NETWORK-SIMULATION-BASED EVALUATION OF SMART GRID APPLICATIONS

BY

DONG JIN

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Doctoral Committee:

Research Assistant Professor Rakesh Bobba
Assistant Professor Matthew Caesar
Professor David Nicol, Chair
Professor William Sanders

ABSTRACT

The United States and many other countries are conducting a major upgrade of their electrical grids. The new "smart grid" is not a physically isolated network like the older power grid was, but a complicated network of networks. That greatly increases the security concerns, ranging from hackers who gain access to control networks or create denial-of-service attacks on the networks themselves, to accidental causes, such as natural disasters or operator errors. Therefore, it is critical to build a safe, resilient and secure communication environment for protecting the smart grid. Under this central theme, our research work has two strongly correlated streams.

First, to analyze large-scale networked systems (e.g., smart grid communication networks) with high fidelity, it is necessary for a testing system to offer both effective emulation (to represent critical software execution) and realistic simulation (to model background computation and communication). We have developed a network testbed using both parallel simulation and virtual-machine-based, virtual-time-embedded emulation to provide both functional and temporal fidelity for running large-scale networking experiments, so that technologies can be appropriately evaluated with modeling and simulation methodologies as well as with real software/hardware testing before they are integrated into the grid.

Second, we have utilized the testbed to study various cyber attacks in the smart grid, including a distributed denial-of-service attack (DDoS) in an advanced metering infrastructure (AMI) and an event buffer flooding attack on a supervisory control and data acquisition (SCADA) system (both for the Trustworthy Cyber Infrastructure for the Power Grid (TCIPG) Center at the University of Illinois at Urbana-Champaign), and also used it to evaluate a demand response design in a hierarchical transactive control network (as part of the Pacific Northwest smart grid demonstration project).

To my father, Shunrong Jin
my mother, Peiyun Huang
my wife, Xuan Zhuang
for their endless love and support

ACKNOWLEDGMENTS

I am very fortunate to get to know many people who have made my Ph.D. journey a pleasant and unforgettable experience.

First of all I would like to express my sincere gratitude to my adviser, Prof. David Nicol. You have been a steady influence throughout my Ph.D. life and future career; your ability to select and to approach compelling research problems, your high scientific standards, and your hard work set an example; you have listened to my ideas and discussions with you frequently led to key insights; you have oriented and supported me with promptness and care. Above all, you made me feel a mentor as well as a friend, which I appreciate from my heart.

I also wish to express my appreciation to all the professors serving on my thesis committee. Prof. Matthew Caesar, you have always been encouraging in times of new ideas and difficulties. I cannot tell how much I learned from you on those teaching assignments and research projects that we have worked together. Indeed, you implicitly set an example on how to be an excellent young professor for me. Prof. William Sanders, your vision and leadership has created the success of the entire Trustworthy Cyber Infrastructure for the Power Grid (TCIPG) story, and has guided me into this interesting and important research field. Prof. Rakesh Bobba, discussion with you always led to many valuable suggestions and constructive advice for my dissertation work. Also, I would like to express my gratitude on your grant proposal experience sharing and your help on the TCIPG reading group.

I have been very lucky to collaborate with many other great people who became friends over the last several years. Special thanks to my colleague, Yuhao Zheng and Huaiyu Zhu, I still remember the nights we stayed up together working on our papers, and the photo with only our three cars hand by hand sitting in the parking lot and waiting for us in the dawn. Sincere thanks are extended to my colleagues Tim Yardley and David Bergman for numerous stimulating discussions and help with experimental setup. Working with Tim and David on papers and demos has always been a pleasant and fruitful experience. I also want to express my gratitude to Prof. Carl Gunter for your support and guidance on various research projects we worked together. It was in your house that we installed monitoring/control devices so that we could detect the appliances in your kitchen and remotely control the heater. What you did made the serious and sophisticated research process a joyful experience to me. In addition, I want to thank my intern supervisor, Dr. Guanhua Yan, for many helpful suggestions, important advice and constant encouragement. Thank you for always being there for me as a mentor and a big brother.

I also want to express my thankfulness to the staff at the Information Trust Institute (ITI), Toshua York, Jenny Applequist, Amanda Brown, Tonia Siuts, Misty Houston, Keri Frederick, Lori Melchi, Andrea Fain and many others. Thank you so much for all the birthday and wedding gifts, which have made me feel like I had a home at ITI.

Finally, my special appreciation goes to my parents Shunrong Jin and Peiyun Huang, and my wife, Xuan, for your endless patience, encouragement and love.

The University of Illinois at Urbana-Champaign (UIUC) is a lucky place for me. Here I met so many great people; here I took the first step toward fulfilling my academic dream; here I met my wife (on the plane of our first trip from China to the USA), and will have our first baby, Marina. I wish everyone a peaceful and joyful life ahead. Wo Ai Ni Men!

Sincerely,
Dong (Kevin) Jin

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Motivations

Today's quality of life greatly depends on the successful operations of many large and complex communication networks, such as the Internet, cellular networks, and the communication infrastructure of national power grids. The health of those critical networks is at serious risk from both malicious cyber attack and accidental failure. Currently, the United States and many other countries are conducting a major upgrade of their power grids. Many critical components of the next generation of the grid, such as advanced metering infrastructure (AMI), substation automation, and supervisory control and data acquisition (SCADA) systems, require interconnections among various types of networks. Therefore, the grid is no longer a physically isolated network, but a complicated network of networks. If a design that will be implemented on such a scale has not been appropriately tested and evaluated, the result can be security concerns and performance issues for the critical infrastructure. Testing systems are an important approach to studying the cyber-security and efficiency of the power grid, because of their flexibility, controllability and lack of interference with real systems. These test systems may interact with actual equipment in order to reveal how the equipment will react in various situations. However, the scope of the grid makes it infeasible to create a physical test system anywhere near its full scale.

Network simulation and emulation help to alleviate the concern. Researchers have created various network testbeds that use emulation and/or simulation to conduct medium-to-large-scale communication network experiments. The emulation testbeds coordinate real physical devices and provide a configurable environment for live experiments, but with respect to networking are constrained by budget and the inherent limitations of lab equipment, restricting scalability and flexibility. On the other hand, network simulation provides better scalability and 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 and error-prone. Therefore, we have developed a parallel network simulator [1], and an OpenVZ-based network emulator embedded in virtual time [2], and have integrated the two systems based on the virtual time to provide a unique testbed to study smart grid applications in realistic large-scale settings [3]. The network emulation is used to represent the execution of critical software to ensure functional fidelity, and the parallel network simulation is used to model an extensive ensemble of background computation and communication. For example, to investigate a DDoS attack in an AMI network, we used emulation to run real smart meter programs (rather than meter models) to accurately represent behaviors of compromised meters, and used simulation to model a large number of intermediate meters including the Zigbee networking communication environment and background traffic generation.

1.2 Research Objectives and Contributions

1.2.1 A Large-Scale Network Simulation/Emulation Testbed

We want the capability to embed a smart grid subsystem within a high-fidelity virtual environment and quantitatively assess the behaviors under realistic conditions, the reliability in the face of faults, the effectiveness of security defenses, and the presence of unknown vulnerabilities. The high-fidelity virtual environment is the key.

Our first contribution is that we developed a large-scale, high-fidelity network emulation/simulation testbed, which is currently serving as a central component in the TCIPG smart grid laboratory at the University of Illinois at Urbana-Champaign [4]. The testbed integrates a parallel network simulator, for which we developed a rich set of network protocols and applications used in the smart grid (such as ZigBee, Ethernet, DNP3, Modbus, and OpenFlow), to model the large-scale network environment, and an OpenVZ-based network emulator to represent the execution of critical software. Because of the parallel simulation kernel, the lightweight virtualization technology, and the use of efficient models to reduce simulation cost, the unique testing system provides good scalability for conducting large-scale network experiments on smart grid technologies. The controllability, flexibility, and repeatability offered by the testbed are also helpful in the evaluation and analysis of many emerging technologies in the smart grid.

1.2.2 Virtual-Time Integration of Simulation and Emulation

Integration of emulation and simulation systems is not a trivial task, since emulation advances the state of the program with respect to real, "wallclock" time, and simulation advances the state of the model with respect to the more abstract, "virtual" time. Therefore, there are research problems related to interactions and management of virtual time between emulations and simulation. Our design needs to address the inherent errors due to the VM control as well as the exploitation of parallelism. Our contributions include the following:

- We integrated the "virtual time" concept in a virtual-machine-based (OpenVZ) emulation system to ensure temporal fidelity.
- We developed global synchronous scheduling algorithms to enable seamless connection between the emulation (VE entities) and the simulation (simulation threads).
- By finding analytical bounds for the error, we developed a response to serious concerns about the unavoidable uncertainties involved in emulation behavior, and we produced empirical data showing that the error is as small as the minimum system execution unit.

1.2.3 Modeling Smart Grid Communication Networks

Modeling of smart grid communication networks involves unique challenges. First, the testing system may need to interact with real devices (e.g., relays, phasor measurement units, and/or data aggregators) in the course of experiments, and our simulation/emulation testbed must keep up with those devices in real-time. Second, networks in the smart grid are often large-scale (e.g., SCADA networks or AMI networks), ranging from thousands to millions of entities. Therefore, scalability and performance of a testbed are critical. In addition to developing the parallel simulation kernel and the lightweight kernel-level virtualization technology, we also investigated means to reduce simulation cost. The major contributions are summarized as follows.

Switch Models

Switched networks, such as SCADA networks and utility enterprise networks, are used widely in smart grid communication. From real traces, we observed that the time-scale difference between applications and switches suggests that exact latency is not as important as average latency, but that a packet loss under TCP impacts application behavior. Therefore, we developed latency-approximate scheduling for a weighted fair queuing discipline [5]. The new switch models significantly reduce simulation cost with only a small loss of fidelity.

Background Traffic Models

The cost of simulating a network can easily represent the overwhelming majority of the overall cost of performing a simulation experiment. In some applications, only a small fraction of traffic is of specific interest. For example, in models of SCADA networks, we are mainly interested in the detailed behavior of specific flows (e.g., a DNP3 or Modbus connection between a control station and a particular substation), and are interested in other flows only in-so-far as they consume resources and affect the behavior of the detailed flows.

Structured traffic patterns enable compact and efficiently executed background traffic. We have developed techniques for modeling background traffic through switches that use Fair Queuing scheduling and First-Come-First-Served scheduling [6]. The background traffic models enable experiments that include low-cost background traffic mixed with detailed foreground traffic. The new models run extremely fast relative to packet-based flow simulation (with speedups exceeding 3000), and the foreground flows are still accurate enough for our purposes.

Software-Defined Networks

A software-defined network (SDN) design decouples the data plane and the control plane of a switch or a router. The logically centralized controller can directly configure the packet-handling mechanisms in the underlying forwarding devices (e.g., drop, forward, modify, or enqueue). The benefits of applying SDN in the context of the smart grid include elimination of the need to configure network devices individually; consistent policy enforcement across network infrastructures (such as policies for access control, traffic engineering, quality of service, and security); the ability to define and modify the functionality of a network after deployment; and evolution of products at software speeds rather than at standards-body speed. Therefore, we extended the testbed to support OpenFlow-based SDN simulation and emulation. However, the centralized controller designs of SDN impose potential performance issues in parallel discrete-event simulation.

Our contributions include a demonstration of how to exploit typical SDN controller behavior to deal with the performance bottleneck of centralized controllers, and the results of our investigation of methods for improving model scalability, including an asynchronous synchronization algorithm for passive controllers and a two-level architecture for active controllers [7]. The techniques not only improve the simulation performance, but also are valuable for designing scalable SDN controllers. In addition, the SDN simulation/emulation testbed is a useful tool not only for smart grid networks, but also for SDN-based research in general.

1.3 Thesis Outline

The remainder of this dissertation is structured as follows.

Chapter 2 presents our network testbed consisting of a parallel simulator and virtual-machine-based emulator for conducting large-scale network experiments with high functional and temporal fidelity. We start with background on network simulation and emulation, the parallel simulation kernel, and virtual time in Section 2.1, followed by an overview of the system design in Section 2.2 and implementation details of each component, as well as the system integration, in Section 2.3. We then report on system evaluations, including error analysis in Section 2.4, performance analysis in Section 2.5, and application-level fidelity analysis in Section 2.6. We then describe featured models developed in the system, including background traffic models in Section 2.7 and software-defined networks in Section 2.8.

Chapter 3 describes how we use the network testbed to conduct security and performance studies of various smart grid applications. Section 3.1 investigates a distributed denial-of-service attack using the C12.22 trace service in an advanced metering infrastructure (AMI) network. Section 3.2 explores an event buffer flooding attack in DNP3-controlled supervisory control and data acquisition (SCADA) networks. Section 3.3 proposes multiple designs for the demand response algorithms and evaluates the designs on a large-scale hierarchical transactive control network.

Finally, Chapter 4 summarizes the conclusions made in this dissertation and sketches the directions for future research.

CHAPTER 2

A NETWORK TESTBED WITH PARALLEL SIMULATION AND VIRTUALIZATION-BASED EMULATION

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 motivates us to create a high-fidelity and large-scaled network testbed, since many critical component in the smart grid, such Advanced Metering Infrastructure (AMI) and Supervisory Control and Data Acquisition System (SCADA), are large-scaled network, and smart grid itself is a complex network of networks. Our testbed uses a version of OpenVZ modified to operate in virtual time [8] with a new parallel network simulator, S3F [1], which was inspired by SSF [9], and RINSE [10].

Our testbed uses emulation to represent the execution of critical software, and simulation to model an extensive ensemble of background computation and communication. For example, 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 wireline networks, as well as models of meters that contribute to the network traffic load but are not otherwise particular objects of study.

2.1 Background

2.1.1 S3F — The Scalable Simulation Framework

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. More details about S3F are in [1]. 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. Figure 2.1 depicts the basic elements described above.

S3F supports parallel execution, which requires synchronization among the timelines. Multiple timelines may run simultaneously to exploit parallelism, but they have to be carefully synchronized to guarantee global causality. Much of the motivation and design of S3F is to support synchronization more-or-less transparently, yet provide hooks to the sophisticated modeler to transfer modeling information to the synchronization engine that is used to improve performance.

