DCE Cradle: Simulate Network Protocols with Real Stacks

Hajime Tazaki, Frédéric Urbani, Thierry Turletti

DCE Cradle: Simulate Network Protocols with Real Stacks for Better Realism

Hajime Tazaki
NICT, Japan
tazaki@nict.go.jp

Frédéric Urbani
Inria, France
frederic.urbani@inria.fr

Thierry Turletti
Inria, France
thierry.turletti@inria.fr

Abstract

Reusing real protocol implementations of the kernel network stack in network simulators can increase the realism of network experimentations as well as decrease the cost of protocol development. In this paper, we introduce `DCE Cradle`, a novel framework that allows to use any features of Linux kernel network stack with existing `ns-3` applications. `DCE Cradle` uses Direct Code Execution (DCE) to address the brittleness of Network Simulation Cradle (NSC). We validate `DCE Cradle` with TCP goodput measurements, and analyze its performance overhead with micro-benchmarks in a large scale simulation. Then we show with an example, an actual implementation of the DCCP transport protocol, how easy it is to simulate a real implementation using `DCE Cradle`. We believe that this tool can highly benefit the network community by enabling more realistic evaluation of network protocols.

Categories and Subject Descriptors

I.6.7 [**Simulation Support Systems**]

Keywords

Direct Code Execution, Simulation, Real Code.

1. INTRODUCTION

Motivation: A large number of network protocols for the Internet have been proposed and studied for decades. Some of them are widely used in real networks but others are still in the development phase, having not yet been implemented or operated in detail because they have been just proposed. Network simulators are widely used to develop and evaluate in depth network protocols for several reasons: (i) no physical network devices and actual network topology constructions are required, (ii) easier application development is provided by simulators, (iii) variety of parameters into network functionality are deeply configurable, and also (iv) the reproducibility of performance results is ensured.

However, while the user perspectives of network simulators give us a straightforward way to studying network protocols, adding new network protocols (a.k.a. network stacks) into network simulators is relatively hard. Indeed, implementing a protocol itself requires a huge effort to developers: e.g., the amount of Linux SCTP codes is about 34K LOC[1], resulting in difficulties to implement the protocol for a specific network simulator from scratch. Moreover, once the protocol is successfully implemented with a significant amount of effort, it still requires an important work to ensure the validity of the protocol operation; especially ensuring the interoperability is hard after implementing new protocols. Furthermore, if the behavior of implemented simulation models differs from real protocols, who can trust the performance result obtained from network simulators? From these observations, we argue that reusing real protocol code in network simulators is a good alternative and can prevent the above issues.

Existing Work: Reusing real code still requires development effort to be used in network simulators. Network Simulation Cradle (NSC) [3] allows us to use real kernel network stacks with the simulator's facility like traffic generators and analyzers. However, it turns out that we need to update the *glue* code of NSC, which bridges the gap between the network simulator's code and kernel network stack's code, when we want to add new features of the kernel network stack. That also makes it hard to track and support the latest kernel code into network simulator.

Direct Code Execution (DCE) [6], the follower of NSC, carefully considered this version tracking issue of NSC in the initial design phase by introducing a transparent configuration system of kernel building [6]. It allows to easily enhance and replace the kernel by adding new feature into the kernel[2]. While DCE currently provides a bridge function between POSIX socket-based application and Linux kernel over `ns-3`, it is still unable to use `ns-3` native applications (e.g., traffic generator), which NSC already supports.

In summary, NSC has interesting features but can

[1] `wc -l net-next/net/sctp/*.c`

[2] in theory, by simply adding CONFIG_NET_XXX into the configuration file.

