libVNF: A Framework for Building Scalable High Performance Virtual Network Functions

Priyanka Naik, Mythili Vutukuru
Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, India
ppnaik,mythili@cse.iitb.ac.in

ABSTRACT

One of the benefits of Network Function Virtualization (NFV) is the lower cost of building software appliances as compared to custom hardware. The development of software Virtual Network Functions (VNFs) can greatly benefit from a framework to enable code reuse, but such a framework does not exist today. This paper describes libVNF, an open-source library to develop scalable and high performance VNFs. The libVNF API provides high-level abstractions that lets a VNF developer leverage high performance packet I/O mechanisms without knowing the low level details of these optimizations. Our API also provides functions to manage shared state and frees the developer from the task of distributed state management in a clustered implementation. We rewrite two existing VNFs using our API and demonstrate its expressiveness and utility. We believe that the widespread use of a library like ours to build VNFs would significantly ease the burden of VNF development and reduce the cost to NFV adoption.

CCS CONCEPTS

•**Networks** → **Programming interfaces;** *Middle boxes / network appliances;*

KEYWORDS

Network Function Virtualization, mTCP, DPDK, netmap

ACM Reference format:
Priyanka Naik, Mythili Vutukuru.2017. libVNF: A Framework for Building Scalable High Performance Virtual Network Functions. In *Proceedings of APSys '17, Mumbai, India, September 2, 2017,* 8 pages.
DOI: 10.1145/3124680.3124728

1 INTRODUCTION

Network Function Virtualization (NFV) [9] is a recent trend in the networking and communications industry. NFV advocates moving network functions that were traditionally implemented as custom hardware appliances to software appliances running on commodity hardware, possibly on virtual machines (VMs) or containers in a cloud. The network functions being considered for virtualization include middleboxes in enterprise networks, e.g., firewalls and load balancers, and several appliances within telecommunication operator networks, e.g., the LTE EPC (Long Term Evolution Evolved Packet Core) [5]. There are several purported benefits to virtualizing network functions. Software appliances can be scaled horizontally by adding more replicas, making it easy to expand networks for future demands. Building software appliances is also expected to cost lesser time and effort than developing custom hardware. While there have been concerns about whether a software appliance can meet the performance of hardware, rapid advances in commodity hardware, coupled with frameworks like Data Plane Development Kit (DPDK) [3] that promise good packet processing performance in software, seem to alleviate these concerns.

For NFV to really take off, one needs an ample supply of low cost and high performance virtual network functions (VNFs). We note that several parts of the code of a VNF, like the logic of scaling a network function horizontally or the logic of communicating data via high performance software stacks, are common to every network function. Therefore, VNF development could consume lesser time and cost if a library were to be available to provide this common code. While some frameworks to build VNFs do exist in the research literature [8, 10, 14, 19, 21, 24], we argue in §2 that no one framework is generic enough to enable easy development of the many different kinds of VNFs in use today. Without a mechanism for code reuse, and it is not clear that building VNFs is going to be cost-effective as compared to building hardware appliances, bringing into question one of the main benefits of adopting NFV itself.

This paper introduces **libVNF** (§3), a reusable library for building high performance and scalable network functions. The most important contribution of our work is our libVNF API, that cleanly separates the application-specific processing of VNFs from the software stack that is reusable across

VNFs, and is generic and expressive enough to build a wide variety of VNFs across domains as varied as enterprise networks and telecommunication networks. We implement this API in an open-source library that is under active development. Our library is implemented atop the mTCP [13] multi-core scalable network stack, which in turn uses high performance packet I/O mechanisms like netmap [23] or the DPDK [3] kernel bypass mechanism to efficiently send and receive packets. Our API exposes a high level abstraction over these lower layer optimizations to simplify VNF development. Our API also provides a way for applications to identify shared state, and the library takes care of synchronizing this shared state across the distributed system using a shared data store.

