Towards More Realistic Network Simulations: Leveraging the System-Call Barrier

Roman Naumann(✉), Stefan Dietzel, and Björn Scheuermann

Humboldt-Universität zu Berlin, Berlin, Germany
{roman.naumann,stefan.dietzel}@hu-berlin.de,
scheuermann@informatik.hu-berlin.de

Abstract. Network simulations play a substantial role in evaluating network protocols. Simulations facilitate large-scale network topologies and experiment reproducibility by bridging the gap between analytical evaluation and real-world measurements. A recent trend in discrete event network simulations is to enhance simulation realism and reduce duplicate implementation efforts by maximizing code reuse. Despite such efforts, it is not yet possible to run arbitrary network applications in state-of-the-art network simulators. As a consequence, researchers are required to maintain separate protocol implementations: one for real-world measurements and one for simulations. We review existing approaches that maximize code reuse in simulations, compare their limitations, and propose a novel architecture for protocol simulation that overcomes those restrictions.

Keywords: Simulation · Virtualization · Code reuse · Simulation realism

1 Introduction

When developing novel ad hoc networking protocols, extensive evaluation to gauge their performance and fitness to fulfill use case requirements is an integral part of the protocol design. The same is true when existing protocols are to be revised or improved. Arguably, network simulations are among the most widespread tools used to evaluate protocols for ad hoc networks. Network simulators strike a balance between the fundamental and asymptotic results that formal, analytical protocol evaluations can provide and the realism that testbed implementations on real hardware can provide.

To implement network simulations, the lower layers of the protocol stack are typically approximated by more or less simplified models, whereas higher protocol layers are re-implemented to mimic real-world protocol stacks. Clearly, the physical layer needs to be simulated, and state-of-the-art physical network models have largely been confirmed by empirical measurements [1,2]. For the protocol under evaluation, it is desirable that it is evaluated using the same code that would be used in real deployments; only then can we draw meaningful conclusions from simulative evaluation results. Designing the protocol layers in

Y. Zhou and T. Kunz (Eds.): ADHOCNETS 2016, LNICST 184, pp. 180–191, 2017.
DOI: 10.1007/978-3-319-51204-4_15

between is challenging, because they need to operate with the simulated physical layer while at the same time allowing the protocol under evaluation to be implemented under realistic conditions.

Today, simulator implementations for intermediate layers are often based on standards or available specifications, whereas real-world implementations contain further optimizations and extensions that affect performance. For instance, real-world TCP variants are a complex interaction of several optimizations performed by the operating system's implementation [3]. Likewise, the most widely used implementation [4] of Optimized Link State Routing [5], a routing daemon for ad hoc networks, implements non-standard link quality extensions that drastically improve performance in wireless mesh networks, whereas a state-of-the-art network simulator's version is based on the official specification only.

In the worst case, the network system – be it a protocol or an application – has to be developed twice: once in the simulator and once for real-world deployment. Such duplicate implementations have several negative implications. First, the implementations' behavior may diverge due to implementation differences. In addition to increased development effort, the differences between the implementations may invalidate simulation results, since they no longer match real-world behavior. Second, the choice of a network simulator may require to use a specific programming language that the development team would not use for real deployments. Finally, the additional effort slows down the development and evaluation of network systems. These and other issues with current simulators have been acknowledged by the simulation community, spawning a trend to increase protocol code reuse [3,6–8].

Code-reuse issues are emphasized by the abstraction level of widely used network simulators, such as OMNeT++ [9] and ns-3 [10]. When developing network protocols for ad hoc networks, researchers interact with artificial interfaces towards the network, medium access control, and physical layers. We argue that shifting the abstraction between actual implementation parts and modeled parts towards the lower layers will provide more realistic simulation results and facilitate more widespread code reuse. To do so, we propose to utilize the system call interface, which is well established in Unix-like operating systems. As ad hoc network systems often use Unix-like operating systems, the system call (short "syscall") interface provides a clear interface to separate real protocol implementations from simulated parts of the network.

