Scalable and Resource Efficient Control Plane for Next Generation Cellular Packet Core

ABSTRACT

With the increase in cellular-enabled IoT devices with diverse traffic characteristics and service level objectives, handling control traffic in a scalable, resource-efficient manner in the cellular packet core network is critical. We focus on the most crucial control plane component of the core network, the *Mobility Management Entity (MME)*. Traditional monolithic design of the MME is inflexible with respect to the diverse requirements and bursty loads of IoT devices, specifically for properties such as elasticity, customizability, and scalability. We present a design approach called `MMLite` based on the concept of *stateless microservices*, where microservices are used to implement individual control procedures and the states are decoupled from the processing. We also develop the related state migration and fault tolerance frameworks, and design inter- and intra-host load balancing approaches based on skewed consistent hashing to efficiently distribute incoming connections. We build the `MMLite` prototype using DPDK and OpenNetVM frameworks and demonstrate its performance potential with respect to raw throughput, fault tolerance, seamless scaling, and load balancing.

1 INTRODUCTION

One of the grand challenges in the design of the future cellular core network is its efficient scaling with the projected growth of signaling or control traffic. Much of it is expected to come from the tremendous growth of IoT devices (≈12 billion by 2022 [10, 13]). Compared to traditional user equipments (UE) such as smartphones, IoT devices generate at least twice the volume of control messages, growing 50% faster than the data traffic [2, 11, 19, 30, 40]. Control messages do not directly contribute to the revenue of cellular service providers, and thus represent a significant overhead. Worse, the traffic characteristics and performance requirements of cellular-based IoT devices are very different from that of other types of UEs [3, 14, 27, 29, 33, 40].

An immediate concern now is the scalability and efficient resource utilization in the cellular core network - also called the *Evolved Packet Core* or EPC in connection with the LTE networks. Our specific interest in this work is *Mobility Management Entity* or MME. MME is the most intensive control plane component in EPC and handles 5 times more control messages than any other entity in EPC [31, 45]. Our main concerns are the following three issues:

(1) *Elasticity*: IoT applications create bursty traffic [21, 40], necessitating dynamic capacity provisioning. The insufficient capacity at MME may lead to connection failures and rejections from MME, triggering retry messages and further increasing the load on the MME [14]. The current practice of over-provisioning the MME to handle such scenarios runs counter to the interest in maintaining a low price point for cellular connectivity for IoT devices. Further, UEs and all entities inside the EPC maintain contextual information specific to their association of connection from UE called *static bindings*. This makes it difficult to migrate connections to other MMEs in case of overloads and failures.

(2) *Customizability*: Different UEs, specifically IoT devices, can have very different requirements [40, 43]. For example, IoT devices in smart cars require stringent Service Level Objectives or SLOs to react to changing traffic conditions, while smart home IoT devices may simply require IP connectivity. Unfortunately, today's cellular networks make use of monolithic MME devices which are rigid and do not possess the needed functional or performance flexibility.

(3) *Scalability*: A key bottleneck for large-scale networks is the centralized load balancing mechanism that must immediately assign incoming connections to an MME. Given the heterogeneity in the entire ecosystem, traditional approaches such as round-robin or least number of connections will no longer be effective [23]. While recent approaches based on consistent hashing (CH) are scalable [3, 36], they are static in nature (i.e., unaware of the traffic characteristics or load conditions of the MME), and can lead to "hot spots" whereby a few MME hosts are overloaded. Meeting user-specified SLOs while being scalable and resource efficient will thus require a careful reconsideration of load balancing decisions in the network.

To address the above limitations we propose "`MMLite`", an agile MME architecture that exploits recent advances in network functions virtualization for responsiveness to the scaling and customization needs outlined above. The key enabler of `MMLite` is its *stateless* design that decouples the MME from the UE context and static bindings by externalizing it in shared memory. This allows agile provisioning of MMEs in response to changes in cellular traffic.

To address customizability, we *decompose* the MME functionality into a set of microservices (or NFs) based on the specific control procedure they handle, such as attach, service, handover, migrate, etc. This decomposition allows us to customize MMEs based on the requirements of the UE. We support differentiated service by assigning *slices*, i.e., a dedicated set of NFs with a physical resource, to incoming connections based on their SLO requirements. The performance of slices is managed by introducing NF-level priorities and regulating their resource allocation. Finally, to address scalability, we develop a novel SLO-aware and slice-aware MME load balancer and MME forwarder that optimizes the resource utilization within and across hosts.

We implement MMLite using the Intel DPDK [17] and OpenNetVM [32] frameworks that provide high performance and low latency packet processing capabilities required to build a scalable and high performance architecture. The DPDK library provides zero copy architecture to store the packets in shared memory. We extend this architecture to build stateless MME microservices by externalizing their states from processing and other fast path components such as MME load balancer and MME forwarder. Performance evaluations show that while the the stateless, functionally decomposed design of MMLite provides a raw throughput at par with the traditional monolithic stateful design, MMLite provides the anticipated scaling benefits, can satisfy stringent SLO requirements, provides near-optimal load balancing, and results in better resource utilization.

In summary, we make the following contributions:

- We demonstrate the performance limitations of current generation monolithic MME design that uses static binding between state information and host machines (§2.2).
- We propose a design approach based on functional customization and statelessness that uses the concept of a slice as a unit of resource provisioning (§4).
- We develop a load balancing approach based on skewed consistent hashing that takes into account the SLO requirements and dynamic load conditions of hosts and NFs (§5).
- We develop a prototype using Intel DPDK and OpenNetVM frameworks (§6) and demonstrate the performance potential in terms of raw throughput, fault tolerance, scaling, and resource/SLO-aware load balancing (§7).

2 BACKGROUND & MOTIVATION

We provide a primer on current generation LTE architecture and highlight the challenges related to elasticity, customizability and scalability for the MME functionality. This sets the stage for our work.

2.1 Basics of LTE Architecture

Present day LTE network has two main components: Radio Access Network (RAN) and Evolved Packet Core (EPC). See Figure 1. The User Equipment[1] (UE) communicates with Internet through eNodeB (another name for the Base Station) of RAN via EPC. The Serving Gateway (SGW) is responsible for forwarding the packets between the eNodeB and Packet data network Gateway (PGW). PGW provides packet data services to UE such as QoS (Quality of Service), packet filtering services, and billing. The Mobility Management Entity (MME) provides control plane functionality, specifically to establish data sessions and releasing them. MME verifies the subscription details of a user for authentication and maintains the control channel with eNodeB for exchanging the control information. MME verifies the subscription details of an UE for authentication with Home Subscriber Server (HSS). Since MME is the centerpiece of our work, we discuss the control procedures executed in the MME in more detail below.

[1]In our work, we use User Equipment (UE) and Cellular-enabled IoT device interchangeably.

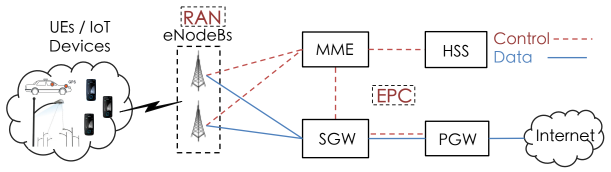


Figure 1: LTE architecture with key components and connection specific mapping details

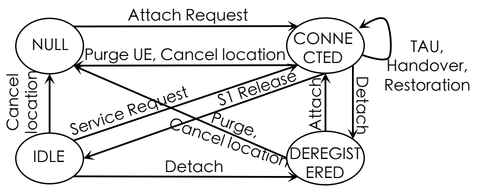
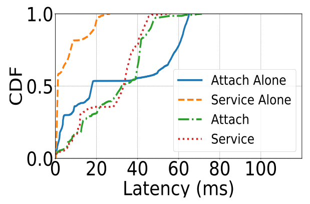
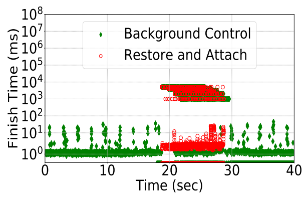


Figure 2: MME connection state machine

Control Procedures and Bindings: Figure 2 shows key MME states and control procedures associated with each of the states. The *Attach request* is issued by UE to register itself with EPC for Internet connectivity. This is an infrequent procedure that is invoked a few times per day per UE. The *Service request* procedure is performed when an inactive UE in Idle state wishes to send or receive data. This is a frequent procedure in LTE (e.g., 45 requests per hour per UE during busy hours [26]). The *Handover* or *TAU* (Tracking Area Updates) procedures handle mobility and responsible for migrating each device's association and states. The TAU procedures are also used for migrating the connections in case of overloads or MME scaling. LTE also supports other control signaling procedures for providing necessary services to the cellular devices [35]. But the procedures described above are the most frequently used and we concentrate only on these in our work. In an ideal scenario, these procedures take tens or few hundreds of milliseconds to complete; in case of overload or failures, these procedures can take seconds or even minutes to complete [4, 5, 22]. Typically, each invocation of



(a) Interference among control procedures



(b) Snowball effect after failure

Figure 3: Experiments demonstrating limitations

a control procedure triggers a sequence of control messages between UE, eNodeB, and EPC.

