Analysis of Programming Language Overhead in DCE

Jared S. Ivey
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA, USA
j.ivey@gatech.edu

George F. Riley
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA, USA
riley@ece.gatech.edu

ABSTRACT

In the context of network simulation, directly executing the code for new protocols and applications can add a sense of realism to the simulation while saving time that would have been required for development and validation of sufficiently representative models. Direct Code Execution (DCE) in ns-3 provides this functionality for debugging and analyzing new network applications directly in simulation. However, DCE currently requires that these applications are executed as native code that has been compiled from source code written in C or C++. This work exploits the fact that languages such as Python and Java launch their applications through native code that ultimately translates their source code into instructions that the underlying system understands. The efforts of this study provide a framework for allowing applications written in Python or Java to be executed within the simulated environment similarly to how DCE currently allows applications written in C or C++ to be introduced into simulation. This framework is tested on multiple simulated topologies for a variety of applications, and its overhead is examined in the context of memory usage and wallclock execution time.

CCS Concepts

•**Computing methodologies** → **Discrete-event simulation;** *Simulation evaluation;* •**Software and its engineering** → *Object oriented languages;*

Keywords

Network Simulation, ns-3, Direct Code Execution, Programming Languages, C/C++, Java, Python

1. INTRODUCTION

The field of network communications is constantly evolving and expanding with the introduction of new technologies and protocols aimed at providing greater quality, resiliency, and security to the vast amounts of data traversing current networks. In this ever-changing field, modeling and simulation provide an avenue for examining the traffic within new or existing network topologies. 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. Popular network simulators such as ns-3 are effective tools for studying these network behaviors. However, adequate coverage of new network programs and protocols within simulation requires that models of these new applications are ported by developers into the simulators. These efforts require a significant amount of time in terms of simply writing the code but also in testing its validity against real-world behaviors. Instead, a mechanism for directly deploying real-world network applications within a simulated environment can alleviate these issues while supplying an air of realism. The Direct Code Execution (DCE) framework in ns-3 can deliver this functionality. It allows real-world protocols and applications to be installed and executed directly on the simulated nodes of a topology created in ns-3.

The concept of direct code execution is primarily constrained to network applications and protocols written in the programming language of the employed simulator. For DCE in ns-3 and some similar network simulators, prior work has demonstrated their effective use for applications written in C or C++, which is conveniently accomplished because these simulators are written in C++. The Python-based flow simulator *fs* includes extensions that can connect it to external applications simply because those applications are also written in Python[1]. The efforts of this work introduce a framework for allowing DCE to accommodate and execute code for network applications written in languages other than C or C++, specifically Python and Java. The executables that launch applications written in these languages are simply native code binaries that are built from C/C++ source code. This work exploits this fact in order to launch Python and Java applications within the DCE environment, allowing their source code to be interpreted line by line from within the context of the simulation. In this way, entirely new sets of network applications can be conveniently examined through the DCE environment without source code modification or the need for a translated port to the languages that DCE understands.

The remainder of this paper is organized into the following sections. Section 2 briefly describes ns-3 and provides detailed information on DCE. The programming languages C/C++, Python, and Java as well as considerations required for their usage in DCE are discussed in section 3. Section

WNS3, June 15-16, 2016, Seattle, WA, USA

DOI: http://dx.doi.org/10.1145/2915371.2915383

4 details the applications, topologies, and experiments examined for each programming language. The results are examined and discussed in section 5. Section 6 outlines prior work and use cases of DCE and similar libraries. Section 7 concludes the work and provides potential directions for future research.

2. NETWORK SIMULATOR NS-3

This work uses the network simulator ns-3, a popular discrete event network simulator written in C++. It is used for a variety of educational and research-oriented purposes within the field of computer networking. ns-3 provides simulation and emulation frameworks for developing network topologies and analyzing their network characteristics. In creating network simulations, users of ns-3 have the choice of accessing its libraries by programming in C++ or through Python bindings. Stock or user-created applications may be installed on the nodes of simulated topologies in ns-3, generating and monitoring packets within the simulated network. Data may be collected from the artifacts of these network simulations in order to better understand and gauge the performance of existing and proposed real-world networks.

2.1 Direct Code Execution

DCE is an additional module for ns-3 that can test real-world network applications within the ns-3 environment. It provides the capability to execute userspace and kernelspace network protocols and applications directly within an ns-3 simulation to provide additional realism to the simulated network. These real-world applications are installed on specified nodes within the simulated topology in a similar manner to the stock applications in the ns-3 baseline. The applications themselves typically require no modifications, but they must be rebuilt such that they can act as dynamic binaries rather than static executables. This procedure simply requires that some additional configuration flags are added to the compile and link instructions. Once installed on a node, the applications can interact with the rest of the simulated network through either the ns-3 simulated network stack or the Linux kernel stack. DCE interacts with the installed binary similarly to how an actual operating system would. DCE applications are executed within the simulated environment under a single-process model such that all applications inside their own simulated processes are managed by a single process on the underlying system. This single-process approach provides a convenient debugging alternative to complex, distributed debuggers.

