TimeSync: Virtual Time for Scalable, High-Fidelity Hybrid Network Emulation

Florin Sultan, Alex Poylisher, Constantin Serban, John Lee, Ritu Chadha, C. Jason Chiang
Applied Communication Sciences (ACS)
{fsultan, apoylisher, cserban, jlee, rchadha, jchiang}@appcomsci.com

Keith Whittaker, Chris Scilla, Syeed Ali
U.S. Army CERDEC
{keith.d.whittaker.civ, christopher.a.scilla.civ, syeed.e.ali.civ}@mail.mil

ABSTRACT

Hybrid network emulation (HNE) comprises a discrete event simulated network and virtual machines that send and receive traffic through the simulated network. It allows testing network applications rather than their models on simulated target networks, particularly mobile wireless networks. Scalability of this test approach is hindered by the time divergence problem: for complex, large-scale simulations, discrete event simulation time advances slower than real time, distorting packet propagation characteristics. To address this problem, we developed TimeSync, a system that uses discrete event simulation time to control and synchronize time advance on virtual machines for large-scale hybrid network emulation. In this paper, we describe how TimeSync controls and synchronizes time perception in hybrid network emulation between simulator and virtual machines, and present experimental results.

I. Introduction

Hybrid network emulation [1]–[3] (HNE) comprises primarily a discrete-event simulated network and virtual machines that send and receive traffic through the simulated network. It allows testing network applications rather than their models on simulated target networks, particularly mobile wireless networks. In some HNE approaches, e.g., [4]–[6], applications can run on top of their native operating systems without any code modification, so the same executable binary can be used in both HNE and real networks.

HNE can potentially address both feasibility and scalability concerns associated with testing applications over target networks. With respect to feasibility, as testing applications over hybrid emulated network only requires the models of network elements, the availability of network element hardware (e.g., next generation radio hardware) will not be an issue, and simulation can allow for testing over various different network topologies and configurations. With respect to scalability, as simulation is used to enable HNE, theoretically the scale of the target network is constrained only by a simulator's capabilities and hardware resource availability.

While the feasibility argument stands valid, the scalability of HNE is actually hindered by the *time divergence problem*: for complex, large-scale simulations, discrete-event simulation time advances slower than real time (typically in a non-uniform way), thus distorting packet propagation characteristics. For example, in a hybrid emulated network where the simulation time advances constantly two times slower than real time, the packet propagation latency perceived by applications running on virtual machines will be twice the expected value dictated by the simulation. To minimize and bound the possible loss of fidelity in HNE, time synchronization between simulation and the external OS domains becomes a necessity, particularly for large scale models where the loss of fidelity can be substantial.

To address the time divergence problem, we developed a system that uses the discrete-event simulation time to control and synchronize time advance on virtual machines for large-scale HNE. Our objectives are: (1) tight constraint on simulation time to advance no faster than real time, (2) tight synchronization of the VM system time with simulation time, (3) tight synchronization of the rate of flow of VM time (as perceived by software running inside a VM) with that of simulation time, (4) small footprint/low overhead.

This paper presents TimeSync[1], our solution to the time divergence problem, and its integration with an operational Xen [7]-based HNE testbed (VAN [6]). The core idea of TimeSync is to create a simulator-driven virtual timeline in the VMs participating in emulation (as opposed to the 'real' timeline created by the hardware platform that VMs normally follow).

In addition to the experiments discussed in this paper, the VAN testbed (with TimeSync) was used to support a test/demonstration based on a tactical network scenario consisting of over 270 nodes running actual military applications, including JBC-P. Currently there are two installations of the VAN testbed – one located in the Applied Communication

[1]The research reported in this document/presentation was performed in connection with contract number W15P7T-08-C-P213 with the U.S. Army Communications Electronics Research and Development Engineering Center (CERDEC). The views and conclusions contained in this document are those of the authors and should not be interpreted as presenting the official policies or position, either expressed or implied, of the U.S. Army CERDEC, or the U.S. Government unless so designated by other authorized documents. Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

978-1-4673-3/12/$31.00 ©2013 IEEE

Sciences' laboratory in New Jersey and the other is on the U.S. Army CERDEC's C4ISR campus at Aberdeen Proving Ground, Maryland.

