rVNF: Reliable, scalable and performant cellular VNFs in the cloud

Antonios Katsarakis†, Zhaowei Tan‡, Matthew Balkwill*, Božidar Radunović*, Andrew Bainbridge*, Aleksandar Dragojević*, Boris Grot†, Yongguang Zhang*
*Microsoft Research, †University of Edinburgh, ‡UCLA

Abstract

State management is one of the main design challenges for virtual network functions (VNFs). While progress has been made with conventional IP middle-boxes, much less has been done in the cellular space. Cellular VNFs have different design requirements. In addition to high performance, they impose high availability and they manage a complex set of states that require a strong notion of transaction to achieve reliability. We argue that the current state-of-the-art does not address well these requirements.

This work introduces *rVNF*, an in-memory distributed transactional data store designed for cellular VNFs. rVNF has two parts. The first is a novel transactional protocol that explores state access locality for efficiency. The second one is a fast and replicated load balancer that enforces the access locality through customized routing. The combination of the two allows us to build a flexible state access API that can be combined into arbitrary transactions. rVNF is efficient, low-latency, highly available and offers strong transactional semantics. Our evaluation shows that rVNF can process several 100k transactions per second, can virtualize existing cellular core components and protocols with little effort, and several times outperforms existing external data stores.

1 INTRODUCTION

Telecommunication operators and other service providers increasingly move their cellular and other network workloads to private or public clouds. This is especially evident with the move to 5G that champions the cloud-native architecture as one of its cornerstones. The cloud-native network architecture introduces containerized VNF (virtualized network function) deployments featuring intelligent self-scaling and self-healing capabilities. This promises to reduce network operational expenses and improve quality of service. While operators are starting to understand and embrace the operational value of the cloud-native approach, the right technical abstractions remain elusive [10, 21, 35].

One of the most difficult issues in cloud-native architecture is managing network state. This is especially pronounced in a cellular control plane. A cellular control plane updates complex user and connection state and has several distinguishing requirements for state management. One requirement is high performance. A large cellular network has to manage millions of transactions per second. Another requirement is high availability. Each state update should either execute in its entirety or not execute at all; a failure to do so may cause long outages [25]. Cellular VNF state is also often implemented using large and complex data structures that require flexible access interfaces and strict transactional semantics to prevent inconsistencies. Finally, local processing improves performance. The locality is difficult to extract as cellular protocols use different identifiers (e.g. IMSI, tunnel ID, IP 5-tuple, etc.) to denote the same context. We elaborate these requirements in detail in Section 2.

Most of the proposed 5G VNF designs in the industry today address these requirements by borrowing microservice design patterns from web-based architectures (e.g. [24]). However, these rely on existing open-source data stores that are not optimized for cellular workloads and can be very inefficient (e.g. they are reported to process 10k-20k memory-bound transactions per multi-core server [30]). In parallel, there has been much work in the academic networking community on designing scalable, reliable and efficient stateful VNFs (also called middle-boxes) [12, 14, 17, 31, 32, 38, 44]. These designs are much more efficient, processing 100k to 1M requests per second per server. However, they have mainly focused on conventional IP middle-boxes, such as NATs, firewalls, intrusion detection systems and web proxies. None of them meet the full set of requirements for cellular VNFs, particularly those for high reliability and a strict transactional semantics. We present a detailed comparison in Section 2.3.

One way to address the missing requirements would be to build cellular VNFs directly on top of one of the existing state-of-the-art distributed transactional in-memory data stores [9, 15, 42, 43]. These systems process general-purpose transactional workloads and provide strict transactional semantics with general state access

and management primitives (e.g. object pointers and B-trees [9]). However, they also address a more general set of requirements, such as providing very large storage space and supporting access patterns without locality (e.g., graph search), and thus have to implement a general and expensive reliable distributed atomic commit (DAC [39]). In contrast, cellular and other VNFs have relatively small replicated per-flow state (a few KBs, at most), and mainly local access patterns (each flow mainly accesses its own state [44]).

The question we pose in this paper is whether we can design a distributed transactional in-memory data store that is optimized for cellular VNFs and its state access patterns. To that end, we introduce a novel concept of *transparent locality-enforcing transactions* and we design and build a distributed transactional in-memory data store called *rVNF* based on this concept. rVNF has the following key design principles:

Reliable Transactions: rVNF has strict transactional semantics (highly-available and strictly serializable [36]). A programmer can modify any object comprising the state during a transaction. If a transaction is successfully committed, all modified objects are guaranteed to be seamlessly replicated on other nodes. Otherwise, the transaction is aborted without altering the state. All state is stored across rVNF nodes and there is no external data store.

General memory abstraction: rVNF provides a transactional memory abstraction and enables access to objects in memory using object references, similarly to [9]. This allows programmers to create arbitrary data structures on top of it, such as linked lists, nested structures, etc. It also makes porting existing code much easier.

Access locality: rVNF instances are collocated with the application instances to minimize access latency. Moreover, rVNF enforces state locality whenever possible to improve performance. Each object in transactional memory has an owner server that stores it locally and has exclusive access to it. Most commonly, a transaction is processed by a node that owns all relevant memory locations. This allows us to implement a dynamic primary backup transactional system that is highly efficient and requires only one round-trip to commit a transaction. To help enforce locality, rVNF implements an object ownership management protocol that transfers ownership among nodes when required. Ownership transfer is atomic and resilient to failures.

Scalability: Finally, rVNF comes with a flexible and reliable load balancer that facilitates locality enforcement and provides dynamic state sharding that is fast and resilient to failures. It allows an external controller to implement a scale in/out by simply modifying routes for an arbitrary set of requests – the ownership management protocol will automatically migrate corresponding states when needed.

We have fully implemented rVNF in C over DPDK. We ported the main components of a cellular core on rVNF. We also ported SCTP transport protocol, allowing several compute instances to share the same SCTP transport protocol state and provide resilience to an instance failure. We show that rVNF can forward over 2M packets per second per instance while processing up to 1M small transactions per second. In our cellular core evaluation, we show that the overhead of rVNF replication is less than 30% compared to no replication. rVNF with full replication offers several times better performance than an external Redis data store without replication. We also show that a fully replicated SCTP session can achieve over 0.5 Gbps and recover from a failure.

In summary, our contributions are:

- We propose a novel architecture for scalable and reliable VNFs that provides strong transactional guarantees, state access through transactional memory, and high reliability through seamless replication.
- We introduce the notion of transparent locality-enforcing reliable transactions and build a new distributed transactional protocol, optimized for VNF access patterns through aggressive locality enforcement.
- We build an efficient implementation of rVNF and present an evaluation on several key components and protocols of cellular core control plane.

To the best of our knowledge, rVNF is the first system that demonstrates the possibility of seamless fault tolerance and scalability of all performance-critical components of the control path of a cellular core.

2 MOTIVATION

In this section we present a brief overview of the state management challenges and use them to derive major design requirements, followed by a brief review of the most relevant existing works with respect to the requirements.

2.1 Overview and Challenges

Figure 1 shows a simplified architecture of a 5G core network. The core network consists of control (AMF, SMF, etc) and data plane network functions (UPF). The control plane deals with connection and mobility management (AMF), session management and data plane setup

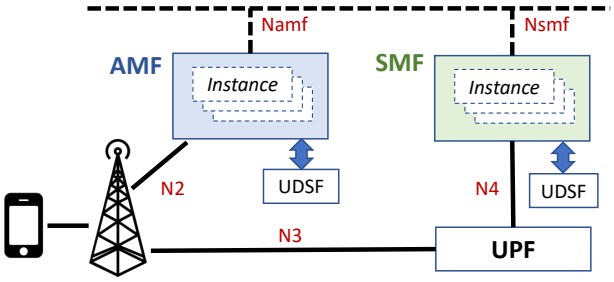


Figure 1: Simplified 5G core network overview (we omit many more control plane elements for clarity).

