Performance Comparison of State Synchronization Techniques in a Distributed LTE EPC

Pratik Satapathy, Jash Dave, Priyanka Naik, Mythili Vutukuru
Department of Computer Science and Engineering, Indian Institute of Technology, Bombay
Email: {pratik,jashdave,ppnaik,mythili}@cse.iitb.ac.in

***Abstract*—With Network Function Virtualization (NFV) generating significant interest in the network operator community, many network functions, including the LTE EPC, are being built as virtualized software appliances running on commodity hardware, as opposed to custom hardware. To provide fault tolerance and scalable performance, the virtualized network functions are typically built in a clustered architecture, with a front-end load balancer distributing incoming traffic across multiple replicas, and the replicas synchronizing shared state with each other for resilience and consistency. Our work compares several distributed designs of the LTE EPC along the axis of the frequency of state synchronization across the replicas, and quantifies the performance overhead of state synchronization for control plane and data plane operations. Using experiments with our distributed EPC prototypes, we show that synchronizing state for every message incurs a prohibitive performance penalty (over 70% reduction in throughput as compared to the case of no synchronization). On the other hand, synchronizing state at session boundaries, while providing lower fault tolerance guarantees, imposes a smaller performance penalty. We believe that our open-source prototypes and experimental results will guide future distributed EPC designs, and help operators pick the right design that is appropriate for their fault tolerance and performance requirements.**

I. Introduction

The recent proliferation of mobile devices has resulted in a significant rise in mobile data traffic in cellular data networks, placing enormous stress on existing network architectures. Network Function Virtualization (NFV) is one of the design paradigms to improve the performance of future cellular networks. With NFV, the various components of a cellular network are built as software modules running on an elastic commodity server platform, as opposed to custom-built hardware. In addition to lower cost of developing software, NFV also enables easier scaling of network components with increasing load as compared to a custom hardware-based design. There has been significant interest in NFV within the telecom network operator community, with components like the LTE Evolved Packet Core (EPC) being actively considered for virtualization.

For NFV to be really useful, the various Virtual Network Functions (VNFs) in a network must be built, not as monolithic pieces of software, but as horizontally scalable clustered components. While a clustered VNF may appear as a single logical entity to other network components, it is internally implemented as multiple *replicas*, with the number of replicas scaling according to load. Any clustered VNF design typically has a front end or a *load balancer* to distribute incoming traffic amongst the various replicas, and a *data store* of some kind to synchronize state amongst the replicas. The multiple replicas must synchronize state for various reasons like fault tolerance (e.g., when one of the replicas goes down) and consistency (e.g., when one replica must access the state being used by another replica). This state synchronization can be done at various time granularities, and in general, higher the frequency of synchronization, better the fault tolerance and consistency benefits, but also greater the performance overhead. Therefore, a VNF designer must weigh these competing concerns in the context of a given VNF to make a suitable state synchronization design choice.

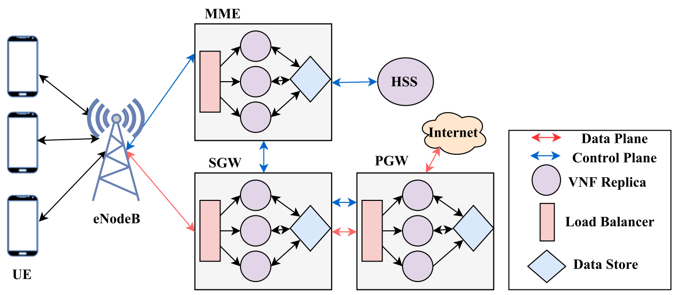


Fig. 1: Distributed LTE-EPC

In our paper, we compare the performance overhead of various state synchronization options in the design of a distributed LTE EPC. The LTE EPC connects the Radio Access Network (RAN) of a cellular data network to various external networks. The EPC primarily consists of the Mobility Management Entity (MME) and the Home Subscriber Server (HSS) in the control plane, and the Serving and Packet Gateways (S/P-GWs) in the data plane. The MME, in consultation with the HSS database, handles various signaling messages (e.g., the attach request when a user initiates data traffic) and sets up packet forwarding state for the user in the gateways, while the gateways use this state to correctly forward user traffic through the cellular core. In a virtualized EPC, each of these EPC components will likely be built as distributed scalable software running on commodity hardware, as illustrated in Figure 1. In this context, we compare the performance of the following state synchronization choices in the LTE EPC VNFs: (i) *Always sync*, where each replica of a VNF synchronizes state with other replicas, or with a state database, before/after every message, (ii) *Session sync*, where a replica synchronizes state once for a *session*, defined as a coherent group of messages that arrive in quick succession (e.g., the many steps that comprise the attach procedure, or the multiple packets in one TCP flow of a user), and (iii) *No sync*, where the replicas hold all state locally without incurring any state synchronization overhead.