When the initial connectivity is established via *Attach request* the MME establishes the necessary state information that it uses to process subsequent control procedures from the UE. UE and MME retain these association details (called *static bindings or associations*) until the UE is completely detached from the core network. In this association, each UE maintains identifiers such as GUTI (Globally Unique Temporary UE Identifier), which contains: (*i*) TMSI (Temporary Mobile Subscriber Identification) for temporarily identifying the current UE session until it is detached from the network and (*ii*) MME identifier (i.e., MMEID-UE-S1AP). Similarly, the MME maintains the necessary contextual state information specific to that connection (such as authentication and security keys), TMSI and other session details. The TMSI information is used by MME to subsequently identify the UE connection. The control packets from UE to MME includes MMEID that identifies the MME that has the necessary contextual information for this UE. These static bindings hinder elastic scaling. Only that MME that has the state for a specific UE must handle all control signaling for the UE.

2.2 Limitations and Challenges

The rigid design of MME faces elasticity, customizability and scalability challenges in the face of two factors: 1) diverse nature of IoT traffic e.g., sporadic traffic, periodic and predictable traffic, high-frequency triggers, and busty traffic [40, 43] and various SLO requirements, 2) much larger number of devices to be supported with low per-unit cost. We describe the key challenges below, sometimes highlighting them with performance numbers collected from our testbed described in later in §6.

Unreliable Overload Protection: In case of bursty IoT traffic [21, 40], the stateful static bindings makes it difficult to migrate connections in case of MME overload. Current overload protection methods are [7, 8]: 1) *Migrating connections* from an overloaded MME to other lightly loaded MMEs by triggering the UE to migrate the connection requests thus necessitating a large amount of control messages for reestablishing the UE's context and 2) *Rate throttling* by the overloaded MME by either dropping or rejecting the control messages beyond a certain limit. In an experiment using the testbed in §6 a stateful MME operating at 80% CPU utilization when attempting to migrate some or all of its connections to another MME has $> 30\%$ of its control procedures incurring significantly higher latency, resulting in $> 50\times$ data transfer instantiation times (from 100ms to 5s). We observe similar results with *rate-throttling* approach due to the reconnection attempts made by the UE following dropped messages.

Need for Functional Decomposition/Isolation: Control procedures from one device may interfere with the processing of control procedures from other devices, resulting in service degradations and SLO violations for these devices. Figure 3a shows this scenario where, the heavier *Attach Request* is affecting the lighter *Service Request* by more than 50%. Thus there is a need to isolate the performance of various control procedures across various UEs so that SLOs could be satisfied subject to resource constraints. This also allows for flexibility in case of new SLO models that could be upcoming.

Fault Tolerance: MME failures could result in service outages of up to 10s of minutes [4]. Current LTE networks address fault tolerance in two way [16, 25]: *i*) Active-Passive High Availability via $N + 1$ (N Active, 1 Passive) resiliency and *ii*) session restoration procedures. The $N + 1$ resiliency approach requires additional hardware and cannot handle multiple MME failures. The second approach is more efficient in this regard. Here, a *session restoration server* maintains UE session information of each of the MMEs [25]. In case of MME failures, the session restoration server redistributes the UE session information pertaining to the failed MME servers among other active MMEs. The active MMEs will trigger the affected UEs to reattach to the network to reassociate the UEs with new MME servers through session restoration procedures, still resulting in a large number of control procedures flooding the core network. Any session restoration procedure that fails to complete with in 5 seconds will retry with reattach procedures (i.e., 3GPP specific timer and retry duration [24]), resulting in additional attach floods (*snowball effect* [14]). Figure 3b) shows that UEs take on the order of seconds and even minutes to reattach to the network.

Resource- & SLO-aware Load Balancing: Common approaches for distributing MME connections among UEs [23] are via weighted round robin (W-RR) or consistent hashing (CH). In these approaches downscaling the number of MMEs (e.g., when the load is low) means that idle UEs need to woken up to reestablish sessions using other MMEs. This generates a large number of control messages. Also, consistent hashing [3] uniformly distributes UEs among MMEs, but this may not be fair in terms of the load and may generate hot spots and SLO violations.

3 SYSTEM OVERVIEW

From the description above, it is apparent that there are two core issues: 1) the stateful nature of MME and the static binding it engenders make moving UE connections from one MME to another significantly resource intensive, and 2) the current monolithic design is contrary to the need for functional decomposition and performance isolation. Our design of `MMLite` thus fundamentally uses two core design principles - 1) *Statelessness* and 2) use of *functionally decomposed microservices*. These principles are used in conjunction with *slicing* - a unit of physical resource that procedures needing specific SLO requirements are mapped to. A *load balancer* is responsible for mapping of control messages to MME hosts/slices.

Thus, the `MMLite` architecture introduces the following functional components (Figure 4):

- *Stateless MME Microservices*: We decompose the MME functionality into individual network functions (NFs) that handle specific LTE control procedures. The NFs and their functionality are implemented as microservices. The state is maintained externally making these microservices stateless (see §4).
- *Slices*: The microservices are bundled into 'slices', with each slice hosting a given number of microservices for different control procedures. Slices are units of physical resources, such as a fraction of a logical execution unit of a processor (lcore [37] in Intel DPDK-speak). Multiple slices can run on a single MME host machine. Many such hosts are possible within the carrier's datacenter.
- *Load Balancing*: The load balancer distributes load both within and across MME hosts in a resource-efficient fashion. It has two functional components: (*i*) *MME Load-Balancer*: An external entity that distributes control messages from UEs across multiple MME hosts on the basis of their resource and SLO requirements; and (*ii*) *MME Forwarder*: An NF-based forwarding entity on each MME host that distributes control messages to suitable MME NFs on the basis of the SLO requirements of UE's control connections. The details of inter-host MME load balancing and intra-host NF-level control traffic distribution are presented in §5.

For providing the necessary logic and infrastructure support for above mentioned key functional capabilities `MMLite` supports two different controllers: (*i*) *NF Controller*: A controller local to each MME host for managing the externalized states, state migration, and NF scaling; and (*ii*) *EPS Controller*: A centralized controller infrastructure that manages

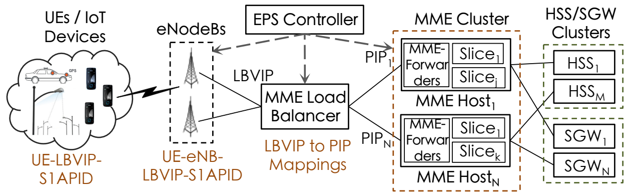


Figure 4: `MMLite`: High Level LTE Cellular Core System Architecture

the MME hosts scaling on the basis of SLO violations and resource requirements.

Overview of Operation: All control procedures are implemented as independent NFs running as microservices. The states are external to the NFs and are maintained in the shared memory inside the MME hosts. There are number of such hosts actively executing MME functions. Additional hosts are available from an idle pool to respond to any immediate needs such as faults or load balance requirements. The states corresponding to each UE/slice are assumed to be replicated among a subset of the MME hosts (also called *replica hosts*). More on this in §4.2. This helps different hosts work on the same connection at different times as dictated by the MME load balancer.

The load balancer steers all control packets for the MME to the right MME host. Typically, the very first packet of any new invocation of a control procedure is used to choose the host that can satisfy the SLO requirements from the set of replica hosts. Subsequently all packets for the same invocation goes to the same host. This is tracked by the identifier `mmeID` maintained by the load balancer.

