J. Parallel Distrib. Comput. 104 (2017) 88–98

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

NEAT: Network link emulation with adaptive time dilation

Hee Won Lee [*], Mihail L. Sichitiu, David Thuente

North Carolina State University, Raleigh, NC 27695, United States

HIGHLIGHTS

- We present an approach for emulating networking links using adaptive time dilation.
- Our system accurately emulates delay/throughput while allowing for varying TDFs.
- Our system emulates distributed systems running different operating systems.

ARTICLE INFO

Article history:
Received 23 January 2015
Received in revised form
3 January 2017
Accepted 10 January 2017
Available online 16 January 2017

Keywords:
Network emulation
Time dilation
Virtual time
Distributed system
Virtual machines
KVM
QEMU

ABSTRACT

In evaluating the performance of highly complex networked systems, emulation is often used as it maintains much of the realism of testbeds, while offering increased flexibility and scalability. In large emulation systems, multiple and heterogeneous virtual machines can be deployed in relatively few general purpose physical hosts. Time dilation is a technique that allows virtual time to pass at a different (and potentially variable) rate with respect to real time, allowing for increased scalability of the emulated system. In this paper we present networking links in a large emulated system employing adaptive time dilation. The link emulation focuses on accurate delay and throughput emulation while allowing varying time dilation factors. To evaluate our system, we measure the delay and throughput of the virtual links under variable system loads.

1. Introduction

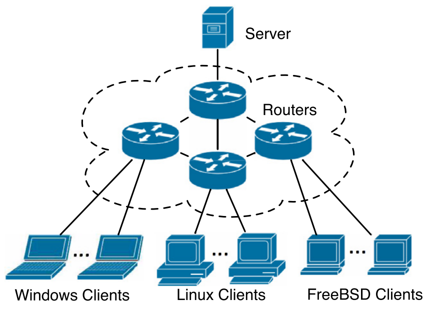
Network emulation is a technique that combines real elements of a deployed networked application such as end hosts and protocol implementations with synthetic, simulated, or abstracted elements such as network links, intermediate nodes and background traffic. When network emulation is comprised of exclusively emulated nodes and links, without connecting to any real elements, time passage can be dynamically controlled. *Time dilation* creates the illusion of improved hardware performance to the emulated nodes and links such that even a large-scale system can be built with relatively limited physical resources. Time dilation allows the passage of virtual time (i.e., time from the perspective of a virtual node) to proceed at a slower rate than real time by a specified factor, which is referred to as *time dilation factor* (TDF) [15]. When using the time dilation technique, DieCast [14] statically scales the data rates and delays of emulated links for achieving a target network performance. When a virtual machine is scaled by a factor TDF $= 10$, for example, DieCast changes a network link rate of 1 Gbps to 100 Mbps, and a network link delay of 100 μs to 1 ms.

When TDF statically increases to a greater extent than needed, we underutilize the physical resources, and thus unnecessarily increase emulation time. Our previous work [23] overcomes this conservative approach by dynamically adapting the TDF to system loads; we refer to this as *adaptive time dilation*. The adaptive time dilation allows virtual machines (VMs) to emulate higher performance than the actual performance of the physical machines. Our previous paper does not cover inter-VM's emulated network performance under varying virtual time. As TDF dynamically changes, it is not easy to maintain a constant network link performance between virtual hosts. If TDF increases, the data rate of a virtual link between VHs appears to increase, and the delay of the virtual link appears to decrease, and vice versa. Consequently, when packets pass through the virtual link, the network measurement result is not accurate.

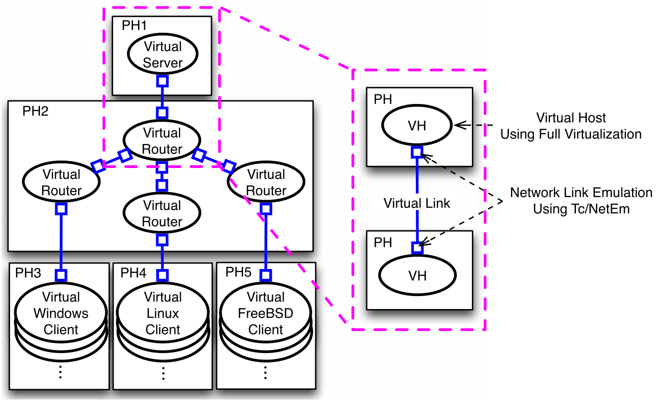
In this paper we propose an approach to controlling the value of the emulated link rates and delays such that the total end-to-end rates and delays between VHs are maintained near their target

* Corresponding author.
E-mail addresses: hlee17@ncsu.edu (H.W. Lee), mlsichit@ncsu.edu (M.L. Sichitiu), djthuen@ncsu.edu (D. Thuente).

http://dx.doi.org/10.1016/j.jpdc.2017.01.013
0743-7315/© 2017 Elsevier Inc. All rights reserved.



(a) Real-world topology.



(b) Emulation topology.

Fig. 1. Overview of proposed network emulation with the possible mapping of the virtual elements to physical hosts.

values despite changes in the TDF. We also show that for each target delay there exists a lower TDF bound that the TDF has to exceed for the target delay to be achievable. In particular, when emulating low latency – e.g., emulating InfiniBand having 5-µs latency with 1 Gb Ethernet having 500-µs latency between VMs – we have to operate the emulation system at or above the lower TDF bound.

The remainder of the paper is organized as follows. We first propose our approach for accurate network emulation under adaptive time dilation in Section 2, and then present our system implementation in Section 3. We evaluate our emulation system in Section 4, and discuss related works in Section 5. Finally, conclusions follow in Section 6.

2. Proposed approach

In this paper we propose an approach to emulate networks composed of fully-virtualized nodes and virtual links using adaptive time dilation. As the relative passage rate of virtual time ($=\frac{1}{TDF}$) changes in order to control system and network loads caused by the applications, the virtual links' emulated data rate and emulated delay should be scaled such that virtual hosts (VHs) perceive constant link characteristics. While many emulation approaches use lightweight virtualization for scalability [43,6,21], our system uses full virtualization for the VHs. Hence, the system can employ unmodified applications running on heterogeneous, unmodified OSs.

For our network emulation system, each network element in the desired topology is instantiated as a virtual element that runs on a PH. For scalability, multiple PHs can be used for mapping the virtual network elements to host machines.

Fig. 1(a) depicts a sample network with Windows, Linux, and FreeBSD clients connected to a server through four routers. Fig. 1(b) shows a possible mapping of the elements in the real-world

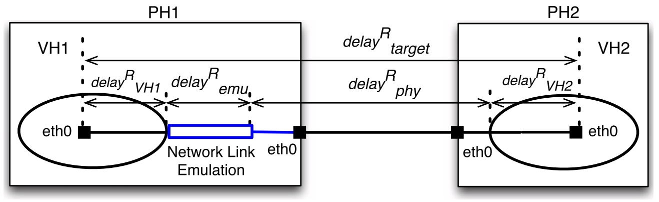


Fig. 2. Delay components of a virtual link.

topology into virtual elements emulated on five physical machines. The clients, server, and routers are emulated in our system using QEMU-KVM that supports full virtualization. For link emulation, we use Tc [25] and NetEm [16,28] to produce the effects of designed rates and delays on packets passing through a virtual link in our system.

We apply the adaptive time dilation mechanism [23] to control the PHs' system loads. The adaptive time dilation allows virtual elements to operate at a target emulation performance by preventing their PHs from being overloaded. Under the adaptive time dilation, all virtual elements' time passes in a synchronized manner, while the passage rate is dynamically changing in response to PHs' loads generated by VHs' traffic.

2.1. Virtual link design

When time dilation is applied to a virtual network, the network can appear faster or slower to the VHs depending on the TDF. In order to accurately emulate target delays independent of the changes in the TDF, the data rates and delays of emulated links have to change in concert with TDF.

For a virtual link, the virtual-time target data rate $rate^V_{target}$ can be controlled by the real-time emulated rate $rate^R_{emu}$, which is set by the Token Bucket Filter of Tc, such that:

$$rate^R_{emu} = \frac{rate^V_{target}}{TDF}. \qquad (1)$$

In addition, a virtual link's error rates due to packet duplication, corruption, reordering, etc. can also be emulated by setting them using the NetEm command options, because the error rate is represented as a percentage and is independent of TDF.

However, it is significantly more difficult to emulate the target delay of a virtual link. We target NEAT at emulating a delay time that a network packet would encounter when traveling from a VH's application to the other VH's via a virtual link.

Fig. 2 shows the components of a delay incurred when a packet travels from VH1 to VH2 through a virtual link. The virtual link's target delay, $delay^V_{target}$, is given by:

$$delay^V_{target} = \frac{delay^R_{target}}{TDF}, \qquad (2)$$

where $delay^R_{target}$ is the real-time target delay of a virtual link.