While prior work (§II) has proposed several distributed designs for the MME, each system makes only one particular choice with respect to state management. To the best of our knowledge, there has been no performance comparison across the spectrum of state synchronization design choices in the control plane (MME) and data plane (S/P-GW) components of the EPC. To this end, we propose several distributed designs of the EPC components (§III), with varying frequencies of state synchronization between the replicas. We implement these designs (§IV) over an existing NFV-based LTE EPC codebase [1]. Our code is available as open-source for other researchers to experiment with.

Our experiments (§V) show that state synchronization before/after every session has 51% lower throughput and doing so with every message has 71% lower throughput, as compared to the base case of no synchronization in the control plane. Similarly, synchronization before/after every packet has 76% lower throughput as compared to synchronization once per user flow in the data plane. We see from our results that, while synchronizing state with every message provides the best fault tolerance guarantees, it also suffers a high performance penalty. Synchronizing state at the granularity of sessions provides a reasonable middle ground with respect to both performance and fault tolerance. We believe that research such as ours provides useful guidelines to designers of virtualized EPCs. Network operators deploying clustered EPC components can weigh the fault tolerance requirements against their performance requirements for various classes of traffic to adopt suitable state synchronization strategies. While the absolute values of performance reported in our work can change with implementation choices and the functionalities of the VNFs, we believe that the broad conclusions drawn in our study would hold across other NFV systems as well.

II. Related Work

Most related work in the area of distributed EPC designs has dealt with proposing distributed architectures for the MME component of the EPC. DMME [2] proposes a geographically distributed architecture for the LTE control plane, where smaller MMEs are collocated with every eNodeB to serve users locally. The various distributed MME replicas store state locally and synchronize with an object store at the end of the user's activity, or when the user moves to the region covered by another MME replica. The clustered LTE EPC architectures evaluated in our work do not consider the case of geographic distribution, but only cover the case where multiple VNF replicas are spawned to scale performance. SCALE [3] proposes a scalable distributed architecture for the MME, where users are partitioned among replicas using consistent hashing. MME replicas that perform computations store state locally, and synchronize their state with a small number of other replicas at session boundaries for fault tolerance and load balancing. Takano et al. [4] and Gopika et al. [5] also provide similar distributed MME designs, where the replicas synchronize state using a common data store at session boundaries. While all of these papers propose various distributed MME designs, none compares the entire spectrum of possible designs. Basta et al. [6] proposes a model for placement of virtualized S/P-GW in multiple data centres but does not consider distributing the S/P-GW components of the EPC, as we do in our work.

Prior works like split-merge [7], pico-replication [8] and, FTMB [9] provide a generic framework for building fault-tolerant and scalable VNFs. Kablan et al. [10] propose and evaluate a generic VNF architecture where replicas keep no state and always synchronize with a datastore. Our work, on the other hand, explores all possible state synchronization choices, for the specific case of EPC VNFs.

III. Design

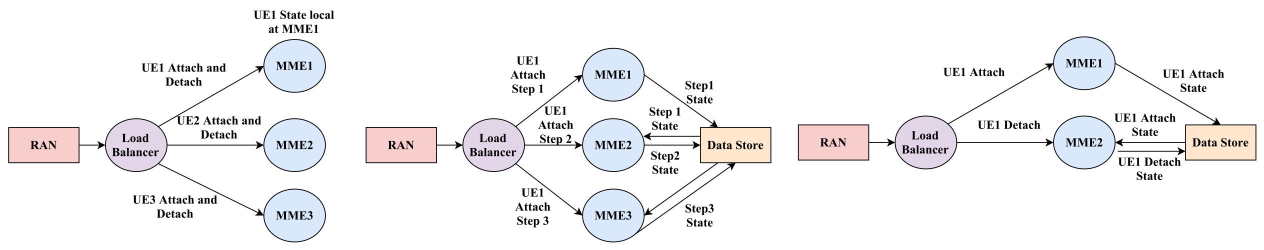
We now discuss the different distributed EPC designs we compare, focusing on their state synchronization aspects. We begin with a brief overview of the LTE EPC.