The library and API provide a wide variety of design choices (e.g., with respect to shared state management in the distributed implementation, or the depth of the network stack processing in the VNF), giving developers the flexibility to customize the VNF design to suit their needs. The VNF application code written using libVNF will only contain the application-specific processing logic, and calls to the libVNF API function with suitable arguments for customization, saving developer effort. We attempt to build two VNFs from very different domains using our API (§4): the VNFs that make up the LTE packet core in telecom networks, and a high-performance load balancer used in enterprise networks. We show that refactoring existing code using our API can result in around 50% reduction in the lines of code of the VNFs, indicating significantly reduced development cost and effort.

An open-source library such as ours serves to establish a clear separation between the roles of a VNF domain expert and the systems expert. While the former would only focus on the application-specific packet processing logic of the VNF and easily build standards-compliant network functions, the latter would only focus on building horizontally scalable high-performance packet processing stacks without worrying about the network functions they would be used for. We hope that efforts such as ours will enable the VNF development community to come together and build a high performance software stack that is reusable across VNFs, reducing the time and cost of VNF development.

2 BACKGROUND AND GOALS

We begin by describing the assumptions and goals of our work (§2.1), and show how none of the existing frameworks for VNF development satisfy all our goals (§2.2).

2.1 Assumptions and Goals

A typical NFV deployment consists of a chain of VNFs connected in a service chain. At this point, our library does not address optimizations that are possible by considering all the VNFs in a chain holistically, e.g., optimizing packet transfers between VNFs in a chain. The goal of our work is to ease the development of a single VNF component, using high performance packet I/O mechanisms and a clustered architecture of multiple replicas for horizontal scalability. Our library manages distributed shared state across the multiple replicas of a VNF, but does not concern itself with questions like the number of replicas to spawn, when to scale-up or scale-down, or where to place the replicas in the underlying physical infrastructure. Such questions have already been investigated as part of management and orchestration frameworks for NFV like E2 [18]. While the scope of our library can expand to include such optimizations in the future, we do not attempt to address them in our current work.

To enable a horizontally scalable implementation of a VNF component, we assume that the processing and state of the application are separable. Further, we assume that all application state can be stored as key-value pairs, so that an in-memory key-value store (e.g., [16, 17, 22]) can serve as the shared data store. Our API provides mechanisms to get/put state in the shared data store, at the frequency desired by the VNF developer (e.g., once every packet, or once per "session"). Upon a failure, our library does not attempt to recover any local state beyond what is stored in the data store. We assume that, upon a failure, the application handles any errors resulting from local state that has not been synchronized with the data store, e.g., by retrying the request. Finally, our library runs over the mTCP [13] multicore scalable network stack, which scales packet processing across multiple cores by partitioning incoming traffic to cores. Therefore, we assume that the processing of the VNF is parallelizable (and not pipelined) across multiple cores.

The requirements below are desirable in any VNF development framework, and serve as our design goals.

R1: High performance packet I/O. A VNF framework must leverage recent high performance I/O mechanisms like DPDK [3] and netmap [23] that significantly improve packet I/O performance over the kernel stack. However, the framework must abstract out the low-level details of using these mechanisms via a high-level API that is stable across future developments in high-performance packet I/O.

R2: High level abstractions for application development. While some VNFs operate at the network layer, and only modify headers of packets in transit (e.g., a Layer 3 load balancer), some VNFs terminate transport layer connections and operate at the application layer. For example, the Mobility and Management Entity (MME) VNF in the LTE packet core receives requests from subscribers (e.g., to attach to the network, to handover from one region to another) over a transport layer SCTP connection, and communicates with other VNFs in the packet core also at the application layer. Similarly, a HTTP L7 load balancer acts as a TCP endpoint in order to parse the HTTP request before load balancing. A general VNF development framework must enable the development of such a variety of VNFs that operate at different depths of the network stack. The API must support high-level transport layer abstractions like sockets, connections, or application-layer requests, and must leverage recent work on multicore scalable transport layers like mTCP [13] to provide a high performance network stack.

