A High Performance Packet Core for Next Generation Cellular Networks

Zafar Ayyub Qazi
UC Berkeley

Melvin Walls
Nefeli Networks, Inc.

Aurojit Panda
UC Berkeley

Vyas Sekar
Carnegie Mellon University

Sylvia Ratnasamy
UC Berkeley

Scott Shenker
UC Berkeley

ABSTRACT

Cellular traffic continues to grow rapidly making the scalability of the cellular infrastructure a critical issue. However, there is mounting evidence that the current Evolved Packet Core (EPC) is ill-suited to meet these scaling demands: EPC solutions based on specialized appliances are expensive to scale and recent software EPCs perform poorly, particularly with increasing numbers of devices or signaling traffic.

In this paper, we design and evaluate a new system architecture for a software EPC that achieves high and scalable performance. We postulate that the poor scaling of existing EPC systems stems from the manner in which the system is decomposed which leads to device state being duplicated across multiple components which in turn results in frequent interactions between the different components. We propose an alternate approach in which state for a single device is consolidated in one location and EPC functions are (re)organized for efficient access to this consolidated state. In effect, our design "slices" the EPC by user.

We prototype and evaluate *PEPC*, a software EPC that implements the key components of our design. We show that PEPC achieves 3-7× higher throughput than comparable software EPCs that have been implemented in industry and over 10× higher throughput than a popular open-source implementation (OpenAirInterface). Compared to the industrial EPC implementations, PEPC sustains high data throughput for 10-100× more users devices per core, and a 10× higher ratio of signaling-to-data traffic. In addition to high performance, PEPC's by-user organization enables efficient state migration and customization of processing pipelines. We implement user migration in PEPC and show that state can be migrated with little disruption, e.g., migration adds only up to 4μs of latency to median per packet latencies.

CCS CONCEPTS

• **Networks** → *Network components*; *Middle boxes / network appliances*;

KEYWORDS

Cellular Networks, EPC, Network Function

ACM Reference format:
Zafar Ayyub Qazi, Melvin Walls, Aurojit Panda, Vyas Sekar, Sylvia Ratnasamy, and Scott Shenker. 2017. A High Performance Packet Core for Next Generation Cellular Networks. In *Proceedings of SIGCOMM '17, Los Angeles, CA, USA, August 21-25, 2017,* 14 pages.
https://doi.org/10.1145/3098822.3098848

SIGCOMM '17, August 21-25, 2017, Los Angeles, CA, USA

ACM ISBN 978-1-4503-4653-5/17/08…$15.00
https://doi.org/10.1145/3098822.3098848

1 INTRODUCTION

Cellular networks are experiencing explosive growth along multiple dimensions: (i) traffic volumes (e.g., mobile traffic grew by 74% in 2015), (ii) the number and diversity of connected devices (e.g., projections show that by 2020 there will be 11.6 billion mobile connected devices including approximately 3 billion IoT devices [11]), and (iii) signaling traffic (e.g., signaling traffic in the cellular network is reported to be growing 50% faster than data traffic [31]).

These trends impose significant *scaling* challenges on the cellular infrastructure. In particular, there is growing concern regarding the scalability of the cellular evolved packet core (EPC) [21] infrastructure. The EPC is the portion of the network that connects the base stations to the IP backbone and implements cellular-specific processing on user's data and signaling traffic. Recent industrial and academic studies have provided mounting anecdotal and empirical evidence showing that existing EPC implementations cannot keep up with the projected growth in cellular traffic [10, 18, 19, 23, 37].

We postulate that the poor scaling of existing solutions stems from the manner in which existing EPC systems have been decomposed. More specifically, EPC systems today are factored based on traffic type, with different components to handle signaling and data traffic: the Mobility Management Entity (MME) handles signaling traffic from mobile devices and base stations, while the Serving and Packet Gateways (S-GW and P-GW) handle data traffic. The problem with this factoring is it complicates how *state* is decomposed and managed. As we elaborate on in §2, current designs lead to three problems related to state management:

(1) Duplicated state leads to frequent synchronization across components. In current EPCs, per-user state is often duplicated between components. For example, a user request to establish a cellular connection is processed by the MME which instantiates user state and then communicates this addition to the S-GW which in turn locally instantiates similar per-user state. A similar interaction takes place when user state is updated after mobility events. This duplication introduces complexity (e.g., implementing the protocols

between components) and the performance overhead associated with cross-component syncronization.

(2) Migration is hard. As we explain in §2, migrating a user's state to a new MME/S-GW/P-GW instance is often desirable to allow elastic scaling of these components and/or to enable performance optimizations [35]. However with the existing decomposition, migrating a user's state is hard since it requires coordination across multiple components (e.g., updating multiple logical tunnels) [1].

(3) Customizing/optimizing the processing pipeline is hard. Current EPCs implement a fixed processing pipeline for all data packets. However, as we show, there is a significant opportunity to increase efficiency by customizing the EPC processing pipeline based on the nature of the user/device. For example, a stationary IoT thermostat can omit much of the complex processing required by devices like smart phones. However, with the current EPC decomposition, enabling these customizations can be hard when the processing pipeline is spread across different components (§2).

The above state of affairs is in stark contrast to many Internet services that enjoy (relatively) simple and efficient elastic scaling. We observe that many of these services achieve elasticity by pursuing a "share nothing" decomposition in which components that are to be scaled-out are independent, with little/no shared state across them. A natural question then is whether there exists an analogous decomposition for EPC processing that enables simpler horizontal scaling.

To answer the above question, we examine the various forms of EPC state and corresponding processing logic and propose an alternate *state-driven* decomposition of an EPC system. Specifically, we propose that the state associated with a user be consolidated into a single location which we call a slice. Hence, in contrast to current EPC designs, the signaling *and* data traffic for a user are now processed by the same slice. To maintain performance isolation between data and signaling traffic, we assign each to separate cores.

To mitigate the overheads of contention over shared state, we decompose EPC state and functions such that any given piece of state has only a single writer associated with it. Based on this refactoring, we propose PEPC,[1] a new system design for an EPC. As will be clear in §3, the refactoring in PEPC goes beyond simply consolidating existing components in a single process with a 1:1 mapping between MME (S/P-GW) operations and PEPC's control (data) thread.

We implement PEPC using the NetBricks [33] framework, which provides cheaper isolation as compared to VMs/containers. NetBricks allows us to run multiple PEPC slices within the same process, while providing memory isolation between the slices. Hence we avoid the higher overheads associated with implementing a slice as a VM or container. This in turn facilitates vertical slicing and scaling by users.

In summary, this paper makes the following contributions:

- We propose PEPC, a new system design for an EPC that is based on consolidating per-device state in a single location and (re)organizing EPC functions for efficient access to this consolidated state (§3).
- We implement key EPC functions in PEPC, including the data-plane functions such as the GPRS Tunnelling Protocol and Policy Charging Enforcement Function (PCEF), as well as the S1AP protocol used to exchange control traffic with the base station (§4).
- We conduct extensive evaluation of PEPC. We show that PEPC achieves 3-7× higher throughput than existing industrial DPDK-based EPCs and over 10x higher throughput than OpenAirInterface [32] (§5). In addition, PEPC's performance scales linearly with the number of cores and can sustain performance with a large number of devices and high signaling to data ratio (§6).
- We implement state migration in PEPC and show that PEPC can handle 10,000 state migrations/second with only a 5% drop in the data plane throughput and in the worst case increases per-packet latency by 4μs (§6.6).
- We implement customized pipelines for IoT devices and measure the resultant performance improvement for different traffic mixes (§7.4).

The remainder of this paper is organized as follows: we first review the EPC architecture and its scaling problems (§2) and then present the design and implementation of PEPC (§3 and §4 respectively). We evaluate PEPC's performance (§5-§7), discuss related work (§9) and finally conclude.

2 BACKGROUND AND MOTIVATION

We begin by briefly reviewing the existing EPC architecture. We then highlight key scalability issues using experimental results from two industrial software EPC implementations. Finally we deconstruct the current EPC architecture to better understand the root cause of its limited scalability.

2.1 EPC background

The LTE cellular network architecture consists of two main components: the Radio Access Network (RAN), and the Evolved Packet Core (EPC). The RAN consists of the eNodeBs (i.e., base stations), which communicate with the User Equipment (UE)[2] through a radio interface. The traffic from the eNodeBs is forwarded to the EPC. The EPC then forwards traffic to the final destination, e.g., a server on the Internet, or a cellular provider's IP Multimedia System (IMS) in the case of voice traffic.

The EPC consists of a set of cellular-specific functions as well as traditional middleboxes, like NATs and firewalls. There are three key cellular specific functions. The Mobility Management Entity (MME) handles all the signaling traffic from the User Equipments (UEs) and the eNodeBs, and is responsible for user authentication, mobility management and session management. The Serving Gateway (S-GW) and Packet Gateway process all the data traffic. The S-GW handles all the data traffic from the eNodeBs and forwards it to the Packet Gateway (P-GW) which may then send it out to another middlebox or egress node. Besides packet forwarding the S/P-GWs also perform other functions including maintaining statistics for charging and accounting, QoS enforcement, IP address allocation, and policy enforcement.

[1] PEPC stands for Performant EPC, pronounced as pepsi.

[2] We use UE, user and device interchangeably in the rest of the paper.

