Library Operating System with Mainline Linux Network Stack

Hajime Tazaki, Ryo Nakamura, Yuji Sekiya

University of Tokyo
Tokyo, Japan
{tazaki,sekiya}@wide.ad.jp, upa@haeena.net

Abstract

Library operating system (LibOS) is a userspace version of Linux kernel to provide an operating system personalization (or ad-hoc network stack) as well as yet-another virtualization primitive. Although the concept of library operating system is not new and was established in back to 90's, the idea here is adding a hardware independent architecture (i.e., arch/lib) into Linux kernel tree and reusing the rest of networking code as a library for userspace programs in order to avoid 'reinventing the wheel'. Unlike conventional Linux kernel/userspace model, system calls are redirected to the library in the same process or the other userspace processes, but the framework tries to be transparent so that all of the existing userspace applications like nginx and iproute2 are able to be used as-is. The LibOS framework provides several interesting use cases such as 1) a fast-path for the new protocol deployment (no need to replace or insert new kernel code), 2) a feature-rich network stack for a high-speed packet I/O mechanism like Intel DPDK, 3) a continuous integration for testing networking code implemented in Linux kernel tree. Right now, most of in-kernel protocols like TCP, SCTP, DCCP, and MPTCP are tested to work on top of the LibOS. Newly implemented protocol may also work depending on the POSIX API coverage and kernel glue code.

This paper covers the introduction of the LibOS framework and two sub projects, Network stack in userspace (NUSE) and Direct Code Execution (DCE), with the internal design of the indirections, and presents the ongoing work on the multi-process support to share a single userspace network stack (e.g., share a userspace routing table between two processes) via inter-process communication implemented by rumpkernel IPC/RPC framework.

Keywords

userspace network stack, library operating system, NUSE, DCE

1. Motivation

When the price of a packet was expensive and packet handling was a holy operation, network stacks were implemented in kernel space[1]. Fair enough. But in these days, the unit price of packets becomes cheaper, then the assumption of kernel implementation has been changed: there is no longer strong reason staying in kernel space.

The work is thus motivated to design the userspace network stack, as a library operating system (LibOS), in order to bring a lot of benefits to the Linux kernel like 1) rapid evolution of network stack, 2) lightweight virtualization only focusing on network part, and 3) full controllable testing environment.

The outlook of our LibOS design is to reuse the existing Linux codebase and to avoid dedicated modifications to the codebase as much as possible, and attach to the userspace application as a library with a glue. Furthermore, we try to incorporate all known techniques such as the way to abstract, process modeling, and efficient CPU scheduling for the performance, which were done by various previous work like rumpkernel [9], Linux kernel library (LKL [15]), mTCP [8].

This paper briefly reviews our previous work at first, then digs the detail of the design and implementation view, with the newly introduced abstraction giving another instance of execution platform (§ 2). Furthermore, we present the detail of applications, Direct Code Execution (DCE) and Network Stack in Userspace (NUSE) (§ 3, § 4), with the simple performance measurement (§ 5).

2. Design of LibOS

The design of the LibOS (*archlib*) consists of three distinct components, 1) Host Backend Layer (virtualization core layer), 2) Kernel layer, and 3) POSIX glue layer. Figure 1 illustrates the overview of the architecture. The benefit of this design, compared to other techniques such as porting-based Linux stack [19], is completely transparent to the implementation of network stack: we do not have to modify that part.

