Virtual Time Integration of Emulation and Parallel Simulation

Dong Jin*, Yuhao Zheng*, Huaiyu Zhu, David M. Nicol and Lenhard Winterrowd
University of Illinois at Urbana-Champaign
{dongjin2, zheng7, hzhu10, dmnicol, winterr2}@illinois.edu

***Abstract*—A high fidelity testbed for large-scale system analysis requires *emulation* to represent the execution of critical software, and *simulation* to model an extensive ensemble of background computation and communication. We leverage prior work showing that large numbers of virtual environments may be emulated on a single host, and that the timestamped interactions between them can be mapped to virtual time, and we leverage existing work on simulation of large-scale communication networks. The present paper brings these concepts together, marrying the scale emulation framework OpenVZ (modified earlier to operate in virtual time) with a scalable network simulator S3F. Our algorithmic contributions lay in the design and management of virtual time as it transitions from emulation, to simulation, and back. In particular, inescapable uncertainties in emulation behavior force us to explicitly set and reset timestamps so as to avoid either emulator or simulator having to deal with a packet arriving in its logical past. We provide analytic bounds and empirical evidence that the error introduced in resetting timestamps is small. Finally, we present a case-study using this capability, of a cyber-attack with the smart power grid communication infrastructure.**

***Index Terms*—parallel discrete event simulation; network emulation; virtual time**

I. Introduction

The advancement of large-scale computer and communication networks, such as Internet, power grid control networks, heavily depends on the successful transformation from in-house research efforts to real productions. To enhance this transformation, research has created various network testbeds that use emulation, or simulation, for conducting medium to large scale experiments. The emulation testbeds coordinate real physical devices and provide a configurable environment to conduct live experiments, but for networking are constrained by budget and what can be equipped in a lab. This limits scalability and flexibility. On the other hand, network simulation provides better scalability and much more flexibility, but degrades fidelity owing to the sort of model abstraction and simplification necessary to achieve scale. Furthermore, development of simulation models can be labor-intensive.

Our work in studying security in the smart grid's Advanced Metering Infrastructure (AMI) is one of the motivators of this paper. We need to study behavior of software and networking in a system with many meters, connected locally through wireless networks, and through wire-line networks to utilities. We need to study how particular software behaves under cyber-attack, and how the nature of a distributed denial of service (DDoS) attack affects delivery of that attack to victims, and how it impacts the overall network behavior. We use emulation technology to run real software stacks that run in meters, and simulation technology to model wireless and wirelined networks, as well as models of meters that contribute to the network traffic load but are not otherwise particular objects of study. We combine here two prior efforts. We use a version of OpenVZ modified to operate in virtual time [1] with a new parallel network simulator, S3F [2], which was inspired by SSF [3], and RINSE [4].

OpenVZ allows one to run real applications under a real OS and pass messages between simulated and emulated hosts. Users can plug in a real smart meter program rather than be forced to create a simulation model of one. OpenVZ operates in virtual time, not wallclock time, thereby increasing temporal fidelity [1] (unlike most other emulation systems). Freeing the emulation from the real-time clock permits one to run experiments either faster than real time, or slower, depending on the inherent simulation workload. We use S3F for simulating large network scenarios, as it provides sophisticated networking layer protocols and the ability to simulate many many devices such as routers, switches, and hosts creating and receiving background traffic. S3F therefore provides scalability. Coordination of activity between OpenVZ's emulation and S3F's simulation is handled by new extensions to S3F, described in this paper. The current system can run 200+ OpenVZ virtual machines and simulate millions of devices on a single multi-core server.

The contributions of this paper include design of synchronization, event passing and virtual machine control mechanisms in the hybrid system for safe and efficient experiment advancement. We do some small scale experiments to illustrate how changes to timestamps made by the system are bounded, and how these changes behave as a function of the overall simulation load (and are empirically seen to be much smaller than the guaranteed bound).

The remainder of the paper is organized as follows: Section II describes the system design architecture; Section III illustrates the implementation details of the synchronization mechanism and VE controller of the system; Section IV analyzes the system error; Section V shows a case study of our system for a DDoS attack security exercise in the AMI network; Section VI presents the related work and Section VII concludes the paper with future work.

* indicates equal contribution to this work.

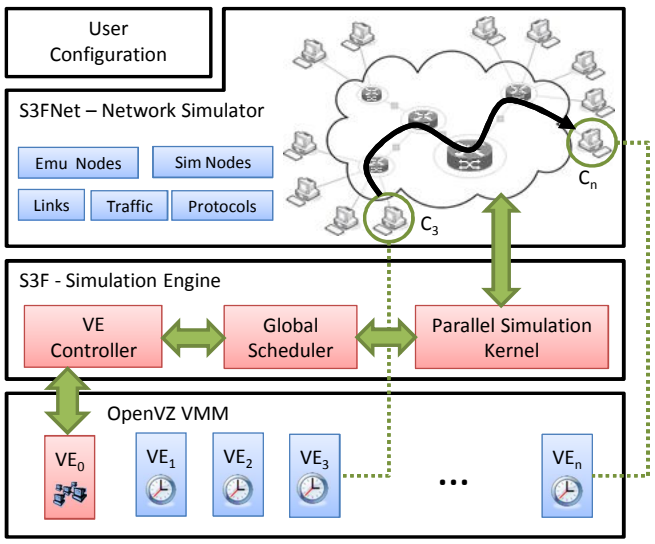

Fig. 1. System Design Architecture

II. Design

A. System Design Architecture

Figure 1 depicts the system design architecture of our system, which integrates the OpenVZ network emulation into a S3F-based network simulator on a single physical machine. The system is capable of running large-scale and high-fidelity network experiments with both emulated and simulated nodes.

1) Background: SSF and S3F: The Scalable Simulation Framework (SSF) is an API developed to support modular construction of simulation models, in such a way that potential parallelism can be easily identified and exploited. Following ten years of use, we created a second generation API named S3F [2]. In both SSF and S3F, a simulation is composed of interactions among a number of *entity* objects. Entities interact by passing *event*s through *channel* endpoints they own. Channel endpoints are described by *InChannel*s and *OutChannel*s depending on the message direction. Each entity is aligned to a *timeline*, which hosts an event list and is responsible for advancing all entities aligned to it. Interactions between co-aligned entities need no synchronization other than this event-list. Multiple timelines may run simultaneously to exploit parallelism, but they have to be carefully synchronized to guarantee global causality. The synchronization mechanism is built around explicitly expressed delays across channels whose end-points reside on entities that are not aligned. We call these *cross-timeline channels*. The synchronization algorithm creates synchronization windows, within which all timelines are safe to advance without being affected by other timelines. More details about S3F are in [2].

