Proceedings of the 2017 Winter Simulation Conference
W. K. V. Chan, A. D'Ambrogio, G. Zacharewicz, N. Mustafee, G. Wainer, and E. Page, eds.

SIMULATING NETWORKS WITH NS-3 AND ENHANCING REALISM WITH DCE

Jared S. Ivey

402 Software Maintenance Wing
Warner Robins Air Logistics Complex
Robins AFB, GA 31098, USA

Brian P. Swenson

Information and Communications Laboratory
Georgia Tech Research Institute
Atlanta, GA 30318, USA

George F. Riley

School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332, USA

ABSTRACT

Communication networks are constantly evolving with new technologies providing greater quality, resiliency, and security to the data traversing current networks. In this ever-changing field, simulation provides an avenue for examining the traffic within new or existing networks. In simulating a network, characteristics and metrics of the topology may be derived without interfering with the existing framework or incurring an immediate hardware or software cost. The popular network simulator ns-3 is an effective tool for studying these network behaviors. This talk presents an overview of ns-3, discussing its design as a discrete event network simulator and its capabilities. Code snippets will be examined, demonstrating how to configure a network topology in simulation, generate packet traffic to traverse the simulated network, visualize the network behaviors, and glean metrics from the simulation. A subproject of ns-3, Direct Code Execution (DCE), is also described, demonstrating a mechanism for deploying real-world applications within the simulation.

1 INTRODUCTION

Network simulation employs representative models based on real-world communication network concepts and paradigms to reproduce realistic behaviors and statistics. It aims to accomplish this task without the required costs of constructing equivalent testsbeds with hardware. This tutorial discusses the design and usage of the network simulator *ns-3*. This popular simulator began initial development in 2006 and was first released in June 2008(ns-3 Project 2017). It employs a discrete-event scheduler to permit researchers and educators to construct network topologies, generate traffic to traverse the simulated network, and gather statistics based on the effects of manipulating characteristics of the network. This tool operates at the packet level, processing individual packets as distinct entities as they move through a network. This design differs from some flow-based network simulation tools which monitor flows of packets with similar header attributes. This design decision ultimately weighs the trade-offs of specificity and speed. In this regard, ns-3 shows preference toward the former.

ns-3 should not be construed as an extension or update to the network simulator *ns-2*(Breslau et al. 2000) as the two projects are distinct and not directly compatible with one another. Although ns-3 indeed borrows some design concepts from ns-2, it has also been heavily influenced by other networks simulators such as the Georgia Tech Network Simulator (GTNetS)(Riley 2008) and Yet Another Network Simulator

(YANS)(Lacage and Henderson 2006). Similarly, ns-3 is not related to GNS3(GNS3 Technologies Inc. 2017), a graphics-based network emulator that permits integration of network kernel images on nodes.

The reliability of the results produced by a network simulation tool such as ns-3 is ultimately determined based on the adequacy of the implementations of the models on which it is based. These models are constructed based on the requirements of various network protocols and standards as well as the models of underlying devices and communication channels. In this way, the realism of such simulators can be limited if simplifications in the models are deemed necessary. Additionally, the time required to design, construct, and validate these models can restrict the toolset, especially when one considers the rapid evolution of the modeled technologies. This limitation motivates an additional module of ns-3 called Direct Code Execution (DCE). This module permits the execution of real-world network applications and protocols directly in the simulated environment instead of attempting to design adequate representations or models.

This tutorial introduces both ns-3 and DCE through the following sections. Section 2 discusses some of the underlying concepts behind the design of ns-3. It specifically describes discrete event simulation, build tools for the source code and dependencies, and the general design of both the simulator core and the components required for simulating a network topology. Visualizing simulated networks and their behaviors is useful toward debugging and better understanding them. As such, Section 3 examines some of the visualization tools that ns-3 provides. DCE is discussed in terms of its architecture and application in Section 4. An example requiring the use of both ns-3 and DCE is studied in Section 5. Finally, the conclusion of this tutorial provides links to additional exploration and information in Section 6.

2 ARCHITECTURE

This section examines the basic architecture of ns-3. It first describes the concept of discrete event simulation as it pertains to the network simulator, discusses the build systems that ns-3 employs, and then describes the building blocks of the simulation architecture.

2.1 Discrete Event Simulation

A discrete event simulation models a system in such a way that changes to its state occur at discrete points in the simulation time. This concept differs from a continuous simulation which monitors system state continuously while the simulation process itself regularly progresses its simulated time. Functions will schedule events to occur at a future time relative to the simulated time at which the function was executed. This scheduling operation is most generally articulated through either the `Simulator::Schedule` or `Simulator::ScheduleWithContext` API. As events are scheduled and thus included in the list of events to be processed, the ns-3 event scheduler automatically orders them such that the simulated time can progress in a chronologically appropriate manner. The ns-3 baseline provides the ability to choose between four data structures for managing and ordering the events: map (*default*), list, heap, and calendar. Initial commencement of the simulation occurs through the `Simulator::Run` function which triggers the initial processing of events in the event list. Subsequently, the simulation may be stopped either manually by scheduling a `Simulator::Stop` event or automatically by allowing the event list to become empty.

2.2 Build Systems

For configuring and building the source code, ns-3 uses the Waf build automation tool(T. Nagy 2017). Waf provides a Python-based framework for configuring, compiling, and installing software applications. It aims to simplify these processes under its singular framework by allowing processes to execute any compilation and scripting distinctly, as necessary, and in parallel. Waf also provides basic compiler and tool configuration support. Waf allows the modularization of the ns-3 source code through *wscript* files that can be discovered in the ns-3 directory hierarchy and can be collected for building by the main `waf` script. This design permits only the necessary components for a particular simulation to be built.

