Combining Simulation and Emulation Systems for Smart Grid Planning and Evaluation

CHRISTOPHER HANNON, JIAQI YAN, and DONG JIN, Illinois Institute of Technology
CHEN CHEN and JIANHUI WANG, Argonne National Laboratory

Software-defined networking (SDN) enables efficient network management. As the technology matures, utilities are looking to integrate those benefits to their operations technology (OT) networks. To help the community to better understand and evaluate the effects of such integration, we develop DSSnet, a testing platform that combines a power distribution system simulator and an SDN-based network emulator for smart grid planning and evaluation. DSSnet relies on a container-based virtual time system to achieve efficient synchronization between the simulation and emulation systems. To enhance the system scalability and usability, we extend DSSnet to support a distributed controller environment. To enhance system fidelity, we extend the virtual time system to support kernel-based switches. We also evaluate the system performance of DSSnet and demonstrate the usability of DSSnet with a resilient demand response application case study.

CCS Concepts: • **Computing methodologies** → **Simulation environments**; *Simulation tools*; • **Networks** → Network design principles;

Additional Key Words and Phrases: Electrical power system simulation, network emulation, software-defined networking, smart grid

ACM Reference format:
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 (August 2018), 23 pages.
https://doi.org/10.1145/3186318

1 INTRODUCTION

The growth in grid modernization highly relies on the successful convergence of a utility's operations technology (OT) networks and advanced information technology (IT) systems. Applying software-defined networking (SDN) technology to OT networks is an emerging research topic with the goal of boosting control efficiency and attaining secure and reliable operation.

This article 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, J. Yan, and D. Jin, Computer Science Department, Illinois Institute of Technology, 10 w 31st street, Chicago, IL 60616; emails: {channon, jyan31}@hawk.iit.edu, dong.jin@iit.edu; C. Chen, Center for Energy, Environmental, and Economic Systems Analysis (CEEESA), Energy Systems Division, Argonne National Laboratory, 9700 Cass Ave, Lemont, IL 60439; email: morningchen@anl.gov; J. Wang, Associate Professor, Department of Electrical Engineering, Lyle School of Engineering, Southern Methodist University, Dallas, TX 75275; email: jianhui@smu.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Association for Computing Machinery.
1049-3301/2018/08-ART27 $15.00
https://doi.org/10.1145/3186318

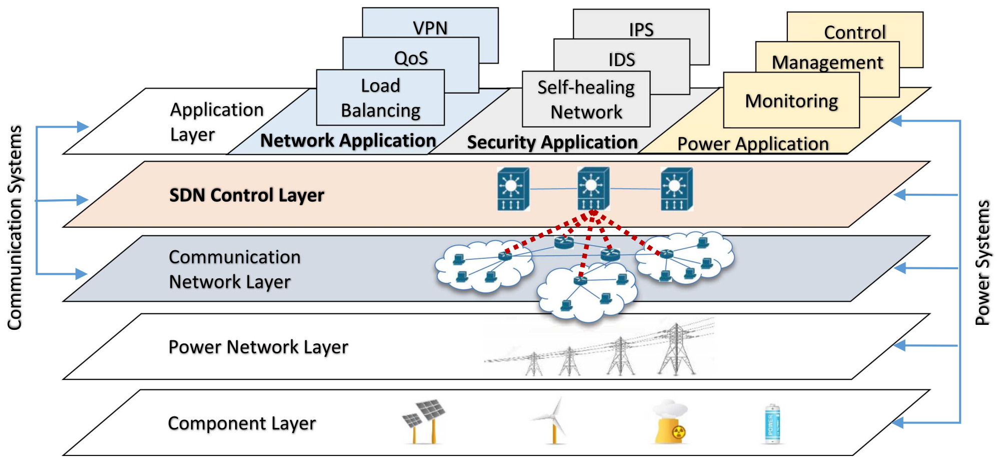


Fig. 1. A multilayered SDN-enabled smart grid.

Recent works include applications in substation automation Cahn et al. (2013), Molina et al. (2015), reliability evaluation Pfeiffenberger and Du (2014), quality of service (QoS) optimization Sydney et al. (2014), Goodney et al. (2013), Dong et al. (2015), fast failover mechanism Sydney et al. (2014), Molina et al. (2015), resilience enhancement Ren et al. (2017), evaluation under SDN controller failure Ghosh et al. (2016), and an energy sector SDN-enabled Ethernet switch Schweitzer Engineering Laboratories (2014).

Figure 1 depicts a smart grid with an SDN-based communication infrastructure. SDN is a programmable, open-source approach to designing, building, and managing networks onf (2015). It decouples network control from forwarding logic in network devices and offloads its functions to logically centralized SDN controller(s). SDN offers global network visibility, which enables detailed system virtualization; offers direct network programmability, which facilitates efficient network management and traffic engineering; and offers centralized control of communication networks, which would be integrated with existing power grid control applications to allow more intelligent security and network applications to blossom, such as system-wide configuration verification and context-aware detection systems. However, incorporating SDN into critical control systems is very challenging. Challenges include the following: (1) the continuous operation of physical processes is extremely important, (2) such systems must operate to meet real-time deadlines, and (3) new designs must integrate with many resource-constrained legacy devices. Therefore, a high-fidelity and scalable testing platform is needed to evaluate SDN-based technologies and their impact on the power grid systems before deployment on real systems.

We present the **D**istribution **S**ystem **S**olver **Net**work (DSSnet), a testbed incorporating an electrical power distribution system simulator and an SDN-based communication network emulator Hannon et al. (2016). Traditional emulators execute real programs with reference to the system clock to advance experiments, while simulators execute models to advance experiments with reference to the simulation's virtual clock. To ensure the correct and efficient synchronization of time and events between the two systems, DSSnet leverages a prior container-based virtual time system in Linux kernel Yan and Jin (2015a). Previous work concentrates on the development of virtual time for the hosts or containers of the emulation and the synchronization of simulation and emulation systems. In this article, we enhance the scalability, fidelity, and usability of DSSnet in order to provide a feature-full testbed and smart grid development platform in order to enable smart grid researchers to access better tools for planning and evaluation of future grid infrastructures.

- We integrate the Open Networking Operating System (ONOS) Berde et al. (2014) to manage the communication network. ONOS supports a distributed controller environment, which significantly increases the size of network experiments that one can emulate. This feature also enables the study of resilient controllers, a key component for deploying an SDN-enabled smart grid in practice. ONOS also provides many unique features to enhance the usability of DSSnet, such as host-to-host intents, multipath forwarding, reactive forwarding, and traffic statistic monitoring. We extend the virtual time system in DSSnet to provide an API to enable such integration. We also utilize ONOS's native GUI for customization and visualization of an experimental network.
- We extend the virtual time support to Open vSwitch Pfaff et al. (2015), the underlying SDN switch implementation in Mininet, for fidelity and performance improvement of DSSnet. This feature ensures the consistency among all components of DSSnet using virtual time. We demonstrate the improvement in fidelity with forwarding applications running on the SDN switches, such as spanning tree protocols (STP and RSTP), as well as the effect of STP on power systems (see Section 5). Additionally, we extend DSSnet with the capability of real-time process monitoring of all the virtual-time-specific states.
- We perform fidelity evaluation of the enhanced DSSnet. We also demonstrate the usability of DSSnet with a new case study analyzing the effectiveness of a demand response application in an SDN-enabled network undergoing link failure.

In the remainder of the article, Section 2 presents related work and shows the differences of DSSnet with those existing tools. Section 3 describes the system design and how it addresses the synchronization challenges across the two core systems. Section 4 presents the virtual time system implementation. Section 5 evaluates the system performance. Section 6 demonstrates a demand response application that illustrates the features and benefits of DSSnet. Finally, Section 7 concludes the article with future work.

2 MOTIVATION AND RELATED WORK

2.1 Combining Power with Communication

The power grid is composed of power generation, transmission, distribution, and loads. Traditionally, power is generated in mass quantities from hydro, coal, nuclear, and gas sources. The power is then transmitted at high voltages to distribution systems, where the power is distributed to residential and commercial consumers. As the power grid is moving toward a smarter grid, efficient energy management is increasingly dependent on the underlying communication network supporting reliable information transfer among the various entities in the grid.

With distributed power generation—such as solar and wind energy—and more storage technology, there is a need for understanding the state of the power network in real time. A challenge with the integration of such generation is the uncertainty and intermittency of the availability of power generation. In order to combat this challenge, there needs to be an infrastructure that allows for the monitoring and control of the system state. Doing this effectively requires a reliable and resilient communication network.

Researchers have developed systems to cosimulate the power and network components of the smart grid Godfrey et al. (2010), Montenegro et al. (2012), Dufour and Belanger (2014), Lin et al. (2012), Hopkinson et al. (2006), Ciraci et al. (2014a, 2014b). Mets et al. (2014) survey the existing technologies and motivations for cosimulation.

Montenegro et al. (2012) propose a system using OpenDSS to allow for sending real-time signals to hardware integrating with electric power simulation. Real-time simulators are used for hardware-in-the-loop simulations, allowing for simulation-emulation closer to the real system

Dufour and Belanger (2014). This gives high fidelity but requires power equipment and often specific simulator hardware. Using a network emulator, we make the system closer to that of real hardware deployment, but without the cost or complexity associated with real hardware.

Lin et al. (2012) create a cosimulation between PSLF and ns-2. They use a global event-driven mechanism for sending synchronization messages between the two simulators. In simulation, events are sorted by timestamps, typically in a priority queue. To enforce the temporal order of events, we take inspiration from the global event queue and adapt this strategy to integrate the network emulation with the distributed power simulation in DSSnet.

EPOCHS Hopkinson et al. (2006) uses commercial power simulators to cosimulate network and power systems through the use of agents. This platform uses agents to effectively cosimulate power and communication elements. The authors define agents as having the properties of autonomy and interaction and exhibit properties of mobility, intelligence, adaptivity, and communication. In DSSnet, our models run real processes in the network emulation. This allows us to make use of agents as entities that exist in both systems.

