# Conjoining Emulation and Network Simulators on Linux Multiprocessors

Jereme Lamps, Vladimir Adam, David M. Nicol, Matthew Caesar
University of Illinois at Urbana-Champaign
{lamps1, adam4, dmnicol, caesar}@illinois.edu

## ABSTRACT

Conjoinment 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, ns-3, and S3F. We describe for each of these the modifications made to the tools to support this integration, and examine experiments designed to assess the accuracy of the combined models. Timekeeper permits much tighter sychronization of emulation and simulation than has ever been achieved before.

## 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*

## General Terms

Experimentation

## Keywords

Simulation, Emulation, LXCs, Virtualization, Time Dilation, CORE, ns-3, S3F, 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 model building is that it enables high fidelity execution behavior and traffic generation. However, that fidelity comes with a computational cost. Simulation usually requires fewer resources per unit model, achieved by abstraction. The use of emulation need not exclude simulation, or vice-versa, but to use them simultaneously suggests that the emulated portions of a model operate within the same virtual time coordinates as does the simulated portion. The research applications that motivate our interest call for models with large numbers of simulated and emulated entities, with fairly tight synchrony between them. This drives us towards the lightweight end of the emulation spectrum where as much as possible is shared between co-hosted virtual machines, with mechanisms for tightly integrating the simulation and emulation. Conjoining lightweight emulation and simulation within virtual time, with support for close synchronization, is our objective.

This paper is by no means the first foray into this space. An early effort was motivated by the objective of testing distributed applications [14]. 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 $\alpha$ is consider to run $\alpha$ 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[5] emulation platform and the S3F[20] simulator were integrated and studied [27]. 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 $\alpha$ that modeled the duration of an uninterrupted execution burst that took $t$ units of measured time as a duration of $t/\alpha$ units of virtual time. A key differentiator between this work and prior art was that viewed from a wallclock 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 $\Delta$, stop, and exchange traffic with the S3F simulator, which alternatively is also run for $\Delta$ units of simulation time.

Follow-on work produced *TimeKeeper*[17], which brings virtual time to general Linux processes using a small modi-


*SIGSIM-PADS'15,* June 10–12, 2015, London, United Kingdom.
ACM 978-1-4503-3557-7/15/06.
http://dx.doi.org/10.1145/2769458.2769481.

fication 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 use something other than OpenVZ for conjoined emulation and simulation.

In the present paper we take TimeKeeper and show how to integrate it with three different simulators: CORE [7], ns-3 [15], and S3F [20]. The integration with ns-3 and S3F use in particular Linux LXC containers as the emulation envelopes. This work is in the spirit of conjoining OpenVZ and S3F, but is significantly different in that it shows how the generality of TimeKeeper support three different systems, in three different ways. We study the behavior of models using these systems, and assess the realism of the behaviors that are produced. We find that the limiting factor is the granularity of the Linux timer, in the following sense. TimeKeeper schedules a process to run for $\delta$ units of wall-clock time as measured by the system clock. The process will run for $\delta + \epsilon$ time, the variation being introduced by operating system overhead. TimeKeeper's ability to control virtual time advancement depends on the magnitude of $\epsilon$ relative to the magnitude of $\delta$.

We will show that the integration of TimeKeeper with CORE and with ns-3 is trivial. Integration of TimeKeeper with S3F is likewise trivial if we use the same approach as did the integration of S3F and OpenVZ. Recall however that in this approach there is no interaction between emulated hosts and simulated network while either system is running. We want to explore a tighter coupling where we place the advancement of LXC containers under control of the S3F composite synchronization method. In this approach, from the S3F synchronization point-of-view, an LXC container 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.

The take-away message of this paper is that TimeKeeper can be integrated with various discrete-event simulations, and that in each of the three instances we consider it enables those conjoined systems to model systems with more accuracy than they could before, and/or model larger systems than they could before. This point is important. The applications that motivate our research are embedded real-time systems. We are building a cyber-physical system testbed that brings together actual devices, emulations of large numbers of devices (such as smart meters in the power grid), and simulation of communication networks, including wireless, local area wireline, and wide area wireline. The issues of interest include (a) what are the performance characteristics of this network, e.g., latency, throughput, and packet loss; (b) does the system being modeled still meet real-time constraints as we introduce new technologies to improve security? Our earlier work using OpenVZ forced a granularity of interaction between emulation and simulation where the approximations used for introducing traffic from one system to the other were of a scale larger than interlink latencies seen in actual systems. In order to have the highest possible confidence that the results produced by our testbed reflect those that will be observed in the field, we seek ways to more tightly integrate in virtual time the emulated processes and the simulated processes.

# 2. TIMEKEEPER

TimeKeeper [17] is a kernel module for Linux that brings the notion of virtual time to Linux processes in general, and LXC containers in particular. TimeKeeper affects the scheduling of processes under its control, and the notion of "time" these processes experience. TimeKeeper modifies the value returned to one of its processes when it calls *getTimeOfDay* to be virtual time. The value returned is a function of the the time dilation factor (TDF) TimeKeeper ascribes to it, and the amount of real execution time that has been allocated and used by the process.

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.

# 3. RELATED WORK

The related work falls into one of three categories: virtualization methods, simulation/emulation, and other virtual time systems. Each category will be discussed individually.

## 3.1 Virtualization Methods

Virtualization grants the ability to allocate system resources such as memory or computing power into separate Virtual Machines (VMs). Each VM will consider itself to be a unique entity, although it runs on the same hardware as the host machine. There are three levels of virtualization: full virtualization, para virtualization, and OS level virtualization. Full virtualization solutions such as VMware [21] and VirtualBox [6] completely abstract out the underlying physical system without needing to modify the guest OS. This solution has the most overhead, but allows for complete isolation, as well as the ability to concurrently run many different operating systems. Para virtualization solutions such as Xen [9] modify the guest VM's kernel to allow direct communication to the host machine. Finally, OS level solutions such as OpenVZ [5] and Linux Containers [2] require all guest VMs to share the same kernel as the host machine. This scenario has the least amount of overhead, but it comes at the cost of flexibility, as you are not able to host VMs with different kernels.

