The Cuttlefish API is implemented using hashmaps synchronized between the centralized root and the local controllers. The get/put/delete operations on the offloadable state are performed on these synchronized hashmaps. The use of synchronized hashmaps is expensive because for every put/delete operation at the synchronized hash maps, our application performs additional computations related to version control and concurrency control, to ensure state consistency.



(a) Centralized mode.

(b) Offload mode.

Figure 4.4: Cuttlefish API functions.

We implement a few optimizations to deal with the high overhead of synchronized hashmaps, as follows.

1. While a traditional SDN application may use several hashmaps to store the offloadable state, Cuttlefish stores all the state in a single synchronized hashmap (for each partition). Otherwise, multiple synchronization channels have to be maintained between the root and local controllers, one for each hashmap. The use of a single hashmap for all offloadable state reduces the synchronization overheads. The programmer's view of the state should not be changed, therefore the key stored in Cuttlefish is a concatenation of the map_name and the original key.

2. When operating in the centralized mode, all the messages are processed by the root controller. That is, we do not require to synchronize the offloadable state. Cuttlefish reduces the synchronization overheads by using local hashmaps (no version control) instead of synchronized hashmaps, whenever possible. The details are described below.

Cuttlefish API implementation: centralized mode. During the centralized mode of operation, there is no need to synchronize the updates to the offloadable state with the local controller since all application messages are processed at the root controller. To speed up the put/delete operations in centralized mode, we temporarily cache the offloadable state in local hashmaps. That is, the application state in centralized mode is split between synchronized hashmaps (which would have been populated when the application was in offload mode) and the local hashmap cache (which is used only in centralized mode). The API function implementation for the centralized mode is shown in Figure 4.4(a). Our goal is to avoid unnecessary state synchronization and speed up the put/delete operations. For example, if a particular user was active when the system was in offload mode, the synchronized map has the user's state as *Connected*. After this, the system switches to centralized mode. Now, if the user turns idle, the eNB initiates a context release procedure, and the user's state should be changed to *Idle*. In centralized mode, we apply all the put operations to the local hashmap for fast processing. Get operations are first performed on the local hashmap since it has the most recent state. If the state is not found in the local hashmap, it indicates that the state during the offload mode is the latest; hence the get operation fetches the state from the synchronized hashmap. Delete operations are performed on both local and synchronized hashmaps for consistency.

Cuttlefish API implementation: offload mode. When operating in offload mode, all offloadable messages are processed at local controllers, and all offloadable state accesses (get/put/delete) by the offloaded messages are performed on the synchronized hashmaps, as shown in Figure 4.4(b). All non-offloadable messages are handled at the root controller (e.g., because processing such messages depends on other global states), and these messages may also generate concurrent put/delete requests to the offloadable state. To optimize performance in offload mode, we batch updates to synchronized hashmaps at the local controller and push multiple updates at a time to the root controller. However, updates to offloadable state at the root controller are immediately pushed to the local controllers without batching, in order to ensure that the get operations at the local controller never see the stale state. If any of the non-offloadable messages (e.g., handover message of EPC application) requires access to the offloadable state cached at the local controller, such messages are routed to the root controller via the local controller. The cached offloadable state is piggybacked with the non-offloadable message, deleted from the local controller, and the message is forwarded to the root controller.

We implement synchronized hashmaps and batching by extending the fault tolerance module of the open-source Floodlight SDN controller [44]. The Cuttlefish framework implements TCP communication channels between the root and local controllers to transport

updates to the synchronized hashmaps. We batch up to 500 updates at a time at the local controller. Note that we currently do not handle pending updates in a batch being lost due to the failure of the local controller. Our changes spanned about 350 lines of code in the Floodlight controller code base.

4.3.4 The adaptation approach

The Cuttlefish adaptation module dynamically monitors the cost of synchronizing the offloadable state across the root and local controllers and weighs the benefits of offload against the cost to decide the appropriate mode of operation (centralized vs. offload) for the SDN application. The adaptation module can run as a separate application at the root controller or as a standalone application.

When is the centralized mode better than the offload mode? The non-offloadable messages that write to the offloadable state at the root controllers trigger state synchronization and, therefore, form a significant part of the synchronization cost. The state synchronization process uses a large number of CPU cycles to perform functions like version control for every state update, and also uses a small slice (typically, state sizes are small) of network bandwidth for agreement on the current value of states. The amount of network bandwidth utilization is negligible even when the root controller is saturated due to a high state synchronization rate, so we cannot use this metric to quantify the state synchronization cost. Instead, the rate at which the non-offloadable messages (at the root controller) update the offloadable state can be a good proxy to quantify the state synchronization cost (cost of the offload).

When the Cuttlefish application is operating in offload mode, and the fraction of offloadable traffic is high, we achieve high throughput and latency gains, since the root controller load is offloaded to local controllers, and the state synchronization cost is low. Therefore, the rate at which offloadable message arrive at the local controllers can be used as a proxy to measure the benefits obtained due to offload. The Cuttlefish adaptation module can decide to switch from offload mode to centralized mode when the synchronization cost exceeds the gains due to offloading, i.e., when the cost of the offload exceeds the benefits due to the offload.

Determine the metrics that decide the SDN mode switch. Let us define the metrics that quantify the offload gains at the local controller, and the synchronization cost at the root controller. Let 'f_{NOFF}' be the fraction of non-offloadable messages in the incoming traffic, that generate writes to the offloadable state at the root controller, during an epoch (say, epoch=10sec). Let 'f_{OFF}' be the fraction of offloadable messages in the incoming traffic, that access (read/write) the offloadable state at the local controller, during an epoch. Let

k_R be the average number of writes to the offloadable state at the root controller, by any non-offloadable message. Let k_L be the average number of accesses to the offloadable state at the local controller, by any offloadable message. We have manually obtained the k_R and k_L values. Although, given the input as the set of offloadable control plane messages, non-offloadable control plane messages, offloadable state variables, and non-offloadable state variables, k_R and k_L values can be automatically derived. Let N_R be the number of root controller CPU resource, and N_L be the number of local controller CPU resource. The local controller runs over the local switch CPU or the commodity server, close to the edge. The local switch CPUs aren't typically powerful unless the switches are custom built by network operators [119]. For example, the Pica8 3290 OpenFlow switch uses a 825 MHz PowerPC CPU [120]. Switch local CPUs can widely vary in their packet I/O performance. Therefore, the CPU resource parameters, N_R and N_L, should be provided as normalized values. Our implementation runs the root and local controllers over the commodity server; therefore, we use absolute values

The state synchronization cost at the root controller can be quantified by the *put_rate* at the root controller defined in equation 4.1. The *put_rate* is the average number of offloadable states written by non-offloadable messages at the root controller during an epoch, normalized to the root controller CPU.

$$put\_rate = (f_{NOFF} * k_R)/N_R \tag{4.1}$$

The gains due to computation offload can be quantified by the *access_rate* at the local controller defined in equation 4.2. The *access_rate* is the average number of offloadable states accessed by offloadable messages at the local controller during an epoch, normalized to the local controller CPU.

$$access\_rate = (f_{OFF} * k_L)/N_L \tag{4.2}$$

The rate at which the offloadable state is written at the root controller (*put_rate*) is a good proxy for estimating the synchronization cost. The rate at which the offloadable state is accessed at the local controller (*access_rate*) is a good proxy for estimating the benefits obtained due to computation offload.

If the *put_rate* at the root controller is higher than the *access_rate* at the local controllers, it implies that there is not enough offloadable load to be processed at local controllers. The state synchronization cost (*put_rate*) is an overhead at the root controller, and the local controllers are underutilized. Under low load conditions, we still observe lower response latencies, as the offloadable messages are processed close to the user. But,

the performance of the offload mode degrades when the state synchronization due to the high *put_rate* saturates the root controller CPU, and there is not enough CPU available for application message processing. Under such high non-offloadable traffic-mix conditions, the performance of centralized mode is better than the offload mode in terms of throughput as well as latency, since centralized mode does not require state synchronization.

Cuttlefish adaptation metric computation. We assume that the application programmer has the k_R, k_L values, and the network administrator who deploys the application has the knowledge of N_R, and N_L. The k_R, k_L, N_R, and N_L values are provided as input to the Cuttlefish adaptation module. The Cuttlefish adaptation module monitors the values of f_{NOFF} and f_{OFF} dynamically. At the end of each epoch, the Cuttlefish adaptation module queries the edge switches to obtain the statistics of the number of packets received for each control plane message type. The message type information is provided as input by the programmer as defined in §4.3.1. The Cuttlefish adaptation module uses the programmer input to identify the offloadable messages that read/write to offloadable state and non-offloadable messages that write to the offloadable state. The adaptation module computes the count of offloadable messages, non-offloadable messages, and total messages received during the epoch. These calculated values are used to obtain the fractions f_{NOFF} and f_{OFF}. At the end of each epoch, the Cuttlefish module substitutes the computed f_{NOFF} and f_{OFF} values in the equations 4.1 and 4.2 to obtain the *put_rate* and *access_rate* that estimates the synchronization cost and the offload benefits for the last epoch.

Cuttlefish adaptation conditions. When operating in offload mode, if the *put_rate* at the root controller is higher than the *access_rate* at the local controller, it implies that the synchronization cost is higher than the offload benefits. If the root controller CPU is also saturated ($> 90\%$), there is not enough CPU available for application message processing; therefore, the Cuttlefish framework should switch to the centralized SDN mode. Of course, the root controller CPU saturation condition will always hold, as the problem solved by this thesis is to alleviate the root controller bottleneck.

When operating in centralized mode, if the *put_rate* at the root controller is lower than the *access_rate* at the local controller, the Cuttlefish framework should switch to offload mode since the performance gains would be high.

Equation 4.3 states the condition when the Cuttlefish adaptation module decides to migrate from the offload mode to the centralized mode, whereas, equation 4.4 states the condition when Cuttlefish adaptation module chooses to migrate from the centralized mode to offload mode. We keep a guard band of Δ to ensure the buffer between mode switch decisions and avoid flip-flops.

$$\frac{put\_rate}{access\_rate} \geq 1 + \Delta \tag{4.3}$$

$$\frac{put\_rate}{access\_rate} \leq 1 - \Delta \tag{4.4}$$

The instrumentation to the Floodlight controller to gather the statistics of *put_rate*, *access_rate*, and the adaptation algorithm logic were implemented in about 200 lines of code.

Assumptions. Note that, the proposed adaptation metrics and conditions are not directly applicable for multiple Cuttlefish applications.

- We assume that the SDN application is bottlenecked by the root controller CPU, which will be the case in scenarios where SDN controller scalability solutions are deployed. In such a case of root controller CPU bottleneck, CPU cycles spent on synchronization reduce the amount of CPU available for application processing at the root. Therefore, the put rate is a good metric to capture the cost of the offload.

- If the *access_rate* is high due to a small set of flows, Cuttlefish will run in offload mode even for the flows that do not use the state at the local controller, and root controller CPU is wasted in synchronization of such state. Similarly, if the *put_rate* is high due to a small set of flows, Cuttlefish will switch to the centralized mode for all flows, affecting the performance of other flows. The adaptation conditions choose the best performing mode based on overall application statistics and does not capture per-user or per-traffic-class statistics. Assuming that the number of flows (or users) is very large, we may rarely observe this situation.

Alternative adaptive offload decision metrics.

Cuttlefish uses the offloadable and non-offloadable message frequencies normalized with the controller resources to estimate the offload costs and benefits. We can translate the *put_rate* and *access_rate* to absolute CPU and network utilization costs and benefits. This translation would involve profiling the underlying hardware every time the SDN controller is migrated. We observed that the Cuttlefish adaptation decisions were the same for the proposed adaptation metric and the absolute resource utilization metric. Such absolute metric values could be used to enforce policies such as enforcing the CPU upper bound for offloadable state synchronization.

4.3.5 Enforcing the offload mode

When the Cuttlefish adaptation algorithm decides to switch from the offload mode of operation to a centralized mode, or vice versa, the SDN switches in the data plane must be configured in real-time to redirect messages to the suitable controller. We now describe how this redirection happens in our system.

Our framework has been implemented over the OpenvSwitch (OVS) [57] SDN switches managed by the Floodlight controller. The OVS switches are configured with rules to identify the various message types specified in the user input. When the system switches modes, the controller and switches must redirect specific offloadable message types to the appropriate controller (root/local) based on the mode of operation. The controller in our implementation did not come with this support to direct packets to a specified controller; all switches forwarded traffic to all configured SDN controllers by default. Therefore, we developed an extension to the Floodlight controller by implementing the `NiciraSetControllerId` feature in the Loxigen library[44], which allows the Floodlight controller to identify and communicate with specific switches. To adaptively switch between modes, we added logic to the controller to automatically generate Openflow commands that add/delete/modify rules to direct specific message types to specific controllers at the OVS switches. Finally, we added a new Openflow action type `of_action_nicira` to Floodlight that allows adding routes at switches to direct packets to a specific controller (instead of forwarding to all controllers, as in the default implementation). These changes required modifying ~150 lines of code in the controller (Java), and Loxigen library (C++) code base and required no changes to the OVS switch implementation.

4.3.6 Transition between controller modes

When transitioning between modes, the Cuttlefish framework avoids race conditions during the installation of switch rules to divert traffic, and the process of synchronizing state across the root and local controllers. We now describe the detailed algorithm for Cuttlefish transition from the offload mode to the centralized mode, and back.

Offload mode to centralized mode. Figure 4.5 shows the timeline of tasks performed by the Cuttlefish framework when it decides to switch from the offload mode to centralized mode. Recall that the offloadable state is synchronized in batches from the local controllers to the root controller, in offload mode. When we want to switch from offload to the centralized mode, we must immediately synchronize the offloadable state. The root controller instructs the local controller to flush all pending updates from the synchronized hashmaps, immediately. After waiting for a grace period for the synchronization to com-

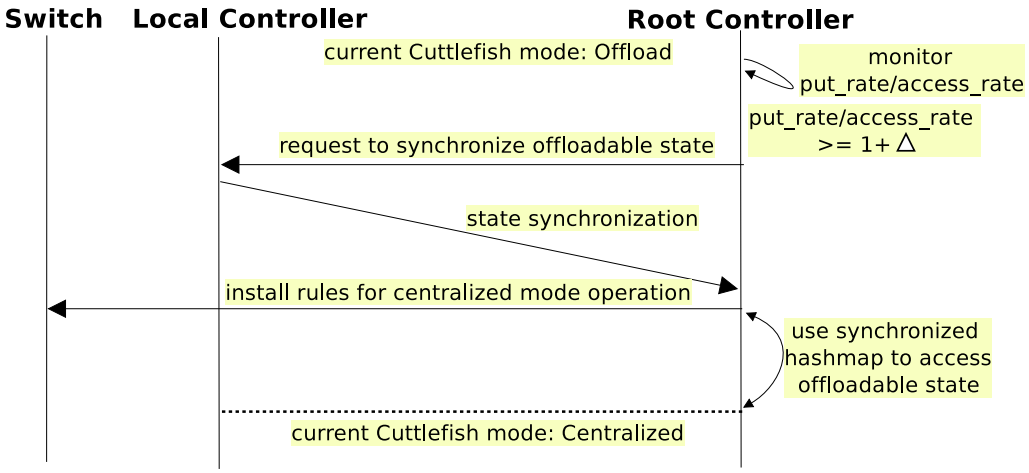


Figure 4.5: Switch from offload mode to centralized mode.

plete, the root controller is ready to switch to the centralized mode. The root controller first pushes the rules onto the OVS at the local controller to forward all the messages (offloadable and otherwise) to the root controller. However, there could still be packets in the pipeline at the switch that arrive at the local controllers, and continue to update the offloadable state for a short duration after the switch rules have been installed. In order to correctly handle such packets, the root controller accesses the offloadable state from synchronized hashmaps for a brief waiting period. Further, new packets arriving at the root are buffered until the packets in the local switch's pipeline have been processed, to avoid reordering. Once this grace period for flushing the switch pipeline has expired, the root controller stops state synchronization of synchronized hashmaps, since no packets will be serviced by the local controllers. The root controller can now switch to centralized mode with consistent offloadable state, and store newly created offloadable state in the local hashmap cache for better application performance. The values of the grace periods are a few milliseconds in our implementation, and will have to be configured based on the processing latency of the local controller and the network latency between the root and local controllers for other deployments.

Centralized mode to offload mode. Figure 4.6 shows the timeline of tasks performed by the Cuttlefish framework when it decides to switch from the centralized mode to offload mode. When Cuttlefish is operating in centralized mode, some of the offloadable state is stored in the local hashmap cache at the root controller, and some in the synchronized hashmaps. When the framework decides to switch from centralized to offload mode, we must migrate the offloadable state from the local hashmap cache to the synchronized hashmap at the root controller. During the state migration phase, all the delete operations at the root are performed on both the local and synchronized hashmaps, all put operations are performed only on the synchronized hashmaps. In contrast, all get operations are

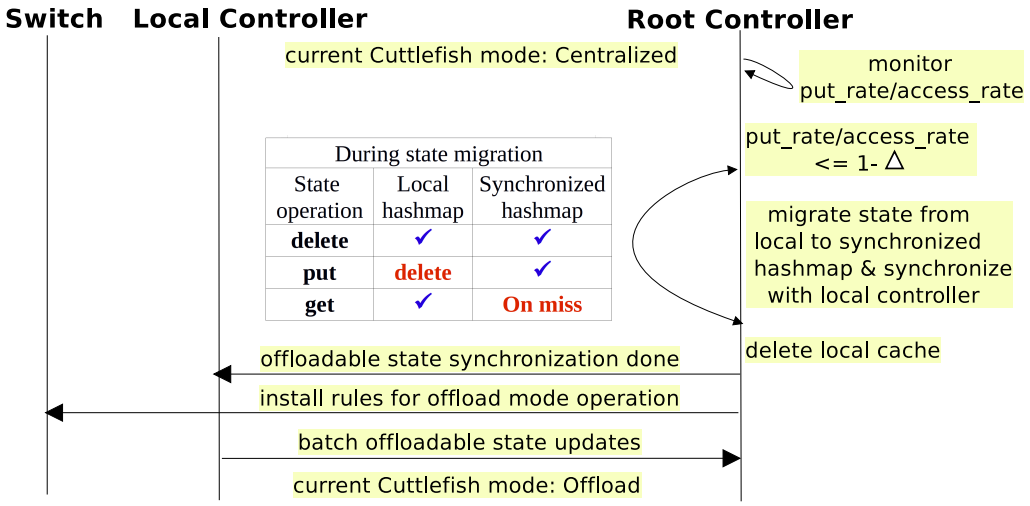


Figure 4.6: Switch from centralized mode to offload mode.

handled normally (get from the local cache, and on a miss get from the synchronized hashmap). Also, for all put operations during state migration, we first perform delete on local hashmap to avoid state inconsistency. After the local hashmap has been transferred to the synchronized hashmaps at the root, the local cache is cleared to avoid stale state. We then wait for a grace period for the synchronized hashmap updates to propagate to the switches. After that, we push rules on to the switches to forward all offloadable messages to the local controller. Finally, we also enable batching of updates to offloadable state at the local controller in offload mode.

4.3.7 Implementation of use cases

We implement the two sample applications — a key-value store, and the SDN-based LTE packet core (§3.2.2)—over the Cuttlefish framework, to demonstrate and evaluate the benefits of our framework. The source code of the Cuttlefish framework and the implemented use cases is available at [39] for innovation.

Key-value store. We implemented a centralized key-value store as the basic application. We partition key-value space into offloadable and non-offloadable states by randomly marking a subset of key-value pairs as offloadable and rest as non-offloadable. We defined the get/put/del messages that only access the offloadable key-value states; these messages are marked as offloadable. We also defined the get/put/del messages that access both the non-offloadable and offloadable key-value states. These messages induce state synchronization between the root and local controllers in offload mode and are marked as non-offloadable messages. The application implements the offloadable state access using the Cuttlefish API.

We have implemented a load generator that can generate traffic with varying ratios of offloadable and non-offloadable requests. We use the IP ToS field in packet headers to identify the application messages at the switches. The application and the load generator were implemented in about 1400 lines of Java/C++ code.

SDN based LTE EPC. We implement the SDN-based LTE EPC application by extending an existing version of the code [121] built atop the Floodlight controller and OVS SDN switches, and adapting it to use the Cuttlefish API. We extended the load generator in the existing code to tag packets with message types in the IP ToS field, to enable identification of the various control plane messages. We also modified the load generator to generate traffic of varying mixes, e.g., vary the ratio of the attach requests (non-offloadable) and the service requests (offloadable). Our changes modified 1800 lines of Java/C++ code in the original application codebase.

4.4 Evaluation

We describe the evaluation of Cuttlefish framework in this section. Our evaluation aims to answer two broad questions:

- What are the performance gains of adaptively offloading computation across local controllers? (§4.4.2)
- How efficiently does Cuttlefish accomplish the process of adaptively switching modes? (§4.4.3)

4.4.1 Experimental setup

Testbed. We deployed the Cuttlefish applications over our testbed consisting of a Floodlight v1.2 as the root and local controllers, and OVS v2.3.2 switches as the data plane switches. The local controller was colocated with one of the switches. All components (controller and switches) used Ubuntu 14.04 and were hosted over separate LXC containers to ensure isolation. The containers were distributed amongst two 16-core Intel Xeon E312xx @2.6Ghz servers with 64GB RAM. The root and local controllers, and all gateway switches, were allocated 1 CPU core and 4GB RAM each.

In the offload mode of operation, when the non-offloadable traffic rate was low, we were unable to generate enough load to saturate the root controller and measure saturated throughput. So we allocated six forwarding chains for each application (i.e., six load generators and six local controllers) to generate more traffic for the root controller. We did not have enough CPU/memory resources to add more forwarding chains; therefore,

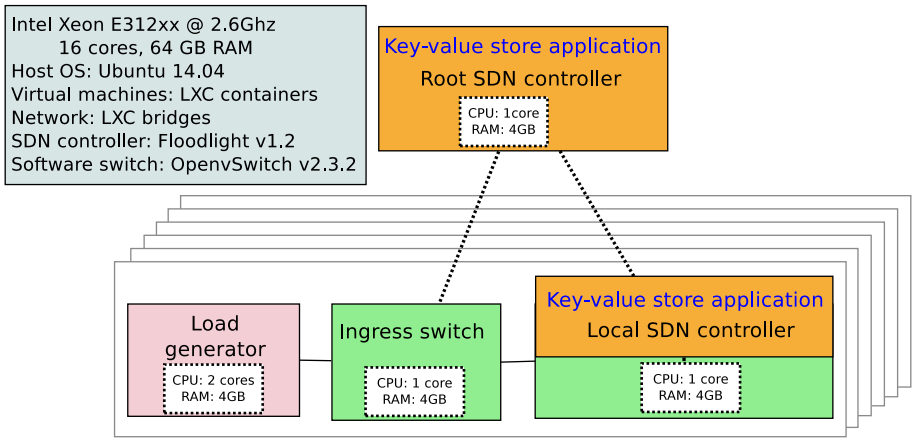


Figure 4.7: Experimental setup for the key-value store application.

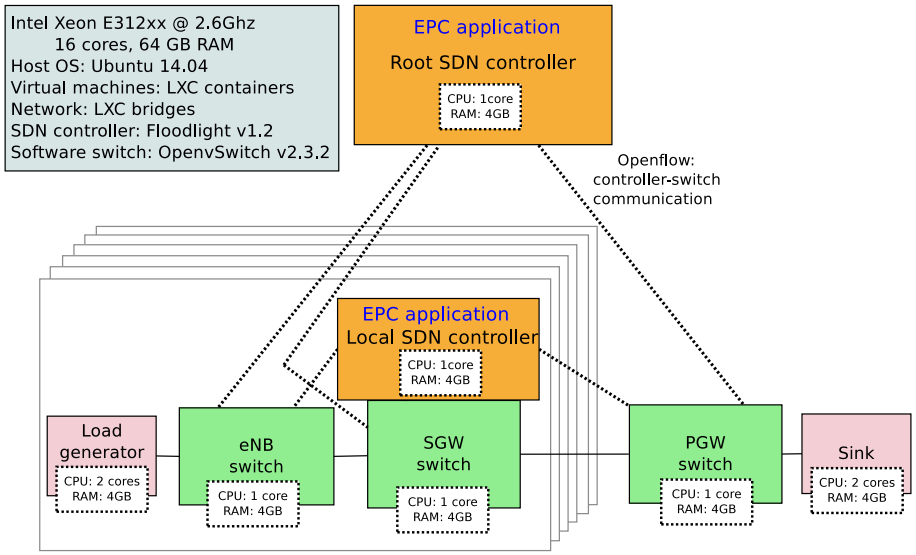


Figure 4.8: Experimental setup for the SDN-based EPC application.

there were cases when we could not saturate the root controller, but the local controllers were saturated.

We now describe the specific setup components for each application. Figure 4.7 shows the setup for the key-value store application. Each chain comprises of a load-generator, an ingress switch that routes the offloadable and non-offloadable messages to the appropriate controller based on the current SDN operation mode, and a local controller that serves the offloadable messages in the offload mode. Figure 4.8 shows the setup for the LTE-EPC application. Each chain comprises a load-generator, an eNB switch that routes the data traffic and control plane messages (offloadable/non-offloadable), an SGW switch that hosts a local controller to process offloadable messages, a PGW switch, and the sink node which is the end node for EPC data traffic.

Parameters and metrics. We generate different experiment scenarios by varying the mix of offloadable and non-offloadable messages in the control plane traffic processed by

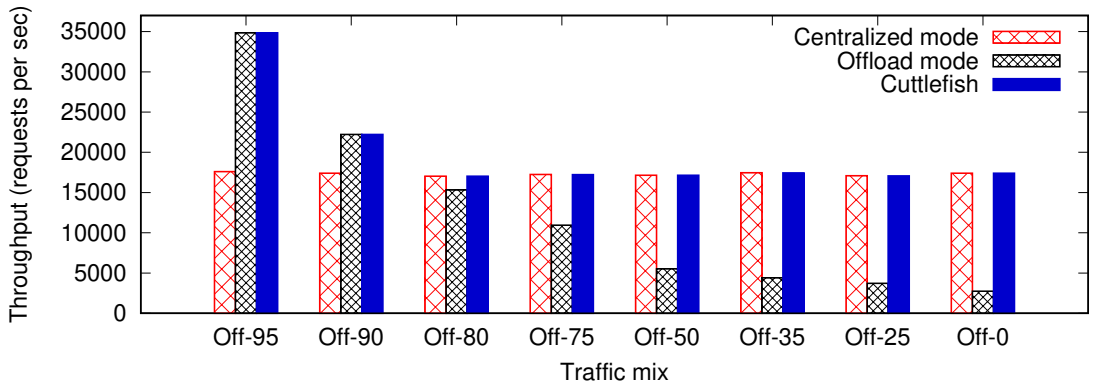


Figure 4.9: Key-value store: control plane throughput.

