Direct Code Execution: Revisiting Library OS Architecture for Reproducible Network Experiments

Hajime Tazaki, Frédéric Urbani, Emilio Mancini, Mathieu Lacage, Daniel Camara, Thierry Turletti, Walid Dabbous

Direct Code Execution: Revisiting Library OS Architecture for Reproducible Network Experiments

Hajime Tazaki*, Frédéric Urbani°, Emilio Mancini°, Mathieu Lacage†,
Daniel Câmara°, Thierry Turletti°, Walid Dabbous°
*University of Tokyo, Japan   †ALCMEON, France   °INRIA, France

ABSTRACT

We describe the first capability, to our knowledge, to execute nearly unmodified applications and Linux kernel code in the context of a widely-used open source discrete event networking simulator (`ns-3`). We propose Direct Code Execution (DCE), a framework that dramatically increases the number of available protocol models and realism available for `ns-3` simulations. DCE meets the goals recently proposed for fully reproducible networking research and runnable papers, with the added benefits of 1) the ability of completely deterministic reproducibility, 2) the scalability that simulation time dilation offers, 3) capabilities supporting automated code coverage analysis, and 4) improved debuggability via execution within a single address space. In this paper, we describe in detail DCE, report on packet processing benchmarks and showcase key features of the framework with different use cases. We reproduce a previously published Multipath TCP (MPTCP) experiment and highlight how code coverage testing can be automated by showing results achieving 55-86% coverage of the MPTCP implementation. Then we demonstrate how network stack debugging can be easily performed and reproduced across a distributed system. Our first benchmarks are promising and we believe this framework can benefit the network community by enabling realistic, reproducible experiments and runnable papers.

Categories and Subject Descriptors

D.4.8 [**Operating Systems**]: Performance—*Simulation*; D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

Keywords

Direct Code Execution; Emulation; Experimentation; Linux; Network Stack; Simulation; Software Development

1. INTRODUCTION

The need for reproducible research in computational sciences has been expressed many times for decades [8, 9, 23]. However, it was not common for networking researchers to reproduce results shown in the literature because of many reasons: not enough details in papers on scenarios, complexity to reproduce the same scenarios, no access to code and scripts, difficulty to reproduce the same in-field conditions, etc. Only recently networking researchers have started to alleviate the problem, by describing in further details their experimentation scenarios, making their code and scripts available to the network community and sometimes by using tools that improve the repeatability of the experiments [29].

Ideally, any researcher should be able to reproduce results shown by her colleague, not only to verify the results published in the paper but also to easily evaluate and debug the protocol on other scenarios with different scales, compare it with other approaches and possibly propose enhancements. In this paper, we define *full reproducibility* as the ability to provide all the above-mentioned requirements. This would lead to more credible and runnable publications [14, 18, 33].

Handigol et al. introduced the following requirements, which are more or less complex to satisfy to ensure experiments reproducibility [14]:

Experimentation realism. This requirement is met when the three following properties are provided: functional realism, i.e., the software implementation of the system under test (SUT) is the same than the one used in the real world; timing realism, i.e., the timing behavior of the SUT is similar to real world; and traffic realism, i.e., the traffic sources used in the experiment reflect the ones from the real world.

Topology flexibility. Experimental environments and input parameters of the SUT should be configurable with fine-grained control so that any network topology is possible to evaluate.

Easy and low cost replication. It should be easy and inexpensive to replicate an experiment.

Container-based Emulation (CBE) with fidelity monitoring (Mininet-HiFi [14]) has been demonstrated as an efficient tool to address the above requirements. However, this approach has two main concerns[1]. First, it mandates that

---

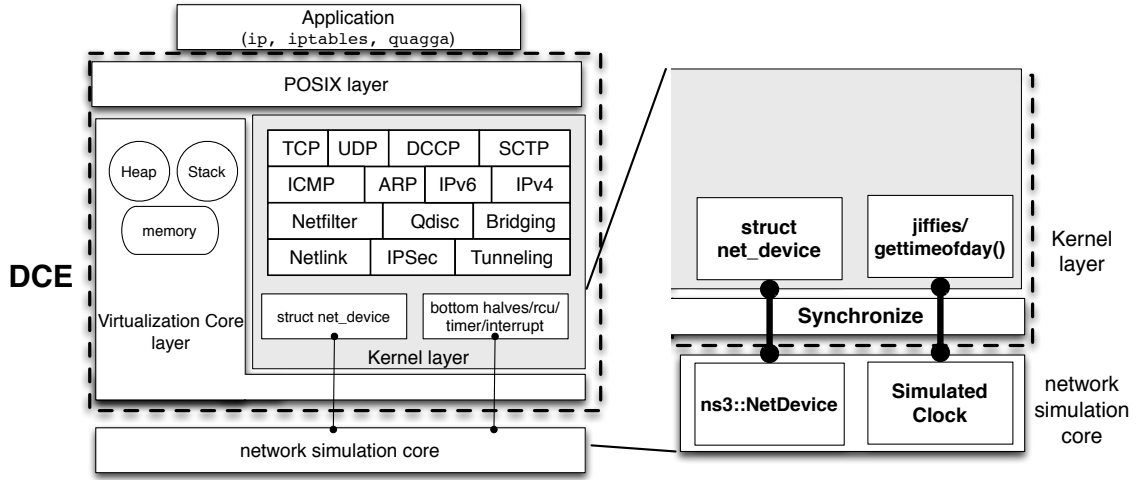[1]A more detailed description of Mininet-HiFi is provided in Section 6.

Figure 1: Architecture of Direct Code Execution. Kernel network devices and timers are synchronized with simulated NetDevice and clock.

enough computing resources are available to run the scenario in real time and requires to monitor the CPU load of the emulation machine to ensure that performance results are meaningful. This constraint significantly restricts the range of possible experimentation scenarios that can be evaluated. Second, since each experimental node runs in a distributed way with CBE, identifying and debugging implementation issues of the SUT is a painful task because there is no integrated control of the software execution.

Therefore, we argue that it is important to satisfy the two following requirements in addition to the aforementioned:

Experimentation scalability. The range of possible experimentation scenarios should not be limited by the resources of the machine that run the experiments.

Easy debugging. It should be easy to identify possible issues in the SUT and debug them, in particular in presence of a distributed system running on multiple nodes.

