ECHO: A reliable distributed cellular core network for hyper-scale public clouds

Binh Nguyen*, Tian Zhang*, Bozidar Radunovic‡, Ryan Stutsman*
Thomas Karagiannis‡, Jakub Kocur†, Jacobus Van der Merwe*

*University of Utah, ‡Microsoft Research, †Core Network Dynamics

ABSTRACT

Economies of scale associated with hyper-scale public cloud platforms offer flexibility and cost-effectiveness, resulting in various services and businesses moving to the cloud. One area with little progress in this direction is cellular core networks. A cellular core network manages the state of cellular clients; it is essentially a large distributed state machine with very different virtualization challenges compared to typical cloud services. In this paper we present a novel cellular core network architecture, called ECHO[1], particularly suited to public cloud deployments, where the availability guarantees might be an order of magnitude worse compared to existing (redundant) hardware platforms. We present the design and implementation of our approach and evaluate its functionality on a public cloud platform. Analysis shows ECHO promises higher availability than existing telco solutions.

ACM Reference Format:
Binh Nguyen*, Tian Zhang*, Bozidar Radunovic‡, Ryan Stutsman*
Thomas Karagiannis‡, Jakub Kocur†, Jacobus Van der Merwe* . 2018. ECHO: A reliable distributed cellular core network for hyper-scale public clouds. In *The 24th Annual International Conference on Mobile Computing and Networking (MobiCom '18), October 29-November 2, 2018, New Delhi, India.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3241539.3241564

1 INTRODUCTION

Recent years have seen a tremendous uptake of cloud computing. More and more companies move their services to the public cloud to take advantage of the economies of scale, the resource elasticity and scalability that the cloud offers. In stark contrast, the telco industry today faces major challenges in equipment upgrading, scaling, and introducing new services [20]. Cellular core networks are largely still based on custom-built hardware mandated by the strict reliability requirements posed by running a mobile core network [57, 63].

To alleviate these challenges, telcos and cellular operators are attempting to virtualize their core networks through network function virtualization (NFV) [9]. Typically, this is in the form of a move to a private-cloud setting, where the telco provider has full control of the infrastructure and can optimize the whole stack for its particular services. Indeed, owning the whole cloud stack can allow for the addition of specialized functionality for fault tolerance and management [50, 51, 64]. However, this functionality typically imposes extra constraints on cloud design (e.g. imposes locality, low-level network access). A super-optimized cloud stack for a particular core service might not be able to scale to the size of a public cloud, and may be at odds with the requirements of a new service to be introduced. Telco providers will have to manage and maintain the new private cloud deployments and will not be able to take full advantage of the economies of scale a public hyper-scale cloud deployment can offer.

Instead, the question we address is whether it is feasible to implement a cellular core network on top of a *hyper-scale public* cloud, such as Amazon AWS or Microsoft Azure. To achieve this, one has to address three main challenges. First, reliability - a typical public cloud availability SLAs are "four 9s" (i.e., availability of 99.99%) or less, but only if a service is deployed *across several VMs* in different availability sets [6, 46]. In contrast, a cellular core network today often requires up to five 9s reliability [18, 49], and comprises different *monolithic* components. Such a reliability requires replication of components across VMs and across multiple data centres. Secondly, failover architectures in hyper-scale clouds are very different from private data centers because of their scales. A typical fault detection in a public cloud is of order of 10 seconds [7, 47], in order to limit false positives [26]. This is too slow for a cellular core whose timeouts are of order of 1 second, hence we cannot apply the same fail-over techniques from private telco clouds. Finally, a cellular core

[1] After "Echo, the Nymph of Steady Reply" from Greek mythology.

MobiCom '18, October 29-November 2, 2018, New Delhi, India

ACM ISBN 978-1-4503-5903-0/18/10…$15.00
https://doi.org/10.1145/3241539.3241564

requires consistency of mobile clients' session state across multiple network components *and* end-user devices. This makes it fundamentally different than virtualizing middleboxes [22, 23, 58, 60] and other web services.

In this paper, we redesign the EPC's control plane[2] for a hyper-scale public cloud deployment with three goals in mind. First, it should provide high availability while taking into account the unpredictability of the public cloud. Second, it should be backward compatible with the existing hardware and network deployments (phones and base stations). Third, it should allow minimum modifications to the existing EPC design and avoid any new dependencies of the cellular signaling protocols which can and will change in time.

To this end, we introduce ECHO, a distributed network architecture for the evolved packet core (EPC) on the public cloud. We propose all ECHO components to be distributed across multiple VMs as no single VM can provide sufficient reliability. Each component (e.g., MME, SPGW) in ECHO must be redundant. However, redundancy introduces consistency issues, which we found can also lead to customer-observed unavailability. To solve this, ECHO lightly augments EPC with extra information that can be used to detect stale states and stale requests, allowing ECHO to enforce inter-VM consistency. In sum, ECHO is available and provides consistent operation as long as a majority of VMs, in a single or *across* data centers, are reachable - regardless of whether failures are due to software, host or network failure.

Our contributions can be summarized as follows:

• We propose ECHO, a distributed EPC architecture for the public cloud that achieves availability superior to conventional hardware EPC by continuing safe operation even in the presence of software, host, network, or data center failures. ECHO uses *conventional* distributed systems techniques to solve the availability with a focus on correctness. Its key contribution is that it uses the techniques properly on an unmodified EPC protocol while eliminating correctness issues and edge cases that otherwise result from unreliable and redundant components.

• The core of ECHO is an end-to-end distributed state machine replication protocol for a software-based LTE/EPC network running on an unreliable infrastructure. ECHO ensures atomic and in order execution of side-effects across distributed components using a necessarily reliable agent, and atomic and in-order execution on cloud components. Cloud components in ECHO are always non-blocking to ensure performance and availability.

• We demonstrate the feasibility of the proposed architecture by implementing it in full. We implement and deploy the entry-point agent software on a COTS LTE small cell [28]. We implement the required EPC modifications into OpenEPC [17] and deploy ECHO on Azure.

• We perform an extensive evaluation of the system using real mobile phones as well as synthetic workloads. We show that ECHO is able to cope with host and network failures, including several data-center component failures, without end-client impact. From analytics, ECHO can promise higher availability compared to existing telco solutions. ECHO shows performance comparable to commercial cellular networks today. Compared to a local deployment, ECHO's added reliability introduces an overhead of less than 10% to latency and throughput of control procedures when replicated within one data center. We evaluate ECHO client on five base stations in a 3 months long live trial and show no observable performance overhead.

To the best of our knowledge, ECHO is the first attempt to run a cellular core on a public cloud and the first attempt to replicate the LTE/EPC state machines in an NFV environment. ECHO is a step toward relieving telcos from the burden of managing their own infrastructure. We also hope it will allow (small) operators to deploy cellular networks in communities where it wasn't previously economical to do so.

2 BACKGROUND

This section presents a brief overview of the mobile core control plane and makes an observation that, effectively, the control plane implements multiple distributed state machines, one per user.

2.1 Mobile Core Network Control Plane

Control Plane Components: The control plane of the cellular network does not participate in packet forwarding. It instead installs forwarding rules on the data plane. The main component of the control plane in LTE/EPC is the Mobility Management Entity (MME) which authenticates mobile clients (i.e., User Equipment – UE), sets up data plane connection, and pages UE's location. The data plane consists of a base station (eNodeB), Serving Gateway (SGW) and a Packet Gateway (PGW). Co-located with each data plane component is a control plane component (i.e., eNodeB-C, SGW-C, PGW-C) that coordinates with the MME and with each other to implement a data path for the UE on the data plane.

UE context: The network stores an UE context for each *attached UE* which consists of subscriber information (authentication key, UE's capability), the current state (connected or idle), and the data connection (Evolved Packet System bearer or EPS bearer) of the UE. The UE context is stored across the control plane components; the MME stores all of the UE context, while the eNodeB-C, S/PGW-C only store information of the data connection.

[2]Throughout the paper, by EPC we refer to the EPC's control plane.

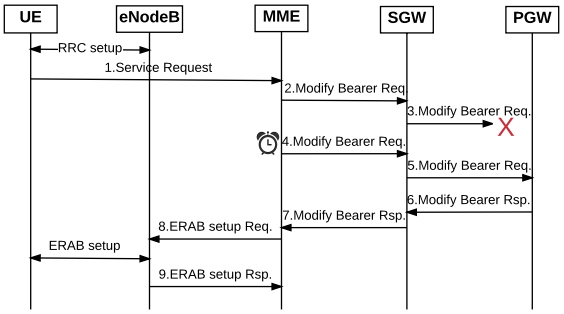


Figure 1: Service Request procedure in LTE/EPC. After setting up a Radio Control channel (i.e., RRC setup), the UE sends a *Service Request* to the MME. The MME modifies its UE context and sends a *Modify Bearer Request (MBR)* as a side effect request to the Serving Gateway (SGW). This side effect message sets up a tunnel endpoint for the UE on the SGW and triggers another *MBR* to Packet Data Gateway (PGW) to set up a tunnel endpoint on the PGW. If the MME doesn't receive a reply, a timer expires and the MME retries (msg. #3,4). When the PGW, SGW acknowledges the MME (msg. #6,7), the tunnel endpoints are past to the eNodeB (msg. #8.) that set up a E-UTRAN Radio Access Bearer (ERAB) at the eNodeB.

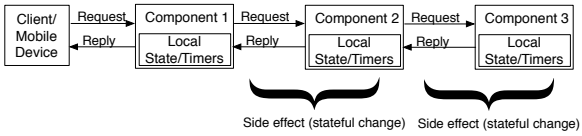


Figure 2: Distributed state in core mobile network. Components 1, 2 and 3 map to the MME, the SGW and the PGW in Figure 1.