2.1 Host Backend Layer

The purpose of Host Backend layer is to provide an abstraction of underlying platform: right now, the implementation supports two backends, network simulator (ns-3[2]) backend and Linux userspace backend, but thanks to this abstraction, this list can be extended to other platforms like (general) POSIX userspace application, Xen domU, and KVM as NetBSD rumpkernel [9] does. Each backend needs to expose required resources by private functions such as accessing clock source, receiving and sending packets thorough

[1] https://fosdem.org/2015/interviews/2015-antti-kantee/

[2] http://www.nsnam.org/

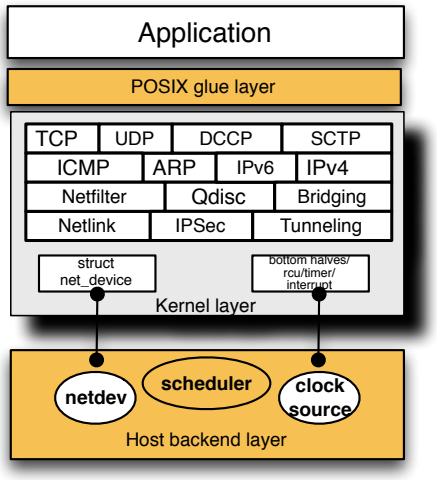


Figure 1: Overview of library operating system architecture. Network stack part such as TCP, IPv4/v6 implementation remains untouched with the surround components above and below parts.

(virtual) network devices, or allocation of memory, scheduling processes, etc. Then the network stack in Linux kernel tree will access these resource via well-defined interfaces like `kmalloc()`, *jiffies*, and *struct net_device*.

2.2 Kernel layer

Kernel layer sits in the middle of POSIX and host backend layer. It also contains unmodified network protocol implementation as shown in figure 1. The bottom part connected to obtain and access the required resource provided by host backend. Each resource required by upper code like memory allocation, clock source, or (virtual) network interface is exposed transparently via a Host backend.

All the kernel context primitives such as interrupt, bottom halves, timer, thread (e.g., workqueue) are carefully reimplemented by using these exposed resources, resulting other code remains untouched.

The configurations of LibOS are done by existing utilities such as `iproute2` or `iptables`, thanks to the system call hijack achieved by POSIX glue layer (detailed later).

All of the wrapped code is implemented as a hardware independent architecture, called `lib`, in order to avoid a bunch of *#ifdef* clauses in the main body of network stack, which is a sort of disaster (i.e., hard to track the frequent updates of upstream codebase).

2.3 POSIX glue layer

Upon the top of Kernel layer, POSIX layer lays to bridge our LibOS kernel and applications in order to provide transparent interface to applications. Theoretically, any kind of existing applications can execute with LibOS by indirections of system calls. But in practice, it depends on the coverage of POSIX API. For instance, our current design concentrates on the socket API and related system calls so, some applications fall into a failure of missing indirections.

The destination of system calls is dispatched by this glue layer: if a call attempts to use a resource managed by LibOS, the system call is *hijacked* and redirected into LibOS. Otherwise, the call reaches the underlying platform (i.e., calls defined by glibc). For example, `gettimeofday(2)` system call returns a different value based on different clock source managed by underlying platform, while `chmod(2)` uses the host system call since LibOS does not need to intercept it.

3. Implementation

The implementation of each component presented in the previous section is slightly different between each Host backend layer. We are going to present our two backends, Direct Code Execution (DCE) and Network Stack in Userspace (NUSE) with the detail of each differences and common parts.

3.1 Direct Code Execution (DCE)

Direct Code Execution (DCE)[3] is a way to reuse network protocol implementations of Linux kernel on top of the `ns-3` network simulator. The project itself started for the research purpose, which is investigation of a protocol behavior with a reproducible environment, but also presented a useful toolset for the protocol development such as distributed debug in a userspace with a flexible network configuration, deterministic regression testing, etc. The detail descriptions and figure of overall architecture are available in our previous manuscript [20] so, we try to highlight only unique part as a Host backend layer.

Memory management

Since DCE gives a multiple-node (instance) environment of network stack in a single host process (by `dlmopen(3)` call), the Host backend layer needs to take care of memory allocation including heap and stack management of each (simulated) process. This is done by the network simulator part with its own allocator implementation in order to support `fork(2)` and `exec(2)` system calls: the memory block is saved and restored upon the context switches in order to avoid a conflict of global symbols.