## 3.2 Simulation/Emulation

In a simulation, the entire experiment is modeled in software. It is considered an attractive solution in research, as the experiments can easily be scaled and repeated. Many popular network simulators exist today, some of the more

notable ones are J-Sim [3], OMNeT++ [23], ns-2 [1], and ns-3 [15]. J-Sim is developed in Java, and considers each node, link, and protocol to be a different component. Each component in the experiment has a *component contract* describing how the component should behave if a packet were to arrive on a particular port. OMNeT++ and ns-2 are both discrete event network simulators. Previous research was done comparing the two network simulators [25][16][4]. It was determined that OMNeT++'s hierarchical network model made it both easier to use, as well as more scalable, than ns-2's flat model. Finally, there is ns-3, which was developed to succeed in the areas where ns-2 was lacking. It is a completely independent project from ns-2, and aims to be modular and scalable. Previous studies have found ns-3 to be more efficient than both OMNeT++ and ns-2 [25].

In contrast to simulation, emulation utilizes a testbed or physical network to run an experiment. This provides more realism, as physical packets are being sent across an actual network. However, there are two fundamental drawbacks to consider when using emulation. First, scalability may be limited, as your experiment can not be larger than the testbed itself. Second, it may be hard to reproduce the results from an experiment, as some physical conditions may change. Two very popular emulation testbeds are Emulab [26] and PlanetLab [22]. 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. On the other hand, PlanetLab consists of connected machines around the world. PlanetLab is continually growing, with 1343 nodes at 657 sites to date. Unlike Emulab which gives you sole access to the machine, PlanetLab gives you a Linux LXC container on each machine in the experiment. Therefore, you will not be able to run a custom OS. You get the same Linux as every other processon the machine.

Finally, there exist hybrid solutions which attempt to bring the benefits of both simulation and emulation in one package. Some popular hybrid solutions are the Common Open Research Emulator (CORE) [7], ns-3 [15] and S3F [20] with OpenVZ [5]. We integrated TimeKeeper with these simulators, and we will discuss them at greater depths in later sections.

## 3.3 Virtual Time Systems

Bringing virtual time to a computer system is not a new idea, as many recent papers have explored this concept [27, 12, 13, 11, 24, 18]. SVEET! [11] and DieCast [13] 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 considered a performance evaluation testbed and introduces a static TDF which will slow down both the VMs and the simulator. This will help the experiment if it had been previously overloaded. DieCast is able to scale the perceived performance of various hardware components. This is useful if you need 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 the dynamic change of TDFs, instead of a static TDF. Next, TimeKeeper can run multiple LXCs with different TDFs simultaneously, while synchronizing their virtual times. Finally, TimeKeeper supports lightweight LXCs and minor Linux Kernel modifications instead of a Xen-based approach. Our solution is most similar to the work done by Zheng et al. [27], who developed a virtual time system for emulation and simulation using OpenVZ for virtualization. Similar to our solution, they modified certain system calls to change the VM's perception of time. However, there are key differing distinctions. Our solution uses a more lightweight virtualization method, and has demonstrated the ability to run much larger experiments [17].

# 4. CORE

CORE is considered a hybrid emulator/simulator, as it uses lightweight virtualization to emulate the networking stack of end hosts and routers, while simulating the links between these devices. The proceeding sections will give a brief overview of how CORE works under the hood, followed by a list of the required changes to fully integrate TimeKeeper with CORE. The section will conclude with some CORE specific experiments we have conducted.

## 4.1 CORE Overview

CORE uses network namespaces to divide a process into a logically separate networking entity (each process will have its own routing tables, network adapters, and so forth). Internally, LXCs also use network namespaces to achieve the same goal; however, LXCs provide additional features which are not necessary for CORE's functionality. For this reason, CORE bypasses LXCs entirely and uses network namespaces directly.

To illustrate what CORE does behind the scenes, consider a simple 2-router topology. The network will be modeled with CORE's basic on/off wireless connectivity model. If the two routers are close enough (as determined by the distance from one another in the CORE GUI), they will be able to communicate. If the routers are too far away from one another they will not be able to communicate. When an experiment is started, a *vnoded* daemon is spawned for every node in the topology. Each *vnoded* daemon is responsible for creating its own network namespace [8]. To establish the network connectivity, CORE uses a combination of virtual Ethernet pair drivers (*veth*), Linux Bridging, and Ethernet Bridging Tables (*ebtables*). A *veth* is simply a Ethernet-like device, where one end is installed on the host while the other end is installed in the *vnoded* daemon. When a packet is sent to one end device, it will simply come out the other end. The necessary *veths* will then be tied together with Linux bridges. Finally, appropriate *ebtable* rules will be applied to the bridge. The rules will determine if packets should be dropped or not, and are modified as the distances between the routers change. See Figure 1.

## 4.2 TimeKeeper with CORE

Only a few changes needed to be made to the CORE source code to allow TimeKeeper's functionality to be fully integrated with CORE. Specifically, modifications were needed to allow both CORE's GUI and each *vnoded* daemon to communicate with TimeKeeper. It is worth noting CORE does not use LXCs directly; however, TimeKeeper was still easily integrated with CORE. This attests to a flexible API capable of assigning TDFs to not just LXCs, but any process

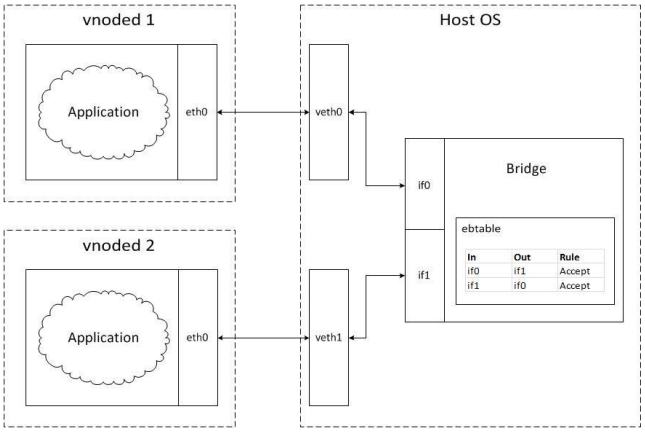


Figure 1: Core Networking Internals

on the Linux system. The necessary changes to the CORE source code are as follows:

1. First, CORE's GUI was modified in order to support additional topology information for each node. If a user double clicks on a node, the option is provided to set that node's TDF. In addition, the user has the ability to set the TDF for every node in the experiment at once, reducing setup time.