2.2 Mobile core: a distributed state machine

The cellular control plane implements a distributed state machine for each UE, as illustrated on the Service Request example in Figure 1. The state machine is distributed across multiple components and a transition may involve communications and state changes across multiple other components. The control plane runs many such state machines in parallel, *one for each UE*. A generalized depiction of this distributed state machine is in Figure 2.

Specifically, the distributed state machine deals with the following events and messages:

Request from UE: Most of the changes in the state machine are triggered by a client or mobile device. For example, when an idle UE has data to send, it sends a Service Request (message #1 in Figure 1) to the MME.

Side effect request & reply: Upon receiving a request from a UE, the MME may alter the states at other components. In the Service Request example above, the MME must set up a bearer in the data plane. The MME sends a bearer setup request to the SGW (message #2) which triggers another request to the PGW (message #3). We call the messages that are generated by components in the cellular core, and are indirectly triggered by the main request that originated in the UE, *side effect requests*.

Timers: A state transition can also be triggered by a timeout. For example, if a SGW does not respond to the bearer setup request, the MME will trigger a retry when its timer expires (message #4). This retry generates another side effect request to the system. A timer can be set by any component, if so required by the protocol.

3 RELIABILITY IN CLOUD-BASED EPC

Through examples, we show the strict reliability requirements of the cellular core network. We then present the state of the art of reliability in the current cellular core network. We contrast today's cloud availability with hardware reliability through a 3-month long study.

3.1 Mobile network reliability requirements

We conducted experiments with a Nexus 5 mobile device, an LTE eNodeB (ip.access smallcell) and the OpenEPC core network to demonstrate the core network's reliability needs. The experiments also highlight the key requirements of the cellular network that could be summarized as following: The control plane components of the cellular network (1) must be as highly available as possible, and (2) must process requests from the UE *atomically* and *in-order*.

High availability: A service outage on an MME would also be interpreted as a congested mobile network, so UEs are required to back-off from the network. We demonstrate this with an example scenario in which, after 5 unsuccessful Attach attempts lasting 1 minute in total, the UE entered silent state for 12 minutes before it retries to attach again. Hence, a short MME outage can result in disproportionate experienced outages in UEs. To illustrate this behavior, we triggered the UE to attach to the LTE network and left a bug in the MME so that the UE failed to attach. Figure 3a shows the MME's log with timestamps illustrating this experiment.

Persistent state: If the MME loses a UE context, it cannot process UE requests, leading to a long outage on the UE[3]. Unfortunately, the LTE protocol doesn't deal with this state inconsistency proactively; UE doesn't reattach when detects this issue. We show experimentally that when the MME loses the UE context, the UE loses connectivity for 54 mins!

Figure 3b shows the MME's log with timestamps when the MME loses context for an attached UE. The UE attached to the network (*17:05:26*). The UE went to idle state due to inactivity (*17:06:14*) while the MME still keeping its state. After this the MME crashed and the UE context was lost. The UE went from idle to active and requested for services but did not get any service for 54 mins (from *17:10:01* to *17:54:03*). 54 mins after the Attach, the UE performed a

[3]While OpenEPC has an option to enable persistent UE state storage on MME, naively enabling this feature is not sufficient to solve the consistency issue described in the next example.

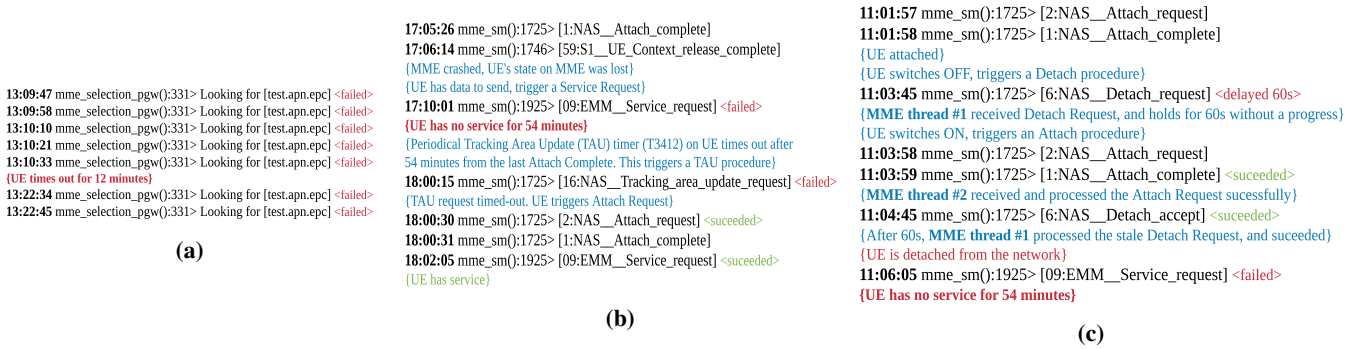


Figure 3: Examples of real-world outages caused by reliability issues: (a) 5 consecutive Attach failures caused UE to sleep for 12 mins; (b) UE did not have service for 54 minutes because MME crashed and UE context was lost; (c) Violation of FIFO order execution caused state inconsistency and 54 minutes outage.

periodic Tracking Area Update (TAU) procedure (*18:00:15*) as defined in its protocol. This TAU also failed because the MME does not have any context of the UE. The result of this unsuccessful TAU is that the UE timed out and moved to a "deregisterd" state which requires the UE to reattach [1] (*18:00:30*). The Attach Request let the UE exchanged its context with the MME. After having the UE context, the MME was able to serve the UE as normal (*18:02:05*), ending the extended UE service outage.

In-order message delivery and execution: If the mobile core network execute requests from the UE in an out-of-order manner, the state between the network and the UE will be inconsistent which leads to a long outage. We demonstrate this with an experiment in which a UE requests $\langle R_1, R_2 \rangle$ but the network executes the requests in a different order (i.e., $\langle R_2, R_1 \rangle$). This causes state inconsistency between the network and the UE which results in a long outage.

Figure 3c shows the MME's log with timestamps describing this experiment. After attaching to the network, the radio interface of the UE was turned off to trigger a detach (11:03:45). That detach was processed by a *MME1 thread* which is a slow MME thread. Later the UE was turned on to trigger another attach request which arrived at *MME2 thread* (11:03:58), updating the state of the UE Context with the *Attached* state. This was successfully verified by the MME2 and replied to (11:03:59). However, the slow MME1 thread later was executed and updated the UE Context with *Detached* state (11:04:45). The Detached Accept message was ignored by the UE. This results in an inconsistent state between the network (*Detached*) and the UE (*Attached*). This caused a UE outage of 54 mins as in our previous experiment.

3.2 Reliable EPC: state of the art

Telecom-grade reliable hardware is built with $N + M$ redundancy [18, 48, 49, 66]. Active-standby techniques [36] allow for state synchronization (e.g., UE context) between the active and standby instances with the active one switching over to the standby one in case of a failure. This technique is extended to an NFV setting where a resource scheduler can quickly detect a fault and migrate service from a faulty component [50, 51, 64].

Further redundancy is introduced at the protocol layer. The standard EPC architecture supports a pool of MMEs [3] behind a load balancer. If one MME instance fails, the eNodeB will notice that its Stream Control Transmission Protocol (SCTP) connection to MME is broken and connect to another MME instance in the pool. State (UE context) is either stored in a common Session Restoration Server (SRS) [38, 39] or synchronously replicated among MME instances [18]. This mechanism, however, doesn't deal with the out-of-order execution problem. For example, if the SCTP connection of a MME instance is broken (e.g., because of a network card crash) while the instance is still active, the instance could generate stale requests that might cause inconsistency.

There are several aspects of the existing designs which do not map well to the public cloud infrastructure. In order to allow for almost instantaneous fault isolation and repair, hardware appliances and VNFs offer fine-grained availability information and scheduling control (active standby or service migration), low-level network access and assume locality among instances of the same VNF. However, these techniques do not apply to hyper-scale public clouds, where instances often do not share the same rack, faults detection are much slower because of false positives [7, 26, 47] and network virtualization prevents standard approaches for fault migration [33].

Furthermore, due to higher inherent reliability of conventional nodes, the types of faults that can occur are different. Public clouds run all software on VMs that can delay executions (e.g., due to an upgrade), causing stale requests and inconsistent side-requests (as in example 3c). Overall, a public cloud EPC deployment has to deal with failures proactively and in software. Our failure measurements in the next section also suggest this.

	Cloud				World-wide			
Outage type	VM	DC Cluster	DC	Across DCs	VM	DC Cluster	DC	Across DCs
All	four 9s	five 9s	five 9s	five 9s	three 9s	four 9s	four 9s	five 9s
$>$ 10 sec	four 9s	five 9s	five 9s	five 9s	three 9s	four 9s	four 9s	five 9s
$>$ 1 min	four 9s	five 9s	five 9s	five 9s	three 9s	four 9s	four 9s	five 9s

Table 1: Inter-DC availability in a major cloud provider

3.3 Availability of public clouds

Cloud providers, such as AWS and Azure, do not advertise availabilities of individual VMs but only of "availability sets" of carefully selected VMs that belong to different fault and upgrade domains. Even so, the advertised availability is "four 9s" – an order of magnitude larger total outage compared to the five 9s availability of telco appliances. Besides the overall availability in the number of 9s, the mobile network reliability requirements outlined in Section 3.1 highlight that the duration of an outage can be critical. For example, the system may be able to recover from many short 1-second outages using transport or other mechanisms, but a few outages lasting minutes can be catastrophic (example 3a). It is thus crucial to understand the availability properties (total outage instances and their duration) of public clouds in practice, beyond advertised SLAs.