(SMF), authentication, charging, etc. A user first authenticates and establishes a connection through AMF, which in turns finds the appropriate SMF to set up a data plane session on a data plane element (UPF). This is similar to OpenFlow where the control plane sets up the paths and the data plane (UPF) forwards packets. Consequently, we focus our attention on the control plane elements as almost all of the network state is stored there[1]. For more details on the architecture and protocols we refer the reader to [1].

One of the main novelties in 5G is the service-based architecture (SBA) of a core network. Previous generations of cellular used connection oriented interfaces between components. These are stateful interfaces that require peers to maintain a dedicated connection between themselves in order to issue or receive requests. In 5G, most network function exposes a stateless RESTful API (e.g. Namf and Nsmf in case of AMF and SMF, Figure 1). This API can be used by other network elements to request a service from any other element without having to set up a dedicated connection. However, 5G architecture still maintains some connection oriented interfaces, such as SCTP-based N2 interface between AMF and a base station.

The architecture also proposes (but does not mandate) a functional split of a network function into multiple stateless instances executing the application logic and a shared data store. The number of instances can scale up or down depending on the demand and an external request can be served by any instance. All instances store their shared state in the data store (called UDSF - unstructured data storage function). 3GPP standard [1] does not specify the architecture of the data store nor its interfaces with other network functions. This is left to a specific implementation.

This architecture leaves several open questions regarding the optimal state management of cellular VNFs. The first set of questions is on what is the optimal split between an application logic and a data store. Should they be collocated or remote? If remote, should the application fetch the entire state of the user before processing the request (increasing the network overhead as parts of the state may not be needed)? Or should it fetch only the relevant parts of the state as the request is being executed (increasing the processing latency with an extra network RTT after each fetch)? Or should it cache the state locally and only store modified parts of the state at the data store (but it is not clear how to enforce caching locality)?

The second set of questions is on what is the optimal design of the data store layer. How to make it fast, scalable and replicated for redundancy, and also expose flexible interfaces? Should it perform some processing itself by exposing more complex, application specific calls? How can it be optimized for VNF access patterns?

The third set of questions is on how to support the remaining stateful, connection-oriented interfaces. One such example is the N2 interface that connects all base stations to the network core. Similarly, a 5G network will have to support existing 4G connection-oriented interfaces (e.g. diameter) for interoperability for the foreseeable future. The challenge with the connection-oriented interface is that if a connection fails, it is interpreted as if that the peer is down and the service is interrupted. This is why SCTP is commonly used on these interfaces as it offers a degree of fault tolerance on network issues. SCTP supports multi-homing and is able to switch from one access network to another in case of a network failure, without dropping a connection. However, current SCTP cannot move connection state from one server to another, for scaling and reliability [2]. The question is whether we can make a data store flexible enough to virtualize connection-oriented interfaces and transport protocols.

2.2 Design Requirements

In order to address the above challenges, in this section we discuss the main design requirements that state management of cellular VNFs has to meet.

Reliability (REL): Cellular VNFs require high reliability. This is important because different cellular VNFs store different parts of a state for a single connection. If one part of the state gets lost (e.g. due to a failure of a VNF instance), this can create significant outages. For example, [26] shows that if two core elements (in a 4G network) get out of sync, the inconsistency can create an outage for its users for up to 45 minutes.

[1] See [28] for an efficient design of data plane elements.

[2] Techniques such as S1Flex [41] and TLNA [1] provide some support for scaling and reliability but are not supported by most of the small cells today.

Performance and scaling (PERF): Mohammadkhan et al. [22] estimate that a 4G network with 8M customers generates about 100k user requests per second on average on a control plane, which is consistent with other traffic estimates [40]. This is likely to increase 10x-100x in 5G with many more cellular IoT nodes predicted [22]. We expect a single deployment of a cellular control plane to support up to 10M control plane transactions per second as a peak load. The deployment should also scale down at times of lower traffic to save compute cost.

Flexible state access (FLEX): Cellular network components may have to deal with a complicated state. E.g., as discussed in Section 2.1, one may need to virtualize SCTP transport protocol to guarantee connection persistence in case of endpoint failures. However, SCTP is originally implemented as a part of Unix kernel and extensively uses data structures form kernel header files (lists, hash tables, etc.). A virtualization framework needs to support all these data structures to avoid the need for a complete rewrite of the protocol stack.

Strong notion of transaction (TRANS): In the operations described above, a single control plane operation may need to either update multiple objects atomically or none. If it replicates the updated states in a sequence of replications, it may fail half way through, leading to inconsistencies and user-facing outages. A cellular VNF thus needs transactions that are highly-available and strictly serializable [36]; in other words, each transaction must either be committed in its entirety or rolled back completely.

Dynamic multi-key routing (MULTI): State access locality (when a state is almost always accessed only by a single compute instance) allows for state caching and more efficient processing. IP-based VNFs enforce state locality when processing requests by using IP-tuples to route packets across instances. It is much more difficult to do so with cellular VNFs as they often access the same state with diverse, dynamically created keys. To see that, consider an example of a session management protocol between an AMF and an SMF. A session is removed whenever it is idle for more than 15-30 seconds, and re-established when a new data packet for that connection appears. This is one of the most frequent control plane operations in a cellular network.

An AMF requests a new session by posting a create session request over *Nsmf* interface (Figure 1) to the SMF with a set of parameters describing the user. If the request is successful, the SMF responds with a unique new session ID `smContextRef`. When the AMF wants to subsequently delete the session, it posts a delete session request to the SMF using the same `smContextRef` as a key. However, it is difficult to enforce state locality of the first (create) and the second (delete) requests because `smContextRef` is known only after the create message is executed. Therefore, any external load balancer will not have seen the `smContextRef` by the time the delete request arrives. Instead, we must be able to insert a custom route for the newly created `smContextRef` key.

2.3 Overview of Existing Works

We present an overview of the most relevant related works in Figure 2, categorized according to the key properties required for cellular VNFs. Most prior works offer performance/scaling (**PERF**) and partial reliability (**REL**), but none satisfy the full set of requirements. For instance, only FTMB [38] satisfies the **FLEX** requirement through a flexible state access API; however, FTMB runs on a single node and does not support scaling (**PERF**). Other works use custom state access primitives, mainly constrained on key-value abstractions, which are difficult to use to meet the **FLEX** requirement.

	PERF	REL	TRANS	FLEX	MULTI
Split/Merge [32]	✓				
Pico Replication [31]	✓	P			
OpenNF [12]	✓				
FTMB [38]		✓		✓	
StatelessNF [14]	✓	✓			
S6 [44]	✓				
CHC [17]	✓	P			
SCALE [8]	✓	P			✓
ECHO [25]		✓	✓		
rVNF (this work)	✓	✓	✓	✓	✓

Figure 2: Summary of the most relevant related works (P - partial).

Pico Replication [31] and CHC [17] choose to cache state locally and commit on occasional check-points for performance reasons. They are thus not able to always recover a lost state, violating **REL**. StatelessNF [14], ECHO [25] and CHC [17] use an external data store. StatelessNF [14] incurs a throughput penalty due to a latency of an external store. CHC [17] does not consider scalability and resilience properties of the data store. SCALE [8] provides scalability but does not offer reliability and transactions. ECHO [25] is the only one implementing a strong transactional semantic but it does not scale (uses a slow external data store). Cellular-focused systems, SCALE [8] and ECHO [25], both serve much lower volume of requests (in 1000s req/s per server) than the other IP-based middle-boxes (in 100,000s req/s per server).

SCALE [8] is the only system that addresses **MULTI** requirement. All others base their flow routing on IP 5-tuple. In each of them, routing is done on a separate routing component that is unaware of the application semantics.

