Journal of Simulation (2016) © 2016 The Operational Research Society. All rights reserved. 1747-7778/16

www.palgrave.com/journals

A lightweight container-based virtual time system for software-defined network emulation

Jiaqi Yan and Dong Jin*

Illinois Institute of Technology, 10 W 31st Street, Room 226E Stuart Building, Chicago, IL 60616, USA

Container-based network emulation offers high fidelity and a scalable testing environment to bridge the gap between research ideas and real-world network applications. However, containers take their notions of time from the physical system clock, and thus the time-stamped events from different containers are multiplexed to reflect the scheduling serialization by the Linux operating system. Conjoining the emulator and other simulators is also challenging due to the difficulties of synchronizing the virtual simulation clock with the physical system clock. Virtual time systems for network emulation shed light on both issues. In this paper, we develop a lightweight container-based virtual time system in Linux Kernel. We use time dilation to trade time with system resources by precisely scaling the time of interactions between containers and physical devices. We develop a time freezer to enable the precise pause and resume of an emulation experiment, which offers the virtual time support to interface with simulators for close synchronization. We integrate the virtual time system into a software-defined networking emulator, Mininet, and evaluate the system accuracy, scalability, and overhead. Finally, we use the virtual-time-enabled emulation testbed to conduct a case study of equal-cost multi-path routing protocol analysis in a data center network.

Journal of Simulation (2016). doi:10.1057/s41273-016-0043-8

Keywords: network emulation; virtual time; software-defined networking; Linux kernel

1. Introduction

The advancement of an emerging network technology is highly dependent on the successful transformations of in-house research ideas into real system deployments. To enable such transformations, a testbed offering a scalable and high fidelity environment for evaluating network designs is extremely valuable. To analyze a networked computer system, researchers are often concerned with questions of scale. What is the impact of a system if the communication delay is X times longer, the bandwidth is Y times larger, or the processing speed is Z times faster? Simulation-based testbeds typically offer a cost-effective means to improve the testing scalability. However, losing fidelity is inevitable due to model abstraction and simplification. For example, large ISPs today prefer to evaluate the influence of planned changes of their internal networks through tests driven by realistic traffic traces rather than complex simulations. Network emulation technology is very attractive for such scenarios by executing real application software inside lightweight virtual machines (VMs) over controlled networking environments. Researchers have developed various network emulators with different virtualization technologies. Examples include DieCast (Gupta *et al*, 2011), TimeJails (Grau *et al*, 2008), VENICE (Liu *et al*, 2010), which are built using full or para-virtualization (such as Xen (The Xen Project, 2014)), as well as Mininet (Handigol *et al*, 2012) (Lantz *et al*, 2010), CORE (Common Open Research Emulator (CORE), 2016) ,and vEmulab (Hibler *et al*, 2008), using OS-level virtualization (such as OpenVZ (OpenVZ, 2014), Linux Container (Linux Containers, 2014), and FreeBSD Jails (Jails under FreeBSD 6, 2014)). All those network emulators offer functional fidelity through the direct execution of unmodified code. In this work, we focus on improving the Linux container technology for scalable network emulation, in particular in the context of software-defined networks (SDN). Our approach is to develop the notion of *virtual time* for containers to improve emulation temporal fidelity and scalability. A key insight is to trade time for system resources by precisely scaling the system's capacity to match behaviors of the target network. In this paper, we develop a time-dilation-based virtual time system in the Linux kernel, and integrate it to Mininet. The virtual time system also offers the capability to freeze and unfreeze the advancement of emulation, which facilitate further integration with other scalable simulation-based testbeds. The source code of our system is available to the research community on GitHub (Yan, 2016).

1.1. Time-dilation-based virtual time

Mininet (Lantz *et al*, 2010), like many other container-based network emulators, is hard to reproduce correct behaviors of a real network with large topology and high traffic load because of the limited physical resources. For example, "on a commodity machine with 2.98 GHz CPU and 4 GB RAM providing 3 Gbps internal bandwidth, Mininet is only capable to emulate a

*Correspondence: Dong Jin, Illinois Institute of Technology, 10 W 31st Street, Room 226E Stuart Building, Chicago, IL 60616, USA.
E-mail: dong.jin@iit.edu

network up to 30 hosts, each with a 100 MHz CPU and 100 MB RAM and connected by 100 Mbps links (Lantz *et al*, 2010)." It is because a host OS *serializes* the execution of multiple guest VMs, rather than in parallel like a physical testbed. Also, VMs take their notions of time from the host system's clock, and hence time-stamped events generated by the VMs are multiplexed to reflect the host's serialization, which badly harms temporal fidelity of the emulation result. To improve fidelity and scalability of the container-based network emulation, we take a time-dilation-based approach to build our virtual time system. The time dilation factor (TDF) is defined as the ratio between the rate at which wall-clock time has passed to the emulated host's perception of time passing rate (Gupta *et al*, 2005). A TDF of 10 means that for every ten seconds of real time, applications running in a time-dilated emulated host perceive the time advancement as one second. This way, a 100 Mbps link is scaled to a 1 Gbps link from the emulated host's viewpoint.

1.2. Network emulation time freezer

Combining lightweight emulation with scalable and flexible simulation environment is useful for high fidelity system evaluation in large-scale settings. A key challenge is the time synchronization of the two systems, which typically follow two different clock systems. One approach to use the two systems simultaneously is to use virtual time to coordinate the operations of the emulation and the simulation portions of the model. A key objective of our virtual time system is to offer the capability to conjoin container-based emulation and simulation with support for close synchronization. We design a time freezer module in the Linux kernel so that a running emulation experiment can now be paused and resumed, both in functionality and (virtual) time. This module allows tight management of virtual time transitions and event exchanges from emulation to simulation and back.

Our contributions in this paper are summarized as follows:

- We develop an independent and lightweight middleware in the Linux kernel to support virtual time for Linux container. Our system transparently provides the virtual time to unmodified application processes inside the containers, without affecting other processes outside the containers.
- To the best of our knowledge, we are the first to apply virtual time in the context of SDN emulation, and have built a prototype system integrated in Mininet, and have conducted an extensive evaluation on system accuracy, scalability, and usability.
- We develop a time freezer module for conjoining container-based emulation and simulation with support for close synchronization in the notion of virtual time.

In the remainder of the paper, Section 2 presents the related work. Section 3 elaborates design of the overall system and its two key components – dilation-based virtual time and time freezer. Section 4 describes the implementation details. Section 5 evaluates the system performance in terms of accuracy, scalability, and overhead. Section 6 demonstrates the virtual-time-enabled network emulation capability with a routing protocol analysis case study in a data center network. Section 7 concludes the paper with future work.

2. Related work

2.1. Software-defined network emulation and simulation

Researchers have developed a number of SDN simulation and emulation testbeds based on OpenFlow (McKeown *et al*, 2008). Examples include Mininet (Lantz *et al*, 2010), Mininet-HiFi (Handigol *et al*, 2012), EstiNet (Wang *et al*, 2013), ns-3 (Jurkiewicz, 2013), S3FNet (Jin and Nicol, 2013), fs-sdn (FS: a network flow record generator, 2013), and OpenNet (Chan *et al*, 2014). Mininet, today's most popular SDN emulator, uses the process-based virtualization technique to provide a lightweight and inexpensive testing platform. Mininet offers high functional fidelity by executing real SDN-based switch software, such as Open vSwitch (Open vSwitch, 2014), and controller software, such as POX (The POX Controller, 2013), Floodlight (Project Floodlight, 2016), RYU (Ryu SDN Framework, 2014), OpenDaylight (The OpenDaylight Platform, 2013). ns-3 supports models to simulate SDN-based networks (Jurkiewicz, 2013) as well as SDN controller emulation capability via the direct code execution (DCE) technique using TAPBridge devices. S3FNet supports scalable simulation/emulation of OpenFlow-based SDN through a hybrid platform that integrates a parallel network simulator and an OpenVZ-based network emulator (S3F/S3FNet, 2015). fs-sdn (FS: a network flow record generator, 2013) extends the SDN capability to fs (Sommers *et al*, 2011), a flow-level discrete-event network simulator.