To this end, we perform a 3-month long measurement study in a major public cloud provider. We expect our findings to be indicative of other providers as well. We monitor the VM uptime and reachability at multiple levels: data center (DC) cluster, single DC, and across DCs. A DC cluster (or availability set) consists of three VMs in different availability zones behind a load-balancer within the same regional data center. There are two clusters per DC. In total, we use 3 DCs, two in Europe and one in the US. We measure uptime and reachability both within Azure and from the public Internet; we generate TCP pings every 1 second from each VM to all other VMs and use Azure's Application Insights monitoring service [42] to monitor VM reachability from the public Internet. The service tests reachability of VMs every 10 minutes from 10 locations across 4 continents. The cloud is available if *at least* one VM in a cluster is available.

Results: Our results are summarized in Table 1. Each row in the table shows the observed availability constrained on an outage duration (e.g., in row $>$ 1 min we only account for outages that are longer than 1 min – outages that would result in 12-min outages for UEs as in example 3a). We observed intra-cloud outages of more than 1 second, 2,400 times during our study. The Cumulative Distribution Function (CDF) of the durations of such outages is depicted in Figure 4. In all, there are 7 outages that last more than 1 minute and they can all be attributed to VM failures. We observe that the advertised SLAs of four 9s are generally met by the cloud.

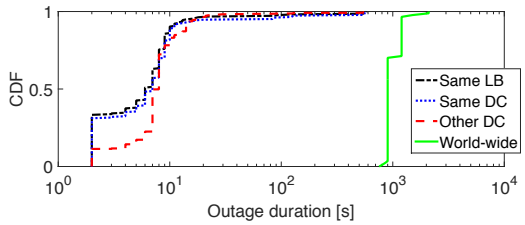


Figure 4: Outage duration distribution across all pairs of VMs

The picture is significantly different from the Internet ("World-wide" in Table 1). Availability is roughly an order of magnitude less compared to intra-DC measurements, with more than 20% of outages last 20 minutes or more. This suggests that most "outages" are due to public Internet connectivity problems reaching the cloud.

Implications: In summary, we observe that our five 9s availability can be achieved only if the service is replicated across multiple VMs *across availability zones in a single DC (columns 3,4,5 in Table 1)*; additionally, coping with public Internet reachability problems requires service presence *across multiple regional data centers (column 9 in Table 1)* unless a dedicated connectivity service to the cloud [5, 41] is deployed which can incur extra cost. In short, in order to deal with both VM failures and public-facing connectivity issues, our design must be able to replicate the service both within a DC *and* across multiple DCs.

We note that the measured availability exceeded the cloud's advertised SLA in some cases (columns 2,3,4,5). However, this could be explained by the rather short duration of our measurement (3 months). We also note that [25] also studies public cloud reliability over a seven year period and points out that a median reliability across 32 public clouds is below 99.9%, and that the median duration of an outage varies between 1.5 and 24 hours. This reinforces the need for software replication and proactively dealing with the unpredictability of public clouds.

4 ECHO DESIGN

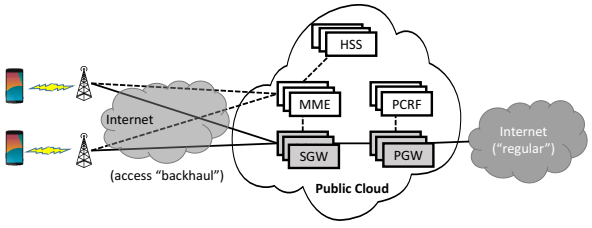


Figure 5: ECHO's high level network deployment.

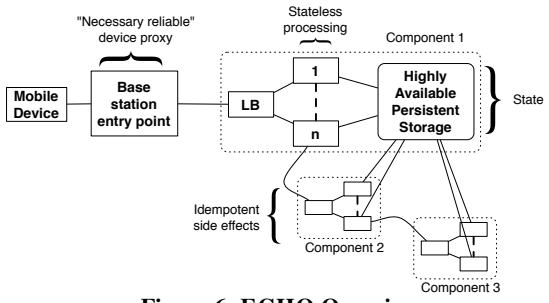


Figure 6: ECHO Overview

Figure 5 presents a high-level depiction of ECHO's operation. ECHO moves the main components of EPC into the cloud. The base station connects to a MME behind a public cloud load-balancer via an Internet-based access "backhaul". The PGW in EPC provides "regular" Internet connectivity also via the cloud load-balancer. We now discuss in more details the problem space, ECHO's architecture and operation.

4.1 Problem space

As in §2.2, EPC is a distributed state machine comprised of several components, each storing state for each user. ECHO must achieve high availability ($\geq$five 9s) compared to conventional telco appliances despite VM and network failures. Besides availability[4], ECHO must produce the same results as the original core network that assumes reliable components (i.e., correctness property). In particular, ECHO must execute requests *atomically* and *in the order that the (per-device) requests arrive at the base station*. ECHO must also scale to support a large number of users. Finally, a particular challenge is that one of the component that stores the state is a user's mobile device, which cannot easily be modified.

4.2 ECHO Overview

Figure 6 depicts an overview of ECHO. Each control plane component (Components 1, 2 and 3 that correspond to MME, SGW, and PGW) is replicated (instances 1 to n) behind a data center load balancer (LB) [40, 52]. Each component instance is refactored into a stateless processing frontend, paired with a high availability persistent storage backend that maintains state for all replicas (and all components). This allows quick replacement of a malfunctioning component and scaling based on demand.

At each base station, i.e., eNodeB, there is a "necessarily reliable" entry point. This entry point is an *assumed reliable* component (§4.3) that (1) queues requests of the mobile device and tags them with a unique *sequence ID* for each request and (2) keeps resending a request until receiving an acknowledgment from the Component 1 (i.e., MME) before moving to the next request.

In ECHO, the stateless instances of a component implement the same unmodified 3GPP LTE/EPC state machine. Stateless instances use an optimistic, non-blocking approach to enforce whole-operation atomicity. They can process the same request in parallel without mutually blocking each other. If an instance processing an operation fails, the *necessarily reliable agent* timeouts and retries. Eventually, when one instance makes progress (i.e., externalizes state in the state storage), the component moves to the next state; other instances processing the same request in parallel will abort and discard any buffered changes if another instance has already completed processing the operation.

Each component in ECHO processes requests from the mobile device *in-order* using the sequence ID embedded in the requests. During processing a request, if the component generates a side effect to another component, then the sequence ID is also embedded in the side effect message. The component receives the side effect uses the sequence ID to guarantee the in-order execution. This happens at every component to enforce in-order execution across all components. More details about atomic and in-order execution are in § 4.4.

ECHO bears resemblance to other transactional systems that leverage optimistic concurrency control [4], especially those that have to deal with nested transactions on distributed state [37]. However, those systems have to deal with general applications, so they are complex (and, consequently, not commonly deployed in distributed environments). Since, ECHO is tailored to EPC it is much simpler: it can rely on the fact that UE state is partitioned, that UEs do not issue concurrent requests, that UE operations are totally ordered by a single entry point, and that UEs can only observe effects through transitions of the state machine (and never through external effects). Together, these assumptions simplify ECHO and remove the need for stronger transactional isolation properties. This also makes ECHO more practical to deploy since it can be built with off-the-shelf key-value stores that are easy to deploy and scale.

4.3 Necessarily reliable entry point

The ECHO entry point approach relies on the fact that the base station (eNodeB) is a *necessarily reliable* component. Because the network connectivity of a mobile device relies on wireless access to the base station, connectivity is lost if the base station crashes; there is no point designing the system to deal with base station failures. Therefore, since the entry point is as reliable as the base station, it is seen as a "reliable" component of the system.

The entry point agent is a thin software layer deployed on a base station. It is similar to a sequencer in other distributed

[4]ECHO uses consistent, linearizable storage. By the CAP theorem, it cannot remain available under *all* network failures. However, using consensus it is only offline when no majority of data centers hosting ECHO can communicate with one another. This is *extremely* rare (columns 5,9 in table 1); such an extreme case would already likely result in a partitioned-away base station.

databases [65], however, because ECHO needs only provide atomicity and eventual completion of distributed operations, it avoids the full complexity of a more general distributed concurrency control. It provides the following functionality.

Sequential request IDs: The entry point assigns a sequential ID to each request from a given UE; different UEs have independent ID sequences. The request is queued locally and forwarded to the next component (the MME). The entry point serializes the requests using a FIFO queue: the oldest unacknowledged request is resent until it is acknowledged and removed from the queue. The sequential IDs are used to ensure that requests are processed at components in the same order as the UE issued them.

Eventual completeness: After queuing a request, the entry point persistently retries until the request is acknowledged before moving to the next one. This ensures a component failure in the cloud won't be visible to the mobile device; if an instance of a component crashes in the middle of an operation, the entry point transparently issues a retry and the retry will reach another instance of that component to recover from the crash. As the entry point is the "reliable" component, its retries ensure a request is eventually processed and is processed by *all* core components regardless of failures.

Reliable timers: Components in EPC must set a timer whenever they receive a request. However, if components crash timers are lost. In ECHO, since the entry point is considered reliable, components' timers are maintained and triggered by the entry point instead of by the components; after receiving a request, the component creates the timer event by sending a *set timer request* to the entry point. The set timer request includes a unique ID of the mobile device that the timer applies to, a unique timer ID, and a timeout value; the request ID of the event is returned. To cancel a timer event, the component sends a *cancel timer request* with the user ID and the previously returned request ID of the timer event.

State coordination with clients: Since request IDs are added (and removed) at the entry point, unlike components, client devices cannot rely on them to reliably receive correctly ordered responses. A failed state update at a component may produce a message that is sent to a client, and a retry may produce another copy of the same message. This must be handled by the entry point. Each client-bound reply is labeled with a request ID and the sequence number of the message within the request. The entry point ignores replies that have already been forwarded to the client. Retries always produce the same responses, but it is important that one and only one gets forwarded. Necessarily reliability means the client and the entry point can be expected to maintain a single, ordered, reliable connection (e.g., TCP connection), which safely deals with message loss on the last hop as long as the entry point correctly orders replies.