Since our system scales time but does not scale CPU cycles, the delay caused by the hypervisor and the guest OS's network protocol stack, $delay^R_{VHs}$ ($=delay^R_{VH1} + delay^R_{VH2}$), is practically independent of TDF. Physical delay $delay^R_{phy}$ is also independent of TDF. Hence, the $delay^R_{target}$ can be controlled by an emulated delay $delay^R_{emu}$, as illustrated in Fig. 2. In addition to controlling $delay^R_{emu}$ set by NetEm, the range of values of TDF also has to be limited for small target delays, $delay^V_{target}$.

To illustrate the need for a lower limit on TDF, we present two sample scenarios. For these examples, assume that $delay^R_{VH1} =$ 1 ms, $delay^R_{VH2} =$ 1 ms, and $delay^R_{phy} =$ 1 ms in Fig. 2.

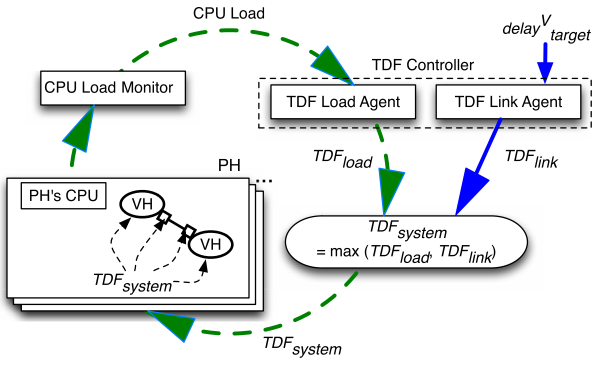


Fig. 3. TDF control mechanism.

- **Scenario 1**: Without time dilation, i.e., when TDF = 1, a target delay $delay^V_{target} = 3$ ms can be emulated by setting $delay^R_{emu}$ to 0 ms, because $delay^R_{VH1}$, $delay^R_{VH2}$, and $delay^R_{phy}$ already generate a total of 3 ms. Another target $delay^V_{target} = 5$ ms can be produced by setting $delay^R_{emu} = 2$ ms. However, a target delay $delay^V_{target} = 1$ ms is not possible, because setting $delay^R_{emu} = -2$ ms is physically impossible in the real world.
- **Scenario 2**: When the virtual time passes three times slower than real time, i.e., TDF = 3, it is possible to create $delay^V_{target} = 1$ ms by setting $delay^R_{emu} = 0$ ms, since $delay^R_{target} = 3$ ms in (2). Moreover, when TDF = 6, $delay^V_{target} = 1$ ms can be created by adding $delay^R_{emu} = 3$ ms, so that the total $delay^R_{target}$ becomes 6 ms.

In scenario 2, $delay^V_{target} = 1$ ms is feasible only when $TDF \geq 3$, since $TDF < 3$ requires a negative value of $delay^R_{emu}$. We refer to the minimum TDF required to emulate a virtual link's target delay as TDF_{link}. For the example in scenario 2, TDF_{link} is 3 for the corresponding target delay $delay^V_{target} = 1$ ms.

2.2. TDF control

NEAT aims at accurate emulation of virtual links towards their target delays even when TDF dynamically changes. When VHs' workloads load the PH, our system increases TDF, and vice versa. As TDF changes due to the PH's overload, NEAT accordingly controls the emulated rate and delay, $rate^R_{emu}$ and $delay^R_{emu}$, to maintain a target rate and delay respectively, as discussed in Section 2.1.

NEAT changes TDF to control PHs' loads using TDF_{load}; we refer to a TDF required to prevent VHs from overloading the PH, which is based on its PH's CPU loads, as TDF_{load}. In addition to TDF_{load} that we introduced in our previous work [23], NEAT controls TDF to satisfy the requirement of a minimum TDF (=TDF_{link}) for emulating the target delay of a virtual link.

Our TDF control mechanism is illustrated in Fig. 3. For each virtual link the TDF link agent first computes a new TDF_{link}, which is a minimum TDF required to create the target delay of that virtual link. For multiple virtual links, we take the largest value among each virtual link's TDF_{link} such that all virtual links are able to create their target delays. Also, when the CPU load monitor is periodically checking the PHs' CPU loads, the TDF load agent computes a new TDF_{load} based on the CPU loads.

The TDF controller then broadcasts the maximum of TDF_{link} and TDF_{load} to all the other PHs; we refer to the maximum of TDF_{link} and TDF_{load} as system TDF (or TDF_{system}). The maximum of TDF_{link} and TDF_{load} guarantees that all virtual links can create their target delay and no PHs are overloaded. NEAT synchronizes TDF_{system} in all VHs and virtual links distributed over multiple PHs.

2.3. TDF synchronization

In this section we summarize the synchronization method used by NEAT to ensure that all VHs use the same TDF throughout the emulation. The system is very similar to the one used in [23].

Each PH hosts a process called TDF controller whose main purpose is to determine what the minimum TDF is for the system; this value is called TDF_{local}. Each TDF controller on a PH is running two TDF agents for determining the minimum TDF required for that PH: a TDF load agent that determines the minimum TDF to avoid overloading the CPU system (called TDF_{load}) and a TDF link agent that determines the minimum TDF required to accurately emulate the link (called TDF_{link}). The TDF_{local} for a PH is the maximum of the TDF_{load} and TDF_{link}, essentially slowing down the emulator to avoid overwhelming the CPU as well as correctly emulating the links.

Synchronizing the TDF in multiple PHs is done by periodically broadcasting the locally computed value TDF_{local} to all the other PHs by using a dedicated LAN interface interconnected through a 1 Gbps switch. The broadcasting period is 10 ms, whose value is a compromise between timely synchronization and system overhead [23]. Upon receiving new TDF messages from different TDF controllers, the maximum of the locally determined TDF_{local} and the other values of TDF_{local} are used as the TDF_{system}; i.e., in essence the entire system is running as slow as the slowest PH. PHs are synchronized at the granularity of the synchronization message interval. VHs that run on the same PH are informed of the current TDF_{system} by using a shared memory (the TDF controller writes it, and all the VHs read it).

We use User Datagram Protocol (UDP) packets for synchronization messages. Synchronization delay may occur due to the UDP packet loss, and the synchronization however will be recovered at the subsequent synchronization message. Although Transmission Control Protocol (TCP) can be used instead, UDP is a better choice because TCP may introduce additional delay due to its congestion and flow control mechanism; if a packet is lost, then TCP will retransmit that packet and delay the subsequent packets. For our controller, the next UDP packet is not only a retransmission, but also has new information.

3. System implementation

We first present our system architecture for TDF control. We then present our virtual link implementation. We also describe how our system measures the physical link delay, which is a delay component of a virtual link. Finally, our system load control mechanism is briefly discussed.

3.1. System architecture

Fig. 4 depicts our system architecture for a sample virtual network with three VHs. The three VHs are deployed in two physical hosts (i.e., PH1 and PH4) as shown in Fig. 4(b); a larger virtual network may need to use more physical hosts. The VHs are connected through TAP interfaces and bridges. Each VH creates a TAP interface, which can be bridged to a real network interface to communicate with VHs running in other PHs.

When receiving a synchronization message, the TDF controller computes a new system TDF (i.e., TDF_{system}) and store it into the shared memory of each PH. The VHs then use this TDF value to control the progress of their virtual clocks. Based on the new system TDF, the TDF link agent updates the emulated rate and delay, $rate^R_{emu}$ and $delay^R_{emu}$, using Tc and NetEm in the TAP device of each virtual link.

VHs use a virtual network channel (using each PH's eth0) to communicate with each other. TDF synchronization messages use a physically-isolated control channel (using each PH's eth1). The isolated control channel assures stable TDF control operations by minimizing control-related side effects caused by heavy virtual network traffic.

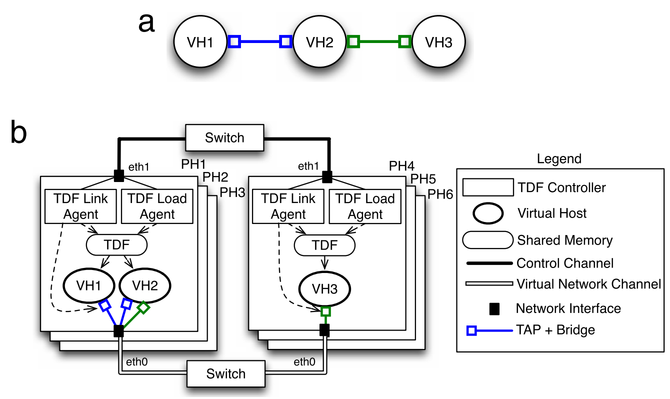


Fig. 4. A sample virtual network in (a) can be emulated by our system, as depicted in (b).

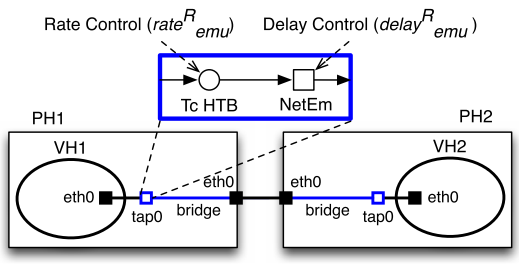

Rate Control ($rate^R_{emu}$)
Delay Control ($delay^R_{emu}$)
Tc HTB
NetEm
PH1
PH2
VH1
VH2
eth0
tap0
eth0
bridge
eth0
bridge
tap0
eth0

Fig. 5. Virtual link implementation.