The remainder of the paper is organized as follows. In Section II, we present the TimeSync architecture. Due to space limitations, implementation details are omitted and the interested reader is referred to [8]. We present an evaluation of the TimeSync prototype in Section III, discuss related work in Section IV, and conclude in Section V.

II. System Architecture

The key idea in TimeSync is to make the time flow inside a VM closely *track* simulation time by dynamically controlling the Xen-VM time interface. Ideally, this should happen continuously, at infinitely small time granularity. In practice, however, extracting time information from the simulator and control of VM time can only be done at finite granularity. Our solution is to track the value of simulation time in small discrete steps, along with an approximation of its average rate of change between consecutive steps. Following simulation time dynamics in both discrete value and rate of progress is essential for good accuracy.

To this end, TimeSync uses two mechanisms: *(i)* simulator-side introspection, to extract time information as the simulation is running, and *(ii)* dynamic time virtualization in Xen, to apply this information dynamically to the VMs via the narrow hypervisor-VM interface. The time information consists of the value of simulation time at a given instant and its projected rate of progress relative to the real time.

To illustrate the time divergence problem, Fig. 1 depicts the simulation and real timelines during a slowdown period in the simulation. The simulation time advance is *discrete*, driven by processed events (shown as dots on the simulation timeline). Processing an event may take a variable amount of time, thus the advance of simulation time versus real time is also *non-linear*. As shown, if we sample the simulation time every Δ units of real time, the sampled values (the top line) yield intervals $d < \Delta$, likely of variable length [2].

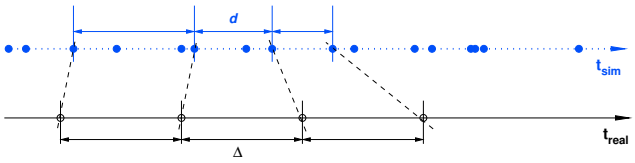


Fig. 1: The flow of simulation time under slowdown (top), compared with real time (bottom).

Let ST(t) be the simulation time as a function of the real time t. Let also VT(t) be the virtual time (as perceived by a VM) as a function of real time t. Ideally, we would like VT to track ST, i.e., VT(t) = ST(t) for any t. Since in a real system this cannot be done continuously we propose a piece-wise linear approximation of ST(t) achieved as follows:

Introspection: every interval of constant length Δ in *real time*, we sample ST = ST(t) and predict a *slowdown factor* $SF \geq 1$ of the simulation time in the next interval.

Control: we constrain the simulator to run no faster than real time, which enforces that SF is never less than 1.

Dynamic Time Virtualization: we make VT(t_i) = ST at the beginning t_i of an interval and approximate VT(t) inside the interval as a linear function of t, VT(t) = ST + $(t - t_i)$ / SF.

Fig. 2 shows the TimeSync system architecture. On the simulator side, a simulator introspection and control module (SICM) samples ST and computes SF every sampling period Δ, then sends them to the VM clock control module (CCM), running as a privileged process on all testbed machines. The CCM injects the (ST, SF) tuple into the Xen hypervisor. There, our time virtualization mechanism (details in [8]) uses it to control all aspects of time perceived by VMs involved in the emulation: system time, its rate of progress, and timers. VMs run freely under the control of the Xen scheduler, but their time is virtualized: ideally, system time is set to ST at the beginning of an interval and flows at a rate of 1/SF until the next update.

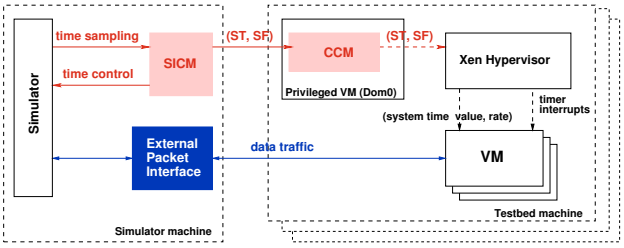


Fig. 2: TimeSync system architecture.