Handovers: After a handover, the target entry point (target eNodeB) must handle the client's operations, so the request ID must be transferred from the source eNodeB to the target eNodeB. This could be done by augmenting the 3GPP Handover procedure [3]. The old entry point embeds the client's current request ID in the Handover Required message which is sent to the MME. The MME then forwards the state to the target entry point in the Handover Request message. When the client successfully associates with the target eNodeB (i.e., after it receives the Handover Confirm from the client), the target entry point uses the transferred request ID for the Handover Notify message and following messages.

4.4 Non-blocking cloud components

Given the requests with monotonic request IDs, ECHO needs to guarantee atomicity and in-order execution properties on each component *and* across components. Algorithm 1 shows how a ECHO component processes a request. Note that the algorithm describes two types of components, with and without side effects (§2.2), in a single algorithm. The algorithm is designed to dovetail with required processing in conventional EPC components; the red lines (14, 17, 18, 19) already exist in EPC components. The algorithm is *non-blocking*; multiple stateless instances of a component execute the algorithm in parallel without causing a stall on other instances.

Component's atomicity: Replication of components in ECHO and retries from the entry point mean that a single request could be processed by multiple instances of the same component. To prevent inconsistency caused by interleaved processing of the same request across instances, ECHO uses atomic conditional writes provided by the persistent storage (we discuss our persistent storage implementation in Section 5). When committing changes to the reliable storage (line 22 in the algorithm), each component instance ensures that the stored UE context (session) remained unmodified while it was processing the request by checking the *version* number of the session. If the conditional write fails, then another component instance has already processed the request, so this instance discards the local session state and backs off. This assures that, even though multiple component instances can process the same request, only one instance is able to commit the changes at step 22, guaranteeing atomicity.

Component's in-order execution: A component in ECHO needs to execute requests in the order that they arrive at the entry point. However, concurrent retries of a request issued by the entry point can cause processing of an obsolete message at a component instance. Without care, this causes the state in the session store to regress, leading to inconsistency, as illustrated in Figure 3c.

A component in ECHO uses the monotonic request ID to filter out obsolete requests. As in line 6 in the algorithm,

Algorithm 1 Non-blocking cloud component

Input event: Receive a request from eNodeB's entry point (agent) *R*, with UE's ID (*R.UE*) and request ID (*R.ID*).
Output event: Send reply and timeout message to eNodeB's agent.

1: Fetch session from storage: *(session, version) = read(R.UE)*, where *version* is version number of session.
2: **if** *session* not found in storage **then**
3: Create a session locally. Set *session.ID = R.ID*
4: Go to step 14.
5: **end if**
6: **if** *R.ID < session.ID-1* **then**
7: {Received an obsolete request}
8: Return
9: **end if**
10: **if** *session.reply* and *session.timer* exist **then**
11: (Re)send *session.reply* and *session.timer*
12: Return.
13: **end if**
14: Update *session*.
15: Increment request ID: *session.ID += 1*
16: Set request ID in side effect msg: *session.side_effect.ID=R.ID*.
17: Send side effect message: *session.side_effect*.
18: Receive side effect reply.
19: Update *session*.
20: Prepare reply message: *session.reply*, set request ID in reply message *session.reply.ID = R.ID*
21: Prepare timeout message: *session.timer*, set request ID in timeout message *session.timer.ID = R.ID*
22: Write session to storage: *write(session, version)*
23: If write OK: Send reply and timeout messages: *session.reply*, *session.timer*

before processing a request, the component instance checks if the request ID is less than the last executed request ID (which is in the persistent storage). If it is, then the request is obsolete and is discarded. When externalizes the persistent storage, the component increments the session's request ID (line 15) and acknowledges the entry point (line 20).

E.g., in example 3c, the stale Detach Request at *11:04:45* would have been discarded as its request ID would have been lower than the request ID of the Attach Request that is last processed at *11:03:58*.

Atomic and in-order execution across components: Given each single component operates atomically and in order as described, ECHO needs to ensure atomicity and in order execution *across* its distributed components.

A *side effect* is triggered when one component processes a request that generates a message to another component. Consistency must be maintained across components despite side effects, but retries from the entry point can create multiple *duplicated* side effect requests, and slow instances can generate *stale* side effect requests. Without care, duplicated and stale side effect requests could cause inconsistency.

Service Requests (Figure 1) illustrate the inconsistency that can arise from duplicated side effect requests. Suppose an MME instance *A* receives a Service Request. In step 17 of algorithm 1, it sends a *Modify Bearer Request* (request #1) to the SGW component. An SGW instance receives the request #1, creates and installs a tunnel endpoint *TEID1*, stores it in persistent storage and replies to the MME with the information. Meanwhile, suppose that the entry point times out and retransmits the Service Request. Another MME instance *B* receives the retry and sends a *duplicated Modify Bearer Request* (request #2) in step 17. Later a SGW instance receives the request #2, and it *overwrites* and replaces *TEID1* with a new tunnel endpoint *TEID2* and replies. The MME component ignores the second reply because it already moved to a new state when the first reply arrived. In the end, the MME component (and the UE) contains *TEID1* while the SGW records *TEID2*; this inconsistency breaks the data plane.

To keep multiple duplicated side effect requests from mutating component state, retries of a side effect must induce the *same* effect on the target component (i.e., side effects must be idempotent). Algorithm 1 enforces this. When a message is processed, the response is recorded in the session store with its corresponding request ID, so lost responses can be reproduced without repeating execution. If an instance receives a request and the committed session in the persistent storage contains a reply, then another component instance has already executed the transaction, and it only needs to reply (lines 10, 11, 12). Since responses are recorded in the persistent storage, they can be obtained by other instances, in case the current instance crashes before replying.

To solve the inconsistency problem caused by stale side effect requests, a component also passes the request ID of received requests to the side effect requests it generates (line 16). The target component then ensures the side effect requests are executed in the order specified by the request ID. This happens at *every* ECHO component, so no stale side effect requests are processed.

Base station failures: If a base station fails, the mobile device will connect to another nearby base station. The entry point on the new base station needs to synchronize its sequence ID with the sequence ID in ECHO's storage. However, given the current sequence ID in the state storage is n, it is possible that the old base station already begun propagating a request $n + 1$ to the instances. To suppress the effects of this stale request and to ensure each request has a unique id, ECHO the new base station's agent uses 2-phase protocol similar to Paxos [34]. In the prepare phase, the entry point first asks if the instances haven't accepted request $n + 1$ and that they will only accept a no-op $n + 1$ request. When all instances are prepared, the entry point sends a no-op $n + 1$ to commit the synchronization. It then sends other normal requests with the sequence ID starting from $n + 2$. If any instance has already seen the older request $n + 1$ it is returned

to the entry point, which aborts the commitment of the no-op and reissues the discovered operation as $n + 1$.

4.5 Correctness

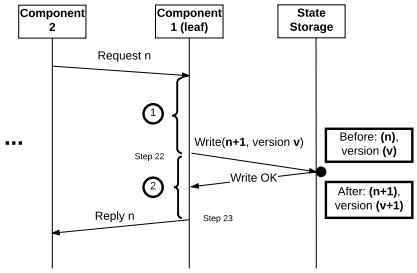


Figure 7: ECHO's leaf component is linearizable

Here we give a sketch of why ECHO is safe even though components are redundant and non-blocking under failure. Showing that ECHO *appears* to process operations atomically, in client FIFO order, one-at-a-time demonstrates safety.

The proof first needs to show that a leaf component (a component that does not trigger side effects to other components) operates linearizably (i.e., serializable and in FIFO order). It then uses induction, where the base case is the leaf component, to show that every component in ECHO operates linearizably and so does ECHO as a whole. Finally, because the requests arrive at ECHO in the client FIFO order, ECHO operates in client FIFO order.

ASSUMPTION 1 (IDEMPOTENT OPERATION). *Each leaf component instance processes requests from other components idempotently; that is, a retry causes the same effect on the component.*

With the assumption of idempotent operation of a leaf component, we prove that a leaf component operates linearizably.

LEMMA 4.1 (A LEAF COMPONENT IS LINEARIZABLE). *Each transition on a leaf component instance is linearizable; that is, it processes operations atomically in some total order consistent with the request ID.*

Proof. As shown in Figure 7, given that the update to the shared State Storage is atomic, individually, each processed request with ID $n + 1$ results in one of four outcomes: 1) *aborted*-the component is not in state n or $n + 1$, so the request is invalid and ignored; 2) *successful*-the operation completes successfully, the component instance updates the State Storage, moves the component from state n to $n + 1$ at step 22 and replies; 3) *crashed before update*-the operation fails in ① before updating the state leaving the component in state n; or 4) *crashed after update*-the operation fails in ② after updating the state leaving the component in state $n + 1$ without a reply.

In case 3, because of the eventual completeness, there must be another instance that progresses to either case 4 (in-completed) or case 2 (completed). In case 4, another component instance when receives a retry, simply replies with the recorded reply which eventually results in case 2. Therefore, a component only executes requests in the specified order, either completely successful or failed. □

Given that a leaf component operates linearizably, we can show that ECHO system is linearizable.

LEMMA 4.2 (ECHO'S LINEARIZABILITY). ECHO *is linearizable; that is, it appears to process operations atomically in some total order consistent with the requests order of the client.*

Proof. ECHO's linearizability could be proved using induction with the base case is the *leaf component*.

Base case: As in lemma 4.1, a leaf component operates linearizably.

Induction hypothesis: Now assuming component M operates linearizably, we need to prove component $M+1$ operates linearizably.

If there are multiple $M + 1$ instances trigger multiple side effects on component M, the effect is linearizable as the induction hypothesis. Therefore, component $M + 1$ operates similarly to an ECHO's leaf component, which is proved to be linearizable. □

Lemma 4.2 can be immediately strengthened, since the total order above is precisely mobile device's FIFO order.