3.2. Virtual link implementation

In order to emulate a link with a target data rate and delay between VHs, a virtual link is constructed as illustrated in Fig. 5. The virtual link connects VH1 and VH2, each running in a different PH. Each VH creates a TAP interface to communicate with the other VHs. Each TAP interface is connected to its physical interface through a bridge. The rate and delay controls are implemented in the TAP interfaces using Tc's Hierarchical Token Bucket (HTB) [17] and NetEm.

For the emulation of the target data rate of a virtual link, whenever the system TDF changes, a rate control, $rate^R_{emu}$ (in Fig. 5), is changed by using (1).

For the emulation of the target delay of a virtual link, we use (2). As illustrated in Fig. 2, the real-time target $delay^R_{target}$ can be obtained by:

$$delay^R_{target} = delay^R_{emu} + delay^R_{VH1} + delay^R_{VH2} + delay^R_{phy}, \tag{3}$$

where $delay^R_{VHi}$ is the delay caused by the hypervisor and the guest OS's protocol stack in VHi, and $delay^R_{phy}$ is the real-time physical link delay. Once we measure $delay^R_{VHi}$ and $delay^R_{phy}$ before running our system, the emulated delay $delay^R_{emu}$ can be computed by:

$$delay^R_{emu}(n) = delay^V_{target} \cdot TDF(n) - delay^R_{VH1} - delay^R_{VH2} - delay^R_{phy}. \tag{4}$$

Setting up a negative delay is physically impossible, i.e., $delay^R_{emu}(n) \geq 0$ in (4). Hence, a minimum TDF required for a target delay, TDF_{link}, is obtained from (4) by:

$$TDF_{link} = \frac{delay^R_{VH1} + delay^R_{VH2} + delay^R_{phy}}{delay^V_{target}}. \tag{5}$$

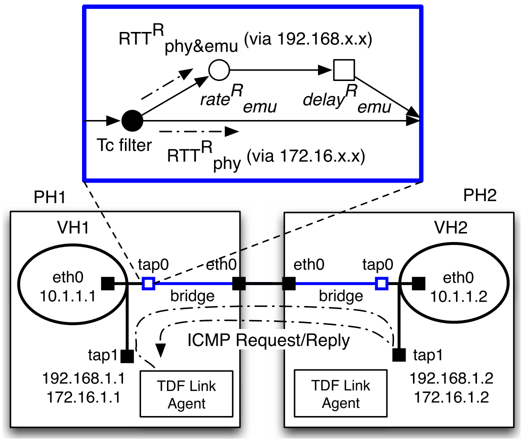


Fig. 6. Physical link delay measurement.

3.3. Physical link delay measurement

In order to emulate a target delay of a virtual link, NEAT uses the minimum TDF, TDF_{link} from (5), and the emulated delay, $delay^R_{emu}$ from (4), both of which need the physical link delay, $delay^R_{phy}$. For simplicity we can measure $delay^R_{phy}$ before starting emulations. However, for flexibility, NEAT periodically measures $delay^R_{phy}$ in the system.

Fig. 6 shows how NEAT measures physical link delay. The TDF link agent measures RTT^R_{phy} and $RTT^R_{phy\&emu}$. RTT^R_{phy} measurement packets use a 172.16.x.x path to eliminate the effect of link emulation, and $RTT^R_{phy\&emu}$ measurement packets use a 192.168.x.x path. Note that both RTT^R_{phy} and $RTT^R_{phy\&emu}$ exclude VHs' delay (i.e. $delay^R_{VH1}$ and $delay^R_{VH2}$) because they are measured by the TDF link agents running on PHs.

The physical link delay $delay^R_{phy}(n)$ can be obtained by measuring $RTT^R_{phy}(n)$ as:

$$delay^R_{phy}(n) = \frac{RTT^R_{phy}(n)}{2} \tag{6}$$

assuming that the delays comprising RTT^R_{phy} are symmetrical.

In order to minimize the variance of RTT^R_{phy}, we use an exponential moving average (EMA) for $RTT^R_{phy}(n)$ as:

$$RTT^R_{phy,EMA}(n) = (1-\alpha) \cdot RTT^R_{phy,EMA}(n-1) + \alpha \cdot RTT^R_{phy}(n), \tag{7}$$

where $0 \leq \alpha \leq 1$.

By using $RTT^R_{phy}(n)$ for a virtual link, NEAT can compute the minimum TDF TDF_{link} from (5) by:

$$TDF_{link} = \frac{delay^R_{VH1} + delay^R_{VH2} + \frac{RTT^R_{phy,EMA}(n)}{2}}{delay^V_{target}}. \tag{8}$$

A TDF $\geq TDF_{link}$ allows our system to emulate a virtual link's target delay $delay^V_{target}$.

In addition to TDF_{link}, if $delay^R_{phy}(n)$ changes, NEAT should adjust $delay^R_{emu}$ according to (4) in order to emulate the target delay $delay^V_{target}$.

The physical link delay $delay^R_{phy}(n)$ can also be obtained by:

$$delay^R_{phy}(n) = \frac{RTT^R_{phy\&emu}(n)}{2} - delay^R_{emu}(n), \tag{9}$$

where $RTT^R_{phy\&emu}(n)$ is measured by the TDF link agent, as illustrated in Fig. 6. Hence, a new emulated delay $delay^R_{emu}(n+1)$ can be obtained from (4) by:

$$delay^R_{emu}(n+1) = delay^V_{target} \cdot TDF(n) - delay^R_{VH1} - delay^R_{VH2} - \frac{RTT^R_{phy\&emu}(n)}{2} + delay^R_{emu}(n). \quad (10)$$

In terms of system architecture and its implementation, it is simpler to measure physical link delays before running the system. A system with the simpler architecture can emulate each target delay using (4) and (5). The simpler architecture is suitable for a physical configuration where the physical link delay is small and approximately invariant; for instance, all PHs running VHs are located in a LAN and there is no interference from other network traffic. The architecture of NEAT is more flexible because it measures physical link delays in the system. NEAT emulates each target delay using (8) and (10). The advantage of the flexible architecture is that target link delays can be emulated even if physical link delays incur large variance; for instance, we can create a virtual link that connects geographically-distributed VHs (e.g., one is in the West Coast and the other in the East Coast in the United States). The disadvantage is that the system is more complex.

3.4. System load control

In this section we summarize the system load control mechanism covered in our previous paper [23]. Our system load control mechanism seeks to maintain system loads at or below a target CPU load level ($Load_{target}$). When a PH's load increases above a target level, the TDF controller sends a larger TDF_{load} to each PH, thereby increasing the system TDF, such that virtual time passage rate ($=\frac{1}{TDF}$) decreases. As VHs proceed at a decreased rate in virtual time, VHs produce lighter loads to their PHs. Conversely, if a PH's load decreases below the target CPU load, the TDF controller decreases TDF_{load}.

4. Performance evaluation

In this section, we determine system parameters, and then evaluate emulated delay and throughput under scenarios with, and without traffic loads. We also evaluate how system load control allows for emulating a target throughput under TCP and UDP traffic scenarios. Finally, we evaluate our emulation system using real-world applications. NEAT emulates a heterogeneous system with distributed nodes running different operating systems, and runs a real-world application on it.

Our emulation system is built on general purpose servers (Dell PowerEdge R210). We use Ubuntu Linux (ubuntu-10.04-server-amd64) for PHs and VHs. In this setup, we run our modified version of QEMU-KVM (qemu-kvm-0.13.0) and our TDF agent [29].

4.1. System parameters

In this section, we describe how we determine the system parameters, i.e., the virtual host delay $delay^R_{VHs}$, the exponential moving average coefficient α in (7), and a TDF_{link} change threshold.

4.1.1. Virtual host delay $delay^R_{VHs}$

In Fig. 2, the delay introduced by VHs' hypervisor and network protocol stack, $delay^R_{VHs}$ ($=delay^R_{VH1} + delay^R_{VH2}$), is nearly constant for a given guest OS and PH, and independent of TDF, as this delay is measured in real time units. Note that the $delay^R_{VH1}$ and $delay^R_{VH2}$ are not affected by system overload because we prevent overload of the PHs by controlling system loads using TDF_{load}. After measuring $RTT^R_{VH\_VH}$, i.e., a real-time RTT from the guest OS of a VH to the other VH, and RTT^R_{phy}, i.e., a real-time RTT bypassing the rate and delay

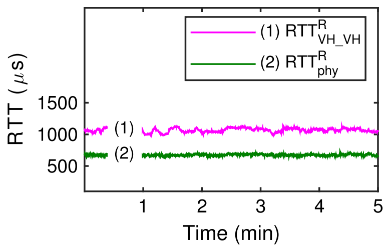


Fig. 7. Measurement of virtual host delay $delay^R_{VHs}$.

emulation elements, $rate^R_{emu}$ and $delay^R_{emu}$ in Fig. 6, the $delay^R_{VHs}$ can be computed by:

$$delay^R_{VHs} = \frac{RTT^R_{VH\_VH} - RTT^R_{phy}}{2}. \quad (11)$$

Experimentally, for our setup, the five-minute average of $delay^R_{VHs}$ computed by (11) is 197 µs, as shown in Fig. 7.