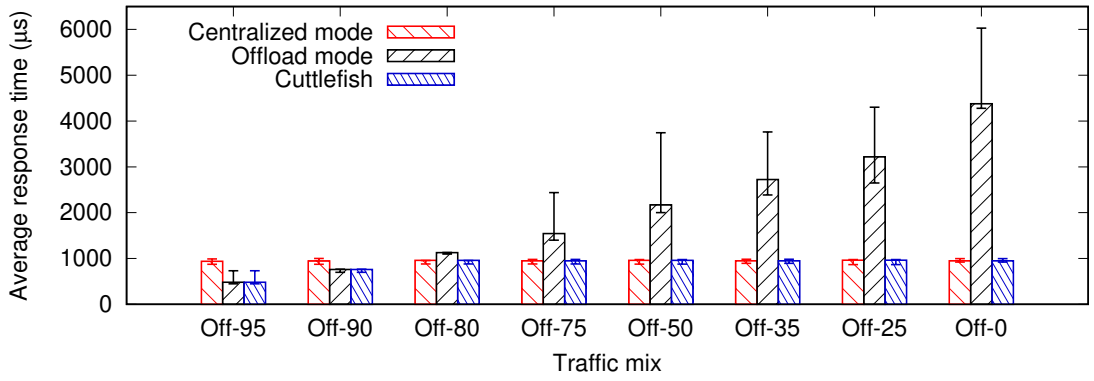


Figure 4.10: Key-value store: control plane latency.

the SDN controllers. All experiments ran for 300 seconds, and the results are averaged over three runs unless mentioned otherwise. The performance metrics measured in our experiments were the control plane throughput (number of control plane messages processed/sec) and response latency of control plane messages. We compare these metrics across three modes of operation of the application: (a) centralized mode, where all control plane messages are handled at the root controller, (b) offload mode, where non-offloadable messages are processed at the root controller and all offloadable messages are always processed at local controllers, and (c) the Cuttlefish adaptive offload mode, where offloadable messages are processed at the local controller only if the Cuttlefish adaptation algorithm detects that the synchronization costs are lower than the offload gains.

4.4.2 Efficacy of adaptive offload

We first quantify the performance gains due to the adaptive offload mechanism of Cuttlefish. We vary the mix of get and put requests in the incoming traffic and measure the performance of the Cuttlefish key-value store application. Traffic mix *Off-x* denotes

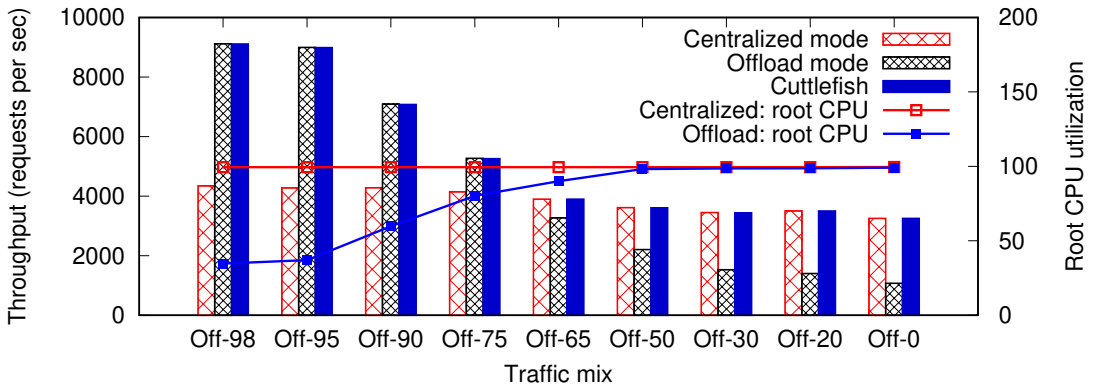


Figure 4.11: LTE EPC: control plane throughput.

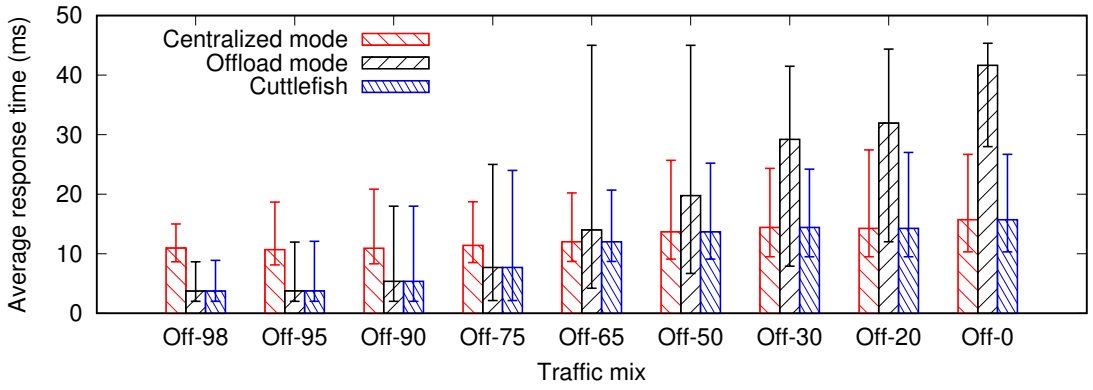


Figure 4.12: LTE EPC: control plane latency.

x% offloadable messages, i.e., get/put to the offloadable state at the local controller, and $(100-x)$% non-offloadable messages, i.e., put to the offloadable state at the root controller.

Figure 4.9 shows the throughput of all the controller modes, and Figure 4.10 shows the response latency with errorbars that represent min-max latency values. As expected, the performance of the offload mode degrades as compared to the centralized mode, as the proportion of non-offloadable traffic increases. However, across all traffic mixes, we see that the performance of the Cuttlefish adaptive offload mode matches that of the best non-adaptive mode for that traffic mix.

We observe that the Cuttlefish throughput is up to 2X higher than that of the traditional centralized mode, and its latency is up to 50% lower. Also, Cuttlefish throughput is up to 6.4X higher than that of the offload mode, and its latency is up to 80% lower. Further, the throughput and latency of Cuttlefish are almost equal to that of the optimal mode (whether centralized or offload) for a given traffic mix, because the cost of running the adaptation module is almost negligible.

Figure 4.11 and Figure 4.12 show the control plane throughput and response latency respectively of the LTE EPC application, with varying traffic mix . The errorbars shown in

Figure 4.12 represent the min-max latency values. Here, traffic mix *Off-x* denotes (100 – *x*)% non-offloadable attach and detach requests, and *x*% offloadable service requests and context release requests.

Our observations remain similar for this application, as well. That is, we see that the performance of the Cuttlefish adaptive offload mode matches that of the best non-adaptive mode for that traffic mix. The throughput of Cuttlefish is up to 2X higher than that of the traditional centralized mode, and its latency is up to 66% lower. Cuttlefish throughput is also up to 3X higher than that of the offload mode, and its latency is up to 62% lower. Similar to our previous observations, the performance of Cuttlefish matches the best performing mode for a given traffic mix. The y2-axis of Figure 4.11 shows the root controller CPU utilization for the centralized and offload modes. We observe that, in the offload mode of operation, the root controller CPU is not saturated up to traffic-mix *Off-75*, but the local controller CPU is saturated for all six forwarding chains. As we said earlier, we did not have enough CPU/memory resources to add more forwarding chains to saturate the root controller. The root CPU utilization for Cuttlefish/offload mode for *Off-98* is ~35%, which means that the performance values of the offload and Cuttlefish mode will be better than the above indicated values when the root controller is saturated.

4.4.3 Convergence of adaptive offload

In our next set of experiments, we demonstrate the effectiveness of the adaptation mechanism and measure the amount of time taken by Cuttlefish to compute the correct mode of operation and switch to it when the traffic mix changes. Our last experiment presents the limitations of the adaptation mechanism and discusses the parameters that impact the accuracy of the adaptation decision.

At the end of each epoch, Cuttlefish gathers the parameter values for adaptation decision, as discussed in §4.3.4, and substitutes the parameter values in equation 4.1 to estimate the synchronization cost at root controller (*put_rate*), and in equation 4.2 to estimate the offload benefits (*access_rate*). The computed *put_rate* and *access_rate* values are substituted in equation 4.3 when in offload mode, and in equation 4.4 when in centralized mode, to test if controller mode migration is required ($\Delta = 0.2$).

In our first experiment, the key-value store application generates get/put traffic for a duration of 2400 seconds, while varying the traffic mix during the experiment as follows. During the first 300s of the experiment, 90% of the traffic comprises of offloadable messages. The offloadable fraction changes to 95% for the next 300s, to no offloadable traffic for the next 300s, to 80% offloadable messages for the next 300s, to 95% offloadable mes-

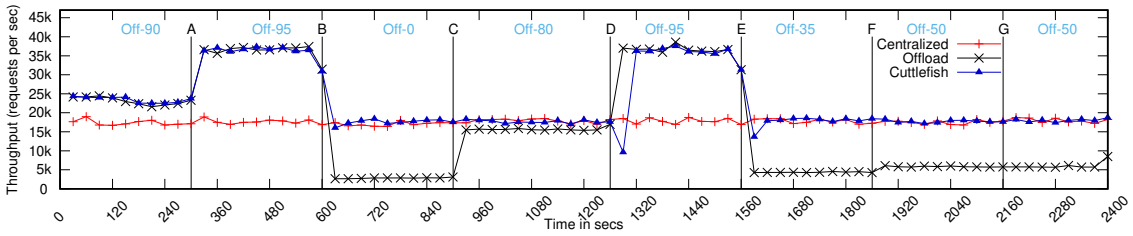


Figure 4.13: Throughput with varying traffic mix for the key-value store application.

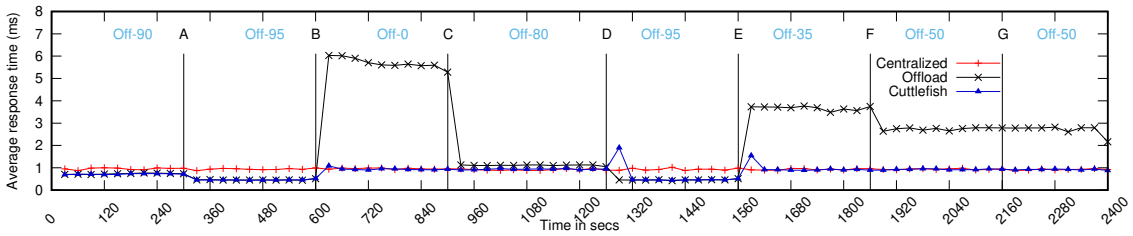


Figure 4.14: Latency with varying traffic mix for the key-value store application.

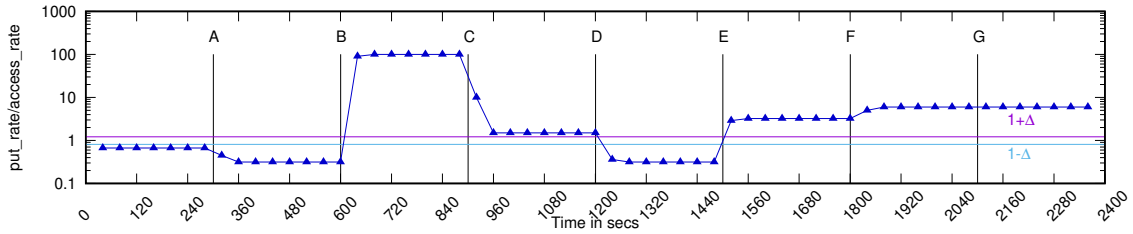


Figure 4.15: Key-value store: adaptation metric for mode switch.

sages for the next 300s, to 35% offloadable messages in the next 300s, and finally 50% offloadable messages in the final 600s.

Figures 4.13 and 4.14 show the throughput and the response latency, respectively, of the key-value store application, sampled every 30 seconds for the duration of the experiment. Figure 4.15 shows the fraction $\frac{put\_rate}{access\_rate}$ and the thresholds for migration from the offload to centralized mode $(1 + \Delta)$, and vice-versa $(1 - \Delta)$. The Cuttlefish adaptation module observes the fraction and the thresholds, at the end of each epoch to make the mode switch decision. From the graphs, we see that when the traffic consists of predominantly offloadable requests in the first 600s (up to point B in the graphs), Cuttlefish operates in offload mode. After point B, the non-offloadable component in the traffic-mix exceeds such that the cost of the offload is much higher than the offload benefits ($\frac{put\_rate}{access\_rate} > 1 + \Delta$), the adaptation algorithm switches from the offload mode to centralized mode, and stays in this mode up to point D. After point D, the non-offloadable traffic reduces ($\frac{put\_rate}{access\_rate} < 1 - \Delta$), the Cuttlefish adaptation algorithm switches to offload mode, and stays there up to point E. After point E, the traffic mix incurs a high synchronization cost, and Cuttlefish switches to centralized mode, and remains in this

mode for the rest of the experiment. Throughout the experiment, we observe that the Cuttlefish adaptation algorithm always correctly identifies the best performing controller mode and correctly switches to it. We observe a transient drop in performance after points B, D, and E, due to the mechanisms of migrating between modes in Cuttlefish. We find that the Cuttlefish framework takes around 20–30 seconds to switch to a new mode of operation after change in the traffic mix. This switching duration is obviously a function of the frequency at which we invoke our decision algorithm (every 10 seconds), and on size of the application-centric state requiring synchronization (700 key-value pairs in this experiment).

In our second experiment with the LTE EPC application, we generate traffic for the EPC setup for a duration of 1200 seconds, while varying the traffic mix during the experiment as follows. The fraction of offloadable traffic (service request, context release request) is 95% during the first 300s of the experiment, which changes to 20% in the next 300s, then back to 95% for next 300s, and it is 20% for the final 300s.

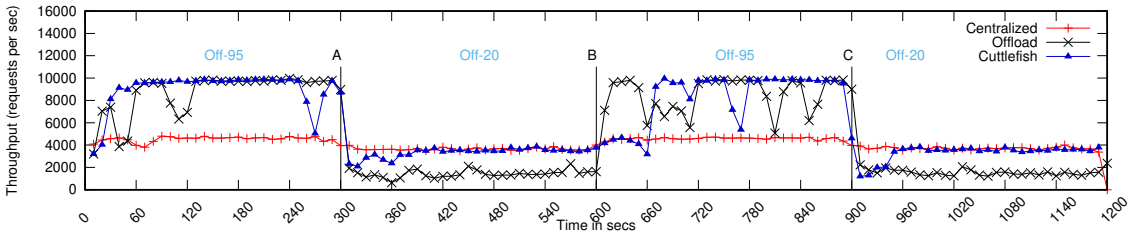


Figure 4.16: Throughput with varying traffic mix for the LTE EPC application.

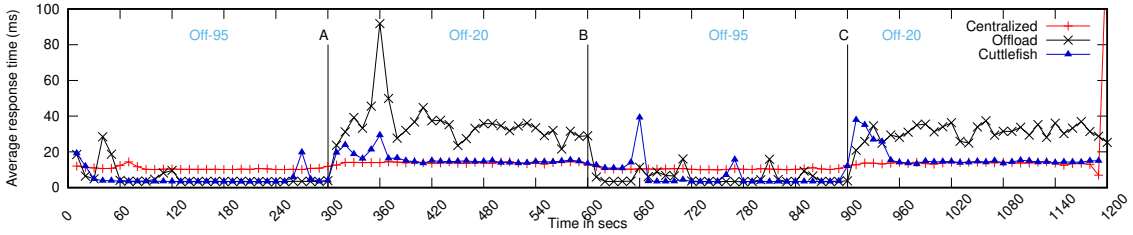


Figure 4.17: Latency with varying traffic mix for the LTE EPC application.

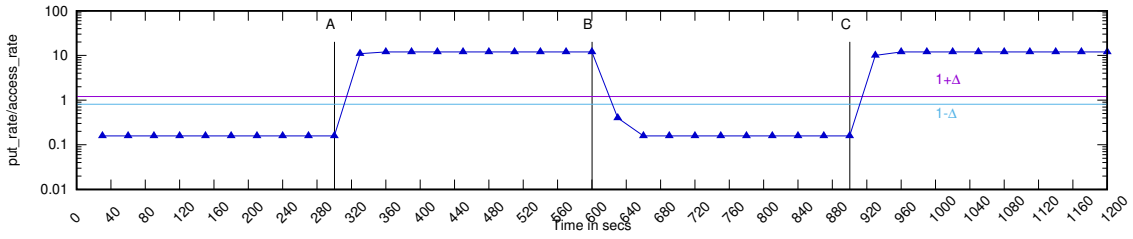


Figure 4.18: LTE-EPC: adaptation metric for mode switch.

Figures 4.16 and 4.17 show the control plane throughput and latency, respectively, of the LTE EPC application, sampled every 10 seconds for the duration of the experiment. Figure 4.18 shows the value of $\frac{put\_rate}{access\_rate}$ computed by the Cuttlefish adaptation algorithm during the experiment. The Cuttlefish framework behaves similar as explained for the previous experiment. Cuttlefish stays in offload mode up to point A, after which the non-offloadable component in the traffic-mix exceeds such that $\frac{put\_rate}{access\_rate} > 1 + \Delta$, and the Cuttlefish adaptation algorithm switches the controller to centralized mode, and stays there upto point B. After point B, the non-offloadable component in the traffic-mix reduces ($\frac{put\_rate}{access\_rate} < 1 - \Delta$), and Cuttlefish shifts to offload mode and stays up to point C. After point C up to the end of the experiment, the synchronization cost is high, and Cuttlefish switches to the centralized mode and stays. We conclude from this experiment that the Cuttlefish framework takes around 30–70 seconds to identify and switch to a new mode of operation after a change in traffic mix. This switching duration also depends on the frequency at which we invoke our decision algorithm (every 10 seconds), and on size of the application-centric state requiring synchronization (1000 key-value pairs in this experiment).

In our third experiment, we discuss the limitations of the proposed adaptation mechanism. Given that Cuttlefish takes a few tens of seconds to identify the correct mode and switch between modes, it is expected that Cuttlefish will not perform well if the traffic mix changes very frequently. Also, Cuttlefish may not adapt to the correct SDN mode if the monitoring interval (epoch size) is too long or too short. To observe the limitations, we planned an experiment with bursty traffic and configured the monitoring interval of the adaptation mechanism to 30s. Figures 4.19 and 4.20 show the control plane throughput and latency, respectively, of the LTE EPC application, sampled every 10 seconds for the duration of the experiment. Figure 4.21 shows the value of $\frac{put\_rate}{access\_rate}$ computed by the Cuttlefish adaptation algorithm during the experiment. To demonstrate the bursty traffic scenario, we generate traffic for the EPC setup for a duration of 480 seconds, while varying the traffic mix during the experiment as follows. The fraction of offloadable traffic (service request, context release request) is 60% during the first 240s of the experiment, which changes to 90% in the next 60s, then to 35% for next 60s, and it is 90% for the final 120s.

For the initial traffic-mix, the adaptation metric, $\frac{put\_rate}{access\_rate} > 1 + \Delta$, so Cuttlefish runs in centralized mode. After point A, the non-offloadable component in the traffic-mix reduces such that $\frac{put\_rate}{access\_rate} < 1 - \Delta$. Since the monitoring interval is configured to 30s, the adaptation algorithm identifies the need for mode switch at `t=270s` and takes around 30s (`t=300s`) for migrating to the offload mode. After point B, the non-offloadable

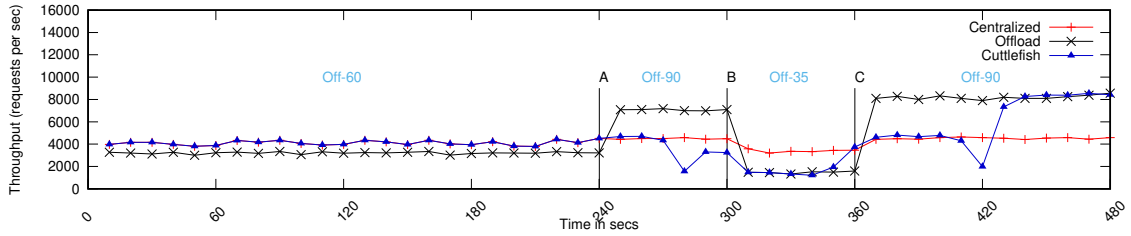


Figure 4.19: Throughput with bursty traffic for the LTE EPC application.

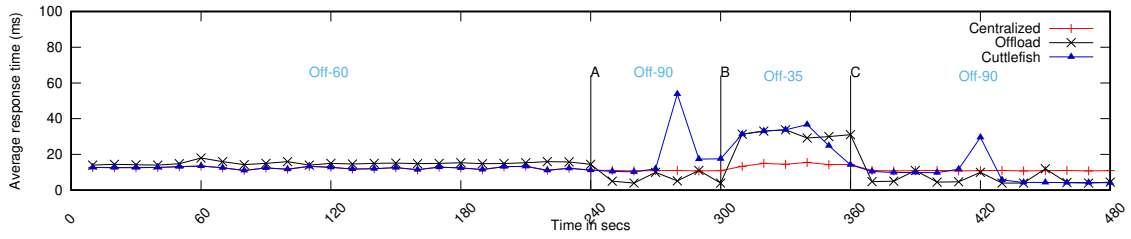


Figure 4.20: Latency with bursty traffic for the LTE EPC application.

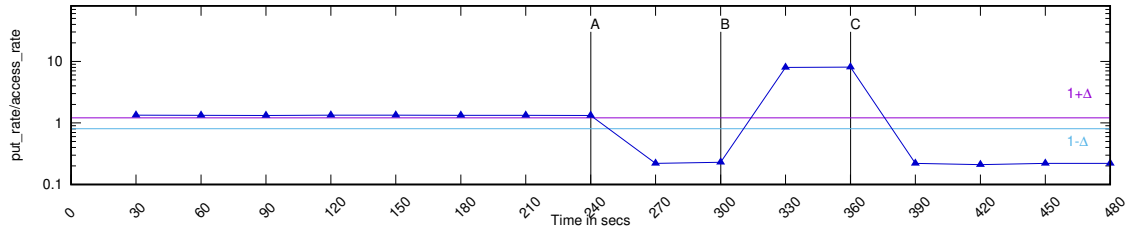


Figure 4.21: LTE-EPC bursty traffic: adaptation metric for mode switch (epoch = 30s).

component in the traffic-mix exceeds ($\frac{put\_rate}{access\_rate} > 1 + \Delta$), but this change is identified at the next epoch, `t=330s`. It takes around 25s (`t=355s`) for migrating to the centralized mode. After point C up to the end of the experiment, the synchronization cost is low ($\frac{put\_rate}{access\_rate} < 1 - \Delta$). Cuttlefish identifies the change in traffic-mix at `t=390s` and takes around 40s (`t=430s`) to switch to the offload mode and stays.

We conclude from this experiment that the Cuttlefish framework takes a long time to switch mode after traffic-mix changes. The delay is caused due to the large epoch size (30s) and the switching time (25–30s). The switching duration depends on the size of the application state that requires synchronization (1000 key-value pairs in this experiment). We also observed rapid oscillations between the centralized and offload modes due to bursty traffic characteristics.

Of course, Cuttlefish can perform better if it reduces the monitoring interval from 30s to something smaller. We tested our algorithm with epochs as low as 5s. We found that the algorithm occasionally makes wrong decisions for shorter epochs due to incorrect estimation of the adaptation metric. The monitoring interval of 10s helped Cuttlefish to converge quickly under most scenarios. Also, the root controller CPU usage was within

2% for 10s epoch size, which is acceptable. Note that, even after careful calibration of monitoring interval, Cuttlefish will not perform well if the traffic-mix changes frequently.

To summarize, the Cuttlefish adaptation metric would fail to decide the correct mode of operation for the following conditions.

- If the traffic-mix changes very frequently resulting in traffic spikes (bursty traffic). Such traffic behavior could cause the system to oscillate between the centralized and offload mode, causing unnecessary state migration and application performance degradation.
- If the monitoring interval is not configured correctly (too high or too low), the adaptation metric based decision could be delayed or incorrect.
- Suppose Cuttlefish is operating in the offload mode, and the network link traffic causes a bottleneck between the root and local controllers, but the root controller CPU is not the bottleneck. In that case, Cuttlefish will continue to function in the degraded offload mode.

4.4.4 Summary of results

Our evaluation of the key-value store SDN application demonstrated that Cuttlefish improved control plane throughput by ~2X and control plane latency by ~50% as compared to the traditional SDN design, and improved control plane throughput by ~6.4X and control plane latency by ~80% as compared to the offload SDN design. Also, our evaluation of the LTE-EPC SDN application demonstrated that Cuttlefish improved control plane throughput by ~2X and control plane latency by ~66% as compared to the traditional SDN design, and improved control plane throughput by ~3X and control plane latency by ~62% as compared to the offload SDN design.

The root controller was saturated for all traffic-mix in the centralized mode, for both the applications. In contrast, for the EPC application case, the root controller utilization in the offload mode for the best performing traffic-mix, *Off-99*, was only 35%. In fact, the root controller was not saturated until *Off-75* while operating in offload mode. It means that the offload performance will improve if enough load is generated (with additional forwarding chains), and the current performance improvement values are pessimistic. Our evaluation depicts that Cuttlefish correctly chooses the SDN controller mode, centralized or offload, to optimize application performance.

4.5 Summary

We presented the design and implementation of Cuttlefish, a hierarchical SDN controller that offloads a subset of global (offloadable) SDN application computations and the corresponding offloadable states to the local controllers on switches, to scale SDN control plane capacity. Cuttlefish incorporated an adaptive state offload capability to balance the tradeoff between performance gains due to offloading of offloadable state, and the cost of synchronizing this state across the root and local controllers. We developed two sample applications—the SDN-based LTE packet core and a key-value store—and demonstrated the efficacy of the Cuttlefish framework. Our framework, based on the popular Floodlight SDN controller, is available for use by SDN application developers [39].

