A

Temporal Integration of Emulation and Network Simulators on Linux Multiprocessors

JEREMY LAMPS, Sandia National Laboratories
VIGNESH BABU, University of Illinois at Urbana-Champaign
DAVID M. NICOL, University of Illinois at Urbana-Champaign
VLADIMIR ADAM, Real-Time Innovations
RAKESH KUMAR, University of Illinois at Urbana-Champaign

Integration of emulation and simulation in virtual time requires that emulated execution bursts be ascribed a duration in virtual time, and that emulated execution and simulation executions be coordinated within this common virtual time basis. This paper shows how an open source tool TimeKeeper for coordinating emulations in virtual time can be integrated with three different existing software emulations/simulations: CORE, mininet, and EMANE, and with two existing network simulators, ns-3 and S3F. The integration does not require modification to those tools. However, the information TimeKeeper needs to administer these emulations has to be extracted from each, and we discuss the issues and challenges we encounter there, and the solutions. The S3F integration is specialized, and shows how we can treat bursts of emulated execution just like an event handler in a discrete-event simulation. Through these case studies we show the impact that the time dilation factor has on available resources, execution time, and fidelity of causality, and that deleterious behaviors suffered under best-effort management of emulation processes can be corrected by integration with TimeKeeper. The key contribution is that we've shown how, using TimeKeeper, it is possible to bring virtual time to many existing emulators, without needing to change them.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; D.4.4 [**Operating Systems**]: Communications Management—*message sending, communication management*; D.4.8 [**Operating Systems**]: Performance—*measurements, simulation*; I.6.3 [**Simulation and Modeling**]: Applications—*Miscellaneous*

Additional Key Words and Phrases: Simulation, Emulation, LXCs, Virtualization, Time Dilation, CORE, ns-3, S3F, EMANE, Linux Kernel

1. INTRODUCTION

Software emulation involves running executable applications on some virtual platform. Sometimes the operating system is virtualized, sometimes the underlying hardware is virtualized, sometimes the network is virtualized. The main attraction of emulation for network models is that it enables high fidelity execution behavior and traffic generation. Emulators such as CORE[Ahrenholz et al., 2008], mininet[min, 2015], and EMANE[nrl, 2010] execute software stacks to generate and consume traffic. They include ways of simulating network delays and the impact of congestion, all within the package. The delay simulation is tied to the system wallclock. To emulate a link delay of 10 ms the emulator will use a timer to wait 10 ms before delivering (or recognizing) a message which is so delayed. There is a limitation however. Any computation whose actions record time or depend on time measurements will reference the system clock. Concurrency in the modeled system that would be evidenced in the real system (e.g., two processes sampling a clock more or less simultaneously and observing more or less the same time value) is not preserved in the emulation. Something as simple as measuring latency of a ping is at risk. This limitation is exacerbated as the size of

DOI: http://dx.doi.org/10.1145/0000000.0000000

network models grow relative to the ability of the computing platform to execute the model workload. Computations that depend on time measurements can deviate very seriously in their behavior from what would be observed in a real system.

This problem can be ameliorated though if the real-time clocks referenced by the emulation processes are replaced with virtual time clocks tied to each process. By managing the advancement of virtual time at each process it is possible to provide a veneer of concurrency to a model. This time management is a form of simulation, with execution of the emulated processes playing the role of event executions. This paper concerns this coupling of emulated software stacks (and sometimes network communication) with simulation, with the objective of providing better temporal accuracy of emulation based models. The research applications that motivate our interest call for models with large numbers of simulated and emulated entities. This drives us towards the lightweight end of the emulation spectrum where as much as possible is shared between co-hosted virtual machines. Integrating lightweight emulation and simulation within virtual time is our goal.

Others have brought virtual time to emulations. An early effort was motivated by the objective of testing distributed applications [Gupta et al., 2005]. The idea was to have the virtual time within a virtual machine progress more slowly than real time, in order to make the (real) network appear to be performing faster. The technique used (with heavier weight emulators) is simply to rescale the clock time in the virtual machine, essentially with a multiplicative factor called the *time dilation factor* (TDF). A virtual machine (VM) with TDF α is considered to run α times more slowly in the modeled system than on the actual execution platform.

A more controlled approach to transforming real time to virtual time was explored as the lightweight OpenVZ[ope, 2014] emulation platform and the S3F[Nicol et al., 2011] simulator were integrated and studied [Zheng et al., 2012a]. This approach embedded OpenVZ containers in virtual time by the simple expedient of transforming returns of calls to the system clock. A container had a TDF α that modeled the duration of an uninterrupted execution burst that took t units of measured time as a duration of t/α units of virtual time. A key differentiator between this work and prior art was that viewed from a wall-clock time perspective, a VM's advancement in virtual time is not continuous. It runs for a burst during which virtual time advances, is paused during which time virtual time is not advanced at all, and advances again when the VM is running again. Modifications to the OpenVZ scheduler allowed OpenVZ to run all its containers through a window of virtual time of size Δ, stop, and exchange traffic with the S3F simulator, which alternatively is also run for Δ units of simulation time.

A follow-on effort by the same group produced *TimeKeeper*[Lamps et al., 2014], which brings virtual time to general Linux processes using small modifications to the Linux kernel. The real-to-virtual time translation mechanism is the same as used with OpenVZ, and the scheduling capability is like OpenVZ's. The purpose for TimeKeeper was to bring virtual time to standard Linux, and not rely on a software platform with smaller exposure and user base.

In the present paper we take TimeKeeper, extend it further, and show that it seamlessly extends virtual time to three lightweight network emulators: CORE [Ahrenholz et al., 2008], Mininet [min, 2015], and EMANE [nrl, 2010]. Each of these contains its own network simulation, with physical-clock based emulation of communication delays. For each of these networks we illustrate through example either that without TimeKeeper there are network models for which the emulations demonstrably fail to do the right thing, and/or the TimeKeeper causes the emulated system to behave as expected as the model size grows or as TDF grows. For these three systems integration with TimeKeeper is simply a matter of bringing those applications up under TimeKeeper's control, and it does the rest. An important contribution here is the utter transparency of the integration from the point of view of

the emulator implementations. However, some specialized effort is required to extract from each the information TimeKeeper needs, which we will describe shortly.

We then show how TimeKeeper extends the capabilities of two network simulators, ns-3 [Henderson et al., 2006] and S3F [Nicol et al., 2011]. ns-3 is designed already to integrate with emulation based devices, but approaches the integration by aligning ns-3's simulation clock advancement directly to the wallclock. The emulated devices are encapsulated in Linux LXC containers and from ns-3's point of view are black boxes that generate and consume traffic. Time advancement is solely the responsibility of the ns-3 event list scheduler. However, as the model size and corresponding emulation and simulation workload increases, it is possible for the simulator to fall behind, the rate at which it is able to advance virtual time is slower than the wallclock. We demonstrate this phenomena but then show that by putting ns-3 under TimeKeeper control we can correct the problem simply by slowing the advancement of virtualized real time to not exceed the rate at which ns-3 is naturally able to advance.

All four integrations previously mentioned involve TimeKeeper giving a Linux process controlling an emulated software stack a time-slice of real time during which it runs continuously. Each active process is given the same slice of real time, and the virtual time advancement is maintained to reflect concurrent execution of these processes. TimeKeeper is essentially time-stepping the emulated software stacks. But because inter-process communication is simulated within these packages and we aren't altering them, TimeKeeper is not involved in coordination of messages. TimeKeeper is capable of supporting more sophisticated time-management of emulated traffic generation and message delivery, which we illustrated using S3F. The advancement of emulated devices is placed under control of the S3F composite synchronization method [Nicol and Liu, 2002a]. In this approach, from the S3F synchronization point-of-view, an emulated device attached to an S3F "timeline" is a means of advancing that timeline's virtual time, just as executing discrete-event simulation model events are a means of advancing a different timeline's virtual time. The synchronization between timelines is independent of the means by which virtual time advances on those timelines. Unlike TimeKeeper's integration with other emulators, with S3F the execution durations vary in order to stop an emulated device from progressing beyond a point in virtual time when it might receive a message.

The take-away message of this paper is that TimeKeeper can be integrated with various network emulators and simulators, and that in each of the five instances we consider it enables those integrated systems to model systems with higher fidelity than they could before, and/or model larger systems than they could before. It shows that integration of TimeKeeper is often possible in a way that is transparent to the emulator/simulator under its control, but requires scripting to effect the integration. The utility of TimeKeeper for these and other possible emulators and simulations is quite high.

2. TIMEKEEPER

TimeKeeper [Lamps et al., 2014] is a kernel module for Linux that brings the notion of virtual time to selected Linux processes. TimeKeeper affects the scheduling of processes under its control, and the notion of "time" these processes experience. What TimeKeeper does is conceptually simple. For each application we write a simple script to bring up the application and TimeKeeper, and use it to notify TimeKeeper of the application processes just spun up that it will place under its control. In time-stepped integrations TimeKeeper allocates common-size time-slices to processes in rounds, with each process getting one execution burst each round. The advancement in virtual time of a process by a execution burst is a function of the length of the burst, and the time dilation factor (TDF) ascribed to the process. During an execution burst, any system call the process makes involving time is intercepted. Instead of reporting the system clock at the time of the call, TimeKeeper reports a virtual time, which is a function of the virtual time at the beginning of the execution burst, the length of execution time since the beginning of the execution burst,

and the TDF. System calls intercepted by TimeKeeper include *gettimeofday*, *time*, and *clock_gettime*. A Linux process can also schedule sleep operations or poll on network sockets or files for new data using *sleep*, *nanosleep*, *select*, *poll*, *epoll* and *timerfd*. TimeKeeper modifies the implementation of these system calls so that when called by a process under TimeKeeper administration the specified relative span is accorded in virtual time.

In addition TimeKeeper can,

— Ascribe to, and dynamically change the TDF assigned to a process.
— Cause a stopped process to start, immediately.
— Cause a running process to stop, immediately.
— Schedule its processes so that even when different processes have different TDFs, they advance together in virtual time.