LEMMA 4.3 (FIFO PROCESSING). *ECHO appears to processes operations atomically, in client FIFO order, one-at-a-time.*

Finally, since the ordinary, unreplicated protocol precisely processes messages atomically, in client FIFO order, one-at-a-time, this gives the essential safety property:

PROPERTY 1 (ECHO SAFETY PROPERTY). *The set of states observed by ECHO clients is equivalent to the unreplicated protocol.*

5 IMPLEMENTATION

Section 4 outlines general design principles ECHO uses to provide safety and reliability. Here we discuss specifically how this design applies to a cellular control plane and a public cloud. The summary of changes to the standard EPC architecture is illustrated in Figure 8.

ECHO agents: ECHO's agents are lightweight software proxies that provide entry-point functionality on eNodeB and an interface between eNodeB and MME. There are eNodeB's agent and MME's agent, as illustrated in Figure 8. The eNodeB's agent is implemented as a separate user-mode daemon written in standard C (≈ 5,000 LOC), deployed on top of

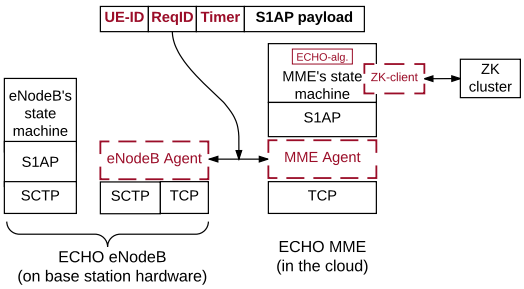


Figure 8: Modifications of ECHO in LTE/EPC

embedded Linux running on a commodity small cell [28]. This allows us to easily port it to any COTS eNodeB without affecting the time-critical LTE radio code. The MME's agent is integrated in the source code of the S1AP processing module of OpenEPC [17].

One of the agent's functions is to proxy S1AP control messages. 3GPP eNodeB and MME use SCTP protocol for S1AP messages. However, Azure and other public clouds do not support SCTP protocol, so we implement a proxy agent that replaces SCTP by TCP. The ECHO agent on the eNodeB opens an SCTP connection to the rest of eNodeB software stack on one side (which is unmodified and unaware of the agent's existence) and a TCP connection to an ECHO MME agent on the other side. The eNodeB agent relays messages between the two connections. The agent reestablishes the TCP connection on a failure, in order to attach to a new MME instance (in the same DC or a different DC).

Furthermore, the agent implements the entry point design, described in Section 4.3. The agent adds an extra network layer (ECHO or agent layer) into the LTE/EPC control plane stack, as in Figure 8. The ECHO layer header consists of the *Request ID*; a *UE-ID*, a unique identifier of the UE, composed of tunnel identifiers readily available from S1AP messages; and a *Timer value*, used to set up timers and to inform components about timer expiry.

Stateless EPC components: We have augmented the most important EPC components (MME, SGW and PGW) in OpenEPC [17] with ECHO functionality. In the example of MME, our implementation preserved the original implementation that extracts information from a received S1AP message, generates side effects and updates the client's state (e.g., steps 14, 17, 19 in the algorithm). We extended handlers to accept request IDs from the ECHO layer and to add duplicate/stale request checks that adapt processing accordingly (step 6). When the original MME code finishes processing a request, ECHO sends an acknowledgment to the eNodeB agent together with an S1AP reply. We made SGW/PGW operations idempotent by making the SGW reply with a stored message (i.e., with the same bearer information) for duplicate requests from the MME (so, the duplicates don't forward effects to the PGW).

In all, ECHO's extensions to OpenEPC required changing 1,410 lines in 12 files.

We added two additional blocks to the conventional EPC: an *agent* (described previously) and a *ZooKeeper client* (ZK-client). The ZK-client provides a *read/write/delete* interface to a ZooKeeper [27] (ZK) cluster that acts as a reliable, persistent storage. ZooKeeper is a reasonable choice of storage because of its consistency guarantees, small amount of stored information (a few KBs per UE context) and relatively low request rate. The UE context (which is extended to include UE replies) is stored as a binary string in a znode in ZK. ECHO uses the *version number* of a znode in ZK to realize an atomic state update at step 22 of the algorithm; ZK only allows updating the znode if the version number hasn't changed since the beginning of the request.

Unique key for UE context: Each UE in ECHO should have a unique key to identify the UE context in the storage. ECHO uses the International Mobile Subscriber Identity (IMSI) of the UE – a number embedded in the SIM card of each device which *uniquely* identifies a device in a network and *never changes* – to be the name of the znode that stores the UE context. The IMSI is included in the very first message (Attach Request) from the UE. The ECHO's MME extracts the IMSI in the message and notifies the UE to use the IMSI as the UE-ID for following requests.

Cloud deployment: Multiple instances of the same component are deployed in a private network in Azure behind a load balancer. The load balancer performs consistent hashing on the connection's 5-tuple, so a connection sticks with the same instance unless there is a failure or a new instance is added. When ECHO is deployed across multiple data centers, requests that time out a few times are retransmitted to another data center by the ECHO agent on the eNodeB.

6 EVALUATION

We evaluate ECHO in the Azure public cloud across several dimensions. We examine the correctness of our implementation, the latency introduced across various components of the architecture, the observed throughput and simulate potential failure scenarios. Our main findings are:

• We demonstrate that our cloud-based implementation correctly serviced 6,720 requests over one week without any failures in the ECHO system. A 3-month trial with five smallcell eNodeBs with maximum 14 users per eNodeB showed no performance degradation on ECHO eNodeB agent. A throughput test with 1000 emulated UEs showed good throughput compared to conventional EPC.

• ECHO introduced reasonable overheads as a trade-off for a public-cloud reliable deployment. When replication within a single data center was used, the response latency was increased by less than 10% and there was no visible drop in

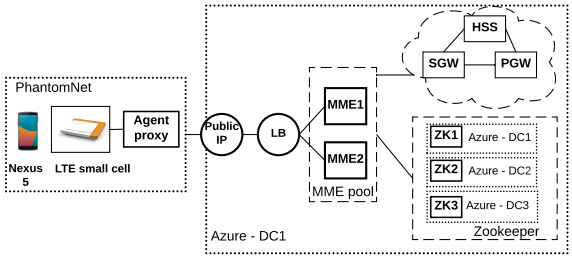


Figure 9: ECHO evaluation topology set up

throughput. Even in more extreme deployments, we showed that total latency was well below standard 3GPP timeouts and would not be noticeable by users. The result showed user-perceived latency was similar in ECHO and T-mobile.

• By emulating failures that are more extreme than typical data center failures, we showed that ECHO gracefully handled all such cases without user experience impact.

Evaluation setup. Our base deployment is given in Figure 9. It consisted of radio equipments (a UE - Nexus 5 device, an LTE eNodeB - ip.access E40 LTE smallcell [28]) in PhantomNet [10]. For a larger scale testing, we use 1000 emulated UEs running on an emulated eNodeB to mimic a large number of UEs. The ECHO EPC core ran on the Microsoft Azure cloud on Standard_DS3 general purpose instances with 4 cores and 14GB memory [45]. The ECHO EPC core consists of an MME pool with 2 MME instances, a ZooKeeper (ZK) cluster with 3 ZK nodes, and other EPC components – SGW, PGW, HSS – all ran on Azure. We compared ECHO with a conventional virtual EPC deployment, which was an OpenEPC Release 6 instance [17] – the most mature virtual EPC software as we are aware of at the time of this evaluation – deployed in PhantomNet.

Trial setup. We also had a trial deployment [43] with five E40 LTE small cells and 40 users. Since our full codebase wasn't production ready when we started the trial, we haven't deployed a full state replication system. We deployed and evaluated eNodeB and MME agent on all small cells.

Reliability options. We considered two availability options as depicted in Figure 9: single data center – all ZooKeeper nodes in the cluster were collocated in the same data center, and multiple data centers – ZK nodes were located in multiple DCs. A single DC deployment provided less reliability but also lower latency than a multi-DC deployment. We evaluated both of them as both can be relevant for different application scenarios. The network latency between the eNodeB (deployed in PhantomNet) and Microsoft Azure was around 22 ms round-trip. The 3 Azure DCs used in our experiments were 20 ms round-trip away from each other.

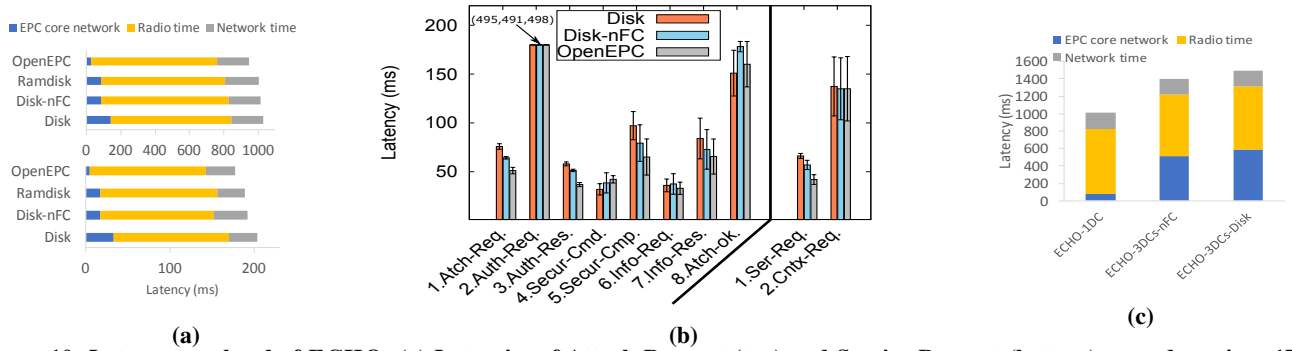
The reliability of ECHO also depends on ZooKeeper operational parameters. We evaluated three ZK logging configurations: *synchronous disk* (Disk), *asynchronous disk logging* (Disk-nFC, no force sync) and *logging to ramdisk* (Ramdisk). The synchronous disk logging is the most robust and quickest to recover, but introduces most latency (because ZK logs to disk synchronously which incurs I/O latency). The Ramdisk and Disk-nFC configurations (log to disk but don't wait before acknowledging) are two trade-offs that reduce latency but also slightly reduce the ability and speed of recovery (under extreme scenarios, data written to ZK could be lost during crash due to the async-write). Table 2 shows the deployment options and failure scenarios that they can tolerate. We compared ECHO with OpenEPC which stores UE context in memory. We also compared user perceived performance of ECHO deployments and T-mobile. We introduced node crashes to the prototype and illustrated that ECHO was robust against failure events.

