ECHO: A reliable distributed cellular core network for 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 public cloud platforms offer flexibility and cost-effectiveness, resulting in various services and businesses moving to the cloud. One area with little progress is cellular core networks. A cellular core network manages states 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.

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 [18]. Cellular core networks are largely still based on custom-built hardware mandated by the strict reliability requirements posed by running a mobile core network.

To alleviate these challenges, telcos and cellular operators are attempting to virtualize their core networks through network function virtualization (NFV) [7]. 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 provide specialized additional for fault tolerance and management – open source cloud software stack OpenStack and its OpNFV layer provide such services (e.g., see Vitrage [38] and Doctor [39]). However, such a deployment model still cannot take full advantage of the economies of scale a public deployment can offer. Telco providers will have to manage and maintain the new private cloud deployments, while at the same time, super-optimized cloud stacks 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.

Instead, the question we address is whether it is feasible to implement a cellular core network on top of a *public* cloud, such as Amazon AWS or Microsoft Azure. To achieve this, one has to address two main challenges. First, reliability - a cellular core network today requires "five 9s" reliability (i.e., availability of 99.999%) [16, 37]. Typical public cloud availability SLAs are four 9s or less, which means an order of magnitude more expected outages. Second, service abstractions mismatch. Naturally, public clouds are optimized for general workloads, offering basic network abstractions such as a network node, a private network, or a load balancer. Cellular core networks are complex, with multiple different components implementing distributed state machines that will need to be redesigned atop the cloud's abstractions.

In this paper, we introduce ECHO, a distributed cellular network architecture for the public cloud. We focus on the evolved packet core (EPC) [1], which is a key component of a cellular network without which a network cannot run. EPC manages user devices (Section 2.1) and provides core network control and data plane functionality for all cellular radio technologies (2G, 3G and 4G/LTE). ECHO is specifically designed as a distributed EPC cloud-based architecture. While operator networks consist of the EPC and middleboxes, ECHO focuses on the core EPC which is fundamentally different than other middle-boxes as it requires consistency across multiple types of network components *and* end-user devices. Although much effort has already been devoted to virtualizing conventional middleboxes [21, 46, 20, 44], these do not address the main EPC design challenges in distributed environments.

ECHO provides the same properties that EPC guarantees, but it also remains correct and available under failures. To make EPC safe against failures, ECHO must ensure the state machine remains consistent in spite of potential component and network failures. To do this, ECHO must ensure two properties: (i) a state change across multiple distributed functional components and mobile devices must appear to be atomic –

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

the distributed state machine must be either in a "before" or "after" state; and (ii) the distributed components must appear to execute requests in the order that the requests are generated by the user's mobile device. In contrast, conventional middleboxes typically share state only across multiple instances of the same functional component.

Implementing a generic distributed state machine is a challenging task. ECHO proposes a novel architecture that is specifically taylored for EPC and thus much simpler. To achieve atomicity across distributed components, ECHO leverages the "necessary" reliability of access points - mobile devices are only connected to the network as long as their associated access points are operational. ECHO introduces a thin software layer (entry point agent) on access points which ensures the eventual completion of each request - the entry point agent keeps sending a request over and over until all of the distributed components in the core network agree on a state before it moves to the next request. If a core component instance crashes in a middle of an execution, another instance can safely recover from another retry from the agent. ECHO also guarantees in-order execution of requests generated by the user's mobile device.

Our contributions can be summarized as follows:

• We propose ECHO, a distributed EPC architecture for the public cloud. ECHO uses conventional distributed systems techniques like stateless redundant components, external state storage, and load balancing for high availability and scalability but with a focus on correctness. Its key contribution is that it uses the 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*, an *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 the entry-point agent software and deploy it on a COTS LTE small cell [26]. Additionally, we implement the required EPC modifications into OpenEPC [15] 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-clients noticing it. 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.

To the best of our knowledge, ECHO is the first attempt to run an EPC 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 hope it will help inspire the next generation of 5G cellular networks, which will require greater scale and decentralization than the current architecture.

2. BACKGROUND

This section presents a brief overview of today's mobile core architectures and makes the observation that, effectively, the network core implements multiple distributed state machines, one per user.

2.1 Mobile Core Network Architecture

A cellular network consists of a wireless radio access network and a wired mobile core network. The core network consists of a control plane and a data plane.

Control Plane: The main component of the control plane in LTE/EPC is the Mobility Management Entity (MME) which is responsible for handling registration/authentication, connection setup, device mobility, etc., for mobile clients (also called User Equipment – UE). MMEs do not participate in packet forwarding but only modify routing entries in the gateways on the data plane.

Data Plane: The data plane consists of a Serving Gateway (SGW) and a Packet Gateway (PGW). They are responsible for routing and forwarding the data packets from the UEs to and from an external IP network (e.g., the Internet). When it receives a connection set up request from the control plane (MME), a data plane gateway installs a state locally and in some cases, triggers requests to other gateways, that in turn create local state associated with the request.

UE context: The MME keeps track of a *UE context* for each *attached UE*. The UE context consists of subscriber information (authentication key, UE's capability), the current state (connected or idle), and the data connection information the UE has on the data plane. The UE context on the MME reflects the "real-world" states on the data plane gateways - it contains a data connection profile (called a Evolved Packet System bearer or EPS bearer) that consists of QoS profile, end-point IDs that are set up on-demand.

Most of the UE context information is created and exchanged between the MME and the UE the first time the UE attaches to the network. However, the UE context can change after attachment depending on whether the UE is active or not; when the UE is idle for a certain amount of time (e.g., half a minute), the eNodeB releases its radio resource and triggers a resource release procedure on the gateways. If the UE has data to send, it requests the MME to re-create the data connection again via a Service Request procedure (Figure 1). Note that in all cases, it is required that the UE context be updated to match the actual states on the data plane.