In this paper, we aim to satisfy all of the five abovementioned requirements by proposing Direct Code Execution (**DCE**), a framework that enables *fully reproducible* network experimentation. DCE takes the traditional library operating system (LibOS) approach such as Exokernel [19] in its core architectural design to enable running and evaluating real network protocol implementations. Since DCE uses a single-process model as a virtualization primitive, the amount of glue code is relatively higher than others (as detailed in Section 2.4). However, tightly integrated design with the `ns-3` discrete-event network simulator benefits from a rich network environment allowing *fully reproducible* experiments.

Our contributions in this paper include:

- The design and implementation of Direct Code Execution[2], a framework that enables realistic and reproducible network experiments at large scale with debugging facilities by integrating real Linux kernel and application code with the `ns-3` network simulator.

---
[2]DCE is available at `http://code.nsnam.org/ns-3-dce`.

- Packet processing benchmarks to analyze its performance and comparison with the Mininet-HiFi CBE approach.

- Reproducible network experiments with different use cases that demonstrate the benefits of DCE.

The paper is organized as follows: we present the design and implementation of DCE, our proposed framework, in Section 2. Then we present micro-benchmarks obtained with DCE and Mininet-HiFi in Section 3, and showcase features of DCE with different use cases in Section 4. We discuss some future research directions in Section 5. Finally, in Section 6, we review the prior work done to enable reproducible network experiments and conclude the paper.

2. DCE ARCHITECTURE

The design of DCE takes its core idea from the library operating system (LibOS) architecture [19] to satisfy the requirements for reproducible network experimentation. DCE is structured around three separate components as depicted in Figure 1. First, the lowest-level *core* module handles the virtualization of stacks, heaps, and global memory. Second, the kernel layer takes advantage of these services to provide an execution environment to the Linux network stack within the network simulator. Third, the POSIX layer builds upon the *core* and *kernel* layers to re-implement the standard socket APIs used by emulated applications.

2.1 Virtualization Core Layer

Contrary to other user space virtualization environments such as UML [10], DCE executes every simulated process within the same host process. This single-process model makes it possible to synchronize and schedule each simulated process in turn from the simulator event loop without having to use inter-process synchronization mechanisms. Moreover, it allows users to trace the behavior of an experiment across multiple simulated processes without the need of a distributed and usually complex debugger.

Table 1: Supported environment of fast custom ELF loader.

Version	i386 arch	x86_64 arch
Ubuntu 10.04	✓	✓
Ubuntu 11.04	✓	✓
Ubuntu 12.04		✓
Ubuntu 13.04		✓
Fedora 14	✓	✓
Fedora 15	✓	✓
Fedora 16		✓

One of the downsides of this single-process design is that we cannot rely on the host operating system to release the resources associated with each simulated process. So, it is necessary to carefully track each resource allocated by each process to handle gracefully their termination within a long-running simulation.

For instance, by default, we manage the stack area and the Program Counter of each simulated process by creating, switching to/from and destroying a host-level thread as necessary. Threads in each simulated process are managed by our task scheduler with the synchronization in simulated host and isolated from the other simulated hosts' threads. This allows the host-level debugger to automatically gain knowledge about the location of our simulated stacks through the list of threads, thus ensuring very reliable backtraces during debugging sessions. Optionally, we provide a more efficient *ucontext*-based [3] stack manager to allocate stack space with `mmap` and control the Program Counter of each simulated process by saving and restoring CPU registers entirely in user-space.

Similarly, we take great care to track and isolate the heap of each simulated process. In particular, we allocate each heap within large `mmap`ed blocks that can easily be reclaimed as needed and then slice each of these memory blocks with a Kingsley [22] allocator to implement the `malloc` and `free` functions for simulated applications.

The most challenging aspect of the single-process model, though, is the virtualization of the global memory. Since the objective of the host program loader is to ensure that every process contains no more than one instance of each global variable, we provide a specific loading mechanism to instantiate once the same global variables for each simulated instance.

To do so, each simulated process lazily saves and restores upon context switches its private copy of the global variables to/from the shared data section which was setup by the host ELF loader upon program loading.

Optionally, on a few host systems (Table 1), we provide a replacement ELF loader that is able to avoid these data copies upon context switches by directly allocating a new pair of code and data sections for each instance of the same simulated process. This improves the memory usage of typical experiments very marginally but runtime often improves by a factor of up to 10 [24].

Table 2: The number of POSIX API functions supported in DCE over time.

Date	# functions
2009-09-04	136
2010-03-10	171
2011-05-20	232
2012-01-05	360
2013-04-09	404

2.2 Kernel Layer

On top of these core virtualization primitives, the Kernel layer embeds the network protocol implementations found in the Linux kernel. This layer communicates with `ns-3` through two well-defined interfaces. At the bottom of the Linux network stack, MAC-level network packets enter and leave the kernel through a fake `struct net_device` that communicates directly with the `ns-3` C++ equivalent, `ns3::NetDevice`. At the top of the network stack, application-level payload is exchanged with socket-based applications through the kernel-level socket data structures.

Since most of the network stack configuration happens through netlink sockets, users can benefit from the standard Linux user space command-line tools (`ip`, `iptables`) to set up the necessary IP-level configuration (IP addresses, forwarding tables, firewalls, etc.). Other parameters that are only accessible through the `sysctl` filesystem can also be controlled by specifying path/value pairs. Each pair is set automatically by accessing the `sysctl` tree of static configuration variables.

Because we have tracked the most recent version of the Linux kernel for almost five years, the support code which provides the execution environment for this Kernel layer has been designed to be robust to kernel version changes. First, we minimized the number of changes to existing kernel code (20 lines across 2 files) and we targeted these changes to files that are stable. Second, we took advantage of the Linux platform abstraction API by integrating the remainder of the support code as a new independent architecture.

2.3 POSIX Layer

Given the significant size of the POSIX specifications, our POSIX implementation used to replace the traditional `glibc` has been developed in an incremental way to provide support for the subset[3] of features used by the applications we already tested over DCE. As such, although we do not provide full API coverage, we have made steady progress (see Table 2) towards being able to run most C-based applications of interest out of the box.

Most API implementations are trivial pass-thru to the corresponding function in the host C library except for those which access kernel-level resources. In particular, time-related functions (e.g., `gettimeofday(2)`) return simulation time instead of the wall clock time; signals are checked upon re-

---

[3]The detailed list of supported functions is available at URL `http://www.nsnam.org/docs/dce/release/1.0/manual/html/dce-user-tech.html`.

turn from every interruptible function; local files are open relative to a node-specific filesystem root to ensure that two different node instances see different data and configuration files, etc.

The new socket implementation is similarly un-eventful since it merely acts as a straightforward translator layer between the application and either kernel sockets from the Kernel module or `ns-3` sockets that provide access to the `ns-3` TCP/IP stack.

