Distributed Virtual Time-Based Synchronization for Simulation of Cyber-Physical Systems

CHRISTOPHER HANNON, Illinois Institute of Technology and CRCL GmbH
JIAQI YAN and DONG JIN, Illinois Institute of Technology

Our world today increasingly relies on the orchestration of digital and physical systems to ensure the successful operations of many complex and critical infrastructures. Simulation-based testbeds are useful tools for engineering those cyber-physical systems and evaluating their efficiency, security, and resilience. In this article, we present a cyber-physical system testing platform combining distributed physical computing and networking hardware and simulation models. A core component is the distributed virtual time system that enables the efficient synchronization of virtual clocks among distributed embedded Linux devices. Virtual clocks also enable high-fidelity experimentation by interrupting real and emulated cyber-physical applications to inject offline simulation data. We design and implement two modes of the distributed virtual time: *periodic mode* for scheduling repetitive events like sensor device measurements, and *dynamic mode* for on-demand interrupt-based synchronization. We also analyze the performance of both approaches to synchronization including overhead, accuracy, and error introduced from each approach. By interconnecting the embedded devices' general purpose IO pins, they can coordinate and synchronize with low overhead, under 50 microseconds for eight processes across four embedded Linux devices. Finally, we demonstrate the usability of our testbed and the differences between both approaches in a power grid control application.

CCS Concepts: • **Computer systems organization → Embedded and cyber-physical systems**; • **Computing methodologies → Simulation types and techniques**; *Distributed computing methodologies*; • **Networks** → Network properties;

Additional Key Words and Phrases: Cyber-physical systems, simulation, distributed systems, synchronization, testbed

ACM Reference format:
Christopher Hannon, Jiaqi Yan, and Dong Jin. 2021. Distributed Virtual Time-Based Synchronization for Simulation of Cyber-Physical Systems. *ACM Trans. Model. Comput. Simul.* 31, 2, Article 10 (April 2021), 24 pages.
https://doi.org/10.1145/3446237

This work is partly sponsored by the Air Force Office of Scientific Research (AFOSR) under Grant YIP FA9550-17-1-0240, the National Science Foundation (NSF) under Grant CNS-1730488, and the Maryland Procurement Office under Contract No. H98230-18-D-0007. 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 AFOSR, NSF, and the Maryland Procurement Office.
Authors' addresses: C. Hannon, Illinois Institute of Technology, 10 W. 31st Street, Chicago, Illinois, 60616, CRCL GmbH, 10 W. 31st Street, Chicago, Illinois, 60616; email: channon@iit.edu. J. Yan and D. Jin, Illinois Institute of Technology, 10 W. 31st Street, Chicago, Illinois, 60616; emails: jyan31@hawk.iit.edu, dong.jin@iit.edu.

1049-3301/2021/04-ART10 $15.00
https://doi.org/10.1145/3446237

1 INTRODUCTION

From the first ring of an alarm clock in the morning to the commute home from work, and from the manufacturing of products or robot-assisted surgery to the food on the table, computers are ingrained in every aspect of modern life. When computers interact with the physical world, they pull information from sensors in the form of measurements and signals. These observations are analyzed, transformed, and processed to interact and shape the physical world around them through actuators, motors, and controllers. These complex systems called **cyber-physical systems (CPSs)** require thorough testing and evaluation to ensure safety, security, and correct function. While great care is used to deploy modern cyber-physical systems, they are continuously being updated and expanded. As embedded computers monitor and control mission-critical physical processes in real time (e.g., an electrical power system), performing evaluation directly on them poses challenges. The mission-critical nature of some CPSs requires less intrusive techniques than manipulating them directly. Virtual testbeds are tools designed to address this challenge. A capable testbed combines both physical and virtual components, including but not limited to real sensors, embedded devices, virtual machines, emulated communication networks, simulation models of physical processes, analytical models of background traffic, etc.

A key challenge in simulating CPSs is to combine seamlessly the physical and cyber worlds to conduct high-fidelity experiments, as real components execute applications with the real-world wall clock and virtual components advance model states with a virtual clock. One solution is to provide a notion of virtual time to the physical processes so that their executions can be explicitly scheduled with simulation models and advance together in virtual time. Virtual time is a concept that was designed to enable multiple virtual machines to be multiplexed on a single physical hardware. We can use virtual time to schedule sequentially executed processes so that from their perspective they are being run in parallel. In simulation and modeling, virtual time is a technique used to synchronize emulated processes to make them execute in a reproducible way and behave more like traditional simulation models [Lamps et al. 2015, 2018, 2014], which also enables simulation models to be integrated with emulation. Furthermore, virtual time can also be used to slow down a running process and thus increase the perception of computer resources [Yan and Jin 2015a, 2015b]. If time moves half as fast to a process, i.e., in virtual time, it perceives that its CPU allocation is twice that of its non-virtual time counterpart. As a result, emulated processes can be executed on hardware that has fewer resources than required by the processes. For example, in communication network emulation, bandwidth on a virtual link can exceed the physical bandwidth of the hardware by slowing down the processes' perception of time by some ***time dilation factor (TDF)*** [Gupta et al. 2005].

A number of virtual time systems have been developed for different types of virtual machines running on a single physical machine (e.g., Xen [Gupta et al. 2005], Linux container [Lamps et al. 2014], and OpenVZ [Jin et al. 2012a]). Taking a set of processes and programs and using virtual time to schedule their execution, one can enable fine-grained control over the execution and interaction of processes. These sets of processes can be merged with traditional simulation systems (e.g., communication networks [Lantz et al. 2010; Yan and Jin 2015b] and power grids [Hannon et al. 2016]) using virtual time to integrate high-fidelity executing processes with simulation models.

In this work, we further enhance the capacity of a virtual testbed by developing the first distributed virtual time system on embedded Linux. The system enables efficient synchronization between the simulation of the physical aspect of CPS and real hardware computing devices running embedded Linux. Our contribution is a distributed system architecture uniquely consisting of a common virtual time Linux kernel module and three communication channels, one for virtual time synchronization using **general-purpose input and output (GPIO)** hardware interrupts,

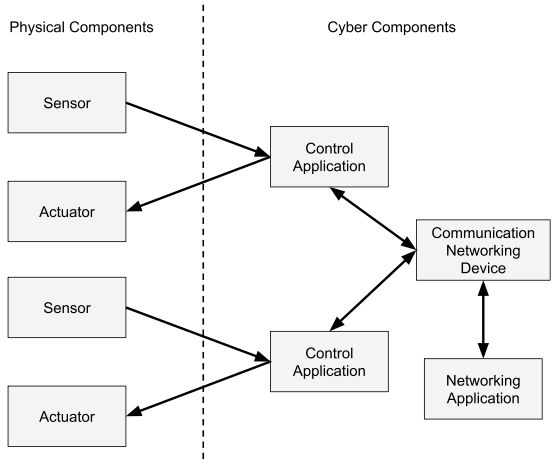

Fig. 1. Overview of a general cyber-physical system. Distributed virtual time enables the integration of the simulation of the physical state with the emulation using real hardware and processes of the control applications and communication networks.

one for connecting the embedded Linux devices, and one for interfacing with the physical system simulation that performs an offline computation. Rather than integrate emulated processes with simulation models, our work leverages the concept of virtual time to perform the reverse, namely to take simulation systems and integrate them into emulated systems. This distributed virtual time system enables the creation of a CPS testbed that can run real emulated processes while simulating the effects of the physical system. By creation of such a testbed, we enable higher-fidelity experimentation while exploiting the benefits of simulation.

Considering the electric power grid, for example, sensors feed data into control applications, which interact and in turn send control signals back to actuators to modify the state for the cyber-physical system. Figure 1 illustrates a general CPS. The left side represents the physical components of the system, while the right side represents the cyber components of the system. The distributed virtual time system enables the establishment of a high-fidelity hardware-in-the-loop testbed, with which we can simulate the state of the physical world, including the sensors and actuators, and inject the data into emulated control processes running across a distributed platform.

2 VIRTUAL TIME SYSTEM DESIGN ACROSS A DISTRIBUTED PLATFORM

Virtual time is a coordinated system that interfaces with running processes to provide an altered perception of time. We design a virtual time system running in synchronization across a distributed platform. Specifically, we design two modes of synchronization to support the needs of a cyber-physical system testbed. We design *periodic mode* as a scheduling-based approach allowing for repetitive synchronization, while *dynamic mode* is designed to allow on-demand synchronization.

2.1 Virtual Time

Virtual time serves two functions. The first function allows a system to pause and resume virtual-time-enabled processes, and the second function enables a system to dilate the running clocks of processes to make the processes' perception of virtual time faster or slower than the rate of the real wall clock. With the design of a CPS testbed, the system must perform offline calculations and simulation to pass to the hardware compute devices. Offline computation refers to calculations and simulations that are used to inject data about the physical component but does not advance the

overall time of the experiment. The CPS testbed and our use cases motivate the need for pausing and resuming processes and their clocks. However, it is also conceivable that dilation could be useful depending on the application of concern. For multiplexing applications running on common hardware, time dilation can alleviate problems of resource contention.