A. Background: LTE EPC

A typical 4G LTE network consists of two components: a RAN (Radio Access Network) and an EPC (Evolved packet core) that connects the RAN to various other networks. The RAN consists of eNodeBs, which are mobile towers responsible for serving the UE (user equipment) over the wireless channel. The EPC consists of control plane elements such as MME, HSS, and an optional PCRF (Policy and Charging Rules Function), and data plane elements such as SGW and PGW. When a device powers on and connects to an LTE network, it sends an attach request to the EPC via the eNodeB. The attach request involves mutual authentication between the UE and the network, primarily performed by the MME in consultation with the HSS database. At the end of the attach procedure, a tunnel or a forwarding path is setup for the user's data via the LTE gateways (the SGW serving the user's area, and the PGW that connects to the user's desired destination) by the MME. All subsequent data sent by the user are forwarded via the chosen gateways using this forwarding state. The MME also processes other control plane requests like disconnecting an idle user from the network, and handling user mobility via handovers between components.

B. Distributed LTE EPC

In an NFV-based implementation of the LTE EPC, a single software instance (hosted on a virtual or physical machine) of any EPC component like the MME or SGW can quickly become a bottleneck under high load, and the underlying hardware limitations may not allow provisioning the software instance beyond a limit. Therefore, any reasonable virtualized EPC deployment would consist of distributed or clustered



(a) No sync (b) Always sync (c) Session sync

Fig. 2: Distributed EPC Designs with Varying State Synchronization.

EPC components, where a single logical EPC component is implemented as multiple replicas sharing the incoming load. With such a design, extra software instances of a component can be spawned or taken down according to the offered load, making the NFV-based EPC much more elastic to varying demands than a traditional hardware implementation. Figure 1 shows our reference design of a distributed LTE EPC, where each of MME, SGW and PGW are implemented as clusters of replicas. We omit the PCRF in our design for simplicity. We believe that this simple reference design suffices to evaluate the performance overheads of state synchronization in the LTE EPC, across a range of design choices.

Each clustered component internally consists of a load balancer that distributes incoming load across one or more replicas, which perform the actual processing of user requests. The replicas can synchronize state either using a common data store, or by replicating their local state at other replicas. Without loss of generality, we assume that the replicas use a common data store to synchronize state; our discussion holds for when other replicas are used instead of a common data store. Our inspection of shared state at EPC components shows that all shared state is *UE-centric*, e.g., user context, forwarding state, location updates, and so on. While some of this state is persistent (e.g., user's subscriber information), most state only lasts for the duration that the user is connected to the network. Therefore, all shared state in the EPC can be represented as key-value pairs, keyed by a unique user identifier, and a key-value store could serve as the data store.

Note that state synchronization primarily involves creating or updating state in the control plane components, while it involves fetching or reading state in the data plane components. For example, during an attach procedure, the MME creates and stores forwarding state for the user at the gateways. When the gateways must forward a user's data packet, they must fetch this forwarding state from their local memory or a common data store. We use the term "state synchronization" to mean both writing as well as reading shared UE state.

Let us now consider the different design options with respect to synchronizing UE state across replicas. At one extreme is a design that completely eschews state synchronization across replicas. Each replica stores its state locally. Users are partitioned to replicas either statically, or dynamically (e.g., based on load or other factors), and requests of a certain user are always handled by the same MME or S/P-GW replica. The load balancer or the front end to these replicas must be aware of this mapping of users to replicas, and must correctly redirect traffic belonging to a certain user to the correct replica. This redirection can be accomplished in many ways in the load balancer, e.g., consistently hashing a user identifier to obtain a replica identifier (as done in SCALE [3]). Note that while this design incurs no state synchronization overhead, it has very low fault tolerance, and failure of a replica could result in complete loss of state of a user, forcing the user to reconnect to the network and re-establish state. As an example, Figure 2(a) shows a clustered MME without any state synchronization, and all signaling messages from the user would be handled by the same replica.

