DCE: Test the real code of your protocols and applications over simulated networks

Daniel Câmara[1], Hajime Tazaki[2], Emilio Mancini[3], Mathieu Lacage[4], Thierry Turletti[3], and Walid Dabbous[3]

[1]Telecom ParisTech, France
[2]University of Tokyo, Japan
[3]INRIA, France
[4]ALCMEON, France

Abstract

We present the Direct Code Execution (DCE) environment for `ns-3`, notable for being the first free, open source framework for integrating Linux kernel and application code within a leading discrete-event network simulator. This new approach has many potential advantages over virtual machine-based frameworks in terms of realism, reproducibility, avoidance of real-time execution constraints, configuration management and ability to debug a network-wide experiment from a single address space using common debugging tools. We provide an overview of DCE and illustrate some key features of this framework with two use cases, one involving `thttpd`, an HTTP server implementation, and another one involving the `udp-perf` traffic generator.

1 Introduction

When designing a new network system (protocol or application), testing and verifying the behavior of the developed code is an essential but time consuming task. In order to guarantee the correctness, robustness and reliability of the system under test (SUT), it is important to perform experiments under realistic conditions. Moreover, the experiments should be reproducible, in order to give other researchers/developers the opportunity to track errors in their code and verify the results obtained by other research teams. Also, debugging is crucial to identify possible implementation issues and this task is not trivial, especially when the SUT is running over multiple distributed

nodes. Running code in a deterministic way not only can help in the identification of possible bugs, but is also important to evaluate the impact of changes performed on the SUT.

Today, simulators and emulators are widely used in the networking community to evaluate the performance of new protocols. The two approaches are interrelated, but differ in various points [1].

Network simulators imitate the behavior of real systems and predict the outcome of real experiments by computing the interactions between the different network entities composing network experiments. They are helpful in evaluating in a reproducible way, trends of network protocols by providing various configurable parameters without the burden of deploying and repeating the experiments on a real testbed. However, as they are unable to use a protocol's implementation directly, they have limited utility during the development process of the final code. Furthermore, their realism is limited by the simplifications inherent to the models used.

Network emulators combine real systems and controllable models. Different types of emulators exist, depending on which components are real and which components are modeled: (1) application level emulation replaces the synthetic traffic generation by live applications or pre-captured traffic; (2) protocol-level emulation uses real protocol stacks instead of simulated ones; (3) link-level emulation connects nodes through real wired or wireless links. Most emulators are Virtual Machine (VM)-based and run in real-time. They are therefore limited by the hardware capacities of the computer used for emulation. Furthermore, debugging an emulated distributed network protocol is complex as different nodes of the scenario run in different processes; additionally, the behavior of the network stack is not completely reproducible because it depends on the virtual machine scheduler and the host's available resources.

In this article, we present the Direct Code Execution (DCE) [2] framework, which benefits from the advantages of both simulation and emulation, while limiting the main drawbacks of these two approaches. To the best of our knowledge, DCE is the only free open source framework that evaluates real implementations of network systems in a scalable and reproducible manner, and also allows easy debugging within a deterministic and reproducible environment. More precisely, DCE can run unmodified network systems written in C/C++ on top of ns-3, a widely-used open source discrete event networking simulator. By using the ns-3 simulated time clock thus removing the real time execution constraint, it can scale to complex scenarios. Such scenarios will take longer than they would in a real experiment, but the results will be accurate. DCE allows to simulate with a Linux network stack

in user space. It also ensures that multiple nodes involved in the same ns-3 experiment can use independent network protocol instances without interfering with each other. Finally, DCE can be used with popular debugging tools such as valgrind and gdb to debug distributed network protocols.

In the following sections, we provide an overview of the Direct Code Execution framework. Then we describe how to use it, illustrate its functionalities with two use cases and discuss the related work.

