libVNF: Building Virtual Network Functions Made Easy

Priyanka Naik, Akash Kanase, Trishal Patel, Mythili Vutukuru
Department of Computer Science and Engineering, Indian Institute of Technology, Bombay, India
{ppnaik,akanase16,trishal16,mythili}@cse.iitb.ac.in

ABSTRACT

Network Function Virtualization (NFV) aims to reduce costs and increase flexibility of networks by moving functionality traditionally implemented in custom hardware into software packet processing applications, or virtual network functions (VNFs), running on commodity servers in a cloud. This paper describes the design and implementation of libVNF, a library to build high performance, horizontally scalable VNFs. Unlike existing frameworks for VNF development, our library (i) can be used for the development of L2/L3 middleboxes as well as VNFs that are transport layer endpoints; (ii) seamlessly supports multiple network stacks in the backend; and (iii) enables distributed implementation of VNFs via functions for distributed state and replica management. We have implemented a variety of VNFs using our library to demonstrate the expressiveness of our API. Our evaluation shows that building VNFs using libVNF can reduce the number of lines of code in the VNF by up to 50%. Further, optimizations in our library ensure that the performance of VNFs built with our library scales well with increasing number of CPU cores and distributed replicas.

CCS CONCEPTS

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

KEYWORDS

Network Function Virtualization, kernel-bypass, mTCP, DPDK, netmap

ACM Reference Format:
Priyanka Naik, Akash Kanase, Trishal Patel, Mythili Vutukuru. 2018. libVNF: Building Virtual Network Functions Made Easy. In *Proceedings of SoCC '18: ACM Symposium on Cloud Computing , Carlsbad, CA, USA, October 11–13, 2018 (SoCC '18),* 13 pages.
https://doi.org/10.1145/3267809.3267831

1 INTRODUCTION

Network Function Virtualization (NFV) [25] envisions replacing networking elements traditionally built on dedicated hardware with software components running on commodity servers in a cloud. NFV brings many benefits to network operators, including the ability to elastically scale the number of software instances to match load (instead of provisioning hardware to handle peak load), and the flexibility to upgrade network functionality quickly (instead of waiting for long hardware development cycles). Recent techniques that enable the development of high-performance packet processing software (e.g., [3], [16]) have spurred the adoption of NFV. The network functions being considered for virtualization today range from simple layer 2/3 middleboxes that manipulate packet headers (e.g., firewall, L3 load balancer, NAT), to networking elements that terminate transport layer connections and perform higher layer processing.

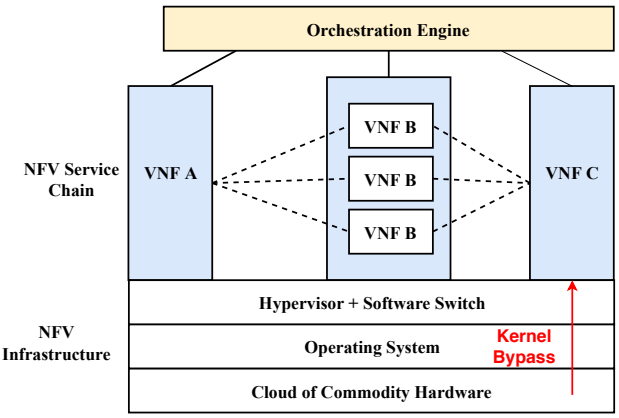

Figure 1: The NFV ecosystem.

Figure 1 shows a simplified view of an NFV deployment. Network traffic is forwarded through *service chains* of one or more *virtual network functions* (VNFs), hosted on the underlying NFV infrastructure consisting of a cloud of commodity servers. The VNFs in the chain can be monolithic (VNFs A and C in the figure), or horizontally scaled clustered implementations (VNF B). The VNFs can run directly on the bare metal servers, or inside virtual machines (VMs) or containers for easy provisioning and management. The VNFs can either run over the traditional network stack within the operating system, or can use kernel bypass techniques (e.g., DPDK [3], netmap [42]) to directly access packets from the hardware in userspace and then process them using optimized userspace network stacks (e.g., mTCP [30], TLDK [15]). Finally, an orchestration layer runs across the system and performs functions such as traffic steering, elastic scaling, and lifecycle management of the VNFs.

This paper focuses on the problem of building high-performance VNFs. For NFV to become a cost saving paradigm, a VNF developer should only have to write the core packet processing logic of the VNF. The other parts of the VNF code—including the logic of efficiently communicating with other VNFs of the service chain over a network stack, and managing distributed state across other replicas

SoCC '18, October 11–13, 2018, Carlsbad, CA, USA

ACM ISBN 978-1-4503-6011-1/18/10…$15.00
https://doi.org/10.1145/3267809.3267831

of the same VNF—should be readily available in reusable optimized VNF development libraries. However, we find that the reality of VNF development is far from it. For example, we examined the code of industry-grade VNFs that constitute the packet core of mobile telecommunication networks (LTE EPC, or Long Term Evolution Evolved Packet Core) [1]. We found that about 38% of the code of these VNFs pertained to reading and writing network data efficiently via DPDK, and had nothing to do with the core logic of the VNF.

The VNF development frameworks proposed in prior work cannot be used to easily build scalable implementations of complex VNFs such as the EPC components for the following reasons. First, most frameworks (e.g., [16, 17, 38]) focus on providing richer APIs for packet header manipulation in L2/L3 middleboxes, and do not have abstractions that support the development of VNFs with transport layer endpoints such as the EPC components. Second, frameworks that support the transport layer endpoint abstraction (e.g., [15, 30]) expose an event-driven socket-like API over a multicore-scalable userspace stack to VNF developers. However, it is well known that writing event-driven code is complicated by the fact that the processing of a single application-layer request is split across multiple callbacks, resulting in the VNF developer having to maintain significant state across callbacks [18]. Therefore, building VNFs with transport layer endpoints over such frameworks still requires significant developer effort.

Third, while some frameworks exist to build horizontally scalable middleboxes (e.g., [26, 31, 41, 47]), none of them comes with a transport layer stack that enables the development of VNFs with transport layer endpoints. Finally, existing frameworks force the VNF developer to choose a network stack apriori (e.g., the regular Linux kernel stack or a kernel bypass stack), and VNFs written on one stack are not easily portable to another. However, our experiments with different types of VNFs show that there is no one right choice when it comes to network stacks. For example, when processing a CPU-intensive workload in the VNFs that comprise the IP Multimedia Subsystem (IMS) in telecommunication networks, a kernel bypass mechanism that is non-trivial to setup and configure achieves a throughput gain of only 1.4× over the more readily available Linux kernel stack (see §4.2 for details). On the other hand, for the I/O intensive LTE EPC gateway VNFs, the kernel bypass stack improves throughput by 33× as compared to the Linux kernel stack, because the impact of reducing kernel I/O processing overheads is more promounced with an I/O intensive workload. These results indicate that VNF developers would benefit from the ability to easily switch and experiment with multiple network stacks, to pick the one that is best suited to the application needs. However, existing VNF development frameworks do not provide this flexibility, and switching network stacks requires significant changes to the VNF code.

This paper describes the design and implementation of libVNF, a C++ library that eases the development of high performance, horizontally scalable VNFs [6]. The libVNF API provides a set of high-level functions for efficient network communication, and can be used to build both L2/L3 middleboxes as well as VNFs that act as transport layer endpoints. The libVNF library implements these API functions over multiple backend network stacks; we currently support the Linux kernel stack and the mTCP userspace stack (over the netmap/DPDK kernel bypass mechanisms). The libVNF communication API is asynchronous and event-driven, in order to leverage event-driven multicore-scalable network stacks like mTCP. VNF developers embed packet processing logic within callback functions that are invoked by the library when events (e.g., packet arrivals) occur on connections. Unlike prior work, libVNF significantly eases the management of application state across callbacks by exposing the abstraction of an application-layer request to the VNF developer. With libVNF, VNF developers can store state across multiple connections and callbacks in a single *request object* that is efficiently managed by the library. libVNF can also be used to easily build distributed clustered implementations of a VNF; our API provides functions to store and retrieve state in a shared datastore, and functions to monitor the health of multiple replicas of a component.