Table 2: ZK configurations and cloud deployment options in ECHO evaluation with their latency and reliability profiles. The Disk-nFC and Ramdisk configurations have smaller latency but can't tolerate DC failures. The 3DCs cloud deployment has higher latency but can tolerate 1 DC failure.

Option	Latency	Robust against failures		
		Node	Avail. Zone	DC
OpenEPC	Low	No	No	No
1DC,Disk	Moderate	Yes	Yes	No
1DC,Disk-nFC	Low	Yes	Yes	No
1DC,Ramdisk	Low	Yes	Yes	No
3DCs,Disk-nFC	High	Yes	Yes	Yes

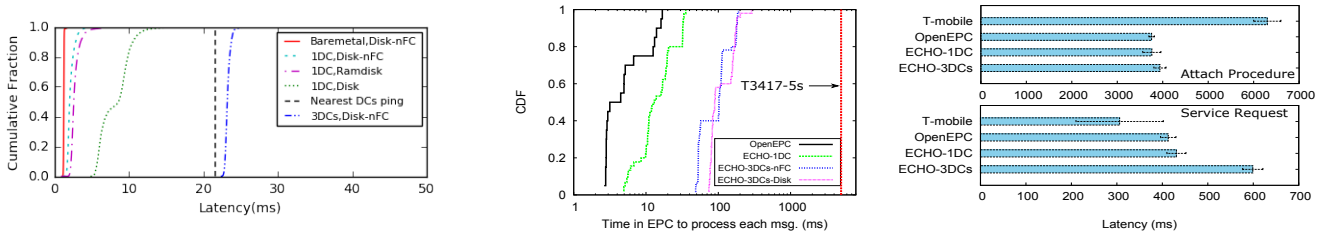
Correctness and availability analysis. We deployed ECHO on one Azure data center and ran it for 7 days. We periodically generated a Service Request and a Context Release Request every 3-minute period. In total, there were 6,720 Service and Context Release requests (i.e., 20,160 S1AP messages) generated from a Nexus 5 device attached to a ip.access LTE eNodeB. The system remained stable and all requests were correctly processed. We next randomly introduced node reboot and process crash events happened on 1% of S1AP messages (i.e., mimicking two 9s availability of the DC, which is orders of magnitude less available than the measured availability). ECHO recovered from crashes and proceeded all S1AP messages successfully.

We were also interested in ECHO's availability using the measured data in § 3.3 and analytics. Multiple data centers have a high degree of failure independence, and cloud providers try to extend that property to availability zones [44]. In practice, some degree of correlated failures are unavoidable, but assuming independence we can calculate ECHO's availability. While ECHO can't assure the calculated 9s in practice, the calculated result suggests ECHO's promise of availability. Given p is the probability that a VM is available, the probability that a ZK ensemble (with 3 nodes) is available is $P(ZK) = p^3 + 3p^2(1-p)$ (i.e., either 3 nodes are



(a) (b) (c)

Figure 10: Latency overhead of ECHO: (a) Latencies of Attach Request (top) and Service Request (bottom) procedures in a 1DC deployment with different ZK configurations, observed on eNodeB; (b) Latency of each individual message in an Attach Request (left part) and Service Request (right part) procedure in the 1DC deployment scenario with different ZK configurations; (c) Latencies of an Attach Request procedure in 1DC and 3DC deployment scenarios.



(a) (b) (c)

Figure 11: Latency vs. reliability trade-off: (a) Network latency CDF for ZooKeeper write. Baremetal showed optimal, non-virtualized performance; (b) CDF of message latency for an Attach Request procedure with different deployments and different ZK configurations; (c) UE-perceived latency for Attach Request procedure (top) and Service Request procedure (bottom), compared with T-mobile

available or 2 nodes are available and 1 node fails). With $p = 0.99947$ (Table 1), $P(ZK) = 0.999999$. ECHO's availability also depends on the availability of each component. Given a component with t presence points (i.e., DC clusters, observed from the Internet), a component is available if at least one DC cluster is available. Given q is the probability that a DC cluster is available, ECHO's component availability is $P(component) = 1 - (1 - q)^t = 0.99999999$ (with $q = 0.99991$ as in Table 1 and $t = 2$). This suggests six 9s availability of ECHO with 2 DC clusters and a ZK ensemble with 3 nodes in different availability zones.

Latency. Figures 10a shows latency of an entire Attach (top) and Service Request (bottom) procedures with different ZK configurations running in a DC. The latency was broken down into EPC core network – the latency incurred within the EPC core (i.e., processing time and network time incurred on EPC components and the ZK cluster); Network time – the round-trip time between eNodeB and Azure; and Radio – the latency to set up radio bearers on UE and eNodeB hardware. Overall, ECHO introduced only about 7% (70 ms) extra latency for an Attach Request compared to OpenEPC. The overall latency was dominated by the radio configuration latency between UE and eNodeB (i.e., RRC latencies).

Individual message overheads. Figure 10b shows the latency overhead ECHO introduced to each message exchanged

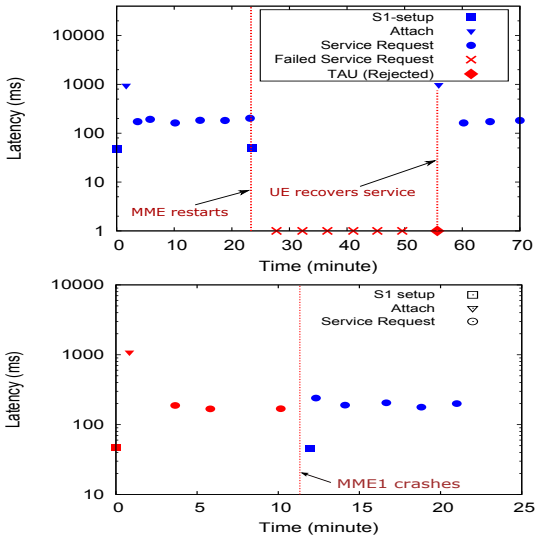


Figure 12: An MME crash in a conventional EPC deployment resulted in a long outage for the UE (upper Figure). In the other hand, despite the MME crash a ECHO deployment with two ECHO MME instances seamlessly proceeded requests from the UE as if there were no failures (lower Figure).

between UE and MME in an Attach Request (left part) and Service Request (right part). The odd-numbered messages (i.e., 1-Attach Request, 3-Authentication Response, 5-Security

Mode Complete, 7-UE Information Response, 1-Service Request) were sent by the UE and processed by ECHO. The even-numbered messages (i.e., 2-Authentication Request, 4-Security Mode Command, 5-UE Information Request, 8-Attach Accept, 2-Context Setup Request) were sent by ECHO and processed by the UE. The results confirmed that radio bearer setup and authentication on UE (msgs. 2-left, 8, 2-right) dominated the total procedure latency. Looking at ECHO latency (i.e., msgs. 1-left, 3, 5, 7, 1-right) we can see a latency trend among ECHO-Disk, ECHO-nFC and OpenEPC. Overall, using disk logging incurred the most latency overhead while using disk without force sync (Disk-nFC) incurred less latency.

Reliability vs. Latency trade-off. Figure 10c shows latency of an Attach Request with ZK deployed in a single DC and 3 DCs. As expected, compared to the single DC deployment, ECHO with multiple-DC deployments incurred extra latency because of network latency between ZK nodes (40% or 400 ms more for the Attach Request procedure). Note that the multiple-DC deployment, however, can tolerate a single data center outage. Depending on the response time and reliability characteristics required, one may favor one option over the other. For example, public Internet outages can simply be relayed from reachable data centers if this is a viable option for a particular deployment. However, even with the most extreme deployment, ECHO incurred overhead is still tolerable for UE operating 3GPP protocols. We further probed into this by showing the CDF of the latency of each ZK write (Figure 11a) and each message on ECHO MME in an Attach Request procedure (Figure 11b). Replication to 3 DCs incurred 10× messaging latency as it invoked several ZK writes. Yet, this was still only a fraction of the total latency and well below the smallest timeout value of an UE – 5s for T3417 (see section 10.2 in 3GPP NAS timers [1], 3GPP S1AP timers [2].)

UE-perceived latency. Figure 11c shows the latencies of Attach Request and Service Request procedures perceived by a UE on ECHO and T-mobile. Since we couldn't capture T-mobile control messages inside their proprietary EPC deployment, we measured the latency by triggering the Attach Request and Service Request on the Nexus 5, using the same methodology on both platforms for a fair comparison. To trigger an Server Request we let the device idle to release its radio connection, and then issued a ICMP request. We then measured the UE-perceived latency as the RTT reported by the first ICMP reply subtracted by the RTT of the following ICMP reply which doesn't include LTE control latency. Note that these latencies include both the phone's operating system latency and control plane latency. Overall, latency of a single-DC deployment of ECHO was comparable to T-mobile. ECHO's latency was worse than T-mobile if it was deployed in 3 data centers. Because of potential differences in authentication methods and radio performance of the E40 smallcell and T-mobile's macro eNodeB, ECHO's Attach was faster while Service Request was slower than T-mobile.