The contribution of our work is the design of a physically distributed but centrally coordinated virtual time system for cyber-physical systems. Our virtual time system supports two modes of synchronization: periodic and dynamic modes. Inherent execution differences in simulation and emulation introduce the main challenge to the synchronization of cyber-physical system testbeds. The problem is that emulation and real hardware applications are run in wall-clock time, while the physical components of the system are simulated offline. Simulation uses a simulation clock that advances arbitrarily to the wall clock. If the simulation clock advances much faster than the wall clock, it can be artificially slowed down to match the wall clock. However, the simulation clock may advance at a slower rate than the wall clock, which is our problem of concern. For example, a sensor can take a measurement very quickly in real life, but in simulation, the state of the simulation may take much longer to advance. By utilizing a virtual time system between running processes and the operating system's clock, the virtual time system can modify processes' perception of time transparently to the running process. For cyber-physical systems, virtual time can ensure the accuracy of data acquisition protocols. As an example, sensor data reading frequency may be rate-limited by the communication channel bandwidth. Virtual time can ensure that the perceived (and effective) bandwidth remains consistent with real-world behavior.

When a process subscribed to virtual time requests the current time, the operating system returns the process's virtual time, which is equal to the wall clock time less the time since the process was started, less total time paused, scaled by the time dilation factor and then added back to the time that the process started. For example, given start time T_s, current wall clock time T_{wc}, time dilation factor tdf, and total cumulative time paused T_p, the process's virtual time, T_{VT}, is given by the formula

$$T_{VT} = \frac{T_{wc} - T_s - T_p}{tdf} + T_s.$$

Therefore, when a process requests the current time, the process receives the total time running (while not paused) scaled by the dilation factor. Our design objective is to make this time function transparent to any process so that no code modification is required for a process to use virtual time. As a corollary, a process may not know that it is in virtual time.

2.2 Distributed Virtual Time

However, synchronization challenges emerge when applying virtual time to processes running across distributed hardware. Figure 2 illustrates the desired behavior for a distributed virtual time system with n devices. The horizontal axis shows the time with respect to the wall clock time. The black lines illustrate when a process and its clock are advancing. When a process on any device, e.g., Device 1, requests an offline computation from some external simulation source, all processes across devices using virtual time should be paused. Similarly, when the offline computation completes, all processes should resume uniformly. At time $T1$, a virtual-time-enabled process on Device 1 makes a synchronization request as shown by the red solid arrow. This request triggers the virtual time system to pause or freeze all the clocks and processes that are virtual time enabled across all devices. At time $T2$, the processes and their clocks are paused. Between times $T2$ and $T3$, the offline computation request is executed. Upon receiving a reply at time $T3$, a notification is sent to the virtual time system on Device 1. At time $T4$, all virtual time processes and their clocks resume. Ideally, ΔT_a and ΔT_b, the overheads of the pause and resume routines, should be zero.

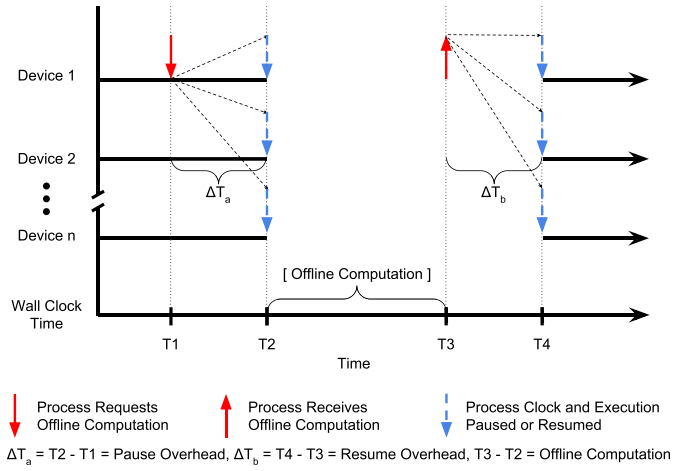

Fig. 2. A high-level design of distributed virtual time synchronization. When a virtual time process requests an offline computation, processes across all devices are paused and resumed uniformly.

Additionally, the pause and resume operations ($T2$ and $T3$) must occur across devices at the same wall clock time to ensure virtual time synchronization. In implementation, the non-simultaneous pausing and resumption across processes is the skew, which we measure in the evaluation section. In Figure 2, the total virtual time that has elapsed for the processes on Devices $[1 : n]$ is the wall clock time less $T4 - T2$, which is the total paused time.

In order to synchronize virtual time across multiple hardware devices, a communication channel is needed to connect the virtual time system on each device to distribute the pause and resume signals. However, in order to maintain high accuracy, the communication channel must have minimal latency. Shared channels, such as Ethernet, have too large of an overhead and rely on system scheduling. A dedicated link is ideal to prevent any congestion. Additionally, since the cyber-physical system itself relies on a communication network, this network must be disjoint from the virtual time system's communication channel, ensuring accuracy. Finally, a third communication channel is needed to interface with an external compute engine for offline computations including simulating the physical components of the CPS.

2.3 System Design

We use hardware interrupts to coordinate and synchronize virtual time across devices. Each device runs a common virtual time system, and they communicate with each other via hardware interrupts and voltage signals. The design of the virtual time system must ensure that any process on any device is able to pause the virtual time across all devices in real time and must be highly responsive. Hardware interrupts are very fast and can be set to call a handler function upon changes on an incoming voltage to a physical pin on the hardware device. Because many devices (e.g., sensors) in cyber-physical systems are low power, we utilize embedded Linux devices, which enables us to construct a low-cost hardware and emulation testbed while maintaining high fidelity.

While embedded Linux hardware and software have support for peripherals such as sensors and actuators, some FPGA hardware has ultra-high-resolution clocks to take frequent, accurate, timestamped measurements. One example is phasor measurement units in the power grid, which report with nanosecond accuracy the complex phasors of current and voltage being observed. Performing high-resolution measurements via virtual time synchronization events from the user space in Linux would provide some inaccuracy analogous to jitter. This jitter can be caused by the

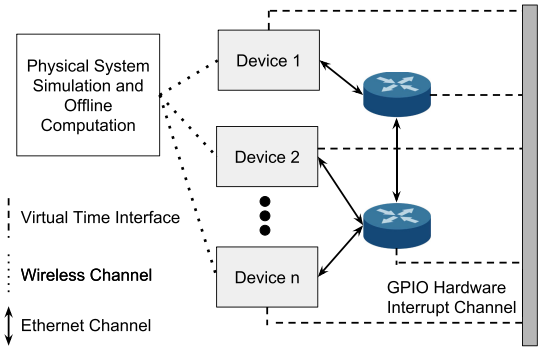

Fig. 3. The architecture of the virtual time system enables a virtual time channel, a channel for the testbed, and a channel to interface with a source for simulating the physical component of the CPS. All physical devices are required to be virtual time enabled.

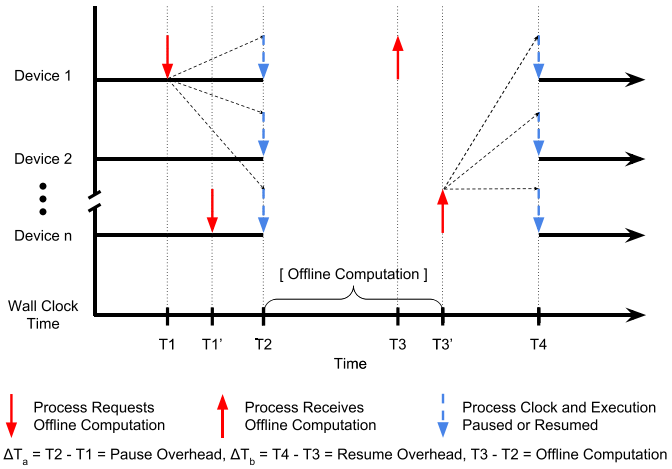

Fig. 4. When two processes on different devices create a synchronization event simultaneously, both virtual time systems handle the request. The processes on all devices should pause their clocks when receiving the synchronization event. All processes should only resume when all pending offline computation events are completed. This is accomplished by utilizing voltage signals on a common bus.

userspace timer, process scheduling limitations, clock resolutions, file system access, and so forth. While it is impractical without the similar precision of FPGAs to achieve the same accuracy, we make it a design decision to enable reliable repetitive synchronization in *periodic mode*. Additionally, *dynamic mode* enables controllers to change the state of actuators in the CPS system through synchronization events on-demand based on the logic and networking of the controller.

Figure 3 illustrates our design of a virtual-time-enabled simulation and emulation testbed. Devices can be machines that have a cyber and physical presence or just a cyber presence. The devices are Linux machines and are networked together through embedded Linux routers. All the physical devices run an instance of the distributed virtual time system, and any of them can have a connection to an external simulation and offline computation system.

Furthermore, our design of distributed virtual time must account for multiple hosts making offline requests at the same time or nearly at the same time. In Figure 4, we illustrate the desired behavior. When processes request an offline computation at nearly the same time, it can be possible

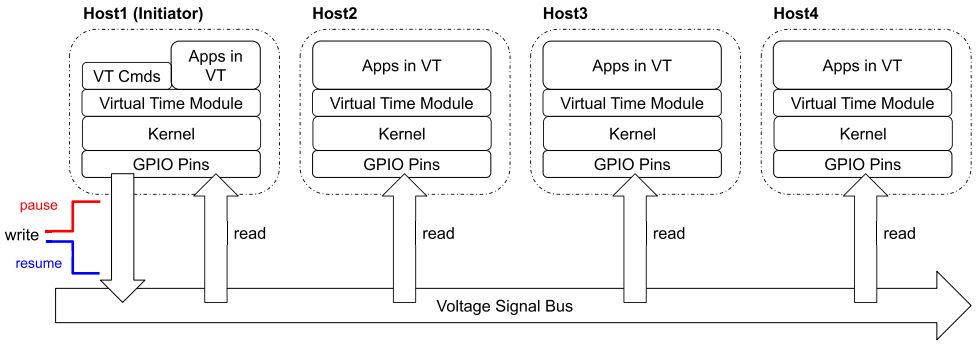

Fig. 5. Distributed virtual-time-enabled emulation testbed built on four embedded devices.

that two requests are sent before the first one pauses the system. For example, this scenario can occur when sensors collect data at synchronized periodic increments. While virtual time supports parallel requests, the power simulator executes requests sequentially.