2.2. Virtual time system

Researchers have explored various virtual time techniques to build more scalable and accurate virtual-machine-based network emulation. One direction is based on time dilation, a technique to uniformly scale the virtual machine's perception of time by a specified factor. It was first introduced in (Gupta, *et al*, 2005) and has been adopted to various types of virtualization techniques and integrated with a handful of network emulators. Examples include DieCast (Gupta *et al*, 2011), SVEET (Erazo *et al*, 2009), NETbalance (Grau *et al*, 2011), TimeJails (Grau *et al*, 2008), and TimeKeeper (Lamps *et al*, 2014, 2015). The other direction focuses on synchronized virtual time by modifying the hypervisor scheduling mechanism. Hybrid testing systems that integrate network emulation and simulation have adopted this approach. For example, S3F (Jin *et al*, 2012) integrates an OpenVZ-based virtual time system (OpenVZ Linux Container, 2014) with a parallel

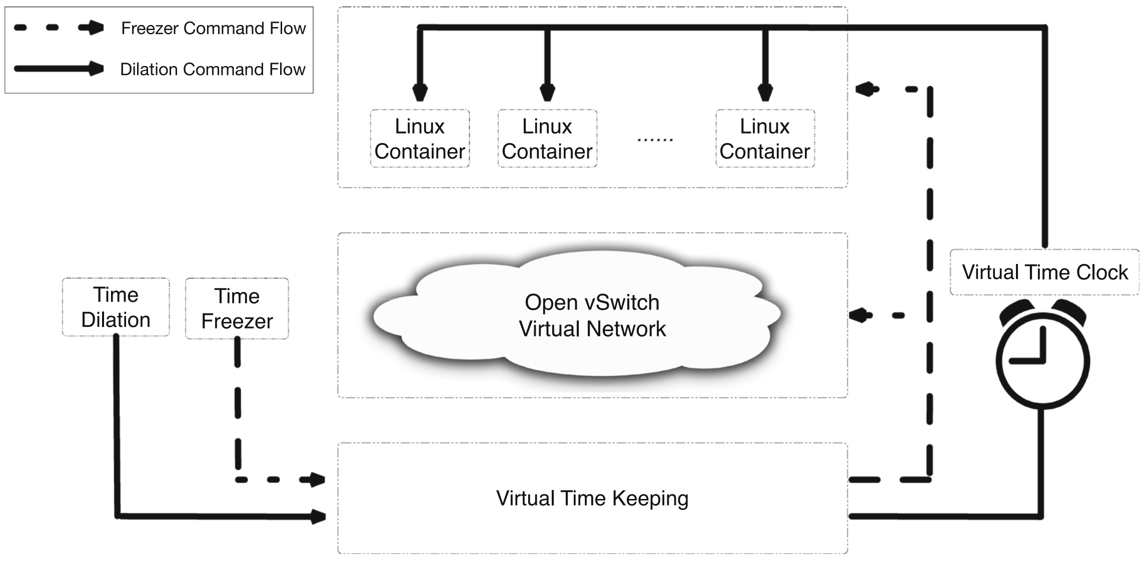

Figure 1 Virtual time system architecture for a container-based network emulator.

discrete-event network simulator by virtual timer. SliceTime (Weingartner *et al*, 2011) integrates ns-3 (Henderson *et al*, 2008) with Xen to build a scalable and accurate network testbed. Our approach is technically closest to TimeKeeper (Lamps *et al*, 2014, 2015) through direct kernel modifications of time-related system calls. The main differences are as follows: (1) Our virtual time system offers the capability of freeze/unfreeze emulation execution that facilitates the integration of simulation and emulation, and TimeKeeper uses a different approach. In TimeKeeper, the execution of a process is centrally controlled. TimeKeeper calculates the next lookahead and uses hrtimer to trigger the freezing and unfreezing of a process group. (2) Our virtual time system utilizes the *proc* virtual file system to expose a clean and unifying interface. (3) Our virtual time system covers a much wider range of time-related system calls. (4) We are the first to apply virtual time in the context of SDN emulation.

3. Container-oriented virtual time system design

Linux container (Linux Containers, 2014) is a lightweight virtualization technique that enables multiple instances of Linux OS sharing the kernel. It is a highly lightweight virtualization technique as compared with full/para-virtualization techniques, such as Xen, QEMU, or VMware, in which separated kernels are required for each VM. Therefore, Linux container has been applied in the area of scalable network emulation. For example, Mininet is a Linux-container-based emulation platform supporting SDN research.

Each container represents a virtual host with its own private network namespace and virtual interface. Applications are encapsulated in the containers as processes. The containers are connected by software switches with virtual interfaces and links, and they are multiplexed onto the physical machine. All the containers use the same system clock of the physical machine, and the execution of the containers is scheduled by the OS in serial. Unfortunately, this leads to the incorrect perception of time in a container, because a container's clock keeps advancing even if it is not running (e.g., in idle or suspended state). Such errors are particularly severe when emulating high-workload network experiments where the number of VMs exceeds the number of cores. In this work, we develop a virtual time system to improve the temporal fidelity. A capable virtual time system for scalable network emulation ought to have the following requirements:

- Lightweight design with low system overhead
- Transparent virtual time provision to applications in containers, i.e., no code required modification
- Universal virtual time support within the emulator and invisible to other processes on the same machine
- Ease of integration to the emulator as well as external simulators

We design a virtual time system to meet the aforementioned requirements. Figure 1 depicts the architecture of the virtual time system, as a lightweight middleware layer in the Linux kernel, for a Linux-container-based network emulator. We employ the time dilation technique to provide the illusion that a container has as much processing power, disk I/O, and network bandwidth as a real physical host in a production network despite the fact that it shares the underlying resources with other containers. The two main components are the time dilation manager and the time freezer.

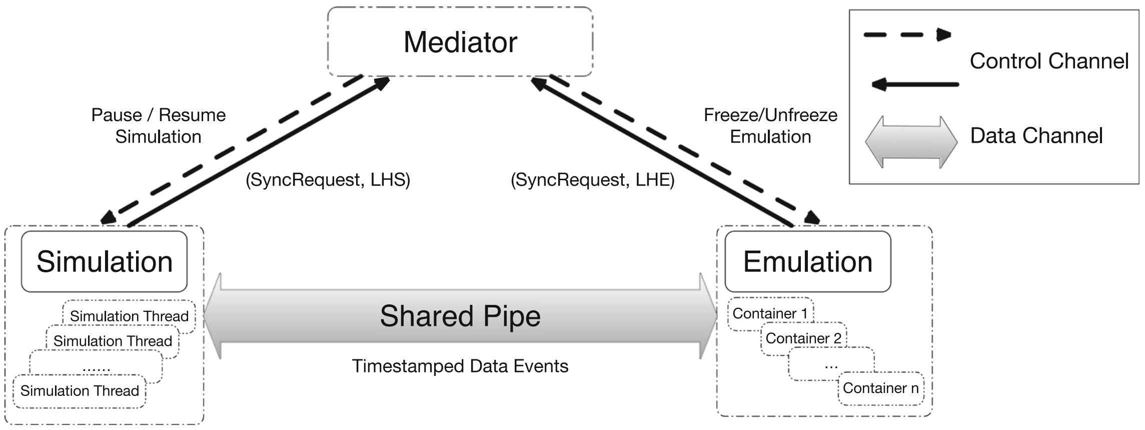

Figure 2 System design for coordinating simulation/emulation experiment advancement. *LHS* lookahead offered by simulation, *LHE* lookahead offered by emulation.

3.1. Time dilation for container-based network emulation

The time dilation manager is responsible for computing and maintaining the virtual time, according to a given TDF for all the containers. It can offer per-container virtual time or the global virtual time for the emulator. We have made a set of changes in the kernel, in particular, a modified data structure to present virtual time, and a new algorithm to convert the elapsed wall-clock time to the dilated virtual time, with no dependency on third-party libraries.

We attach an integer-valued TDF for every container. A TDF of *k* slows down a container's time advancement rate by a factor of *k*, and thus re-scales a container's notion of time with reference to a physical network. This way, Mininet can emulate a seemingly *k* times faster network owing to the accelerated rate of interaction between the containers and the virtual network. Note that our design cannot scale the capacity of hardware components such as main memory, processor caches, and disk, firmware on network interfaces. The integration with Mininet, and potentially other container-based software is straightforward. We provide a set of simple APIs to (1) initialize containers with virtual time, and (2) inquire virtual time at run time. The system returns precise virtual time to container processes and transparently to all their child processes while returning the ordinary system time to other processes.

3.2. Time freezer for simulation/emulation synchronization

Our virtual time system also facilitates the integration of network emulation and simulation, and the two systems can now be combined based on the notion of virtual time. This way, modelers do not need to worry about the causality errors introduced by a slower-than-real-time simulation experiment, which is often true for large-scale network scenarios. However, unlike simulation, to pause and resume a network emulator requires us to carefully modify the Linux kernel to achieve this capability. We develop a time freezer module for this purpose, and the details are discussed in Section 4.