R3: Horizontal scalability. A VNF development framework must easily enable a clustered implementation of a component, with minimal effort from the VNF developer. It must provide implementations of standard techniques for horizontal scalability, like load balancing incoming traffic across multiple replicas for scalability and the synchronization of shared state across the replicas for consistency and fault tolerance. Further, the API must be expressive enough to let the developer trade-off performance for higher consistency and fault tolerance or vice versa, without having to implement the low level mechanisms of state management.

2.2 Related Work

There has been prior research on frameworks for building middleboxes and VNFs. Split/Merge [21] and OpenNF [10] enable development of stateful middleboxes, and manage the migration of per-flow TCP state and across replicas during scale-in and scale-out. Pico Replication [20] and FTMB [24] provide frameworks to build fault tolerant network-layer middleboxes, by replicating state at the granularity of TCP flows in the former and by logging and replaying input packets in the case of the latter. To avoid the overhead of state migration and recovery of these mechanisms, StatelessNF [14] assumes that the replicas of a component are stateless, and uses a remote data store to synchronize shared state during fail-over and scaling, much like in our work. StatelessNF also runs on top of DPDK for high performance. However, it does not provide a transport-level API required to develop application-layer VNFs.

Netbricks [19] provides a DPDK-based framework to build network functions, using a high-level API inspired by the Click [15] framework. Flick [8] is another DPDK-based framework to build application-layer NFs, that provides a custom programming language atop the mTCP network stack. Both Netbricks and Flick do not enable a clustered VNF implementation, and horizontal scalability should be managed by the application developer. Table 1 summarizes how prior work compares with libVNF with respect to our requirements (§2.1).

We now describe work that is complementary to ours. mOS [12] is an extension of the mTCP stack used in our work, and modifies the scalable mTCP network stack to enable reconstruction of transport layer state even in middleboxes that are not transport layer endpoints. Our framework can leverage such work to provide an even richer API to the library users. E2 [18] is a framework to manage and orchestrate network functions, and to decide when to scale-in or scale-out a VNF based on load. Our library can work along with such a framework in the future to orchestrate the multiple VNF replicas. OPNFV [7] is an industry effort to build an open-source NFV platform for deploying VNFs; our work is orthogonal to this effort and concerns itself with developing VNFs to run on such platforms.

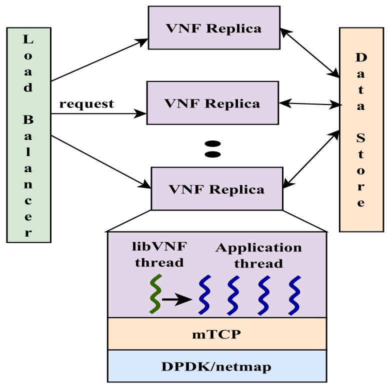

Figure 1: VNF architecture with libVNF.

3 DESIGN

We now describe the design of our framework, and the API exposed to VNF developers.

3.1 Architecture

An NFV system has several VNFs connected in a service chain or processing graph, and each VNF can be independently built using libVNF. Figure 1 shows the architecture of a libVNF-based VNF. The VNF code written by a developer using our API consists primarily of the VNF packet processing logic, and is compiled and linked against our library and run at one or more *replicas* in a clustered implementation. A *load balancer* takes care of splitting incoming traffic to the replicas, and a *data store* is used to manage shared state. The load balancer operates at the network layer, and transparently rewrites packet headers without exposing the internal clustered architecture of the VNF to upstream and downstream components in the service chain.