DCE is composed of three layers: the core, the kernel, and a POSIX layer. The *core* layer provides virtualization mechanisms to coordinate the actions of the simulated processes within the context of the ns-3 event scheduler. At this layer, global variables are loaded in physical pages on the underlying single process to be shared by the multiple simulated processes. Threads, stack space, and the heap for each particular process are managed in the core layer as well. The *kernel* layer connects the real-world Linux TCP/IP stack to the simulated physical layer (L2) of ns-3. Traffic from the installed applications can then traverse this hybrid simulated stack from application-level socket function calls.

The POSIX layer of DCE replaces the real GNU C Library (*glibc*), the standard library for the C programming language. Calls to *glibc* functions are caught by this layer to determine how they should be handled. In many cases, DCE does not need to manage the interactions and results of certain *glibc* calls within the context of the simulation. Calls to functions in `string.h` or `math.h`, for example, can simply be passed to the *glibc* of the underlying system. Time-related functions return information about the simulated time rather than the wallclock time. Socket function calls are effectively wrappers to the simulated sockets in relation to the ns-3 stack. Subprocess and threading functions are also handled by this layer to appropriately manage their contexts. Local files are managed in the POSIX layer relative to specific file space for each node running DCE-configured applications. The file space holds all output generated by the application and may also be used to hold any runtime file dependencies[2, 3, 4].

3. PROGRAMMING LANGUAGES

This section provides a simple historical and technical overview of the programming languages examined in this work. It begins by examining the languages for which DCE was originally intended to be compatible, namely C and C++. Through extensions for DCE performed in this work, two additional languages have been made operable within the ns-3/DCE environment. The interpreted language Python and its predominant runtime library CPython are described. Additionally, the Java programming language and its runtime environment, the Java Virtual Machine (JVM), are discussed.

3.1 C/C++

The C programming language is considered one of the first mainstream, general-purpose programming languages. It was developed by Dennis Ritchie from 1969 to 1973 at Bell Telephone Laboratories (now AT&T Bell Labs). It is a statically typed, procedural language that has been adopted for use in a variety of systems. It is one of the "lower" high level languages available with many newer languages, including C++, Java, and Python, employing it as an intermediary at some point in their respective compiler/runtime pipelines. As stated in section 2.1, the standard library for the C programming language, referred to as *glibc*, provides a substantial level of functionality. String and memory manipulation, mathematical functions, system time information, file and socket handling, parallel processing, and a variety of other capabilities are available in *glibc*.

The C++ programming language, created by Bjarne Stroustrup at AT&T Bell Labs in 1983, was originally intended as an object-oriented enhancement to C. In addition to similar features as C, C++ enables class creation complete with abstraction, encapsulation, inheritance, and polymorphism, templates, and operator overloading. A standard API is provided by the C++ Standard Library, which *glibc* currently supports. Both C and C++ are compiled to native code that an underlying computer system can recognize equivalently. This characteristic results in them being effectively identical from the viewpoint of the DCE environment.

To operate as DCE applications, programs written in C or C++ must be recompiled such that DCE recognizes them as dynamic libraries rather than static executables. In this way, DCE can load the `main` function of a particular program as if it was just another addressed symbol in the memory space of the dynamically loaded library. Compiling the source code into position-independent code with the `fPIC` flag allows it to be loaded similarly to a shared object library. Subse-

quently, linking with `pie` produces a position-independent executable and `rdynamic` ensures that all of its symbols will be loaded into the dynamic symbol table. Regarding operability within DCE, the addition of these flags to the compile and link steps is generally the only modification required for building a target application for execution in DCE. However, additional considerations may be necessary if certain functions used by the application are not currently implemented in the *glibc* coverage of the DCE baseline.

3.2 Python

The Python programming language is a multi-paradigm, general-purpose, high-level language that aims to be more readable than other general-purpose, object-oriented languages, such as C++ or Java. It was developed by Guido van Rossum and initially released in 1991. Today, programming in Python is available in two versions: Python 2 (most recently 2.7) and the backwards-incompatible Python 3 (currently 3.5). The reference implementation for both versions is CPython, whose source is written in C. CPython acts as a source code interpreter more so than a compiler. When Python programs (or *scripts*) are provided as inputs to the `python` command, the code will be read and directly executed by the Python runtime. This fact suggests that, with the proper configuration, the `python` command can be built to operate in the DCE environment. In this way, when it is provided a Python script, the source lines will be interpreted, and the underlying *glibc* calls can be handled by DCE. The focus of this work is the CPython implementation of Python 2.7.