2 Overview of DCE

The goal of DCE [2] is to enable the execution and debugging of real implementations of network protocols and applications. DCE enables unmodified protocol/application implementations, as well as the unmodified Linux network stacks to be run within the ns-3 network simulator with the caveat that using a real network stack requires more memory and processing power than running the same experiment with the ns-3 native TCP/IP stack. This characteristic of DCE improves the realism of network simulations by allowing the SUT to be tested using "real code" rather than just simplified implementations or models of the system. By real code we mean implementations of protocols and applications used in production in real world setups. Moreover, it makes it possible to experiment with the full range of protocols and applications available on standard Linux systems. DCE allows to benefit from ns-3 support of technologies such as LTE, WiFi and WiMax, and of seamless transitions between these technologies in an easy, controllable, repeatable and inexpensive way. Figure 1 illustrates the interactions between DCE, ns-3 and high layer protocols/applications.

Figure 1

To allow the integration of ns-3 simulations with a kernel network stack, DCE encapsulates the stack into a user space library using a single-process model, i.e. all the experiment takes place within a single OS process. In particular, DCE controls the execution of protocols and guarantees reproducibility of large scale experiments. Reproducible experiments can be conducted using DCE because it uses simulated clock instead of wall clock; the simulated clock ensures a deterministic behavior for each experiment. Furthermore, as DCE uses a single-process model approach, i.e. all network protocol instances are inside the same process, it is an attractive approach to debug network communication protocols distributed on multiple nodes with a popular debugger such as gdb.

With DCE, the fact that the traffic is passing through a simulated network instead of a real one, is transparent to both the SUT and the Linux

stack. The ns-3 simulator offers a widely configurable network environment. For instance, it can be used to define possibly complex network topologies using a mix of different technologies, e.g. Ethernet, LTE, Wi-Fi and WiMax. Nodes can be static or mobile, and run a variety of routing protocols and applications. Ns-3 also enables the injection of real traffic inside the simulated network. Additionally, it is possible to perform hybrid experiments, where part of the nodes uses the default ns-3 network stack and another part uses a real-world Linux stack.

As ns-3 scenarios involve the simulation of multiple nodes, DCE must ensure that multiple instances of the same protocol implementation can be run on the same physical machine, and that, consequently: (1) Different instances of the protocol do not share global and static variables; and (2) System calls are captured and processed by DCE, so that when the simulated program calls, for example, the system function *gettimeofday()*, DCE has to return the simulated time instead of the real wall-clock time.

DCE is composed of three layers, namely, core layer, kernel layer and POSIX layer.

- **The core layer** implements minimized virtualization primitives to synchronize and schedule every simulated process from the ns-3 simulator event-loop in a single process. It includes a smart loading mechanism to instantiate the global variables, once for each simulated instance. This ensures that multiple processes can share the same underlying physical pages instead of having their own set of unshared modified pages. This is important as it minimizes the experiment memory footprint.

- **The kernel layer** enables embedding an unmodified real world Linux TCP/IP stack within ns-3. The bottom of the Linux TCP/IP stack interacts with the MAC/PHY layers simulated in ns-3, whereas the top of the TCP/IP stack is exported to ns-3 as a single function allowing to map application-level socket function calls with their kernel equivalents.

- **The POSIX layer** re-implements the subset of the standard socket APIs (glibc) used by the simulated applications. This was required to enable the capture of the relevant system calls and treat them accordingly.

Further details on the inner workings of the DCE architecture are available in [2].

DCE does not impose any hard limit on the size of the simulations performed. It provides accurate results whatever the scale of the experiment. However, the execution time and memory requirements increase with the complexity of the scenario. These limitations can be overcome since ns-3 allows parallelization of simulations over multiple computers, aka Message Passing Interface (MPI)-based distributed simulations. For instance, in a recent study Renard et al. [8] simulated a scenario composed of 360,448,000 ns-3 nodes using a cluster of 176 computers. However, note that an MPI-based simulation does not run in a single process, and so is more complex to debug.

