Proceedings of the 2012 Winter Simulation Conference
C. Laroque, J. Himmelspach, R. Pasupathy, O. Rose, and A. M. Uhrmacher, eds.

VALIDATION OF APPLICATION BEHAVIOR ON A VIRTUAL TIME INTEGRATED NETWORK EMULATION TESTBED

Yuhao Zheng
Dong Jin
David M. Nicol

University of Illinois at Urbana-Champaign
Information Trust Institute
1308 West Main Street
Urbana, IL , 61801, USA

ABSTRACT

Combination of emulation and simulation offers the hope of both functional and temporal fidelity when modeling large scale networks and the applications that use them. Emulation of unmodified software gives functional fidelity, but not necessary temporal fidelity. We addressed this in prior work by embedding the OpenVZ virtual machine system in virtual time. Validation reveals that there are network timing errors whose magnitude depend on the length of a virtual machine execution timeslice. A natural question asks to what degree these errors impact the behavior of *applications*. For instance, if an application is relatively insensitive to these errors, we can increase performance by allowing larger emulation timeslices. We study a variety of applications with different network and CPU demands. We find, surprisingly, that difference in application behavior due to simply using OpenVZ often dominate the errors, implying that we need not be overconcerned about errors due to larger timeslices.

1 INTRODUCTION

The successful deployment and advancement of many large-scale networked systems requires high performance, high reliability, and secure underlying communication networks. A testbed is an important tool for studying these networks and their applications. Researchers have created various testbeds that use emulation or simulation, for conducting medium to large scale experiments. Simulation testbeds provide a highly configurable, repeatable and scalable network environment at low cost, but trade functional fidelity for scale and speed. Network emulation ensures functional fidelity by running real applications on physical machines, but is limited by budget. To draw upon strengths of both types of testbed, we integrate the light-weighted OpenVZ-based emulation system (OpenVZ 2012) with a parallel network simulator (Jin, Zheng, Zhu, Nicol, and Winterrowd 2012).

Our testbed provides both functional and temporal fidelity, by embedding the virtual machines in virtual time (Zheng and Nicol 2011). However, small temporal errors are introduced by the OpenVZ design, on the scale of a timeslice given to virtual machines, because OpenVZ interacts with a virtual machine only at the beginning and end of a timeslice. This paper asks how temporal errors affect behavioral fidelity with respect to application-specific metrics. We study applications that are network-intensive, and ones that are CPU-intensive. We also evaluate behavioral fidelity on ICMP, UDP and TCP by studying FTP, web browsing, ping, and iperf. Our study concerns three configurations: native Linux, native OpenVZ, and our emulation/simulation testbed. By comparing native Linux and native OpenVZ we identify deviations that are due solely to OpenVZ's implementation. Comparing native Linux and our emulation/simulation testbed we see the impact of those errors and errors introduced by our testbed.

The experimental results show that application timing errors introduced by the virtual time system are bounded by the size of an emulation timeslice in both network-intensive and CPU-intensive applications, and that the network round-trip-time (RTT) may increase by as much as 2 emulation timeslices. Applications whose metrics are not affected by changes in the RTT of that scale are insensitive to emulation errors. We also see that errors introduced by OpenVZ are generally larger than errors introduce by the virtual time system, especially for TCP-based applications. The importance of these observations is that we can tune the virtual time errors by changing the size of the emulation timeslice, and so, as a function of application, choose timeslices that do not significantly impact application performance.

The remainder of the paper is organized as follows: Section 2 overviews the design of the testbed and describes the virtual time system implemented in the OpenVZ-based emulation; Section 3 shows the setup of the experiment framework for conducting the validation tests under various network conditions; Section 4 presents and analyzes the experimental results for various type of applications; Section 5 discusses the related work; and Section 6 concludes the paper.

2 SYSTEM OVERVIEW

2.1 System Architecture Design

We have developed a large-scale, high-fidelity network testbed by integrating the OpenVZ network emulation into a S3F-based network simulation framework. Figure 1 shows the overall system architecture on a single physical Linux box. The system consists three layers: an OpenVZ virtual machine manager for network emulation, S3F—the kernel of the system, and S3FNet—a parallel network simulator.