2.2 Mobile core as a distributed state machine

The cellular control plane implements a distributed state

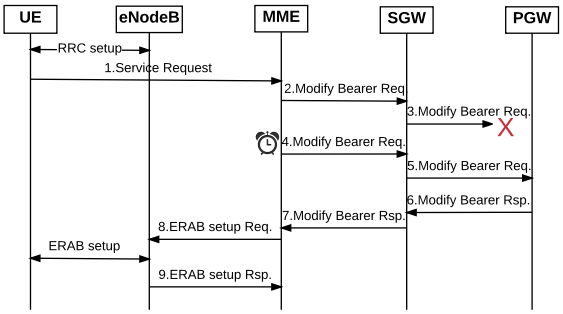

Figure 1: Service Request procedure in LTE/EPC. Once a control channel has been established across the Radio Access Network (RAN) (i.e., RRC setup), a request from the UE - *Service Request* - triggers the MME to make changes to the UE context and sends a *Modify Bearer Request (MBR)* as a side effect request to a Serving Gateway (SGW). This side effect message requests to set up a data bearer for the UE on the SGW. When it receives the *MBR*, the SGW sends another *MBR* to Packet Data Gateway (PGW) to set up a tunnel endpoint for the UE on PGW. If this request fails, a timer on the MME expires and as a result the MME retries (message #3,4). If the retry goes through successfully, the PGW acknowledges the SGW which acknowledges the MME. At this point, the MME knows that a data bearer is created for the UE (messages #6,7). The MME then informs the eNodeB with information of the data bearer (message #8.) The eNodeB then sets up a E-UTRAN Radio Access Bearer (ERAB) with the UE acknowledges the MME (message #9) when the ERAB is created.

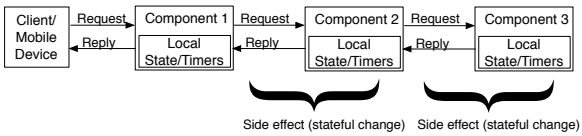

Figure 2: Distributed state in core mobile network. Components 1, 2 and 3 map to the MME, the SGW and the PGW in Figure 1.

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, which is common across all mobile core operations and components, is depicted 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: 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 sends a bearer setup request to the PGW (message #3). We call these two messages *side effect requests*. They are generated by components in the cellular core; they are indirectly triggered by the main request that originated in the UE.

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 and triggered by any component, if so required by the protocol.

Control messages: Components in the system communicate through various control messages, such as messages that trigger requests, ACKs, NACKs and other state update messages (message #1 – #9 in Figure 1).

3. RELIABILITY IN A CLOUD-BASED EPC

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

3.1 Mobile network reliability requirements

We conducted experiments with a real mobile device (Nexus 5), an LTE eNodeB (IP.Access smallcell) and the OpenEPC core network to demonstrate the core network's sensitivity to failures and its reliability needs. The results motivate ECHO's key design requirements (§4).

High availability: An MME outage would immediately cause a service outage on many UEs. Moreover, 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 outages 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. This suggests the network must be highly available.

Persistent state: The UE context exchanged during the attach procedure is kept in the MME. If this context is lost, the MME cannot process UE requests, leading to a service outage for the UE. 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*), and after a period of inactivity, the UE released a portion of the connection (at *17:06:14*, note that the UE context should be kept by the MME). After this the MME crashed and the UE context was lost. The UE then requested service (i.e., it had data to send) but did not get any service (from *17:10:01* to *17:54:03*). Approximately 54 mins after the Attach, the UE performed a periodic Tracking Area Update (TAU) procedure (*18:00:15*). This TAU also failed because the MME does not have any context of the UE. The TAU timed-out after 15 seconds. The result of this unsuccessful TAU is that the UE is moved to the EMM dereg-

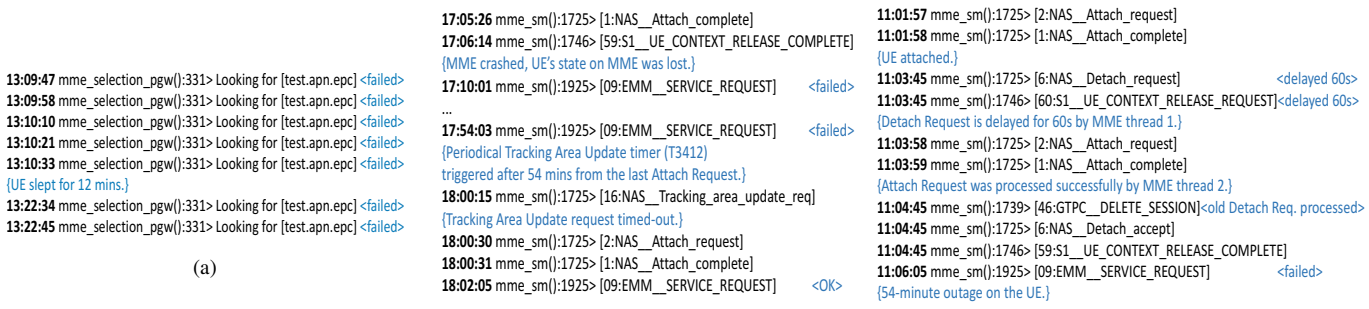

Figure 3: Examples of real-world outages caused by reliability issues: (a) 5 consecutive Attach failures caused UE to sleep for 12 mins; (b) UE did not have service for 54 minutes because MME crashed and UE context was lost; (c) Violation of FIFO order execution caused state inconsistency and 54 minutes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

istered state and as defined in the protocol [2] it performed a new Attach Request (*18:00:30*). This Attach Request was performed successfully and 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*). This suggests that the EPC network must persist UE's state to guarantee continuous operation.