Listing 1: Sample Waf script (wscript) for the CSMA module in ns-3.

```
def build(bld):
    obj = bld.create_ns3_module('csma', ['network', 'applications'])
    obj.source = [
        'model/backoff.cc',
        'model/csma-net-device.cc',
        'model/csma-channel.cc',
        'helper/csma-helper.cc',
        ]
    headers = bld.new_task_gen(features=['ns3header'])
    headers.module = 'csma'
    headers.source = [
        'model/backoff.h',
        'model/csma-net-device.h',
        'model/csma-channel.h',
        'helper/csma-helper.h',
        ]

    if bld.env['ENABLE_EXAMPLES']:
        bld.add_subdirs('examples')

    bld.ns3_python_bindings()
```

Interacting with Waf typically involves the use of three distinct command arguments. To configure particular environment variables or specific ns-3 modules, the command `./waf configure` is used. If a wscript file is modified, ns-3 will need to be reconfigured with this command. For the source code to be compiled and linked, the `./waf build` command (or simply `./waf`) is called. Waf will build all of the necessary modules and dependencies based on the specified configuration and then display those modules upon successful completion. To actually execute an instance of network simulation complete with topology creation and traffic generation, referred to as an ns-3 *script*, one supplies the command `./waf --run` (or `./waf --pyrun` for Python scripts) along with the name of the script to be run.

Listing 1 displays the wscript for the CSMA module of ns-3. From the listing, the process for creating the module and specifying its source code and header files can be seen. Also, one can see how additional functionality can be described, either by specifying certain environment variables during configuration, traversing further subdirectories in order to discover additional wscript files, or calling other Python scripts contained in other objects.

Above ns-3 and Waf in terms of directory hierarchy is an extra layer of build automation that can be used by way of the Python-based *bake* tool(INRIA 2012). Bake provides the capability to specify additional configurable options and remedy additional dependency requirements through an XML-based specification file. In this way, external software libraries can be retrieved and installed for local use by ns-3. This functionality is especially useful for DCE as it resides external to the baseline ns-3 source code and must be configured appropriately to recognize and employ the libraries that ns-3 creates. Bake can perform all of the necessary configurations and installations automatically, hiding these complexities from the user such that he/she can simply use the software. Like Waf, bake requires an initial configuration step in order to specify both the configuration file that will be used and a specific configuration set in that file that will be built. Unlike Waf, bake next requires a download step that will retrieve any and all necessary

Listing 2: Sample script describing the simulator. Excerpt of source code from https://www.nsnam.org/docs/release/3.26/doxygen/sample-simulator_8cc_source.html

```
#include <iostream>
#include "ns3/simulator.h"
#include "ns3/nstime.h"
#include "ns3/command-line.h"
#include "ns3/double.h"
#include "ns3/random-variable-stream.h"
using namespace ns3;
...
int main (int argc, char *argv[])
{
  CommandLine cmd;
  cmd.Parse (argc, argv);
  MyModel model;
  Ptr<UniformRandomVariable> v = CreateObject<UniformRandomVariable>();
  v->SetAttribute ("Min", DoubleValue (10));
  v->SetAttribute ("Max", DoubleValue (20));
  Simulator::Schedule (Seconds (10.0), &ExampleFunction, &model);
  Simulator::Schedule (Seconds (v->GetValue ()), &RandomFunction);
  EventId id = Simulator::Schedule (Seconds (30.0), &CancelledEvent);
  Simulator::Cancel (id);
  Simulator::Run ();
  Simulator::Destroy ();
}
```

source code packages. Once all of these various packages have been downloaded, bake can then be used to build and install them in a way that ns-3 and/or DCE can make use of them.

2.3 Simulator Core

The ns-3 simulator core integrates a number of components to provide the key aspects of a discrete event simulator: time, events, and the simulator and event scheduler themselves. Additionally, the interfaces for command line arguments and random variables assist in providing a more dynamic nature to an ns-3 simulation both programmatically and analytically. Listing 2 provides a simple example detailing some of the commands required when interacting directly with the simulator.

Time within the context of ns-3 represents either a relative or absolute time within the simulator. When referring to time in simulation, a distinction is made between real time, referred to as wallclock execution time, and simulation time, an abstraction modeling the progression of time in terms of the events processed by the simulator. The Time object in ns-3 implements the latter of these two concepts. To avoid floating point discrepancies across various hardware and operating system platforms, the simulated time is stored as a large integer. Manipulation of the simulated time occurs through operations across the ns-3 Time objects. Most standard operators are overridden to permit interactions with Time objects similarly to integers or floating point values. The default time resolution is nanoseconds, but representation of days, hours, minutes, and seconds is allowed as well. Time resolution is permitted to femtoseconds (10^{-15}). Furthermore, although the underlying value is stored as an integer, the API of the ns-3 Time class allows values to be set with and exported as floating point values.

Events in ns-3 are simply function calls that will be processed at a specific simulated time. In this way, the events act as callback operations that will be triggered in relation to some other occurrence in the simulation. As shown in Listing 2, events can be scheduled with the `Simulator::Schedule` API. The event specifies the relative simulated time when it should be processed, a function or callback that should be executed when the event is processed, and any input variables required based on the function signature. The relative simulated time provided to the `Schedule` function represents a time based on the current simulation time at which the function is called. In the case of Listing 2, the simulated time is zero since `Simulator::Run` has not yet been called, and the resulting scheduled times will occur in simulation at their specified times in the future. Additionally, ns-3 events possess identifying values to allow for status checking or removal from the scheduler, as evidenced by the `Simulator::Cancel` function.

As previously discussed briefly in Section 2.1, the `Simulator` class comprises the mechanisms and data structures for implementing the discrete event simulator for ns-3. By default, events are maintained in a map and can be added in chronological order in terms of simulation time through either the `Simulator::Schedule` or `Simulator::ScheduleWithContext` methods. As Listing 2 shows, a number of operations beyond event scheduling are available under the `Simulator` class. Of particular importance is the `Simulator::Run` function which initiates the program loop through which events can be processed. For this particular example, only two events will be processed (since one of the `Schedule` operations is accompanied by a call to `Cancel`). Once these two events have been processed within the `Run` loop, the simulator will exit the program loop on its own. Alternatively, the `Simulator::Stop` function may be used to specify a relative time in simulation when the simulator should be stopped. Following completion of the `Run` operation, the call to `Simulator::Destroy` is used to destruct the employed object and clean up memory used by the simulator.