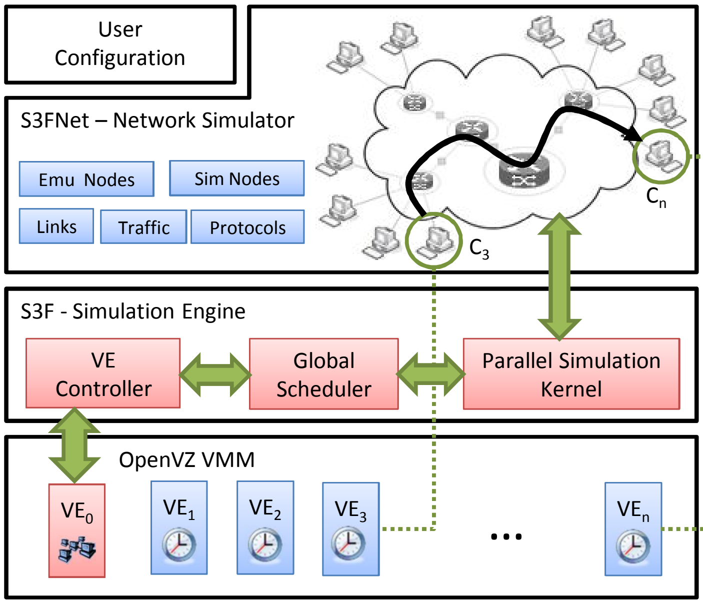

Figure 1: System Design Architecture.

2.1.1 OpenVZ-based Network Emulation

OpenVZ is an OS-level virtualization technology, which creates and manages multiple isolated Linux containers, also called Virtual Environments (VEs), on a single physical server. Each VE performs exactly like a stand-alone host with its own root access, users and groups, IP and MAC address, memory, process tree, file system, system libraries and configuration files, but shares a single instance of the Linux OS kernel for services such as TCP/IP. Compared with other virtualization technologies such as Xen (para-virtualization) and QEMU (full-virtualization), OpenVZ is light-weight (e.g., we are able to run more than 300 VEs on a commodity server with 8 2GHz cores and 16 GB memory), at the cost of diversity in the underlaying operating system.

OpenVZ emulation allows users to execute real software in the VEs to achieve high functional fidelity. It also frees a modeler from developing complicated simulation models for network applications and the corresponding protocol stacks. Besides functional fidelity, our version of OpenVZ also provides temporal fidelity through a virtual clock implementation (Zheng and Nicol 2011), allowing an integration with the simulation system. More details on virtual clock will be discussed in Section 2.2.

2.1.2 S3F – Kernel of the Emulation/Simulation System

SSF (Scalable Simulation Framework) defines an API that supports modular construction of simulation models, with automated exploitation of parallelism. Building on ten years of experience, we developed a second generation API named S3F (Nicol, Jin, and Zheng 2011), which supports the OpenVZ-based emulation experiments based on the notion of virtual clock. The VE controller and the global scheduler shown in Figure 1 are responsible for managing the interaction between the simulation and the emulation systems.

The *VE controller* is designed for controlling all the emulation hosts according to S3F's command, as well as providing necessary communications between VEs and their corresponding simulated nodes. VE controller is a special API that offers the following three services: advance emulation clock, transfer packets bidirectionally between simulation and emulation, and provide emulation lookahead. More details on the VE controller are in (Jin, Zheng, Zhu, Nicol, and Winterrowd 2012).

The *Global scheduler* coordinates safe and efficient advancement of the two systems. It ensures that the emulation runs ahead of the simulation, generating the network's offered load, but not so far ahead to miss delivery of traffic. It also schedules the sequence and length of each emulation and simulation execution cycle to preserve causal relationships. Each VE has its own virtual clock, which is synchronized with the simulation clock in S3F. The complete synchronization algorithm used by the global scheduler is described in (Jin, Zheng, Zhu, Nicol, and Winterrowd 2012). In addition, the global scheduler makes the emulation integration nearly transparent to the upper layer network simulator.

2.1.3 S3FNet – Network Simulator

S3FNet is a network simulator built on top of the S3F kernel. While it is capable for creating models of network devices (e.g., host, switch, router) with layered protocols (e.g., IP, TCP, UDP) like other network simulators, the primary usage of S3FNet in the entire system is to simulate a sophisticated underlying network environment (e.g. detailed CSMA/CD for traditional Ethernet, CSMA/CA for wireless communication), and to efficiently model the extensive communication and computation in large-scale experiment settings (e.g., background traffic simulation model (Nicol and Yan 2006) (Jin and Nicol 2010)).

S3FNet has the global view of the network topology. Every VE in the OpenVZ model is represented in the S3FNet model as a host within the modeled network, together with other simulated nodes. Within S3FNet, traffic that is generated by a VE emerges from its proxy host inside S3FNet, and, when directed to another VE, is delivered to the recipient's proxy host as shown in Figure 1. Only the global scheduler in S3F knows the distinction between an emulated host (VE-host) or a virtual host (non-VE host).

2.2 Virtual Time System

In our system VEs perceive time exactly as they do in the real world (Zheng and Nicol 2011). In a real system, an application perceives time elapses when doing computation or when waiting for I/Os. Similarly, a virtual clock advance between when its VE starts to execute and stops, and when the VE starts a blocking I/O, and when it finishes.

We use a timeslice-based mechanism to implement our virtual time system. A VE can run only after the VE controller has given it a timeslice and released it. Without being given a timeslice, a VE will remain suspended and its clock does not advance. When an event occurs to unsuspend it (e.g., arrival of a message being blocked for), the clock is advanced to the time of the event, leaping over an idle epoch. While a VE is suspended, the simulator can perform arbitrarily long simulation computation as this will not affect the VE's state. This frees our emulation from having to run in real time—emulation can now run either faster or slower than real time, depending on the system load and the simulation model that is being used.