Our implementation of the libVNF API employs several optimizations for high throughput and multicore scalability. Upon initialization, the library spawns one event-driven application thread per core, and the threads use cache-optimized per-core data structures for lockfree operation wherever possible. To avoid the overhead of dynamic memory management, the library preallocates memory using optimized slab allocators for frequently recycled objects, e.g., packet buffers and request objects.

We build several VNFs using our library to show that our API is expressive enough to cater to a wide variety of VNFs, from a simple L3 load balancer to complex VNFs that make up the EPC and IMS subsystems in the packet core of mobile telecom networks. We find that using our library to build VNFs saves up to 50% lines of code in the VNF, and the performance of VNFs built over libVNF is within 10% of optimized VNFs built without the library. Further, we show that the performance of VNFs built with our library scales well with increasing replicas of a VNF, and with increasing CPU cores within a single replica.

This paper builds upon and improves our position paper [35] that only proposed an API without an implementation. The experience of implementing the API over multiple network stacks has led us to refine our API further from this prior work. The rest of the paper is organized as follows. §2 provides an overview of our system. §3 describes the libVNF API and its implementation. §4 presents the evaluation of our system. §5 describes related work and §6 concludes the paper.

2 LIBVNF OVERVIEW

We designed libVNF with the following goals:

- The API should be useful for the development of both L2/L3 middleboxes as well as VNFs that terminate transport layer connections.
- The API should be agnostic to the choice of the network stack (kernel vs. kernel bypass) and the deployment platform of the VNF (baremetal vs. VM).
- The API should ease state management and replica coordination in a distributed VNF implementation.
- The performance of the library should scale well with increasing CPU cores of the VNF.

We now illustrate VNF development in our framework with a simple example of a service chain consisting of VNFs A, B, and C, shown in Figure 2. Let the communication between the VNFs

Figure 2: Example VNF service chain.

```
1 struct b_state{
2   char* a_req;
3   int serverID;
4 };
5 //callback for request from A
6 void handle_a_req(a_sockid,requestObj,a_req){
7   requestObj = allocReqObj(a_sockid,1);
8   //connect to C
9   int c_sockid = createClient(C_IP_ADDR,C_PORT,TCP);
10  //callback for read on C's socket
11  registerCallback(c_sockid,READ,handle_c_reply);
12  linkReqObj(c_sockid,requestObj);
13  b_state *x = static_cast<b_state*>(requestObj);
14  //store state in request object
15  x->serverID = a_sockid;
16  x->a_req = setPktDNE(a_req);
17  //send request to C
18  c_request = getPktBuf(c_sockid);
19  //... fill packet buffer ...
20  sendData(c_sockid,c_request,LEN);
21 }
22 //handle function for reply from C
23 void handle_c_reply(c_sockid,requestObj,c_reply){
24  //... read reply from C ...
25  //retrieve state from request object
26  b_state *x = static_cast<b_state*>(requestObj);
27  a_sockid = x->serverID;
28  //... compute reply using a_req and c_reply ...
29  //write reply back to A
30  a_reply = getPktBuf(a_sockid);
31  // ... fill packet buffer ...
32  sendData(a_sockid,a_reply,LEN);
33  unsetPktDNE(x->a_req);
34  freeReqObj(c_sockid);
35  freeReqObj(a_sockid);
36 }
37 int main(int argc, char *argv[]){
38  int serverID = createServer("",B_IP,B_PORT,TCP);
39  //callback function for read on server socket
40  registerCallback(serverID,READ,handle_a_req);
41  //initialize request object pool
42  int reqpool[1] = {sizeof(struct b_state)};
43  initReqPool(reqpool, 1);
44  startEventLoop();
45  return 0;
46 }
```

Listing 1: VNF B written with libVNF

proceed as follows: A generates requests to B over a TCP connection. To process A's request, B first opens a connection to C, sends a request, and waits to get a response back. After receiving C's reply, B proceeds to do some computation based on data received from A and C, and then sends a reply back to A.

```
1 run(){
2   //called in thread after accept
3   read(a_sockid,a_req,LEN);
4   connect(c_sockid,&daddr,size);
5   write(c_sockid,c_request,LEN);
6   //read reply from C
7   read(c_sockid,c_reply,LEN);
8   //compute a_reply using a_req and c_reply
9   //write reply back to A
10  write(a_sockid,a_reply,LEN);
11 }
```

Listing 2: VNF B written over blocking sockets

Listing 1 shows some snippets of the code of B built over libVNF. The `main` function of B first invokes the initialization API (§3.1, lines 38–44) to begin VNF execution. During initialization, the library creates one affinitized VNF thread per CPU core, and starts each thread in an event-driven loop to process packets. The library also configures the packet I/O path through the network stack, the details of which depend on the type of network stack (kernel vs. kernel bypass), the layer at which the VNF is operating (L2/L3 vs. transport), and the deployment platform (baremetal vs. VM).

After initialization, the VNF uses the functions in the libVNF communication API (§3.2) to communicate with other VNFs, e.g., by opening connections (line 9) and sending data (lines 18–20). In order to leverage existing work on multicore-scalable network stacks like mTCP that expose an event-driven API, our communication API is also asynchronous and event-driven. The VNF developer registers callback functions to be invoked by the library on events such as packet arrivals (lines 11, 40), and the main packet processing logic of the VNF is written within these callback functions (e.g., `handle_a_req` and `handle_c_reply`). We currently implement this API over the regular Linux stack and the mTCP userspace stack with the DPDK/netmap kernel bypass mechanisms. Our implementation uses per-core pools of packet buffers that are preallocated using optimized allocators, and accessed in a lockfree manner by the application threads. We also employ optimizations such as batched transfer of packets to/from the network card wherever possible.

Developing applications over non-blocking event-driven APIs such as ours is harder than when using blocking APIs, because application state that would have been on the process stack in the blocking case must now be manually marshalled across callbacks by the developer of the event-driven application [18]. For example, Listing 2 shows how the code of B would be written over blocking sockets, while Listing 3 shows B written over an event-driven API with non-blocking sockets (but without using libVNF). We can see from the code snippets that state required in B to reply to A's request (e.g., the packet received from A) is readily available as local variables on the stack when using blocking sockets. However, in the case of event-driven APIs, when B replies to A's request within the callback invoked upon receiving C's reply, the packet received from A is no longer on the stack. Therefore, when not using libVNF, B must explicitly allocate memory and maintain data-structures to track this state (lines 22–30 of Listing 3), and this problem only gets worse as the VNFs get more complex.

```
1 //map socket id to state
2 map<int,void*>state_store;
3 //set to maintain socket status
4 set<int>c_conn_map,a_conn_map;
5 struct b_state{
6     char* a_req;
7     int serverID;
8 };
9 void run(){
10 if (events[i].data.sockid == listen_sockid){
11  //store the socket of A in a set
12  a_conn_map.insert(newsockfd);
13 }
14 else if (events[i].events & EPOLLIN) {
15  if(a_sockid in a_conn_map){
16     //read request from A
17     read(a_sockid,a_req,LEN);
18     //connect to C
19     connect(c_sockid,&daddr,size);
20     //store the socket of C in a set
21     c_conn_map.insert(c_sockid);
22     void* requestObj = malloc(sizeof(b_state));
23     b_state *x = static_cast<b_state*>(requestObj);
24     //store packet from A required
25      x->a_req = malloc(LEN);
26     memcpy(x->a_req,a_req,LEN);
27     //associate C s socket to A s socket
28     x->serverID = a_sockid;
29     //point C socket to request state
30     state_store[c_sockid]=requestObj;
31     //write request to C
32     write(c_sockid,c_request,LEN);
33   }
34   if(c_sockid in c_conn_map){
35     //read reply from C
36     read(c_sockid,c_reply,LEN);
37     //retrieve A s socket from the map
38     b_state *x = static_cast<b_state*>
39               (state_store[csockid]);
40     a_sockid = x->serverID;
41     //compute a_reply using x->a_req and c_reply
42     //write reply back to A
43     write(a_sockid,a_reply,LEN);
44     free(x->a_req);
45     free(state_store[c_sockid]);
46     state_store.erase(c_sockid);
47     c_conn_map.erase(c_sockid);
48     a_conn_map.erase(a_sockid);
49   }
50 }}
```

Listing 3: VNF B written with an event-driven API

To ease this process of managing state across callbacks, libVNF provides the abstraction of a request object. A request object can be used to store application state of a single request across callbacks and connections, e.g., VNF `B` can embed the packet received from `A` during the callback `handle_a_req` within a request object, and retrieve it to generate a reply to `A` in the callback `handle_c_reply`.