The basic idea behind synchronization is simple. Use information about latencies across communication paths established between outchannels and inchannels to establish a window of simulation time with the property that no activation written to an outchannel at a time within that window will be received by an inchannel *on a different timeline* at a time also within that window. The windows are implemented with barrier synchronizations. The upshot is that cross-timeline events do not have to be immediately

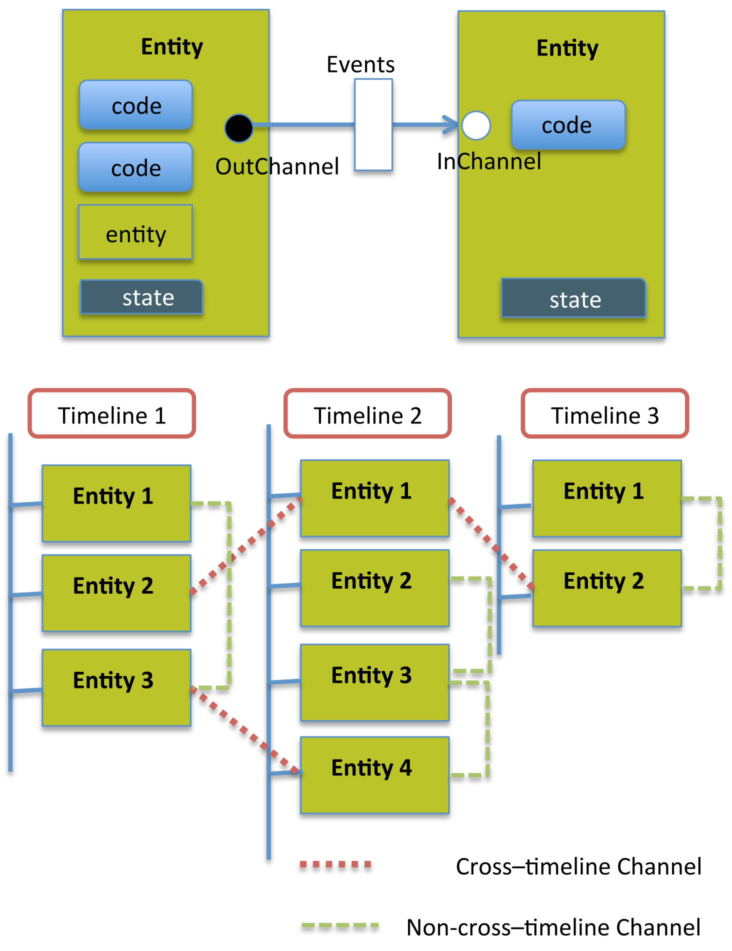


Figure 2.1: S3F Basic Elements

delivered—their receipt lies on the other side of a barrier synchronization, so they can be buffered pending a step where synchronized timelines exchange such activations, and integrate them into their target timeline event lists. The synchronization mechanism is built around explicitly expressed delays across channels whose endpoints 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.

Figure 2.2 illustrates the concept. Suppose the timelines have all advanced to simulation t, and have in their event lists all events known (at that time) to be executed. Execution of some of these events may of course introduce other events into the list, but every future event known at time t is in the event list of the timeline that will execute it. The timelines are working to

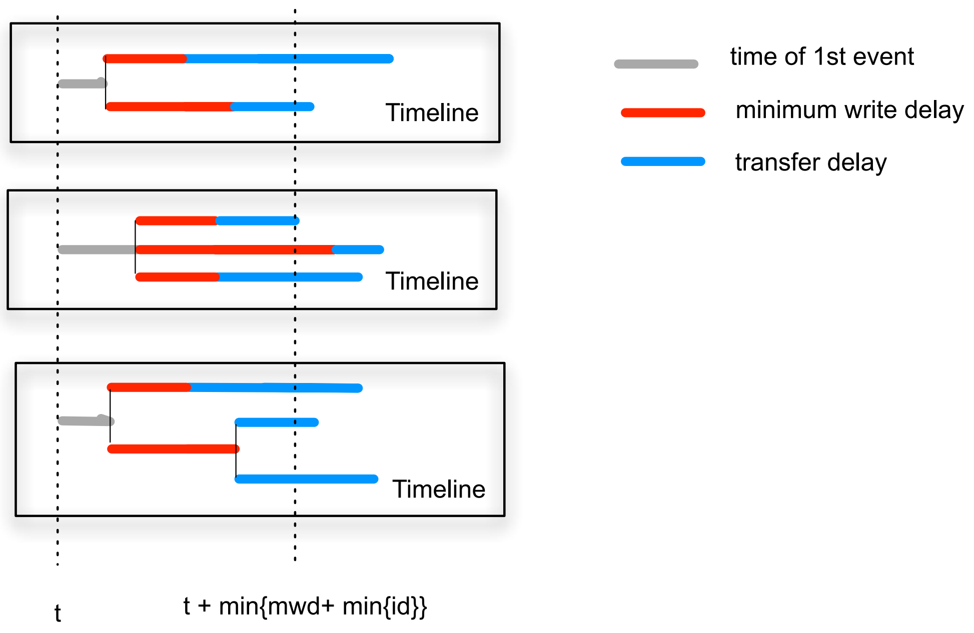


Figure 2.2: Synchronization Window

coordinate how far ahead in simulation time they can safely advance. Each timeline identifies the time of the first event it has in its list, which is a lower bound on when next the timeline will execute a process that performs a write that crosses timeline boundaries. Each timeline also identifies the minimum over all mapped cross-timeline outchannel/inchannel pairs of the outchannel's minimum per-write delay, and the transfer delay to the inchannel. That is, each timeline i identifies a lower bound on the arrival time of any future cross-timeline write as

$$L_i = n_i + B_i$$

where we separate the bound into time of next event n_i, and

$$B_i = \min_{outchannel\ c} \left\{ w_c + \min_{cross\text{-}timeline\ mapped\ inchannels\ k} t_{c,k} \right\}$$

where w_c is the minimum per-write delay declared for outchannel c, and $t_{c,k}$ is the transfer time between c and inchannel k owned by an entity aligned to a different timeline than c's owner. To establish the synchronization window the timelines offer their respective L_i values to a global min-reduction. On being released from the associated barrier, each timeline can read the minimum among all offered values—this is the upper edge of the window. Simulation time may advance to one clock tick less than this value.

The value of n_i may change from window to window, but B_i need not, at least so long as there are no changes in the inchannels mapped, the transfer delays, or the minimum per-write delays, the result of a prior computation can be used. However one cannot always expect the set of mapped outchannels/inchannels to remain constant, and changes in model state may allow (or require) the model to change an outchannel's per-write minimum delay, or some transfer delay. S3F allows dynamic `unmap` and `mapto` calls, and dynamic changes to minimum per-write and transfer delays. S3F tries to minimize the impact of those changes by being smart about recomputing B_i. Once computed, S3F also counts the number of cross timeline outchannel/input pairs that actually achieve the computed value of B_i. Any change in mapped channels or their delays that decrease that count need not trigger a recomputation so long as the change leaves at least one outchannel/input pair with B_i's value. Likewise any change that cannot possibly lower B_i (e.g., *increasing* either the per-write minimum delay or a transfer delay) will not trigger recomputation of B_i. However, requests for changes that do affect B_i raise a flag that later triggers recomputation of B_i.

After a timeline has processed all the window's events, implemented any buffered delay changes, and recomputed B_i (if needed), it sets its clock to the time of the window edge minus one clock tick, and enters another barrier synchronization to wait for all other timelines to do so also. When they are released, they transfer buffered events that resulted from cross timeline writes into their event lists (thereby ensuring that each time of next event n_i is what it needs to be), and compute the upper edge of the next synchronization window.

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 condition. 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 in S3F at the end of a synchronization window, when all timelines are blocked, and events and control information are passed between emulation and simulation. Details about these interactions will be discussed in Section 2.3.1.

2.1.2 Network Simulation and Emulation

Network testing systems are widely used for evaluating, debugging and analyzing new and existing network designs. Network testbeds generally fall into three categories: physical testbeds, emulation testbeds and simulation testbeds. The physical testbeds provide realistic networking environment for users to conduct (sometimes live) networking experiments, such as WAIL [11] and PlanetLab [12]. However, users have limited controllability and flexibility on the network scenarios they could run, e.g., it is difficult to test a network protocol/design on large-scale networks or with different network topologies. Emulation and simulation testbeds have been developed to address the shortcomings of the physical testbeds with better scalability, flexibility, controllability and repeatability, but less accuracy and realism.

Network emulation testbeds utilize virtualization technologies to create flexible virtual network topologies with better scalability as compared with physical testbeds. Emulation runs software in virtual machines which share lower layer resources (even the hardware platform) transparently. The critical differences between emulation and native execution include: (1) native execution is always tied to "wall-clock" time; (2) interfaces to emulation is standard networking; and (3) specialized hardware functionality (e.g., DSP) is hard to emulate. Ordinary network emulators are also embedded in real-time, which causes temporal fidelity issues. Researchers have investigated emulation virtual-time systems to address this, and details are presented in Section 2.1.3. Researchers have built emulation testbeds on a variety of computing platform, including computer clusters (such as EmuLab [13], ModelNet [14], and DETER [15]), distributed computing platform (such as VINI [16], VIOLIN [17], and X-Bone [18]), and specially programming devices (such as ORL[19], ORBIT [20] and CMU wireless emulator [21]).

While emulation executes "unmodified software" to produce behavior and advance the experiments simulation executes "model software" (such as models of network protocol and network devices). Simulation uses abstractions to accelerate changes to model states, and requires lower memory needs than emulation. Hence, simulation testbeds typically have better scalability than emulation testbeds, but less functional fidelity. Simulation can scale-up to explore large-scale networks (e.g., SSFNet [22], GTNetS [23] and ROSSNet [24]). It may run faster or slower than real-time, while most emulation testbeds are tied to "wall-clock" time and are limited by hardware capacity as they need to run in real-time. In addition, developing simulation models is labor-intensive and often error-prone. Representative network simulators include open source ones, such as ns-2 [25], ns-3 [26], SSFNet [22], GTNetS [23], OMNeT [27], and J-Sim [28], and commercial ones, such as OPNET [29] and QualNet [30].

Some systems combine both simulation and emulation, such as ns-2 [25], ns-3 [26], and CORE [31]. Our S3F/S3FNet testbed provides the emulation functionality which 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 as well as ns-2 and ns-3 have no notion of virtual time, while our testbed implemented the emulation virtual-time in the OpenVZ Linux kernel [8].

2.1.3 Virtual Time

Ordinary network emulators are embedded in real-time, but network simulators advance experiments in virtual-time. Therefore, integration of emulation and simulation has temporal fidelity issues. The software managing virtual environments (VEs) takes its notion of time from the host system's clock, which means that time-stamped actions taken by VEs whose execution is multi-tasked on a host reflect the host's serialization. This is deleterious from the point of view of presenting traffic to a network simulator which operates in virtual time. Here is one simple illustrative example: imagine a set of synchronized emulated devices that in the real system all generate

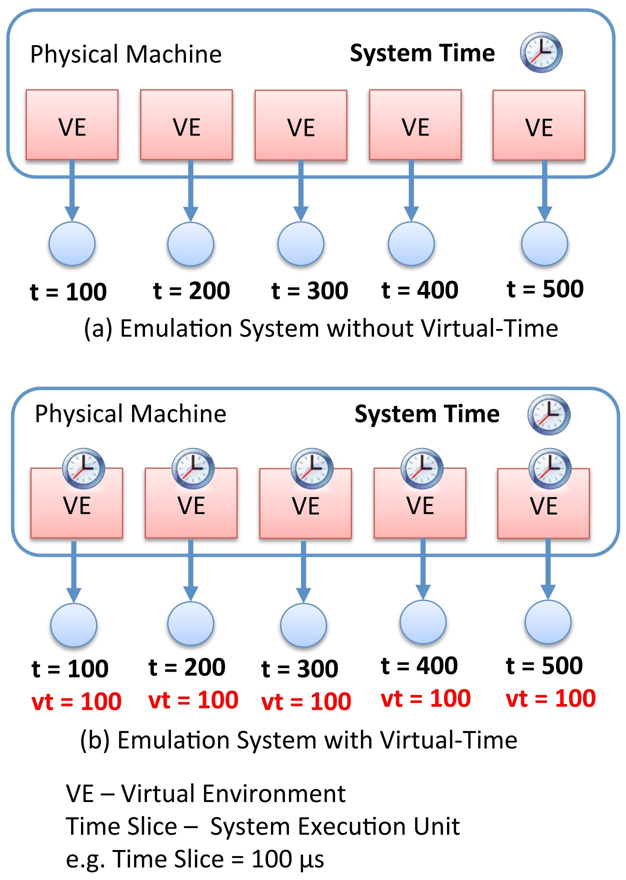


Figure 2.3: Emulation Temporal Fidelity Illustration Example: Simultaneous Traffic Generation among VMs

a message within the same small δ of time ($\delta <<$one time slice). Virtual machine manager (VMM) separates the packet generation in real-time by time-slice allocation as shown in Figure 2.3 (a). From the network simulator's viewpoint, the packets are generated one by one, each is separated by approximately one time slice. Assuming the network scenarios have shared medium, which every packet is competing for medium access, or the packets will join the same queue of a connected switch, the observed behaviors are incorrect. Ideally, each VE would have its own virtual clock, so that time-stamped accesses to the network would appear to be concurrent rather than serialized as shown in Figure 2.3 (b).

Recent efforts have been made to improve temporal accuracy in para-virtualization. DieCast [32] and VAN [33] 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 VEs idle (like an operating system would) simply to advance the virtual time clock. By contrast, we advance virtual time discretely, and only when there is an activity in the applications or network.

Our approach is related to the LAPSE system [34]. LAPSE simulated the behavior of a message-passing code running on a large number of parallel processors, by using fewer physical processors to run the application nodes and simulate the network. In LAPSE, an application code is directly executed on the processors, measuring execution time by means of instrumented assembly code that counted the number of instructions executed; application calls to message-passing routines are trapped and simulated by the simulator process. The simulator process provides virtual time to the processors such that the application perceives time as if it were running on a larger number of processors. Key differences between our system and LAPSE are that we are able to measure execution time directly, and provide a framework for simulating any communication network of interest while LAPSE simulates only the switching network of the Intel Paragon.

2.1.4 OpenVZ-Based Network Emulation

We expand the capacity of S3F/S3FNet by integrating it with the OpenVZ-based network emulation. OpenVZ is a light-weighted container-based virtualization technology in Linux [35]. OpenVZ enables multiple isolated execution environments within in a single Linux kernel, called *Virtual Environments (VEs)*. A virtual time system has been developed in the OpenVZ kernel to make VEs perceive virtual time as they were running concurrently on different physical machines [8].