Besides the default simulation scheduler, ns-3 provides some alternatives to consider when interacting with simulated topologies. A “Real Time” simulator implementation is available when the simulation time needs to align with the wallclock execution time. The requirement for simulation time to coincide with real time is most evident when the simulator is intended to operate as more of an emulation. This capability in turn allows the simulated topology to interact with devices outside of ns-3 and the simulated environment through either its File Descriptor (FD) or Tap Bridge network devices. These device objects can be configured and used by ns-3 to send simulated packets to destinations outside of the simulation. As of ns-3.8, a distributed simulator implementation has also been available to mitigate scalability issues related to simulating significantly large topologies(Pelkey and Riley 2011). In this initial version, the mechanisms underlying this implementation have been based on the Distributed Snapshot algorithm(Mattern 1993). In ns-3.19, an additional distributed simulator option has been provided that employs null message synchronization derived from the Chandy-Misra-Bryant (CMB) algorithm(Chandy and Misra 1979, Bryant 1977).

Command-line arguments are handled through the `CommandLine` class, permitting dynamic configuration of script parameters at runtime. As shown in Listing 2, following creation of the `CommandLine` object, the `Parse` method handles the resolution of command-line arguments by analyzing the inputs to the main program. What is not shown in the listing but will be covered in Section 5 is that an additional set of steps between creation of the `CommandLine` object and calling `Parse` is required to specify valid command-line arguments. This operation is handled through the `AddValue` method.

Random variables in ns-3 augment a simulation script in an effort to better model the stochastic nature of real-world network behaviors while doing so in a manner that can be adequately reproduced. As Listing 2 demonstrates, random variables for ns-3 are provided through the creation of an ns-3 smart pointer `Ptr` object that holds a reference to a pseudo-random number generator through which random values are derived. Attributes such as the minimum and maximum values of the uniform random variable in the listing are manipulated through the `SetAttribute` method in the same way that attributes of other ns-3 `Ptr` objects are handled. Requests for a value from the random variable stream employ the `GetValue` function which will return a floating point value that can in turn be introduced to a `Time` object. (Values may also be acquired as 32-bit integers with the `GetInteger` method.) This process then allows the event scheduler to receive events with a certain level of randomness, enhancing the realism of the simulation. With no additional configuration, the same values will be produced from the random variable stream each time the simulation is executed. However, modification of the random variable seed or run parameters can be introduced(ns 3 project 2017). This manipulation prevents the results from different executions of the simulation script from being identical and allows statistical analyses to be performed across the collected datasets. Recording the seed and run values during experimentation also permits others to reproduce the observed results for verification. The ns-3 baseline currently provides the following selection of random variable models: constant, deterministic, empirical, Erlang, exponential, gamma, log-normal, normal, Pareto, sequential, triangular, uniform, Weibull, zeta, Zipf.

2.4 Network Components

Figure 1 provides a diagram of the components that constitute a simulated topology in ns-3. A `Node` object in ns-3 models a generic computer system that can act as any number of "boxes" in the simulated topology. As an end host, a `Node` object can represent a computer as either a server or client appliance, generating or receiving packet traffic. When they are not residing on the edges of the network, the `Node` objects can alternatively act as routers, switches, firewalls, NATs, or any other middlebox based on the functionality which is installed on them.

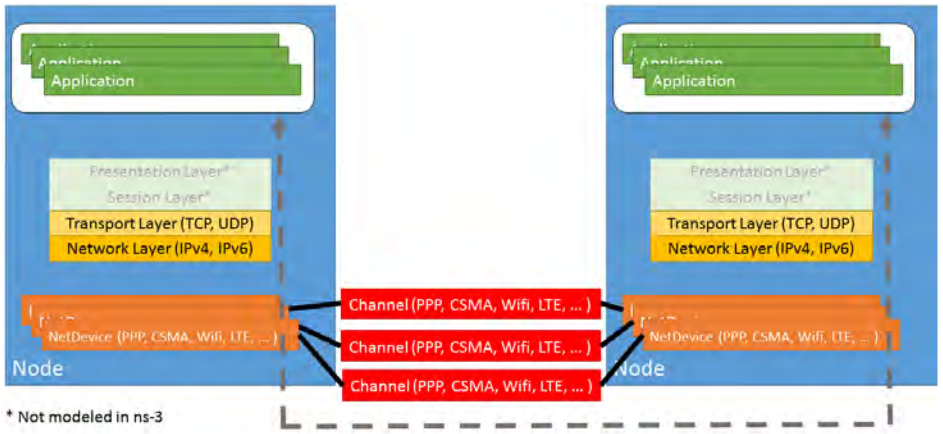
Application
Presentation Layer*
Session Layer*
Transport Layer (TCP, UDP)
Network Layer (IPv4, IPv6)
NetDevice (PPP, CSMA, Wifi, LTE, ...)
Node
Channel (PPP, CSMA, Wifi, LTE, ...)
Channel (PPP, CSMA, Wifi, LTE, ...)
Channel (PPP, CSMA, Wifi, LTE, ...)
Application
Presentation Layer*
Session Layer*
Transport Layer (TCP, UDP)
Network Layer (IPv4, IPv6)
NetDevice (PPP, CSMA, Wifi, LTE, ...)
Node
* Not modeled in ns-3

Figure 1: Diagram of the components modeled in an ns-3 simulated topology.