FNCS Ciraci et al. (2014b) is a federated approach for cosimulation of power and electrical simulators by combining multiple power simulators, both distribution and transmission, and use ns-3 as a communication simulator. Ciraci et al. (2014a) improve the synchronization between systems, which we take inspiration from in our implementation in Section 4. The difference is that DSSnet is focused on network emulation, which has different synchronization challenges due to the inherent difference between the execution mechanisms in simulation and emulation.

There are two main features that set our design apart from the existing tools. The first is that we are using a network emulator rather than a simulator. The emulator allows for higher fidelity by executing real networking programs. To the best of our knowledge, our testbed is the first to combine electric power simulation and communication network emulation. The second is that our network emulator supports SDN-based networks.

2.2 Software-Defined Networking in Utility

SDN is an emerging network technology that separates the data plane from the control plane. The benefit of this is the enhanced ability to have a global view over the network and be able to program network switches to provide functions that were previously too laborious and impossible to do. SDN allows for complex network functions to be created by adjusting network paths and flows in real time—reactively and proactively. This technology can help solve security issues and increase performance in many networks such as data centers, and even in energy infrastructure. However, SDN is not widely used yet and does not solve all problems out of the box.

Ren et al. (2017) utilize a hardware-in-the-loop testbed to evaluate an SDN framework developed for a microgrid. The limitation of their work is in the synchronization of simulation to hardware and scalability associated with hardware requirements. We aim to create a purely software-based approach to enable scalable, customizable, and fast development of smart grid SDN applications.

In Kim et al. (2014), SDN is proposed to allow for scalable deployment of utility applications. The authors show how SDN can provide network functions to simplify publisher-subscriber roles in intelligent electrical devices (IEDs), including in phasor measurement unit networks; however, this work is limited to communication networks.

Dong et al. (2015) propose a system that combines an SDN emulator with an off-the-shelf high-voltage solver. The difference between the system they propose and ours is that we are focused on combining open-source tools and that our simulator is for low-voltage distribution networks.

In Goodney et al. (2013), SDN is utilized to increase the performance of SCADA networks. In our testbed, we have also modeled SCADA network elements, which can be used to explore how cyber attacks can impact the power grid using different communication models.

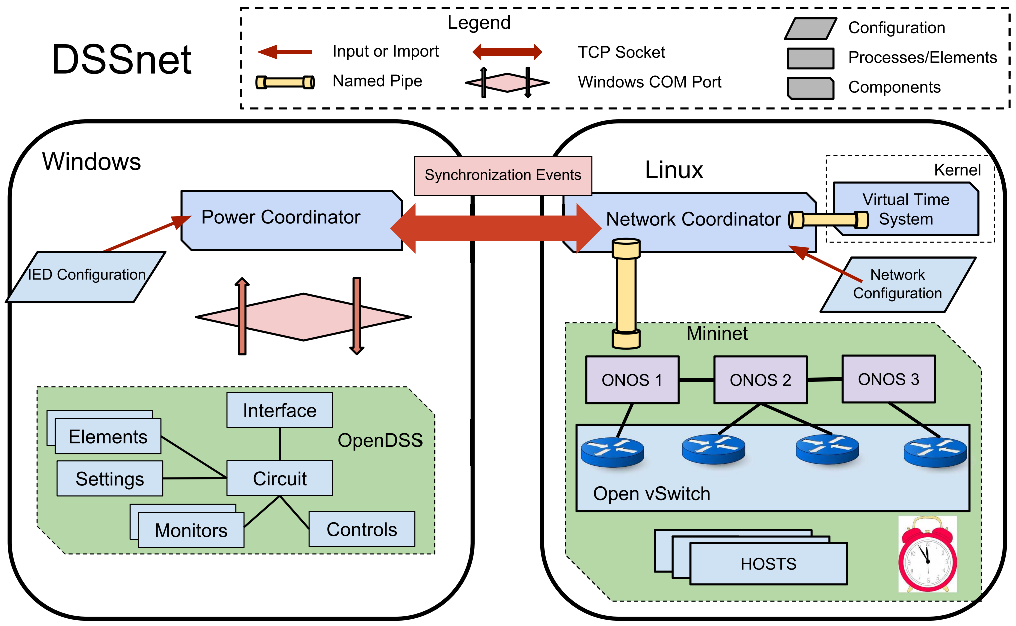


Fig. 2. DSSnet system architecture diagram. Note that the power simulator runs on a Windows machine and the network emulator runs on a Linux machine.

Sydney et al. (2014) analyze utility communication networks for situational awareness including during blackouts. Through the use of a hybrid power and communication system, situational awareness can be enhanced to increase the resilience of the grid.

Additionally, there has been work to bring existing power grid network protocols such as GOOSE and IEC 61850 into SDN networks Molina et al. (2015). Our testbed can be used to emulate IEC 61850-based communication with the advantage of analyzing the effects in the power simulator.

To summarize, our system is built on top of a network emulator rather than the existing works of network simulation for high-fidelity analysis in the context of the smart grid, and the emulator we use supports SDN-enabled software switches and protocols.

3 SYSTEM DESIGN

DSSnet integrates a distribution power system simulator, OpenDSS Montenegro et al. (2016), with a network emulator, Mininet Lantz et al. (2010), using virtual time. The system has the following features:

- Power Flow Studies
- SDN-Based Communication Network Modeling
- Smart Grid Control Applications
- ONOS-Enabled Controller Integration
- Virtual-Time-Enabled Network Emulation

DSSnet is composed of five main components: the communication network emulator, the electrical power simulator, a network coordinator for interfacing with the network and the virtual time system, a power coordinator for interfacing and controlling the simulator, and a virtual time system that manages time and ensures synchronization in DSSnet. Figure 2 depicts the architecture of DSSnet with an emphasis on the extended features from our previous work Hannon et al. (2016).

3.1 System Design Architecture

3.1.1 System Components. The network emulator in DSSnet contains software switches that emulate the function of real SDN switches. In DSSnet, the hosts represent IEDs in a power network, and each host has its own virtual network ports. Hosts in the emulation have their own namespaces Lantz et al. (2010) and can run real processes to model IEDs. Any element in the power network that has a communication requirement is modeled in the emulation, including SCADA elements, such as sensors and phasor measurement units (PMUs), and even relays and generators. In the network, some hosts interact with the simulator indirectly through other models, such as data collection and storage systems, state estimation applications, and voltage and frequency adjustment controllers.

DSSnet models define the power network through elements, such as lines, transformers, relays, meters (sensors), loads, capacitors, and generators. The purpose of the power simulation is to simulate the behavior of utility distribution systems. Functions of the power simulator include power flow snapshot, harmonic study, fault studies, load modeling, and solving dynamic time-step power flow Dugan (2013).

The network topology and the IED models are loaded through the network coordinator to configure network properties. The role of the network coordinator is also to interface with the network emulator for synchronization between the communication and power systems. When the coordinator receives a synchronization request, it interfaces with the power coordinator and with the virtual time system to control DSSnet's virtual clock. Start-up options for the network coordinator include controller selection, virtual time options, paths for configuration files, Open vSwitch options for emulated switches, and logging and debugging options.

The power coordinator interfaces with the power simulator and the network coordinator. The power coordinator provides an API to modify and extract values in the simulator. A role of the module is to advance the simulation clock to the timestamp of the current event request from the emulator, and to solve the power flow at that time. Currently, the power coordinator supports dynamic loads, generators, and energy storage devices through configuration files. Additionally, DSSnet supports controllable loads, generators, energy storage devices, and relays. The API also provides support for faults, sensor readings of voltage, current, and power values. The power coordinator also supports trace-based simulation from predefined synchronization events, allowing for debugging, testing, and replays for previous DSSnet runs.

Switches in the emulation are modeled with Open vSwitch Pfaff et al. (2015). Open vSwitch is a software switch that runs on top of the networking hardware. It consists of a user-space server, a database (ovs-db) maintaining settings and configurations, and a kernel module for performance enhancement. Open vSwitch creates a *bridge* for each switch in the emulation. The ports of the emulated switch as well as configuration are maintained by the ovs-db. When a packet is sent between hosts, the Open vSwitch server processes the packet to establish a micro-flow in the kernel module, querying the ovs-db if necessary. The kernel module ensures that subsequent packets will be processed in the kernel (while the flow exists), thus avoiding laborious context switching. The emulation essentially provides a high-level abstraction to configuring bridges and namespaces for switches and hosts.

There are drawbacks to using emulation. With each host running its own processes and having its own virtual network adapter, the system becomes more complex, making debugging a challenge. In order to make debugging easier, we create a GUI tool extended from htop presented in the following section. Most importantly, an emulation system cannot scale as large as a simulation system can due to the virtualization of hosts that require many resources. Our future work includes the development of distributed emulation to achieve better scalability, with reference to a prior work on the distributed OpenVZ-based network emulator Zheng et al. (2013).

3.1.2 ONOS. Extended from our previous work, DSSnet now incorporates ONOS Berde et al. (2014), the Open Network Operating System. In order to integrate ONOS into DSSnet, we emulate both the data plane network that consists of the hosts in DSSnet and the switches that connect them, as well as the control plane that consists of multiple ONOS instances. Each ONOS instance maintains the same state. Intercontroller communication through the ONOS design enforces this consistent state. Therefore, the distributed SDN controller environment can be treated as a single entity from a logical control standpoint, allowing for scalability in processing requests from many switches, as well as reliability and resiliency in case of controller failure.

ONOS provides many features that enable DSSnet to better evaluate the SDN's role in the power grid. ONOS includes a distributed controller environment, which enables scalability in managing many switches, and provides resiliency by allocating backup controllers. Additionally, ONOS provides many out-of-the-box applications, such as host-to-host intents, reactive forwarding, and flow statistic collection. The ONOS GUI further helps to debug and configure DSSnet's emulator component.

3.1.3 Virtual Time System. Unlike simulation, the emulation clock elapses with the real wall clock. Therefore, pausing the emulation requires more than just stopping the execution of the emulated entities, but also the pausing their clocks. Virtual time can be used to achieve this goal Lamps et al. (2014), Yan and Jin (2015a). We choose to extend the work of Yan and Jin (2015b), in which virtual time support is included in Mininet. However, their motivation is different from ours.