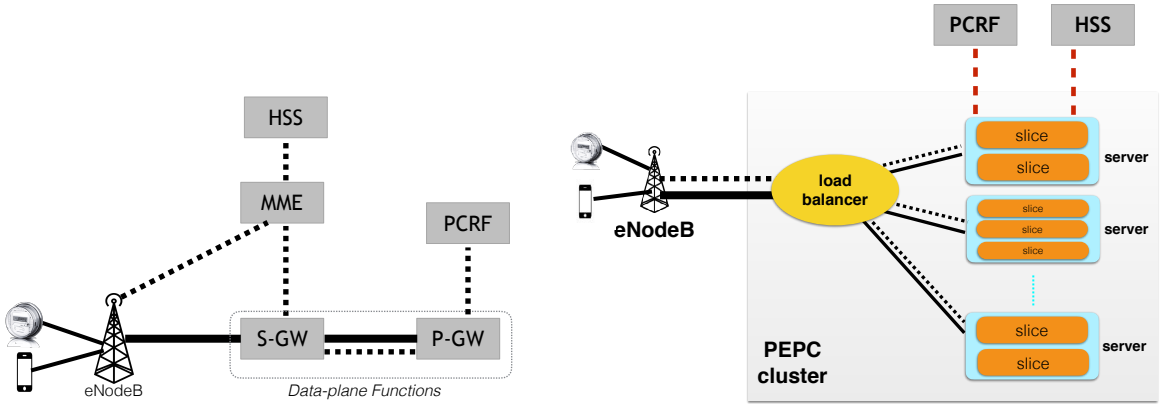

Figure 1: (a) Current EPC architecture (left) (b) PEPC overall architecture (right)

State type	MME	S-GW	P-GW	PEPC control thread	PEPC data thread	Update frequency
User location	w+r	w+r	NA	w+r	r	per-event
User id	w+r	w+r	w+r	w+r	r	per-event
Per-user QoS/policy state	w+r	w+r	w+r	w+r	r	per-event
Per-user control tunnel state	w+r	w+r	w+r	NA	NA	per-event
Per-user data tunnel state	w+r	w+r	w+r	w+r	r	per-event
Per-user bandwidth counters	NA	w+r	w+r	r	w+r	per-packet

Table 1: State taxonomy for current EPC functions (§2.3) and PEPC (§3).

Besides these core functions, the EPC also contains a Home Subscriber Database (HSS) containing subscriber information such as their billing plan and a Policy Charging Rules Function (PCRF) which installs new charging and policy control rules in P-GW.

Figure 1(a) shows the above EPC components and how they interact, as described in the 3rd Generation Partnership Project (3GPP) [4]. The dotted lines represents signaling traffic while the solid lines represent data traffic.

All signaling traffic between the UE/eNodeB and the EPC is sent over the S1-MME interface to the MME using the S1AP protocol [4]. When a user-device first connects to the eNodeB, the eNodeB sends a signaling event ('attach request') to the MME. This triggers a series of signaling messages between different EPC functions: e.g., the MME queries the HSS for the device's credentials, it then authenticates the device, once authenticated it sends device-state to the S-GW, which in turn sends the device state to the P-GW. The S-GW receives control traffic from the MME over the S11 interface using the GPRS tunneling (GTP) protocol, specifically GTP-C. The S-GW communicates control messages to the P-GW on the S5 interface using GTP-C.

Once the UE is authenticated, it can start sending data over the S1-U interface to the S-GW via the eNodeB. The eNodeB tunnels the UE data-packets using GTP-U, where the eNodeB performs an IP-in-IP encapsulation, adds a GTP-U header to the data-packets, and sends it to the S-GW. The S-GW similarly performs GTP-U decapsulation, and again encapsulation before sending the packet out to the P-GW over another GTP-U tunnel. In the current LTE architecture, tunneling is used to support mobility, QoS, and traffic aggregation.

2.2 Scaling issues in existing EPC systems

In this section, we highlight some key scalability issues with the current EPC design using experimental results from two industrial software EPC implementations. We treat these systems as black-boxes in this section and describe them in more detail in §5.

1. **Poor scalability with increasing signaling:** The data-plane performance can be severely impacted by the number of signaling events. In one industrial EPC implementation that we tested, the performance drops to almost 0 pps as we increase the number of attach requests per second to more than 10 K (§5). A different EPC implementation that was evaluated in prior work [37] reports a 15% drop in data-plane performance as the number of S1-based handover events[3] per second are increased to 3000. As we explain shortly, both attach requests and S1-based handover events lead to complex synchronization between the MME, S-GW and P-GW. For example, measurements in the OpenEPC system [6] revealed that the time between the MME sending an update to the data-plane following an attach event, to when the corresponding user-state has been updated at the S/P-GW is as high as 2-3 milliseconds. As real-world signaling traffic is growing 50% faster than data traffic [31], this presents a significant scalability issue, with several high-profile outages being associated with signaling [10].

2. **Poor scalability with increasing devices:** Throughput on the data plane also scales poorly with increasing numbers of devices. With one industrial EPC implementation, the performance

[3] S1-based handover is triggered when a device moves between two eNodeBs which are not directly connected.

drops by 50% from 128K to 300K users [37]. Similarly, with another EPC industrial implementation the performance drops from 1 Mpps with 250 K users to 0.1 Mpps with 1M users (§5). With increasing numbers of devices, the number of signaling events grows. In addition, performance also drops because of the increased overhead of state lookups: EPC components maintain per-user state and the data plane components access this state on a per-packet basis. As the number of devices is increasing rapidly, with projections of 11.6 billion mobile devices including around 3 billion IoT device by 2020, this presents another major scalability problem.

Next, we discuss why these issues exist.

2.3 Deconstructing EPC state

One of the key reasons why multiple EPC components need frequent synchronization is because they maintain per user state which needs to be synchronized for most of the signaling events. In order to understand how this can lead to the above scalability issues, we deconstruct the state maintained by the network functions in EPC and the type and frequency of their state operations. To perform this exercise, we draw from the 3GPP design specifications [3], recent work on EPC state analysis [20], and an open source EPC implementation [32].

2.3.1 Types of state. Below, we describe the user state variables in EPC and group them into the following categories:

- **User identifier:** These include subscriber identifiers such as IMSI, GUTI (a temporary identifier used instead of IMSI over the radio link), and user IP addresses.
- **User location:** This group contains variables that store the cell-level location information of the UE (ECGI) as well as bounds on where a UE can go without reporting its location back to the network (TAI, Tracking Area List, etc.).
- **Per-user control tunnel state:** This contains variables that maintain the identifiers and state for user-specific control tunnels related to the S11 and S5/S8 interfaces. These include tunnel end point identifiers (TEIDs) and UE IP addresses.
- **Per-user data tunnel state:** This contains variables that maintain the identifiers and state for user-specific data tunnels, related to S1U and S5/S8 interfaces. These include tunnel end point identifiers (TEIDs) and UE IP addresses.
- **Per-user QoS/policy state:** This group includes QoS and policy parameters, such as Guaranteed Bit Rate (GBR), Maximum Bit Rate (MBR), and Traffic Flow Templates (TFT). These parameters are per bearer, where a bearer is a logical connection between two EPC components, e.g, between eNodeB and S-GW. Each UE has one or more bearers associated with it.
- **Per-user bandwidth counters:** This group includes various counters that track a user's bandwidth usage.

2.3.2 Types of state accesses and implications. Table 1 describes how state variables are accessed by different EPC functions and the frequency at which these state variables are updated. We note that the MME, S-GW, and P-GW, all maintain state with the per-user QoS/policy state, per-user data/control tunnel state as well as user ids. This duplicated state leads to frequent synchronization: signaling events result in updates to this state and these changes must be propagated to the S-GW and P-GW. For instance, for an attach signaling event, the MME, S-GW and P-GW must all update their per-user QoS/policy state. Similarly for a S1-handover event which happens when a UE moves from one base to another base station (and the base stations are not directly connected), all components update per-user data tunnel state. Similarly for a modify-bearer event, all the components may update some parameters related to QoS/policy state. The key insight here is that some state variables are replicated at the MME, S-GW, and P-GW and may need to be updated for every signaling event of a certain type.

The duplication of state found in current EPC designs leads to synchronization between components as a result of signaling events. As the number of signaling events increases, this frequent synchronization can impact the performance of the data-handling components. In addition, with the existing decomposition, migrating a user's state to a new MME/S-GW/P-GW is hard since it requires coordination across multiple components.

Fixed processing: Current EPCs implement a fixed processing pipeline for all data packets, even though many devices (e.g., IoT devices) have very different packet processing requirements [41] from traditional smart phones. The data-plane functions, S-GW and P-GW, maintain the same state and processing logic for all types of devices. We show (§3 and §5) that for many IoT devices, packet processing can be optimized through customization (e.g., reduce state, avoid state lookups) to allow EPC to scale well with the number of devices.

3 PEPC DESIGN

In PEPC, we rearchitect the implementation of the MME, S-GW, and P-GW components of the current EPC architecture while leaving unchanged the design of base stations (eNodeB) and the PCRF/HSS components. The PEPC implementation consists of software that runs on a cluster of commodity servers as shown in Figure 1; we use the term PEPC *node* to refer to a server running our PEPC software. As is common with cluster-based services, we assume that the PEPC cluster is abstracted by a single virtual IP address [17]; external components such as the eNodeB direct their traffic to this virtual IP address and the cluster's load balancer takes care of appropriately demultiplexing user traffic across the PEPC nodes (§3.4).