`Node` objects are connected to one another via either wired or wireless links. Specific link classes in ns-3 can model a variety of capabilities, with wired functionality such as the point-to-point protocol (PPP) and CSMA or with wireless operations such as Wi-Fi or LTE as shown in Figure 1. The simulated links in ns-3 are coupled such that the `NetDevice` subclass will only interact with its appropriate `Channel` subclass. For example, simulation of PPP requires that `Node` objects have `PointToPointNetDevice` objects installed on them and individual `Node` objects are connected using `PointToPointChannel` objects.

Working its way up the network stack, the `Channel` objects model the physical layer (L1) while the `NetDevice` objects act as the link layer (L2). Following this path, the network layer (L3) is implemented in a number of ways. IPv4 models for ARP, ICMP, UDP, and TCP are well supported in ns-3. Some IPv6 support is provided as well with neighbor discovery (ND), ICMPv6, IPv6 extension headers, UDP, and TCP. Routing with IPv4 can be accomplished through static or global means, the on-demand NIx vector mechanism, RIPv2, or the wireless protocols OLSR, AODV, DSR, or DSDV. Routing with IPv6 is less robust with static routing or RIPng being the two major ways to maneuver traffic.

The transport layer (L4) provides both UDP and TCP support and connects the packet-forwarding components in the lower layers to packet-generating aspects of the application layer. It accomplishes this coordination through a sockets API specific to ns-3 but mirroring traditional Berkeley sockets. TCP in ns-3 can be handled in a variety of ways either natively in ns-3, through the Network Simulation Cradle (NSC), via DCE with the Linux kernel stack, or through the use of virtual machines. Furthermore, congestion control for native ns-3 TCP is accomplished through any one of the following implementations: BIC, HighSpeed, Hybla, Illinois, Scalable, Vegas, Veno, WestWood(+), or YeAH.

Traffic generation and reception occurs at the application layer through the previously mentioned ns-3 sockets API using subclass objects of the `Application` class. Users can create their own applications to act as processes on the ns-3 `Node` objects. They may also use any of the provided applications in the ns-3 baseline: `BulkSendApplication`, `OnOffApplication`, `PacketSink`, `UdpClient`, `UdpServer`, `UdpEchoClient`, `UdpEchoServer`, `V4Ping`, `Ping6`, or `Radvd`.

The design of ns-3 provides two levels of granularity when developing a simulation script. At the lower level, users can directly interact with the various models provided by ns-3 to configure the components of their simulation with a finer granularity. However, this level of granularity would result in a simulation script that would be unnecessarily long and tedious. Instead, a “helper” API is included with the ns-3 baseline to abstract and simplify commonly collected program functionality. Anything written with the helper API could also be coded using the low-level API. Since network simulations often perform sets of identical actions on groups of components, ns-3 provides container objects as part of the helper API to realize this functionality. For example, multiple nodes can be handled via a `NodeContainer` object. Using a more specific example, the `Install` method of the `PointToPointHelper` takes one of these `NodeContainer` objects and returns a `NetDeviceContainer` object, holding the `NetDevice` objects and their link layer information (i.e. Ethernet addresses) associated with that installed connection. Subsequently, the `Assign` method of an `Ipv4AddressHelper` object can take that `NetDeviceContainer` and return an `Ipv4InterfaceContainer` object, holding `Ipv4Interface` objects and the network layer information (i.e. IPv4 addresses) now associated with those `NetDevice` objects. Through the use of these simple method calls on the associated helper objects, the helper API aims to make ns-3 scripts easier both to read and write.

3 VISUALIZATION

Visualization of a simulated network topology contributes to demonstrating its operation and assists in gauging its correctness. Graphical observation of the network simulation allows a user to see how a network is acting and reacting based on the transmission of packets. Furthermore, visualization acts as a convenient method for debugging a network simulation, verifying that the topology has been constructed and connected as the user intends and that traffic is traversing the topology as expected. The network simulator ns-3 does not have a preferred method of visualization. However, several tools have been developed, and this section discusses three of them: NetAnim, PyViz, and FlowMonitor.

3.1 NetAnim

NetAnim, developed by George Riley and John Abraham, is a network animator that operates based on an XML trace file collected during the ns-3 simulation. It supports IPv4 and can animate both wired and wireless links, displaying packets and their metadata as they traverse these links. It visualizes nodes and their mobility while also being able to display node information such as its IPv4 and Ethernet addresses. Additionally, routing tables for each node can be displayed as they develop over the course of the simulation. Packet timelines and node statistics can also be displayed through additional sections of the NetAnim tool.

NetAnim is developed using the Qt4 GUI framework and as such is built separately from ns-3 using the Qt build tools. After NetAnim has been built, it can load an XML trace file generated specifically for NetAnim animation by an ns-3 simulation script. To create this trace file, the ns-3 simulation script must be configured properly. The `netanim` module must be specified as a dependency in the wscript of the script module, the simulation script must include the `netanim-module.h` file, and an `AnimationInterface` object must be created in the script. This `AnimationInterface` object can be used to set the positions of the simulated nodes in the topology so they appear in the NetAnim visualization.

3.2 PyViz

PyViz differs from NetAnim in that it is a live visualizer. It actively scans the running simulation script to determine how to visualize the topology and its traffic without the use of trace files. Similarly to NetAnim, PyViz can display information specific to each simulated node. However, rather than displaying individual packets, it visualizes packet flow bitrates during packet transmission. As a live visualizer, it provides the ability to move forward in a running simulation while also including an interactive Python console for analyzing the instantaneous state of running objects. PyViz is written in Python but works for both Python and C++ ns-3 simulations. When all package dependencies for this tool are met, it can be enabled for use by ns-3 by simply including the flag `--vis` to the `./waf --run` command.