Finally, we look at the performance results of industry deployments of scalable cellular VNF. There are no publicly available performance reports on commercial cellular cores. Instead, we look at an example of an IMS core, a set of VNFs that provide voice calls on top of a cellular network. Clearwater IMS core [24] by Metaswitch, is an open source version of their commercial product deployed in many operational networks. In a report [33], it is stated that 5-nodes cluster, deployed on AWS c5.4xlarge (16 vCPUs each), are required to process state for 15k calls and 3k registrations per second. Clearwater IMS core [24] uses Cassandra as its transactional store, which is flexible but not very efficient; it is reported in [30] that a 10 server cluster processes around 20k requests/second per 8 cores server. Although an IMS workload is not directly comparable with a cellular core, it suggests that these kinds of designs cannot meet the scalability (**PERF**) criterion with the performance similar to the IP-based middle-boxes [12, 14, 17, 31, 32, 38, 44], which measures in 100,000s requests/sec per server.

3 rVNF DESIGN

rVNF is designed to satisfy the aforementioned requirements. It offers a strict transactional semantics and a flexible API to access state data. It is efficient, has low latency, and it is able to scale in and out by moving all relevant state to a different node. Each state is reliably replicated after every transaction commits on two or more replicas and it is designed to recover from a failure of one or more nodes.

We first overview the rVNF design and then discuss the design of each component in detail.

3.1 Design Overview

rVNF architecture consists of two architectural components: a group of application nodes (ANs) that can scale in or out as needed, and a set of cheap and reliable load-balancers (LB) collocated with the ANs. Each application node runs application logic that has access to a shared transactional memory and sockets.

We explain the role of each component in a walk-through, illustrated in Figure 3. Packets coming from external peers are received by LB ①. The LB parses the packet headers using a custom stateless parser and extracts context metadata (e.g., IP 5-tuple, GPRS tunnel, etc.). It then looks up the metadata in the *routing store*.

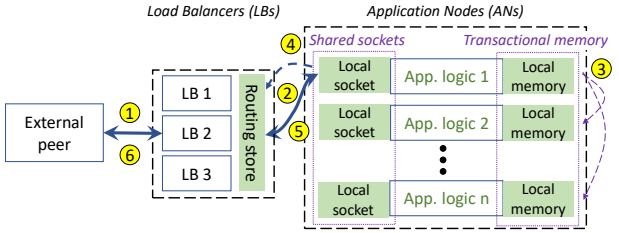


Figure 3: rVNF software architecture. An LB is collocated with an AN on the same physical server.

If the destination AN is found, it forwards the packet to it ②. If not, it selects an arbitrary destination using a user-defined load-balancing algorithm (e.g., based on the load at each AN) and stores the destination in the routing store. The routing store is replicated across LBs.

The packet is next received by a local instance of a *shared socket* layer on the AN, which provides a network socket-like interface to applications. The application logic fetches the packet through the socket API layer and processes it. During processing, the application logic may start a transaction and alter various objects that are part of the state. Our transactional protocol introduces a new concept of *transparent locality-enforcing transactions*. Each object has a primary owner that is exclusively allowed to modify it. The LB ensures that requests are routed to primary owners as often as possible. Should that not be the case (e.g., due to a scale-in), the instance processing the packet requests to become a primary owner for the relevant memory objects before processing the transaction. All this is transparent to the application logic. The transaction protocol is detailed in Section 4.

The application accesses the state objects through *transactional memory* that provides object references (similar to pointers) and key-value store abstractions. An application can build various higher level memory interfaces on top of it (lists, hash tables, etc.), as required by flexible state access (**FLEX**) property. Primary ownership is managed per (memory and key-value store) object. Note that the routing store is not a part of the transactional memory, and operates independently.

During packet processing, the application may also create response packets and send them out using the shared socket interface. At the end of processing, the application commits the transaction. At that point, all state modifications are replicated to one or more replicas ③, and the outgoing packets are transmitted ⑤ through a load balancer ⑥. Operations ③ and ⑤ are performed atomically so that either all succeed, or none. This provides strong transaction guarantees (**TRANS**) and ensures reliability of each transaction (**REL**).

If an application logic creates a new state (e.g. for

a user or a flow) as a part of packet processing, it can insert a new routing metadata, corresponding to the new state, into the routing store with an API call ④. This addresses the multi-key routing issue (**MULTI**). In the example from Section 2, an SMF can insert the session ID (`smContextRef`) in the routing store once it has been created. This will guarantee that the subsequent messages with the same session ID get routed to the same SMF instance.

When an application decides to scale out (**PERF**), it simply instantiates a new AN. An external controller can then modify an arbitrary set of existing routes to point to the new AN, at a desired pace. There is no need to pause for scaling, the state will be transferred to the new AN when needed using the ownership protocol. Scale in is analogous. Recovery protocol (**REL**) works in a similar fashion, as follows: the transactional protocol first recovers all states and creates new state replicas to replace those that resided on the crashed node. It then assigns primary ownership to new nodes for those objects that belonged to the failed node, and adds corresponding routes to LBs.

In the context of the cellular network architecture from Figure 1, *transactional memory*, *shared sockets* and LB can be seen as implementing the data store layer (UDSF) functionality, and providing service to the *application logic* (hosting AMF, SMF, etc.). For performance, our design proposes to always collocate an instance of an application with an instance of a data store. However, these instances are logically separate thanks to well defined interfaces. The application logic remains stateless in the sense that a programmer does not have to worry about replicating state (as proposed by 5G service-based architecture) but we provide a much richer data store API than conventional stateless architectures.

3.2 Load Balancer (LB)

The load balancer serves three purposes. The first one is to perform application-level routing on custom metadata extracted from packets. The second one is to participate in the dynamic state sharding and ownership protocol. Finally, it provides a reliable group membership for application nodes.

At the core of the load balancer is a *routing store*, a replicated key value store. It uses the Hermes protocol [16] for replication, which provides strong consistency, fault-tolerance, and high throughput. A load balancer instance is collocated with an application node instance, which reduces extra latency of going through a load balancer.

Load balancer can be configured to accept different protocols, and for each of them it provides a custom header processing function that extracts routing metadata (e.g., IP 5-tuple, GPRS tunnel ID, etc.). The extracted metadata is used as a key, and if a matching routing entry is found in the routing store, the packet is routed accordingly. Otherwise, a custom routing algorithm picks a new route (e.g., based on the overall load, locality of AN, etc.) and stores it in the routing store along with a time-stamp. Application logic can insert, modify or delete routes through an API to enforce locality where needed.

The second function of the load balancer is to maintain a reliable record of the *sharding metadata*: the locations of the replicas of each object and the identity of the primary owner. An update of a sharding metadata needs to atomically modify all replicas across the load balancer ensemble as well as the local states on ANs (since not all of the ANs might be collocated with load balancers). For this, the Hermes protocol does not suffice as it cannot guarantee that the state ownership transfers will occur only after any relevant ongoing transaction is fully committed. We design our own transactional protocol, described in Section 4.

Finally, the load balancer provides a reliable group membership protocol, similar to Zookeeper. This is needed to avoid inconsistencies during network partitioning of ANs, where several groups of disconnected ANs believe themselves as the only surviving group. It is the LBs that make the ultimate decision on which ANs are still functioning.

3.3 Elastic Application Nodes (AN)

Application nodes (ANs) run application code. They also run an in-memory distributed transactional data store and provide two APIs to the applications: transactional memory and sockets. These APIs are designed to hide complexities of state management and reliability, while providing interfaces that are familiar to programmers. The data store is collocated with the application to minimize access latency.

Transactions, replication and ownership: Cellular VNF state is a set of memory objects. It is mostly local per-user context (e.g. object pertaining to a flow or a device) and only a small amount of the context is shared (c.f. [44]), so our design optimizes for the former. We use the load balancer to keep the state of a single per-user context on the same node as much as possible. This state sharding leads us to use a primary-backup transactional model [6]. Each object has a single primary node and several backups nodes. The primary replica is the only one allowed to modify the object, and it is called an

owner of the object. Each modification is replicated to the backups for reliability. The transaction and ownership protocols are described in detail in Section 4.