In this section, we first discuss our goals in designing PEPC and the overall approach they lead to (§3.1). We then present our two key system components: the PEPC slice (§3.2) and node (§3.3). Finally, we discuss how PEPC operates in the end-to-end cellular architecture (§3.4).

3.1 Design goals and approach

The primary goal driving the design of PEPC is **high performance**. A PEPC node must sustain high throughput with low per-packet latencies. In addition, performance must **scale** well with the number of devices and the volume of signaling traffic. We meet these performance goals in two steps. First, we develop a new system architecture for EPC processing that is based on *consolidating* device state and *refactoring* EPC functions for efficient access to this state. This change gives us a baseline design that achieves high per-node

throughput with good scaling behavior; e.g., our results in §5 show the even with 10M users, PEPC can achieve a data plane throughput of 5 Mpps. Next, leveraging our new system architecture, we introduce a few key optimizations that exploit common characteristics of EPC workloads to improve PEPC's baseline performance by between 2-50% (§7). In what follows, we elaborate on the key ideas behind each step.

A new system architecture for EPC processing. PEPC's system architecture is based on two key guidelines: (1) *consolidate* per-device state in one location, and (2) *decompose* existing EPC functions for efficient access to this consolidated state.

Consolidation. As mentioned earlier in §2, in current EPC systems, device state is often duplicated at the MME, S-GW and P-GW. PEPC instead consolidates state in one location which we call a PEPC *slice*. A slice lives entirely within a single server; a server can host multiple slices and multiple user devices can be processed by a single slice. State consolidation removes the inefficiencies that result from duplicating state across multiple EPC components and hence improves performance. As we shall see, consolidation also simplifies user migration to enable further performance optimizations.

Decomposition. We decompose EPC functions based on the nature of their accesses to per-device state. Consolidating state means that all traffic that accesses device state must be processed by the same slice; this in turn implies that signaling and data traffic for a device is processed by the same slice. However, if done naively, this could violate the performance isolation that EPC implementations typically maintain between signaling and data traffic. Cellular providers typically support various QoS options on data traffic such as guaranteed bit rates on voice traffic, maximum bit rates for total user traffic, or priority forwarding based on the application type [46]. Supporting these QoS options is challenging if signaling and data traffic are processed on the same core because the time to process signaling events can be large and highly variable [30].[4]

Hence, to ensure performance isolation, we implement two types of threads within each slice: (1) control threads that process signaling traffic and (2) data threads that process data traffic. Data and control threads are assigned to separate cores for performance isolation.[5] With this arrangement, per-device state is shared between the two types of threads. Again, naively done this could lead to sub-optimal performance due to cross-core contention as the threads access shared state. We avoid this by partitioning device state and EPC functions such that each piece of state has only a single writer; i.e., either the control or the data thread (but not both) can write a piece of state but both control and data threads can read all per-device state. Thus, our slice architecture provides both performance isolation between data and signaling traffic as well as high performance (since it avoids contention over writing to shared state).

[4]One might consider preemptive scheduling but this typically hurts throughput and hence we aim for a run-to-completion model as is common in packet-processing applications [12, 25, 33].

[5]As the cores on the same socket will share the LLC cache, there still can be some possible contention.

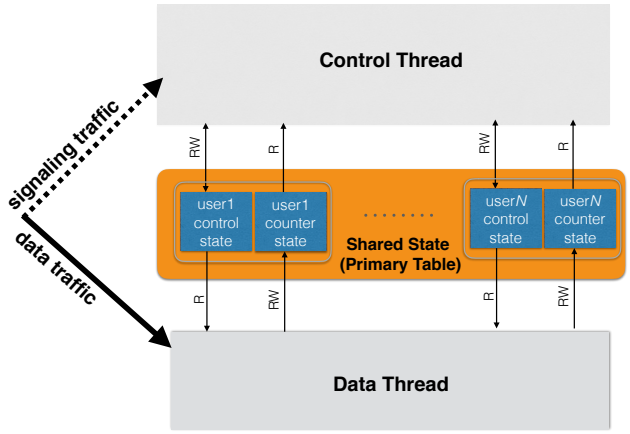

Figure 2: Internal view of a PEPC slice. (The secondary state table is not shown for simplicity.)

Performance optimizations. With PEPC's system architecture, all state and processing for a device is now localized within a single slice. This greatly simplifies our ability to support two important features: *migrating* a user's processing across PEPC nodes and *customization* of a device's processing pipeline. Migration and customization enable a range of performance optimizations. For example, migration enables elastic scaling for efficient use of resources, or offloading a user's processing to reduce end-to-end application latency as proposed by [35]. Similarly, the ability to easily customize the processing pipeline on a per-device basis creates many opportunities to streamline processing. For example, unlike smart phones which generate traffic from diverse applications, many IoT devices only use a single application [41]. However, all the data traffic in EPC functions is run through a classifier that aims to identify different applications so that they can be subject to different QoS processing. The per user state on the data plane functions serves this purpose of mapping incoming traffic to a QoS class. For IoT devices that run a single application, and require a default best effort service, we bypass this state lookup, allowing the data performance to scale better.

Resultant architecture. Figure 1 shows the overall PEPC architecture. The eNodeB still speaks the same S1AP [4] protocol, directing both signaling and data traffic to the PEPC cluster where a load balancer transparently directs traffic to the appropriate PEPC node. Each node in turn can run several PEPC slices and the node internally steers traffic to the appropriate slice (§3.3). As we describe in §3.3, each PEPC server also runs a proxy that communicates with backend components like the PCRF and HSS using existing protocols.

3.2 PEPC Slices

Figure 2 shows the internal view of a PEPC slice. Below we describe the key pieces of a slice.

Shared state with fine-grained locks: In PEPC, shared state is partitioned at two levels. First, state is partitioned by user. Second, per-user state is partitioned based on whether it is written by the control or data thread. State that is written by the control thread

is listed in Table 1 and includes state related to the QoS policies associated with the user (e.g., various rate limits, priority levels, and filters), the data tunnels associated with the user, and so forth. Such state is only updated by signaling and other control events handled by the control thread; processing data packets may require reading this state (e.g., to enforce rate limits) but does not write to such state.[6] We refer to this state as a user's *control state*.

State that is written by the data thread consists of various counters that track a user's bandwidth consumption and are used for charging and QoS-related functions. Such state is updated when processing data packets and may be read by the control thread to implement functions related to accounting and charging (e.g., to be communicated back to the PCRF). We refer to such state as a user's *counter state*.

We use fine-grained locks for efficient access to shared state as shown in Figure 2. A control thread holds a read/write lock for each user's control state and a read-only lock for each user's counter state. Similarly, a data thread holds a read/write lock to per-user counter state but a read-only lock to per-user control state. In §7, we show that the use of fine-grained locks offers performance improvement of upto 5 times over designs that use coarse-grained locks over all user state and about 5% when we allow the control and data threads to contend in writing to per-user state.

Primary vs. Secondary tables for storing user state: Many current EPC implementations store all user state in a single table [37]. As the number of user devices grows, this table is poorly contained by the CPU cache and hence performance drops. In PEPC, we aim to alleviate this effect by maintaining two tables – a primary and secondary table. State for active devices is stored in the primary table and accessed by both the control and data threads. Once a device is no longer active (as determined by a timeout or an explicit signaling event), its state is moved from the primary to the secondary table. Similarly, when an incoming packet's user state is not found in the primary table, the state is located in the secondary table and moved back to the primary one. Moving user state between the primary and secondary tables is handled by the slice control thread as detailed in §4.

This two-level architecture exploits the fact that the active time for many users is often relatively short. This is particularly true for many classes of IoT devices [41] and can yield significantly improved scalability. For example, as we show in §7.3, two-level storage can improve performance by upto 29% depending on the number of devices that are always on and level of churn in the remaining devices.

PEPC control threads: A PEPC control thread processes the signaling traffic for users associated with its slice. Thus functions typically implemented in the MME are implemented in the control thread. In addition, the control thread implements certain functions currently implemented in the P-GW related to interactions with the PCRF. These include accepting updates to the user's charging/accounting rules from the PCRF (this involves writing to the user's control state) as well as communicating usage statistics back to the PCRF (this involves reading the user's counter state). Finally, the control plane also handles moving user state between the primary and secondary tables as well as handling any state migration requests as described in §4. The control thread is also responsible for reserving resources for a user[7] (e.g., bandwidth and CPU). When migrating user processing from one slice to another the resources associated with the user are released from the old slice to ensure efficient utilization.

[6] In the current EPC architecture, the S-GW and P-GW do write to this control state but this is only because these components maintain a duplicate copy of per-user state and hence they must update the state to reflect writes made at the MME. In PEPC, since state is shared, updates by the control thread are immediately available to the data thread.

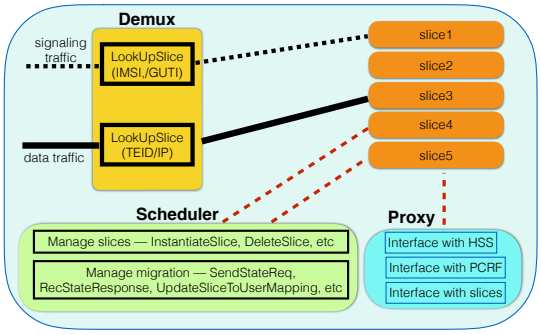

Figure 3: A PEPC server.