Given the real and simulation time values at the beginning (rt_{prev} and st_{prev}) and at the end (rt and st) of the last sampling interval, SICM computes the actual slowdown factor SF for the interval as $SF = (rt - rt_{prev})/(st - st_{prev})$.

If the simulator did not advance at all in the last interval (so the denominator in the formula above is 0), we set SF to a predefined maximum slowdown factor SF_{max}.

Ideally, when applying SF to a VM, we would like to use the future SF (in the *next* interval). Since we cannot know it, we could rely instead on a prediction based on the history of slowdown across a window of recent intervals. In the current implementation, the SICM predicts SF by assuming that simulation time advances exactly as much in the next interval as in the last one, i.e., uses the actual SF value from the last interval as the projected SF value in the next.

Fig. 3 illustrates the impact of this dynamic time control mechanism on the time progression in a VM. The x axis shows the discrete moments in real time at which a (ST, SF) update is injected into Xen. The blue curve depicts the progression of simulation time used as a reference for sampling ST and for computing each SF value. The red curve shows the progression of VM system time.

Every time CCM injects an (ST, SF) update (every sampling period), our Xen time virtualization mechanism forces the VM system time up or down by a difference δ_i, in an attempt to set it to the exact simulation time. In addition, an update adjusts VM timers and changes the progression rate of VM time according to the predicted SF value in the next

[2]Simulation slowdown can be due to any such factors as heavy computational load for some event, large number of events, and transient lack of resources.

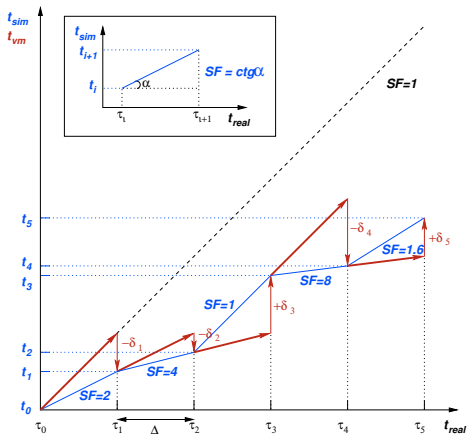


Fig. 3: Progression of simulation (blue, thin solid line) and VM time (red, thick solid line with arrows) with dynamic SF control.

interval. As a result, in each interval between two updates the red curve grows linearly with the same slope that the blue curve had exhibited in the previous interval (the inverse of the predicted SF). This causes the divergence between the two curves seen in the figure over each interval, marked by δ_i at the end of an interval. However, as the vertical arrows show, at the end of each interval this divergence is corrected by a new incoming update that forces the VM system time to the most recent simulation time sample ST received in the update. The error induced by SF prediction is bounded by Δ (reached only if the predicted SF was 1 but the real SF was infinity). Because the interval between updates is small (Δ is on the order of milliseconds), the instantaneous divergence between the two curves is small.

III. Evaluation

In this Section, we present the goals, methodology, and results of an evaluation of the TimeSync implementation.

A. Goals

The goal is to evaluate the impact of TimeSync on the fidelity of the emulation in a large-scale simulation. Intuitively, fidelity refers to how "close" the values of relevant network metrics are when measured in the end hosts, compared to the values for the same metrics as dictated by the simulation. We are thus interested in the distortion that *hybrid* emulation introduces in these metrics.

It is *not* our goal to validate the accuracy of a given simulated network model at a certain scale. Since all our experiments are large-scale, we could not conduct similar experiments on real hardware at such scales to validate against (this is exactly the problem that TimeSync solves). In all experiments, we assume the simulated network is an accurate approximation of a similar real network.

Since our end goal is to evaluate a system that provides time virtualization in hybrid network emulation (TimeSync), we argue that the primary figure of interest in such a system is the *measured* end-to-end packet latency through the emulated network. Because measuring one-way latency requires taking local time samples at different hosts (the sender and the receiver of a packet), this provides good insight into two distinct areas: *(i)* how effective is TimeSync at reducing or eliminating latency distortion, and *(ii)* what other discrepancies can be exposed once TimeSync eliminates the dominating components caused by simulator slowdown.