Figure 2 shows how the time freezer is used to synchronize the emulation and simulation systems. The mediator is designed to coordinate the control events between the two systems, and their data communication is through a shared pipe. Both simulator and emulator can initiate a synchronization request with a lookahead value to the mediator. A lookahead offers a lower bound of the time that one system will affect the other system and its components in the future, such as simulation threads or emulation containers. The emulator will freeze all its containers for the duration of the offered lookahead upon receiving the freezing command from the mediator. Note that emulator itself does not stop running, since it may need to write time-stamped data events to the shared pipe. We have successfully combined the virtual-time-enable emulator with a power distribution simulator in a prior work (Hannon *et al*, 2016). With the mediator design, multiple simulators and potentially multiple emulators can be added into this framework in the future. Otherwise, the pairwise synchronization will end up with $O(N^2)$ control channels.

4. Implementation

4.1. The /proc filesystem interface

The virtual file system needs to provide an interface from the kernel to user-space applications. A straightforward way of exposing virtual time to user space is through modifying existing system calls and adding new system calls. However, we need to update the modified system calls across different Linux kernel versions. An extremely simple interface is also desired to encourage emulation developers to use the virtual time functionality. Therefore, we revisit the existing virtual time implementation and develop a new virtual time system using the */proc* file entry under the directory specific to each process. Our virtual time interface consists of two extra file entry under */proc/$pid*.

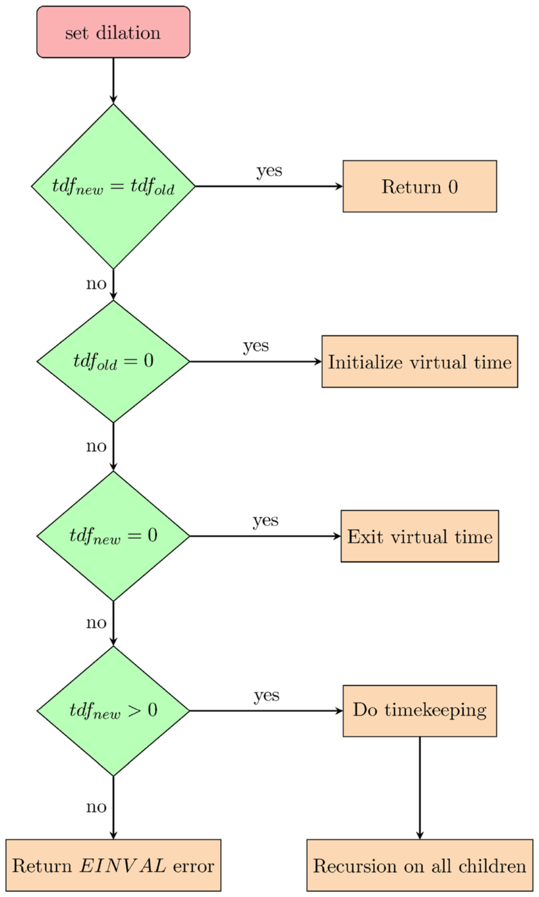

Figure 3 Set time dilation factor for container *tsk*.

- */proc/$pid/dilation* a non-negative integer value, to start and exit the virtual time mode for a process *$pid*, as well as to update the TDF value during the run time
- */proc/$pid/freeze* a Boolean value, to control to freeze or unfreeze a process *$pid*

Now manipulating a process's virtual time is as simple as executing the *echo* command to those file entries. Changing TDF from zero to a non-zero value will detach the process's clock from the system clock to enter the virtual time mode. Conversely, setting a process's TDF value to zero will make the process exit the virtual time mode.

4.2. Data structure of operating TDF

The following new fields are added into *task_struct*, so that a process can have its own perception of time.

- *dilation* represents the TDF of a time-dilated process.
- *virtual_start_ns* represents the starting time that a process detaches from the system clock and uses the virtual time, in nanoseconds.
- *physical_start_ns* represents the starting time that a process detaches from the system clock and uses the virtual time, in nanoseconds. It changes whenever a process's *dilation* changes.
- *virtual_past_ns* represents how much virtual time has elapsed since the last time the process requested the current time, in nanoseconds.
- *physical_past_ns* represents how much physical time has elapsed since the last time the process requested the current time, in nanoseconds.
- *freeze_start_ns* represents the starting moment that a process or a process group is frozen. It is always zero when the process is not frozen.
- *freeze_past_ns* represents the accumulative time, in nanoseconds, that a running process or a process group was put into the freezing state. A special process *leader* is responsible for maintaining and populating this variable for the members.

Setting a new TDF for a process has several possible cases as shown in Figure 3. In the case of "do timekeeping for switching TDF," Algorithm 1 describes how to handle the time calculation. Note that we need to repeat this process to all *tsk*'s child processes recursively so that the TDF is updated for the entire container (e.g., the process tree).

Algorithm 1 Do Timekeeping when Switching to a New TDF

```
function switch_tdf (tsk, new_tdf)
    now = undilated wall-clock time in nanoseconds
    delta_ppn ← now - tsk.physical_past_ns -
tsk.physical_start_ns - tsk.freeze_past_ns
    delta_vpn ← delta_physical_past_ns / old_tdf
    tsk.virtual_past_ns += delta_vpn
    tsk.physical_start_ns ← now - tsk.freeze_past_ns
    tsk.physical_past_ns ← 0
    tsk.dilation ← new_tdf
    return 0
end function
```

4.3. Signal-based time freezer

In this work, our time freezer is implemented in the container level, i.e., to freeze all the application processes inside a container. For example, when we freeze host h_1 (container), we should also stop the clock of *ping*, *iperf,* or other applications running inside h_1. We set process h_1 to be the leader of applications running in h_1, so that every process does not have to maintain its own frozen period and not to communicate with other processes in the same group.

The implementation of the time freezer is presented in Algorithm 2. After we *kill* a group of processes, we record the current time in order to calculate the group frozen duration for the next unfreezing command. When unfreezing a container, we send *SIGCONT* to all processes in the end. The reason is that if we resume the process group first, an unfrozen process may be scheduled to run and possibly query time before we populate the *freeze_past_ns* value to the entire container. In fact, the current per-container design of time freezer can be easily extended to the per-process level if the experiment scenario requires that level of granularity.

Algorithm 2 Freeze and Unfreeze a Container *tsk*

```
function populate_frozen_time(tsk)
  for each tsk's child do
    child.freeze_past_ns ← tsk.freeze_past_ns
    populate_frozen_time(child)
  end for
end function

function freeze(tsk)
  kill_pgrp(task_pgrp(tsk), SIGSTOP, 1)
  now ← undilated wall clock time in nanoseconds
  tsk.freeze_start_ns ← now
end function

function unfreeze(tsk)
  now ← undilated wall clock time in nanoseconds
  tsk.freeze_past_ns += now − tsk.freeze_start_ns
  tsk.freeze_start_ns ← 0
  populate_frozen_time(tsk)
  kill_pgrp(task_pgrp(tsk), SIGCONT, 1)
end function
```

4.4. Virtual time keeping

Virtual time keeping maintains the virtual time by considering both time dilation and time freezer as described in Algorithm 3. *do_virtual_time_keeping*() calls *update_physical_time*() to get the physical duration since last time we inquired the time excluding the frozen period, and then dilates the physical duration using *update_virtual_time*(). At the end of both methods, we complete the timekeeping by updating *physical_past_ns* and *virtual_past_ns*. In addition, operations like sending freezing commands or writing time dilation values to multiple containers are mutually independent. Therefore, we also provide two *pthread*-based utility programs, *freeze_all_procs* and *dilate_all_procs*, to enhance the parallelism for virtual time operations.