2. As mentioned above, when the CORE experiment is started a *vnoded* daemon is created for every host and router. This daemon will create the network namespace via a *clone*() system call. Once the newly created process is spawned in its own network namespace, a message is sent to TimeKeeper to give it the appropriate TDF.

3. After a short time, all of the time-dilated *vnoded* daemons will be initialized, and the CORE experiment will begin. At this point, all *vnoded* daemons will progress in time with respect to their TDFs. From here, the user may tell TimeKeeper to start a synchronized experiment, in which all *vnoded* daemon's virtual times will progress uniformly through time. This was done by adding an additional button to CORE's GUI, which will send the start message to TimeKeeper when it is clicked.

## 4.3 CORE Experiments

Here, we will investigate CORE specific experiments. Unless otherwise specified, 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 is running 64-bit Ubuntu with a modified 3.10.9 Linux Kernel.

The basic idea of the experiments is to show that the conjoined emulation and simulation give the sort of results one expects from the system being modeled. The specific experiment involves measurement of the maximum bandwidth that can be squeezed out of a network using TCP (alternatively, using UDP). The Unix tool *iperf* is executed to make this measurement. We set up within CORE models with clients and servers, running *iperf* to measure the bandwidth achieved with a *simulated* network. The maximum network bandwidth is independent of the speed of the processors attached to it, and so we tested the sensitivity of *iperf* measurements as a function of changing TDF factors in processes used to make and interpret those measurements.

Within TimeKeeper's synchronized experiment, some containers may be frozen for periods of time to allow other container's virtual times to catch up. For example, when one container has a TDF of 1 and another has a TDF of 2, the container with the TDF of 1 will only be given 1/2 the amount of CPU time as the container with a TDF of 2. Thus, it is important for the fidelity of the experiment that this synchronization of the container's virtual time does not interfere with the packet flow of the application, and hence the measurements made by *iperf*. The experiment consisted of a 3-node topology, with one switch, one server, and one client. We used *iperf* to measure the bandwidth between the client node and the server node. 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, and the results are collected in Figures 2 and 3. Figure 2 displays the resulting bandwidth across time with various time dilation factors.

As one expects, regardless of whether the experiment was running in realtime, or much slower than realtime (as in the case where the TDF was 50), the resultant bandwidth was approximately the same. The bandwidth is the same because TimeKeeper is forcing the process to run for the same amount of CPU time, but over a longer period of time. This increases our confidence that a time dilated experiment will give us similar results as if it were run without TimeKeeper in realtime. Next, Figure 3 explores how the TDF affects both CPU utilization and the physical time required to run an experiment. Without a TDF, *iperf* takes approximately 10 seconds to complete, and in that time *iperf* demands 100% of a CPU. The default case is not displayed on the graph in order to show more detail as to what is going on in a synchronized experiment. As the TDF of an experiment increases, the amount of time a process spends on the CPU is throttled back; however, this comes at the cost of increased experiment runtime. As Figure 3 demonstrates, when the TDF of the experiment was 50, the *iperf* process spends only roughly 2% of the time on the CPU. However, the time required to actually run *iperf* for a 10 second long analysis now took 500 seconds to complete. This tradeoff is beneficial and being able to advance time more slowly will allow us to run more complex experiments, which is further explored in Section 5.3.

We extended this experiment by more processes, to convince ourselves that TimeKeeper will still function properly even while synchronizing the virtual times of a large number of containers. We spawned an additional 100 containers, each running a script that sends traffic with randomly chosen destinations. This is done concurrently with *iperf*, and once again the experiment was run with many different TDFs. As before, we found the maximum measured bandwidth to be consistent across all runs. The additional containers did not affect TimeKeeper's ability to maintain a consistent bandwidth, it only caused the experiment to take longer due to the additional overhead. The overall bandwidth of this new experiment was lower than the previous one, but is only due to the extra background traffic saturating the network while *iperf* was executing.

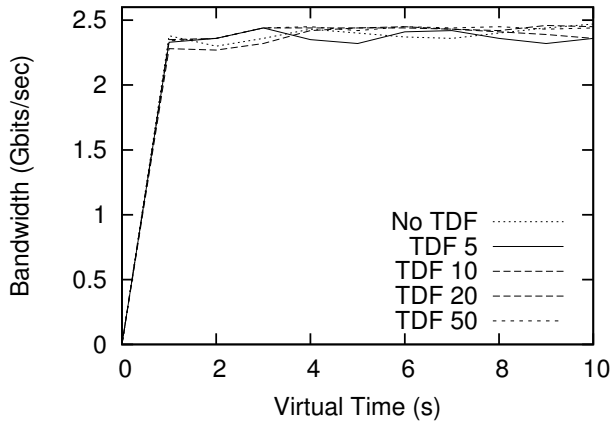


Figure 2: Average Bandwidth Across Various TDFs

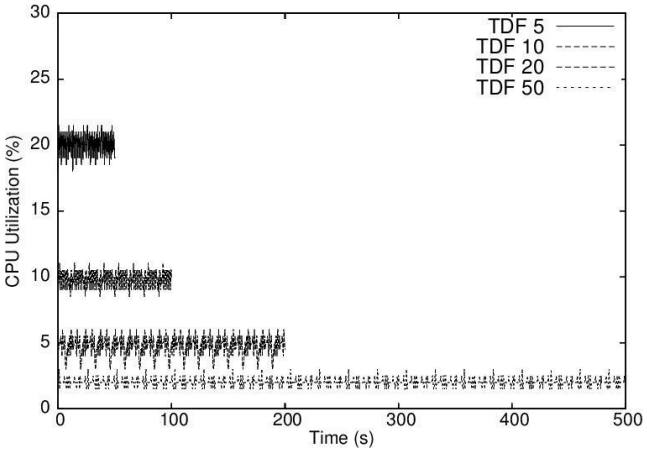


Figure 3: Relationship Between CPU Utilization and Experiment Time

# 5. 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 or Wi-Fi. With respect to TimeKeeper, we are interested in ns-3's ability to interact with real systems, e.g., 'simulation-in-the-loop' experiments. This can be done with ns-3's Realtime Scheduler option. The following sections will give a brief overview of how ns-3 works, followed by necessary changes to support the TimeKeeper integration, and concluding with some ns-3 experiments that were conducted once TimeKeeper was integrated.