With this functionality the interface to TimeKeeper provides an application with the ability to change a process' TDF, and to run a process for a specified duration of virtual time, and know when the processes has stopped at the end of that epoch. We will use this functionality to integrate emulated execution of hosts within Linux processes with network simulators that carry the traffic they generate.

2.1. Scheduling in TimeKeeper

TimeKeeper can be used to control LXC containers, processes with network name-spaces, or ordinary Linux processes. The way it transparently accommodates all of these is by tracking the growth of process trees. After a process is initially identified to TimeKeeper for administration, TimeKeeper tracks the appearance of new processes spawned by processes known to already be under its administration. This means, effectively, that TimeKeeper is keeping track of process trees, each of whose root is a process originally signaled to TimeKeeper. In this way all of the processes associated with an LXC container or name-space are grouped together. TimeKeeper allocates execution time-slices then to process-groups, which we'll call a *procgroup*.

In time-stepped mode TimeKeeper allocates to each procgroup a burst of execution time, in rounds. Each round each procgroup is serviced once. Each round the processes in a procgroup are advanced in virtual time by the same amount W. Allowing for different procgroups to have different TDF values, the burst length for a procgroup with TDF α in real time is αW. With every round TimeKeeper scans the process table to update its knowledge of extant processes and threads. New processes are credited to the procgroup of the processes that spawned them.

TimeKeeper assigns all computational workload associated with a procgroup to one CPU, and schedules the execution of tasks, approximating round-robin scheduling with process priorities. For each procgroup TimeKeeper maintains its own run-queue of live processes, and takes responsibility for scheduling them to run. When it schedules the process at the head of the queue to run, the execution time quanta assigned is proportional to the relative process priority (i.e. higher priority means a higher positive weight) times the number of threads. However, TimeKeeper also strives to advance all the procgroups in virtual time uniformly.

TimeKeeper starts a procgroup running in a burst by sending the process at the head of the TimeKeeper queue for that container a Linux SIGCONT signal. It stops that process at the end of its residual service time or at the end of the synchronization window (which ever comes first) by sending it a SIGSTOP signal. If the synchronization window is not reached, it proceeds in a round robin fashion to the next runnable process in the container's queue. From a logical point of view, the allocation of execution quanta to procgroups is orthogonal to the maintenance of temporal synchrony among them.

The use of kernel level signalling techniques allows TimeKeeper to bypass the linux scheduler. Processes under the administration of TimeKeeper are assigned to the real time

scheduling group (RT-SCHED) and when a SIGCONT signal is sent to such a process by TimeKeeper, it is added to the head of the linux scheduler run queue. It starts to run quickly by pre-empting other non administered processes which may have been scheduled on the associated CPU. When the process is stopped by TimeKeeper, it is immediately taken off the run queue and the CPU may be used for other tasks. Thus, by simply manipulating the scheduling groups and using signalling techniques, TimeKeeper is able to leverage the capabilities of the linux scheduler to control the scheduling order and run time of each administered process.

2.2. Role of the Time Dilation Factor

In its original application, the TDF was used to artificially change the speed of one system component (e.g. the network) with respect to the other (e.g. hosts). Another application is to use the TDF to rescale measured execution time in order to model the time needed to execute that code on a different CPU. So by changing the TDF we can model the code running on a processor that is four times faster ($\alpha = 0.25$) or four times slower ($\alpha = 4$) than the processor used to execute the emulation.

Another important role for the TDF turns out to be in helping to preserve causality, when such support is needed. It is sometimes necessary to make the virtual time scheduling window size W so small that a message sent by one device during a round will not be received by a different device in that same round. This ensures that the recipient process has a chance to see the message before emulating past what its receive time would be in an actual system, and so preserves causality by ensuring that the computation it performs in "the next" round has all the state information it should have.

In a time-stepped approach W introduces a certainly sloppiness in any latency computations. Now starting and stopping an emulation process is controlled by a timer and inter-process signalling, which means that when W is translated into a very small *real time* window αW, the uncertainties in timer firings and the magnitude of the time needed to send, receive, and recognize a signal can allow considerable variation in what is actually executed between when the process starts and when it recognizes a signal to stop. We can mitigate these effects by increasing α, which has the effect of greatly reducing the relative variation in the number of instructions executed within a window (understanding that this reduction in comes from increasing the number of instructions executed per window.) From the model point of view this can be just a interpretation of time units. As we will see in our assessment of CORE, this comes at the cost of increased execution time for the overall experiment. Still, it is important to understand the issue, to understand a means of addressing the issue, and to understand the implications of that solution.

2.3. Integration of TimeKeeper

The initiation of a TimeKeeper controlled session requires that it be told of the processes to be placed under its virtual time administration. We have to extract this information from different emulators/simulators in different ways. Once abstracted the process id's are presented to TimeKeeper with other experiment specific parameters such as the TDF values for each process, and the synchronization quanta. Details for extracting the process ids from the emulators are as follows.

CORE. We modified CORE's GUI to integrate TimeKeeper, basically to accept TDF descriptions for the processes identified in the GUI. The GUI starts CORE, the internals of which spins up specific processes called *vnodes* where the code execution occurs. Mostly as a matter of expediency we modified CORE slightly so that when a *vnode* starts it reports its process id, which the interface then gives to TimeKeeper. We believe it would be possible to re-engineering the CORE interface with TimeKeeper to resemble the script-driven interfaces of the software tools (which were developed later), and then

analyze the characteristics of executing processes to identify the *vnodes*. The existing approach allows one to give distinct TDFs to the *vnodes*, mining of the process table is easiest when the TDF is the uniformly the same.

mininet. We wrote a Python script which brings up mininet, extracts process ids and passes them to TimeKeeper. The mininet configuration file describes the network topology and identifies the binary executables that will run in each emulated device. An understanding of how mininet spawns processes is important. For each device a process is spun up to represent it. That device process consults the mininet configuration and spawns processes bound to that device. After being notified that mininet is up and running, the star-tup script mines the list of active processes, discovers procgroups and reports the roots of each to TimeKeeper (which will discover the children with its own analysis.)

EMANE. The EMANE start-up process is similar to that of mininet. We wrote a python script in which the experimenter specifies each device's TDF, model parameters, initial physical locations and applications to run on each LXC. From these parameters (except for TDF) it creates various XML formatted files that EMANE needs for any experiment. It launches EMANE in a distributed deployment configuration. Like the mininet script, once EMANE is running the host processes can be identified from the process table and their ids are passed (with the configured TDFs) to TimeKeeper. The processes spawned by the host processes are also detected by TimeKeeper, which then assembles procgroups.
Proper causality is enhanced if the synchronization window is small enough that no message whose transmission begins within a round also finishes within the round. A lower bound on the time between when the first bit of a packet is transmitted and when the last bit is received is the minimum packet size (in bits) divided by the channel bandwidth (in bits per second). As these quantities are not specified in the EMANE configuration file, but rather are buried within EMANE, the user needs to put this lower bound in the start-up script as the TimeKeeper synchronization window size W.

ns-3. Every ns-3 model is initialized executing a user-written program that creates the topology and initializes the network data state. For ns-3 models that use LXC containers, this program identifies and launches them as well. When using TimeKeeper, we require that this program acquire the process ids of the core ns-3 simulator and LXC containers, assign a TDF to each, and pass the ids to TimeKeeper for administration.

S3F. We extended S3F's input configuration parser to allow specification of emulated hosts with TDF values in project files. The S3F startup process was modified to recognize these, start the processes, and communicate their process ids to TimeKeeper. Section 9 describes a number of further extensions to S3F to support fine-grained interaction with TimeKeeper-administered processes.

2.4. Further Extensions to TimeKeeper

TimeKeeper is a work in progress, and has evolved since the PADS paper which motivates the present paper. We report here changes made since which increase its utility.

procgroups and scheduling. Our work in integrating EMANE and mininet in particular use cases revealed some gaps. In both systems a "host" process spawns application processes, potentially dynamically. In the real system the host will share its computational

resources with these processes. Correspondingly we modified TimeKeeper to dynamically detect which processes need to share resources (what we have called a procgroup), and to schedule these. We encountered subtle issues with fair scheduling of individual threads in multi-threaded dilated processes. We solved this by keeping track of CPU time for each thread and ensuring that it is shared fairly.

dilation through `netem`. Unlike CORE and EMANE, mininet emulates a packet's transmission delay with the use of a Linux kernel module called `netem` (network emulation). `netem` intercepts all the packets sent from the emulated host's network interface and applies the specified propagation delay. We extended TimeKeeper to associate each host's interface with the dilated processes that use that interface for sending packets. For such dilated processes, we modified `netem` to dilate the transmission delay in virtual time. This new TimeKeeper extension will have utility for any other emulator which employs `netem`.

traffic monitoring in virtual time. Another modification was motivated by our use of mininet/TimeKeeper in observing emulated network traffic using standard tools such as tcpdump [tcp, 2015] and wireshark [wir, 2015] to capture and record packet captures in virtual time. We include as part of TimeKeeper modifications which put virtual timestamps on packets transmitted and received at the interfaces associated with a dilated process. The ability to delay and capture packets in virtual time is an extension to [Yan and Jin, 2017], and is necessary to dilate both computation and communication delays in a mininet experiment.

3. EVALUATION METHODOLOGY

Our evaluations are designed to demonstrate how selection of TDF impacts the execution of a model, to show how use of TimeKeeper ameliorates model mis-behaviors that are possible when emulation is not coordinated in virtual time, and to raise the art of integrating emulation and simulation to new levels through our integration with S3F. For each emulator or simulator we design experiments to highlight some subset of these three goals.

For CORE we consider an experiment that measures available network bandwidth, using the Linux utility *iperf*. We don't here show ways that CORE misbehaves without TimeKeeper, but do show that as one changes TDF, the system allocates CPU resources to the CORE experiment as one would expect, with the impact of lengthening the real time required to perform an experiment over a fixed length of virtual time.