At the other extreme is a design where replicas are completely stateless and always synchronize state at the end of every packet. For example, Figure 2(b) shows a stateless MME implementation, where various authentication steps of a single attach request of a user can go to different MME replicas, and the replicas must get (put) the updated state of a user before (after) processing every packet. An example in the data plane would be an SGW replica fetching forwarding state of the user from a common data store or from another replica, for every packet of the user's flow. The load balancer is very simple in this design, as any incoming packet can be redirected to any replica. While this design obviously has a high synchronization overhead, it also has very high fault tolerance: the failure of a replica even in the middle of a multi-step user request will be tolerated well, and another replica can continue where the previous one left out. The underlying reliable transport protocols like TCP and SCTP over which most signaling messages or user data are sent can easily mask the loss of a small number of packets that happen during the transition from one replica to the other.

The middle-ground between both of these designs is one in which state is synchronized at the granularity of a session. That is, shared state is fetched from the data store at the start of the session, stored back at the end of the session, and maintained locally for the duration of a session. We define a session as series of logically related consecutive packets from a user. For example, the various authentication steps of an attach request can constitute a session in the control plane, while all packets of a given TCP connection can be considered as a session in the data plane. In such a design, the load balancer must maintain affinity and direct packets of a given session to the same replica, but packets of the same user that belong to different sessions, can be served by different replicas. For example, Figure 2(c) illustrates a clustered MME with session-level synchronization, where one MME replica processes all packets of an attach request, while another may process the detach request from the same user. An analogous example in the data plane would be the gateways fetching user forwarding state from the data store only once at the start of a user flow, and caching it locally for the rest of the packets of the flow. If a replica fails in the middle of a session, the user will perceive the failure, and will have to retry from the beginning of the session at another replica. For example, the user may have to restart the handover procedure if an MME replica fails in the middle of the request. However, the user will not face major disruptions, e.g., he will not have to reattach, as the attach context will have been saved in the data store and can be retrieved by the new MME replica.

It is easy to see that different state synchronization choices result in different performance overheads at the VNF replicas, the load balancer, and the common data store, and provide very different fault tolerance properties. While it is intuitive that more frequent synchronization will lead to lower performance, we aim to quantify this performance penalty, so that operators can make an informed choice on how much performance to trade off for a desired level of fault tolerance.

IV. Implementation

In this section we describe our implementations of various distributed EPC candidates. Our implementations are extensions of our in-house monolithic NFV-based LTE EPC [1]. Our NFV-based EPC consists of multi-threaded high performance implementations of the MME, SGW, and PGW, along with a RAN simulator and sink. The RAN simulator is a multi-threaded client that emulates multiple UEs, which perform simple signaling procedures like attach and detach, and transfer data through the EPC for specified durations. The RAN module connects to the MME over the S1AP interface, which runs on SCTP. Therefore, the RAN simulator acts as an SCTP client while the MME acts as an SCTP server. The MME also connects over GTP-C on UDP to the SGW and PGW to setup and modify forwarding state during signaling procedures. The RAN simulator sends GTP data over UDP via the gateways in the data plane. Our EPC implementation implements only a subset of the LTE signaling protocols, and is not fully standards compliant. However, it captures the computational complexities of the important LTE procedures, and is suitable for modification into a clustered implementation.

A. Load Balancer

We first modified each of the EPC components to have a load balancer as the front end. All traffic from a downstream EPC component in a service chain passes through the load balancer before being processed by a replica of the component. We had two design choices in implementing a load balancer: a network layer load balancer that simply rewrites the destination IP address of packets to that of the replica, or an application layer load balancer that performs protocol processing as well. A network layer load balancer imposes a lower performance overhead, whereas an application layer load balancer provides more functionality (e.g., selecting replicas based on application layer semantics) at a higher processing cost. While most prior work (e.g., SCALE [3]) has used an application layer load balancer in their designs, we chose a network layer load balancer as it suffices for our goal of comparing state synchronization options.

	RAMCloud	LevelDB	Redis	Memcached
Throughput	9159	20616	23667	22747

TABLE I: Performance (req/s) of the key-value store client library.