In general, virtual time has at least two categories of application. The first one is to slow down emulation so that it appears to emulate entities that have sufficient *virtual* resources. Slowing down execution also alleviates the problems caused by resource multiplexing. The work in Yan and Jin (2015b) and Yan and Jin (2015a) fall into this category. Another usage of virtual time is for emulation-simulation synchronization. In DSSnet, we assign every container a private clock, instead of using the global time provided by the Linux OS. In addition, we assign virtual clocks for Open vSwitch processes. The containers now have the flexibility to slow down, speed up, or stop their own clock when synchronizing with the simulator.

However, the emulator needs to manage the consistency across all containers. This is achieved by a centralized timekeeper in Lamps et al. (2014), and by a two-layer consistency mechanism in Yan and Jin (2015a). A more flexible virtual time system implemented by Yan and Jin (2015a) avoids this problem as emulation takes charge of this responsibility. In practice, the emulator configuration guarantees that all containers are running with one shared virtual clock. Similarly, the container leverages the Linux process hierarchy to guarantee that all the applications inside the container are using the same virtual clock. The two-layer consistency approach is well suited to this work for pausing and resuming because

(1) all hosts, controllers, and switches should be paused or resumed when we stop or restart the emulation, and
(2) all applications running on a host should be paused or resumed when we stop or restart the emulation.

The first task is done by the network coordinator. The second task is implemented based on the fact that processes inside a container belong to the same process group.

3.2 Synchronization

A key challenge in DSSnet is the synchronization between connecting the emulated communication network and the simulated power system. The root cause is that two different clock systems

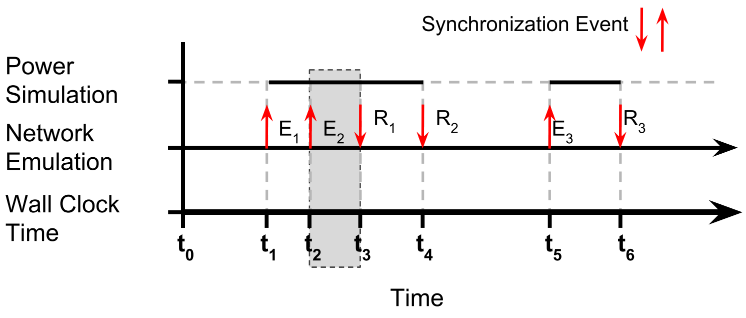


Fig. 3. The execution of DSSnet is shown with respect to the wall clock. The network emulation runs concurrently with the power simulation and is not paused, which allows for synchronization errors to occur when requests arrive before the responses are sent; e.g., R_1 occurs after E_2. The shaded box highlights the location of the error.

are used to advance experiments. Ordinary virtual-machine-based network emulators use the system clock, and a simulator often uses its own virtual clock. This difference would lead to causality errors as shown in the following example.

In Figure 3, there are three cross-system events (E_i), each with a response (R_i). E_1 occurs before E_2; however, E_2 may require information from R_1. Since the response occurs after the second event, the global causality is violated, and thus reduces experiment fidelity. An example of E_1 is a request to retrieve power flow values while E_2 sets the value of a discharging battery based on the value returned previously. Since the reply R_1 occurs after E_2, this can introduce an error. Furthermore, such errors can be accumulated if the simulation remains out of synchronization with the emulation.

To address this issue, we develop a virtual time system in Mininet with the new capability to pause the emulator without advancing the emulation virtual clock while the simulator is running. We adopt this idea, since the experiment advancement in DSSnet is designed to be driven by the emulation. This means that synchronization requests are originated at the emulator. Before the coordinators permit the simulator to advance over a time interval [a,b), we first ensure that all processes in the emulator have advanced their own clocks to at least time b, to ensure that all input traffic that arrives at the simulator with timestamps in [a,b) are obtained first. The emulation system is driven by a virtual system clock; therefore, the simulation system will not advance past the virtual clock time of the emulation. While the inverse does not hold, the emulation is permitted to advance beyond the simulation. This is acceptable because all synchronization events are originated from the emulator.

Figure 4 shows the execution of the DSSnet. The total execution time (Equation (1)) is the total time the emulation is running plus the sum of the time spent executing the simulation. DSSnet's clock, the overall clock of both systems (Equation (2)), is equal to the total time of the emulation plus the sum of the returned simulation virtual times. In this illustrative scenario, we do not include factors like synchronization overhead, parallel execution based on simulation and (possibly) emulation look-ahead, and time dilation effects in emulation virtual time, for simplicity.

$$Time_{wall\_clock} = \sum t_{E\_i} + \sum t_{S\_i} \tag{1}$$

$$Time_{DSSnet} = \sum t_{E\_i} + \sum t_{S'\_i} \tag{2}$$

$$ret = \frac{t_{S'\_i}}{t_{S\_i}}, \tag{3}$$

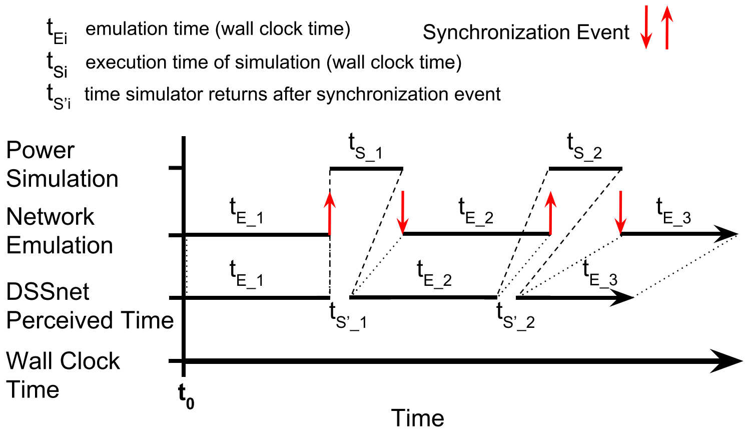


Fig. 4. The execution of DSSnet is shown with respect to its own perceived time, i.e., the sum of the emulation execution time (can be dilated or not dilated) and the virtual time elapsed in simulation. The network emulation is paused to allow for the simulation to catch up to the emulation—this also ensures synchronization errors in the previous example do not occur.

where *ret*'s value range is

— (1,∞) if the power simulation takes a longer time to execute than the real time. Thus, emulation virtual time is essential for synchronizing the two systems;
— (0,1] if the power simulation takes less or equal time to execute than the real time, i.e., with real-time simulation capability; and
— 0 if the power simulation time is not considered by the emulation, for instance, recomputing voltage and current change along power lines at nearly light speed.

Synchronization events occur when either system influences the other. One optimization is to divide the global queue into two queues, because synchronization events can be created in two ways: onblocking events and blocking events. For each type of event, we design a queue sorted by timestamps to organize the requests. The nonblocking event queue contains premeditated synchronization events and events that do not require a response to the communication network. For example, the nonblocking event queue can be used to pass messages to the simulation to sample the power flows with meters at periodic intervals. Other examples are power events such as line faults that occur at a specific time. Nonblocking events enable the power simulator and communication network emulator to run in parallel as the event is defined not to have a direct impact on the communication network. The IEDs are able to influence the power simulation by sending a synchronization event message using the blocking queue. Examples of these classes of synchronization events are PMUs requesting values from the power simulation and controllable loads changing power values or turning on or off.

We have revisited the design of our synchronization queues. Because many networked power devices (e.g., sensors) frequently synchronize at the same time in relation to each other, there are often multiple events arriving simultaneously into the queues. When this occurs, we synchronize these events as a batch to the power coordinator. This reduces the overhead of blocking events and prevents the unnecessary overhead of pausing and resuming the emulation.

By using the nonblocking event queue, we can speed up the overall execution time. In other words, we do not need to pause the emulation for nonblocking events (E_1 and E_3 in Figure 5). However, if a blocking synchronization event (E_2 in Figure 5) occurs before the response R_1, then

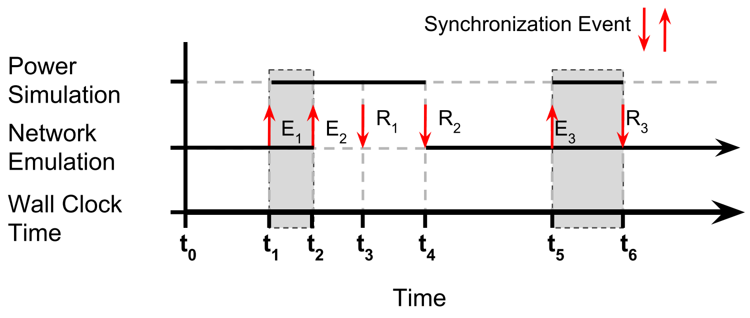


Fig. 5. E_1 and E_3 are nonblocking synchronization events and E_2 is a blocking synchronization event from an IED. The network emulation is not paused unless an event in the blocking queue occurs, i.e., the one that requires a response to the communication network. The shaded box represents the portion of the experiment that is running in parallel.

the emulation is paused at t_2, i.e., the timestamp of E_2. The emulation is resumed at t_4, when response R_2 is returned. In this work, we demonstrate the advantage of having a nonblocking queue with sample events. How to classify the events is not a focus for this article. In addition, the container-based emulation system introduces opportunities for offering real application-specific look-aheads to improve the parallelism performance, which we will explore as our future work.

4 IMPLEMENTATION

DSSnet combines OpenDSS, an electrical power distribution system solver simulator; Mininet, an SDN emulator; and ONOS, a distributed SDN controller with the use of a virtual time system. This section mainly presents implementation details of the virtual time system, which integrate the simulation and emulation systems.

We extend our prior work on time-dilation-enabled Mininet Yan and Jin (2015b) with the capability to pause and resume emulation. The emulation clock runs x times slower than the wall clock during the experiment execution with a time dilation factor of x. To address the synchronization problem discussed in Section 3, we develop two routines *freeze* and *unfreeze* in the virtual time system. Specifically, we utilize the signal mechanism in the Linux kernel to stop both the process execution and the timing information passage.

4.1 Freeze/Unfreeze Implementation

To track virtual time using the OS software clock, we add several new fields into the process descriptor `task_struct`, including the following:

— `freeze_start_ns` represents the start time to freeze a process.
— `freeze_past_ns` represents the cumulative frozen time of a process.
— `physical_past_ns` represents the elapsed wall clock time.
— `virtual_past_ns` represents the elapsed virtual time.