For mininet we design an experiment which shows that lack of virtual time control allows processes with heavy weight computation to skew *iperf* measurements. However, under TimeKeeper administration the bandwidth measurements are what they should be, because in the real system bandwidth is not affected by unrelated computation.

The EMANE experiment uses a more complex example. We emulate wireless routers, using a moderately complex software stack to compute routes. With this example we show that without virtual time control the functional behavior of the application (in terms of the nature of routes produced and end-to-end latencies) can deviate significantly from what would be observed in the field, but that using TimeKeeper these problems disappear.

Our evaluation of ns-3 demonstrates that using TimeKeeper can solve a problem already known to ns-3 designers and users. Because the network simulation portion of ns-3 is tied to the wallclock in a 1-1 ratio, if a simulation model has so much computational work that it cannot advance the simulation clock as fast as the wallclock advances, the model exhibits "jitter", meaning the network simulation and emulation are out of virtual time synchronization. Simply running the ns-3/emulation problem under TimeKeeper with larger TDF can greatly reduce jitter.

Finally, we demonstrate TimeKeeper's ability to synchronize device emulations selectively, to become a natural extension to S3F. Device emulations that produce or consume traffic are treated just like event handlers in a discrete-event simulation. We show by example how

this level of control improves the quality of fine-grained temporal measurements, such as end-to-end latency.

4. RELATED WORK

The notion of integrating emulated software stacks with simulated networks is not new. The Wisconsin Wind Tunnel [Reinhardt et al., 1993] project used parallel simulation of a parallel machine network to carry traffic generated and consumed by executing application code; the LAPSE [Dickens et al., 1996] system did the same, differing from the WWT in its treatment of parallel network simulation. The idea is re-expressed in [Erazo et al., 2015], where attention is paid to how lack of controlled integration can lead to anomalous behavior. The capacity of emulation to produce anomalous results is nicely shown in [Chertov et al., 2009]. [Zheng et al., 2012b] show how to generalize these approaches by integrating OpenVZ containers with network simulation, which is extended in [Jin and Nicol, 2015] specifically to simulation of software defined networks (using OpenVZ) and in [Yan and Jin, 2017] using Linux containers. The present work is an extension of [Lamps et al., 2014], where it is shown how the emulation-control platform TimeKeeper can be applied to a number of different emulators. The contribution of TimeKeeper is its generality; we make the point in this paper that the TimeKeeper approach is largely transparent to the emulators controlled.

All of this work rests on a history of virtualization e.g., [Rosenblum, 1999; Watson, 2008; Barham et al., 2003; ope, 2014; qem, 2017; Watson, 2008], involving software used by professionals daily. In the space of virtualization approaches differences exist in the resources shared by the virtual machines. The approach studied in this paper is applied to very lightweight virtualization, although in principle this is simply because it is relatively easy to connect a simulator and a lightweight container system running under the same operating system, Linux.

An emulated approach to system study uses virtualization, and is a common means of understanding models. Three very widely used emulation testbeds are Emulab [White et al., 2002], DETER [Wei et al., 2009], and PlanetLab [Vahdat et al., 2002]. In Emulab, the experimenter creates the desired network, then is granted access to the nodes within the testbed for a specific period of time. The experimenter is given root access and may install any OS on each node. PlanetLab consists of connected machines around the world. Deter is an emulation platform originally based on Emulab which is focused on cyber-security. PlanetLab gives you a Linux LXC container on each machine in the experiment. Therefore, cannot run a custom OS, and will competing for system resources with other PlanetLab users on the same node.

In addition, emulators used in our study include CORE [Ahrenholz et al., 2008], mininet [min, 2015], and EMANE [nrl, 2010] which are interesting in that each contains within it a network simulator. Latency of messages is modeled directly through timers.

Finally, related work concerns embedding emulators in virtual time. Example systems include [Zheng et al., 2012b; Grau et al., 2008; Gupta et al., 2011; Erazo et al., 2009; Weingärtner et al., 2011; Lee et al., 2014]. SVEET[Erazo et al., 2009] and DieCast [Gupta et al., 2011] both make modifications to the Xen hypervisor in order to give each VM a notion of virtual time. The modification changes the rate at which interrupts are sent from the Xen hypervisor to each VM. SVEET is a performance evaluation testbed that integrates emulation and simulation, scaling simulation advancement in proportion to the wallclock (we will see this also in ns-3). DieCast scales the perceived performance of various hardware components. This is useful if one needs to create a large experiment, where the number of required experiment nodes is larger than the number of nodes in your testbed. TimeKeeper is different from both DieCast and SVEET in three ways. First, our solution supports dynamic changing of TDFs. Next, TimeKeeper can manage concurrent emulation processes that have different TDFs, while synchronizing their virtual times. Finally, TimeKeeper

supports lightweight virtualization and uses minor Linux Kernel modifications instead of a Xen-based approach.

5. CASE STUDY-1: CORE

CORE is a standalone network emulation platform which uses lightweight virtualization to emulate the networking stack of end hosts and routers while simulating the links between these devices using Linux bridges. CORE uses network namespaces to create logically separate networking entities (each process will have its own routing tables, network adapters, and so forth). For the purposes of this discussion, the most important point is that CORE uses a timer to emulate the latency of a sent message over its channel.

5.1. Integration with TimeKeeper

Integration of CORE with TimeKeeper is simply a matter of reporting to TimeKeeper the processes involved in emulating each host (called *vnoded* daemons). CORE's GUI and back end daemon control progress of the emulation phase and handle topology creation. The GUI allows a user to specify the link model, location of hosts and connectivity between them. The specified information is communicated by the GUI process to the back end daemon which spawns a vnoded daemon for each host and uses Ethernet bridging tables to configure connectivity.

Our modifications to CORE simply allowed an user to specify a TDF for each host and communicate all process IDs to TimeKeeper. The CORE's GUI was modified to include an option to specify TDFs to each node and this information was transferred to the back end daemon. The back end CORE daemon was also modified to acquire process IDs of newly created nodes and communicate them to TimeKeeper along with the assigned TDFs. The GUI was additionally modified to allow the user to signal start/stop of the synchronized virtual experiment.

5.2. Experiments

We use CORE to illustrate the impact that increasing TDF has on the availability of system resources, and on the length of real time needed to evaluate an experiment for a given duration of virtual time. The experiments were conducted on a Dell Studio XPS Desktop, with 24 GB of RAM, and 8 Intel Core i-7 CPU X 980's @ 3.33GHz. The machine was running 64-bit Ubuntu with a modified 3.10.9 Linux Kernel.

The experiment involves measurement of the maximum bandwidth that can be squeezed out of a network using TCP (alternatively, using UDP). We use the Unix tool *iperf* to make this measurement. *iperf* works by pushing messages as fast as possible over the channel, and measures the achieved throughput, which is converted to network bandwidth. We set up CORE models with clients and servers, using the basic on/off wireless model. These ran *iperf* to measure the bandwidth achieved with a *simulated* network. The model is IO bound: The computational work of running *iperf* is negligible compared to message latency. If the nominal timer-based latency for a message is, say, 1 ms, then under a TDF of α the latency actually observed is α ms. The implications are two-fold. First, since it takes α times longer to deliver a message than in the nominal case and the application is IO-bound, we expect the overall emulation of a given period of virtual time to take α times longer. Second, since the same amount of work is being accomplished (e.g. the same number of messages sent) over a longer period of time, we expect the CPU utilization to decrease by $1/\alpha$.

We see both of these effects in experiments using a 3-node topology comprised of one switch, one server. We used *iperf* to measure the bandwidth between the client and server over a period of 10 virtual seconds. The experiment was repeated numerous times and the average bandwidth, CPU utilization, and length of experiment was recorded. This process was repeated across many different TDFs. As one expects, bandwidth (in virtual time) is unaffected by TDF, because all we are really changing is the length of a timer which

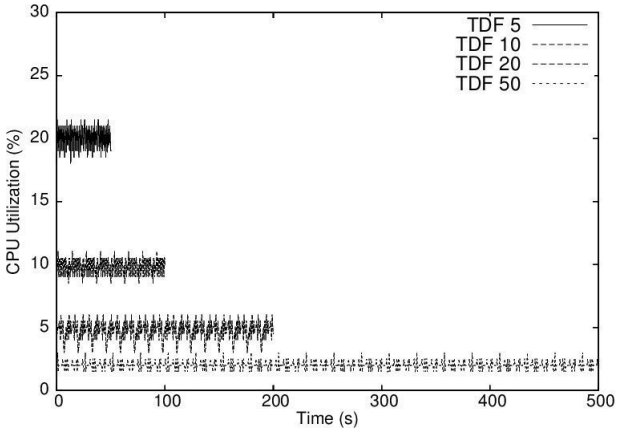


Fig. 1: Relationship between CPU utilization and experiment time

we ascribe to a fixed duration in virtual time. We also performed the experiment without TimeKeeper, where we observed the experiment takes approximately 10 seconds, and has the CPU utilization pegged at 100%.

Figure 1 explores how the TDF affects both CPU utilization and the physical time required to run an experiment, and we observe what we expect we should. For a given value of α, the CPU utilization is close to $1/\alpha$, and the experiment takes approximately 10α seconds. While seemingly an artifact, the decrease in CPU utilization with increasing α means there are more system resources available to simulate more model, in scaled-by-α real time. This turns out to be important for our integration of TimeKeeper with ns-3.