Another advantage is that the VE controller is able to control VE executions to prevent some VEs from running too far ahead (in terms of virtual time), potentially leading to causal violations. In regards to VE synchronization, we currently use a barrier-based mechanism. In particular, S3F computes a barrier (a virtual time value) to which all VEs are safe to advance, and let all VEs run to this barrier. Within two consecutive barriers, VEs are considered independent, and any VE interactions are postponed to the barrier. Details of the synchronization algorithm is provided in (Jin, Zheng, Zhu, Nicol, and Winterrowd 2012).

Our virtual time system introduces some timing errors via the timeslice-based mechanism. Once a VE receives a timeslice, the VE controller cannot interact or stop the VE until the timeslice has expired, on the hardware interrupt. In particular, the VE controller cannot deliver an event (i.e. network packet arrival) to the VE at the exact virtual time the packet leaves the simulation. Events occur at timeslice boundaries and so any error is bounded in magnitude by the length of the timeslice (Zheng and Nicol 2011). We may reduce the error bound by setting a smaller hardware interrupt interval (Zheng, Nicol, Jin, and Naoki 2012). Nevertheless, this cannot be arbitrarily small due to efficiency concerns and hardware limits. We typically use 100 μs as the smallest timeslice.

3 EXPERIMENT SETUP

Figure 2 illustrates our experiment framework, which consists of three components: an end-host running a server side application, an end-host running a client side application and an intermediate host that serves as a traffic controller. The traffic controller is a Linux application for configuring test scenarios with various network conditions including bandwidth, packet drop rate and packet delay. We duplicate the same network topology onto three platforms. The platform in Figure 2(a) consists only the physical hosts, which serves as the ground truth data collector. The second platform, as shown in Figure 2(b), has end-host applications running inside the OpenVZ virtual machine (VE) instead of the real operation system; comparison of behaviors on this with those of the applications on the first platform reveals the difference introduced by the OpenVZ techniques. The same topology is also created in our virtual-time system enabled testbed, as shown in Figure 2(c). The setup is composed of three virtual machines running on a single physical machine. Comparison of behaviors on this with behaviors on the pure OpenVZ topology reveals the errors our virtual time techniques introduce.

We use the identical hardware across all three platforms. Each physical machine is equipped with a 2.0 GHz dual-core processor, 2 GB memory and gigabit Ethernet network interface cards. Also, we create the same software environment for all platforms, including the same OS (Red Hat Enterprise Linux 5 with 2.6.18 kernel), the same version of libraries and drivers, the same testing applications and the same setting of network parameter (e.g. sending/receiving buffer, ip routing table). Finally, the traffic controller alters data packets in a deterministic manner, coordinated across architectures, using a random number generator to select packets to drop on the flows of interest. Therefore, when the i^{th} packet is dropped in any one of the configurations, the i^{th} packet is dropped in all of them. In this way, we ensure that on an experiment-by-experiment basis, we are comparing precisely the same context for measuring the application-level network metrics.