Within each replica, the VNF application code runs over the multicore scalable mTCP [13] network stack, which can communicate with the NIC using either the DPDK [3] or the netmap [23] I/O frameworks. The user code runs an initialization function of the library, which spawns one application thread per computing core, as is the norm with mTCP. The libVNF API provides functions for the application to open sockets (both as server and client) and communicate with the upstream and downstream VNFs in the VNF graph. The API

Framework	**Opt. packet I/O (R1)**	**Transport stack (R2)**	**Horizontal scaling (R3)**
Split/Merge, OpenNF	no	no	**yes**
StatelessNF	**yes**	no	**yes**
Netbricks	**yes**	no	no
Flick	**yes**	**yes**	no
libVNF	**yes**	**yes**	**yes**

Table 1: Comparison of VNF development frameworks against the requirements of §2.1.

also has an option to directly receive packets coming over an interface without going through the mTCP stack, in case the VNF wants to operate at the network layer alone. The application registers *callback* functions via our API, for the events corresponding to packet reception, new connection arrival (via `accept`), and successful packet transmission on sockets (or interfaces, when bypassing the TCP stack). These callback functions implement most of the processing logic of the VNF, and would constitute the bulk of the code written by the VNF developer. Upon occurrence of said events, the library invokes the callback functions registered on that connection, along with a pointer to the received packet. Our library sends and receive packets using `epoll` style system calls that are part of the mTCP event-driven I/O framework. The VNF application thread on every core calls a library function to enter the epoll wait loop after initializations.

The callback functions would also need to access application-specific state, and the API provides functions to manage this state. State is stored as key-value pairs in an in-memory data store; we use the popular Redis cluster as our data store. The API exports functions to create tables, and get/put key-value pairs in specific tables. A VNF must use these functions to access all state that it wishes to recover at any other replica, say, after a failure, scale-out, or scale-in. The library maintains a connection to the Redis server from every core on every replica, which is used to convey these get/put requests and receive replies. The VNF developer provides a callback function as part of the get/put API, and the library invokes this callback upon completion of the request, passing the result of the operation back to the VNF code.

Storing state in a key-value store can be an expensive operation, and libVNF provides an option to VNF designers to trade-off performance for fault tolerance by choosing the frequency of state synchronization on a per-packet basis. The library maintains a local cache of key-value pairs recently fetched from the data store in local memory, and the get/put API functions can choose to read from or write to this cache instead of the remote data store. The library takes care to ensure that key-value pairs modified locally are not evicted from the local cache until the dirty entries have been stored in the remote data store. This option of a variable granularity of state synchronization gives the VNF developer flexibility in tailoring the distributed VNF design to application needs. For example, for state that is critical, the VNF can store state in the remote data store always, incurring a performance penalty. Application state that is not critical can be modified locally in the cache, and check-pointed only periodically, say once per application session. Failures resulting from the loss of local state that has not yet been stored in the data store must be recovered by the VNF application, say, by making the downstream VNF retry requests.

The load balancer is implemented as a kernel module based on Netfilter hooks, and transparently rewrites packet headers to steer traffic to replicas using a provided list of VNF replicas. In the future, this list of replicas can come from the orchestration framework, based on the dynamic scaling decisions made by the orchestrator. The current design of our load balancer steers traffic based on the hash of the TCP 5-tuple. Now, if the application stores state at a granularity other than a TCP flow, then a given application state can be accessed concurrently from different cores or different replicas. For example, in a telecom network function that maintains per-subscriber state, the multiple flows of a subscriber could end up at different replicas, forcing the replicas to perform concurrent get/put operations on the same subscriber's state from different application layer threads on different replicas. Our library handles these concurrent operations, and serializes them for correct performance using the distributed locking and transaction features provided by key-value stores. As an alternative, our library can avoid this concurrent state access by steering traffic at the granularity at which application state is maintained, e.g., by ensuring that traffic of one subscriber is always routed to the same replica/core. We plan to enhance the traffic steering logic in our load balancer and the network stack in the future, to steer traffic based on any application-specific predicate by taking suitable inputs from the VNF developer.