## 5.1 ns-3 Overview

Describing all of the components in ns-3 would be out of the scope of this paper. Instead, we will focus on the two components that allow us to connect LXCs to the ns-3 simulator: the TapBridge Model and the RealTime Scheduler. They will be discussed individually.

- **TapBridge Model**
  The TapBridge Model was created to allow real internet hosts (LXCs) to interact with an ns-3 simulation. Essentially, it works by connecting the inputs and outputs of a Linux TAP device 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 will be able to communicate with other *Nodes* in the simulation. For every LXC we wish to integrate with the ns-3 simulation, the TapBridge Model will create 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 will bring the packet to user-space to the corresponding *Ghost Node*, which will push it to the NetDevice. This setup allows an LXC to interact with the ns-3 simulator. See Figure 4.

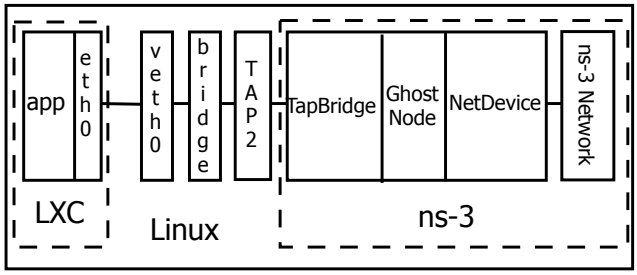


Figure 4: LXCs and ns-3

- **RealTime Scheduler**
  By default, the ns-3 scheduler is not real-time. Rather, when an event is processed, the simulator's time will jump to the time of the next scheduled event. Obviously, this technique will not work if ns-3 is tied to an external entity, as the entity may send a network packet at any time. Thus, the RealTime Scheduler was implemented, which attempts to keep ns-3's simulation time synchronized with respect to an external time source (commonly the wall clock). The RealTime Scheduler works as follows: When the next event in the simulation is ready to be processed, the scheduler will compare the system time with the scheduled time of the event. If the scheduled time of the event is in the future, the simulator will sleep until the system clock catches up to the scheduled time of the event, then execute the event. It is possible for the simulator to fall behind the system clock (if the simulator can not process a sequence of events fast enough). When this happens, the RealTime Scheduler has two options which is set by the user: *BestEffort* or *HardLimit*. If the scheduler is running in *BestEffort* mode, the simulator will repeatedly process events to the best of it's ability in an attempt to catch up to the system clock. However, this may never happen if the packets are continuously sent. The *HardLimit* option will also try to

catch up to the system clock, but will terminate the simulation if the difference between the system clock and simulation clock becomes too large.

## 5.2 TimeKeeper with ns-3

Integrating TimeKeeper with ns-3 was actually surprisingly simple. In fact, no changes were necessary to the ns-3 source code. This is because ns-3's RealTime Scheduler utilizes the *gettimeofday()* system call to determine how far away the simulation time is from the system time. Recall that TimeKeeper implements a modified version of the *gettimeofday()* system call, which will return a modified virtual time based on the TDF of the process. With this knowledge, you can can place ns-3 and all of its LXC containers under control of TimeKeeper scheduling set the TDF of the ns-3 process in order to make it progress more slowly in time (allowing more time to process events) and in synch with the TDF of the LXC containers holding its emulated hosts. Therefore, all that was necessary was to add the ns-3 process and its containers to TimeKeeper's concept of a synchronized experiment and assign them TDF values. When this is done, ns-3's notion of simulation time will progress at the same rate as the other LXCs in the experiment, and keep the LXCs synchronized in virtual time.

## 5.3 ns-3 Experiments

Here we discuss experiments applied to the conjoined ns-3 and TimeKeeper system.

### 5.3.1 Measuring Jitter

In ns-3, *jitter* is defined as the difference between when an event should leave the simulator and when it actually does. With respect to the RealTime Scheduler, minimizing the jitter is important to increase the fidelity of the experiment. When the scheduler is running in *HardLimit* mode, the experiment will abruptly stop if the jitter becomes too large. To investigate how TimeKeeper can help reduce jitter, we ran a series of experiments. First, we created a ns-3 network using the WiFi network model and performed an *iperf* test between two LXCs. Throughout the test we measured the jitter of every single event, and repeated this experiment using different TDFs. This setup will not overload the ns-3 simulator. The results are found in Figure 6. The jitter was always below the default 100ms *HardLimit*. The average jitter in the non time-dilated experiment was 40ms. When the experiment was repeated with TDFs of 2, 3, and 4, the resulting jitters were 18.7ms, 12.2ms, and 9.1ms respectively. This makes sense, as the TDF specifies how long an experiment will take to run, and the average jitter is reduced by the factor of the TDF.

Next, we look at how the jitter is affected when the simulator is overloaded. From the previous CORE experiments, we know a high TDF will slow down the experiment and reduce the stress on the simulator. Therefore, a previously overloaded simulation should be able to complete if it is given a sufficiently large TDF. For the overloaded experiment, we create a ns-3 network using the CSMA network model, and performed an *iperf* test between two LXCs. Once again we measured the jitter of every event, and repeated the experiment with different TDFs. The results are found in Figure 7.

This setup originally overloads the ns-3 simulator, because the CSMA network model tries to provide higher bandwidth