One of the most challenging components of this POSIX implementation is the support for the `fork()` function. Traditionally, single address space-based POSIX implementations provide only the `vfork()` function because it is not easy to make two processes, sharing the same address space, see different values at the same memory location. By contrast, DCE supports the two above functions to facilitate the integration of a larger set of applications. This feature is implemented by tracking which memory locations are shared by which processes and by lazily saving and restoring these shared locations upon context switches.

2.4 Discussion

The LibOS approach that encapsulates kernel network stacks into a user space library, combined with the single-process model used in the DCE virtualization core, offers broad capability in code inspection. As a result, our design provides a complete solution for reproducibility, debuggability (through controllability), and experimentation scalability to network experiments.

The essential strengths of the LibOS approach are characterized as follows. First, it uses minimized virtualization, which allows to run multiple instances of a node on a single-process to obtain the controllability of experimental entities, as well as fast execution by simplifying the set up and by removing the need for (virtual) hardware initialization, filesystem checking and mounting. This optimization is particularly important when conducting a large number of experiments involving many instances of a node, as they usually take a large amount of time to initialize and execute. This has a significant impact on the easy and low cost replication requirement and represents a clear plus for the DCE framework. Second, the integration with the `ns-3` network simulator enables to obtain a deterministic network stack behavior, which is required for reproducible experimentation. This determinism also benefits the experimental scalability, since experimental scenarios are not bound to the real-time constraint of available resources. Moreover, it offers a widely configurable network environment for testing purpose, which is useful to analyze previous work or network experiments using different parameters space to the System Under Test. Finally the single-process model used in DCE facilitates debugging across distributed nodes.

However, DCE suffers from several limitations. First, the use of virtual clock prevents possible interactions with the external world of `ns-3` such as real routers in the Internet. Second, contrary to lower-level CPU virtualization technologies, DCE requires API-specific glue code for its POSIX and kernel support. New protocol implementations that at-tempt to use previously un-implemented APIs need extra work. In practice, though, as our coverage of the POSIX API increases, the probability of needing a missing function decreases. Our experience leads us to believe that we have reached sufficient coverage for a wide variety of applications. Finally, in very rare cases, protocol implementations that execute busy loops require modifications to behave correctly within DCE.

One of the concerns of using virtual clock is the timing accuracy with respect to real environments. A previous study [33] presents a high-level analysis of the TCP performance between DCE-based Linux network stack and real Linux environment, and the result shows some differences between them (25-30% low goodput of DCE), but also gives similarity between them.

Although the single-process model used in DCE brings key features, it is not able to scale with very large simulation scenarios, involving millions of nodes. In such a case, a solution is to use Message Passing Interface (MPI)-based distributed simulation as a built-in service of `ns-3` [31]. This gives the opportunity to add experimental physical machines when necessary, although it breaks the DCE single-process model, and so, makes debugging more complex.

2.5 Integrating a New Protocol

While the details[4] of how a new protocol can be integrated within DCE vary a lot, we have observed a few recurring patterns over the past few years. In general, most of the work is relatively minor: it ranges from adding a few pass-through functions for newly-used POSIX functions to adding to the kernel build the files that contain newly-used generic functions. In some cases, extra work might be required to add support for new functionality, especially for POSIX-based protocol implementations. For example, when a new protocol uses a thread synchronization primitive that we do not support yet.

Kernel-based protocols rarely create problems in and of themselves. However, when these protocol implementations are based on ancient versions of the kernel, it can be sometimes necessary to first port them to a more recent Linux kernel before they can be used in DCE.

In the following sections, we evaluate the performance of DCE and compare our approach with other techniques.

3. PERFORMANCE EVALUATION

To investigate the performance overhead of packet processing that is consumed during transmitting and forwarding at simulated nodes, we measure the maximal packet processing rate that the host machine is able to process. The experiments were performed on an Intel Xeon 2.8 GHz with 8 GB of RAM, and the virtualized network stack from the Linux 2.6.36 kernel. We set up a linear daisy chain topology, where nodes are arranged in a line. Every node, apart

---

[4]See the DCE manual at `http://www.nsnam.org/docs/dce/manual/html/index.html`. Note also that the `bake` building and integration tool is used to make easier the DCE installation procedure, see `http://www.nsnam.org/docs/bake/tutorial/html/index.html`.

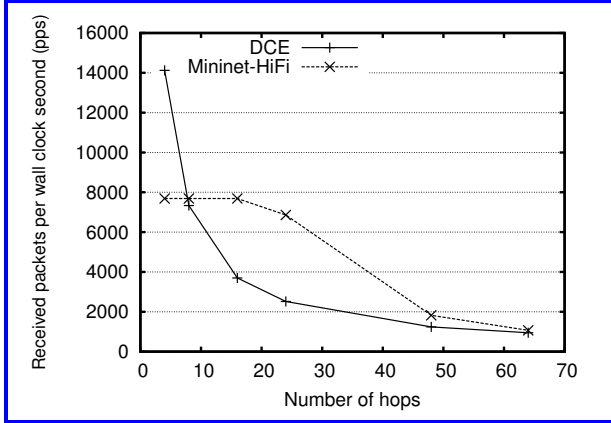Figure 2: **Experimental topology to estimate packet processing rate.**

Figure 3: **Packet processing performance as a function of the number of nodes: Mininet-HiFi has lower overhead than DCE.**

Figure 4: **Sent and received packets in function of the number of hops, when running a client/server CBR UDP session for 50s. Note that there is no packet loss in DCE, while Mininet-HiFi starts losing packets when the number of hops exceeds 16.**

Figure 5: **Wall clock time for different sending rates and number of hops with DCE, running a client/server UDP session for 100s. DCE runs slower or faster than real time depending on the scale of scenario but provides accurate results in all cases.**

from the two nodes at the ends, is connected to two other nodes as depicted in Figure 2. A UDP constant bitrate flow (100 Mbps) is transmitted from the client node to the server node. To avoid congestion issues, the link bandwidth is set to 1 Gbps, higher than the data sending rate.

Emulated network technologies working in real time have their capacity bounded by the host machine processing power and by the size of the emulated topology. DCE, instead, runs in simulated time so it can handle through time dilation all the traffic without data losses, even though the experiment may sometimes take longer to run than the real-time scenario would take. In other words, in DCE only the execution time of the experiment depends on the hardware capacity, while the experiment results are not impacted by the available resources of the machine. Here, we compare DCE with Mininet-HiFi [14], which is one of the most promising real time emulation networks tool available. We used Mininet-HiFi Version 2.0.0 over an Ubuntu machine running Linux 2.6.36 kernel, the same used for the DCE tests.