From operating system's point of view, a process can either have CPU resources and be running, or be blocked and waiting for I/O (ignoring a "ready" state, which rarely exists when there are few processes and ample resources). The wall-clock continues to advance regardless of the process state. Correspondingly, in the our OpenVZ-based emulation system, the virtual time of a VE advances in two ways. When a VE is having the CPU and is running, its virtual clock keeps advancing in the same speed as the wall-clock. This is the same as real world. On the other hand, when the VE is waiting for an I/O request, and such I/O request should be a simulated one (e.g. network requests), the scheduler needs to make the VE perceives time in the same way as real world. Specifically, while waiting for I/O, the VE is suspended and therefore its virtual clock is not advancing. Instead, the scheduler captures the I/O request, simulates it, and returns it to the VE. Then the scheduler adds the simulated I/O time to the VE's virtual clock, and release the VE to let it run again. Consequently, the VE perceives virtual time as if it were running in real world. Such notion of virtual time is shown in Figure 2.4. Note that simulated I/O time is normally irrelevant to the wall-clock time. The actual (wall-clock) time it takes to simulate an I/O request can be either faster or slower than real (wall-clock), depending on the model complexity and the simulator.

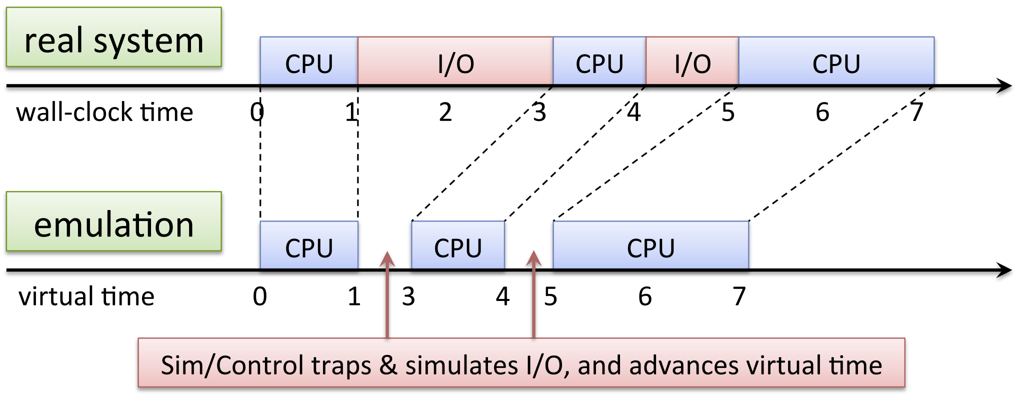


Figure 2.4: Time Advancement: Wall-Clock vs. Virtual Time

A VE runs real applications which interact with emulated IO devices (e.g. disks), generate and receive 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.

2.2 System Design

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. 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 wall-clock time, thereby increasing temporal fidelity [8] (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. Coordination of activity between OpenVZ's emulation and S3F's simulation is handled by new extensions to S3F, described in this section. The current system can run 100+ OpenVZ virtual machines and simulate millions of devices on a single multi-core server.

Figure 2.5 depicts the overall system design architecture of our system, which integrates the OpenVZ network emulation into a S3F-based network simulator on a single physical machine. Structurally, every VE in the OpenVZ model is represented in the S3FNet model as a host within the modeled network, where S3FNet is a network simulator built on top of S3F. 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 2.5. 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

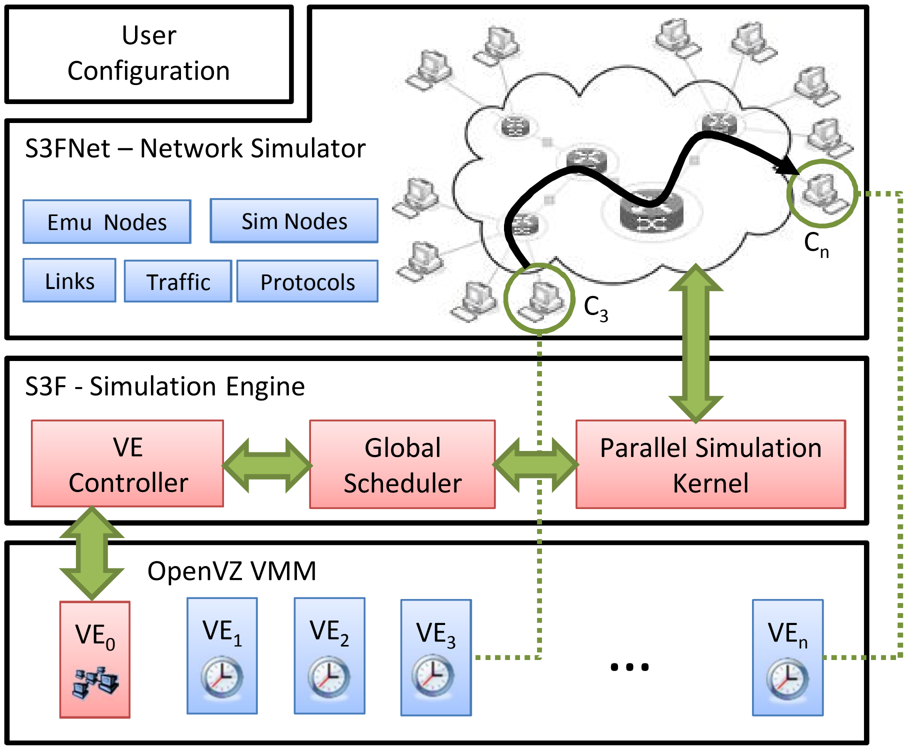


Figure 2.5: System Design Architecture

of the two systems and to make the emulation integration nearly transparent to S3FNet. The system capable of running large-scale and high fidelity network experiments with both emulated and simulated nodes.

2.2.1 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 [8], 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 2.5, 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 2.3.

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 [36], [37], 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.

2.2.2 Simulation/Emulation Coordination

Our design prohibits OpenVZ VEs from executing concurrently with the S3F simulation timelines. Our view is that the VE's operate as traffic sources. Correspondingly, 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 timestamp 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.

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 have 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.

2.2.3 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 [2], and the details are discussed in Section 2.2.4 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 $t_0 + \alpha * TS$.

The integration of the simulation and emulation framework is based on the fact the both system can run on virtual time. The work includes design of synchronization, event passing and virtual machine control mechanisms in the hybrid system for safe and efficient experiment advancement. We also 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). Details are described in Section 2.4.

2.2.4 Variable Timeslice

Our virtual time system in OpenVZ can reduce the temporal error by using smaller timeslices. But the minimal allowed timeslice is subject to the frequency of hardware timer interrupts. To achieve smaller timeslices, we need to raise the frequency of timer interrupts, i.e. the HZ value in Linux kernel. For example, by raising the HZ value from 1000 to 4000, the smallest allowed timeslice is 250 μs, rather than 1 ms. Actually, with HZ = 4000, any timeslice length of n $\cdot$ 250 μs is allowed, where n is an integer. However, changing the HZ value has some side effects to the Linux kernel, which must be dealt with for the kernel to work properly. Such side effects are mainly due to counter overflows or integer operation overflows [38], as some Linux kernel developers did not anticipate that the HZ value might be set so large.

Higher HZ frequency and smaller timeslice has overhead, which comes from at least two sources: (1) more frequent timer interrupt handlings, and (2) more frequent context switches. We model the extra time consumed by each timer interrupt as T_{int}, and model that consumed by each context switch as T_{CS}. Let TS be the length of a scheduler timeslice. Then the ratio of time spent actually working during a timeslice to the length of that timeslice (i.e., system efficiency) is

$$\rho = \frac{TS - T_{CS} - k \cdot T_{int}}{TS} = 1 - T_{CS} \cdot \frac{HZ}{k} - HZ \cdot T_{int}$$

where $TS = \frac{k}{HZ}$, making k the total number of timer interrupts within one timeslice of length TS. Observe that for fixed HZ, system efficiency is an increasing concave function of k (i.e., increasing TS), which suggests (and will be verified) that increasing TS from very small values will have the largest positive impact on efficiency, after which efficiency approaches an asymptote of $1 - HZ \cdot T_{int}$. As TS increases, accuracy decreases linearly. All of this means that for a given accuracy constraint, e.g., no error greater than E, we seek to maximize the expression above subject to $\frac{k}{HZ} \leq E$. By concavity, this occurs when $k = 1$, and $HZ = \frac{1}{E}$.

2.3 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.

2.3.1 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 [1]. 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. First, 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. Second, our system design ensures that the simulation can never run ahead of the current emulation time. Third, once the OpenVZ emulation starts to run, it has to run for at least one timeslice [8], 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 provided in the following list:

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 2.2.3
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 (2.1) illustrates how ESW is calculated:

$$ESW = \max\left\{\alpha * TS, \min_{timeline\ i}\{P_i\} - t_{emu}\right\} \tag{2.1}$$

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 (2.2) illustrates how SSW is calculated:

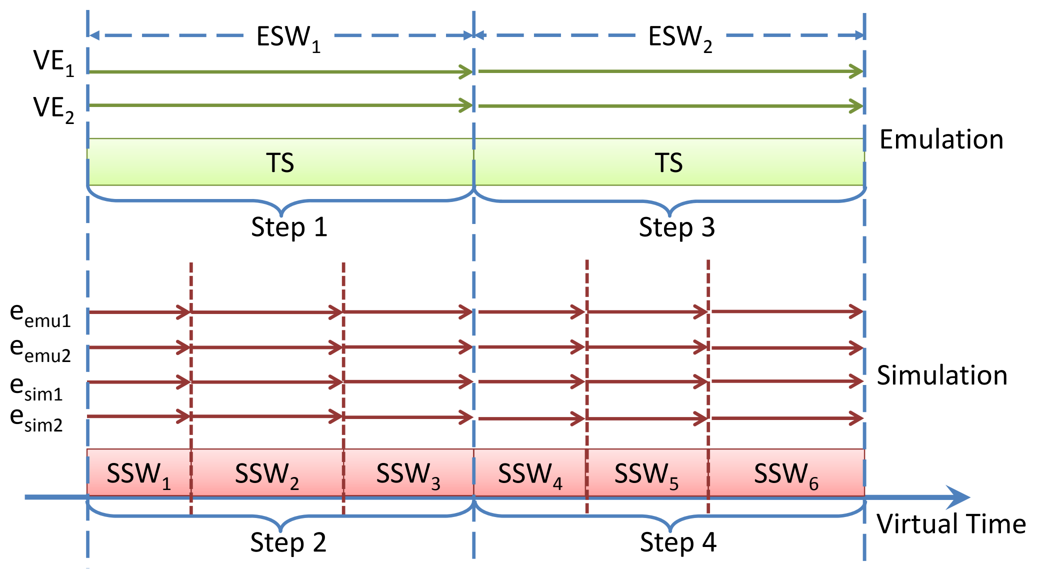


Figure 2.6: System Advancement with Global Synchronization, Emulation Timeslice $\geq$ Simulation Synchronization Window

$$SSW = \min\left\{t_{emu}, \min_{timeline\ i}\{Q_i\}\right\} - t_{sim} \tag{2.2}$$

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 2.6 and Figure 2.7 illustrate the behavior of the system in the two cases respectively.

In both cases, the simulation advancement is bounded by ESW. How-

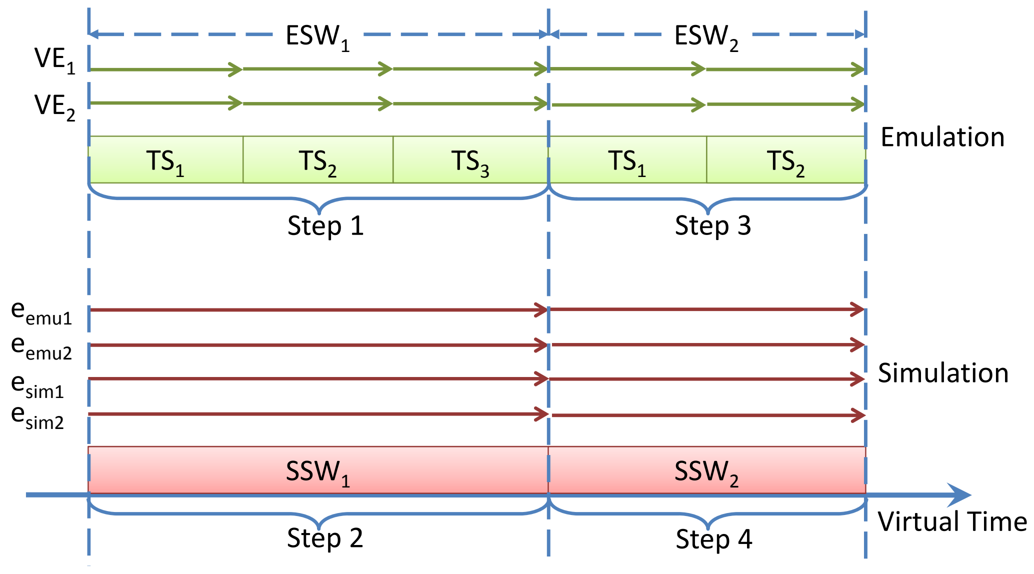


Figure 2.7: System Advancement with Global Synchronization, Emulation Timeslice < Simulation Synchronization Window

ever, in case 2, the simulation helps to improve 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 2.3.2.

2.3.2 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 [8].

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.