We developed two network layer load balancers to use in our prototype implementation. The first one is based on the Linux Virtual Server (LVS) [11] framework. LVS is an open source kernel module based network load balancer with some preset load distribution algorithms. The kernel module rewrites the destination IP address of incoming traffic to that of one of the chosen VNF replicas, in a manner that is transparent to the peers of a VNF component. The LVS framework comes with simple algorithms for selecting one of the replicas for any given packet: a simple round-robin policy picks replicas in a round robin fashion on incoming packets, while a source-hash based policy uses the hash of the source IP and port of the TCP/UDP header to pick a replica (ensuring packets from the same source IP and port reach the same replica). We use the LVS module in the "direct server return" mode, where the reply from the replicas are sent directly to the source without going through the load balancer again, for performance reasons.

The LVS based framework does not provide enough flexibility to pick replicas based on application layer information like user identifiers. Therefore, we also implemented a custom load balancer as a Linux kernel module to redirect packets based on user identifier information that is often embedded inside the payload of transport layer packets. Our kernel module uses Netfilter hooks to intercept packets before they reach the replica. The module takes as inputs (via sysfs) the various message types processed by the VNF, and the location of the application layer key in each packet type, and uses this information to extract the application layer key from packets. The hash of the application layer key (when available and applicable) is then used to assign packets to replicas. The module also takes an input information about session boundaries of various messages in order to maintain session-level affinity to replicas. This custom kernel module can pick a replica IP to rewrite the destination IP field of a packet using much richer semantics than what is possible by the LVS server. We evaluate both load balancers for their performance in §V.

B. Shared Data store

Clustered EPC components in our implementation synchronize state either by storing state at other replicas, or by storing

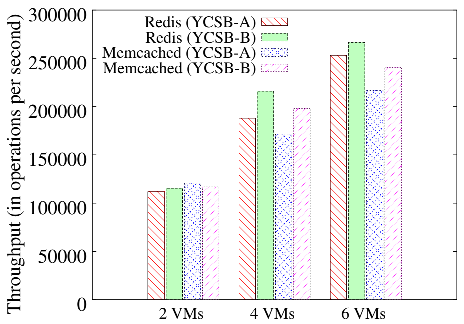


Fig. 3: Performance comparison of Redis v/s Memcached.

it in a common shared data store. We find that all shared state in the EPC components can be represented as key-value pairs, and the EPC procedures require simple operations like get, put, multi-get, multi-put, and delete on these key-value pairs, and do not require any transaction processing support. We also find that all the components, except for the HSS, do not need to store any shared state persistently. Therefore, when replicas use a common data store for synchronization, we use in-memory key value stores to synchronize data. Our EPC components were modified to make all state access operations to pass through a simple multi-threaded client library developed by us, which simply acts as a wrapper around get/put operations and in turn performs get/put operations on a backend high performance key-value store.

We ran simple benchmarking experiments on existing high performance in-memory key-value stores to pick a suitable candidate. We compared the following data stores: LevelDB [12], Memcached [13], RAMCloud [14] and Redis [15]. We tested the key-value stores with simulated LTE EPC data (30 bytes key and 2000 bytes value), with a YCSB-A or YCSB-B [16] type workload. The client and server VMs were hosted on single server with 24 CPU cores (Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz) and 64GB RAM.

We first evaluate the performance impact of accessing the key-value store at the VNF replicas, which act as clients to store data in the key-value stores. The client program was hosted on single VM with single CPU core and 4GB of RAM. Server VMs were configured differently for different data stores, to ensure that they were well provisioned, to isolate only the performance impact at the client side. Table I shows the throughput of our client library (in terms of average number of get/put operations performed by the library) when accessing different key-value stores in the backend. Our results show that the RAMCloud client is the most CPU-intensive, while the Redis and Memcached clients give the highest throughput, and hence impose the lowest performance overhead at our VNF replicas.

Next, we evaluate the performance of the Redis and Memcached key-value stores at the server side by provisioning comparable resources to both. We used 2 client VMs each with 8 CPU cores and 4GB of RAM to load a varying number of server VMs. Each server VM hosted a single instance of the Redis or Memcached server, and was configured with 1 CPU core and 4GB RAM. We turned off the disk writes for Redis to keep it comparable to Memcached. Figure 3 shows the performance comparison of Redis vs. Memcached under YCSB-A and YCSB-B workloads. We see from the figure that while Redis and Memcached perform comparably, Redis performs slightly better. Further, Redis also provides persistence as a feature. Therefore, we choose Redis as the common data store in the rest of our experiments.