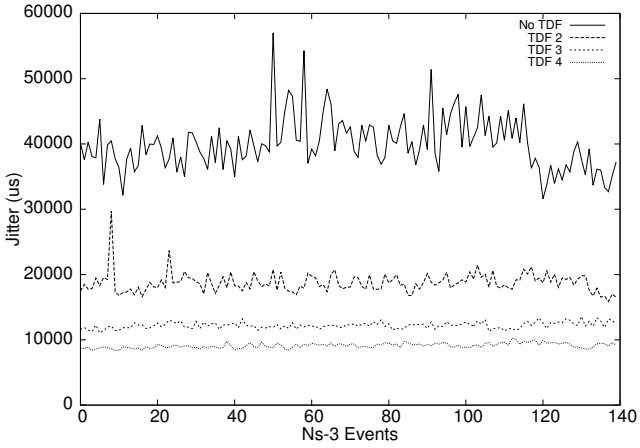


Figure 5: Measuring Jitter When Simulator Is Not Overloaded

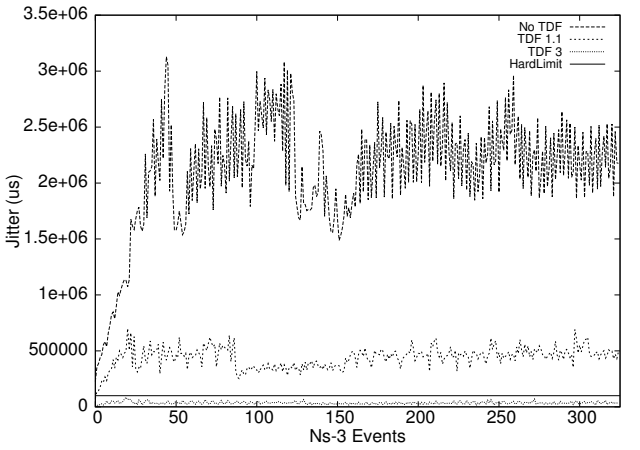


Figure 6: Measuring Jitter When Simulator Is Overloaded

than the WiFi model, and the additional packet events bog down the simulation. When the simulator is overloaded, the average jitter is hurt dramatically. With no TDF, the average jitter was 2,108ms, or 20x greater than the RealTime Scheduler's default *HardLimit*. A little improvement is seen when the experiment is slowed down with a TDF of 1.1, resulting in a average jitter of 441ms. Increasing the experiment's TDF to 3 further reduces the average jitter, this time to 35.5ms. This is a successful experiment, as the jitter never exceeds the *HardLimit*.

### 5.3.2 Increasing ns-3 Complexity

While we demonstrated that increasing the experiment TDF will increase jitter, we wanted to ensure this held true for more complicated ns-3 experiments as well. We constructed a network topology consisting of 100 ns-3 nodes which communicate with one another over the WiFi 802.11b model in order to create background traffic. In addition, we tied in two LXCs who were connected to the same network model, and performed a bandwidth test. This more complicated network was able to overload the simulator, unlike in the previous WiFi example. We ran this experiment with

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

Table 1: Average Jitter with Large ns-3 WiFi Model

varying TDFs, and calculated the average jitter. The results can be found in Table 1.

Similar to the previous experiment, as the TDF of the simulator increased, the average jitter decreased. When the TDF of the simulator was either 5 or 10, the jitter stayed under the default *HardLimit* of the RealTime Scheduler, and would be able to finish successfully. We are able to run more complicated experiments while keeping the jitter low, but this comes at the cost of the overall experiment runtime.

# 6. S3F

## 6.1 S3F and Synchronization

The S3F system was developed after ten years of experience with the Scalable Simulation Framework (SSF) [10]. 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. Every S3F Entity is aligned to exactly one timeline; entities on the same timeline are synchronized with each other using an ordinary discrete event 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 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. Figure 8 illustrates the S3F architecture.

S3F uses *composite synchronization* [19], which works as follows. Given some threshold $\tau$, every cross-timeline channel is categorized as being "fast" or "slow', depending on whether its minimum latency is no greater than $\tau$ (in which case it is fast), or otherwise. Composite synchronization defines a synchronization window of size $\tau$, and does a global barrier among timelines every $\tau$ units of virtual time. No communication that crosses a slow channel will affect the recipient within the same window as the message is sent. This means that the simulation can wait to physically pass that message between timelines until all timelines are stopped at the end-of-window barrier. In contrast, interactions that use fast channels have to be synchronized dynamically during the concurrent execution of timelines within a window. If a channel extends from an entity $E_s$ on one timeline $T_s$ to

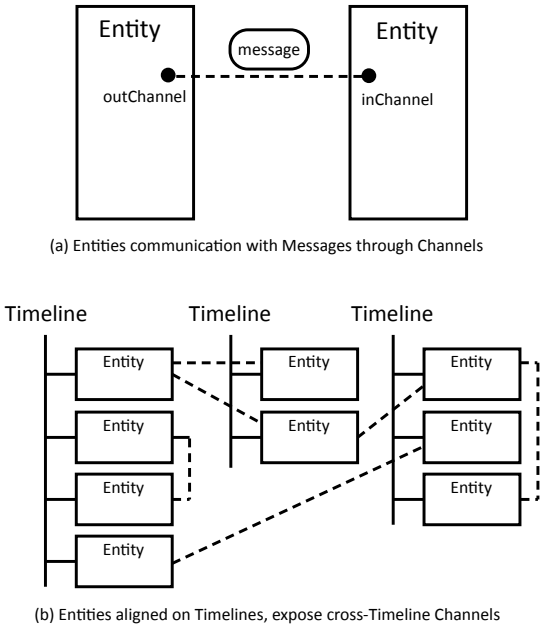


Figure 7: S3F Logical Architecture

an entity $E_d$ on another timeline $T_d$, and this channel has a minimum latency of $\gamma < \tau$, then the two timelines involved synchronize with each every $\gamma$ units of simulation. This means that at times $k\gamma$ (for $k = 1, 2, \ldots$) $T_d$ does not advance beyond time $k\gamma$ until it learns that $T_s$ has reached that time, and has passed to $T_d$ any message it created within virtual time window $[(k-1)\gamma, k\gamma)$. These synchronization points are called *appointments*. The timeline action by $T_s$ associated with indicating to a potential recipient $T_d$ that the timeline's virtual time has reached an appointment time is placed in $T_d$'s event list. Likewise, the action by timeline $T_d$ associated with waiting for a sending timeline $T_d$ to reach the next appointment time is likewise placed in $T_d$'s event list.

Messages received either at appointments or at barriers all have virtual receive-times that are in the future, and so the actions associated with receiving and processing each message can be scheduled in the recipient timeline's event-list.