4.1.2. Exponential moving average coefficient α and TDF_{link} change threshold

In measuring physical link delay in Section 3.3, we introduced the parameter of α of EMA in (7) in order to reduce the measurement variance. While decreasing α from 1 to $\frac{1}{16}$, we compute the corresponding TDF_{link} by using (8) where NEAT measures physical link delay RTT^R_{phy} in the system and uses virtual host delay $delay^R_{VHs}$ ($=delay^R_{VH1} + delay^R_{VH2}$) of 197 µs that we measured before running the system as shown in Section 4.1.1. As α decreases from 1 to $\frac{1}{16}$, the computed TDF_{link} becomes smoother, as shown in Fig. 8(a), (b), and (c).

To further eliminate variability in TDF_{link}, we introduce a threshold, $\theta_{TDF_{link}}$; if the computed TDF_{link} does not change more than $\theta_{TDF_{link}}$, then the TDF_{link} change is not enforced (neither locally, nor communicated to the other PHs). As shown in Fig. 8(d) and (e), a change threshold of $\theta_{TDF_{link}} = 10\%$ completely removes the noise remaining after the EMA. Therefore, we use $\alpha = \frac{1}{16}$ and $\theta_{TDF_{link}} = 10\%$.

4.2. Minimum TDF

Our emulation system operates at TDF $\geq TDF_{link}$ to emulate a target delay $delay^V_{target}$ in each virtual link. In order to evaluate a minimum TDF, TDF_{link}, that is required to create the target delay of a virtual link, we temporarily disable the exchange of the TDF synchronization messages from the TDF controller. We then measure $RTT^V_{VH\_VH}$ (i.e., a virtual-time RTT measured from inside a VH to the other VH) while increasing TDF from 0.01 to 100 manually.

NEAT dynamically adjusts the emulated delay $delay^R_{emu}$ using (10), while measuring $RTT^R_{phy\&emu}$ in the system. For $delay^R_{VHs}$ ($=delay^R_{VH1} + delay^R_{VH2}$), NEAT use 197 µs obtained from Section 4.1.1.

Fig. 9(a) shows that for the topology in Fig. 6, i.e., two VHs and one virtual link, when the one-way target delay $delay^V_{target}$ is 500 µs, the measured values of $RTT^V_{VH\_VH}$ (i.e., a round-trip time from the guest OS of VH1 to VH2) are approximately 1000 µs when TDF $\gtrsim$ 1. The target delay can be successfully emulated as the emulated delay $delay^R_{emu}$ is appropriately adjusted when TDF $\gtrsim$ 1. Fig. 9(b) shows that for all TDF values larger than the corresponding TDF_{link} the target delays are successfully achieved. The minimum TDF for a virtual link, TDF_{link}, can be computed by (8). The corresponding TDF_{link} is approximately 0.25, 0.5, 1, 2, and 4 for $delay^V_{target}$ = 2000 µs, 1000 µs, 500 µs, 250 µs, and 125 µs respectively.

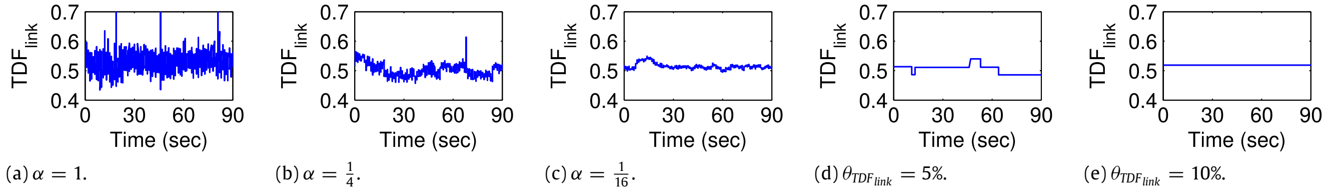


Fig. 8. When $\theta_{TDF_{link}} = 0\%$, TDF_{link} changes are reduced as α decreases from 1 to $\frac{1}{16}$ in (a), (b), and (c). When $\alpha = \frac{1}{16}$, as $\theta_{TDF_{link}}$ increases from 0 to 10%, TDF_{link} changes are further reduced in (d) and (e).

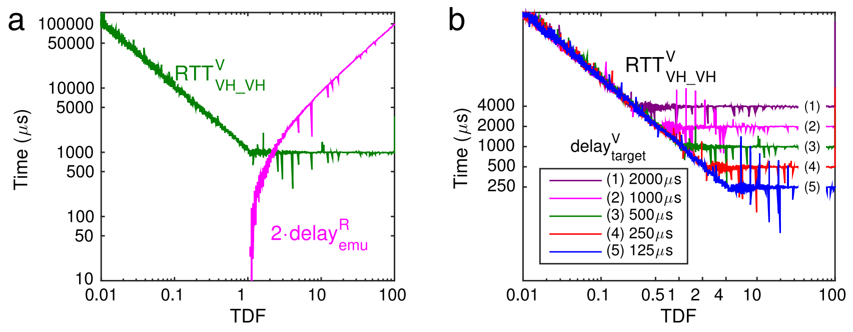


Fig. 9. (a) Emulated delay $delay^R_{emu}$ is appropriately adjusted to achieve the target RTT ($2 \cdot delay^V_{target} = 1000$ μs) when $TDF \gtrsim 1$. (b) Each $RTT^V_{VH\_VH}$ (measured from a VH) becomes the target RTT $= 2 \cdot delay^V_{target}$ when TDF $\geq TDF_{link}$.

4.3. Network link delay emulation

In this section we evaluate how well NEAT emulates network link delay by comparing their distributions for real-world and emulation situations. We test a local area network scenario with hundreds of microsecond delay, and a wide area network scenario with tens of millisecond delay.

4.3.1. Local area network scenario

For the comparison of real-world delays and emulated delays, we use a topology where two physical hosts (host 1 and host 2) are connected through 1-Gbps Ethernet in a local area network (LAN), as shown in Fig. 10(a). This real-world network topology is emulated as depicted in Fig. 10(b), where PH1 emulates VH1 for real-world host 1 and PH2 emulates VH2 for real-world host 2 respectively. The target data rate of a virtual link between VH1 and VH2 is set to 1 Gbps (corresponding to the real 1-Gbps Ethernet link).

The distribution of 1000 real round-trip delays between host 1 and host 2 is shown in Fig. 10(c). Since the average round-trip time is 235 μs, we use 118 μs ($=\frac{235}{2}$ μs) for the target delay of the virtual link between VH1 and VH2, $delay^V_{target}$, in Fig. 10(b).

When we do not control time by using TDF = 1 (i.e., no time dilation), the resulting emulated round-trip time has an average of 1091 μs, as shown in Fig. 10(d). Our TDF controller enables a target round-trip time (i.e., $2 \cdot delay^V_{target}$) to be accurately emulated by operating TDF at or above TDF_{link}. When running our emulation system at TDF $= TDF_{link}$ (4.1 in our experiment), the emulated round-trip delays are distributed, as shown in Fig. 10(e). The emulated round-trip time distribution of Fig. 10(e) closely resembles the real-world ping's round-trip time distribution shown in Fig. 10(c).

4.3.2. Wide area network scenario

We evaluate emulated delay in a wide area network (WAN) illustrated in Fig. 11(a). The real-world topology is emulated by our emulation system, as shown in Fig. 11(b). For a virtual link between VH1 and VH2, we use a target data rate $rate_{target}$ of 1 Gbps and a target delay $delay^V_{target}$ of 35.7 ms. When host 1 located in North

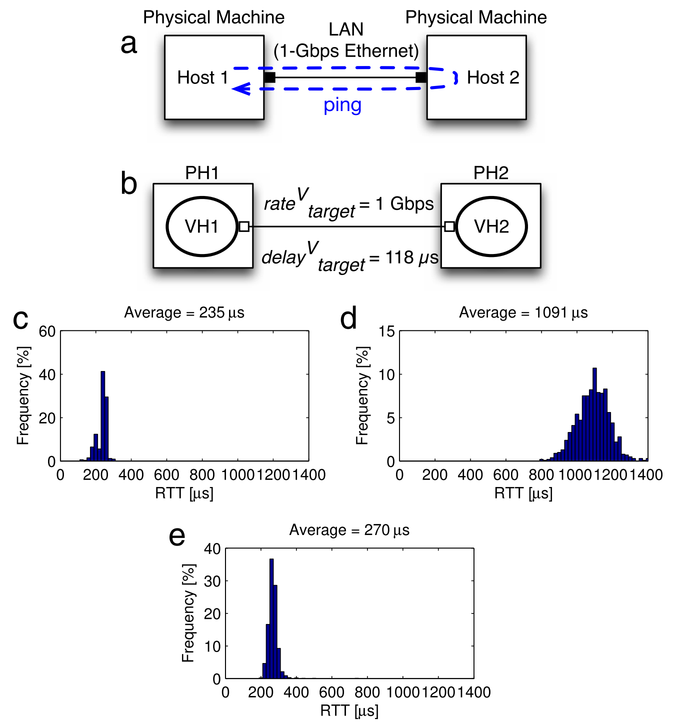


Fig. 10. Validation of delay emulation for a local area network. (a) Real-world topology; (b) Emulation topology with $delay^V_{target} = \frac{235}{2}$ μs; (c) Real round-trip time distribution for scenario (a); (d) Emulated round-trip time distribution without time dilation (TDF = 1); (e) Emulated round-trip time distribution with adaptive time dilation—the TDF controller maintains TDF at $TDF_{link} = 4.1$.