Chapter 5

Offload of SDN Applications to Programmable Switches

In the previous chapter (Chapter §4), we observed that offload of control plane computations to local controllers resulted in high throughput and latency benefits. This chapter describes how we can further accelerate applications by offloading control plane computations to the hardware programmable switches, close to the user. The offload to the hardware switch avoids the packet traversal latency through the local controller's network stack and the application stack, and provides high throughput. To demonstrate the efficacy of our idea, we implement our offload idea for the real-life use case, the CUPS-based (Control User Plane Separation) mobile packet core [97]. We present the design and implementation of our proposed system, TurboEPC, a redesign of the mobile packet core that revisits the division of work between the control and data planes.

5.1 Motivation and problem description

The telecom industry has endorsed the SDN design to gain benefits like control plane programmability and individual component scaling. Therefore, the future mobile packet core networks adopt the CUPS-based model that we have described in §2.4. We have discussed the SDN controller scalability problem earlier (§2.2). The centralized SDN controller located at the mobile core network can become a bottleneck with the increase in the control plane traffic. We have discussed the efforts of the research community towards scaling the software mobile packet core in §2.4.3, but we believe that they are not good enough to ensure the real-time experience to the mobile users. TurboEPC takes a few steps towards this goal.

EPC procedure	Number of transactions/sec
Attach	9K
Detach	9K
S1 release	300K
Service request	285K
Handover	45K

Network load when total subscribers in the core = 1 million

Table 5.1: Sample EPC load statistics [5, 6].

Our work is motivated by following observations pertaining to the signaling traffic in the mobile packet core.

1. As discussed in §2.4.3, the signaling traffic is proliferating [100, 101] due to the increase in the number of mobile devices, and the increased number of signaling messages exchanged [100] between the mobile user and the network. The high signaling load at the mobile packet core with the centralized software control plane makes it challenging to satisfy signaling traffic SLAs.

2. We have classified the mobile packet core signaling procedures into two types, offloadable and non-offloadable procedures (§3.2.2), based on their frequency (see Table 5.1) and nature of the processing. A small percentage of the signaling traffic consists of procedures like the *attach and detach* procedure (1–2% of total traffic, as per [5, 6]), the *handover* procedure (~5%) that is executed when the user moves across regions of the mobile network. These procedures are identified as non-offloadable because they access/update global state (non-offloadable state), for example, the free IP address pool and the HSS (Home Subscriber Server) database. Whereas, a significant fraction of the signaling traffic (~63–90%) is made up of procedures like the *S1 release*, and the *service request* procedure. These procedures are identified as offloadable because they only access/update the per-session user context (offloadable state) like the tunnel identifiers for data forwarding. We have described more details about these EPC procedures in §2.4.2. We can protect the core from high signaling load with a positive side-effect of lower response time latencies if we process the high-frequency signaling messages at the hardware programmable edge switch, closer to the user.

3. The mobile network minimizes the UE's power consumption and network resource usage by switching the UE to *IDLE* state whenever possible. Suppose that the UE

is in *IDLE* state, and the data request arrives; the mobile core invokes the *service request* control plane procedure. The service request procedure assigns resources and switches the UE to *CONNECTED* state, and after that, data can be sent or received. The current mobile standards transit the UE state from IDLE to CONNECTED state in the order of *~50 ms* [122]. But, the 5G standards have end-to-end latency requirements of *10 ms* for broadband data access [30]. Therefore, it is impossible to satisfy the 5G latency requirements using the current mobile network standards. One solution is to keep all the UEs in the CONNECTED state, but this increases power consumption and also generates considerable amount of signaling traffic to keep the connection alive. It is not advisable to waste power, especially for low power, battery-operated IoT devices. If we offload the processing of the service request and S1 release control plane procedures close to the UE (at/close to the base station) along with the user state, we can easily satisfy the 5G latency requirements. Also, the devices can save power by more frequently transiting to the idle state.

As discussed in §2.3, the data plane switches are evolving from fixed-function hardware towards programmable components that can forward traffic at line rate while being highly customizable [36, 37]. TurboEPC improves the control plane performance of the CUPS-based mobile packet core by offloading the offloadable control plane procedures (S1 release and service request) from the control plane onto the programmable hardware switches, close to the mobile user.

5.2 Key idea and challenges

The offload of frequent offloadable signaling procedures to the programmable hardware switches improves both control plane throughput (by utilizing spare switch capacity for handling signaling traffic) and latency (by handling signaling traffic closer to the end-user at the switches). TurboEPC modifies the processing of the non-offloadable messages (like the attach and handover procedure) in the control plane so that modifications to the user-specific context generated/modified during such procedures is immediately pushed to the data plane switches. This user context is stored at the switches along with the forwarding state needed for data plane processing, and is used to process offloadable signaling messages within the data plane switch itself.

Challenges and contributions

We have already addressed the challenge of identification of offloadable EPC signaling messages in §3.2.2, using our offload guide (§3.2.1) to enable offload of the EPC

	User state (in bytes)	**Forwarding state (in bytes)**
eNB	0	32
SGW	64	28
PGW	0	19

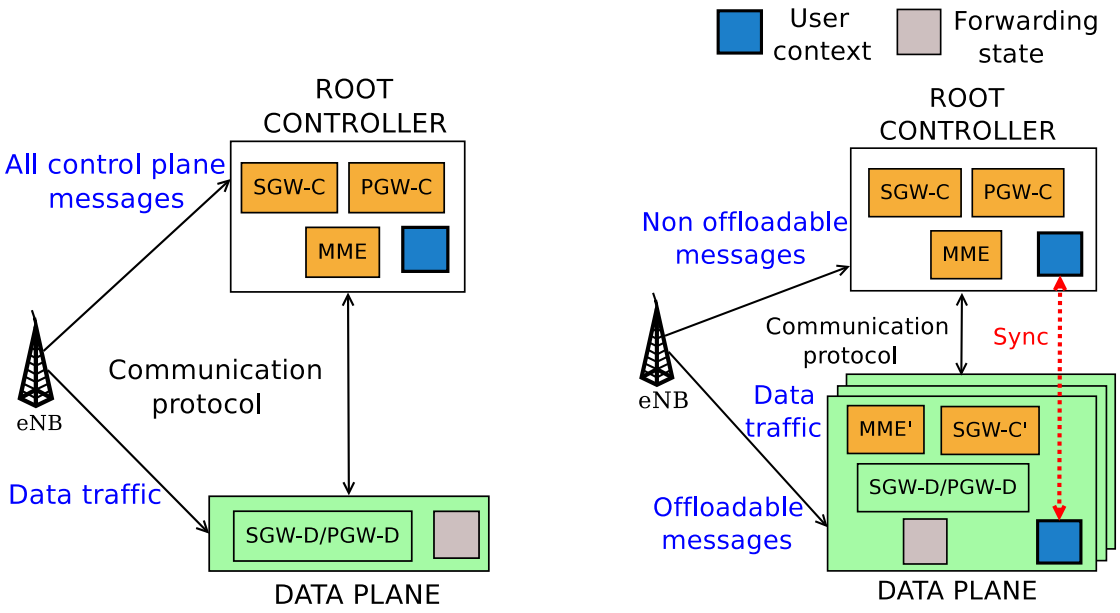
Table 5.2: Size of state stored at TurboEPC switches.

application to the programmable switch hardware. TurboEPC addresses the following other challenges while implementing the offload for the CUPS-based LTE-EPC application.

- **Inconsistency of offloaded state.** The offloadable state is cached at the programmable hardware switches for offloadable message processing. Non-offloadable messages like the handover message require access to both the offloadable and non-offloadable state, and are processed at the root controller. TurboEPC lazily synchronizes the offloadable state updates from the switch to the root controller. Therefore, the handover message may use stale offloadable state, which may result in incorrect application behavior. TurboEPC employs an on-demand state synchronization technique that ensures state consistency and reduces synchronization costs, as well (discussed in §5.3.1).

- **Memory limitations at the programmable switches.** The hardware programmable switches have a small amount of memory to store the application state. A typical mobile core must handle millions of actively connected users [5, 38]. Table 5.2 shows the size of the offloadable user-context at the TurboEPC switch. The recent high-end programmable switches like Barefoot Tofino [36] can only store the user context for a few 100K users, whereas the Netronome programmable NIC hardware used in our prototype implementation [117] could only store user context for 65K users. Therefore, it is unlikely that a single data plane switch can accommodate the contexts of all the users connected to the mobile network core. To overcome this challenge, TurboEPC partitions the offloadable state across multiple switches, which increases the probability of storing the offloadable state for all users at the data plane. We discuss multiple state partitioning techniques in §5.3.2.

- **State losses due to failure of target switches.** The programmable hardware switches have the most recent version of the offloaded user context, and if they fail, the latest user context is lost. The UE's view and the network's view of the user's



(a) Traditional CUPS-based EPC

(b) TurboEPC

Figure 5.1: TurboEPC Design.

context become inconsistent, and future message processing may result in incorrect application behavior. TurboEPC overcomes this challenge by employing state replication techniques for the offloadable state across the programmable switches. The state replication runs as a data plane application to support line-rate replication. The SDN controller monitors the network topology, and when a switch failure is detected, it implements a failover mechanism to tackle switch failures (discussed in §5.3.3).

5.3 TurboEPC design

This section describes the TurboEPC design that enables the offload of EPC messages to the programmable switches. We also describe the details on how TurboEPC addresses the challenges mentioned in §5.2. We begin with an overview of TurboEPC's basic design (§5.3.1) and then describe design features related to scalability (§5.3.2) and fault tolerance (§5.3.3).

5.3.1 Design overview

Figure 5.1 compares the CUPS-based traditional EPC design with TurboEPC. In the traditional CUPS-based EPC design (Figure 5.1(a)), the MME, SGW-C, and PGW-C components are implemented within a centralized *root* SDN controller in the control plane, while the data plane processing is performed at data plane switches (SGW-

D & PGW-D). The eNB forwards all control plane (a.k.a., signaling) traffic to the root controller, which processes these messages and installs forwarding state at the S/P-GW switches. All control plane state, including the per-user context, is maintained only in the control plane.

In contrast, in the TurboEPC design (shown in Figure 5.1(b)), the eNB forwards offloadable messages (e.g., S1 release and service request) to the data plane S/P-GW switches. We assume that the eNB is capable of analyzing the header of a signaling message to determine if it is offloadable or not. To enable the processing of offloadable signaling messages in the data plane, the root controller in TurboEPC pushes the offloadable per-user context generated/modified by non-offloadable signaling messages into the data plane switches. The user context that is pushed to the data plane consists of a mapping between the UE identifier and the following subset of information pertaining to the user: the tunnel identifiers (TEIDs), GUTI (Globally Unique Temporary Identifier), and the UE connection state (*CONNECTED/IDLE*). This user context is stored in data structures of data plane switch, much like the forwarding state, and consumes an additional ≈64 bytes of memory over and above the ≈32 bytes of forwarding state in our prototype (as shown in Table 5.2).

Offloadable signaling messages that arrive at the edge data plane switches (close to the eNB) are processed within the switch data plane itself, by accessing and modifying the offloaded per-user context. For example, the S1 release request processing requires the TurboEPC switch data plane to delete the uplink/downlink TEIDs at the eNB and the downlink TEID at the SGW, change the user connection state to idle, and update GUTI if required. Because these offloadable messages reach the switch at least a few tens of seconds (idle timeout) after the root controller pushes the context, the state offload does not cause any additional delays while waiting for the state to be synchronized. If the signaling message requires a reply to be sent back to the user, the reply is generated and sent by the switch data plane as well.

Consistency of offloadable state. Note that, the user context can be modified by the offloadable signaling messages within the switch data structures, and the latest copy of this state resides only in the data plane. TurboEPC does not synchronize this state back to the root after every modification to the offloaded state, because doing so nullifies the performance gains due to the offload in the first place. Instead, TurboEPC lazily synchronizes this state with its master copy at the root controller only when required. That is, all future offloadable messages will access the latest copy of the offloaded state within the data plane itself, and non-offloadable messages that do not depend on this offloaded state will be directly forwarded to the root by the eNB. However, some non-offloadable mes-

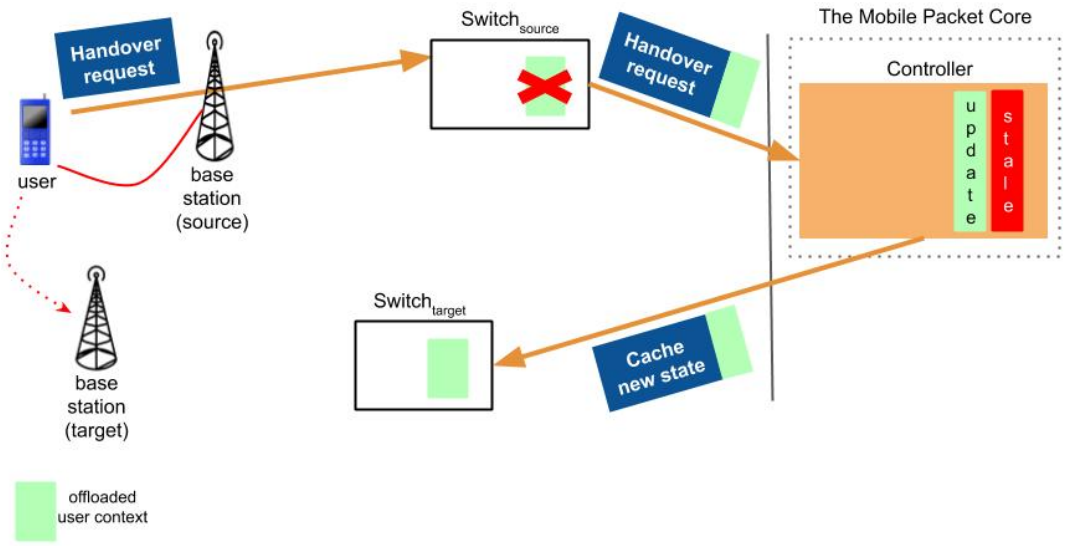


Figure 5.2: Handover message processing in TurboEPC.

sages in EPC (e.g., handover messages) require access to both the latest offloaded user context in the data plane and the non-offloaded state stored in the root. Figure 5.2 shows how TurboEPC processes handover messages when the user is moving from the source base station (eNB) to the target base station. The handover message is first sent to the data plane switches by the eNB, and the switch performs the part of message processing that does not require access to the global non-offloadable state. Next, the message is forwarded from the data plane switch to the root controller, with a copy of the modified user context (that is subsequently invalidated at the switch by the root controller) appended to the packet, in order to complete the rest of the processing at the root correctly. Once the mobile user is successfully migrated to the target network, the most recent user context is pushed to the target switch to help processing of offloadable messages at the edge programmable hardware switch.

We acknowledge that TurboEPC introduces a small amount of overhead during the processing of non-offloadable handover messages since we need to piggyback the user context from the switch to the root controller, as described above. This overhead may be acceptable in current networks, because the handover messages comprise only 4–5% [5, 6] of all signaling traffic. However, the handover traffic can increase for future networks, e.g., with small cells in 5G. We plan to revisit our handover processing to reduce overhead in such use cases as part of our future work.

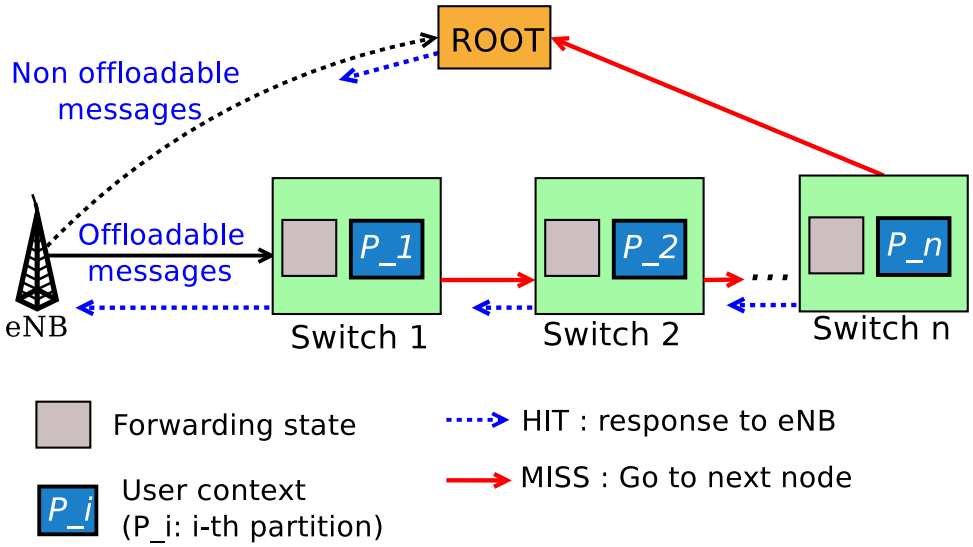


Figure 5.3: User context distributed over set of switches connected in series.

5.3.2 Partitioning for scalability

To overcome single switch memory limitations, and maximize handling of offloadable messages at the data plane, TurboEPC relies on multiple programmable switches in the core network. TurboEPC partitions the user context required to handle offloadable messages and distributes the partitions among multiple data plane switches along the path from the eNB to S/P-GW (possibly including the S/P-GW itself) [97]. Further, if the data plane switches cannot accommodate all user contexts even with partitioning, some subset of the user contexts can be retained in the root controller itself. With this design, any given data plane switch stores the contexts of only a subset of the users and handles the offloadable signaling messages pertaining to only those users. The switches over which the partitioning of user context state is done can be connected in one of two ways, as we describe below.

Series design. In the *series design* shown in Figure 5.3, the contexts of a set of users traversing a certain eNB to S/P-GW path in the network are split amongst a series of programmable switches placed along the path. When an offloadable control plane message arrives at one of the switches in the series, it looks up the user context tables to check if the state of the incoming packet's user exists on the switch. If it exists (a hit), the switch processes the signaling message as discussed in §5.3.1. If the user context is not found (a miss), the packet is forwarded to the next switch in the series until the last switch is reached. If the user context is not found even at the last switch, the message is forwarded to the root controller, and is processed like in the traditional EPC.

Parallel design. Figure 5.4 depicts a *parallel design*, where the user context is distributed amongst programmable switches located on multiple parallel network paths between the

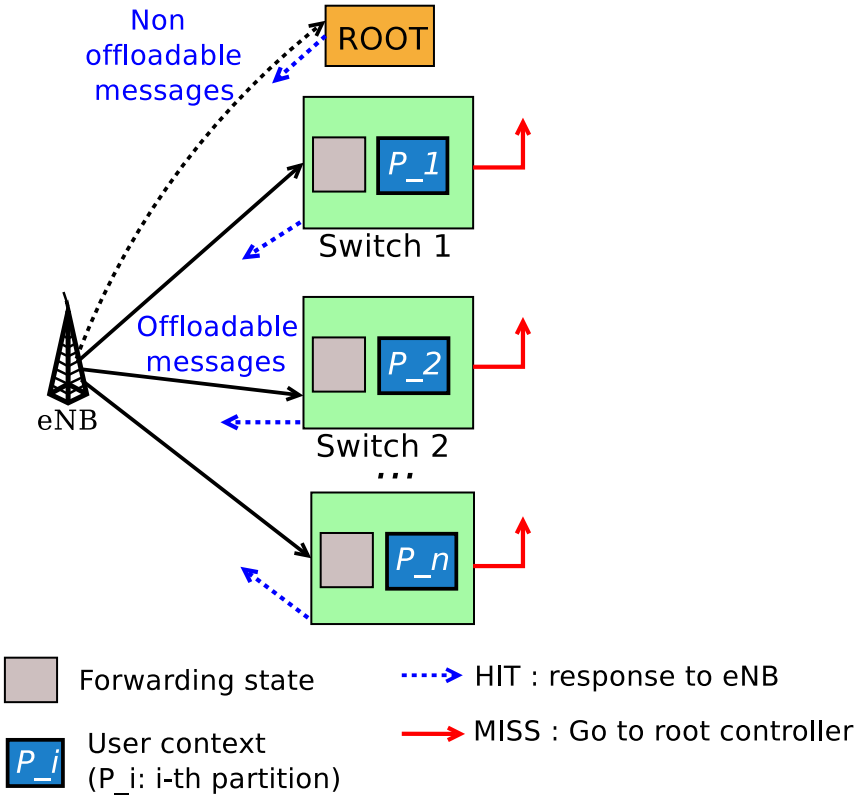


Figure 5.4: User context distributed over set of switches on parallel network paths.

eNB and the S/P-GW in the network. The difference from the series design is that the eNB now needs to maintain information on how the user contexts are partitioned along multiple paths, and must forward offloadable messages of a particular user along the correct path that has the user's state. The parallel design entails the extra step of parsing the signaling message header to identify the user, and an additional table lookup to identify the path to send the message on, at the eNB. Offloadable signaling messages that do not find the necessary user context at the switches on any of the parallel paths are forwarded to the root. While the series design leads to simpler forwarding rules at the eNB, the parallel design lends itself well to load balancing across network paths. Note that, while our current implementation supports only the simple series and parallel designs described above, a network could employ a combination of series and parallel designs, where user contexts are partitioned across multiple parallel paths from the eNB to the S/P-GWs, and are further split amongst multiple switches on each parallel path. Across all designs, the root controller installs suitable rules at all switches to enable forwarding of signaling messages towards the switch that can handle it. §5.5 compares the performance of both designs and evaluates the impact of partitioning state on TurboEPC performance.

Partitioning user context. Given a fixed and limited amount of storage in the programmable dataplanes, the question of how best to partition user contexts across multiple

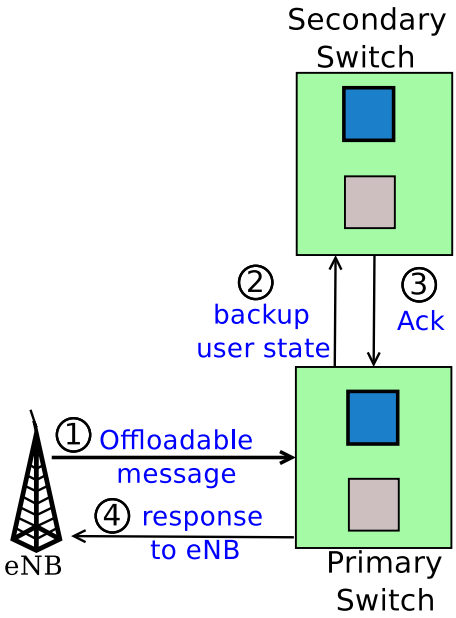


Figure 5.5: Fault tolerance in TurboEPC.

programmable switches in a large network is vital to address. The partitioning decision depends upon many factors, including the number of active users, the size of the core network, the capacity of the programmable switches, and the routing and traffic engineering policies employed within the network, and is beyond the scope of this work. Another interesting question that we defer to future work is deciding which users should be handled at which switches. With the advent of new use cases such as vehicular automation, IoT, smart sensors, and AR/VR in next-generation networks, it is becoming essential to provide ultra-low latency and ultra-high reliability in processing signaling traffic of some users. Subscribers who require low latency for their frequent signaling requests but are not highly mobile (e.g., smart sensors) are ideal candidates to offload to the data plane. It is also conceivable to think that an operator would wish to offload the contexts of premium subscribers. TurboEPC can support any such operator-desired placement policy.

5.3.3 Replication for fault tolerance

In TurboEPC, a subset of the user context is pushed into the data plane switches during the attach procedure. This context is then modified in the data plane tables during the processing of subsequent offloadable signaling messages. For example, the S1 release message changes the connection state in the context from connected to idle. In the case of a switch failure, such modifications could be lost, leaving the UE in an inconsistent state. For example, a UE might believe it is idle while a stale copy of the user context at the root controller might indicate that the user is actively connected.

To be resilient to such failure scenarios, TurboEPC stores the user context at one *primary* data plane switch, and another *secondary* switch. During the processing of non-offloadable messages such as the attach procedure, the root controller pushes the user context to the user's primary as well as the secondary switch. The root controller also sets up forwarding paths such that offloadable signaling messages of a user are directed to the primary switch of the user. Upon processing an offloadable message, the primary switch first synchronously replicates the updated user context at the secondary switch, before generating a response to the signaling message back to the user, as shown in Figure 5.5. Our current implementation uses simple synchronous state replication from the primary to one other secondary switch, and is not resilient to failures of both the primary and secondary switches in quick succession. We plan to evolve our design for replication across multiple secondary switches as part of future work, using techniques from recent research such as Netchain [123] and SwiShmem [124]. For example, SwiShmem uses a register data structure to store the distributed state. SwishShmem proposes in-network mechanisms to provide different consistency levels (strong, weak, eventual) and failure management using state replication.

In our implementation, suppose the message from the primary switch to the secondary switch or the ACK from the secondary switch to the primary switch is lost; the user application will retry the signaling message and recover from the loss. If a primary switch fails before replication completes, no response is sent to the user, the user will retry the signaling message, and will be redirected to a new switch after the network repairs the failure. If the primary switch fails after successful replication, the SDN controller will be notified of the failure in the normal course of events, e.g., in order to repair network routes, and the TurboEPC application installs forwarding rules to route subsequent offloadable messages of the user to the secondary switch. The root controller also synchronizes itself with the latest copy of user context from the now primary (former secondary) switch and repopulates this context at another new secondary switch. Users served by the failed switch may see a temporary disruption in offloadable message responses (along with a disruption in data plane forwarding) during the time of failure recovery, and we evaluate the impact of such disruptions in §5.5.