3 DCE in action

In this section, we demonstrate how DCE can be used to analyze or extend a network protocol, debug errors, reproduce and scale the experiments. The detailed results, and instructions to reproduce the results, are available by clicking the figures. These results are also available at `http://yans.pl.sophia.inria.fr/trac/DCE/wiki/dcehttp`.

3.1 Integrating a new protocol in DCE

To enable POSIX and Linux kernel support, DCE uses some API-specific glue code to let interoperation of different parts of the code. Note that the amount of glue code required to run unmodified protocol implementations is relatively higher for DCE compared to other lower-level CPU virtualization technologies. New protocol implementations that attempt to use previously un-implemented APIs need extra work. However, in practice, as our coverage of the POSIX API increases, the probability of needing a missing function decreases. Indeed, DCE already supports hundreds of POSIX functions, and examples of tested applications over DCE include Quagga, iperf, torrent, thttpd, CCNx and various Linux kernel versions (from 2.6.36 to 3.12 versions). For example, in the scenario shown in Section 3.3.1, we study the performance of the http protocol using two widespread tools: `thttpd` and `wget`. The integration of these tools in DCE was done with virtually no code modification as the corresponding POSIX functions were already supported in DCE. To run the executables inside the DCE sandbox, a user needs to associate them to a specified ns-3 topology, set their command line parameters and schedule their execution at a defined time.

All the necessary steps to integrate and run a new protocol are detailed

in the DCE manual.[1]

3.2 Debugging with DCE

DCE can encapsulate a network stack into a user space library within the ns-3 network simulator, and the whole simulation runs as a single process. This feature is particularly useful to debug a distributed network stack running on multiple nodes because debugging tools such as `gdb` and `valgrind` can be used with DCE.

In [2] we explained in detail how we used DCE and valgrind to detect two Uninitialized Data Access errors on version 2.6.36 of the Linux kernel. In particular, we used `valgrind` on a series of ns-3 test suites we developed to test the Linux network stack. Note that the errors we detected still exist in the kernel 3.12, the current version of the Linux kernel.

In Figure 2 we present a DCE debugging session using the Eclipse IDE, where we set a breakpoint in the `thttpd` binary.

3.3 Using DCE to conduct reproducible experiments

In the following, we provide two examples of using DCE to conduct reproducible experiments. The goal of our first experiment was to demonstrate the realism of DCE. In particular we compare the results obtained by using DCE with the results obtained by conducting the same experiment using Mininet-HiFi [5], another network emulation tool, and also by conducting the experiment in a real network. The goal of the second experiment was to evaluate the performance of DCE and its capacity to perform automatic time dilation, where simple scenarios may run faster than real time and complex scenarios slower, without any impact on the accuracy of the experiment results.

3.3.1 Realism of DCE experiments

We demonstrate the realism of DCE with the help of an experiment in which we study time evolution of TCP throughput during an HTTP file transfer. For our experiment we use a simple two nodes topology, depicted in Figure 3a, to transfer a 2 GB file using the HTTP protocol. The two nodes are directly connected through an 1000BASE-T Ethernet link. An `thttpd` server is deployed at node A and the 2 GB file is requested with `wget` from a client at node B. The two nodes used for the real testbed are Intel Core

[1] DCE manual: `http://www.nsnam.org/docs/dce/manual/html/index.html`.

i5 processor machines with 4 GB of RAM and run the Ubuntu Linux 12.04 distribution with kernel Linux 3.4.0. One of these nodes is used to perform in turn the DCE and the Mininet-HiFi experiments.

To evaluate the realism of DCE experiments, we compare the HTTP throughput obtained with DCE using a ns-3 simulated Ethernet link with the throughput obtained in the same scenario running in a real testbed on top of a the real Ethernet link.

We also perform the same experiment with Mininet-HiFi [5], which is one of the most promising real time network emulation tool available, see Section 4 for further comparison between the Mininet-HiFi and DCE approaches.

In the three cases (real testbed, DCE and Mininet-HiFi), the same `thttpd` and `wget` code and the same TCP stack are used for running the experiments. Note that it is possible with DCE to select a specific network stack, from ns-3 or from one of the supported Linux kernels. Note also that the `thttpd` server and `wget` software are run without any source code modification in all experiments.