In-order message delivery and execution: To maintain state consistency, requests from the same UE should be executed in *First-In-First-Out (FIFO)* order on the MME. We demonstrate this with an experiment which shows an example where the FIFO execution is violated, resulting in state inconsistency and service outage for the UE. In the experiment, a sequence of requests $\langle R_1, R_2 \rangle$ of the same UE arrive at the MME. However, request R_1 was delayed and executed *after* R_2. The result was that the stale request R_1 overwrote the effect of request R_2 which causes inconsistency between MME's state and UE's state. Figure 3c shows the MME's log with timestamps describing this experiment.

In 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. We intentionally caused a delay of 60s on this thread through a sleep timer while at the same time releasing the session lock. Later the Nexus 5 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. At this point, the UE state recorded in the *UE Context* and the actual UE state is inconsistent: recorded *Detached*, while the actual state is *Attached*. This caused a UE outage of 54 mins as in our previous experiment. This suggests that the EPC network must execute requests in the order that they are generated from the UE.

Summary: The above experiments imply that the mobile network must be highly available, must persist UE context (state), and must maintain state consistencies both between components and between the mobile device and components. The state consistencies mean that the core network must ensure (i) *in order execution* per mobile device - the distributed components must appear to process the requests in order from the mobile device's perspective (see example 3c) and (ii) *atomic execution* - the distributed components must be in either a "before" or "after" state.

3.2 Reliable EPC: state of the art

The conventional way to maintain high availability is to introduce redundancy in hardware. Telecom-grade reliable hardware is built with $N+M$ redundancy (N active blades have M back up blades) [16, 36, 50]. Active-standby techniques [30] 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 [38, 39].

Further redundancy is introduced at the protocol layer. The standard EPC architecture supports a pool of MMEs [4]. An eNodeB can connect to any of the MME instance in the pool. The MME instances share a common Session Restoration Server (SRS) [32, 31] which acts as persistent storage for UE context in real time. If one MME instance fails, the eNodeB will notice that its Stream Control Transmission Protocol (SCTP) connection to MME is broken. It will then attempt to reconnect, and will be connected to another MME instance.

To maintain atomic and in order execution, there is an SCTP connection between the eNodeB and the MME that provides retransmission, de-duplication, and request ordering. The reliable MME then maintains a FIFO queue of the requests and executes them in order, one at a time until the states are consistent across EPC components. In order for the MME pool mechanism to work, the failed MME instance must crash cleanly - after the SCTP connection is broken, all of the requests that are already in the crashed MME's queue are completely discarded so that no stale requests exist.

There are several aspects of the existing designs which do not map well to the public cloud infrastructure. Unlike public cloud, hardware appliances and VNFs offer a fine-grain availability information and scheduling control (active standby or

service migration). This allows for almost instantaneous fault isolation and repair, which is impossible in a public cloud (c.f. [24]). A public cloud EPC deployment has to deal with failures proactively, before the software is certain that a fault has occurred, as we illustrate with public cloud measurements in the next section. 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 explained in the examples above). Again, a public cloud EPC deployment has to deal with these issues in software, as most of today's distributed systems deployed in the cloud do.

3.3 What does public cloud provide?

Typical telco appliances like the ones described in the previous section provide availability of 99.999% [37], also referred to as "five 9s" availability. With five 9s availability an appliance would experience an overall outage of 1 minute in two months. Instead, cloud offerings today, such as AWS and Azure, advertise VM availabilities of "four 9s", or outages of 1 minute every week – an order of magnitude larger total outage compared to reliable hardware architectures.

Besides the overall availability in the number of 9s, the state machine 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. 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 uptimes as well as the reachability of VMs at multiple levels: data center (DC) cluster, single DC, and across DCs. A DC cluster 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 and perform TCP pings every 1 second from each VM to all other VMs. Further, we use Azure's Application Insights service [35] to monitor the reachability of our VMs from the public Internet. The service initiates web request to all 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; at least some of these cannot be handled by MME retransmissions, as illustrated in example 3a in §3.1). We observe that the advertised SLAs of four 9s are generally met by the cloud. Most of the outages are very short, and can possibly be attributed to network

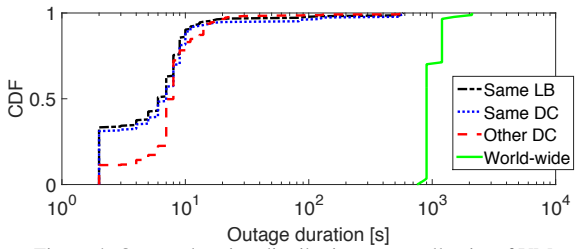

Figure 4: Outage duration distribution across all pairs of VMs

congestion, or other instantaneous problems. 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 longer than 1-second 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. However, VMs in the same DC cluster do not tend to fail at the same time.

The picture is significantly different as observed from hosts in the Internet ("World-wide" in Table 1). Availability is roughly an order of magnitude less compared to intra-DC measurements, implying that most "outages" are due to public Internet connectivity problems reaching the cloud. The CDF of the durations of the outages longer than 10 minutes (measurement interval) is depicted in Figure 4, and we can see that more than 20% of them last 20 minutes or more.

Implications: In summary, taking into account the limited duration of our study, we observe that our key requirements (high availability and state persistency) can be achieved with five 9s only if the service is replicated across multiple VMs across availability zones in a single DC; additionally, coping with public Internet reachability problems requires service presence across multiple regional data centers unless a dedicated connectivity service to the cloud [5, 34] is deployed which can incur extra cost.