The algorithms to dilate time and freeze/unfreeze processes are presented in Yan and Jin (2015a) and Hannon et al. (2016). In this article, we made several changes so that freezing and time dilation work simultaneously. Algorithm 1 shows the procedure to enable, disable, and update the time dilation. A trick scenario is that when both the old and new time dilation factor (TDF) have nonzero values, setting TDF may occur during the emulation execution. To handle this scenario, we implement `SwitchDilationTimekeeping` to calculate the wall clock time offset by subtracting both the physical elapsed time (line 4) and the elapsed time that a process is in the frozen state (line 5).

After dilating the time offset with the old time dilation factor, the system resets the starting point of both the virtual time and the physical time.

ALGORITHM 1: Set Time Dilation Factor

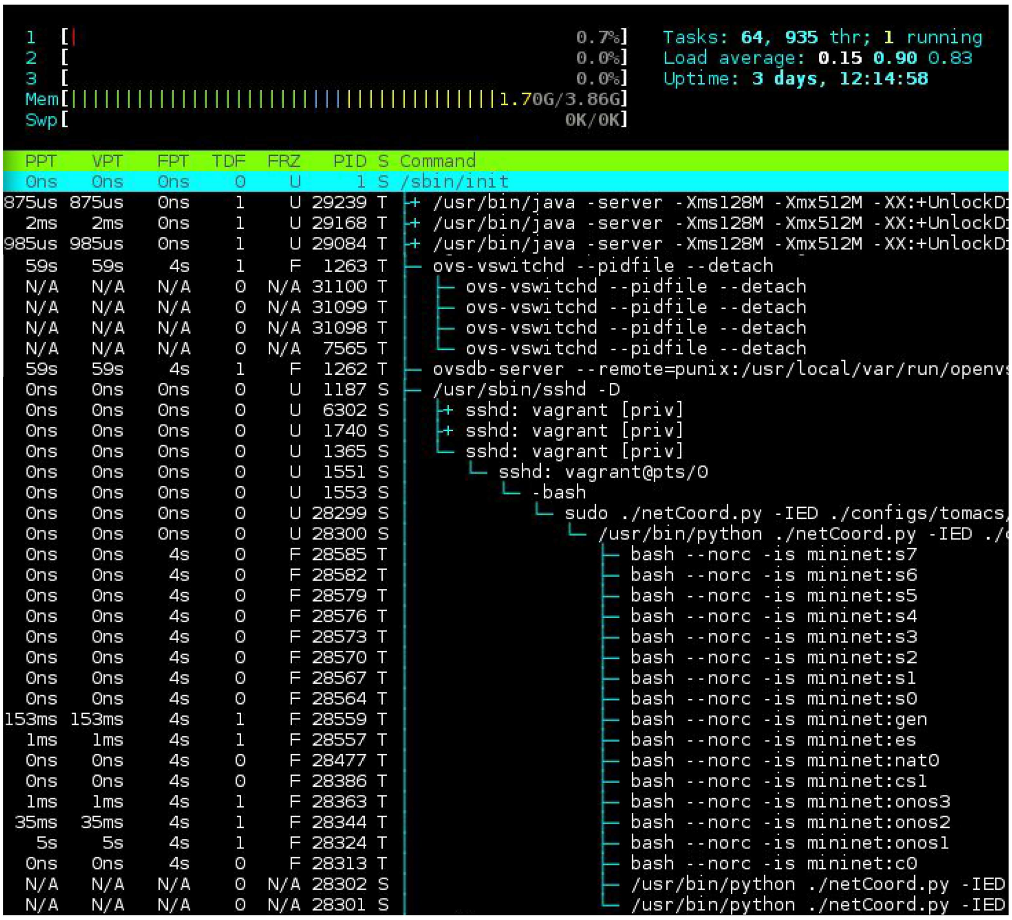
```
Data: Process $tsk$, time dilation factor $tdf$
Function SwitchDilationTimekeeping($tsk, old\_tdf, new\_tdf$)
    $now \leftarrow$ wall clock time in nanoseconds
    $\Delta_{ppn} \leftarrow now - tsk.physical\_start\_nsec$
    $\Delta_{ppn} \leftarrow \Delta_{ppn} - tsk.physical\_past\_nsec$
    $\Delta_{ppn} \leftarrow \Delta_{ppn} - tsk.freeze\_past\_nsec$
    $\Delta_{vpn} \leftarrow \Delta_{ppn} / old\_tdf$
    $tsk.virtual\_past\_ns \leftarrow tsk.virtual\_past\_ns + \Delta_{vpn}$
    $tsk.virtual\_start\_ns \leftarrow now$
    $tsk.physical\_start\_ns \leftarrow now$
    $tsk.physical\_past\_ns \leftarrow 0$
    $tsk.dilation \leftarrow new\_tdf$
return

Function SetDilation($tsk, new\_tdf$)
    $old\_tdf \leftarrow tsk.dilation$
    if $new\_tdf = old\_tdf$ then
        return 0
    else if $old\_tdf = 0$ then
        InitVirtualTime($tsk, new\_tdf$)
    else if $new\_tdf = 0$ then
        ExitVirtualTime($tsk$)
    else if $new\_tdf > 0$ then
        SwitchDilationTimekeeping($tsk, old\_tdf, new\_tdf$)
    else
        return -EINVAL
    end
    foreach child of tsk do
        SetDilation($child, new\_tdf$)
    end
return
```

4.2 User-Space Interface to Virtual Time

The virtual file system provides an interface between the kernel and the user space and is an ideal location to place the interface to virtual time. Under each process's directory in `/proc`, we expose the virtual-time-related variables via the following file entries:

- `/proc/$pid/dilation`. This entry allows us to enable/disable the virtual time of a process and to set the TDF value.
- `/proc/$pid/freeze`. This entry allows us to freeze/unfreeze a process `$pid`. It also indicates whether a process is frozen.
- `/proc/$pid/fpt`, `vpt`, and `ppt`. The three entries indicate the values of `freeze_past_ns`, `virtual_past_ns`, and `physical_past_ns` of a process.



```
  1  [|                                                  0.7%]   Tasks: 64, 935 thr; 1 running
  2  [                                                   0.0%]   Load average: 0.15 0.90 0.83
  3  [                                                   0.0%]   Uptime: 3 days, 12:14:58
  Mem[||||||||||||||||||||||||||||||||||||||||||||1.70G/3.86G]
  Swp[                                                  0K/0K]

  PPT    VPT    FPT  TDF  FRZ    PID S Command
  0ns    0ns    0ns    0    U      1 S /sbin/init
875us  875us    0ns    1    U  29239 T ├+ /usr/bin/java -server -Xms128M -Xmx512M -XX:+UnlockD
  2ms    2ms    0ns    1    U  29168 T ├+ /usr/bin/java -server -Xms128M -Xmx512M -XX:+UnlockD
985us  985us    0ns    1    U  29084 T ├+ /usr/bin/java -server -Xms128M -Xmx512M -XX:+UnlockD
  59s    59s     4s    1    F   1263 T ├─ ovs-vswitchd --pidfile --detach
  N/A    N/A    N/A    0  N/A  31100 T │  ├─ ovs-vswitchd --pidfile --detach
  N/A    N/A    N/A    0  N/A  31099 T │  ├─ ovs-vswitchd --pidfile --detach
  N/A    N/A    N/A    0  N/A  31098 T │  ├─ ovs-vswitchd --pidfile --detach
  N/A    N/A    N/A    0  N/A   7565 T │  └─ ovs-vswitchd --pidfile --detach
  59s    59s     4s    1    F   1262 T ├─ ovsdb-server --remote=punix:/usr/local/var/run/openv
  0ns    0ns    0ns    0    U   1187 S ├─ /usr/sbin/sshd -D
  0ns    0ns    0ns    0    U   6302 S │  ├+ sshd: vagrant [priv]
  0ns    0ns    0ns    0    U   1740 S │  ├+ sshd: vagrant [priv]
  0ns    0ns    0ns    0    U   1365 S │  └─ sshd: vagrant [priv]
  0ns    0ns    0ns    0    U   1551 S │     └─ sshd: vagrant@pts/0
  0ns    0ns    0ns    0    U   1553 S │        └─ -bash
  0ns    0ns    0ns    0    U  28299 S │           └─ sudo ./netCoord.py -IED ./configs/tomacs
  0ns    0ns    0ns    0    U  28300 S │              └─ /usr/bin/python ./netCoord.py -IED ./
  0ns    0ns     4s    0    F  28585 T │                 ├─ bash --norc -is mininet:s7
  0ns    0ns     4s    0    F  28582 T │                 ├─ bash --norc -is mininet:s6
  0ns    0ns     4s    0    F  28579 T │                 ├─ bash --norc -is mininet:s5
  0ns    0ns     4s    0    F  28576 T │                 ├─ bash --norc -is mininet:s4
  0ns    0ns     4s    0    F  28573 T │                 ├─ bash --norc -is mininet:s3
  0ns    0ns     4s    0    F  28570 T │                 ├─ bash --norc -is mininet:s2
  0ns    0ns     4s    0    F  28567 T │                 ├─ bash --norc -is mininet:s1
  0ns    0ns     4s    0    F  28564 T │                 ├─ bash --norc -is mininet:s0
153ms  153ms     4s    1    F  28559 T │                 ├─ bash --norc -is mininet:gen
  1ms    1ms     4s    1    F  28557 T │                 ├─ bash --norc -is mininet:es
  0ns    0ns     4s    0    F  28477 T │                 ├─ bash --norc -is mininet:nat0
  0ns    0ns     4s    0    F  28386 T │                 ├─ bash --norc -is mininet:cs1
  1ms    1ms     4s    1    F  28363 T │                 ├─ bash --norc -is mininet:onos3
 35ms   35ms     4s    1    F  28344 T │                 ├─ bash --norc -is mininet:onos2
   5s     5s     4s    1    F  28324 T │                 ├─ bash --norc -is mininet:onos1
  0ns    0ns     4s    0    F  28313 T │                 ├─ bash --norc -is mininet:c0
  N/A    N/A    N/A    0  N/A  28302 S │                 ├─ /usr/bin/python ./netCoord.py -IED
  N/A    N/A    N/A    0  N/A  28301 S │                 └─ /usr/bin/python ./netCoord.py -IED
```

Fig. 6. vthtop: Real-time monitoring of per-process virtual-time-related information.