Figure 4 shows the throughput of the file transfer with `thttpd`/`wget` over the real Ethernet link, integrated in DCE over an ns-3 simulated network and run in a Mininet-HiFi virtual machine on top of an emulated Ethernet link. Note that to minimize the bias caused by the massive I/O operations, the `wget` output is not saved on the disk of the client machine. In Figure 4a, we show the performance obtained when the link speed is limited to 100 Mbps. We can observe that the results are very similar for the three different cases. The average deviation obtained from the real network transmission is about 0.86% for DCE and 1.6% for Mininet-HiFi. This means that both DCE and Mininet-HiFi approaches are representative and able to accurately emulate the real network behavior in this case.

Figure 4

Figure 4a

Figure 4b shows the performance obtained when the link speed is not limited, i.e. the 2 GB file is transmitted at the maximum speed. We observe that the transmission rate with the real Ethernet link is close to 1Gbps, the limit for a 1000BASE-T Ethernet link. In this case, DCE still presents accurate results, with an average deviation of 1.41% from the real data transmission. However, we observe that Mininet-HiFi does not provide meaningful results (with 54.5% deviation observed from the real case). On inspecting the traces and Mininet logs, we observe that the decrease in throughput is because of packet losses that take place when the machine is overloaded by the execution of Mininet-HiFi. Indeed, container-based emulation approaches such as Mininet-HiFi are bounded by the computational resources available in the emulation machine. Mininet-HiFi should be used jointly with a monitoring

Figure 4b

tool to check that enough CPU is available while running the experiment. DCE does not have such limitations and can scale to more complex scenarios as it uses the ns-3 simulated time.

Using a simulation environment such as DCE/ns-3, the time experienced by the SUT inside DCE is elastic, and may expand or contract to process all the events in the scenario. With this automatic time dilation, low CPU usage experiments may run faster than they would in reality, while CPU-greedy ones would run slower, but this has no impact on the accuracy of the results.

3.3.2 Scalability and flexibility of DCE

In this second example, we chose a Daisy chain topology, shown in Figure 3b, to demonstrate the scalability and flexibility of DCE. Basically, we run the `udp-perf`[2] application in DCE on the daisy chain nodes that are connected by Ethernet links. An udp-perf server placed on the first node (Node 0) transmits packets at a constant bit rate to a receiver located on the last node of the chain (Node N). The network link capacity is set to 1Gbps and the sending bit rate is set using udp-perf. We analyze the execution time required in DCE to run the simulation for different values of CBR and different number of hops. We ensure that the CBR, set with udp-perf, is respected and that the client does not experience packet loss.

We run different scenarios by varying the number of hops (from 8 to 255) and the CBR values (5, 100, 500 and 1000 Mbps). The definition of the scenario for DCE is a simple C parameterized script, with the same format characteristics of the ones used in standard ns-3 simulations. This flexibility is convenient when, as in this case, we need to run several times the same experiment, but just changing a few parameters. All these scenarios correspond to a 60 seconds transmission over a real network.

Figure 5a shows the real time (CPU time) required to run the different scenarios using DCE. We can observe that when we increase the number of hops or the sending rate, the simulation takes longer to complete. Indeed, when we increase the number of hops or the sending rate, we increase the number of events DCE/ns-3 has to handle. We can note that low usage CPU experiments run faster than real time. Recall that the same scenario on a real network will take 60s, represented by the black horizontal line in Figure 5a. For example, DCE performs the scenario with 8 nodes at 5 Mbps CBR in about 0.75 second. On the other hand, the scenario with 255 nodes and

Figure 5a

Figure 5a

[2]The `udp-perf` source code is available at URL `http://code.nsnam.org/ns-3-dce`.

1Gbps takes about 5990 seconds to be executed, so nearly 100 times more than real time. The network behavior is the one we would expect from this experiment, no packet loss is experienced and the data rate are respected, as it would happen in the real setup, but without the painful need to set up of a 255 daisy chained network.