We also note that other studies, such as [23] that gives account of public cloud reliability over a seven year period, point out that a median reliability across all public clouds (32 public clouds studied) is below 99.9% and that the median duration of an outage varies between 1.5 and 24 hours, depending on the type of failure. This reinforces the need for software replication and proactively dealing with faults.

4. ECHO DESIGN

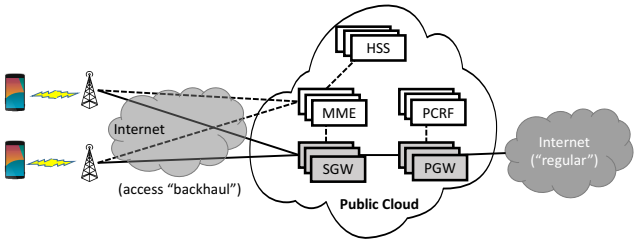

Figure 5: ECHO components at a high level.

Figure 5 presents a high-level depiction of ECHO's oper-

	Cloud				World-wide			
Outage type	VM	DC Cluster	DC	Across DCs	VM	DC Cluster	DC	Across DCs
All	99.9947%	99.9998%	99.9999%	100%	99.988%	99.991%	99.9921%	100%
$>$ 10 sec	99.9947%	99.9998%	99.9999%	100%	99.988%	99.991%	99.9921%	100%
$>$ 1 min	99.9948%	99.9998%	99.9999%	100%	99.988%	99.991%	99.9921%	100%

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

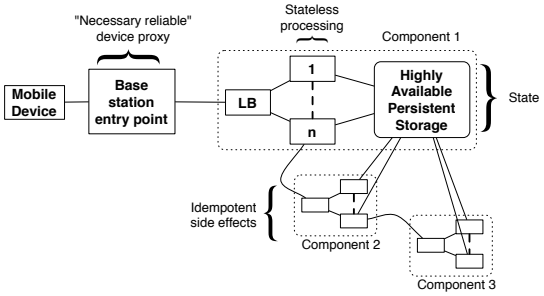

Figure 6: ECHO Overview

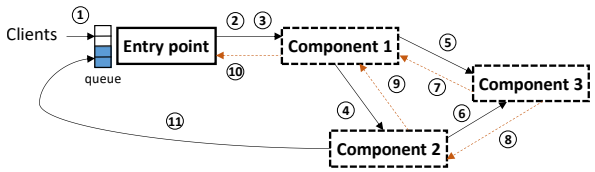

Figure 7: ECHO request example. (1) request from client, queued at entry point; (2) request forwarded to MME (but fails to be processed); (3) resent after timeout; (4,5) side effects triggered by (3); (6) side effect triggered by (4); (7,8) ack for (5) and (6); (9) ack for (4); (10) ack for (1); (11) EPC-level timeout triggered by Component 2.

ation. ECHO moves the main components of EPC into the cloud, replicating them for reliability while connectivity to the UEs is through the base stations which implement our entry point serialization. Existing public cloud load-balancers provide load-balanced connectivity to the "regular" Internet and to the Internet-based access "backhaul" between base stations and the public cloud.

We now discuss in more details the problem space, ECHO's architecture, operation and a proof of correctness.

4.1 Problem space

As discussed in §2.2, the EPC can be viewed as a distributed state machine comprising multiple components. Each component stores state for each user. ECHO must assume nodes and the connections between them can fail; the VM or container hosting a component could crash and restart or it could be arbitrarily slow, and connectivity between components is not reliable. Alternatively a node might reply with a correct response, but the replies could be late or lost.

ECHO must continue operation despite these failures while producing the same results as the original core network that assumes components are reliable; ECHO must appear to 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. Moreover, the EPC protocols are complex and constantly evolving; we must avoid modifying the protocols or relying on its implicit semantics for correctness. Finally, a particular challenge is that one of the component that stores the state is a user's mobile device, which cannot be modified.

4.2 ECHO Architecture Overview

Figure 6 depicts an overview of ECHO. Each control plane component (Components 1, 2 and 3) is replicated (instances 1 to *n*) behind a data center load balancer (LB) [40, 33]. 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 in LTE/EPC parlance, there is a "necessarily reliable" entry point. This entry point is the to ECHO's availability and correctness (§4.3).

In ECHO, each request originates at the UE and is proxied by the entry point at the base station or access point (Figure 7). Each request gets a unique, sequential ID from the entry point, and it is queued until completion. The *sequence ID* captures the order that the requests arrive at the eNodeB from the mobile device. A component *n* acknowledges a request to a previous component *n-1* once all of its downstream requests (requests to components *n+1,n+2,...*) are acknowledged. An acknowledgment to the entry point means the request from the mobile device has been applied at all components. After a timeout, the entry point retransmits the request until it is acknowledged, only after which will it move to the next request.

4.3 Necessarily reliable entry point

The *necessarily reliable entry point* 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. Moreover, note that because the number of users served by each entry point is limited by its wireless resources, its scalability constraints are different from core network components.

The entry point is a thin software layer deployed on a base station. It is similar to a sequencer in other distributed systems [49] with an additional eventual completeness property. Moreover, placing the entry point at the base station eliminates the need to enhance its reliability (e.g., by using the

state machine replication). Its API provides the following.

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: As in other protocols, components in EPC must set a timer whenever they receive a request. However, if components crash timers could be 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 an 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: Occasionally, a client moves between two eNodeBs and requires a handover. Another entry point must handle the client's operations, so state must be transferred between the entry points. This state is very light; it consists of the contents and ID of the last processed request and any registered timers or control packets. The handover procedure is augmented so the old entry point sends the context to the new entry point, which becomes the anchor for the client once the procedure is finished.