We make a clear distinction between regular processes and virtual-time-enabled processes. The `/proc/$pid/freeze` entry is valid only if `/proc/$pid/dilation` already has a nonzero value. The emulator can place a container in virtual time by writing 1,000 to `/proc/$pid/dilation`. This turns on the per-container freezing capability without unnecessarily modifying the clock speed.

DSSnet frequently queries the virtual time variable entries for debugging or monitoring purposes. We extended *htop* Muhammad (2016), a popular process viewing tool for Unix-family systems, to monitor all the virtual-time-related variables in real time, including the physical past time (PPT), virtual past time (VPT), freeze past time (FPT), time dilation factor (TDF), and freeze status (FRZ) of each process, as depicted in Figure 6. This tool can verify whether a set of virtual hosts share the same TDF to ensure that their virtual time is consistent (e.g., all the frozen processes in Figure 6 has the same FPT). In the case of synchronization events, FPT offers a good estimate of the synchronization overhead. Users can log the resource utilization information (e.g., CPU usage and IO rate) as well as the virtual clock information. The tool also enables us to adjust TDF when the emulation system load is too heavy to preserve the temporal fidelity. We have open-sourced the virtual time monitoring tool to facilitate the usage of the virtual time system Yan (2016).

In addition, we develop two user-space utility programs `freeze_all_procs` and `dilate_all_proc` for freezing and time-dilating processes. Both programs support parallel execution. `freeze_all_procs` spawns one `pthread` for each network host and writes the freezing/unfreezing flag to the `/proc/$pid/freeze` entry. This optimization significantly reduces the execution overhead in large-scale network settings. Users can also create their proxies in Mininet by referencing `net.dilateEmulation()` and `net.freezeEmulation()`.

4.3 Open vSwitch in Virtual Time

In order to guarantee that the emulated switches are embedded in virtual time, we modify the source code for the Open vSwitch library.

The reason is that flows installed on the switches must ensure correct timeouts with respect to their virtual clocks. Additionally, traffic statistics and network protocols must use their virtual clocks to maintain consistency. Specifically, we force Open vSwitch to use the `gettimeofday` system call, which in turn ensures that the Open vSwitch daemon, i.e., *ovs-vswitchd* and *ovs-db*, uses virtual clocks. We also include their process IDs when calling `freeze_all_procs` from Mininet. The motivation for requiring switches to be in virtual time is presented in Section 5.3 in more detail.

5 SYSTEM EVALUATION

5.1 Virtual Time System Overhead in Network Emulation

As described in Section 3, the synchronization between the power simulator and the network emulator requires us to freeze and unfreeze all emulated hosts. These operations bring overhead to synchronization. The overhead is not tolerable when the scale of the networking system grows to hundreds of emulated hosts on a single physical machine, which is quite common in practice Lantz et al. (2010). Note that the overhead to freeze/unfreeze processes does not affect the emulation temporal fidelity, which is evaluated in the next section.

We measured the overhead of our `pthread`-based implementation by repetitively freezing and unfreezing emulated hosts. We varied the number of hosts as 10, 50, 100, 250, and 500 in Mininet. For each setting, we repeated the freezing and unfreezing operations 1,000 times and computed the overhead as the duration from the moment the coordinator issues a freezing/unfreezing operation to the moment that **all** hosts are actually frozen/unfrozen. We added the overhead of the freezing operation and the overhead of the associated unfreezing operation and plotted the CDF of the emulation overhead in Figure 7.

We observe that more than 90% of the operations take less than 100 milliseconds in the 500-host case. For all other cases, more than 80% of the operations consume less than 50 milliseconds. We also observe that the average overhead time grows linearly as the number of hosts increases in Figure 8. The error bars indicate the standard deviations of the overhead time, which are caused by the uncertainty of delivering and handling the pending `SIGSTOP` and `SIGCONT` signals.

5.2 Accuracy Evaluation

End-to-end throughput and latency are two important network flow characteristics. In this section, we use these two metrics to evaluate the communication network fidelity. We created two emulated hosts connected via an Open vSwitch in Mininet. The links are set to 800Mbps bandwidth and $10\mu s$ latency. `iperf` Gates and Warshavsky (2014) was used to measure the throughput, and `ping` Hideaki (2015) was used to measure the round-trip time (RTT) between the two hosts.

5.2.1 End-to-End Flow Throughput. We used `iperf` to transfer data over a TCP connection for 30 seconds for throughput testing. In the first run, we advanced the experiments without freezing

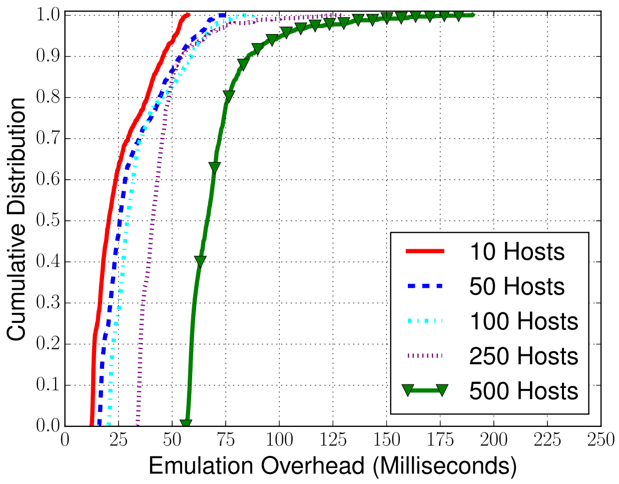


Fig. 7. CDFs of network emulation overhead caused by freezing/unfreezing operations.

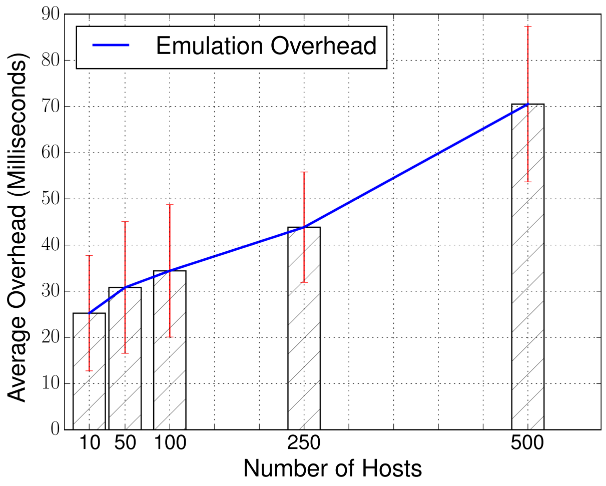


Fig. 8. Average network emulation overhead.

the hosts. In the second run, we froze the emulation for 1 second and repeated the operation every 1 second 64 times during the data transmission. We coupled the two experimental results and reported the average throughputs between the 11th second and the 30th second in Figure 9. The error bars represent the 99% confidence interval of the throughputs.

We observed that the average throughputs of the "interrupted" emulation match well with the baseline results. However, pausing emulation introduces around 11% to 18% deviation. Several sources lead to this deviation. First, while we explicitly generate `SIGSTOP` and `SIGCONT` signals to the containers, those signals are only in the pending state. The actual deliveries depend on the OS scheduler, and the deliveries usually occur when exiting from the interrupt handling. Second, the actual freezing duration depends on the accuracy of the sleep system call. Sleeping for 1 second has a derivation about 5.027 milliseconds on the testing machine, the Dell XPS 8700 with Intel Core i7-4790 3.60GHz processor.

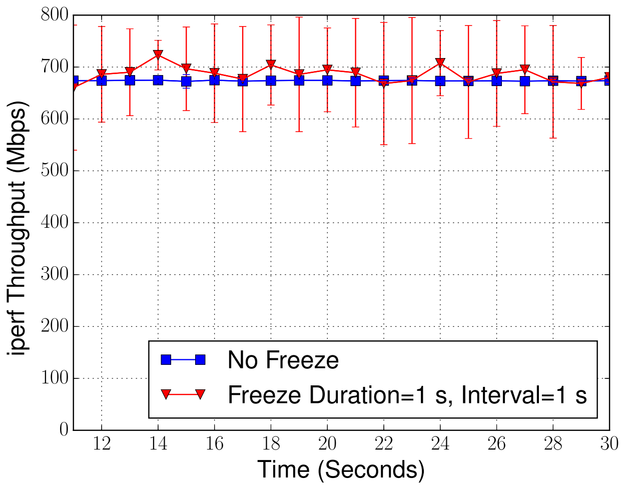


Fig. 9. TCP flow throughput comparison, 800Mbps bandwidth and 10μs link latency.

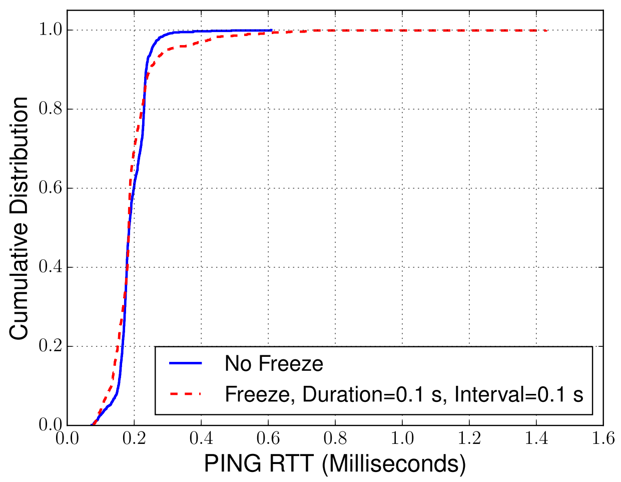


Fig. 10. Ping round-trip time comparison, 800Mbps bandwidth and 10μs link latency.