Figure 5b show for the corresponding number of packets processed per second (pps) for each scenario. This number is computed by dividing the number of processed packets by the CPU execution time used to run each scenario. We observe that DCE processes the maximal number of packets for networks including between 32 and 80 nodes. This happens because the experiment setup time (e.g. libraries load, memory allocation) does not change significantly between small and large experiments. Thus, this nearly constant initial time, where packets are not being treated, has a higher impact on the performance of faster experiments.

Figure 5b

4 Related Work

Network virtualization frameworks such as **NetKit** [3] and **Cloonix** [4] are based on User Mode Linux (UML). They help researchers and developers design and validate network protocols and kernel network stacks with configurable network setups. These ephemeral network setups, which only last the duration of the experiment, have many advantages over real testbeds, including lower costs, simple setup procedures, and integrated experimental results collection. However, debugging a protocol using these frameworks is just as complex as debugging a distributed system over multiple machines. Moreover, as they use fully virtualized Linux machines, this strongly limits the maximum number of nodes in the experimental scenario. DCE shares the protocol stack with all nodes, saving resources and improving the capacity of the framework to describe larger network scenarios.

Mininet-HiFi [5] is based on light-weight virtualization technology that enables a single emulator to run a large number of virtual machines, when compared to NetKit and Cloonix. However, as demonstrated on section 3.3.1, because it must be run in real-time, this framework cannot support hosts exchanging data at high throughput. Mininet-HiFi is limited by the CPU resources available on the emulation machine. DCE, by contrast, uses the simulation time of ns-3 and encapsulates the kernel communication stack into a user space library as a single-process model. This makes the approach scalable and allows to easily debug distributed network protocols.

SliceTime [6] provides speed adjustment for a real software prototype running in a VM on top of a simulated network topology. A synchronizer controls the execution of the network simulation and the software prototypes and interrupts the execution of the prototype or the simulation when necessary to achieve precise clock alignment. SliceTime provides better scalability and reduces variability in the execution. The approach is different than DCE that directly integrates real network or application software in the `ns-3` simulator. In particular, DCE provides implicit synchronization between the software and the simulated network topology as only the simulation time is used, which makes also the framework scalable.

Network Simulation Cradle (NSC) [7] is the ancestor of DCE. It parses and transforms different operating system's network stacks (such as FreeBSD and Linux) into new C files compiled and linked with shared libraries used in a network simulator. Both NSC and DCE use the network simulator's virtual time and facilities to provide a wide range of network environments. However, as NSC relies on a language-dependent source-level parser, its usage is restricted to the validation of TCP protocols. This is not the case with DCE, which uses carefully designed abstractions of network devices and time-related kernel API.

To the best of the authors' knowledge, DCE is the only framework that provides experimental reproducibility and scalability, as experiments are no longer bound to run in real time, and easier debugging, which derives from controllability of the single-process model, discussed in Section 2.

5 Conclusion

There is a growing need in the network community for more realistic evaluation environments. DCE enables developers and researchers to develop their protocols and applications in a fully controllable and deterministic environment, where tests can be repeated with reproducible results. This open source framework allows unmodified protocol implementations and application code to be tested over large and possibly complex network topologies through ns-3, a leading discrete-event network simulator. The single-process model used in the DCE virtualization core brings key features, such as the possibility to easily debug a distributed system over multiple simulated nodes without the need of a distributed and complex debugger.

Acknowledgments

This research has been 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.

The authors thank the anonymous reviewers for their comments and *Ashwin Rao* and *Marc Mendonca* for their proofreading on this paper.

References

[1] I. McGregor, "The relationship between simulation and emulation", Proc. of Winter Simulation Conference (WSC'02), San Diego, CA, December 2002.

[2] H. Tazaki, F. Urbani, E. Mancini, M. Lacage, D. Camara, T. Turletti, W. Dabbous, "Direct Code Execution: Revisiting Library OS Architecture for Reproducible Network Experiments", Proc. of ACM CoNEXT, Santa Barbara, CA, December 2013.