In this paper, we contribute a taxonomy of different abstraction levels for network simulations, which we use to survey existing approaches to achieve more realistic network simulations. Moreover, we discuss a novel syscall-level approach to combine the benefits of realistic protocol implementations with those of discrete event network simulations.

We discuss the general role of simulation in network evaluation and introduce discrete event network simulation in Sect. 2. Section 3 then structures options for code reuse in network simulations and discusses existing approaches' advantages and limitations. We present a less restrictive simulation architecture that maximizes code reuse in Sect. 4. In Sect. 5 we summarize and conclude the paper.

2 Discrete Event Network Simulation

When designing or implementing a new network system, verifying correctness and efficiency is an important but difficult task. A network system is generally developed with a use-case environment in mind. In an ideal world, the system's designer could test her protocol in this exact environment as long and as often as necessary. In reality, the exact environment is often not available due to practical considerations such as cost of components or the time it takes to perform measurements, which is especially true when a large number of systems are involved. Likewise, real-world measurements are not easily reproducible, since external influences often cannot be controlled.

For these reasons, several accepted techniques for network system evaluation aim to increase scalability and reproducibility over real-world measurements by controlling external influences to different degrees. The system designer can use these techniques to evaluate a network system without requiring the exact deployment environment. Common approaches to evaluation fall in the categories *analytical evaluation*, *simulation*, and measurements using a *testbed.* As sketched in Fig. 1, evaluation by analytical evaluation, simulation, and testbeds typically offer scalability and reproducibility in decreasing order, and they offer closeness to real-world measurement results, i.e., "realism," in increasing order.

The testbed is the method closest to real-world measurements; protocols are evaluated on real hardware. Testbeds provide partially controlled environments where measurement time and topology of the nodes are pre-determined, whereas external interference, for instance, cannot be predicted in the general case. Scalability (i.e., number of nodes, size of topology, number of measurements) is limited by practical considerations, such as cost of hardware, available space, and the time it takes to perform measurements. The benefit of testbeds is that results are close to real-world measurements when the topology resembles the network system's use-case environment.

Analytical evaluation is on the opposite end of the spectrum. It involves finding the right abstractions to formally model a network protocol and its environment, and it allows to mathematically assess their interaction. Once such a model is found, it is generally possible to arbitrarily scale parameters, such as, number of nodes or size of topology. Realism of analytical evaluation results is highly dependent on the choice of abstractions, since it is seldom possible to formalize all facets of a protocol in a tractable analytical model.

From the viewpoint of realism, network simulations fill the middle ground between analytical protocol evaluation and real-world measurements performed in a testbed. In comparison to testbeds, network simulations provide better scalability and reproducibility. Using modern network simulators, it is possible to simulate hundreds or thousands of nodes in arbitrary topologies and repeat experiments with fully controlled randomness.

The most common technique for network simulators is discrete event network simulation. A discrete event network simulator is driven by events, which can be a timer running out or a packet being received by a simulated network card.

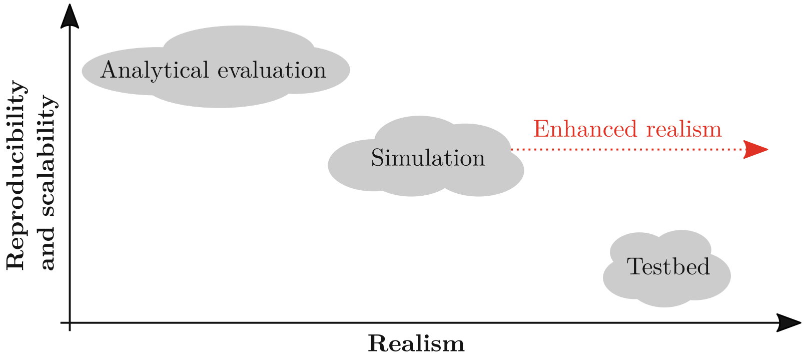

Fig. 1. Evaluation realism

An important property of discrete event network simulators is that events are mere points in time, i.e., no time passes *during* an event. Similarly, no time passes *between* events. Instead, the simulator skips to the next event after processing one event, not simulating anything in between. This approach scales well since it only requires to process what happens during events. Simulated time in a discrete event network simulator is different from system time: simulated time may run faster or slower than system time, depending on the system load of the machine running the simulator and the simulation's complexity.