Inside each MME host, a dedicated forwarder and a set of NFs are assigned to each slice. The slice is provided with predetermined amount of resource (i.e., fraction of an lcore) as mentioned before. Control messages for the same invocation of a control procedure are processed in the same slice. This is tracked using `sliceID` by the forwarder. Finally, during execution of the NF the `TMSI` value in the message identifies the UE and thus its state in the shared memory of the host.

The load balancer uses *skewed consistent hashing* for assigning MME hosts to slices. More on this in §5. The EPS controller infrastructure helps the `MMLite` architecture to evaluate the performance of the control messages served at different slices and hosts and scale accordingly by adjusting the number of MME hosts and/or the number of NFs assigned to a slice within an MME host. We discuss the important components of `MMLite` in detail in the following sections.

4 STATELESS MME MICROSERVICES

In this section, we describe the core of the MME design that provides two main functionalities: 1) elastic scaling and 2) functional decomposition. These are achieved by decomposing the MME NFs on a dedicated basis for each control

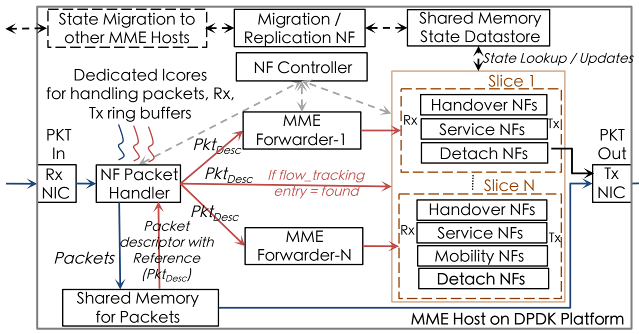

State Migration to other MME Hosts
Migration / Replication NF
Shared Memory State Datastore
State Lookup / Updates
NF Controller
Dedicated lcores for handling packets, Rx, Tx ring buffers
PKT In
Rx NIC
NF Packet Handler
Pkt_{Desc}
MME Forwarder-1
Pkt_{Desc}
If flow_tracking entry = found
Pkt_{Desc}
MME Forwarder-N
Packets
Packet descriptor with Reference (Pkt_{Desc})
Shared Memory for Packets
Slice 1
Rx
Handover NFs
Service NFs
Detach NFs
Tx
Slice N
Rx
Handover NFs
Service NFs
Mobility NFs
Detach NFs
Tx
PKT Out
Tx NIC
MME Host on DPDK Platform

Figure 5: Block diagram of key `MMLite` components running on a single MME host

procedure thereby reducing the interference among them. Second, we make the MME stateless by decoupling the states from the NFs. The description below refers to the Figure 5 that describes the architectural components.

4.1 Functional Decomposition

Functional decomposition targets two issues. First, the control procedures that need to be invoked are temporally distributed in an unpredictable fashion. Some procedures such as *Attach* are infrequent, while *Service* procedures could be more frequent. The latter may even exhibit periodic, synchronous or semi-synchronous behavior (e.g. IoT sensors) [1]. Second, some IoT devices have limited functional needs and do not require certain types of control functions such as Handover, TAU-based state migration, QoS procedures and so on (e.g., stationary IoT sensors). Some IoT devices may be very dependent on certain types of control functions (e.g., IoT devices on smart transport platforms may invoke significant mobility related control functions). Mapping of individual control procedures to microservices allow for only specific microservices to be instantiated depending on load conditions and SLO requirements.

Thus, `MMLite` decomposes the traditional monolithic MME into following set of microservices targeted to handle specific control procedure: *a*) *Attach request*, *b*) *Service request*, *c*) *Detach request*, *d*) *Handover request* and *e*) *State management* microservice-based NFs. A full-fledged implementation will cover other control procedures as well. We cover only these in our prototype for demonstration as they cover the most frequently used procedures.

4.2 Statelessness

`MMLite` externalizes the states of all MME NFs inside a host i.e, the states are maintained outside the NF in the shared memory inside the MME host. We use *NF Controller* to allocate two shared memory pools: One for storing packets for zero-copy architectural support and other for storing the UEs states. These memory pools are used later by the MME NFs to store the packets and to get the UE context information.

When a control packet arrives at the NIC of the MME host: (*i*) The *NF Packet Handler* interfaces with the DPDK platform's poll mode driver to bypass the operating system to DMA, which can be readily accessed by all the NFs. (*ii*) The other *NF Packet Handler* threads access the packets stored in the shared memory to create a *packet descriptor* for each packet, which includes the handler to the packet in the shared memory and details on how the packet need to be handled inside the host by different services. The *packet descriptors* are then placed onto the RX queues of the MME-forwarders for distributing the packets further across MME NFs depending on the slice the packet belongs to. This is facilitated by a set of hash tables maintained in the host that maps slice ids to forwarders, control message types to SLOs and UEs (identified by TMSI) to its state.

Multiple NFs handling different procedures run on each slice. The mechanism of dynamically choosing the *viable* MME NF (i.e., which satisfies the SLO requirement of this control message in a resource efficient manner) by the MME-Forwarder is described in the Section 5.2 and respective intra-host load balancing algorithm is described in the Algorithm 2.

State Migration: State migration support is helpful to both load balancing and fault tolerance. This is assisted by the *state migration utility.* The migration utility helps maintain up-to-date copies of the states on other hosts (replica hosts). Replication of states and number of replicas may be limited to those with tighter SLO requirements or other priorities in order to conserve resources. Similar to prior work [36], we maintain two level state storage, i.e., the portion of UE's state that is frequently updated and portion of state that are less frequently or not updated at all. Partitioning the states in this manner allows us to efficiently consolidate while migrating the states. There are two key mechanisms:

(*i*) *Cold Migration*: The UE contextual information from within a MME host is migrated to other replica hosts only upon completion of the complete control procedure. Upon completing the procedure MME marks each UE contextual information for migration. The state migration utility consolidates the states belonging to the same slice together and sends it to the other replica hosts.

(*ii*) *Hot Migration*: In case of hot migration, each time a message is handled by MME, the specific context (this is just a part of the state) is marked for migration because it could be updated. The state migration utility continuously polls and migrates those parts of the state that are marked for migration. Though this approach provides better fault tolerance compared to the cold migration but increases the volume of migration traffic inside the core network.

Fault Tolerance & Scaling: We have seen before in §2.2 that in case of conventional stateful MME failure, the UEs that are already attached to the failed MME host are redistributed to new MME hosts. This triggers an avalanche of restoration and reattach procedures to change UEs' MME associations significantly affecting performance. `MMLite` avoids this with the following mechanisms.

(i) Host Failure: The EPS Controller continuously monitors the MME hosts (i.e., with heartbeat messages) for failures and performs following tasks: *i*) It updates the load balancer about such MME host failures and *ii*) provisions resources to each slice of the failed MME host by adding the new MME hosts into the consistent hash ring of that slice with the same amount of resources. If the existing set of hosts does not have sufficient resources, new MME hosts are brought up from an idle pool, for example. If state replicas are available corresponding to the slices of the failed hosts (see above), these replicas are migrated to the appropriate hosts. Otherwise, the UEs must invoke reattach, but with proper load balancing the avalanche effect would be avoided.

(ii) NF Failure: The NF controller monitors NFs and helps in instantiating new MME NFs from an idle pool of MME NFs. The idle MME NFs are maintained to reduce the new MME NF instantiation times. In case of NF failure, the NF controller invokes the *NF Packet Handler* thread that is dedicated to handle the failure scenario. This handler registers an MME NF in the idle MME pool with NF controller and reassigns the packets already in the RX queue of the failed NF to that of the new NF. (Note that the packets themselves are in the shared memory).

5 LOAD BALANCING

A major limitation in the design of current cellular networks is their inelastic nature. Today's MME architectures fail to effectively scale in response to changes in load due to: (*i*) static load balancing used by eNodeBs (e.g., weighted round robin) to distribute traffic across MMEs, and (*ii*) static binding of UEs (User Equipment) and eNodeB with MME. To improve resource efficiency, we first present an inter-host load balancer that determines the *slice- and resource-aware MME host* for the incoming packets. We then present our intra-host forwarder that selects the *SLO-aware MME NF* within the host for serving the packets.