[3] M. Pizzonia, M. Rimondini. "Easy Emulation of Complex Networks on Inexpensive Hardware", Proc. of Tridentcom, Innsbruck, March 2008.

[4] Cloonix: dynamical topology virtual networks, `http://clownix.net/`, [Accessed Nov. 12th, 2013].

[5] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation", Proc. of ACM CoNEXT, Nice, France, December 2012.

[6] E. Weingartner, F. Schmidt, H. Lehn, T. Heer, K. Wehrle, "SliceTime: a platform for scalable and accurate network emulation", Proc. of ACM NSDI, Boston, MA, April 2011.

[7] S. Jansen, A. McGregor, "Simulation with real world network stacks", Proc. of the 37th Winter Simulation Conference (WSC), Orlando, FL, December 2005.

[8] K. Renard, C. Peri, J. Clarke, "A Performance and Scalability Evaluation of the ns-3 Distributed Scheduler", Workshop on ns-3 (WNS3), Desenzano del Garda, Italy, March 2012.

Biography

Daniel Câmara is a research engineer at Telecom ParisTech, France. He holds a Ph.D. in Telecommunications from Telecom ParisTech and a PhD in Computer Science from Federal University of Minas Gerais, Brazil. His research interests include bio-inspired algorithms, vehicular networks, routing, data dissemination and topology management protocols for wireless networks.

Hajime Tazaki is a Lecturer of University of Tokyo. He received his Ph.D.from Keio University in 2011 and then worked as a senior researcher of National Institute of Information and Communications Technology (NICT), Japan. He is interested in distributed network system in general, and software and network architecture.

Emilio P. Mancini is a research fellow at INRIA Sophia Antipolis, France. He has a Ph.D. and an M.S. in Computer Science from Università degli Studi del Sannio-Benevento. His research interests are in the areas of autonomic and parallel computing. He has been working on, French, Italian and European projects on grid computing, mobile services and simulation of parallel systems.

Mathieu Lacage is CTO at Alcmeon where he leads their efforts in building a scalable Semi-Automatic Q&A Engine. Previously, he served as Software Lead for ns-3 in the PLANETE team at the INRIA. He received a M.S. (2001) from Telecom ParisTech and a Ph.D. in computer science from the University of Nice - Sophia Antipolis, France (2011).

Thierry Turletti is a senior research scientist in the DIANA team at INRIA. He received the MSc and the PhD degrees in Computer Science from the University of Nice - Sophia Antipolis, France. His research interests include software defined networking, trustable network evaluation platforms and wireless networking. He is currently serving in the editorial boards of the Wireless Networks and Advance on Multimedia journals.

Walid Dabbous is a senior researcher at INRIA where he is a currently leader of the DIANA team. His research interests include Future Internet Architecture and Protocols, Networking Experimental Platforms and Simulators, Experimental Methodology for Networking Protocols.