Carolina State University (NCSU) sends 1000 ICMP packets to host 2 located in University of California, Los Angeles (which is about 4111 km away from NCSU), the average round-trip time is 71.4 ms, as shown in Fig. 11(c). Hence, we set the target delay from VH1 to VH2, $delay^V_{target}$, to 35.7 ($=\frac{71.4}{2}$) ms.

With adaptive time dilation, the TDF controller operates TDF at $TDF_{link} = 0.014$ to emulate a target delay of 35.7 ms. Since virtual time passes 71 ($=\frac{1}{0.014}$) times faster than real time, the emulated

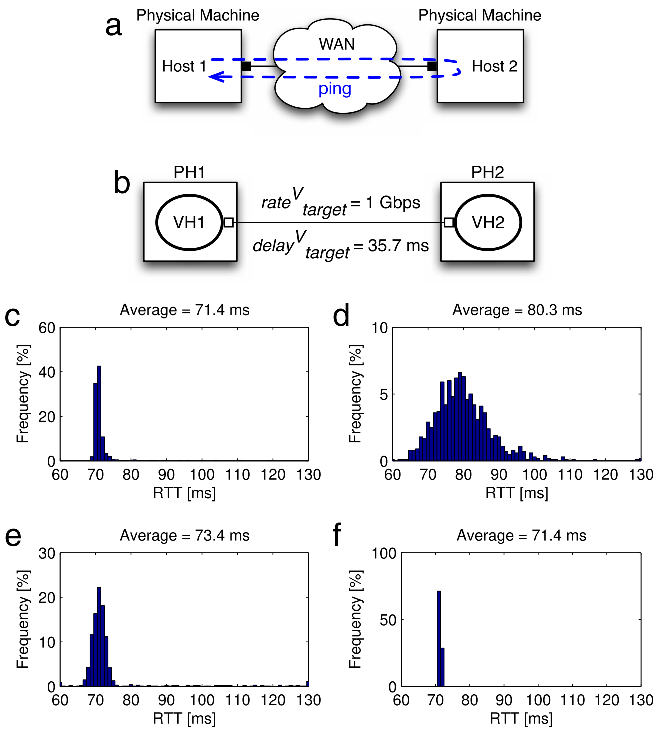


Fig. 11. Validation of delay emulation for a wide area network. (a) Real-world topology (b) Emulation topology with $delay^V_{target} = \frac{71.4}{2}$ ms; (c) Real round-trip time distribution; (d) Emulated round-trip time distribution with adaptive time dilation—the TDF controller maintains TDF at $TDF_{link} = 0.014$; (e) Emulated round-trip time distribution when TDF $= 0.1$; (f) Emulated round-trip time distribution when TDF $= 1$.

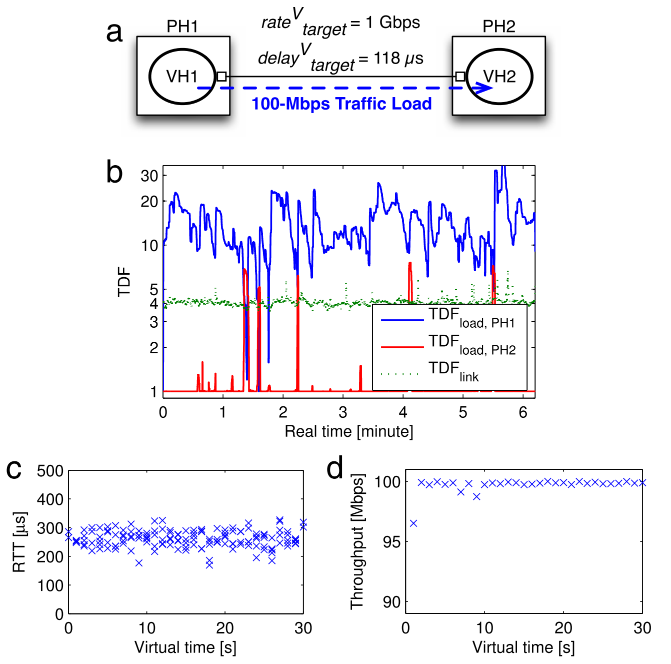


Fig. 12. Validation of delay and throughput emulation under a traffic load in a VH–VH scenario. (a) Emulation topology where 100-Mbps UDP traffic passes through a virtual link with $rate^V_{target} = 1$ Gbps and $delay^V_{target} = 118$ µs. (b) System TDF ($= \max(TDF_{load,PH1}, TDF_{load,PH2}, TDF_{link})$) dynamically changes. (c) Emulated round-trip time is close to 236 µs ($=2 \cdot delay^V_{target}$). (d) Emulated throughput is close to the offered load, 100 Mbps.

round-trip time measured in a VH has a relatively high variance, as shown in Fig. 11(d).

When we increase TDF to 0.1 (from $TDF_{link} = 0.014$ computed by the TDF controller), the variance of emulated round-trip time is reduced, as shown in Fig. 11(e). If TDF is increased even more, the variance of emulated delay is further reduced, as shown in Fig. 11(f).

This experiment shows that operating the emulation system at TDF < 1 results in excessive variance of the link delays. Therefore, we restrict the range of TDF values to be larger than one.

4.4. Virtual link under traffic loads

In this section, we evaluate the accuracy of emulated delay and throughput in scenarios where TDF dynamically changes due to traffic loads. As PHs are heavily loaded, the TDF load agent adaptively changes TDF to control the load. PHs' CPU loads are controlled to the target load 60%, at which VHs are able to generate traffic without packet losses.

4.4.1. Single link scenario

We test emulated delay and throughput for the topology depicted in Fig. 12(a). In this test, we consider the LAN scenario used in Section 4.3; for the virtual link between VH1 and VH2, the target data rate $rate^V_{target}$ and the target delay $delay^V_{target}$ are 1 Gbps and 118 µs respectively.

As VH1 sends to VH2 1000-byte UDP packets at 100 Mbps, their physical hosts (i.e. PH1 and PH2) are heavily loaded to process the packets. The TDF controller dynamically changes $TDF_{load,PHi}$ (i.e., a minimum TDF required to control PHi's loads offered by the VH traffic generation) depending on the traffic loads, as shown in Fig. 12(b). The minimum TDF required for a target delay 118 µs, TDF_{link}, is approximately 4.1. System TDF is obtained by taking the maximum of TDF_{link} and TDF_{load} for all PHs. Since $TDF_{load,PH1}$ is at all times the largest value among $TDF_{load,PH1}$, $TDF_{load,PH2}$, and TDF_{link}, the $TDF_{load,PH1}$ drives the system TDF.

While the system TDF dynamically changes over more than one order of magnitude as shown in Fig. 12(b), the emulated round-trip time is relatively constant and close to the target of 236 µs ($=2 \cdot delay^V_{target}$), as shown in Fig. 12(c). The emulated throughput is approximately the same as the offered load (i.e., 100 Mbps), as shown in Fig. 12(d). We observe that these emulated delays and throughput closely resemble real delays and throughput measured in a real-world scenario depicted in Fig. 10(a), where physical host 1 generates 100-Mbps UDP traffic towards physical host 2. Since the 1-Gbps network link is lightly utilized by the 100-Mbps traffic (i.e. 10% usage), the real round-trip time distribution is similar to the distribution without traffic shown in Fig. 10(c).

4.4.2. Multiple links scenario

We test emulated delay and throughput by mapping multiple VHs in a single PH, and running an application that generates network loads. We construct the emulation topology depicted in Fig. 13(a); the mapping on the physical hosts is shown in Fig. 13(b).

Every virtual host VHi running on PH1 is connected to virtual host VHi' running on PH2 through a virtual link with a target data rate 1 Gbps and a target delay 118 µs. A total of 8 VH–VH′ pairs are created. Each VH on PH1 runs an application which generates 1000-byte UDP packets towards their counterpart on PH2 at 3 Mbps.

As eight VHs on PH1 generate a total of 24-Mbps ($=3$ Mbps $\times$ 8 VHs) UDP traffic, the VHs heavily load PH1, thus increasing $TDF_{load,PH1}$ (i.e., a TDF required to control the PH1's load), as shown in Fig. 13(c). A TDF for PH2's load control, $TDF_{load,PH2}$, is one because PH2's CPU loads are always maintained below the target load 60% and we operate at TDF $\geq$ 1 for accurate emulation, as discussed in Section 4.3. The system TDF $= \max(TDF_{load,PH1}, TDF_{load,PH2}, TDF_{link})$ is most of emulation time driven by $TDF_{load,PH1}$, as seen in Fig. 13(c).