5.2.2 End-to-End Flow Latency. To evaluate the end-to-end flow latency, we issued 1,000 `pings` with and without freezing the emulator. We skipped the first ping in the results to exclude the effect of ARP and the switch rule installation from the SDN controller. Figure 10 plots the CDF of the RTT for both sets of the `ping` experiment. We observed that the two lines are well matched in the case of a 10μs link delay, and pausing the emulator does not affect the distribution of RTT. About 80% of ping packets are received at around 0.2ms.

When we increased the link latency to 1 millisecond, the observed RTTs in the freezing emulation case were around 1ms slower than the nonfreezing case. One solution is to reprogram the `hrtimer`, but if the target kernel only supports low-resolution timers, we need to search in the complicated time-wheel structure; otherwise, we can search in a red-black tree. Another approach is to explore the emulation look-ahead to increase the synchronization window size, and thus reduce the synchronization frequency between the two systems. We will leave those enhancements as our future work.

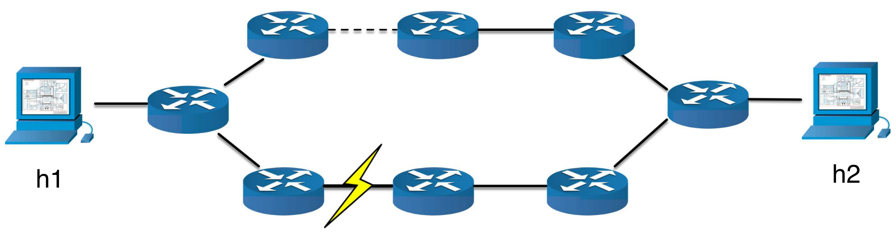


Fig. 11. Ring network topology. Blocking ports are denoted by the dashed line. When the primary path experiences a link failure, the blocking ports transition to forwarding ports per STP.

5.3 Open vSwitch Virtual Time Evaluation

We make Open vSwitch (OVS) support virtual time by modifying how OVS queries the system clock. In this section, we demonstrate how the virtual-time-enabled OVS improves the temporal fidelity of DSSnet by studying the convergence time of the spanning tree protocol with and without virtual time and its effect on the power simulator.

5.3.1 Spanning Tree Protocol. OVS can run the distributed spanning tree protocol (STP) 802.1D. In STP, the switches collectively compute a spanning tree in a network to prevent loops. The switches elect a root node by selecting the switch with the lowest priority, breaking ties by comparing MAC addresses. The switches then compute the lowest cost from their nodes to the root node. The lowest-cost paths are kept, while higher-cost paths are put into a blocking state to prevent forwarding traffic in a loop. Ports in STP can be in one of the following states: *blocking, listening, learning,* and *forwarding*. When the topology changes, i.e., through a link failure, a blocking port transitions to a forwarding state to maintain the complete spanning tree. The reconvergence process goes through three stages in the switches: link failure detection, transition to the listening state, and transition to the learning state. For example, if we set the Max_Age to 20 seconds, $Message\_age$ to 2 seconds, and $Forward\_Time$ to 15 seconds, then the total time for reconvergence is Lapukhov (2009)

$$Max\_Age - Message\_Age + 2 * Forward\_Time = 48 \text{ seconds.}$$

5.3.2 Convergence Time in Spanning Tree Protocol. Because the STP protocol runs on the switches, the reconvergence time will be affected by the virtual clock. To evaluate the effect, we set up the experiment by using two hosts: host 1 (h1) sends UDP packets to host 2 (h2). Upon receiving the packet, host 2 timestamps the packet and prints the header. The network topology is a loop topology depicted in Figure 11.

Figure 12 shows the effects of STP convergence with OVS in and out of virtual time. By causing a link down event simulating a link failure on the primary path, the ports that are in a blocking state must transition to forwarding states to re-establish the spanning tree. The OVS bridges are used with the default spanning tree settings. In the base case, we run OVS without virtual time and we do not create any synchronization events to freeze the emulation. In this case, it takes approximately 48 seconds to re-establish communication between the hosts, measured by the timestamps of captured UDP traffic. When we pause the emulation for a duration of 100ms at an interval of 100ms and do not use the virtual time for OVS, we observe a perceived convergence time of approximately 24 virtual time seconds. This is because the virtual time in the hosts is equal to half the wall clock time due to the pausing and resuming of the emulation. However, when we put the OVS processes into virtual time, we observe the correct behavior of approximately 48 seconds to re-establish the traffic path.

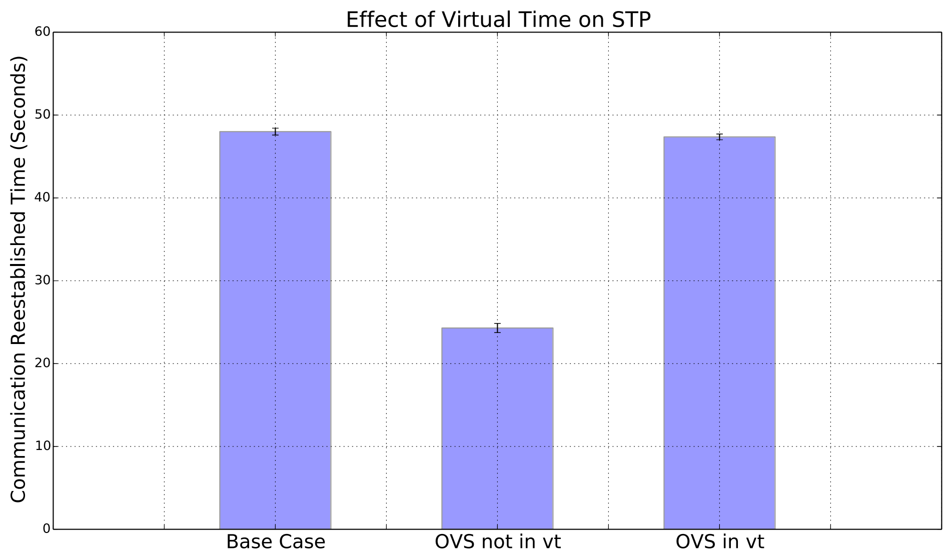


Fig. 12. Open vSwitch (OVS) in and out of virtual time. In the base case, we determine that it takes approximately 48 seconds to re-establish communication between the hosts with pausing of the processes. Without OVS in virtual time, we pause OVS and the host processes at an interval of 100ms for a duration of 100ms. Since OVS does not use the same clock as the hosts, it appears to re-establish the communication in 24 seconds DSSnet time (i.e., 48 seconds wall clock time). This causes an error with respect to the baseline case. By including OVS in virtual time, we observe the proper convergence time of 48 seconds.

5.3.3 Effect on the Power Simulator. We consider a simple demand response application consisting of a controllable energy storage device and a wind turbine. The wind turbine acts as a dynamic power generator, while the energy storage device charges and discharges power to stabilize the voltage of the system. Using the same ring topology in Figure 11, we send traffic from a sensor monitoring the dynamic generation to the energy storage device at an interval of 100ms that computes its discharge rate to maintain system stability.

In Figure 13, we can see how the power system is affected by the inaccuracies in the STP convergence time. The base case shows the correct operation of the system under normal operating conditions without a link failure. In the case where we have put OVS processes in virtual time, we see that after the link failure, the energy storage device receives new messages at about $t = 67$ seconds to re-establish the voltage of the system. This result corresponds to the previous experimental data in Figure 12. In the case that OVS is not in virtual time, it appears that the system stabilizes much quicker, and it is actually due to the error observed in Figure 12. The incorrect convergence time may hide system instability that causes the system to fail.

Therefore, experiments with OVS not in virtual time may lead to incorrect observations of the power system, potentially missing important events that could lead to system instability or failure. While we highlight the effect of STP not in virtual time, errors will be introduced for many other applications, such as flow metrics and rule timeouts.

6 CASE STUDY: DEMAND RESPONSE

DSSnet is designed for testing smart grid applications that affect both the power grid and the communication network. We now present a case study to analyze the behavior of a microgrid demand response application using DSSnet and to illustrate the benefits of an SDN-enabled microgrid.

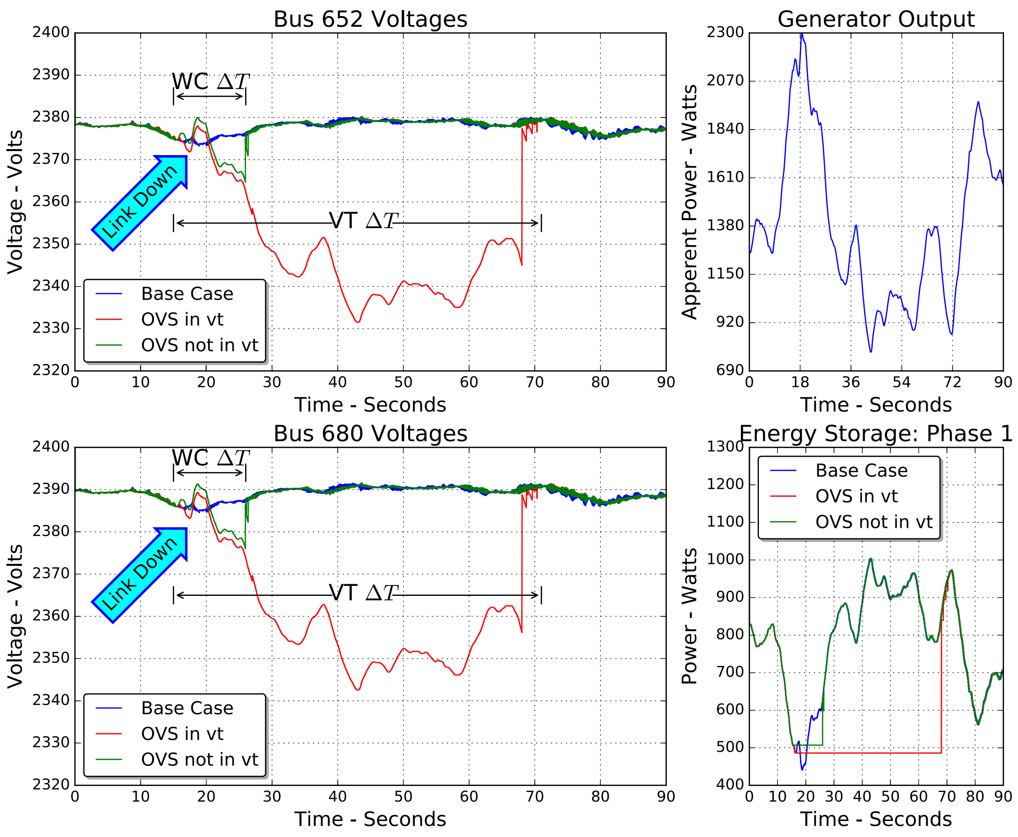


Fig. 13. Effect of STP on power system: the time uses DSSnet's clock. In the base case, we show the correct operation of the system without link failure. In the case of OVS in virtual time, we observe system instability until the communication network reconverges around $t = 67$ seconds. In the case without virtual time, we see that the network improperly reconverges prematurely. The wall clock time between the link failure and the reconvergence in the case of no virtual time for OVS (i.e., WC ΔT) is equal to the virtual time for the case with virtual time (i.e., VT ΔT), and WC ΔT =VT ΔT.