3.3 FlowMonitor

FlowMonitor(Carneiro, Fortuna, and Ricardo 2009) is not a graphical visualizer but instead an analytical visualizer, aiding researchers in easily deriving flow metrics from ns-3 simulations. It is a statistics capability that can interact with PyViz. Figures 2 and 3 visually describe the architecture and the statistics gathered by FlowMonitor, respectively. Integrated into the ns-3 baseline as one of its modules, it installs probes on the simulated nodes to track packets and measure the parameters found in the `FlowProbe::FlowStats` component for each flow. Most typical interactions with FlowMonitor can be accomplished through its helper similarly to other ns-3 helper classes. Through the helper, the monitor can be installed, configured, and used to print statistics. When the simulation completes, the FlowMonitor data can be exported into an XML file.

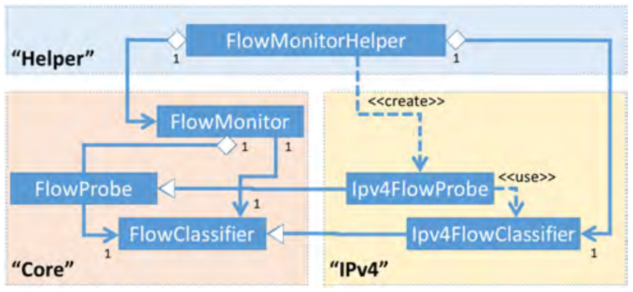

Figure 2: FlowMonitor architecture.

4 DIRECT CODE EXECUTION

DCE(Camara et al. 2014, Tazaki et al. 2013, Inria 2013) is a subproject of ns-3 that permits testing of real-world network applications within the simulated environment of ns-3. It allows execution of user space and kernel space network protocols and applications from within an ns-3 simulation, enhancing the realism of the simulated network. Simulated nodes in an ns-3 topology have these network applications installed on them, much in the same way stock applications in the ns-3 baseline are installed. The applications generally require no modifications to their source code. The source code for a particular application must be available though because it must be rebuilt to act as a dynamic binary rather than a static executable. Rebuilding applications in this way simply requires additional configuration flags for the compile and link instructions in the build process of the applications. When an application is installed on a node using DCE, it is then able to interact with the simulated network through either the ns-3 simulated network stack or

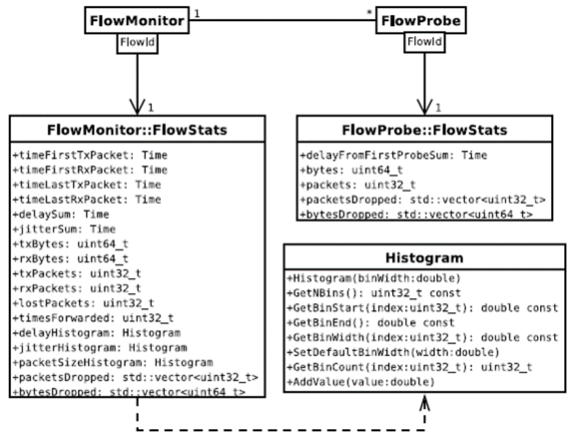
FlowMonitor
1
*
FlowProbe
FlowId
FlowId
1
1
FlowMonitor::FlowStats
+timeFirstTxPacket: Time
+timeFirstRxPacket: Time
+timeLastTxPacket: Time
+timeLastRxPacket: Time
+delaySum: Time
+jitterSum: Time
+txBytes: uint64_t
+rxBytes: uint64_t
+txPackets: uint32_t
+rxPackets: uint32_t
+lostPackets: uint32_t
+timesForwarded: uint32_t
+delayHistogram: Histogram
+jitterHistogram: Histogram
+packetSizeHistogram: Histogram
+packetsDropped: std::vector<uint32_t>
+bytesDropped: std::vector<uint64_t>
FlowProbe::FlowStats
+delayFromFirstProbeSum: Time
+bytes: uint64_t
+packets: uint32_t
+packetsDropped: std::vector<uint32_t>
+bytesDropped: std::vector<uint64_t>
Histogram
+Histogram(binWidth:double)
+GetNBins(): uint32_t const
+GetBinStart(index:uint32_t): double const
+GetBinEnd(): double const
+GetBinWidth(index:uint32_t): double const
+SetDefaultBinWidth(width:double)
+GetBinCount(index:uint32_t): uint32_t
+AddValue(value:double)

Figure 3: Statistics gathered by FlowMonitor.

the Linux kernel stack. The installed binary functions within DCE similarly to how a process running on an actual operating system would.

DCE has seen significant adoption in the realm of networking research as a means to enhance the realism of network simulations. The fields of mobile and wireless networks have been some of the most permissible areas for DCE adoption due to the frequency with which these technologies are updated. Introducing implementations of new technologies from these areas is more direct via DCE as compared to maintaining appropriate models of these protocols in simulation. The OLSR model in ns-3 has been compared against a real-world OLSR daemon with DCE, identifying issues in both implementations(Bikov and Boyko 2011). By tuning both the simulated model and the real-world program, both capabilities were updated to more adequately represent the OLSR RFC. Additional use cases for DCE have been demonstrated for content-centric networking (CCN) over mobile ad-hoc networks (MANETs) and multipath TCP over LTE and wireless in (Kwon et al. 2015, Tazaki et al. 2013). In both cases, no source code modifications were required for the original programs. Furthermore, research that enables the use of additional languages such as Python and Java has also been examined, expanding the number of potential applications that can be studied directly and portably through DCE(Ivey and Riley 2016).

DCE comprises three layers: the core, the kernel, and a POSIX layer. The *core* layer coordinates the actions of the simulated processes within the context of the ns-3 event scheduler. At this layer, global variables are loaded in a way that they can be shared by the multiple simulated processes. Threads, memory stack space, and the heap for each particular process are maintained here as well. The *kernel* layer links the Linux TCP/IP stack to the simulated physical layer (L2) of ns-3. The installed applications then use this hybrid simulated stack for packet transmission using application-level socket function calls.