PEPC data threads: A PEPC data thread executes the processing pipeline for its users' data traffic. This includes many of the functions currently found in the S-GW and P-GW with certain exceptions as described above. We describe the functions in a typical pipeline as well as the customizations that we implement for IoT devices in §4.

3.3 PEPC Nodes

Figure 3 illustrates the internals of a PEPC node. Each node may run several independent PEPC slices. There are three key components that sit outside the PEPC slices, and inside a PEPC node. We describe these below:

Demux: PEPC's Demux function is responsible for steering incoming signaling and data traffic to its associated slice. For this, the Demux component maintains a mapping from a user to a slice. For signaling traffic, the Demux function uses the IMSI or GUTI (each signaling request contains either one of these user identifiers) to look up the slice corresponding to incoming signaling traffic. And it uses the TEID (for uplink) or user device IP address (for downlink) to map incoming traffic to a specific slice.

Scheduler: The PEPC node scheduler is responsible for (i) managing slices (e.g., instantiating slices, assigning hardware resources to slices with dedicated cores for a slice's data and control threads, and so forth), and (ii) managing migration (e.g., receiving state migration requests from an external controller, initiating state transfers from slices, etc.). The PEPC node scheduler can also reconfigure user to slice mappings.

Proxy: The PEPC node proxy interfaces with the backend servers like HSS and PCRF. Specifically, the interface between the HSS and Proxy is the same as the current interface between the MME and HSS. This interface is currently referred to as S6A and usually runs the Diameter protocol [4]. Similarly the interface between the

[7] More specifically a bearer associated with a user device.

proxy and PCRF is the same as the current interface between the P-GW and PCRF. This interface is referred to as Gx [4].

3.4 End-to-end Architecture

We elaborate on how PEPC interacts with other components in the overall architecture by elaborating on its interface with the base station (eNodeB) and walk through how two of the key signaling events – "attach request" and "mobility" – are handled with PEPC.

Interface to the eNodeB: PEPC assumes that eNodeBs remain unmodified; i.e., a eNodeB uses the S1AP protocol over the S1-MME interface to connect to the PEPC. Currently, when a user entity (UE) attaches to an eNodeB, the eNodeB perform a DNS lookup to identify an MME to which it should forward the attach request. In PEPC the DNS lookup instead resolves to the virtual IP address associated with a PEPC cluster; traffic thus reaches the cluster's load balancer which transparently steers the incoming user traffic to a specific PEPC node (Figure 1). Such load-balancers may be implemented as specialized hardware appliances [14], as part of the top-of-rack switch [8], or as a software service [5, 13, 17, 33]; all of these options are commonly available today. Finally, the Demux function maps the traffic to a specific PEPC slice. For communication between two cellular users, a packet will traverse the PEPC slice for both source and destination in exactly the same manner as today it would traverse through the S-GW/P-GW/MME for both endpoints.

Attach request: The user initiates the attach procedure by sending an attach request message containing its IMSI identifier. The eNodeB then performs a DNS lookup to find the PEPC cluster that the user traffic is to be routed to. As describe above, the traffic is eventually routed to an individual PEPC slice where the PEPC slice's control thread uses the proxy to query the HSS about the user. If the user is a valid subscriber, the control thread thus authenticates the user device. Once authenticated, the control thread inserts the relevant user QoS/policy and tunnel state in the per-user control state and shares a read-only reference to the same with the data thread. The PEPC control thread also updates the user location information with the cells to which the user device can contact.

Mobility handling: We consider here only the case where a user device is moving within a network managed by the same operator (there are other cases of mobility; e.g., roaming). In this case, there can be two further scenarios: a) the eNodeBs that serve the user device during the handover are connected to each other, or b) the eNodeBs are not connected to each other. In both cases, the PEPC slice's control thread may need to update the location information of the user device (e.g., the cell that the user is connected to), and data plane tunnel state (i.e., the new eNodeB's TEID and IP address). The control thread then updates this information in the shared user state which the data thread can then read when sending/receiving packets to/from the eNodeB.

3.5 PEPC vs. existing EPC design tradeoffs

We discuss a number of potential trade-offs between PEPC and the legacy EPC design.

The existing EPC architecture allows the MME and S/P-GWs to be separately deployed (possibly in different data centers) [35]. However, because of the dependencies between these functions in the current EPC design, scaling one function invariably requires scaling the other due to the need for frequent synchronization. PEPC addresses this challenge by consolidating user state and minimizing the impact that the control plane and data plane have on each other.

In existing EPC designs, the S-GW can be selected based on its geographical proximity to the user, without necessarily changing the MME. However, changing the S-GW requires a handover procedure [1], which requires coordination with MME and P-GW. On the other hand, in PEPC, moving processing closer to the user entails a simplified migration procedure (§5) which leads to traffic being processed by a PEPC slice closer to the UE.

Current EPC designs offer carriers the flexibility to mix and match EPC functions from different vendors. PEPC as implemented right now does not allow mix and match.

4 IMPLEMENTATION

In this section we describe the implementation of the individual components of PEPC. We begin by briefly describing NetBricks [33], the underlying framework we use to program and run the EPC functions, and discuss why and how we use it.

4.1 Background on NetBricks

The NetBricks framework provides a set of customizable network processing elements for writing network functions and an execution environment which provides the same memory isolation as VMs or containers, without incurring the same performance penalties. We use NetBricks for PEPC because it allows us to run multiple PEPC slices within the same process, instead of running these inside VMs or containers. Compared to VMs and containers, our PEPC binaries in NetBricks are lightweight, with a size of approximately 5MB. Any single PEPC binary can have many slices, and each reuses common code modules[8].

NetBricks uses the Rust programming language [38], and PEPC is also written in rust, and runs inside NetBricks execution environment.

4.2 PEPC slice

Listing 1 describes the implementation of a slice in PEPC. It consists of a set of configurations, containing policy rules, charging filters, etc. Each PEPC slice consist of its own control and data plane. The PEPC control and data plane threads are pinned to separate cores. A slice also maintains state tables corresponding to the data and control thread. The shared state is implemented through Read/Write locks per user state. The PEPC slice scheduler transmits the outbound packets, in accordance with the QoS requirements of the traffic. A slice also maintains a migration channel with the PEPC node/server scheduler, allowing to receive state transfer requests and send state transfer responses.

Below we describe the data plane functions, control plane functions and customizations we implement. We also describe how we implement the two level cache.

[8]We note that this is the size of the binary with our existing implementation and protocol support. This does not include support for interacting with backend servers such as HSS (S6 interface) and PCRF (Gx interface).

Listing 1 A slice in PEPC.

```
struct Slice {
    pub config: EpcConfig,
    ctrl_core: i32,
    data_core: i32,
    dp_state: HashMap<id,RwLock<UeContext>>,
    cp_state: HashMap<id,RwLock<UeContext>>,
    pub sched: Scheduler,
    // State migration
    from_node_sched: Receiver<StateTransferMessage>,
    to_node_sched: SyncSender<StateTransferMessage>,
    ...
}
```

Slice data plane: Our data path consists of a chain of network functions programmed using NetBricks. We currently implement GTP-U encapsulation and decapsulation, user state look-up which involves mapping downlink traffic to the appropriate GTP-U tunnel. We also implement the Policy Charging and Enforcement Function (PCEF), as a match-action table, consisting of BPF programs over the 5-tuple and operator specified actions. Our PEPC implementation can handle data-traffic from existing base-stations, and our evaluation uses traces from a open source base-station emulator [32].

Slice control plane: Our current slice control plane implementation emulates state operations corresponding to two types of control events; attach request and S1-based handovers. In an attach event, state is allocated for a new user device, the control plane inserts this into its local state table, and then notifies the data plane. S1-based handovers require modification of specific elements of the user state, specifically eNodeB tunnel identifier, used to identify a GTP-U tunnel with the base-station, and the IP address of the new base-station. In addition, we have built support for S1AP protocol for parsing request messages and sending response messages from/to the UE/eNodeB on the S1-MME interface. We also have support for handling NAS messages [2], which is used to convey non-radio signalling between the UE and the MME, and sits on top of the S1AP in the protocol stack. We presently only have support for handling the attach procedure over S1AP/NAS.

Customization: We implement customized processing for a class of IoT devices running a single application, with default best effort service. We refer to these as *Stateless IoT devices*. For these devices, the data plane avoids the state lookups, only applies policy and charging rules. For these devices PEPC assigns the tunnel endpoint identifiers (TEIDs) and IP addresses from a pre-assigned pool, and this information is then used to infer the service required by these devices. In PEPC, an operator can assign these devices to their own independent slice with customized processing.

Two-level state storage: PEPC slice control-plane has a two-level state storage architecture: a small primary storage containing state for active devices and a secondary state table for all devices. A device in primary storage may be evicted after being idle for some specified duration by the control plane. In the case the state for a device does not exist in the data plane, it can query the control plane to retrieve the reference of the state from the secondary state table.

4.3 PEPC Node

We describe how we implement the Demux function and PEPC node scheduler.

Demux: PEPC implements a *LookUpSlice* (Figure 3) function that reads the TEID/IP from the data packets and searches a mapping table to assign a slice. The Demux functions currently does not implement the dynamic mapping of signaling traffic to a slice.