B. Experiment Setup

The testbed consists of one simulator machine running QualNet [3] under Linux and 7 testbed machines. All machines are Dell PowerEdge 1950 dual-processor Intel Core 2 Quad Xeon, having a total of 8 cores at 2.66 GHz and 16 GB of RAM each and two Intel e1000e 1Gbps network interfaces. Testbed machines run Xen 3.3.2 with TimeSync support, and use the default credit scheduler. Guest VMs run Linux kernel 2.6.18-164.11.1.el5xen in Dom0 and 2.6.21-2950.fc8xen in DomU, respectively. Each testbed machine runs seven end-host VMs, each with 1 GB of memory and 1 VCPU; a Dom0 has 8 VCPUs and 1 GB of memory. The data traffic between the simulator and testbed machines goes over a 1 Gbps switched LAN. A separate 1Gbps LAN is used to multicast the clock signal from the simulator to the testbed machines.

We simulate a network of 200 stationary nodes, structured into seven 802.11a subnets in ad hoc mode, connected by a core FastEthernet subnet (Fig. 4). Each 802.11 subnet consists of 25 nodes except for one subnet of 50 nodes. These subnets are geographically separated and do not interfere with one another. The core subnet consists of seven gateway nodes, one from each subnet. Inter-subnet traffic is forwarded through the core Ethernet. The main reason for choosing this setup is that wireless models are in general more complex and can achieve significant slowdown with a smaller number of nodes.

For unicast routing, we run OLSR with 2 seconds HELLO interval within the 802.11 subnets and OSPF within the core. For routing multicast traffic, we use PIM-DM with an add-on that supports ad hoc networks. The data rate of the 802.11 subnets is 18 Mbps and the radio range is 350 m, with free range propagation. The data rate of the core subnet is 100 Mpbs and the delay between two gateways is 1 ms.

We assign one VM to each of 49 nodes spread across all seven subnets as shown in Fig. 4. The rest of the nodes participate in the simulation but do not generate and do not receive application-layer traffic. The VMs generate application traffic and participate in IGMP and ARP, whereas all other network layer and MAC control traffic is generated by the nodes in the simulated network.

In all experiments, one VM (located in subnet 1) sends UDP packets of fixed size (460 bytes) at a constant rate to a single destination multicast group. All the other VMs in the network run a receiver application that joins the group and logs a trace of packets received. We specifically choose multicast traffic because it allows us to timestamp packets at a single source in the network (the unique sender), which makes comparison and aggregation of latency results over multiple hosts meaningful. We use `mgen` [9] as the sender/receiver application, and associated tools such as `trpr` to compute one-way end-to-end latency based on send/receive timestamps placed in the packets as they travel through the emulated network from the sender to receivers.

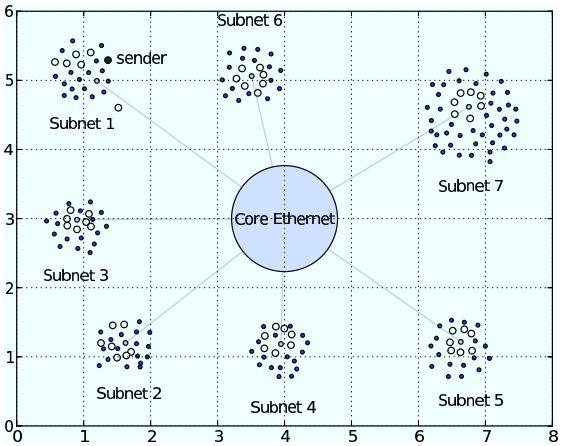


Fig. 4: Layout (at scale, grid in km) of the simulated 200-node network. Receiver nodes (in white) and the sender each have an associated VM in the testbed (49 VMs).

C. Methodology

To compute the latency distortion, we instrument the packet path to record two pairs of timestamps: *(i)* at the two endpoints (sender/receiver), to obtain the "user-perceived" end-to-end latency in VM time, and *(ii)* at the ingress/egress points to the simulator, to obtain the simulated (reference) packet latency in simulation time. Both the sender and each individual receiver timestamp a packet locally, before sending and after receiving it, respectively, using `gettimeofday()` system calls: the difference δT between the timestamps gives us the one-way end-to-end latency. The simulator timestamps the packet at ingress/egress using *simulation time*: the difference δt between these two timestamps represents the simulated packet latency through the simulated network. Finally, we compute the *latency distortion metric* as $dist = \delta T - \delta t$.