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. ECHO's operation is different from distributed ACID transactions because it enhances *both* atomicity and in-order execution (linearizability). Also, ECHO cannot simply use the state machine replication technique [43] because ECHO's operation is not atomic and deterministic; components induce side effects on other components, making determinism hard to guarantee.

Algorithm 1 shows how an ECHO component processes a request. Note that the algorithm describes two types of components, with and without side effects (as explained in §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. Note, the algorithm is *non-blocking*; multiple stateless instances of a component can execute the algorithm in parallel without causing any stall on other instances.

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 znode.
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*

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 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 monotonicity (in order execution): Each 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 could cause the state in the session store to regress, leading to inconsistency, as illustrated in example 3c in §2.

A component in ECHO uses the monotonic request IDs to filter out obsolete requests. As in line 6 in the algorithm, before processing a request, the component instance checks if the request ID of the request is less than the last executed request ID (which is stored in the persistent storage). If it is, then the request is obsolete and is discarded. When updating the persistent storage, the component increments the request ID of the session (line 15) and acknowledges the request ID to the entry point (line 20).

For example, in the 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*.

ECHO's atomicity and monotonicity: 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.

4.5 Correctness

ECHO is equivalent to the unmodified LTE/EPC network running on reliable hardwares even though components are redundant and non-blocking under failure. Please refer to appendix A for the full proof.

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 proxy agents that provide entry-point functionality on eNodeB and an interface between eNodeB and MME. There are two agents, one that resides on eNodeB and one on MME, as illustrated in Figure 8. The eNodeB's agent is implemented as a separate user-mode daemon written in standard C, deployed on top of embedded Linux running on a commodity small cell [26]. 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 [15].

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 eNodeB opens an SCTP connection to the rest of eNodeB software stack on one side (which is unmodified and unaware of the

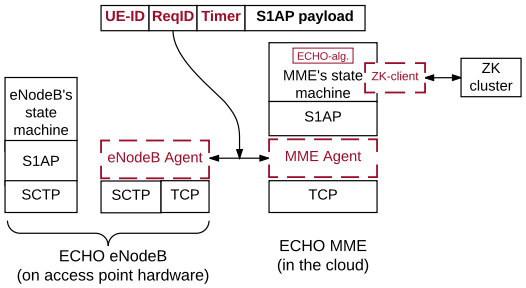

Figure 8: Modifications of ECHO in LTE/EPC

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 network stack, as shown 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 [15] 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 [25] (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.

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. In particular, we examine the correctness of our implementation, the potential latency introduced across various components of the architecture, the observed throughput and simulate potential failure scenarios. Our main findings can be summarized as follows:

• We demonstrate that our cloud-based implementation correctly services 6,720 requests over one week without any failures in the ECHO system.

• ECHO introduces reasonable overheads as a trade-off for a public-cloud reliable deployment. When replication within a single data center is used, the response latency is increased by less than 10% and there is no visible drop in throughput. Even in more extreme deployments, we show that total latency is well below standard 3GPP timeouts and would not be noticeable by the users. The result shows user-perceived latency is similar in ECHO and T-mobile.

• By emulating typical data center failures, we show that ECHO gracefully handles all such cases without noticeable user experience impact.

Evaluation setup. Our base deployment is given in Figure 9. It consists of radio equipment (a UE - Nexus 5, LTE eNodeB - IP.Access small-cell) in PhantomNet [8] and an EPC core (MME pool with 2 MMEs, a ZooKeeper ensemble with 3 nodes, and other EPC components - SGW, PGW, HSS) in Azure. Each node is a Standard_DS3 VM with 4 cores. We also use a local OpenEPC deployment in PhantomNet to compare ECHO's performance.

Reliability options. We consider two availability options. One is a single data center deployment, in which all ZooKeeper (ZK) nodes are collocated in the same data center. The other is a ZooKeeper deployment across multiple data centers, as depicted in Figure 9. The network latency between the eNodeB and Azure is around 22 ms round-trip. The 3 Azure DCs are 20 ms round-trip away from each other. A single DC deployment provides less reliability but also lower latency than a multi-DC deployment. We evaluate both of them as both can be relevant for different application scenarios.

The reliability 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). Synchronous disk logging is the most robust and quickest to recover, but introduces most latency. Ramdisk and Disk-nFC

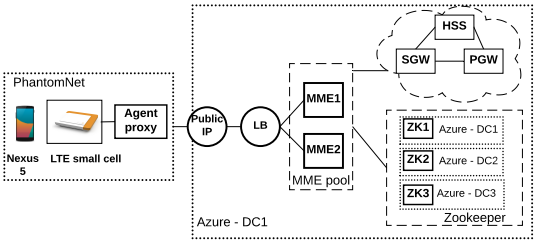

Figure 9: ECHO evaluation topology set up

(log to disc but don't wait before acknowledging) are two trade-offs that reduce latency but also slightly reduce the ability and speed of recovery. 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 and T-mobile. We introduced node crashes to the prototype and illustrate ECHO is robust against failure events.

Table 2: ZK configurations and cloud deployment options in ECHO evaluation with their latency and reliability profiles: Disk-nFC and Ramdisk configurations have smaller latency while 3DCs cloud deployment could 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. We deployed ECHO on one Azure data center and ran it for 7 days. We generated 6,720 Service and Context Release requests (20,160 messages) from a Nexus 5 device attached to a eNodeB. The system remained stable and all requests were correctly processed. We next randomly introduced node reboot and process crash events on 1% of control messages; ECHO recovered from crashes and all messages were correctly processed.

Latency. Figures 10a shows latency of entire Attach (top) and Service Request (bottom) procedures with different ZK configurations running in a DC. The latency is broken down into EPC core network - the latency between EPC components (including ZK); Network time - network round-trip time between eNodeB and Azure; and Radio - latency to set up radio bearers on UE and eNodeB hardware. Overall, ECHO introduces about 7% (70 ms) more latency for an Attach compared to OpenEPC which stores UE context in memory, which is almost negligible. The overall latency is dominated by radio bearer configuration between UE and eNodeB.