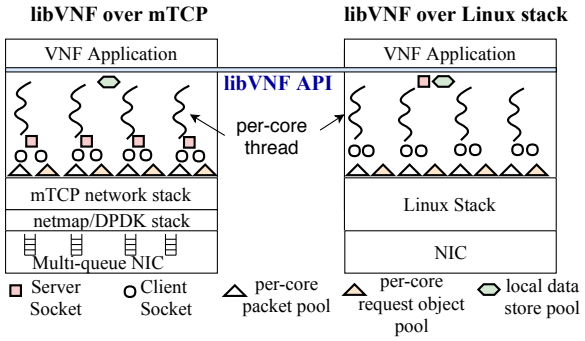

Figure 3: VNF Architecture with libVNF.

Our API provides functions to allocate and manage request objects (§3.3). These objects are maintained in per-core, preallocated, cache-optimized slabs in the library. From comparing Listing 1 and Listing 3, it is clear that the request object abstraction significantly simplifies VNF development in libVNF as compared to frameworks that only expose an event-driven socket-like API.

Finally, libVNF makes it easy to develop VNFs that are implemented as a cluster of replicas for fault tolerance and scalability. Our state management API functions (§3.4) can be used to transparently store and retrieve state across VNF replicas, and our orchestration API (§3.5) can be used to monitor and manage the replicas.

3 API DESIGN AND IMPLEMENTATION

The libVNF API consists of 19 functions that are implemented in around 2500 lines of code in an open-source library [6]. Table 1 summarizes our API, and Figure 3 illustrates the architecture of a VNF built over our API. We now describe our API and its implementation.

3.1 Initialization

Rows A1–A2 in Table 1 show the functions used to initialize a VNF built over libVNF. A VNF starts a packet receiving endpoint, or a *server* using function A1. If the VNF is operating as a transport layer endpoint, the arguments to A1 will consist of the IP address, port number, and protocol of the server. This API function is implemented in the library by creating a listening server socket in the Linux or mTCP network stacks, and binding it to the specified address and port by the library. Instead, if the VNF is operating as a L2/L3 middlebox without transport layer processing, only the network interface is provided as an argument to A1. This functionality is realized by the library using raw sockets in case of the Linux stack, and by directly communicating with the underlying kernel bypass packet I/O mechanism (netmap or DPDK) in the case of the mTCP stack. Our library currently supports registering only one server socket per VNF, due to a similar restriction in the mTCP stack. We currently support only TCP as the transport layer protocol.

We use the terms *connection endpoints* and sockets interchangeably, to refer to both raw sockets over interfaces and transport layer

No.	Function	Inputs	Return value
A1	`createServer`	receiving interface (for L2/L3 VNFs) or IP, port, protocol of listening socket (for transport layer VNFs)	connection identifier
A2	`startEventLoop`	None	None
C1	`createClient`	local server connection id, local client IP, transmitting interface (for L2/L3 VNFs) or IP, port, protocol of remote server (for transport layer VNFs)	connection identifier
C2	`registerCallback`	connection id, event (READ/ACCEPT/ERROR), pointer to callback function with inputs: connection id, pointer to packet buffer, pointer to request object, error code	None
C3	`getPktBuf`	connection id	pointer to empty packet buffer
C4	`sendData`	connection id, pointer to packet buffer, size of data to write	None
C5	`setPktDNE`	connection id, pointer to packet buffer	pointer to packet buffer
C6	`unsetPktDNE`	connection id, pointer to packet buffer	None
C7	`closeConn`	connection id	None
R1	`initReqPool`	array containing sizes of request object types, number of request object types	None
R2	`allocReqObj`	connection id, index of request type	`void *` pointer to request object
R3	`linkReqObj`	connection id, pointer to existing request object	None
R4	`freeReqObj`	connection id, index of request type	None
D1	`setData`	connection id, table name, key, pointer to value, size of value, location (LOCAL/REMOTE), pointer to callback function with arguments: connection id, key, pointer to request object, error code	None
D2	`getData`	connection id, table name, key, location (LOCAL/REMOTE/CHECKCACHE), pointer to callback function with arguments: connection id, key, pointer to value, size of value, pointer to request object, error code	None
D3	`delData`	table name, key, location (LOCAL,REMOTE)	None
D4	`setKeyDNE`	table name, key	None
D5	`unsetKeyDNE`	table name, key	None
O1	`registerfor Notification`	orchestrator (IP, port), pointer to callback function with arguments: task (ADD/REMOVE), vnf_name, vnf_ip_address, event (MAINTENANCE/FAILURE/OVERLOAD)	None

Table 1: The libVNF API functions.

sockets at particular port numbers. API functions that create connection endpoints (including A1) return a *connection identifier* to the VNF code, which is provided as an argument to several other API functions. The connection identifier returned by the library to the VNF is the concatenation of the socket file descriptor and the CPU core number on which the VNF thread is running, because the socket file descriptors are per-core local and not globally unique in mTCP.

After creating a server, the VNF registers callback functions which must be invoked by the library when any event like packet reception occurs on the socket (as described in §3.2). The VNF then invokes function A2 to enter an event loop. The function A2 blocks, and control returns to a callback function in the VNF code when an event of interest occurs. The implementation of A2 depends on the backend network stack. Across all stacks, the library creates one VNF thread per core, and pins the thread to its core. In case of the Linux stack, all threads register for events on the shared server socket created by A1 (with suitable locking), and invoke the `epoll` system call to block for events; the edge-triggered listen socket semantics ensure that every new connection request is delivered to one of the VNF threads only. The mTCP stack uses per-core accept queues and listen sockets for multicore scalability. So, when running over mTCP, the VNF threads invoke mTCP's epoll-like API to block for events on their own per-core listen sockets. In case of L2/L3 VNFs running over the mTCP stack, the underlying kernel bypass mechanism is directly polled for packets, completely bypassing the transport layer in mTCP.

Before invoking the event loop, the library initializes the I/O paths for sending and receiving packets through the network stack. The I/O path configuration depends on whether the VNF is running on

bare metal or inside VMs, and on the specific network stack being used. Note that mTCP scales performance across multiple cores by running one copy of the network stack per core, and binding a separate RX/TX packet queue to each core via a kernel bypass mechanism. When the VNF runs mTCP directly on the physical server, physical NICs can expose multiple RX/TX packet queues to mTCP via the multi-queue functionality. When the VNF runs mTCP inside a VM, we provision multiple queues in a software switch that is compliant with the kernel bypass mechanism (e.g., VALE [43] for netmap and BESS [46] for DPDK), and bind one such queue per core of the VM. Of course, all of this configuration is much simpler when running on the kernel network stack; the library simply sends and receives packets from the network interface exposed by the host/guest kernel. Note that all of this configuration is transparent to the VNF developer, and is completely managed by the library.

3.2 Communication

Rows C1–C7 in Table 1 correspond to functions that are used to communicate with other VNFs in a service chain. Function C1 is used to create a client-side connection endpoint for initiating transmission. Much like A1, C1 can also be used to create both endpoints for L2/L3 VNFs and transport layer VNFs. Function C1 takes the connection identifier of the local server socket as an argument in order to identify the CPU core of the running thread and use per-core data structures where possible. C1 is non blocking, and returns immediately to the VNF after initiating the process of connection setup within the library. Packets sent on a socket by the VNF before connection setup completes are buffered internally by the library, and flushed once connection setup completes. Our API does not notify connection setup failures to the VNF; instead, this error will be discovered and handled by the VNF when it tries to write data. A VNF can create any number of transport layer client sockets using C1. However, if the VNF is opening a connection endpoint to send/receive packets directly from an interface (for L2/L3 VNFs), only one endpoint per interface RX/TX queue will be permitted for obvious reasons.