3.2 High-level Abstractions

The event-based architecture of mTCP, with a separate TCP thread per core, is key to ensuring a scalable network stack. However, the multiple callbacks required to finish processing a given request at a VNF, and the need to maintain state across these callbacks, can make VNF development tricky in event-driven architectures. For example, consider a simple

S. No.	Function	Explanation
1	`connID = createServer(interface, localServerIP, localServerPort, protocol)`	Create listening socket
2	`connID = createClient(interface, remoteServerIP, remoteServerPort, protocol)`	Connect as client
3	`registerCallback(connID, event, callbackFnPtr)`	Register callbacks for packet events on a connection
4	`sendData(connID, packetToSend, size)`	Send data on a connection
5	`getState(table, key, localOrRemote, callbackFnPtr,connID)`	Get data from the data store
6	`addReqCtxt(connID, requestObj)`	Associate a request context with a connection
7	`ptr = getRefPtr(packet)`	Get a reference-counted pointer to a packet to add to request context
8	`startEventLoop()`	Start the main event loop

Table 2: The main libVNF API functions.

VNF chain of three VNFs A–B–C. VNF B receives a request from VNF A (say, over TCP). In order to process this request, B must open a connection to C, send data to C and obtain a reply, and possibly get some shared application state (stored by another replica of B) from the data store, before it can generate a reply to write back on the connection to A. Thus the state required to process A's request to B is spread over the data returned in callback functions invoked when data is received from C and from the data store.

In order to ease the job of the VNF developer, the libVNF framework provides an abstraction of a *request context* as part of its API. The definition of what constitutes a request is left to the VNF developer, and the developer can create request objects using the API based on application semantics. For example, a HTTP L7 load balancer can allocate a request object for the duration of a HTTP request on a TCP connection. The API lets the VNF developer define the structure of the request object, based on the state that the application must store as part of the request. The API provides functions to associate a request object with a socket or an interface, and this request object is passed to callback functions invoked after events on that socket. For VNFs that do not need to store state across callbacks (e.g., a Layer 3 NAT that simply rewrites packet headers using simple rules), the VNF developer can completely forgo using request objects.

To avoid the overhead of dynamic memory allocation, the library preallocates memory for caches of recent packets sent/received from mTCP, key-value pairs fetched from the data store, and request objects associated with connections. The packets are only copied once from mTCP, and a pointer to the packet is passed to the callback function to avoid further packet copies. Normally, the packet buffer would be reused at the end of the callback to store other packets. However, if a request requires access to the packet beyond

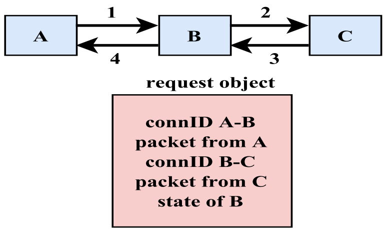

Figure 2: An illustration of request context.

the callback (say, until some state arrives from the data store), the API provides a function to obtain a reference-counted pointer to the packet to store in the request object. Packets with a non-zero reference count are not recycled in the packet cache, and the reference is decremented when a request object is freed by the VNF. Similarly, request objects can also store reference counted pointers to key-value pairs in the cache. Finally, request objects can also store other state like connection/socket identifiers (to reply to a request at a later point). Figure 2 shows an example request object created at VNF B to handle the request from VNF A, in the example VNF service chain A–B–C discussed earlier.

3.3 The libVNF API

Table 2 summarizes the main API functions exposed by our framework to VNF developers. API functions 1–2 setup server or client sockets for application-layer VNFs. If null IP address/port values are provided, the transport layer processing is bypassed and all packets arriving on an interface are considered part of the same "connection". API function 3 is used to register a callback function on the occurrence of a events defined by libVNF (e.g., a new connection accepted, or data received) on a connection, and the callback function is subsequently invoked with the connection identifier, the request object, and the received packet as arguments by the library. API function 4 is used to send data on a connection. API function 5 is used to fetch a key from a specified table, either from the local cache of key-value pairs or from the remote data store. Functions also exist to put/delete keys, and create/delete tables. Function 6 shows how a request object can be associated with a connection; functions also exist to allocate/free request objects. Function 7 shows how a request object can store a reference-counted pointer to a packet. Function 8 is used to start the main event loop on each application core, after all initializations. In the interest of space, we omit listing other straightforward functions.