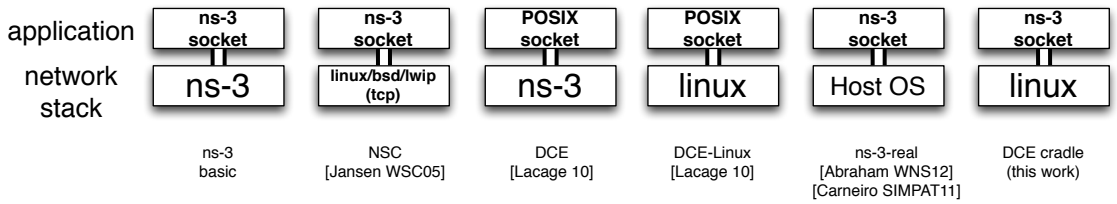


Figure 1: Current possible combinations of network stacks and applications.

hardly update the network stacks, while DCE alleviates the concern of network stack modification but does not allow `ns-3` to benefit from the features.

In this paper, we propose `DCE Cradle` for the introduction of new protocols into network simulators. Instead of implementing models for a specific network simulator, we reuse real code (e.g., an existing protocol) and provide a *bridge* between the real code and simulator. We carefully design `DCE Cradle` without breaking the existing functionality of DCE and `ns-3` socket architecture by considering the gaps between asynchronous ns-3 socket API and general POSIX socket API. In this manner, researchers do not need to reinvent the wheel and are also able to test the real code instead of a simulated one, which is difficult to trust.

`DCE Cradle` adds a feature of `ns-3` native application with Linux kernel integration, which is provided by DCE. It currently supports several types of protocol socket (i.e., TCP, UDP, Raw, DCCP) with `OnOffApplication` and `PacketSink` transparently. Preliminary evaluations done with our implementation demonstrates the validity of the TCP socket on `DCE Cradle`, and micro-benchmarks give a reasonable performance of `DCE Cradle` without adding much overhead from `ns-3` native network stack.

In summary, the contributions of this paper are threefold.

1. We propose `DCE Cradle` to facilitate the development and testing of network protocols by enabling the use of Linux network stack and `ns-3` native applications.

2. We validate the implementation of `DCE Cradle` with the behavior of TCP implementation in congested links, and then study its performance by focusing on the simulation time and network scale. The performance study shows that `DCE Cradle` is at most 1.3 times faster than NSC, while it is about 2.2 times slower than the `ns-3` native stack.

3. As a general contribution of our work, we propose a complete and fully functional implementation of `DCE Cradle` to the network community[3].

2. RELATED WORK

At this moment, there are several ways to utilize a network stack (such as ipv4, ipv6, tcp or udp) in `ns-3`. We briefly review the existing solutions and explain how `DCE Cradle` can make easier the development of network protocols.

Network Simulation Cradle (NSC) [3] is a pioneer of introducing alternative network stacks into `ns-3` network simulator. By parsing source code automatically to avoid global variables conflicts between multiple instances of simulated network stack, encapsulated userspace library of kernel network stack could be used as a model of simulations. While NSC is successfully integrated as a TCP model of `ns-3`, considerable effort is still required to add new features such as UDP or DCCP transport protocol. In particular, ad-hoc modifications are necessary in the *glue* layer between the kernel network stack and the simulator. This also makes it difficult to track new versions of kernel to utilize.

Direct Code Execution (DCE) [6] was proposed to utilize existing POSIX socket-based applications (e.g., `Quagga` or `iperf`) in `ns-3` to reduce the development cost of network protocols. With DCE, an application without any modifications is able to run on `ns-3` with several kinds of network stacks, such as `ns-3` native stacks, NSC (Linux, FreeBSD, OpenBSD, etc), or Linux via DCE (described next).

Linux kernel emulation over DCE (DCE-Linux) [6] is another way to integrate a Linux network stack in a simulation as NSC does. By introducing a virtual architecture so-called *sim*, the network stack functionality in kernel space can be transparently integrated as a userspace library, resulting a spontaneous upgrade of the kernel network stack. For instance, Mobile IPv6 simulation[4] with the interaction of userspace signaling daemon has been introduced without much changes of *glue* code.