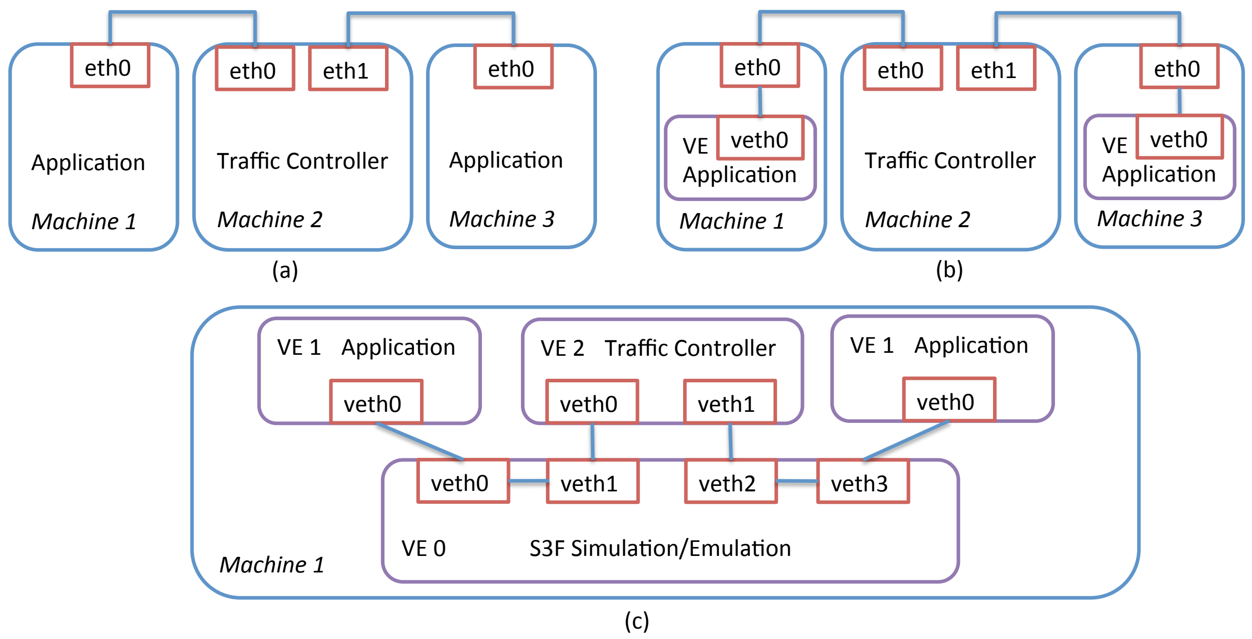

Figure 2: Testbeds Setup (a) Native Linux (b) Native OpenVZ (c) OpenVZ with Virtual Time.

4 EXPERIMENT DATA AND ANALYSIS

4.1 Network-intensive Applications

The first set of applications we study are network-intensive applications (ICMP, UDP, TCP). The experiments, run on each testbed platform, vary bandwidth, delay and loss. The data shown is based on a $100\mu s$ timeslice. The platform index number 1, 2 and 3 used in every table in this section represents the native Linux, native OpenVZ and OpenVZ with virtual time system respectively as shown in Figure 2.

4.1.1 Icmp

We use the ICMP protocol by pinging from one end-host to the other end-host under different network conditions controlled by the intermediate node. Ping is the commonly used utility application for testing the reachability of a host on an IP-based network and for measuring the round-trip time (RTT) for messages (ICMP echo request and response packets) sent from the originating host to a destination host and record any packet loss. The measured RTTs are listed in the Table 1.

Comparison of Testbed 1 (native Linux) and Testbed 2 (OpenVZ) shows the processing delay in bridging the veth and eth interface. We see the total processing overhead is approximately 0.1 ms, a cost due entirely to using OpenVZ, independent of virtual time overheads. Comparison between Testbed 2 and Testbed 3 shows the timeslice error explained in Section 2.2. A round-trip in this topology contains four hops (from Machine 1 to Machine 2 and back to Machine 1) and thus the worst case error is $400\mu s$. Indeed, the largest observed error is about $200\mu s$, and this matches the error bound of our system.

4.1.2 Udp

We set up an iperf (Iperf 2012) UDP server and client pair at the two end-hosts. The iperf client sends constant bit rate (CBR) UDP traffic under various network conditions, and the packet loss rate, throughput and jitter are recorded in Table 2 for comparison.

The client sends data at a CBR equal to the link bandwidth. However, the bandwidth specified in iperf is the end-to-end (application layer) bandwidth. When the data is transmitted over the network and

Table 1: Ping.

Network Condition		**Result**														
Loss	**Delay**	**Loss (%)**			**RTT min (ms)**			**RTT avg (ms)**			**RTT max (ms)**			**RTT mdev**		
(%)	**(ms)**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**
0	1	0	0	0	2.19	2.32	2.58	2.24	2.36	2.67	2.27	2.40	2.69	0.03	0.02	0.04
0	10	0	0	0	20.1	20.2	20.5	20.2	20.3	20.6	20.2	20.3	20.6	0.04	0.05	0.05
0	100	0	0	0	200	200	200	200	200	200	200	200	200	0.00	0.00	0.00
20	1	30	30	30	2.19	2.33	2.68	2.24	2.36	2.69	2.27	2.41	2.69	0.04	0.02	0.01
20	10	30	30	30	20.2	20.3	20.5	20.2	20.3	20.6	20.2	20.3	20.6	0.00	0.00	0.05
20	100	30	30	30	200	200	200	200	200	200	200	200	200	0.00	0.00	0.00
50	1	80	80	80	2.22	2.30	2.68	2.23	2.33	2.68	2.24	2.34	2.69	0.01	0.02	0.01
50	10	80	80	80	20.1	20.3	20.5	20.2	20.3	20.6	20.2	20.3	20.6	0.06	0.00	0.06
50	100	80	80	80	200	200	200	200	200	200	200	200	200	0.00	0.00	0.00

is added the network headers, the raw network data rate slightly exceeds the available bandwidth. This is the reason we observe packet losses even in the cases that link are not lossy, and those losses are caused by buffer overflow at the traffic controller.

We observe nearly identical results across all three platforms, especially the throughput has smaller than 1% error. Unlike TCP, there is no feedback loop in UDP, and the temporal error of a single packet does not propagate and cascade.

Table 2: UDP - Iperf.