6. CASE STUDY-2 MININET

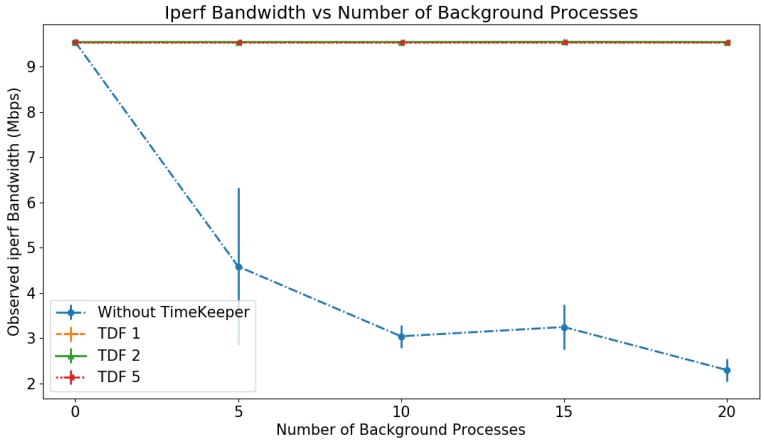
Mininet [min, 2015] is a popularly used network emulator, well-suited for emulating Software Defined Networks (SDNs). Hosts in mininet are assigned separate network namespaces while sharing process namespaces with each other. Mininet switches are processes which can interact with SDN controllers using the openflow protocol.

6.1. Mininet Overview

Like CORE, mininet uses separate network namespaces for hosts and switches. In an SDN emulation, the controller populates the switch routing tables with flow rules. Mininet allows creation of bi-directional links from either a host or a switch to another switch. Each link results in creation of two virtual interfaces in the appropriate namespace. The link can be assigned a specified propagation delay and a maximum bandwidth which is enforced by a part of the linux kernel called `netem` [net, 2016]. Every packet transmitted from a host or a switch interface is intercepted by `netem`, which applies bandwidth and propagation delay to the transmitted packet before queuing it for transmission. After a packet is transmitted from one interface of the link, it simply shows up at the other interface with no further delay.

6.2. Experiments

Without TimeKeeper, Linux schedules mininet process executions using ordinary Linux scheduling policies. Now imagine an experiment where, like the CORE experiment, we are measuring bandwidth using *iperf*, but also imagine that unlike the CORE experiment we have additional background processes running on each mininet host that do no communication, but perform heavy-weight computations. Recalling that *iperf* measurements are



Variation of observed iperf bandwidth

Fig. 2: Mininet: Comparison of observed iperf bandwidth in a simple 5 node mininet topology for best-effort emulation and TimeKeeper controlled experiments as the number of background processes and TDFs is varied

based almost solely on emulated link delays, imagine the impact of heavy outside load on recognition of a kernel timer. If the timer was set to fire after 1 ms, but on each host there are multiple background processes being scheduled as well, there could be a significant delay between the instant when the timer fires and when the process which scheduled it is allocated CPU resources again. Effectively the 1 ms delay becomes larger, potentially much larger, which will diminish measured bandwidth. However, under TimeKeeper, the computation time allocated to the background processes is accounted for *in parallel* with the *iperf* experiment, which then is unaffected by them.

We demonstrate these points using experiments that were conducted on an Intel i-7, 2.4 GHz Lenovo Laptop with 16 GB RAM. The experiments were restricted to use only two cores out of the 8 cores available on the system. We created a simple 5-node linear mininet topology with nodes at either end running an iperf client and server respectively. We configured the iperf client to use a maximum udp bandwidth of 10 Mbps and ran cpu intensive background processes on all other nodes. Background processes were assigned higher priority and performed factorial computation tasks.

In Figure 2, we see that as the total number of background processes is increased, under best effort emulation the *iperf* measurement is increasingly impacted by the background processes, resulting in smaller observed bandwidth. However, under the control of TimeKeeper, the observed bandwidth was very close to the actual bandwidth regardless of the number of background processes or operating TDF.

In the next experiment, we demonstrate the benefits of recording/monitoring traffic in virtual time. We use a hypothetical smart grid network topology depicted in Figure 3. One host was attached to each link and all links were assigned a latency of 1ms. The host at switch s_1 runs a DNP3 master process which periodically polls a DNP3 slave process running on the host attached to switch s_3. The polling frequency is 10 ms, i.e, after receiving a reply from the slave, the master waits 10 ms before sending the next poll request. The host at switch s_1 also runs a UDP server. Every other host is running a UDP client and also performs some compute intensive operations. Multiple UDP flows can exist between the master and slave hosts and they constitute back ground traffic in the network. This

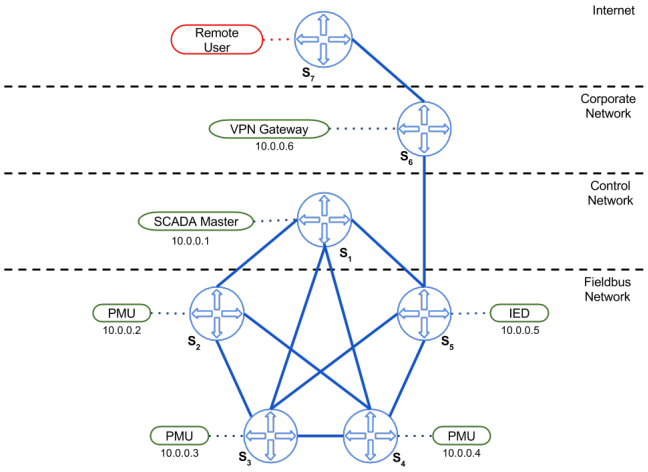


Fig. 3: A hypothetical smart grid control network topology created on Mininet

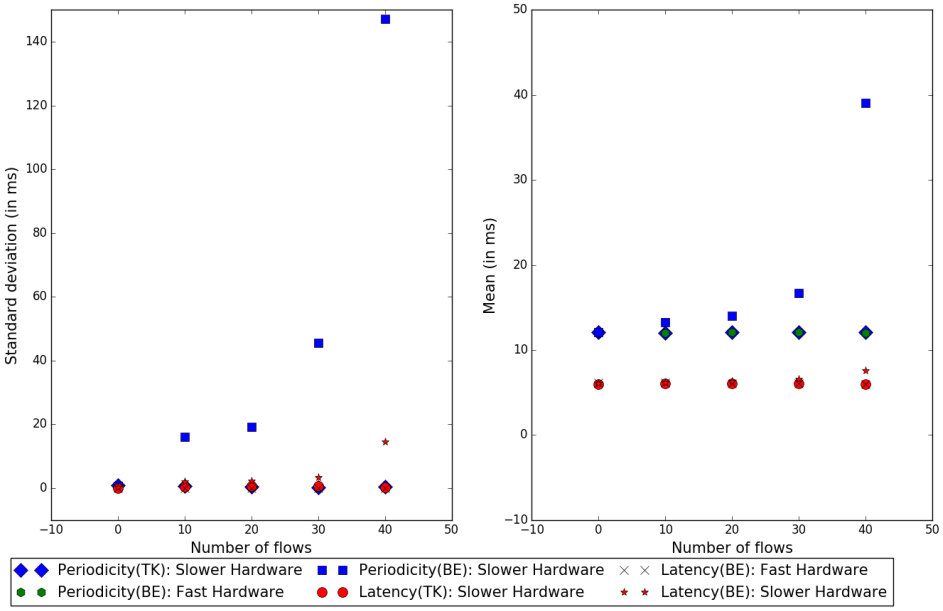


Fig. 4: The observed standard deviation and means of the periodicity and latency with and without TimeKeeper. The expected mean latency is 6 ms and periodicity is 12 ms.

models a smart grid control network which has a single SCADA master and a few SCADA slaves which not only communicate with the master and other entities of the control center but also run some computation before sending data.

The experiment was performed with two machines: (1) a constrained Lenovo Laptop with 6 usable cores and 16 GB RAM also referred to as **Slow Hardware** and (2) a powerful

Xeon server with 56 cores clocked at 2.0 GHz with 64 GB of RAM also referred to as **Fast Hardware**. Figure 4 shows the mean and standard deviation of the DNP3 transaction latency and polling periodicity of the packets captured on link $s_1 - s_3$ under three configurations: Best-Effort (BE) emulation on slow and fast hardware machines and emulation with TimeKeeper (TK) on the slow hardware machine. The number of background UDP flows was increased monotonically in each configuration. The results indicate that when the experiment is run without TimeKeeper on a slow hardware machine and is tasked with synthesizing a large number of background flows, both latency and periodicity experience higher mean and standard deviation. This is because the processes are no longer scheduled on time as they should. However, when TimeKeeper is used on the slow machine, the means and standard deviations are close to the corresponding observations made on the fast hardware machine. It supports our claim that by removing the notion of shared time across emulated processes, TimeKeeper decouples experiment outcomes from the amount of available computational resources.

7. CASE STUDY-3: EMANE

The Extensible Modile Adhoc Network (EMANE) Emulator, developed by [nrl, 2010], is a widely used ad-hoc wireless simulator. EMANE can be used as a standalone package or it can be integrated with CORE and other network simulators to enable robust simulation of wireless links. In this section, we give a brief overview of EMANE and describe the integration of TimeKeeper with the standalone EMANE package.

7.1. EMANE Overview

EMANE provides Network Emulation Modules (NEMs) to emulate data-link and physical layer models. EMANE provides models of data-link layer models such as IEEE 802.11a/b/g and RF-Pipe. The default physical layer model referred to as the Universal Physical Layer is responsible for emulating several effects of wireless links such as fading, collisions, interference and noise. In EMANE's implementation (using LXC containers), the physical layer adds a header to each outgoing packet and every outgoing packet is broadcast to every other node in the network. On receiving a packet a node examines the header to determine if the packet should be passed up the stack or dropped. The fields in the header include information such as the sender's location (which can be used to estimate signal strength), sender frequency and the transmit time from which the propagation delay is calculated.

7.2. Integration with TimeKeeper

Integration of EMANE with TimeKeeper required no modifications to EMANE source code. We developed an user interface in python to allow the experimenter to easily specify the dilation factors for each node, model parameters, initial physical locations for every node and specify the applications to run on each LXC. The interfacing code launches EMANE in a distributed deployment configuration. Initial locations specified in the user interface are broadcast to all participating nodes at the start of the experiment. The application is responsible for simulating node movement through sharing of so-called location events.