We illustrate the benefits of using our API by providing a snippet of code from VNF B of the simple VNF chain shown in Figure 2. Code snippets 1 and 2 show a part of the code of VNF B written without using libVNF and with using libVNF respectively. When not using the library, the user has to explicitly store the different pieces of information belonging to a request (e.g., A's socket, C's socket, data received from A, and so on) in maps. On the other hand, the request object abstraction provided by our library (see Figure 2) simplifies this task, leading to a simpler code. This example illustrates the complexity of writing event-driven code in the mTCP framework, and how our API hides most of this complexity from the VNF developer.

```
//Request object passed as a parameter to handle state
//User does not have to maintain a map

//handle function for reply from C
void handle_c_reply(conn_id, request, packet){
   //store data
   request.data=getRefPtr(packet);
   sendData(request.serverID, id, request.data, 5);
}
//handle function for request from A
void handle_a_request(conn_id, request, packet){
   //connect to C
   int c_id = createClient(id, C_IP_ADDR, C_PORT, TCP);
   registerCallback(c_id, "read", handle_c_reply);
   addReqCtxt(c_id, request);
   //map C's socket to A's socket
   request.serverID = conn_id;
   //store data which would be pointed by both A and C's socket
   request.data = getRefPtr(packet);
   //send request to C
   sendData(c_id, id, data, len);
}
int main(int argc, char *argv[]){
    int serverID = createServer(B_IP,B_PORT,TCP);
    setRequest(serverID, sizeof(struct));
    registerCallback(serverID, "read",handle_a_request);
    startEventLoop();
}
```

Listing 2: VNF B written using libVNF

4 EXAMPLE VNFS

Our library development is a work in progress. We have completed the definition of the API, and have implemented the functionalities of the network stack, the load balancer, and the data store interface. The optimizations corresponding to maintaining request objects and packet caches are part of ongoing work. We are unable to present a performance evaluation of our library at this point. However, in this section, we show how two very different VNFs—the LTE packet core and the Galera load balancer [4] used in MySQL clusters—can be implemented using our API, demonstrating its expressiveness. We also show how using our library leads to a significant reduction in the lines of code (LoC) of the VNFs, indicating reduced effort for VNF development using our framework. We hope to demonstrate end-to-end development and execution of VNFs over our framework in the near future.

```
mctx_t mctx = mtcp_create_context();
int ep = mtcp_epoll_create(mctx, N);
mtcp_listen(mctx, listen_sockid, 4096);
while(1){
  nev = mtcp_epoll_wait(mctx, ep, events, N, -1);
  for(int i=0;i<nev;i++) {
    if (events[i].data.sockid == listen_sockid){
       //Accept connections from A
       newsockfd = mtcp_accept(mctx, listen_sockid, NULL, NULL);
       ev.events = MTCP_EPOLLIN;
       ev.data.sockid = newsockfd;
       mtcp_setsock_nonblock(mctx, newsockfd);
       mtcp_epoll_ctl(mctx, ep, MTCP_EPOLL_CTL_ADD, newsockfd, &ev);
       //store the socket of A in a map
       a_conn_map.insert(newsockfd);
    }
    else if (events[i].events & MTCP_EPOLLIN) {
       if(sockid in a_conn_map){
         //read request from A
         mtcp_read(mctx, sockid, data,LEN);
         //connect to C
         csockid = mtcp_socket(mctx);
         c_conn_map.insert(csockid);
         mtcp_connect(mctx, csockid, &daddr, size);
         //associate C's socket to A's socket to send reply back to A later
         conn_map[csockid] = sockid;
         //associate data with A's socket
         data_map[sockid]=data;
         //associate data with C's socket
         data_map[csockid]=data;
       }
       if(sockid in c_conn_map){
        //read reply from C
        mtcp_read(mctx, sockid, dataC,LEN);
        //retrieve A's socket from the map
        asocket = conn_map[sockid];
        //update the data in map for socket of A and C
        data_map[sockid]=dataC;
        data_map[asocket]=dataC;
        mtcp_write(mctx, asocket, dataC, len);
       }
    }
    else if (events[i].events & MTCP_EPOLLOUT){
      if(sockid in c_conn_map){
        //connection to C done, send request to C
        mtcp_write(mctx, sockid, data_map[sockid], len);
      }
    }
   }
}
int main(){
run();
}
```