Clock source

A clock source is derived from the clock managed by network simulator: it is a clock based on a discrete sequence of event in time, resulting a deterministic scheduler with arbitrary kind of tasks. Timestamp in kernel (i.e., *jiffies*) is accordingly updated on an entry and exit of LibOS space and any time related function in kernel space like system calls, timer, or interrupts refers this value as the source.

Network devices

Network devices with DCE cover a variety of virtual devices implemented in `ns-3` such as Ethernet-like device, Wi-Fi, LTE, WiMAX, etc. All these devices appear as a generic Ethernet device on the network stack side. Various configuration utilities such as `ethtool`, `iw` are not available as-is but might be able to interact with a particular bridge implementation between tools and devices.

Process and interrupt primitives

The execution contexts used in the original kernel code are

[3] `http://www.nsnam.org/overview/projects/direct-code-execution/`

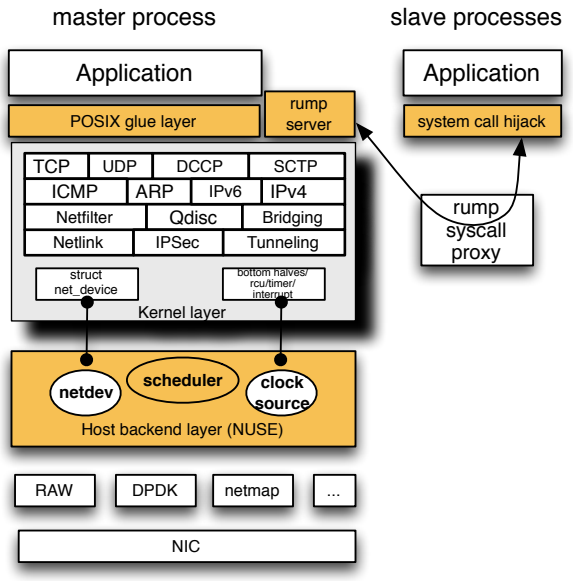


Figure 2: Overview of NUSE architecture. Multiple processes interact via system call proxy implemented by rumpkernel.

carefully synchronized to the one of the Host backend layer (i.e., network simulator). On the Host backend side, the context is implemented with POSIX thread based as well as ucontext [6] based primitive as an option.

Note that even the context in the network simulator side is implemented to be executed parallel, all of events are serialized in the scheduler queue, which means there is no race condition during the execution of kernel code[4].

Process communication

A process abstraction in this Host backend is implemented at the network simulator side: the *struct task_struct* simply refers and synchronizes the simulated process. Channels for the inter process communication such as pipe and UNIX domain socket are directed to the one of host operating system.

3.2 Network Stack in Userspace (NUSE)

Network Stack in Userspace (NUSE)[5] exposes the Linux kernel network stack to userspace in order to achieve the network stack personalization. The basic structure is built upon the LibOS with the Linux operating system as a Host backend layer, and a dedicated POSIX glue codes to redirect system calls. Figure 2 illustrates the high-level overview of NUSE.

We also highlight the Host backend layer in this section to grasp the difference of this application.

Memory management

Contrary to DCE, the memory management for NUSE does not have to take special care since processes running on NUSE always runs a single process with a single memory address space. Thus, the memory resource is simply obtained from host userspace memory allocator provided by standard library (e.g., glibc): all of memory management functions in the original kernel code (e.g., kmalloc) are redirected to the library call (e.g., malloc).

Clock source

The current implementation of clock access for NUSE lazily uses the system call provided by host operating system: we use `clock_gettime(2)` system call to synchronized *jiffies* variable to refer from timer related functions (e.g., add_timer).

Network devices

Even if we are using *struct net_device* and all the upper code for our network stack, NUSE requires the dedicated entry/exit interface to exchange packets with the outside of NUSE since it totally bypasses the kernel network stack. No queueing are triggered to the physical NIC. Thus we implemented virtual network interface, as most of userspace approaches do, with various ways ranging from raw socket based one, tun/tap devices, `pipe(2)` based, Intel DPDK [3], and netmap [17]. Any of channels can be implemented with this interface if required.