Clearly the alignment of entities to timelines greatly affects performance. The objective of the alignment algorithm is to balance workload, while keeping entities that share channels with small (or zero) minimal latencies aligned on the same timeline (thereby synchronizing those entities extremely efficiently through an event list rather than a cross-timeline synchronization algorithm). For a given alignment, the choice of $\tau$ can also impact performance. At one extreme one might choose $\tau$ equal to the smallest minimum latency among all cross-timeline channels. In this case every channel is a slow one, and *all* synchronization and message exchange occurs at barrier. Alternatively, one could choose $\tau$ to be so large that every cross-timeline channel is fast, in which case all synchronization and message exchange occurs at appointments. Situations where the former extreme is attractive include when an entity is connected to many many others. In such a case the cost of appointment synchronization is se-

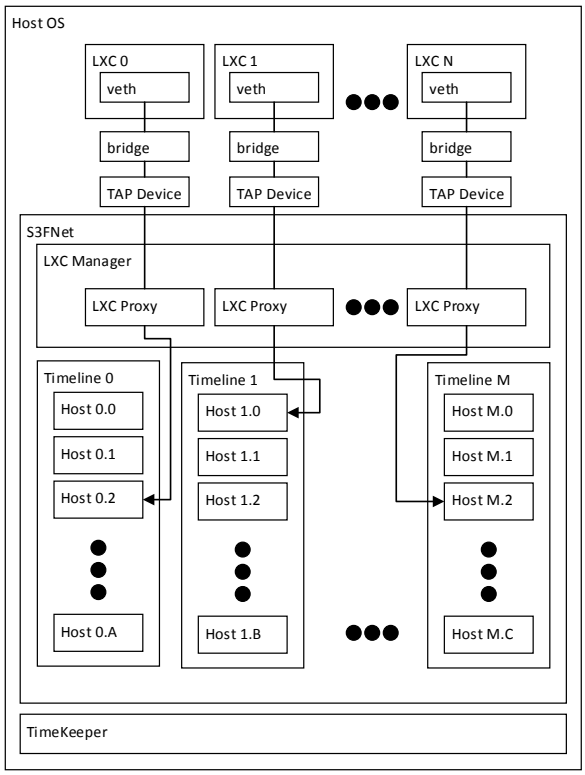


Figure 8: S3F Conjoined with LXC Containers

rialized over the number of channels, but this is avoided by a barrier because the cost of synchronization is insensitive to the communication topology. Situations where the latter case is attractive include when the minimum cross-timeline latency is very small relative to the average cross-timeline latency, which means that overly many barrier synchronizations are called for just to deal with the synchronization requirements of a few channels with small minimum latencies. Implementation of composite synchronization can dynamic alter $\tau$ and search for the sweet spot that gives the best performance [19].

## 6.2 S3F and TimeKeeper

The integration of S3F and OpenVZ limited communication between OpenVZ and S3F to be at barrier synchronizations points; appointments were not used. 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 CORE and ns-3, more sophisticated kernel-level functionality is available (TAPs and bridges) 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 9. Here 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 what looks like a normal interface to it, 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 event list. 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).

The architecture of trapping traffic at the TAP has the benefit of not requiring any modification to the executing code within the LXC, but has a consequence of a certain amount of inaccuracy in virtual time measurement. This occurs because the LXC will write a message to the network interface *and keep executing*, while the virtual time associated with that transmission is measured by the LXC Manager, when it notices the write, at which point it queries the LXC for the the virtual time. Conceptually it would be possible to measure the virtual time at the point of transmission, but this would require further modification to the kernel and to the packet to carry the time-stamp along. This is illustrated in Figure 10. We have determined by instrumentation that the delay between when the packet is sent and when the clock returns a value for the send time is close to 16 $\mu s$. This obviously impacts the granularity of accuracy we can expect, but note that this depends on the time dilation factor, for the lag in *virtual time* is on the order of (16/TDF) $\mu s$. We will see the impact of this in our experiments.

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.

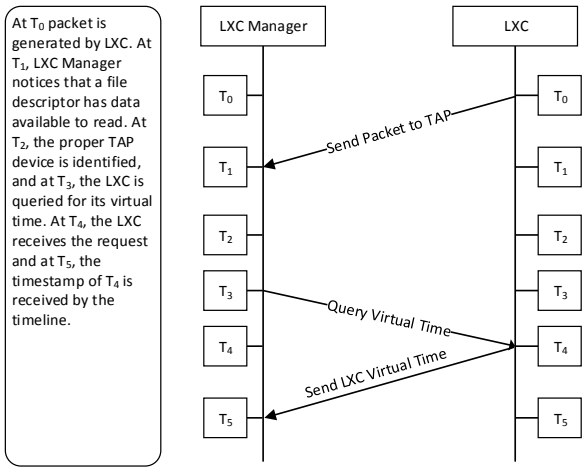


Figure 9: Determining Timestamp of LXC Generated packet.

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 $\mu s$ and then stop" are new. That said, control over LXC execution is not exact. TimeKeeper uses a timer to govern 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 $\mu s$. Our approach includes buffering advancement requests. The LXC's manager has an accumulator of advancement requests. When a request comes in to advance by $\delta$ wallclock time, the accumulated value and $\delta$ are added together. If that sum is less than 10 $\mu s$, then the accumulated value becomes the sum, the LXC's clock is advanced by the virtual time corresponding to $\delta$, but the LXC is not run. If instead that sum exceeds 10 $\mu s$, the LXC is run for a wallclock time equal to the sum, the accumulator is cleared, and when the LXC completes its virtual clock is advanced by the amount to virtual time corresponding to $\delta$ units of wallclock time.

## 6.3 Experiments

We conclude this section by evaluating the accuracy and practicality of S3FNet's integration with TimeKeeper. One experiment highlights the most significant source of inaccuracy, and shows how models with high TDF are largely unaffected by it. The second experiment demonstrates that this integration is capable of handling models with thousands of containers, and measures the overhead as a function of overload model activity.

### 6.3.1 Accuracy

Accuracy experiments were conducted on a desktop running a modified 3.10.9 Linux Kernel with 8 Intel Core i7-2600s @3.40GHz. Since the motivation behind virtual time is to artificially scale performance of LXCs, we explored how varying TDF affects results of a simulation, specifically by bouncing a message between containers. In the first experiment, we repeat the following experiment 3000 times: a container sends a minimally sized UDP packet to the other, which on receipt immediately returns a minimally sized UDP response. The virtual time at the point of transmission is measured by the process running in the LXC, as is the virtual time at the point of receipt. Note that these virtual times are different from the virtual times ascribed to the UDP packets by S3F, because they are made from within the container, and are not the times acquired by the LXC manager. The channel delay between the 2 LXCs is 100 $\mu s$ and the transmission time is 271 $\mu s$, which means the "perfect" round-trip time between LXCs should be $(100 \cdot 2 + 217 \cdot 2) = 634$ $\mu$s.