For all endpoints (server and client), the function C2 registers a callback function with the library. The callback is registered for one of the following events: ACCEPT (new connection), READ (packet arrival), and ERROR. The same connection endpoint can register different callback functions for different events. Once callbacks are registered, the VNF cedes control to the event wait loop in the library. Upon the occurrence of one of these events on any socket, the event loop in the library returns, and the library invokes the suitable VNF callback function to handle the event. The arguments passed to the callback function by the library depend on the event.

The library maintains per-core pools of packet buffers, allocated using slab allocation in the boost library [22]. Much like the TCP socket buffers in Linux, the size of the packet pools must be tuned based on the network linerate; our current implementation allocates 2048 buffers per core, each of size 1024B. When `epoll` or a similar function returns in the library indicating packet arrival (EPOLLIN), the packet is first copied into a free packet buffer in the per-core userspace buffer pool within the library. The library then invokes the corresponding callback function on the socket, providing a pointer to this packet buffer as an argument. If the event loop returns to indicate an error on any socket (e.g., ENOTCONN), the callback is invoked with the suitable error code as an argument, and with a null pointer for the packet buffer. If the event loop indicates a new TCP connection request (EPOLLIN on the listen socket), the connection identifier of the new connection is passed as an argument. Finally, every socket can optionally have some application state associated with it in the form of a request object (§3.3), and a pointer to this request object, if it exists, is also returned.

To send data on any connection, the VNF must first obtain a pointer to a free packet buffer using API function C3. The connection identifier provided as an argument to C3 helps the library identify the core-local packet buffer pool to allocate from. Once the VNF fills the buffer with the desired contents, it must invoke API function C4 to transmit the packet. The VNF can just fill in the payload if it is sending packets via a transport layer stack, but must generate all network layer headers itself when it is acting as a L2/L3 VNF and communicating over raw sockets. API C4 to send packets is a non-blocking function, and errors that occur during the transmission are notified to the VNF via a later error event callback.

Prior work has shown that batching the transmission and reception of packets can improve network I/O performance [42]. Therefore, libVNF strives to incorporate batching in its implementation wherever the underlying network stack supports it. mTCP already incorporates batching at the transport layer and transport layer VNFs built over mTCP automatically gain the benefits of batching. However, when implementing L2/L3 VNFs that do not undergo transport layer batching, our library enables batching directly via the kernel bypass mechanism. For example, our library reads a batch of packets from the NIC, and then invokes callback functions for every packet in order. Similarly, on the transmit path, the library collects a batch of packets before handing them over to the kernel bypass mechanism. Batching is not implemented over the kernel stack because the socket API of Linux does not support it.

Packet buffers are reused by the library after the VNF has processed the received packet and the callback returns. However, if a VNF does not wish the buffer to be recycled immediately (e.g., to store a pointer to the packet in a request object, and access it in another callback), it can use API function C5 to set the *Do Not Evict (DNE)* flag on the packet buffer. Buffers with this flag set are not recycled until the flag is unset using function C6. The requirement of storing packet buffers beyond the callback is the reason why libVNF maintains its own pool of packet buffers in userspace, even when running on stacks like mTCP that have a similar mechanism internally. If the number of free packet buffers in the pool fall below a threshold, the library proceeds to allocate more memory from the kernel and expand its pool. We currently do not have mechanisms to detect malfunctioning VNFs that exhaust the packet buffer pool, e.g., by not un-setting the DNE flag, and a better mechanism of garbage collecting unused packet buffers is part of future work.

The final API function pertaining to communication is C7, used to close a connection and release all state pertaining to it.

3.3 Request Objects

libVNF provides the abstraction of request objects to easily manage application state across multiple callbacks. libVNF has no knowledge of specific application semantics, so VNF developers indicate

the size of application state that they wish to track across callbacks to the library via API function R1. This function takes as argument the number of types of request objects and their sizes. Upon receiving this information from the VNF, the library provisions multiple pools of request objects per core, one pool for each object size, after rounding up the request object sizes to the smallest power-of-2 block that can accommodate the object. Our implementation currently allows 4 pools of 2^{20} objects each per core, and the pools are preallocated using a slab allocator [22] to avoid dynamic memory allocation overheads. The per-core request object pools are cache-optimized, and accessed by the VNF threads in a lockfree manner.

Every connection endpoint can be optionally associated with a request object, and this object is returned on every callback associated with that endpoint. If a connection has no request object associated with it yet, API function R2 can be used to allocate a new request object of a specific size and associate it with a connection endpoint. Further, multiple connection endpoints (e.g., sockets to communicate with `A` and `C` at `B`) can also be associated with the same request object, because handling one application layer request can rely on multiple asynchronous network communications. In such cases, API function R3 can be used to associate an existing request object to another connection endpoint. Once a VNF has completed processing a request (or has encountered an error and wishes to abort a request), the request object can be deallocated using API function R4. Note that the API gives applications complete control over what state they wish to track across callbacks, and what the granularity of a request is. We currently assume that the application has one outstanding request at a time on a connection, and we plan to relax this assumption in future work.

Request objects can store pointers to other objects managed by the library, e.g., packet buffers. Recall that API function C5 is used to obtain a pointer to a non-evictable packet buffer, and such pointers can be stored in request objects without worrying about the buffers being recycled. The VNF developer should take care to ensure that the pointers stored in a request object are released (e.g., packet buffer marked as evictable using function C6) before a request object is released. We assume that the memory allocated for a request object is freed by the VNF developer at the end of the request processing.

3.4 State Management

The libVNF API provides functions to store application state (assumed to be in the form of multiple tables of key-value pairs) across per-core threads of the VNF application, and across multiple distributed replicas of a VNF. The API supports non-blocking get, set, and delete of these key-value pairs. Our framework provides a local datastore that is shared across all threads of a single VNF instance, and a remote shared data store that is shared across multiple VNF replicas. In addition, the library also maintains a local FIFO cache of key-value pairs fetched from the remote datastore, which can be used to improve performance when accessing remote state. Our current implementation assumes that keys are integers and values are dynamically allocated blobs of arbitrary size.

API function D1 is used to store a key-value pair, in the local or remote datastores. The function is non-blocking, and an associated callback function is invoked when the set operation has completed by the library (successfully or otherwise). Function D2 is used to retrieve a key-value pair. In addition to fetching from the local or remote datastore, the API can also indicate whether the local cache must be checked before contacting the remote server. Note that applications can use the cache for enhanced performance, but may choose to forgo checking in the local cache in case the application logic deems the cached value to be stale. The get operation is also non-blocking, and a callback function is invoked by the library when the get completes. Fetched key-value pairs are also stored in the datastore cache (if not present already), and a pointer to the dynamically allocated value is provided to the callback function. Finally, API function D3 is used to delete keys from local and remote datastores. Note that key-value pairs stored in the cache can be evicted by the FIFO eviction policy any time after the callback returns. However, if the VNF requires to hold on to a value for a little while longer (e.g., for storing its pointer in a request object) it may use API functions D4/D5 to set/unset the *Do Not Evict (DNE)* flag of the cache entry.

The libVNF library implements the local datastore and the datastore cache as hashmaps that point to pre-allocated blocks of memory storing values. The local datastore and cache are accessed from the per-core VNF threads through locking. We use the Redis in-memory key-value store as the remote datastore in our implementation because of its widespread use as a distributed fault tolerant datastore in cloud applications; other datastores can be considered as part of future work. The library opens one TCP connection per CPU core to the datastore during initialization. In order to ensure that all communication to the datastore is non-blocking, we built a wrapper over Redis (using hiredis-vip([2]) that runs at the Redis server and communicates asynchronously with our library threads running on VNF replicas. We use the transaction support provided by Redis to correctly handle multiple writes to the same key by different replicas, and guarantee atomic updates of keys across replicas.