Process and interrupt primitives

The context primitives are also reimplemented with the POSIX thread API (pthread) in the Host backend layer as DCE does. Unlike DCE implementation of context primitives in which all the events are serialized, the concurrent execution of multiple threads happens as usual and requires avoiding the race condition between them. The current implementation of NUSE lazily blocks all the thread with `sched_setaffinity(2)` if there is another schedule context running on.

Process communication

Since the NUSE host backend embeds a network stack into a single process, other processes have no access neither to the process nor the network stack. As a result, we cannot benefit to reuse standard configuration tools such as `iproute2` and `iptables` as DCE does.

Our design choice for supporting multiple processes with NUSE is, also, reusing an existing idea of remote communication model designed and implemented by rumpkernel [9]. The right part of figure 2 depicts the relationship between a hosted process on NUSE (i.e., master process) while slave processes communicate with the master via rump system call proxy and eventually arrive at the system call in NUSE. We simply implemented its hypercall implementations in order to provide system call interface to external processes.

3.3 How it works

The initialization of LibOS is triggered by calling `lib_init()` function, initiated by `dlmopen(3)` call or a constructor section of ELF binary. It configures Host backend layer at first, then iterates initialization vector required by kernel space (i.e., *init_call* vector). Since it is a subset of boot procedure of conventional Linux kernel, the initialization finishes immediately (though we do not have any measurement result).

[4]There is an experimental implementation of parallel event scheduling in ns-3 for speedup, but not working at the time of writing.

[5]`http://libos-nuse.github.io/`

4. Applications

Since the LibOS is a library, which should be neutral as much as possible, it does not have a strong claim of its position as the use cases are defined by applications. But we highlight several use cases of LibOS in this section with our current experience during the development.

4.1 Network Stack Personalization

Network stack personalization, a.k.a., ad-hoc network stack, is a primary motivation of LibOS development: deploying a new feature without involving a network stack running upon host operating system makes easier investigation to determine whether the implementation works well or not.

This use case is of course motivated by Filesystem in Userspace (FUSE)[6] where NUSE focuses on the network stack.

4.2 Combination with Any Kernel Bypass Technologies

Network stack personalization also plays well together with kernel bypass technology such as high-speed packet I/O mechanism achieved by Intel DPDK [3] or netmap [17]. There are a bunch of dedicated network stack implementations for the kernel-bypass technology (mTCP [8], lwIP [4], ipaugenblick [19], dpdk-tcpipstack [16]), but all of them are implemented from scratch, or ported with a snapshot of a particular version of network stack, and thus lose an important property of *interoperability*, which has been grown since a couple of decades. Without implementing a network stack from scratch, NUSE allows us to reuse a matured network stack tested and operated for a long time.

4.3 Testing platform

Examples of debugging facilities like (single) gdb and valgrind debugging session with multiple nodes, and code coverage measurement with a flexible network configuration are presented in our previous papers [21][2].

As a result of such debugging and development facilities, it is trivial to implement a continuous integration (CI) platform for Linux network stack development, which allows a whole development community to provide a stable code base tested in every hackers' commit.

Figure 3 presents a screenshot of Jenkins CI[7] web with the code coverage measurement, and regression test implemented in fine-grained network topology configurations. We have been operated a nightly regression test with the latest `net-next` tree, and found a couple of regressions over the past few years. What we learnt for a recurred pattern of regression, as usual software, is a lack of users, obviously in some network subsystems like Mobile IPv6, as well as the existence of untested code path which a committer was not aware of it[8].

[6] `http://fuse.sourceforge.net/`

[7] `http://jenkins-ci.org/`

[8] Our nightly tested environment to the Linux net-next tree with Jenkins CI is available at `http://ns-3-dce.cloud.wide.ad.jp/jenkins/job/daily-net-next-sim/`.