Tradeoff between scalability and fault-tolerance. To scale the mobile network core application, TurboEPC partitions the state across multiple switches in the network. However, to achieve fault tolerance, TurboEPC creates a backup copy of the state requiring twice the amount of state. With multiple replica copies, the required memory further increases. We have a fixed number of programmable switches in the network; therefore, we have an upper bound on the total memory in the network dataplane. Therefore, we need a

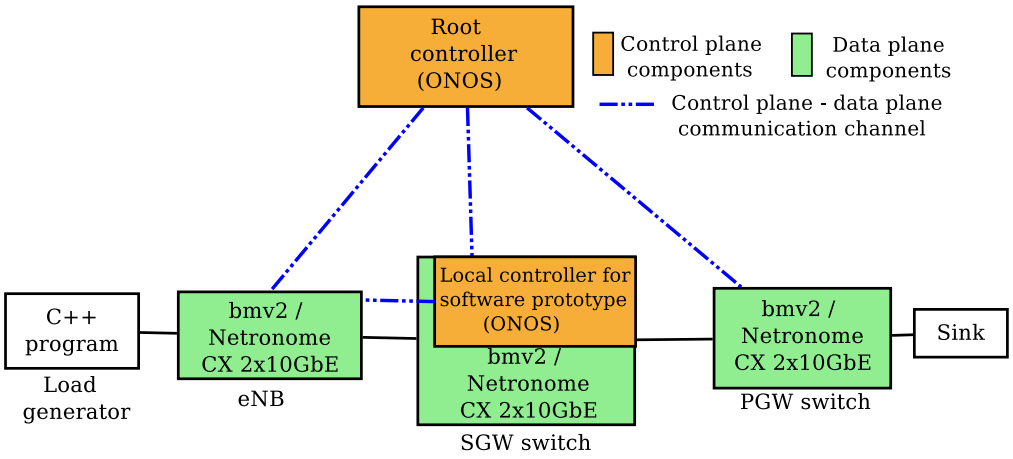


Figure 5.6: TurboEPC implementation.

tradeoff to be made between scalability and fault tolerance. For the mobile packet core application, we cannot ignore fault tolerance, as the mobile user's state should be consistent and available. To optimally utilize the limited switch memory, we should implement state eviction policies based on parameters such as the traffic pattern, the frequently accessed traffic class (latency-sensitive control plane or not), the mobility rate, and the user priority. However, for general applications, we must tradeoff fault-tolerance for the ephemeral state for improved scalability.

5.4 TurboEPC implementation

We implemented simplified versions of the CUPS-based traditional EPC and TurboEPC in order to evaluate our ideas. We have built our prototype by extending the SDN based EPC implementation available at [39, 121]. Our implementation supports a basic set of procedures: attach, detach, handover, S1 release, and service request in the control plane, and GTP-based data forwarding. While our implementation of these procedures is based on the 3GPP standards, complete standards compliance was not our goal, and is not critical to our evaluation. The source code of TurboEPC is available at [40].

Figure 5.6 shows the various components of our implementation. A load generator emulates control and data plane traffic from multiple UEs to the core, a simplified eNB switch implements only the wired interface to the core, and a sink consumes the traffic generated by the load generator. The load generator is a multi-threaded raw-sockets based program of 5.3K lines, that generates EPC signaling messages and TCP data traffic. The load generator can emulate traffic from a configurable number of concurrent UEs. Further, the emulated traffic mix (i.e., the relative proportions of the various signaling and data plane messages) is also configurable.

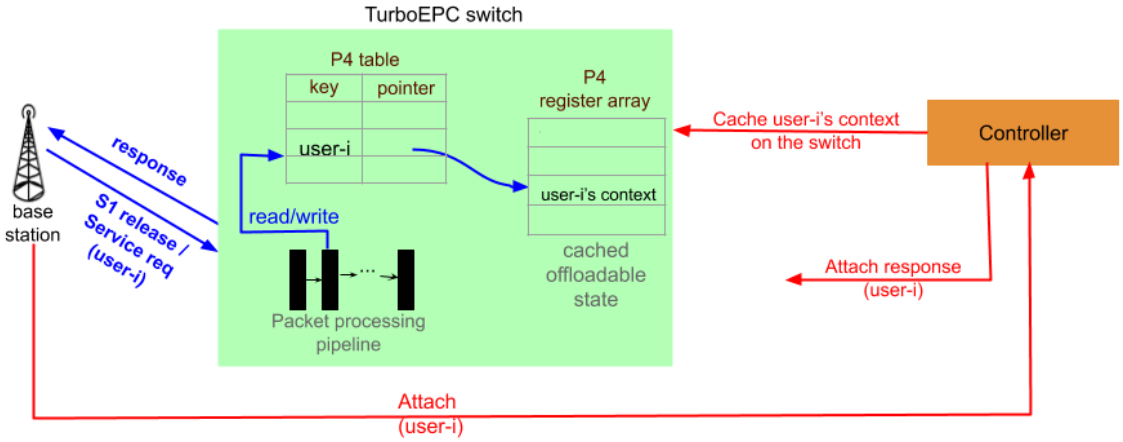


Figure 5.7: Message processing at the TurboEPC hardware switch.

The control plane components of the packet core (MME, SGW-C, PGW-C) are implemented within an SDN controller. The data plane switches (eNB, SGW-D, PGW-D) are implemented as P4-based packet processing pipelines in approximately 3K lines of P4 code. While the data plane performs only GTP-based forwarding in the traditional CUPS-based EPC prototype, it also performs additional processing of offloadable signaling messages (MME' and SGW-C' of Figure 5.1(b)) in TurboEPC. We have compiled our TurboEPC P4 code to run on two targets: the bmv2 simple_switch_grpc [116] software switch target, and the Netronome CX 2x10GbE [117] smartNIC hardware target. We now describe these hardware and software switches.

TurboEPC software switch. In the software switch based TurboEPC prototype, the SDN application that forms the EPC control plane is implemented in the ONOS controller [19] in 10K lines of Java code. The offloadable message processing is implemented within a local ONOS controller that is co-located with the P4-based software data plane switches. This local controller configures and modifies the P4 software switch tables that contain the offloaded state. We use P4Runtime [74] as the communication protocol between the ONOS controller and the P4 software switch. However, the current P4Runtime v1.0.0 does not support multiple controllers (e.g., local and root controllers) configuring the same data plane switch. Therefore, we built custom support for this feature by modifying the *proto/server* package of the *P4Runtime* [74] to send/receive packets to/from multiple controllers. Our initial implementation broadcasted control plane messages to all the controllers, which resulted in unnecessary message processing overhead at the controllers. Therefore, we further modified the P4Runtime agent at the bmv2 switch and the ONOS controller to enable the P4 switch to identify the specific controller where the control packet should be forwarded. This optimization required significant code changes but also improved performance.

TurboEPC hardware switch. Our hardware-based TurboEPC switch did not integrate with the ONOS SDN controller used in the software prototype, due to the limitations of the control to data plane communication mechanisms (P4Runtime support) in the programmable hardware we used. Therefore, we implemented our own channels for the control to data plane communication, but we still could not dynamically install rules on the hardware switch. So, we pre-populated the table rules on the hardware, and the rule population code at the controller generates the rule packets for the switch. When the switch receives these rule packets, it silently discards them.

Another difference with the software switch is in how offloadable messages are processed. Figure 5.7 shows the message processing flow at the TurboEPC hardware switch, and we also describe how it differs from the TurboEPC software switch design. The software prototype stores the offloadable user context and forwarding state generated by the non-offloadable procedures in switch tables, and the local controller is invoked to modify these tables when processing offloadable messages. However, this local controller can consume the limited switch CPU available in hardware switches. Therefore, the hardware prototype stores the offloadable state not in switch tables but in switch register arrays, which are distinct from switch tables. While a switch table can only be modified from the root/local control plane, a register can be modified by P4 code running within the data plane. Therefore, we modified our design so that the switch tables only store a pointer from the user identifier to this register state, and not the actual state itself. The root controller takes care of maintaining the free and used slots in the register arrays of the switches. The root controller creates the table entries that map from user identifiers (which are either available in packet headers, or can be derived from the packet headers) to register array indices when the user context is first created during the attach procedure. After the entries are created, offloadable messages (S1 release, service request) that change the offloaded state do not require to invoke the switch control plane (that consumes switch CPU) to modify the tables. Rather, the offloadable messages can fetch the register index from the table and directly modify the registers from within the data plane.

TurboEPC packet processing pipeline. We now briefly describe the P4-based packet processing pipeline of both hardware and software TurboEPC data plane switches (Figure 5.8). Incoming packets in an EPC switch are first run through a *message redirection table* that matches on various header fields to identify if the incoming message is a signaling message, and if yes, where it should be forwarded to. This table is populated by the root controller to enable correct redirection of non-offloadable signaling messages to the root, and offloadable messages to the switch that has the particular user's context. Packets that

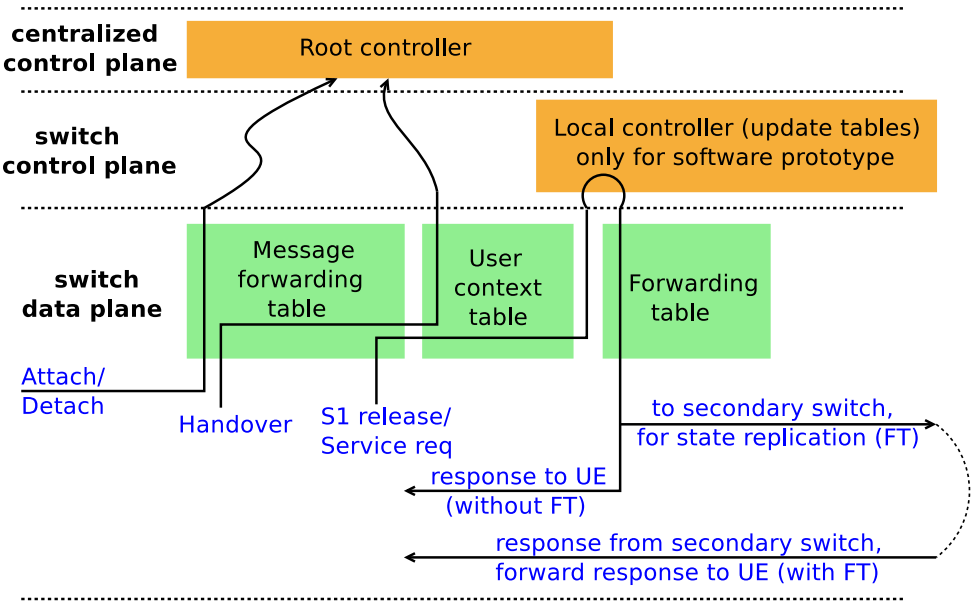


Figure 5.8: Packet processing pipeline in TurboEPC.

do not match the message redirection table continue along the pipeline, and are matched through multiple *GTP forwarding tables* for GTP-based data plane forwarding.

Offloadable signaling messages destined to the current switch are first run through the *user context table* to find any existing offloaded user context. The signaling message is processed by modifying or deleting the user context and/or GTP forwarding state stored on the switch. The switch data structures are either updated by the local controller (software prototype) or within the data plane itself (hardware prototype). After message processing, the packet may be forwarded to the secondary switch for state replication. On successful replication (within the data plane), the secondary switch generates the response packet for the user, and forwards it to the primary switch as an acknowledgement for successful state replication. The primary switch data plane forwards the response packet back to the user, indicating the successful execution of the signaling message. If the signaling message processing could not complete at the switch (e.g., the user context is not found, or the handover message requires further processing at the root), the packet is forwarded to the root controller for further processing. In the case of series design (not last switch), if the user context is not found, the message is forwarded to the next switch on the path.

5.5 Evaluation

We now evaluate the TurboEPC software and hardware switch prototypes, and quantify the performance gains over the traditional CUPS-based EPC.

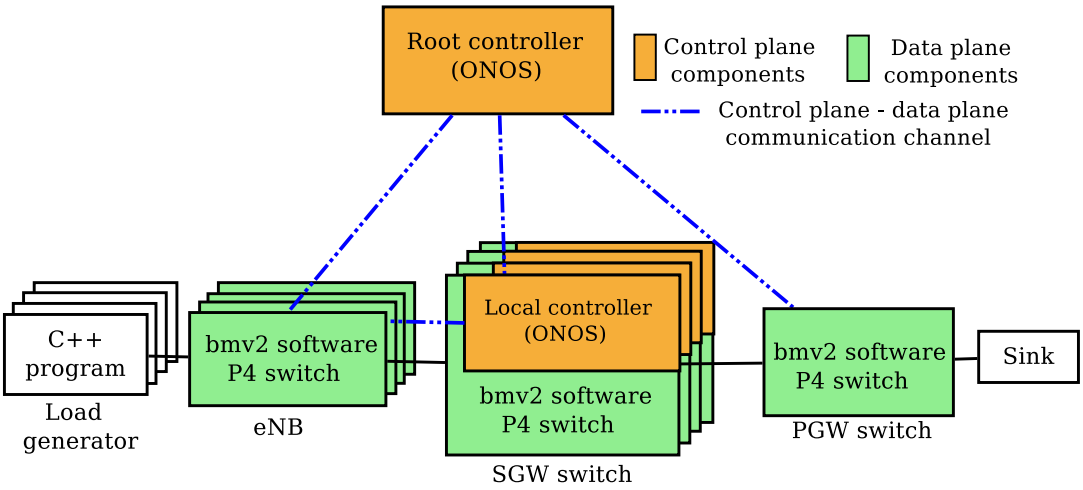


Figure 5.9: TurboEPC software evaluation setup.

5.5.1 TurboEPC software prototype

We first evaluate the TurboEPC prototype implemented on P4-based software switches. We primarily aim to evaluate the benefits of our TurboEPC design as compared to the traditional EPC design. Further, we also seek to demonstrate the correctness and efficacy of the various mechanisms for scalability and fault tolerance in our design.

Setup. Figure 5.9 shows the components in our software TurboEPC setup that include the load generator, a sink node, ONOS v1.13 SDN controller, and multiple P4-based programmable bmv2 software switches (simple_switch_grpc) for the eNB, SGW, and PGW components of LTE EPC. We use multiple "forwarding chains" of load generators and switches in the data plane, to generate enough load to saturate the root SDN controller. All components run on Ubuntu 16.04 hosted over separate LXC containers to ensure isolation. The root controller container is hosted on an Intel Xeon E5-2697@2.6GHz (24GB RAM) server, and the rest are hosted on an Intel Xeon E5-2670@2.3GHz (64GB RAM) server. The root/local controllers and all P4 software switches are allocated 1 CPU core and 4GB RAM each. Our load generator is a closed-loop load generator that emulates multiple concurrent UEs generating signaling and data plane traffic. The number of concurrent emulated UEs in our load generator is tuned to saturate the control plane capacity (root or local or both) of the system in all experiments, and is varied between 4 and 100.

Parameters and metrics. We generate different workload scenarios by varying the mix of offloadable (S1 release and service request) and non-offloadable (attach, detach, and handover) signaling messages in the control plane traffic generated by the load generator. Table 5.3 shows the relative proportions of the various signaling messages in the traffic mixes used, along with a typical traffic mix found in real user traffic [6]. *Off-x* indicates

Traffic Mix	Attach, Detach %	S1 release, Service request %	Handover %
Off-99	1	99	0
Off-95	5	95	0
Off-90	10	90	0
Off-50	50	50	0
HO-5	10	85	5
Typical [6]	1–2	63–94	5

Table 5.3: LTE-EPC traffic mix used for experiments.

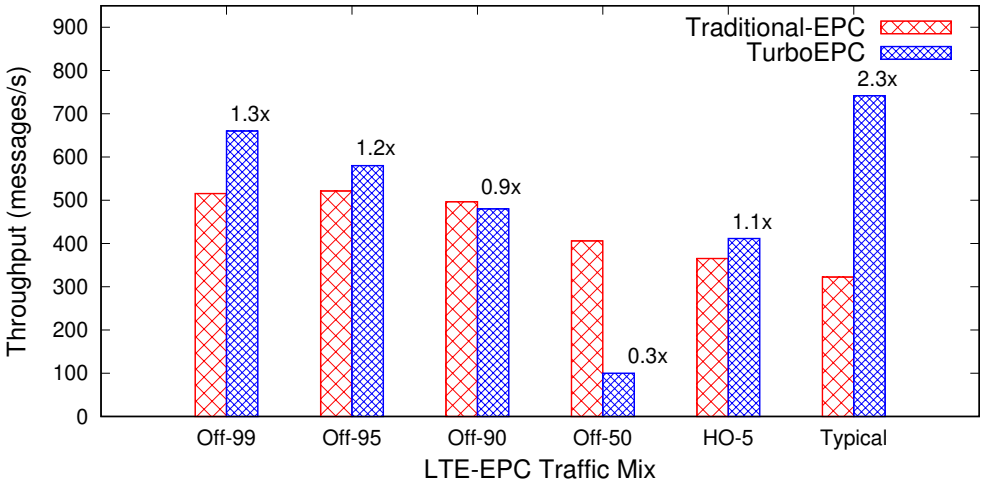


Figure 5.10: TurboEPC vs. traditional EPC: Throughput.

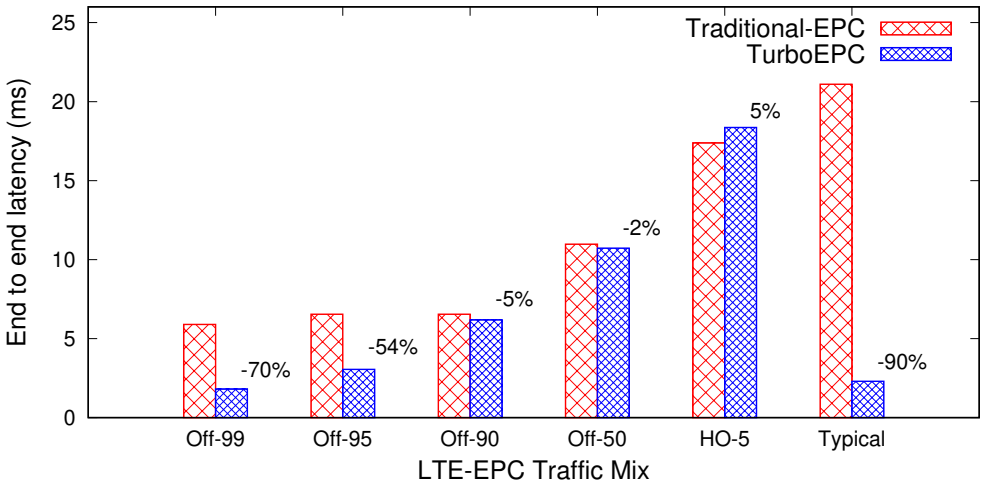


Figure 5.11: TurboEPC vs. traditional EPC: Latency.

that the traffic mix comprises of x% of offloadable messages. All results reported are averaged over three runs of an experiment conducted for 300 seconds, unless mentioned otherwise. The performance metrics measured are the average control plane throughput

(number of control plane messages processed/sec) and average response latency of control plane requests, as measured at the load generator over the duration of the experiment.

TurboEPC vs. Traditional EPC. We first quantify the performance gains of the basic TurboEPC design as compared to the traditional EPC design. In these set of experiments, we assume (and ensure) that all UE context state fits in the memory of a single switch. We also do not perform any replication of the data plane state for fault tolerance because we are interested in measuring maximum control plane capacity; therefore, our load generator does not generate any data plane traffic. Figures 5.10 and 5.11 show the control plane throughput and latency respectively of the traditional EPC and TurboEPC, for various traffic mixes of Table 5.3. As can be seen, performance gains of TurboEPC over traditional EPC are higher for traffic mixes with a greater fraction of offloadable messages. For example, for the typical traffic mix, we observe that TurboEPC improves control plane throughput by 2.3× over traditional EPC, while control plane latency is reduced by 90%. Further, we note that the root controller was fully saturated in the traditional EPC experiments, while CPU utilization was under 20% with TurboEPC because most signaling traffic was processed using data plane switch CPU. However, when the traffic consists of a high proportion of non-offloadable messages (e.g., mix Off-50, which is unrealistic), TurboEPC has lower throughput than traditional EPC (0.3×), because it incurs an additional overhead of pushing user context to the data plane switches during the processing of non-offloadable messages. In summary, we expect TurboEPC to deliver significant performance gains over the traditional EPC over realistic traffic mixes, which contain a high proportion of offloadable signaling messages.

The performance gains of TurboEPC are more pronounced when the distance between the "edge" and "core" of the network increases, and with the increasing number of switches that can process offloadable messages in the data plane, both of which are likely in real-life settings. Figures 5.12 and 5.13 show the performance of TurboEPC as a function of the distance to the root controller (emulated by adding delay to all communications to the root) and the number of forwarding chains of data plane switches. We see from the figures that TurboEPC with 4 chains provides 4× – 5× throughput over traditional EPC. We also observe that TurboEPC latency does not increase with the distance to the core network, and the latency is reduced by two orders of magnitude compared to traditional EPC when the round trip latency to the core is higher than 5 ms.

While TurboEPC improves average control plane performance, it can (and does) degrade performance for some specific non-offloadable messages. For example, as discussed in §5.3.1, processing non-offloadable messages like the attach request incurs the extra cost of pushing offloaded user context to data plane switches. Similarly, handover

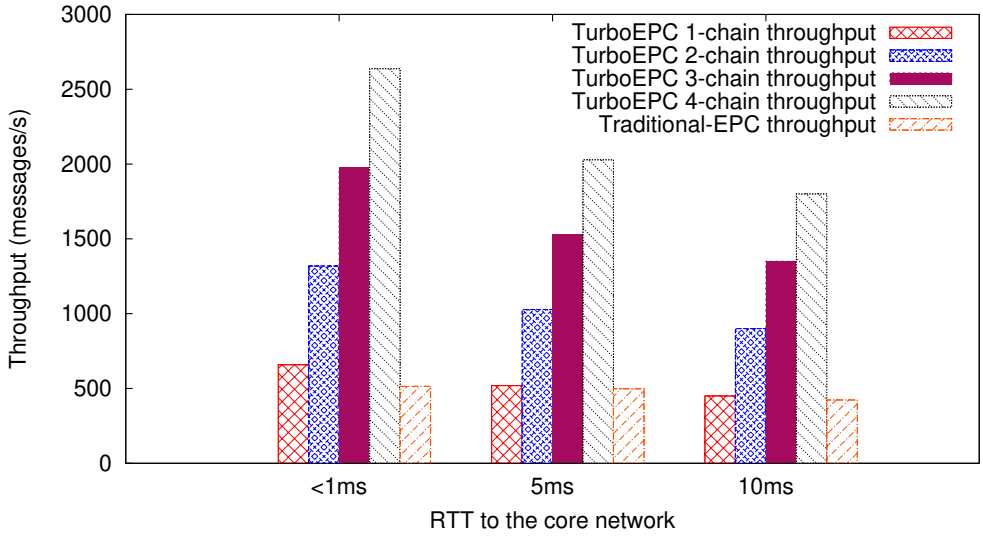


Figure 5.12: Throughput with varying distance to core, and varying number of dataplane switches.

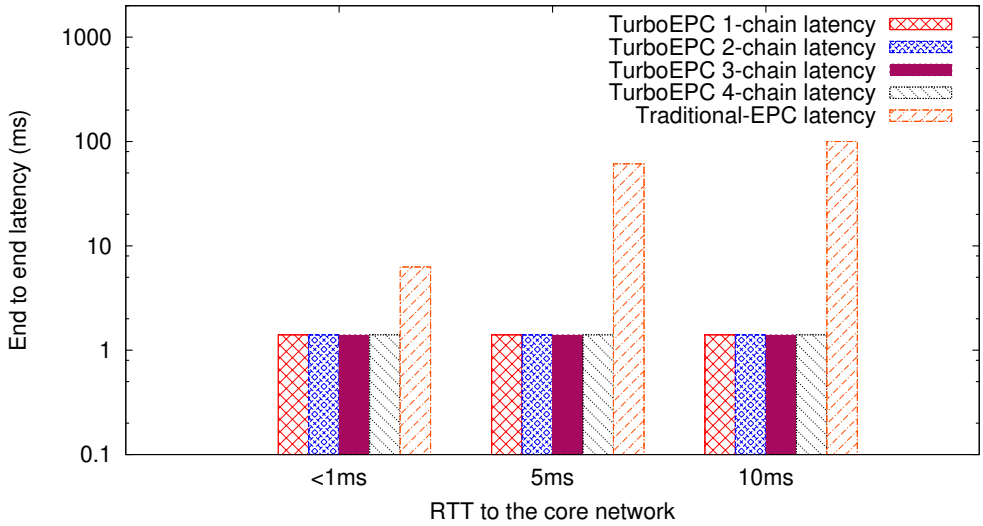


Figure 5.13: Latency with varying distance to core, and varying number of dataplane switches.

message processing incurs a higher overhead with TurboEPC because we need to piggyback the offloaded state and synchronize it with the root. Table 5.4 shows the average processing latency of various individual signaling messages in TurboEPC and the traditional EPC, in the setup with a single forwarding chain. The generated load followed the typical traffic distribution, as shown in Table 5.3. Table 5.4 shows the latency results for two scenarios: (i) when the EPC core is close to the edge ($RTT < 1ms$), and (ii) when the EPC core is far from the edge ($RTT = 10ms$). We see that the processing latency reduces by up to 86–94% for offloadable messages like S1 release and service request, but increases by 2–5% for non-offloadable messages like attach requests and handovers.

Design	Attach, Detach	S1 release, Service request	Handover
RTT to the core is less than 1 ms			
Centralized	10.72	10.28	17.38
TurboEPC	10.98	1.44	18.36
RTT to the core is 10 ms			
Centralized	200	38	549
TurboEPC	205	2.4	580

Table 5.4: Average end-to-end latency for typical LTE-EPC traffic distribution (in *ms*).

Because offloadable messages form a significant fraction of signaling traffic, TurboEPC improves the overall control plane performance of the mobile packet core, even though a small fraction of signaling messages may see a slightly degraded performance.

Series vs. parallel partitioning. Next, we perform experiments with the series vs. parallel state partitioning design variants of the TurboEPC software switch prototypes, to evaluate the performance impact of the additional complexity of these designs. This experiment was performed with traffic mix *Off-99* of Table 5.3 (1% attach-detach requests), and results for other traffic mixes were similar. We use multiple (up to 3) TurboEPC switches in series and parallel configurations, and partition 100 active users uniformly over these switches. Besides these 100 users, our load generator also generates traffic on behalf of an additional 20 users whose contexts were not stored in the data plane switches, to emulate the scenario where all contexts cannot be accommodated in the data plane. Figure 5.14 shows the average control plane throughput and latency of the TurboEPC-Series(n) and TurboEPC-Parallel(n) designs, for a varying number of switches n in series and parallel, both when the context of the users is found within one of the switches (hit) and when it is not (miss). We see from the figure that the TurboEPC throughput scales well when an additional switch becomes available to process offloadable signaling messages. The scaling is imperfect when there are 3 switches in series or parallel because the eNB switch became the bottleneck in these scenarios. This eNB bottleneck is more pronounced in the parallel design case because the eNB does extra work to lookup the switch that has the user's context in the parallel design. We hope to tune the eNB software switch to ameliorate this bottleneck in the future.