The performance of DCE and Mininet-HiFi shown in Figure 3 are calculated by counting the number of received packets and dividing it by the elapsed wall clock time of each experiment[5]. Mininet-HiFi experiments run for 50 seconds and DCE experiments run for 50 simulated seconds. The packet size used in both cases is set to 1470 bytes. Even though DCE is able to process faster with a small number of nodes (less than 8), the packet processing rate per wall

---

[5]In all of the experimental results in this paper, you can click on figures/tables in the PDF to get more information about each result and instruction to reproduce it.
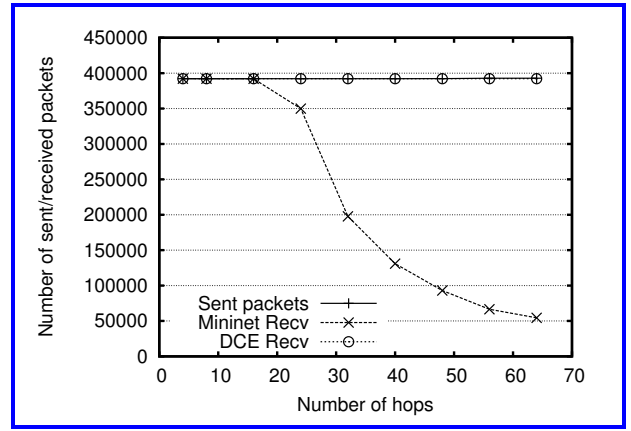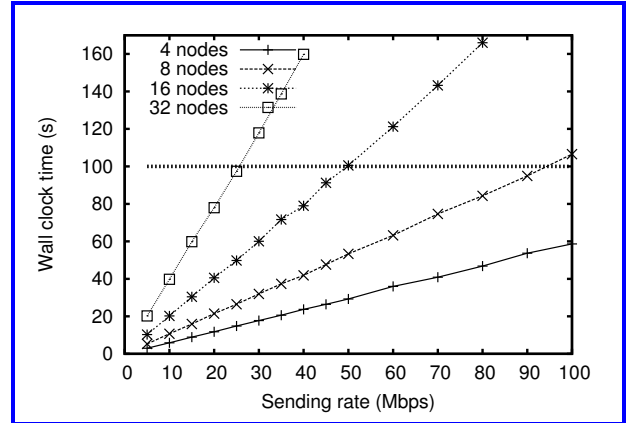
clock time decreases as the number of nodes in the network increases. This indicates Mininet-HiFi has lower overhead than DCE to process packets which means that reproducing experiments with DCE might take longer time than with Mininet-HiFi. This is not, however, the most important performance metric when reproducing network experiments.

In order to illustrate the limit beyond which experimental results are not accurate, we report in Figure 4 the maximum throughput that Mininet-HiFi is able to handle. With the machine used to perform the tests, the upper-bound traffic for Mininet-HiFi remains stable when the number of nodes is less or equal to 16. When the number of nodes exceeds 16, we observe the presence of packet loss and that the packet processing capacity starts to decrease. This behavior is expected because the machine used for emulation has limited
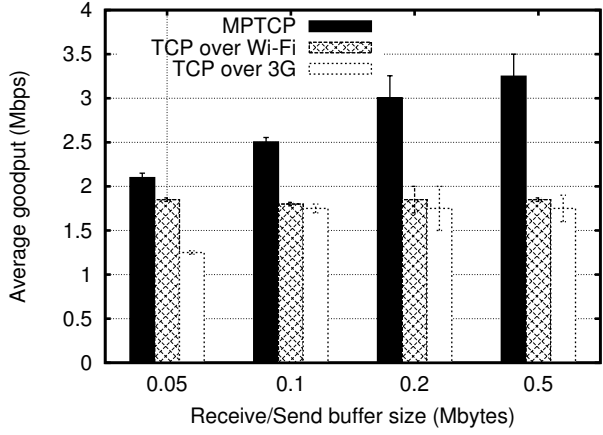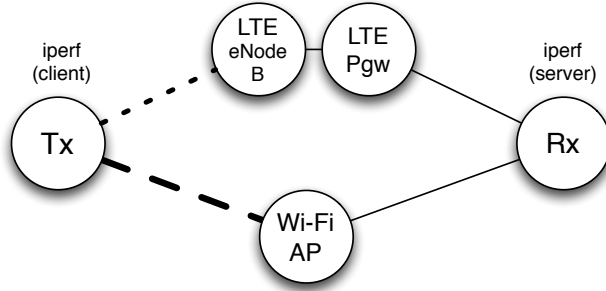
Figure 6: Network setup for the MPTCP experiment (left) and the result of original experiment [30] (right).

available hardware resources which may impact the performance of real time emulation of CPU-greedy experiments.

To analyze the relationship between the execution time and the data rate for different number of nodes with DCE, we sent a constant bitrate for 100 simulated seconds. We ran the experiment for different data rates and number of hops. In particular, in Figure 5, the data rates from 5 to 100 Mbps, and for 4 to 32 hops are reported. For less complex experiment scenarios (with lower number of nodes and/or sending rate), DCE runs the experiments faster than it would take to run in the real world (i.e., 100 seconds). As expected, the measured execution time linearly increases with the amount of traffic handled during the simulation, matching closely their linear regression.

4. DCE USE CASES

This section illustrates key features of the DCE framework with different use cases. First, we replay an existing network experiment found in the literature with the same software over DCE (§ 4.1). Then, we evaluate the degree of configurability of the network environment offered by DCE using the code coverage metric (§ 4.2). Finally, we showcase DCE debugging facilities on distributed nodes (§ 4.3).

4.1 Experimentation Reproducibility

In this section, we evaluate the **reproducibility** of DCE with Linux network stack experimentation by replaying an experiment described in the literature: a Multipath TCP (MPTCP) experiment presented in paper [30]. We present 1) the ability of rough reproducibility of an existing experiment by using the same software, and 2) *full reproducibility* of existing experiments when conducting the experiment with DCE on different versions of the OS.