3.5 Orchestration

Multiple VNFs hosted on a cloud in an NFV deployment will typically be managed by an orchestration framework that is part of a cloud management software. Among other things, an orchestrator takes care of instantiating VNFs on physical servers and managing the life-cycle of the VNFs. For example, NFV orchestrators like OpenBaton [11] and OPNFV [14] use simple heuristics such as CPU utilization thresholds to trigger scaling events. Once an orchestrator detects a scaling or failure event, it takes suitable action to address the situation, e.g., spawning a new replica of a VNF during failure or overload, or decommissioning an existing replica during underload or maintenance. libVNF does not seek to replicate this functionality of detecting failures or overload/underload scenarios. However, libVNF can integrate with orchestrators and learn of these events, in order to notify VNFs that wish to adapt to these events. For example, a load balancer VNF built over libVNF may wish to adapt its traffic steering algorithm based on the number of replicas it is steering traffic to. Any VNF that wishes to know the status of another VNF from the orchestrator for such use-cases may register for notifications using orchestration API function O1. libVNF interfaces with an orchestrator to learn of failure and scaling events, and invokes a callback function in all VNFs registered for orchestration notifications. The callback is invoked with the following details that are available

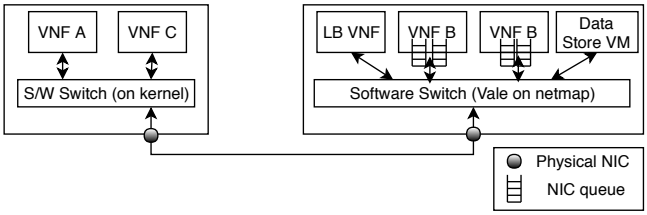

Figure 4: Experimental setup for microbenchmarking.

at the orchestrator: the identity of the VNF whose status has changed, the type of change (addition or removal of replicas), and the reason for the change (e.g., failure). The VNF receiving these notifications can then take suitable action to handle the failure or scaling event. We have currently integrated libVNF with a homegrown orchestrator built on simple VM monitoring tools, and have implemented the orchestration API in this setup. We plan to explore the integration with a more sophisticated orchestrator as part of future work.

We would like to note that, while libVNF does not contain explicit mechanisms for fault tolerance beyond whatever state has been stored in the remote datastore by the VNF itself, it has enough hooks to enable VNF developers to build fault tolerant distributed implementations customized to the needs of the application. When failures happen, our library returns error codes back to the VNF developer in callbacks, and orchestrator notifications about new replicas being spawned. With these inputs, the VNF developer can handle faults in a manner that is appropriate for the application semantics. For example, when a transport layer connection to an upstream VNF times out, the VNF could choose to abort the request or retry once again at another replica. Similarly, when a VNF replica fails, the new replica can decide how it should resume execution based on what state has survived the failure in the remote datastore.

4 EVALUATION

Our evaluation of libVNF comprises two parts. First, we run microbenchmarks and show that the performance of VNFs built with our library is usually within 10% of the performance of optimized implementations built without using the library, and that the performance scales well, both with increase in number of CPU cores in the VNF, as well as increase in the number of replicas of a distributed VNF implementation (§4.1). Second, we build several complex real-world VNFs over our library, and show that our API is expressive enough to develop a wide variety of VNFs, and that using our library results in up to 50% reduction in development effort (as measured by LoC) (§4.2).

4.1 Microbenchmarks

Setup. We consider a simple service chain consisting of VNFs A, B, and C, as shown in Figure 2. VNFs `A` and `C` are implemented as multithreaded C++ applications over `pthreads`, while `B` is the VNF of interest for our benchmarking exercise. VNF `A` generates multiple requests to `B` in a closed loop fashion in order to saturate `B`. Upon receiving a request, `B` performs a CPU-intensive computation (updating a value in a busy for loop), communicates with `C`, and then replies back to `A`. We measure the performance of `B` in terms of end-to-end throughput (requests completed/sec) and average latency of request completion, as measured at `A`. Across all experiments, we ensure that the VNFs `A` and `C` are not the bottleneck. All graphs report the minimum, maximum, and average values of metrics over 5 runs of 300 seconds each.

We implement `B` both using the libVNF API (we call this version BLib), and as a regular non-blocking C++ application (we call this version BNoLib). We run each of BLib and BNoLib on three different network stacks: the regular Linux kernel, the mTCP stack over DPDK, and mTCP over netmap. Note that the code of BLib did not change across the various network setups, while BNoLib was significantly rewritten for each network stack. We also ran BLib in a clustered configuration, with multiple replicas of `B` synchronizing state using a remote datastore. Migrating to a distributed implementation also did not require rewriting the code of BLib. We used a simple load balancer built over our API (see §4.2) to steer traffic across multiple replicas of `B`.

We use two Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz servers with 24 cores, 64GB memory to host the VNFs. VNFs `A` and `C` run inside VMs on one Linux (Ubuntu 14.04) server with KVM hypervisor, while one or more replicas of `B` (along with the load balancer and remote datastore VMs) run on a second Linux (CentOS7) server that is connected to the first over a 1Gbps link. When testing `B` on the mTCP/netmap stack, `B` runs inside a VM that is hosted over the netmap-based VALE software switch, as shown in Figure 4. When testing `B` on the mTCP/DPDK (dpdk-17.08) stack, `B` runs directly on the physical server, and not inside a VM, because we could not get mTCP to run inside a VM using the DPDK kernel bypass mechanism. (Specifically, mTCP did not recognize the multiple queues exposed by BESS, which is a DPDK-based software switch.) When testing `B` on the Linux and netmap stack, `B` runs inside a VM hosted on the KVM hypervisor. VMs hosting `A` and `C` were provisioned with 4 CPU cores and 4GB RAM each, while each replica of `B` was provisioned with 4GB RAM. The number of cores and replicas of `B` varied across experiments. In all the above setups, we first verified correct packet transmission and reception using simple test programs before running our benchmarking experiments.

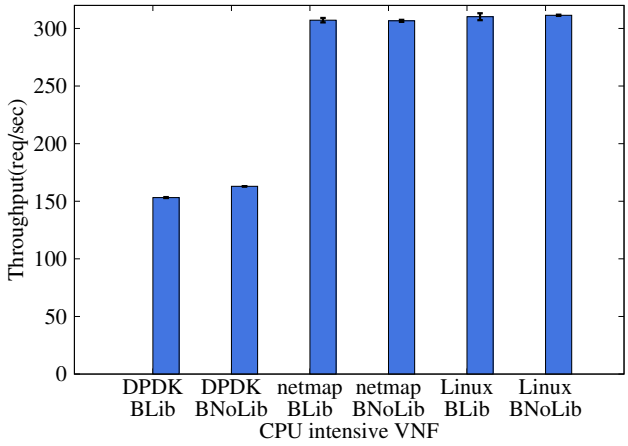

Figure 5: Single core CPU intensive application: throughput comparison with and without library.

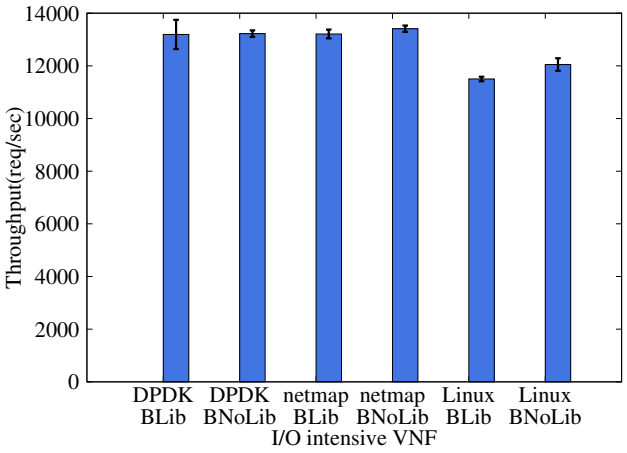

Figure 6: Single core I/O intensive application: throughput comparison with and without library.

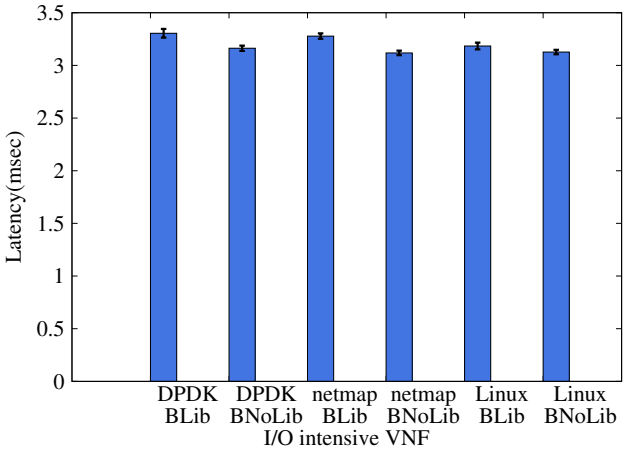

Figure 7: Single core I/O intensive application: latency comparison with and without library.