If this occurs, we must ensure that the virtual-time-enabled processes are paused correctly. All processes should not be resumed until all pending events are finalized. Therefore, for n processes p_i on all the devices in the testbed, we define a binary function $S(p)$ such that $S(p_i) = 1$ maps to a request for process p_i; otherwise $S(p) = 0$. The pausing criteria becomes

$$\sum_{i=0}^{n} S(p_i) \geq 1, \tag{1}$$

while the resuming criteria is

$$\sum_{i=0}^{n} S(p_i) = 0. \tag{2}$$

In Figure 4, at time $T1$, a request is created on Device 1. However, at time $T1' \leq T2$, another request is created on Device n before the virtual time system completes the pause routine. Therefore, instead of resuming at time $T3$ as in Figure 2, the virtual time system must enforce the system to resume at time $T3'$ when all pending synchronization requests are completed.

3 IMPLEMENTATION

3.1 Distributed Virtual-Time-Enabled Emulation

Our emulation testbed supports distributed virtual time. By design, any physical host is allowed to play the role of the virtual-time-initiating host during experiment runtime to initiate synchronization requests (i.e., pause or resume the emulation) to all the running hosts. We illustrate the system using a four-host scenario as shown in Figure 5, where `Host1` sends the synchronization request to the other three hosts.

For example, `Host1` issues the request to pause the distributed emulation testbed; the request is handled by the virtual time **Linux kernel module (LKM)**; the OS kernel eventually converts it to a hardware signal. In the case of pause, the kernel changes the state on one of `Host1`'s GPIO pins to HIGH. This change outputs a voltage signal (the red arrow in Figure 5). Since all the hosts (including `Host1`) are connected to the voltage signal bus, they can read the voltage change. Each host's OS kernel has an interrupt registered to a rising/falling voltage signal on two of its hardware GPIO pins (different from the output pin). Depending on the type of the signal, i.e., rising edge or falling edge, the kernel triggers a software interrupt that the virtual time LKM has registered with an associated software routine. In the case of a rising voltage signal, the triggered software

interrupt instructs the virtual time module to pause all processes subscribed to the virtual time system. Resuming applications in virtual time works similarly except that a falling voltage signal (shown in blue in Figure 5) is created by changing the output pin's state on the virtual-time-initiating host from HIGH to LOW. The change in signal then triggers an interrupt on the pins registered to the falling voltage. Due to the oscillations in the electronic signal change, a debounce time of 20 microseconds is required on the rising and falling interrupts. This is set in the software. A hardware debouncing method could further reduce any skew or error introduced through this method. Therefore, we have kept the external hardware requirements to a minimum in our testbed. The embedded Linux computers have the required pull-up/down resistors to enable a bus to form between the GPIO pins. The hardware interrupts are configured and triggered using the generic interrupt controller (Corelink GIC-400) [Allwinner Technology Co., Ltd. 2013], which connects peripherals to the A20 Allwinner CPU [ARM 2011]. The rising and falling interrupts are registered to separate pins due to this configuration.

Apart from initially sending the rising or falling voltage signal to the bus, the virtual-time-initiating host `Host1` handles the signal on the voltage bus in the same way as `Host2`, `Host3`, and `Host4` do. Therefore, no bias occurs at the virtual-time-initiating host through any in-advance pause/resume operations. In addition, any host in Figure 5 is allowed to take the role of the virtual-time-initiating host to request pause and resuming operations. Throughout the life of an experiment, many hosts take turns to serve as the virtual-time-initiating host as they require an offline computation.

The design of the interrupts using rising and falling signals satisfies the case described in Figure 4. When multiple requests are created at the same time from distinct processes (across different devices), multiple signals can be sent out simultaneously. However, since all pins are connected to a common voltage bus, the rising signal interrupt is only triggered once. Similarly, the falling signal interrupt is only triggered when all output pins are changed from HIGH to LOW. Conveniently, the entire distributed testbed will initiate a pause at the moment of the first rising signal, equivalent to the pause criteria (1). Moreover, the testbed will not resume until the last signal changes to LOW, triggering the falling interrupt matching the resume criteria (2). As a result, the signal bus frees the kernel module from having to deal with overlapping synchronization requests.

3.2 Virtual Time Linux Kernel Module

Figure 6 depicts the detailed implementation of the virtual time LKM, specifically for an embedded Linux system with GPIO support (i.e., the Banana PI single board computer). The virtual time LKM works in two phases, namely proactive mode and reactive mode. In the proactive mode, virtual time LKM outputs a signal to the underlying bus; in the reactive mode, a hardware signal is sent to the virtual time LKM. Our implementation works for various embedded Linux devices including the Banana Pi M1s, the Banana Pi R1 Routers, and the Raspberry Pi devices. Additionally, our solution is designed to be compatible with any hardware that runs Linux and has the GPIO programmability. Barebone OS-less devices, such as 8- and 32-bit microcontrollers, are also compatible, but some system-specific modification is required due to the lack of an operating system. Microcontrollers can enable emulation of full sensor and actuator components of cyber-physical systems. The I/O functionality for hardware interrupts, clock interrupts coupled with the lack of multiprocessing, and context switching can make virtual time on microcontrollers highly accurate. We leave this line of exploration for future work.

3.3 Userspace Virtual Time Commands

Controls in proactive mode are initiated from the user-space commands performing the following operations, which are built using the virtual file system: (1) **register** a process in virtual time,

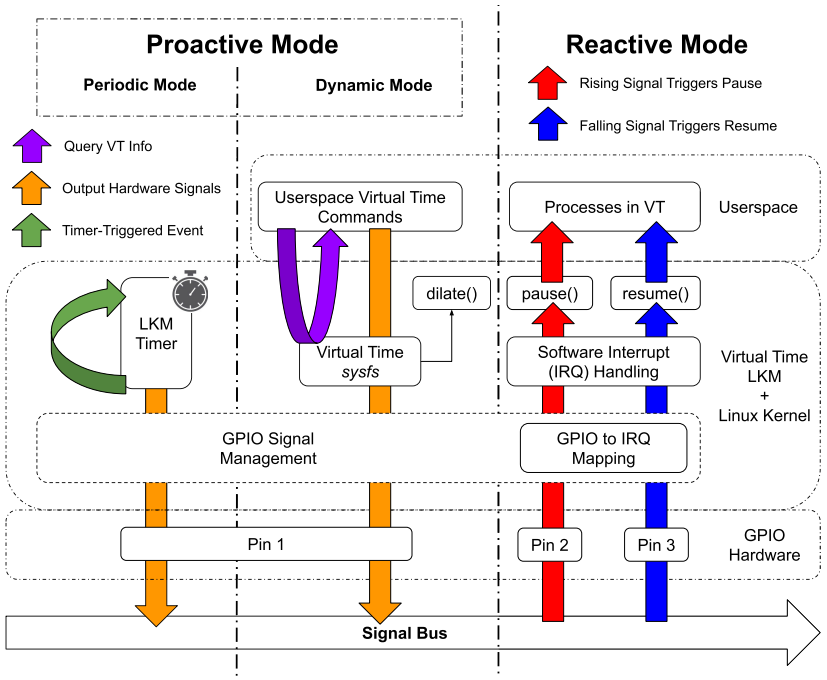

Fig. 6. GPIO signal and software interrupt-based virtual time Linux kernel module.

(2) **query** the virtual time information for a given process, (3) **dilate** the time of one or more processes, (4) **pause** all processes registered in virtual time, and (5) **resume** all processes registered in virtual time. After a process is registered in virtual time, LKM makes the rest of the virtual time commands available to the process.

3.3.1 Proactive Mode. Proactive mode is enacted when a host makes a synchronization request as the initiating host. There are two sub-modes in proactive mode: the first is *dynamic mode* and the second is *periodic mode*. Any host in the distributed system can create synchronization events in either mode interchangeably.

Regarding the query user command (the purple arrow in Figure 6), the virtual time system utilizes the sysfs subsystem to return the values of all virtual-time-related variables of that process. Regarding the dilate command, sysfs invokes `dilate(pid, tdf)`, which enables virtual time on a given process with a user-specified time dilation factor `tdf`.

In dynamic mode, pause and resume commands work in the same way as other sysfs entries (the orange arrow in Figure 6). For a pausing request, the virtual time sysfs triggers a callback to output a HIGH voltage signal on Pin 1 via GPIO signal management API. For a resume command, the output voltage signal returns to LOW.

For periodic mode, however, at load time of the LKM, the parameter `period` is passed in. This value, set in milliseconds, is the interval of the periodic mode of synchronization. When the timer expires, the system is paused from the LKM in the same way that the pause command is triggered by the user space process. However since a user space process did not initiate the request, it sets a flag in sysfs to indicate to a user space process that the system has initiated a synchronization event from a periodic mode event. Resuming the process is done the same as in dynamic mode with the addition that the LKM resets the timer to expire at the next interval. It is important to

note that the design of virtual time is blocking by nature; a user space process is responsible for resuming the distributed system or else the system can result in a deadlock.