We first vary the sender rate to study the impact of increasing simulation load on latency distortion. For each receiver host, we collect traces of packets received. We discard the first packet in a trace to eliminate the impact of ARP.

Depending on its location, not every receiver may get the same number of packets. However, since the loss rates we observe are fairly small (under 1%), we can compute meaningful statistics over *any* of the receiver traces and compare them across different receivers. For each receiver, we compute statistics of the per-packet $dist$ metric.

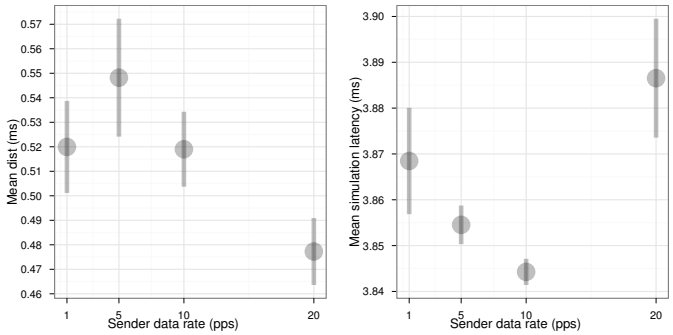
In order to factor out transients and noise caused by simulator startup initialization and transient regimes of routing protocols in the simulated network, traffic injection in all experiments starts consistently at 130 s in *simulation* time. Traffic injection lasts 500 s measured in *end-host VM* time.

D. Experiments

First, we study the effect of varying traffic load. Fig. 5 shows the average per-packet $dist$ (a) and the average simulated latency (b), at 1, 5, 10, and 20 pps, respectively, over traces collected for a receiver in Subnet 2 (results are similar for all receivers). We note that: *(i) TS* achieves a mean $dist$ between 0.47 and 0.54 ms; the relative error of the measured end-to-end latency over the simulated latency is 12 - 14%, and *(ii)* the mean $dist$ is fairly stable across all rates: it is statistically similar at 1, 5, and 10 pps, and about 23 μs lower at 20 pps.

Next, we measure $dist$ at a fixed traffic load (20 pps) and vary simulated network size (and thus mean slowdown due to model complexity). This is achieved by adding or removing nodes in the 802.11 subnets of the 200-node topolgy of Fig. 4, while keeping the locations of the sender/receiver nodes constant. As shown in Fig. 6, mean $dist$ is (a) well under 1 ms, and (b) is generally higher for lower slowdown and lower for higher slowdown. (b) can be explained by lower contention for testbed resources with lower load per unit of real time, and presents an interesting tradeoff between $dist$ and running time.

These results show that TimeSync achieves a useful fidelity of the emulation, under scalable traffic load, with a complex network model and large network sizes. Furthermore, the same experiments conducted with a system that does not use TimeSync (not described here) demonstrated that TimeSync is non-intrusive, i.e., it does not affect simulation behavior.[3]



(a) (b)

Fig. 5: *TS*: Mean $dist$ (a) and mean simulated latency (b) with 95% confidence, at various traffic loads, over one receiver trace (results are similar for all receivers).

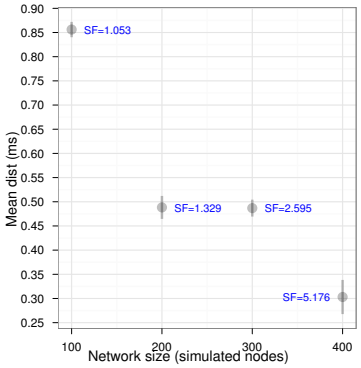


Fig. 6: *TS*: Mean $dist$ with 95% confidence, at the load of 20 pps and various network sizes, over one receiver trace (results are similar for all receivers). Mean slowdown (SF) over the 500 s of traffic is also shown for each size.