Scheduler: The PEPC node scheduler currently implements the following functions, (i) it instantiates PEPC slices based on a given operator configuration, and (ii) it handles state migrations. The PEPC scheduler currently implements migration within a PEPC node, i.e., migrating user state between two slices within the same server. For state migrations, the PEPC scheduler is responsible for sending state transfer messages to a specific slice, and receiving state transfer responses. To ensure state migrations do not lead to packet losses or any inconsistent updates to user state, the PEPC scheduler buffers the packets which are undergoing migration. Specifically, it implements per-user migration queues, which are drained once a user state is migrated. It then sends these packets to the new slice. We discuss the feasibility of state migrations in PEPC in §6.6

5 EVALUATION SETUP

In this section, we describe our evaluation setup, baselines and parameters.

5.1 Testbed

Our test environment has two servers running Ubuntu 14.04 with Linux kernel 4.4.0, each with two sockets populated with 22 core Intel Xeon E5-2699 v4 CPUs clocked at 2.2GHz (hyperthreading disabled) and 64 GB of RAM, split evenly between each socket. Both servers are also equipped with an Intel X710 40 Gb (4 x 10) NIC. Both the servers are patched directly to each other using all available ports. The server running PEPC uses NetBricks, which uses DPDK 16.04 and the nightly build of Rust.

We setup one server to generate traffic, while the other server runs PEPC. In the uplink direction, to emulate sending data-traffic from the eNodeB, we use traces collected using `OpenAirInterface` [32], which provides a user device and base-station emulator. The data-traffic in these traces is encapsulated using GTP-U, and is replayed by the traffic generator. In the downlink direction, EPC functions receive normal IP traffic.

For emulating a large number of user device tunnels, we synthetically modify TEID before the traffic is processed by PEPC.

For testing signaling traffic we divide our experiments into two categories. In the first set of experiments, we test with an implementation of the S1AP and NAS protocol, and implement the handling of request and response messages between the UE/eNodeB and EPC for an attach request procedure. These experiments are run with real signaling traces from a commercial traffic generator and RAN emulator from ng4T [28].[9] In these traces SCTP is used as the underlying transport protocol,[10] and S1AP and NAS protocols are implemented. However, because of lack support for handling SCTP in NetBricks, we use kernel-based SCTP implementation which impacts how much we can scale the handling of signaling events.

[9] Some of the NG4T signaling traces are available online [29].
[10] SCTP is the 3GPP recommended underlying transport protocol for S1AP protocol and also used in the signaling traces from ng4T.

The second set of experiments are aimed at testing with signaling traffic at scale. In these experiments, we synthetically generate only control updates corresponding to attach requests and S1-based handovers. When a attach event is received, the user device creates the appropriate user device state, and adds it to state table. In the event of a S1-based handover, the device state gets modified, including the IP address of the eNodeB it is connected to, and TEID. In our evaluation experiments, the control updates are uniformly distributed across the number of user devices. However, no signaling messages are generated over the wire.

In the evaluation results that follow up in §6 and §7, the Figure 4, 5, 8, 10 correspond to the first set of experiments. And the rest of the results correspond to the second set of experiments.

5.2 Baselines

We compare the scalability benefits of PEPC, against the following EPC implementations.

1. An industrial software EPC implementation developed in collaboration between carriers and our industrial partners for deployment. We refer to this system as `Industrial#1`.[11] It uses DPDK to bypass the kernel for fast I/O, and its data plane includes support for GTP and Application Detection and Control (ADC). In PEPC, we implement the same functions along with the Policy and Charging Enforcement Function (PCEF). In comparing the test results of `Industrial#1` with PEPC, we use the same workloads for testing, including traffic mix (uplink to downlink traffic ratio) and signaling event rate (e.g., number of attach requests per second).[12]
2. We also use as reference data points from another industrial EPC implementation used in a recent study [37]. We refer to it as `Industrial#2`. It also uses DPDK. We used the same test parameters and workloads (number of tunnels, control events per second, uplink to downlink traffic ratios) cited in the paper for comparing these results with PEPC. We were also able to verify with the authors of the study the data plane functions they implemented. Its data plane supports GTP, but does not implement ADC and PCEF functions. The results that we cite here for `Industrial#2` are directly taken from the paper [37].
3. **OpenAirInterface**: We also test `OpenAirInterface` release 0.2, an open source EPC implementation, used to experiment with features for next generation EPC designs [32]. It currently supports GTP, but not ADC and PCEF.
4. **OpenEPC**: We also test `OpenEPC` using the PhantomNet testbed [6]. Under PhantomNet, restricted binary-only OpenEPC images are available for testing.

Both `Industrial#1` and `Industrial#2` run as processes and not inside VMs or containers.

5.3 Parameters

We evaluate PEPC across a number of evaluation parameters that can potentially impact performance of EPC functions. Table 2 lists the evaluation parameters and their default values. Unless otherwise stated, the parameter values used during our experiments are the default ones and the reported data plane throughputs are per core.

Parameter	Default value
Ratio of uplink to downlink traffic	1:3
Downlink packet size	64 bytes
Uplink packet size	128 bytes
Signaling event type	attach request
Signaling events per second	100K
Number of users	1M

Table 2: Evaluation parameters and default values.

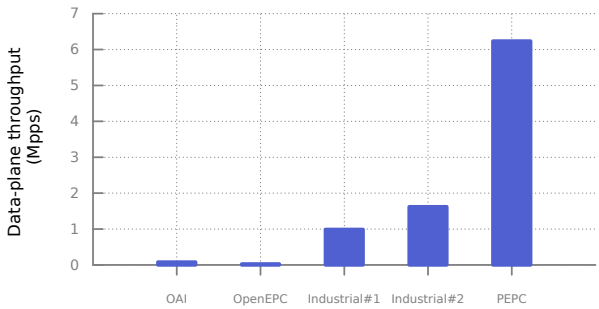

Figure 4: Data plane performance comparison.

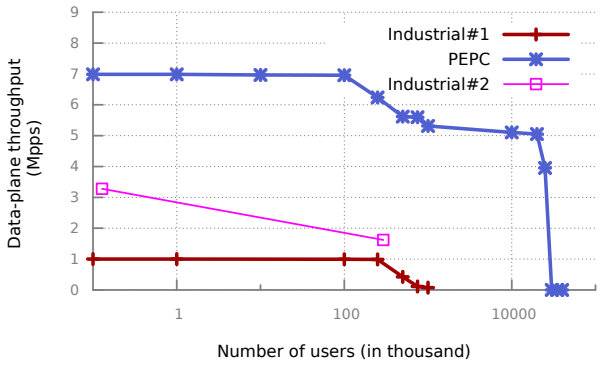

Figure 5: Data plane performance with number of users.

6 PEPC SCALABILITY

We first compare PEPC's performance against all the baselines. We then test PEPC's scalability, with increasing number of (i) user devices, (ii) signaling events, (iii) cores, and (iv) state migrations.

6.1 Comparison with all the baselines

Figure 4 shows PEPC performance in comparison with `OpenAirInterface`, OpenEPC on Phantomnet [6], `Industrial#1` and `Industrial#2`. PEPC data plane throughput is more than an order of magnitude higher as compared to `OpenAirInterface` (OAI) and `OpenEPC`, 6 times higher than `Industrial#1`, and more than 3

[11] Anonymized as the current license does not allow us to reveal name.
[12] Both Industrial#1 and PEPC were run on the same hardware with the same configuration.

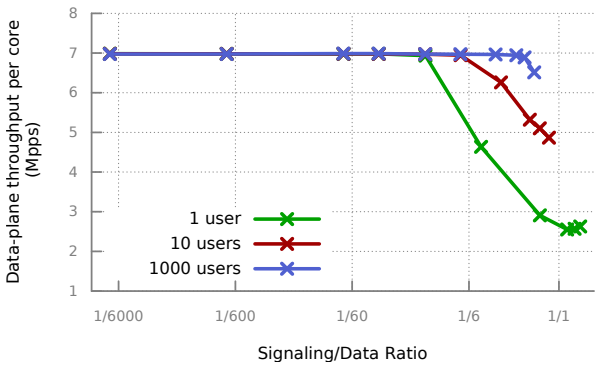

Figure 6: PEPC data plane performance as a function of signaling/data traffic ratio.

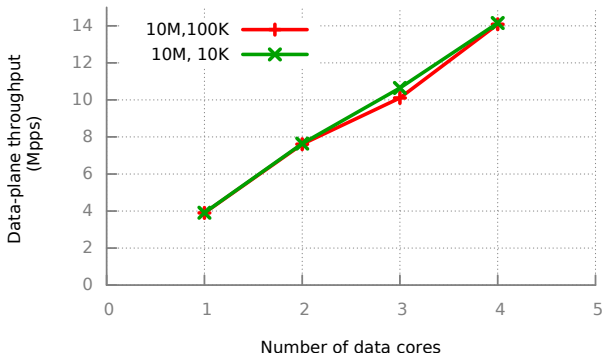

Figure 7: Data-plane performance with number of cores.

time higher than Industrial#2. OpenAirInterface and OpenEPC achieves a very low data plane throughput in comparison with other implementations[13]. The bottleneck in OpenAirInterface and OpenEPC is that it does not use kernel bypassing mechanisms (e.g. DPDK). As a result, for our next set of scalability evaluations, we drop OpenAirInterface and OpenEPC.