Algorithm 3 Virtual Time Keeping for Time Dilation and Time Freezer

```
function update_physical_time(tsk, ts)
  now ← convert ts to nanoseconds
  delta_ppn ← now − tsk.physical_past_ns − tsk.physical_start_ns
  tsk.physical_past_ns += delta_ppn
  return delta_ppn
end function

function update_virtual_time(delta_ppn, tdf)
  if tdf != 0 then
    delta_vpn ← delta_ppn / tdf
    tsk.virtual_past_ns += delta_vpn
  end if
end function

function do_virtual_time_keeping(tsk, ts)
  if tsk.dilation > 0 then
    delta_ppn ← update_physical_time(ts)
    update_virtual_time(delta_ppn, tsk.dilation)
    virtual_now ← tsk.virtual_start_ns + tsk.virtual_past_ns
    virtual_ts ← convert virtual_now to timespec struct
    return virtual_ts
  end if
end function
```

4.5. Link bandwidth dilation in traffic control

Network emulators like Mininet use the program *tc* (traffic control), a network quality-of-service control module in Linux (Almesberger, 1999) to control communication link properties. For instance, Mininet can use *tc* to rate-limit a link bandwidth to 100 Mbps using Hierarchic Token Bucket (HTB) *qdisc*. However, as a network module in Linux, *tc* does not reference to the Linux kernel time as many user-space applications do. Therefore, our virtual time system does not take effect on *tc*. For example, if the TDF is set to 8, the link bandwidth would be approximately 800 Mbps from the emulated hosts' view-points, as we observed from the *iperf* testing results. We tackled this problem by modifying the network scheduling code in the kernel to provide *tc* with a dilated traffic rate. To correctly emulate a 100 Mbps link, *tc* module should be configured to rate-limit the link bandwidth to 12.5 Mbps.

4.6. Bypassing virtual dynamic shared object

We discover that the 64-bit Linux kernel running on the Intel-based architecture uses the Virtual Dynamic Shared Object (vDSO) and the *vsyscall* mechanism to reduce the overhead of context switches caused by the frequently invoked system calls, including *gettimeofday*() and *time*() (Corbet, 2014). Therefore, applications may bypass our virtual-time-based system calls unless we explicitly use the *syscall* function. Our solution is to disable vDSO with respect to *__vdso_gettimeofday*() in order to transparently offer the virtual time to applications in the containers.

4.7. Limitation of virtual time system

The current implementation does not cover applications that read CPU cycles from the Time Stamp Counter (TSC) register. Since TSC can be directly read via the RDTSC assembly instruction, it is useful in a number of applications, such as measuring the exact time cost of certain specific instructions or operations. In practice, however, the usage of TSC reduces application portability, because this hardware-based feature is not available on every processor. On multiprocessor platforms, the TSCs on different processors may drift away from each other over time. Therefore, kernel developers in general discourage the usage of TSC (Molnar, 2006). Therefore, the impact of not considering applications using TSC is limited.

Another limitation of time dilation is that we cannot dilate interrupt-triggered events in the network stack. This is because interruptions are generated via timers. For example, consider a timer of duration d is set for a network event, such as a packet transmission or reception, at time t_{set}, before a dilated process enters virtual time at time $t_{vt\_enter} > t_{set}$. When a timer fires, either a software or a hardware interrupt event is triggered.

The correct dilated time for this event should be $t_{out\_dilated} = t_{vt\_enter} + \frac{(t_{current} - t_{vt\_enter})}{dilation\ factor}$. However, when the kernel sets up the timer, it already knows the interrupt event should occur at $t_{out\_induced} = t_{set} + d$. As a result, this kernel-inducted event timestamp is not dilated which could negatively affect application fidelity.

5. Evaluation

All the experiments are conducted on a Dell XPS 8700 Desktop with one Intel Core i7-4790 CPU, 12 GB RAM, and gigabit Ethernet interface. The machine runs a 64-bit Ubuntu 14.04.1 LTS with our customized 3.16.3 Linux kernel. Our virtual-time-enabled Mininet was built on Mininet (2.1.0), also named Mininet-Hifi, at the time of development.

5.1. Benchmarking virtual time system

5.1.1. Error of time dilation We exam the error of time dilation with different TDFs. The error is the difference between the elapsed time duration in virtual time and that in wall-clock time, and then multiplied by TDF. The duration spans from 200 microseconds to 10 s, i.e., {200 μs, 400 μs, 800 μs, 1 ms, 10 ms, 40 ms, 80 ms, 100 ms, 400 ms, 800 ms, 1 s, 4 s, 8 s, 10 s}. Figure 4 shows the average dilation errors in microseconds with standard deviations across 100 repetitions for TDF chosen from {1, 10, 20, 50} respectively. Note that the x-axis is in logarithm scale. We observe that the time dilation errors do not propagate. The error of time dilation does not increase as the duration of dilated time grows, up to 10 s. In addition, the increasing TDF does not affect the dilation error either. Comparing subfigures (a)–(d) in Figure 4 shows that as TDF increases from 1 to 50, the average errors are always bounded by 300 microseconds.

5.1.2. Overhead of signal-based time freezer We measured the overhead of our *pthread*-based implementation of the time freezer by repetitively freezing and unfreezing emulated hosts. We varied the number of hosts as {10, 50, 100, 250, 500} in Mininet. For each setting, we repeated the freezing and unfreezing operations for 1000 times, and computed the overhead as the duration from the moment the mediator issued a freezing/unfreezing operation to the moment that all hosts were actually frozen/unfrozen. We added the overhead of freezing operation and the overhead of the associated unfreezing operation, and plotted the CDF of the emulation overhead in Figure 5.

We observe that more than 90% of the operations take less than 100 ms in the 500-host case. For all other cases, more than 80% of the operations consume less than 50 ms. We also observe that the average overhead time grows linearly as the number of hosts increases, as shown in Table 1. The observed standard deviation of the overhead time is due to the uncertainty of delivering and handling the pending *SIGSTOP* and *SIGCONT* signals.

5.2. Evaluating virtual-time-enabled Mininet

We evaluate how our virtual time system improves Mininet's fidelity and scalability through a basic network scenario: a single TCP flow transmission through a chain of switches in an emulated SDN network. As shown in Figure 6, the network topology consists of a client–server pair connected by a chain of Open vSwitch switches in Mininet. We setup the default OpenFlow controller to function as a learning switch.

5.2.1. Fidelity evaluation In this set of experiments, we connected two hosts through 40 switches in Mininet, and all the links are configured with 10 microseconds delay. We used *iperf* (iperf3, 2014) to generate a TCP flow between the client and the server. TDF was set to 1 (i.e., no virtual time) and 4 for comparison. We also setup a real testbed for "ground truth" throughput data collection. The testbed was composed of two machines connected by a 10 Gbps Ethernet link. We varied the bandwidth link from 100 Mbps to 10 Gbps and measured the throughput using *iperf*. In the real testbed, we manually configured the link bandwidth and delay using *tc*, and the delay was set as the corresponding round-trip times (RTTs) measured in the switch chain topology in Mininet, so that the networking settings were tightly coupled for comparison. Although we did not setup an exact network with SDN switches, the stabilized TCP throughputs generated by the physical testbed should reflect what occurs in a real SDN network. Each experiment was repeated 10 times and the results with bandwidths 4 Gbps, 8 Gbps, and 10 Gbps are reported in Figure 7.

We observe that when the bandwidth was no greater than 4 Gbps (we only displayed the 4 Gbps case in the figure), Mininet was able to accurately emulate the TCP flow with and without virtual time, as the average throughputs were very close to the ones collected from the physical testbed. However, when we continued to increase the bandwidth, Mininet was not able to produce the desired throughputs, e.g., 28% (8 Gbps) and 39% (10 Gbps) smaller than the physical testbed throughputs.

With virtual time (TDF = 4), Mininet was capable to accurately emulate the TCP flow even at high bandwidth, and the results were nearly the same as the ones measured in the physical testbed. Note that we only emulated a single flow, and the results could be much worse and unpredictable in complicated multi-flow scenarios. Results show that virtual time can significantly enhance the fidelity by "slowing down" the experiments so that the system has sufficient resources and time to correctly process the packets. We further illustrate the effect by plotting the time series of throughput changes for the 10 Gbps cases in Figure 8. With virtual time, the throughputs measured in Mininet closely match the real testbed results; without virtual time (TDF = 1), the ramp up speed was much slower, in particular, 22 s (TDF = 1) rather than 4 s (TDF = 4), and the throughput was incorrectly stabilized below 6.2 Gbps.