A transaction typically involves modifying several objects from a single context. Since in most cases the primary owner of all the objects is the same AN, this AN executes the entire transaction locally, without a wait. Moreover, it can continue executing the next transaction without waiting for the replication to finish, as the ownership guarantees that no one else is modifying the same state at the time. This makes transactions fast. A transaction has to wait only if the AN has to acquire ownership (e.g. in case of scaling or failure), which is transparent to the application. For very few cases of shared memory objects that are frequently written by multiple user contexts (such as flow counters), we introduce special atomic primitives that operate without ownership (much like `MultiWriter` in [44]).

Transactional memory: A new memory object of arbitrary size can be created using a malloc-like primitive `tr_alloc`. It is then assigned a *virtual address*, a unique opaque 64-bit pointer as an identifier. At the end of the use, the object is released using `tr_free`. For convenience, a memory object of a fixed size can also be created and accessed using a key-value store abstraction `tr_get, tr_set, tr_del`. If a programmer wants to modify a memory object created using `tr_alloc`, she has to call `tr_open_write` primitive with the rVNF object pointer. This call creates a copy of the value of the memory object and returns a real pointer to that copy which the programmer can further modify. `tr_open_write` can also be used to access only a subset of an object, to reduce overhead when modifying large data structures.

```
trans *tr_create();
void tr_commit(trans *t);
tr_addr tr_malloc(trans *t, size_t size);
void tr_free(trans *t, tr_addr addr);
void *tr_open_write(trans *t,
                    tr_addr addr, size_t size);
void *tr_open_read(trans *t,
                   tr_addr addr, size_t size);
int tr_get(trans *t, uint8_t *key,
           size_t len, uint8_t *val);
int tr_set(trans *t, uint8_t *key,
           size_t len, uint8_t *val, size_t vlen);
int tr_del(trans *t, uint8_t *key, size_t len);
```

Figure 4: rVNF Transactional API.

rVNF offers strict transactional semantics. Each transaction starts with a command `tr_create` that returns a transaction handle. Each subsequent memory operation is called with the transaction handle. A transaction is committed using command `tr_commit`. When a commit is issued, all modified memory objects are replicated to their backup nodes. If backups are successful (i.e. no failures), a transaction is committed.

Whenever a program attempts to modify a memory object (using either `tr_open_write` or `tr_set`), a node executing a transaction checks whether it is the owner. If not, it issues an ownership request to the object's owner. The transaction is paused until the request is granted, or it is rolled back if the request is rejected or timed out. The ownership is also used to coordinate access across multiple local threads.

The transactional API also supports read-only operations, `tr_open_read` and `tr_get` that do not require ownership. These two operations return the most recent consistent value of the backup without needing to acquire an ownership, similar to [44] (though unlike [44], reads from backups in rVNF are never stale). A full list of API calls is given in Figure 4.

Network access: rVNF also provides a shared socket network abstraction to a programmer to receive and send packets. One AN is responsible for opening and binding a socket. All ANs use `select` or `epoll` model to get notified about incoming packets. rVNF framework routes a packet to one of the instances and triggers a notification through `select`. rVNF natively supports raw and UDP sockets; other network protocols can be implemented on top of it, as we demonstrate with the SCTP example.

rVNF socket can be reliable or unreliable. Unreliable socket is suitable for UDP traffic. If an instance that has received a packet fails before processing it, this can be seen as a network error. In some cases, such as providing a reliable SCTP abstraction, a socket should not loose a packet once it has been acknowledged to the other end of the connection. In this case, an incoming packet is stored to a transactional memory when received, and is atomically acknowledged when the transaction is committed. Similarly, an outgoing packet is buffered in a transactional memory and sent atomically only after a transaction is committed. The rVNF socket API is very similar to the POSIX socket API, except that `send`, `recv` and `select` macros have an additional parameter that points to an active transaction (which is NULL in case of unreliable sockets).

4 rVNF TRANSACTION PROTOCOL

rVNF implements transactions that are strictly serializable [36] and highly-available through replication. In this

section we first overview the motivation for our novel transactional protocol design and then we present the design in detail. We finish by discussing fault handling and verification.

4.1 Protocol Design Overview

General-purpose transactional memories shard objects and execute requests across servers (or nodes) for scalability and reliability. This poses two challenges. The first challenge is handling concurrent transactions on the same object. If two nodes try to access the same object at the same time, each through its own independent transaction, one of them has to abort. Detecting and handling these conflicts, particularly in the presence of faults, requires extra signalling across nodes. Most of the distributed transactional systems (e.g. [9, 15, 42, 43]) implement a variant of a Distributed Atomic Commit (DAC) protocol [39] that needs numerous RTTs to commit each transaction. More importantly, a node cannot start the next transaction on the same object until the commit is finished, as it cannot be sure that it will not have to abort. This introduces several RTTs of delay in the critical path of commit, which can significantly reduce the transactional throughput.

The second challenge is accessing the objects. Different object placement schemes have been proposed, such as random [5], using user-supplied hints [9], or dynamically optimized [13]. However, in all cases, object placement is best-effort and never guarantees that all objects accessed by a single transaction reside on the same node. This is particularly true for objects with high churn (e.g. flow states). If one or more objects involved in a transaction are stored remotely, the execution must stall until the objects are fetched. A programmer may try to optimize the placement at run-time, but that requires extra development effort and an understanding of the underlying transactional protocol.

rVNF takes a different approach. It introduces *transparent locality enforcement* through object ownership. Each object has a single *owner*, a node with an exclusive right to write the object, and one or more *readers* that serve as backup replicas. In order to execute a transaction, a *coordinator* (node executing the transaction) has to become the owner of all objects that are involved in a transaction. This is a potentially expensive phase that requires coordination across several nodes. However, transactions in cellular VNFs involve objects that typically access a single context (e.g. a single flow or session). Thus when a coordinator becomes the owner for a particular context, it should continue receiving packets for that context, thus avoiding the need to acquire object

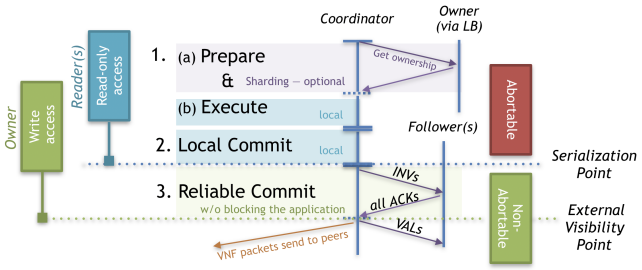


Figure 5: Locality-aware Reliable Transactions in rVNF.

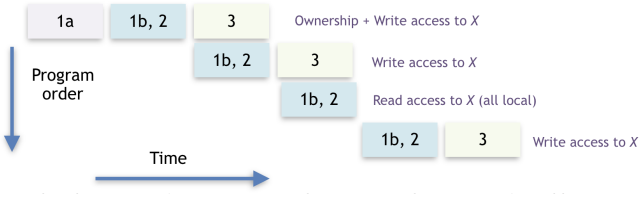


Figure 6: rVNF's pipelined execution of transactions for context X, on the same coordinator as enforced by LB.

ownership for subsequent transactions. This is precisely what LBs help to accomplish by routing all packets with the same context to the same node. Thus, in the steady state, a node is the owner of all objects accessed by requests routed to it.

Our *transparent locality enforcement* addresses the two main issues of previous works. Firstly, all objects accessed by a transaction are accessible locally the majority of the time, which avoids stalls during the execution phase. Secondly, only a single node (the owner) can execute a transaction on an object at a time, so a transaction cannot be aborted remotely.

rVNF further leverages these observations by breaking the commit phase of a transaction into two sub-phases: (i) local commit; and (ii) reliable commit. The local commit does not require any communication with the replicas. Because the coordinator has exclusive write access to the objects included in the transaction, the local phase does not need to perform any remote ordering or conflict resolution. Thus, a subsequent transaction at the same coordinator may access all locally committed objects without any wait.