Table 2 reports statistics from this experiment, varying the time-dilation factor. The reduction in error with increasing TDF is largely (but not exactly) explained by the lag between actual send/receive time and queried send/receive time.

| TDF | Mean RTT | $\rho$ RTT | Mean Error |
|---|---|---|---|
| 1 | 704.57 | 11.5 | 70.57 |
| 5 | 647.03 | 2.5 | 13.03 |
| 10 | 642.21 | 1.9 | 8.21 |
| 20 | 638.86 | 1.5 | 4.86 |
| 30 | 637.66 | 1.2 | 3.66 |
| 40 | 636.70 | 1.1 | 2.70 |
| 50 | 635.95 | 0.8 | 1.95 |
| 100 | 634.54 | 0.7 | 0.54 |

Table 2: Accuracy of simulation with varying TDFs, sampled from 3000 round-trips. Time units are $\mu s$, mean and standard deviation $\rho$ are shown.

### 6.3.2 Large Scale Model

This experiment measures the overhead incurred on a larger network model that fully exercises composite synchronization, and compares the overhead of managing LXCs under conditions of no traffic, and under conditions of heavy traffic. Figure 11 illustrates the topology. The upper half of the figure shows the subnetwork that is aligned to a timeline. It is compromised of 200 hosts that are joined by a router that has a fast (50 $\mu s$) connection to a gateway router, and 200 hosts that are joined by a router that has a slow (1 $ms$) connection to that same router. There are 10 time-

lines, with 10 gateways, with the gateways being joined in a ring. The traffic pattern has randomly chosen pairs of hosts performing the UDP back-and-back experiment described earlier. Under composite synchronization, the fast channels (including those between gateways) use appointments while the traffic that crosses the slow channels are exchanged at barrier synchronization points.

These experiments were run on an enterprise server with 24 Intel Xeon X5650s running at 2.67 GHz, with 24 GB of RAM. For any given experiment we can measure the time spent in LXC management in a timeline by subtracting the measured time executing the emulations from the total running time. We did this in two scenarios, one in which there was no traffic at all generated by the containers, the other in which the back-and-forth UDP messaging was constant. As Table 3 and 4 show, the simulation with traffic has higher overhead, on the order of 10%. This happens because the communication between the LXC Manager and Timekeeper greatly increases as LXCs advance more frequently with smaller intervals. Without traffic, an LXC advances from virtual time $T_1 \rightarrow T_2$ by a single interval. However, with traffic, an LXC may advance (unfreeze and freeze) multiple times when going from $T_1 \rightarrow T_2$, yielding greater overhead.

Two things are revealed by this experiment. First, that S3FNet with Timekeeper support is capable of advancing at least 4000 containers using less than half the resources of an ordinary enterprise server. This experiment was not designed to see where the system breaks, the upper bound is yet to be discovered. The second lesson is that traffic load increases overhead, but not overwhelmingly so. This also speaks to the practicality of the system.

| TL | Total (s) | Emulation (s) | Overhead (%) |
|---|---|---|---|
| 1 | 1200.43 | 1199.52 | 0.076 |
| 2 | 1200.2 | 1199.52 | 0.057 |
| 3 | 1199.93 | 1199.52 | 0.034 |
| 4 | 1200.18 | 1199.52 | 0.055 |
| 5 | 1200.65 | 1199.52 | 0.094 |
| 6 | 1200.44 | 1199.52 | 0.077 |
| 7 | 1200.52 | 1199.52 | 0.083 |
| 8 | 1200.36 | 1199.52 | 0.07 |
| 9 | 1199.95 | 1199.52 | 0.036 |
| 10 | 1200.1 | 1199.52 | 0.048 |

Table 3: 4000 LXCs Aligned on 10 Timelines With No Traffic

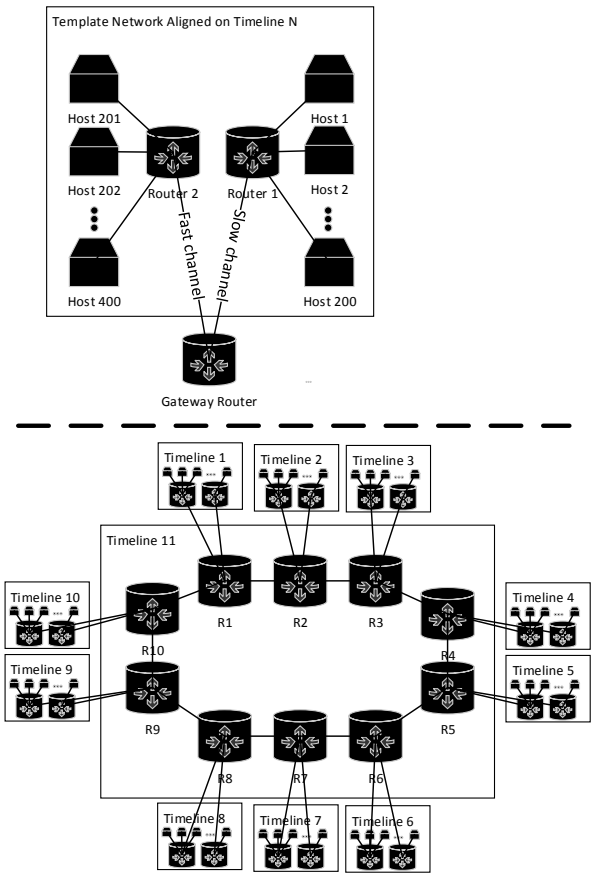


Figure 10: Topology supporting 4000 LXCs

## 7. CONCLUSION

We have shown that emulation can be conjoined with simulation in Linux multiprocessors, using a kernel module TimeKeeper. To emphasize the generality of the approach, we have shown how to bring TimeKeeper's capabilities to three simulation/emulation systems: CORE, ns-3, and S3F. In each case TimeKeeper brings new capabilities to conjoined models. It makes CORE's behavior more accurate by synchronizing the execution of its emulation modules in virtual time, something CORE 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. Finally, it showed for the first time how to integrate 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.