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**
      $VE_i.offset \mathrel{+}= VE_i.clock - barrier$
      $VE_i.clock = barrier$
      $VE_i.done =$ true
    **end if**
  **end while**
  calculate emulation lookahead for $VE_i$
**end for**
$t_{emu} = barrier$
```

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.

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.

2.4 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
$listener$	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 2.8 for the 400 Mb/s case. The x-axis indexes the packet, the y-axis shows the times associated

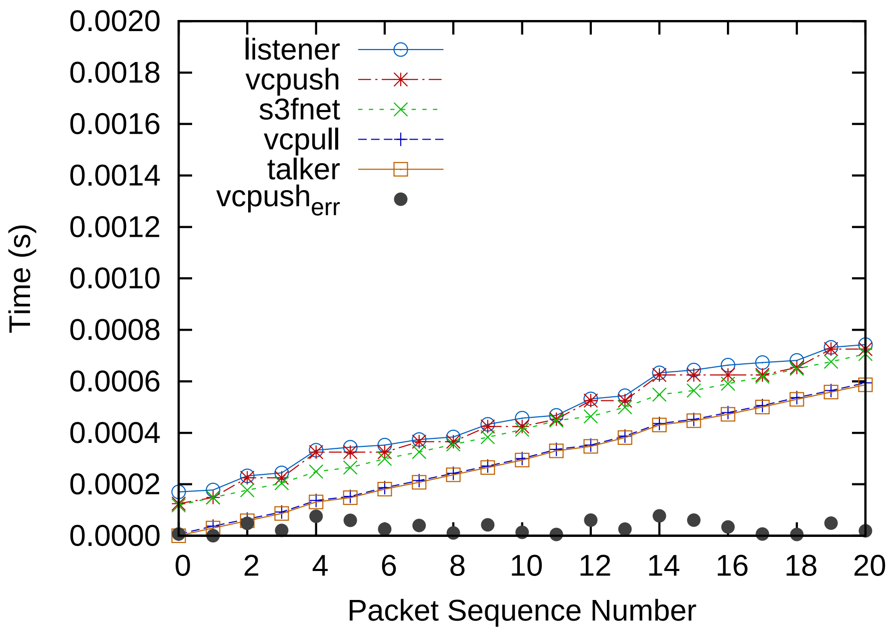


Figure 2.8: Timestamps during Packets Traverse Route: Sending Rate = 400 Mb/s

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 2.8. As explained in Section 2.2.3, 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 [2].

2.5 Performance Analysis

The testing platform for conducting all the experiments is built on a Dell PowerEdge R720 server with two 8-core processors (2.00GHz per core) and 64 GB RAM, and installed with 64-bit Linux OS. By enabling the hyper-threading functionality, our network simulator can concurrently explore up to 32 logical processors. Given the same size of a network model, i.e. identical topology, and traffic volume and pattern, we would like to investigate means to improve the simulation execution speed without losing accuracy, particularly, performance impact of (1) the number of timelines, and (2) the size of synchronization window.

Number of Timelines: The concept of "timeline" in our testbed is designed for exploiting parallelism of a network model. Simulated nodes, such as hosts, switches, and routers, are partitioned into groups with user-specified rules, such as geographic location or traffic balancing. Each group is aligned to a timeline, and each timeline is assigned to a core during the model execution. Timelines are synchronized through barriers. By increasing the number of timelines, the entire simulation work is distributed to more cores and thus increases the degree of parallelism, at the overhead cost of synchronization. In the first set of experiments, we set up a network with 1024 simulated hosts, among which every two hosts pair up a server-client connection (512 links in total) with 1 Gb/s bandwidth. Each sever is sending constant bit rate traffic flow to its client, and the server-client pairs are evenly distributed among timelines. The above settings minimize the impact of unbalanced simulation workload among multiple timelines with no cross-timeline events, and well-

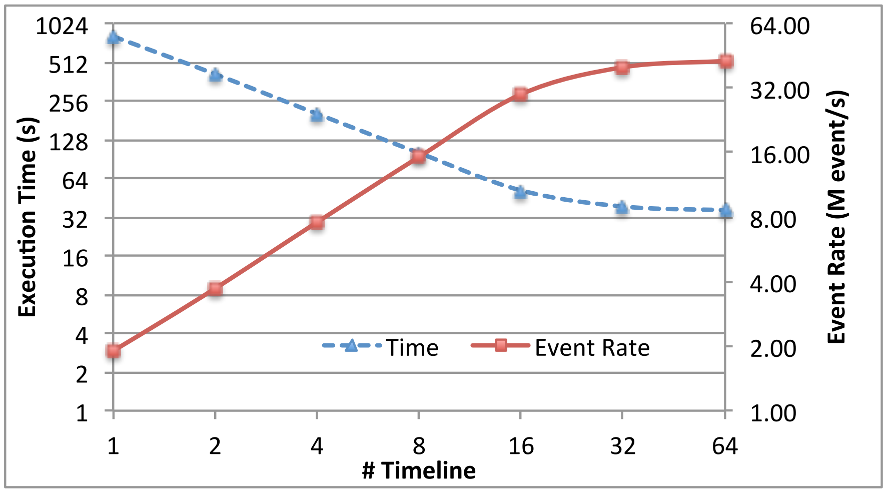


Figure 2.9: Optimal Simulation Speedup on a Multi-Core Architecture Platform

balanced traffic patterns. The inter-packet gap is set to 1 μs to generate sufficient simulation workload. Each experiment is scheduled to run for 1 second in virtual time, and generates 1.54 G events in total. Figure 2.9 plots the execution time in wall-clock time and event execution rate as we increase the number of timelines.

The simulator shows a nearly perfect speedup behavior as we increase the number of timelines up to 16: the execution time continues to drop in half, and the event execution rate continues to be doubled as we double the number of timelines. The improvement slows down when the number of threads grows up to 32 due to the heavy involvement of the hyper-threading technique. We have conducted the same set of experiments again with hyper-threading turned off, and the comparison shows that using hyper-threading can achieve a 1.36 speedup factor for our simulator when the number of timelines is greater than or equal to 32. The results show that given sufficient workload within each synchronization window, our simulator can maximize the hard-ware parallelism capabilities by creating the same number of timelines (i.e. software threads) as the number of available logical processors, with an event rate as high as 40 million-events/second. Further increasing the number of timelines gives us little performance gain. The aforementioned experimental results profile the best achievable speedup performance of our simulator by maximizing the work in the parallel section of our simulator. The rest overhead is due to OS-dependent control elements, such as multi-thread scheduling and inter-processor management that are beyond our control.

Synchronization Window: Synchronization windows indicate how long the emulation or the simulation can proceed without affecting entities on other timelines. The length of the windows are computed at synchronization barriers based on the 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. In this set of experiments, we investigate the performance impact of synchronization window on our simulation/emulation testbed. We set up a network with 64 emulated hosts, among which every two hosts pair up a server-client connection, and 32 links in total. Each sever sends constant bit rate UDP traffic flow with constant 1500 byte packet size to its client. The emulation time slice is set to 1 ms, and the networking environment created in S3FNet has 1 Gb/s bandwidth and 1 ms link delay. We vary the sending rate with 100 Kb/s, 1 Mb/s and 10 Mb/s in the preceding scenarios and record both the emulation time and the simulation time every 10,000 packets. We then pre-calculate a constant lookahead based on the observed average inter-packet gap, essentially creating larger synchronization windows, and re-run the experiments for comparison. We observe the nearly identical sending/receiving traffic patterns for each scenario with and without lookahead, which indicates little experiment accuracy loss with the presence of the lookahead. Figure 2.10 compares the execution times for both emulation and simulation.

With emulation lookahead, the execution time, both emulation and simulation, significantly reduces for 100 Kb/s and 1 Mb/s sending rate. Here is the explanation: for 1 Mb/s sending rate, given 1500 byte packet size, the average inter-packet time is around 12 ms, which equals to 12 timeslices; and accurate emulation lookahead should predict that amount – promise that a VE will not generate any events within the next such amount of time, and offer this value to S3F for computing the next emulation synchronization window (ESW). The new ESW increases to approximately 12 timeslices in length and thus minimizes the emulation over-head. Since the emulation is now running far ahead of time as compared with the case without emulation lookahead, and no events are injected into the simulation's event lists, the simulation also takes advantage of the empty event lists to compute a large simulation synchronization window. Therefore, the execution times on both simulation and emulation are significantly reduced, and same is true for 100 Kb/s case. However, little improvement is observed for the 10 Mb/s case, because ESW generated by emulation lookaheads (1.2 ms) is close to one time slice. Another observation is that by offering accurate lookahead, the execution times for transmitting 10,000 packets with all the sending rates are very close. This is actually one benefit with our virtual-time-embedded emulation system that we can accelerate low traffic load emulation experiments by utilizing available system processing resources.

Providing good simulation lookahead is always challenging for conservative network simulation, and our virtual-time-based simulation/emulation testbed gives another opportunity for providing good emulation lookahead. The aforementioned experimental results clearly indicate the huge performance gain that a good lookahead mechanism can bring to our system, and thus strongly motivate our ongoing work of investigating emulation lookahead from source code/binary analysis.

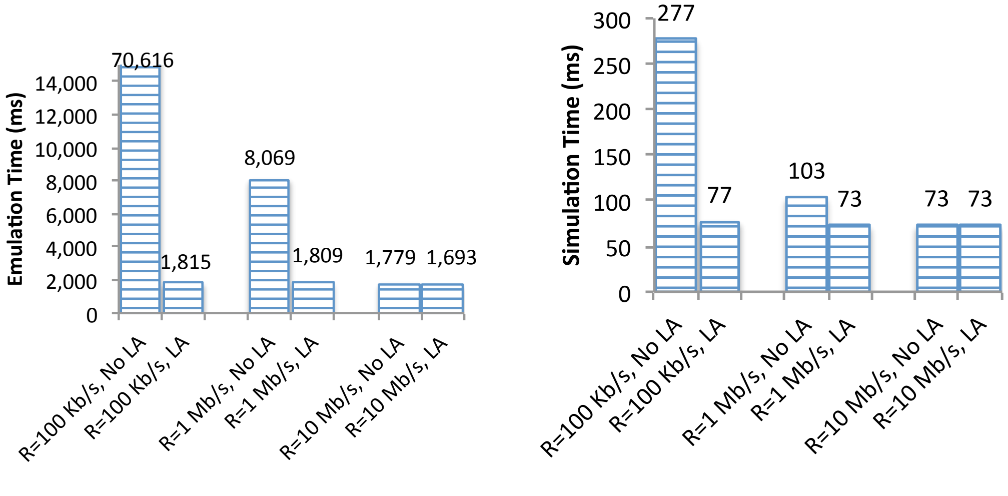


Figure 2.10: Execution Time Comparison with Lookahead (R - Sending Rate, LA - Lookahead)

2.6 Application-Level Fidelity Analysis

Validation reveals that there are network timing errors whose magnitude depend on the length of a virtual machine execution timeslice. A natural question asks to what degree these errors impact the behavior of *applications*. For instance, if an application is relatively insensitive to these errors, we can increase performance by allowing larger emulation timeslices. We study a variety of applications with different network and CPU demands. We find, surprisingly, that difference in application behavior due to simply using OpenVZ often dominate the errors, implying that we need not be overly concerned about errors due to larger timeslices.

2.6.1 Overview

Our testbed provides both functional and temporal fidelity, by embedding the virtual machines in virtual time [8]. However, small temporal errors are introduced by the OpenVZ design, on the scale of a timeslice given to virtual machines, because OpenVZ interacts with a virtual machine only at the beginning and end of a timeslice. This work asks how temporal errors affect behavioral fidelity with respect to application-specific metrics. We study applications that are network-intensive, and ones that are CPU-intensive. We also evaluate behavioral fidelity on ICMP, UDP and TCP by studying FTP, web browsing, ping, and iperf. Our study concerns three configurations: native Linux, native OpenVZ, and our emulation/simulation testbed. By comparing native Linux and native OpenVZ we identify deviations that are due solely to OpenVZ's implementation. Comparing native Linux and our emulation/simulation testbed we see the impact of those errors and errors introduced by our testbed.

The experimental results show that application timing errors introduced by the virtual time system are bounded by the size of an emulation timeslice in both network-intensive and CPU-intensive applications, and that the network round-trip-time (RTT) may increase by as much as two emulation timeslices. Applications whose metrics are not affected by changes in the RTT of that scale are insensitive to emulation errors. We also see that errors introduced by OpenVZ are generally larger than errors introduce by the virtual time system, especially for TCP-based applications. The importance of these observations is that we can tune the virtual time errors by changing the size of the emulation timeslice, and so, as a function of application, choose timeslices that do not significantly impact application performance.

2.6.2 Experiment Setup

Figure 2.11 illustrates our experiment framework, which consists of three components: an end-host running a server side application, an end-host running a client side application and an intermediate host that serves as a traffic controller. The traffic controller is a Linux application for configuring test scenarios with various network conditions including bandwidth, packet drop rate and packet delay. We duplicate the same network topology onto three platforms. The platform in Figure 2.11(a) consists only the physical hosts, which serves as the ground truth data collector. The second platform, as shown in Figure 2.11(b), has end-host applications running inside the OpenVZ virtual machine (VE) instead of the real operation system; comparison of behaviors on this with those of the applications on the first platform reveals the difference introduced by the OpenVZ techniques. The same topology is also created in our virtual-time system enabled testbed, as shown in Figure 2.11(c). The setup is composed of three virtual machines running on a single physical machine. Comparison of behaviors on this with behaviors on the pure OpenVZ topology reveals the errors our virtual time techniques introduce.

We use the identical hardware across all three platforms. Each physical machine is equipped with a 2.0 GHz dual-core processor, 2 GB memory and gigabit Ethernet network interface cards. Also, we create the same software environment for all platforms, including the same OS (Red Hat Enterprise Linux 5 with 2.6.18 kernel), the same version of libraries and drivers, the same testing applications and the same setting of network parameter (e.g. sending/receiving buffer, ip routing table). Finally, the traffic controller alters data packets in a deterministic manner, coordinated across architectures, using a random number generator to select packets to drop on the flows of interest. Therefore, when the i^{th} packet is dropped in any one of the configurations, the i^{th} packet is dropped in all of them. In this way, we ensure that on an experiment-by-experiment basis, we are comparing precisely the same

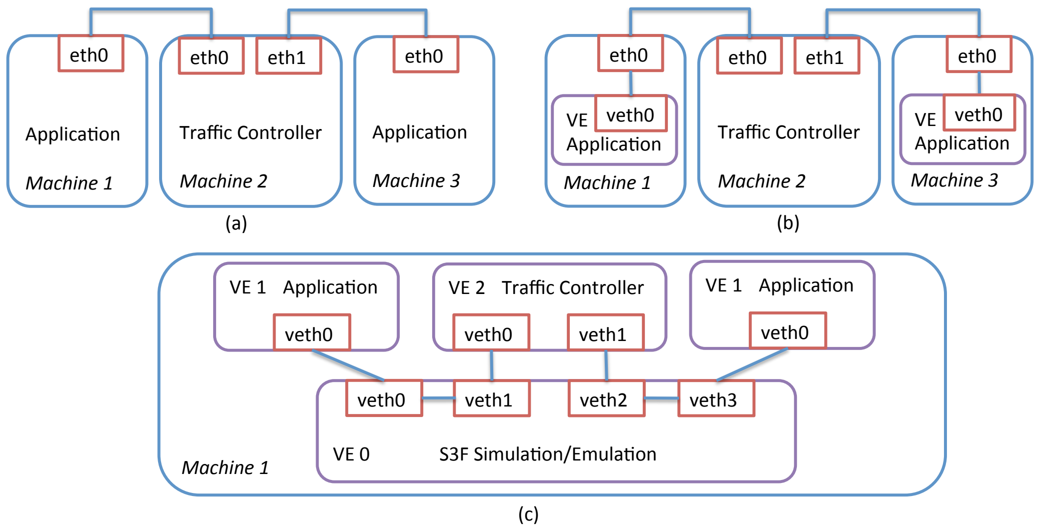


Figure 2.11: Testbeds Setup (a) Native Linux (b) Native OpenVZ (c) OpenVZ with Virtual Time

context for measuring the application-level network metrics.

2.6.3 Experiment Data and Analysis

Network-Intensive Applications