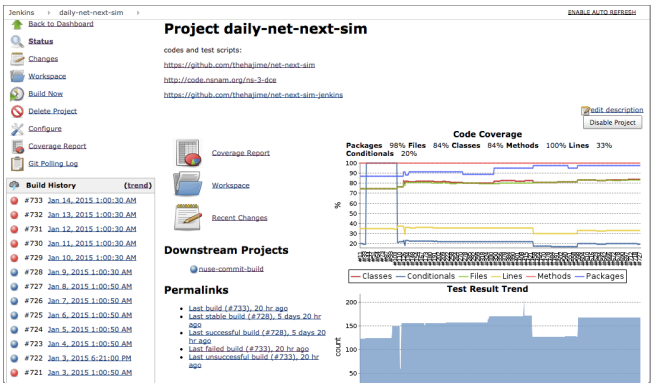


Figure 3: A typical configuration of continuous integration by Jenkins.

5. Micro-benchmark

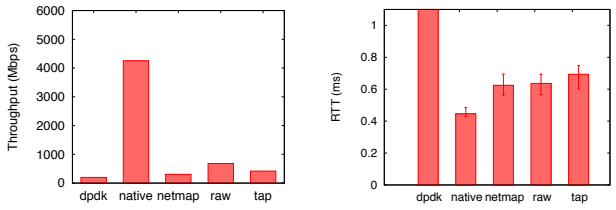
Although the performance is not a primary target of NUSE development, we conducted a simple micro-benchmark to reveal the potential of our network stack. This is an encouragement of the code improvement to every developer, where there are available technologies for high-performance userspace network stack [8][13]. Note that since the micro-benchmark for DCE was introduced in the other paper [20], we do not present here.

To measure the packet forwarding performance of NUSE, we put NUSE on top of a Linux box between two Linux boxes, and injected test traffic through NUSE. In benchmark tests for routing performance, one Linux box sent test traffic or ICMP request (`ping`), NUSE process on relay box routed and forwarded them, and another Linux box received the traffic and sent back ICMP reply. In benchmark tests for transmitting performance of NUSE, the NUSE process sent test traffic or ICMP request to a Linux box. The specification of Linux boxes on this benchmark is summarized in Table 1. In addition, all connections were 10 gigabit Ethernet with direct attached cables without any switches and routers in between boxes. In throughput tests, `flowgen`[9] was used to generate test traffic on the Linux box, and `vnstat` was used to count received packets on the other Linux box. The test traffic was one UDP flow consisting of 1024-bytes packets that means all packets had same source and destination addresses and same source and destination port numbers. All throughput results are average of ten seconds on each permutation. Moreover, the `ping` command was used to measure round trip time as routing and transmitting delays in delay tests.

Table 1: The specification of machines on the micro-benchmark.

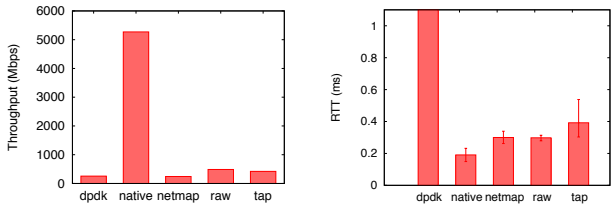
	Linux Boxes	NUSE Box
CPU	Xeon L3426 1.87GHz	Xeon E5-2650 2.60GHz
Memory	4GB	32GB
NIC	Intel X520 82599ES	
OS	Ubuntu 14.04 kernel 3.13.0-32-generic	

[9] `https://github.com/upa/flowgen`



(a) Throughput (b) Delay

Figure 4: The results of benchmark for routing performance of NUSE.



(a) Throughput (b) Delay

Figure 5: The results of benchmark for TX performance of NUSE.