While the throughput increases with extra TurboEPC switches, the control plane latency also increases due to extra hop traversals and extra table lookups compared to the basic TurboEPC design. This impact on latency is more pronounced in the series

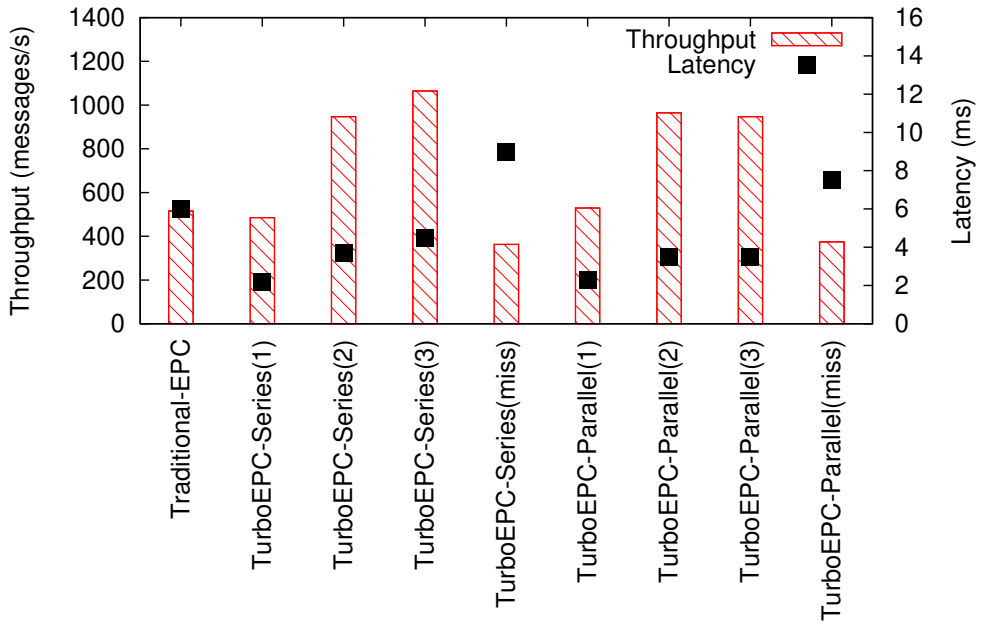


Figure 5.14: Series vs. parallel partitioning.

designs, where each switch adds an extra hop to latency. However, even with 3 switches in series or parallel, TurboEPC latency is still lower than that of the traditional EPC. We also see from the figure that the miss latency of offloadable message processing is worse than the message processing latency of the traditional EPC, because the messages undergo multiple table lookups within the data plane before eventually ending up at the root controller.

TurboEPC fault tolerance. Next, we evaluate the fault tolerance of the TurboEPC design by simulating a failure of the primary switch in the middle of an experiment and observing the recovery. Figure 5.15 shows the average throughput and Figure 5.16 shows the average latency of the fault-tolerant TurboEPC for an experiment of duration 1200 seconds, where the primary switch was triggered to fail after 600 seconds. Also shown in the graphs are the throughput and latency values of the basic TurboEPC without any fault tolerance, for reference. We see that the throughput of the basic TurboEPC is 40% higher, and the latency is 33% lower than the fault-tolerant design due to the lack of replication overhead. After the failure of the primary switch, we found that the root controller takes about 15 seconds to detect the primary switch failure, ~2 ms to push rules to eNB that would route incoming packets to the secondary switch, and ~30 ms to restart offloadable signaling message processing at the secondary switch. During this recovery period, we observed ~200 signaling message retransmissions, but all signaling messages were eventually correctly handled by TurboEPC after the failure.

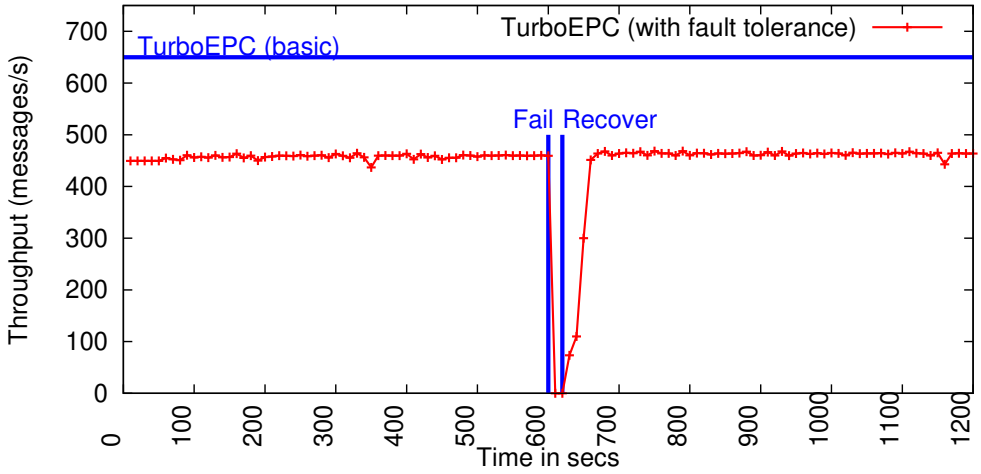


Figure 5.15: TurboEPC throughput during failover.

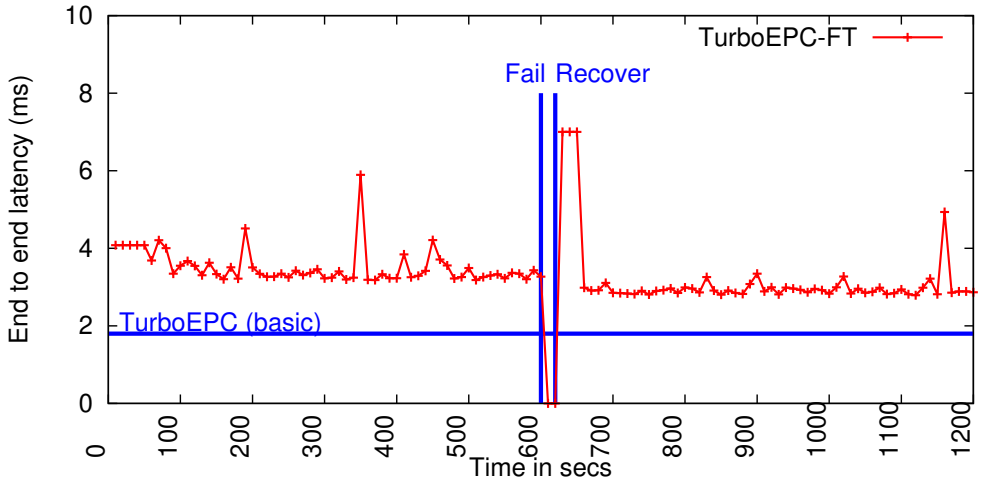


Figure 5.16: TurboEPC latency during failover.

5.5.2 TurboEPC hardware prototype

We now evaluate our hardware-based TurboEPC prototype, built using the P4-programmable Netronome Agilio smartNIC [117].

Setup. Figure 5.17 shows the TurboEPC hardware setup which was hosted on three Intel Xeon E5-2670@2.3GHz (128GB RAM) servers, each connected to one Netronome Agilio CX 2x10GbE smartNIC. The three servers hosted the single chain of the load generator+eNB, SGW, and PGW+sink, respectively. An ONOS controller is assigned 4 CPU cores and is hosted on the SGW switch and served as the root control plane, for TurboEPC hardware switch setup as well as the traditional CUPS-based EPC setup.

Parameters and Metrics. Our load generator generated a mix of offloadable/non-offloadable signaling messages and data plane traffic (using iperf3) in the experiments. The smartNIC hardware could accommodate the user contexts of 65K users within the

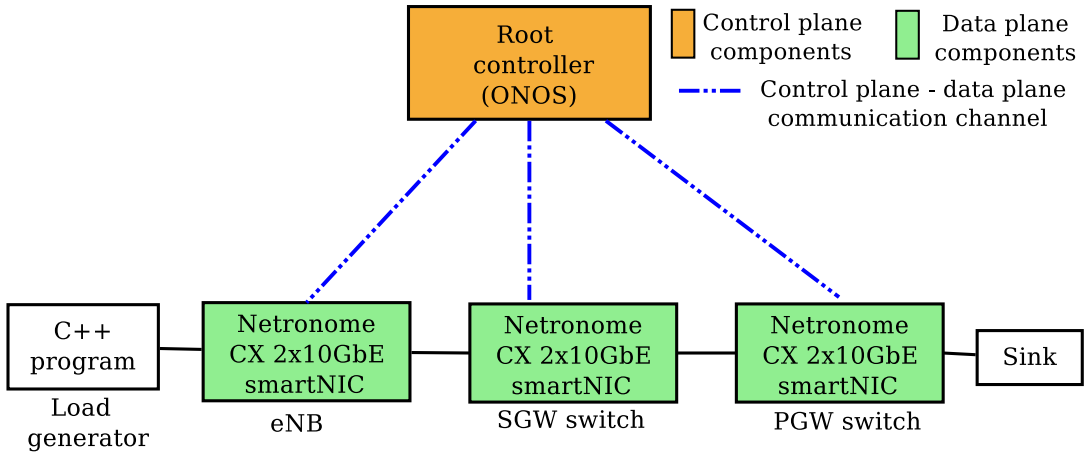


Figure 5.17: TurboEPC hardware evaluation setup.

switch hardware tables, and the load generator could generate traffic for these users in all experiments. The maximum forwarding capacity of our smartNICs (without any TurboEPC changes) was measured at 8 Gbps, so our load generator also limited its maximum data plane traffic rate to 8 Gbps. All experiments were run for 300 seconds, and we report the maximum throughput and latency of processing offloadable signaling messages in the hardware prototype.

Performance of TurboEPC hardware switch vs. traditional CUPS-based EPC. First, we measure the performance of our hardware TurboEPC switch, without any interfering data plane traffic, and compare it with that of the traditional CUPS-based EPC setup. We evaluate the saturation throughput and response latency with the smartNIC loaded with the state for 65K users. Figure 5.18 and Figure 5.19 compare the performance of the TurboEPC hardware switch and traditional CUPS-based EPC in terms of throughput and response latency, respectively. The errorbars in the latency plot shows the minimum and maximum latency values. We observe that when the offloadable traffic rate is high (*Off-99*), the hardware-based TurboEPC throughput is 11× higher, and the average latency is 97% lower than the traditional EPC. The traditional EPC root CPU is saturated (~400%, which refers to all the 4 CPU cores running at 100%), but the TurboEPC root CPU utilization for this traffic is only 45%, and the local TurboEPC switch is saturated. The TurboEPC root controller does not saturate because the amount of non-offloadable traffic that is served by the root controller is very low. In order to obtain TurboEPC saturated root controller throughput, we need to add more hardware TurboEPC chains that could pump more non-offloadable traffic. With additional hardware chains at saturated root CPU (400%), the TurboEPC throughput improvements would be more significant. However, for the *Off-20* traffic-mix, we observe TurboEPC throughput to be 1.4× higher and average latency 68% lower than traditional EPC. The TurboEPC performance gains

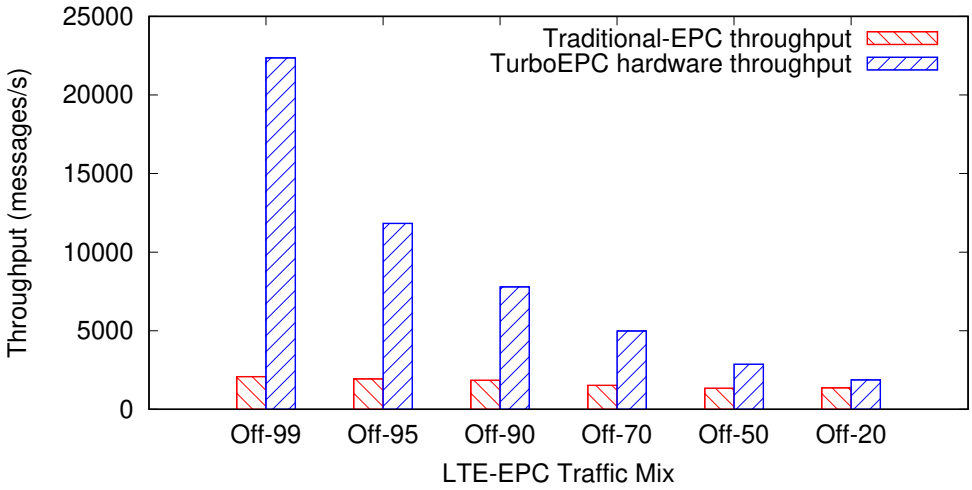


Figure 5.18: TurboEPC-hardware vs. traditional-EPC throughput.

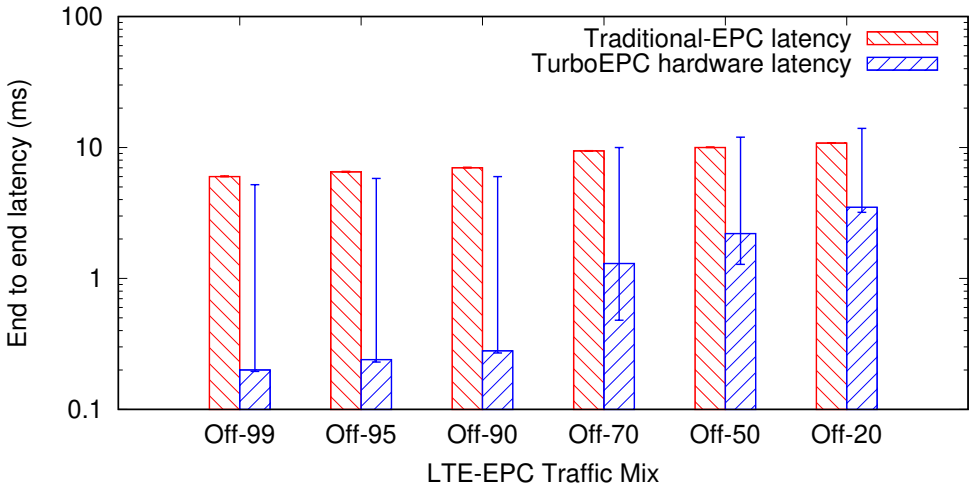


Figure 5.19: TurboEPC-hardware vs. traditional-EPC response latency.

decrease because as the non-offloadable traffic rate increases, the traffic processed at the hardware switch reduces, and the root controller saturates. We observe high tail latencies for TurboEPC due to the processing of non-offloadable traffic at the root controller.

Capacity of TurboEPC hardware switch. Now, we measure the maximum control plane capacity of our hardware TurboEPC switch such that the switch only processes offloadable EPC control messages. In this setup, the switch resource was not shared for data plane traffic processing or non-offloadable message processing. For this purpose, the load generator offloaded the user state to the switch (attach request), after which it only performed the offloadable operations of the S1 release and service request. We also tested the effect of varying user-state size on the switch performance. Figure 5.20 shows the throughput and latency of a single TurboEPC hardware switch. We evaluate the maximum throughput with the smartNIC loaded with user state size varying from 100 to 65K.

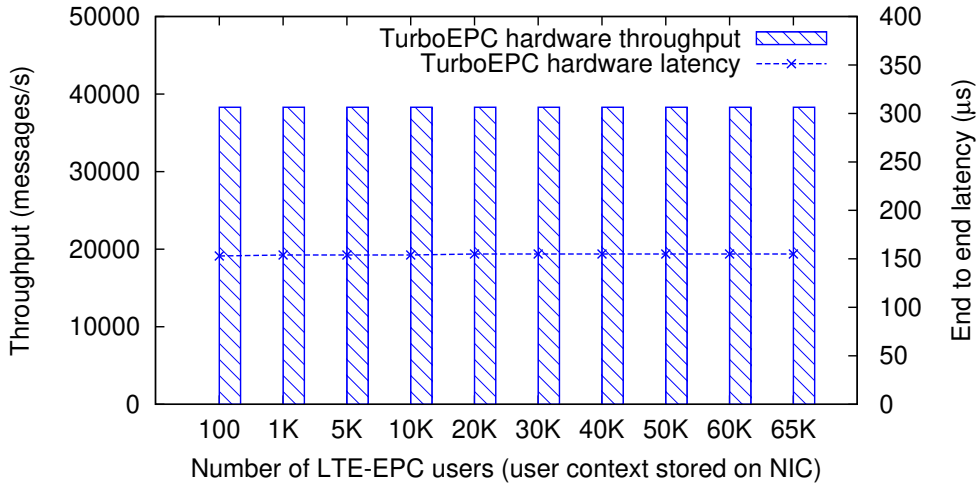


Figure 5.20: TurboEPC throughput vs. number of users

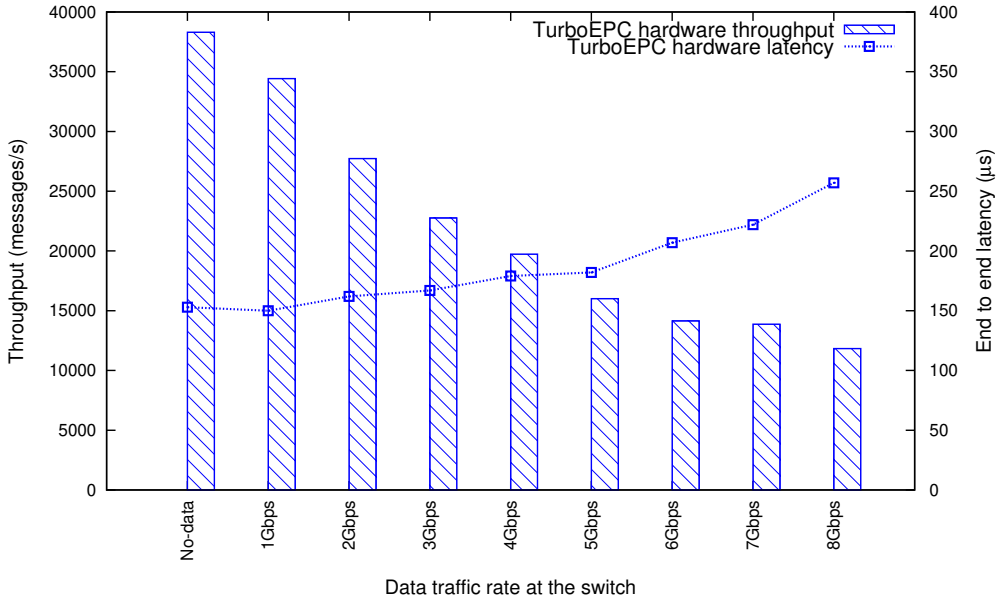


Figure 5.21: TurboEPC throughput with data traffic interference.

We found that the throughput does not vary when we add the state of more users to the smartNIC. We see from the figure that our single TurboEPC hardware switch can successfully serve up to 38K EPC messages per sec, while providing 20× higher throughput and 97% lower latency than traditional EPC.

Performance with dataplane traffic. TurboEPC improves control plane throughput over the traditional EPC by leveraging the extra capacity at data plane switches for offloadable signaling message processing. However, the performance gains of TurboEPC may be lower if the switch is busy forwarding data plane traffic. We now measure the impact of this data plane cross-traffic on the control plane throughput of TurboEPC. We pump increasing amounts of data plane traffic through our TurboEPC hardware switch (with

the state for 65K users) and measure the maximum rate at which the switch can process offloadable signaling messages while forwarding data traffic simultaneously. Figure 5.21 show the signaling message throughput and latency, respectively, as a function of the data plane traffic forwarded by the TurboEPC hardware data plane switch. We see from the figure that as the data traffic rate increases, the offloadable signaling message throughput decreases, and response latency varies between 150μs to 250μs. The throughput and latency values for the traditional CUPS-based EPC (RTT to the root < 1 ms) are same as in the previous experiment (refer Figure 5.18). We observe that when the switch is idle, the hardware-based TurboEPC throughput is 20× higher, and the latency is 97% lower than the traditional EPC. However, even when the switch is forwarding data at line-rate (8Gbps), we observe throughput to be 10× higher and latency 96% lower than traditional EPC, confirming our intuition that spare switch CPU can be used for handling offloaded signaling traffic.

5.5.3 Summary of results

Our evaluations demonstrated that offloading signaling messages to the data plane improves the throughput by 1.4× to 20× and reduces the latency by 68% to 97% than traditional EPC, with the increase in the proportion of offloadable control plane messages. The performance range depicts the TurboEPC efficacy with the increase in the proportion of offloadable control plane messages. Even when the switch is forwarding data at line-rate, the TurboEPC data plane improves the control plane throughput of offloadable messages by 10× and reduces the latency by 96%.

5.6 Summary

We described TurboEPC, a redesigned mobile packet core that offloads a significant fraction of signaling procedures from the control plane to the programmable data plane to improve control plane performance. TurboEPC data plane switches store a small amount of control plane state in switch tables, and use this state to process some of the more frequent signaling messages at switches closer to the edge. We implemented TurboEPC on P4-based software switches and programmable hardware. Our TurboEPC code is open-sourced and available for use for the developers [40].

Chapter 6

Comparison of Control Plane Scaling Approaches

In this chapter, we use empirical results to compare the performance of SDN control plane scalability approaches proposed by existing research as well as the methods proposed in this thesis. We use the common testbed, framework, and common SDN application (4G LTE mobile packet core) to compare these scalability designs. We chose the mobile packet core application because it is complex enough to cover all the patterns that are found in other applications too. We conclude the chapter with the performance summary of all scalability designs and provide guidelines on the scalability design choice based on the application and traffic characteristics.

6.1 SDN control plane scaling approaches

The goal of this thesis is to ensure scalability and low response latency for SDN control plane applications. The term *scalability* here implies that the application throughput scales with the addition of resources to the SDN control plane. *Response latency* is the latency between the user initiating the request and receiving the response. We have already discussed the existing SDN control plane scalability approaches in §2.2. We have contributed two new approaches, Cuttlefish (Chapter 4) and TurboEPC (Chapter 5), towards a scalable SDN control plane. Next, we briefly describe all control plane scalability approaches that we compare (see Figure 6.1).

1. **Centralized SDN controller design.** The traditional SDN model runs the SDN controller over a commodity server (Figure 6.1 (a)). POX [125] is an example of a single threaded SDN controller. Beacon [43], Floodlight [44], NOX [45], Maestro [46] are some of the popular multithreaded SDN controllers. The centralized

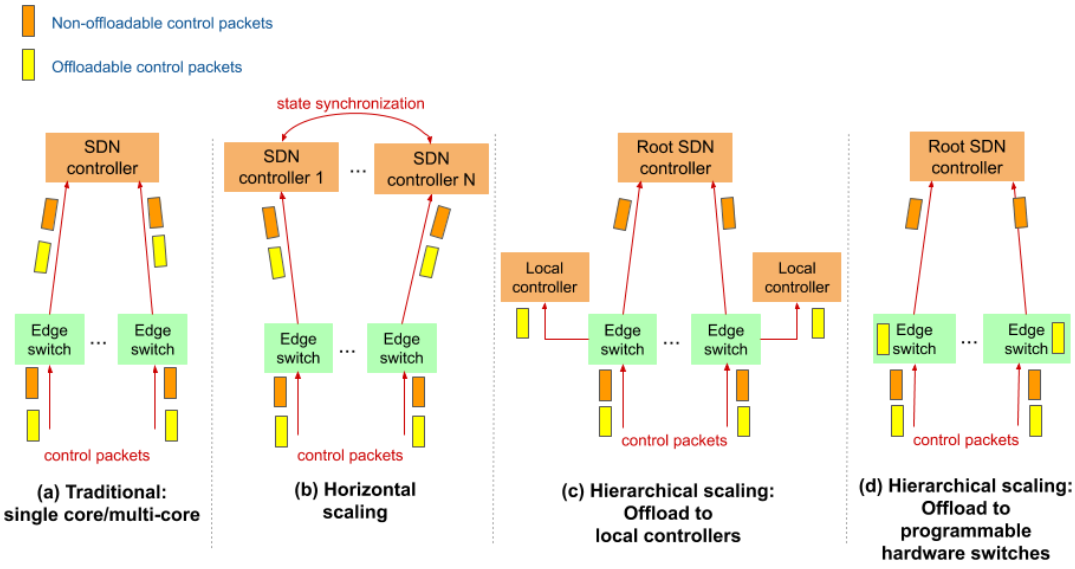


Figure 6.1: SDN control plane scalability approaches.

SDN controller processes all the control plane requests that arrive at the data center network. We have already seen in §2.1 that the centralized SDN controller can become a bottleneck with high frequency control plane traffic. In our experiments, we evaluate two centralized controller configurations, single-core and multi-core, where the number of cores assigned to the centralized root controller is the configuration parameter. The multi-core configuration scales the SDN controller but, there is a limit on the number of cores a dedicated server can have. Also, the cost of a single dedicated server with $M * N$ cores is higher than that of M servers with N cores each [126]. So, it is better to split the load amongst multiple server replicas, as adopted by the horizontally-scaled controller design.

2. **Horizontally-scaled SDN controller design.** The horizontally-scaled controller design distributes the incoming control plane traffic among multiple homogeneous controller replicas (Figure 6.1 (b)). But, the SDN paradigm requires maintenance of a consistent network-wide view at the SDN controller. Therefore, the controller replicas in the horizontally-scaled design use synchronization mechanisms to maintain a consistent global network-wide view.

   Proposals like Onix [13], Hyperflow [14], and Beehive [20] implement this design. For example, Hyperflow divides the network topology into subsets, and the designated controller replica processes the control traffic at each subset. These controller replicas implement publish-subscribe state synchronization mechanisms to maintain the global network-wide view. We can scale the SDN control plane by the

addition of controller replicas, but the state synchronization process will use some CPU cycles. Therefore, the application scales but the throughput is slightly lower than the expected linear scaling. The response latency slightly increases due to state synchronization, as compared to traditional centralized technique.