S3FNet is a network simulator built on top of S3F. In this work, we expand the capacity of S3F by integrating it with the OpenVZ-based network emulation. OpenVZ enables multiple isolated execution environments within in a single Linux kernel, called *Virtual Environments (VEs)*. A VE runs real applications which interact with emulated I/O devices (e.g. disks), generates and receives real network traffic, passing through real operating system protocol stacks. The only mechanism available to control a VE is the OpenVZ scheduler. When the scheduler frees a VE to execute, the VE runs without interruption or interaction with any other VE for the period of one "timeslice", a configurable parameter. This presents us with two challenges. One is that the actual length of time the VE runs is somewhat variable, the starting and stopping of that process being handled by the native operating system. In particular, a set of VEs run concurrently will not necessarily receive *exactly* the same amount of CPU service. This has ramifications for transforming observed real execution durations into virtual time durations. A second challenge is that all interactions between a VE and the network simulator must occur when the VE is not executing. This too has ramifications on assignment of virtual time to message traffic, and on how synchronization is performed.

Structurally, every VE in the OpenVZ model is represented in the S3FNet model as a host within the modeled network. Within S3FNet traffic that is generated by a VE emerges from its proxy host inside S3FNet, and, when directed to another VE, is delivered to the recipient's proxy host. The synchronization mechanism needs to know the distinction though between an emulated host (VE-host) or a virtual host (non-VE host), as shown in Figure 1. However, the type of host should make no difference to the simulated passing and receipt of network traffic. The global scheduler we added in S3F is designed for coordinating safe and efficient advancement of the two systems and to make the emulation integration nearly transparent to S3FNet.

S3F synchronizes its timelines at two levels. At a coarse level, timelines are left to run during an *epoch*, which terminates either after a specified length of simulation time, or when the global state meets some specified conditions. Between epochs S3F allows a modeler to do computations that affect the global simulation state, without concern for interference by timelines. Good examples of use include periodically recalculating of path loss delays in a wireless simulator, or periodic updating of forwarding tables within routers. States created by these computations are otherwise taken to be constant when the simulation is running. Within an epoch, timelines synchronize with each other using barrier synchronization, each of which establishes the length of the next *synchronization window* during which timelines may execute concurrently. Synchronization between emulation and simulation is managed by the global scheduler at the end of a synchronization window, when all timelines are blocked. Here it is that events and control information pass between OpenVZ and S3F, using S3F's global scheduler and *VE controller*. Details about these interactions will be discussed in Section III-A.

2) OpenVZ emulation and VE controller: OpenVZ is an OS level virtualization technology, which enables multiple isolated execution environments (called *Virtual Environments (VEs)*) within in a single Linux kernel. A VE has its own process

tree, file system, and network interfaces with IP addresses, but shares a single instance of the Linux operating system for services such as TCP/IP. Compared with other virtualization technologies such as Xen (para-virtualization) and QEMU (full-virtualization), OpenVZ provides excellent performance and scalability, at the cost of diversity in the underlaying operating system.

A given experiment will create a number of guest VEs, each has representation by an emulation host within S3FNet. Each VE has its own virtual clock [1], which is synchronized with the simulation clock in S3F. The VEs' executions are controlled by S3F simulation engine, such that the causal relationship of the whole network scenario can be preserved. As shown in Figure 1, S3F controls all emulation hosts through VE controller, which is responsible for controlling all emulation VEs according to S3F's command, as well as providing necessary communications between S3F and VEs. More details are provided in Section III.

The VE controller uses special APIs to control all guest VEs. It has the following three functionalities. (a) Advance emulation clock: while the VE controller communicates with OpenVZ to start and stop VE executions, it does so under the direction of the S3F global scheduler. Guest VEs are suspended until the VE controller releases them, and they can at most advance by the amount specified by S3F. When guest VEs are suspended, their virtual clocks are stopped and their VE status (e.g. memory, file system) remains unchanged. (b) Transfer packets bidirectionally: the VE controller passes packets between S3FNet and VEs. Packets sent by VEs are passed into S3FNet as simulation inputs and events, while packets are delivered to VEs whenever S3FNet determines they should. By doing so, we provide the notion to the emulation hosts that they are connected to a real network. (c) Provide emulation lookahead: S3F is a parallel discrete event simulator using conservative synchronization [5], and its performance can be significantly improved by making use of lookahead. While S3F may have sufficiently knowledge of the network model state when calculating lookahead, it has no knowledge of the future behavior of an emulation. The VE controller is responsible for providing such emulation lookahead to S3F, the details of which are of course application dependent.

B. Simulation/Emulation Coordination

Our design forces the OpenVZ emulation to always runs ahead of the S3F simulation model, so that VEs operate as traffic sources. Before S3F permits the simulator to advance over a time interval $[a, b)$, we first ensure that all VEs have advanced their own virtual time 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. A packet generated within a VE is given a virtual time stamp based on the VE's clock at the beginning of its timeslice, and the measured execution time until the application code calls the OS to send the packet. The initial send time is as accurate as we can make it. Potentially more parallelism could be exploited if the emulation and simulation executed concurrently. This is a topic we will explore later, as there is sufficient parallelism for the size of problems we're interested in now, and tighter synchrony could paradoxically reduce performance owing to more complex scheduling.

A packet bound for a VE proxy host transits the network model, reaches the proxy host, and is passed to the VE controller, stamped with the arrival time, t. The VE controller delivers the packet to the target VE at the initialization of the first timeslice when the target VE clock is at least as large as t—for a very practical reason. All VEs share the same operating system and its state, and all packets are ultimately obtained by the VE through calls to the operating system; only by extensive modifications to the OS kernel could we build in a per-VE buffering capability that would accept a future packet arrival, and not present it to a VE before the packet's arrival time. We've adopted an approach that is much easier to implement, at the cost of it always being the case that the virtual time at which a packet is recognized (e.g. by a socket read) can be larger than the packet's arrival time.