Discrete event simulators facilitate reproducibility: if properly implemented, running a simulation twice with the same parameters yields the exact same results, enabling precise debugging of rare corner cases. Randomness in simulations is controlled by the simulator's (pseudo-)random number generator, which can be initialized with different seeds to select a statistically meaningful sample size.

From a protocol-implementation perspective, a simulated environment is therefore different from a real-world environment. When considering code reuse, we need to carefully consider the effect of discrete event simulation on real-world implementations. If too many aspects of the implementation are affected by the simulator interface, code reuse is difficult or impossible, limiting meaningfulness of simulation results. If too few aspects are affected, we lose the benefits of discrete event simulations, foremost its reproducibility.

3 Options for Code Reuse

We have established that finding the right level of abstraction between realistic implementations and simulated parts is key to reusable yet scalable and reproducible network simulations. The level of abstraction is determined by the extent to which code can be reused between simulations and real deployments. In this section, we identify different options for code reuse and survey existing approaches within this structure.

3.1 Partial Source Reuse

The simplest form of code reuse is what we term *partial source reuse*. When the protocol implementation's programming language is compatible with the simulator language, it is trivially possible to copy-and-paste chunks of source code to the network simulator implementation and execute them as part of the simulation.

For instance, the state-of-the-art discrete event network simulators OMNeT++ [9] and ns-3 [10] support the C++ programming language for protocol simulations. Therefore, C or C++ protocol implementation source code can be used as part of a simulation. Likewise, existing Java protocol source code can be used in the JiST/SWANS simulator [11].

There are several software components, however, that need porting or re-implementation to work in a discrete event simulator:

- real-world socket APIs cannot be used in a simulator; instead, the network abstractions provided by the simulator need to be used;
- time is different from the system time in a network simulator, so no system time queries must be made;
- random numbers have to stem from the simulators pseudo-random number facility exclusively;
- concurrency is often not supported by discrete event simulators, instead the asynchronous event dispatcher of the simulator has to be used;
- global variables may prevent spawning more than one application instance in a simulator; and
- likewise, file system operations may conflict when more than one application instance is simulated.

We conclude that partial code reuse can help alleviate duplicate implementation efforts, but by no means eliminates them, because all of the above issues have to be addressed manually.

3.2 Full Source Reuse

Recent research [3,6,8,12,13] has investigated how duplicate implementation effort can be minimized by increasing code reuse. Here, we discuss approaches based on sharing the entire source code of a protocol implementation for simulation and real-world deployment. We distinguish two different approaches for full source reuse: employing a software compatibility layer and using alternative compilation methods.

Software Compatibility Layer. A special case of code reuse is the approach taken by Click [14], where network protocols are implemented in a modular fashion in C++ and a domain-specific router configuration language. Click protocol implementations can be deployed on real hardware or integrated in a simulator such as ns-3 [6]. Click's aim is to find suitable programming abstractions for the critical software components listed in Sect. 3.1. The protocol is implemented against this compatibility layer instead of APIs that are specific to the real world or simulation environments.

Of course, the Click approach only works when developing a new protocol, as tight integration with Click is needed. Another restriction is that the compatibility layer can only support those features that *all* supported platform APIs provide.

Mayer *et al.* [13] consider a more lightweight compatibility layer for the OMNeT++ simulator. Instead of compiling a user space protocol's sources into an executable, a shared library is built and dynamically loaded into the simulator. The authors suggest to replace the network functionality with a compatibility wrapper, so that it can quickly be exchanged depending on whether the protocol is built for real-world deployment or for simulation. Likewise, calls that query the current time are replaced by pre-processor macros that switch between simulation time and system time depending on the compilation mode.