Listing 1: VNF B written without libVNF

4.1 The LTE packet core

The 4G LTE mobile data network consist of a radio access network (comprising of mobile users and base stations) and a packet core called the Evolved Packet Core or LTE EPC. Increase in mobile traffic, the pressure to lower costs, and the desire to add new features easily are pushing telecom operators to consider virtualizing the components of the LTE EPC, and several industry players are actively developing virtualized EPC network functions [1, 2, 6]. As shown in Figure 3, the LTE EPC consists of the Mobility and Management Entity (MME) and the Home Subscriber Server (HSS) in the control plane, and the Serving Gateway (SGW) and Packet Data Network Gateway (PGW) in the data plane. The MME processes control plane requests to connect a subscriber to the network, release resources when the subscriber is idle, handover the subscriber during mobility, and so on. The MME consults with the HSS during such request processing, and sets up or modifies forwarding state of the subscriber in the dataplane SGW and PGW. Once forwarding state is setup by the MME, subscriber traffic is forwarded via the SGW and PGW to the external networks. The MME, SGW, and PGW are all application-layer network functions that communicate with each other, and with the upstream and downstream components over transport layer connections.

We use an existing open-source EPC codebase [11] as an example VNF. The implementation we started out consisted of distributed implementations of the MME, SGW, and PGW, built over the kernel stack (and not an optimized userspace stack). We re-factored these VNFs to use our API, and added support for high performance I/O in the process. In this exercise, we found that our API functions were sufficient to abstract out all the VNF code that was not application-specific. Table 3 shows the LoC in each of the EPC components in the original implementation, and in the implementation using our API. We can observe from the table that our API has resulted in reduction of LoC in the range of 48% to 51%, which amounts to a significant reduction in development effort.

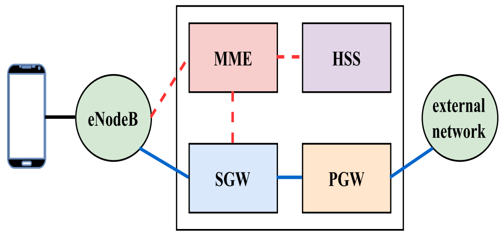

Figure 3: A simplified LTE EPC Architecture.

VNF	LoC w/o library	LoC with library	%reduction in LoC
MME	1956	1016	48.05%
SGW	1600	775	51.56%
PGW	1345	654	51.37%

Table 3: LoC reduction in LTE EPC using libVNF.

4.2 The Galera load balancer

The next example VNF we consider is the Galera load balancer [4], which is used as a load balancer in MySQL cluster deployments. The original code of the load balancer has a polling based multi-threaded implementation over the kernel stack written in C, but did not support a distributed implementation or a high-performance I/O stack. We started out with this code and re-factored it to use our API, separating out the application-specific logic (e.g., backend selection) from the code dealing with packet I/O. Our refactoring led to a 25% reduction in the LoC of the Galera implementation (from 2279 to 1698). Further, this new code written using our API can be run as a clustered implementation over a high-performance I/O stack without requiring any further developer effort.

5 CONCLUSION

This paper presented libVNF, a framework to develop scalable and high performance network functions. The main contribution of our work is the identification of an API that separates the application-specific logic of a VNF from the application-agnostic software stack, and is versatile enough to build a wide variety of VNFs. VNF developers using the libVNF API only need to write VNF-specific packet processing logic, and the library handles the tasks common to all VNFs, like efficient communication via high-performance network stacks and state management across distributed horizontally scaled replicas. Our API provides high level abstractions over lower-layer packet I/O optimizations, making it easy for VNF developers to leverage such optimizations. We have developed two very different VNFs using our API, and have demonstrated a significant reduction in LoC if our framework is used.