The reliable commit propagates the transactional updates to the readers of all objects (called *followers*). Because the transaction is already serialized by virtue of exclusive ownership at the coordinator, only a single round-trip is needed for the coordinator to perform reliable commit by informing all followers. At this point, all changes are externally visible and any generated network output is released to the network. As depicted in Figure 6, a reliable commit can be overlapped with execution and local commit of subsequent transactions,

hence lowering latency and maximizing concurrency.

4.2 Transactions in Detail

We next explain the transaction phases in detail. Each AN has an instance of a transactional memory where it stores objects for which it is an owner or a reader. For each object, it stores its actual value (`t_value`) as well as per-object metadata: a version (`t_version`) and a state (`t_state`). There are three possible object states: *Valid*, *Invalid* and *Write*. All objects are initially *Valid*. We next review various phases of a transaction, as illustrated in Figure 5, and the relevant protocol messages. The protocol is inspired by Hermes [16]; however, Hermes provides only single value semantics and does not support transactions. In contrast, rVNF protocol manages multiple objects, their owners and readers.

Prepare and Execute: In the first phase of a transaction, an AN executes the application code. During the execution, the application code can access an object using the primitives from Figure 4. Some primitives (`tr_open_read` and `tr_get`) require a *Read-only access*. If the coordinator of a transaction is also a reader of the object, these primitives can be executed immediately in an optimistic, lock-free manner. Otherwise, the coordinator has to request to become a reader, and receive the value of the object, through the ownership protocol. Other primitives require a *Write access* to an object. rVNF first verifies that the coordinator is the owner of the object. If not, it stalls the execution until it obtains the ownership. Once it gets the ownership, it creates a local copy of the object and offers it to the application to modify.

Local commit: After the application calls `tr_commit`, rVNF begins local commit. For each object that was granted *Write access*, it first copies the locally modified copy into the transactional memory. Then, it increments the object's `t_version` and sets `t_state` to Write. At this point, the transaction is serialized (cannot be aborted any more). Finally, the coordinator sends an INV message to the followers. The INV message includes the new value (`t_value`) and the version (`t_version`) of each object updated in the transaction.

After this, the coordinator can start executing a new transaction without delay. Transactions become visible to the rest of the nodes at the end of reliable commit. However, if the coordinator owns all objects in a subsequent transaction, it is guaranteed that it has a consistent view of the state and can thus proceed before the end of the reliable commit phase.

Reliable commit: Upon receiving an INV message, a follower iterates over the updated objects in the message. For any object that it stores locally with a smaller `t_version`, it updates the `t_value`, `t_version` and transitions its `t_state` to *Invalid*. Subsequently, the follower buffers the INV packet and responds to the coordinator with an ACK message.

Once the coordinator receives ACKs from all followers, it finishes its own reliable commit by transitioning all of its modified objects to the Valid `t_state`. At this point the transaction is visible externally. The coordinator sends a VAL message to the followers and releases any outgoing application response messages that were generated during the transaction.

When a follower receives a VAL message, the follower transitions all involved objects of the corresponding buffered INV message into the *Valid* `t_state`, thus finishing its reliable commit phase. Lastly, the follower discards the buffered INV message.

Ownership protocol: The messaging structure of the ownership protocol is similar to the transactions. For each object, the object's owner and all LB nodes store a sharding state and a vector indicating the object's readers and the owner. They also store the ID of the last ownership request in a form {`r_Seq`, `node_ID`}, where `node_ID` is the unique ID of the node that made the last request and `r_Seq` is the sequence number of the request for that particular object, which increases after each request.

A transaction coordinator who wants to become a new owner of an object issues an ownership request to one of the LBs. This LB is called the ownership *driver*. The driver sends an INV message to the other LBs and the object's current owner, which together are referred to as *arbiters*, and increases `r_Seq`. The INV message includes the new pair {`r_Seq`, `node_ID`}.

The {`r_Seq`, `node_ID`} pair is used to resolve conflicts if two ownership requests are issued concurrently. Upon reception of an INV message, each arbiter checks its local value of {`r_Seq`, `node_ID`} for the requested object. If it is higher (in lexicographic ordering), the arbiter sends a NACK response directly to the coordinator indicating that a concurrent request has taken precedence. Otherwise, the arbiter places the object into the Invalid sharding state, updates its local request ID and responds with an ACK. The ACK from the owner also contains the latest value of the object and the metadata. At this point, the coordinator is the new owner and can proceed with its transaction. The coordinator also sends VAL messages to the arbiters to set the sharding states of the respective objects back to Valid.

4.3 Handling of Failures and Verification

In order to explain how rVNF resolves failures, consider a case when a coordinator fails in the middle of the commit, after sending the INV messages. Some followers might have received the INV messages, which leaves the relevant objects in the *Invalid* state. After a failure of the coordinator is detected (through a group membership, Section 3.2), all followers send out any buffered INV messages originating from the failed coordinator to all other followers. Since each INV message contains the full record of the transaction (namely, `t_version` and `t_value` of each object), the other followers can update their states from it. Moreover, it is safe to do so since the INV message has already been globally serialized by the original coordinator – the only owner of the context at that time.

The recovery of the ownership and sharding protocol is based on the same principles. Each INV message contains all the information necessary to recover the request, and they get replayed in the case of failure.

Due to a lack of space, we do not discuss or evaluate the full recovery in this paper. We have formally specified and model checked the transaction protocol[3] in TLA^+ for safety and the absence of deadlocks. We consider the following failures: crash-stop node failures, message reorderings and duplicates. Under these failure scenarios, we ensured that all sharers of an object in *Valid* t_state agree on their data (**transactions**) and arbiters in *Valid* sharding state agree on sharding vectors, which correctly indicate the single object's owner and readers (**ownership**).

5 SYSTEM

In this section we discuss the rVNF implementation and applications ported on top of it.

rVNF implementation: rVNF is implemented in C over DPDK, and it consists of two parts. One part is the rVNF module that runs as a separate process. This implements the main rVNF functionality, including load balancing, transactions, reliable communication, etc. The other part is the rVNF library that is linked to a VNF application over shared memory without imposing any constraints on the VNFs architecture (it can be a separate process, container, etc). This is illustrated in Figure 7.

In a typical deployment, and unless specified otherwise, rVNF uses 2 cores. DPDK and Routing KVS (if used, on an instance with LB) are pinned to one core. The rest of

[3]The model checked specification is available in http://bit.ly/rVNF-Zeus

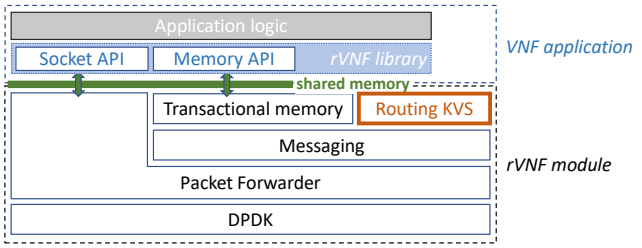


Figure 7: rVNF system architecture. If routing KVS is present, rVNF module acts as an LB. Otherwise, it connects to another node to access a LB.

the rVNF library runs on another core. Our evaluation shows that this mapping is enough to achieve the required performance of rVNF module (as specified in Section 2). We leave the remaining cores to the application.

Within a single rVNF instance, a packet forwarder interconnects all other internal modules and transfers packets and buffers from one to another. All active rVNF instances exchange information using an internal messaging protocol. A node without an LB (i.e. without a routing store) does not perform load balancing functionality. It connects to another node with an LB using messaging.