Keeping track of the latest kernel version with DCE-Linux is still complex because it requires following the changes of the upstream kernel code. However, DCE was intended to track the kernel version automatically as it was originally designed to be merged with Linux kernel source tree. If DCE-Linux code is included in the

[3] `https://codereview.appspot.com/6856090/`

[4] `http://code.nsnam.org/thehajime/ns-3-dce-umip/`

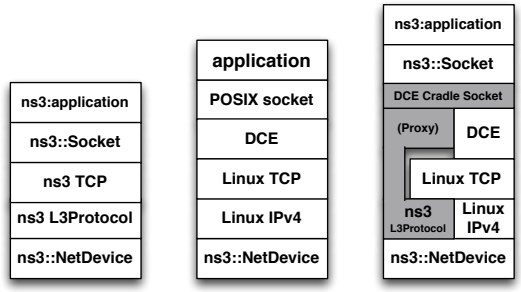
kernel source tree as User Mode Linux, it is possible to reduce the effort needed to track the latest version of the kernel.

ns-3-real is a smart alternative to use different network stacks from `ns-3` native applications. Carneiro [2] and Abraham [1] proposed a mechanism to use the network stack of host operating system directly from `ns-3` applications. Here the goal is to improve the re-usability of the `ns-3` application code using the socket library abstraction. However the network stack of host operating system is only available when the application in the simulator communicates with the outside world: we cannot utilize this feature in a usual simulation scenario.

Figure 1 summarizes the possible combination of applications and network stacks available in `ns-3`. This paper addresses the brittleness of NSC when adding a new feature in the kernel code, and provides a new approach using Linux kernel for `ns-3` native applications via DCE. The goal is to allow users to benefit from useful `ns-3` applications, such as traffic generators, and to reuse protocol stacks of the latest Linux kernel.

3. DESIGN AND IMPLEMENTATION

In this section, we present an overview of `DCE Cradle`. The main idea of `DCE Cradle` consists of three parts: 1) introducing wrapper socket factory objects to access DCE-Linux, 2) adding a proxy Layer-3 object to make transparent to the `ns-3` side, and 3) enhancing DCE to support `ns-3` socket communications.



(a) `ns-3` (b) DCE-Linux (c) `DCE Cradle`

Figure 2: Overview of `DCE Cradle` and other network stacks available on ns-3. The gray part is the modified part for `DCE Cradle`.

Socket Factory: In order to provide a transparent interface to `ns-3` applications (i.e., `OnOffApplication` or `PacketSink`), we introduce new socket factory classes to utilize the Linux network stack. The sockets support several transport protocols (e.g., TCP, UDP, DCCP) and the raw socket as well.

The socket factory classes bridge the DCE-provided Linux kernel socket (i.e., `LinuxSocketFd`) to access the network protocol implementation of the kernel space. Packets generated by `ns-3` applications thus go through Linux kernel using an extended socket, then go back to the network device of the simulator. Figure 2 illustrates the components of the different ns-3 protocol stacks. While DCE completely replaces `Socket` and Layer-3 protocol stack (`L3Protocol`) of `ns-3` with emulated POSIX API and the kernel network stack respectively, `DCE Cradle` silently replaces `Socket` part with inherited features of the original one and makes applications available to access Linux network stack.

Proxy Layer-3 Object: The new Layer-3 protocol stack objects inherited from `ns-3` native stack also makes applications transparent to the underlying network stack. IP address and route configuration are extended to proxy from the original simulation core to the kernel network stack; existing services such as IP addresses and route configuration can be transparently used from `ns-3` simulation scenarios.

Thus, users of `DCE Cradle` only have to care about the helper library (i.e., `LinuxStackHelper`) to benefit from the kernel functionality.