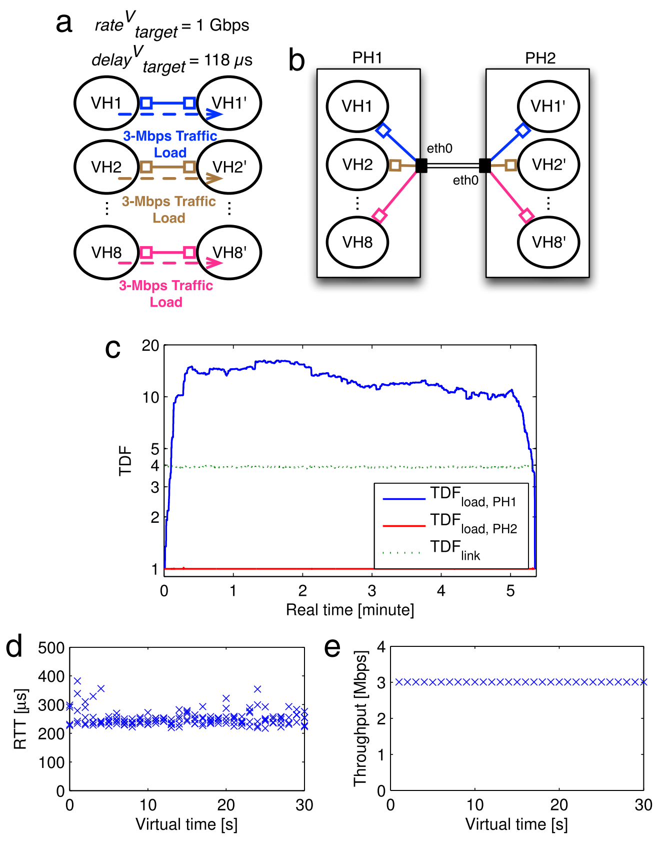


Fig. 13. Validation of delay and throughput emulation under a traffic load in a 8VHs–8VHs scenario. (a) Emulation topology where VH*i* generates 3-Mbps UDP traffic towards VH*i*′ through a virtual link with a target data rate 1 Gbps and a target delay 118 μs, and the total offered load is 3 Mbps * 8 VH–VH′ pairs = 24 Mbps. (b) Physical deployment of 8VHs-8VHs in two PHs. (c) $TDF_{load,PH1}$ primarily drives the system TDF, while changing dynamically. (d) Emulated round-trip time is close to 236 μs (=2 · $delay^V_{target}$). (e) Emulated throughput of a VH*i*-VH*i*′ pair is close to the offered load 3 Mbps.

While $TDF_{load,PH1}$ in the VH–VH scenario fluctuates mostly between 10 and 20 for 100-Mbps traffic, as seen in Fig. 12(b), $TDF_{load,PH1}$ in the 8VHs–8VHs scenario also changes between 10 and 20 for a total of only 24-Mbps traffic. Therefore, multiple VHs produce larger CPU loads for the same amount traffic.

Independent of TDF changes shown in Fig. 13(c), our system emulates a round-trip delay of ∼236 μs (=2 · $delay^V_{target}$), as shown in Fig. 13(d). In addition, Fig. 13(e) shows that our system emulates 3-Mbps throughput for 3-Mbps traffic loads per VH*i*–VH*i*′ insensitive to TDF.

4.5. High throughput and low latency

In this section we evaluate our emulation system in a high throughput and low latency scenario.

We emulate an InfiniBand-like network link of 10 Gbps and 5-μs latency using a physical link of 1-Gb Ethernet, as shown in Fig. 14(a). The 1-Gb physical link between VMs has approximately 500-μs latency without time dilation. VM1 generates TCP traffic using *Iperf* [18] towards VM2.

Fig. 14 shows the results. Before generating TCP traffic, TDF_{system} is maintained at approximately 100 by TDF_{link}, since we emulate 5-μs latency over a physical link with a latency of 500 μs. When TCP traffic is generated, TDF_{load} increases TDF_{system} to approximately 300, as shown in Fig. 14(b). The throughput reaches approximately 9.55 Gbps, which is reasonable for TCP over a 10-Gbps link. The round-trip time is maintained at approximately 10 μs (= 5 μs× 2-ways), as shown in Fig. 14(d).

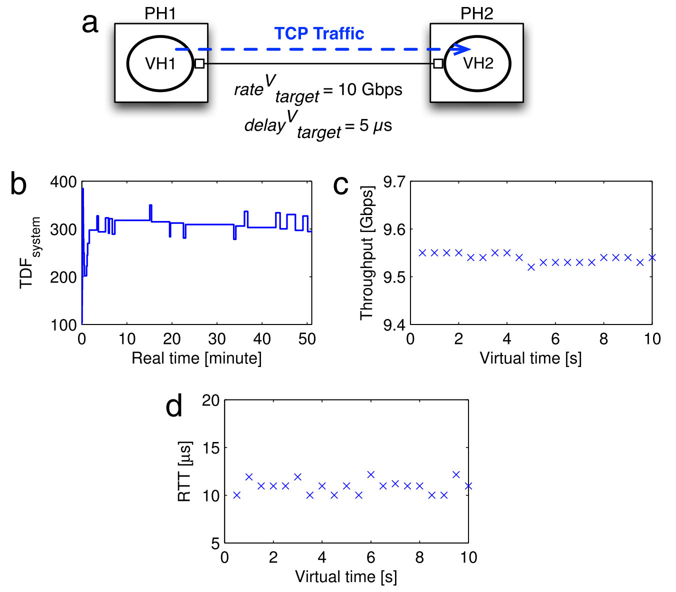


Fig. 14. When TCP traffic is generated on a virtual link that emulates 10 Gbps and 5 μs latency, the TDF increases approximately 300, as shown in (a) and (b). The increased system TDF allows for approximately 10-Gbps TCP throughput and 10-μs round-trip time, as shown in (c) and (d).

Table 1
File transfer time (virtual time).

Target rate (Mbps)	File size (MB)	File transfer time (s)
10	500	434.36
100	500	41.70
1000	500	5.68

4.6. Real-world application

In this section we evaluate our emulation system using two real-world applications. We first run the very secure FTP daemon *vsftpd* [38] on a virtual link connecting two virtual Linux boxes or devices. In addition, we run the VLC media player from VideoLan [35] on a virtual network with devices running on a variety of OSs including Linux, FreeBSD, and Junos.

4.6.1. vsftpd

In the virtual network topology shown in Fig. 15(a), *vsftpd* runs in PH1's virtual Linux server and the Linux default ftp client in PH2's virtual Linux client downloads a 500-MB file from the virtual Linux server. We perform three tests where a 500-MB file in the virtual Linux server is transferred to the virtual Linux client, while only changing the target rate of the virtual link from 10 Mbps to 100 Mbps, and then 1000 Mbps. As shown in 15(b), our system controller adaptively controls TDF depending on dynamic traffic loads caused by the different link data rates.

While the TDF is dynamically changing, the file transfer time in virtual time is only affected by virtual link's target rates, as shown in Table 1. When disregarding TCP/IP overhead and processing time, for a 500-MB (i.e. 4000-Mb) file the file transfer times should be 400, 40 and 4 s for virtual links of 10, 100 and 1000 Mbps respectively. Table 1 shows that our system correctly emulates each file transfer time.

4.6.2. VLC media player

We test a real application, the VLC media player, on a network configuration where a virtual FreeBSD server, three Junos routers, and a virtual Linux client run on three physical hosts, as shown in Fig. 16(a). The target data rate and delay of a virtual link is denoted by $\{rate^V_{target}, delay^V_{target}\}$.

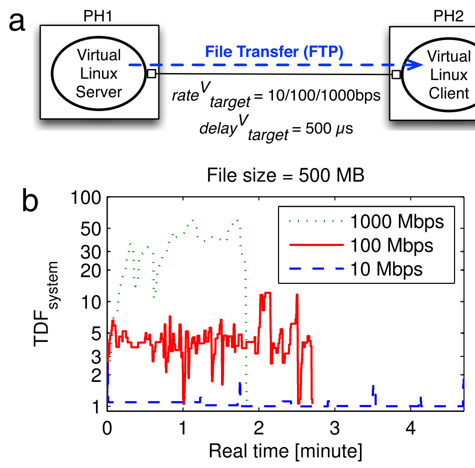


Fig. 15. When transferring a 500-MB file using *vsftpd* through a virtual link shown in (a), as the virtual link is configured for a higher target data rate, our controller increases the TDF accordingly, as shown in (b).