Individual message overheads. Figure 10b shows the latency overhead ECHO introduced to each message exchanged between UE and MME in an Attach (left part) and Service Request Procedure (right part). The odd-numbered messages (1-Attach Request, 3-Authentication Response, 5-Security Mode Complete, 7-UE Information Response, 1-Service Request) are sent by the UE and are processed by ECHO. The even-numbered messages (2-Authentication Request, 4-Security Mode Command, 5-UE Information Request, 8-Attach Accept, 2-Context Setup Request) are sent by ECHO and processed by the UE. The results confirm that radio setup and authentication processing on UE (msgs. 2-left, 8, 2-right) dominate the total procedure latency. Looking at ECHO latency (i.e., msgs. 1-left, 3, 5, 7, 1-right) we can see a clear latency overhead trend among ECHO-Disk, ECHO-nFC and OpenEPC. Overall, using disk logging incurs the most latency overhead while using disk without force sync (Disk-nFC) incurs less latency. The per message overhead ECHO introduced is small but noticeable, about 40%.

Reliability vs. Latency trade-off. Figure 10c shows latency of an Attach Procedure with ZK deployed in a single DC and 3 DCs. ECHO with multiple-DC deployments will survive DC failures (Table 2) yet incur higher latency because of network latency between ZK nodes (40% or 400 ms more for attach procedure). 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 probe into this by showing a CDF of the latency of each ZK write (Figure 11a) and each message on ECHO MME in an Attach procedure (Figure 11b). Replication to 3 DCs can incur 10× messaging latency as it may invoke several ZK writes. Yet, this is 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 [2], 3GPP S1AP timers [3].) In future, this could be improved by using closer data centers or closer integration of a consensus protocol into Algorithm 1 to reduce the number of writes.

UE-perceived latency. Figure 11c shows the latencies of Attach and Service Request procedures perceived by a UE on ECHO and T-mobile. Since we can't capture T-mobile control messages inside their proprietary EPC deployment, we measured the latency by triggering Attach and Service Request on the Nexus 5, using the same methodology on both platforms for a fair comparison. To trigger an Attach we toggle the airplane mode in the Nexus 5. To trigger an Server Request we let the device idle to make sure it releases radio connection, and trigger a Ping request from it to the Internet. We then measure the time it takes for the Nexus 5 to be network-available (from the trigger-time to the first Ping packet gets through.) As these latencies include phone-level overheads, they correspond to end-user perceived latencies, and they are much larger than network measured latencies (Figure 10). Overall, ECHO control procedure latency on one DC is comparable to T-mobile, but worse on 3 DCs. However, control events are infrequent, so we expect that this will not affect end-user experience.

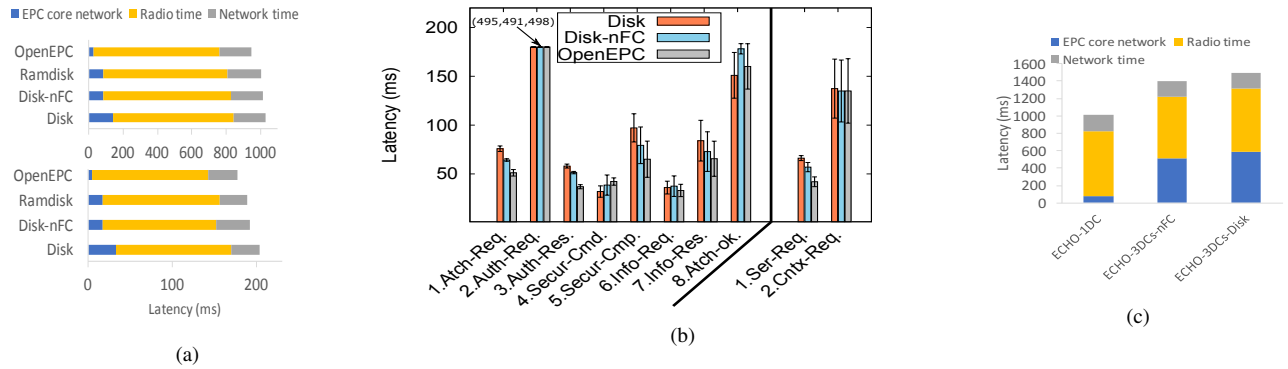

Figure 10: Latency overhead of ECHO: (a) Attach (top) and Service Request (bottom) procedures latency on 1DC deployment, observed on eNodeB; (b) Latency of each individual message in an Attach (left part) and Service Request (right part) procedures on 1DC deployment; (c) Latency for attach procedure on 1DC and 3DC deployments.

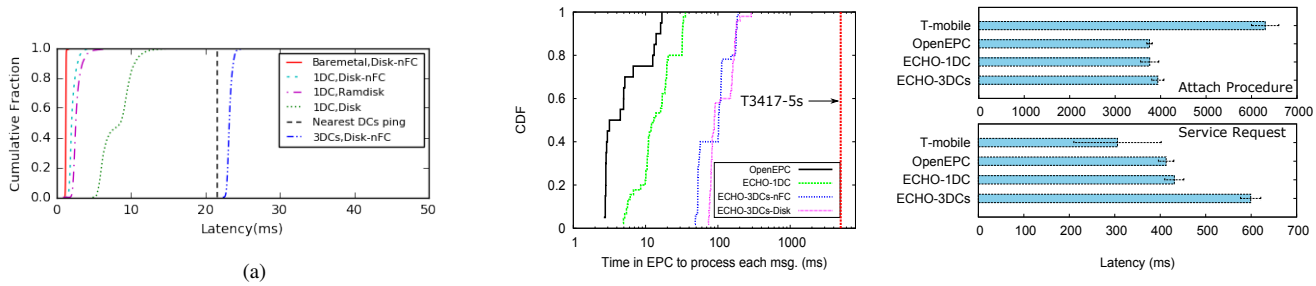

Figure 11: Latency vs. reliability trade-off: (a) Network latency CDF for ZooKeeper write. Baremetal shows optimal, non-virtualized performance; (b) Network latency CDF for attach procedure; (c) UE-perceived latency for attach procedure (top) and UE-perceived latency for service procedure (bottom)