DCE core extension: In order to utilize `ns-3` native sockets over DCE Linux network stack, we extended the DCE core functionality. A brief overview of DCE and its extensions are illustrated in Figure 3. In a nutshell, the extension is derived to *fill the gap between the asynchronous ns-3 socket API and the POSIX socket API.*

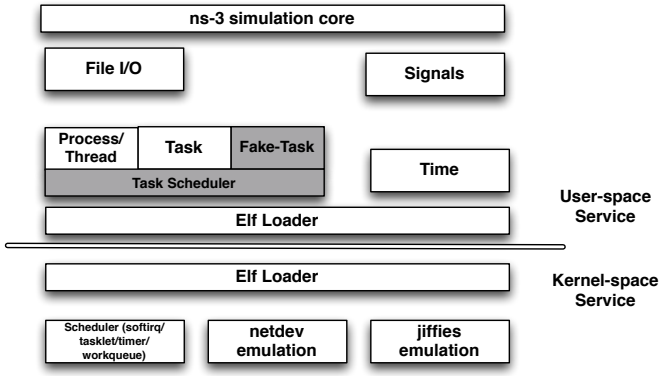


Figure 3: Overview of DCE component and its extensions (in gray) for `DCE Cradle`.

While the original DCE is designed to make possible both blocking and non-blocking socket operation, defined in the POSIX socket-based application's code, `ns-3` sockets only support non-blocking operations. Non-blocking socket operations require that the Linux kernel immediately returns to the application without waiting for completion of processing. In order to support this feature in the DCE extension, we simply use the `LinuxSocketFd::Fnctl` call to configure the Linux kernel socket created by `ns-3` application (via `DCE Cradle`)

being *non-blocked*.

In addition to the above socket operation, we introduced *fake-task* to execute kernel code within the task scheduler. Since the timer in the kernel code of the original DCE-Linux requires to be managed with the timer of the simulation core, socket operations need to be properly scheduled. This scheduling task is provided by the DCE task manager, but with the following limitations: (i) socket operations initiated by `ns-3` applications are not allowed to wait and (ii) queuing in the scheduler is not possible. This constraint brings the design of *fake-task* so that socket operations in the kernel return immediately and the scheduler design remains untouched.

4. PROTOCOL VALIDATION USING DCE Cradle

In order to validate our `DCE Cradle` design and implementation, we conducted a simple experiment with the dumbbell topology shown in Figure 4 using multiple TCP flows. The bottleneck link was configured with uniform random packet loss of 5%, a latency of 100ms and a bandwidth of 2Mbps. The simulations were run on a single machine equipped with an Intel Core i7-2600 (3.4GHz) processor, 16 Gbytes of memory and Ubuntu 10.04 64-bit with kernel versions 2.6.32.29.

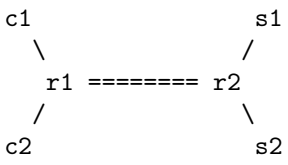


Figure 4: A dumbbell topology to validate `DCE Cradle` with TCP, where `c1, c2` denote TCP clients, `s1, s2` denote servers, and `r1, r2` are routers connected with a bottleneck link.

As an example of protocol stack under test, we picked TCP over IPv4 and compared several existing implementations: 1) `DCE Cradle` with Linux net-next 2.6 kernel (July 2010 version), 2) `DCE Cradle` with linux-stable (July 2012 version, a.k.a Linux 3.4.5)[5], 3) `ns-3` native TCP stack, 4) NSC with Linux 2.6.26 stack, and 5) a real network stack behavior under Linux 2.6.32-28 with `netem` to configure the delay, bandwidth, and packet loss behavior at the bottleneck link. Two clients generate 1Mbps TCP flows using `OnOffApplication` of `ns-3` over the configured dumbbell topology and the server side measures the application goodput after 60 seconds traffic injection. In case of 5) using a real Linux stack, we used `iperf` for the goodput measurement. For all the scenarios above, the tests were run 100 times using different random seeds.

[5]`http://code.nsnam.org/furbani/ns-3-linux-3`

Additional configurations for this validation are the followings: checksum calculation is enabled on all the nodes, the DCE configurations, ucontext-based fiber and DlmLoader [6], are chosen because of their performance benefits under the DCE environment.