Throughput. We measured throughputs of Attach Request and Service Request procedures in ECHO and compared the throughputs with OpenEPC using identical hardwares. We used 1000 emulated UEs on an eNodeB to generate simultaneous requests until we saturated the systems and measured the throughputs. For the two deployment scenarios (single DC and across 3 DCs), the ECHO's two throughput results were similar to OpenEPC's with ≈900 messages/s (figures omitted). This confirmed that ECHO offered high availability without scarifying the system's throughput.

Failure scenarios. Figure 12 shows in detail how differently OpenEPC and ECHO reacted to a failure. Under the conventional EPC deployment in the upper Figure, the attached UE was not able to use the network after the MME crashed; the UE didn't receive service despite sending Service Requests repeatedly from the 23rd minute until the 56th minute (denoted by red crosses in the Figure). Only until the 56th minute into the experiment, the UE issued a periodic Tracking Area Update (TAU) and failed (denoted by the red diamond shape), it then re-attached and succeeded (the blue triangle at the 56th minute). After the successful re-attach, the UE had service again and the following Service Requests (blue round dots after the 60th minute) were also successful. In total, the UE experienced a 33-minute outage in the conventional EPC deployment. On the other hand, in a ECHO deployment with 2 MME instances in the lower Figure, the UE seamlessly had services as if the were no failures even when one MME instance crashed. As in the lower Figure of Figure 12, there were 2 MME instances, MME1 and MME2, running in a ECHO deployment. At the 12th minute into the experiment, we restarted the MME1 instance. The eNodeB received a S1AP reset signal from the MME1 instance and reestablished the S1AP connection with the MME2 instance (the blue square at 12th minute). The UE's Service Requests were processed *successfully* by the MME2 (blue round dots after the 12th minute). Note that the 1st Service Request after the crash of the MME1 instance (at around 12th minute) had a slightly higher latency. This was because the MME2 instance didn't has the UE's context in its memory and had to fetch the UE's context from the ZK cluster, which resulted in the extra latency.

eNodeB client. In order to test the overhead of the ECHO's eNodeB client, we deployed it on five IP Access E40 eNodeBs [28] running a live trial with 40 users. According to the specifications, each E40 eNodeB supports up to 16 concurrently active users. The highest load we registered was 14 users per eNodeB. During the 3-month trial, we have not

observed any performance degradation in any of the eNodeBs. We also observed that eNodeB client was able to quickly re-establish TCP connection on intermittent failures without eNodeB noticing, which avoided connection drops and made network more stable.

7 DISCUSSION

End-to-end security: ECHO doesn't break the end-to-end security contexts of EPC. ECHO adds another layer on top of the EPC's SCTP protocol and therefore adds an extra encapsulation out of the original S1AP packets. This extra encapsulation, however, is only visible to the ECHO layer and is pealed off before the packet is delivered to the S1AP layer. Since the keys and counters are kept in the central storage, a new MME instance can fetch the information and continue to process S1AP messages after failures.

Further optimizations to reduce latencies: ECHO introduced latencies could be further reduced to support future low-latency use cases. For example, using closer data centers could reduce ZK read/write latencies. A closer integration of a consensus protocol into Algorithm 1 could also reduce the number of ZK writes.

8 RELATED WORK

Our work is related to efforts in network function virtualization in general [19, 21, 23, 67, 68], as well as more closely related virtualized mobile network efforts focused on resource management and scalability [30, 53–55], orchestration of virtualized core network functions [61], and virtualizing specific core network components [12]. PEPC [54] only deals with data plane performance of the mobile core and doesn't consider reliability. Perhaps most closely related to ECHO are the virtualized MME architectures proposed in SCALE [11] and DMME [8]. SCALE and DMME proposed to horizontally scale the MME using load balancing and state replication. However, SCALE and DMME focus only on scalability of a single (MME) component. They do not deal with reliability issues – if an MME instance is slow or crashes, stale requests could cause state inconsistencies.

Various studies have dealt with availability and reliability concerns of cloud platforms [13, 14, 24, 25]. Alternative approaches to our work to address these concerns include mechanisms to make clouds inherently more reliable [62], service abstractions to hide the complexities of dealing with cloud failures from application developers [29] and attempts to add specialized cloud features to deal with cloud fault tolerance [50, 51]. ECHO took a different approach to assume the cloud infrastructure is not reliable and instead used software and protocols to enhance availability.

ECHO's replication strategies relate to state machine replication (SMR) [56], a well-known approach to building fault-tolerant, highly available services [16, 27]. However, naively reimplementing MME logic in replicated state machines does not work. The side effects that an MME performs on state transitions make this hard; the (distributed) effects fundamentally cannot be atomic with the state change in the MME. If an MME replica crashes in the middle of interacting with other entities, then the intended effect may not have been achieved. Another MME replica cannot know where to resume to avoid duplicating effects. So the system must work correctly even when side effects are performed more than once. SMR also intertwines scaling, partitioning and fault-tolerance, since state machines are stateful. SMR plays a role in ECHO, but in the form of ZooKeeper's [27] fault-tolerant atomic broadcast protocol, Zab [31].

ECHO's enforcement of FIFO and atomicity is similar to virtually synchronous CBCAST from the ISIS toolkit [15]. However, ECHO is the first to combine atomic and FIFO processing over distributed components in a cellular network leveraging the reliable base station. The key challenge is in minimizing changes to the existing EPC protocol and in interactions with the outside UE, which cannot be modified. Others observed this issue with clients in other contexts [35]. The necessary reliability between the UE and its eNodeB simplify this, since the radio control link offers a reliable, ordered connection with the UE. Setty et. al. [59] proposed "locks with intent" for building fault-tolerant systems on cloud storage. In ECHO, each client only affects its own state, which eliminates the need for intents.

The idea of separating state from computation is also proposed in StatelessNF [32]. Logical NF in StatelessNF, however, is deployed in isolation, i.e., it assumes crash failures. Unlike StatelessNF, ECHO is non-blocking on failure; because conflicting operations are resolved at the state store, having multiple components affecting the same state at the same time is safe. This makes it safe ECHO to continue operation even when nodes are slow rather than crashed.

9 CONCLUSIONS

We present and evaluate ECHO, a scalable architecture that enhances availability of the LTE/EPC's control plane. ECHO provides the same properties offered by the conventional LTE/EPC control plane despite failures. ECHO is proven correct and promises higher availability compared to commercial cellular networks today with potential lower cost and minimal overheads when deployed in a public cloud.

10 ACKNOWLEDGMENT

We would like to thank our anonymous reviewers and shepherd for their valuable comments. This work was initiated when the primary author was an intern at Microsoft Research, and is supported in part by the National Science Foundation under grant numbers CNS-1305384 and CNS-1566175.

REFERENCES

[1] 3GPP. Non-Access-Stratum (NAS) protocol for Evolved Packet System (EPS). http://www.etsi.org/deliver/etsi_ts/124300_124399/124301/10.03.00_60/ts_124301v100300p.pdf.

[2] 3GPP. S1 Application Protocol (S1AP)(Release 12) - Network, Evolved Universal Terrestrial Radio Access, 2011.

[3] 3GPP. 3GPP TS 23.401 - General Packet Radio Service (GPRS) enhancements for Evolved Universal Terrestrial Radio Access Network (E-UTRAN) access. http://www.etsi.org/deliver/etsi_ts/123400_123499/123401/08.14.00_60/ts_123401v081400p.pdf, 2015.

[4] ADYA, A., GRUBER, R., LISKOV, B., AND MAHESHWARI, U. Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 1995), SIGMOD '95, ACM, pp. 23–34.

[5] AMAZON. AWS Direct Connect. https://aws.amazon.com/directconnect/.

[6] AMAZON AWS. Amazon Compute Service Level Agreement. https://aws.amazon.com/ec2/sla/.

[7] AMAZON AWS. Configure Health Checks for Your Classic Load Balancer. https://docs.aws.amazon.com/elasticloadbalancing/latest/classic/elb-healthchecks.html.

[8] AN, X., PIANESE, F., WIDJAJA, I., AND GUNAY ACER, U. Dmme: A distributed lte mobility management entity. *Bell Lab. Tech. J. 17*, 2 (Sept. 2012), 97–120.

[9] AT&T. AT&T Domain 2.0 Vision White Paper. https://www.att.com/Common/about_us/pdf/AT&TDomain2.0VisionWhitePaper.pdf, 2013.

[10] BANERJEE, A., CHO, J., EIDE, E., DUERIG, J., NGUYEN, B., RICCI, R., VAN DER MERWE, J., WEBB, K., AND WONG, G. Phantomnet: Research infrastructure for mobile networking, cloud computing and software-defined networking. *GetMobile: Mobile Computing and Communications 19*, 2 (2015), 28–33.

[11] 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.

[12] BASTA, A., KELLERER, W., HOFFMANN, M., HOFFMANN, K., AND SCHMIDT, E. D. A virtual sdn-enabled lte epc architecture: A case study for s-/p-gateways functions. In *2013 IEEE SDN for Future Networks and Services (SDN4FNS)* (Nov 2013), pp. 1–7.

[13] BENSON, T., SAHU, S., AKELLA, A., AND SHAIKH, A. A first look at problems in the cloud. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing* (Berkeley, CA, USA, 2010), HotCloud'10, USENIX Association, pp. 15–15.

[14] BIRKE, R., GIURGIU, I., CHEN, L. Y., WIESMANN, D., AND ENGBERSEN, T. Failure analysis of virtual and physical machines: Patterns, causes and characteristics. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (June 2014), pp. 1–12.

[15] BIRMAN, K., SCHIPER, A., AND STEPHENSON, P. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems (TOCS) 9*, 3 (1991), 272–314.

[16] BURROWS, M. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 335–350.

[17] CND. OpenEPC - Core Network Dynamics. http://www.corenetdynamics.com/.