5.1 Inter-Host MME Load Balancer

Our inter-host load balancer uses a *skewed consistent hashing* mechanism to distribute incoming connections to hosts. For fine-grained (slice-level) resource management, we maintain a *separate* consistent hashing ring for each slice based on its `sliceID` (see Figure 6). Thus, each slice can be served by at most a subset of all hosts; however, multiple slices can be served by a single host. Within the slice, we assign a subset of hosts to each UE based on their `TMSI`. To serve the connection, a specific host is chosen from this subset based on its load conditions and the required SLO.

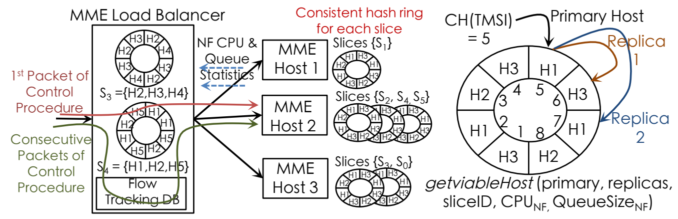

MME Load Balancer
1st Packet of Control Procedure
Consecutive Packets of Control Procedure
S_3 ={H2,H3,H4}
S_4 = {H1,H2,H5}
Flow Tracking DB
NF CPU & Queue Statistics
Consistent hash ring for each slice
MME Host 1
MME Host 2
MME Host 3
Slices {S_1}
Slices {S_2, S_4, S_5}
Slices {S_3, S_0}
CH(TMSI) = 5
Primary Host
Replica 1
Replica 2
H1
H2
H3
getviableHost (primary, replicas, sliceID, CPU_{NF}, $QueueSize_{NF}$)

Figure 6: `MMLite` slice and SLO-aware MME Load balancer architecture.

ALGORITHM 1: Inter-Host Load Balancing.

```
pkt → sliceID, mmeID, GUTI.TMSI, msgID;
msgType ← getProcedureType(msgID);
track_entry ← Hash database of (LTMSI, hostID) ;
if track_entry[LTMSI] exists then
    mmeHostID ← track_entry[LTMSI] ;
else if TMSI ≠ 0 then
    if msgType ≠ "attach" then
        key ← pkt.GUTI.TMSI;
        sliceCHRef ← getSliceCH(sliceID);
        hostReplicas ← lookupHosts(key, sliceCHRef);
        hostMMEID ← getViableHost(hostReplicas, sliceID);
        track_entry[LTMSI] = mmeHostID ;
else
    /* First message of Attach Procedure.    */;
    LTMSI ← rand();
    mmeHostID ← CHGetHost(LTMSI);
    track_entry[LTMSI] = mmeHostID ;
mmeHost ← getHost(mmeHostID);
send packet to mmeHost;
```

Our Algorithm: Algorithm 1 details our slice-aware inter-host load balancing. The first procedure for any UE is the attach procedure. For the first packet of attach (for which the `TMSI`, `mmeID`, and `sliceID` values are not assigned), the load balancer assigns a TMSI, i.e., `LTMSI`. The `LTMSI` is used to calculate the MME host, say $host_0$, to which this message will be sent for service using the hashing ring reserved for `sliceID=0`. After service, $host_0$ assigns the `mmeID` to this connection as part of the reply. The load balancer uses this `mmeID` to directly send subsequent packets of attach to $host_0$.

After successful completion of the attach procedure, the `sliceID` of the UE is resolved. This `sliceID` is then used to select the specific slice-aware hash ring. Within the hash ring, the hash of the `TMSI` value (same as `LTMSI`) is used to select the primary host and a few replica hosts, as shown in Figure 6. The number of replicas can be decided based on the expected load for each UE; in our experiments, we initially assign 2 replicas for each UE. *host*$_0$ then migrates the UE context to the primary and replica hosts so they can serve subsequent procedures from the UE based on the `TMSI`.

For subsequent procedures (after attach), from among the primary and replica hosts, we derive the set of *viable hosts*, i.e, hosts that satisfy the procedure's SLO requirements.

Calculation of viable MME hosts: At a high-level, viable hosts are those that contain at least one NF that meets the SLO requirements of the procedure, say T_{SLO}. To obtain the set of viable MME hosts, we compute the *total estimated time* required by a (primary or replica) host to handle the incoming control procedure. This, in turn, requires the current load statistics at each host. Each MME host propagates the CPU utilization and queue sizes of their NFs to the load balancer. To minimize the overhead of communication, hosts periodically send the *moving average* of CPU utilization and NF queue sizes. In our implementation, this period is set to a few hundred milliseconds, resulting in only a few kilobytes of data overhead on the network.

Let t be the type (e.g., service) of the incoming control procedure. Let T_t be the total completion time required for a type t procedure when handled in isolation on a core (obtained via profiling). Let m be the total number of NFs that handle type t procedures across all primary and replica hosts of `TMSI`. For each NF i, the load balancer is aware of the moving average of queue sizes, q_i, and CPU assigned, c_i. The waiting time for the procedure at NF i is then estimated as:

$$W_i = (q_i \cdot T_t)/c_i \ \forall i = 1, \ldots, m \tag{1}$$

Assuming that the moving average is stable, every message of the procedure assigned to NF i will see a backlog of q_i. If there are p_t messages in a type t procedure, the total backlog experienced by the procedure is $p_t \times q_i$. Since T_t is the time for a procedure when run in isolation (on a full core), we can approximate the time per message as $T_t/(c_i \cdot p_t)$, and thus the wait time is $p_t \times q_i \times T_t/(c_i \cdot p_t) = q_i \times T_t/c_i$. The completion time is then computed as $T_i = W_i + T_t = (\frac{q_i}{c_i} + 1) \times T_t$.

We now find the *viable hosts* as those that contain at least one NF i for which $T_i \leq T_{SLO}$. If no viable hosts exist, then the messages are sent to the host that has the NF with the least q_i/c_i ratio. Such violations are reported to the EPS Controller, which can then add hosts to the slice (see §5.3).

Selection of final host from viable hosts: From among the set of viable hosts, we select the host that is *most loaded*, i.e, the host that has the highest CPU usage. While this may seem counter-intuitive, recall that we are only picking from among viable hosts, each of which has at least one NF that can satisfy the SLO. We prefer the most loaded host to maintain the lighter load on other hosts, which can then be scaled down during low cluster usage. Instead, if we select the least loaded host, over time we will have balanced hosts, making it difficult to identify less loaded hosts to drain connections from in case of a scale down. This *load unbalancing* technique has been shown to facilitate server scaling in web clusters [6, 12]; we evaluate its impact on our scaling in §7.2.

Once the final host is selected for a control procedure, we maintain the connection tracking information for this `TMSI` to forward subsequent messages of this procedure to the same host without having to recalculate the viable hosts. This information is refreshed each time a new control procedure (not message) is seen by the MME load balancer.

ILP Formulation: The inter-host load balancing problem can also be addressed exactly using an Integer Linear Programming (ILP) formulation. We provide the formulation from the point of view of an arriving procedure of type t that must be assigned to a host to meet its SLO, T_{SLO}, while minimizing the number of hosts needed. We will compare our algorithm's performance with that of the ILP in §7.2.

Let there be n hosts, with each host H_k ($k = 1, \ldots, n$) having the set C_k of cores, with $|C_k| = l_k$. Thus, the total number of cores available is $c = \sum_k l_k$. Each core may support several NFs. Let there be m total NFs. Let x_{ij} denote the proportion of core i (out of c) assigned to NF j. Then:

$$\sum_{j=1}^{m} x_{ij} \leq 1, \ \forall i = 1, \ldots, c \tag{2}$$

To satisfy the SLO constraints for the incoming procedure, we have (similar to Eq. (1)):

$$(\frac{q_i}{c_i} + 1) \times T_t \leq T_{SLO} \tag{3}$$

To account for resource usage, let u_{ij} be an integer denoting whether core i hosts NF j. Then $u_{ij} = \lceil x_{ij} \rceil$. The above can be converted into integer constraints as follows:

$$u_{ij} \geq x_{ij}, \ u_{ij} \in \{0, 1\}, \ \forall i = 1, \ldots, c, \forall j = 1, \ldots, m \tag{4}$$

Every NF can be on exactly one core:

$$\sum_{i=1}^{c} u_{ij} = 1, \ \forall j = 1, \ldots, m \tag{5}$$

Let v_k be 1 (0) if host H_k is (not) used. Then,

$$v_k \geq \{u_{ij}\}, \ \forall i \in C_k, \forall j = 1, \ldots, m, \forall k = 1, \ldots, n \tag{6}$$

The ILP's objective is to minimize the number of hosts used:

$$\textbf{Minimize} \sum_{k=1}^{n} v_k \tag{7}$$

5.2 Intra-Host Load Balancing (Forwarder)

Each slice inside an MME host is assigned a dedicated MME forwarder and a number of MME NFs of different service types (e.g., attach, detach, handover). For each service type, there can be multiple NFs depending on the load conditions of that specific control procedure. Once a host is chosen for handling an incoming message (inter-host load balancing), the next task is to decide the specific NF within that host that will serve the request. *MME Forwarders* provide an effective means to load balance the control traffic across multiple MME NFs inside each MME host.