Alternative Compilation. Tazaki *et al.* [15] propose a refined shared library approach for the ns-3 simulator that, with some restrictions, allows a protocol implementation's sources to run unmodified in the simulator, i.e., without a compatibility layer. Again, a shared library is built from the implementation's sources, as depicted in Fig. 2, and dynamically loaded into the simulator. However, instead of using a compatibility layer (which requires in-source changes), calls to the operating system's standard library are redirected to a wrapper library. The wrapper library decides whether to pass the call to the operating system (for most calls), or provide an alternative implementation based on simulator facilities. For example, a call to the function that returns the length of a string (`strlen`) can safely be passed through, as it does not perform input or output operations, whereas a call to a function that normally returns the current system time (e.g., `gettimeofday`) is replaced by a wrapper that returns simulation time. The approach can be used for kernel-space protocols in a similar fashion [3,7].

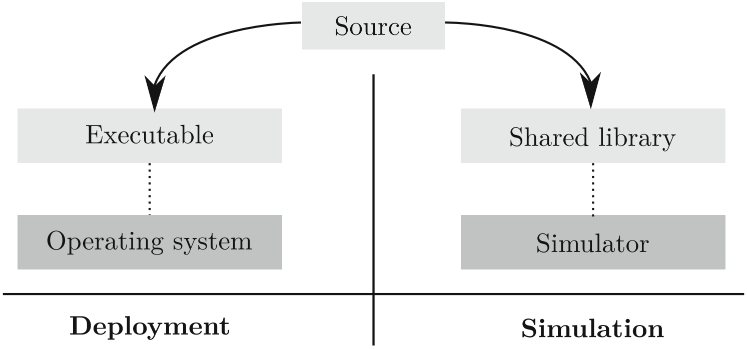

Fig. 2. Shared library approach

The shared library approach is the first to allow running unmodified applications without reducing the reproducibility guarantees provided by discrete event simulation, but it has restrictions on a conceptual level: compilation to a shared library requires that the source code is available and in fact can be compiled into such a shared library. The former is not necessarily true when proprietary implementations are evaluated and the latter does not usually hold for most interpreted programming languages and even many mainstream compiled languages, such as Java or Go.

3.3 Process Reuse

Another approach is to run node processes or even the nodes' operating systems via virtualization and solely exchange network traffic between these real processes and the simulation. In the context of the OMNeT++ simulator, this approach was first briefly discussed in [13] and later implemented by Staub *et al.* [12].

The advantage of network traffic exchange between simulation and real processes is that full code reuse is trivial, since processes run in the same environment as they would when deployed. No programming language limitations or tool chain restrictions apply when the system is implemented as in [12]. Unfortunately, this approach does not maintain perfect reproducibility, because only network operations are simulated. Processes or operating systems do not run in the simulated time domain but in their respective system time domain; system (pseudo-)random numbers cannot be predicted, i.e., reproduced.

4 Leveraging the System Call Barrier

The shared-library approach that we saw in Sect. 3.2 chose the operating system's standard library as the barrier between simulation and a user's protocol implementation. What we propose here is to use a lower-level abstraction as the border between simulation and real-world applications. Operating systems already provide a natural barrier between user-space and kernel-space that can only be transgressed via so-called system calls (syscalls).

4.1 The Syscall Interface

An obvious property of the system call barrier is that the operating system is agnostic towards programming language details: every process, regardless of whether it is a compiled executable, an interpreter, or a just-in-time compiled program fragment, uses the same syscalls to interface with the operating system's kernel. So the approach supports running all of these protocol implementations, even proprietary ones, with zero modification, thereby maximizing code reuse.

Techniques to capture and modify system calls, often called *syscall wrapping*, have been used before in the security context [16] and for operating system emulation [17]. To make use of the system call barrier for discrete event network simulation, it is necessary to filter and selectively re-implement system calls.