The POSIX layer intercedes the system GNU C Library (*glibc*), the standard library for the C programming language. Calls to *glibc* functions are captured by this layer for handling. In many cases, such as string manipulation or mathematical operations, DCE requires no additional management and can simply pass the calls to the *glibc* of the underlying system. Time-related functions are managed in relation to the simulated time rather than the wallclock time. Socket function calls use the simulated sockets of ns-3 rather than actual system sockets that would interact with the external network. Subprocesses and threads are handled in terms of their simulated contexts instead of using the subprocessing and threading capabilities of the underlying system. The POSIX layer also manages local files for each DCE-configured application in terms of specific file space allocated for each node. For example, a node identified as Node 0 would handle files for a DCE-configured application at runtime from a system directory named *files-0*. This file space would contain the output files created during execution of the application. It could also be used for configuration files or other runtime file dependencies specific to the application on that particular node.

Listing 3: Instructions to install ns-3 and DCE

```
hg clone http://code.nsnam.org/bake bake
mkdir dce-wsc
cd dce-wsc
../bake/bake.py configure -e dce-ns3-1.9
../bake/bake.py download -vvv
../bake/bake.py build -vvv
```

DCE provides a number of options when determining how to configure the operation and management of the DCE-configured applications. Generally, users will choose whether to use the simulated internet stack of ns-3 or the Linux kernel stack. Less typical configuration options allow the user to determine how processes and binaries are handled. The `FiberManager` class in DCE handles the management of execution contexts for the processes and subprocesses of DCE-configured applications. DCE provides the option within the `FiberManager` class to either use a system modeled after the *ucontext.h* POSIX API or the *pthread* library. These choices provide the user with the choice of either efficiency or easier debugging, respectively. Similarly, the user has an option when determining how DCE will load the binaries for the installed applications through its `LoaderFactory` class. This class by default uses the `CoojaLoaderFactory`, which supports `fork`. Alternatively, a more efficient but somewhat less reliable option is the `DlmLoaderFactory`.

C and C++ programs that will operate as DCE applications must be recompiled to be recognized as dynamic libraries rather than static executables. The application must be configured in a way that DCE can load the `main` function of a network application as if it was just another function of the "shared" library. Compiling the source code into position-independent code with the `fPIC` flag configures it similarly to a shared object library. Linking with `pie` creates a position-independent executable and `rdynamic` prompts the resulting binary to load all of its symbols into the dynamic symbol table. The addition of these flags when rebuilding a network application is typically the only requirement for compatibility with DCE. However, certain functions used by the application may not currently be implemented in the *glibc* coverage of the DCE baseline. If so, they will need to be included in a simulation-appropriate manner.

5 EXAMPLE WALKTHROUGH

This example takes a simplified version of the *dce-iperf* example found in the DCE baseline. It creates a network of two nodes connected and configured with a single wired link using ns-3 simulation components. It then installs two DCE-configured applications onto each simulated node such that one node will act as an *iperf* client transmitting packets to the other node acting as an *iperf* server. It uses the latest releases of the ns-3 and DCE libraries, which at the time of this writing are ns-3.26 and DCE-1.9. For this tutorial, it assumes that the operating system in use is Ubuntu 14.04. Initial installation instructions using bake are provided in Listing 3. Prerequisite software packages can be found at https://www.nsnam.org/wiki/Installation#Ubuntu.2FDebian.

An excerpt of the bake configuration file used to download and build ns-3 and DCE is shown in Listing 4. Based on this part of the configuration, a version of iPerf 2.0 will be downloaded, built, and installed. This installation is not system-wide but simply local to the overarching bake installation. This excerpt demonstrates how bake will download the source code for *iperf* (specifically into a directory named `iperf-2.0.5` under `dce-wsc/source/`) and then build and install it in a location that DCE can find it (specifically in `dce-wsc/build/bin`). The bake installation tool understands based on the configuration that the source code is retrieved from an archive rather than an online repository. The configuration also allows commands to be executed and patches to be introduced prior to installation. Configuration of the build instructions specifically adapt the resulting *iperf* executable for compatibility with DCE as noted by the `pie` and `rdynamic` flags introduced during linkage. (Compile configuration using the `fPIC` flag is introduced in the patch file.)

Listing 4: Excerpt of bake configuration file for installing iperf for DCE usage

```
<module name="iperf">
<source type="archive">
  <attribute name="url"
    value="http://iperf.fr/download/source/iperf-2.0.5-source.tar.gz"/>
  <attribute name="extract_directory" value="iperf-2.0.5"/>
</source>
<build type="make" objdir="yes">
  <attribute name="pre_installation"
    value="cd $SRCDIR;./configure --prefix=$INSTALLDIR"/>
  <attribute name="patch"
    value="$SRCDIR/../ns-3-dce/utils/iperf_4_dce.patch"/>
  <attribute name="build_arguments"
    value="LDFLAGS=-pie LDFLAGS+=-rdynamic"/>
</build>
</module>
```

The source code for this tutorial is shown in Listing 7. (It can be copied into the `dce-iperf.cc` file located in `dce-wsc/source/ns-3-dce/example`.) Lines 1-6 display the necessary header files to be included. Lines 10-12 demonstrate the usage of the `CommandLine` object to permit command line arguments, in this case specifying the number of buffers that the *iperf* client will transmit. Two simulated nodes are created via the `NodeContainer` helper class (line 14), and these two nodes are connected using a `PointToPointHelper` object (line 18). This helper is configured to create a wired point-to-point link between the two nodes with a bandwidth of 5Mbps and a speed-of-light delay of 1ms. The `InternetStackHelper` object installs the ns-3 simulated TCP stack onto the two nodes (line 19). IPv4 addresses are then assigned to the devices on each node with the help of the `Ipv4AddressHelper` class (line 22). This assignment will result in Node 0 possessing an IP address of 10.1.1.1 while Node 1 will be assigned 10.1.1.2. The final line of ns-3 topology configuration (line 23) is not necessary for this example but demonstrates the simplicity with which routing can be enabled in ns-3. Rather than employing a coordinated routing mechanism, such as BGP, for each node in the simulated topology, the simulation collectively manages routing for the entire topology and realizes this mechanism in conjunction with the `Ipv4GlobalRoutingHelper` class. In this way, simulated packet routing can be accomplished in a simpler, more efficient manner.