Figures 4(a) and 4(b) show the results of benchmark for the routing performance, and 5(a) and 5(b) show the results of benchmark for the transmitting performance. The "native" label on the x-axis indicates the performance when using native Linux kernel to forward packets. Other items indicate network I/O mechanisms of NUSE, where `dpdk` is Intel DPDK [3] with version 1.7.1, `netmap` is netmap [17] with API version 11, `raw` is the result with raw socket-based network I/O, and `tap` indicates using tun/tap driver. When using tap driver as a network I/O, Linux kernel bridge was used to switch packets from a tap interface attached to NUSE process to a physical network interface. Other mechanisms transfer packets from the NUSE process to physical network interfaces directly.

Raw socket based I/O achieves the best performance with the NUSE. Contrary to the expectation, the performance using DPDK or netmap is worse than raw socket and tap based I/O. The reason of this poor performance is probably due to the current implementation of the glue between *struct net_device* on NUSE and host backend. The current implementation of virtual network interface for NUSE process does not support API for packet batching and reducing lock. As a result, current NUSE cannot benefit from of recent high-speed packet I/O mechanisms. However, to burst packets from *struct net_device* on NUSE process to physical interfaces, adapting a dedicated batch mechanism such as `xmit_more` API might improve the performance.

6. Alternatives

User-mode Linux (UML) is a way to execute the Linux kernel completely on a userspace. The design and implementation are close to LibOS but UML fully virtualizes the behavior of an operating system for the transparency, while LibOS concentrates on a relevant part of an operating system. Moreover, a single-process model taken by DCE (an instance of Host backend) totally differs from UML to ease debug and obtain controllability.

There are considerably a large number of userspace network stacks: Alpine [5] and nfsim [18] aimed to provide an automated test framework for kernel network stacks, with reproducibility by using their own clocks. Nowadays there are many github projects working on userspace stacks: ipaugenblick [19], DPDK-tcpipstack[16] and mTCP[8] for instance. OpenOnload [7] provides a userspace network stack for dedicated NIC, achieving independent performance improvement out of host network stack. All of them stick on the specific version of Linux kernel code or implement a network stack from scratch. This is not maintainable in a long term in our humble opinion, resulting less chance to be broader deployment.

Containers (LXC [1], OpenVZ [14], VServer [11]) are very lightweight technologies for a virtualization and provides a good isolation with a lot of recent development effort, but the guest OS restricts to use the same kernel of host OS, resulting lack of personalization to the network stack.

Recent new direction in operating systems is going to the different shape of kernel: Mirage [12] created a thin library OS for Xen platform implemented from scratch in OCaml. OSv [10] takes a similar approach to port various operating system code into a library, which is executable on a hypervisor. rumpkernel [9], which is already integrated in the NetBSD kernel, extends the idea to the filesystem code as well as the network stack. We think rumpkernel is the most similar approach to us in terms of design choice as well as implementation.

7. Summary

The concept of library operating system itself is not new but Linux does not benefit from LibOS in a number of ways, network stack personalization for each application, combination with kernel bypass technology. Testing and development environment were highlighted as a killer use case of User-Mode Linux (UML) but single-process model of multiple VM instances with Direct Code Execution allows us more controllability which makes more flexible configuration of testing and debugging.

Future work includes to improve more generality of the code to support more applications, performance improvement by adapting known techniques such as packet batching, connection locality, efficient processors distribution, etc. These require a large effort to develop so, we are happy to invite all of you who are willing to join the project.

Our implementation is available at `https://github.com/libos-nuse/net-next-nuse`.

8. Acknowledgments

The authors thank Shrijeet Mukherjee and Jamal Hadi Salim, our shepherd, for the guidance in improving the abstract and the paper. We also thank Thierry Turletti for his valuable comments on the paper. This work is partially supported

by "Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures" in Japan. This research has been supported by the Strategic International Collaborative R&D Promotion Project of the Ministry of Internal Affairs and Communication, Japan, and by the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement No. 608533 (NECOMA). The opinions expressed in this paper are those of the authors and do not necessarily reflect the views of the Ministry of Internal Affairs and Communications, Japan, or of the European Commission.