While the synchronization window $[a, b)$ was constructed to ensure that no traffic created within $[a, b)$ is also delivered across timelines within $[a, b)$, it is possible for the VEs to have advanced so far that S3FNet presents a packet to a VE's proxy with a timestamp that is smaller than the VE's clock. This risk seems unavoidable, owing to the coarse grained control we have over VE execution, and when this occurs we deal with it by changing the packet's timestamp.

To understand and bound the extent to which timestamps may be modified, we need to carefully step through the assignment of timestamps, described in the next section.

C. Virtual Time Advance

Our modification of the OpenVZ system converts execution time into virtual time; a VE that has advanced in simulation time to t_0 is given T units of execution time, and run. At the end of the execution its clock is advanced to time $t_0 + \alpha * T$, where α is a scaling factor used to model faster ($\alpha < 1$) or slower ($\alpha > 1$) processing. It is important to realize that this is an approximation that treats only at a coarse level factors that affect execution time, e.g., caching and pipelining effects. In addition, the scheduling mechanism is not so precise that *exactly* T units of execution time are received, and the VE's actual execution time T' may slightly deviate from T. Nevertheless, in order to keep all VEs in sync with respect to the clock, after execution the VE's virtual clock is explicitly set to $t_0 + \alpha * T$.

In the OpenVZ system, the unit of scheduling (minimum execution time) is a timeslice. We currently set timeslice length $TS = 100\mu s$, but TS is tunable [6]. For the sake of efficiency, the VE does not interact with the VE controller until after its full timeslice has elapsed, at which point packets sent by the VE may be collected, and packets may be delivered to the VE. As we arrange that the emulation always runs ahead of the network simulation, we are assured that each packet arrival

lies in the temporal future of the VE-host, and so the packet retains the timestamp received in the emulation.

During the execution, if a message send is performed by the VE, the timestamp on the message is the computed virtual time at which the message leaves the VE to enter the network. In particular, if that departure occurs x units of measured execution time after the beginning of the timeslice, the virtual time of the VE is computed as $t_s = t_0 + \alpha * \min\{x, T\}$, where t_0 is the virtual time at the beginning of the timeslice. The min term is introduced as it is possible for the VE to run longer than T units even though its clock will be advanced only by $\alpha * T$ units, and we need to have virtual time be consistent with that fact. We can bound the amount by which any virtual timestamp is artificially smaller up to $\alpha * T_\epsilon$, where T_ϵ denotes the maximum deviation between T' and T. For the magnitude of TS we have used typically (100 μs), T_ϵ has tended to be relatively small. It can be up to TS in the worst case, but has proven to be much smaller than that in practice.

At some point the timeline in S3F on which the VE-host is aligned advances its time to recognize the arrival, and normal simulation time advancement techniques deliver the packet to its destination VE-host, say at time t_d. Mechanisms yet to be described ensure that the simulation does not advance farther in time than the VEs have advanced, and so t_d necessarily arrives to a VE with a timestamp smaller than VE's clock. Conceptually anyway, it arrives to the VE later, precisely at the time when the VE begins its next timeslice of execution. In some circumstances this can cause a functional deviation in VE behavior. For example, if the VE in any way "looked" for a packet arrival during its previous timeslice at times t_d or greater, it would not see it, and would react to the absence as coded. However, if the VE behavior in the previous timeslice is insensitive to the presence or absence of a packet, the late arrival poses no logical difficulties. When the VE looks for a packet it will find one. From this we see that the effective arrival time of the packet cannot be later than one timeslice TS than its timestamped arrival time.

These observations are summarized more formally below.

Lemma 1: Let t_0 be a VE's clock at the beginning of an execution, and suppose a packet is sent x units of execution time later. The timestamp on the message presented to the network simulator is less than $t_0 + \alpha * x$ by no greater than $\alpha * TS$, where α is the virtual/real time scaling factor and TS is the timeslice length.

Lemma 2: Suppose a packet is delivered to a VE-host at virtual time t_d. That packet is available to the VE no later than time $t_d + \alpha * TS$.

It is worth pointing out that we cannot construct an end-to-end bound on the error of the packet's timestamp without making some assumptions about how network latencies are different between an arrival at time $t_0 + \alpha * x$ versus an earlier arrival at time $t_0 + \alpha * TS$.

III. Implementation

We added two components to the S3F simulation engine to support integration with OpenVZ: the global scheduler, which coordinates the time advancement of both emulation VEs and simulation entities; and the VE controller, which is responsible for VE scheduling and message passing, such as packets, emulation lookahead, between VEs and simulation entities. This section will illustrate how the system works by explaining the implementation details of the two components and the decisions we made behind them.

A. Simulation/Emulation Synchronization

S3F supports parallel execution, which requires synchronization among multiple-timelines. Latencies across communication paths established between outchannels and inchannels are used to establish a simulation synchronization window, within which no events from an outchannel can be delivered to any cross-timeline mapped inchannels. The windows are implemented with barrier synchronizations. The upshot is that cross-timeline events do not have to be immediately delivered since their receipt lies on the other side of a barrier synchronization. The events can be buffered until the end of the synchronization window, when the synchronized timelines exchange such events, and integrate them into their target timelines' event lists [2]. The larger the synchronization window size is, the less frequent a simulator needs to stop for global synchronization, and so achieve better performance. In terms of pure network simulation, the channel mapping can be used to model links among host network interfaces, and the latencies across the channels can be constructed by the packet transfer time and the link propagation delay.

However, integration with the OpenVZ-based emulation brings new features and constraints to the existing synchronization mechanism. Firstly, emulation and simulation never operate concurrently, therefore two clocks actually exist in the system: the current simulation time and the current emulation time; there exist also two types of synchronization window: the *emulation synchronization window (ESW)* and the *simulation synchronization window (SSW)*. The system first computes an ESW and runs the emulation for that long, and then injects packets created during that window into the simulator, at the end of the ESW. The new emulation events contribute to the computation of SSW for the next simulation cycle. Both ESW and SSW are calculated at S3F. Secondly, our system design ensures that the simulation can never run ahead of the current emulation time. Thirdly, once the OpenVZ emulation starts to run, it has to run for at least one timeslice [1], during which no simulation work can interrupt any VE. This system level constraint affects the granularity of the system. Finally, the OpenVZ system introduces opportunities for offering real application specific lookahead for increasing the size of ESW.