ALGORITHM 1: Pause/Resume Processes in Virtual Time

```
Data: Key variables maintained by virtual time LKM:
      procList, list of processes controlled by virtual time LKM.
      tdf, time dilation factor.
      frzNow, the moment of pausing all processes in wall clock time.
1  Function pause()
2      for i ← 0 to length(procList) by 1 do
3          send SIGSTOP signal to procList[i]                    // Pause processes.
4      end
5      frzNow ← __getnstimeofday()                          // Record wall clock time.
6  return
7  Function resume()
       /* Calculate virtual time.                                                  */
8      duration ← (__getnstimeofday() − frzNow)/tdf
9      for i ← 0 to length(procList) by 1 do
10         increase procList[i].freeze_past_nsec by duration
11     end
12     for i ← 0 to length(procList) by 1 do
13         send SIGCONT signal to procList[idx]                  // Resume processes.
14     end
15 return
```

3.3.2 Reactive Mode. The proactive mode, however, only handles the part of the work to pause or resume the processes in virtual time. The other half is accomplished in the reactive mode when the voltage signal sent by the proactive virtual time LKM is received on Pin 2 and Pin 3. Reactive mode is handled uniformly without regard to being dynamic or periodic mode. The red arrow depicts how a rising signal triggers `pause()`. During the initialization, the virtual time LKM maps Pin 2's GPIO to a software interrupt number, *irq*. In addition, it registers the software interrupt handler `pause()` for *irq*. Only upon detecting a rising signal on Pin 2 will the GPIO signal management trigger the pre-mapped software interrupt *irq*, which is handled automatically by the virtual time LKM using the `pause()` handler. Algorithm 1 describes how the system pauses the processes attached to the virtual clock. The system sends `SIGSTOP` signals to all virtual-time-enabled processes before querying the current wall clock time. In the case of the resuming operation (shown in blue), the differences are (1) the source is a falling voltage detected on Pin 3, and (2) the destination is another pre-registered software interrupt handler `resume()`, listed in Algorithm 1, for a separate software interrupt number $irq' \neq irq$. Software interrupt handler `resume()` first calculates the duration since the pause moment in virtual time and then updates the variable $freeze\_past\_nsec$ for each process. This way, all processes running on a single host are conceptually paused for the same duration of virtual time. Then the `SIGCONT` signal commands the kernel to wake up all processes registered in the virtual clock. Even if the moments to wake up are different for processes in *procList*, the perceived pause duration is identical among the processes. Our signal-bus-based hardware design also simplifies Algorithm 1 as it is not necessary for the software module to cope with the tangled synchronization requests from multiple devices.

3.4 Virtual Time Retrieval

In addition to the coordination of the virtual time LKM described in Section 3.2, to correctly retrieve virtual time we also modify the system calls that return time to a process. When a process requests time through the `gettimeofday()` system call, the kernel returns a virtual time if the requesting process is a virtual-time-enabled process. We port the virtual time kernel in Yan and Jin [2015a] to the ARM Linux kernel 3.14. Our implementation behaves the same as the original kernel with necessary modifications regarding the ARM instruction set. The new virtual time kernel adds the file system entries to the `/proc/$PID` directory to contain the virtual time metadata. When a process calls `gettimeofday()`, the function checks the virtual time metadata and returns the virtual time to a process. A major advantage of this approach is that the virtual time retrieval procedure is transparent to applications. The only requirement is that the time source must use the monotonic clock representation. The detailed kernel implementation is described in Yan and Jin [2015a].

4 EVALUATION

In this section, we present the evaluation results of the distributed virtual time system in terms of overhead and correctness. The testbed used to evaluate the virtual time kernel module is made from eight Banana PI devices. Four devices are M1 single-board computers and four devices are R1 embedded routers. Banana PI devices have the A20 ARM Cortex-A7 Dual-Core CPU at 1GHz with 1GB DDR3 Memory and a 1Gb Ethernet controller.

4.1 Virtual Time Overhead

Measuring the performance of any system naturally introduces further overhead. The approach tries to minimize the impact of observing the system while providing useful profiling information. The overhead can be divided into two components: the overhead added to the `gettimeofday()` system call and the overhead in pausing and resuming processes. To measure the overhead of the `gettimeofday()` system call, we employ the following methods.

4.1.1 System Call gettimeofday(). The first way to measure overhead is the function tracer `ftrace`, which enables fine-grained measurements of kernel function latencies. `ftrace`, specifically the function graph tracer, works by probing a function on both its entry and exit using a dynamically allocated stack of return addresses, which it overwrites to calculate latencies [Rostedt 2017]. When a program is compiled, these probes are set in each function call; however, when not in tracing mode, the impact to normal performance is limited by using `nops` in place of the function hooks.

We compile the kernel with the `ftrace` option to measure the overhead of virtual-time-related system calls. Table 1 illustrates the system call progression in and out of virtual time for the `gettimeofday()` system call. This is the overhead a process experiences when querying the clock. Virtual time clocks are transparent to the processes. A virtual time is returned in place of a real time. Virtual-time-based timekeeping in the `gettimeofday()` system call increases the kernel space overhead from an average of 8.625 microseconds to 13.333 microseconds. Seemingly virtual time adds substantial overhead in calculating the offset from the real clock.

However, the values reported are quite inflated due to `ftrace`. In reality there are not many instructions inside the `gettimeofday()` call, so when measuring, the `ftrace` work is a substantial portion of these values. We should expect this system call without `ftrace` to perform significantly better. We provide these values just to reassure the reader that the changes we make to the kernel do not cause drastic changes to the performance.

Table 1. Duration of sys_gettimeofday Function Call

Virtual-time-enabled process	
Time	System Call
	sys_gettimeofday() {
	do_gettimeofday() {
	getnstimeofday() {
0.250 us	arch_counter_read();
	ns_to_timespec() {
0.541 us	ns_to_timespec.part.0();
3.041 us	}
8.250 us	}
+ 10.583 us	}
+ 13.292 us	}

Regular non-virtual-time-enabled process	
Time	System Call
	sys_gettimeofday() {
	do_gettimeofday() {
	getnstimeofday() {
0.250 us	arch_counter_read();
2.792 us	}
5.333 us	}
8.000 us	}

The durations include the time of nested calls. (Top Function trace is for gettimeofday() system call for virtual-time-enabled processes; The total time is 13.292 microseconds. (Bottom) Function trace is for regular processes not registered to virtual time. The total time is 8.000 microseconds.

The function tracer ftrace only monitors the kernel function sys_gettimeofday(). In order to measure the total time from the user space call, another method is employed. We create a C program that calls gettimeofday() 1,000,000 times and calculates the average duration of the call for a process registered to virtual time and a non-registered process. When not in virtual time, the average overhead of a regular process calling gettimeofday() is 11.833 microseconds per call, while in virtual time it is 17.387 microseconds. For comparison, ext4_mark_inode_dirty() requires 58 microseconds, unlock_new_inode() 36 microseconds, and writing 2.7MB to a file takes 975 microseconds with sys_write().

4.1.2 Linux Kernel Module Proactive Mode. As explained in the previous sections and illustrated in Figure 6, there is a proactive mode and a reactive mode of the LKM. In the proactive mode, a dynamic synchronization event is created by a process writing to the virtual file system to trigger a callback that changes the output on GPIO Pin 1. While maintaining the above disclosure about ftrace's overhead, we employ ftrace to measure the latency in the LKM including writing to the virtual file system. The main function in the kernel in the proactive mode is sys_write(). This function takes on average 262.292 microseconds. The main functions sys_write() calls are sysfs_write_file() (204.5 microseconds), which calls mode_store() (105.833 microseconds) and in turn either calls gpio_direction_output() (47.250 microseconds) or gpio_direction_in() (38.833 microseconds) for pause and resume functions, respectively.

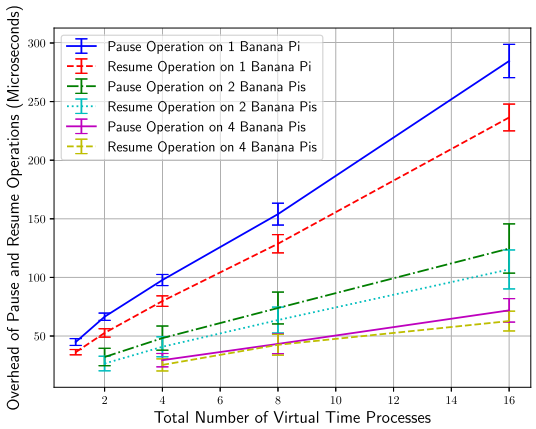

Fig. 7. The overhead of pausing and resuming scales linearly with the number of processes subscribed to virtual time. Using more boards allows the parallelization of virtual-time-enabled processes, which enables a linear speedup as expected. 95% confidence intervals are shown.

The top-level function writes a value to the virtual file system `/sys/VT/mode`, which determines if `pause` or `resume` should be called inside the LKM. Next `mode_store()` changes the state of the GPIO Pin 1. A full trace of a proactive `pause` routine can be found in the online repository [Hannon and Yan 2016] due to the length. In the periodic synchronization events, a timer in the kernel is responsible for triggering virtual time and the write-in user space can be avoided.