3. **Offload computations to local controllers.** The SDN controllers discussed in the previous approaches can be physically distant from the users and cause response time latencies in the order of a few 10's to 100's of milliseconds. A hierarchical control plane scaling approach is used to scale the centralized root controller and reduce the response time latency. This approach offloads a subset of application computations to the local controllers at/close to the edge switches, typically close to the user (Figure 6.1 (c)). The rest of the computations are processed at the centralized root controller.

   Devoflow [15], Difane [21], and Kandoo [16] are examples of hierarchical scaling approaches. These approaches identify the computations that require local switch-specific states alone and offload their processing to the local controllers. We have seen that our hierarchical control plane scaling framework, Cuttlefish, extends the existing approaches by offloading additional computations that depend on the offloadable global state (§3.1.2) to the local controllers at the edge switches, close to the user. It involves lazy synchronization of the offloadable global state since some control plane messages may access this offloadable state at the root controller.

   The offload of the subset of computations significantly reduces the response latency for the offloaded control plane messages. The response latency slightly increases for non-offloadable messages because the message processing can create/update/delete the offloadable state, that must be synchronized consistently with the copy of the state at the local controllers. This approach performs worse than the centralized approaches when the state synchronization cost negates the benefits of offload. Therefore, our Cuttlefish framework monitors the state synchronization costs in real-time and adapts to the best SDN mode, centralized or offload.

4. **Offload computations to programmable hardware switches.** The hierarchical scaling approach benefits significantly by offloading the processing of offloadable messages to the programmable hardware switches or smartNICs since the packets do not have to travel the network and application stack of the controller and undoubtedly hardware runs faster than the software (Figure 6.1 (d)).

   Eden [25] and FOCUS [17] are examples of solutions that offload subset of computations to the hardware switches. Our proposal, TurboEPC, offloads the *S1 release*

Approach	Scalable?	Response latency	Bottleneck reason	Suitable workload
Centralized: multi-core	Yes, limited to number of controller cores	Depends on RTT between user and root controller	Root controller CPU saturation	Workload with low frequency control traffic
Horizontal scaling	Yes	Depends on RTT between user and root controller & state synchronization overhead	CPU saturation of all controller instances & state synchronization overhead	Workload that generates low synchronization traffic
Offload to local controllers	Yes, scalable for offloadable traffic	Depends on RTT between user and local controller (for offloadable traffic)	High offloadable state synchronization cost (put_rate>access_rate)	Workload with higher fraction of offloadable traffic
Offload to programmable hardware switches	Yes, scalable for offloadable traffic	Depends on RTT between user and hardware switch (for offloadable traffic)	High offloadable state synchronization cost (put_rate>access_rate), switch memory, & spare switch CPU	Workload with higher fraction of offloadable traffic & the offloadable computations should be implementable on programmable hardware

Table 6.1: Comparison of SDN control plane scaling approaches.

and the *service request* offloadable control plane messages to the programmable hardware (smartNIC) that resides close to the base station. We achieve significant throughput improvement as these hardware devices run at line-rate. Also, the response latencies are low (order of 100's of μsec) as the control plane messages are processed at the data plane itself. Apart from the synchronization cost limitation, there are additional programmable hardware limitations like small hardware instruction set and memory size that we have discussed in §5.2.

6.1.1 Comparison of control plane scaling approaches

Table 6.1 summarizes the key points of the SDN control plane scaling approaches. Both the horizontal scaling and hierarchical scaling approaches help scale the SDN control plane. The horizontal scaling approach must implement strict state synchronization between the controller replicas to maintain a logically centralized network view. In contrast, the offload techniques scale better because they employ lazy synchronization for the offloadable state. The response latency provided by the offload techniques is very low as compared to horizontal scaling techniques since the offloadable messages are processed at the edge, close to the user. In this chapter, the local controller offload design does not implement the Cuttlefish idea of adaptive switching between the offload and centralized SDN modes, since we are interested in identifying the individual design (centralized, horizontal, or offload) that provides the best performance.

Offload to programmable hardware provides significant throughput and latency gains. To implement this design, we incur additional costs for replacing the edge switches

by the programmable hardware. The price of a quad-core Intel Xeon processor is 422 USD [127], and the value of the Netronome 10Gbps smartNIC is 444 USD [128]. The cost for the two is similar, but the performance gains of smartNIC-based offload are significant, so it is preferred to use the hardware offload approach, if the offloadable computations are programmable with the instruction set of the underlying programmable hardware.

The horizontal scaling approach can scale any control plane traffic, whereas the proposed offload techniques can only scale offloadable control traffic. The offload techniques have to consistently update the copy of the offloadable state at the local controller/switch whenever it is updated at the root controller by the non-offloadable messages. Therefore, the response latency for non-offloadable messages increases and the performance of the offload design degrades with the increase in the proportion of non-offloadable messages in the total traffic. So, it is better to implement a framework like Cuttlefish, that adapts to the best approach based on incoming control traffic-mix. In the next section, we discuss the empirical evaluation of SDN control plane scalability designs to validate our hypothesis.

In this section, we describe the implementation, experimental testbed setup, and evaluation of all the SDN control plane scaling designs.

6.2 Implementation

Our testbed uses the ONOS controller framework and data plane switches (P4-based bmv2 (simple_switch_grpc [116]) switch for all software setups, Netronome Agilio CX 4000 smartNIC [117] for hardware setup. We run the same SDN application (4G LTE mobile packet core) for all the designs to have a fair comparison. Figure 6.2 shows the experimental setup that is common to all the scaling designs. The mobile packet core application components eNB, SGW, and PGW are the data plane switches controlled by the root ONOS controller, i.e., the root controller can configure the switches and also populate switch tables. Each switch is assigned 2 CPU cores, and the ONOS root controller is assigned 4 CPU cores unless specified otherwise. In the case of all software setups, the P4-based bmv2 switches used the P4Runtime API for controller-switch communication, i.e., to populate switch rules and send control/data traffic.

We know that our offload designs distribute the computation processing between the root controller and the local nodes (controllers/switches) such that the root controller processes the non-offloadable computations, and the local nodes process the offloadable computations. If the traffic-mix consists of a small fraction of non-offloadable messages, the root controller load is very low. Therefore, we may be unable to saturate the offload design's root controller with a single forwarding chain (load generator + eNB + SGW).

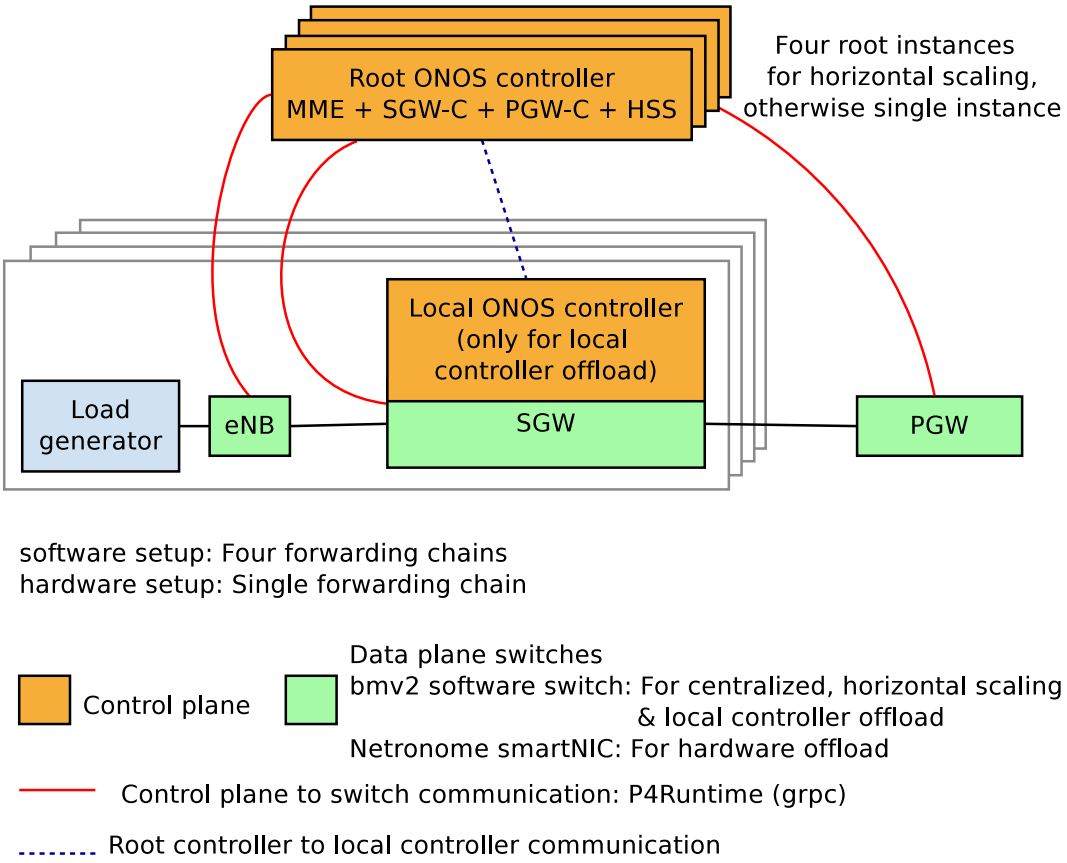


Figure 6.2: Experimental setup diagram for all scaling designs.

Similar to what we have discussed in §4.4.1 and §5.5.1, we set up four forwarding chains with the intention to saturate the root controller. We set up a single forwarding chain for the hardware offload design since only three smartNICs were available with us. We limited our setup with four forwarding chains because the CPU and the memory at the servers that hosted our switch containers were exhausted. Despite four forwarding chains, there were cases when we could not saturate the root controller CPU due to low fractions of non-offloadable traffic, and we have highlighted such results.

Some of the discussed scalability designs comprise of multiple controller nodes. For example, the horizontal scaling design comprises multiple homogeneous root controllers, whereas the local controller offload design comprises multiple local controllers and the root controller. Therefore, our framework should have the capability to allow multiple controllers to configure/populate the switch tables simultaneously. The current P4Runtime implementation does not have this support. Therefore, we modified the existing P4Runtime code at the bmv2 switches and the ONOS controllers to enable multi-master support for all designs comprising multiple controller nodes. We have discussed more details in §5.4. Next, we describe the specific implementation details for each scalability design.

1. **Centralized single-core and centralized multi-core design.** The centralized design is the traditional SDN controller design. We use the TurboEPC's centralized design implementation for LTE-EPC application (refer §5.4) since it uses the same framework, i.e., the ONOS SDN root controller and P4-based bmv2 software switches. We evaluate two centralized design configurations, single-core, and multi-core, where the purpose of the multi-core configuration is to demonstrate vertical scaling of the root controller. We allocate 1 CPU core and 4 CPU cores to the root controller for the single-core and multi-core configurations, respectively.

2. **Horizontal scaling.** We implemented this design since we do not have an existing LTE-EPC implementation that scales using horizontal scaling design. Our root controller runs the modified version of the centralized controller code mentioned in §5.5. We have implemented four root ONOS controllers that run the EPC application, and each of them is assigned 1 CPU core and one forwarding chain. The root controller with 4-core capacity can have a fair comparison with the centralized multi-core design that is assigned 4-cores.

   In horizontal scaling, controllers use synchronization mechanisms to maintain the logically centralized view. We implement the synchronization mechanism similar to Hyperflow [14]. All the root controllers update the state at the global Redis key-value server (publish). All the updates at the global Redis store are synchronized immediately with the local Redis store at the root controllers (subscribe). Since all the root controller replicas have all user state, any root replica can service the control plane message of any user. We modified the centralized code for all EPC control plane messages to access the state from the Redis datastore, and we require all the root controller replicas to have control over all the switches.

3. **Local controller offload.** The proposed local controller offload design, Cuttlefish was implemented over the Floodlight SDN controller (§4.4). This implementation did not implement the standard EPC security algorithms for encryption and authentication, as Cuttlefish's goal was to demonstrate adaptive offload and not build standards-compliant EPC application. Since all other designs have the standard implementation for EPC security algorithms, we implemented the local controller offload prototype using TurboEPC's base code (§5.4).

   We implemented four forwarding chains with four local ONOS controllers, each with 2 CPU cores. The local ONOS controller that resides at the SGW processes the offloadable control traffic that arrives at the corresponding forwarding chain. In contrast, the incoming non-offloadable traffic from all the forwarding chains is pro-

cessed by the root controller. We implemented synchronization channels between the root and local ONOS controllers for synchronization of offloadable state. We could not use the ONOS built-in controller-to-controller communication channels because it interfered with our P4Runtime implementation.

4. **Offload to programmable hardware.** In this chapter, the hardware offload design evaluation refers to the LTE-EPC application implementation and results presented in §5.5.2.

6.3 Experimental setup

The components in our evaluation setup (see Figure 6.2) include the load generator, ONOS v1.13 SDN controller (root as well as local), multiple switches for the eNB, SGW, and PGW components of LTE EPC. Our load generator is a closed-loop load generator that emulates multiple concurrent UEs generating signaling and data plane traffic.

Setup specific to centralized, horizontal scaling, and local controller offload design. The switches for the centralized, the horizontal scaling, and the local controller offload design are the P4-based bmv2 software switches. We have described the resource allocations in §6.2. All components run on Ubuntu 16.04 hosted over separate LXC containers to ensure isolation. The root controller container was hosted on an Intel Xeon E5-2697@2.6GHz (24GB RAM) server, and the rest were hosted on an Intel Xeon E5-2670@2.3GHz (64GB RAM) server. All the containers were allocated 4GB RAM each.

Setup specific to hardware offload design. The hardware offload setup was hosted on three Intel Xeon E5-2670@2.3GHz (128GB RAM) servers with Ubuntu 18.04, each connected to one Netronome Agilio CX 2x10GbE smartNIC [117]. The three servers hosted the single chain of the load generator+eNB, SGW, and PGW+sink, respectively. The root ONOS controller was hosted on the SGW switch. We have described the resource allocations in §6.2.

Parameters and metrics. We generate different workload scenarios by varying the mix of offloadable (S1 release and service request) and non-offloadable (attach and detach) signaling messages in the control plane traffic generated by the load generator. All results reported are averaged over three runs of an experiment conducted for 300 seconds, unless mentioned otherwise. The performance metrics measured are the average control plane throughput (number of control plane messages processed/sec) and average response latency of control plane requests, as measured at the load generator over the duration of the experiment.

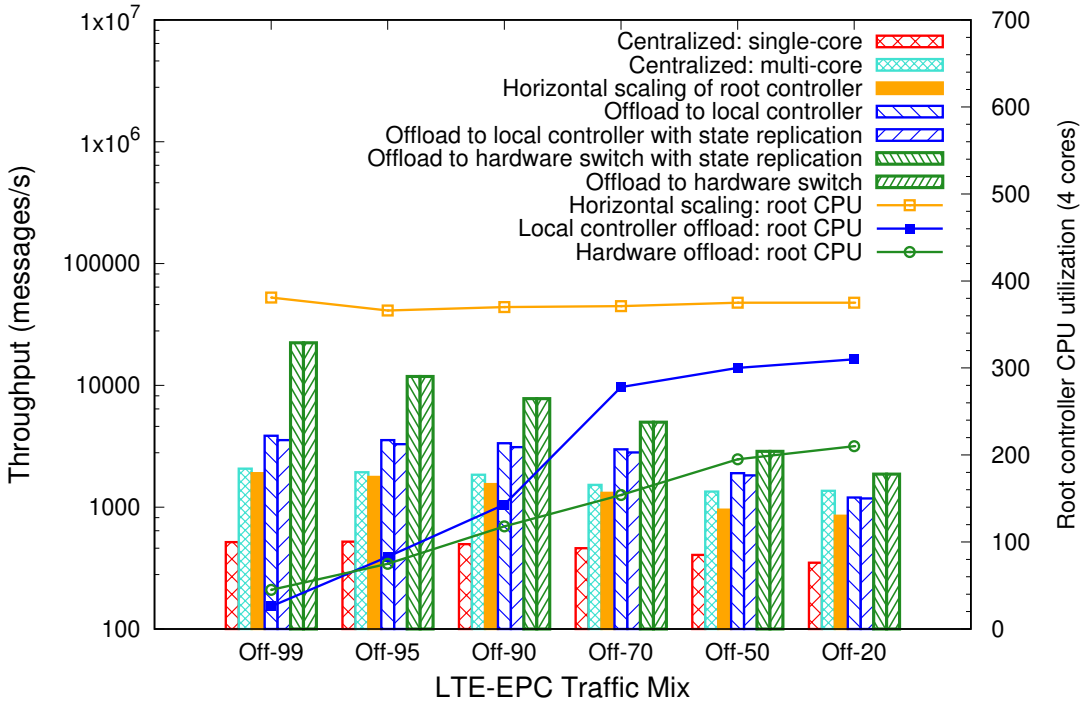


Figure 6.3: Throughput for SDN-based EPC application.

6.4 Evaluation

Our evaluation aims to answer the following questions:

1. Which design performs better with respect to the application throughput and response latency?

2. What is the impact of the distance of the root controller from the end-user on the application performance?

6.4.1 Performance comparison of scaling designs

Figure 6.3 compares the throughput for all the SDN control plane scaling designs. As the hardware offload throughput is very high, it dominates the plot (y-axis plotted using logscale). Therefore, we use Figure 6.4 to show the throughput without the hardware offload results so that the difference between the rest of the results is clear. The figures also show the root controller CPU utilization on y2-axis. The x-axis shows the traffic-mix, where the traffic-mix, *Off-x*, means that the fraction of offloadable traffic is x%, and the fraction of non-offloadable traffic is (100-x)%.

We know that horizontal scaling design inherently replicates state to maintain a consistent network-wide view; therefore, this design is fault-tolerant by default. To implement a fault-tolerant solution for the proposed offload designs, we replicate the offloaded

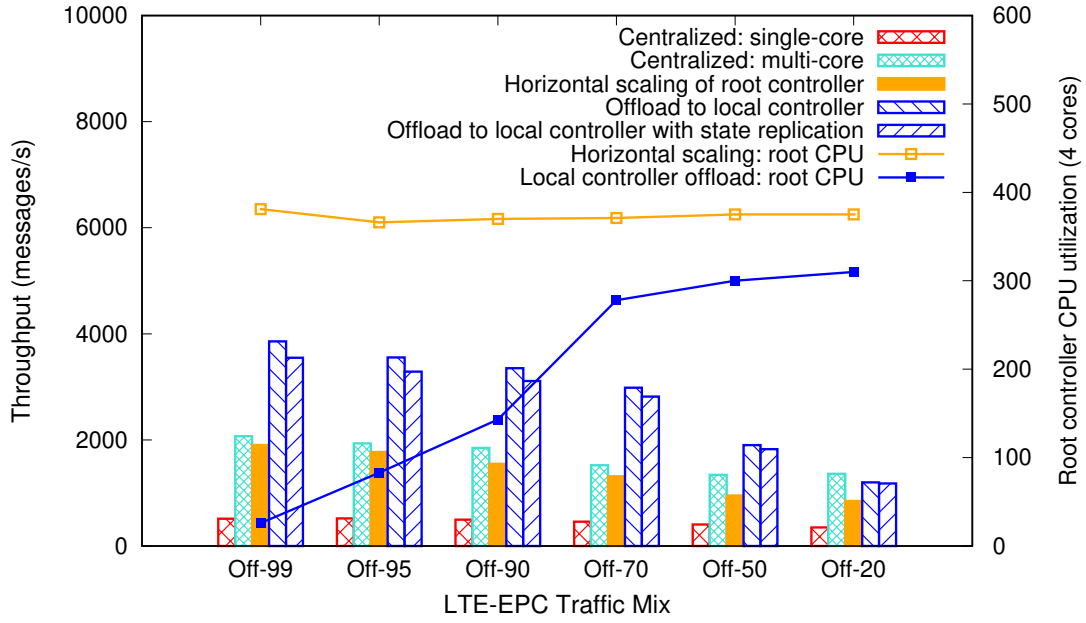


Figure 6.4: Throughput for SDN-based EPC application without hardware offload results.

state at the local controllers and programmable switches. The experiment results compare the performance of fault-tolerant offload techniques (i.e., with replication of offloadable state at the local nodes) with the performance of horizontal scaling. We observe the following from the throughput plots:

- Figure 6.4 shows that the saturation throughput of centralized multi-core (4 cores) design for all traffic-mixes was 3.8× (average) higher than the centralized single-core design implies that the multi-core design scales linearly.

- The saturation throughput of horizontal scaling was 3.7× higher than the centralized single-core design for *Off-99* traffic-mix, whereas it was 2.4× higher for *Off-20* traffic mix (see Figure 6.4). The horizontal scaling design scales linearly when the synchronization cost is low, and the throughput gains decrease with the increase in synchronization rate due to increase in the fraction of non-offloadable traffic.

- The centralized multi-core design performed better than the horizontal scaling design since both of them have the same number of root controller cores, and the centralized multi-core design did not require state synchronization. Practically, there is a limit on the number of server CPU cores, so horizontal scaling must be preferred over centralized multi-core design when the control traffic rate is high. In case of horizontal scaling, we can scale the SDN application by spawning a large number of controller replicas, while reserving some CPU resource for state synchronization.

- The throughput of the local controller offload was 2× higher than the centralized multi-core design for the *Off-99* traffic-mix. Note that the local controllers were saturated, but the root CPU utilization was only 26% because of a small fraction of non-offloadable messages. As mentioned earlier, we could not generate enough load to saturate the root controller with four forwarding chains (up to *Off-70* traffic-mix); otherwise, the throughput improvements would be more significant with saturated root CPU (400%, which refers to all the 4 CPU cores running at 100%). Figure 6.4 shows that the local controller offload design shows better performance than the centralized multi-core design up to *Off-50* (1.4×). At *Off-20*, we observe that the throughput of centralized multi-core design was better than the local controller offload design, as the synchronization costs become more prominent than the offload benefits, i.e., $\frac{put\_rate}{access\_rate} > 1 + \Delta$ (§4.3.4).

- The throughput of local controller offload design with state replication for *Off-99* traffic-mix was 1.8× higher than horizontal scaling, and the root CPU utilization was only 30%. It implies that the throughput improvement must be better with saturated root CPU (400%). Even for *Off-20* traffic-mix, the saturation throughput of local controller offload design was 1.4× higher than horizontal scaling.

- Like the local controller offload design, the hardware offload design demonstrated significant throughput improvements when the fraction of non-offloadable messages in the traffic-mix was small (see Figure 6.3). As the fraction of offloadable messages reduces, fewer computations are processed on hardware, and the non-offloadable messages processed at the root controller waste the CPU cycles towards offloadable state synchronization. The throughput of the hardware offload design was 12× higher than the centralized multi-core design, with the root CPU utilization of 45% for *Off-99* traffic-mix. The addition of hardware chains to saturate the root CPU (400%) would further improve the throughput. When the traffic-mix was *Off-20*, the throughput gains of the hardware offload design dropped to 1.4× of the centralized multi-core design.

- The hardware offload design replicates the state at the data plane switch using the primary-backup mechanism (discussed in §5.3.3). The data plane switches run at line-rate; therefore, the application throughput was not impacted by replication.

Figure 6.5, Figure 6.6, and Figure 6.7 compare the response latency for offloadable EPC messages, non-offloadable EPC messages, and all EPC messages, respectively. We observe the following from the latency plots:

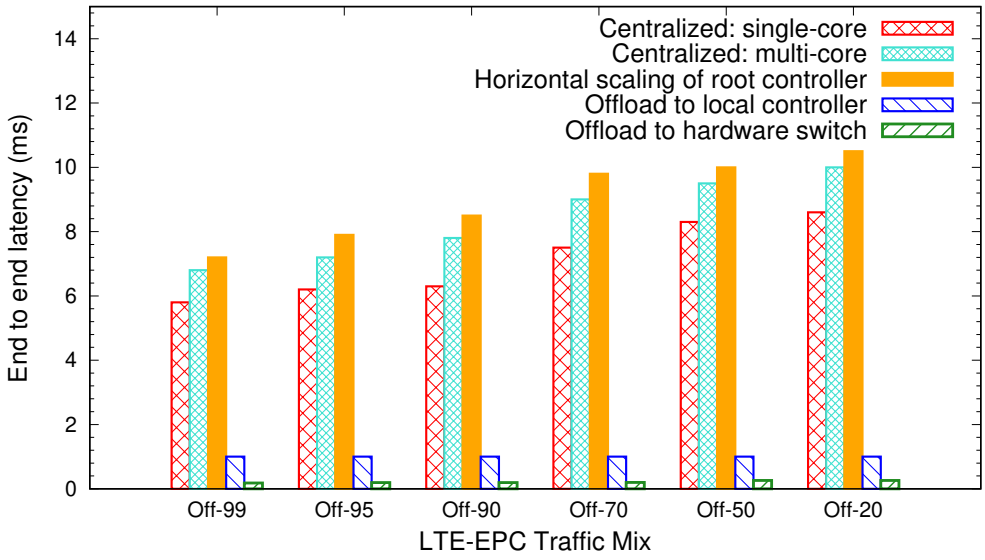


Figure 6.5: Response latency of offloadable EPC messages.

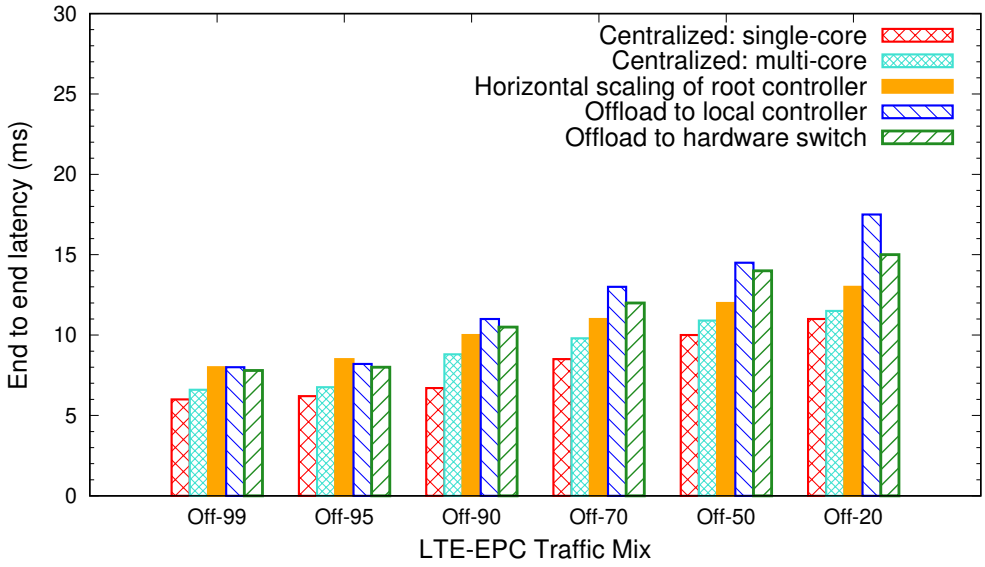


Figure 6.6: Response latency of non-offloadable EPC messages.