Comparison of single core VNFs. We first compare the performance of BLib and BNoLib when running on a single core VM, across three different network stacks: Linux, mTCP/netmap, and mTCP/DPDK. Figure 5 shows the throughput of `B` across all 6 experiments. We see from the figures that the throughput of BLib is within 5% of that of BNoLib, indicating that our library implementation does not introduce any overheads. However, we noticed that the mTCP/DPDK stack underperformed in this experiment. We found the reason to be the polling variant of the DPDK kernel bypass driver (unlike the interrupt-based driver of netmap), which did not share the single CPU well with the CPU-intensive application. To verify this fact, we modified `B` to skip the CPU computation on each request, i.e., upon receiving `A`'s request, `B` simply communicated with `C` and replied back to `A`. Figure 6 shows the throughput of `B` in this I/O intensive mode of operation. We now see that both mTCP/DPDK and mTCP/netmap outperform the kernel stack, and once again, the throughputs of BLib and BNoLib are comparable across all stacks. Further, Figure 7 shows that the request processing latencies of BLib and BNoLib are comparable as well.

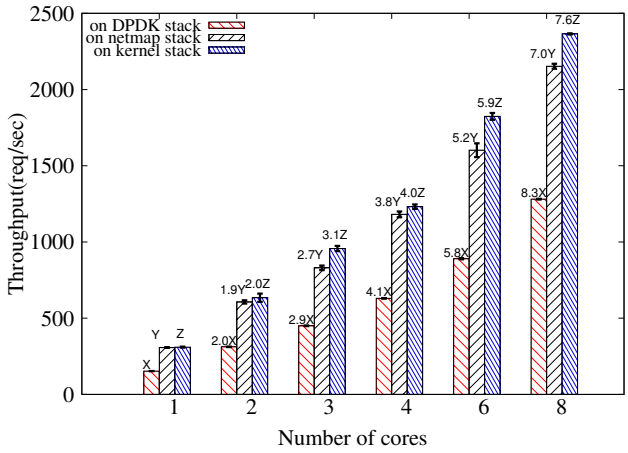

Figure 8: Performance scaling with increasing CPU cores.

Multicore scalability. We now consider the CPU-intensive version of `B` built over the library (i.e., BLib) and test how its performance scales with increasing number of CPU cores of the VNF `B`. Figure 8 shows the throughput of `B`, across all three network stacks, as the number of CPU cores are varied from 1 through 8. We see from the figure that the performance of `B` scales well with increasing cores due to several implementation choices made in our library (e.g., use of per-core data structures where possible).

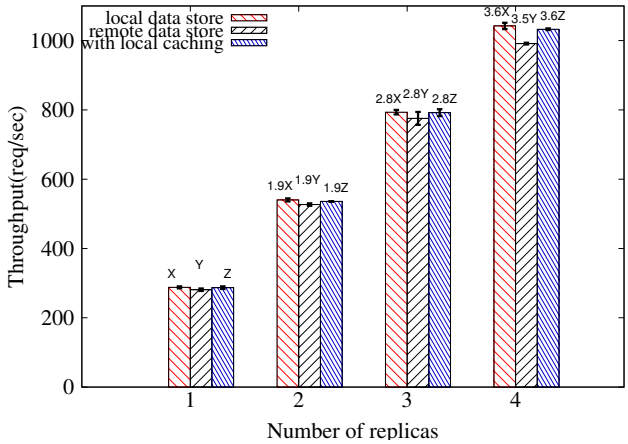

Figure 9: Performance scaling with increasing replicas.

Scalability across replicas. We now consider single core `B` built using the library (i.e., BLib) over the mTCP/netmap stack, and run it as a distributed VNF with varying number of replicas. We consider various methods of storing/retrieving state at `B` (by changing the API parameters in `B`): always accessing the local datastore, always accessing the remote Redis datastore running on another VM, and accessing the remote datastore via a cache (with a very high hit ratio of 97%). Figure 9 shows the throughput of `B` as the number

of replicas of B varied from 1 to 4. We see from the figure that the performance of B scales well with increasing replicas in all the three state management options. Further, we see that B's performance degrades slightly when using the remote datastore as compared to the local datastore, due to the extra network stack processing required for remote communication. However, caching state locally as done by our library can reduce this performance degradation for workloads with good cache locality.

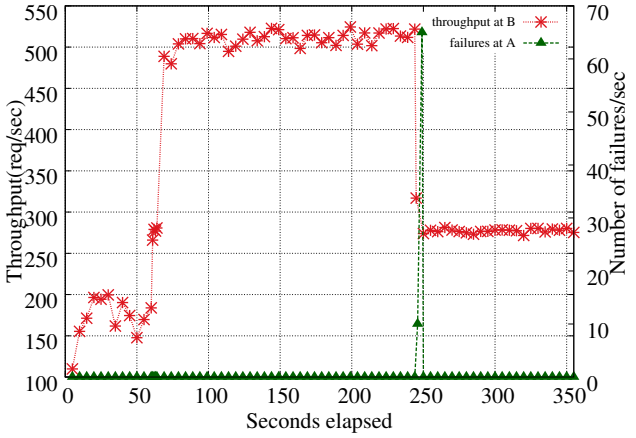

Figure 10: Scale-out and failure timeline.

Handling orchestration events. We now consider the clustered implementation of B, where the load balancer of the cluster is built over the orchestration API of libVNF and uses the orchestrator notifications to adapt its traffic steering. Figure10 shows the aggregate throughput of a clustered implementation of B and the failure rate at A as a function of time. This is across two events: the load from A was increased at around 60 seconds leading to the orchestrator increasing the number of replicas of B from 1 to 2, followed by one of the replicas being taken down at around 245 seconds. Across both events, we found that the load balancer was notified of the event and could successfully adapt its traffic steering to the change in the status of B's replicas within 5 seconds of the event. This recovery time is a function of the granularity at which libVNF synchronizes with our simple orchestrator, and can be improved further in the future.

VNF	Throughput No library	Throughput With libVNF	LoC saved
IMS	5483 reg/s	5678 reg/s	42%
EPC	7988 reg/s	7548 reg/s	38%
LB	2.7 Gbps	2.3 Gbps	52%

Table 2: VNF development with and without libVNF.

4.2 Building VNFs with libVNF

We now build three sample VNFs using our library. The VNFs we build range from a simple L3 load balancer to several complex VNFs deployed in the packet core of mobile telecommunication networks. We implement all VNFs over the Linux and mTCP stacks, with and without using the libVNF APIs. Table 2 summarizes the key results comparing the VNF implementations using our library with those that were built without libVNF.

IMS. IMS (IP Multimedia Subsystem) is a set of networking elements used to setup voice and video calls over IP within modern telecommunication networks. IMS comprises of multiple components: Proxy Call Session Control Function (P-CSCF), Interrogating CSCF (I-CSCF), Serving CSCF (S-CSCF), and a Home Subscriber Server(HSS). These components communicate with each other as part of several service chains, and together handle requests to register mobile users and setup calls between pairs of registered users, among other things. The VNFs within IMS are all transport layer endpoints that receive requests from mobile users (and from other VNFs in the service chain) over TCP and perform CPU-intensive computation (such as authentication) to process the requests.

We implemented simplified versions of the IMS VNFs that handle user registrations alone [8], both using libVNF and without the library. We then ran experiments to compare the performance of the two versions, when both versions were running on the Linux stack. All VNFs run within VMs provisioned with 2 CPU cores and 4GB RAM, hosted on an Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz server with 24 CPU cores and 64 GB RAM. We also built a multi-threaded closed loop load generator to fire registration requests and saturate the IMS VNFs, and measure the saturation throughput of the system as the average number of registrations successfully completed per second at the load generator. Row 1 of Table 2 shows that there was no performance overhead in implementing the IMS VNFs over the library. In fact, we noticed a slight improvement in performance when using the version built over the library. Upon profiling our code, we found that this improvement in performance was due to the preallocation of several data structures within the library, as compared to the dynamic memory allocation overhead incurred by the version built without using the library. Further, we found that using the library API reduced the LoC in the implementation by over 40%.