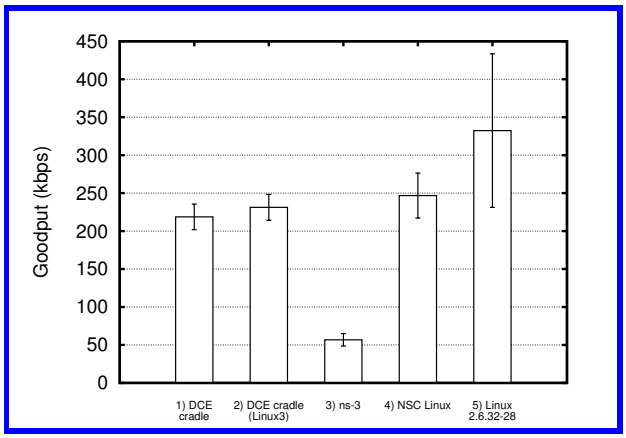


Figure 5: Mean goodput with standard deviation from 100 replications.

Figure 5 plots the goodput performance along with the standard deviation from 100 runs of the experiment[6]. While the experimentation with `ns-3` native TCP stack shows about 83% less bandwidth than with the real environment, NSC and `DCE Cradle` obtain more realistic performance (only 25-30% less bandwidth than with the real environment) under the simple topology scenario above. This is an example scenario that the `ns-3` TCP implementation does not model correctly while NSC and `DCE Cradle` are able to report a close performance to real environment. Further investigations are required to ensure the validity of TCP behaviors, but this preliminary result shows that `ns-3` applications can be utilized with `DCE Cradle` with a realistic behavior of TCP goodput like NSC with the latest version of network stack using DCE. Furthermore, it allows to verify that the experimental results obtained are compatible with the ones observed on actual network and real software environment.

Note that this high level analysis of the `ns-3` behavior is similar to the validation conducted in paper [3] at the exception that [3] uses `ns-2` as a reference simulator of NSC.

5. PERFORMANCE

In this section, we conducted micro-benchmarks to analyze the performance of the `DCE Cradle` and other

[6]For each experiment shown in the following text, figures have a link to web pages to reproduce experiments. Click on figures in the PDF (when viewed electronically) to get the instructions on how to replicate the experiment. `http://www.nsnam.org/~thehajime/ns-3-dce-doc/dce-cradle-usecase.html`

approaches. The micro-benchmark scenario is the same as described in the NSC's paper [3], and aims to measure the time required to simulate a specific simulation scenario. In the first scenarios, we varied the number of client nodes (i.e., senders) in the simulated scenario and measure the actual simulation time (i.e., wall-clock time). Each run corresponds to 60 seconds of traffic. Then, in the second scenario, we fixed the number of nodes to two and varied the duration of TCP traffic from one second to 400 seconds.

We also used the same TCP protocol stacks, network topology and configuration parameters of the bottleneck link as the ones described in Section 4.

Varying the number of nodes

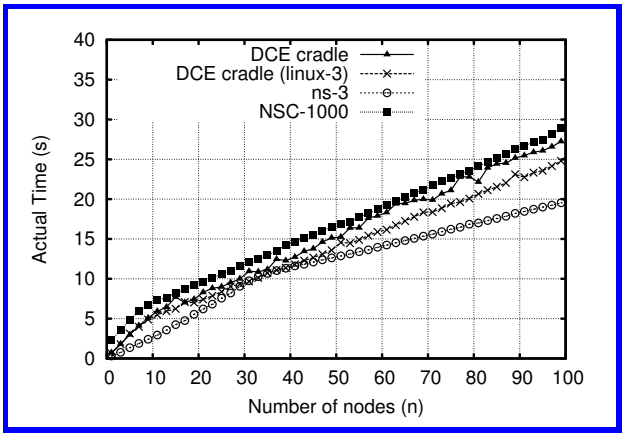


Figure 6: Duration of the simulation in function of the number of client nodes.