4.1.3 Linux Kernel Module Reactive Mode. Tracing the reactive mode of the LKM is more difficult as the function tracer is unable to probe the entrance and exit of the interrupt functions. In order to measure the overhead of the reactive component of the LKM, we timestamp the entrance into and exiting the `pause` and `resume` functions. For each process registered in virtual time, the LKM sends a `SIGSTOP` or `SIGCONT` signal. Additionally, it writes some timekeeping data to the process's `/proc/$PID/` directory. Figure 7 illustrates the overhead of pausing and resuming virtual time processes in the reactive mode of the LKM. The overhead of a single machine, two machines, and four machines are plotted for up to 16 virtual time processes with their 95% confidence intervals. The overhead scales linearly with the number of virtual-time-enabled processes for all scenarios. On a single machine, 16 virtual-time-enabled processes take 282 microseconds to pause and 234 microseconds to resume. Using more boards allows us to parallelize the pausing and resume, which reduces the overhead as shown in Figure 7.

4.2 Virtual Time Correctness

In addition to determining the latency imposed by virtual time, it is also critical that the virtual time does not introduce unexpected errors in applications. To evaluate the correctness of the distributed virtual time system, it is important that the systems' clocks do not skew over time. Additionally, we evaluate the bandwidth between two Banana Pi hosts in and out of virtual time to ensure that errors are not introduced.

4.2.1 Clock Skewness. The clock skewness is the difference in the value of time reported by two clocks at a given instance in time. Two factors can contribute to the clock skewness. First, as in all commodity systems, the onboard clock is not perfect; the clocks on the Banana PIs will naturally drift with time. Additionally, our virtual time design supports any embedded Linux device that supports the ARM kernel, which can have different accuracy in their underlying clock hardware. Furthermore, we observe variations in clock accuracy from different distributors of the

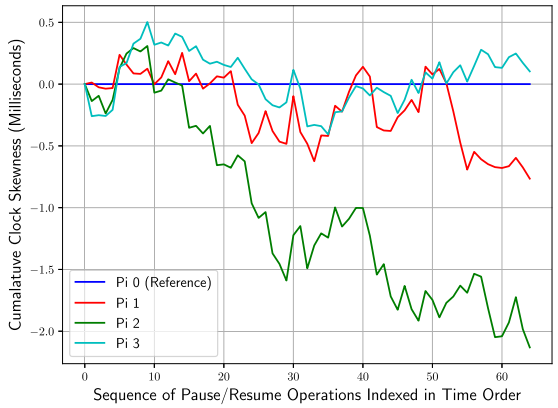

Fig. 8. The processes are periodically paused and resumed using the virtual time system. We measure the skewness in milliseconds on the y-axis and plot the index of the pause/resume operations on the x-axis. The processes are paused and resumed 64 times. The skewness of the clocks over time remains within acceptable limits in comparison with typical clock skewness. The skewness is measured by querying the cumulative paused time of each virtual-time-enabled process.

same hardware. The **Network Time Protocol (NTP)** can be used to correct drift errors. The second source of clock skewness is introduced by the virtual time system. Since it is infeasible to query two clocks simultaneously with high fidelity to determine the clock skewness, we employ an indirect method. The virtual time is given by the wall clock time minus paused time, and one can query the paused time while the virtual time processes are running. Additionally, paused time advances only when processes are paused and resumed. To evaluate the clock skewness, we apply the pause and resume operations on four virtual-time-enabled processes across four Banana Pi devices. Each process within a device shares the same virtual time offset. Each device takes its turn as the *master* node shown in Figure 5 in a round-robin order. After each resume operation, the cumulative paused time is recorded by querying the virtual time interface. In Figure 8, we plot the skewness of the virtual time clocks with respect to an arbitrary virtual time Pi as the reference. The results show that the clocks do skew over time but that they remain within a close range after 64 pause and resume operations. Mani et al. [2018] examine clock drift over Internet-of-Things devices including Raspberry Pi devices, which are similar to the Banana Pi devices used in our work. The authors conclude that with their implementation they can maintain a clock accuracy of within 15 milliseconds. Using NTP, the clock skewness we observe in virtual time remains tolerable.

While 64 synchronization events through the distributed virtual time system may be realistic for some cyber-physical system analysis and simulation, we also explore the clock skewness as the number of synchronization events increases to 2,048 in Figure 9. The skewness from six clocks is plotted with a reference clock. Observed are trends in the skewness for different batches of devices. In Figure 10, we show the 95% confidence interval for skewness between all device clocks. The skewness grows over the life of an experiment. Therefore, the length of the experiment impacts the clock skewness, and in some applications, it is perceivable that this may exceed experiment requirements.

Potential Mitigation 1. One approach to reducing the clock skewness over time is to expose writeable entries into the Linux kernel module allowing user space processes to modify the state of the LKM virtual time. This, incorporated with a user space synchronization algorithm that runs while the devices are performing offline calculations, could adjust the clocks monotonically.

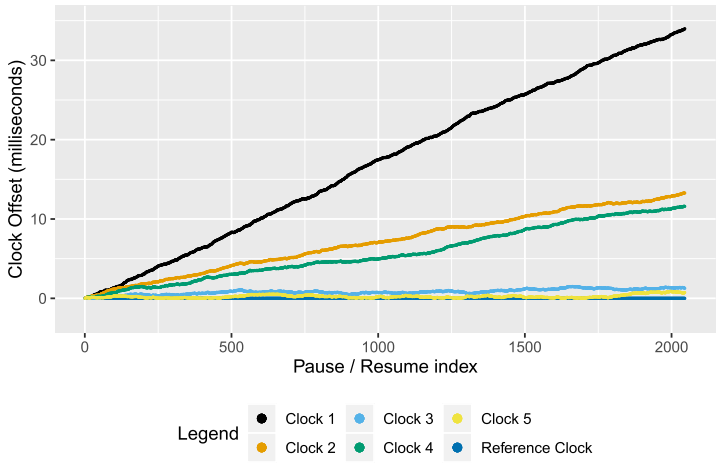

Fig. 9. The processes are periodically paused and resumed using the virtual time system. We measure the skewness in milliseconds on the y-axis and plot the index of the pause/resume operations on the x-axis. The processes are paused and resumed 2,048 times.

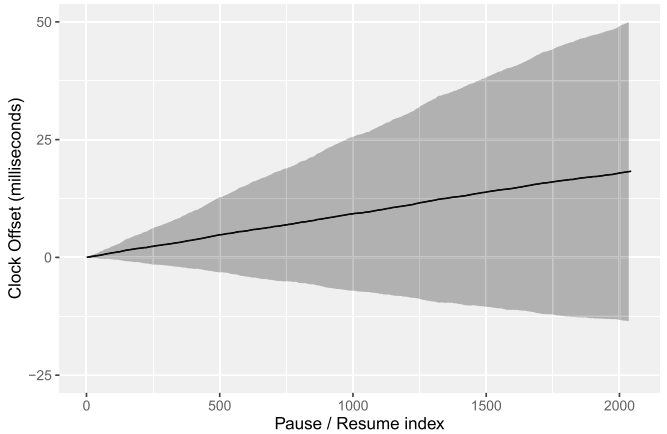

Fig. 10. The processes are periodically paused and resumed using the virtual time system. We compare all device clock pair offsets and plot the average pairwise skewness with 95% confidence intervals. It can be observed that over the course of a simulation, the clock's skewness increases. At 2,048 synchronization events, the average skewness is under 20 milliseconds.

Potential Mitigation 2. An alternative approach to reducing clock skewness is to take advantage of the periodic nature of periodic mode events. When a periodic mode event occurs, it is because a timer has elapsed. To exploit periodic events, the frequency of periodic events is known at load time. When the periodic event occurs, all clocks across the distributed system would be able to checkpoint and set their virtual clocks to the frequency. The catch is that the reactive mode will need to differentiate between dynamic and periodic events. This would require an additional channel of GPIOs, software interrupts, and state variables in the LKM. While the embedded Linux devices used have GPIO pins to spare, it adds additional complexity in the setup. Another downside to this is that the clock updates would not be guaranteed to be monotonic and that the distributed

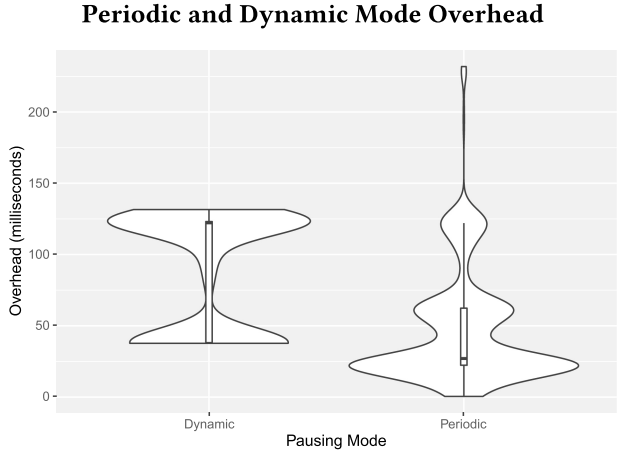

Fig. 11. The processes are periodically paused and resumed using the virtual time system at an interval of 250 milliseconds. We measure the actual interval by comparing the resume and pause timestamps in the kernel. Both dynamic mode and periodic mode are compared and distributions shown. We find that the average overhead in periodic mode is 54% of that in dynamic mode with a few outliers.

LKM will become a leader/follower system rather than purely distributed. We leave the exploration of these approaches as future work when experiments requiring strict clock restrictions exist.