LTE EPC. Long Term Evolution Evolved Packet Core (LTE EPC) is the packet core of modern 4G telecommunication networks, and connects the radio network with the rest of the Internet. Mobile users register with the network and send mobile data, which is then tunneled through the EPC using packet forwarding state that is created during the registration procedure. The main components of the EPC are the Mobility Management Entity (MME) and, Serving and Packet Gateways (S/P-GWs). MME handles user registration and other CPU-intensive control plane procedures and sets up forwarding state to tunnel user traffic through the SGW and PGW gateway VNFs.

We implemented simplified versions of all three EPC components (MME, SGW, PGW), along with a closed loop load generator that generates control and dataplane traffic to saturate the EPC [9]. Our simplified version of the EPC code only handles user registration/deregistration, datapath setup, and data transfer; we do not handle other EPC procedures, e.g., handovers. We then ported the simplified MME component to use the libVNF API, and compared the performance of the VNFs built with and without the library over the mTCP/netmap stack. (We have not yet ported the SGW/PGW components to the library as they run over the UDP transport layer stack that is currently not supported by mTCP.) In the experiment,

we run all EPC components and the load generator inside VMs provisioned with 1 CPU core and 4GB RAM, hosted on an Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz server with 24 cores and 64 GB RAM. We let the load generator saturate the MME with control plane traffic and measure the saturation throughput (in terms of the number of registration successfully completed per second) of the versions of MME. From Row 2 of Table 2, we see that using our library imposes a low overhead of under 6%, and results in LoC savings of around 38%.

As discussed in §1, we also compared the performance of the mTCP/netmap and Linux stack versions of IMS and EPC (both built without the library) in order to understand the benefits of kernel bypass mechanisms over the kernel stack, as part of the motivation of our work. We found that the mTCP-based kernel bypass stack resulted in very small performance gains (1.4× higher than that of the Linux stack in case of IMS) when running CPU intensive workloads, but performed very well (e.g., 33× higher than the Linux stack in case of the SGW/PGW of EPC) when running I/O intensive workloads. Upon profiling the code, we found that the CPU cycles saved during I/O processing in a kernel bypass stack were insignificant in comparison to the CPU cycles spent on application-layer processing in the case of CPU-intensive workloads, resulting in a small performance gain for the kernel bypass techniques.

L3 Load Balancer. Next, we build a simple L3 load balancer (LB), both with and without the library, over the netmap kernel bypass mechanism. Recall that this LB was also used to steer traffic to multiple VNF replicas in Figure 4. Our LB is not very sophisticated—it simply uses the hash of the source port to distribute incoming flows to a fixed set of backend replicas—but it suffices to demonstrate the ability of our library to build L2/L3 VNFs. The experience of building a L3 VNF using the same API functions that are used to build a transport layer VNF was a challenging experience, and led us to refine our library implementation in significant ways.

In order to test the performance of our LB, we ran an iperf [4] client and server on two separate VMs, with the LB VM intercepting the flow to rewrite packet headers, and measured the throughput achieved by iperf. All VMs were provisioned with one core and 1GB RAM, and were hosted on an Intel i5 server with 4 cores and 12GB RAM. From Row 3 is Table 2, we see that the forwarding capacity of the LB VNF is 14% lower when built over our API as compared to when built directly over the netmap API. We found that this difference in throughput between the implementations was due to the additional API calls being made within the library. Note that the impact of the additional API calls by the library is higher in this very simple VNF that does very little work per packet, but this overhead was negligible in comparison with the VNF processing in more complex VNFs (IMS and EPC).

5 RELATED WORK

Network stack design optimized for NFV. Prior work has identified several shortcomings in the network I/O subsystem of standard operating systems, and has provided solutions for the same. One set of solutions (e.g., Megapipe [27], FastSocket [33]) propose fixes to the Linux network stack to improve network I/O performance and make it scale well across multiple CPU cores. On the other hand, kernel bypass mechanisms like DPDK [3] and netmap [42] choose to bypass the Linux kernel altogether and process network packets directly in userspace, with the help of clean-slate multicore scalable network stacks (e.g., mTCP [30]). Clean slate operating systems designs such as IX [21], Arrakis [39] propose a refactored kernel design that is better suited to fast I/O processing. Finally, modern NICs (e.g. smartNIC [7]) are evolving to offload network stack processing from the CPU in order to improve I/O performance. Our work on libVNF is orthogonal to this body of work, and the implementation of the libVNF API can leverage such optimized network stacks in the backend.

Frameworks to build L2/L3 VNFs. Click [32] is a seminal work on providing suitable abstractions to ease the development of L2/L3 middleboxes. More recently, VNF development frameworks such as Netbricks [38], YANFF [17], and VPP [16] are built on top of DPDK, and expose a rich packet header manipulation API to build L2/L3 VNFs. PISCES [44] allows developers to specify a L2/L3 packet processing pipeline in the P4 [23] language, and compiles it to a C-based software switch code. However, unlike libVNF, these frameworks cannot be easily used to build VNFs that also act as transport layer endpoints, nor do they have mechanisms to enable a distributed clustered implementation of a VNF.

On the other hand, these frameworks expose a richer API for packet header manipulation than libVNF, which only delivers raw packets to the VNF. While we plan to explore enhancements to our API to support richer packet manipulation in the future, we believe that the strength of libVNF really lies in the ability to develop both L2/L3 and transport layer VNFs within the same framework. Therefore, we envision libVNF being used to primarily develop transport layer VNFs and the L2/L3 VNFs that need to coexist with these transport layer VNFs (e.g., L3 load balancer to steer traffic to the multiple transport layer VNF replicas), while frameworks with a richer API for L2/L3 VNF development can be used when the ecosystem has no VNFs with transport layer endpoints.

Frameworks to build transport-layer VNFs. Flick [19] proposes a domain-specific language to build VNFs over the mTCP userspace stack, but does not support the development of horizontally scalable VNFs. Network stacks such as mTCP [30], TLDK [15] build transport layer (TCP/UDP) functionality in userspace over a kernel bypass mechanism like DPDK, and expose an event-driven epoll-like API for network communication. mOS [29] is a network stack derived from mTCP that provides the abstraction of a monitoring socket and an event-driven API over it, so that TCP-level flow state can be reconstructed even on middleboxes that do not terminate TCP connections. Our work is complementary to such efforts; the libVNF provides a higher-level API than sockets, and can use most of these network stacks in the backend.

Frameworks to build horizontally scalable VNFs. Split/Merge [41] and OpenNF [26] address the problem of efficiently migrating application state across distributed stateful middleboxes during scale-out and scale-in. On the other hand, StatelesNF [31] proposes to handle failures and scaling in distributed middleboxes not via migrating state across replicas, but by storing all shared state in a remote data store. S6 [47] provides a shared state space abstraction to distributed middleboxes, and transparently migrates state to optimize shared state access. However, unlike libVNF, these frameworks are not integrated with a transport layer stack to build VNFs that act as transport layer endpoints.

Frameworks to build fault-tolerant VNFs. The Pico-Replication [40] framework enables the development of fault tolerant middleboxes by replicating TCP flow state across replicas, while FTMB [45] achieves the same goal by carefully replaying input packets. libVNF does not explicitly implement mechanisms for fault tolerance; VNF developers can customize the fault tolerance logic of their applications using the the libVNF API functions for checkpointing state, detecting connection errors, and performing efficient load-balancing on failures (e.g., as described in recent research on load balancers [20, 24, 36]).

NFV infrastructure and orchestration. While our work focuses on buildng VNFs, other efforts in industry and academia have addressed the complementary problems of developing an optimized NFV infrastructure to deploy the VNFs, and mechanisms to orchestrate and manage the deployed VNFs. NetVM [28] and OpenNetVM [50] propose NFV platforms based on DPDK, and optimize communication across VNFs of a service chain hosted on the same physical machine via zero copy packet transfer. Flurries [49] is an NFV platform that is designed to host lightweight VNFs in containers. ClickOS [34] is a Xen-based NFV platform optimized to host middleboxes. OPNFV [14] aims to create a standardized NFV software platform, comprising of several open-source systems such as the OpenStack cloud management software [13], the OVS (Open vSwitch) software switch [12], and the KVM hypervisor [5]. ONAP [10] is an industry effort to create an orchestration platform for VNFs, while E2 [37] and Dysco [48] are academic efforts that address the orchestration and placement problem.