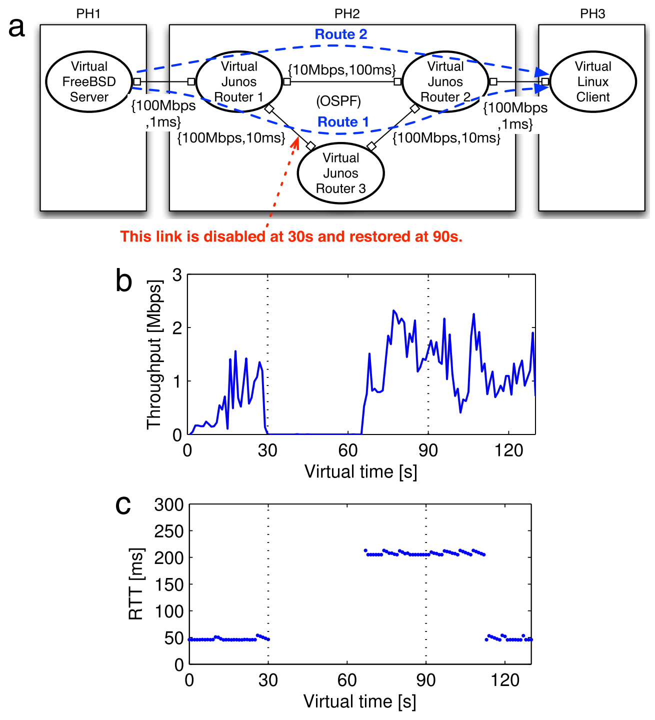


Fig. 16. (a) While a VLC media player on a virtual FreeBSD server generates UDP packet streams towards a virtual Linux client, we disable a virtual link between two virtual Junos routers at 30 s, and we restore the link at 90 s; three virtual routers running OSPF first route the VLC packet streams over Route 1, and then switches the route to Route 2 at the link's breakdown. (b) During the link's breakdown, the throughput becomes zero for 36 s and then recovers with a new route (i.e., Route 2). (c) The round-trip time between the virtual FreeBSD server and the virtual Linux client is around 46 ms over Route 1; during the link's breakdown, all packets are lost, and then the round-trip time becomes around 206 ms over Route 2; after the link's restoration, the packets are routed over Route 1 and the round-trip time gets back to about 46 ms.

A VLC media player operates as a streaming server on the virtual FreeBSD server running on PH1. The other VLC operates as a streaming client on the virtual Linux client running on PH3. The streaming packets pass through three Junos routers, which route the packets using the Open Shortest Path First (OSPF) routing protocol. We set each OSPF cost metric of the router interfaces to 1 for the 100-Mbps link and 10 for the 10-Mbps link.

In our experiment scenario, a VLC player on virtual FreeBSD server starts to generate UDP packet streams using a video file with 480p resolution and Xvid Codec; this video file generates 1-Mbps packet streams on average. As shown in Fig. 16(a), at 30 s in virtual time, we disable the virtual link between Virtual Junos Router 1 and Virtual Junos Router 3, and at 90 s we restore the virtual link. When the virtual link is disabled, the route of VLC's packet streams is switched from Route 1 to Route 2. When the virtual link is restored, the route is switched back from Route 2 to Route 1.

Fig. 16(b) shows the throughput results. As a VLC player starts to generate video streams from the virtual FreeBSD server, the throughput measured at virtual Linux client increases to around 1 Mbps. When a virtual link break down, the throughput decreases to zero. However, since the virtual routers regularly exchange link state update packets (i.e., link state advertisements), they find a new route, Route 2 in Fig. 16(a). Hence, the throughput is recovered at virtual second 66, as it takes 36 s for the OSPF protocol in the default configuration to update its state and calculate new routes.

Additionally, while the virtual link between the virtual Junos routers goes down at 30 s and is restored at 90 s, we continue to measure the ping's round-trip time between the virtual FreeBSD server and the virtual Linux client. Fig. 16(c) shows the result. Since ping packets are routed over Route 1 before the link's breakdown, the round-trip time measures approximately 46 ms, whose value can be calculated as $(1 + 10 + 10 + 1 \text{ ms}) \times 2$ (i.e., two-ways) + virtual elements' processing time. During the link's down time, all ping packets are lost. However, when the OSPF protocol finds a new path (i.e., Route 2), the round-trip time becomes approximately 206 ms as given by $(1 + 100 + 1 \text{ ms}) \times 2$ + virtual elements' processing time. When the broken link is restored at virtual second 90, the OSPF eventually switches the route from Route 2 to Route 1 because Route 1's routing cost of 4 $(1 + 1 + 1 + 1)$, is less than Route 2's cost of 12 $(1 + 10 + 1)$. Since link state updates require some time (23 s in our experiment), the route switch occurs at virtual second 113, resulting in the same 46-ms round-trip time.

In summary, we have shown that our system can emulate network links in a real-world application running on heterogeneous real-world OSs.

4.7. System overhead

Our system controls virtual links to emulate target network performance. Our system synchronizes the time passage rate in virtual hosts distributed over multiple physical hosts by exchanging two different types of synchronization messages, TDF_{link} and TDF_{load}. System loads caused by the synchronization messages should be minimized such that VHs can maximize the use of their PH's computing resources, but a larger synchronization interval degrades the responsiveness. Experimentally, when a synchronization interval decreases to less than 10 ms, the synchronization messages start to affect the overall system loads. To avoid this system overhead, our system uses a synchronization update period of 10 ms as a reasonable tradeoff between responsiveness to change and overhead.

5. Related work

Our approach provides a methodology for accurate network emulation as the time dilation factor dynamically changes for system load control. Hence, we first introduce prior work related to time dilation and then to dynamic time dilation. We also discuss prior work regarding node emulation and link emulation, which are used in our system. Finally, we introduce previous network emulators.

5.1. Time dilation

SliceTime [40] and DieCast [14] manage virtual time with different approaches. The mechanism used in SliceTime [40] is to alternately suspend and resume the entire system in order to connect VMs to discrete event simulations [22] that may lag behind in time under heavy system loads [39]. In contrast, DieCast [14] controls how fast virtual time proceeds by using a *time dilation factor* (TDF). If the TDF is greater than one, then virtual time passes slower than real time, so physical resources appear to virtual nodes to be TDF times faster [15]. Similarly, the Distributed Open Network Emulator (dONE) uses a temporal model referred to as *relativistic time* [3].

The approaches in [39,40,15,14] are static; i.e., the relative ratio between real and virtual time is fixed for the life of the VMs. NETplace [11] focuses on an initial placement of virtual nodes onto physical nodes using prior knowledge of the average load in order to minimize the network emulation runtime. NETbalance [12] extends NETplace for experiments with varying and unknown load by dynamically changing TDF using epoch-based virtual time [13,10]. NETbalance migrates virtual nodes during experiments to distribute the load and reduce the experiment runtime. NETplace and NETbalance create multiple virtual nodes in each VM for highly scalable emulation, whereas NEAT creates a virtual network using VMs, each of which is mapped onto a fully-virtualized node, focusing on the emulation of network link connecting the VMs.

5.2. Node emulation

There are several hypervisors that can create and run virtual machines (VMs), each running a separate OS instance. QEMU-KVM [31,20], Xen [41], and VirtualBox [36] are popular and free hypervisors, while VMware [37] is a commercial product.

Many academic research efforts have been based on Xen [41], but the QEMU-KVM hypervisor is recommended as the optimal choice for high performance computing environments [42]. Furthermore, QEMU-KVM runs each guest OS as a separate process in a PH, so established OS mechanisms such as shared memory and interprocess communications can be used to add new features to the hypervisor. QEMU-KVM can perform at near native speed with the KVM kernel component addition that provides a full virtualization solution. Hence, we choose QEMU-KVM to create *virtual hosts* (VHs) in our emulation system.

5.3. Link emulation

From the perspective of a host OS, when using QEMU-KVM for virtualization, each VH is a process, and the VHs communicate with one another via virtual links, which are implemented through inter-process communications [33] or using the Universal TUN/TAP device driver [34]. TUN/TAP provides packet reception and transmission for user space programs.

As an inter-process communication method, QEMU-KVM can connect multiple guest systems on a VLAN using TCP or UDP sockets. The VLAN here is a network switch running in the context of a QEMU process. Another method is to use TAP interfaces to provide full networking capability [32]. Virtual links between VHs are built such that TAP interfaces on VHs connect to each other through an Ethernet bridge [26] provided by the Linux kernel. Furthermore, an Ethernet bridge enables a TAP interface of the VH to connect to the PH's interface. This allows a VH to connect to another VH on a different PH. We use TAP interfaces and Ethernet bridges for virtual links in our emulation system.

A virtual link emulates the link's bandwidth, packet delays, and error rates. Dummynet [7] intercepts packets in the protocol stack, and passes them through one or more objects called queues and pipes that can be used to produce the effects of bandwidth limitation, propagation delay, bounded-size queue, packet losses, multipath, etc. EmuNET [19] has been designed to test protocols under a variety of conditions, such as bit rate limitations, network delay and jitter, different queuing schemes, etc. NetPath [1] is a scalable software-based link emulator that features fixed and probabilistic packet propagation delay emulation, probabilistic bit errors, probabilistic packet loss, packet duplication, and packet reordering capability. NetEm [28,16] emulates variable delays, packet losses, duplication and reordering. Tc [25] can classify traffic and limit bandwidth.

The approach in [24] constructs a virtual link between a VH and a simulator. There is no direct network link between VHs, and all network links are implemented through simulation models. In contrast, our current paper proposes a solution for emulating a direct network link between VHs.