Socket API and Memory API are implemented as described in Section 3.3. We used modified snmalloc [20] as a memory manager for the transactional memory. rVNF supports multi-threading with several optimizations for performance, including the use of sharding across threads based on the readily available routing meta-data.

Currently, porting an application to rVNF requires manual code modification on every pointer access. We note that this can be automatized at a compiler level, similar to [38].

Applications: There is no open source 5G cellular core available at the moment. Instead, we modify a 4G core since 4G cellular signaling is similar to the 5G one. In particular, we evaluate MME and SPGW components of a 4G core, whose functionalities loosely map to AMF and SMF in 5G. We use OpenEPCv8 [27] 4G implementation. Because it has been reported slow [28], we optimize it by removing all existing replication mechanisms and accesses to the slow external data store (mySQL). We then port it to rVNF by virtualizing all state accesses using the rVNF API. Unlike [28], we focus on the cellular control plane which requires more complex transactions.

We also evaluate how well can connection-oriented protocols be virtualized using rVNF, as discussed in Section 2.1. One connection-oriented protocol we evaluate is SCTP. We modify a user-mode implementation of SCTP protocol [34] to save all its state in rVNF. We start a

transaction for every received or sent packet and commit it when the packet processing is finished. We also start a transaction whenever a timer triggers. Usrsctp has a main hash table that stores all open sockets. We store this table in the rVNF transaction key value store so different rVNF application instances can serve different SCTP sockets without having to contend for ownership. The memory layout for all per-socket state is maintained without modifications. Usrsctp uses BSD macros for basic data structures (e.g., lists, hash tables), and we modify the macros to support our memory management functions (Figure 4). Usrsctp is designed to use three worker threads (TX, RX and timer), and we keep the same design.

We also virtualize GTP-C protocol that connects MME and SGW [2]. It runs over UDP and implements its own simple retransmission protocol. It keeps timer for each packet at a transmitter until it is acknowledged, and retransmits it if a timer times out. It also keeps a timer for each packet at a receiver to prevent duplicate delivery of a retransmitted packet. We virtualize both sets of timers with rVNF.

6 EVALUATION

We run all our experiments on a dedicated cluster with 6 servers. Each server has a dual socket Intel Xeon Skylake 8168 with 24 cores per socket, running at 2.7GHz, 192 GB of DDR4 memory and Mellanox CX-3 card. They are connected to a Dell S6100-ON ToR switch over 40 Gbps links. We also run our experiments on a cluster of Standard_D8s_v3 VMs (with 8 CPU cores) in Azure, as an example of a more constrained setup. Azure VMs with fewer cores such as this one have less network bandwidth (see [7] for details). Also, DPDK on Azure does not allow jumbo frames and Azure does not support core pinning. The application suite is described in Section 5. Due to a lack of space, we do not describe or evaluate full failure recovery in this paper.

6.1 Cellular Core

We start by evaluating performance of a cellular core. We focus our evaluation on service requests that set up a user data plane session. It is the most common operation in a cellular network that comprises the bulk of the signalling workload [22]. Also, these are very similar to session management operations in 5G [3].

SGW: We first evaluate the service requests on a service gateway (SGW). We create a custom load generator that issues Create Session and Delete Session requests on an SGW in such way that SGW is always loaded. We measure and report the number of requests per second SGW can process.

We consider three cases. The first case is an SGW without any external storage. The second one is an SGW that stores its state to a single instance (not replicated) of an external Redis server. This is a straight-forward implementation of a UDSF using a popular open-source in-memory database. It is reported to be significantly faster than any other replicated open source data store [30]. The third case is an SGW that stores state in rVNF, replicated across two or three servers.

The results are depicted in Figure 8a. On our rack, an SGW without external storage can process around 3450 requests/sec/core (1750 create and 1750 delete requests). An SGW with rVNF achieves 20% less while replicating the state after each request to two other servers. An SGW with an external Redis store and without data replication can process around 590 requests/sec/core – 4.8× less than SGW and rVNF. On Azure, SGW with rVNF has approximately the same performance as SGW without replication, and it is 14.7 × more than the external Redis store. It is possible that performance with Redis can be improved by rearchitecting SGW to issue asynchronous requests to the data store. This is not required with rVNF, which collocates storage and processing and affords an intuitive programming model. In Figure 8a, we also show scaling of SGW with rVNF, with the total load balanced across 1, 2 and 3 replicated SGW instances.

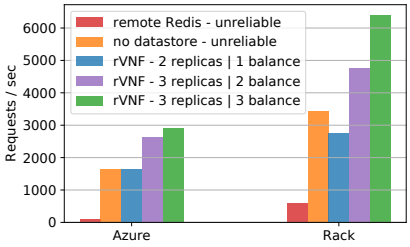
MME: We next evaluate the service requests on a mobility gateway (MME). We conduct similar experiments as for SGW, creating an artificial load from multiple simulated eNodeBs. We measure that a single MME without replication can process 5600 requests/sec/core. An MME on top of rVNF processes 35% less, but still 6.7 × more then the external Redis store.

6.2 Connection-Oriented Protocols

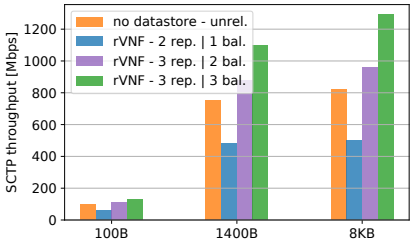
We next study how rVNF can be used to virtualize two protocols commonly used in cellular networks: SCTP and GTP-C.

SCTP: We evaluate SCTP performance using iperf3. We run usrsctp and iperf3 server as an application on top of rVNF. We run an unmodified iperf3 client on a Linux server and create a test traffic of SCTP 100 flows. In two sets of experiments, we use different iperf3 packet sizes, 100B and 1440B. All packets are received by one rVNF LB and balanced across a varying number of application instances. All SCTP state is replicated across all active rVNF instances.

The achieved aggregate SCTP throughput is shown



(a) *Performance of service requests (per core) for different SGW configurations.*



(b) *SCTP performance when varying the traffic size [Rack].*

Figure 8: Throughput of reliable rVNF compared with unreliable local and remote data stores. rep: replicas; bal: balancing nodes.

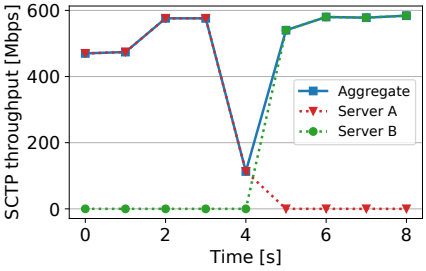


Figure 9: SCTP failover performance. The failure of the primary instance happens at 3s. [Rack]

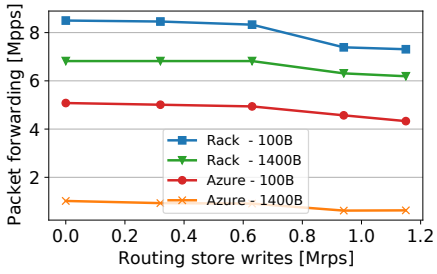


Figure 10: LB micro benchmark: Aggregate packet forwarding rate over 3 LBs as a function of a number of routing KVS writes.

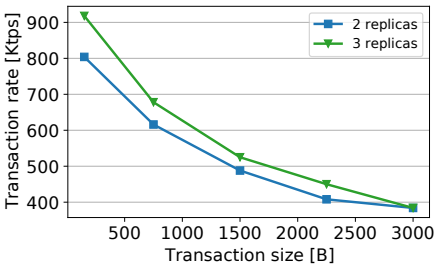


Figure 11: Transaction micro benchmark: Maximum rate for two- and three-way replication and different transaction sizes. [Rack]

in Figure 8b. To the best of our knowledge, no other data store is able to virtualize SCTP. On our rack, a replicated SCTP attains 60% of the throughput achieved without replication, both with big and small packets. On Azure, a replicated version achieves 80% and 40% of the throughput of the non-replicated one, with big and small packets respectively. We postulate that this is because the replication latency is higher in Azure than in our local rack. In both cases the performance does not change significantly with more replicas, indicating that replication throughput is not a bottleneck.