Our intra-host load balancing algorithm is presented in Algorithm 2. Control messages that do not have any `sliceID` details (i.e, attach procedure's messages), will be put on the receive ring buffer of the forwarder that is dedicated to `sliceID=0`. This forwarder distributes the messages to the attach MME NF that has the least queue size. Consecutive messages from control procedures that have a `sliceID` (i.e., procedures from UEs whose attach is successful), will be sent to the *optimal* MME NF, as discussed next.

ALGORITHM 2: Intra-Host Load Balancing.

```
1  $pkt \longrightarrow sliceID, mmeID, GUTI.TMSI, msgType$;
2  $sID \leftarrow$ getServiceID(msgType);
3  $nfIDs \leftarrow$ getNFInstances($sID$, $sliceID$) ;
4  $track\_entry \leftarrow$ Hash database of (TMSI, mmeNFID) ;
5  if $track\_entry[TMSI]$ exists then
6      $mmeNFID \leftarrow track\_entry[TMSI]$ ;
7  else
8      if $sliceID = 0$ then
9          $mmeNFResources \leftarrow$ getNFQueue($nfIDs$) ;
10         $mmeNFID \leftarrow \min\{mmeNFResources\}$ ;
11     else
12         $mmeNFID \leftarrow$ getOptimalNF($mmeNFLoads$) ;
13         $track\_entry[TMSI] = mmeNFID$ ;
14 send packet to NF [$mmeNFID$] ;
```

Determining the optimal MME NF: A common approach to intra-host load balancing is round robin or consistent hashing [3]. However, such strategies do not provide any performance guarantees, and may create hot spots, resulting in SLO violations (see §2.2).

Our approach, by contrast, selects the NF that provides the lowest latency for the incoming procedure. Specifically, we use the same latency model as in Eq. (1), except: (i) we use current queue size at the NF, say Q_i, instead of moving average, and (ii) we use the current lcore allocation (based on updated priorities, see §5.3), say C_i, as opposed to moving average. The current values can be quickly obtained from within the host by monitoring these metrics, and they provide more accurate and timely estimates of current load conditions at the NF. The estimated latency is then:

$$T_i = \left(\frac{Q_i}{C_i} + 1\right) \times T_t, \ \forall i = 1, \ldots, r, \tag{8}$$

where r is the number of NFs in the host. Based on Eq. (8), finding the optimal NF for a given procedure type, t, and a given slice that minimizes the estimated latency is equivalent to finding the NF *opt* such that:

$$opt = \arg\min_{1 \le i \le r} Q_i / C_i \tag{9}$$

If the estimated latency violates the SLO, i.e., $T_i > T_{SLO}$, then, in addition to sending the packet to the computed *opt* NF, the predicted violation is reported to the NF Controller within the MME host. The Controller then instantiates additional MME NFs as needed (see §5.3).

In general, the number of NFs within a host for a given procedure and for a given slice is not too large, and so the optimal NF can be determined without much overhead. However, we find that the overhead of centrally computing the queue statistics and cpu allocation values for NFs in the kernel is infeasible due to the high packet arrival rate. Thus, we customize the MME NF code so that each NF itself maintains the required statistics, considerably alleviating the overhead. To amortize the overhead, we maintain the flow tracking entry for each procedure with TMSI and MME instanceID (`TMSI::instanceID`) once its optimal NF has been determined. Subsequent packets of this procedure can bypass the forwarder and directly go to the *opt* NF.

5.3 NF Prioritization & Resource Scaling

Cellular-based IoT devices, such as (critical) motion sensors and (nonessential) temperature sensors in self-driving car, have heterogeneous traffic characteristics and a wide range of SLO requirements. Efficiently managing the MME resources while ensuring SLO compliance for UEs thus requires careful prioritization and resource management to avoid interfere among connections.

Prioritization: Multiple NFs on a host may be assigned to the same core, creating contention. To provide performance isolation, our `MMLite` architecture leverages CPU-based prioritization and processor scheduling. When a core is assigned to n MME NFs with different priority levels, P_i, we compute, for each NF i, the CPU core allocation fraction, C_i:

$$C_i = \frac{P_i}{\sum_{j=1}^{n} P_j} \ \forall i = 1, \ldots, n \tag{10}$$

P_i ranges from 0 to 1, with higher values representing higher priority. If only one active NF exists on a core, it will be allocated the entire core. Idle NFs are not considered in the C_i calculation. To enforce the C_i allocation, we schedule the core's time slices across resident NFs in proportion to their C_i values in a round robin manner. The priorities are recalculated every time an NF becomes idle or when an NF is added to the core for packet processing.

To assign priorities, we first note the minimum SLO value across users, say SLO_{min}. We then set the priorities for each

NF *inversely proportional* to their respective SLO values, normalized by SLO_{min}. Thus, for an NF with SLO value SLO_i, we set $P_i = SLO_{min}/SLO_i$. For example, consider two NFs a and b that share a core with $SLO_a = 5msec$ and $SLO_b = 10msec$. If $SLO_{min} = 1msec$, we set $P_a = 1/5$ and $P_b = 1/10$. This gives us, from Eq. (10), $C_a = 2/3$ and $C_b = 1/3$. Prior work on shared storage workloads has shown that priorities that are set inversely proportional to performance requirements work well in practice [47]. We evaluate the impact of our priority assignment on performance in §7.2.

Lower and upper limits on C_i may be predefined for specific slices. If SLO violations occur for an NF, we reactively increment its priority by a small fraction (e.g., 0.01). If violations continue to persist, we inform the NF Controller, which may then instantiate additional NFs, as discussed below.

Scaling: Hosts or NFs can be dynamically added for each slice (or user) reactively in response to overload or failures. On the other hand, resources (NFs or hosts) can be removed in response to low utilization. NF-level scaling is carried out by the NF controller within a host (see Figure 5), whereas host-level scaling is carried out at the EPS controller.

We add NFs at a host (or at other hosts if the CPU is saturated) in response to persistent SLO violations that are reported to the NF controller. To minimize the impact of waking up MME NFs, we maintain a pool of idle NFs that can be quickly instantiated, as needed; the overhead of idle NFs is negligible in our experiments. Likewise, we add a new MME host in response to persistent SLO violations that cannot be addressed by NF scaling alone. Further, we also add hosts if the load on all existing hosts is high. For scaling down, we first transition NFs to the idle state and return them to the pool of idle NFs if there are no outstanding messages in the queue. If the pool of idle NFs contains more than a threshold (say, 1, as in our experiments) number of NFs of a given type, then additional idle NFs of that type are turned off. We scale down hosts (or move to sleep or idle state) if the total available CPU across MME hosts is high, say significantly greater than 100%.

When the number of hosts changes, the EPS controller triggers the hosts to recalculate the slice-specific consistent hashing to determine the new set of hosts to migrate the states. When the number of NFs change in a host, our *viable* and *optimal* NF selection strategies will gradually redistribute the queues themselves as they prefer NFs with smaller queue sizes. However, we can redistribute the packets in the receive queues more aggressively, e.g., in response to an NF failure.