The notations used in this section are listed below:

t_{emu}	current emulation time: OpenVZ virtual time
t_{sim}	current simulation time
E_{emu}	the set of VE-proxy entities in S3F
E_{sim}	the set of non-VE-proxy entities in S3F
ESW	emulation synchronization window: the length of the next emulation advancement

SSW simulation synchronization window: the length of the next simulation advancement

α a scaling factor used to model faster ($\alpha < 1$) or slower ($\alpha > 1$) processing time in OpenVZ system, details in Section II-C

TS timeslice length in OpenVZ system, unit of VE execution time

EL_i the event list of timeline i

EL_i^{emu} the set of events in EL_i that may affect the state of a VE, e.g. a packet delivery to a VE

EL_i^{sim} the set of events in EL_i that will not affect the state of a VE, $EL_i^{sim} \cup EL_i^{emu} = EL_i$

n_i timestamp of next event in EL_i; $n_i = +\infty$ if $EL_i = \varnothing$

n_i^{emu} timestamp of next event in EL_i^{emu}; $n_i^{emu} = +\infty$ if $EL_i^{emu} = \varnothing$

n_i^{sim} timestamp of next event in EL_i^{sim}; $n_i^{sim} = +\infty$ if $EL_i^{sim} = \varnothing$

$w_{i,j}$ minimum per-write delay declared by outchannel j of timeline i

$r_{i,j,k}$ transfer time between outchannel j of timeline i and its mapped inchannel k

$s_{i,j,x}$ transfer time between outchannel j of timeline i and its mapped inchannel x, where x aligns with a timeline other than i

$l_{i,e}$ emulation lookahead of entity e in timeline i, computed by VE Controller in every ESW; $l_{i,e} = +\infty$ if $e \in E_{sim}$

The scheduling mechanism used in the global scheduler is described in Algorithm 1. It makes the emulation run first, and ensures the simulation time never exceeds the emulation time. When the simulation catches up with emulation, emulation is advanced again.

Algorithm 1 Global Scheduler

```
while true do
  if t_sim = t_emu then
    compute ESW
    run OpenVZ emulation for ESW (Algorithm 2)
    inject packets to simulation
  else
    compute SSW
    run S3F simulation (all timelines) for SSW
  end if
end while
```

Equation 1 below illustrates how ESW is calculated:

$$ESW = \max\left\{\alpha * TS, \min_{timeline\ i}\{P_i\} - t_{emu}\right\} \quad (1)$$

where P_i is the lower bound of the time when an event from timeline i can potentially affect a VE-proxy entity in the simulation system, for the global scheduler to decide the next ESW:

$$P_i = \min\left\{\left[\min\left(n_i^{sim}, \min_{entity\ e}\{l_{i,e}\}\right) + B_i\right], n_i^{emu}\right\}$$

and B_i is the minimum channel delay from timeline i:

$$B_i = \min_{outchannel\ j}\left\{w_{i,j} + \min_{inchannel\ k}\{r_{i,j,k}\}\right\}$$

In our system, a packet is passed to VE Controller for delivery right after the packet is received by a VE-proxy entity in S3F. As simulation is running behind, the packet is not available to VE Controller until simulation catches up and finishes processing that event. The P_i calculation prevents an VE from running too far ahead and bypassing a potential packet delivery event.

Equation 2 below illustrates how SSW is calculated:

$$SSW = \min\left\{t_{emu}, \min_{timeline\ i}\{Q_i\}\right\} - t_{sim} \quad (2)$$

where Q_i is the lower bound of the time that an event of timeline i can potentially affect an entity on other timeline, for the global scheduler to decide the next SSW:

$$Q_i = n_i + C_i$$

and C_i is the minimum cross-timeline channel delay from timeline i:

$$C_i = \min_{outchannel\ j}\left\{w_{i,j} + \min_{inchannel\ x}\{s_{i,j,x}\}\right\}$$

As the simulation runs behind the emulation in virtual time, i.e. t_{sim} can be at most advanced to t_{emu}, events potentially generated from emulation can be ignored when calculating Q_i, as they all have timestamps no smaller than t_{emu}.

When SSW is smaller than $\alpha * TS$, the simulation has to run multiple synchronization windows to catch up to the emulation. On the other hand, when SSW is larger than $\alpha * TS$, the emulation can run multiple time slices in one emulation cycle. Figure 2 and Figure 3 illustrate the behavior of the system in the two cases respectively.

In both case, the simulation advancement is bounded by ESW. However, in case 2, the simulation helps to improve

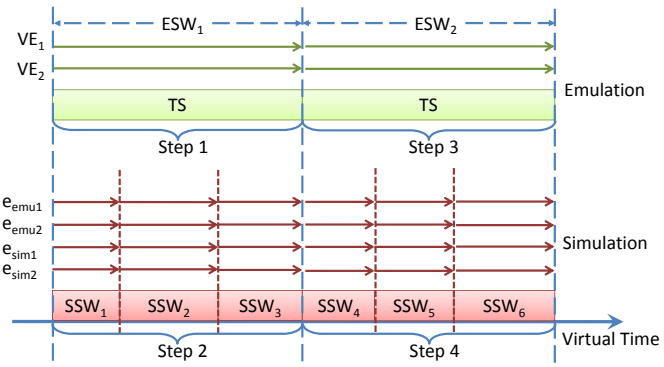

Fig. 2. System Advancement with Global Synchronization, Emulation Timeslice $\geq$ Simulation Synchronization Window

Fig. 3. System Advancement with Global Synchronization, Emulation Timeslice < Simulation Synchronization Window

the emulation performance by computing a large ESW, so that emulation can run through over multiple timeslices before interacting with the VE controller, thereby enjoying less synchronization overhead. In return, the emulation also provides event information to the simulator, which could improve the simulation performance with a larger SSW. A large SSW can be obtained by utilizing detailed network-level and application-level information, such as minimum link delay and minimum packet transfer time along the communication paths, or network idle time contributed by the simulated devices that not actively initiate events (e.g. server, router, switch), or from the lookahead offered by the OpenVZ emulation, refer to Section III-B.