Next, we evaluate the ability of the virtualized SCTP to sustain faults. We study the following scenario. A client using iperf3 establishes a multi-homed SCTP connection to two rVNF servers (server A and B). Server A gets chosen by the protocol as a primary receiver and all data traffic goes to it. After each data packet reception, server A replicates all of its state to server B. Initially, we observe SCTP throughput of 577 Mbps (see Figure 9). We induce a network fault between the client and server A at 3 seconds into the experiment. After an SCTP timeout (set to 100ms), the client switches the traffic to the IP address of server B. When the fault happens, performance initially drops as SCTP waits for the timeout before changing the destination. Subsequently, throughput is quickly restored through server B. Because the entire state of the flow is replicated, the same connection is resumed and the client does not observe any issues.

GTP-C: We study the performance of SGW on top of rVNF when GTP-C state is virtualized. We run the same experiments described in Section 6.1, this time virtualizing GTP-C state. We observe no drop in performance on our rack compared to the experiment without virtualizing GTP-C state. The performance on the Azure setup drops by 10%.

6.3 Micro Benchmarks

We next present a series of micro benchmarks to further understand performance of various components of rVNF.

Load balancer and data forwarding performance: In rVNF, each incoming packet hitting the LB has to query a routing table before being forwarded to its destination. If the routing table is being concurrently modified (e.g., due to new flows being inserted or due to a scale-in), reads to the routing table may stall while the respective entries are being replicated. Thus, forwarding performance can be affected if the routing table is faced with a large number of writes. To understand forwarding performance under such an adversarial scenario, we set up the following benchmark. One server node acts as a generator of UDP data traffic sending

it to three instances of rVNF. Another node generates a stream of write requests to the routing store, writing one randomly-selected key out of 1M keys. We vary the number of routing requests from 0 to 1.2M requests/sec, and we measure how many packets the load balancer can forward in parallel with routing store updates.

The result is shown in Figure 10. We see that on our rack, each LB instance can forward up to 2.8Mpps of small packets and 26 Gbps of large packets. The forwarding throughput drops by at most 10-15% when the additional write load to the routing store is introduced. The LB instances can respond to up to 1.15M write requests/sec while forwarding traffic. The same experiments on Azure clusters show about 30% lower packet throughput in number of packets per second.

Transactions: We evaluate a synthetic workload to microbenchmark rVNF's transactional performance. We use 2 and 3 rVNF instances to replicate and execute transactions in parallel. Each transaction writes to a number of objects, where each object is 150B in size (comparable to the size of a typical small state). We vary the transaction size by varying the number of objects that are accessed. Each transaction is replicated on all other nodes partaking in the test. Test application runs on one thread and transaction replication (as a part of rVNF module) on another thread.

We present the aggregate rate of all transactions in the system in Figure 11. The aggregate transaction throughput is almost the same for 2 and 3 nodes. On our rack, rVNF can serve a little less than a million of small (150B) transactions per second and 0.4 Mtps for large transactions (3KB) across 2 and 3 nodes.

Benefits of dynamic routing: One of the features of rVNF is dynamic routing (**MULTI**). In case of session management, this allows SMF to proactively add a route to the session being created, as explained in Section 2.2. We now quantify benefits of dynamic routing. We repeat the experiments from Section 6.1 but with dynamic routing disabled. We observe that in the absence of dynamic routing, system throughput, in terms of requests/sec, drops by 30% while the total number of ownership requests increases by 37%. Thus, we conclude that dynamic routing is essential for high performance as it allows to exploit locality across requests.

7 RELATED WORK

Reliable VNFs. In Section 2, we have introduced a major body of work in making VNFs reliable or scalable. We have systematically compared rVNF with them. In addition, [37] distributes the VNF state as KV store in processing nodes, similar to what rVNF does. REINFORCE provides an efficient replay-based failure recovery method for chained NFs [19]. [18] proposes an efficient and proven scheme to automatically find the state that needs to be shared. Although partially achieving some requirements for cellular VNF, they fail to address all goals as rVNF does.

Virtualizing cellular core. [29] consolidates state from control and data plans and reduces the performance overhead for EPC. MMLite from [23] further borrows the ideas from [14] and completely decouples processing and state. The aforementioned solutions provide good performance and scalability for EPC, while rVNF achieves these properties in addition to satisfying the critical reliability requirement of cellular core.

Stateless Connection. [4] aligns with rVNF's efforts to make socket reliable by storing TCP state in persistent datastore, while Yoda [11] is a layer-7 load balancing that deploys stateless TCP in load balancers for high availability. However, cellular VNFs use SCTP which poses new challenges. Techniques such as S1Flex [41] partially tackle reliable SCTP but require extensive configuration, hence cannot be applied in dynamically changing networks. rVNF, on the other hand, delivers a fully reliable socket connection for cellular VNFs.

Distributed Transactions. There are many distributed key-value store systems [5, 16], but they do not support transactions. Several systems try to optimize DAC [9, 13, 15], but fundamentally keep 3 RTTs on the fast path after each transaction. Hermes [16] allows for local reads and fast reliable updates to individual keys from all replicas but it does not support multi-key reliable transactions.

8 CONCLUSION

In this paper we present rVNF, a replicated in-memory transactional data store for cellular VNFs. rVNF leverages unique access patterns of cellular and networking VNFs to improve data store performance and offer transactional guarantees. It introduces a novel concept of transparent locally-enforcing transactions, in which rVNF transactional protocol seamlessly and atomically moves object ownership across nodes. This allows for local processing and improved performance, and more flexible API. rVNF replication adds 30%-40% overhead to unreplicated system. rVNF outperforms a commonly used in-memory data store Redis by several times. We believe that its design principles can further improve the state of art in the service-based cellular architectures.

REFERENCES

[1] 3GPP. 2019. 3GPP TS # 23.501 System architecture for the 5G System (5GS). https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=3144. (2019). (V16.3.0 Updated on 12/22/2019).

[2] 3GPP. 2019. 3GPP TS # 29.274 Tunnelling Protocol for Control plane (GTPv2-C) (release 15). https://portal.3gpp.org/desktopmodules/Specifications/SpecificationDetails.aspx?specificationId=1692. (2019).

[3] 3GPP. 2019. 3GPP TS # 29.502 5G Session Management Services (5GS). https://www.etsi.org/deliver/etsi_ts/129500_129599/129502/15.00.00_60/ts_129502v150000p.pdf. (2019).

[4] Marcelo Abranches and Eric Keller. 2019. Stateless TCP. In *Proceedings of the 15th International Conference on emerging Networking EXperiments and Technologies*. 70–71.

[5] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. 2019. Fast Key-Value Stores: An Idea Whose Time Has Come and Gone. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*. Association for Computing Machinery, New York, NY, USA, 113–119. https://doi.org/10.1145/3317550.3321434

[6] Peter A. Alsberg and John D. Day. 1976. A Principle for Resilient Sharing of Distributed Resources. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE '76)*. IEEE Computer Society Press, Washington, DC, USA, 562–570.

[7] Azure. [n. d.]. General purpose virtual machine sizes. https://docs.microsoft.com/en-us/azure/virtual-machines/linux/sizes-general#dsv3-series-1. ([n. d.]). (Accessed on 14/01/2020).

[8] Arijit Banerjee, Rajesh Mahindra, Karthik Sundaresan, Sneha Kasera, Kobus Van der Merwe, and Sampath Rangarajan. 2015. Scaling the LTE Control-Plane for Future Mobile Access. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT '15)*. Association for Computing Machinery, New York, NY, USA, Article Article 19, 13 pages. https://doi.org/10.1145/2716281.2836104

[9] Aleksandar Dragojevič, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*. USENIX Association, USA, 401–414.