6 IMPLEMENTATION

In this section we describe our prototype implementation of `MMLite`. We use the Intel DPDK platform [17] and the OpenNetVM [32] to build our stateless MME microservices and other components of `MMLite`. DPDK boosts packet processing performance by allowing better control over packet processing. OpenNetVM provides a high-performance NFV platform built using DPDK. This platform provides the ability to process packets directly from the NICs allowing the packets to be DMA'd (Direct Memory Accessed) into a shared memory region. NFs can thus directly access packets with no additional copies (i.e, zero copy I/O). We next describe the functional components developed for our prototype. All functional components are built in C language.

UE Emulator: The UE application is built as a multi-threaded program. We tune the linux kernel's parameters, such as *thread-max* and *pid_max*, beyond the default 32K limit to generate up to ≈400K threads which can continually generate traffic to achieve line rates. Each UE is run as an individual thread with the ability to generate LTE control traffic. We adapt the OpenAirInterface OAISIM's approach of bypassing the radio to generate traffic that can be directly handled by the MME [9]. Our UEs generate control procedures such as attach, service, handover, detach, restoration, and TAU-based procedures. The UE emulator allows the user to configure each thread or group of threads to generate traffic with specific characteristics.

MME Load Balancer & Forwarder: We implement our skewed consistent hashing along with round robin (RR) and consistent hashing (CH) mechanisms on DPDK. We use Jenkins hash function [18] as the baseline hashing mechanism for CH and our skewed CH methods.

MME as Microservice: We build our MME using the code base of OpenAirInterface OAISIM [9]. OAISIM is built using the linux networking stack and suffers from scalability because of network dependencies. We remove the network dependencies from the code and build it as an application-level networking using the DPDK platform. We implement the control procedures as microservices with their states externalized to shared memory. The microservice instantiation time is on the order of a few milliseconds compared to tens of seconds with OpenAirInterface OAISIM and OpenEPC [44]. Each MME NF is provided with two different identifiers – service ID and instance ID. The service ID defines the type of the microservice, while the instance ID identifies the specific MME NF inside a host. Each MME, when instantiated, is registered with the NF controller using these identifiers. This allows the NF controller and the MME forwarder modules to identify the MME on the basis of type of service.

State Migration Utility: We implement state migration as a standalone module interfaced with the EPS controller to trigger bulk and device-specific state transfers. Each MME, when handling a control message, updates the state machine for each connection. The states marked for migration are transfered to other MME replica hosts (§4.2) using separate

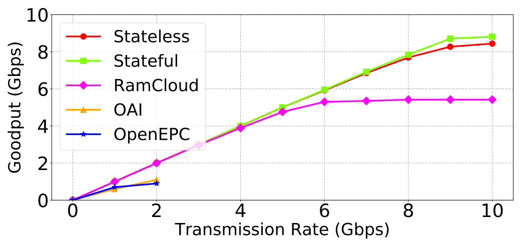


Figure 7: Baseline throughput comparison MME prototypes

threads assigned with dedicated cores. These threads continuously monitor the state updates to perform state transfers to other MME hosts. This utility is designed to perform both hot and cold migrations.

Controller infrastructure: The EPS controller and NF controller modules are designed to work as hierarchical controllers interfaced with the MME NFs, load balancer, and forwarder modules. The controller infrastructure is built with following key functional capabilities: (*i*) initiates the state migration across hosts, (*ii*) performs scaling of resources (MME NFs and hosts) based on observed SLO violations and cpu usage, (*iii*) updates the load balancer with the MME failure and recovery states, and (*iv*) assigns the globally unique LTMSI values to UEs when attached to the network for the first time.

7 EVALUATION

In this section, we demonstrate the performance of `MMLite` and compare it with the conventional MME architecture. We use the following platforms: (*i*) DPDK Compatible Intel Ethernet 10G 2P X520 NIC cards, (*ii*) Dell R710 servers with 48GB RAM, 12 cores (2.6GHz) with Ubuntu 4.4.0-97-generic kernel used as MME hosts, and (*iii*) Dell R710 servers with 48BG RAM, 12 cores and 10G Mellanox InfiniBand adapter integrated with RAMCloud infrastructure for centralized data store. The testbed has a UE emulator host interfaced with two 10Gbps NICs to the MME load balancer. The MME load balancer interfaces with multiple MME hosts and EPS Controller using 10Gbps NIC each.

7.1 Baseline Comparison

We compare the throughput of `MMLite` with the following prototypes: 1) *OpenAirInterface (OAI)*: OAI is the most widely used open source EPC implementation [39]. We benchmark the performance and scale of OAI in the OAISIM mode [9], where the UE and eNodeB are integrated together into a single node, bypassing the radio interface. 2) *OpenEPC*: We benchmark OpenEPC using the PhantomNet testbed [44]. 3) *RAMCloud-based MME*: We customize our DPDK-based MME code to use the RAMCloud-based centralized data store model. RAMCloud [38] stores all the data in DRAM allowing the remote servers to access the RAMCloud data objects

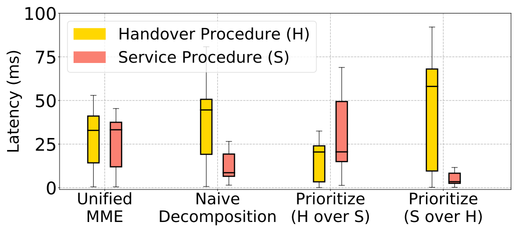


Figure 8: Functional decomposition and prioritization of control procedures

with low latency (as little as 5μsec). 4) *Stateful DPDK-based MME*: The stateful DPDK model implementation uses the same code base as the stateless implementation of `MMLite` .

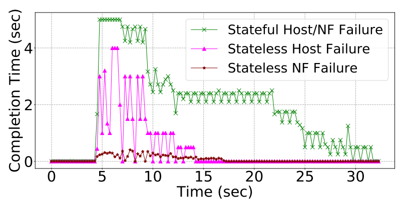
Figure 7 summarizes the goodput for the above approaches relative to our stateless DPDK-based approach in `MMLite` with increasing load. Note the poor scaling of OAI and OpenEPC/PhantomNet due to limitations of these platforms as available to us. The stateless and stateful models provide very similar throughput and only saturate close to the line rate (10 Gbps). The stateless model does take a slight performance hit at saturation and beyond. One can ignore this small impact as MMEs are unlikely to operate at such saturated loads.

7.2 `MMLite` Performance

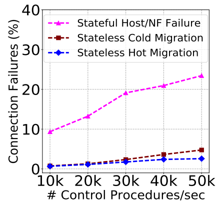
We now show the performance of our `MMLite` prototype. We use the following different types of traffic characteristics generated using our UE emulator: `T1`, a constant rate of control procedures; `T2`, a steady increasing rate of control procedures; `T3`, traffic rate from each UE using a Markov modulated Poisson process.

Functional Decomposition & NF Prioritization: In this experiment, we illustrate the benefits of functional decomposition with two different types of control procedures i.e., *handover request* and *service request*. We generate traffic traces from UE emulator with a mix of *service request* and *handover* procedures from multiple UE threads. We use `T1` traffic pattern at a rate of 1000 requests/sec.

Figure 8 shows the latency of the two control procedures in different scenarios. First, as expected the unified stateful MME has an average of ≈35ms of latency for each procedure. Though the average latency for handling the *service request* is a lot smaller (i.e., < 10ms), it is impacted when the procedures are unified given the more significant processing requirements of *handover requests*. In the second scenario with our naive decomposition, the interference induced by *handover* over the *service request* procedure is alleviated. The service request procedure latency is brought back to the ideal case of < 10ms. However, there is an increase in the *handover* latency since the *service request* NF stays idle after finishing the control procedures destined to it, wasting resources.



(a) Latency of control procedures after failure



(b) Connection Drops

Figure 9: Demonstrating fault tolerance

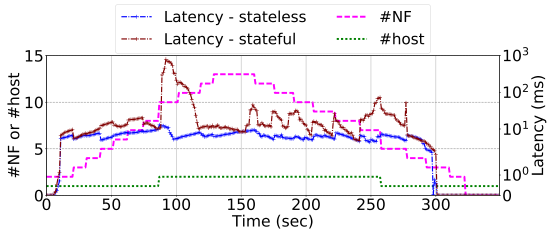


Figure 10: Demonstrating scaling: average latency of control procedures with increasing/decreasing load and scaling of NFs and hosts

The latency can be further optimized by effectively setting the NF priorities when sharing CPU resources. Figure 8 shows that the *handover* procedure can be brought down to ≈ 20ms average latency while sacrificing the *service* latency by ≈10 ms. Alternatively, we can prioritize *service* procedure NF to get an enhanced priority compromising the *handover* latency. These results showcase how tighter SLOs can be obtained by appropriately allocating resources in the decomposed implementation in `MMLite`. Note that such prioritization is not possible for the unified MME design.