B. VE Controller

The VE controller's main responsibility is to advance the emulation clock. The VE controller does not drive VEs directly, but allocates timeslices in which to run. A VE is suspended and its virtual clock is paused, except during an allocated timeslice. Once released, a VE runs until the timeslice expires, with its virtual clock increasing as a scaled function of elapsed execution time [1].

Each time the VE controller is invoked by the global scheduler, it is given a window size (ESW) within which all VEs are to advance. Within an ESW, all VEs are independent, i.e. no events from a VE can affect another VE. This independence is either guaranteed by S3F according to channel delays, or derives from the minimum VE scheduling granularity. The VE controller delivers packets to VEs just before they begin to execute, and collects generated packets from them after they execute. The logic of VE controller is described in Algorithm 2.

When the VE controller gets control back after a non-idle VE has run a timeslice, there is variability in the actual length of timeslice the VE consumed, primarily due to the timing resolution of the Linux scheduler. Instead, for a given ESW, whatever length of execution ends up being allocated, the VE controller *assumes* it is precisely ESW and adjusts the clock accordingly. Algorithm 2 is slightly more complex than this description, containing some correction terms and handling idle VEs slightly differently.

Algorithm 2 VE Controller

$barrier = t_{emu} + ESW$
for all VE_i **do**
  $VE_i.stop = barrier - \alpha * TS/2 - VE_i.offset$
  $VE_i.done =$ false
  while $VE_i.done =$ false **do**
    deliver due packets to VE_i
    give a timeslice to VE_i
    ($VE_i.clock$ keeps advancing while VE_i is running)
    wait until VE_i stops
    collect sent packets from VE_i
    if VE_i is idle (has no runnable processes) **then**
      $VE_i.offset = 0$
      $VE_i.clock = \min(VE_i.nextPacket, VE_i.stop)$
    end if
    if $VE_i.clock \geq VE_i.stop$ **then**
      $VE_i.offset$ += $VE_i.clock - barrier$
      $VE_i.clock = barrier$
      $VE_i.done =$ true
    end if
  end while
  calculate emulation lookahead for VE_i
end for
$t_{emu} = barrier$

At the end of a VE controller cycle, the emulation lookahead is calculated and conveyed to S3F through an API. The emulation lookahead is a duration of future virtual time within which a VE will not send packets, so that it will not affect the states of other hosts. In the test cases studied here we use constant bit rate (CBR) traffic source which makes emulation lookahead computation straightforward. In this paper, we demonstrate the promise of emulation lookahead; estimating it and tolerating errors in it is an area of future research.

IV. Error Analysis

We have seen already that timestamps may be changed, and have bounded the magnitude of those changes. We now examine these changes empirically, using a simple network which contains two emulation hosts. These two hosts are connected via a link with 1 Gb/s bandwidth and 100 μs delay. The timeslice TS is also 100 μs, and α is set to 1. During the experiment, a sender application sends constant bit rate (CBR) traffic—meaning the packet inter-arrival time is as constant as virtual time advance can make it—to a receiver application in the other VE. The receiver loops over a blocking socket read, yet has a background computation thread to keep the VE non-idle. For each packet we trace its arrival time at different points along its path, which will reveal where and by how much the virtual time changes. We record the following times:

- $talker$: the packet is generated by the sender app
- $vcpull$: the sending timestamp presented to S3FNet
- $s3fnet$: the delivery timestamp computed by S3FNet
- $vcpush$: the packet is delivered and available to the VE

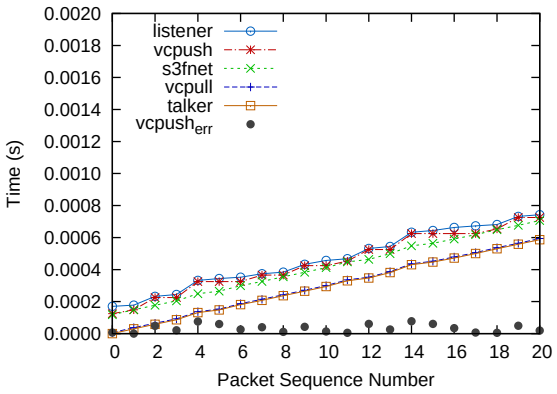

Fig. 4. Timestamps during Packets Traverse Route

- $listener$: the packet is received by the receiver app
- $vcpush_{err}$: system error, equals to $vcpush - s3fnet$

For the sender application, we have tested 25 Mb/s, 100 Mb/s, and 400 Mb/s sending rate. The results are shown in Figure 4 for the 400 Mb/s case. The x-axis indexes the packet, the y-axis shows the times associated with each packet. Slopes decrease with increasing sending rate because inter-packet arrival times decrease.

Although we plot only one sending rate, the behavior of the error in each case is very close, and in this case is bounded by 100 μs—the length of a timeslice. Likewise, the effect of communication latency is the same in each plot, and can be seen in Figure 4. As explained in Section II-C, the sending timestamp we put on a packet is exactly the virtual time when it leaves its VE. There we see clear that $talker$ and $vcpull$ are nearly indistinguishable—the only difference is constant processing delay from application layer to IP layer. The gap between $vcpull$ and $s3fnet$ is the (constant) network latency. Any gap between $s3fnet$ and $vcpush$ is due to the effect described before, that a packet is not pushed to a VE until the VE's clock is at least as large as the packet's arrival time. We know this gap is no larger than $\alpha * TS$. The data here confirms the theory, and shows that in this experiment the gap is on average considerably small than one timeslice. The data occasionally shows a gap between $vcpush$ and $listener$, but this is not caused by our system. Instead, it is caused by multi-task scheduling delay inside the VE, in the same way it exists on a real machine.

The $\alpha * TS$ error bound is an absolute value. When the sender is sending at a very fast rate, e.g. 400 Mb/s as shown, and the inter-packet duration is small, such error and delay approach the inter-packet delay. When the sender is sending at a slower rate, e.g. 100 Mb/s or 25 Mb/s, such error and delay become negligible compared with the relatively large inter-packet delay. We conclude that our system can provide sufficient accuracy for those scenarios that can tolerate these errors. For scenarios that require higher accuracy, one can reduce the length of timeslice, but at the cost of slower execution speed [6].