C. *VNF Replicas*

We now describes changes made to the VNF components of our monolithic NFV EPC [1]. We first modified all components to use our wrapper client library when accessing any shared state, to enable a quick switch between various synchronization options in our experiments. When using a shared data store for state synchronization, all EPC components use the same data store: the MME pushes forwarding state to the data store during signaling, and the gateways pull the state from the same data store during forwarding in the data plane.

We also integrated all EPC components to work with a load balancer. We modified our RAN simulator slightly to work correctly with our load balancer across various designs as follows. When evaluating the design where the MME replicas store state locally and do not synchronize, the MME load balancer needs to ensure UE-level affinity and direct all traffic from a UE to the MME replica that has its state. To enable this in our LVS load balancer that cannot use UE information to pick replicas, we modified our load generator to send all signaling messages from the same user over the same SCTP connection. With this modification, the source hash mode of the LVS automatically ensures that traffic of the same UE lands up at the same MME replica. We did not place any such restriction when this affinity was not required, ensuring that different sessions of a UE, or different packets of one session of a UE go to different replicas. A similar modification was also required at the MME, when it communicates with the SGW and PGW to setup user state at the end of signaling procedures. To ensure that the forwarding state of the same user reaches the same SGW replica (when the SGW load balancer only uses source hash to decide replicas), we picked the source port of the UDP socket at the MME replica based on a hash of the user forwarding state information (the user tunnel identifier, to be precise). This ensured that forwarding state setup messages of a particular user from the MME to the SGW always reached the same SGW replica when such affinity was desired, even though the simple LVS load balancer had no visibility into user identifier information. All of these changes ensured that we could direct traffic correctly to replicas in different clustered designs, in order to correctly evaluate the state synchronization overhead, without adding extra complexity to the load balancer.

V. Evaluation

We now evaluate the performance of various distributed EPC designs and their state synchronization overheads using our prototype implementation. Our experimental setup consists of the various EPC components, along with the RAN simulator

Mode	LVS	Custom
UDP (UE affinity)	18.1	18.8
UDP (Session affinity)	18.1	19
UDP (No affinity)	19.1	19.6
TCP (No affinity)	17.7	19.2

TABLE II: Throughput (Gbps) of LVS and custom load balancer.

and sink, hosted as VMs on physical servers with 24 CPU cores (Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz) and 64GB RAM. The hypervisor used is KVM and the VMs are connected using virtio over KVM bridge. The RAN and the sink are assigned 8 cores and 4GB RAM each, while the MME, SGW, and PGW replicas are assigned 1 core and 2GB RAM each. The load balancer modules are assigned 2 cores each with 4GB RAM. All VMs run on Ubuntu 14.04. The datastore consists of a Redis 3.4 cluster running on 3 VMs with 4 cores and 4GB RAM each. Each VM run 4 Redis servers (2 masters, 2 slaves) with a replication factor of 1.

A. *Load Balancer Benchmarking*

We begin with measuring the performance of our load balancers. Because the eventual goal of our evaluation is to quantify the performance of the EPC components, we benchmark our load balancers to ensure that their capacity is higher than that of the EPC components, so that our experiments are not bottlenecked at the load balancer. We use multiple iperf [17] clients and servers to generate traffic to saturate our load balancer. The load balancers rewrite the destination IP address of the packets in various modes of operation (ensuring affinity to a replica, or round robin across replicas) for both UDP and TCP over the iperf traffic. This rewriting simulates the load balancer rewriting headers within the EPC components, by inspecting the TCP/IP headers and payload. The load balanced throughput at the iperf receivers is shown in Table II. Note that UE-level affinity for TCP flows is infeasible to evaluate in our network layer load balancer with iperf, because the payload which contains the UE identifier is not available at the time the TCP handshake messages are being assigned a replica. In all modes, we find that the load balancer is able to handle over 18 Gbps of traffic while running on one core, and is limited by the network throughput of the vhost-daemon of the load balancer VM. Since the throughput of our EPC components is lower in comparison, both load balancers were not the performance bottlenecks in all further experiments.