The first set of applications we study are network-intensive applications (ICMP, UDP, TCP). The experiments, run on each testbed platform, vary bandwidth, delay and loss. The data shown is based on a 100 μs timeslice. The platform index number 1, 2 and 3 used in every table in this section represents the native Linux, native OpenVZ and OpenVZ with virtual time system respectively as shown in Figure 2.11.

ICMP We use the ICMP protocol by pinging from one end-host to the other end-host under different network conditions controlled by the intermediate node. Ping is the commonly used utility application for testing the reachability of a host on an IP-based network and for measuring the round-trip time (RTT) for messages (ICMP echo request and response packets) sent from the originating host to a destination host and record any packet loss. The measured RTTs are listed in the Table 2.1.

Comparison of Testbed 1 (native Linux) and Testbed 2 (OpenVZ) shows the processing delay in bridging the veth and eth interface. We see the total processing overhead is approximately 0.1 ms, a cost due entirely to using OpenVZ, independent of virtual time overheads. A round-trip in this topology contains four hops (from Machine 1 to Machine 2 and back to Machine 1) and thus the worst case error is 400 μs. Indeed, the largest observed error is about 200 μs, and this matches the error bound of our system.

UDP We set up an iperf [39] UDP server and client pair at the two end-hosts. The iperf client sends constant bit rate (CBR) UDP traffic under various network conditions, and the packet loss rate, throughput and jitter are recorded in Table 2.2 for comparison.

The client sends data at a CBR equal to the link bandwidth. However, the bandwidth specified in iperf is the end-to-end (application layer) bandwidth. When the data is transmitted over the network and is added the network headers, the raw network data rate slightly exceeds the available bandwidth. This is the reason we observe packet losses even in the cases that link are not lossy, and those losses are caused by buffer overflow at the traffic controller.

We observe nearly identical results across all three platforms, especially the throughput has smaller than 1% error. Unlike TCP, there is no feedback loop in UDP, and the temporal error of a single packet does not propagate and cascade.

TCP - Iperf We set up the iperf TCP server and client and use the traffic controller to adjust the length of delay, loss rate and the available bandwidth to create various network testing scenarios. All the TCP related parameters, such as size of sending buffer and receiving buffer, are set to be the same (128 KB) in native Linux and OpenVZ. We keep sending traffic for 30 seconds for all the experiments and record the throughput, which is the primary indication of TCP connection performance, in Table 2.3.

Throughputs from platforms 2 and 3 are very close, under all cases, suggesting that the small errors in virtual time do not impact throughput evaluation. However, in our first trial of these experiments we saw a large difference between platforms 1 and 2, with (surprisingly) platform 2 yielding a significantly larger throughout! Investigation revealed that OpenVZ configuration allows for control of certain TCP buffer sizes, and that were set to be larger

Table 2.1: Ping

Network Condition		**Result**														
Loss	**Delay**	**Loss (%)**			**RTT min (ms)**			**RTT avg (ms)**			**RTT max (ms)**			**RTT mdev**		
(%)	**(ms)**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**
0	1	0	0	0	2.19	2.32	2.58	2.24	2.36	2.67	2.27	2.40	2.69	0.03	0.02	0.04
0	10	0	0	0	20.1	20.2	20.5	20.2	20.3	20.6	20.2	20.3	20.6	0.04	0.05	0.05
0	100	0	0	0	200	200	200	200	200	200	200	200	200	0.00	0.00	0.00
20	1	30	30	30	2.19	2.33	2.68	2.24	2.36	2.69	2.27	2.41	2.69	0.04	0.02	0.01
20	10	30	30	30	20.2	20.3	20.5	20.2	20.3	20.6	20.2	20.3	20.6	0.00	0.00	0.05
20	100	30	30	30	200	200	200	200	200	200	200	200	200	0.00	0.00	0.00
50	1	80	80	80	2.22	2.30	2.68	2.23	2.33	2.68	2.24	2.34	2.69	0.01	0.02	0.01
50	10	80	80	80	20.1	20.3	20.5	20.2	20.3	20.6	20.2	20.3	20.6	0.06	0.00	0.06
50	100	80	80	80	200	200	200	200	200	200	200	200	200	0.00	0.00	0.00

Table 2.2: UDP - Iperf

Network Condition			**Result**								
Loss	**Delay**	**BW**	**Loss (%)**			**Throughput (Mb/s)**			**Jitter (ms)**		
(%)	**(ms)**	**(Mb/s)**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**
0	1	10	1.6	1.6	1.6	9.73	9.73	9.73	0.067	0.043	0.108
0	1	100	2.5	2.4	2.4	97.9	98.0	98.0	0.070	0.046	0.055
0	1	400	3.3	3.4	3.3	389	392	392	0.046	0.032	0.038
0	10	10	1.7	1.7	1.7	9.73	9.73	9.73	0.052	0.048	0.057
0	10	100	2.5	2.5	2.5	98.0	98.0	98.0	0.056	0.050	0.060
0	100	10	2.6	2.7	2.5	9.73	9.71	9.73	0.101	0.128	0.145
5	10	10	5.1	5.1	5.1	9.48	9.54	9.48	0.039	0.044	0.037
5	10	100	5.0	5.1	4.9	95.5	95.4	95.6	0.042	0.050	0.062
10	10	10	10	10	10	8.98	9.03	8.98	0.056	0.051	0.055

than native Linux uses. After we aligned all TCP configurations possible, we still see differences between native Linux and native OpenVZ. In particular, in all the loss-free scenarios, TCP traffic in the native Linux has better performance than the OpenVZ-based Linux.

To understand the root cause of this difference, we instrumented the code to print out the size of the TCP send window, as a function of segment number; Figure 2.12 plots the result when we induce network conditions delay = 1 second, loss = 0, and bandwidth = 1 Mb/s. Platforms 2 and 3 have essentially identical results, but a significant difference is seen between platforms 1 and 2, with the send window size in congestion avoidance mode bing 34 for native Linux and 29 for OpenVZ. We also see different growth in the window size during the slow start mode. The differences strongly suggest some fundamental difference between the TCP implementation or configuration in native Linux and native OpenVZ.

TCP - FTP FTP traffic is generally very tolerant of delay and loss. We setup an FTP server and an FTP client, programming the client to download a file. All the cases use a network bandwidth of 10 Mb/s. The throughput, transfer time, and connection establish time are recorded in Table 2.4.

In the first four loss-free cases, Testbed 2 and Testbed 3 behave similarly, but they are slightly different than Testbed 1 (up to 3% difference in throughput). The reason is the same as previous iperf TCP, as FTP uses TCP. The difference affects only the file transfer time but not the connection estab-

Table 2.3: TCP - Iperf

Network Condition			**Result**		
Loss	**Delay**	**BW**	**Throughput (Mb/s)**		
(%)	**(ms)**	**(Mb/s)**	**1**	**2**	**3**
0	1	10	9.63	9.59	9.59
0	1	100	94.1	94.5	95.7
0	1	400	131	129	133
0	10	100	17.9	15.8	15.8
0	100	10	1.79	1.60	1.61
0	1000	1	0.157	0.137	0.133
1	10	10	4.06	4.89	4.83
2	10	10	2.93	3.25	3.26
5	10	10	1.74	1.70	1.78

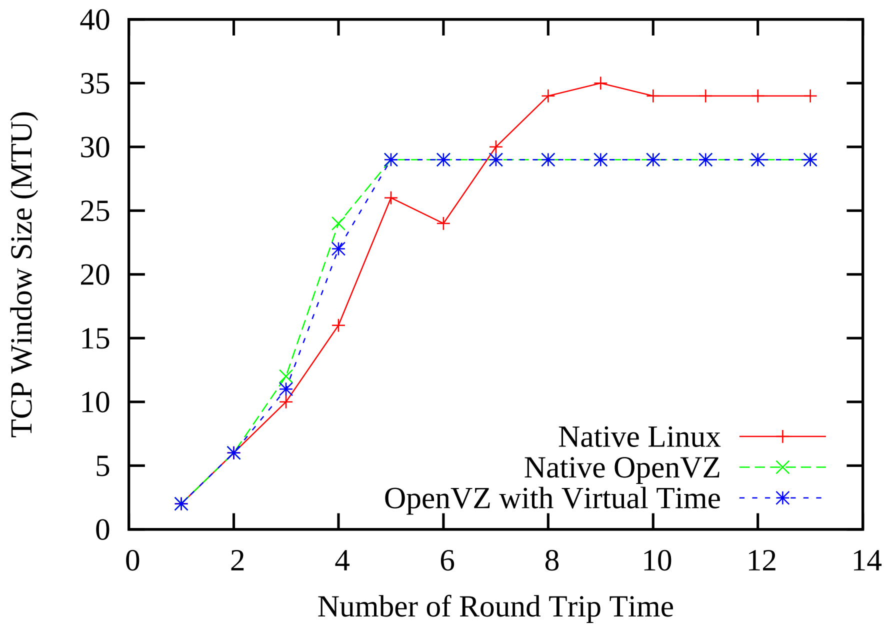


Figure 2.12: TCP Window Size

lish time, as the server and client only exchange control messages during the connection establish phase. These messages are delay sensitive but not throughput sensitive.

In the last three cases with losses, the throughput differences among three testbeds are enlarged. Again this is due to the difference in TCP implementation. Although our traffic controller outputs deterministic packet losses, they may behave differently on a same single packet loss, causing the different in achievable TCP throughput. During the connection establish phase, no packet losses are observed, thus all three testbeds have similar connection time.

TCP - HTTP Hypertext Transfer Protocol (HTTP) is the data communication protocol for the world wide web. Web browsing is generally tolerant of moderate delay and loss. We setup an apache server on one end-host [40] and a text-based web browser, named lynx [41], on the other end-host. We grabbed the openvz.org site with one level depth (105 files, 2848KB in total) and host those contents in our apache server. In this way, we can produce some typical web traffic which consists of a series of small and bursty file transfers. In the experiments, the client is configured to traverse all the first-level links and reports the total traversal time. The cache is cleared at the beginning of every run. The network bandwidth is set to 10 Mb/s for every experiment. We run each experiment for 10 times and the results are shown in Table 2.5.

We observe that traversing all web pages takes longer time in the OpenVZ-based Linux than in the native Linux. This is due to the processing delay in bridging the virtual network interface in OpenVZ (e.g., “veth0”) and the real Ethernet interface (e.g., “eth0”).

Also, testbed 3 has a smaller traversal time than testbed 2 in all the loss-free scenarios. The reason is that our embedding of OpenVZ in virtual time does not yet account for delays in file I/O, a deficiency in our implementation we will shortly be rectifying.

In addition, large randomness is observed for all the cases with packet loss. We carefully studied traces of different runs, and discovered this is due to multi-threaded web client/server applications. In particular, the web client application launches multiple TCP connections to request objects from the server and each connection is a thread. Since the multi-thread scheduling

Table 2.4: FTP

File	**Network Condition**		**Result**											
Size	**Loss**	**Delay**	**Throughput (KB/s)**			T_{Total} (s)			$T_{Transfer}$ (s)			$T_{Initial}$ (s)		
(MB)	**(%)**	**(ms)**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**	**1**	**2**	**3**
10	0	1	1140	1140	1140	8.9	8.9	8.9	8.8	8.8	8.8	0.1	0.1	0.1
10	0	10	1130	1140	1130	9.1	9.0	9.1	8.8	8.8	8.8	0.3	0.2	0.3
1	0	100	180	186	185	8.1	7.9	8.0	5.7	5.5	5.5	2.4	2.4	2.5
1	0	1000	20.4	18.3	18.9	74.3	80.1	78.1	50.2	56.0	54.2	24.1	24.1	23.9
1	1	100	79.5	96.1	108	15.3	13.2	11.9	12.9	10.7	9.5	2.4	2.5	2.4
1	2	100	41.6	42.4	41.4	27.0	26.5	27.2	24.6	24.2	24.7	2.4	2.3	2.5
1	5	100	29.6	29.1	29.3	36.9	37.7	37.3	34.6	35.2	34.9	2.3	2.5	2.4

in Linux is non-deterministic, the packet sending sequences can be different across multiple runs of each test case. Therefore, the traffic controller could drop different packets though itself is designed to produce packet loss pattern deterministically. The dropped HTTP packets are not uniformly important — control packet losses have larger impact on the overall timing than do data packet losses. Such randomness in the application behavior is not introduced by our system and should be expected in a multi-thread execution environment.

Table 2.5: HTTP

Network Condition		**Result**					
Loss	**Delay**	**Website Traversal Time (s)**			**Stddev (s)**		
(%)	**(ms)**	**1**	**2**	**3**	**1**	**2**	**3**
0	1	5.1	6.0	4.5	0.0	0.0	0.0
0	10	12.3	12.5	11.9	0.1	0.1	0.1
0	100	91.6	92.2	91.6	0.1	0.1	0.1
1	100	107.4	108.8	109.1	2.2	3.7	1.4
2	100	128.1	129.9	130.2	9.2	9.8	4.6
5	100	230.7	231.5	230.6	36.3	45.0	20.5

Larger emulation timeslice The network round-trip time is increased by the emulation timeslice. As much as one timeslice is added on the server side, as it may take that much additional time for it to recognize a packet, and as much as one additional timeslice may be added for the client to recognized the server's response. These delays obviously directly impact "ping", but they also impact TCP and applications that use it. TCP throughput is approximately RWS/RTT, where RWS is the receiver window size and RTT is the round-trip delay. So, for instance, in our experiments where the network latency is 1 ms, using an emulation timeslice of 1 ms causes RTT to increase from 2 ms to 4 ms, effectively cutting throughput in half. Experiments using larger emulation timeslices confirm this sensitivity. This gives us some guidance on choice of emulation timeslice in network intensive applications using TCP.

CPU-Intensive Applications

For CPU-intensive applications, we implemented the Client Puzzle protocol [42], which is used in many proof of work schemes for managing limited resources on a server and providing resilience to denial of service (DoS) attacks. In this protocol, when a client initiates a connection to a server, server will send client a puzzle to solve. The connection will be established only if the client correctly solve the puzzle. In particular, a puzzle is essentially a hash inversion problem, which currently has no efficient algorithm to solve but using brute force search. Table 2.6 documents the elapsed time for the client to set up a connection with the server. For consistency, the server is always using the same puzzle, but the client has no caches of previous puzzles. Each experiment uses a network bandwidth of 10 Mb/s.