4.2.2 Periodic Mode vs. Dynamic Mode. The *periodic* mode implementation utilizes a timer in the Linux kernel. At load time, a parameter setting an interval in milliseconds for a periodic synchronization event can be passed to the LKM. When the timer expires, it pauses the system using the proactive mode. When a *dynamic* mode event occurs, any timer currently in use gets paused by the interrupt handler. The advantage of periodic mode is twofold. First, the pause process does not need to go through the `sysfs` filesystem, which results in a potential speedup. Second, devices like sensors in the system do not need to utilize a programming language-specific user space timer, which may also help to reduce error. In Figure 11, we observe that the periodic mode can reduce the average error from a sensor-like device requesting synchronization. Although there are a few outliers, the periodic mode has on average 50% less overhead.

4.2.3 Network Performance in Virtual Time. In order to verify the correctness of our virtual time system, we measure the performance of a common networking application in and out of virtual time. We connect two Banana Pi hosts together via Ethernet and run iperf3 in TCP mode. In the first case, we run iperf for 30 seconds with processes not in virtual time. Although the network interface is rated for 1Gbps, we observe that the practical limit is closer to 450Mbps. In the second case, we add the iperf processes on both hosts to virtual time and periodically pause and resume from the iperf client. The virtual time system is completely transparent to the iperf applications. The client host is paused every 1 second for a duration of 1 second. The iperf test is also run for 30 virtual time seconds, which takes 60 wall clock seconds. The bandwidth we observe in Figure 12 is similar to the base case where processes are not using virtual time. The average throughput of the normal case is 453.89Mbps with a variance of 113.9Mbps, while the virtual time case is 453.90Mbps with a variance of 116.76Mbps.

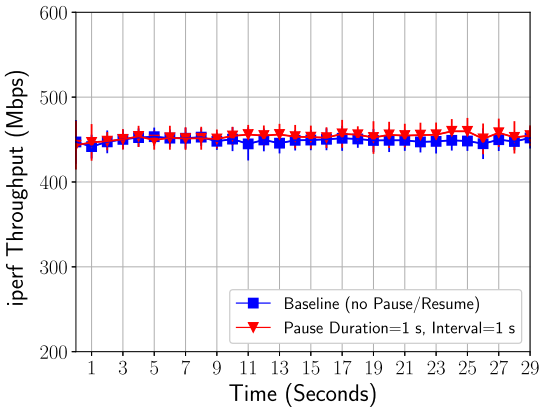

Fig. 12. We use iPerf to measure the bandwidth between two hosts. This graph shows that the performance of the networking application iPerf is correct when a process is in virtual time.

5 CYBER-PHYSICAL SYSTEM USE CASE

One of the largest cyber-physical systems is the electric power grid, which is composed of many physical components including generators, transformers, loads, capacitors, and so forth. Due to the emergence of distributed and renewable generation such as solar power and wind generation, maintaining the stability of the electric power grid is becoming increasingly important and expensive. When electric energy generation is dynamic due to fluctuations in the physical environment, electric grid operators must implement appropriate strategies including demand response [Siano 2014], virtual power plants [Pudjianto et al. 2007], energy storage systems [Carrasco et al. 2006], and so forth. These techniques require greater observability and controllability, which in turn requires more extensive communication infrastructure within the power grid.

5.1 Voltage Stability Application

We demonstrate the usability of our virtual-time-enabled testbed by evaluating the cyber-security of a voltage stabilization application. In order to maintain grid stability, an actuator compensates for the changes in the dynamic generation in real time. It consists of an energy storage device and a control system and works as follows. A sensor collects voltage and frequency data from the distributed energy resource in the electric power grid. It transmits the data to the actuator's control system that determines the required settings for the energy storage device. If there is an excess of energy being produced, the control system will reduce the discharge rate of the energy storage device. If the excess is large, the energy storage device will switch from discharge to charge mode to consume energy and replenish its storage. On the other hand, if the energy produced from the generator is low, the energy storage device will supplement the grid with power.

In this use case, we consider the voltage stabilization application run in the IEEE 13-bus power system [Kersting 2001] illustrated in Figure 13. The power system is composed of busses, which are nodes in the power system topology, such as substations, that distribute power to loads to residential communities. Busses also connect generators to the power grid, such as solar, wind, and hydro resources. The IEEE 13-Bus test case is used with added dynamic loads at Bus 611, Bus 652, and Bus 692. A dynamic generator is added at Bus 634 and an energy storage device is added at Bus 675. The point of common coupling to the transmission network is at Bus 650. The base system voltage is 2.4kV provided from this bus. The sensor is located at Bus 634 near the distributed generation and the application is run at node 675.

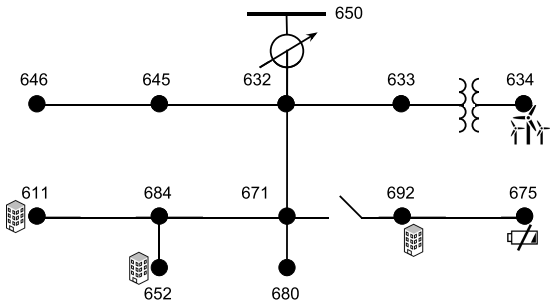

Fig. 13. 13-Bus distribution system.

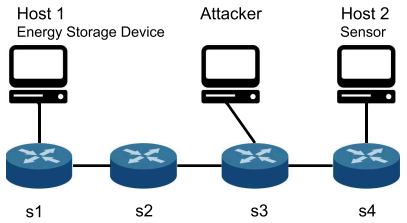

Fig. 14. The communication network used in the power grid application.

The cyber components are run on our distributed virtual time testbed while the physics of the power system are simulated. We adopt a linear communication network topology to connect the hosts in the control application. Figure 14 illustrates the linear communication network we created using the Banana Pi hosts and routers to emulate the cyber-components of the system. The energy storage device (Host 1) is connected to switch s1, while the sensor-measuring generation output (Host 2) is connected to switch s4. All hosts are additionally connected to a power grid simulator, OpenDSS [Dugan 2013; Montenegro et al. 2016], through a separate network to simulate the physical state of the system. The power simulator is run on a Windows 10 virtual machine with 8GB of memory and four 4GHz cores. The sensor retrieves voltage, frequency, and current data from the power simulator, and the actuator changes the state of the system by modifying the state of the energy storage device. The simulator advances when either the sensor requests data or the energy storage device changes the output state. The simulator updates and advances its state up to the virtual clock time at a millisecond time step. The emulation experiment takes up to 10 times as long as the wall clock advances, due to the computational complexity of both calculating the state changes and sampling the circuit properties of the electric power system.

When the sensor determines it is time to retrieve a value from the power simulator, it sends its request to a management process that makes the call to the virtual time module. The management process interfaces with the power simulation server to relay the request. When the request is fulfilled, the management process places the sensor data in the sensor process's memory. The final step is to interact with the virtual time system to resume the processes as in Figure 6.

The blue line in Figure 15 shows the operation of the power grid under normal behavior. The upper right plot shows the variability in the generation of the wind turbine over time in apparent power. The lower right plot illustrates the functionality of the voltage stability application over 90 seconds. The energy storage device charges for a short period between 18 and 21 seconds while discharging at a variable rate for the remainder of the experiment. Since there still remains randomness in the quantity of power consumed by loads, there are some voltage fluctuations as can

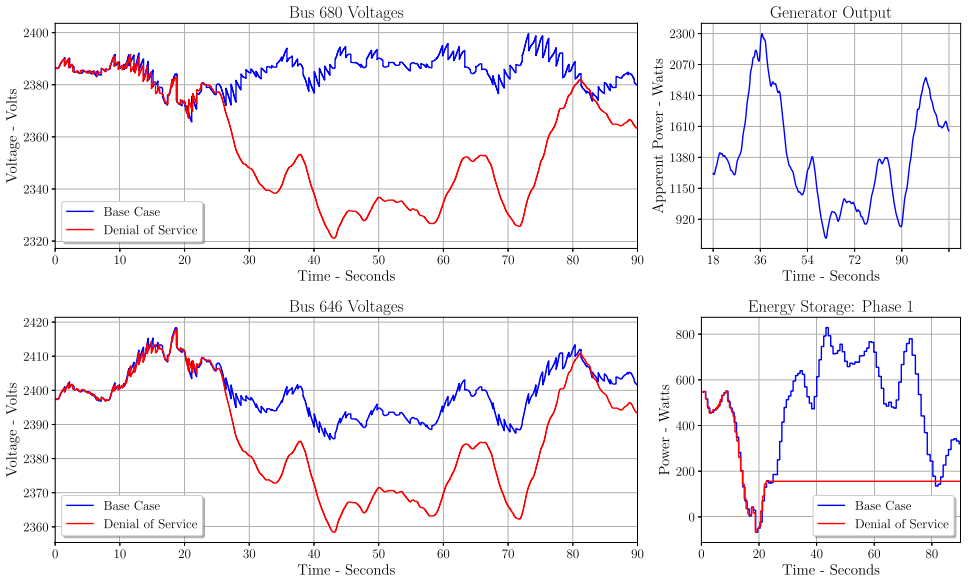

Fig. 15. Left: Voltage of the power system at Bus 680 (Up) and Bus 646 (Down). Top Right: The apparent power in watts generated by the wind turbine. Bottom Right: The energy storage device's supplied power and load.

be seen in the left two figures, which illustrates the voltage at Bus 680 and Bus 646 in the power system. However, the voltage does not deviate significantly from the baseline of 2,400 volts.