Channel delays in EMANE are not explicitly specified. Instead, propagation delays are calculated inside the emulator based on the distance between the transmitting and receiving node. Similarly, the transmission delay is calculated based on the transmission bandwidth and the received packet size. On receiving a packet, EMANE first estimates when to process a packet by computing the transmission and propagation delays. An internal timer is then scheduled using the linux *timerfd* interface to fire after the computed delay elapses and subsequently, the packet is processed and pushed up the stack.

The challenge with distributed simulation of EMANE (indeed, a typical challenge for distributed simulation of ad-hoc networks in general) is that generation of message traffic is unpredictable, and the propagation delay of a wireless message is very small. In the

TimeKeeper context, temporal accuracy means using a synchronization window for each round that is small enough so that a packet is not both sent and received in the same round. A lower bound on the time between when the first bit of a packet is transmitted and when the last bit is received is the minimum packet size (in bits) divided by the channel bandwidth (in bits per second). As these quantities are buried within EMANE, we do ask the user to provide them. The TimeKeeper window size W is set to this product.

7.3. Experiments

We use EMANE to illustrate how lack of synchrony in virtual time can cause a best-effort approach to exhibit behaviors that are likely very far from those in the actual system. We accomplish this by emulating a system of wireless routers, in particular the OLSRD [ols, 2015] routing module available for use in EMANE. OLSRD is a link state based routing protocol in which neighbouring nodes exchange periodic keep-alive messages. Keep-alive messages carry link state and the sender's view of the overall topology. Through these messages each node constructs an overall view of the topology and determines the best path to each destination. The protocol implementation allows specification of a periodic keep-alive exchange interval and duration of time for which the link-state information for a particular node is valid. If a node does not receive a routing update from a neighbor within the validity period, it considers the link between them to be broken. This of course impacts the routes it subsequently chooses. Thus we see that the behavior of the emulation depends on the network simulation delivering keep-alive messages to the emulation in a timely fashion with respect to the emulation's view of time.

In our mininet case study we showed that under best-effort management we could induce delayed recognition of timer signals by including backgrounds processes in the emulation. We use the same approach here, recognizing that delayed recognition of keep-alive messages will cause the router nodes to have a different view of the topology, and hence compute different routes. Dropped links can also induce dropped packets, when no known route yet exists as a result.

All the EMANE experiments were conducted on an Intel i-7, 2.4 GHz Lenovo Laptop with 16 GB RAM. The experiments were restricted to use only two cores out of the 8 cores available on the system. The evaluation topology is a linear chain of nodes, with each node able to communicate with the nearest three neighbors on either side of it in the chain. We used EMANE's RF-Pipe model with an assumed bandwidth of 1Mbps. A node at one end of the chain contains an application client process that sends UDP packets to a server at the other end of the chain. The intermediate nodes route these messages using routes computed continuously by OLSRD. We place stress on EMANE's ability to deliver keep-alive messages between nodes by placing on alternate nodes a background process that runs at high priority. High priority processes are allotted larger time quanta by the Linux scheduler and so provide a mechanism by which we can interfere with the best effort delivery of keep-alive messages, by starving the OLSRD processes the ability to run and send those messages, or run and receive them. We can furthermore increase the stress by increasing the size of the topology run on a fixed number of cores; this has the same effect of withholding CPU resources from OLSRD processes that need them to keep up with the flow of wallclock time. We used a dual core system for the evaluation.

Our experiments used EMANE parameters which ensured that packet losses were not caused explicitly by the RF-Pipe model. The data rates were low enough so that in a real system packet loss would be very very infrequent. Under these operating conditions, we found the stressing technique to be very effective. In the best-effort experiments we observed significant packet loss rates (and hence reduction in throughput) as we increased the stress by increasing the size of the topology, see Figure 6, where the independent axis is the number of LXCs assigned to each of two cores. If a routing update does not arrive within a timeout, the routing daemon flushes its routing tables and all packets which arrive

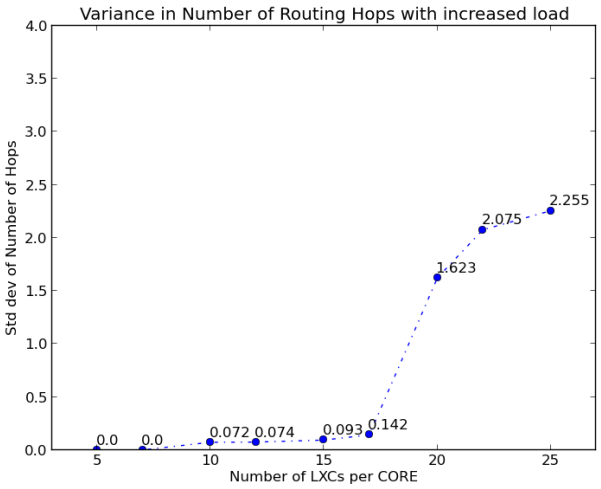


Fig. 5: EMANE: Variance in number of routing hops in best-effort emulation

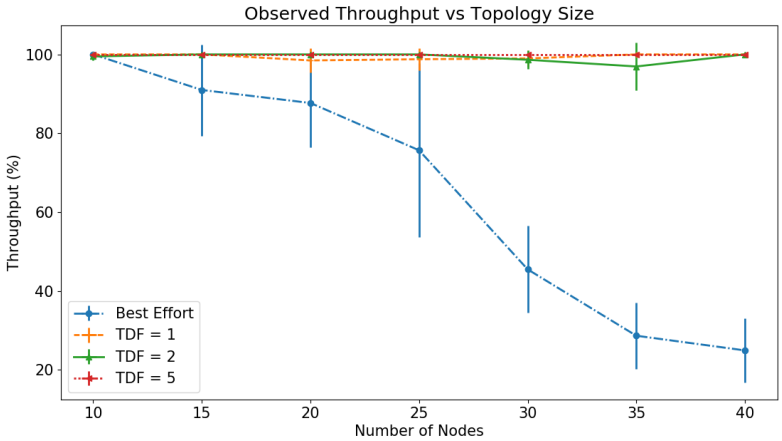


Fig. 6: EMANE: Comparison of throughput for best-effort emulation and TimeKeeper controlled experiments as the topology sizes and TDFs are varied

without available routes are dropped. Other packets could experience longer routes because of unstable routing tables. We see in Figure 5 that there is considerable variance in the number of hops per end-to-end route as the frequency of routing table oscillation increases. In the real system there is essentially no variance.

Figure 6 compares the observed throughput as the size of the topology (number of LXCs per core) is increased. We see that best effort emulation struggles to cope with the increased load and starts to lose packets. The TimeKeeper controlled experiment is largely unaffected by the imposed stress or the TDF and manages to deliver nearly of the injected packets. Figure 7 compares the observed end to end transmission time of the transmitted packets as the topology size is increased. We notice that the end to end latencies are slightly higher in

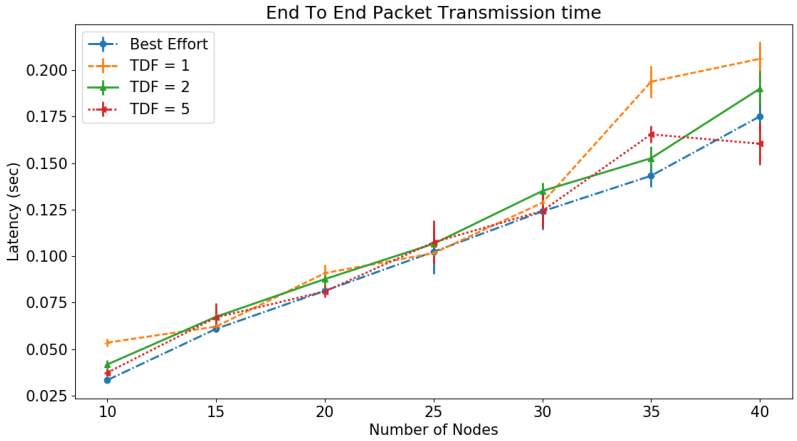


Fig. 7: EMANE: Comparison of end to end latencies for best-effort emulation and TimeKeeper controlled experiments as the topology sizes and TDFs are varied

TimeKeeper controlled experiments because processes within each container are scheduled in round robin fashion which is less responsive than sophisticated scheduling strategies used by the Linux scheduler. However, the increase in end to end latencies shows the same trend as Best-Effort emulation and is largely invariant to TDF as expected.

8. CASE STUDY-4: NS-3

ns-3 is a very popular discrete-event network simulator, designed primarily for research and educational use. It consists of numerous 'network' models, such as LANs and Wi-Fi. ns-3 is able to do simulation-in-the-loop experiments where a simulation model interacts with code executing within a Linux LXC container, using what is called its Realtime Scheduler option. We will show the utility of integrating TimeKeeper with ns-3 when the ns-3 model interacts with LXCs. As we have seen with other systems, we accomplish this without changing ns-3.

8.1. ns-3 Overview

A complete description of ns-3 is out-of-scope. Instead, we will focus on the two components that allow one to connect LXCs to the ns-3 simulator: the TapBridge Model and the RealTime Scheduler. They will be discussed individually.