6.2 Increasing user devices

Figure 5 shows the data plane performance (Mpps/core) with increasing number of user devices.[14] During these experiments we consider 10K attach events per second. PEPC can achieve a performance of more than 5 Mpps for 1M user devices, and can sustain a throughput of 4 Mpps for upto 3M users. In comparison, Industrial#1 data plane throughput drops to less than 0.1 Mpps/core for a 1M users, a drop of more than 90% from the rate sustained with 100 K users. Similarly, Industrial#2 EPC exhibits a drop in data plane throughput of about 50% (from 3.28 Mpps to 1.62 Mpps), when increasing the number of user devices from 128 to 292K users, (however with these data points there are no signaling events). [15]

6.3 Increasing signaling

Figure 6 shows the data-plane performance (Mpps/core) plotted against increasing signaling/data traffic ratio. We vary the signaling to data traffic ratio, by varying the number of signaling events per second. We consider three different mix of signaling and data traffic patterns by varying the number of user devices. We observe in Figure 6 that PEPC can sustain a performance of 7 Mpps/core for signaling to data traffic ratio of 1:10. The signaling to data traffic ratio is estimated to rise to 1:17 [23]. In the worst case, with 1:1 signaling to data traffic ratio and only a single user, PEPC still achieves a throughput of 2.6 Mpps. In contrast, Industrial#1 performance drops to close to 0 for more than 10K signaling events per second (signaling to data ratio of 1:100) , whereas Industrial#2 performance drops by 15% with 3000 signaling events per second (signaling to data ratio of 1:100).

6.4 Scaling with number of data cores

Figure 7 shows how our data plane performance scales with increasing number of data cores. The label (X,Y) refers to X users and Y signaling events. For this experiment we generate 10 Gbps of traffic on all four ports available in our test setup. To ensure the same traffic ratios as the other tests, we then split received traffic before uplink and downlink processing. With 4 data cores for data traffic, the aggregate data plane performance scales linearly to 14 Mpps, with a total 10M users and 100K signaling events.

6.5 Scaling the control plane

Figure 10 shows the total number of cores needed to handle a given signaling to data ratio. The signaling to data ratio specifically refers to the ratio between the number of attach requests and the number of data packets. Note that each attach request leads to several request/response messages between the UE/eNodeB and the EPC. For these experiments, there is full support for parsing and sending S1AP messages, including NAS messages, over SCTP. We vary the signaling to data ratio, by increasing the rate at which attach requests are generated[16], while keeping the data plane load constant. The data plane load is the maximum data rate that can be handled by a single data core.

To put the results in perspective, with a signaling to data ratio of 1:304, which according to estimates is the peak signaling to data ratio for current workloads [23][17], PEPC would need a data core and a separate control core. For our test experiments, this translates into 5.2 Mpps of data traffic and <19K attach requests per second.

Figure 11 shows the maximum rate of attach requests that can be handled as we increase the number of control cores. With 1 core we can handle about 20K attach request per second and with 8 cores about 120K requests per second. In our current implementation, we use the SCTP implementation in the Linux kernel because of lack of existing SCTP support in the NetBricks framework. We

[13]The results assume 250K user devices and 10K attach/s for for PEPC and Industrial#1, 292K users for Industrial#2 with 3000 signaling events, whereas for OpenAirInterface/OpenEPC uses a single user.
[14]We use the number of users as a proxy for number of tunnels. It is possible that a user device may have multiple tunnels.
[15]Note for comparison with Industrial#2, we changed the traffic ratio of downlink to uplink to 1:3 to match their configurations. PEPC had the same data plane throughput with the new traffic ratio.
[16]For simplicity, in these experiments each core has a separate SCTP connection, over which the S1AP messages are exchanged. The total number of users were evenly divided over these connections.
[17]This is the estimate for the peak rate of signaling events that lead to some user state synchronization.

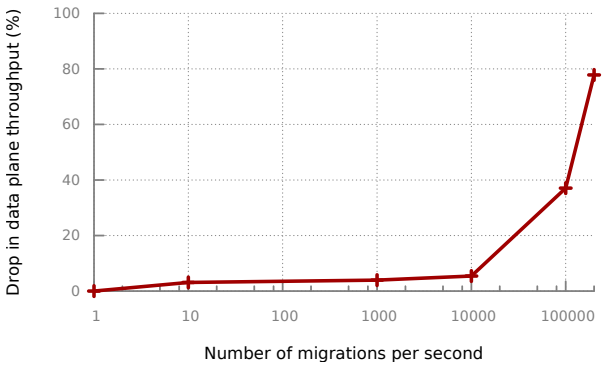

Figure 8: The impact of state migrations on data plane throughput.

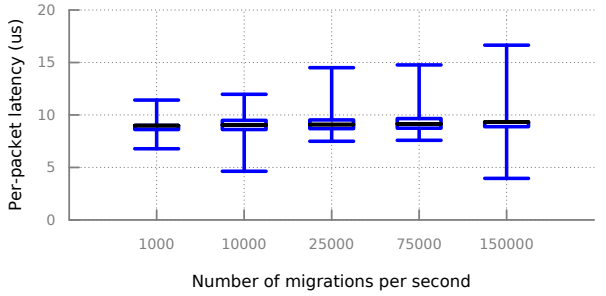

Figure 9: The impact of state migrations on per-packet latency.

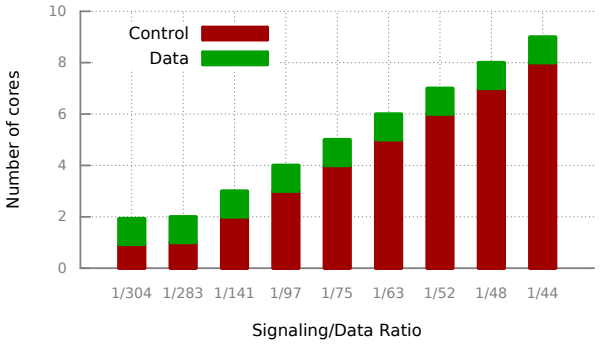

Figure 10: The number of cores needed to handle a given signaling to data ratio, with increasing number of attach requests and support for handling S1AP and NAS messages.

observe that such a kernel implementation can become a potential bottleneck when handling signaling messages.

6.6 Scalability with state migrations

Impact on data plane throughput: Figure 8 shows that PEPC only experiences a drop of 5% in data plane throughput with 10K user migrations per second. We start to see a more significant drop in performance about 37% with 100K migrations per second.

Impact on per-packet latency: In Figure 9 shows the end-end per-packet latency distribution during state migrations. We do not observe any appreciable increase in median per-packet latency, but as we increase the number of migrations, in the worst-case, there can be an increase of 4μs in per packet latency with 25K migrations per second.

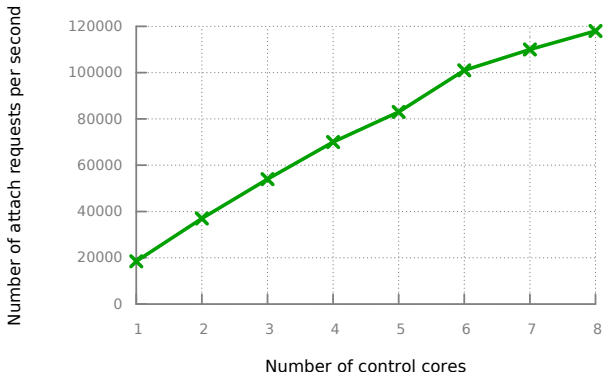

Figure 11: The number of attach requests that can be handled with increasing number of control cores. In these experiments, S1AP and NAS protocols are used.

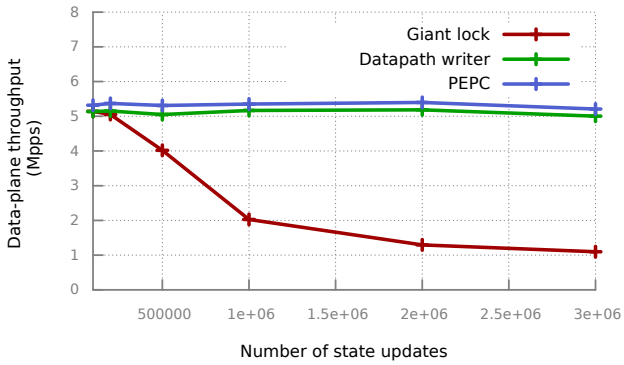

Figure 12: Comparison of different shared state implementations.

7 FACTOR ANALYSIS

To better understand PEPC design choices and separate out the benefits of PEPC's different design ideas, we perform a series of micro-benchmarks. These include (i) comparison between different shared state implementations, (ii) impact of batching, (iv) impact of two-level state tables, (v) impact of customization.

7.1 Shared state implementations

We consider three different implementations for the shared state in PEPC slice. The "Giant lock" uses a single giant lock to protect access to the entire state table, consisting of state of multiple users. In the "Datapath writer" implementation, we have a fine-grained Read/Write lock associated with each user state, but there is a single

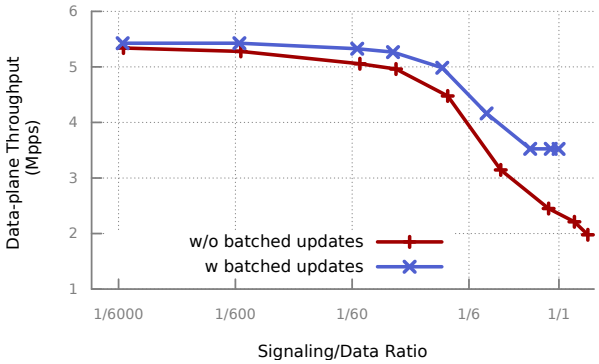

Figure 13: The impact on data plane performance by batching updates.