MPTCP is an extension of the standard TCP allowing to use multiple subflows with different IP addresses without modifying user space applications. Basically, this new transport protocol makes it possible to increase the throughput of an application by running it over multiple links. In this use case, two wireless links (LTE and Wi-Fi) are set up on a simulated host and these two links are simultaneously used
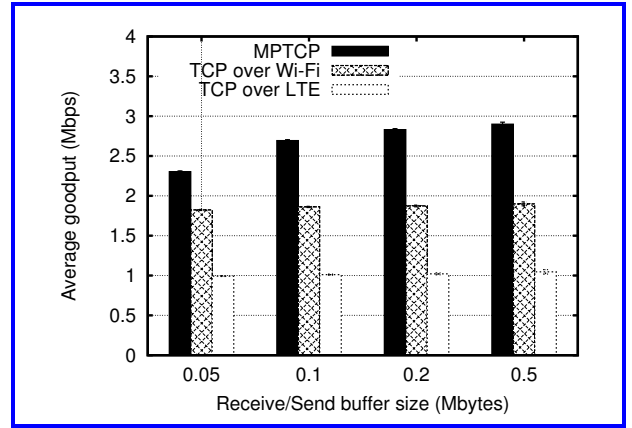

Figure 7: Goodput obtained with MPTCP and TCP over LTE/Wi-Fi using `ns-3` DCE in function of receive/send buffer size.

to increase the application throughput. Note that in the original scenario [30], a 3G connection was used instead of LTE, but we had to replace it with a LTE link of similar characteristics because such a 3G link is not yet available within `ns-3`.

Figure 6 illustrates the network setup for this experiment along with the plot from the original experiment[6]. We created this topology with `ns-3` version 3.17 running over Ubuntu 10.04 x86_64 version. Then we configured DCE to run the MPTCP Linux implementation [5], the `iproute` utility, and `iperf` without any modification to the original implementations[7]. Our approach, based on simulation virtual time, does not require specific machine setup as it is the case in the original experiment since the CPU and memory re-

---

[6]For copyright reasons, we plotted the original data by hand instead of including the original curve from the NSDI paper.
[7]Note that DCE requires a minor modification of `iperf` when using UDP, but this is not the case in this setup as it uses TCP. See `http://www.nsnam.org/docs/dce/release/1.1/manual/html/dce-user-newapps.html#example-dce-with-iperf-dce-iperf` for more information.

Table 3: Measured goodput by different platforms: full reproducible experimental results achieved among different platforms.

Environment	MPTCP (Mbps)	LTE (Mbps)	Wi-Fi (Mbps)
Ubuntu1204-64-Phy	2.67787e+06	1.00382e+06	1.85387e+06
CentOS6.2-64-KVM	2.67787e+06	1.00382e+06	1.85387e+06
Ubuntu1210-64-KVM	2.67787e+06	1.00382e+06	1.85387e+06
Ubuntu1204-64-KVM	2.67787e+06	1.00382e+06	1.85387e+06

Table 4: Code coverage of network tests for the MPTCP implementation, where Lines, Functions and Branches are the ratio of declared and tested lines/functions/branches in the code file, respectively.

	Lines	Functions	Branches
mptcp_ctrl.c	76.3 %	86.7 %	59.9 %
mptcp_input.c	66.9 %	85.0 %	57.9 %
mptcp_ipv4.c	68.0 %	93.3 %	43.8 %
mptcp_ipv6.c	57.4 %	85.0 %	45.2 %
mptcp_ofo_queue.c	91.2 %	100.0 %	89.2 %
mptcp_output.c	71.2 %	91.9 %	58.6 %
mptcp_pm.c	54.2 %	71.4 %	40.5 %
Total	68.0 %	85.9 %	54.8 %

sources will not impact the performance results. The buffer size was configured with the following set of Linux kernel parameters: `.net.ipv4.tcp_rmem`, `.net.ipv4.tcp_wmem`, `.net.core.rmem_max`, and `.net.core.wmem_max`.

Figure 7 reports the average received goodput at the right node (Rx) using a single TCP connection over Wi-Fi, a single TCP connection over LTE and an MPTCP connection, along with the 95% confidence interval computed for 30 replications using different random seeds in function of the receive/send buffer size. Unsurprisingly, we note that the goodput obtained with the DCE-based experiment increases when the buffer size increases as the original paper demonstrated, but the performance differs from the original result in the following ways. First, for the two TCP measurements (over LTE and Wi-Fi), no significant goodput improvement is observed when increasing the buffer size, while the impact is more noticeable on the TCP over 3G link experiment of the original paper. Second, the maximum goodput achieved for MPTCP is ranging from 2.2Mbps to 2.9Mbps while it is ranging from 2Mbps to 3.2Mbps in the original paper. The differences observed could be due to different end-to-end delays between the two experiments, as the round-trip-time can have a significant impact on the throughput performance.

Although we observe some differences with the original performance results when using the same software implementations, DCE could reproduce a similar trend of results for the MPTCP goodput. Note that DCE performance results are similar to the ones reported in [1] with Mininet-HiFi, where authors also noticed that link characteristics have a high impact on goodput performance.

In addition to the above experiment and for proof of concept, we conducted the same simulation with four different environments (Ubuntu 12.04 64bits version on physical ma-

chine, CentOS6.2 64bits, Ubuntu 12.10 64bits, and Ubuntu 12.04 64bits versions over KVM). Performance results obtained (shown in Table 3) are rigorously identical across all the different environments. This *full reproducibility* obtained with deterministic performance results is an important asset while (1) reproducing experiments of other researchers, (2) analyzing the impact of some parameters on the performance of the system or (3) comparing different implementations of the same protocol in the same network conditions.

4.2 Increasing Code Coverage

The second use case aims to demonstrate DCE **flexibility** to configure the network environment parameters of experiments by examining the code coverage of the networking protocol stack under test. Code coverage is usually used as a quality metric of software development to measure how thoroughly test programs exercise with the system under test. However, it is also useful to understand how many parameters an experimental system exposes in a specific network experiment since the number of lines, functions, and branches covered by test programs reflect the number of different inputs (i.e., parameters) injected into the system under test with the support of `ns-3`. Moreover, coverage tests shall be deterministically executed in order to ensure they cover the whole implementation. So, the virtual clock of `ns-3` is helpful in such a case.

For this use case, we used the same MPTCP code as in § 4.1 and wrote four test programs by using `iproute` utility for IPv4 and IPv6 addresses configuration, `quagga` to set up route information, and `iperf` as a traffic generator in the experimental topology. We also added an Ethernet type of link with different packet loss ratio and link delay to induce the behaviors of protocols. Then we ran these test programs

Table 5: Memory check obtained with valgrind on Linux (2.6.36).

	type of error
tcp_input.c:3782	touch uninitialized value
af_key.c:2143	touch uninitialized value

to measure and analyze the code coverage of the MPTCP kernel code by `gcov` tool .