The **TapBridge Model** was created to allow hosts running inside of Linux LXC containers to interact with an ns-3 simulation. It works by connecting the inputs and outputs of a Linux TAP device [Krasnyansky and Yevmenkin, 2007] with the inputs and outputs of an ns-3 NetDevice. A Linux TAP device allows a user space program to send and receive packets without needing to traverse physical media. The Linux TAP acts as the glue connecting an LXC with the ns-3 simulation. An ns-3 NetDevice is an abstraction which covers the simulated hardware as well as the software driver. When a NetDevice is installed on a ns-3 *Node*, it is able to communicate with other *Nodes* in the simulation. For every LXC we wish to integrate with the ns-3 simulation, the TapBridge Model creates a *Ghost Node*. A *Ghost Node* is simply a *Node* in the ns-3 experiment that represents an external entity (where the upper levels of the network stack are not being simulated). For every *Ghost Node* there is a corresponding NetDevice acting as the connection to the simulation. Every time an LXC sends a packet, the Tap Device brings the packet to the corresponding *Ghost Node*

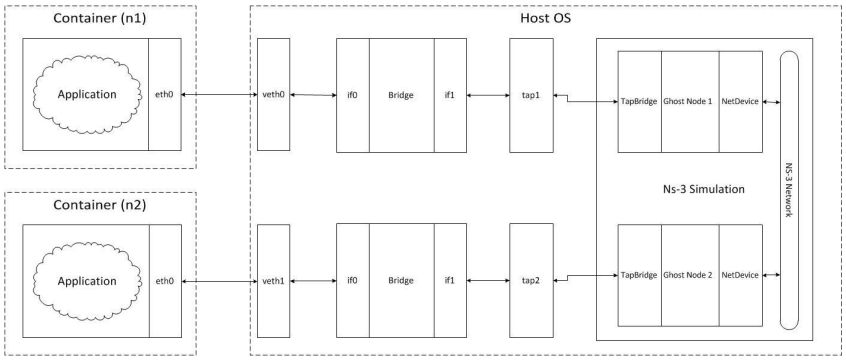


Fig. 8: LXCs and ns-3

in user space, which will push it to the NetDevice. This setup allows an LXC to interact with the ns-3 simulator. See Figure 8.

The **RealTime Scheduler** attempts to keep simulation time synchronized with the wallclock. An event's virtual time stamp is interpreted as a wallclock time. When the simulator is idle and able to execute the next event, that execution is delayed until the wallclock advances to the next event's interpreted time.

It is possible for the simulator to fall behind the wallclock, when there are more events with greater computational workload than can be processed in faster-than-real time. This is easily detected at the instant the simulator is free to execute the next event in the event-list, and finds that the real time which corresponds to the event-time has already past. When this happens, the RealTime Scheduler has two user-selected options: *BestEffort* or *HardLimit*. Under *BestEffort* the simulator will keep running, trying to catch up to the wallclock. The *HardLimit* option terminates the simulation if the difference between the system clock and simulation clock becomes too large. At events that have fallen behind, the difference between the actual wallclock time and the lapsed targeted wallclock time is called *jitter*.

8.2. Integration with TimeKeeper

As described in Section 2, the high degree of transparency in integration of ns-3 with TimeKeeper is due to the fact that ns-3's RealTime Scheduler aligns virtual time with real time using the *gettimeofday()* system call; and under TimeKeeper the value returned is the current virtual time. Time-oriented system calls made by the LXC containers are likewise intercepted and re-interpreted by TimeKeeper.

The benefit of the integration is enjoyed when otherwise the ns-3 model cannot keep up with real time and suffers significant jitter. The simple expedient of putting the model under TimeKeeper administration and using a sufficiently large value of TDF $\alpha > 1$ will apparently keep the wallclock time from racing past the simulation's targeted execution times. Traffic generation in the LXCs based on timers (e.g., launch an http request every 5 seconds) will be slowed down by the same amount. As we saw in the CORE experiments increasing α will increase the execution time, but also frees up system resources available to execute the model.

8.3. Experiments

We designed experiments to illustrate the difference in jitter suffered without TimeKeeper, and with TimeKeeper for a variety of TDF values. We perform the comparison for one model where TimeKeeper-free ns-3 has relatively low jitter and stays below the *HardLimit* threshold, and another model where the jitter for ns-3 is much larger than this threshold.

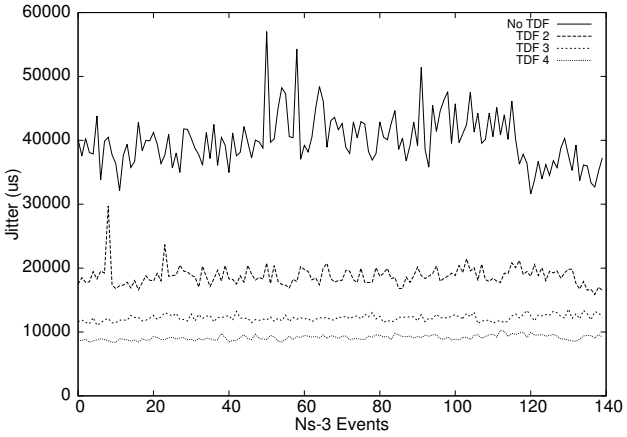


Fig. 9: Measured jitter when simulator is not overloaded

We will see in both cases that under TimeKeeper increasing α decreases the jitter, and in the overloaded case can reduce it below the *HardLimit* threshold.

Like our other experiments the model we use measures network bandwidth by running *iperf* in LXC containers. While in the CORE and mininet experiments the network simulation was contained within the emulator, in this case the ns-3 WiFi model simulates the network, moving packets between the LXC containers.

Unlike the mininet and EMANE experiments, we do not need to introduce background processes to stress the system. The computational difference between two standard network models is enough to illustrate our points. We use ns-3's WiFi model for the low workload case, and ns-3's CSMA model for the high workload case. All of the experiments were executed on a Dell Studio XPS Desktop, with 24 GB of RAM, and 8 Intel Core i-7 CPU X 980's @ 3.33GHz. The machine was running 64-bit Ubuntu with a modified 3.13.1 Linux Kernel.

Figure 9 illustrates the results observed with the low workload case, where the jitter of each event is plotted when the system is simulated without TimeKeeper, and for a variety of TDF values. In all cases the jitter is below the default 100ms *HardLimit*, with an average average jitter in the non time-dilated experiment of 40ms. Under TimeKeeper with TDF values of $\alpha \in 2, 3, 4$ the resulting average jitter is close to $(40/\alpha)$ ms. This agrees with intuition and our earlier CORE experiments.

Workload of the same *iperf* model is increased by using ns-3's CSMA network model, which generates more events per unit virtual time, with different degrees of computational complexity than the WiFi model. Running the same experiments we see that in the stock ns-3 system jitter is sometimes three orders of magnitude larger than the *HardLimit*. An curious point is that the effects of introducing TimeKeeper are non-linear. The difference between a TDF of 1.1 and non-dialated control is a factor of 5. This may be understood in that ns-3 is completely at the mercy of the Linux scheduler, that ns-3 is a compute-heavy process, and whatever scheduling priorities are involved are significantly delaying ns-3 execution. Just the mere act of putting ns-3 under TimeKeeper brings the LXC containers and the ns-3 simulator into temporal alignment and isolates them from the vagrancy of other processes running on the platform. However there is another factor of 10 reduction in magnitude of jitter when the TDF is increased from 1.1 to 3. This non-linearity may be understood considering that the computational load of the network traffic when the simulator is running behind at $\alpha = 1.1$ is larger than when $\alpha = 3$, due to network contention. The two runs

TDF	None	2	3	4	5	10
Runtime	1700ms	738ms	312ms	101ms	30ms	4.25ms

Table I: Average jitter with large ns-3 WiFi model

are not performing the same amount of workload, are not getting the same bandwidth estimates, and are not seeing the same levels of congestion and packet loss.

By increasing α the network is brought to behavior that is closer to what we'd expect a real system to perform, and as it happens, simulating and emulating that behavior takes less computation. With a average jitter of 35ms the *HardLimit* is never approached.

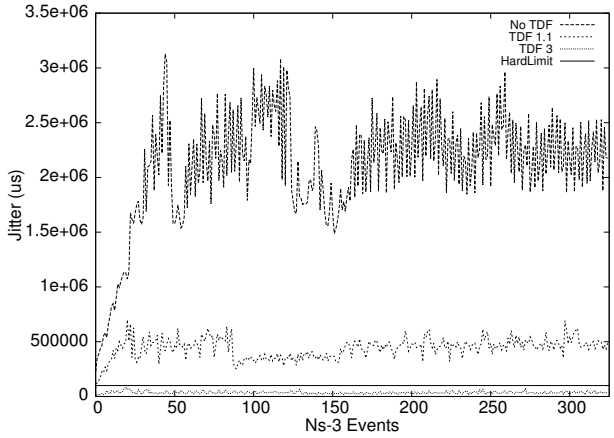


Fig. 10: Measured jitter when simulator is heaviliy loaded

To demonstrate that TimeKeeper reduces jitter also on larger networks we constructed a network topology of 100 ns-3 nodes which create background traffic by communicating over via WiFi 802.11b. We include in this model two LXCs which use *iperf* to measure available bandwidth. We ran this experiment with varying TDFs, and calculated the average jitter. The results are given in Table I.

Without TimeKeeper this model cannot keep up, the jitter is 20 times larger than the default *HardLimit* threshold. Increasing TDF to 5 or larger, jitter remains under the threshold, yielding successful simulation runs.

9. CASE STUDY-5: S3F

9.1. S3F Overview

S3F is a parallelized simulation framework that was based on ten years of experience with the Scalable Simulation Framework (SSF) [Cowie et al., 1999]. A prime objective was to make the system simpler to understand and maintain. S3F eschews use of any libraries other than those which are standard with C++, and it backs away from direct support of process oriented simulation. It shares SSF's notion of Entities, inChannels and outChannels, and communication through those channels.

Timeline is an important concept in S3F; one thinks of it as a construct that advances simulation time. Each timeline has its own simulation clock, and synchronization between timelines is needed to ensure proper execution. Every S3F Entity is aligned to exactly one timeline; entities on the same timeline need no coordination beyond the timeline's simulation event-list. Channels where the inChannel side is bound to a timeline different than that of the entity to which the out-channel side is bound expose communication that must be

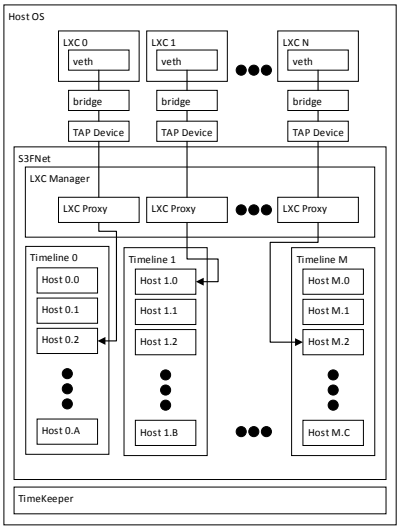


Fig. 11: S3F integrated with LXC containers

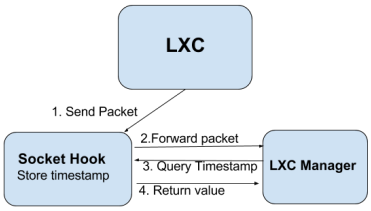


Fig. 12: Socket Hook Monitor

synchronized using parallel simulation time synchronization techniques. S3F requires that any channel which has its inChannel and outChannel in different timelines must have also declared a minimum latency time. This minimum time is a lower bound on the time between when the first bit of a communication enters the channel, and when the entity receiving the message can be affected by the arrival of the message. In network-oriented models minimum latencies are typically derived from network characteristics.