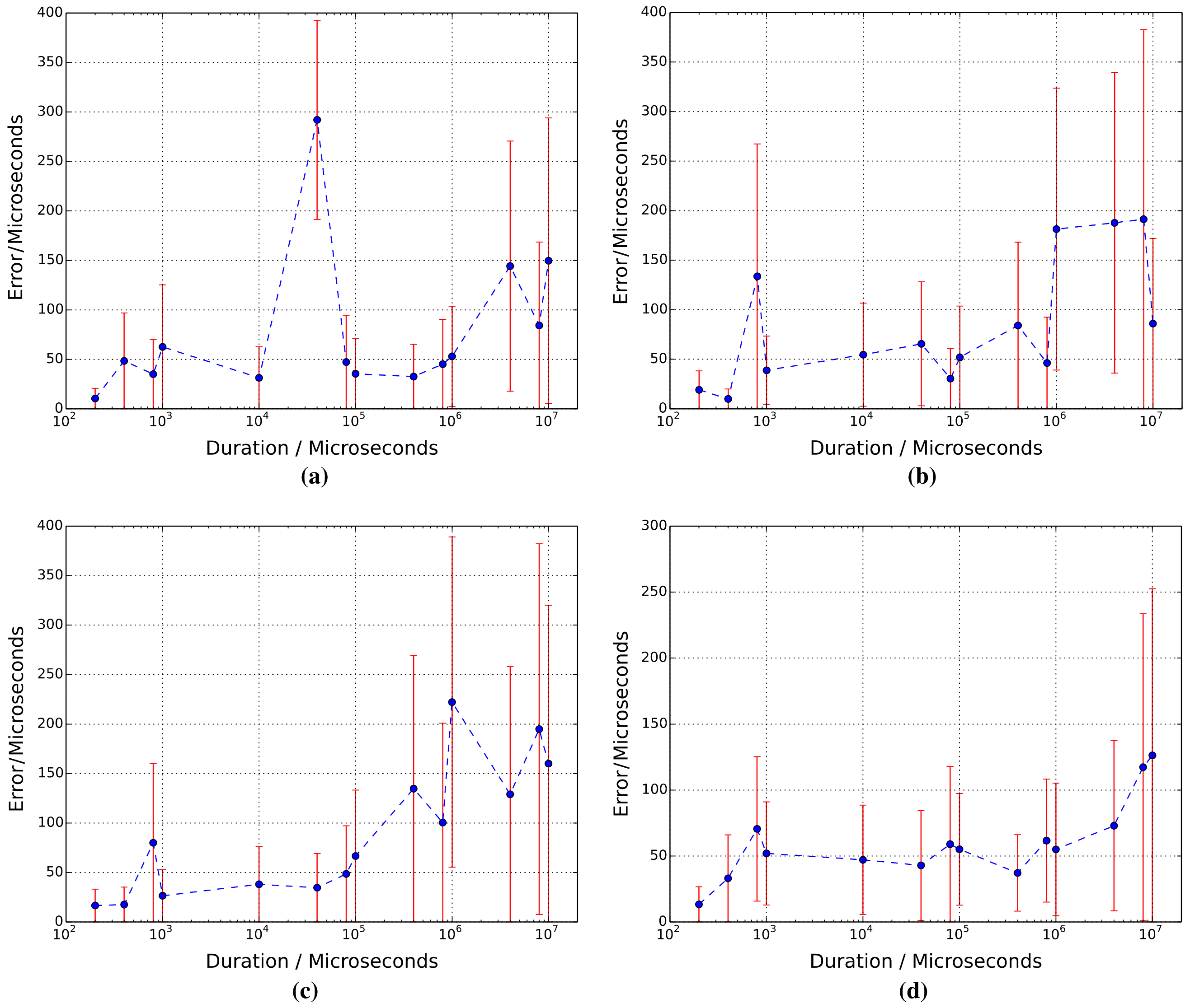

Figure 4 Time dilation error for different TDFs and test durations. **a** Dilation Error when TDF = 1, **b** Dilation Error when TDF = 10, **c** Dilation Error when TDF = 20, **d** Dilation Error when TDF = 50.

5.2.2. Scalability evaluation Virtual time also improves the scale of networks that one can emulate without losing fidelity. In this set of experiments, we used the same switch chain topology in Figure 6, and set the link bandwidth to 4 Gbps. We want to investigate, with virtual time, how many switches Mininet is capable to emulate without losing fidelity, i.e., preserving nearly 4 Gbps throughput. This time we increased the number of switches with the following values {20; 40; 60; 80; 100}, and TDF was selected to be 1 (no virtual time) and 4. We ran *iperf* for 25 s between the client and the server. Each experiment was repeated 10 times, and the throughput measurement is reported in Figure 9.

In the case of TDF = 1, the average throughput kept decreasing as the number of switches grew over 20. The throughput decreased dramatically when the number of switches was greater than 60 (e.g., decreased by 41% for 60 switches, 65% for 80 switches, and 83% for 100 switches). The standard deviation, indicated the undesired high disturbance, also grew as number of switches increased. When virtual time was applied with TDF = 4, the throughput was always around 3.8 Gbps with small standard derivations in all the experiments. It is clear that virtual time helps to scale up the emulation. In this case, Mininet can emulate 100 switches with 4 Gbps links and still generate the accurate throughputs, rather than being saturated at 20 switches without virtual time. We also recorded the running time in Figure 10. Longer execution time is the tradeoff for the fidelity and scalability improvement. When TDF = 4, the execution time was about 3.7 times longer than the time required in the case of TDF = 1 in all the experiments.

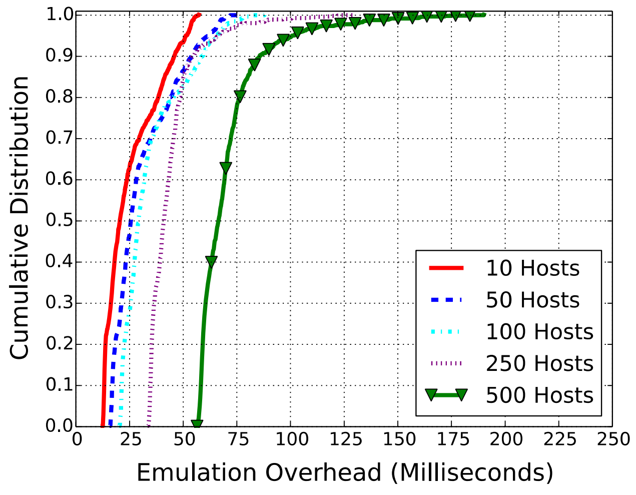

Figure 5 CDF of freeze/unfreeze overhead.

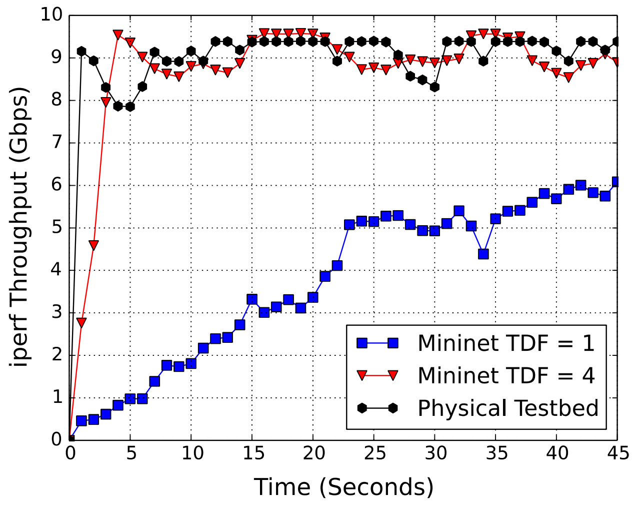

Figure 8 TCP throughput with 10 Gbps links.

Table 1 Statistics of freeze/unfreeze implementation's overhead

# Hosts	*10*	*50*	*100*	*250*	*500*
Avg. overhead	25.244	30.812	34.418	43.858	70.516
Std. overhead	12.476	14.259	14.339	11.951	16.840

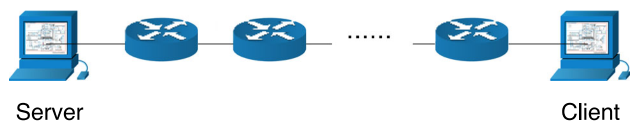

Figure 6 A switch chain network topology for fidelity and scalability evaluation.

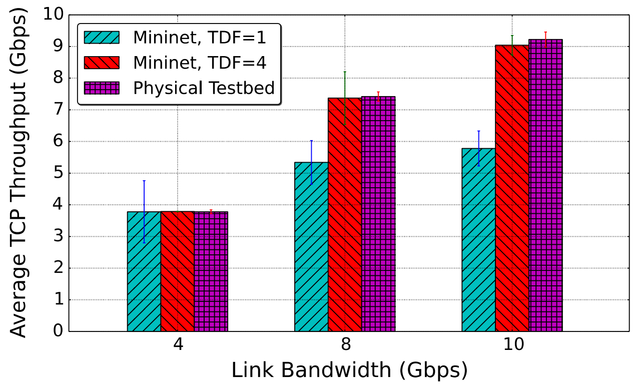

Figure 7 TCP throughput with different link bandwidths.