There is no free lunch however. LXC containers are controlled by timers whose firings cause an LXC to halt. 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 $\epsilon$ will vary with every execution. The number of instructions executed by a container within a given interval of time will vary also, depending on things like cache hits/misses, page faults, and other variation induced by the operating system. Likewise, in the current implementation, inaccuracy creeps in because of a difference between when a container sends or receives a packet and when a time-stamp is ascribed to that event. Both causes of inaccuracy are inaccuracies in physical time, and so decrease in *in virtual time* as the time-dilation factor increases.

The experiments we provide show that conjoined emulation/simulations using TimeKeeper behave as would be hoped for, illustrate the magnitude of the the variation due to non-determination, and the overhead of managing many many LXCs concurrently. We believe TimeKeeper will be a valuable tool for bringing emulation capabilities to other

| TL | Total (s) | Emulation (s) | Overhead (%) |
|---|---|---|---|
| 1 | 1353.31 | 1199.52 | 11.364 |
| 2 | 1456.51 | 1199.52 | 17.644 |
| 3 | 1385.29 | 1199.52 | 13.41 |
| 4 | 1318.87 | 1199.52 | 9.049 |
| 5 | 1371.41 | 1199.52 | 12.534 |
| 6 | 1340.93 | 1199.52 | 10.546 |
| 7 | 1377.82 | 1199.52 | 12.941 |
| 8 | 1404.09 | 1199.52 | 14.57 |
| 9 | 1313.71 | 1199.52 | 8.692 |
| 10 | 1314.11 | 1199.52 | 8.72 |

Table 4: 4000 LXCs Aligned on 10 Timelines With Heavy Traffic

simulators, and to bring virtual time to other Linux applications.

## Acknowledgments

This material is based upon work supported in part by the Boeing Corporation, in part by the Department of Energy under Award Number DE-OE0000097, and in part by the Maryland Procurement Office under Contract No. H98230-14-C-0141 [1].

## 8. REFERENCES

[1] The network simulator - ns-2. http://www.isi.edu/nsnam/ns/, 1997. Accessed: 2015-02-03.

[2] Lxc: Linux containers. https://linuxcontainers.org/, 2009. Accessed: 2015-02-30.

[3] J-sim official. https://sites.google.com/site/jsimofficial/, 2014. Accessed: 2015-02-03.

[4] Omnet++ vs ns-2: A comparison. http://ctieware.eng.monash.edu.au/twiki/bin/view/Simulation/OMNeTppComparison, 2014. Accessed: 2015-03-06.

[5] Openvz: a container-based virtualization for linux. http://openvz.org/Main_Page, 2014. Accessed: 2015-01-30.

[1] Disclaimer: Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Maryland Procurement Office. This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

[6] Virtualbox. https://www.virtualbox.org/, 2014. Accessed: 2015-02-15.

[7] J. Ahrenholz, C. Danilov, T. R. Henderson, and J. H. Kim. Core: A real-time network emulator. In *Military Communications Conference, (San Diego, CA, Nov 16-19, 2008)*, pages 1–7. IEEE, 2008.

[8] J. Ahrenholz, T. Goff, and B. Adamson. Integration of the core and emane network emulators. In *MILITARY COMMUNICATIONS CONFERENCE, (Baltimore, MD, Nov 7-10, 2011)*, pages 1870–1875. IEEE, 2011.

[9] 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 *Proceedings of the 19th ACM Symposium on Operating System Principles, (Bolton Landing, New York, Oct 19-22, 2003)*. ACM, 2003.

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

[11] M. A. Erazo, Y. Li, and J. Liu. 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, (Washington, DC, April 6-8, 2009)*, pages 1–10. IEEE, 2009.

[12] A. Grau, S. Maier, K. Herrmann, and K. Rothermel. Time jails: A hybrid approach to scalable network emulation. In *Principles of Advanced and Distributed Simulation, 2008. PADS'08. 22nd Workshop, (Rome, Italy, June 3-6, 2008)*, pages 7–14. IEEE, 2008.

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

[14] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, and G. M. Voelker. To infinity and beyond: time warped network emulation. In *Proceedings of the twentieth ACM symposium on Operating systems principles, (San Jose, CA, May 8-10, 2006)*, pages 1–2. ACM, 2005.

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

[16] M. Koksal. A survey of network simulators supporting wireless networks. In *Middle East Technical University (Ankara, Turkey)*, pages 1–11, 2008.

[17] J. Lamps, D. M. Nicol, and M. Caesar. Timekeeper: a lightweight virtual time system for linux. In *Proceedings of the 2nd ACM SIGSIM/PADS conference on Principles of advanced discrete simulation, (Denver, CO, May 18-21, 2014)*, pages 179–186. ACM, 2014.

[18] H. W. Lee, D. Thuente, and M. L. Sichitiu. Integrated simulation and emulation using adaptive time dilation. In *Proceedings of the 2nd ACM SIGSIM/PADS conference on Principles of advanced discrete simulation, (Denver, CO, May 18-21, 2014)*, pages 167–178. ACM, 2014.

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

[20] D. M. Nicol, D. Jin, and Y. Zheng. S3F: The scalable simulation framework revisited. In *Proceedings of the Winter Simulation Conference, (Phoenix, AZ, Dec 11-14, 2011)*, pages 3288–3299. Winter Simulation Conference, 2011.

[21] M. Rosenblum. Vmware's virtual platform. In *Proceedings of hot chips, (Stanford, CA, Aug 15-17, 1999)*, volume 1999, pages 185–196, 1999.

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

[23] A. Varga et al. The omnet++ discrete event simulation system. In *Proceedings of the European simulation multiconference, (Prague, Czech Republic, June 6-9, 2001)*, volume 9, page 65. ESM, 2001.

[24] E. Weingärtner, F. Schmidt, H. Vom Lehn, T. Heer, and K. Wehrle. Slicetime: A platform for scalable and accurate network emulation. In *NSDI, (Boston, MA, March 30-April 1, 2011)*, 2011.

[25] E. Weingartner, H. Vom Lehn, and K. Wehrle. A performance comparison of recent network simulators. In *Communications, 2009. ICC'09. IEEE International Conference, (Dresden, Germany, June 14-18, 2009)*, pages 1–5. IEEE, 2009.

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

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