The Linux operating system kernel version 4.5, for instance, supports a total of 385 system calls for file manipulation, signal handling, concurrency, socket operations, and so forth. While this number may appear to be large, most system calls are rarely used and implementing only a subset would already support numerous protocol implementations. For example, our experiments show that a simple web page served by the Nginx web server utilizes 46 distinct syscalls and the olsrd [4] daemon uses 26 unique system calls when running in minimal mode. Both implementations invoke largely the same – frequently used – system calls and jointly require only 51 distinct system calls. Of these commonly used syscalls, only a fraction needs to be modified during execution, whereas most system calls need not be modified to support reproducibility in discrete network simulations. System call groups that can be passed through instead of being re-implemented include security options, memory manipulation, process manipulation, and most concurrency operations, since these do not usually involve network traffic or system time [15].

4.2 Syscall-Barrier Process Simulation

Figure 3 shows an overview of our proposed process simulation architecture: Topmost are user-space processes, and bottommost is the operating system kernel. Two components constitute the simulator process in the middle: the discrete event simulation logic to the right and the syscall wrapper to the left. We propose to use syscall wrapping, as shown in Fig. 3, to selectively redirect syscalls to the simulator logic and emulate them there. Non-emulated system calls are forwarded to the operating system as is. The simulator process thereby implements a secondary system call barrier to run real-world processes within the simulation environment. Previous work that uses syscall wrappers for process virtualization suggests that the syscall barrier's performance is lower than hardware virtualization as in, e.g., XEN [18]. Dike *et al.* [17] notes that the performance penalty is dominated by additional context switches. This factor can, however, be mitigated on platforms that provide special operating system support for syscall wrapping [19,20].

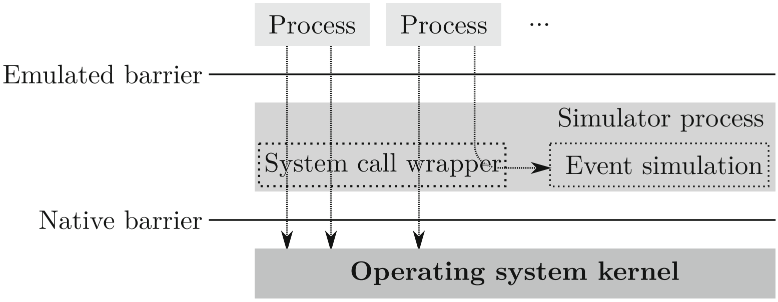

Fig. 3. Syscall-barrier process simulation

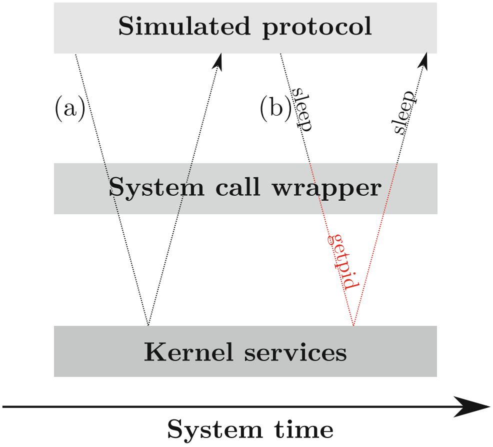

Fig. 4. System call wrapping

To illustrate our approach, we discuss how to wrap two example syscalls. Namely, we discuss two system calls that the OLSR mesh routing daemon issues: the first system call (a) is `mprotect`, a system call that changes access permissions on a memory region. The second system call (b) is `nanosleep`, which causes the kernel to suspend the calling thread's execution via a high-resolution timer. As shown in Fig. 4, (a) is an example for a syscall that can be passed through to the actual kernel services, whereas (b) is a syscall that needs to be caught and handled by the wrapper.