[18] ERICSSON. High Availability is more than five nines. https://www.ericsson.com/real-performance/wp-content/uploads/sites/3/2014/07/high-avaialbility.pdf.

[19] ETSI. Network Functions Virtualisation (NFV); Management and Orchestration. ETSI GS NFV-MAN 001 V1.1.1 (2014-12).

[20] ETSI. NFV White Paper. https://portal.etsi.org/nfv/nfv_white_paper.pdf.

[21] ETSI. Network Functions Virtualisation (NFV); Architectural Framework. ETSI GS NFV 002 V1.1.1 (2013-10), 2013.

[22] FAYAZBAKHSH, S. K., REITER, M. K., AND SEKAR, V. Verifiable network function outsourcing: Requirements, challenges, and roadmap. In *Proceedings of the 2013 Workshop on Hot Topics in Middleboxes and Network Function Virtualization* (New York, NY, USA, 2013), HotMiddlebox '13, ACM, pp. 25–30.

[23] GEMBER-JACOBSON, A., VISWANATHAN, R., PRAKASH, C., GRANDL, R., KHALID, J., DAS, S., AND AKELLA, A. Opennf: Enabling innovation in network function control. In *Proceedings of the 2014 ACM Conference on SIGCOMM* (New York, NY, USA, 2014), SIGCOMM '14, ACM, pp. 163–174.

[24] GILL, P., JAIN, N., AND NAGAPPAN, N. Understanding network failures in data centers: Measurement, analysis, and implications. *SIGCOMM Comput. Commun. Rev. 41*, 4 (Aug. 2011), 350–361.

[25] GUNAWI, H. S., HAO, M., SUMINTO, R. O., LAKSONO, A., SATRIA, A. D., ADITYATAMA, J., AND ELIAZAR, K. J. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing* (New York, NY, USA, 2016), SoCC '16, ACM, pp. 1–16.

[26] GUO, C., YUAN, L., XIANG, D., DANG, Y., HUANG, R., MALTZ, D., LIU, Z., WANG, V., PANG, B., CHEN, H., LIN, Z.-W., AND KURIEN, V. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication* (New York, NY, USA, 2015), SIGCOMM '15, ACM, pp. 139–152.

[27] HUNT, P., KONAR, M., JUNQUEIRA, F. P., AND REED, B. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference* (2010), vol. 8, p. 9.

[28] IP ACCESS. E-40 Access point.

[29] JHAWAR, R., PIURI, V., AND SANTAMBROGIO, M. Fault tolerance management in cloud computing: A system-level perspective. *IEEE Systems Journal 7*, 2 (June 2013), 288–297.

[30] JIN, X., LI, L. E., VANBEVER, L., AND REXFORD, J. Softcell: Scalable and flexible cellular core network architecture. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies* (2013), ACM, pp. 163–174.

[31] JUNQUEIRA, F. P., REED, B. C., AND SERAFINI, M. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)* (2011), IEEE, pp. 245–256.

[32] KABLAN, M., ALSUDAIS, A., KELLER, E., AND LE, F. Stateless network functions: Breaking the tight coupling of state and processing. In *NSDI* (2017), pp. 97–112.

[33] KEEPALIVED. Software Design. https://tinyurl.com/y94whg8h.

[34] LAMPORT, L. The Part-time Parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169.

[35] LEE, C., PARK, S. J., KEJRIWAL, A., MATSUSHITA, S., AND OUSTERHOUT, J. Implementing Linearizability at Large Scale and Low Latency. In *Proceedings of the 25th Symposium on Operating Systems Principles* (New York, NY, USA, 2015), SOSP '15, ACM, pp. 71–86.

[36] LEUNG, K. K. Mobile ip mobility agent standby protocol, Feb. 27 2001. US Patent 6,195,705.

[37] LISKOV, B. Distributed Programming in Argus. *Communications of the ACM 31*, 3 (Mar. 1988), 300–312.

[38] LUCENT, A. LTE Subscriber Service Restoration - Application Note. http://www.tmcnet.com/tmc/whitepapers/documents/whitepapers/2014/10085-lte-subscriber-service-restoration.pdf.

[39] LUCENT, A. Study of EPC Nodes Restoration - technical report. ftp://ftp.3gpp.org/specs/archive/23_series/23.857/23857-140.zip.

[40] MICROSOFT. Azure Load Balancer overview. https://docs.microsoft.com/en-us/azure/load-balancer/load-balancer-overview.

[41] MICROSOFT. ExpressRoute. https://azure.microsoft.com/en-us/services/expressroute/.

[42] MICROSOFT. Monitor availability and responsiveness of any web site. https://docs.microsoft.com/en-us/azure/application-insights/app-insights-monitor-web-app-availability.

[43] MICROSOFT. Project Belgrade. https://www.microsoft.com/en-us/research/project/project-belgrade/.

[44] MICROSOFT. Regions and Availability. https://docs.microsoft.com/en-us/azure/virtual-machines/windows/regions-and-availability.

[45] MICROSOFT AZURE. Linux Virtual Machines. https://tinyurl.com/z4rlnzs.

[46] MICROSOFT AZURE. SLA for Virtual Machines. https://tinyurl.com/y8dtlad9.

[47] MICROSOFT AZURE. Understand Load Balancer probes. https://docs.microsoft.com/en-us/azure/load-balancer/load-balancer-custom-probe-overview.

[48] NOKIA. Nokia 7750 Service Router - Mobile Gateway - Data Sheet. http://resources.alcatel-lucent.com/?cid=141247.

[49] NOKIA. Nokia 9471 Wireless Mobility Manager Mobility Management Entity/Serving GPRS Support Node - Data Sheet. https://resources.alcatel-lucent.com/theStore/files/Nokia_9471_WMM_MME_SGSN_WM9_Data_Sheet_EN.pdf.

[50] OPENSTACK. Vitrage. https://wiki.openstack.org/wiki/Vitrage.

[51] OPNFV. Doctor. https://wiki.opnfv.org/display/doctor/Doctor+Home.

[52] PATEL, P., BANSAL, D., YUAN, L., MURTHY, A., GREENBERG, A., MALTZ, D. A., KERN, R., KUMAR, H., ZIKOS, M., WU, H., KIM, C., AND KARRI, N. Ananta: Cloud scale load balancing. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM* (New York, NY, USA, 2013), SIGCOMM '13, ACM, pp. 207–218.

[53] QAZI, Z. A., PENUMARTHI, P. K., SEKAR, V., GOPALAKRISHNAN, V., JOSHI, K., AND DAS, S. R. Klein: A minimally disruptive design for an elastic cellular core. In *Proceedings of the Symposium on SDN Research* (New York, NY, USA, 2016), SOSR '16, ACM, pp. 2:1–2:12.

[54] 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* (2017), ACM, pp. 348–361.

[55] RAJAN, A., GOBRIEL, S., MACIOCCO, C., RAMIA, K., KAPURY, S., SINGHY, A., ERMANZ, J., GOPALAKRISHNANZ, V., AND JANAZ, R. Understanding the bottlenecks in virtualizing cellular core network functions. In *Local and Metropolitan Area Networks (LANMAN), 2015 IEEE International Workshop on* (Apr. 2015), pp. 1–6.

[56] SCHNEIDER, F. B. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys 22*, 4 (Dec. 1990), 299–319.

[57] SDX CENTRAL. AT&T Misses Its 2016 OpenStack Deployment Goal. https://www.sdxcentral.com/articles/news/att-misses-2016-openstack-deployment-goal/2017/02/.

[58] SEKAR, V., EGI, N., RATNASAMY, S., REITER, M. K., AND SHI, G. Design and implementation of a consolidated middlebox architecture. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2012), NSDI'12, USENIX Association, pp. 24–24.

[59] SETTY, S., SU, C., LORCH, J. R., ZHOU, L., CHEN, H., PATEL, P., AND REN, J. Realizing the Fault-tolerance Promise of Cloud Storage Using Locks with Intent. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2016).

[60] SHERRY, J., HASAN, S., SCOTT, C., KRISHNAMURTHY, A., RATNASAMY, S., AND SEKAR, V. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2012), SIGCOMM '12, ACM, pp. 13–24.

[61] SYED, A., AND VAN DER MERWE, J. Proteus: A network service control platform for service evolution in a mobile software defined infrastructure. In *International Conference on Mobile Computing and Networking (MobiCom)* (2016).

[62] TALEB, T. Toward carrier cloud: Potential, challenges, and solutions. *IEEE Wireless Communications 21*, 3 (June 2014), 80–91.

[63] TELCO TRANSFORMATION. Seen and Heard: Verizon, AT&T Target Enterprises With NFV. https://tinyurl.com/ya7j5mv9.

[64] THE LINUX FOUNDATION. Carrier Grade Linux Requirements Definition. https://tinyurl.com/y9wfn9rk.

[65] THOMSON, A., DIAMOND, T., WENG, S.-C., REN, K., SHAO, P., AND ABADI, D. J. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), ACM, pp. 1–12.

[66] WALE, K. Implementing ATCA Serving Gateways for LTE Networks. http://go.radisys.com/rs/radisys/images/paper-atca-implementing.pdf.

[67] WOOD, T., RAMAKRISHNAN, K., HWANG, J., LIU, G., AND ZHANG, W. Toward a software-based network: integrating software defined networking and network function virtualization. *Network, IEEE 29*, 3 (May 2015), 36–41.

[68] XILOURIS, G., TROUVA, E., LOBILLO, F., SOARES, J., CARAPINHA, J., MCGRATH, M., GARDIKIS, G., PAGLIERANI, P., PALLIS, E., ZUCCARO, L., REBAHI, Y., AND KOURTIS, A. T-NOVA: A marketplace for virtualized network functions. In *Networks and Communications (EuCNC), 2014 European Conference on* (June 2014), pp. 1–5.