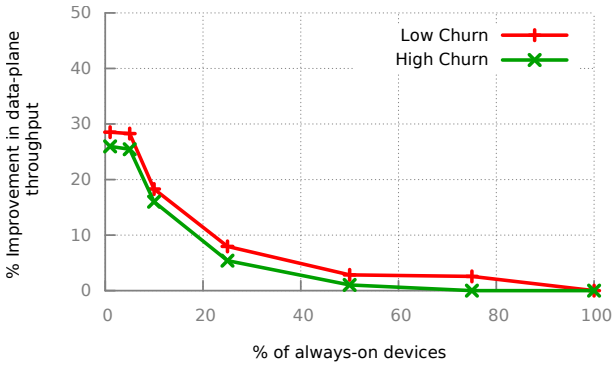

Figure 14: Performance improvement with two-level state tables over a single state table.

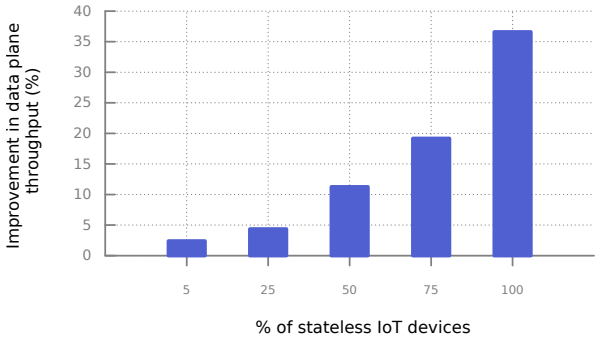

Figure 15: Benefits of customizing IoT devices in PEPC.

per-user shared state, and the data-plane also has write access to this shared state. "PEPC" uses fine-grained per user locks, but separate out per user charging record, for which the data plane is the only writer and control plane only reads it. In Figure 12, we observe with a 'Giant lock", data plane performance drops severely, for large state updates. For 3M state updates the performance drops close to 1 Mpps, whereas with fine grained locks both "Datapath writer" and "PEPC" maintain a consistent data plane throughput irrespective of the number of state updates. However, with write accesses to both data and control plane, in the case of "Datapath writer" we see only upto 0.3 Mpps drop in data plane throughput as compared to "PEPC".

7.2 Impact of batching updates

PEPC batches updates to the data plane, related to the insertion or deletion of a specific user state Figure 13 shows the benefits of batching updates at the data plane in the case of an attach event which leads to a new user state being inserted. In the case of batched updates, data plane syncs updates from the control plane only every 32 packets. Figure 13 shows that with a signaling to data traffic ratio of 1:1, batched updates result in a performance improvement of more than 1 Mpps.

7.3 Impact of two-level state tables

In Figure 14, we show the data plane performance improvement with two-level state tables as compared to a single state table. Our goal is to (i) measure the data plane performance improvement with two state tables as a function of number of always-on devices and (ii) investigate how this is impacted by the churn between the primary and secondary state tables.

We consider a total of 1M devices, and vary the fraction of always-on devices; the state for these devices is always maintained in the primary state table. The remaining devices are maintained in the secondary state table, and depending on the level of churn, we move the state of some of these devices into the primary state table, and similarly evict the state of some of the devices from the primary table to the secondary table. 'Low Churn' refers to 1% of all devices moving into the primary state table per second and 1% of all devices getting evicted from the primary state table. Similarly, 'High Churn' refers to 10% of all devices moving into and getting evicted from the primary state table.

We observe close to 29-27% improvement in data plane performance when 1% of the devices are always-on. The primary reason for the performance improvement is the lookup performance on the data path i.e., with a smaller primary table, the data plane's per-packet state lookup performance improves. As the % of always-on devices increases, the performance gap decreases, with 50% always-on devices there is a performance improvement of 1-3% and with 100% always-on devices, the performance is the same as with a single state table. We also observe that increasing the churn rate does not have a significant impact on the data plane performance ($\leq$ 2%).

7.4 Impact of customization

Figure 15 shows the benefits of customizing *Stateless IoT devices* in PEPC. Specifically, we consider a total of 10M devices, and vary the percentage of IoT devices. For 5% of these devices, we observe about 3% improvement in data plane throughput as compared to the case where there is no customization for these devices. As we increase the percentage of these devices, the performance increases sharply, with a performance improvement of about 38% in the case when there are only *Stateless IoT devices*.

8 DISCUSSION

Failure handling: In the current EPC architecture, network functions (e.g., MME and S-GW) can fail independently. Thus, if a MME fails, the S/P-GW can still process data traffic correctly until there are any signaling events. However, if a S/P-GW fails, no data/voice traffic can be processed correctly until the S/P-GW state is reconstructed and a connection between the MME and the new S/P-GW is established.

In PEPC, there is primarily a single failure mode (i.e., a PEPC node fails). Thus, if a PEPC node fails, both the user's data and control traffic cannot be processed until the necessary user state is recovered. To handle failures in PEPC, we can borrow from recent work on providing fault tolerance for middleboxes [36, 42].

Relevance for 5G: Future 5G cellular networks are expected to see significant change in traffic workloads [23]. There are a number of trends that are expected to contribute to this, including, (i) a shift to small cell sizes, which will likely cause more mobility handoffs, and hence higher signaling load, (ii) the emergence of many IoT devices with high signaling to data ratio [41], (iii) new latency sensitive application like driverless cars. As discussed earlier (2.1), the existing EPC design is poorly equipped to handle these workloads due to poor scalability and lack of elasticity in the current design which leads to a largely centralized deployment of EPC systems [35]. On the contrary, PEPC is targeted to handle such future workloads, e.g., in PEPC there is little impact of increasing signaling traffic on the data traffic, and it can enable distributed cellular core deployments by facilitating horizontal scaling by users and providing support for efficient migration of user processing. In this respect, PEPC's goals are aligned with recent initiatives such as MCORD [21] which are also aimed at enabling such deployments.

Potential savings with PEPC: We perform back of the envelop calculations to analyze the potential cost savings with PEPC for a large carrier. We compare against `Industrial#1`. Taking into account the total number of subscribers for the carrier [44] and maximum possible data rate supported by the carrier when using LTE [45], we infer the total number of servers needed to support all the subscribers at their peak data load. We assumes that similar amount of signaling traffic from the eNodeBs can be handled by both implementations. For our analysis, we use the per-core performance for PEPC and `Industrial#1` from our experimental evaluation results. The analysis reveals that a large carrier can potentially save upto 100K servers with PEPC in EPC costs.

9 RELATED WORK

SDN based cellular core designs: SoftCell [15] and SoftMoW [24] argue for a SDN-based architecture for the cellular core, with flexible wide area routing. SoftCell addresses the challenges in supporting fine-grained policies for mobile devices in such a cellular network architecture, such as minimizing forwarding table entries in the core switches. SoftMoW proposes a recursive and reconfigurable SDN-based cellular WAN architecture with a logically centralized control plane for achieving global optimizations, such as reducing routing path lengths for latency sensitive applications. These are complementary to PEPC, which addresses the scaling issues with current decomposition of EPC functions and proposes an alternate state driven EPC system design, which scales with future cellular workloads, such as increasing user devices and signaling traffic. For global routing optimizations, PEPC can potentially benefit from these proposals for a programmable cellular WAN.

Virtualizing EPC functions: KLEIN [35] proposes a NFV-based cellular core, which assumes a software-based EPC implementation. It addresses the problem of managing EPC resources in a virtualized EPC, with the objective of distributing load optimally across EPC instances. KLEIN is complementary to PEPC, as PEPC targets the specific system design issues of EPC functions, whereas KLEIN provides a orchestration layer for managing EPC functions across a cellular network. KLEIN uses OpenAirInterface as the underlying virtual EPC, and we compare against it in §5. There are other proposals [9, 16, 34, 39] that present backwards compatible mechanisms for virtualizing core EPC functions like S-GW and P-GW, and using SDN to route traffic through software instances.

Managing signaling traffic: Recent proposals [7, 26] argue for handling the increasing signaling traffic from the base stations. SCALE [7] proposes mechanisms for scaling a software MME to handle increasing signaling and whereas ProCel [26] argues for reducing the signaling traffic from the base stations which is forwarded to the cellular core. Both these designs are in a sense complementary to PEPC, which addresses the problem of frequent synchronization between EPC functions due to state duplication.

IoT customizations: Recent proposals also call for clean-slate proposals for customizing EPC and base station processing for IoT devices [22, 27] and designing a more evolvable software defined cellular architecture to support new services (e.g., related to machine-to-machine communication) [43]. In contrast, PEPC design enables customization of EPC functions for IoT devices without modifying the base stations.

Consolidated middlebox architectures: There are general middlebox architectures that aim to consolidate multiple middlebox applications on a single hardware platform, such as CoMb [40]. PEPC is a specific system design of EPC functions that goes beyond simple consolidation and addresses specific scaling issues in EPC functions related to state duplication, inefficient migration and fixed packet processing pipelines.