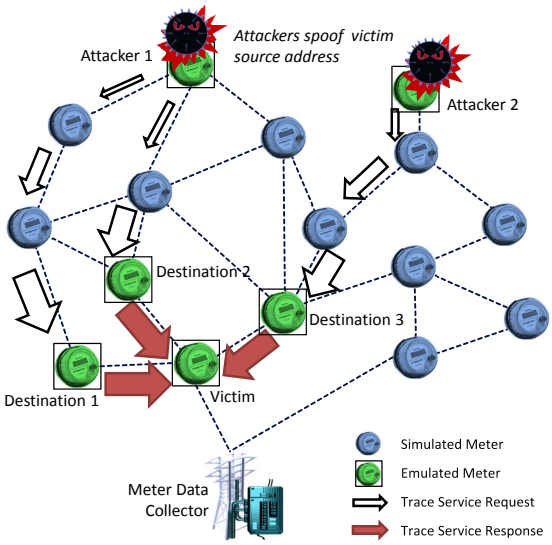

Fig. 5. C12.12 Trace Service DDoS Attack in AMI Network

V. Case Study: DDoS Attack in AMI Network

A. Overview of the DDoS Attack Using C12.22 Trace Service

Advanced metering infrastructure (AMI) systems use metering devices to gather and analyze energy usage information. The emergence of AMI is an important step towards building a smart grid that provides both cost efficiency and security. Various communication models, protocols and devices can be combined to form the communication backbone of an AMI network. In North America, the major deployment of AMI network is based on Radio Frequency Mesh network architecture, wireless metering devices and the ANSI C12 protocol suite.

In this section, we present a case study that highlights a potential Distributed Denial of Service (DDoS) attack we discovered in an AMI system that uses the C12.22 transport protocol, and we find our testbed well supports this experiment scenario. In this scenario we require detailed functional behavior of some meters—the ones directly involved in the attack, but only routing behavior from the others. This suggests an approach where a few meters are emulated, with the rest of the meters and the communication network being simulated. The whole experiment can be done on a single multi-core machine, and this makes the experiment economic and easy to set up.

ANSI C12.22 protocol is widely used for AMI systems, defining the application used to exchange information between AMI devices. It provides a *trace service* to return the route between source and destination that a particular C12.22 message traverses. The main purpose of the trace service is for network administration and failure detection. However, the design does not include any security features, and it can be exploited by malicious users to launch DDoS attacks.

We next explain how DDoS attacks can be launched. When a node wants to trace the route to a target node, it sends out a message with its own ID and the target node's ID

enclosed. Whenever an intermediate node on the route receives the message, it appends its ID to the message and forwards it to the next hop. Once the destination node receives the request, it replies with a sequence of all intermediate nodes' IDs, and thus the route the initiator seeks to know. Once the trace request reaches the target, the message is returned to the source—and herein lies an important element of the attack. A malicious source puts a *victim's* ID in as the message source. Thus, a number of compromised meters, working in concert, can generate many trace requests, each carrying the spoofed source identity of a victim. The long messages "reflect" and converge on the victim. Figure 5 illustrates this attack.

B. Attack Experiment Analysis

The AMI network we created for the case study models a typical 4×4 block neighborhood in a town. There are a total of 448 meters, distributed evenly (approximately) along the street edges, as shown in Figure 6. The meters responsible for parsing and processing C12.22 packets are emulated by applications in VEs using real OS protocol stack. This set includes five attacking meters that generate trace service requests, five meters to which the traces are directed (each attacker targets its own), and one victim device, whose source address is spoofed by the attackers. The rest meters and the underlying communication network (802.15.4 ZigBee wireless network) with 1 Mb/s bandwidth are modeled and simulated by S3F. The radio channel path-loss model is the simple $1/d^2$ line-of-sight model. More sophisticated models can be introduced as needed.

Figure 6-A1 and A2 illustrate key meters in the experiment. The egress point is seen on the lower right edge. All the meters send routine traffic to that device, around 100-byte packet per 10 seconds. Attacking this choke-point maximizes the impact of the DDoS attack, and thus we set all the five attacks choose this point as the victim, and choose one of its close neighbors as the destination of the trace service request (and hence, reflection point). The figures mark out the location of the attackers, and the locations of their trace request destinations. Each attacker sends a trace service packet every 0.05 seconds (200 times faster than a normal meter) and each intermediate meter will add additional 20 bytes into the payload. Experimenting on the testbed shows that attackers initializing a few large-size packets rather than many small-size packets can improve attacking efficiency due to eliminating frequent back-off times, therefore the trace service packet size is set to 500 bytes. Also learning from the experimental results, arranging the attacker meters in such a way that each of them covers a long (around 15 to 30 hops in this scenario) and spacial-separated route to the victim's surrounding meters could effectively render the entire network useless. More details will be covered soon in the result analysis.

We investigate and evaluate the impact of the DDoS attack by the following three metrics from each meter's viewpoint. The experiments data are collected in a 100 second window and the results are shown in Figure 6 for the normal scenario and the attacking scenario respectively.

- r_u channel utilization, fraction of time that a meter is transmitting packets
- r_c channel contention, fraction of time that a meter senses busy channel
- r_l packet loss, fraction of lost packets

Figure 6-A1 illustrates the fraction of time a meter is in a transmitting state during a 100-second period, where the size of a point reflects its transmitting rate. Compared with A2 (the same experiment, but with attackers) we see that meters which route attack traffic have much higher transmitting rates than others. By tracing the highlighted meters, we can easily observe the routing paths between attackers and the victim. The results also clearly illustrate the most interesting behavior of trace service – the packet along the forwarding path takes longer transmitting time and consumes more power of the relay meter, as one expects because of longer packet length. Another interesting observation is that when two or more attackers share a common path (e.g., attacker1 and attacker2 in Figure 6-B1), they tend to block out each other. Therefore, an efficient strategy requires the attackers to smartly select routes covering the entire network, especially the area around the victim, with minimum overlaps.