The CPython source code did not require any modifications. In building it into the `python` executable and libraries, it required the typical flags such that DCE could recognize and load it, i.e. `fPIC` for compile and `pie` and `rdynamic` for linking. Similarly to *glibc*, Python is equipped with the Python Standard Library (PSL), a set of objects and APIs written in Python that provide a standardized baseline of Python programming capabilities. Because the PSL is written as Python scripts and not a "built" library in the same sense as *glibc*, it is loaded in a different way. A Python script is generally "imported" into the Python runtime through the `python` command. In the context of DCE, this step occurs after the `python` executable is loaded and executed. In this way, the effective kernel space has transitioned into simulation. At this point, the location of the user-defined Python application and baseline Python scripts in the PSL such as `os`, `socket`, and `string` must be placed where the DCE-enabled node can "see" them. To make the files visible to the simulated node, the user-defined script is copied into the filespace for the node, and the PSL is symbolically linked under this filespace as well.

Integrating CPython into the DCE framework is relatively straightforward and follows the typical process for installing and testing new programs on DCE-enabled nodes in an ns-3 topology. Code within new programs that is currently not recognized by the POSIX layer of DCE can be marked to use the system *glibc* library with the `NATIVE` macro. A number of `math.h` functions are needed within the Python library such as `exp`, `log`, `pow`, `__isnan`, `__isinf`, etc. Some *pthread* attribute functions are set to execute natively as well such as `pthread_attr_init`, `pthread_attr_setscope`, and `pthread_attr_destroy`. Since these functions only handle pthread attributes and not the actual threads, they can be manipulated natively with relatively low risk. For a function that cannot be run natively, the `DCE` macro can be utilized to either override the function behavior or wrap it within the context of the simulation. Conveniently, the Python library required relatively few functions needing the `DCE` macro. One example is `sendfile` which is intended to move data between file descriptors more quickly than a combination of `read` and `write`. In the context of simulation, this efficiency is less of a requirement, and as such, its DCE version can simply be the `read-write` combination. Another function – `__rawmemchr` – is only available in the binary standard rather than the source standard in the *glibc* base. Hence, DCE cannot recognize it natively. Instead, `__rawmemchr` must be forward-declared and wrapped around a simpler call to `strchr` as a workaround. Furthermore, a patch providing `epoll` support is incorporated as well from [5].

3.3 Java

Java is a general-purpose, object-oriented programming language similar in some ways to C++. It was created by James Gosling and released in 1995 through Sun MicroSystems (acquired by Oracle). Java applications are compiled but not to native code. Instead, they are converted to Java bytecode which can be executed on the JVM. The JVM along with the standard Java Class Library (JCL) comprises the Java Runtime Environment (JRE) which provides the APIs and executable environment for running Java programs. When running a compiled Java program, the JRE will initialize the environment, and then the JVM will interpret the provided bytecode into native code that the underlying system can understand. One version of the JVM, HotSpot, provides performance optimizations such as adaptive compilation as well as efficient heap management and garbage collection. Development of Java programs is enabled through the Java Development Kit (JDK), which allows the applications to be compiled and packaged.

The OpenJDK library, an open-source implementation of the Java Standard Edition (SE) Platform, provides a configurable mechanism to build an interface between DCE and the Java programming language. The JRE and JDK provided by Oracle is only distributed as binaries such that they are only available in specific build configurations. In contrast, OpenJDK is available as source code that can be built – compiled and linked – with the position-independent and dynamic flags that allow DCE to load its programs into simulation. The "program" of specific interest is the `java` command. The source code that produces the command and the JRE and JVM libraries that it calls to configure and execute the Java environment are all written in C. In this way, applications written in Java that are compiled to class files that the `java` command will accept will ultimately be interpreted to *glibc* symbols. These symbols can then be loaded and executed by the DCE environment. The implementation of OpenJDK used by this work is OpenJDK 8.

The JRE, designed around the JVM, requires additional considerations beyond simply addressing *glibc* symbols that the DCE baseline has yet to include. This process did require the inclusion of additional symbols, some related to determining the process location for newly created Java threads within virtual memory. However, accommodating the JVM also needed to address determining networking interfaces for the DCE-enabled nodes. Under the baseline OpenJDK source code, information about network interfaces

and next-hop routes for a system running the JVM is gathered from the `/proc/net/if_inet6` and `/proc/net/ipv6_route` files, respectively. (The JVM handles translating addresses between IPv4 and IPv6 schemes.) The information in these files is relatively easy to gather. However, an interesting issue presents itself in the form of buffer limitations for standard I/O methods such as `sscanf` within the DCE environment. Limitations are implicitly imposed by DCE on the applications it manages simply due to the nature of handling virtual kernel space inside of a simulation. To model the environment the JVM expects as closely as possible, filespace for a DCE-enabled node is created with the `/proc/net` directory in place. However, to accommodate the noted buffer issues, modified versions of the `if_inet6` and `ipv6_route` files are written that only hold the information that the JVM needs. For `if_inet6`, this information is the hexadecimal representations of the IPv6 addresses for a node as well as their indexes and device names. The `ipv6_route` file that is responsible for configuring routes for the node holds the base IPv6 addresses as 32-character hexadecimal values, the hexadecimal prefix length (similar to the IPv4 subnet mask), configuration flags, and the device names.