We can see both Testbed 2 and Testbed 3 have very similar runtimes, yet Testbed 1 has slightly smaller ones. This is due to the overhead introduced by OpenVZ virtualization. Such overhead is small (around 3%), and it matches the advertised overhead of OpenVZ. The simulation/emulation overheads are excluded from the virtual clock of a container, and the container perceives time as if it were running independently. Moreover, we notice that Testbed 3 has a smaller standard deviation in runtime, indicating that its runtime is more stable and more repeatable. This is due to the virtual time system, which only counts the execution time performed by a VE into its virtual clock, excluding most other activities that may affect its runtime.

We conclude that our timeslice-based virtual time implementation yields high temporal fidelity not only for network packets but also to CPU computations. When the scheduler gives a timeslice to a VE, the actual amount of execution time received by the VE is usually slightly different from the timeslice length, due to some overhead and some interrupt-disabling routines in the Linux kernel. Our virtual time system uses an offset mechanism to compensate such difference, making the CPU computation time correct in long term [3]. Without such mechanism, application runtime is less accurate and less repeatable.

Table 2.6: Puzzle

Network Condition		**Result**					
Loss	**Delay**	**Average Time (s)**			**Stddev**		
(%)	**(ms)**	**1**	**2**	**3**	**1**	**2**	**3**
0	1	5.405	5.585	5.595	0.060	0.086	0.001
0	10	5.407	5.630	5.657	0.076	0.071	0.014
0	100	5.947	6.119	6.189	0.077	0.075	0.004
0	1000	11.383	11.593	11.585	0.091	0.059	0.004

2.7 Fast Background Traffic Simulation Model

While virtual-machine-based emulation allows us to run unmodified software in our testbed, the simulation primarily offers the accurate and fast network environment for large-scale experiments, such as providing an accurate and fast network environment (layer 3 and below), efficiently simulating and modeling the background traffic generated by the large number of virtual nodes. When conducting large-scale network simulation experiments, the cost of simulating the network can easily overwhelm the overall cost of performing the simulation experiment. Therefore, we develop several efficient Ethernet switch models based on the data collected on the real switches [5]; we also develop techniques for efficiently modeling background traffic through switches/routers that use First-Come-First-Serve and Fair Queueing scheduling in our testbed.

2.7.1 Overview

In many network applications, only a small fraction of traffic is of specific interest. For example, in models of the power grid we may be more interested in the detailed behavior of some specific flows (e.g. a DNP3 or Modbus connection between a control station and a particular substation), and be interested in other flows only in so far as they consume resources and affect the behavior of the detailed flows. In experiments where real devices are embedded into the simulator, it is challenging to meet the real-time constraint given the fact that the network being simulated has hundreds of thousands of devices and (typically) tens of thousands of flows represented at any given time. Therefore, there is a strong motivation to have efficient multi-resolution traffic models, in which background traffic is modeled with much less details than foreground traffic.

A high performance background traffic model is proposed and validated in [43]. The model aggregates the background flows between network access points, transforms the dependencies among the flows into a reduced system of non-linear equations, and efficiently solves the system using fixed-point iteration. One foundational assumption of the model is that the network switches and routers manage queueing using First-Come-First-Serve (FCFS) scheduling. However, in reality, such devices often have schedulers that provide rate proportional service [44], such as round robin (RR), weighted round robin (WRR) [45] and virtual clock [46]. Indeed, many commercial switches use round robin based scheduling due to the low time complexity O(1) and low implementation cost [47].

We investigate two Gigabit Ethernet switches, 3COM 3CGSU08 and NetGear GS108v2 based on the traces collected with a unique testbed. The testbed we created can generate and capture Gigabit Ethernet traffic at line rates, and measure precisely what the latency and loss patterns are through a switch, for sequences of millions of frames. The experimental results reveal the FCFS behavior in the NetGear switch and Fair Queuing (FQ) behavior in 3COM. Based on analysis of the real data, we developed an algorithm for the FQ scheduler, integrated it with the existing algorithm based on the FCFS scheduler. We then validated the model in terms of convergence behavior, simulation speed and accuracy, and compared the simulation results between the FCFS networks and the FQ networks.

2.7.2 Measurement

Testbed

A comprehensive study of a Gigabit Ethernet switch, such as the frame loss and delay patterns, output rate and drop rate of each flow, requires a testbed to do the following:

- Generate traffic up to line rate with user-configured parameters such as frame size, sending rate and inter-frame gap.

- Record frame delays and arrival orderings with microsecond resolution.
- Capture frames at line rate with little loss.

We built a testbed that uses hardware to instrument, transmit, and capture Ethernet frames at line rates. Figure 2.13(a) depicts our solution. Traffic is generated using a four-port NetFPGA card [48], [49]; the frames to send can be loaded from a pcap file with user-specified sending rate. A 4.5 G4 Endace DAG card [50] is the receiving end, which places time-stamps on received frames using a clock that has 10 ns resolution; the card can also capture and store millions of frames with zero loss at 1 Gb/s. In order to time the passage of a frame through a switch, we took advantage of the NetFPGA card's ability to simultaneously send identical flows from each port. Two identical flows are generated from the NetFPGA to the same destination, for example, in Figure 2.13(a) a frame is sent simultaneously out on ports N1 and N4 of the NetFPGA card. One instance arrives at port D4 of the DAG and is time-stamped on receipt. The other enters the switch via port S1, is routed out via port S5, and arrives at the DAG on port D1 where it is time-stamped. The difference in time stamps is the delay through the switch (and time on the wire from switch to DAG). To validate the approach, we replaced the switch with a wire and calculated the difference under a 1 Gb/s sending rate. The measured delay has mean 0 ns and standard deviation 0.004 ns, which is low enough given the microsecond delay in the switch. This design does not require clock synchronization between the NetFGPA card and the DAG card.

The data flows generated in the experiments were Constant Bit Rate (CBR) Ethernet raw frames in pcap format, and a sequence number was added into each frame. The frame size is fixed at 1500 bytes to minimize the inter-frame gap and so maximize the sending/receiving rate. We generated one million frames for every flow, in every experiment. Post analysis of received frames can identify the missing frames by analyzing the sequence numbers. Likewise the difference between timestamps on frames with identical sequence numbers (one which passed directly to the DAG, the other which passed first through the switch) gives that frame's delay. In addition, we can also calculate the output rate and the drop rate of each flow. Both cards are placed on the same PC (four dual-core 2.0 GHz CPUs running CentOS 5.2). We captured data from two 8-port Gigabit Ethernet switches: 3COM 3CGSU08 and NetGear GS108v2. They are all simple low-end commodity switches with no configuration interface available, and every port is identical. All ports were connected by cat-6 Ethernet cables.

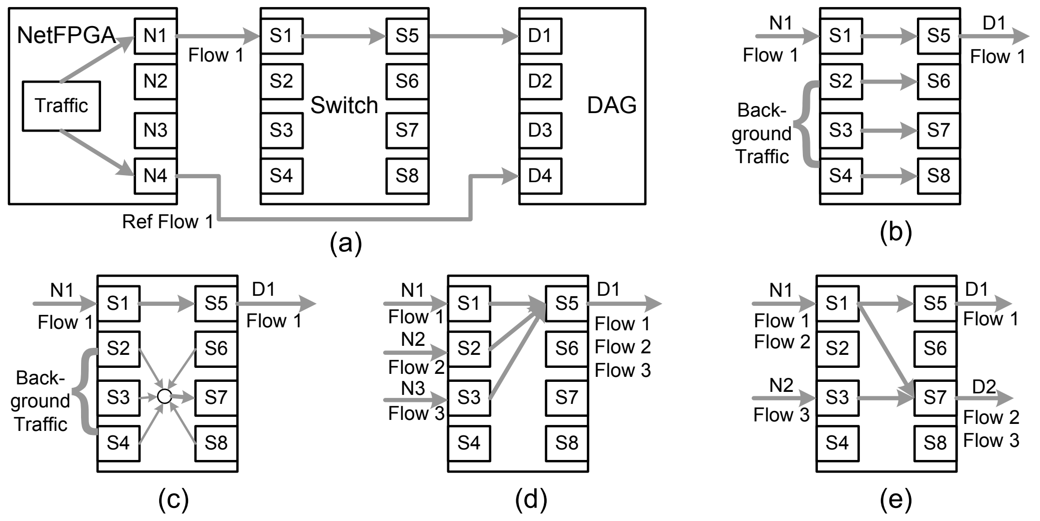


Figure 2.13: Testbed Overview and Experiment Setup

Experiment and Data Analysis

The first set of experiments (see Figure 2.13(a)) monitored a single flow whose sending rate varies from 100 Mb/s to 1 Gb/s with 100 Mb/s increment, with no background traffic. We observed that the delay in both switches behaves constantly with no loss, about 13 μs for the NetGear switch and 18 μs for the 3COM switch. The delay added by each switch has the same mean value under any sending rate up to 1 Gb/s and the variance is close to 0 μs. This shows that the switches handle line rate without loss, the switches have significantly different delays, and that with deterministic inter-arrival times those delays are constant.

In the second set of experiments, we kept the single flow, and added various combination of background traffic flows going through other ports, such as three parallel 1 Gb/s CBR flows (see Figure 2.13(b)) and five 1 Gb/s CBR flows from five input ports to one output port (see Figure 2.13(c)), *other* *than* the one targeted by the monitored flow. These experiments revealed the same behavior of delay and loss on the monitored flow as we saw in the first experiments when there were no background flows. From these experiments we learned that (1) both switches can handle nearly 1 Gb/s flows at each output—no frame losses were observed until the total rate of the flows to the same output port reached 988 Mb/s, and (2) flows coming from different source ports and going to different destination ports do not affect each other.

The third set of experiments monitored the output rate of three flows from three input ports to the same output port, as shown in Figure 2.13(d). Flow 1 moves at 100 Mb/s, flow 2 moves at 500 Mb/s, and flow 3 varies from 100 Mb/s to 1 Gb/s with a 100 Mb/s increment. Table 2.7 shows the results from the NetGear switch and Table 2.8 shows the results from the 3COM switch. When total input rate of the three injected flows does not exceed the switch's service capacity (which is 1 Gb/s in theory and 988 Mb/s as observed), both switches experienced zero drop rate on every flow. When the total input rate exceeds the maximum service rate, the NetGear switch dropped frames in all the three flows and the drop rate is almost same. However, for the 3COM switch, we observed that (1) all frames in flow 1 (100 Mb/s) were received, and (2) frames in flow 2 and 3 ($\geq$ 500 Mb/s) equally share the remaining bandwidth. The implication is that all flows have the same bandwidth reservation ($\approx$ 333 Mb/s) and the switch served the connections in proportion to their reservation, and then distributes the extra bandwidth equally among the active flows.

To investigate switch behavior, we collected the individual flow's detailed delay and loss patterns. By utilizing all four ports of the NetFPGA card, the testbed can monitor delay and loss patterns of two flows. We removed flow 3 in Figure 2.13(d) and ran the fourth set of experiments to monitor flow 1 and flow 2. The sending rates on both flows were varied from 100 Mb/s to 1 Gb/s with a 100 Mb/s increment. Figure 2.14(a) and (b) show one sample pattern of two input flows with sending rates 900 Mb/s and 300 Mb/s—Figure 2.14(a) describes the 3COM switch and Figure 2.14(b) describes the NetGear switch. Frames are ordered according their arrival timestamp. The delay is plotted on the y-axis and a value zero represents a loss. The NetGear switch dropped frames from both 900 Mb/s and 300 Mb/s flows, in bursts, while the 3COM switch dropped frames only from the 900 Mb/s flow, not in bursts. Indeed, as we increased the rate of the slower flow, the 3COM switch

Table 2.7: NetGear, Results of the Three Input Flows to One Output Port Experiment

Input Rate (Mb/s)			Output Rate (Mb/s)			Drop Rate		
Flow 1	Flow 2	Flow 3	Flow 1	Flow 2	Flow 3	Flow 1	Flow 2	Flow 3
100	499	100	100	499	100	0	0	0
100	499	200	100	499	200	0	0	0
100	499	299	100	499	299	0	0	0
100	499	399	98	492	394	0.02	0.01	0.01
100	499	499	90	447	447	0.10	0.10	0.10
100	499	599	83	414	487	0.17	0.17	0.19
100	499	699	77	383	525	0.23	0.23	0.25
100	499	799	72	359	554	0.28	0.28	0.31
100	499	899	68	339	578	0.32	0.32	0.36
100	499	988	65	316	604	0.35	0.37	0.39

Table 2.8: 3COM, Results of the Three Input Flows to One Output Port Experiment

Input Rate (Mb/s)			Output Rate (Mb/s)			Drop Rate		
Flow 1	Flow 2	Flow 3	Flow 1	Flow 2	Flow 3	Flow 1	Flow 2	Flow 3
100	499	100	100	499	100	0	0	0
100	499	200	100	499	200	0	0	0
100	499	299	100	499	299	0	0	0
100	499	399	100	486	399	0	0.03	0
100	499	499	100	443	442	0	0.11	0.11
100	499	599	100	443	442	0	0.11	0.26
100	499	699	100	443	442	0	0.11	0.37
100	499	799	100	443	442	0	0.11	0.45
100	499	899	100	443	442	0	0.11	0.51
100	499	988	100	442	443	0	0.11	0.55

did not drop any frames until it reached 500 Mb/s. The delays of both flows increase together and stabilize at same level for the NetGear switch, while the delay of the two flows stabilized at different values in the 3COM switch. The distinctive behavior of the two switches can be explained by different scheduling policies as studied in [5]. The scheduler in the NetGear switch and the 3COM switch can be modeled respectively by a FCFS server and a weighed round robin server, which belongs to the class of rate proportional server [44].