The AMI network uses ZigBee wireless as communication model which means the attacking traffic does not only affect the meters who forwards the traffic but also jams the channels of meters around them. Figure 6-B2 presents the wireless channel contention in AMI network. The size of the point codes the utilization of the wireless channel sensed by each meter. From the figure we can see that, wireless channels are free before the DDoS attack. Few channel competitions can be found around the place where the gateway is located. However, when we turn five meters (roughly 1% of all meters) into attacking nodes, the injected DDoS traffic cause considerable channel contention in the traversed areas. In Figure 6-B2, one of the most busy zones is at victim's location. Due to the collision avoidance protocol used by ZigBee, the meters will back-off until the channel is free. In this case, it is very difficult for legitimate traffic to pass through the channel busy area. In addition, when the attacking traffic from different meters meet each other in the network, they will compete against each other for wireless channel and their battle field becomes a noticeable busy area in Figure 6-B2.

In the third set of experiments, we compared results in Figure 6-C1 and C2, and show the ultimate negative impact that the trace service DDoS attack has imposed on the entire AMI network by measuring loss rate of legitimate traffic at each meter. Packets are dropped after four unsuccessful transmission attempts, or when a buffer (assumed here to hold 100 packets) overflows. It is not surprising that most of the legitimate meters in AMI network experience increasingly high packet drop ratio under DDoS attack since the only egress point has been efficiently blocked by attacking traffic. This is achieved by compromising fewer than 1% of the meters with properly selected attacking routes.

The case study shows how our system can be used for exploring security in a critically important infrastructure. It

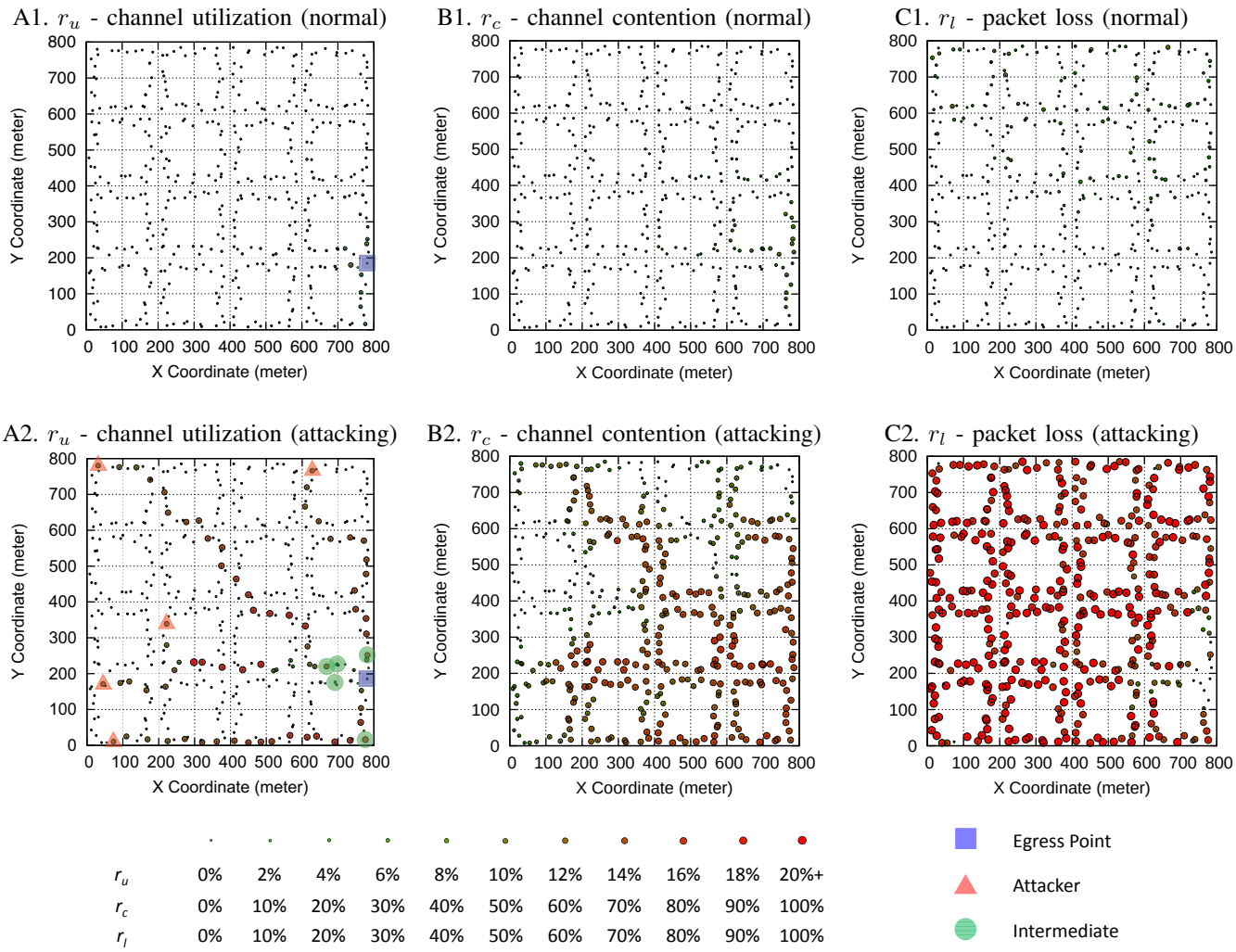

Fig. 6. Experimental Results of the C12.22 Trace Service DDoS Attack

provides the capability and flexibility to set up a testing scenario with real emulated hosts modeling complicated applications and large-scale simulated network environment. The detailed study of the trace attack and other attacks in smart grid and their corresponding defense mechanisms in our testbed remains future work.

C. Scalability

A simulator's performance scales if the ratio of simulation time to wallclock time increases only linearly as the model size increases. For our model we simultaneously increased the number of neighborhoods (hence numbers of meters and length of trace path), and the number of attackers and destinations. We have run the system on an 8-core machine, with 2GHz processor each and 16 GB memory. We have simulated a system as large as 32×32 neighborhood, with 100 attacker/intermediate pairs and 10 egress points (hence 210 VEs) and 28672 simulated nodes, and noted the desired scaling property. We also noted, and are correcting, abnormally large memory use for forwarding tables. We should be able to simulate models two or three orders of magnitude larger with these optimizations.