- Figure 6.5 shows that the response latency of offloadable EPC messages for horizontal scaling design was 8% (average) higher than the centralized multi-core design. The response latency of offloadable EPC messages for local controller offload design was 87% (average) lower than the centralized multi-core design. The response latency of offloadable EPC messages for hardware offload design was 97% (average) lower than the centralized multi-core design.

- Figure 6.6 shows that the response latency of non-offloadable EPC messages for the horizontal scaling design and the offload design were higher than the centralized multi-core design. The latency increase was due to the synchronization of

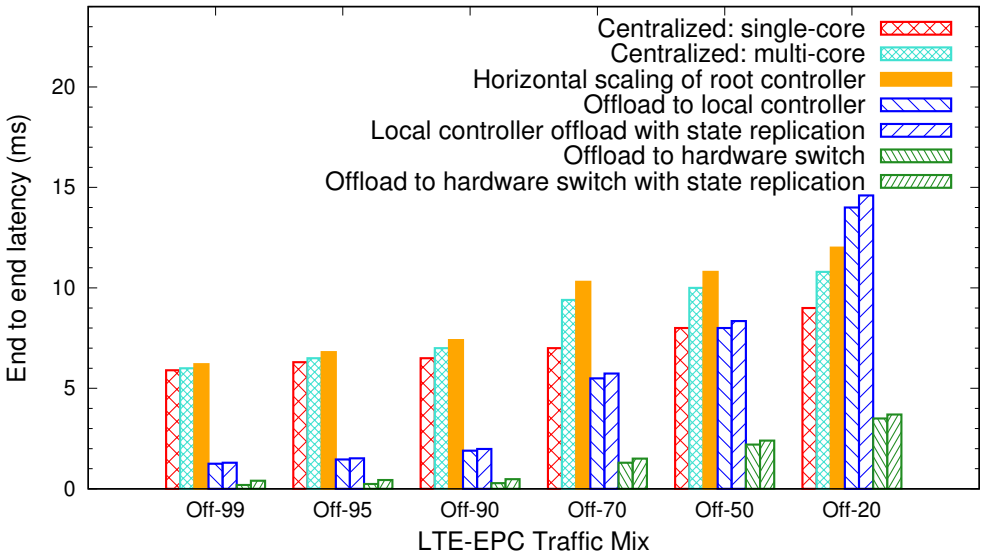


Figure 6.7: Response latency of all EPC messages.

updates to the offloadable state. The latency increase for the horizontal scaling design was observed to be lower than the offload designs because the root controller replicas are placed close to each other, within the data center core, whereas the root controller (at the core network) and the local controller/switch (close to end-user) are generally distant. In the case of offload design, the latency increase is higher when the fraction of non-offloadable traffic is high.

  - We observe that the response latency of non-offloadable EPC messages for the local controller offload design was 20% higher for *Off-99* and 52% higher for *Off-20* traffic-mix than the centralized multi-core design.
  - The non-offloadable EPC message response latency for the hardware offload design was 18% higher for *Off-99* and 30% higher for *Off-20* traffic-mix than the centralized multi-core design.

- Figure 6.7 shows that the average response latency of EPC messages for the offload designs was lower than the centralized multi-core design, except at *Off-20* for local controller offload design (cost of the offload > offload benefits). The processing of offloadable messages close to the user results in very low latencies, which amortize the latency increase due to non-offloadable message processing.

  - We observe that the average response latency for the local controller offload design was 80% lower for *Off-99* and 30% higher for *Off-20* traffic-mix than the centralized multi-core design.

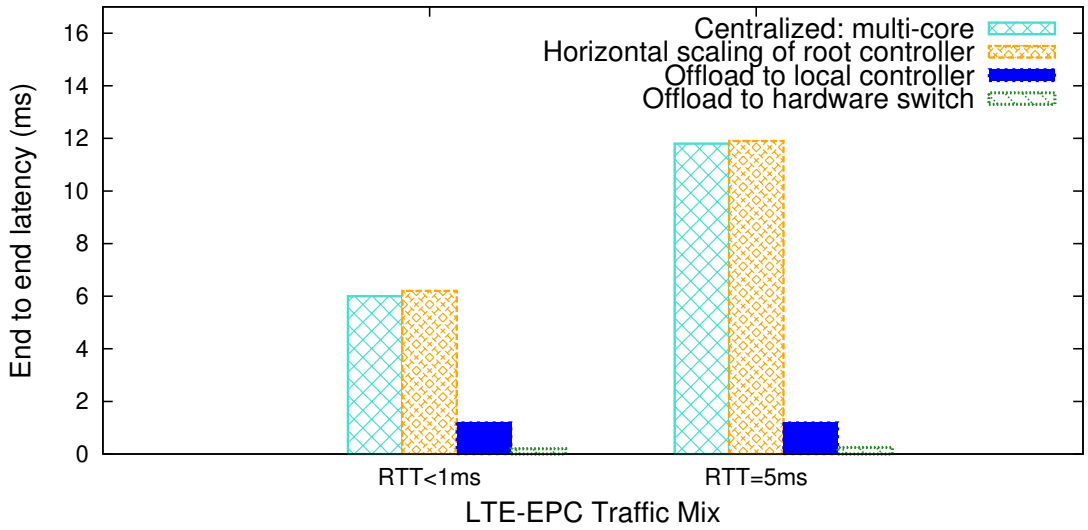


Figure 6.8: Response latency with varying distance to the root controller.

- Despite state replication at the local controllers, the latency for the local controller offload design was 79% lower than the horizontal scaling design for *Off-99* traffic-mix because a large fraction of messages were processed close to the user. But, the latency was 17% higher for *Off-20* traffic-mix because apart from state replication latency, a high fraction of non-offloadable messages update the offloaded state and induce synchronization delays.
- The average message response latency for the hardware offload design was 97% lower for *Off-99* and 60% lower for *Off-20* traffic-mix, compared to the centralized multi-core design.
- There was latency degradation observed for hardware offload design with state replication as the packets travel to/from the backup switch. The latency degradation depends on the RTT from the primary switch to the backup switch, which, in our case, was *~0.2 ms*. Despite the latency degradation, we observed that the latency of hardware offload for *Off-99* traffic-mix was 94% lower than horizontal scaling design, and 70% lower for *Off-20* traffic-mix.

6.4.2 Impact of distance of the root controller from the end-user

Figure 6.8 shows the response latency for EPC application messages for different RTT values between the UE (end-user) and the root SDN controller. The plot is for the *Off-99* traffic-mix since we wanted to observe the impact of RTT on offloadable messages. We have the following observations from the plot:

- The response latency of the offloadable messages for the offload designs is roughly the same even if the distance to the root controller increases. Also, the latency of the offload designs is significantly lower than the centralized and horizontal scaling designs. The near-constant, low latency was expected since the offloadable messages are processed at the local controllers or switches close to the user.

- The response latency of the offloadable messages for the centralized and horizontal scaling designs was *12 ms* for RTT = 5 ms, and they would increase further with higher RTT. These designs cannot be implemented for the 5G mobile network core as they cannot satisfy the *10 ms* latency constraint for data transfer when the user is in idle state (discussed in §5.1).

6.5 Choosing the right scalability design

We have observed from the empirical evaluation in §6.4 that every scalability design has strengths as well as limitations. We have demonstrated that our proposed offload designs provide significant performance gains in throughput and response latency. But, the offload design may not always be favorable. In this section, we define guidelines that suggest the choice of suitable scalability design. We have discussed the essential and desirable conditions for the application messages to be offloadable in §3.2. Next, we provide specific conditions for offloading application messages to programmable hardware.

6.5.1 Checklist to determine offload to programmable hardware

Typically, a packet processing application is the right candidate for offloading computations to programmable hardware. The performance gains are significant because the packets do not have to travel through the application server's network and application stack. We experience lower latencies and also save the server CPU that would have been used for packet processing. Virtual network functions (VNFs) like firewalls, load balancers, and intrusion detection systems are typical examples of packet processing applications. Further, the computations should satisfy the following conditions if they were to be offloaded to the programmable hardware.

1. The computations should be identified as offloadable. We have defined the conditions for computations to be offloadable in §3.2. This condition ensures that the states accessed by the computations could be made available on the hardware switch without any compromise on offloaded state consistency and application accuracy.

2. The hardware-dependent conditions that determine if the computations are offloadable are as follows (few of these were described in Table 2.2 of §2.3.3):

   (a) There should be no stalls during packet processing since the data plane switch commits to line-rate performance, i.e., the packet cannot wait at any stage for data or completion of other tasks, it has to move from one stage to the other at every clock tick.

   (b) The on-switch memory of the programmable hardware should be enough to store the offloadable state.

   (c) We should over-provision the programmable hardware to have enough spare capacity for application processing.

   (d) The offloadable computations should be programmed using the instruction set supported by the programmable hardware. The programmable hardware has a limited instruction set to ensure line-rate processing.

      i. The programmable hardware may not support all possible arithmetic operations; for example, the division operation may not be supported.

      ii. Arithmetic operations may be supported on an integer number of bytes due to alignment and padding constraints

      iii. Some architectures may only support multiplication with small constants, or shifts with small values due to operand constraints.

      iv. The programmable hardware has a limit on the number of packet processing pipeline stages. All the tables and features independent of each other can be part of the same pipeline stage. For example, IPv4 and IPV6 match-action table processing can share the same pipeline stage since the same packet will never match to both the tables.

      v. The code should not have any loops, and it cannot be recursive. This constraint ensures a deterministic number of pipeline stages and hence adheres to the line-rate performance commitment.

      vi. We should be able to store the application state at the switch tables or registers, but they have a limit on the maximum width. We need to split the state if its size is greater than this width. A typical programmable hardware also supports abstract global data structures called counters and meters. A counter can be used to maintain statistics. A meter is an advanced counter, that triggers an action when some condition on the counter value is satisfied.

Example computations that cannot be offloaded to the programmable hardware

A traffic-shaping application ensures that the packet rate of any flow does not exceed the given threshold. The implementation involves two components, (a) monitor the rate of each flow, (b) buffer the packets when the flow rate crosses the threshold. The second component requires the packet to halt at the switch, which is not possible. The packet has to leave through some interface or dropped, and the switch does not have enough memory to buffer these packets. But, it is possible to monitor the flow rates during packet processing and flag an anomaly when a flow's rate exceeds the threshold. Therefore, we can offload the flow monitoring component on the hardware switch.

Solutions like AccelTCP [88] have offloaded the subset of TCP protocol operations to the programmable hardware. TCP is a stateful protocol; therefore, TCP protocol offload implies that we must maintain the state for each TCP flow on the hardware switch. There are certain complex tasks like maintaining timers for each flow and retransmit the packets on timeout. Maintaining the state for a large number of flows on the hardware may not be feasible. AccelTCP implements tasks like connection establishment, termination, and splicing for a subset of flows on the hardware, which would require less memory. The implementation of timers on the programmable hardware is challenging; hence AccelTCP designs an optimized data structure for timers. The end-hosts handle operations like error control and congestion control since the functions are complicated, and the data packets require more memory than SYN/FIN/ACK packets.

Machine learning techniques comprise of feature extraction, which look like a decision tree. Dream [96] performs the task of feature extraction from the packet headers by mapping the classification process to the match-action tables at the programmable hardware. But, many machine learning models require complex features that may involve loops and recursion, so it may not be possible to extract them on a programmable switch.

6.5.2 Choice of the scalability design

This thesis solves the research problem of alleviating the root controller bottleneck and proposes the control plane scalability solutions. Therefore, we assume that the control plane traffic always saturates the centralized root SDN controller (refer §1.1). With this assumption in mind, we should choose one of the scalability solutions from horizontal-scaling design, local controller offload design, and hardware offload design. We now describe the decision making process for the choice of scalability design for a given application.

1. We have provided the necessary and desirable conditions for an application message to be declared as offloadable in §3.2. The *essential conditions* for offload

ensure application correctness when the application uses the offload design. One of the essential conditions is that all the states accessed by the application message should be offloadable. An application state is said to be offloadable if it is switch-local or has session-wide scope, and it should not be concurrently accessed from multiple network locations. We have an additional condition for hardware offload; the programmable hardware target should support the computations required for message processing. The *desirable conditions* guarantee application performance.

If at least the essential conditions are satisfied by the application, then test condition (2). Otherwise, the application should be implemented over the horizontal scaling framework.

2. We have provided a checklist that should be used to verify if the application can benefit from the hardware offload design in §6.5.1. The offloadable messages identified in (1) should satisfy conditions like: there should be no stalls during the message processing, the on-board memory should be enough to store the offloadable state, the programmable hardware should have spare capacity, and the message processing should be completed within a limited number of pipeline stages.

   If all the checklist conditions for programmable hardware offload are satisfied by the application, implement the application over the hardware offload framework. Otherwise, the application should be implemented over the local controller offload framework.

3. If condition (1) is satisfied, we know that the application can gain the benefits of offloading. But, if the fraction of non-offloadable and offloadable messages in the traffic-mix change dynamically, we must implement the Cuttlefish framework that adapts to the best SDN mode.

6.6 Summary

We demonstrated the empirical evaluation of the centralized, horizontal scaling, and the proposed offload designs. We found that the offload scalability design provides substantial performance improvements over the status-quo when the incoming traffic-mix comprises of a large fraction of offloadable messages. Finally, we provide guidelines to the application deployer to help her choose a suitable control plane scaling design.

Chapter 7

Future Work

This chapter discusses the insights gained for further system improvements while working on the thesis. These insights require additional exploration to understand the underlying state-of-the-art and feasibility. These ideas are open-ended, and we consider to pursue these as part of future work. We discuss two open-ended problems in this chapter.

7.1 TurboEPC extensions for 5G mobile packet core

The research community suggests the CUPS-based (Control User Plane Separation) architecture for the 4G mobile packet core network, but the current 4G packet core components are implemented over traditional hardware. But, the next-generation 5G mobile packet core is designed using the CUPS-based architecture and provided the specifications [29] for the same. The 5G control plane is physically separated from the user plane (or data plane), and the control plane components are implemented in software as VNFs (Virtual Network Functions).

The 5G control plane components perform functions that are similar to the corresponding 4G elements. Figure 7.1 shows the basic 5G components and the mapping with the relevant 4G components. The 4G MME functions are performed by the 5G Access and Mobility management Function (AMF) and Session Management Function (SMF). SMF also performs the control plane functions of the 4G SGW and PGW. The 4G HSS is implemented as the 5G Authentication Server Function (AUSF) and User Data Management (UDM). The UDM stores the state for all the users that can be accessed by any 5G component. The 4G PCRF is renamed as PCF in 5G. The 5G User Plane Function (UPF) refers to the data plane switches.

According to the 5G 3GPP standards [122] (discussed earlier in §5.1), the response latency for control plane messages that switch between *IDLE* and *CONNECTED* states

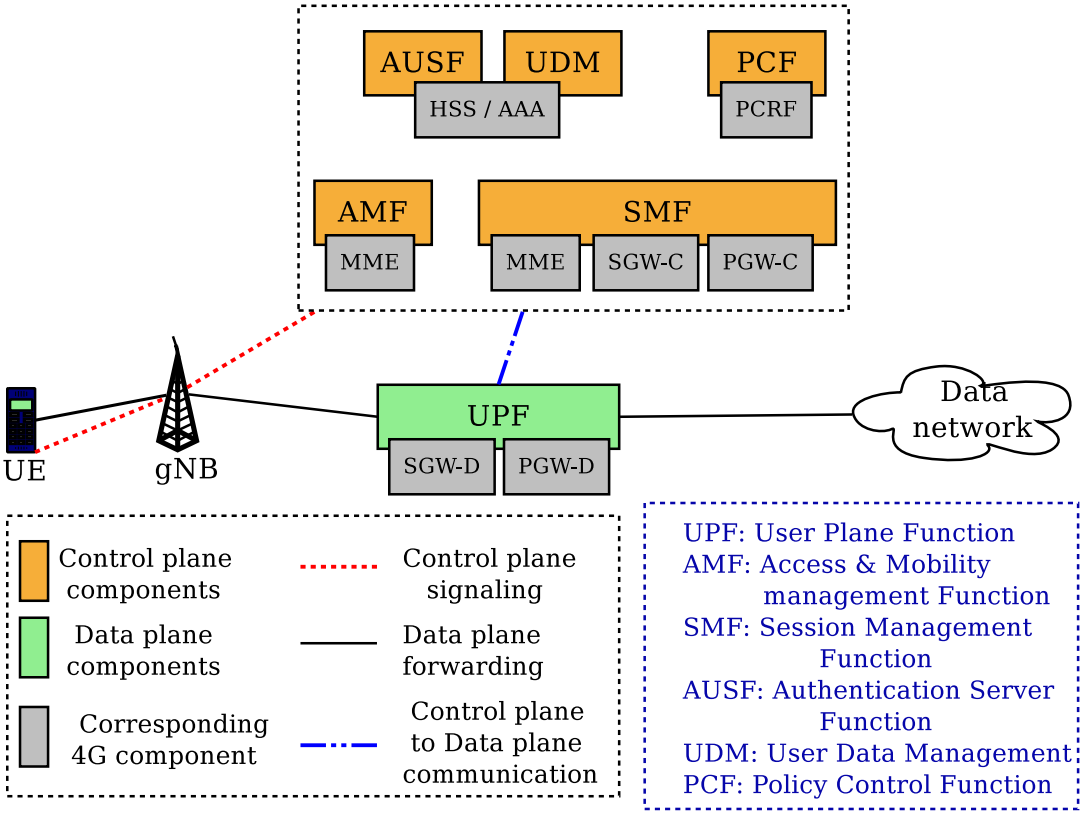


Figure 7.1: 5G components with corresponding 4G components.

should be less than 10 ms so that the data transfer can be initiated quickly and SLAs are protected. We have seen that our TurboEPC design is capable of providing such low control plane latency (~0.3 ms) by offloading the control plane computations to the programmable hardware. The offloadable 4G control plane functions like the S1 release and service request correspond to the 5G control plane functions, AN release, and session modification request. The AN release and session modification request control plane functions are responsible for the IDLE-CONNECTED user state switch. We want to answer the following questions for improvements to the 5G mobile packet core:

- Can we extend the current TurboEPC design to offload the AN release and session modification request 5G control plane functions to the programmable hardware? For example, the 5G standard wraps the control messages into HTTP packets; therefore, we will have to process HTTP connections in hardware. We need to identify additional challenges.

- The authentication, encryption, and message integrity procedures processed by the AMF/AUSF consume most of the host CPU. Can we offload these procedures to the programmable hardware so that the control plane can use the spare CPU for other important tasks like network management and control?

- All the 5G components frequently talk to the UDM for state access and state update operations, and this state is replicated for failure management. Can we reduce the latency of these operations by offloading the UDM state access and replication functions to the programmable hardware?

We want to explore these design questions and build an accelerated 5G mobile packet core over programmable hardware.

7.2 Three-tier adaptive hierarchical design

Our TurboEPC design proposed to process the offloadable control plane computations at the programmable hardware switch close to the user. The control plane performance is accelerated by utilizing the spare capacity at the switch. Let us look at a few concerns.

- Under peak load conditions, the data traffic could saturate the switch. The control plane traffic and data plane traffic will interfere with each other and cause performance degradation for both control and data traffic.

- Our current TurboEPC implementation always runs in offload mode because the non-offloadable traffic fraction for 4G networks is always less than 8% (Attach/Detach: 1–2%, Handover: 5%), and the synchronization costs are low. But, the TurboEPC design applies to a general class of applications. What happens if the state synchronization cost is higher than the offload benefits for a generic TurboEPC application? For example, the 5G mobile packet core comprise of a class of users that have high mobility rate, so the handover rate will be high for such users.

Towards solving these concerns, we want to explore the broad scheme of offload. We propose a three-tier hierarchical scaling design (see Figure 7.2) where the tiers are: (1) the root controller, (2) the local controller, and (3) the programmable hardware switch. The root controller can process all computations, but it is expensive (centralized SDN design). The local controller and the programmable hardware can process offloadable computations, where the programmable hardware is preferred if the application computations are offloadable to the hardware.

There are three conditions when the computations already offloaded to the programmable hardware have to be revoked.

1. The synchronization rate due to updates to the offloaded state by the non-offloadable computations is higher than the offload benefits.

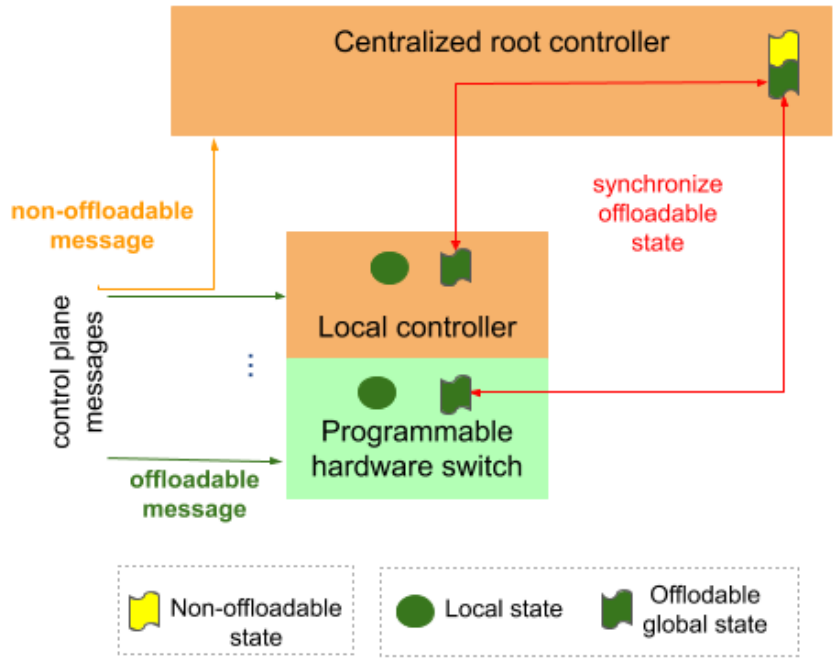


Figure 7.2: Three-tier adaptive hierarchical design.

2. The switch is saturated due to high data plane traffic, and there is no spare capacity for control plane packet processing.

3. The switch memory is exhausted, and the state of additional users/flows cannot be stored.

If condition (1) holds, our proposal, Cuttlefish, states that we must migrate the application processing to the centralized root controller. Can we do this intelligently? There may be few flows responsible for the increased synchronization cost. We should be able to dynamically classify the flows, and identify the set of flows that cause high synchronization at the root controller, and migrate the processing of such flows to the root controller. The rest of the flows can benefit from the offload design. For example, in the 5G mobile packet core application, there is a class of users with high mobility, so the handover rate of such users would be very high (high synchronization cost). We should identify such users and migrate their processing to the root controller. We have to explore the challenges of gathering per-user or per-traffic-class statistics. Of course, the size of these statistics would be large. We need to identify the additional overheads at the data plane, switch control plane, and the root controller. We also need to verify if the benefits obtained by such a selective offload mechanism are significant.

If condition (2) holds, it does not make sense to demote offloadable computations processing to the root controller (centralized mode). Instead, we must migrate the offloadable computations from hardware switch to the local controller at (or close to) the switch.

Since the state synchronization cost is low, the local controller offload will provide better performance than the traditional centralized design. We need to explore the feasibility of this proposal. We need to quantify the overheads of maintaining the state consistency for the two-level cache. That is, the master copy of the offloadable state is available at the root, and the copy of this state is cached at the local controller and the hardware switch. We need to ensure that the stale state is not accessed and application accuracy is preserved. Further, we need to quantify the overheads during mode-switch between the three hierarchical nodes. Finally, we need to explore various cost-benefit aspects to evaluate the gains of such a hierarchical offload design.

In chapter 5, we have already suggested the solution for condition (3). The offloadable state is partitioned and stored across multiple hardware switches, and the root controller manages the partitions. Alternatively, we can process the computations of the additional flows (whose state is not stored on the hardware) at the local controllers. To achieve this, we should implement cache eviction policies (similar to Netcache [85]) such that the more frequently accessed state remains in the dataplane or local controllers, whereas the less frequently accessed state is moved to the root controller dynamically during runtime. We must compare the performance of the hierarchical offload design with the two-level cache with the design where the state is partitioned across multiple switches, to decide if one of the designs is better.

7.3 Summary

We have defined and described two broad research problems that have been identified while working on the thesis. We have presented a few alternative design options for our proposed systems. With the evolution of programmable network hardware and accelerators, there are many open, challenging, and exciting problems in the domain that the system and network researchers would address in the future.

Chapter 8

Conclusion

The primary focus of this thesis was to alleviate the control plane bottleneck for SDN applications. We presented the challenges and existing solution directions for alleviating the control plane bottleneck. Out of the existing control plane scaling approaches,i.e., horizontal and hierarchical scaling, we chose the hierarchical scaling approach and advanced the state-of-the-art. Traditional hierarchical scaling solutions offload the computations that depend only on the switch-specific local state to the local controllers/switches that are close to the user. This offload reduces the load at the centralized root controller hence scales the application and also reduces the response latency. Not many applications have these types of computations, limiting the applicability of the hierarchical design. The key ideas of this thesis were as follows:

1. We classified the application state and identified a class of the global application state that can be offloaded to the local controllers/switches (offloadable state). We proposed an offload approach where the application computations that depend only on offloadable state can also be offloaded locally, with reduced synchronization costs compared to horizontal scaling solutions.