Like [17], we assume a Linux system and a syscall wrapper based on the `ptrace` framework [20]. The wrapper runs solely in user space and leverages the `ptrace` system call to trace protocol processes. `ptrace` enables syscall inspection and modification at two points: (1) just before the system call is processed by the kernel and (2) just after the system call was processed by the kernel, but before the protocol process is notified. As soon as the `mprotect` system call (a) is issued, but before the kernel processed the call, the syscall wrapper would be notified by the `ptrace` framework. It can inspect the system call and decide, based on a lookup table, that `mprotect` does not affect the network simulation. The wrapper hands back execution to the kernel, which processes the system call as usual, and is notified again when the system call's processing is finished but before the protocol process is notified. Again, the syscall wrapper continues execution without modification.

The second system call (b) is `nanosleep`. Time-related system calls need embedding in a discrete event simulation environment, so the wrapper intercepts the call: the syscall wrapper first registers an event with the simulator that notifies the wrapper once the requested *simulation* time has passed. If operating system support is available, the original system call can be skipped altogether [19]. Otherwise, the syscall is replaced by a dummy system call without input or output, such as `getpid` [17], as indicated in Fig. 4. After the (dummy) system call is processed, but before the protocol process is notified, the original system call's result is emulated by modifying the protocol processes registers. In particular, the syscall's return value is replaced by zero, which indicates success for `nanosleep`. Next, the syscall wrapper waits until the event it has registered with the simulator expires. Once notified that the event has expired, it continues the protocol process. By following these steps, the `nanosleep` system call is transparently emulated by the simulator, replacing system time with simulation time – which is crucial for experiment reproducibility.

Other system calls can be implemented in a similar fashion. Some syscalls can be passed through, because they do not interfere with the simulation time. In some cases, syscalls may be forwarded, but their parameters need to be modified. Examples are file system operations, where potentially path prefixes should be modified by the syscall wrapper. Others, such as timing and network interactions, need to be intercepted entirely and handled internally.

4.3 Syscall-Barrier System Simulation

Pushing the border even further towards the operating system level, we can emulate the whole operating system while maintaining the syscall barrier as the interface to the network simulator. Normally, this approach would require that the simulator emulates hardware on which a node's operating system can run. Dike *et al.* [17] shows that instead it is possible and feasible to port an operating system "to itself" in terms of system calls.

Instead of creating an environment that the virtualized operating system can run on, the virtualized operating system is ported to run on an existing system call environment. In theory, we can utilize these findings to run a node's operating system and all associated protocol implementation's processes via the same interface that we propose in Sect. 4.2 – the system call barrier.

This approach maximizes code reuse: nodes run fully virtualized, running real-world protocol stacks on all layers above the medium access and physical layers, which the network simulator models and simulates.

It is an open research challenge to evaluate how a real-world operating system behaves when running on top of a discrete event network simulator, since time-related syscalls behave differently in a discrete simulation environment. However, due to the successful porting of kernel-space UDP and TCP implementations [3,7], we are positive that this next level of code reuse can be obtained without much modification to the kernel.

5 Conclusion

During the design and implementation phase of a network system, it is important to verify the system's correctness and performance. Simulating a protocol during the design phase allows to carefully tune parameters and quickly assess a proposed modification's performance impact. Unfortunately, with today's tool support, it is often required to maintain two separate implementations for simulation and real-world deployment. This undermines both correctness – as implementation differences question simulation results – and efficiency – as two implementations duplicate development efforts.

We reviewed and structured a number of approaches that maintain the reproducibility and scalability of discrete event network simulation, and at the same time, improve correctness and reduce duplicate effort by increasing code reuse. Among those approaches, the recently proposed shared library approach [3,7] facilitates full source reuse with a state-of-the-art simulator, albeit with a number of restrictions.

We proposed a system-call barrier design as an alternative abstraction level to form the border between simulator and protocol stack, i.e., model and real-world code. Our design has the potential to solve the remaining restrictions that are inherent to the shared-library approach. It is agnostic to programming language, it can run compiled, interpreted, or just-in-time compiled code, and it does not require a modified tool chain nor modified source code. The proposed design is based on a technique called system call wrapping, which has been used for security and virtualization previously. We also describe an extended design that utilizes an operating system's port to itself to simulate nodes' operating systems via the system-call barrier.