5.2 Cyber-Attacks on Power Grid

We demonstrate how a cyber-attack can affect the voltage stability of the power system and impact the security of the power application. We make the assumption that a malicious actor has access to a compromised device on the network. Using `dsniff` [Song 2001], the attacker floods the ARP tables of the devices on the network in order to (1) convince the sensor that the attacker is the energy storage device's controller and (2) convince the energy storage device's controller that it is the sensor. Equivalently, the attacker is able to eavesdrop on all communications between the sensor and the controller. Furthermore, by removing the compromised host's network port from promiscuous mode, the attacker can create a **denial-of-service (DoS)** attack, preventing network traffic from routing between the honest hosts (energy storage device controller and the sensor). In Figure 15, the red curves illustrate the case when the attacker performs a DoS attack at time 22.5 seconds. After this point, the communication channel between Host 1 and Host 2 is severed. Because the energy storage device is unable to receive new control signals, it maintains its state of discharging. The effects of the DoS attack can be seen in the figures on the left. At Bus 680, the voltage deviates 3.33% from nominal to 2,320 volts. If the grid experiences instability for a prolonged period of time, the frequency of the power system will deviate, which causes larger generators to protectively shut down, further contributing to voltage instability and power outages. Because of the magnitude of the power deficit, the voltage drop at the busses will follow the generator's output. The base case has less uniformity than when there is an attack because the system is constantly updating the charging to compensate for the dynamic generation. The denial of service deviates the stability of the power system based on the generator output. Since the energy storage unit is discharging at a constant rate, the patterns between Bus 680 and Bus 646

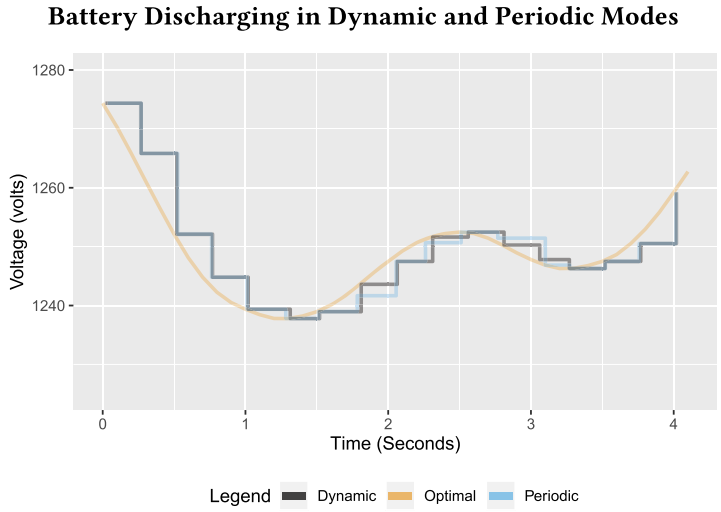

Fig. 16. Dynamic and periodic modes are compared, which illustrate inaccuracies introduced through overhead. Periodic mode introduces less error, thus reducing the propagation of error.

follow the same curve at close voltage. The base case looks less uniform but more stable around 2,400 volts because the battery is adjusting based on the power generation. The differences between the voltage levels at the bus are attributed to the power system characteristics.

After an attacker has access to the network and is able to eavesdrop on real-time traffic, there is the potential for spoofing attacks, replay attacks, false data injection attacks, and more. Spoofing attacks can impersonate traffic to send data, configuration, or even control messages to cyber-physical devices in the grid, potentially opening relays or putting the grid in an unstable state. Replay attacks enable the attacker to resend network packets that have the potential to change the state of the system. Even encrypted packets have the potential to become susceptible to this family of attacks. Furthermore, by modifying packets in transit, applications can become victims to false data injections that force control applications to make improper decisions. There exist preventative and reactionary measures to prevent such attacks. For example, having a static routing policy at the switch level can prevent ARP spoofing and poisoning attacks. Encryption under proper configuration can mitigate eavesdropping and spoofing attacks. The more cyber components that are adopted, the more cyber security plays a role in creating resilient and robust systems.

5.3 Performance Comparison between Dynamic Mode and Periodic Mode

The *periodic* mode is designed for sensors that take reoccurring scheduled measurements. The reduced overhead allows for higher fidelity of synchronization, which in turn leads to higher-fidelity experiments. Actuators, on the other hand, do not have the ability to be scheduled in advance. Actuators are triggered by events from the controllers. Therefore, their influence on the physical world is dynamic and on-demand, thus using the *dynamic* mode. The difference in the modes is shown in Figure 16. In this case, we focus on a subset of the battery discharge. If the sensor takes four readings per second, the difference in the accuracy of dynamic and periodic mode can be seen. Figure 16 illustrates a 4-second window where the optimal battery discharge is plotted next to the battery discharge actuator. In this comparison, the overhead is sampled from the distributions displayed in Figure 11. The differences in the periodic mode and dynamic mode are shown by the time that the periodic mode and dynamic mode return to the optimal line. While these changes

have minor significance here, over the course of an experiment with many synchronization events, these errors get compounded.

In summary, we demonstrate the role of distributed virtual time in a physical testbed equipped with simulation modeling using a voltage stability application. The distributed virtual time is critical because simulating the actuator's control system is computationally intensive and cannot be performed in real time. With distributed virtual time, it runs in synchrony with the physical networking system, which is a must-have for reproducing various attacks on a power grid launched from its underlying networking layer and for evaluating possible countermeasures.

6 RELATED WORK

Research work on virtual time in network emulation can be generally classified into two categories based on the application objectives. The first objective is to improve the scalability and fidelity of network emulation testbeds. They typically adopt the virtual time technique to uniformly scale the emulation entity's perception of time by a specified factor. It was first introduced as time dilation in Gupta et al. [2005] and has been adopted to various types of virtualization techniques (e.g., virtual machines, virtual nodes, Linux containers) and integrated with a handful of network emulators [Erazo et al. 2009; Grau et al. 2011, 2008; Lamps et al. 2014; Yan and Jin 2015b]. For example, VT-Mininet [Yan and Jin 2015b] used time dilation to virtually scale up the system resources on a single-commodity machine to support high-fidelity analysis of large-scale software-defined networks. TimeKeeper [Lamps et al. 2018] studied how time dilation enables "moderate hardware" to emulate a smart grid control network that requires "powerful hardware." In addition, Grau et al. [2012] explored means to minimize the running time of dilated network emulation experiments based on the historical average resource requirements of virtual nodes.

The second objective is on the temporal integration of network emulation and simulation. Several hybrid testing systems that integrate network emulation and simulation have adopted this approach [Hannon et al. 2018; Jin et al. 2012b; Lamps et al. 2014; Weingärtner et al. 2011]. For example, Hannon et al. [2016] combined a power distribution system simulator with an SDN emulator to support evaluation of communication network applications and their impacts on corresponding power systems. The synchronization of the two systems is achieved via freezing (and then resuming) Linux containers including their own virtual clocks. By embedding Linux processes in virtual time, TimeKeeper [Lamps et al. 2014] successfully integrated two network simulators (ns-3 and S3F) to a network emulator with negligible modification. SliceTime [Weingärtner et al. 2011] used a common barrier to block both simulation processes and virtual machines in emulation until both systems complete a rational time slice. On the other side of the coin, temporal integration of emulation and simulation can be approached from the simulation side, for the case of **parallel discrete event simulation (PDES)**. The general idea is to run PDES fast enough so that it always catches up with the real-time emulation. For example, extreme simulation [Carothers 2002] can be adopted to generate a statistically valid result by a real-time deadline, typically within 100 milliseconds. It is also possible to adaptively control the progress of simulation time using an elastic time algorithm [Srinivasan and Reynolds 1998].

Distributed virtual time relies on software triggering hardware interrupts to coordinate and synchronize between cyber and physical simulation systems. Interrupts are also used in simulation systems including parallel systems to preempt events [Pellegrini and Quaglia 2017]. Silvestri et al. [2020] use an interrupt-based synchronization method to prevent the waste of CPU resources in the event of an out-of-order event. However, the previous uses of interrupts are used to synchronize more efficiently in optimistic simulation, while our use is to uniformly halt the cyber system.

Cyber physical systems have gained research attention over recent years. Nutaro and Ozmen [2019] simulate a building control network and show it is feasible. The authors expand their work

into simulating transactive energy management systems in Özmen et al. [2020]. Al-Hammouri [2012] present a co-simulation framework for cyber-physical systems. Koutsoukos et al. [2018] present a testbed for transportation cyber-physical systems. In our work we present a hybrid simulation/emulation testbed based on virtual time.

Distributed virtual time facilitates the integration of simulation to emulated or physical hardware-based testbeds that are temporally transparent to the emulated processes. This article presents the first working virtual time system that is synchronized across multiple physical devices.

7 CONCLUSION

We present a distributed version of virtual time across multiple embedded Linux devices, which are essential components of a high-fidelity cyber-physical system simulation/emulation testbed. We develop two modes of synchronization, *periodic* and *dynamic* modes, to support reoccurring and on-demand events. Through the use of hardware interrupts across a common voltage bus, multiple embedded Linux devices can be synchronized in virtual time. Each device runs a Linux kernel module that supports distributed virtual time for processes across devices. Through a use case, we demonstrate the usability of a distributed virtual-time-enabled testbed for evaluating the cyber-security of a voltage control application in the electric power grid. We also evaluate the difference between *periodic* and *dynamic* modes and implications on the use case. In the future, we will further evaluate cyber-physical system operations using the testbed supporting the distributed virtual time. Specifically, we would like to evaluate the impact of disruptions in the communication network for frequency adjustment control applications in the electric power grid as this is a cyber-physical system application that utilizes real-time sensing and actuation.