2. Since the typical traffic-mix is dynamic, the fraction of non-offloadable messages in the traffic-mix that modify offloadable state may increase, which increases the synchronization cost. There could be conditions when the cost of the offload (synchronization costs) exceeds the offload gains (throughput and latency); therefore, the proposed offload approach is not always the right solution. We designed an adaptive framework for SDN applications that dynamically identifies the best SDN operation mode and automatically switches between the traditional centralized and our proposed offload SDN modes based on the synchronization costs introduced by the current traffic-mix.

3. We implemented our proposed hierarchical design concepts to offload SDN applications over fast programmable hardware devices and accelerate the SDN control plane of the mobile packet core application.

While incorporating our key ideas, we identified many challenges like high synchronization costs, the inconsistency of offloaded state, state losses due to local node failure, and limited memory to store offloaded state for programmable hardware. We have addressed these challenges and advanced the current state-of-the-art with the design and implementation of two hierarchical scaling designs—Cuttlefish and TurboEPC. Cuttlefish offloads the application computations to the software local controllers, whereas TurboEPC offloads the application computations to the hardware programmable P4-based switches. Our proposal Cuttlefish incorporated an adaptive offload capability to balance the tradeoff between performance gains due to state offload, and the cost of synchronizing this state across the root and local controllers—a win-win design approach.

We performed an empirical evaluation of the existing and proposed control plane scaling approaches. We observe that the choice of the suitable control plane scaling framework depends on various parameters like:

- the application characteristics like whether the application instructions are stateless or stateful (offloadable or not) and the application message's SLA constraints
- the traffic characteristics like the fraction of offloadable traffic in the total traffic
- the capabilities of the target node where the computations are offloaded, for example, the on-chip memory size and the supported instruction set

We have provided guidelines on how to choose an appropriate SDN scalability design based on these parameters. We have also provided guidelines on how to identify the offloadable computations of an application.

We presented the scaling of one of the popular use-case — the mobile packet core, using both the hierarchical Cuttlefish framework that offloads computations to local controllers and TurboEPC that offloads computations to programmable hardware switches (close to the user). We have observed significant throughput and latency gains by processing some of the more frequent signaling messages at local controllers/switches closer to the edge. Our ideas can be applied to other applications with offloadable computations and can be identified using our guidelines. There are other ways of alleviating the control plane bottleneck that we have presented as part of our future work.

References

[1] M. Karakus and A. Durresi, "A survey: Control plane scalability issues and approaches in software-defined networking (sdn)," *Computer Networks*, vol. 112, pp. 279–293, 2017.

[2] C. El Khalfi, A. El Qadi, and H. Bennis, "A comparative study of software defined networks controllers," in *Proceedings of the 2nd International Conference on Computing and Wireless Communication Systems*, 2017, pp. 1–5.

[3] L. Zhu, M. M. Karim, K. Sharif, F. Li, X. Du, and M. Guizani, "Sdn controllers: Benchmarking & performance evaluation," *arXiv preprint arXiv:1902.04491*, 2019.

[4] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Computer Communication Review*, vol. 44, 2014.

[5] S. Tabbane, "Core network and transmission dimensioning," https://www.itu.int/en/ITU-D/Regional-Presence/AsiaPacific/SiteAssets/Pages/Events/2016/Aug-WBB-Iran/Wirelessbroadband/core%20network%20dimensioning.pdf, 2016.

[6] D. Nowoswiat, "Managing LTE Core Network Signaling Traffic," https://www.nokia.com/en_int/blog/managing-lte-core-network-signaling-traffic, 2013.

[7] O. W. Paper, "Software-Defined Networking: The New Norm for Networks," https://pdfs.semanticscholar.org/a3f6/9f6181a0b4d481073a21eafbcc434a800db6.pdf, 2012.

[8] A. Zaslavsky, C. Perera, and D. Georgakopoulos, "Sensing as a service and big data," *arXiv preprint arXiv:1301.0159*, 2013.

[9] J. Ochs, "Scalability in cloud computing: using virtualization to save money," https://www.servercentral.com/blog/scalability-in-cloud-computing/, 2014.

[10] M. Nelson, B.-H. Lim, G. Hutchins *et al.*, "Fast transparent migration for virtual machines." in *USENIX Annual technical conference, general track*, 2005, pp. 391–394.

[11] "Sdn adoption in enterprises," https://www.wipro.com/en-IN/infrastructure/sdn-adoption-in-enterprises/, 2019.

[12] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn," *ACM Queue*, vol. 11, 2013.

[13] T. Koponen *et al.*, "Onix: A distributed control platform for large-scale production networks," in *Proceedings of the OSDI*, 2010.

[14] A. Tootoonchian and Y. Ganjali, "Hyperflow: A distributed control plane for openflow," in *Proceedings of the the INM/WREN*, 2010.

[15] A. R. Curtis *et al.*, "Devoflow: Scaling flow management for high-performance networks," in *Proceedings of the ACM SIGCOMM*, 2011.

[16] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: A framework for efficient and scalable offloading of control applications," in *Proceedings of the HotSDN*, 2012.

[17] J. Yang, Z. Zhou, T. Benson, X. Yang, X. Wu, and C. Hu, "Focus: Function offloading from a controller to utilize switch power," in *Proceedings of the IEEE NFV-SDN*, 2016.

[18] A. S. . Tam, Kang Xi, and H. J. Chao, "Use of devolved controllers in data center networks," in *2011 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2011.

[19] "ONOS SDN controller," https://github.com/opennetworkinglab/onos, 2017.

[20] S. H. Yeganeh and Y. Ganjali, "Beehive: Simple distributed programming in software-defined networks," in *Proceedings of the SoSR*, 2016.

[21] M. Yu *et al.*, "Scalable flow-based networking with difane," in *Proceedings of the ACM SIGCOMM.*, 2010.

[22] M. A. S. Santos, B. A. A. Nunes, K. Obraczka, T. Turletti, B. T. de Oliveira, and C. B. Margi, "Decentralizing sdn's control plane," in *39th Annual IEEE Conference on Local Computer Networks*, 2014.

[23] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a globally-deployed software defined wan," *SIGCOMM Computer Communication Review*, vol. 43, no. 4, 2013.

[24] K.-K. Yap, M. Motiwala, J. Rahe, S. Padgett, M. Holliman, G. Baldus, M. Hines, T. Kim, A. Narayanan, A. Jain, V. Lin, C. Rice, B. Rogan, A. Singh, B. Tanaka, M. Verma, P. Sood, M. Tariq, M. Tierney, D. Trumic, V. Valancius, C. Ying, M. Kallahalla, B. Koley, and A. Vahdat, "Taking the edge off with espresso: Scale, reliability and programmability for global internet peering," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*, 2017.

[25] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea, "Enabling end-host network functions," in *Proceedings of the ACM SIGCOMM*, 2015.

[26] M. A. Togou, D. A. Chekired, L. Khoukhi, and G. Muntean, "A hierarchical distributed control plane for path computation scalability in large scale software-defined networks," *IEEE Transactions on Network and Service Management*, vol. 16, no. 3, pp. 1019–1031, 2019.

[27] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, 2012.

[28] "Ericsson Mobility Report," http://www.ericsson.com/res/docs/2016/ericsson-mobility-report-2016.pdf, 2016.

[29] 3GPP, "5g 3gpp specifications," https://www.3gpp.org/ftp/Specs/archive/23_series/23.502/, 2017.

[30] R. E. Hattachi, "Next generation mobile networks, ngmn," https://www.ngmn.org/fileadmin/ngmn/content/downloads/Technical/2015/NGMN_5G_White_Paper_V1_0.pdf, 2015.

[31] D. Kim, "5g stats," https://techneconomyblog.com/tag/economics/, 2017.

[32] C. Kachris, K. Kanonakis, and I. Tomkos, "Optical interconnection networks in data centers: recent trends and future challenges," *IEEE Communications Magazine*, vol. 51, no. 9, pp. 39–45, 2013.

[33] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, 2010, pp. 267–280.

[34] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, 2015.

[35] T. Holterbach, E. C. Molero, M. Apostolaki, A. Dainotti, S. Vissicchio, and L. Vanbever, "Blink: Fast connectivity recovery entirely in the data plane," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019.

[36] "NoviWare 400.5 for Barefoot Tofino chipset," https://noviflow.com/wp-content/uploads/NoviWare-Tofino-Datasheet.pdf, 2018.

[37] N. Systems, "vEPC Acceleration Using Agilio SmartNICs," https://www.netronome.com/media/documents/SB_vEPC.pdf, 2017.

[38] TRAI, "Highlights of Telecom Subscription Data," https://main.trai.gov.in/sites/default/files/PR_60_TSD_Jun_170817.pdf, 2017.

[39] R. Shah, M. Vutukuru, and P. Kulkarni, "Cuttlefish project," https://github.com/rinku-shah/cuttlefish, 2018.

[40] R. Shah, V. Kumar, M. Vutukuru, and P. Kulkarni, "Turboepc github code," https://github.com/rinku-shah/turboepc, 2015.

[41] M. Alsaeedi, M. M. Mohamad, and A. A. Al-Roubaiey, "Toward adaptive and scalable openflow-sdn flow control: A survey," *IEEE Access*, vol. 7, pp. 107 346–107 379, 2019.

[42] N. McKeown *et al.*, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, 2008.

[43] D. Erickson, "The beacon openflow controller," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, 2013.

[44] "Floodlight SDN controller," https://github.com/floodlight, 2016.

[45] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: Towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, p. 105–110, Jul. 2008.

[46] Z. Cai, A. L. Cox, and T. Ng, "Maestro: A system for scalable openflow control," Tech. Rep., 2010.

[47] Y. Fu, J. Bi, K. Gao, Z. Chen, J. Wu, and B. Hao, "Orion: A hybrid hierarchical control plane of software-defined networking for large-scale networks," in *IEEE 22nd International Conference on Network Protocols*, 2014.

[48] M. Li, X. Wang, H. Tong, T. Liu, and Y. Tian, "Sparc: Towards a scalable distributed control plane architecture for protocol-oblivious sdn networks," in *28th International Conference on Computer Communication and Networks (ICCCN)*, 2019.

[49] F. Kandah, I. Ozcelik, and B. Huber, "Mars: Machine learning based adaptable and robust network management for software-defined networks," in *The 10th Annual Computing and Communication Workshop and Conference (CCWC)*, 2020.

[50] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes." in *Proceedings of the NSDI*, 2013.

[51] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, "Opennf: Enabling innovation in network function control," in *ACM SIGCOMM Computer Communication Review*, vol. 44, 2014.

[52] S. Rajagopalan, D. Williams, and H. Jamjoom, "Pico replication: A high availability framework for middleboxes," in *Proceedings of the SoCC*, 2013.

[53] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. R. Kompella, "Elasticon; an elastic distributed sdn controller," in *Proceedings of the ANCS*, 2014.

[54] "NVGRE standard," https://tools.ietf.org/html/rfc7637, 2015.

[55] "VXLAN standard," https://tools.ietf.org/html/rfc7348, 2014.

[56] "STT standard," https://tools.ietf.org/html/draft-davie-stt-01, 2012.

[57] OVS, "OpenvSwitch," https://github.com/openvswitch, 2011.

[58] "OVS-DPDK software switch," https://software.intel.com/en-us/articles/open-vswitch-with-dpdk-overview, 2016.

[59] "BESS software switch," http://span.cs.berkeley.edu/bess.html, 2015.

[60] "Intel Vector Packet Processor(VPP)," https://wiki.fd.io/view/VPP, 2018.

[61] L. Rizzo and G. Lettieri, "Vale, a switched ethernet for virtual machines," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*, 2012.

[62] R. Ozdag, "Intel Ethernet Switch FM6000 Series - Software Defined Networking," https://people.ucsc.edu/~warner/Bufs/ethernet-switch-fm6000-sdn-paper.pdf, 2019.

[63] "Cisco highlights next big switch." https://www.biztechafrica.com/article/cisco-announces-next-big-switch/5448/, 2013.

[64] "Cavium Xpliant ethernet switch product line." https://people.ucsc.edu/~warner/Bufs/Xpliant-cavium.pdf, 2015.

[65] "EZchip." https://www.mips.com/partner/ezchip/, 2019.

[66] "Xilinx." https://www.xilinx.com/, 2019.

[67] "Altera." https://www.mouser.in/manufacturer/altera/, 2019.

[68] "EBPF (extended Berkeley Packet Filter." https://www.iovisor.org/technology/ebpf, 2019.

[69] "Intel Data Plane Development Kit(DPDK)," https://www.dpdk.org/, 2018.

[70] "P4-16 language specifications," https://p4.org/p4-spec/docs/P4-16-v1.2.0.html, 2019.

[71] "P4Runtime Specifications," https://p4.org/p4runtime/spec/master/P4Runtime-Spec.html, 2020.

[72] "General purpose RPC (gRPC)," https://grpc.io/, 2018.

[73] "P4Runtime Github code," https://github.com/p4lang/p4runtime, 2018.

[74] "P4Runtime server implementation," https://github.com/p4lang/PI, 2018.

[75] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM*, 2015.

[76] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim, "Language-directed hardware design for network performance monitoring," in *Proceedings of the the ACM SIGCOMM Conference*, 2017.

[77] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *Proceedings of the the SoSR*, 2017.

[78] R. Harrison, Q. Cai, A. Gupta, and J. Rexford, "Network-wide heavy hitter detection with commodity switches," in *Proceedings of the Symposium on SDN Research*, 2018.

[79] R. Harrison, S. L. Feibish, A. Gupta, R. Teixeira, S. Muthukrishnan, and J. Rexford, "Carpe elephants: Seize the global heavy hitters," in *Proceedings of the Workshop on Secure Programmable Network Infrastructure*, 2020.

[80] E. Cidon, S. Choi, S. Katti, and N. McKeown, "Appswitch: Application-layer load balancing within a software switch," in *Proceedings of the APNet*, 2017.

[81] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *Proceedings of the the SoSR*, 2016.

[82] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proceedings of the the ACM SIGCOMM Conference*, 2017.

[83] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé, "Netpaxos: Consensus at network speed," in *Proceedings of the the ACM SIGCOMM SoSR*, 2015.

[84] H. T. Dang *et al.*, "Consensus for non-volatile main memory," in *IEEE 26th International Conference on Network Protocols (ICNP)*, 2018.

[85] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the SOSP*, 2017.

[86] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "Kv-direct: High-performance in-memory key-value store with programmable nic," in *Proceedings of the SOSP*, 2017.

[87] H. Giesen, L. Shi, J. Sonchack, A. Chelluri, N. Prabhu, N. Sultana, L. Kant, A. J. McAuley, A. Poylisher, A. DeHon *et al.*, "In-network computing to the rescue of faulty links," in *Proceedings of the 2018 Morning Workshop on In-Network Computing*, 2018, pp. 1–6.

[88] Y. Moon, S. Lee, M. A. Jamshed, and K. Park, "Acceltcp: Accelerating network applications with stateful {TCP} offloading," in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI}s)*, 2020, pp. 77–92.

[89] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. R. Ports, and P. Richtárik, "Scaling distributed machine learning with in-network aggregation," *arXiv preprint arXiv:1903.06701*, 2019.

[90] C. Cascone and U. Chau, "Offloading vnfs to programmable switches using p4," in *ONS North America*, 2018.

[91] A. Aghdai *et al.*, "Transparent edge gateway for mobile networks," in *IEEE 26th International Conference on Network Protocols (ICNP)*, 2018.

[92] E. C. Molero, S. Vissicchio, and L. Vanbever, "Hardware-accelerated network control planes," in *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, ser. HotNets, 2018.

[93] "Noviflow switch," https://noviflow.com/noviswitch, 2018.

[94] K.-F. Hsu, P. Tammana, R. Beckett, A. Chen, J. Rexford, and D. Walker, "Adaptive weighted traffic splitting in programmable data planes," in *Proceedings of the Symposium on SDN Research*, 2020.

[95] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *Proceedings of the ACM SIGCOMM Conference*, 2013.

[96] Z. Xiong and N. Zilberman, "Do switches dream of machine learning? toward in-network classification," in *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, 2019.

[97] "The evolved packet core," http://www.3gpp.org/technologies/keywords-acronyms/100-the-evolved-packet-core, 2017.

[98] "Control and User Plane Separation," http://www.3gpp.org/cups, 2017.

[99] "EPC: S1 release," https://gitlab.eurecom.fr/oai/openairinterface5g/issues/16, 2016.

[100] N. S. Networks, "Signaling is growing 50% faster than data traffic," https://docplayer.net/6278117-Signaling-is-growing-50-faster-than-data-traffic.html, 2012.

[101] "On Signalling Storm," https://blog.3g4g.co.uk/2012/06/on-signalling-storm-ltews.html, 2012.

[102] P. Kiss, A. Reale, C. J. Ferrari, and Z. Istenes, "Deployment of iot applications on 5g edge," in *IEEE International Conference on Future IoT Technologies*, 2018.

[103] A. Banerjee, R. Mahindra, K. Sundaresan, S. Kasera, K. Van der Merwe, and S. Rangarajan, "Scaling the lte control-plane for future mobile access," in *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, 2015.

[104] J. Cho, R. Stutsman, and J. Van der Merwe, "Mobilestream: A scalable, programmable and evolvable mobile core control plane platform," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, 2018.

[105] V. Nagendra, A. Bhattacharya, A. Gandhi, and S. R. Das, "Mmlite: A scalable and resource efficient control plane for next generation cellular packet core," in *Proceedings of the 2019 ACM Symposium on SDN Research*, 2019.

[106] Y. Li, Z. Yuan, and C. Peng, "A control-plane perspective on reducing data access latency in lte networks," in *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*, 2017.

[107] A. Mohammadkhan, K. Ramakrishnan, A. S. Rajan, and C. Maciocco, "Cleang: A clean-slate epc architecture and controlplane protocol for next generation cellular

networks," in *Proceedings of the 2016 ACM Workshop on Cloud-Assisted Networking*, 2016.

[108] M. Pozza, A. Rao, A. Bujari, H. Flinck, C. E. Palazzi, and S. Tarkoma, "A refactoring approach for optimizing mobile networks," in *2017 IEEE International Conference on Communications (ICC)*, 2017.

[109] M. T. Raza, D. Kim, K. Kim, S. Lu, and M. Gerla, "Rethinking lte network functions virtualization," in *IEEE 25th International Conference on Network Protocols (ICNP)*, 2017.

[110] Z. A. Qazi, M. Walls, A. Panda, V. Sekar, S. Ratnasamy, and S. Shenker, "A high performance packet core for next generation cellular networks," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017.

[111] H. Lindholm *et al.*, "State space analysis to refactor the mobile core," in *Proceedings of the AllThingsCellular*, 2015.

[112] X. An, F. Pianese, I. Widjaja, and U. G. Acer, "Dmme: A distributed lte mobility management entity," *Bell Labs Technical Journal*, vol. 17, no. 2, pp. 97–120, 2012.

[113] R. Balakrishnan and I. Akyildiz, "Local anchor schemes for seamless and low-cost handover in coordinated small cells," *IEEE Transactions on Mobile Computing*, vol. 15, no. 5, pp. 1182–1196, 2016.

[114] X. Jin, L. E. Li, L. Vanbever, and J. Rexford, "Softcell: Scalable and flexible cellular core network architecture," in *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, 2013.

[115] ONF, "OpenFlow-enabled SDN and Network Functions Virtualization," https://www.opennetworking.org/images/stories/downloads/sdn-resources/solution-briefs/sb-sdn-nvf-solution.pdf, 2014.

[116] "Behavioral-model," https://github.com/p4lang/behavioral-model/tree/master/targets/simple_switch_grpc, 2017.

[117] "Agilio CX SmartNIC," https://www.netronome.com/m/documents/PB_NFP-4000.pdf, 2018.

[118] S. Filiposka and I. Mishkovski, "Smartphone user's traffic characteristics and modelling," *Transactions on Networks and Communications*, vol. 1, no. 1, Dec.

2013. [Online]. Available: https://journals.scholarpublishing.org/index.php/TNC/article/view/22

[119] S. Choi, B. Burkov, A. Eckert, T. Fang, S. Kazemkhani, R. Sherwood, Y. Zhang, and H. Zeng, "Fboss: Building switch software at scale," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018.

[120] J. Sonchack, "Enabling practical software-defined networking security applications with ofx." 2016.

[121] A. Jain, S. Lohani, and M. Vutukuru, "Opensource SDN LTE EPC," https://github.com/networkedsystemsIITB/SDN_LTE_EPC, 2016.

[122] e. Mikko Saily, "5G Asynchronous Control Functions and Overall Control Plane Design," https://ec.europa.eu/research/participants/documents/downloadPublic?documentIds=080166e5b200001d&appId=PPGMS, 2017.

[123] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "Netchain: Scale-free sub-rtt coordination," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[124] L. Zeno, D. R. K. Ports, J. Nelson, and M. Silberstein, "Swishmem: Distributed shared state abstractions for programmable switches," in *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, 2020.

[125] "POX SDN controller." https://noxrepo.github.io/pox-doc/html/, 2014.

[126] M. Lahn, "Server cost," https://www.servermania.com/kb/articles/how-much-does-a-64-core-server-cost/, 2017.

[127] "Cost of Intel Xeon Quad Processsor." https://www.amazon.in/Intel-Xeon-Quad-Processor-BX80644E51620V3/dp/B00MU046J4/ref=sr_1_2?dchild=1&keywords=intel+quad+core+server+processor&qid=1592073708&s=computers&sr=1-2, 2020.

[128] "Cost of Netronome 10 Gbps CX-4000." https://colfaxdirect.com/store/pc/viewPrd.asp?idproduct=3017&idcategory=0, 2020.

List of Publications

Conference publications

1. Rinku Shah, Vikas Kumar, Mythili Vutukuru, Purushottam Kulkarni. TurboEPC: Leveraging data plane programmability to accelerate the mobile packet core. In Proceedings of the ACM SIGCOMM Symposium on SDN Research (**SOSR**), 2020.

2. Rinku Shah, Mythili Vutukuru, Purushottam Kulkarni. Cuttlefish: Hierarchical SDN Controllers with Adaptive Offload. In Proceedings of the IEEE 26th International Conference on Network Protocols (**ICNP**), 2018.

Workshop publications

1. Rinku Shah, Aniket Shirke, Akash Trehan, Mythili Vutukuru, Purushottam Kulkarni. pcube: Primitives for network data plane programming. In Proceedings of the IEEE 26th International Conference on Network Protocols (**ICNP**), 2018.

2. Rinku Shah, Mythili Vutukuru, Purushottam Kulkarni. Devolve-Redeem: Hierarchical SDN Controllers with Adaptive Offloading. In Proceedings of the ACM First Asia-Pacific Workshop on Networking (**APNet**), 2017.

Acknowledgements

I have learned some of the best life lessons, both personally and professionally, during the time spent at IIT Bombay. I want to express my gratitude to some of the wonderful people who have been instrumental in my success.

First, I want to thank my supervisors Prof. Mythili Vutukuru and Prof. Purushottam Kulkarni (Puru), for their constant motivation, guidance, and valuable feedback. Prof. Mythili's passion for her work, clarity of thought, creative ideas to solve a problem, and knack for fulfilling so much without stressing out is genuinely inspirational. Her belief in me was extremely encouraging; she is instrumental in imbibing the never-give-up attitude and taught me to accept nothing less than excellence. Besides being a great advisor, she has been a mentor and a guide towards personal matters. Thank you, Prof. Mythili. Prof. Puru's bird's eye view towards looking at the problems, rigorous brainstorming, and microscopic observations played a vital role in shaping my dissertation. Besides being a great advisor, he has been a mentor and a friend with excellent people skills. Thank you, Prof. Puru. Stable finances are crucial in a Ph.D. student's life. During my Ph.D., I had to resign from my ongoing job to focus on research. My advisors made sure that I do not have to worry about my finances, which was a great relief.

I want to extend my deep gratitude to my dissertation committee members, Prof. Umesh Bellur and Prof. Varsha Apte, for their encouragement and valuable feedback. Prof. Umesh was the first to lend his hand and accept me as his research student. Your guidance has helped me understand the research process during my initial years. I switched my research domain but was fortunate to have you on my dissertation committee and avail your constant guidance. Your out-of-the-box feedback helped me immensely in the visualization of my research problem from a different perspective. Thank you, Prof. Umesh. Prof. Varsha provided critical feedback and has always been motivating. Her energy has been contagious, inspiring me to get involved in aspects beyond research, such as organizing social lab events. Thank you, Prof. Varsha.

I want to thank my labmate, Vikas, for being a fantastic co-worker and a great friend. You were a great support system when we together started to explore the programmable

network hardware domain. I was fortunate to mentor some incredible BTech and MTech students, Sanjana, Akash, Aniket, Maharishi, Vishal, Arijit, and Kanak. I have learned a lot from you and had a wonderful time working with you. I am fortunate to mentor the Ph.D. student, Abhik. I had a fantastic time listening to his crazy but meaningful discussions.

A special thanks to my labmates, Priyanka, Akanksha, Dhantu, Anshuj, Avinash, Chandra Prakash, Unais, and Nitin. You have been the source for your unconditional support, encouragement, and means for rejuvenation under conditions of stress and self-doubt. Our chai sessions, festive celebrations, and parties were some of the best unforgettable memories.

My friends Anamika, Priyanka, Shreya, Sridevi, and Sushma; you were always there by my side in sound as well as testing times; you were the most vital pillar. Your non-judgemental support helped me let out all my insecurities. The long meaningless chats during the mess hours, the parties, the night-outs, and the Goa trip are the moments that I will carry along for life. Thank you, my girl gang.

I want to thank the most important people, without whose support this journey would have been impossible, my father, Mahendrakumar, my mother, Ranjan, and my sister, Sweety. They always thought that I could achieve anything in life, and there is nobody better in the entire world, which is weird but somehow kept the light in me going. Their belief, love, and support have been the most important reasons for my success. I want to thank my brother-in-law, Nirav, who used his great sense of humor to motivate me towards success and supported me during lows. This journey would be incomplete if it were not with my nephew Parth; he was around three years old when I started my journey. You always bring a smile to my face, even during times of self-doubt and disappointment. Your innocence has unknowingly taught me life's best lessons.

The past seven years have been the most enriching phase of my life, and I look forward to the beginning of the next chapter of my life.

Rinku Mahendrakumar Shah
IIT Bombay
9 February 2021