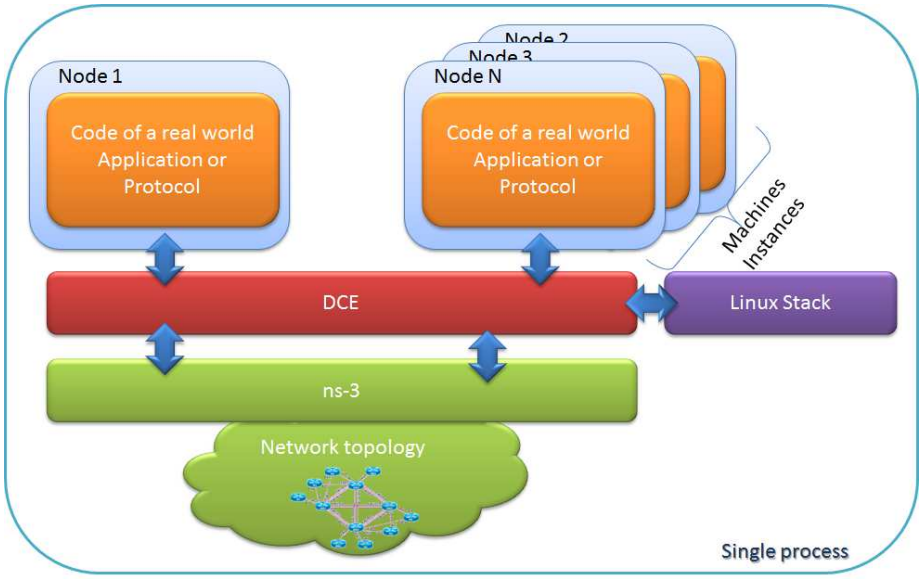


Figure 1: High level interactions between the main components of the DCE/ns-3 framework

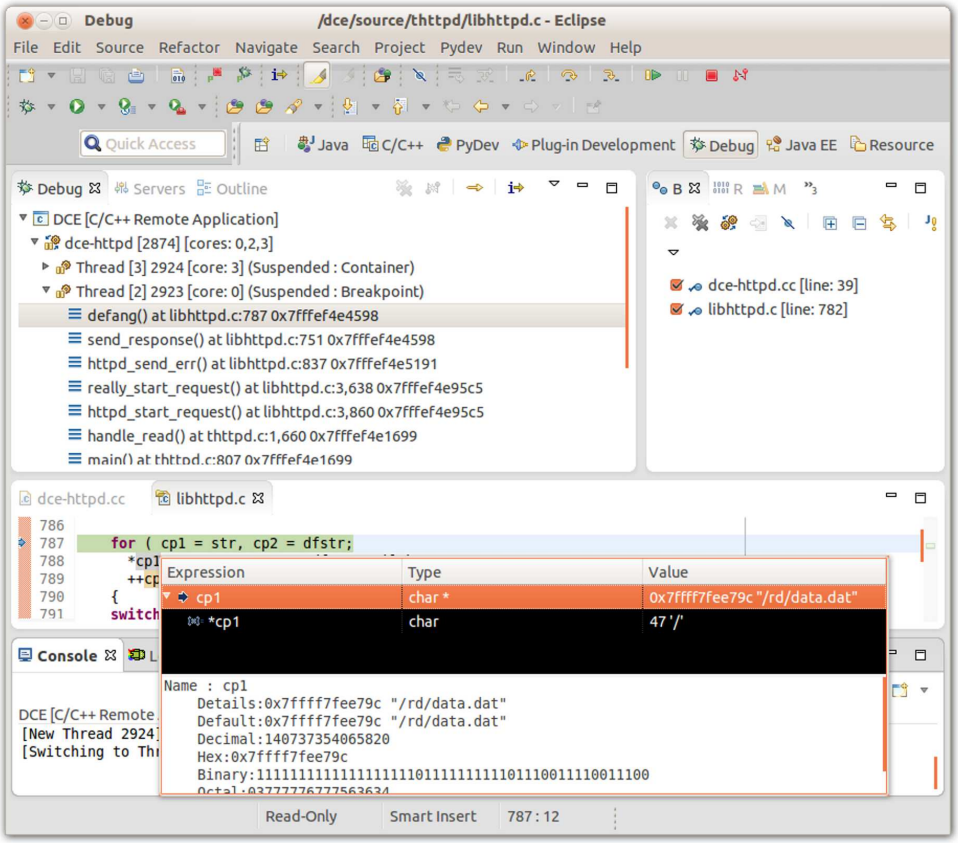


Figure 2: A snapshot of a DCE debugging session: after stopping to a breakpoint (central area), the backtrace (top left area) and a variable value (bottom pop-up) are shown.