B. *EPC Control Plane*

We now evaluate the performance of the distributed EPC control plane under various modes of state synchronization. Our RAN simulator simulates multiple concurrent users performing attach and detach requests in a closed loop for the duration of the experiment, and we measure the number of successful registrations completed per second (throughput) and the average latency of the registrations as metrics of the control plane performance. We use a distributed MME with 3 replicas. Figure 4 shows the throughput of the clustered

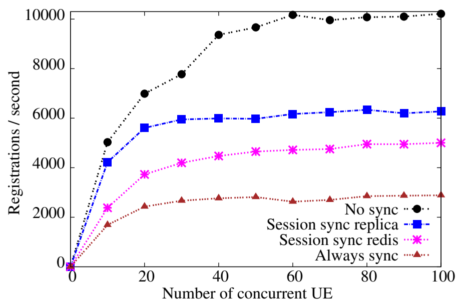


Fig. 4: Control plane throughput of the distributed LTE EPC with increasing load.

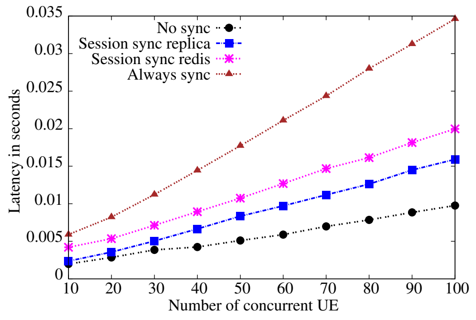


Fig. 5: Control plane latency of the distributed LTE EPC with increasing load.

MME in the following modes: the MME replicas performing no synchronization (*no sync*), the replicas performing synchronization once at the end of attach and detach sessions to the Redis data store (*session sync redis*), replicas synchronizing at session boundaries by replicating state at the other two replicas (*session sync replica*), and the replicas synchronizing with the Redis data store after every step/message of the attach and detach procedures (*always sync*). Figure 5 shows the latency of the attach request in the same modes, as a function of increasing number of concurrent UEs in the RAN simulator. We observe that the performance of the distributed MME is the highest (close to 10K registrations/sec) when the replicas perform no synchronization, and progressively falls with increasing synchronization overhead. The throughput is 51% lower when synchronizing with the Redis data store at session boundaries, 38% lower when synchronizing with other replicas, and 71% lower when synchronizing at every message. The replicas were limited by the CPU processing in all experiments, and we verified that the load balancer and data store components were not the bottleneck. The latency also shows a similar trend. The reduction in throughput can be explained by the time spent blocking for I/O operations to the data store and other replicas during synchronization. From these results, we see that synchronization across replicas for fault tolerance and consistency incurs a steep performance penalty in the EPC control plane, though synchronizing at session boundaries presents a good middle-ground.

Next, we look at how the performance of the various designs scales with increasing number of replicas. Figure 6 presents

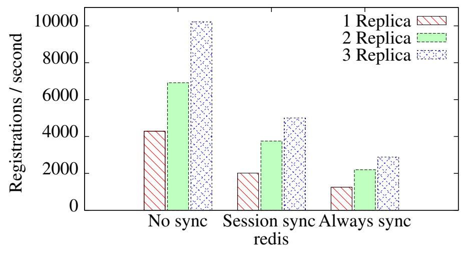


Fig. 6: Control plane throughput of distributed LTE EPC.

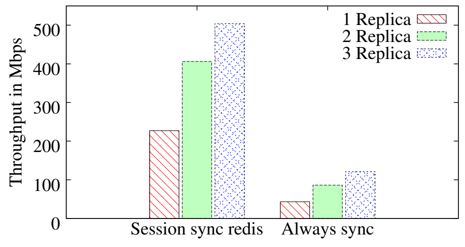


Fig. 7: Data plane throughput of distributed LTE EPC.

the saturation throughput of the three distributed designs with 1, 2, and 3 MME replicas. We find that all designs scale reasonably well with increasing number of replicas, with the session based synchronization providing the best scaling (1.8× with 2 replicas, and 2.5× with 3 replicas).

C. EPC Data plane

We now evaluate the performance of the EPC data plane under various state synchronization modes. We generate data through the EPC by spawning iperf processes in the simulated UE threads in the RAN simulator. We measure the amount of data traffic reaching the sink through the SGW and PGW. All simulated UEs attach before start of the experiment, to ensure no interference from control plane traffic.