To illustrate how TimeSync controls VM time, Fig. 7 plots simulation time against VM time in *TS* at 1, 5, 10, and 20 pps, respectively. We collect two traces of time values during

[3]By design, TimeSync does not affect the simulation correctness since it does not require changes to the underlying models.

a run: the sender VM's timestamp on each packet sent (x axis), and the simulation time at which the packet arrives at the ingress interface to the simulation (y axis). The four curves are practically overlapping lines with a slope of one, which shows that the VM time successfully tracks simulation time at all rates, as visible at this scale. Note that, since trace collection is packet-driven, we do not have data points before traffic starts at 130 s of simulation time. However, we can infer that, prior to that, TimeSync had correctly synchronized VM and simulation time, since they are equal at the first data point (as highlighted by the red lines drawn at 130 s).

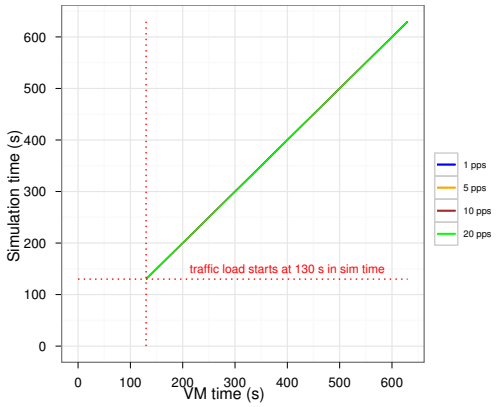


Fig. 7: *TS*: Simulation time vs. VM time at various traffic loads.

E. Discussion

To explain these results, we note that $dist$ has two components: $dist = sync\_err + extra\_err$

where $sync\_err$ is the error due to the disparity between simulation time and VM time, and $extra\_err$ is the error due to the extra latency introduced by various hw/sw components that interpose between the VMs and the simulator.

In *TS*, $sync\_err$ reflects the residual error intrinsic to TimeSync, caused by inaccuracies in tracking simulation time, e.g., due to discrete time sampling, wrong slowdown prediction, applying the slowdown at slightly different instants across testbed machines and VMs. $extra\_err$ reflects the delays that a packet incurs due to propagation (to/from the simulator machine) or processing in the software stacks it traverses (on the testbed and simulator machines). Note that when measuring $dist$ in our experiments these delays are appropriately scaled due to time control by TimeSync. We call the sources of such delays *real-time components* because they operate in the real-world part of the system and introduce delays in real time (as opposed to the simulated delays in the simulator). E.g., in Xen, such delays on the inbound path of a packet to a paravirtualized VM include hardware interrupt processing, virtual physical interrupt mapping and processing, Dom0 backend driver processing and demultiplexing, and Xen inter-domain signaling. Moreover, because of the Xen I/O model in which multiple VMs share a driver domain, the magnitude of these effects is unpredictable and not strictly related to the volume of I/O performed by a given VM.

In an hypothetical ideal system, if $extra\_err$ could be zero, then $dist$ would estimate the error introduced by TimeSync. Conversely, if $sync\_err$ could be zero, $dist$ would estimate the error introduced by the real-time components of the system. In a real system, both $sync\_err$ and $extra\_err$ are non-zero and their contributions to $dist$ cannot be easily separated. Identifying the sources of error and measuring their contribution requires heavy instrumentation on the packet path.

Our measurements show a lower bound of about 200 μs for $extra\_err$ that includes traversal of Xen and simulator network stacks. We believe that in the current implementation and experimental setup we have significantly reduced the impact of these factors, and the residual distortion $extra\_err$ is due to minimal delays on the packet path. While we cannot achieve the ideal case of $dist = 0$ for all packets, the results are well within the bounds of what is possible in the real world, where time on physical machines is synchronized across networks of fairly large latencies.