[10] FierceWireless. 2019. Industry Voices-Containers in 5G and edge still under construction. https://www.fiercewireless.com/5g/containers-5g-and-edge-still-under-construction. (2019).

[11] Rohan Gandhi, Y Charlie Hu, and Ming Zhang. 2016. Yoda: A highly available layer-7 load balancer. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.

[12] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2014. OpenNF: Enabling Innovation in Network Function Control. In *Proceedings of the 2014 ACM Conference on SIGCOMM (SIGCOMM '14)*. Association for Computing Machinery, New York, NY, USA, 163–174. https://doi.org/10.1145/2619239.2626313

[13] Le Long Hoang, Enrique Fynn, Mojtaba Eslahi-Kelorazi, Robert Soulé, and Fernando Pedone. 2019. DynaStar: Optimized Dynamic Partitioning for Scalable State Machine Replication. In *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019*. 1453–1465. https://doi.org/10.1109/ICDCS.2019.00145

[14] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. 2017. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, USA, 97–112.

[15] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, USA, 185–201.

[16] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. 2020. Hermes: A Fast, Fault-Tolerant and Linearizable Replication Protocol. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 201–217. https://doi.org/10.1145/3373376.3378496

[17] Junaid Khalid and Aditya Akella. 2019. Correctness and Performance for Stateful Chained Network Functions. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI'19)*. USENIX Association, USA, 501–515.

[18] Junaid Khalid, Aaron Gember-Jacobson, Roney Michael, Anubhavnidhi Abhashkumar, and Aditya Akella. 2016. Paving the Way for {NFV}: Simplifying Middlebox Modifications Using StateAlyzr. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 239–253.

[19] Sameer G Kulkarni, Guyue Liu, KK Ramakrishnan, Mayutan Arumaithurai, Timothy Wood, and Xiaoming Fu. 2018. Reinforce: Achieving efficient failure resiliency for network function virtualization based services. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. 41–53.

[20] Paul Liétar, Theodore Butler, Sylvan Clebsch, Sophia Drossopoulou, Juliana Franco, Matthew J. Parkinson, Alex Shamis, Christoph M. Wintersteiger, and David Chisnall. 2019. Snmalloc: A Message Passing Allocator. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management (ISMM 2019)*. Association for Computing Machinery, New York, NY, USA, 122–135. https://doi.org/10.1145/3315573.3329980

[21] LightReading. 2019. From NFV to Cloud Native - 4 Key Themes. https://www.lightreading.com/nfv/from-nfv-to-cloud-native---4-key-themes-/a/d-id/755162. (2019).

[22] A. Mohammadkhan, K. K. Ramakrishnan, A. S. Rajan, and C. Maciocco. 2016. Considerations for re-designing the cellular infrastructure exploiting software-based networks. In *2016 IEEE 24th International Conference on Network Protocols (ICNP)*. 1–6. https://doi.org/10.1109/ICNP.2016.7784474

[23] Vasudevan Nagendra, Arani Bhattacharya, Anshul Gandhi, and Samir R Das. 2019. MMLite: A Scalable and Resource Efficient Control Plane for Next Generation Cellular Packet Core. In *Proceedings of the 2019 ACM Symposium on SDN Research*. 69–83.

[24] Metaswitch Networks. 2019. Project Clearwater. https://github.com/Metaswitch/clearwater-website-archive. (2019).

(Accessed on 14/01/2020).

[25] Binh Nguyen, Tian Zhang, Bozidar Radunovic, Ryan Stutsman, Thomas Karagiannis, Jakub Kocur, and Jacobus Van der Merwe. 2018. ECHO: A Reliable Distributed Cellular Core Network for Hyper-scale Public Clouds. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (MobiCom '18)*. ACM, New York, NY, USA, 163–178. https://doi.org/10.1145/3241539.3241564

[26] Binh Nguyen, Tian Zhang, Bozidar Radunovic, Ryan Stutsman, Thomas Karagiannis, Jakub Kocur, and Jacobus Van der Merwe. 2018. ECHO: A Reliable Distributed Cellular Core Network for Hyper-scale Public Clouds. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (MobiCom '18)*. ACM, New York, NY, USA, 163–178. https://doi.org/10.1145/3241539.3241564

[27] PhantomNet. [n. d.]. OpenEPC Tutorial. https://wiki.emulab.net/wiki/phantomnet/oepc-protected/openepc-tutorial. ([n. d.]). (Accessed on 14/01/2020).

[28] Zafar Ayyub Qazi, Melvin Walls, Aurojit Panda, Vyas Sekar, Sylvia Ratnasamy, and Scott Shenker. 2017. A High Performance Packet Core for Next Generation Cellular Networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 348–361. https://doi.org/10.1145/3098822.3098848

[29] Zafar Ayyub Qazi, Melvin Walls, Aurojit Panda, Vyas Sekar, Sylvia Ratnasamy, and Scott Shenker. 2017. A high performance packet core for next generation cellular networks. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 348–361.

[30] Tilmann Rabl, Sergio Gómez-Villamor, Mohammad Sadoghi, Victor Muntés-Mulero, Hans-Arno Jacobsen, and Serge Mankovskii. 2012. Solving Big Data Challenges for Enterprise Application Performance Management. *Proc. VLDB Endow.* 5, 12 (Aug. 2012), 1724–1735. https://doi.org/10.14778/2367502.2367512

[31] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. 2013. Pico replication: a high availability framework for middleboxes.. In *SoCC*, Guy M. Lohman (Ed.). ACM, 1:1–1:15. http://dblp.uni-trier.de/db/conf/cloud/socc2013.html#RajagopalanWJ13

[32] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. 2013. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (nsdi'13)*. USENIX Association, USA, 227–240.

[33] EANTC Independent Test Report. 2018. Metaswitch's Clearwater IMS Core: Performance, Scalability, Reliability and Functionality. http://www.eantc.de/fileadmin/eantc/downloads/News/2018/EANTC-Clearwater-IMS_Marketing-Report_v6.pdf. (2018). (Accessed on 15/01/2020).

[34] I. Rüngeler and M. Tüxen. 2015. Socket API for the SCTP User-land Implementation (usrsctp). https://github.com/sctplab/usrsctp. (2015). (Accessed on 14/01/2020).

[35] sdxCentral. 2020. AT&T Misses Network Virtualization Goal. https://www.sdxcentral.com/articles/news/att-misses-network-virtualization-goal/2020/01/. (2020).

[36] Ravi Sethi. 1982. Useless Actions Make a Difference: Strict Serializability of Database Updates. *J. ACM* 29, 2 (April 1982), 394–403. https://doi.org/10.1145/322307.322314

[37] Xiaozhe Shao, Lixin Gao, and Hao Zhang. 2017. Cogs: Enabling distributed network functions with global states. In *2017 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 1–9.

[38] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and et al. 2015. Rollback-Recovery for Middleboxes. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15)*. Association for Computing Machinery, New York, NY, USA, 227–240. https://doi.org/10.1145/2785956.2787501

[39] Dale Skeen. 1981. Nonblocking Commit Protocols. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data (SIGMOD '81)*. Association for Computing Machinery, New York, NY, USA, 133–142. https://doi.org/10.1145/582318.582339

[40] Source. 2018. Private communication. (2018).

[41] K. Suzuki, K. Kunitomo, T. Morita, and T. Uchiyama. 2011. Technology Supporting Core Network (EPC) Accommodating LTE. *NTT DOCOMO Technical Journal* 13, 1 (June 2011).

[42] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838

[43] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast In-Memory Transaction Processing Using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 87–104. https://doi.org/10.1145/2815400.2815419

[44] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. 2018. Elastic Scaling of Stateful Network Functions. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation (NSDI'18)*. USENIX Association, USA, 299–312.