Lines 25-40 of Listing 7 enable the two nodes created and configured by ns-3 to use the real-world *iperf* application within the simulated environment. The `DceManagerHelper` class prepares the nodes to operate collaboratively with the three layers of DCE. The `DceApplicationHelper` class then composes the proper *iperf* commands and installs them onto each node. The `SetBinary` method (lines 28, 36) specifies which executable should be used. The `ResetArguments` and `ResetEnvironment` methods ensure that no previous information is erroneously connected to the executable. The `AddArgument` function can then be called as many times as necessary to construct the remainder of the command that will be executed. In this way, Node 0 will by default receive the following command: `iperf -c 10.1.1.2 -i 1 -n 1M`; Node 1 will in turn make the following call: `iperf -s`. The start times for each DCE application are controlled in this example using the `Start` method of an ns-3 `ApplicationContainer` object. Since the `Stop` methods are not called for either application or the simulator in general, the simulation will stop once the *iperf* client has transmitted its specified number of buffers. The simulator begins processing events with the call to `Run`, and upon completion, cleans memory using the `Destroy` method (line 42).

The simulation can be run directly from the `dce-wsc/source/ns-3-dce` directory in a terminal using the following command: `./waf --run dce-iperf`. (Alternatively, the number of buffers can be

Listing 5: Output of Node 0 (*iperf* client)

```
Client connecting to 10.1.1.2, TCP port 5001
TCP window size:  128 KByte (default)
------------------------------------------------------------
[  3] local 10.1.1.1 port 49153 connected with 10.1.1.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 1.0 sec   640 KBytes  5.24 Mbits/sec
[  3]  0.0- 1.6 sec  1.00 MBytes  5.16 Mbits/sec
```

Listing 6: Output of Node 1 (*iperf* server)

```
Server listening on TCP port 5001
TCP window size:  128 KByte (default)
------------------------------------------------------------
[  4] local 10.1.1.2 port 5001 connected with 10.1.1.1 port 49153
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0- 2.6 sec  1.00 MBytes  3.20 Mbits/sec
```

modified using this command: `./waf --run "dce-iperf --num=<num>"`.) Once the simulation completes, all output generated for the simulated Nodes 0 and 1 will be sent to files in the following locations, respectively: `ns-3-dce/files-0/var/log/<pid>` and `ns-3-dce/files-1/var/log/<pid>`. Listings 5 and 6 display the contents of the respective `stdout` files of Nodes 0 and 1. These output files mirror the feedback that would be returned from two computer systems transmitting *iperf* data across a real network. This particular example, although simple, demonstrates the potential when using DCE to execute real-world network applications from within the ns-3 simulated environment.

6 ADDITIONAL RESOURCES

Although this tutorial covers a variety of topics concerning ns-3 and DCE, additional information can be found as well through a number of different channels. A brief list is provided to point users to additional links where further assistance can be found:

- Official ns-3 website: https://www.nsnam.org
- API documentation: https://www.nsnam.org/doxygen
- Wiki: https://www.nsnam.org/wiki/Main_Page
- Online tutorial: https://www.nsnam.org/docs/release/3.26/tutorial/html/index.html
- DCE resources: https://www.nsnam.org/overview/projects/direct-code-execution/
- Python/Java integration (based on ns-3.24.1/DCE-1.7): https://github.com/jaredivey/dce-python-sdn
- Mailing lists for:
  - Users: https://groups.google.com/forum/#!forum/ns-3-users
  - Developers: http://mailman.isi.edu/mailman/listinfo/ns-developers

REFERENCES

Bikov, E., and P. Boyko. 2011. "Direct Execution of OLSR MANET Routing Daemon in ns-3". In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools '11, 454–

461. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

Breslau, L., D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu, and H. Yu. 2000. "Advances in Network Simulation". *Computer* 33 (5): 59–67.

Bryant, R. E. 1977. "Simulation of Packet Communication Architecture Computer Systems". Technical report, Cambridge, MA, USA.

Camara, D., H. Tazaki, E. Mancini, T. Turletti, W. Dabbous, and M. Lacage. 2014. "DCE: Test the Real Code of Your Protocols and Applications over Simulated Networks". *Communications Magazine, IEEE* 52 (3): 104–110.

Carneiro, G., P. Fortuna, and M. Ricardo. 2009. "FlowMonitor: A Network Monitoring Framework for the Network Simulator 3 (ns-3)". In *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, VALUETOOLS '09, 1:1–1:10. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

Chandy, K., and J. Misra. 1979. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs". *Software Engineering, IEEE Transactions on* SE-5 (5): 440–452.

GNS3 Technologies Inc. 2017. "GNS3 — The Software that Empowers Network Professionals". https://www.gns3.com/.

INRIA 2012. *BAKE Documentation*. Available via https://www.nsnam.org/docs/bake/tutorial/html/index.html.

Inria 2013. "Direct Code Execution (DCE) Manual". Website. https://www.nsnam.org/docs/dce/release/1.5/manual/html/index.html.

Ivey, J. S., and G. F. Riley. 2016. "Analysis of Programming Language Overhead in DCE". In *Proceedings of the Workshop on ns-3*, WNS3 '16, 41–48. New York, NY, USA: ACM.

Kwon, S., K. Hasan, M. Lee, and S. Jeong. 2015. "Comparative Analysis of Real-time Video Performance in the CCN-based LTE Networks". In *Information and Communication Technology Convergence (ICTC), 2015 International Conference on*, 509–511.

Lacage, M., and T. R. Henderson. 2006. "Yet Another Network Simulator". In *Proceeding from the 2006 Workshop on ns-2: The IP Network Simulator*, WNS2 '06. New York, NY, USA: ACM.