6.1 Experiment Setup

We model a microgrid using the IEEE 13-bus system (see Figure 14). The system's base voltage is 2.4kV and the point of common coupling with the main power system is at bus 650.

We consider a three-phase unbalanced microgrid with added distributed energy resource components and sensors. We add a renewable energy generator at bus 634 with variable power output matching the wind turbine in Figure 13. Additionally, we utilize three dynamic loads, an energy storage device, and a control center to control the energy storage device. The loads are added to buses 611, 652, and 692. Because the wind turbine provides dynamic generation and the loads also consume dynamic power, we consider a control application that measures the output generated by the wind turbine and power consumed by the loads and subsequently charge or discharge the energy storage device. The control center acts as the demand response server, which collects energy usage output from the sensors monitoring the dynamic loads and the generator. The control center processes the information to send control messages to the energy storage device to stabilize the system's voltage. Sensor information is sent every 100ms from the generator and sensors to the

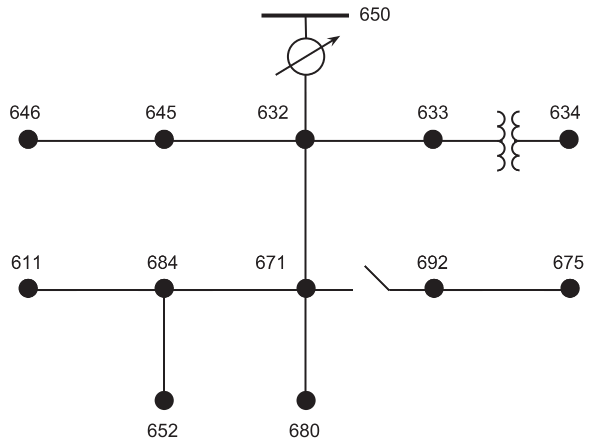


Fig. 14. Thirteen-bus distribution system.

control center. The control center computes the balance per phase and sends the corresponding control messages to the energy storage device.

The communication network is shown in Figure 15. There is one switch at each bus as well as two additional switches enabling backup routes and multipath forwarding. The control plane is a distributed ONOS cluster composed of three ONOS instances, connected to the microgrid communication network via a single switch. Each of the three ONOS instances manages five switches as shown by the color of the switch in Figure 15.

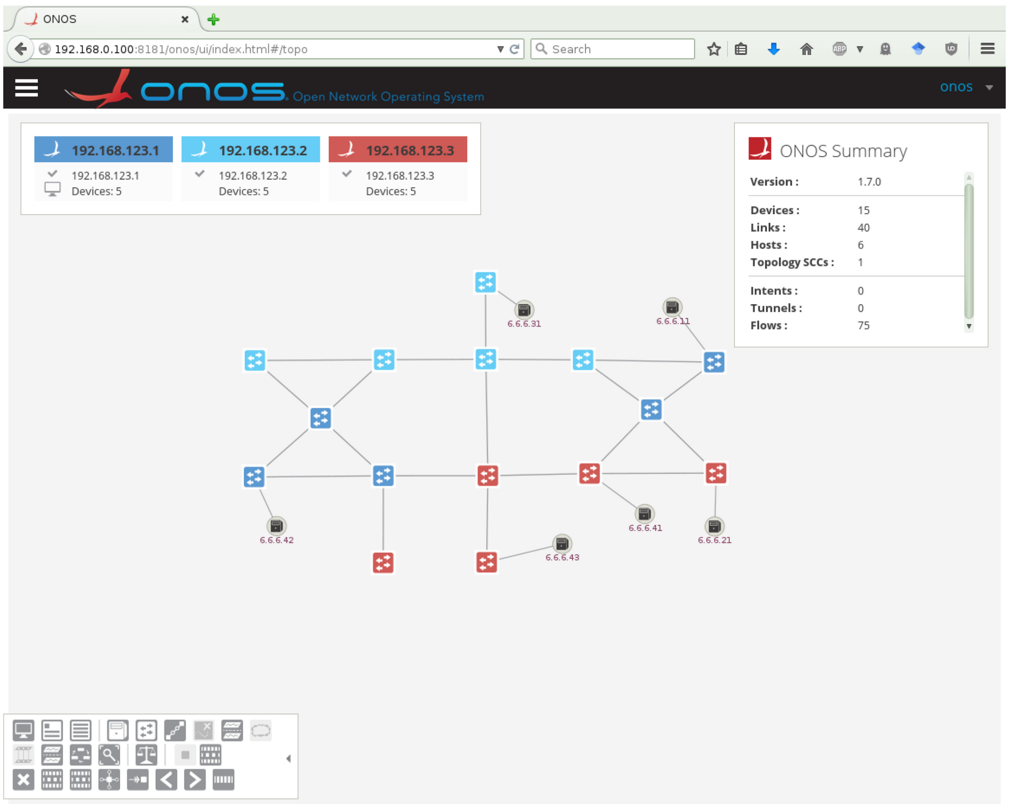
One major benefit of ONOS is the graphical interface, which shows the topology of switches and hosts, as well as established flows, flow intents, and traffic metrics. Additionally, ONOS comes with many helpful out-of-the-box network management applications, enabling statistic collection, multipath routing, and host-to-host intents. In our case study, we use the multipath routing and host-to-host intents to create the paths from the loads to the control center, from the generator to the control center, and from the control center to the energy storage device. Evaluating SDN-based networks is a unique ability of DSSnet. To illustrate how SDN can be utilized in the power network, we design an experiment to study the effects of a link failure between the controller and the energy storage device. By using the reactive forwarding and topology service applications built into ONOS, we see the effects of self-healing in the network.

The power simulator runs in duty mode with a timestep of 1ms. Synchronization events sent from the communication network are sensor readings for the loads and generator, as blocking events, as well as the energy storage charge and discharge commands, as nonblocking events.

6.2 Experimental Results

We plot the single-phase voltages at buses 652 and 675 in Figure 16. The demand response application shows that the voltage stability of the power system does not vary greatly from the nominal voltage in the simulated microgrid. Specifically, we can verify that the system does not deviate greater than 5% of the average value, meaning the system is stable.

The results found in this case study show that our demand response application successfully maintains the correct operation of the power system. The results provide motivation for evaluating and designing SDN-based solutions for cyber attacks on the demand response application as well as the communication network itself. In order to see the effects on the power grid during a link failure, we take down the primary link on the path from the control center to the energy storage



ONOS Summary

Version :	1.7.0
Devices :	15
Links :	40
Hosts :	6
Topology SCCs :	1
Intents :	0
Tunnels :	0
Flows :	75

Fig. 15. The ONOS web GUI depicts the communication network for the simulated microgrid. The color of the switches corresponds with the managing controller in the ONOS cluster. The bottom left shows the options for displaying background maps, flow statistics, intent creation, and more.

device at time $t = 3.2$s, simulating link failure. When the link goes down, SDN's centralized view of the network facilitates the quick response time on the order of milliseconds, causing minimal deviation from the base case. In Section 5, we have shown the effects of link down events on the decentralized spanning tree protocol. With a convergence time in the tens of seconds, the demand response application could suffer more severe outages, causing the power system to reach an unstable state. However, with the SDN applications provided by ONOS, we can see that the effect is significantly reduced.

DSSnet can be used for planning and evaluating more complicated demand response applications before being introduced to real systems. Additionally, since DSSnet provides real emulated network traffic, we can use DSSnet to study data injection attacks that target the power grid applications and test corresponding SDN-based solutions, such as SDN-enabled intrusion detection.

7 CONCLUSION AND FUTURE WORK

We present DSSnet, a testing platform that combines an electrical power system simulator and an SDN-based network emulator. DSSnet can be used to model and simulate power flows, communication networks, and smart grid control applications, and to evaluate the effect of network applications on the smart grid. Our future work includes exploring means to extract emulation lookahead to improve the performance of this hybrid system, as well as developing the