4. EXPERIMENTS

This section describes three sets of experiments that have been performed to confirm – and in the case of C/C++ applications reaffirm – the relative range of functionality DCE provides for a variety of applications. Since both C and C++ programs would compile to roughly similar native code representations, only C programs have been examined. Alternatively, the choice of compiler between the GNU C Compiler (GCC) and the Clang frontend for LLVM is introduced for examination to gauge potential differences in compiler optimizations. In one set of experiments, a single node is tasked to perform some simple applications written in each language. In the second set of experiments, a simple dumbbell topology tests basic networking functionality for client/server-style applications written in each language. The final experiments task a host to ping every other end host in a ring topology multiple times to examine the scalability of the simulations when handling multiple DCE applications. All experiments have been performed using the baseline ns-3.24.1 and a modified version of DCE 1.7 that implements the described updates from section 3. Performance is examined against the unoptimized builds of ns-3 and DCE as they are the defaults and typically provide the lowest risk for interference with DCE applications.

4.1 Performance Benchmarks

The first set of experiments installs programs written in C, Java, and Python on a single DCE-enabled node. These programs provide computational workload through some simple algorithms to confirm basic language capability in terms of data types and operators. They also test simple functionality in threading and local (loopback) networking, two relatively common features in networking applications.

4.1.1 Matrix Multiplication and Digits of Pi

The matrix multiplication applications for each language create a 1000-by-1000 array of integers and populate all columns in each row with the row index, e.g. array[0] = [0 0 ... 0], array[1] = [1 1 ... 1], etc. The program then performs matrix multiplication of the array with itself and inserts the corresponding values in a separate 1000-by-1000 array. Although it is not algorithmically complex, the matrix multiplication example demonstrates successful memory allocation, basic operand functionality, and simple control statement behavior in each language within the context of DCE. Furthermore, while the matrix multiplication applications in C and Java simply use nested `for` loop constructs, the Python application administers array allocation and multiplication "Pythonically" using the following construct:

```
[[x for y in range(0,1000)] for x in range(0,1000)]
```

An application to display 100,000 digits of π is written for each language based on the spigot algorithm in [6]. This application tests control statements similarly to the matrix multiplication program. However, additional mathematical complexity is administered, and string formatting is tested as well.

4.1.2 Simple Threading

A threaded application is examined to confirm successful import and usage of threads in Python and Java. The application written for this benchmark creates 500 threads. When each thread executes, it simply iterates 1,000 times printing a string. Threading capabilities using *pthreads* in C are already available in the DCE baseline. Since Python and Java threading functionality ultimately encapsulates calls to *pthreads* methods, it is expected that similar functionality can be achieved in these languages, most likely with some processing overhead. Java implements a `Thread` class for handling thread-based parallelism. In Python, threads are created and managed by importing the `threading` library.

4.1.3 Local Ping

An application to test local networking functionality performs an Internet Control Message Protocol (ICMP) echo request 1,000 times to the *localhost* (127.0.0.1) address and determines if ICMP echo replies are returned. For C, the source code for the original PING application[7] is recompiled for DCE. In Java, an `InetAddress` object is created using the `getLocalHost` method. Then, the `isReachable` method is called to determine the reachability of the address. For the Python application, ping functionality uses a library that mimics the C version of PING through raw sockets sending ICMP echo requests[8].

4.2 Simple Topology

A simple dumbbell topology is constructed to confirm successful networking capabilities are achieved in C/C++, Java, and Python. The dumbbell consists of two inner nodes acting as routers that are connected by a point-to-point link with a data rate of 100Mbps and a speed-of-light delay of 1ms. The outer 2 nodes each connect to one router with a 10Mbps link with 1ms delays. Additionally, network interfaces for the links are configured on different 255.255.255.0 subnets (CIDR /24) to confirm packet transmission occurs through successful L3 routing and not simply ARP requests.