ACKNOWLEDGMENTS

We would like to thank Yuan-An Liu for his contributions and useful discussions.

REFERENCES

Ahmad T. Al-Hammouri. 2012. A comprehensive co-simulation platform for cyber-physical systems. *Comput. Commun.* 36, 1 (2012), 8–19. DOI: https://doi.org/10.1016/j.comcom.2012.01.003

Allwinner Technology Co., Ltd. 2013. *CoreLink GIC-400 Generic Interrupt Controller: Technical Reference Manual* (1.0 ed.). Allwinner Technology Co., Ltd. Retrieved from http://infocenter.arm.com/help/topic/com.arm.doc.ddi0471a/DDI0471A_gic400_r0p0_trm.pdf.

ARM. 2011. *A20 User Manual (r0p0 ed.).* ARM. Retrieved from http://dl.linux-sunxi.org/A20/A20 User Manual 2013-03-22.pdf.

C. D. Carothers. 2002. XSim: Real-time analytic parallel simulations. In *Proceedings 16th Workshop on Parallel and Distributed Simulation.* 23–30.

J. M. Carrasco, L. G. Franquelo, J. T. Bialasiewicz, E. Galvan, R. C. PortilloGuisado, M. A. M. Prats, J. I. Leon, and N. Moreno-Alfonso. 2006. Power-electronic systems for the grid integration of renewable energy sources: A survey. *IEEE Trans. Ind. Electron.* 53, 4 (June 2006), 1002–1016. DOI: https://doi.org/10.1109/TIE.2006.878356

Roger C. Dugan. 2013. Reference Guide, The Open Distribution System Simulator. https://sourceforge.net/projects/electricdss/.

M. A. Erazo, Yue Li, and J. Liu. 2009. SVEET! A scalable virtualized evaluation environment for TCP. In *Proceedings of the 2009 Testbeds and Research Infrastructures for the Development of Networks Communities and Workshops.* IEEE Computer Society, Washington, DC, 1–10.

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

Andreas Grau, Klaus Herrmann, and Kurt Rothermel. 2012. Scalable network emulation - The NET approach. *J. Commun.* 7, 1 (January 2012), 3–16.

Andreas Grau, Steffen Maier, Klaus Herrmann, and Kurt Rothermel. 2008. Time jails: A hybrid approach to scalable network emulation. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*. IEEE Computer Society, Washington, DC, 7–14.

Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. 2005. To infinity and beyond: Time warped network emulation. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*. ACM, New York, NY, 1–2.

Christopher Hannon and Jiaqi Yan. 2016. Kernel Module for enabling virtual time in distributed heterogeneous embedded Linux environment. https://github.com/JinLabIIT/EmbVT.

Christopher Hannon, Jiaqi Yan, and Dong Jin. 2016. DSSnet: A smart grid modeling platform combining electrical power distribution system simulation and software defined networking emulation. In *Proceedings of the 2016 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS'16)*. ACM, New York, NY, 131–142. DOI: https://doi.org/10.1145/2901378.2901383

Christopher Hannon, Jiaqi Yan, Dong Jin, Chen Chen, and Jianhui Wang. 2018. Combining simulation and emulation systems for smart grid planning and evaluation. *ACM Trans. Model. Comput. Simul.* 28, 4, Article 27 (Aug. 2018), 23 pages. DOI: https://doi.org/10.1145/3186318

Dong Jin, Yuhao Zheng, Huaiyu Zhu, David M. Nicol, and Lenhard Winterrowd. 2012a. Virtual time integration of emulation and parallel simulation. In *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*. IEEE Computer Society, 201–210.

Dong Jin, Yuhao Zheng, Huaiyu Zhu, David M. Nicol, and Lenhard Winterrowd. 2012b. Virtual time integration of emulation and parallel simulation. In *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*. IEEE Computer Society, Washington, DC, 201–210.

W. H. Kersting. 2001. Radial distribution test feeders. In *2001 IEEE Power Engineering Society Winter Meeting. Conference Proceedings (Cat. No.01CH37194)*, Vol. 2. 908–912. DOI: https://doi.org/10.1109/PESW.2001.916993

X. Koutsoukos, G. Karsai, A. Laszka, H. Neema, B. Potteiger, P. Volgyesi, Y. Vorobeychik, and J. Sztipanovits. 2018. SURE: A modeling and simulation integration platform for evaluation of secure and resilient cyber–physical systems. *Proc. IEEE* 106, 1 (2018), 93–112.

Jereme Lamps, Vladimir Adam, David M. Nicol, and Matthew Caesar. 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)*. ACM, New York, NY, 113–124. DOI: https://doi.org/10.1145/2769458.2769481

Jereme Lamps, Vignesh Babu, David M. Nicol, Vladimir Adam, and Rakesh Kumar. 2018. Temporal integration of emulation and network simulators on Linux multiprocessors. *ACM Trans. Model. Comput. Simul.* 28, 1, Article 1 (Jan. 2018), 25 pages. DOI: https://doi.org/10.1145/3154386

Jereme Lamps, David M. Nicol, and Matthew Caesar. 2014. TimeKeeper: A lightweight virtual time system for Linux. In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM PADS'14)*. ACM, New York, NY, 179–186. DOI: https://doi.org/10.1145/2601381.2601395

Bob Lantz, Brandon Heller, and Nick McKeown. 2010. *A Network in a Laptop: Rapid Prototyping for Software-Defined Networks*. ACM Press, 1–6. DOI: https://doi.org/10.1145/1868447.1868466 01341

Sathiya Kumaran Mani, Ramakrishnan Durairajan, Paul Barford, and Joel Sommers. 2018. An architecture for IoT clock synchronization. In *Proceedings of the 8th International Conference on the Internet of Things (IOT'18)*. ACM, New York, NY, Article 17, 8 pages. DOI: https://doi.org/10.1145/3277593.3277606

Davis Montenegro, Roger Dugan, Robert Henry, Tom McDermott, and wsunderm1. 2016. OpenDSS Program, SOURCEFORGE.NET. Retrieved January 2016, from http://sourceforge.net/projects/electricdss.

J. Nutaro and O. Ozmen. 2019. Using simulation to quantify the reliability of control software. In *2019 Winter Simulation Conference (WSC'19)*. 3267–3276.

Alessandro Pellegrini and Francesco Quaglia. 2017. A fine-grain time-sharing time warp system. *ACM Trans. Model. Comput. Simul.* 27, 2, Article 10 (May 2017), 25 pages. DOI: https://doi.org/10.1145/3013528

D. Pudjianto, C. Ramsay, and G. Strbac. 2007. Virtual power plant and system integration of distributed energy resources. *IET Renew. Power Gener.* 1, 1 (March 2007), 10–16. DOI: https://doi.org/10.1049/iet-rpg:20060023

Steven Rostedt. 2017. *ftrace - Function Tracer* (4.13 ed.). Red Hat Inc. Retrieved from https://www.kernel.org/doc/Documentation/trace/ftrace.txt

Pierluigi Siano. 2014. Demand response and smart grids–A survey. *Renew. Sustain. Energy Rev.* 30 (2014), 461–478. DOI: https://doi.org/10.1016/j.rser.2013.10.022

Emiliano Silvestri, Cristian Milia, Romolo Marotta, Alessandro Pellegrini, and Francesco Quaglia. 2020. Exploiting interprocessor-interrupts for virtual-time coordination in speculative parallel discrete event simulation. In *Proceedings of the 2020 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM-PADS'20)*. Association for Computing Machinery, New York, NY, 49–59. DOI: https://doi.org/10.1145/3384441.3395985

Dug Song. 2001. dsniff. Retrieved Janunary 21, 2019, from https://www.monkey.org/ dugsong/dsniff/.

Sudhir Srinivasan and Paul F. Reynolds. 1998. Elastic time. *ACM Trans. Model. Comput. Simul.* 8, 2 (April 1998), 103–139. DOI: https://doi.org/10.1145/280265.280267

Elias Weingärtner, Florian Schmidt, Hendrik Vom Lehn, Tobias Heer, and Klaus Wehrle. 2011. SliceTime: A platform for scalable and accurate network emulation. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association, Berkeley, CA, 253–266.

Jiaqi Yan and Dong Jin. 2015a. A virtual time system for Linux-container-based emulation of software-defined networks. In *Proceedings of the 3rd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (SIGSIM PADS'15)*. ACM, New York, NY, 235–246. DOI: https://doi.org/10.1145/2769458.2769480

Jiaqi Yan and Dong Jin. 2015b. VT-Mininet: Virtual-time-enabled mininet for scalable and accurate software-define network emulation. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR'15)*. ACM, New York, NY, Article 27, 7 pages. DOI: https://doi.org/10.1145/2774993.2775012

Özgür Özmen, James Nutaro, Michael Starke, Jeffrey Munk, Larry Roberts, Xiao Kou, Piljae Im, Jin Dong, Fangxing Li, Teja Kuruganti, and Helia Zandi. 2020. Power grid simulation testbed for transactive energy management systems. *Sustainability* 12 (2020), 4402. DOI: https://doi.org/10.3390/su12114402

Received February 2020; revised August 2020; accepted December 2020