6 CONCLUSION

This paper presents libVNF, a library to ease the development of high-performance horizontally-scalable VNFs. The libVNF API is expressive enough to build a wide variety of VNFs, ranging from L2/L3 middleboxes, to those that terminate transport layer connections and perform significant processing at the application layer. libVNF provides mechanisms for managing states across multiple replicas of a VNF, thereby simplifying the process of building horizontally scalable implementations of network functions. The library API builds over the event-driven communication API exposed by modern network stacks, and provides abstractions that ease the pain of managing application state across event callbacks. The API functions are agnostic to the choice of network stack, and can work with multiple network stacks in the backend. Experiments with VNF prototypes built over our API show that libVNF enables the development of high performance VNFs, saves significant development effort, and introduces minimal overhead. In the future, we propose to extend our API to provide non-blocking functions for file handling and other essential activities, by integrating our library with an asynchronous disk I/O framework. Our library also lacks a rich API for packet header manipulation, like that found in other frameworks for building L2/L3 VNFs [16, 38, 44], and we plan to enhance our API in this direction as well. We believe that a library such as ours, if widely used for VNF development, can significantly accelerate the adoption of NFV.

ACKNOWLEDGEMENTS

We thank Sagar Tikore and Vaishali Jhalani for helping with some of the experiments. We thank Yashasvi Sri Ram and Sai Sandeep for helping make libVNF more user-friendly. We are also thankful to the anonymous reviewers for their insightful comments and feedback. This work is supported by a research grant from Intel Corporation, and a Ph.D. student fellowship from IIT Bombay.

REFERENCES

[1] 2018. CORD Intel EPC. https://gerrit.opencord.org/ngic.
[2] 2018. Hiredis-vip. https://github.com/vipshop/hiredis-vip.
[3] 2018. Intel Data Plane Development Kit. http://dpdk.org/.
[4] 2018. Iperf. https://software.es.net/iperf/.
[5] 2018. Kernel Virtual Machine. https://www.linux-kvm.org/page/Main_Page.
[6] 2018. libVNF. https://github.com/networkedsystemsIITB/libVNF.
[7] 2018. Netronome. https://www.netronome.com/products/smartnic/overview/.
[8] 2018. NFV-IMS. https://github.com/networkedsystemsIITB/NFV_IMS.
[9] 2018. NFV-LTE-EPC. https://github.com/networkedsystemsIITB/NFV_LTE_EPC.
[10] 2018. ONAP. https://www.onap.org/.
[11] 2018. Open Baton. https://openbaton.github.io/documentation/.
[12] 2018. Open vSwitch. https://www.openvswitch.org/.
[13] 2018. Openstack. https://www.openstack.org/.
[14] 2018. OPNFV. https://www.opnfv.org/.
[15] 2018. Transport Layer Development Kit. https://github.com/FDio/tldk.
[16] 2018. Vector Packet Processing. https://github.com/FDio/vpp.
[17] 2018. Yet Another Network Function Framework. https://www.slideshare.net/MichelleHolley1/new-model-for-cloud-network-function-development-yanff.
[18] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. 2002. Cooperative Task Management Without Manual Stack Management. In *Proc. of ATEC '02*.
[19] Abdul Alim, Richard G. Clegg, Luo Mai, Lukas Rupprecht, Eric Seckler, Paolo Costa, Peter Pietzuch, Alexander L. Wolf, Nik Sultana, Jon Crowcroft, Anil Madhavapeddy, Andrew W. Moore, Richard Mortier, Masoud Koleni, Luis Oviedo, Matteo Migliavacca, and Derek McAuley. 2016. FLICK: Developing and Running Application-Specific Network Services. In *Proc. of USENIX ATC'16*.
[20] Joao Taveira Araujo, Lorenzo Saino, Lennert Buytenhek, and Raul Landa. 2018. Balancing on the Edge: Transport Affinity without Network State. In *Proc. of NSDI'18*.
[21] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proc. of OSDI'14*.
[22] Boost. 2018. https://www.boost.org/.
[23] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. 2014. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* 44, 3 (July 2014), 87–95.
[24] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A Fast and Reliable Software Network Load Balancer.. In *NSDI'16*.
[25] ETSI. 2012. Network Functions Virtualisation. https://portal.etsi.org/nfv/nfv_white_paper.pdf.
[26] 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 *Proc. of SIGCOMM'14*.
[27] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proc. of OSDI'12*.
[28] Jinho Hwang, K. K. Ramakrishnan, and Timothy Wood. 2014. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. In *Proc. of NSDI'14*.
[29] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. 2017. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *Proc. of NSDI'17*.
[30] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *Proc. of NSDI'14*.
[31] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. 2017. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In *Proc. of NSDI'17*.
[32] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. 2000. The Click Modular Router. *ACM Trans. Comput. Syst.* 18, 3 (Aug. 2000), 263–297. https://doi.org/10.1145/354871.354874

[33] Xiaofeng Lin, Yu Chen, Xiaodong Li, Junjie Mao, Jiaquan He, Wei Xu, and Yuanchun Shi. 2016. Scalable Kernel TCP Design and Implementation for Short-Lived Connections. In *Proc. of ASPLOS '16*.

[34] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *Proc. of NSDI'14*.

[35] Priyanka Naik and Mythili Vutukuru. 2017. libVNF: A Framework for Building Scalable High Performance Virtual Network Functions. In *Proc. of APSys '17*.

[36] Vladimir Olteanu, Alexandru Agache, Andrei Voinescu, and Costin Raiciu. 2018. Stateless Datacenter Load-balancing with Beamer. In *Proc. of NSDI'18*.

[37] Shoumik Palkar, Chang Lan, Sangjin Han, Keon Jang, Aurojit Panda, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. E2: A Framework for NFV Applications. In *Proc. of SOSP'15*.

[38] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *Proc. of OSDI'16*.

[39] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *Proc. of OSDI'14*.

[40] Shriram Rajagopalan, Dan Williams, and Hani Jamjoom. 2013. Pico Replication: A High Availability Framework for Middleboxes. In *Proc. of SoCC'13*.

[41] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. 2013. Split/Merge: System Support for Elastic Execution in Virtual Middleboxes. In *Proc. of NSDI'13*.

[42] Luigi Rizzo. 2012. netmap: A Novel Framework for Fast Packet I/O. In *Proc. of USENIX ATC'12*.

[43] Luigi Rizzo and Giuseppe Lettieri. 2012. VALE, a Switched Ethernet for Virtual Machines. In *Proc. of CoNEXT '12*.

[44] Muhammad Shahbaz, Sean Choi, Ben Pfaff, Changhoon Kim, Nick Feamster, Nick McKeown, and Jennifer Rexford. 2016. PISCES: A Programmable, Protocol-Independent Software Switch. In *Proc. of SIGCOMM '16*.

[45] Justine Sherry, Peter Xiang Gao, Soumya Basu, Aurojit Panda, Arvind Krishnamurthy, Christian Maciocco, Maziar Manesh, João Martins, Sylvia Ratnasamy, Luigi Rizzo, and Scott Shenker. 2015. Rollback-Recovery for Middleboxes. In *Proc. of SIGCOMM'15*.

[46] Berkeley Extensible Software Switch. 2018. http://span.cs.berkeley.edu/bess.html.

[47] Shinae Woo, Justine Sherry, Sangjin Han, Sue Moon, Sylvia Ratnasamy, and Scott Shenker. 2018. Elastic Scaling of Stateful Network Functions. In *Proc. of NSDI'18*.

[48] Pamela Zave, Ronaldo A. Ferreira, Xuan Kelvin Zou, Masaharu Morimoto, and Jennifer Rexford. 2017. Dynamic Service Chaining with Dysco. In *Proc. of SIGCOMM '17*.

[49] Wei Zhang, Jinho Hwang, Shriram Rajagopalan, K.K. Ramakrishnan, and Timothy Wood. 2016. Flurries: Countless Fine-Grained NFs for Flexible Per-Flow Customization. In *Proc. of CoNEXT '16*.

[50] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, K.K. Ramakrishnan, and Timothy Wood. 2016. OpenNetVM: A Platform for High Performance Network Service Chains. In *Proc. of HotMIddlebox '16*.