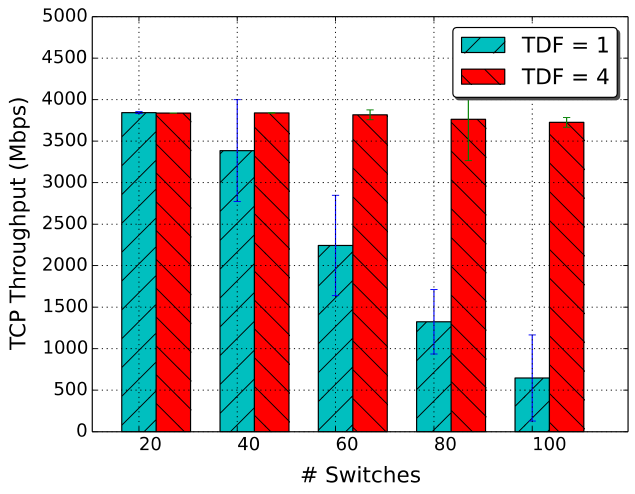

Figure 9 TCP throughput under different number of switches.

5.3. Evaluating time freezer

In this section, we evaluate how the timer freezer in our virtual time system affects network experiments. In particular, we focus on two metrics, i.e., end-to-end throughput and latency. We created two emulated hosts connected via an Open vSwitch in Mininet. The links are set to 1 Gbps bandwidth with 10 microseconds latency. *iperf* is used to measure the throughput, and *ping* is used to measure the round-trip time (RTT) between the two hosts.

5.3.1. End-to-end flow throughput We transferred data over a TCP connection for 30 s for throughput testing. In the first run, we advanced the experiments without freezing the hosts. In the second run, we froze the emulation for one second, and repeated the operation every one second. Data are collected across 64 identical runs. We coupled the two experimental results and reported the average throughputs between the 11th second and the 30th second in Figure 11. The error bars represent the standard deviations of the measured throughputs. We observed that the average throughputs of the "interrupted" emulation match well with the baseline results. However, pausing emulation introduces around 8–14% deviation. Several

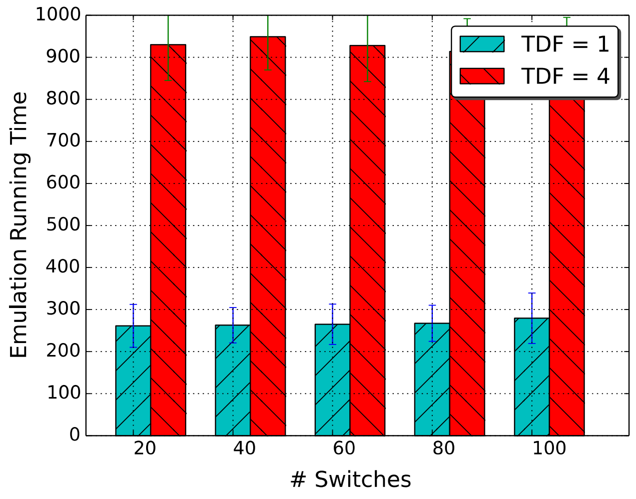

Figure 10 Emulation running time in scalability evaluation experiments.

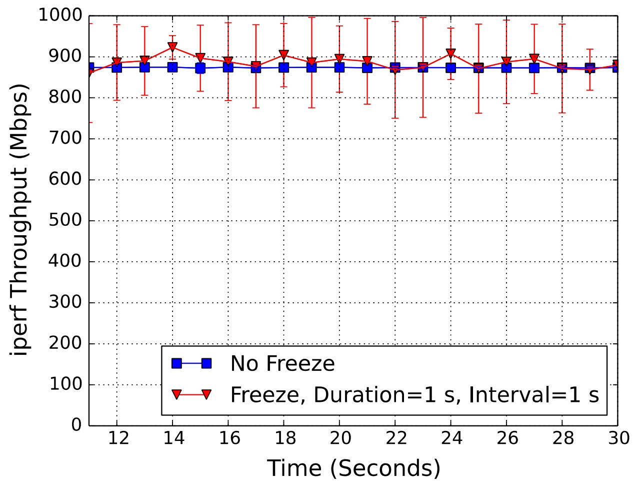

Figure 11 TCP throughput comparison, 1 Gbps bandwidth and 20 microseconds latency.

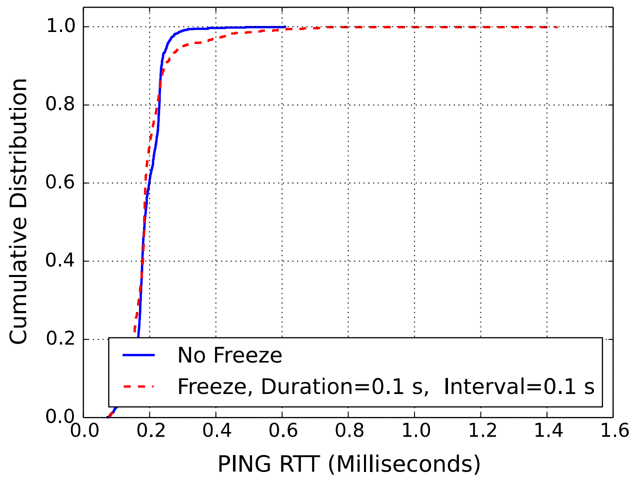

Figure 12 Ping round-trip-time comparison, 1 Gbps bandwidth and 10 microseconds latency.

sources lead to this deviation. First, while we explicitly generate *SIGSTOP* and *SIGCONT* signals to the containers, those signals are only in the pending state. The actual deliveries depend on the OS scheduler, since signal deliveries usually occur when exiting from the interrupt handling. Second, the actual freezing duration depends on the accuracy of the sleep system call. Sleeping for one second has a derivation about 5.027 ms on the testing machine.

5.3.2. End-to-end flow latency To evaluate the end-to-end flow latency, we issued 1000 *ping*s with and without freezing the emulator. We skipped the first ping packet in the results to exclude the effect of ARP and the switch rule installation from the SDN controller. Figure 12 plots the CDF of the RTT for both sets of the *ping* experiment. We observed that the two lines are well matched in the case of 10 microseconds link delay, and pausing the emulator does not affect the distribution of RTT. About 90% of ping packets are received before 0.5 ms, which is most likely dominated by the processing delay instead of the link latency.

6. Case study: evaluation of ECMP routing protocol in data center networks

Network emulation testbeds are widely used to test and evaluate designs of SDN network applications and controllers with the goal of discovering design limitations and implementation faults before the real system deployment. In the context of SDN data center network, we present a case study to demonstrate how our virtual-time-enabled Mininet has been utilized to reproduce and validate the limitations of the equal-cost multi-path (ECMP) routing strategy.

Many modern data center networks employ multi-rooted hierarchical tree topologies, and therefore SDN controllers running ECMP-based routing mechanism (Thaler and Hopps, 2000) are used in data center networks for load-balancing traffic over multiple paths. When an SDN switch has multiple next-hops on the best paths to a single destination, it is programmed by the ECMP controller to select the forwarding path by performing a modulo-N hash over the selected fields of a packet's header to ensure per-flow consistency. The key limitation of ECMP is that the communication bottleneck would occur when several large and long-lived flows collide on their hash and are being forwarded to the same output port (Al-Fares *et al*, 2010). We borrowed the experiment scenario on a physical testbed described in (Al-Fares *et al*, 2010), and created a set of emulation experiments in Mininet to demonstrate the limitation of ECMP. We built a fat-tree topology in Mininet as shown in Figure 13, and generated stride-style traffic patterns, in which *stride*(i) means a host with index x sends to the host with index $(x + i)$ mod n, the number of hosts. The hash-based ECMP mechanism is provided by the

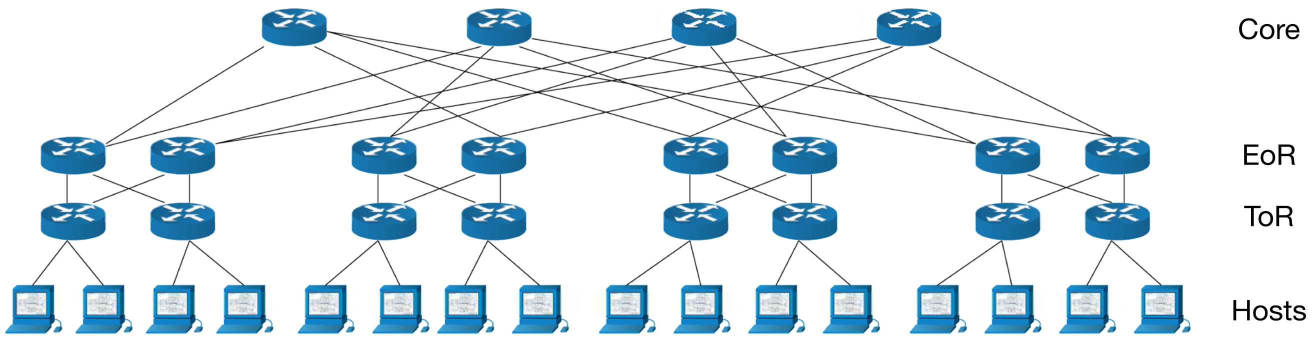

Figure 13 A data center network with a degree-4 fat-tree topology.