Network Condition			**Result**								
Loss	**Delay**	**BW**	**Loss (%)**			**Throughput (Mb/s)**			**Jitter (ms)**		
(%)	**(ms)**	**(Mb/s)**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**
0	1	10	1.6	1.6	1.6	9.73	9.73	9.73	0.067	0.043	0.108
0	1	100	2.5	2.4	2.4	97.9	98.0	98.0	0.070	0.046	0.055
0	1	400	3.3	3.4	3.3	389	392	392	0.046	0.032	0.038
0	10	10	1.7	1.7	1.7	9.73	9.73	9.73	0.052	0.048	0.057
0	10	100	2.5	2.5	2.5	98.0	98.0	98.0	0.056	0.050	0.060
0	100	10	2.6	2.7	2.5	9.73	9.71	9.73	0.101	0.128	0.145
5	10	10	5.1	5.1	5.1	9.48	9.54	9.48	0.039	0.044	0.037
5	10	100	5.0	5.1	4.9	95.5	95.4	95.6	0.042	0.050	0.062
10	10	10	10	10	10	8.98	9.03	8.98	0.056	0.051	0.055

4.1.3 Tcp

Iperf

We set up the iperf TCP server and client and use the traffic controller to adjust the length of delay, loss rate and the available bandwidth to create various network testing scenarios. All the TCP related parameters, such as size of sending buffer and receiving buffer, are set to be the same (128 KB) in native Linux and

OpenVZ. We keep sending traffic for 30 seconds for all the experiments and record the throughput, which is the primary indication of TCP connection performance, in Table 3.

Throughputs from platforms 2 and 3 are very close, under all cases, suggesting that the small errors in virtual time do not impact throughput evaluation. However, in our first trial of these experiments we saw a large difference between platforms 1 and 2, with (surprisingly) platform 2 yielding a significantly larger throughout! Investigation revealed that OpenVZ configuration allows for control of certain TCP buffer sizes, and that were set to be larger than native Linux uses. After we aligned all TCP configurations possible, we still see differences between native Linux and native OpenVZ. In particular, in all the loss-free scenarios, TCP traffic in the native Linux has better performance than the OpenVZ-based Linux.

To understand the root cause of this difference, we instrumented the code to print out the size of the TCP send window, as a function of segment number; Figure 3 plots the result when we induce network conditions delay = 1 second, loss = 0, and bandwidth = 1 Mb/s. Platforms 2 and 3 have essentially identical results, but a significant difference is seen between platforms 1 and 2, with the send window size in congestion avoidance mode bing 34 for native Linux and 29 for OpenVZ. We also see different growth in the window size during the slow start mode. The differences strongly suggest some fundamental difference between the TCP implementation or configuration in native Linux and native OpenVZ.

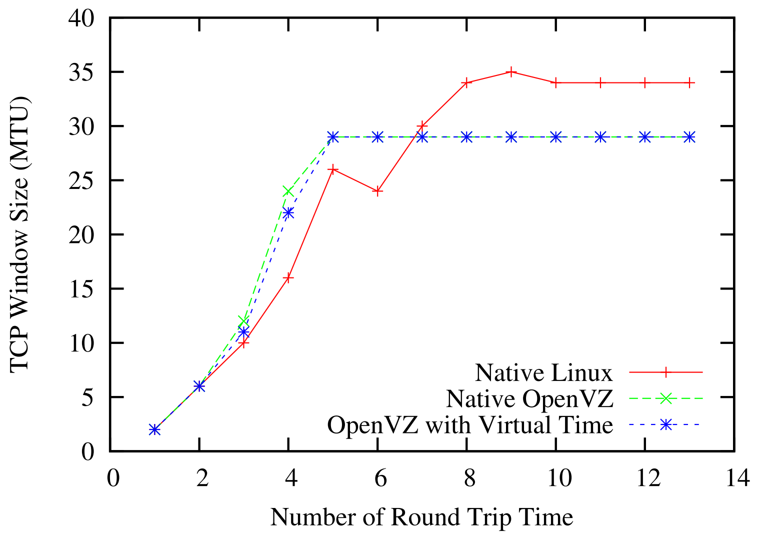

Figure 3: TCP Window Size.

FTP

FTP traffic is generally very tolerant of delay and loss. We setup an FTP server and an FTP client, programming the client to download a file. All the cases use a network bandwidth of 10 Mb/s. The throughput, transfer time, and connection establish time are recorded in Table 4.

In the first four loss-free cases, Testbed 2 and Testbed 3 behave similarly, but they are slightly different than Testbed 1 (up to 3% difference in throughput). The reason is the same as previous iperf TCP, as FTP uses TCP. The difference affects only the file transfer time but not the connection establish time, as the server and client only exchange control messages during the connection establish phase. These messages are delay sensitive but not throughput sensitive.

In the last three cases with losses, the throughput differences among three testbeds are enlarged. Again this is due to the difference in TCP implementation. Although our traffic controller outputs deterministic packet losses, they may behave differently on a same single packet loss, causing the different in achievable TCP throughput. During the connection establish phase, no packet losses are observed, thus all three testbeds have similar connection time.