S3F uses *composite synchronization* [Nicol and Liu, 2002b] to coordinate timelines. For the purposes of this paper the important details are that all timelines synchronize periodically at a global barrier scheduled periodically in simulation time, and within a synchronization window timelines may synchronize on a point-to-point basis, using what are called "appointments". Within a synchronization window a timeline will follow a schedule of appointments with other timelines, where at each appointment it notifies its appointment partner that it is ready to execute events at the appointment's time, and then waits for its partner to make that same notification. Inter-timeline communications are flushed through by these appointment messages.

TimeKeeper as used in the four other emulation/simulators discussed so far implemented the global barrier. This technique makes no allowance for tighter coordination between emulation processes within the same round. The TimeKeeper/S3F integration changes that.

9.2. Integration with TimeKeeper

The integration of S3F and OpenVZ limited communication between OpenVZ and S3F to be at barrier synchronizations points. Messages between S3F and OpenVZ were exchanged only at the end of the synchronization window, using application level communication mechanisms. As we have seen in the discussions of ns-3, more sophisticated kernel-level functionality is available to support concurrent generation, delivery, and receipt of traffic between LXC containers and a discrete-event simulator. This observation led us to re-architect integration of emulation with S3F. The new architecture is shown in Figure 11.

The critical point is that within this framework we selectively execute LXC's for varying amounts of real-time, using the S3F synchronization logic to stop and restart an LXC's execution at an S3F-defined appointment time.

The mechanism for trapping LXC traffic and conveying it to S3F, and vice-versa is the same as used in ns-3: the LXC interacts with *eth0*, and linkages involving virtual interfaces, bridges, and TAP devices connect the LXC with another application. The S3F/LXC interface is managed by an entity we call the LXC Manager. This has a list of data structures called LXC Proxies, with one for each managed LXC. As with the SF3/OpenVZ integration and similar to ns-3, the simulation has a node that represents the LXC host, which for consistency here we will also call a *ghost*. One can think of the LXC as being an application that sends traffic to the network simulator and receives traffic from the network simulator. The LXC's ghost has S3F model code for the simulated network at the bottom of the stack, and exchanges packets with the LXC at the highest layer of the stack.

The LXC Manager has a thread which is responsible for noticing when any LXC has done a write, and conveys the packet written to the timeline to which the LXC's ghost is aligned. The receive time will be in the future, and so the action is to insert an event into the timeline's eventlist. Packets bubbling up from an LXC's ghost are written into the LXC proxy's outgoing queue and are written out to the recipient LXC at the packet's receive time. The LXC Manager interacts with the kernel's TimeKeeper module (which is not shown). The LXC Manger can query TimeKeeper for the virtual time of any specific LXC, and can instruct TimeKeeper to cause a stopped LXC to advance a specific interval of time, and stop again. TimeKeeper is implemented with multiple kernel threads, to allow for many timelines to interact with it concurrently (through the LXC Manager module).

In addition to more fine-grained control over LXC executions, our TimeKeeper/S3F integration brings better precision in assigning time-stamps to packets that transition between emulation and simulation. We want for the send-time as seen by the emulator be given to the packet as it joins the simulator, and vice-versa. Our solution leverages the most commonly used system calls to perform networking operations in Linux, socket send/sendto system calls. We implement a socket hook monitor (Fig. 12), which changes the default implementation of these system calls. It examines each transmitted packet and if the transmitting task is under TimeKeeper administration, the Socket Hook Monitor determines which LXC encompasses the transmitting task. This is done by analysing the control group path of the transmitting task. Each LXC is assigned to separate distinct control group to control access to CPU and memory resource allocations. Once the sending LXC is identified, the socket hook monitor buffers the current virtual time of the LXC. Then when the LXC Manager recognizes the packet arrival it is able to acquire the packet's sending timestamp. Experiments establish that the error in sending time-stamp is now under 1 μ-sec, a significant improvement over our initial technique [Lamps et al., 2015]. Accuracy in the time of a packet's *receipt* transition from simulation to emulation is accomplished through synchronization, described more fully below.

With this architecture and an appropriate interface with TimeKeeper, we are able integrate LXC execution behavior more deeply with the virtual time synchronization structure than has ever before been accomplished. We are able to view an S3F entity as something that holds state, that generates messages, that receives messages, and that synchronizes with other entities *regardless of whether that entity's behavior is simulated or emulated.*

The LXC Manager plays the role of an event-list manager for an LXC. When an LXC ghost's timeline knows that it is safe to advance to time t, it asks the LXC Manager to advance the LXC to time t, and then stop. At time t the LXC Manager may deliver a packet to the stopped LXC, or possibly just wait until additional synchronization information makes it clear that it is safe to advance the LXC even further.

We cannot emphasize this contribution strongly enough. Prior to this work, simulators only interacted with emulators by attempting to continuously track observed passage of time (e.g., ns-3 and CORE), or at fixed points in time (e.g., S3F/OpenVZ). Fine-grained synchronization based on directed commands such as "run for 540 μs and then stop" are new. That said, control over LXC execution is not exact. TimeKeeper uses a timer to govern

TDF	Avg Ping RTT (μs)	Max Advance Error (μs)	Avg Advance Error (μs)	Root Mean Squared Advance Error (μs)	Std Dvt Advance Error (μs)
1	3185	43	1.0007	2.0247	1.7601
2	3137	25	1.3404	2.1849	1.7255
5	3136	40	0.6603	1.0467	0.8122
10	3128	4	0.4726	0.7541	0.5876
30	3124	1	0.0013	0.0380	0.0379

Table II: Prediction errors in small experiments as a function of TDF

when it signals an LXC to stop. The precise firing of that timer, and the precise point in the code execution when that LXC responds is non-deterministic. Furthermore, we observe that the magnitude of that non-determinism suggest we not try to control an LXC with time intervals much smaller than 10 wall-clock μs. Accordingly, if the synchronization logic calls for the LXC to advance by δ in virtual time, and if the translation of δ into wallclock time (δ * TDF) is less than $10\mu s$, TimeKeeper does not schedule the timer, it simply adjusts the LXC clock to be correct, and then applies the action.

9.3. Experiments

Our evaluation of TimeKeeper/S3F is focused on errors that the system creates, as a result of the non-determinism in starting and starting LXC containers. Our approach is to set up models with operating conditions under which we can predict the precise values that network "pings" would have in a pure simulator, and compare those with latencies observed under TimeKeeper/S3F.

In the first experiment, we ping between 2 LXCs, measure the latency, and compute the error. The channel delay between the 2 LXCs is 1000 μs and the transmission time is 561 μs, which means the perfect ping time between LXCs should be $(1000 \cdot 2 + 561 \cdot 2) = 3122$ μs. We consider the mean and standard deviation of the error as a function of the TDF. Intuition suggests that the relative error should decrease as TDF increases, because the contribution of a fixed magnitude of error in wallclock time decreases when scaled to virtual time. Table II confirms this intuition. Error decreases significantly with increasing TDF, coming within 2μs of the expected result (3122 μs.) The maximum advance error varies but it is relatively small.

We also performed a larger experiment that incorporates simulated hosts as well as emulated hosts. We modeled a network consisting of 80 entities on 4 timelines, (four instances of SSF's "campus network [Li et al., 2009]) mixing emulated hosts (each with TDF 10) with simulated hosts. All interfaces on the entities simulate a 1 ms propagation delay. Simulated hosts consist of clients deterministically downloading files from servers, using TCP. Similarly, the emulated LXCs all ping a single LXC. The emulated and simulated traffic co-mingle in the network simulator, potentially affecting each other. In order to isolate the variation to be only that caused by TimeKeeper, we ran the same experiment over and over, with deterministic TCP traffic, comparing for each j the latency of the j^{th} ping, across experiments. Figure 13 shows the fluctuation of pings as emulated traffic interacts with deterministic TCP traffic, plotting the average ping duration as a function of the ping index. Note that the variation is less than 200 μs against an average ping latency of $3ms$. Changes in ping RTT from packet to packet can be attributed to variation in the background traffic from instant to instant. However, by looking at the standard deviation among measurements for a specific packet index, all that variation can be attributed to variation introduced by

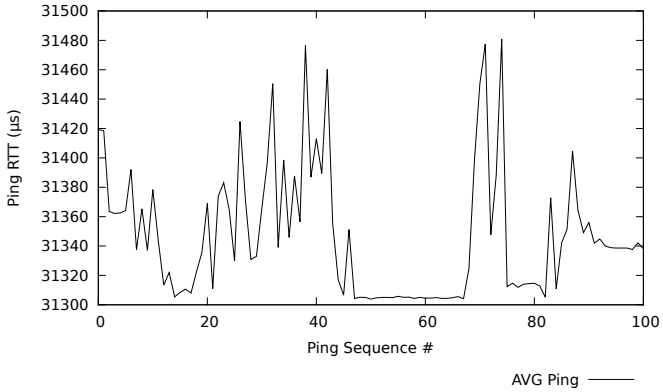


Fig. 13: Average Ping with Simulated and Emulated Traffic

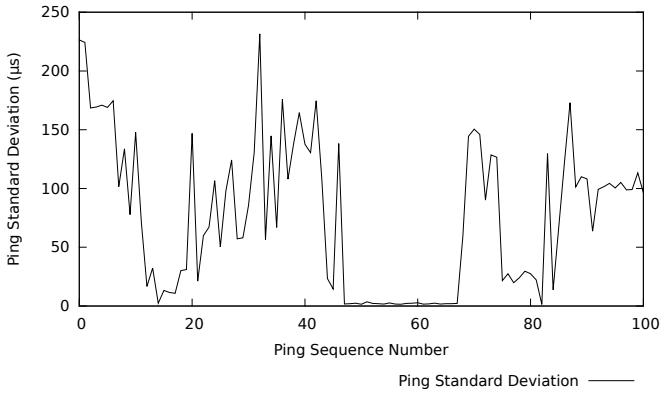


Fig. 14: Standard Deviation of Ping with Simulated and Emulated Traffic

the emulation control mechanisms (Figure 14.) With deterministically replayed background and ping traffic a pure simulator would show *no* variation in RTT for each packet. Hence such variation as we measure is due to TimeKeeper. The main value of this data is to highlight the magnitude of variation in behavior that, without more precise control over LXC executions, may be unavoidable in combined emulation/simulation models. However, the variation shown here is very small, less than 1%.