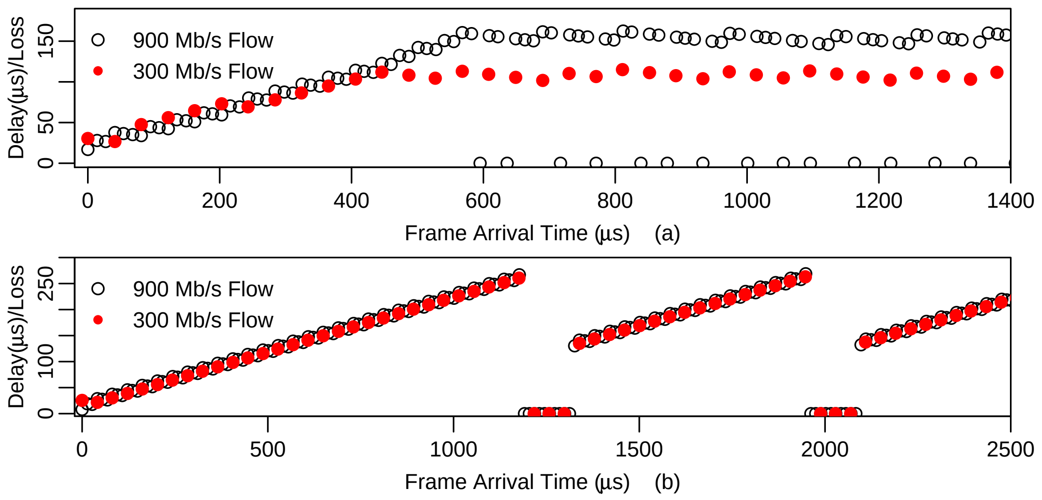


Figure 2.14: Delay/Loss Pattern, Two Input Flows to One Output Port, (a) 3COM (b) NetGear

In the fifth set of experiments, we injected three flows into the switch, where flow 1 and flow 2 shared the same input port (S1) and flow 2 and flow 3 shared the same output port (S7), as shown in Figure 2.13(e). The input rate of flow 3 is fixed to the line rate (988 Mb/s), which ensures the sum of flow 2 and flow 3 would always be greater than the maximum service rate and would result frame losses at outport S7. We want to observe flow 2's impact on flow 1 when flow 2 itself is congested. Table 2.9 shows the output rates for both switches. The results show that the output rate of flow 1 is always equal to its input rate for both switches. Further experiments actually reveal the same behavior of delay and loss on flow 1 as we saw in the first experiment. Therefore, flow 2 does not affect flow 1 at all. This can be explained by the virtual output queue, in which frames are classified according to their destination MAC addresses upon arrival at the input port. The frames with different destination MAC will then be processed independently. In addition, flow 2 and flow 3 share the bandwidth identically as in the third and fourth experiments. We learn then that flows going to different destination ports do not affect each other. It is reasonable to simplify a switch model by modeling its output ports individually and independently. Therefore, our proposed background traffic model uses the output port as a basic element. Details will be discussed in next section.

Table 2.9: Experimental Results: Three Flows with Two Sharing One Input Port and Two Sharing One Output Port

Input Rate (Mb/s)			NetGear Output Rate (Mb/s)			3COM Output Rate (Mb/s)		
Flow 1	Flow 2	Flow 3	Flow 1	Flow 2	Flow 3	Flow 1	Flow 2	Flow 3
99	889	988	99	466	521	99	493	495
198	790	988	197	435	552	197	491	497
329	659	988	329	404	583	329	492	496
494	494	988	493	338	649	493	492	496
659	329	988	658	258	729	658	329	658
790	198	988	790	167	820	790	197	790
889	99	988	888	90	897	888	99	888

We modified the input pcap file of flow 1 by redirecting the source IP address on half of the frame to a different source IP address, and used the modified traffic input to conduct all the five sets of experiments again. The results were exactly same, which means the two switches use only source and destination MAC address to differentiate flows, nothing above the layer 2 header is used by the switches. Therefore, it is reasonable to aggregate flows on a per input port basis to output port pair.

From all the experiments, we learned the following important points which served as guidelines for our background traffic modeling.

- Different switches have different scheduling policy, FCFS (as revealed in the NetGear switch) and FQ (as revealed in the 3COM switch) are two common types.

- Flows going to different output ports do not influence each other, and thus a switch model can be divided into several independent output port models.

- Flows with the same input and output port can be aggregated.

- If the sum of input rates is no greater than the maximum service rate, i.e. the line rate, each flow's output rate is equal to its input rate with zero loss.

2.7.3 Coarse-Grained Background Traffic Modeling and Simulation

Problem Formulation

The network being studied consists of n access points; the backbone is built using Gigabit Ethernet switches. We are interested in how the network shapes coarse-grained background traffic flows along the communication path from an ingress node to an egress node. In our model, the simulation time is discretized into units of length Δ, and we use t_k to denote $k \cdot \Delta (k = 0, 1, 2...)$. For any ingress-egress pair $< P_i, P_j >$ $(1 \leq i, j \leq n)$, we use $T_{i,j}(t)$ to denote the ingress rate of the corresponding aggregate flow at time t. Since the simulation time is advanced by constant time-step Δ, we need to discretize the ingress rate for every aggregated flow. During the time-step $[t_k, t_{k+1}]$, the traffic volume injected at ingress node i to egress node j is $\int_{t_k}^{t_{k+1}} T_{i,j}(t)\ dt$. The burstiness at time scales smaller than Δ is smoothed. To ensure that the same amount of traffic is generated, the ingress rate at the discretized time $k \cdot \Delta$ is $\frac{1}{\Delta} \times \int_{t_k}^{t_{k+1}} T_{i,j}(t)\ dt$.

All the nodes in our model are connected with unidirectional links, understanding that we can model a bi-directional link as a pair of these. The sending endpoint of a link is attached to a switch's output port. When multiple aggregate flows multiplex at the output port at time t_k, the bandwidth allocation to each flow is governed by the scheduling policy, which varies from switch to switch. A switch is modeled as a group of output ports according to the experimental results in Section 3.3.3.

We assume that the resource consumption of certain types of flows as well as the forwarding table of a switch change relatively infrequently. We can define a reasonable time-step, say the maximum TCP round-trip time among all flows. This allows the traffic demand generation logic to be responsive to network conditions at a TCP time-scale (i.e., react to loss on a flow). The resource consumption and the data forwarding paths are determined/re-evaluated only at the beginning of the time-step, and treat the consumption and paths as invariant for the rest of the time-step. These flows can be made sensitive to those flows modeled with higher resolution, but only at the fixed update points.

The model must capture the competition between foreground and background flows for link bandwidth. At the beginning of each time-step, for each port, we compute an estimated foreground input rate based on recent observations (or even known predictions, if such were available). As we compute new flow rates, for that port, we treat the estimated foreground flow exactly as any other flow. The fair contribution of foreground traffic is thus considered as we allocate bandwidth for the background traffic.

Model and Algorithm Description

The goal of our algorithm is this: given a description of flow intensities at ingress nodes, efficiently determine link loads throughout the intermediate switches, and determine flow intensities at the egress nodes. Each flow is associated with a flow rate and a state variable, which can be settled, bounded or unsettled. A settled flow has a finalized flow rate; a bounded flow has a known upper bound on its flow rate; a unsettled flow is neither bounded or settled. A switch is modeled as a few independent output ports. Each port is in one of the three states: resolved, transparent and unresolved. A port is resolved if all its input flows are settled. A port is transparent if not all of its input flows are known, but the sum of the input flow rate is less than the maximum service rate. A port is unresolved if it is neither resolved or transparent. The experimental results in Section 3.3.3 show that for switches using either FCFS scheduling or FQ scheduling, all the output flow rates can be determined at a resolved port; at a transparent port, a flow's output rate is identical to its input flow rate.

The algorithm proposed in [43] works well on networks whose switches use FCFS scheduling. It transitions through four phases, each time-step: flow update computation, reduced graph generation, fixed-point iterations and residual flow update computation. Phase I propagates the flow rate and state from ingress points throughout the network; the goal is to settle flows and resolve as many ports as possible. We are necessarily left with circular dependencies among some flow variables, the remaining phases address these.

Phase II identifies all the flow variables whose values are circularly dependent, and constructs a dependency graph whose vertex set is composed of all output ports with unresolved state and nonzero out degree. The unknown flow rates are the solution of a nonlinear set of equations. Phase III sets up and solves these equations using fixed point iteration. Finally, phase IV substitutes the solutions into the system and finishes the computation.

To modify this algorithm for FQ switching we must reformulation some of the phase I rules to be applied at FQ switches. Other phases are the same; details and validation can be found in [43]. The parameters used by the phase I algorithm are explained as follows:

S_p	State of port $p \in \{Resolved, Transparent, Unresolved\}$
n_p	Total number of input flows of port p
U_p	Bandwidth of port p, i.e. the maximum service rate of port p
F_p	Set of input flows passing through port p
$S_{i,p}^{in}$	State of input flow i at port $p \in \{Settled, Bounded, Unsettled\}$
$S_{i,p}^{out}$	State of output flow i at port $p \in \{Settled, Bounded, Unsettled\}$
$\lambda_{i,p}^{in}$	Input rate of flow i passing through port p
$\lambda_{i,p}^{out}$	Output rate of flow i passing through port p
$\Lambda_{in,p}$	Sum of all input flow rates into port p
$\Lambda_{in,p}^{settled}$	Sum of all settled input flow rate of port p
$R_{i,p}$	Reserved bandwidth for input flow i of port p, for FQ
E_p	Total extra bandwidth available (bandwidth reserved but not actually used) for port p, for FQ
m_p	Number of flows not yet processed at port p, for FQ

In the initialization stage of each time-step, all the ingress nodes generate traffic. Thus, the ingress nodes are in the resolved state, and all the corresponding output flows are settled. All the switches and egress nodes are in the unsettled state and all their connected flows are in the unsettled state. The changes at the ingress nodes then propagate through the entire networks through a three-step algorithm and loops until no more ports and flows change their rates and states.

Step 1 is to decide the state of a port p based on the input flows.

$$S_p = \begin{cases} Resolved & if \quad \forall i \in F_p, S_{i,p}^{in} = Settled \\ \\ Transparent & if \quad \Lambda_{in,p} \leq U_p \\ Unresolved & else \end{cases}$$

Step 2 calculates the rate and state of all the output flows of port p. Rules are selected based on the port state and the scheduling policy of the switch. They are summarized in Table 2.10. For a resolved port, all the input flow rates are known, therefore all the output flow rates can be computed and all the out flow states can be determined. For a transparent port, the rate and state of all the output flows is same as the corresponding input flows. For a unresolved port, the goal is to determine the upper bound of its output flows. If the corresponding input flow is settled, the upper bound is derived by ignoring all bounded input flows. If the corresponding input flow is bounded, the upper bound is derived by ignoring all other bounded input flows and treat itself as settled with rate set to the bounded rate.

Step 3 is to pass the rate and state of the output flow to the next switch's input along the flow's path.

$$\lambda_{i,p+1}^{in} = \lambda_{i,p}^{out}, S_{i,p+1}^{in} = S_{i,p}^{out}$$

2.7.4 Evaluation

We now turn to an experimental evaluation of our algorithms. The topology built for the experiment has 294 hosts, 882 directional links and 11,760 flows. The link bandwidth is 1 Gb/s. The switches used in the network all use the FCFS schedulers or the FQ schedulers for comparison. We use a Possion-Pareto-Burst-Process (PPBP) [51] traffic model to generate the ingress rates for each flow. A Poisson process generates arrivals, each of which contributes a random traffic volume that is sampled from a Pareto distribution. The PPBP is recognized as a good model of traffic arrival to

Table 2.10: Rules of Phase I, Flow Update Computation

Port State	Scheduling Policy	Algorithm
Resolved	FCFS	$\forall i \in F_p, S_{i,p}^{out} = Settled, \lambda_{i,p}^{out} = \lambda_{i,p}^{in} \times min(1, \frac{U_p}{\Lambda_{in,p}})$
	FQ	Initially, sort F_p according to $\lambda_{i,p}^{in}$, $E_p = 0, m_p = n_p$; Every input flow processing round, $\lambda_{i,p}^{out} = \begin{cases} \lambda_{i,p}^{in} & if \quad \lambda_{i,p}^{in} \le R_{i,p} \\ min(\lambda_{i,p}^{in}, R_{i,p} + \frac{E_p}{m_p}) & if \quad \lambda_{i,p}^{in} > R_{i,p} \end{cases}$ $m_p = m_p - 1, E_p = E_p + R_{i,p} - \lambda_{i,p}^{out}$
Transparent	FCFS	$\forall i \in F_p, \lambda_{i,p}^{out} = \lambda_{i,p}^{in}$ and $S_{i,p}^{out} = S_{i,p}^{in}$
	FQ	
Unresolved	FCFS	$\lambda_{i,p}^{out} = \begin{cases} \lambda_{i,p}^{in} \times min(1, \frac{U_p}{\Lambda_{in,p}^{settled}}) & if \quad S_{i,p}^{in} = Settled \\ \lambda_{i,p}^{in} \times min(1, \frac{U_p}{\Lambda_{in,p}^{settled}+1}) & if \quad S_{i,p}^{in} = Bounded \end{cases}$ $S_{i,p}^{out} = Bounded$