References

1. Sukadev Bhattiprolu, Eric W. Biederman, Serge Hallyn, and Daniel Lezcano, *Virtual servers and checkpoint/restart in mainstream linux*, ACM SIGOPS Operating Systems Review **42** (July 2008), no. 5, 104–113.
2. Daniel Câmara, Hajime Tazaki, Emilio Mancini, Mathieu Lacage, Thierry Turletti, and Walid Dabbous, *DCE: Test the real code of your protocols and applications over simulated networks*, IEEE Communications Magazine **52** (March 2014), no. 3, 104–110.
3. Intel Corporation, *Intel Data Plane Development Kit (Intel DPDK) Programmer's Guide, Aug 2013. Reference Number: 326003-003.*
4. Adam Dunkels, *Design and implementation of the lwip tcp/ip stack*, Swedish Institute of Computer Science **2** (2001), 77.
5. David Ely et al., *Alpine: a user-level infrastructure for network protocol development*, Usits, 2001, pp. 1–13.
6. Free Software Foundation, *System V contexts: Complete context control a la System V*. (Accessed January 25th 2013).
7. Solarflare Communications Inc., *OpenOnload*, `http://www.openonload.org/`. (Accessed 14th January 2015).
8. EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park, *mtcp: a highly scalable user-level tcp stack for multicore systems*, 11th usenix symposium on networked systems design and implementation (nsdi 14), April 2014, pp. 489–502.
9. Antti Kantee and Justin Cormack, *Rump Kernels: No OS? No Problem!*, USENIX ;login: **39** (2014), no. 5, 11–17.
10. Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov, *Osv—optimizing the operating system for virtual machines*, 2014 usenix annual technical conference (usenix atc 14), June 2014, pp. 61–72.
11. Linux-VServer.org., *Linux-VServer*, `http://linux-vserver.org/`. (Accessed 9th January 2015).
12. Anil Madhavapeddy and David J. Scott, *Unikernels: Rise of the virtual library operating system*, Queue **11** (December 2013), no. 11, 30:30–30:44.
13. Ilias Marinos, Robert N.M. Watson, and Mark Handley, *Network stack specialization for performance*, Proceedings of the 2014 acm conference on sigcomm, 2014, pp. 175–186.
14. Openvz.org., *OpenVZ Linux Containers.*, `http://openvz.org/Main_Page`. (Accessed 9th January 2015).
15. Octavian Purdila, Lucian Adrian Grijincu, and Nicolae Roedunet International Conference RoEduNet 2010 9th Tapus, *LKL: The Linux kernel library*, Roedunet international conference (roedunet), 2010 9th, 2010, pp. 328–333.
16. rajnesh raturi, *dpdk-tcpipstack*, `https://github.com/rajneshrat/dpdk-tcpipstack`. (Accessed 14th January 2015).
17. L. Rizzo, *Revisiting network I/O APIs: the netmap framework*, Communications of the ACM **55** (2012), no. 3, 45–51.
18. R. Russell and J. Kerr, *nfsim: Untested code is buggy code*, Linux symposium, 2005.
19. Vadim Suraev, *Linux TCP/IP stack port for DPDK*, `https://github.com/vadimsu/ipaugenblick/`. (Accessed 14th January 2015).
20. Hajime Tazaki, Frédéric Uarbani, Emilio Mancini, Mathieu Lacage, Daniel Camara, Thierry Turletti, and Walid Dabbous, *Direct Code Execution: Revisiting Library OS Architecture for Reproducible Network Experiments*, Proceedings of acm conext 2013, December 2013, pp. 217–228.
21. Hajime Tazaki, Frédéric Urbani, and Thierry Turletti, *DCE Cradle: Simulate Network Protocols with Real Stacks for Better Realism*, Proceedings of workshop on ns-3 (wns3) 2013, March 2013, pp. 153–158.