Our library is under active development, and we hope to make an open-source implementation available soon. As part of future work, we plan to build several different VNFs using our library, and demonstrate their end-to-end execution over our framework. We would like to evaluate the performance impact of state synchronization using a shared data store, for different granularities of synchronization. We would also like to quantify the performance benefits of several packet I/O optimizations and kernel bypass mechanisms, and provide suitable guidelines to application developers about the various design options available for VNF development. We believe that the availability of a library such as ours will greatly decrease the time, effort, and cost to building network functions, and accelerate the adoption of NFV in the networking community.

ACKNOWLEDGMENTS

This work was supported by a gift from Intel Corporation, and a seed grant fellowship from IIT Bombay.

REFERENCES

[1] Virtual Mobile Network. http://www.affirmednetworks.com/products-solutions/vepc/.
[2] Brocade virtual evolved packet core. http://www.brocade.com/en/products-services/mobile-networking/vepc.html.
[3] Intel Data Plane Development Kit. http://dpdk.org/.
[4] Galera load balancer. http://galeracluster.com/documentation-webpages/glb.html/.
[5] The evolved packet core. http://www.3gpp.org/technologies/keywords-acronyms/100-the-evolved-packet-core.
[6] vepc solutions. http://www.nec.com/en/global/solutions/tcs/vepc/index.html.
[7] Opnfv. https://www.opnfv.org/.
[8] A. Alim, R. G. Clegg, L. Mai, L. Rupprecht, E. Seckler, P. Costa, P. Pietzuch, A. L. Wolf, N. Sultana, J. Crowcroft, A. Madhavapeddy, A. W. Moore, R. Mortier, M. Koleni, L. Oviedo, M. Migliavacca, and D. McAuley. Flick: Developing and running application-specific network services. In *Proc. of USENIX ATC'16*.
[9] ETSI. Network Functions Virtualisation. https://portal.etsi.org/nfv/nfv_white_paper.pdf.
[10] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella. Opennf: Enabling innovation in network function control. In *Proc. of SIGCOMM'14*.
[11] IITB. Nfv_lte_epc_1.1. https://github.com/networkedsystemsIITB/NFV_LTE_EPC/tree/master/NFV_LTE_EPC-1.1, 2016.
[12] M. A. Jamshed, Y. Moon, D. Kim, D. Han, and K. Park. mos: A reusable networking stack for flow monitoring middleboxes. In *Proc. of NSDI'17*.
[13] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mtcp: A highly scalable user-level tcp stack for multicore systems. In *Proc. of NSDI'14*.
[14] M. Kablan, A. Alsudais, E. Keller, and F. Le. Stateless network functions: Breaking the tight coupling of state and processing. In *Proc. of NSDI'17*.
[15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000. ISSN 0734-2071. doi: 10.1145/354871.354874. URL http://doi.acm.org/10.1145/354871.354874.
[16] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. Mica: A holistic approach to fast in-memory key-value storage. In *Proc. of NSDI'14*.
[17] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, Aug. 2015. ISSN 0734-2071.
[18] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker. E2: A framework for nfv applications. In *Proc. of SOSP'15*.
[19] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. Netbricks: Taking the v out of nfv. In *Proc. of OSDI'16*.
[20] S. Rajagopalan, D. Williams, and H. Jamjoom. Pico replication: A high availability framework for middleboxes. In *Proc. of SoCC'13*, .
[21] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield. Split/merge: System support for elastic execution in virtual middleboxes. In *Proc. of NSDI'13*, .
[22] Redis. http://redis.io/.
[23] L. Rizzo. netmap: A novel framework for fast packet i/o. In *Proc. of USENIX ATC'12*.
[24] J. Sherry, P. X. Gao, S. Basu, A. Panda, A. Krishnamurthy, C. Maciocco, M. Manesh, J. a. Martins, S. Ratnasamy, L. Rizzo, and S. Shenker. Rollback-recovery for middleboxes. In *Proc. of SIGCOMM'15*.