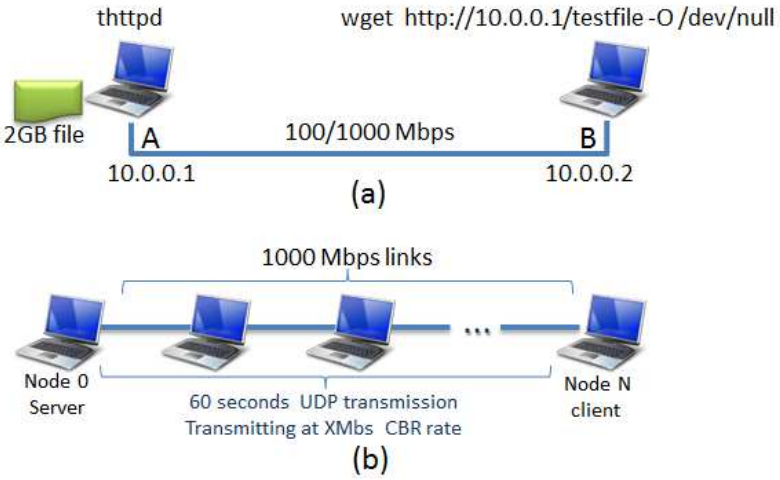
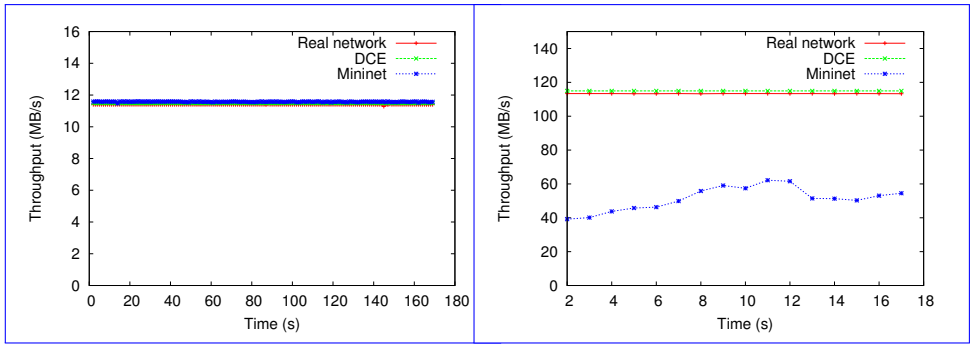
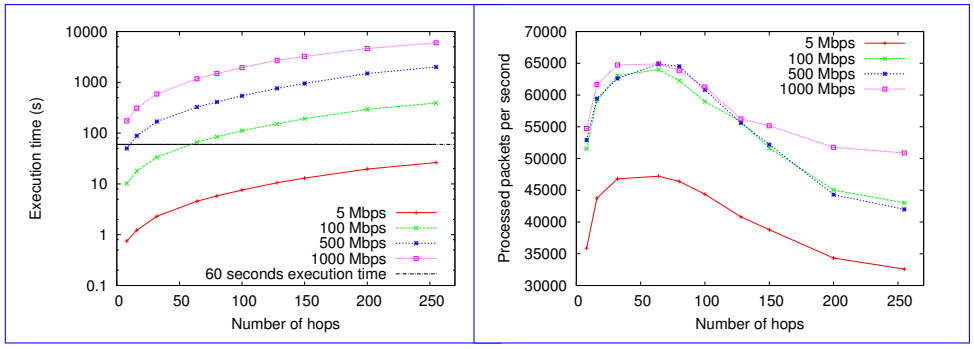


Figure 3: The networks used on the evaluation of DCE. (a) represents the topology used in the realism of DCE experiment, with the `thttpd` server deployed on node A and `wget` on node B. (b) represents the topology used in the scalability and flexibility of DCE experiment, a Daisy chain network where we perform a 60 seconds CBR transmission between the client and server, deployed on the edges of the network.



(a) 100 Mbps (b) 1 Gbps

Figure 4: Throughput of an HTTP transmission over 100 Mbps (a), and 1 Gbps (b) links with the different approaches



(a) CPU execution time (b) Average pps processed

Figure 5: 60 seconds CBR transmission over a daisy chain network