10. CONCLUSION

We have shown that emulation can be integrated with diverse network simulators running on Linux multiprocessors, using a kernel module called TimeKeeper. To emphasize the generality of the approach, we have shown how to bring TimeKeeper's capabilities to five simulation/emulation systems: CORE, mininet, EMANE, ns-3, and S3F. In each case TimeKeeper brings new capabilities to integrated models. It makes CORE, mininet, and EMANE behavior more accurate by synchronizing the execution of its emulation modules in virtual time, something they did not do before. It makes ns-3 able to simulate/emulate larger models than it currently can, by moving the common basis for coordination between ns-3 and emulation from the wallclock to the potentially slower virtual clock. The integration with S3F showed for the first time how to incorporate fine-grained control of

emulated hosts within a parallel simulation synchronization algorithm. From the point of view of synchronization, it removes the distinction between emulated and simulated hosts. We leveraged this capability in [Babu and Nicol, 2016] which used TimeKeeper's integration with S3F for high fidelity emulation of Programmable logic controller (PLC) networks.

There is no free lunch however. Execution of emulation processes are controlled by timers whose firings cause a signal to be sent to the process to stop. There is inescapable variation in the behavior of a timer—telling it to fire in T units of time will result in it firing in $T+\epsilon$ units of time, where ϵ will vary with every execution. There is inescapable variation in the time needed before a running process recognizes a signal. Furthermore, with the current design of TimeKeeper, round robin scheduling mechanism is employed for scheduling processes within a container and this may not be effective for interactive processes.

Integrated emulation and simulation is just another modeling technique. Modeling assumptions must be made, inaccuracies and uncertainties must be tolerated. Nevertheless, experiments we provide show that temporal integration of emulation/simulations using TimeKeeper behave as would be hoped for, and illustrate the magnitude of the variation possible without temporal integration. We believe TimeKeeper will be a valuable tool for bringing emulation capabilities to other simulators, and to bring virtual time to other Linux applications.

Acknowledgements

This material is based upon work supported in part by the Boeing Corporation, in part by the Department of Energy under Award Numbers DE-OE0000097 and DE-OE0000780 and in part by the Maryland Procurement Office under Contract No. H98230- 14-C-0141. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

REFERENCES

(2010). Emane - u.s naval research laboratory. http://www.nrl.navy.mil/itd/ncs/products/emane.

(2014). Openvz: a container-based virtualization for linux. http://openvz.org/Main_Page.

(2015). mininet: An instant virtual network on your laptop. http://mininet.org/.

(2015). Optimized link state routing. http://www.olsr.org/mediawiki/index.php/Main_Page.

(2015). tcpdump. http://www.tcpdump.org.

(2015). wireshark: A network protocol analyzer. http://wireshark.org.

(2016). netem. https://wiki.linuxfoundation.org/networking/netem.

(2017). Qemu: The fast processor emulator. http://www.qemu.org.

Ahrenholz, J., Danilov, C., Henderson, T. R., and Kim, J. H. (2008). Core: A real-time network emulator. In *Military Communications Conference, 2008. MILCOM 2008. IEEE*, pages 1–7. IEEE.

Babu, V. and Nicol, D. M. (2016). Emulation/simulation of plc networks with the s3f network simulator. In *Winter Simulation Conference (WSC), 2016*, pages 1475–1486. IEEE.

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating System Principles*.

Chertov, R., Fahmy, S., and Shroff, N. B. (2009). Fidelity of network simulation and emulation: A case study of tcp-targeted denial of service attacks. *ACM Trans. Model. Comput. Simul.*, 19(1):4:1–4:29.

Cowie, J., Nicol, D., and Ogielski, A. (1999). Modeling the global internet. *IEEE Computing in Science and Engineering*, 1(1):42–50.

Dickens, P., Nicol, D., and Heidelberger, P. (1996). Parallelized direct execution simulation of message passing programs. *IEEE Transactions on Parallel and Distributed Systems*, 7(10):1090–1105.

Erazo, M. A., Li, Y., and Liu, J. (2009). Sveet! a scalable virtualized evaluation environment for tcp. In *Testbeds and Research Infrastructures for the Development of Networks & Communities and Workshops, 2009. TridentCom 2009. 5th International Conference on*, pages 1–10. IEEE.

Erazo, M. A., Rong, R., and Liu, J. (2015). Symbiotic network simulation and emulation. *ACM Trans. Model. Comput. Simul.*, 26(1):2:1–2:25.

Grau, A., Maier, S., Herrmann, K., and Rothermel, K. (2008). Time jails: A hybrid approach to scalable network emulation. In *Principles of Advanced and Distributed Simulation, 2008. PADS'08. 22nd Workshop on*, pages 7–14. IEEE.

Gupta, D., Vishwanath, K. V., McNett, M., Vahdat, A., Yocum, K., Snoeren, A., and Voelker, G. M. (2011). Diecast: Testing distributed systems with an accurate scale model. *ACM Transactions on Computer Systems (TOCS)*, 29(2):4.

Gupta, D., Yocum, K., McNett, M., Snoeren, A. C., Vahdat, A., and Voelker, G. M. (2005). To infinity and beyond: time warped network emulation. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–2. ACM.

Henderson, T. R., Roy, S., Floyd, S., and Riley, G. F. (2006). ns-3 project goals. In *Proceeding from the 2006 workshop on ns-2: the IP network simulator*, page 13. ACM.

Jin, D. and Nicol, D. M. (2015). Parallel simulation and virtual-machine-based emulation of software-defined networks. *ACM Trans. Model. Comput. Simul.*, 26(1):8:1–8:27.

Krasnyansky, M. and Yevmenkin, M. (2007). Universal tun/tap device driver. *URL: http://www. kernel. org/pub/linux/kernel/, FILE: people/marcelo/linux-2.4/Documentation/networking/tuntap. txt*.

Lamps, J., Adam, V., Nicol, D. M., 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*, SIGSIM PADS '15, pages 113–124, New York, NY, USA. ACM.

Lamps, J., Nicol, D. M., 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*, pages 179–186. ACM.

Lee, H. W., Thuente, D., and Sichitiu, M. L. (2014). Integrated simulation and emulation using adaptive time dilation. In *Proceedings of the 2nd ACM SIGSIM/PADS conference on Principles of advanced discrete simulation*, pages 167–178. ACM.

Li, Y., Liu, J., and Rangaswami, R. (2009). Real-time network simulation support for scalable routing experiments. *International Journal of Simulation and Process Modelling*, 5(2):146–156.

Nicol, D. and Liu, J. (2002a). Composite synchronization for parallel discrete event simulation. *IEEE Transactions on Parallel and Distributed Systems*, 13(5):433–446.

Nicol, D. and Liu, J. (2002b). Composite synchronization in parallel discrete-event simulation. *Parallel and Distributed Systems, IEEE Transactions on*, 13(5):433–446.

Nicol, D. M., Jin, D., and Zheng, Y. (2011). S3F: The scalable simulation framework revisited. In *Proceedings of the Winter Simulation Conference*, pages 3288–3299. Winter Simulation Conference.

Reinhardt, S. K., Hill, M. D., Larus, J. R., Lebeck, A. R., Lewis, J. C., and Wood, D. A. (1993). The wisconsin wind tunnel: Virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '93, pages 48–60, New York, NY, USA. ACM.

Rosenblum, M. (1999). VmwareâĂŹs virtual platformâĎć. In *Proceedings of hot chips*, volume 1999, pages 185–196.

Vahdat, A., Yocum, K., Walsh, K., Mahadevan, P., Kostić, D., Chase, J., and Becker, D. (2002). Scalability and accuracy in a large-scale network emulator. *ACM SIGOPS Operating Systems Review*, 36(SI):271–284.

Watson, J. (2008). Virtualbox: bits and bytes masquerading as machines. *Linux Journal*, 2008(166):1.

Wei, S., Ko, C., Mirkovic, J., and Hussain, A. (2009). Tools for worm experimentation on the deter testbed. In *2009 5th International Conference on Testbeds and Research Infrastructures for the Development of Networks Communities and Workshops*, pages 1–10.

Weingärtner, E., Schmidt, F., Vom Lehn, H., Heer, T., and Wehrle, K. (2011). Slicetime: A platform for scalable and accurate network emulation. In *NSDI*.

White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. (2002). An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Review*, 36(SI):255–270.

Yan, J. and Jin, D. (2017). A lightweight container-based virtual time system for software-defined network emulation. *Journal of Simulation*, 11(3):253–266.

Zheng, Y., Nicol, D., Jin, D., and Tanaka, N. (2012a). A virtual time system for virtualization-based network emulations and simulations. *Journal of Simulation*, 6(3):205–213.

Zheng, Y., Nicol, D. M., Jin, D., and Tanaka, N. (2012b). A virtual time system for virtualization-based network emulations and simulations. *Journal of Simulation*, 6(3):205–213.