Figure 6 illustrates the mean actual time of the simulation for five runs in function of the number of client nodes in the simulation. We can note that two network stacks via `DCE Cradle` spend almost the same time as NSC does, however the NSC simulations fail if the number of client nodes exceeds 50. This limitation is due to the static variable used for the maximum number of instances in NSC, which is set to 100 by default. The additional plot in figure 6 was obtained with NSC-1000, i.e. a variation of NSC which sets the maximum number of instances to 1000 instead of the default value 100. The performance obtained with NSC-1000 obtains slightly higher actual time compared to `DCE Cradle` (about 1.05 times more time). Although there is no straightforward way to compare in detail the design of `DCE Cradle` and NSC, this micro-benchmark shows a reasonable performance of `DCE Cradle` while keeping a similar level of realism.

The `ns-3` implementation of TCP spends less time than others due to the low goodput performance, observed from previous experiment, generating less events than others in the simulation.

Varying the duration of traffic

Figure 7 shows the mean value of actual duration for five runs of the simulation in function of the simulated time (described in the scenario). We can observe that the simulation time also affects the actual time of the simulation, especially `DCE Cradle` is 2.2 times slower than `ns-3` native one, but it is 1.3 times faster than NSC-1000.

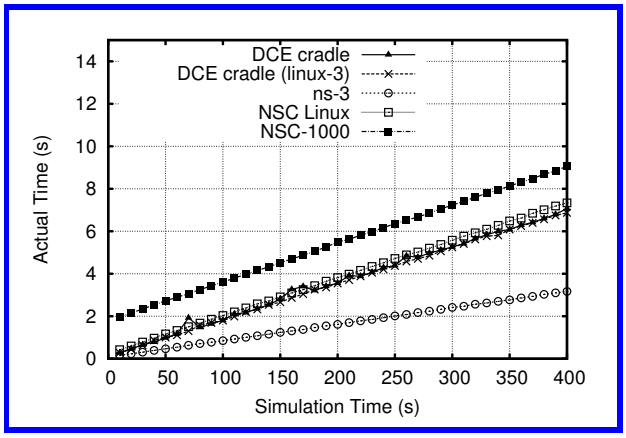


Figure 7: Duration of the actual time in function of the duration of the simulation time.

The slightly high offset observed with NSC is due to the overhead generated for loading a large number of symbols in shared library; indeed about three times more symbols than for DCE is necessary. This is because when the number of instances (i.e., nodes) in the simulation is increased, the parser of NSC generates different symbols for each instance, therefore the number of symbols is also increased, which represents the overhead of library loading. This overhead is remarkable in NSC-1000 as shown in figure 7.

Discussion These two micro-benchmarks capture the considerable overhead of `DCE Cradle` compared to `ns-3`. They show that `DCE Cradle` outperforms NSC while providing a similar level of realism for the simulation results. Further investigations with more simulation scenarios are required, but we believe that from now on, users can benefit from real-code simulation with `DCE Cradle`.

6. A USE-CASE WITH DCCP

While `DCE Cradle` allows us to simulate real protocol code as with NSC, the main objective of this work is to enable users to develop new protocols for `ns-3` without much effort. In this section, we explain how `DCE Cradle` can be used to develop new protocols for `ns-3` with a use case: the Datagram Congestion Control Protocol (DCCP) [5] model with Linux kernel via `DCE Cradle`.

The DCCP was proposed to provide congestion controlled unreliable datagram for real-time applications such as video streaming. Actually, it is a mature protocol as several operating systems already include the experimental DCCP code, and simulation models of the

protocol has already been included for ns-2 and opnet simulators.

Without implementing the 129-pages of DCCP specification [4] from scratch on `ns-3`, we were able to reuse the Linux implementation of DCCP integrated via DCE. We enabled CONFIG_NET_DCCP in DCE-Linux (i.e., ns-3-linux) with a tiny modification in the *glue* code (4 lines of code[7]). Then, we introduced a set of new socket factory classes (i.e., `LinuxDccpSocketFactory`) via `DCE Cradle` to use Linux DCCP implementation over `ns-3` applications.