The code coverage results are shown in Table 4. We do not yet cover 100% code of MPTCP, which requires additional effort to write test scenarios and programs[8]. However, the high code coverage (between 55-86 %) has been achieved with a small amount of effort, with about 1K LoC for four test programs in a couple of days in our case including different network topologies, different traffic patterns, as well as randomized values to link errors such as packet corruptions and losses.

This use case demonstrates that DCE allows one to configure various parameters for network experiments in a flexible way with the interactions of publicly available user space applications (e.g., `iperf`, `quagga`, and several Linux command line utilities), and without much effort for writing test programs.

4.3 Easy Debugging

This third use case illustrates the fine-grained **debuggability** feature of DCE by conducting dynamic memory analysis using the `valgrind` tool, and per-node debugging with `gdb`. What makes the use of such analysis tools with DCE straightforward is that this framework encapsulates the network stack into a user space library with a single process.

valgrind: `valgrind` is a dynamic program analysis tool that includes key features for programmers such as memory error detection. DCE jointly used with `valgrind` allows to investigate, in a reproducible environment, memory errors in a network stack implemented within the kernel space. Furthermore, this functionality is also available for programs that run on multiple distributed nodes using a single `valgrind` profiler.

Table 5 reports errors detected by `valgrind` with 2.6.36 version of the Linux kernel running over DCE. Although all tests including IPv4/IPv6 tcp, udp, raw socket, and Mobile IPv6 are passed, we successfully detected two errors related to invalid access of uninitialized memory, which still exist in the latest version of Linux kernel[9].

gdb: The single-process model used in the DCE virtualization core facilitates debugging of network stacks. For example, it is possible with DCE to inspect a problematic state by putting a breakpoint in the code of a specific node. Although many solutions are available to debug distributed processes in a single debugger front-end, DCE has the specificity to provide *full reproducibility* of bugs with determinis-

---

[8]The latest result is available at `http://ns-3-dce.cloud.wide.ad.jp/jenkins/job/daily-mptcp/cobertura`.
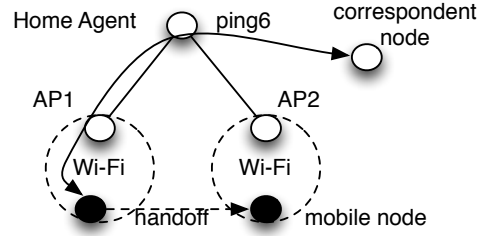[9]Linux 3.9.0 (June 2013)

Figure 8: Scenario to debug node handoff.

```
(gdb) b mip6_mh_filter if dce_debug_nodeid()==0
Breakpoint 1 at 0x7ffff287c569: file net/ipv6/mip6.c, line 88.
 <continue>
(gdb) bt 4
#0  mip6_mh_filter (sk=0x7ffff7f69e10, skb=0x7ffff7cde8b0)
    at net/ipv6/mip6.c:109
#1  0x00007ffff2831418 in ipv6_raw_deliver
    (skb=0x7ffff7cde8b0, nexthdr=135)
    at net/ipv6/raw.c:199
#2  0x00007ffff2831697 in raw6_local_deliver
    (skb=0x7ffff7cde8b0, nexthdr=135)
    at net/ipv6/raw.c:232
#3  0x00007ffff27e6068 in ip6_input_finish (skb=0x7ffff7cde8b0)
    at net/ipv6/ip6_input.c:197
(More stack frames follow...)
```

Figure 9: Call stack backtracking of Linux network stack of Mobile IPv6 code with a break condition.

tic behavior of network protocols, which helps a lot in identifying implementation issues.

To demonstrate the following debugging session, we built the basic network topology illustrated in Figure 8. This scenario simulates a node handoff across two Wi-Fi access points along with signaling messages exchanged to provide mobility transparency through the Mobile IPv6 protocol implementation of the Linux network stack. We used the `umip` [2] user space application for Mobile IPv6 signaling over DCE. Figure 9 shows the debugging session corresponding to the bug detected by the Linux kernel community [35]. Basically, we inspect the state of a specific node when a binding registration message transmitted by the mobile node reaches the Home Agent (HA); we put a breakpoint in node number 0 (corresponding to the HA) in order to analyze the change of state triggered by the mobile node movement.

It is worth noting that the memory analysis done with `valgrind` as well as the sequence obtained with `gdb`, shown in Figure 9, are deterministic. In particular, possible incorrect memory accesses and message transmission times obtained for the `umip` application, for the node movement, and for the handoff duration will remain identical to those obtained in different runs. In this manner, bugs can easily be reproduced, which is a key feature while debugging code, especially in distributed systems. Note that it is possible to add randomness into the simulations, such as packet arrival timings, process scheduling timings, random failure injections in a controlled manner thanks to the `ns-3` pseudo randomizer. In particular, this may help in identifying bugs dependent on specific timings or conditions.

5. FUTURE DIRECTIONS

In this section we present some other uses of DCE along with possible future research directions.

On-demand network stack experimentation over public network testbeds. Numerous research activities concern the design of virtualization primitives specific to network emulation testbeds (NET). One of them focuses on how to enable fine-grained privileged access for users on public testbeds such as PlanetLab [6, 27]. In such a network platform, resources are shared among a bunch of users and are controlled by the testbed operator. In particular, it is necessary to prevent conducting experiments that require network stack modifications on testbed nodes. In such a case, DCE is an interesting approach to consider because it enables to run, at user space, a new network stack without requiring root privileges. For instance, this feature is very useful to study the performance of new network protocols implemented in kernel space over shared testbeds such as PlanetLab.

Lightweight virtualization. A user space network stack is similar to a lightweight virtualization primitive that can be used on resource-constrained environments such as smartphones or tiny sensors. It is likely that the need of lightweight virtualization to introduce new network protocols and architectures over resource-limited nodes, without replacing new kernel network stack, will increase over the next few years.

Foreign OS support. DCE is a self-contained entity in user space. It is able to execute network stacks with multiple versions of Linux kernels[10]. This feature enables to evaluate the impact of different operating systems on the performance of the system under test by just replacing the kernel layer part (§ 2.2) with different operating system, for example, a BSD-based OS on Linux host. This is not possible to perform with CBE approaches [7, 14, 15, 25, 28, 34] because they should share the kernel image between the hosted operating system and the guest emulated host.