HTTP

Table 3: TCP - Iperf.

Network Condition			**Result**		
Loss	**Delay**	**BW**	**Throughput (Mb/s)**		
(%)	**(ms)**	**(Mb/s)**	**1**	**2**	**3**
0	1	10	9.63	9.59	9.59
0	1	100	94.1	94.5	95.7
0	1	400	131	129	133
0	10	100	17.9	15.8	15.8
0	100	10	1.79	1.60	1.61
0	1000	1	0.157	0.137	0.133
1	10	10	4.06	4.89	4.83
2	10	10	2.93	3.25	3.26
5	10	10	1.74	1.70	1.78

Table 4: FTP.

File	**Network Condition**		**Result**											
Size	**Loss**	**Delay**	**Throughput (KB/s)**			T_{Total} **(s)**			$T_{Transfer}$ **(s)**			$T_{Initial}$ **(s)**		
(MB)	**(%)**	**(ms)**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**
10	0	1	1140	1140	1140	8.9	8.9	8.9	8.8	8.8	8.8	0.1	0.1	0.1
10	0	10	1130	1140	1130	9.1	9.0	9.1	8.8	8.8	8.8	0.3	0.2	0.3
1	0	100	180	186	185	8.1	7.9	8.0	5.7	5.5	5.5	2.4	2.4	2.5
1	0	1000	20.4	18.3	18.9	74.3	80.1	78.1	50.2	56.0	54.2	24.1	24.1	23.9
1	1	100	79.5	96.1	108	15.3	13.2	11.9	12.9	10.7	9.5	2.4	2.5	2.4
1	2	100	41.6	42.4	41.4	27.0	26.5	27.2	24.6	24.2	24.7	2.4	2.3	2.5
1	5	100	29.6	29.1	29.3	36.9	37.7	37.3	34.6	35.2	34.9	2.3	2.5	2.4

Hypertext Transfer Protocol (HTTP) is the data communication protocol for the world wide web. Web browsing is generally tolerant of moderate delay and loss. We setup an apache server on one end-host (Apache 2012) and a text-based web browser, named lynx (Lynx 2012), on the other end-host. We grabbed the openvz.org site with one level depth (105 files, 2848KB in total) and host those contents in our apache server. In this way, we can produce some typical web traffic which consists of a series of small and bursty file transfers. In the experiments, the client is configured to traverse all the first-level links and reports the total traversal time. The cache is cleared at the beginning of every run. The network bandwidth is set to 10 Mb/s for every experiment. We run each experiment for 10 times and the results are shown in Table 5.

We observe that traversing all web pages takes longer time in the OpenVZ-based Linux than in the native Linux. This is due to the processing delay in bridging the virtual network interface in OpenVZ (e.g., "veth0") and the real Ethernet interface (e.g., "eth0").

Also, testbed 3 has a smaller traversal time than testbed 2 in all the loss-free scenarios. The reason is that our embedding of OpenVZ in virtual time does not yet account for delays in file I/O, a deficiency in our implementation we will shortly be rectifying.

In addition, large randomness is observed for all the cases with packet loss. We carefully studied traces of different runs, and discovered this is due to multi-threaded web client/server applications. In particular, the web client application launches multiple TCP connections to request objects from the server and each connection is a thread. Since the multi-thread scheduling in Linux is non-deterministic, the packet sending sequences can be different across multiple runs of each test case. Therefore, the traffic controller could drop different packets though itself is designed to produce packet loss pattern deterministically. The dropped HTTP packets are not uniformly important — control packet losses have larger impact on the overall timing than do data packet losses. Such randomness in the application behavior is not introduced by our system and should be expected in a multi-thread execution environment.

Table 5: HTTP.

Network Condition		**Result**					
Loss	**Delay**	**Website Traversal Time (s)**			**Stddev (s)**		
(%)	**(ms)**	**1**	**2**	**3**	**1**	**2**	**3**
0	1	5.1	6.0	4.5	0.0	0.0	0.0
0	10	12.3	12.5	11.9	0.1	0.1	0.1
0	100	91.6	92.2	91.6	0.1	0.1	0.1
1	100	107.4	108.8	109.1	2.2	3.7	1.4
2	100	128.1	129.9	130.2	9.2	9.8	4.6
5	100	230.7	231.5	230.6	36.3	45.0	20.5

Larger emulation timeslice

The network round-trip-time is increased by the emulation timeslice. As much as one timeslice is added on the server side, as it may take that much additional time for it to recognize a packet, and as much as one additional timeslice may be added for the client to recognized the server's response. These delays obviously directly impact "ping", but they also impact TCP and applications that use it. TCP throughput is approximately RWS/RTT, where RWS is the receiver window size and RTT is the round-trip delay. So, for instance, in our experiments where the network latency is 1 ms, using an emulation timeslice of 1 ms causes RTT to increase from 2 ms to 4 ms, effectively cutting throughput in half. Experiments using larger emulation timeslices confirm this sensitivity. This gives us some guidance on choice of emulation timeslice in network intensive applications using TCP.