Both end hosts in the described topology are enabled for installation of DCE applications. One end host is installed with a DCE application that acts as a client pushing data to the other end host. The client will create a TCP socket, establishing a connection with the other end host. Following successful connections, the client allocates 65,536 bytes for transmission. This amount is selected to ensure that

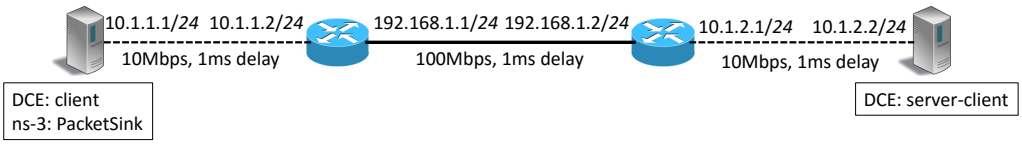

Figure 1: The simple dumbbell topology is used to confirm successful socket handling and data transmission and reception for C/C++, Java, and Python. The topology consists of 4 nodes connected linearly with the outer 2 nodes acting as end hosts and the inner 2 acting as routers.

the socket connection will accommodate multiple segments (based on the ns-3 default segment size of 536 bytes) without encountering congestion control, a process of the simulated stack rather than the examined program. These allocated bytes will be transmitted iteratively multiple times to examine the processing overhead of socket writes for the different programming languages. Upon completion of the specified number of transmission iterations, the client program will exit.

At the other end host, a hybrid server/client program is installed. The server aspect of the program binds a TCP socket to the host and sets it to listen on the same port that the client on the other host will attempt to connect. When the client makes a connection to the server, the server program will accept the connection and reads bytes from the socket until the client closes the connection. Upon closing the initial socket connection, the server program will store the number of received bytes, increment the port number, and relay the same amount of bytes back to the first end host on a new TCP connection. Ready to accept packets on this new port is an ns-3 PacketSink application ready to call the `Simulator::Stop` method when it receives the expected number of bytes. The server-turned-client will iteratively transmit 1,024 bytes until it has transmitted as many bytes as it originally received.

4.3 Ping Ring

A ring topology is simulated to examine the scalability of ns-3 when it must handle multiple DCE applications. The ring consists of a variable number of routers connected to one another in a circle. Point-to-point links connect the routers with a data rate of 100Mbps and 1ms delays. Each router is connected to an end host through a separate point-to-point link with 10Mbps rates and 1ms delays. Network interfaces for all links are configured similarly to those in the dumbbell topology using different 255.255.255.0 subnets (CIDR /24) to again confirm packet transmission occurs through successful L3 routing and not simply ARP requests.

Within the topology, one end host is selected for DCE application installations. The applications are the same applications described in 4.1.3. However, instead of testing for the *localhost* address, the end host attempts to reach every other end host in the topology. Pings are sent 1,000 times for each end host to inflate the amount of processing required for each DCE application.

5. RESULTS AND DISCUSSION

All of the described experiments have been performed on a system running 64-bit Linux Mint 17.3 with a 2.5GHz dual-core Intel i5-3210M processor with 8GB of memory. Simulations are executed 10 times for each set of parameters to obtain the average total memory usage in MB and average wallclock execution time. Gathering memory usage is accomplished through the `ps_mem` Python utility, which determines the core memory usage for a running process[9]. During each simulation run, this tool is called periodically to record the maximum usage realized by the ns-3/DCE simulation at any point in execution. This statistic is important as insufficient memory any time during the life of a program would inhibit successful simulation completion.

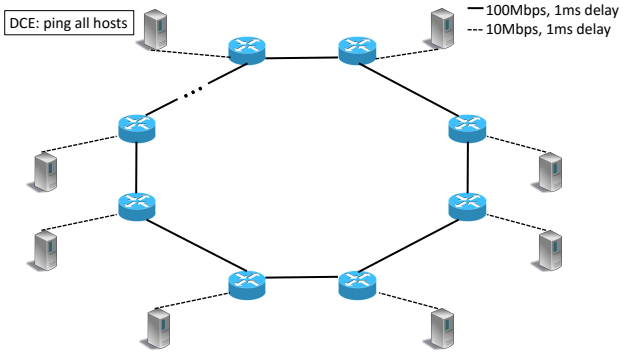

Figure 2: The ring topology is used to examine the scalability of ns-3 simulations when handling multiple DCE applications. The network is comprised of a variable number of router nodes connected in a ring with an end host connected to each one. One host is tasked to ping every other end host a specific number of times.

The benchmark results are shown in Figures 3 and 4 alongside results of each application executed natively in the real Linux environment rather than the ns-3/DCE simulated environment. (The local ping benchmarks are not compared against native versions due to timing differences in the application designs in each programming language.) Java is run both regularly and with the `-Xint` flag to allow it to run in interpreted mode without some of the "performance benefits" of the HotSpot JVM. The programs compiled with GCC and Clang required roughly the same amount of memory and produced similar time results. Based on the programs used in the benchmark simulations, the resource results are as expected. Most of the variables in the C programs had been allocated with stack memory and relatively few variables were utilized, providing little room for any significant compiler optimizations. Interestingly, the Clang-compiled executable for matrix multiplication produced a slight timing improvement over its GCC-built counterpart. This result may suggest that Clang realized some compiler optimizations such as loop unrolling that GCC may not have attempted. Even so,