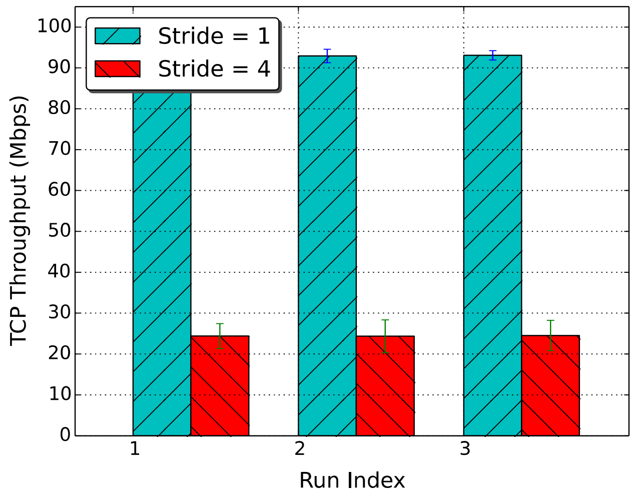

Figure 14 ECMP limitation in a fat-tree data center network with 100 Mbps link bandwidth.

RipL-POX SDN controller (Heller, 2014). The Mininet code was developed with reference to (Reproducing network research, 2014). In all the following experiments, we set up eight sender-receiver pairs transmitting stride-pattern traffic flows using step 1 and 4.

We first set all the link bandwidth (switch-to-switch and switch-to-host) to 100 Mbps, and conducted each experiment over three independent runs. The average throughput of eight TCP flows was plotted in Figure 14. The limitation of ECMP presented in (Al-Fares *et al*, 2010) was clearly observed. When many conflicting flows occurred with stride-4 flow patterns, the average throughput in the fat-tree network dramatically fell below 30 Mbps with up to 75% throughput drop.

However, the link bandwidth configurations in the previous experiments are not realistic. As early as in 2009, links connecting edge hosts to top of rack switches (ToR), ToR to edge of rank switches (EoR), and EoR to Core switches in a data center had been already above gigabit, in particular, 10 Gbps switch-to-switch links and 1 Gbps host-to-switch links (Vahdat, 2009). Can Mininet still show us the limitation of ECMP with such high link bandwidth? If not, can virtual time help to overcome the issue? Using the same configurations except that links were set to 10 Gbps, we re-ran the experiments in Mininet without virtual time (TDF = 1) and with virtual time (TDF = 4). We plotted the average flow throughput in Figure 15 and individual flow throughput in Figure 16.

In the case of stride-1, there were very few collisions among flows. Hence, the network performance ought to be close to the ideal throughput, i.e., 160 Gbps bisection bandwidth and 10 Gbps average bandwidth between each pair. In the experiments that TDF = 4, the average throughput is above 9.0 Gbps, which is close to the theoretical value, and also match well with the results obtained from the physical testbed built upon 37 machines (Al-Fares *et al*, 2010). In the experiments that TDF = 1, however, the throughput barely reaches 3.8 Gbps because of the limited system resources that Mininet can utilize. In addition, as shown in Figure 15, we observe that the variation of throughput is large among flows when TDF = 1. This is incorrect because no flow shares the same link in the case of stride-1. In contrast, when TDF = 4, the throughput of all 8 flows are close with little variation, which implies the desired networking behaviors.

In the case of stride-4, flows may collide both on the upstream and the downstream paths, thus using ECMP could undergo a significant throughput drop, e.g., up to 61% as experimentally evaluated in (Al-Fares *et al*, 2010). The virtual-time-enabled Mininet (TDF = 4) has successfully captured such throughput drop phenomenon. We can see that average throughput dropped about 80% when RipL-Pox controller used ECMP to handle multi-path flows. Large deviation (more than 55% of average throughput value) also indicates that the flows were not properly scheduled with ECMP. When TDF = 1 (no virtual time), Mini*net also* reported plummeted TCP throughput in the case of stride-4. However, we cannot use the result to experimentally demonstrate the limitation of ECMP. It is hard to distinguish whether the throughput drop was caused by insufficient resources to handle 10 Gbps or the limitation of ECMP, given the fact that the throughput was already too low in the

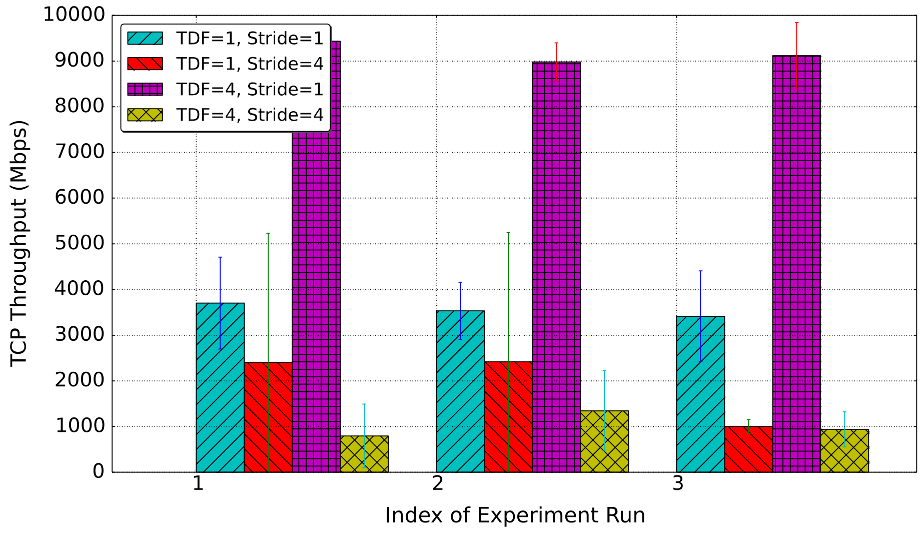

Figure 15 Average TCP flow throughput.

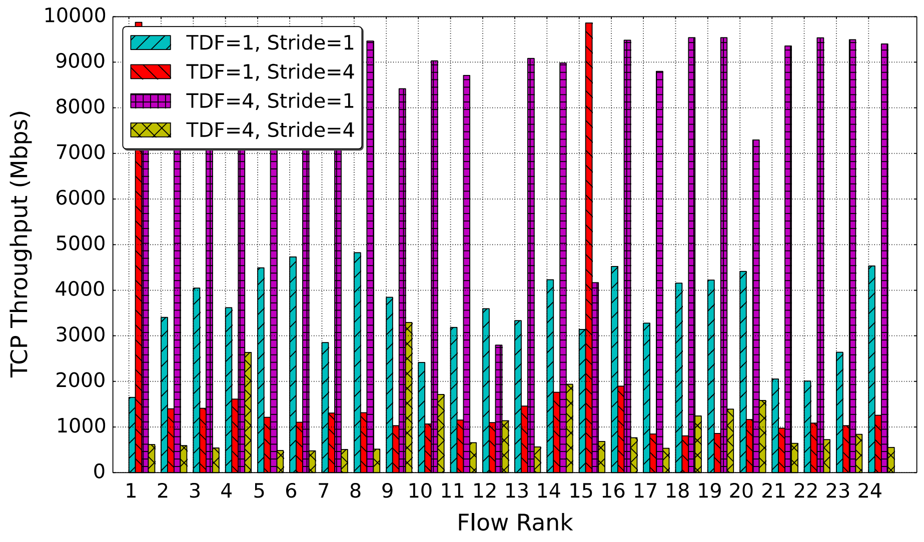

Figure 16 Throughput of individual TCP flow.

collision-free case. Without a correct baseline (benchmark for the collision-free scenario), it is difficult to perform further analysis and qualify the limitation.