4.2 CPU-intensive Applications

For CPU-intensive applications, we implemented the Client Puzzle protocol (Juels and Brainard 1999), which is used in many proof of work schemes for managing limited resources on a server and providing resilience to denial of service (DoS) attacks. In this protocol, when a client initiates a connection to a server, server will send client a puzzle to solve. The connection will be established only if the client correctly solve the puzzle. In particular, a puzzle is essentially a hash inversion problem, which currently has no efficient algorithm to solve but using brute force search. Table 6 documents the elapsed time for the client to set up a connection with the server. For consistency, the server is always using the same puzzle, but the client has no caches of previous puzzles. Each experiment uses a network bandwidth of 10 Mb/s.

We can see both Testbed 2 and Testbed 3 have very similar runtimes, yet Testbed 1 has slightly smaller ones. This is due to the overhead introduced by OpenVZ virtualization. Such overhead is small (around 3%), and it matches the advertised overhead of OpenVZ. The simulation/emulation overheads are excluded from the virtual clock of a container, and the container perceives time as if it were running independently. Moreover, we notice that Testbed 3 has a smaller standard deviation in runtime, indicating that its runtime is more stable and more repeatable. This is due to the virtual time system, which only counts the execution time performed by a VE into its virtual clock, excluding most other activities that may affect its runtime.

We conclude that our timeslice-based virtual time implementation yields high temporal fidelity not only for network packets but also to CPU computations. When the scheduler gives a timeslice to a VE, the actual amount of execution time received by the VE is usually slightly different from the timeslice length, due to some overhead and some interrupt-disabling routines in the Linux kernel. Our virtual time system uses an offset mechanism to compensate such difference (Jin, Zheng, Zhu, Nicol, and Winterrowd 2012), making the CPU computation time correct in long term. Without such mechanism, application runtime is less accurate and less repeatable.

Table 6: Puzzle.

Network Condition		**Result**					
Loss	**Delay**	**Average Time (s)**			**Stddev**		
(%)	**(ms)**	**1**	**2**	**3**	**1**	**2**	**3**
0	1	5.405	5.585	5.595	0.060	0.086	0.001
0	10	5.407	5.630	5.657	0.076	0.071	0.014
0	100	5.947	6.119	6.189	0.077	0.075	0.004
0	1000	11.383	11.593	11.585	0.091	0.059	0.004

5 RELATED WORK

5.1 Virtualization

Virtualization divides computer resources into multiple separated Virtual Environments (VEs). It has become increasingly popular as computer hardware is sufficiently powerful to drive multiple VEs, while providing acceptable performance to each. There are three different level of virtualization: 1) full virtualization, e.g. VMware (VMware) and QEMU (QEMU 2012), 2) para-virtualization, e.g. Xen (Xen 2012) and UML (UML 2012), and 3) operating system (OS) level virtualization, e.g. OpenVZ (OpenVZ 2012) and Virtuozzo (Virtuozzo 2012). Unlike full virtualization or para-virtualization that can run unmodified guest OS, OpenVZ guest VEs are OS templates that share the hosts Linux kernel. Although we can run unmodified applications inside a guest VE, it may perform differently from a real machine due to the shared kernel. Nevertheless, OpenVZ has the smallest overhead and thus offers greatest scalability.

5.2 Virtual Time System

While virtualization improves functional fidelity of network simulation and emulation, some recent effort focuses on temporal fidelity. DieCast (Gupta, Vishwanath, and Vahdat 2008), VAN (Biswas, Serban, Poylisher, Lee, Mau, Chadha, Chiang, Orlando, and Jakubowski 2009), and SVEET (Erazo, Li, and Liu 2009) modify Xen hypervisor that translate real time into a constantly slowed down virtual time, such that multiple guest domains can coexist on a single physical machine, yet they still perceive time as if they were running concurrently. Our virtual time implementation focuses on OpenVZ and achieves better scalability. In addition, unlike the Xen-based virtual time systems in which guest domains run continuously, our timeslice-based implementation advances VEs discretely. While a VE is suspended, its state including its virtual clock remains unchanged, such that the simulator may perform arbitrarily long computation. This mechanism totally frees emulation from real time constraint. In regards to validation, DieCast (Gupta, Vishwanath, and Vahdat 2008) did an extensive work on various applications. While they validate their platform on a larger network but merely focus on only virtualization platforms, we also study the differences between native Linux platforms and virtualized ones.