6. RELATED WORK

Full virtualization provides generic testing environments by running network software prototypes in VMs but suffers from scalability limitation (due to memory and CPU overhead) and non-reproducibility (due to variability introduced by hypervisor scheduling). This section highlights previous research efforts to enhance the evaluation of network experimentation, in particular on virtualization . We classify hereafter the various approaches and discuss if they meet the requirements identified in Section 1, especially the experimentation reproducibility, scalability and easy debuggability.

Container-based emulation (CBE) (CORE [4], Trellis [7], IMUNES [28], vEmulab [15], Crossbow [34], Mininet [25], Mininet-HiFi [14]) is based on lightweight virtualization technologies enabling a large number of VMs to run on the same emulation machine. This approach allows a wide range of network environments to be fed within the ex-

---

[10]The full list of Linux kernel versions supported by DCE is available at URL `https://github.com/thehajime/net-next-sim/branches`.

periments, without the hurdle of building/maintaining real networks. However, as all containers must run on top of the same kernel, there is no strong separation between the virtual systems. The behavior of the network stacks is therefore not completely reproducible as it depends on the type of virtualized OS scheduler and on the resources available on the emulation machine. Recently, Mininet-HiFi [14] has proposed to alleviate this problem by adding resource isolation to reduce variability and monitoring to estimate the performance fidelity obtained from network experiments. However, performance results obtained are only meaningful and reproducible when the CPU resources of the emulation machine are sufficient to run the experiment in real time, which limits the scale of scenarios that can be evaluated with such an approach.

Time Dilation [13] provides the illusion to an operating system and its applications that time is slower than physical time. Data arriving from a network interface will therefore appear to arrive faster, but the system will experience more cycles per perceived second from the processor and the number of cycles available to each arriving byte remains constant. This approach is proposed mainly to check whether the network is the bottleneck for a given system, providing a low-cost mechanism for determining the potential benefits of higher performance network interconnects before committing an upgrade. The idea is however more general and can be applied to adjust the clocks between network stacks running in the VMs and an underlying emulated network, thus removing the real time operation constraint required to get meaningful results.

SliceTime [36] provides speed adjustment for a real software prototype running in a VM on top of a simulated network topology. Instead of slowing down the time by a constant factor as done by time dilation, a synchronizer controls the execution of the network simulation and the software prototypes and interrupts the execution of the prototype or the simulation at times to achieve precise clock alignment. To enable this suspension, the software prototypes are hosted inside virtual machines for means of control. This approach allows better scalability and reduces variability. Our approach based on inserting real network or application software in the `ns-3` simulator provides implicit synchronization between the software and the simulated network topology as we are only using the simulation time. It also provides 'automatic' time dilation so that a large-scale experiment is run with the minimum slowdown while still providing accurate results.

Time traveling-based virtual machines (TTVM [21]) was proposed to address the difficulties associated with debugging operating systems (as cyclic debugging does not work because of non determinism). By recording enough information to replay a long-term execution of an operating system using ReVirt [11], TTVM enables a programmer to navigate backward and forward arbitrarily through the execution history of a particular run and to replay arbitrary segments of the past execution, even in the presence of non determinism. This feature can be exploited to provide timing realism in networking experiments even when they are

Table 6: Reproducible network experimental tools and their pros/cons.

	Functional realism	Timing realism	Topology Flexibility	Easy replication	Easy debug	Experimentation scalability
Container-based emulation [7] [28] [15] [34] [14] [25] [4]	✓	(only [14])	✓	✓		
Time dilation, traveling [13] [21] [36] [26]	✓	✓		✓	✓	✓
Userspace network stack [16] [12] [32] [20]	✓			✓	✓	✓
Network Simulator Cradle [17]	(limited)	✓	✓	✓	✓	✓
Direct Code Execution (this paper)	✓	✓	✓	✓	✓	✓

not run in real time. However, the bootstrap time required for each OS instance takes a significant amount of time[11], which limits the practicality of this technology to repeat experiments. Simics [26] is a full-system simulator used to run unchanged production binaries of the target hardware at high-performance speeds. Simics has the ability to execute a program in the forward and reverse direction. Reverse execution can illuminate how an exceptional condition or bug occurred.

User-space network stack implementation (Entrapid [16], Rump [20], Alpine [12], nfsim [32]) takes a different approach than DCE to enable easy debug of network protocol stacks. Basically, the network stack kernel code is transformed into a user space library, so that applications can bypass the host network stack in favor of the library. Entrapid [16] reuses BSD 4.4 kernel code and allows multiple instances of the network stack to run in a single process. Rump [20], which is already integrated in the NetBSD kernel, extends the latter approach to filesystem code as well as network stacks. While Entrapid and Rump offer debugging facilities and allow reproducing network experiments when enough resources are available on the machine, the use of wall clock time impedes experimentation reproducibility as it is the case with CBE approaches. Alpine [12] and nfsim [32] aim to enable automatic testing of kernel network stacks. By using their own clocks, reproducibility is provided but the per-process virtualization method implemented using the `LD_PRELOAD`ed library makes complex the debugging of network stacks between different processes.

Network Simulation Cradle (NSC [17]) is the ancestor of DCE [24]. It has been originally developed to provide realistic performance results of existing real-world TCP implementations. NSC parses and transforms different operating system's network stacks (e.g., FreeBSD, Linux, OpenBSD, lwip) into new C files compiled and linked with shared libraries used in network simulators. Both NSC and DCE use the network simulator's virtual time and facilities to provide a wide range of network environments. However, the use of NSC is limited to the validation of TCP protocols as it relies on a language-dependent source-level parser, which is unable to cope with the full set of languages and constructs found in other network protocol implementations. By contrast, DCE allows to use broader features of the Linux kernel network stack, through carefully designed abstractions of network devices and time-related kernel API (see further details in § 2.2).

Table 6 summarizes the discussion of prior work in this section. The DCE approach allows to provide both scalability, as experiments are no longer bound to run in real time, and easy debuggability, which derives from controllability of the single-process model, detailed in Section 2.

7. CONCLUSION

This paper proposes Direct Code Execution, the first open source framework that allows to integrate real Linux kernel and application code with a leading discrete-event network simulator. DCE provides a realistic, scalable and easy to use environment to reproduce and debug network experiments. We replicated an experiment described in the literature showing reproducible results by using the same protocol implementations over DCE. We also demonstrated a use case on automated code coverage testing that has shown achieving 55-86% coverage of the MPTCP code, and another use case showing easy debugging of a networking stack.

We are confident this framework can help the network community to conduct reproducible experiments and make results more credible by adopting the principle of runnable papers.

---

[11]The bootstrap time might be reduced by using snapshots of the VMs after initialization.

derson for his insightful comments on this work and Alina Quereilhac for her proofreading on this paper.