7. Conclusion and future works: a new Linux namespaces

In this paper, we present and evaluate a time-dilation-based virtual time system for the Linux-container virtualization technology, and integrate the system to Mininet, a commonly used SDN emulator. One direction to explore further is to design the virtual time as a new Linux namespace. The first namespace *mnt* namespace was introduced in Linux 2.4.19. Currently, there are six namespaces in total. Among them, network namespace, user namespace, and mount namespace provide the critical virtualization environment for network emulation. In this paper, we have demonstrated that the virtual time system could flexibly manipulate the clock of a process, a container, and even an emulator. The behavior fits the characteristics of a namespace: one can isolate the time from the system-wide wall clock by making an entity's time configurable. Therefore, the virtual time concept could be generalized as *Clock Namespace*. However, the current virtual time implementation focuses on the per-process level. Challenges lie in adjusting the granularity of time-sharing in a flexible way, but still ensuring the consistency of virtual timekeeping. Future work will address those issues and develop a virtual time system as a new namespace in Linux.

Acknowledgements—This paper is partly sponsored by the Maryland Procurement Office under Contract No. H98230-14-C-0141, and the Air Force Office of Scientific Research (AFOSR) under grant FA9550-15-1-0190. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Maryland Procurement Office and AFOSR.

References

Al-Fares M, Radhakrishnan S, Raghavan B, Huang N and Vahdat A (2010). Hedera: Dynamic flow scheduling for data center networks. In: *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, USENIX Association, San Jose, California, pp 19–33.

Almesberger W (1999). Linux traffic control: implementation overview. In: *Proceedings of 5th Annual Linux Expo*, pp 153–164.

Chan M, Chen C, Huang J, Kuo T, Yen L and Tseng C (2014). Opennet: A simulator for software-defined wireless local area network. In: *Wireless Communications and Networking Conference*, IEEE Computer Society, pp 3332–3336.

Common Open Research Emulator (CORE). (2016). Retrieved from http://www.nrl.navy.mil/itd/ncs/products/core.

Corbet J (2014). Retrieved from on vsyscalls and the vDSO: http://lwn.net/Articles/446528.

Erazo M, Li Y and Liu J (2009). SVEET! a scalable virtualized evaluation environment for TCP. In: *Proceedings of the 2009 Testbeds and Research Infrastructures for the Development of Networks Communities and Workshops*. IEEE Computer Society, pp 1–10.

FS: A Network Flow Record Generator (2013). Retrieved from https://github.com/jsommers/fs.

Grau A, Herrmann K and Rothermel K (2011). NETbalance: Reducing the runtime of network emulation using live migration. In: *Proceedings of the 20th International Conference on Computer Communications and Networks*. IEEE Computer Society, Maui, HI, pp 1–6.

Grau A, Maier S, Herrmann K and Rothermel K (2008). Time jails: A hybrid approach to scalable network emulation. In: *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*. IEEE Computer Society, pp 7–14.

Gupta D, Vishwanath KV, McNett M, Vahdat A, Yocum K, Snoeren A and Voelker G (2011). DieCast: Testing distributed systems with an accurate scale model. *ACM Transactions on Computer Systems* **29**(2): 1–48.

Gupta D, Yocum K, McNett M, Snoeren A, Vahdat A and Voelker G (2005). To infinity and beyond: Time warped network emulation. In: *Proceedings of the 20th ACM Symposium on Operating Systems Principles*. ACM, Brighton, United Kingdom, pp 1–12.

Handigol N, Heller B, Jeyakumar V, Lantz B and McKeown N (2012). Reproducible network experiments using container-based emulation. In: *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*. ACM, Nice, France, pp 253–264.

Hannon C, Yan J and Jin D (2016). DSSnet: A smart grid modeling platform combining electrical power distribution system simulation and software defined networking emulation. *Submitted to the 4th ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. ACM Alberta.

Heller B (2014). RipL-POX (Ripcord-Lite for POX): a simple network controller for OpenFlow-based data centers. Retrieved from https://github.com/brandonheller/riplpox.

Henderson TR, Lacage M, Riley GF, Dowell C and Kopena J (2008). Network simulations with the ns-3 simulator. *SIGCOMM Demonstration* **15**: 17.

Hibler M, Ricci R, Stoller L, Duerig J, Guruprasad S, Stack T, Lepreau J (2008). Large-scale virtualization in the emulab network testbed. In: *Proceedings of the USENIX 2008 Annual Technical Conference*, USENIX Association, Boston, Massachusetts, pp 113–128.

iperf3 (2014). Retrieved 2014, from http://software.es.net/iperf.

Jails under FreeBSD 6 (2014). Retrieved from http://www.freebsddiary.org/jail-6.php.

Jin D and Nicol D (2013). Parallel simulation of software defined networks. In: *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, ACM, Montreal, Quebec, Canada, pp 91–102.

Jin D, Zheng Y, Zhu H, Nicol DM and Winterrowd L (2012). Virtual time integration of emulation and parallel simulation. In: *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, IEEE Computer Society, Zhangjiajie, pp 201–210.

Jurkiewicz P (2013). Link modeling using ns-3. Retrieved from https://github.com/mininet/mininet/wiki/Link-modeling-using-ns-3.

Lamps J, Adam V, Nicol DM and Caesar M (2015). Conjoining emulation and network simulators on linux multiprocessors. In: *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, ACM, London, pp 113–124.

Lamps J, Nicol D and Caesar M (2014). TimeKeeper: A lightweight virtual time system for linux. In: *Proceedings of the 2nd ACM SIGSIM/PADS Conference on Principles of Advanced Discrete Simulation*, ACM, Denver, Colorado, USA, pp 179–186.

Lantz B, Heller B and McKeown N (2010). A network in a laptop: Rapid prototyping for software-defined networks. In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ACM, Monterey, California, pp 1–6.

Linux Containers (2014). Retrieved from https://linuxcontainers.org.

Liu J, Rangaswami R and Zhao M (2010). Model-driven network emulation with virtual time machine. In: *Proceedings of the Winter Simulation Conference*, IEEE Computer Society, Baltimore, Maryland, pp 688–696.

McKeown N, Anderson T, Balakrishnan H, Parulkar G, Peterson L, Rexford J and Turner J (2008). OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review* **38**(2): 69–74.

Molnar I (2006). GTOD: Mark TSC Unusable for Highres Timers. Retrieved 2016, from https://lwn.net/Articles/209101.

Open vSwitch (2014). Retrieved from http://openvswitch.org.

OpenVZ Linux Container (2014). Retrieved from http://openvz.org/Main_Page.

Project Floodlight (2016). Retrieved 2016, from http://www.projectfloodlight.org/floodlight/.

Reproducing Network Research (2014). Retrieved from https://reproducingnetworkresearch.wordpress.com.

Ryu SDN Framework (2014). Retrieved from https://osrg.github.io/ryu.

S3F/S3FNet (2015). Retrieved from S3F/S3FNet: Simpler Scalable Simulation Framework: https://s3f.iti.illinois.edu.

Sommers J, Bowden R, Eriksson B, Barford P, Roughan M and Duffield N (2011). Efficient network-wide flow record generation. In: *Proceedings IEEE INFOCOM 2011*, IEEE, Shanghai, pp 2363–2371.

Thaler D and Hopps C (2000). Multipath Issues in Unicast and Multicast Next-Hop Selection. Retrieved 2014, from https://tools.ietf.org/html/rfc2991.

The OpenDaylight Platform (2013). Retrieved from https://www.opendaylight.org/.

The POX Controller (2013). Retrieved 2014, from https://github.com/noxrepo/pox.

The Xen Project (2014). Retrieved from http://www.xenproject.org.

Vahdat A (2009). *Scale and Efficiency in Data Center Networks*. UC San Diego.

Wang S-Y, Chou C-L and Chun-Ming Y (2013). EstiNet openflow network simulator and emulator. *IEEE Communications Magazine* **51**(9): 110–117.

Weingartner E, Schmidt F, Lehn HV, Heer T and Wehrle K (2011). SliceTime: A platform for scalable and accurate network emulation. In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, Boston, MA, pp 253–266.

Yan J (2016). Virtual Time System for Linux Kernel. Retrieved 2016, from https://github.com/littlepretty/VirtualTimeKernel.

Received 16 February 2016;
accepted 18 October 2016