Figure 7 shows the saturation throughput of the three distributed designs with 1, 2, and 3 S/P-GW replicas, in the session sync and always sync modes. In the session sync mode, the SGW and PGW replicas incur state synchronization overhead only at the start of a TCP flow to fetch forwarding state from the data store, but cache the result locally at the replica for the duration of the TCP session. In always sync mode, the replicas are stateless and fetch forwarding state for every data packet of the user's TCP flow. One can see from the figure that stateless replicas (and a simple load balancer that ensures no session-level affinity) imposes a huge performance penalty in the data plane performance, resulting in 76% lower saturation throughput in the case of 3 SGW/PGW replicas. Therefore, caching forwarding state at data plane gateway replicas is hugely beneficial for performance. The gateway replicas were limited by CPU in all experiments. The results for when the replicas incurred no synchronization overhead were similar to the case when synchronization was done once at the start of the session.

VI. Conclusion

Our work compared the performance of several designs of a clustered LTE EPC, where the different designs differed in the frequency of state synchronization across the various VNF replicas. Our results show that frequent state synchronization and checkpointing using a high performance datastore imposes a huge performance penalty (71% in the control plane, and 76% in the data plane, as compared to no synchronization), though it provides the best possible fault tolerance guarantees. Maintaining all user connection state locally, while providing the highest performance, will result in significant disruptions when replicas of the distributed system fail. A reasonable middle-ground seems to be to fetch and store shared state at session boundaries (e.g., end of attach procedure in the control plane, and beginning of TCP flow in the data plane), which incurs a somewhat lower performance penalty (51% lower throughput), and somewhat weaker fault tolerance guarantees (replica failure requires restarting the ongoing session), but balances both performance and fault tolerance concerns. We have released the source code of our distributed EPC designs [18], in the hope of furthering research on distributed virtualized network appliances for telecom networks.

References

[1] IITB, "Nfv_lte_epc_1.0," https://github.com/networkedsystemsIITB/NFV_LTE_EPC, 2016.
[2] X. An, F. Pianese, I. Widjaja, and U. Gunay Acer, "Dmme: A distributed lte mobility management entity," *Bell Lab. Tech. J.*, vol. 17, no. 2, pp. 97–120, Sep. 2012.
[3] A. Banerjee, R. Mahindra, K. Sundaresan, S. Kasera, and J. Van der Merwe and Sampath Rangarajan, "Scaling the lte control-plane for future mobile access," in *Proc. of CoNEXT'15*.
[4] Y. Takano, A. Khan, M. Tamura, S. Iwashina, and T. Shimizu, "Virtualization-based scaling methods for stateful cellular network nodes using elastic core architecture," in *Proc. of CloudCom'14*.
[5] G. Premsankar, K. Ahokas, and S. Luukkainen, "Design and implementation of a distributed mobility management entity on openstack," in *Proc. of CloudCom'15*.
[6] A. Basta, W. Kellerer, M. Hoffmann, H. J. Morper, and K. Hoffmann, "Applying nfv and sdn to lte mobile core gateways, the functions placement problem," in *Proc. of AllThingsCellular'14*.
[7] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, "Split/merge: System support for elastic execution in virtual middleboxes," in *Proc. of NSDI'13*.
[8] S. Rajagopalan, D. Williams, and H. Jamjoom, "Pico replication: A high availability framework for middleboxes," in *Proc. of SoCC'13*.
[9] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. a. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker, "Rollback-recovery for middleboxes," in *Proc. of SIGCOMM'15*.
[10] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *Proc. of NSDI'17*.
[11] "Linux virtual server," http://www.linuxvirtualserver.org/.
[12] "LevelDB," http://leveldb.org/.
[13] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, no. 124, pp. 5–, Aug. 2004.
[14] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang, "The ramcloud storage system," *ACM Trans. Comput. Syst.*, vol. 33, no. 3, pp. 7:1–7:55, Aug. 2015.
[15] "Redis," http://redis.io/.
[16] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proc. of SoCC'10*.
[17] "Iperf," https://iperf.fr/.
[18] IITB, "Nfv_lte_epc_1.1," https://github.com/networkedsystemsIITB/NFV_LTE_EPC/tree/master/NFV_LTE_EPC-1.1, 2016.