5.4. Network emulators

Several large testbeds partially use or significantly depend on emulation techniques. PlanetLab [30] is a distributed overlay network for evaluating planetary-scale network services, and allows multiple services to run concurrently, each in its own *slice* [2,4]. Emulab [8] provides integrated access to emulated PC nodes, an 802.11 a/b/g testbed, and universal software defined radios (USRP devices), and can be expanded into PlanetLab testbeds for live Internet experimentation. GENI [9] provides researchers across the country with collaborative environments on which new network architectures and their implementations can be tested, while supporting scalable experimentation on shared and heterogeneous infrastructure. DETERlab [5] supports experimentation on next-generation cyber security technologies, and uses the Emulab cluster testbed software to control and manage a pool of PCs. ModelNet [27] emulates the delays, losses, and throughput of packets traveling between different application instances.

In contrast to large testbeds, Mininet is a lightweight emulator that runs real kernel, switch, and application code on a single machine and allows it to scale to hundreds of nodes [21].

6. Conclusion

In this paper we propose an approach for emulating network links in emulation systems with adaptive TDFs. Our emulation system uses QEMU-KVM that supports full virtualization to create virtual nodes, such that the system can emulate heterogeneous systems with distributed nodes running different OSs. The minimum TDF required for a delay emulation, TDF_{link}, allows our system to accurately emulate the target delays of virtual links. Our performance evaluation shows that NEAT consistently and accurately emulates the delay and throughput of the virtual links despite variable system loads.

References

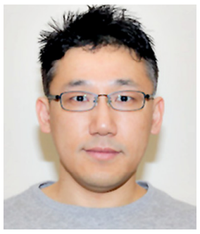
[1] S. Agarwal, J. Sommers, P. Barford, Scalable network path emulation, in: Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005. 13th IEEE International Symposium on, Sept. 2005, pp. 219–228.

[2] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, Mike Wawrzoniak, Operating system support for planetary-scale network services, in: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1, USENIX Association, Berkeley, CA, USA, 2004.

[3] C. Bergstrom, S. Varadarajan, G. Back, The distributed open network emulator: Using relativistic time for distributed scalable simulation, in: Principles of Advanced and Distributed Simulation, 2006. PADS 2006. 20th Workshop on, 2006, pp. 19–28.

[4] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, Mic Bowman, PlanetLab: an overlay testbed for broad-coverage services, SIGCOMM Comput. Commun. Rev. 33 (2003) 3–12.

[5] DeterLab. http://www.isi.deterlab.net.
[6] Vedavyas Duggirala, Srinidhi 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, IEEE Computer Society, Washington, DC, USA, 2012, pp. 101–110.
[7] Dummynet. http://info.iet.unipi.it/~luigi/dummynet.
[8] Emulab. http://www.emulab.net.
[9] GENI Project. http://www.geni.net.
[10] A. Grau, K. Herrmann, K. Rothermel, Efficient and scalable network emulation using adaptive virtual time, in: Computer Communications and Networks, 2009. ICCCN 2009. Proceedings of 18th Internatonal Conference on, Aug. 2009, pp. 1–6.
[11] A. Grau, K. Herrmann, K. Rothermel, NETplace: Efficient runtime minimization of network emulation experiments, in: Performance Evaluation of Computer and Telecommunication Systems, SPECTS, 2010 International Symposium on, July 2010, pp. 265–272.
[12] A. Grau, K. Herrmann, K. Rothermel, NETbalance: Reducing the runtime of network emulation using live migration, in: Computer Communications and Networks, ICCCN, 2011 Proceedings of 20th International Conference on, Aug. 2011, pp. 1–6.
[13] A. Grau, S. Maier, K. Herrmann, K. Rothermel, Time jails: A hybrid approach to scalable network emulation, in: Principles of Advanced and Distributed Simulation, 2008. PADS'08. 22nd Workshop on, June 2008, pp. 7–14.
[14] Diwaker Gupta, Kashi V. Vishwanath, Amin Vahdat, DieCast: Testing distributed systems with an accurate scale model, in: Proc. of NSDI, 2008, pp. 407–421.
[15] Diwaker Gupta, Kenneth Yocum, Marvin Mcnett, Alex C. Snoeren, Amin Vahdat, Geoffrey M. Voelker, To infinity and beyond: time warped network emulation, in: ACM Symposium on Operating Systems Principles, 2005.
[16] S. Hemminger, Network Emulation with NetEm, in: Linux Conf. Au, April 2005.
[17] Hierarchical Token Bucket. http://luxik.cdi.cz/~devik/qos/htb.
[18] Iperf. http://iperf.sourceforge.net.
[19] Ayman Kayssi, Ali El-Haj-Mahmoud, Emunet: a real-time network emulator, in: Proceedings of the 2004 ACM Symposium on Applied Computing, SAC'04, ACM, New York, NY, USA, 2004, pp. 357–362.
[20] KVM. http://www.linux-kvm.org.
[21] Bob Lantz, Brandon Heller, Nick McKeown, A network in a laptop: Rapid prototyping for software-defined networks, in: Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks, Hotnets-IX, ACM, New York, NY, USA, 2010, pp. 19:1–19:6.
[22] Averill M. Law, David M. Kelton, Simulation Modeling and Analysis, third ed., McGraw-Hill Higher Education, 1999.
[23] Hee Won Lee, Mihail L. Sichitiu, David Thuente, High-performance emulation of heterogeneous systems using adaptive time dilation, Int. J. High Perform. Comput. Appl. 29 (2) (2015) 166–183.
[24] Hee Won Lee, David Thuente, Mihail L. Sichitiu, Integrated simulation and emulation using adaptive time dilation, in: Proceedings of the 2Nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation, SIGSIM PADS'14, ACM, New York, NY, USA, 2014, pp. 167–178.
[25] Linux Advanced Routing and Traffic Control HOWTO. http://lartc.org/howto.
[26] Linux Ethernet Bridge. http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge.
[27] ModelNet. http://modelnet.ucsd.edu.
[28] NetEm. http://www.linuxfoundation.org/collaborate/workgroups/networking/netem.
[29] Network Emulation with Adaptive Time Dilation. http://www.ece.ncsu.edu/wireless/MadeInWALAN/AdaptiveTimeDilation.
[30] PlanetLab. http://www.planet-lab.org.
[31] QEMU. http://wiki.qemu.org.
[32] QEMU Networking. http://en.wikibooks.org/wiki/QEMU/Networking.
[33] W. Richard Stevens, UNIX Network Programming, Volume 2 (2nd ed.): Interprocess Communications, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.
[34] Universal TUN/TAP Device Driver. http://vtun.sourceforge.net/tun.
[35] VideoLan. http://videolan.org.
[36] VirtualBox. http://www.virtualbox.org.
[37] VMWare. http://www.vmware.com.
[38] vsftpd. https://security.appspot.com/vsftpd.html.
[39] Elias Weingärtner, Florian Schmidt, Tobias Heer, Klaus Wehrle, Synchronized network emulation: matching prototypes with complex simulations, SIGMETRICS Perform. Eval. Rev. 36 (2008) 58–63.
[40] Elias Weingärtner, Florian Schmidt, Hendrik Vom Lehn, Tobias Heer, Klaus Wehrle, SliceTime: a platform for scalable and accurate network emulation, in: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, NSDI'11, USENIX Association, Berkeley, CA, USA, 2011.
[41] Xen. http://www.xenproject.org.
[42] A.J. Younge, R. Henschel, J.T. Brown, G. von Laszewski, J. Qiu, G.C. Fox, Analysis of virtualization technologies for high performance computing environments, in: Cloud Computing, CLOUD, 2011 IEEE International Conference on, July 2011, pp. 9–16.
[43] Yuhao Zheng, D.M. Nicol, A virtual time system for openvz-based network emulations, in: Principles of Advanced and Distributed Simulation, PADS, 2011 IEEE Workshop on, 2011, pp. 1–10.



Hee Won Lee received a Ph.D. degree in Computer Science from North Carolina State University in May 2015. He received a B.E. in Electrical Engineering from Korea University in 2002, and a Master of Software Engineering from Carnegie Mellon University in 2005. During 2002–2009, he worked for KT Corporation as a Technical Member of Staff. He is currently employed as a Principal Member of Technical Staff at AT&T Labs Research. His primary research interest is in networking and storage systems.

Mihail L. Sichitiu was born in Bucharest, Romania. He received a B.E. and an M.S. in Electrical Engineering from the Polytechnic University of Bucharest in 1995 and 1996 respectively. In May 2001, he received a Ph.D. degree in Electrical Engineering from the University of Notre Dame. He is currently employed as a professor in the Department of Electrical and Computer Engineering at North Carolina State University. His primary research interest is in networking.

David Thuente received a Summa Cum Laude Honors B.S. degree in Mathematics from Loras College. He received his M.S. and Ph.D. degrees from the University of Kansas. He is currently Professor of Computer Science at North Carolina State University. He has done extensive consulting in sonobuoy signal processors and networking protocols and applications for Magnavox Electronic Systems Company. He has also done network consulting for Hughes Systems Company and Raytheon among others. He is a Professor Emeritus of Purdue University. His primary research area is networking.