6 CONCLUSION

We test and evaluate the application behaviors of a virtual-time-integrated, virtual-machine-based network emulation/simulation testbed. The test cases are designed to cover both network-intensive application and CPU-intensive applications over a group of commonly-used protocols. We found the minimal temporal error (100 μs) introduced by the emulation does not introduce additional behavioral errors in all the tested applications than the known errors that are bounded by the scale of a timeslice. We also observe that the errors caused by the native OpenVZ kernel is larger than the ones caused by the virtual time system, especially for TCP-based applications. Overall, this validation study suggests that our emulation/simulation testbed can deliver behaviors with temporal errors no greater than those induced simply by using OpenVZ virtualization.

ACKNOWLEDGMENTS

This material is based upon work supported by the Department of Energy under Award Number DE-OE0000097. (Disclaimer: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.)

REFERENCES

Apache 2012. "Apache: HTTP server project". http://httpd.apache.org/.

Biswas, P., C. Serban, A. Poylisher, J. Lee, S. Mau, R. Chadha, C. Chiang, R. Orlando, and K. Jakubowski. 2009. "An integrated testbed for virtual ad hoc networks". In *Proceedings of the 2009 TridentCom*.

Erazo, M., Y. Li, and J. Liu. 2009. "SVEET! a scalable virtualized evaluation environment for TCP". In *Proceedings of the 2009 TridentCom*.

Gupta, D., K. V. Vishwanath, and A. Vahdat. 2008. "DieCast: testing distributed systems with an accurate scale model". In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

Iperf 2012. http://iperf.sourceforge.net/.

Jin, D., and D. Nicol. 2010, December. "Fast simulation of background traffic through fair queueing networks". In *Proceedings of the 2010 Winter Simulation Conference*, edited by B. Johansson, S. Jain, J. Montoya-Torres, J. Hugan, and E. Yücesan, 2935–2946. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Jin, D., Y. Zheng, H. Zhu, D. Nicol, and L. Winterrowd. 2012. "Virtual Time Integration of Emulation and Parallel Simulation". PADS 2012, To appear.

Juels, A., and J. Brainard. 1999. "Client Puzzles: A Cryptographic Countermeasure Against Connection Depletion Attacks". In *NDSS*.

Lynx 2012. "Lynx: a text browser for the World Wide Web". http://lynx.browser.org/.

Nicol, D., D. Jin, and Y. Zheng. 2011, December. "S3F: The Scalable Simulation Framework Revisited". In *Proceedings of the 2011 Winter Simulation Conference*, edited by S. Jain, R. R. Creasey, J. Himmelspach, K. P. White, and M. Fu. Piscataway, New Jersey: Institute of Electrical and Electronics Engineers, Inc.

Nicol, D., and G. Yan. 2006. "High-performance simulation of low-resolution network flows". *Journal of Simulation* 82 (1): 21–42.

OpenVZ 2012. "OpenVZ Linux Containers". http://wiki.openvz.org.

QEMU 2012. "QEMU, Open Source Processor Emulator". http://wiki.qemu.org/.

UML 2012. "The User-Mode Linux Kernel". http://user-mode-linux.sourceforge.net.

Virtuozzo 2012. "Parallel Virtuozzo Containers". http://www.parallels.com/products/pvc46.

VMware. "VMware virtualization software". http://www.vmware.com.

Xen 2012. "Xen". http://www.xen.org/.

Zheng, Y., and D. Nicol. 2011. "A Virtual Time System for OpenVZ-Based Network Emulations". In *Proceedings of the 2011 Workshop on Principles of Advanced and Distributed Simulation (PADS)*.

Zheng, Y., D. Nicol, D. Jin, and T. Naoki. 2012. "A Virtual Time System for Virtualization-Based Network Emulations and Simulations". *Journal of Simulation*:To appear.

AUTHOR BIOGRAPHIES

YUHAO ZHENG is a Ph.D. student in the Department of Computer Science at the University of Illinois at Urbana-Champaign. His research interests lie in the areas of computer security, large-scale computer and communication system modeling and simulation. His email address is zheng7@illinois.edu.

DONG JIN is a Ph.D. student in the Department of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign. He holds a B.Eng. with first class honors in computer engineering from Nanyang Technological University, Singapore (2005), and a M.S. degree in electrical and computer engineering from the University of Illinois at Urbana-Champaign (2010). His research interests lie in the areas of cyber-security, modeling and simulation of large-scale systems and networks. His email address is dongjin2@illinois.edu.

DAVID M. NICOL is Professor of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign, and is Director of the Information Trust Institute. He holds a B.A. in mathematics from Carleton College (1979), and M.S. and Ph.D. degrees in computer science from the University of Virginia (1983,1985). Prior to joining UIUC, he taught at the College of William & Mary, and Dartmouth College. He has served in many roles in the simulation community (e.g. Editor-in-Chief of ACM TOMACS, General Chair of the Winter Simulation Conference Executive Board of the WSC), was elected Fellow of the IEEE and Fellow of the ACM for his work in discrete-event simulation, and was the inaugural recipient of the ACM SIGSIM Distinguished Contributions award. His current research interests include application of simulation methodologies to the study of security in computer and communication systems. His email address is dmnicol@illinois.edu.