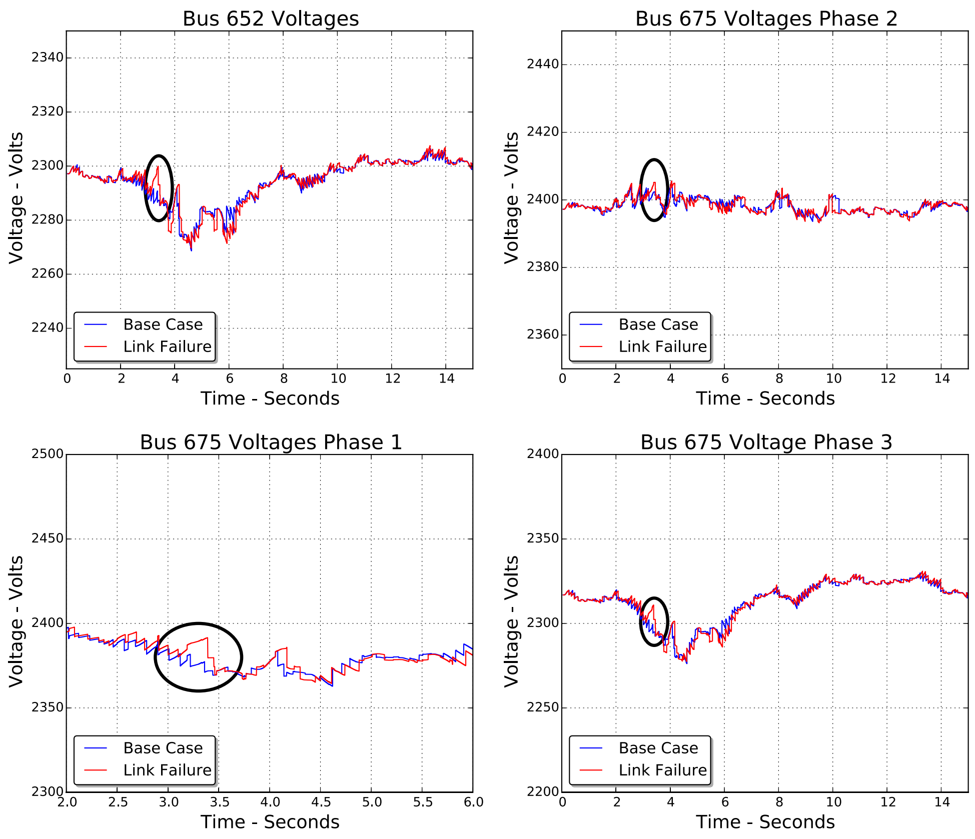


Fig. 16. Voltage at bus 652 and the three phases at bus 675. In the base case, we show the operation of our demand response application in an SDN-enabled network. In the link failure case, we illustrate the self-healing capabilities of ONOS. At time $t = 3.2$s, the link between the control center and the energy storage device fails. This causes a topology change that ONOS processes and assigns alternative flows from the control center to the energy storage device. The fast recovery from the link failure shows that SDN can be beneficial in the power grid. The circle illustrates the location and the effect of the link failure.

hardware-in-the-loop version of the testbed, i.e., to include real SDN switches and hosts. In the future, we will explore how SDN can provide security benefits as well as vulnerabilities, such as controller failure. We will also investigate several novel SDN applications for smart grid security and resilience, such as network-wide configuration verification and context-aware intrusion detection.

REFERENCES

Open networking foundation. Retrieved from https://www.opennetworking.org. Accessed August 2015.

Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. 2014. ONOS: Towards an open, distributed SDN OS. In *Proceedings of the 3rd Workshop on Hot Topics in Software Defined Networking (HotSDN'14)*. ACM, New York, 1–6. DOI: https://doi.org/10.1145/2620728.2620744

Adam Cahn, Juan Hoyos, Matthew Hulse, and Eric Keller. 2013. Software-defined energy communication networks: From substation automation to future smart grids. In *Proceedings of the 2013 IEEE International Conference on Smart Grid Communications (SmartGridComm'13)*. 558–563.

Selim Ciraci, Jeff Daily, Khushbu Agarwal, Jason Fuller, Laurentiu Marinovici, and Andrew Fisher. 2014a. Synchronization algorithms for co-simulation of power grid and communication networks. In *2014 IEEE 22nd International Symposium* *on Modelling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS'14).* 355–364. DOI: https://doi.org/10.1109/MASCOTS.2014.51

Selim Ciraci, Jeff Daily, Jason Fuller, Andrew Fisher, Laurentiu Marinovici, and Khushbu Agarwal. 2014b. FNCS: A framework for power system and communication networks co-simulation. In *Proceedings of the Symposium on Theory of Modeling & Simulation - DEVS Integrative (DEVS'14).* Society for Computer Simulation International, San Diego, CA, Article 36, 8 pages. Retrieved from http://dl.acm.org/citation.cfm?id=2665008.2665044.

Xinshu Dong, Hui Lin, Rui Tan, Ravishankar K. Iyer, and Zbigniew Kalbarczyk. 2015. Software-defined networking for smart grid resilience: Opportunities and challenges. In *Proceedings of the 1st ACM Workshop on Cyber-Physical System Security (CPSS'15).* ACM, New York, 61–68. DOI: https://doi.org/10.1145/2732198.2732203

Christian Dufour and Jean Belanger. 2014. On the use of real-time simulation technology in smart grid research and development. *IEEE Transactions on Industry Applications,* 50, 6 (Nov. 2014), 3963–3970. DOI: https://doi.org/10.1109/TIA.2014.2315507

Roger C. Dugan. 2013. Reference Guide, The Open Distribution System Simulator. http://download2.nust.na/pub4/sourceforge/e/el/electricdss/OpenDSS/OpenDSSManual.pdf. (Accessed January 2016).

Mark Gates and Alex Warshavsky. 2014. iperf3. Retrieved from http://software.es.net/iperf. (Accessed December 2014).

Uttam Ghosh, Xinshu Dong, Rui Tan, Zbigniew Kalbarczyk, David K. Y. Yau, and Ravishankar K. Iyer. 2016. A simulation study on smart grid resilience under software-defined networking controller failures. In *Proceedings of the 2nd ACM International Workshop on Cyber-Physical System Security.* ACM, 52–58.

Tim Godfrey, Sara Mullen, Roger C. Dugan, Craig Rodine, David W. Griffith, and Nada Golmie. 2010. Modeling smart grid applications with co-simulation. In *2010 1st IEEE International Conference on Smart Grid Communications (SmartGridComm'10).* 291–296. DOI: https://doi.org/10.1109/SMARTGRID.2010.5622057

Andrew Goodney, Saurabh Kumar, Akshay Ravi, and Young H. Cho. 2013. Efficient PMU networking with software defined networks. In *2013 IEEE International Conference on Smart Grid Communications (SmartGridComm'13).* 378–383. DOI: https://doi.org/10.1109/SmartGridComm.2013.6687987

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 Annual ACM Conference on SIGSIM Principles of Advanced Discrete Simulation (SIGSIM-PADS'16).* ACM, New York, 131–142.

Yoshifuki Hideaki. 2015. iputils. Retrieved from http://www.skbuff.net/iputils/. (Accessed November 2015).

Kenneth Hopkinson, Xiaoru Wang, Renan Giovanini, James Thorp, Kenneth Birman, and Denis Coury. 2006. EPOCHS: A platform for agent-based electric power and communication simulation built from commercial off-the-shelf components. *IEEE Transactions on Power Systems,* 21, 2 (May 2006), 548–558. DOI: https://doi.org/10.1109/TPWRS.2006.873129

Young-Jin Kim, Keqiang He, Marina Thottan, and Jayant G. Deshpande. 2014. Virtualized and self-configurable utility communications enabled by software-defined networks. In *2014 IEEE International Conference on Smart Grid Communications (SmartGridComm'14).* 416–421. DOI: https://doi.org/10.1109/SmartGridComm.2014.7007682

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, 179–186.

Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets-IX).* ACM, New York, Article 19, 6 pages.

Petr Lapukhov. 2009. Understanding STP convergence. Retrieved from http://blog.ine.com/2009/03/07/understanding-stp-convergence-part-i. (Accessed November 2016).

Hua Lin, Santhosh S. Veda, Sandeep S. Shukla, Lamine Mili, and James Thorp. 2012. GECO: Global event-driven co-simulation framework for interconnected power system and communication network. *IEEE Transactions on Smart Grid,* 3, 3 (Sept. 2012), 1444–1456. DOI: https://doi.org/10.1109/TSG.2012.2191805

Kevin Mets, Juan Aparicio Ojea, and Chris Develder. 2014. Combining power and communication network simulation for cost-effective smart grid analysis. *IEEE Communications Surveys Tutorials,* 16, 3 (2014), 1771–1796. DOI: https://doi.org/10.1109/SURV.2014.021414.00116

Elias Molina, Eduardo Jacob, Jon Matias, Naiara Moreira, and Armando Astarloa. 2015. Using software defined networking to manage and control IEC 61850-based systems. *Computers & Electrical Engineering* 43, C (April 2015), 142–154. DOI: https://doi.org/10.1016/j.compeleceng.2014.10.016

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

Davis Montenegro, Miguel Hernandez, and Gustavo A. Ramos. 2012. Real time OpenDSS framework for distribution systems simulation and analysis. In *2012 6th IEEE/PES Transmission and Distribution: Latin America Conference and Exposition (T D-LA'12).* 1–5. DOI: https://doi.org/10.1109/TDC-LA.2012.6319069

Hisham Muhammad. 2016. htop - an interactive process viewer for Unix. Retrieved from https://hisham.hm/htop/. (Accessed October 2016).

Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J. Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Jonathan Stringer, Pravin Shelar, Keith Amidon, and Martín Casado. 2015. The design and implementation of open vSwitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*. USENIX Association, Berkeley, CA, 117–130. http://dl.acm.org/citation.cfm?id=2789770.2789779.

Thomas Pfeiffenberger and Jia Lei Du. 2014. Evaluation of software-defined networking for power systems. In *Proceedings of the 2014 IEEE International Conference on Intelligent Energy and Power Systems (IEPS'14)*.

Lingyu Ren, Yanyuan Qin, Bing Wang, Peng Zhang, Peter B. Luh, and Ruofan Jin. 2017. Enabling resilient microgrid through programmable network. *IEEE Transactions on Smart Grid* 8, 6 (November 2017), 2826–2836. DOI: https://doi.org/10.1109/TSG.2016.2589903

Schweitzer Engineering Laboratories. 2014. Watchdog Project. Retrieved from https://www.controlsystemsroadmap.net/Efforts/Pages/Watchdog-Project.aspx.

Ali Sydney, David S. Ochs, Caterina Scoglio, Don Gruenbacher, and Ruth Miller. 2014. Using GENI for experimental evaluation of software defined networking in smart grids. *Computer Networks* 63 (2014), 5–16.

Jiaqi Yan. 2016. littlepretty/htop, forked from hishamhm/htop. Retrieved from https://github.com/littlepretty/htop. (Accessed October 2016).

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, 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, Article 27, 7 pages.

Yuhao Zheng, Dong Jin, and David M. Nicol. 2013. Impacts of application lookahead on distributed network emulation. In *Proceedings of the 2013 Winter Simulation Conference (WSC'13)*. 2996–3007.

Received November 2016; revised September 2017; accepted February 2018