Mattern, F. 1993. "Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation". *Journal of Parallel and Distributed Computing* 18 (4): 423–434.

ns-3 Project 2017. *ns-3 Tutorial*. Release ns-3.26 ed. Available via https://www.nsnam.org/docs/release/3.26/tutorial/ns-3-tutorial.pdf.

ns-3 project 2017. "Random Variables – Manual". Website. https://www.nsnam.org/docs/manual/html/random-variables.html.

Pelkey, J., and G. Riley. 2011. "Distributed Simulation with MPI in ns-3". In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques*, SIMUTools '11, 410–414. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

Riley, G. F. 2008. *Using the Georgia Tech Network Simulator*. Available via http://griley.ece.gatech.edu/MANIACS/GTNetS/docs/GTNetS_manual.pdf.

T. Nagy 2017. *The Waf Book*. Available via https://waf.io/book/.

Tazaki, H., E. Mancini, D. Camara, T. Turletti, and W. Dabbous. 2013. "MSWIM Demo Abstract: Direct Code Execution: Increase Simulation Realism Using Unmodified Real Implementations". In *Proceedings of the 11th ACM International Symposium on Mobility Management and Wireless Access*, MobiWac '13, 29–32. New York, NY, USA: ACM.

Tazaki, H., F. Urbani, E. Mancini, M. Lacage, D. Camara, T. Turletti, and W. Dabbous. 2013. "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 '13, 217–228. New York, NY, USA: ACM.

AUTHOR BIOGRAPHIES

JARED S. IVEY is a technical lead engineer for software development and maintenance of the Joint STARS E-8C platform at Robins Air Force Base, GA. He received his B.S. in Biomedical Engineering from the Georgia Institute of Technology in May 2009 and his M.S.E. in Software Engineering from Mercer University in May 2012. He received his Ph.D. in Electrical and Computer Engineering in August 2017 under the supervision of Dr. George F. Riley at the Georgia Institute of Technology. His research interests focus on providing scalable and reliable solutions for network simulation of software-defined networks. His email address is j.ivey@gatech.edu.

BRIAN P. SWENSON is a research engineer for Georgia Tech Research Institute in Atlanta, GA. He received his Ph.D. in Electrical and Computer Engineering at the Georgia Institute of Technology in August 2015. He received his B.S. in Computer Engineering and Computer Science from the University of Wisconsin-Madison and an M.S. in Electrical and Computer Engineering from the Georgia Institute of Technology in 2013. His research interests include distributed discrete event simulation, high performance computing and computer networks. His email address is brian.swenson@gtri.gatech.edu.

GEORGE F. RILEY received his Ph.D. from the Georgia Tech College of Computing in August 2001, and joined the faculty of ECE at that time. Mr. Riley received a MSCS from Florida Tech in 1996, and a BSEE from University of Alabama in 1972. Prior to enrolling at Tech in 1996, Mr. Riley was president and CEO of Infoware, Inc. of Cocoa Beach Florida. From 1987 to 1996 Infoware provided software and system design services to the United States Air Force at Patrick Air Force Base, Florida. During that time, Infoware designed, implemented, and deployed numerous systems in support of the missile launch activities at Cape Canaveral Air Force Station, including a communications front-end processor for real-time data gathering and a real-time distributed flight safety display system. Concurrently, from 1984 to 2000, Mr. Riley was also vice-president and co-owner of CAM Systems Inc. of Atlanta Georgia. CAM systems developed, under Mr. Riley's direction, a suite of PC based software tools for residential property management. His email address is riley@ece.gatech.edu.

Listing 7: Simplified example of dce-iperf.cc with two nodes transmitting data via *iperf*

```
#include ”ns3/network-module.h”
#include ”ns3/core-module.h”
#include ”ns3/internet-module.h”
#include ”ns3/dce-module.h”
#include ”ns3/point-to-point-module.h”
#include ”ns3/applications-module.h”
using namespace ns3;
int main (int argc, char *argv[]) {
  std::string num = ”1M”;
  CommandLine cmd;
  cmd.AddValue (”num”, ”Number of packets. Default 1M.”, num);
  cmd.Parse (argc, argv);

  NodeContainer nodes; nodes.Create (2);
  PointToPointHelper pointToPoint;
  pointToPoint.SetDeviceAttribute (”DataRate”, StringValue (”5Mbps”));
  pointToPoint.SetChannelAttribute (”Delay”, StringValue (”1ms”));
  NetDeviceContainer devices; devices = pointToPoint.Install (nodes);
  InternetStackHelper stack; stack.Install (nodes);
  Ipv4AddressHelper address;
  address.SetBase (”10.1.1.0”, ”255.255.255.252”);
  Ipv4InterfaceContainer interfaces = address.Assign (devices);
  Ipv4GlobalRoutingHelper::PopulateRoutingTables ();

  DceManagerHelper dceManager; dceManager.Install (nodes);
  DceApplicationHelper dce; dce.SetStackSize (1 << 20);
  ApplicationContainer apps;
  dce.SetBinary (”iperf”); // Launch iperf client on node 0
  dce.ResetArguments (); dce.ResetEnvironment ();
  dce.AddArgument (”-c”); dce.AddArgument (”10.1.1.2”);
  dce.AddArgument (”-i”); dce.AddArgument (”1”);
  dce.AddArgument (”-n”); dce.AddArgument (num);
  apps = dce.Install (nodes.Get (0));
  apps.Start (Seconds (0.7));

  dce.SetBinary (”iperf”); // Launch iperf server on node 1
  dce.ResetArguments (); dce.ResetEnvironment ();
  dce.AddArgument (”-s”);
  apps = dce.Install (nodes.Get (1));
  apps.Start (Seconds (0.6));

  Simulator::Run (); Simulator::Destroy ();
  return 0;
}
```