IV. Related Work

The idea of controlling the perception of time in an OS via virtualization has been used in other contexts [10], [11]. TimeWarp [10] slows down time statically by an integer "time dilation factor" (TDF) in order to present a Xen VM with the illusion of a faster (higher capacity) network. TimeWarp differs from TimeSync both in goals and approach: *(i)* it uses time dilation and existing hardware to study the effects of projected scaling of network hardware performance in physical (as opposed to simulated) environments; *(ii)* it sets TDF only once, and keeps it constant throughout the experimental study. In contrast, we address the problem of accurate network emulation in which the details of the network (of which performance scaling is only one of the many possible parameters) are simulated *externally*. We use *dynamic* virtualization of both system time and its rate of change to make the VM perception of time follow that of the simulator.

DieCast [11] is a framework for building a scaled testbed that reproduces a distributed system by running its nodes as Xen VMs while faithfully emulating their hardware capabilities. It uses time dilation by a pre-configured, *static* TDF to reduce a node's CPU requirements (which allows packing more VMs onto a physical machine) and scales down accordingly the perceived network and disk performance. DieCast is orthogonal and complementary to our work by providing a platform for accurate emulation of node capabilities, with constant TDFs. In contrast, time virtualization in TimeSync is *dynamic* and tracks fine-grained variations in the simulation timeline.

In [12] time virtualization is used in a single-OS network emulation system based on [13]. The simulator runs inside a VE (virtualization environment) alongside end-host VEs, on top of a shared Linux OS. The tight coupling enables [12] to control timing of I/O events and VE scheduling, such that VE time tracks simulation time (at the granularity of scheduler time slice in the worst case). This approach correctly slows down VE time under load, but also allows it to run uncontrollably faster than real time when possible. It also modifies the underlying OS, e.g., to control VE scheduling. In contrast, TimeSync addresses time divergence in a hybrid system consisting of a standalone simulator and end-host

Xen VMs spread across multiple machines. It can support multiple VM OSs, and uses only a narrow interface for time virtualization. The loose coupling with the simulator enables TimeSync to easily scale out to multiple physical machines, as opposed to a single-machine system of [12].

[14] employs the TDF idea of [10] to cope with CPU overload that may bias the results of a network emulation. The system uses a Xen VM to host multiple OpenVZ VEs (the virtual end-hosts), and to enable aggregate CPU load monitoring and TDF control. It changes TDF across coarse-grained epochs, in response to variations in CPU load measured for the VM. The thresholds at which TDF changes, the corresponding TDF values and other parameters require extensive tuning and configuration. We address a similar fidelity problem, but in the different context of a fully-virtualized hybrid emulation testbed. It is interesting to note that in TimeSync the logical simulation time acts as the sole (implicit) load indicator. Tracking it at small time scales in the VMs has a natural feedback effect, reducing load on the simulator without complex system level load monitoring and mappings from load to TDF.

SliceTime [15] is most closely related to our work. It targets a similar setup (distributed hybrid network emulation) and also tries to establish a common timeline for the VMs and the simulator. It however follows the different approach of fine-grained lock-step synchronization: at each step, all VMs *and* the simulator run for a fixed time slice in their own time frame, up to a common barrier, under the command of an external (centralized) controller. Strict real-time control of the running time of a VM requires a modified sEDF Xen scheduler. Within a time-slice, VM time advances at full speed, therefore it may arbitrarily drift from the simulation time. In order to hide this divergence, it is crucial for SliceTime to tightly control the time slice down to microsecond granularity. This forces a tradeoff between the drift magnitude and the scheduler overhead needed to bound it.

In [15], the limits of the approach with respect to multiple machines are not discussed, and the system is only evaluated with VMs collocated on a single machine. One problem of SliceTime in a multi-machine setup is that in an heterogeneous system (i.e., with uneven distribution of VCPUs to CPUs across different machines), an oversubscribed machine will slow down the whole system *in real time* since every VM it hosts has to execute its slice before the simulation can proceed. Note that this delay is different from simulation slowdown (e.g., due to overload): it is simply an artifact of the lock-step scheme stalling the simulator at every barrier. In addition, in [15] the signaling overhead is proportional to the number of VMs and inversely proportional to the slice size. Since [15] does not evaluate overhead, it is unclear how it would scale with large numbers of VMs running on multiple machines.