Throughput. Figure 12a shows ECHO's peak throughput on 1 DC and 3 DCs for Attach and Service Requests. ECHO throughput is comparable to OpenEPC. Even though the throughput does not look very high, notice that each procedure consists of multiple messages exchanged (e.g., 8 messages for an Attach), and it is comparable with throughput reported in other papers [9].

Failure scenarios. Figures 12b and 12c show OpenEPC and ECHO operation when an MME crashes. The UE attached to OpenEPC was not able to use the network for 54 minutes because of the crash, whereas with ECHO the UE continued to use the network without disruption. In figure 12b, the UE attached and successfully requested services (via Service Requests) between 0-20 mins. At minute 23rd, the MME restarted. The UE was not able to use the network for 22 mins after the restart (red crosses denotes failed Service Request) until a failed periodical Tracking Area Update (at 55th minute) which triggered a re-attach (similar to example 3b). On the other hand, as in figure 12b, there were 2 MME instances in ECHO. At minute 11 the MME1 instance restarted, the eNodeB reestablished the S1AP connection after the crash, the UE's requests were load balanced to the MME2 instance and were processed successfully (blue dots after 11th minute). Note that the 1st attach request on MME2 experienced a slightly higher latency because MME2 had to contact ZK for the UE's Context. This illustrates the advantage of ECHO over OpenEPC in term of reliability against node crashes.

eNodeB client. We deployed our eNodeB's ECHO client implementation on IP Access E40 eNodeB [26] as a user-mode daemon. We configured four mobile nodes to perform data transfers and then sleep over periods of 1 minute, generating 8 requests per minute. A typical small cell can support up to 64 active users, so this represents a typical load. The induced CPU load was not noticeable on the embedded Linux monitor.

7. RELATED WORK

Our work is related to efforts in network function virtualization in general [21, 52, 51, 19, 17], as well as more closely related virtualized mobile network efforts focused on resource management and scalability [41, 42], orchestration of virtualized core network functions [47], and virtualizing specific core network components [10]. Perhaps most closely related to ECHO are the virtualized MME architectures proposed in SCALE [9] and DMME [6]. 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 and do not deal with out-of-order execution if an MME instance is slow or crashes.

Various studies have dealt with availability and reliability concerns of cloud platforms [22, 23, 12, 11]. Alternative approaches to our work to address these concerns include

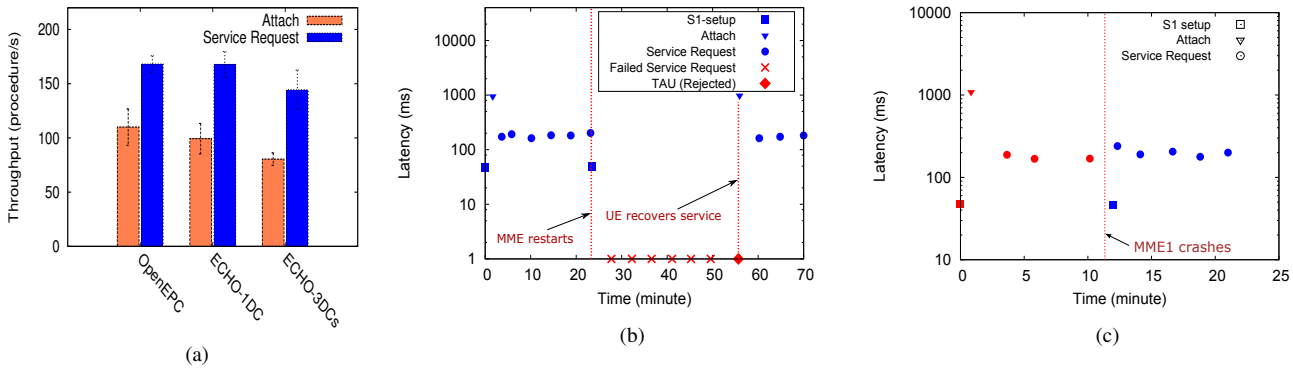

Figure 12: (a) Throughput of Attach and Service Request; (b) Unmodified OpenEPC MME crash results in an outage; (c) ECHO MME crash avoids outages.

mechanisms to make clouds inherently more reliable [48], service abstractions to hide the complexities of dealing with cloud failures from application developers [27] and attempts to add specialized cloud features to deal with cloud fault tolerance [38, 39]. 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) [43], a well-known approach to building fault-tolerant, highly available services [25, 14]. However, as in § 4.4, naively reimplementing MME logic in replicated state machines does not work. It 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 [25] fault-tolerant atomic broadcast protocol, Zab [28].

ECHO's enforcement of FIFO and atomicity is similar to virtually synchronous CBCAST from the ISIS toolkit [13]. However, ECHO is the first to combine atomic and FIFO processing over distributed components in a cellular network. 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 [29]. 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. [45] 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.

8. CONCLUSIONS

Virtualization of cellular core network protocols onto a public cloud introduces new and different challenges. In this paper we present and evaluate ECHO, a scalable and reliable architecture that can easily be implemented on top of existing core networks. ECHO is provenly correct while significantly improves mobile core networks reliability when deployed in a public cloud.