VI. Related Work

A. Network Simulation and Emulation

Network simulation and emulation are commonly used techniques to test and evaluate networking designs. Representative network simulators include ns-2 [7], ns-3 [8], SSFNet [9], GTNetS [10], and QualNet [11]. These network simulators generally cannot capture device or hardware characteristics because they do not involve real devices and live networks. On the other hand, the set of commonly used emulation testbeds include EmuLab [12], ModelNet [13], PlanetLab [14], DETER [15], VINI [16], X-Bone [17], and VIOLIN [18]. These emulators present more realistic alternatives to simulators because they combine real physical devices with emulation, but are limited by hardware capacity as they need to run in real time.

Some systems combine both simulation and emulation. One such example is CORE [19]. Recent work by Zheng et al. [1] is similar to CORE in that both of them use OpenVZ to run unmodified code and emulate the network protocol stack

through virtualization, and simulate the links that connect them together. A difference is that CORE has no notion of virtual time, while [1] implemented it in their work.

B. Virtual Time

Recent efforts have been made to improve temporal accuracy in para-virtualization. DieCast [20] and VAN [21] modify the Xen hypervisor to translate real time into a slowed down virtual time, running at a slower but constant rate. At a sufficient coarse time-scale this makes it appear as though VEs are running concurrently. Our treatment of virtual time differs from DieCast and VAN. The Xen implementations pre-allocate physical resources (e.g. processor time, networks) to guest OSes. In case that the resources have not been fully utilized by guest OSes, the idle VEs (like an operating system would) simply advance the virtual time clock at the same rate as they are busy. By contrast, we advance virtual time discretely, and only when there is an activity in the applications or network.

VII. Conclusion and Future Work

In this paper we present a system that integrates an OpenVZ-based network emulation system [1] into the S3F simulation framework [2]. The emulation allows native Linux applications to run inside the system; and the emulation is based on virtual time, which both provides temporal fidelity and facilitate the integration with the simulation system. We design and study the global synchronization and VE controlling mechanism in the system. Through analysis and experiment, we show that the virtual time error is bounded as a function of timeslice length, which itself is tunable at the cost of different execution speed [6]. We examine this error empirically, noting that, and then apply this technology to a case study of a DDoS attack within an Advanced Metering Infrastructure. Our experiments demonstrate the utility of the approach on an important modeling problem. Ongoing work includes studying the impact of such temporal errors on both CPU-intensive and communication-intensive application behavior.

The current system requires both OpenVZ and S3F to reside on the same shared memory multiprocessor. Our future work includes separating these to support multiple machines running virtual machine managers, not necessarily all running the same virtual system. We are also interested in means of estimating lookahead from within the emulation and providing it to the simulation, to accelerate performance.

VIII. Acknowledgment

This material is based upon work supported in part by the Department of Energy under Award Number DE-OE0000097, and by the Boeing Corporation.

References

[1] Y. Zheng and D. Nicol, "A virtual time system for openvz-based network emulations," in *2011 IEEE Workshop on Principles of Advanced and Distributed Simulation (PADS)*. IEEE, 2011, pp. 1–10.

[2] D. Nicol, D. Jin, and Y. Zheng, "S3F: The Scalable Simulation Framework Revisited," in *Proceedings of the 2011 Winter Simulation Conference*, Phoenix, AZ, December 2011.

[3] D. Nicol, J. Liu, M. Liljenstam, and G. Yan, "Simulation of large scale networks using ssf," in *Proceedings of the 2003 Winter Simulation Conference, 2003.*, vol. 1. IEEE, 2003, pp. 650–657.

[4] M. Liljenstam, J. Liu, D. Nicol, Y. Yuan, G. Yan, and C. Grier, "Rinse: the real-time immersive network simulation environment for network security exercises," in *Proceedings of Workshop on Principles of Advanced and Distributed Simulation, 2005. PADS 2005.* IEEE, 2005, pp. 119–128.

[5] R. Fujimoto, "Parallel discrete event simulation," in *Proceedings of the 21st conference on Winter simulation*. ACM, 1989, pp. 19–28.

[6] Y. Zheng, D. Nicol, D. Jin, and N. Tanaka, "A Virtual Time System for Virtualization-Based Network Emulation and Simulation," *Journal of Simulation*, 2011, To appear.

[7] L. Breslau, D. Estrin, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan, Y. Xu *et al.*, "Advances in network simulation," *Computer*, vol. 33, no. 5, pp. 59–67, 2002.

[8] The ns-3 project. http://www.nsnam.org.

[9] J. Cowie, D. Nicol, and A. Ogielski, "Modeling the global internet," *Computing in Science & Engineering*, vol. 1, no. 1, pp. 42–50, 2002.

[10] Gtnets, http://www.ece.gatech.edu/research/labs/maniacs/gtnets.

[11] Scalable network technologies. http://scalable-networks.com.

[12] Emulab, http://www.emulab.net.

[13] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker, "Scalability and accuracy in a large-scale network emulator," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 271–284, 2002.

[14] Planetlab, http://www.planet-lab.org.

[15] T. Benzel, R. Braden, D. Kim, C. Neuman, A. Joseph, K. Sklower, R. Ostrenga, and S. Schwab, "Experience with DETER: A testbed for security research," in *2nd International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities, 2006. TRIDENTCOM 2006.* IEEE, 2006, pp. 10–388.

[16] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford, "In VINI veritas: realistic and controlled network experimentation," in *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*. ACM, 2006, pp. 3–14.

[17] J. Touch, "Dynamic Internet overlay deployment and management using the X-Bone* 1," *Computer Networks*, vol. 36, no. 2-3, pp. 117–135, 2001.

[18] X. Jiang and D. Xu, "Violin: Virtual internetworking on overlay infrastructure," *Parallel and Distributed Processing and Applications*, pp. 937–946, 2005.

[19] J. Ahrenholz, C. Danilov, T. R. Henderson, and J. H. Kim, "CORE: A real-time network emulator," in *Military Communications Conference, 2008. MILCOM 2008. IEEE*, 2008, pp. 1–7.

[20] D. Gupta, K. V. Vishwanath, and A. Vahdat, "DieCast: testing distributed systems with an accurate scale model," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 407–422.

[21] P. K. Biswas, C. Serban, A. Poylisher, J. Lee, S.-C. Mau, R. Chadha, C.-Y. J. Chiang, R. Orlando, and K. Jakubowski, "An integrated testbed for Virtual Ad Hoc Networks," *Testbeds and Research Infrastructures for the Development of Networks & Communities, International Conference on*, vol. 0, pp. 1–10, 2009.