V. Conclusion and future work

We have described TimeSync, a solution to the time divergence problem in virtualized hybrid emulation testbeds. TimeSync exploits the Xen-VM interface to make the time dynamics in a VM follow the simulation timeline. TimeSync *tracks* the simulation time to within tight error margins, acceptable for testing of applications and network protocols where timeouts are on the order of milliseconds or larger. Due to its small footprint (no coordination between the simulator and the VMs, no interference with simulator or VM execution, no interference with VM scheduling, no change to the VM OS), as we have shown, TimeSync is able to scale to handle large networks and complex simulation models.

We believe the TimeSync idea and mechanisms can be extended to hardware virtual machines (HVMs). Future implementations can take advantage of new features in Xen 4, including improvements in split network driver performance and TSC register virtualization. Another research direction is towards improvement in slowdown prediction, possibly based on longer history and/or internal simulator state (e.g., event queue sizes, processing requirements per event type, etc.). Finally, with the increasing demand for parallel/distributed simulation to help emulate even larger scale networks, an interesting problem is whether and how TimeSync can be integrated with such architectures.

VI. Acknowledgement and notes

The authors would like to thank the Office of the Secretary of Defense (OSD) for their support for developing the VAN testbed technologies. Government programs interested in using the VAN testbed for testing needs can contact one of the CERDEC authors for more information; researchers and developers from the industry and academia who are interested in using the VAN testbed and/or in collaboration can contact one of the ACS authors for information.

References

[1] T. H. J. Ahrenholz, C. Danilov and J. Kim, "CORE: A real-time network emulator," in *Proc. of MILCOM*, 2008.

[2] U.S. Naval Research Laboratory, "Extendable Mobile Ad hoc Network Emulator," 2012. [Online]. Available: http://cs.itd.nrl.navy.mil/work/emane/index.php

[3] Scalable Network Technologies, *QualNet*, 2012. [Online]. Available: http://www.scalable-networks.com/products/developer.php

[4] S. Wang and Y. Huang, "NCTUns Distributed Network Emulator," *Internet Journal*, vol. 4, no. 2, pp. 61–94, 2012.

[5] P. Biswas, C. Serban, A. Poylisher, J. Lee, S.-C. Mau, R. Chadha, C.-Y. J. Chiang, R. Orlando, and K. Jakubowski, "An integrated testbed for virtual ad hoc networks," in *Proc. of TRIDENTCOM*, 2009.

[6] C. Serban, A. Poylisher, and C.-Y. J. Chiang, "Virtual ad hoc network testbeds for network-aware applications," in *Proc. of NOMS*, 2010.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *Proc. of the 19th ACM symposium on OS principles*, 2003.

[8] F. Sultan, A. Poylisher, C. Serban, J. Lee, R. Chadha, C. J. Chiang, K. Whittaker, C. Scilla, and S. Ali, "TimeSync: Enabling Scalable, High-Fidelity Hybrid Network Emulation," in *Proc. of ACM MSWIM*, 2012.

[9] U.S. Naval Research Laoboratory, "Multi-Generator (MGEN)," 2012. [Online]. Available: http://cs.itd.nrl.navy.mil/work/mgen/index.php

[10] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, and G. M. Voelker, "To infinity and beyond: time-warped network emulation," in *Proc. of NSDI*, 2006.

[11] D. Gupta, K. V. Vishwanath, and A. Vahdat, "DieCast: Testing Distributed Systems with an Accurate Scale Model," in *Proc. of NSDI*, 2008.

[12] Y. Zheng and D. M. Nicol, "A Virtual Time System for OpenVZ-Based Network Emulations," 2011.

[13] "OpenVZ: Container-based Virtualization for Linux," 2012. [Online]. Available: http://wiki.openvz.org

[14] A. Grau, S. Maier, K. Herrmann, and K. Rothermel, "Time jails: A hybrid approach to scalable network emulation," in *Proc. of the 22nd PADS Workshop*, 2008.

[15] E. Weingärtner, F. Schmidt, H. V. Lehn, T. Heer, and K. Wehrle, "Slicetime: a platform for scalable and accurate network emulation," in *Proc. of NSDI*, 2011.