Fault Tolerance: Figure 9(a) demonstrates our MME failure handling with three MME hosts using `T1` traffic. We suspend one of the MME host's NFs at about the 5 sec mark to emulate MME failure. The traditional stateful MME uses the same state restoration procedure for NF or host failures. We thus observe the same performance in case of NF and host failure for stateful MME. We see that the control procedure latency shoots up to 5 secs (which is the maximum UE retry time) in response to failures for the stateful MME. This is because during failures, the UE retries for the connection every second. If the UE fails to get a response to its retries within 5 secs, it generates reattach procedure and drops existing data connections associated with it.

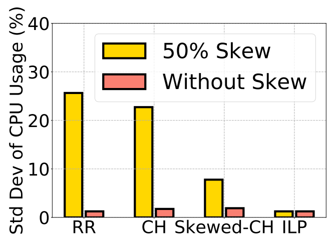
Unlike stateful MME, `MMLite` handles host and NF failures differently as discussed in §4.2. `MMLite` needs state migration in case of host failures, but this does not involve UEs. Figure 9(a) shows that `MMLite` is far more responsive after failures and recovers quickly. The average latency of the control procedure is < 0.5 seconds with NF failure and upto 2.5 seconds with host failure. Clearly, the performance recovery is significantly better for `MMLite` .

Figure 9(b) demonstrates similar failure scenarios, but now shows the number of connection failures right after the fault. We perform multiple experiments at different loads. We see that `MMLite` significantly reduces the number of failures in all cases compared to stateful MME, for both cold and hot migration approaches (§5).

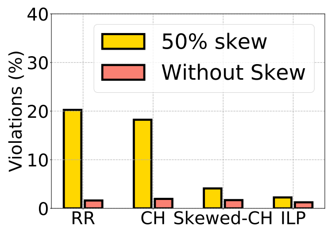
Scaling: To evaluate scaling, we use `T2` traffic to gradually increase traffic until a maximum point (from 2.5 Gbps to about line rate) and then reduce it gradually. As shown in Figure 10, our stateless `MMLite` seamlessly scales the number of NFs and hosts, as needed, in response to the changing traffic. Further, the resulting latency is much lower for stateless when compared to stateful; note that the latency is shown on a log scale. The latency specifically spikes for stateful MME when the number of hosts is scaled up (around 90 s mark) or scale down (around 260 s mark). This is due to the (resource intensive) TAU-based load rebalancing and state migration across hosts that is required for stateful MME. During these periods, the latency for stateful MME is about 50-100× higher than stateless MME.

Resource Utilization and SLOs: To evaluate our load balancing schemes from §5, we use `T3` traffic and skew the load among UE connections. We first compare RR (round robin), CH (consistent hashing), and the ILP balancers with our skewed-CH inter-host load balancer. Figures 11a and 11b show the standard deviation of CPU utilization across hosts and the resulting SLO violations for different balancers when using 3 MME hosts. For uniform load across UEs (no skew), all schemes perform well, as expected. However, for skewed load (wherein the load for 50% of the connections was increased significantly), RR and CH have very high deviation in CPU load across hosts as they simply try to balance the number of connections per host, as opposed to balancing the load per host. This also results in significant SLO violations, about 20% for RR and 18% for CH. By contrast, our skewed-CH results in about 8% standard deviation and only about 3-4% SLO violations. These numbers are almost a factor 3× and 5× lower compared to RR and CH with respect to CPU deviation and SLO violations, respectively. Compared to the optimal ILP, skewed-CH is within 1% of the SLO violations and within 5% of the CPU deviation.

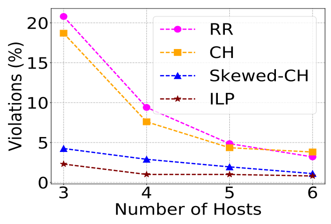
Figure 11c further analyzes the SLO violations when the number of hosts is increased. While the violations decrease with number of hosts for all load balancers, we see that RR and CH continue to have significantly higher SLO violations compared to skewed-CH. Further, as the number of hosts increase, skewed-CH starts to approach ILP; we believe this is because skewed-CH has more opportunities to find viable hosts with larger cluster sizes.



(a) Standard deviation of CPU usage among hosts.



(b) SLO Violations for different skews.



(c) Violations vs. no of hosts for different schemes.

	SD Resource (%)	SLO Violation (%)
Inter + Intra	7.85	4.25
RR + Intra	23.7	6.92
Inter + RR	9.25	12.35
RR + RR	25.75	17.89

(d) Inter- and intra-host load balancing evaluation.

Figure 11: Resource utilization & SLO violations for different load balancing schemes. We outperform RR and CH load balancers, and are within 5% of the ILP balancer.

While the ILP appears to outperform skewed-CH in the above experiments, it must be noted that the ILP's decisions were calculated offline based on collected workload traces. This is because the ILP takes about 10-100 msecs to converge to the solution for our testbed parameters, making it infeasible in practice (as the ILP is run for each arriving procedure). Thus, the ILP should be considered as an optimal but unrealistic comparison point.

Finally, Figure 11d evaluates the performance under different combinations of inter- and intra-load balancing using 3 hosts and the skewed load to illustrate the importance of each component. We see that our load balancing schemes (inter+intra) provide the least SLO violations and deviation in CPU utilization. When we replace our inter-host balancer with RR, the violations increase from 4.25% to 6.92%, but when we replace our intra-host balancer with RR, the violations increase much more substantially to 12.35%. However, the deviation of CPU usage shows the opposite trend. This is because the inter-host balancer is primarily responsible for resource efficiency, whereas the intra-host balancer is primarily responsible for SLO compliance. Replacing both components with RR provides even worse performance.

8 RELATED WORK

Functional Decomposition: There is a significant body of work that decouples control and data planes to remove the interference between control procedures and data [4, 28, 29]. Our work focus on control plane alone and removes interference among control procedures. Recent work, Flurries [46] & Gremlin[15] frameworks have demonstrated the resiliency of such dis-integrated services implemented as micro services. Our stateless MME is designed along the same principles.
Scalability: Several recent studies suggest decoupling of the static bindings of MME with other entities thus allowing the MME to scale [3, 34, 41, 42]. But, the system is stateful and hence the states held by MME specific to each UE/IoT device prevents MME from rapidly scaling and can create hit spots. Our approach to statelessness is motivated by [20] that makes a case for breaking the coupling between the state and processing.
Load balancing: The recent work [3, 36] has proposed consistent hashing based load balancing. However, these methods result in inefficient resource usage and unreliable SLOs. In our work, we have developed a customized approach, skewed consistent load balancing, to alleviate these issues.

9 CONCLUSION

With the advent of IoT devices the need for handling diverse loads with varying traffic characteristics and SLOs in the cellular core networks is critical. This is coupled with the need for cost effectiveness where we no longer can design for peak loads, as billions of IoTs need to be carried in the network at a very modest cost. Given this backdrop, we specifically have focused on handling the control traffic effectively at a critical core network entity - the MME. The proposed design, `MMLite`, is a departure from traditional inflexible approaches that use static binding between the state and the processing. `MMLite` uses a stateless microservices-based approach in order to decouple this binding and also to enable functional customization that is more responsive to SLO requirements and resource availability. We also develop resource/SLO-aware load balancing approach based on skewed consistent hashing. Our evaluations using DPDK and OpenNetVM-based prototypes demonstrate much superior performance of `MMLite` with respect to traditional approaches with respect to fault tolerance, scaling, resource utilization and SLO satisfaction.

REFERENCES

[1] Andres-Maldonado, P., Ameigeiras, P., Prados-Garzon, J., Ramos-Munoz, J. J., and Lopez-Soler, J. M. Optimized LTE Data Transmission Procedures for IoT: Device Side Energy Consumption Analysis. *arXiv preprint arXiv:1704.04929* (2017).