The above changes are the only ones required to simulate with the actual DCCP code. Unfortunately, this is not always as simple for all network protocols, and sometimes additional enhancements are needed on the glue code. However, this example shows the low cost development of a new protocol to simulate by reusing the Linux code with DCE and `DCE Cradle`.

As a proof of concept with our DCCP implementation, we also conducted goodput measurement using multiple flows in a simple dumbbell topology connected via a bottleneck link at the intermediate routers. All parameters including packet loss ratio, injected traffic, delay, and bandwidth are the same as the ones used in the previous experiments. We used `OnOffApplication` to inject traffic on `ns-3`, and the iperf DCCP extension[8] on two Linux boxes to analyze the real Linux stack (version 2.6.32-28) behavior.

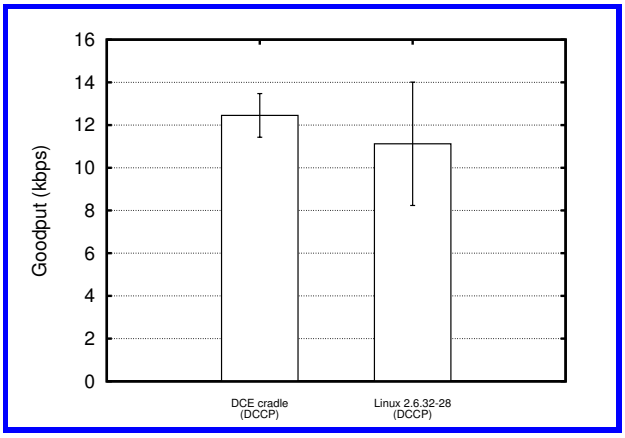


Figure 8: Mean goodput of DCCP with standard deviation from 100 runs.

Figure 8 summarizes the goodput performance measurement results and the standard deviation for 100 replications of the experiment. As we saw with the TCP goodput measurements, DCCP goodput using `DCE Cradle` obtains similar performance compared to that with the real environment.

[7] `http://code.nsnam.org/furbani/ns-3-linux/rev/0f845b1aee21`

[8] `http://www.erg.abdn.ac.uk/~gerrit/dccp/apps/`

7. CONCLUSION

We have proposed `DCE Cradle` to simulate network protocols with existing real code and `ns-3` applications. `DCE Cradle` utilizes the Linux kernel network stack and provides transparent socket interfaces to the `ns-3` applications. Unlike Network Simulation Cradle, `DCE Cradle` can benefit from DCE in terms of kernel code modification, which makes easier the development and validation of new protocols using network simulators. Performance evaluation shows that `DCE Cradle` has a reasonable overhead compared with the `ns-3` native stack, and slightly outperforms NSC in two micro-benchmarks: it is up to 1.3 times faster than NSC while providing a similar level of realism in the results. As a proof a concept, we have shown how easy it is to reuse in `ns-3` an actual implementation of a transport protocol with DCE Cradle. Although further investigation on the performance analysis is required, we believe that `DCE Cradle` can help the network community to develop and test network protocols in an easier and more realistic way.

Acknowledgements

We would like to thank Mathieu Lacage for giving advices on the design of `DCE Cradle`. This research was partially supported by INRIA and the Japanese Society for the Promotion of Science (JSPS) Joint Research Projects program in the context of the Simulbed associated team.

8. REFERENCES

[1] J. Abraham and G. Riley. Simulator-agnostic ns-3 applications. In *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*, WNS3 '12, pages 391–396. ICST, 2012.

[2] G. Carneiro, H. Fontes, and M. Ricardo. Fast prototyping of network protocols through ns-3 simulation model reuse. *Simulation Modelling Practice and Theory*, 19(9):2063–2075, 2011.

[3] 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.

[4] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), Mar. 2006. Updated by RFCs 5595, 5596.

[5] E. Kohler, M. Handley, and S. Floyd. Designing DCCP: congestion control without reliability. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '06, pages 27–38, New York, NY, USA, 2006. ACM.

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