9. REFERENCES

[1] 3GPP. 3GPP TS 23.002 - Network architecture.

[2] 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`.

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

[4] 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.

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

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

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

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

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

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

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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

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

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

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

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

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

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

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

[37] 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.

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

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

[40] 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.

[41] 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.

[42] 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.

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

[44] 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.

[45] 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).

[46] 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.

[47] 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).

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

[49] 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.

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

[51] 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.

[52] 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.

APPENDIX

A. APPENDIX: ECHO'S PROOF OF CORRECTNESS

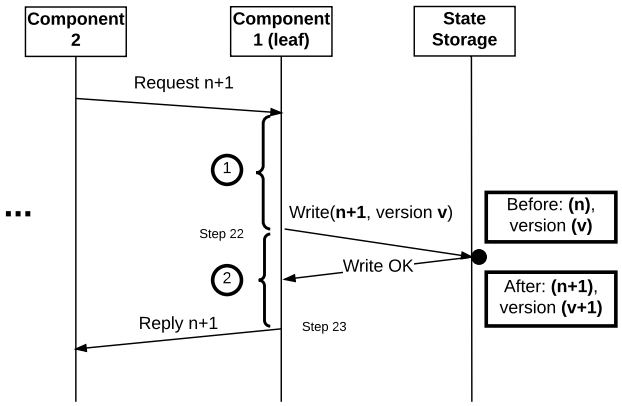

Figure 13: ECHO's leaf instance is linearizable.

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 mobile device's FIFO order (client's FIFO order), one-at-a-time demonstrates safety.

First, we show that a leaf component (a component that does not trigger side effects to other components) operates

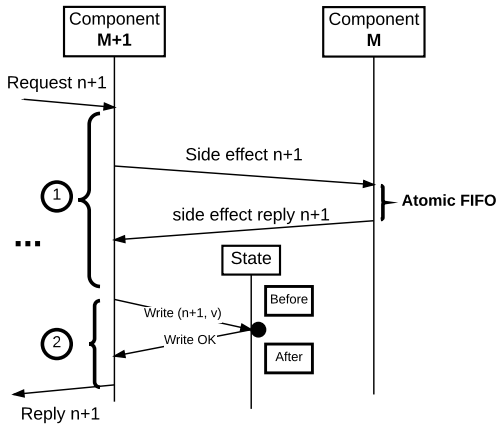

Figure 14: ECHO's components operate atomically and in client's FIFO order.

linearizably (in an order consistent with some total order of the client operations). Next, we show that the total order it is consistent with is the client's FIFO order. Then, using leaf components as the base case, it can be shown that all components appear to process operations atomically in client FIFO order.

LEMMA 1 (LEAF INSTANCE LINEARIZABILITY). *Each leaf component instance appears to process requests atomically in an order consistent with client invocation and response.*

Proof. Figure 13 shows the processing steps of a component instance handling a request. The compare-and-swap on the shared storage acts as a linearization point: before it no (local) effect of the component instance can be perceived, after it all its (local) effects are guaranteed to persist. 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 $n+1$.

In case 3, because of the eventual completeness of the entry point, there must be a retry arrived at another instance that progresses to either case 4 (in-completed) or case 2 (completed). In case 4, when a component instance receives a retry, it 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 completely failed. □

Because each compare-and-swap is issued from a natural number n to $n+1$, only one instance of any component can achieve a successful outcome in the above proof, so Lemma 1 can be strengthened:

LEMMA 2 (LEAF COMPONENT LINEARIZABILITY). *Each leaf component appears to process requests atomically in an order consistent with client invocation and response.*

Finally, because the entry point only issues request $n+1$ after the successful acknowledgment of request n (that is, operations are synchronous), no component instance can attempt to apply $n+1$ to shared storage while the state in storage is tagged with a request number less than n. This strengthens Lemma 2 to:

LEMMA 3 (ATOMIC FIFO LEAF COMPONENTS). *Every leaf component in* ECHO *processes requests atomically in the (FIFO) order client issued them to the entry point.*

Given Lemma 3 as a base case, components that make nested calls to other components can be shown to be atomic and FIFO as well using induction.

LEMMA 4 (ATOMIC FIFO COMPONENTS). *Every component in* ECHO *processes requests atomically in the (FIFO) order client issued them to the entry point.*

Proof. Non-leaf components are identical to leaf components except that they may send requests and wait for responses just before attempting to update shared storage. Let M be the *height* of a component, which is the number of nested requests that must succeed before a leaf component is reached. Leaf components have $M = 1$.

Induction hypothesis: Assuming a component at height M operates atomically in client FIFO order, then a component at height $M+1$ operates atomically and in client FIFO order.

Base case: Lemma 3 proves the case for $M = 1$.

Consider an request arriving at a component at height $M+1$. Similar to the leaf component proof above, the request has 4 outcomes: *1-aborted, 2-successful, 3-crashed before update* and *4-crashed after update*. The *crashed after update* outcome eventually results in the *successful* case as in the proof in lemma 2. In the *crashed before update* case, the component instance crashes in (1) and leaves the component $M+1$ in state n. Eventually, there is a retry $n+1$ from the entry point that arrives at another component instance and triggers the side effect again. If there are multiple retries that trigger multiple side effects to component M, the effect on component M is still atomic and in client FIFO order by the induction hypothesis. Therefore, case 4 eventually results in case 2 or 3 (which eventually results in case 2). Therefore, component $M+1$ operates requests atomically in client FIFO order. □

Finally, since the ordinary, unreplicated protocol precisely processes messages atomically, in mobile device's 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.*