the timing discrepancies between the two compiler versions are still relatively negligible when compared to the Python and Java versions of the applications. Based on the notion that both Python and Java are effectively invoking C/C++ calls within their underlying libraries, a certain amount of computational overhead is expected. In most of the benchmark tests, a tradeoff appears between lower memory usage with higher wallclock times in Python versus higher resource requirements with quicker execution performance in Java. The printing of the digits of π as well as the threading test produced a relatively significant discrepancy between resource usage and execution timing. However, this tradeoff is actually not realized for the matrix multiplication program. This result suggests that dynamically typed `list` allocation and assignment for the Python arrays incurs both a resource and performance cost compared to the statically typed integer array in Java. Additionally, within the DCE environment, the "performance benefits" of the HotSpot JVM appear to provide some level of speed-up to interpreted-only Java. However, these benefits come with increased memory usage in some cases. Threading appears to be one of the few cases that benefit in terms of memory and time from full access to the JVM. The local ping results are relatively flat compared to the other benchmarks most likely implying that the overall program ultimately lacked a significant amount of processing.

Simple dumbbell topology results are graphed in Figures 5 and 6. For the applications written in C, memory remains approximately constant as address space is simply and explicitly reused while performance intuitively requires more processing time as more processing is required. The Java versions of the programs had to be run in interpreted mode in order to complete. Full usage of the HotSpot JVM in this context is prevented due to its alternation between periods of optimization and deoptimization for profiling and debugging. In testing the framework, it was determined that this feature actually presented conflicts for the addressable space that DCE maintains. Again, an overhead is noted for the interpreted Java and Python programs compared to the C versions. However, interpreted Java memory usage appears to approach a limit suggesting potential memory reuse and adequate garbage collection. On the other hand, Python continues to require more memory with more transmission iterations. One possible reason behind this result may be that Python will continue to dynamically allocate memory as its applications transmit and receive data without reaching a point where garbage collection is deemed appropriate by the interpreter.

Figures 7 and 8 display the results of the ring topology experiments. Starting multiple ping applications for the native code did not require significant overhead in terms of memory or time. In this set of experiments, Python produced wallclock times between Java and interpreted-only Java. However, its memory usage was significantly lower than the dumbbell topology results. The Java program benefited from the `-Xint` flag in this set of experiments, producing lower wallclock times than the Python version with a slight memory usage improvement over the full JVM. However, both experimental runs of the Java program produced significantly higher resource usage than the other programs. This memory usage trend may have been a consequence of starting and stopping the JVM multiple times.

During testing of the OpenJDK within DCE, multiple applications running simultaneously within the limited address space of DCE made it difficult for the JVM to find free space for its code heap. Two Java applications running at the same time as in the dumbbell topology experiments appeared to be the stable limit while more than two simultaneous applications produced inconsistent successful results. DCE applications within the ring topology experiments were given staggered start times such that one would complete before another began. When Java applications had dedicated individual access to the Java runtime as in the ring topology experiments, the JVM resources could be successfully launched and deconstructed without interference. Testing for potential solutions to the simultaneous usage issue is ongoing. However, sufficient use cases are available to advocate for usage of the current framework.

6. RELATED WORK

DCE has previously been used in numerous experiments to enhance the realism of network simulations. Demonstrations of DCE in literature have primarily focused on the fields of mobile and wireless networks where updated protocols are introduced frequently as the technology improves. Rather than constantly porting and modifying these procedures into simulation, it can be much quicker to simply introduce these protocol implementations into simulation via DCE. In [10], a comparison of the ns-3 Optimized Link State Routing (OLSR) model with an actual OLSR daemon in DCE uncovered deficiencies in both programs. Addressing tuning issues in the simulated model and the daemon program allowed them to be updated to better fit the OLSR RFC. Demonstrations of content-centric networking (CCN) over mobile ad-hoc networks (MANETs) and multipath TCP over LTE and wireless in [11, 12] provide examples of additional use cases for DCE that required no modifications to the original implementations. However, these examples are limited to software binaries that originated from C/C++ source code whereas this work has provided a proof of concept and framework for testing protocols in Java and Python in addition to C/C++.

The rise of software-defined networking (SDN) in the network communications field over the past few years has provided another valuable avenue for utilizing DCE. In separating the control and forwarding planes of a network, SDN employs a separate controller process for handling the routing decisions of a network. These controller processes are implemented as libraries, such as NOX (C++), POX and Ryu (Python), Floodlight and OpenDaylight (Java), etc. NOX and the Open vSwitch virtual switch kernel have already been demonstrated successfully in [13] for use cases in both wired and wireless networks. The framework discussed in this work will surely benefit the testing of controller applications written for some of these other libraries as well.