[2] Archibald, R., Gupta, D., Jana, R., Gopalakrishnan, V., Rajan, A. S., Ramia, K. B., Dahle, D., Cooper, J., Kennedy, G., Rao, N., Sonnads, S., and Donald, M. M. An IoT control plane model and its impact analysis on a virtualized MME for connected cars. In *2016 IEEE International*

Symposium on Local and Metropolitan Area Networks (LANMAN) (June 2016), pp. 1–6.

[3] Banerjee, A., Mahindra, R., Sundaresan, K., Kasera, S., Van der Merwe, K., and Rangarajan, S. Scaling the LTE Control-plane for Future Mobile Access. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies* (New York, NY, USA, 2015), CoNEXT '15, ACM, pp. 19:1–19:13.

[4] Binh Nguyen et al., ECHO: A reliable distributed cellular core network for public clouds. 2017. https://www.flux.utah.edu/paper/echo-tr.

[5] Call Failures in MME. Nov 2017. https://en.wikipedia.org/wiki/QoS_Class_Identifier.

[6] Chen, G., He, W., Liu, J., Nath, S., Rigas, L., Xiao, L., and Zhao, F. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2008), NSDI'08, USENIX Association, pp. 337–350.

[7] Cisco: ASR5x00 MME Overload Protection Features. 2015. https://goo.gl/LV97b5.

[8] Cisco: MME Overview (Overload Protection). 2017. https://goo.gl/dyF3x9.

[9] End-to-end LTE/EPC network with OpenAirInterface (OAI) simulated eNB/UE and OAI's EPC . Nov 2017. https://goo.gl/kXSvBi.

[10] Ericsson Mobility Report. June 2016. http://www.ericsson.com/res/docs/2016/ericsson-mobility-report-2016.pdf.

[11] Ferdouse, L., Anpalagan, A., and Misra, S. *Congestion and overload control techniques in massive M2M systems: A survey*, 2017. e2936 ett.2936.

[12] Gandhi, A., Harchol-Balter, M., Raghunathan, R., and Kozuch, M. A. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Trans. Comput. Syst. 30*, 4 (Nov. 2012), 14:1–14:26.

[13] Gartner Reveals Top Predictions for IT Organizations and Users in 2017 and Beyond. 2017. http://www.gartner.com/newsroom/id/3482117.

[14] Handling of signaling storms in mobile networks. October 2017. https://goo.gl/fzkSGH.

[15] Heorhiadi, V., Rajagopalan, S., Jamjoom, H., Reiter, M. K., and Sekar, V. Gremlin: Systematic resilience testing of microservices. In *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)* (June 2016), pp. 57–66.

[16] High Availability is more than five nines. Nov 2017. https://goo.gl/o4dV3E.

[17] Intel Data Plane Development Kit. 2017. http://dpdk.org/.

[18] Jenkins hash function. Oct 2017. https://en.wikipedia.org/wiki/Jenkins_hash_function.

[19] Jover, R. P. *Security and impact of the IoT on LTE mobile networks*, 2015.

[20] Kablan, M., Alsudais, A., Keller, E., and Le, F. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)* (Boston, MA, 2017), USENIX Association, pp. 97–112.

[21] Laner, M., Svoboda, P., Nikaein, N., and Rupp, M. Traffic Models for Machine Type Communications. In *ISWCS 2013; The Tenth International Symposium on Wireless Communication Systems* (Aug 2013), pp. 1–5.

[22] Li, Y., Yuan, Z., and Peng, C. A Control-Plane Perspective on Reducing Data Access Latency in LTE Networks. In *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking* (New York, NY, USA, 2017), MobiCom '17, ACM, pp. 56–69.

[23] Load Balance MME in Pool. 2017. https://goo.gl/61CqWz.

[24] LTE EPS mobility management timers - EMM timers. October 2017. https://goo.gl/TgH3QN.

[25] LTE SUBSCRIBER SERVICE RESTORATION. Nov 2017. https://goo.gl/nfmLv6.

[26] Managing LTE Core Network Signaling Traffic. 2017. https://insight.nokia.com/managing-lte-core-network-signaling-traffic.

[27] Matteo Pozza et al., Solving Signaling Storms in LTE Networks: a Software-Defined Cellular Architecture. 2017. http://tesi.cab.unipd.it/53297/1/tesi_Pozza.pdf.

[28] Mohammadkhan, A., Ramakrishnan, K., Rajan, A. S., and Maciocco, C. CleanG: A Clean-Slate EPC Architecture and ControlPlane Protocol for Next Generation Cellular Networks. In *Proceedings of the 2016 ACM Workshop on Cloud-Assisted Networking* (New York, NY, USA, 2016), CAN 16, ACM, pp. 31–36.

[29] Nagendra, V., Sharma, H., Chakraborty, A., and Das, S. R. LTE-Xtend: Scalable Support of M2M Devices in Cellular Packet Core. In *Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges* (New York, NY, USA, 2016), MobiCom Workshop, ATC '16, ACM, pp. 43–48.

[30] Nokia Siemens Networks: Signaling is growing 50% faster than data traffic. October 2017. https://goo.gl/oTbTmM.

[31] Oh, S., Ryu, B., and Shin, Y. Epc signaling load impact over s1 and x2 handover on lte-advanced system. In *2013 Third World Congress on Information and Communication Technologies (WICT 2013)* (Dec 2013), pp. 183–188.

[32] OpenNetVM. 2017. http://sdnfv.github.io/onvm/.

[33] Potsch, T., Marwat, S. N. K. K., Zaki, Y., and Gorg, C. Influence of future m2m communication on the lte system. In *6th Joint IFIP Wireless and Mobile Networking Conference (WMNC)* (April 2013), pp. 1–4.

[34] Premsankar, G., Ahokas, K., and Luukkainen, S. Design and Implementation of a Distributed Mobility Management Entity on OpenStack. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)* (Nov 2015), pp. 487–490.

[35] Protocol Signaling Procedures in LTE. Nov 2017. http://go.radisys.com/rs/radisys/images/paper-lte-protocol-signaling.pdf.

[36] Qazi, Z. A., Walls, M., Panda, A., Sekar, V., Ratnasamy, S., and Shenker, S. A High Performance Packet Core for Next Generation Cellular Networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (New York, NY, USA, 2017), SIGCOMM '17, ACM, pp. 348–361.

[37] RAMCloud. Oct 2017. http://dpdk.org/doc/guides-16.04/linux_gsg/nic_perf_intel_platform.html.

[38] RAMCloud. Oct 2017. https://ramcloud.stanford.edu/docs/doxygen/md_README.html.

[39] repository, T. O. Nov 2017. https://gitlab.eurecom.fr/oai/openairinterface5g.

[40] Shafiq, M. Z., Ji, L., Liu, A. X., Pang, J., and Wang, J. A First Look at Cellular Machine-to-machine Traffic: Large Scale Measurement and Characterization. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 2012), SIGMETRICS '12, ACM, pp. 65–76.

[41] Taleb, T., Ksentini, A., and Sericola, B. On Service Resilience in Cloud-Native 5G Mobile Systems. *IEEE Journal on Selected Areas in Communications 34*, 3 (March 2016), 483–496.

[42] Taleb, T., and Samdanis, K. Ensuring Service Resilience in the EPS: MME Failure Restoration Case. In *2011 IEEE Global Telecommunications Conference - GLOBECOM 2011* (Dec 2011), pp. 1–5.

[43] Traffic models for machine-to-machine (M2M) communications: types and applications. 2014. http://www.eurecom.fr/publication/4265.

[44] using the classic PhantomNet portal, O. T. Nov 2017. https://wiki.emulab.net/wiki/phantomnet/oepc-protected/openepc-tutorial.

[45] Widjaja, I., Bosch, P., and Roche, H. L. Comparison of mme signaling loads for long-term-evolution architectures. In *2009 IEEE 70th Vehicular*

Technology Conference Fall (Sept 2009), pp. 1–5.

[46] Zhang, W., Hwang, J., Rajagopalan, S., Ramakrishnan, K., and Wood, T. Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies* (New York, NY, USA, 2016), CoNEXT '16, ACM, pp. 3–17.

[47] Zhu, T., Kozuch, M. A., and Harchol-Balter, M. Workloadcompactor: reducing datacenter cost while providing tail latency slo guarantees. In *SoCC* (2017).