10 CONCLUSIONS

Existing EPC functions do not scale well with increasing number of user devices and signaling events. We argue the key issue for this poor scalability is how state is decomposed and managed in EPC, which leads to frequent synchronization between multiple components. To address these limitations, we propose a new system level architecture, PEPC, which involves a novel re-factoring of EPC functions, consolidating state in one location and refactoring how state is accessed. This refactoring in turn facilitates other performance optimizations, such as vertical scaling, efficient state migration and customized processing. Based on these design ideas we build PEPC, and show that it achieves roughly 3-7 times higher data plane throughput as compared to two industrial vEPC implementations. We also show that it scales well with the number of user devices and increasing signalling traffic.

11 ACKNOWLEDGEMENTS

We thank our shepherd Suman Banerjee and the anonymous SIGCOMM reviewers for their helpful feedback. We thank Alec Zadikian for his help with the S1AP protocol implementation. We also thank Ashok Sunder Rajan and Christian Maciocco for their technical help and feedback. This research was supported in part by NSF, award number 1553747, and in part by the financial support from Intel Research.

REFERENCES

[1] 3GPP Ref #: 23.401. 2016. *General Packet Radio Service (GPRS) enhancements for Evolved Universal Terrestrial Radio Access Network (E-UTRAN) access.* Retrieved 09/05/2016 from http://www.3gpp.org/DynaReport/23401.htm
[2] 3GPP Ref #: 24.301. 2016. *Non-Access-Stratum (NAS) protocol.* Retrieved 09/05/2016 from www.3gpp.org/dynareport/24301.htm
[3] 3GPP Ref #: 29.272. 2016. *Mobility Management Entity (MME) and Serving GPRS Support Node (SGSN) related interfaces based on Diameter protocol.* Retrieved 09/05/2016 from www.3gpp.org/DynaReport/29272.htm
[4] 3GPP. 2016. *3GPP Specifications.* Retrieved 09/05/2016 from http://www.3gpp.org
[5] aws lb. 2016. AWS Load balancer. (2016). Retrieved 09/05/2016 from https://aws.amazon.com/elasticloadbalancing/
[6] Arijit Banerjee, Junguk Cho, Eric Eide, Jonathon Duerig, Binh Nguyen, Robert Ricci, Jacobus Van der Merwe, Kirk Webb, and Gary Wong. 2015. PhantomNet: Research Infrastructure for Mobile Networking, Cloud Computing and Software-Defined Networking. *GetMobile: Mobile Computing and Communicationss* (2015).
[7] Arijit Banerjee, Rajesh Mahindra, Karthik Sundaresan, Sneha Kasera, Kobus Van der Merwe, and Sampath Rangarajan. 2015. Scaling the LTE Control-Plane for Future Mobile Access. In *CoNEXT'15*.
[8] Barefoot. 2016. Barefoot. (2016). Retrieved 09/05/2016 from https://www.barefootnetworks.com/
[9] A. Basta et al. 2013. A Virtual SDN-enabled LTE EPC Architecture: A case study for S-/P-Gateways functions. In *SDN4FNS'13*.
[10] blog4g. 2017. On Signaling Storm. (2017). Retrieved 01/22/2017 from http://blog.3g4g.co.uk/2012/06/on-signalling-storm-ltews.html
[11] Cisco. 2016. Cisco Visual Networking Index. (2016). Retrieved 09/21/2016 from https://goo.gl/i6rd9e
[12] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. 2009. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *SOSP'09*.
[13] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer. In *NSDI'16*.
[14] f5. 2017. f5: Load balancing. (2017). Retrieved 01/26/2017 from https://f5.com/glossary/load-balancing-101
[15] Xin Jin, Li Erran Li, Laurent Vanbever, and Jennifer Rexford. 2013. SoftCell: Scalable and Flexible Core Network Architecture. In *CoNEXT'13*.
[16] J. Kempf, B. Johansson, S. Pettersson, H. Luning, and T. Nilsson. 2012. Moving the Mobile Evolved Packet Core to the Cloud . In *International Workshop on Selected Topics in Mobile and Wireless Computing 2012*.
[17] Kubernetes. 2016. Production-Grade Container Orchestration. (2016). Retrieved 09/05/2016 from https://kubernetes.io/
[18] Lightreading. 2016. Operators Fight Back on Smartphone Signaling. (2016). Retrieved 09/05/2016 from http://goo.gl/MIbuJ9
[19] Lightreading. 2016. Operators Urge Action Against Chatty Apps. (2016). Retrieved 09/05/2016 from http://goo.gl/mqaaE5
[20] Heikki Lindholm, Lirim Osmani, Hannu Flinck, Sasu Tarkoma, and Ashwin Rao. 2015. State Space Analysis to Refactor the Mobile Core. In *AllThingsCellular'15*.
[21] M-CORD. 2016. *Mobile CORD: Enabling 5G on CORD.*
[22] Ali Mohammadkhan, K.K. Ramakrishnan, Ashok Sunder Rajan, and Christian Maciocco. 2016. CleanG: A Clean-Slate EPC Architecture and Control Plane Protocol for Next Generation Cellular Networks. In *CAN'16*.
[23] A. Mohammadkhan, K. K. Ramakrishnan, A. S. Rajan, and C. Maciocco. 2016. Considerations for re-designing the cellular infrastructure exploiting software-based networks. In *ICNP'16*.
[24] Mehrdad Moradi, Wenfei Wu, Li Erran Li, and Zhuoqing Morley Mao. 2014. SoftMoW: Recursive and Reconfigurable Cellular WAN Architecture. In *CoNEXT'14*.
[25] Robert Morris, Eddie Kohler, John Jannotti, and M Frans Kaashoek. 2000. The Click modular router. In *ACM Transactions on Computer Systems 2000*.
[26] Kanthi Nagaraj and Sachin Katti. 2014. ProCel: Smart Traffic Handling for a Scalable Software EPC. In *HotSDN '14*.
[27] Vasudevan Nagendra, Himanshu Sharma, Ayon Chakraborty, and Samir R. Das. 2016. LTE-Xtend: Scalable Support of M2M Devices in Cellular Packet Core. In *AllThingsCellular'16*.
[28] ng4T. 2016. *Next generation Telecommunication Technology Testing Tools.* Retrieved 09/05/2016 from http://www.ng4t.com/
[29] ng4T. 2016. *Wireshark Recordings.* Retrieved 09/05/2016 from http://www.ng4t.com/wireshark.html
[30] Binh Nguyen, Arijit Banerjee, Vijay Gopalakrishnan, Sneha Kasera, Seungjoon Lee, Aman Shaikh, and Jacobus Van der Merwe. 2014. Towards Understanding TCP Performance on LTE/EPC Mobile Networks. In *AllThingsCellulars'14*.
[31] Nokia. 2016. Signaling is growing 50% faster than data traffic. (2016). Retrieved 09/05/2016 from http://goo.gl/uwnRiO
[32] OpenAirInterface. [n. d.]. *OpenAirInterface: A 5G software alliance for democratising wireless innovation .*
[33] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *OSDI'16*.
[34] K. Pentikousis, Y. Wang, and W. Hu. 2013. Mobileflow: Toward software-defined mobile networks . In *International Workshop on Selected Topics in Mobile and Wireless Computing 2013*.
[35] Zafar Ayyub Qazi, Phani Krishna Penumarthi, Vyas Sekar, Vijay Gopalakrishnan, Kaustubh Joshi, and Samir R. Das. 2016. KLEIN: a Minimally Disruptive Design for an Elastic Cellular Core. In *SOSR'16*.
[36] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. 2013. Pico Replication: A High Availability Framework for Middleboxes. In *SOCC'13*.
[37] A. S. Rajan, S. Gobriel, C. Maciocco, K. B. Ramia, S. Kapury, A. Singhy, J. Ermanz, V. Gopalakrishnanz, and R. Janaz. 2015. Understanding the bottlenecks in virtualizing cellular core network functions. In *LANMAN'15*.
[38] Rust. 2016. The Rust Programming Language. (2016). Retrieved 09/05/2016 from https://www.rust-lang.org/en-US/
[39] M. R. Sama, L. M. Contreras, J. Kaippallimalil, I. Akiyoshi, H. Qian, and H. Ni. 2015. Software-Defined Control of the Virtualized Mobile Packet Core. In *IEEE Communications Magazine 2015*.
[40] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. 2012. Design and Implementation of a Consolidated Middlebox Architecture. In *NSDI'12*.
[41] Muhammad Zubair Shafiq, Lusheng Ji, Alex X. Liu, Jeffrey Pang, and Jia Wang. 2012. A First Look at Cellular Machine-to-machine Traffic: Large Scale Measurement and Characterization. In *SIGMETRICS'12*.
[42] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. Rollback-Recovery for Middleboxes. In *SIGCOMM'15*.
[43] Aisha Syed and Kobus Van der Merwe. 2016. Proteus: A Network Service Control Platform for Service Evolution in a Mobile Software Defined Infrastructure. In *MobiCom'16*.
[44] Fierce Wireless. 2016. *How Verizon, AT&T, T-Mobile, Sprint and more stacked up in Q2 2016: The top 7 carriers.* Retrieved 09/05/2016 from goo.gl/xhsXyx
[45] Yoni. 2016. Mammoth Study Finds Sprint's 4G Network is the Worst – And It's Not Even Close. (2016). Retrieved 09/05/2016 from goo.gl/3ItLYM
[46] Zeljko. 2016. LTE Design and Deployment Strategies. (2016). Retrieved 09/05/2016 from goo.gl/AAuCbR