We expect that, along with the trend to improve code reuse, the use of simulation in the evaluation of network systems will increase. It remains to be seen whether perfect reproducibility can be upheld when modeling arbitrarily complex systems such as full operating systems without modification, but the direction is promising and we expect more results from this line of research in the near future.

Acknowledgments. This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 636892 in the context of the PREVIEW project [21].

References

1. Hashemi, H.: The indoor radio propagation channel. Proc. IEEE **81**(7), 943–968 (1993)
2. Tanghe, E., Joseph, W., Verloock, L., et al.: The industrial indoor channel: largescale and temporal fading at 900, 2400, and 5200 MHz. IEEE Trans. Wirel. Commun. **7**(7), 2740–2751 (2008)
3. Jansen, S., McGregor, A.: Simulation with real world network stacks. In: Proceedings of the Winter Simulation Conference, December 2005
4. OLSR.org Wiki (2016). http://www.olsr.org/mediawiki/index.php/Main_Page. Accessed 26 June 2016
5. Clausen, T., Jacquet, P.: Optimized Link State Routing Protocol (OLSR). Request for Comments, No. 3626, October 2003
6. Suresh, P.L., Merz, R.: ns-3-click: click modular router integration for ns-3. In: Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques, SIMUTools 2011, pp. 423–430. Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering (ICST) (2011)

7. Tazaki, H., Urbani, F., Turletti, T.: DCE cradle: simulate network protocols with real stacks for better realism. In: Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques, SimuTools 2013. Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering (ICST), pp. 153–158 (2013)
8. Camara, D., Tazaki, H., Mancini, E., et al.: DCE: test the real code of your protocols and applications over simulated networks. IEEE Commun. Mag. **52**(3), 104–110 (2014)
9. Varga, A., Hornig, R.: An overview of the OMNeT++ simulation environment. In: Proceedings of the 1st International Conference on Simulation Tools, Techniques for Communications, Networks, Systems and Workshops, Simutools 2008, pp. 60:1–60:10. Institute for Computer Sciences, Social-Informatics, Telecommunications Engineering (ICST) (2011)
10. Henderson, T.R., Lacage, M., Riley, G.F., et al.: Network simulations with the ns-3 simulator. In: SIGCOMM Demonstration, vol. 14 (2008)
11. Barr, R.: An Efficient, Unifying Approach to Simulation Using Virtual Machnies (2004)
12. Staub, T., Gantenbein, R., Braun, T.: VirtualMesh: an emulation framework for wireless mesh and ad hoc networks in OMNeT++. In: Simulation (2010)
13. Mayer, C.P., Gamer, T.: Integrating real world applications into OMNeT++. Technical report TM-2008-2, Karlsruhe Institute of Technology, Karlsruhe, Germany (2008)
14. Morris, R., Kohler, E., Jannotti, J., et al.: The click modular router. ACM Trans. Comput. Syst. **18**, 263–297 (2000)
15. Tazaki, H., Uarbani, F., Mancini, E., et al.: Direct code execution: revisiting library OS architecture for reproducible network experiments. In: Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies, CoNEXT 2013, pp. 217–228. ACM (2013)
16. Watson, R.N.: Exploiting concurrency vulnerabilities in system call wrappers. In: WOOT, vol. 7, pp. 1–8 (2007)
17. Dike, J., et al.: User-mode Linux. In: Annual Linux Showcase & Conference (2001)
18. Emeneker, W., Stanzione, D., et al.: HPC cluster readiness of Xen and user mode Linux. In: CLUSTER (2006)
19. Vivier, L.: User-Mode-Linux SYSEMU Patches (2011). http://sysemu.sourceforge.net/. Accessed 26 June 2016
20. Ptrace(2) Linux User's Manual, June 2016
21. Preview Project EU - Predictive System for Injection Mould Process Optimisation. http://www.preview-project.eu/. Accessed 27 Oct 2015