The Open Network Emulator (ONE) [14] is another network simulator providing direct code execution functionality. It provides a compiler framework that converts real-world network applications into modules that can be integrated into simulated network stacks. The source code for these applications requires no modification prior to compilation within the ONE framework. Applications compiled for ONE are built with LLVM into the LLVM intermediate representation (IR)[15]. This notion provides the claim that ONE modules can be language and architecture independent. However, the published literature on ONE and its dis-

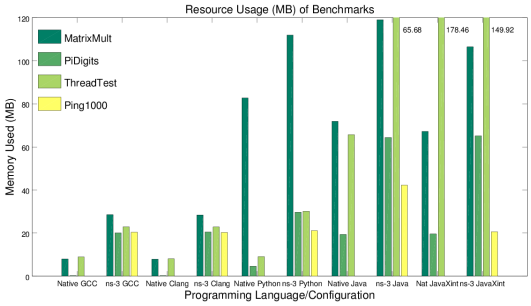

Figure 3: Performance benchmark total memory usage in MB for a single DCE-enabled node and native equivalents for the various programming languages, compilers, and configurations. All data have standard errors that are less than 2% of their respective reported values.

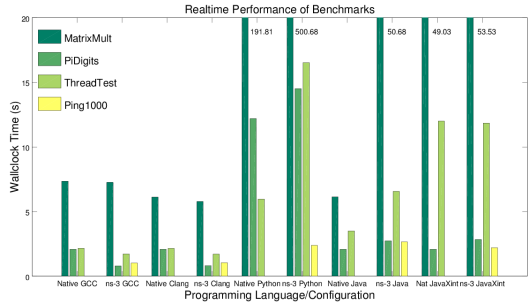

Figure 4: Performance benchmark wallclock execution time for a single DCE-enabled node and native equivalents for the various programming languages, compilers, and configurations. All data have standard errors that are less than 2% of their respective reported values except the GCC and Clang versions of the simple thread test. Their standard errors are 6% of their reported values.

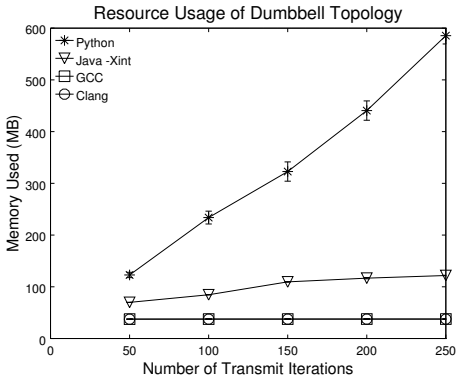

Figure 5: Total memory usage in MB for the simple dumbbell topology for the programming languages, compilers and configurations. Standard error bars are displayed for each data point.

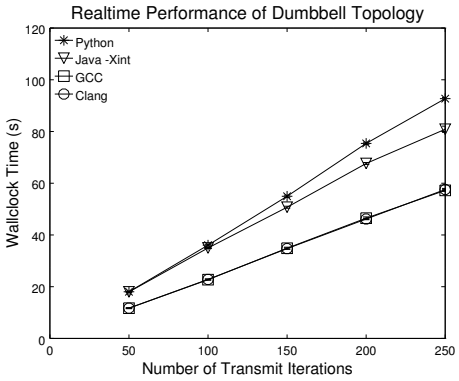

Figure 6: Wallclock execution time for the simple dumbbell topology for the programming languages, compilers and configurations. Standard error bars are displayed for each data point and noted as negligible.

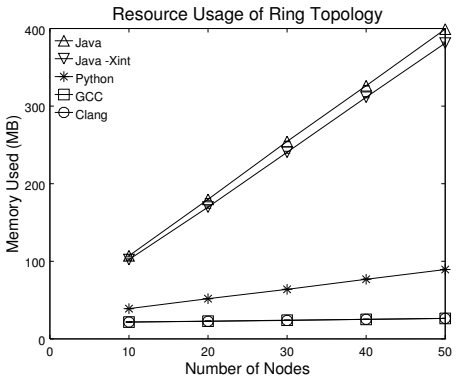

Figure 7: Total memory usage in MB for the ring topology for the programming languages, compilers and configurations. Standard error bars are displayed for each data point and noted as negligible.