Code Availability

The source code described in this paper and the detailed information of each experiment are available at `http://yans.pl.sophia.inria.fr/trac/DCE/` If you viewed this paper electronically, click on figures/tables in the PDF to get more information about each result.

8. REFERENCES

[1] MPTCP Wireless Performance. `http://reproducingnetworkresearch.wordpress.com/2012/06/06/mptcp-wireless-performance-draft/`. (Accessed June 15th 2013).

[2] UMIP - Mobile IPv6 and NEMO for Linux. `http://umip.org/`. (Accessed June 3rd 2013).

[3] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Proceedings of the 2002 USENIX ATC*, 2002.

[4] J. Ahrenholz, C. Danilov, T. Henderson, and J. Kim. CORE: A real-time network emulator. In *Proceedings of Military Communications Conference*, MILCOM 2008, pages 1–7. IEEE, Nov. 2008.

[5] S. Barré, C. Paasch, and et al. Linux Kernel implementation of MultiPath TCP. `https://github.com/multipath-tcp/mptcp`. (v0.86 version, accessed June 3rd 2013).

[6] S. Bhatia, G. Di Stasi, T. Haddow, A. Bavier, S. Muir, and L. Peterson. Vsys: a programmable sudo. In *Proceedings of the 2011 USENIX annual technical conference*, USENIX ATC'11, pages 1–6, Berkeley, CA, USA, 2011. USENIX Association.

[7] S. Bhatia, M. Motiwala, W. Muhlbauer, Y. Mundada, V. Valancius, A. Bavier, N. Feamster, L. Peterson, and J. Rexford. Trellis: a platform for building flexible, fast virtual networks on commodity hardware. In *Proceedings of the 2008 ACM CoNEXT Conference*, CoNEXT '08, pages 72:1–72:6, New York, NY, USA, 2008. ACM.

[8] J. B. Buckheit and D. L. Donoho. *Wavelab and reproducible research*. Springer, 1995.

[9] J. F. Claerbou and M. F. Karrenfach. Electronic documents give reproducible research a new meaning. In *1992 SEG Annual Meeting*, 1992.

[10] J. Dike. User Mode Linux. In *Proceedings of the 5th Anual Linux Showcase and Conference*, ALS'01, pages 3–14. USENIX Association, 2001.

[11] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. *ACM SIGOPS Operating Systems Review*, 36(SI):211–224, Dec. 2002.

[12] D. Ely, S. Savage, and D. Wetherall. Alpine: a user-level infrastructure for network protocol development. In *Proceedings of the 3rd conference on USENIX Symposium on Internet Technologies and Systems - Volume 3*, USITS'01, pages 1–13, Berkeley, CA, USA, 2001. USENIX Association.

[13] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, and G. M. Voelker. To infinity and beyond: time-warped network emulation. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 87–100, Berkeley, CA, USA, 2006. USENIX Association.

[14] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container based emulation. In *Proceedings of the 2012 ACM CoNEXT conference*, CoNEXT '12, 2012.

[15] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau. Large-scale virtualization in the emulab network testbed. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, pages 113–128, 2008.

[16] X. Huang, R. Sharma, and S. Keshav. The ENTRAPID protocol development environment. In *Proceedings of the Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies.*, volume 3 of *INFOCOM '99*, pages 1107 –1115 vol.3, Mar. 1999.

[17] S. Jansen and A. McGregor. Simulation with real world network stacks. In *Proceedings of the 37th conference on Winter simulation*, WSC '05, pages 2454–2463. Winter Simulation Conference, 2005.

[18] G. Jourjon, T. Rakotoarivelo, and M. Ott. A portal to support rigorous experimental methodology in networking research. In T. Korakis, H. Li, P. Tran-Gia, and H.-S. Park, editors, *Testbeds and Research Infrastructure. Development of Networks and Communities*, volume 90 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 223–238. Springer Berlin Heidelberg, 2012.

[19] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, SOSP '97, pages 52–65, New York, NY, USA, 1997. ACM.

[20] A. Kantee. Rump file systems: Kernel code reborn. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX ATC'09, pages 1–14. USENIX Association, 2009.

[21] S. King, G. Dunlap, and P. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 USENIX annual technical conference*, USENIX ATC'05. USENIX Association, 2005.

[22] C. Kingsley. Description of a very fast storage allocator, Documentation of 4.2 BSD Unix release, 1983.

[23] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[24] M. Lacage. *Experimentation Tools for Networking Research*. PhD thesis, Université de Nice-Sophia Antipolis, 2010.

[25] B. Lantz, B. Heller, and N. McKeown. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[26] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50 –58, Feb. 2002.

[27] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. *SIGCOMM Comput. Commun. Rev.*, 33(1):59–64, 2003.

[28] Z. Puljiz and M. Mikuc. IMUNES Based Distributed Network Emulator. *Proceedings of the International Conference on Software in Telecommunications and Computer Networks*, pages 198–203, Oct. 2006.

[29] A. Quereilhac, M. Lacage, C. Freire, T. Turletti, and W. Dabbous. Nepi: An integration framework for network experimentation. In *Software, Telecommunications and Computer Networks (SoftCOM), 2011 19th International Conference on*, pages 1–5, 2011.

[30] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How hard can it be? designing and implementing a deployable multipath tcp. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[31] K. Renard, C. Peri, and J. Clarke. A performance and scalability evaluation of the ns-3 distributed scheduler. In *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*, SIMUTOOLS '12, pages 378–382, ICST, Brussels, Belgium, Belgium, 2012. ICST.

[32] R. Russell and J. Kerr. nfsim: Untested code is buggy code. In *Linux Symposium*, 2005.

[33] H. Tazaki, F. Urbani, and T. Turletti. DCE Cradle: Simulate Network Protocols with Real Stacks for Better Realism. In *Workshop on NS3 (WNS3)*, Mar. 2013.

[34] S. Tripathi, N. Droux, K. Belgaied, and S. Khare. Crossbow virtual wire: network in a box. In *Proceedings of the 23rd conference on Large installation system administration (LISA), USENIX Association*, 2009.

[35] Wakoond. Message corruption with hao and route2 XFRM rules. `http://www.wakoond.hu/2012/07/message-corruption-with-hao-and-route2.html`. (Accessed June 3rd 2013).

[36] E. Weingärtner, F. Schmidt, H. Lehn, T. Heer, and K. Wehrle. SliceTime: a platform for scalable and accurate network emulation. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, pages 1–14. USENIX Association, 2011.