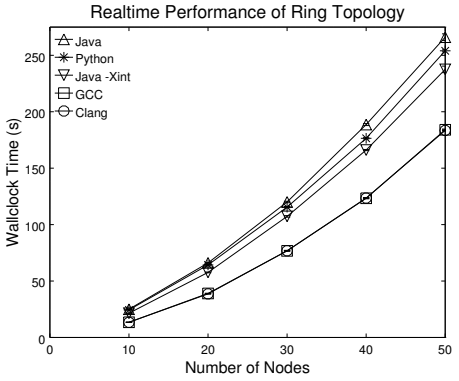

Figure 8: Wallclock execution time for the ring topology for the programming languages, compilers and configurations. Standard error bars are displayed for each data point and noted as negligible.

tributed descendant[16] only examine applications written in C and C++. Based on the subprojects for LLVM[17], these applications are most likely compiled with Clang. For other languages, the DragonEgg plugin allows LLVM to interface with GCC parsers. One such parser, GCJ[18], allows Java source code to be compiled directly to native code. However, it utilizes segmentation fault signals to determine the limits of addressable memory, making it difficult to integrate with DCE.

7. CONCLUSIONS AND FUTURE WORK

This work has described and analyzed a framework using DCE for examining network applications written in Python or Java within simulated topologies in ns-3. This framework provides a convenient way to test entirely new sets of applications written in Python or Java that were not previously compatible to the DCE environment. The framework itself exploits the fact that Python and Java applications are launched and translated ultimately via native code with which DCE can interface. Parts of the underlying language libraries that required modification for successful DCE interaction have been discussed. Multiple applications on topologies of varying complexity have been studied to confirm successful operation of the framework. The resource usage and wallclock execution time of multiple applications written in C, Java, and Python have been gathered and studied to gauge the overhead required to run these applications within simulation.

Future work will continue to enhance the framework to automate some of the discussed environment and library modifications. Efforts to test SDN libraries such as POX and Ryu within the DCE framework have already had preliminary success. Testing will be expanded to determine how well Java controller libraries such as Floodlight and OpenDaylight integrate into the discussed framework. Furthermore, attempts to address the GCJ issue discussed in section 6 will continue in order to better gauge the benefits and shortcomings of the described framework against simply compiling the applications to native code.

8. REFERENCES

[1] M. Gupta, J. Sommers, and P. Barford. Fast, accurate simulation for SDN prototyping. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 31–36, New York, NY, USA, 2013. ACM.

[2] D. Camara, H. Tazaki, E. Mancini, T. Turletti, W. Dabbous, and M. Lacage. DCE: Test the real code of your protocols and applications over simulated networks. *Communications Magazine, IEEE*, 52(3):104–110, March 2014.

[3] H. Tazaki, F. Urbani, E. Mancini, M. Lacage, D. Camara, T. Turletti, and W. Dabbous. 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, pages 217–228, New York, NY, USA, 2013. ACM.

[4] Inria. Direct code execution (DCE) manual. Website, 2013. https://www.nsnam.org/docs/dce/release/1.5/manual/html/index.html.

[5] H. Tazaki. dce-pre.patch. Website, 2014. https://gist.github.com/thehajime/b2efe96e797cd131cac1.

[6] S. Rabinowitz and S. Wagon. A spigot algorithm for the digits of π. *The American Mathematical Monthly*, 102(3):195–203, 1995.

[7] M. Muuss. The story of the ping program. Website, 1999. http://ftp.arl.army.mil/~mike/ping.html.

[8] J. Diemer. python-ping. Website, 2011. https://pypi.python.org/pypi/python-ping.

[9] P. Brady. ps_mem 3.6: Python package index. Website, 2015. https://pypi.python.org/pypi/ps_mem/3.6.

[10] E. Bikov and P. Boyko. 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, pages 454–461, ICST, Brussels, Belgium, Belgium, 2011. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[11] S. Kwon, K. Hasan, M. Lee, and S. Jeong. Comparative analysis of real-time video performance in the CCN-based LTE networks. In *Information and Communication Technology Convergence (ICTC), 2015 International Conference on*, pages 509–511, Oct 2015.

[12] H. Tazaki, E. Mancini, D. Camara, T. Turletti, and W. Dabbous. 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, pages 29–32, New York, NY, USA, 2013. ACM.

[13] E. Mancini, H. Soni, T. Turletti, W. Dabbous, and H. Tazaki. Demo abstract: Realistic evaluation of kernel protocols and software defined wireless networks with DCE/ns-3. In *Proceedings of the 17th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, MSWiM '14, pages 335–337, New York, NY, USA, 2014. ACM.

[14] V. Duggirala and S. Varadarajan. Open network emulator: A parallel direct code execution network simulator. In *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, PADS '12, pages 101–110, Washington, DC, USA, 2012. IEEE Computer Society.

[15] C. Lattner and V. Adve. LLVM: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86, March 2004.

[16] V. Duggirala, C. Ribbens, and S. Varadarajan. Distributed ONE: Scalable parallel network simulation. In *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*, SimuTools '13, pages 10–16, ICST, Brussels, Belgium, Belgium, 2013. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[17] C. Lattner. The LLVM compiler infrastructure project. Website, 2016. http://llvm.org.

[18] GCC Team. GCJ: The GNU compiler for Java. Website, 2016. https://gcc.gnu.org/java.