	FQ	$Initially, \begin{array}{l} \forall j \in F_p \wedge S_{j,p}^{in} = Bounded, \lambda_{j,p}^{in} = 0 \\ if S_{i,p}^{in} = Settled \\ \forall j \in F_p \wedge S_{j,p}^{in} = Bounded \wedge j \neq i, \lambda_{j,p}^{in} = 0 \\ if S_{i,p}^{in} = Bounded \end{array}$ Compute $\lambda_{i,p}^{out}$ using the same algorithm for FQ resolved port case $S_{i,p}^{out} = \begin{cases} S_{i,p}^{in} & if \quad \lambda_{i,p}^{in} \le R_{i,p} \\ Bounded & else \end{cases}$

a backbone. We adjust its input parameter to achieve desired average link utilization. The Hurst parameter for the Pareto distribution is set to 0.8 for all the experiments.

Our evaluation is focused on execution speed and accuracy, both relative to a pure packet simulation. The experimental design sets a target average link utilization and chooses PPBP parameters that produce such loadings. In all experiments a time-step is 1 second of simulation time, and each experiment advances simulation time 1000 seconds. Each time-step we measure how many ports end up in dependency cycles, and how many fixed point iterations are required to solve the dependencies. For each target link utilization we run 10 experiments. Our performance metrics are based on observations from all time-steps.

Convergence Behavior

We consider a solution at a time-step to be converged when the maximum relative difference between successive approximations to any flow rate is 0.001, for example, when $|\lambda_{n+1} - \lambda_n|/\lambda_n \leq 0.001$ for all flow rates λ_n. In addition, the desired average link utilization is 90% in all the convergence study experiments.

Figure 2.15 describes the distribution the number of ports on the dependence graph generated in phase II (over all time-steps and all runs), and the distribution of the number of iterations used to reach fixed-point solutions in phase III, for the FCFS network and the FQ network respectively. This data shows that even under high network load (90% link utilization) our algorithm significantly reduces the number of ports involved in the fixed point iteration, and finds the solution in a small number of iterations.

The data shows that the FQ structure tends to induce fewer ports in dependency cycles (7.6% for the FCFS network and 2.2% for the FQ network), and (consequently) fewer iterations of the fixed point algorithm for solution. We understand the difference as being due to the fact that the reserved bandwidth for an individual flow in the FQ switch can produce more settled flows. For FCFS switches under congestion all the output flows change if one input flow changes rate, whereas for FQ switches, an input flow within its reserved bandwidth will not change its output rate.

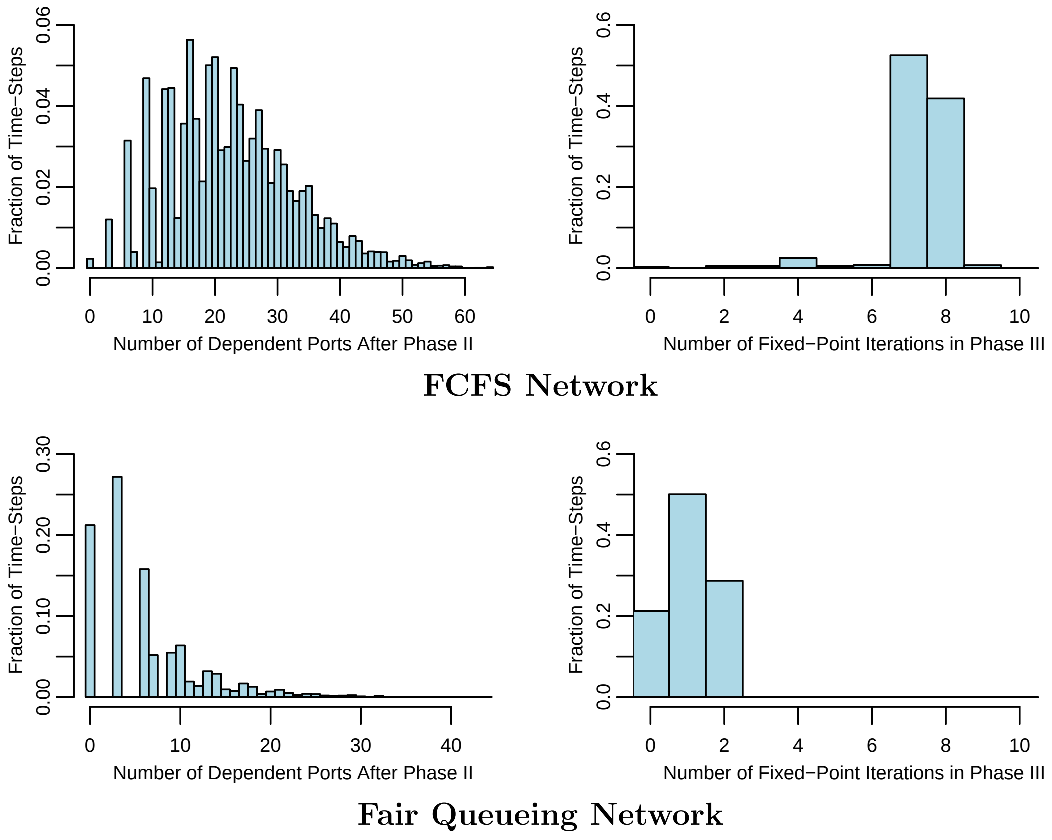


Figure 2.15: Convergence Experiment Results, 90% Link Utilization

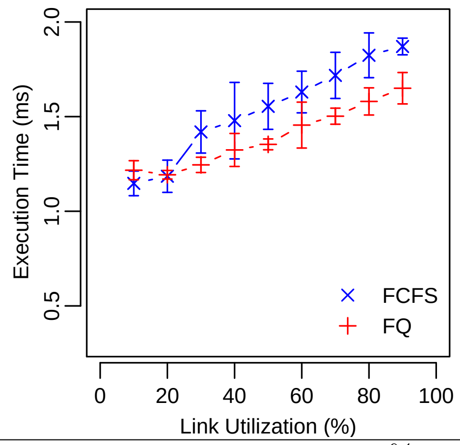
Simulation Speed

The algorithm must execute quickly if it is to be useful for large scale network simulation. The authors in [43] examine the speedup of the algorithm for FCFS networks relative pure packet simulation for background flows and observe speedups exceeding 3000 when executing on an ordinary PC. We infer that similar speedups will be achieved using our algorithm on FQ networks by comparing execution speed with FQ switches with execution speed on FCFS switches, under exactly the same topology and exactly the same traffic loads (varied from 10% to 90%).

These experiments use the same topology as was used in the convergence experiments, and likewise the time-step is 1 simulation second. As before, for each load setting we executed 10 independent runs, each of 1000 simulation seconds. The average execution time per time-step of both switches as well as the corresponding standard derivations are presented in the upper portion of Figure 2.16 and the average time spent on each phase is also listed under link utilization 10% and 90% for comparison.

Each time-step advances the simulation clock by 1 second, while the total time required under both traffic loads, and both scheduling policies is less than 2 ms. Clearly on this sized model the background traffic calculation in no way threatens the ability to run the simulation faster than real time.

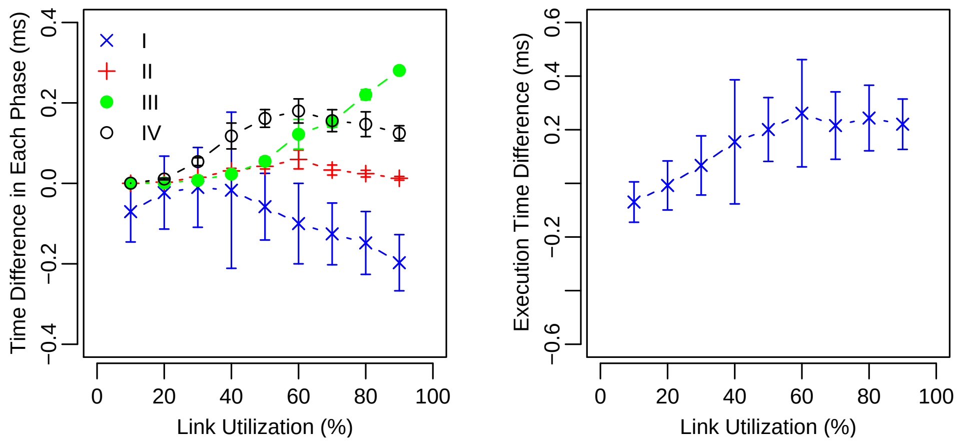
The lower portion of Figure 2.16 describes the difference in execution time between the topology with FCFS switches, and the topology with FQ switches. For each phase we measure the time required each run. We drive the FCFS model and the FQ model with exactly the same samples of traffic demand using synchronized random number streams, so each data point in the estimate is of coupled sampled paths. Here we see the higher overhead in FQ of phase I due to sorting input flows by rate, and related management. However, for all but the lightest load, the extra computational cost is more than accounted for by accelerated performance in phases 3 and 4, due to fewer unresolved ports, and flow values settling faster because of bandwidth reservations. Under very light load there are very few congested ports and so the additional overhead in FQ switches in phase 1 is not recovered from by improved performance later.



Link Utilization = 10%					
Scheduler	Phase I	Phase II	Phase III	Phase IV	Total
FCFS	1.146	0.00021	0.00014	0.001	1.147
FQ	1.217	0	0	0	1.217

Link Utilization = 90%					
Scheduler	Phase I	Phase II	Phase III	Phase IV	Total
FCFS	1.028	0.086	0.328	0.429	1.871
FQ	1.225	0.074	0.047	0.304	1.650

Execution Time per Time-Step (ms)



Impact of Scheduling Policy on Execution Time

Figure 2.16: Analysis of Execution Time

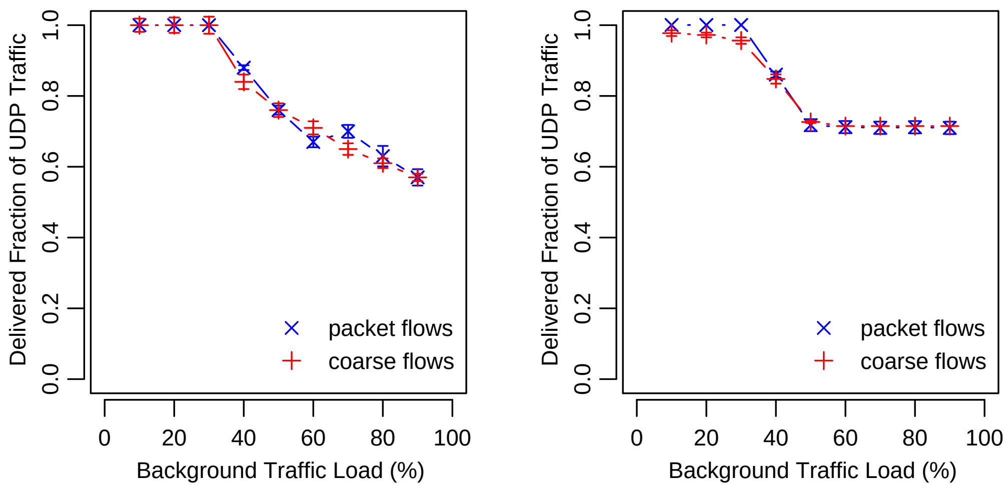
Accuracy

We also consider the accuracy of our approach when used as a background traffic generator, as compared to the pure packet-oriented approach. The idea is to study the properties of a small number of detailed packet-oriented UDP flows, while the background traffic is either packet based or flow based. Toward this end, we modify the topology used in the convergence simulation experiments by attaching a UDP server to each ingress node and a UDP client to each egress node. The background traffic is simulated with two different techniques: our time-stepped coarse-grained simulation and conventional packet-level simulation. We parametrically control the traffic parameters to cause the average background traffic load on a link to vary between experiments, from 10% to 90%. The PPBP traffic model is used to generate ingress traffic for each flow.

For every background traffic load, we simulate 10 experiments. In each experiment, a UDP client randomly picks a UDP server on any other end host. A UDP server's behavior can be modeled as an ON/OFF process: it uses UDP protocol to send a file of 70 Mbytes at the constant rate of 700 Mb/s, and after it finishes the transfer, it remains idle for an exponentially distributed period with a mean of 1 seconds; after the off period finishes, it sends another 70 Mbytes at the same rate, and the above process repeats until the simulation is over. Every experiment is simulated for 1000 seconds (in simulation time).

We compare the behavior of the UDP flows using different methods for background traffic generation on both the FCFS network and FQ network respectively. The experiments are coupled, in the sense that the same pattern of requests is simulated for the UDP flows in the two experiments that use different background flow generation, and furthermore, the background flow generation in the two experiments is driven by the same offered load. In this way, we ensure that on an experiment-by-experiment basis, we are comparing precisely the same context for measuring foreground UDP flows against different background generation techniques.

Figure 2.17(a) and 2.17(b) plot the sample mean and 95% confidence intervals of the delivered fraction of UDP traffic measured in 10 experiments. We see that the fraction of the delivered UDP traffic decreases with the link utilization in both cases due to congestion. The most important feature of the plots is that the mean of the delivered foreground traffic is close between two background flow generation techniques for both FCFS and FQ networks. The FQ network performs better than the FCFS network when the background traffic load is greater than 50%. The explanation here is that we have to approximate packet loss at a congested switches. The more heavily congested the network, the more heavily the approximation is exercised. That approximation is the cause of the error in the FCFS network; in the FQ network, when all the input flows, including the foreground flows, to a output port is higher than its reserved bandwidth, every flow simply gets its own reserved bandwidth, and therefore the packet loss approximation is quite accurate under high traffic load.



(a) FCFS Network (b) Fair Queuing Network

Figure 2.17: Delivered Fraction of Foreground UDP Traffic

2.8 Parallel Simulation of Software-Defined Networks

Existing network architectures fall short when handling networking trends, e.g., mobility, server virtualization, and cloud computing, as well as market requirements with rapid changes. Software-defined networking (SDN) is designed to transform network architectures by decoupling the control plane from the data plane. Intelligence is shifted to the logically centralized controller with direct programmability, and the underlying infrastructures are abstracted from applications. The wide adoption of SDN in network industries has motivated development of large-scale, high-fidelity testbeds for evaluation of systems that incorporate SDN. We leverage our prior work on a hybrid network testbed with a parallel network simulator and a virtual-machine-based emulation system. In this work, we extend the testbed to support OpenFlow-based SDN simulation and emulation; show how to exploit typical SDN controller behavior to deal with potential performance issues caused by the centralized controller in parallel discrete-event simulation; and investigate methods for improving the model scalability, including an asynchronous synchronization algorithm for passive controllers and a two-level architecture for active controllers. The techniques not only improve the simulation performance, but also are valuable for designing scalable SDN controllers.

2.8.1 Overview

Complex networks with large numbers of vendor-dependent devices and inconsistent policies greatly limit network designers' options and ability to innovate. For example, to deploy a network-wide policy in a cloud platform, the network operators must (re)configure thousands of physical and virtual machines, including access control lists, VLANs, and other protocol-based mechanisms. Trends in networking, such as cloud services, mobile computing, and server virtualization, also impose requirements that are extremely difficult to meet in traditional network architectures. Software-defined networking (SDN) is designed to transform network architectures. In an SDN, the control plane is decoupled from the data plane in the network devices, such as switches, routers, access points, and gateways, and the underlying infrastructure is abstracted from the applications that use it. Therefore, SDN enables centralized control of a network with flexibility and direct programability. SDNs have been widely accepted and deployed in enterprise networks, campus networks, carriers, and data centers. For example, Google has deployed SDN in its largest production network: its data-center to data-center WAN [52].

OpenFlow [53] is the first standard communications interface defined between the control and forwarding layers of an SDN architecture. Existing

OpenFlow-based SDN testbeds include MiniNet [54], MiniNet-HiFi [55], OF-Test [56], OFlops [57], and NS-3 [58]. MiniNet probably is the most widely used SDN emulation testbed at present. It uses an OS-level virtualization technique called Linux container, and is able to emulate scenarios with 1000+ hosts and Open vSwitches [59]. However, MiniNet has not yet achieved performance fidelity, especially with limited resources, since resources are time-multiplexed by the kernel, and the overall bandwidth is limited by CPU and memory constraints. Execution of real programs in virtual machines exhibits high functional fidelity, and creation of multiple virtual machines on a single physical machine provides scalability and flexibility for running networking experiments, but low temporal fidelity is a major issue for virtual-machine-based network emulation systems. By default, all virtual machines use the same system clock of the physical machine. As a result, when a virtual machine is idle, its clock still advances. That raises the temporal fidelity issue when applications running on a virtual machine are expected to behave as if they were being executed on a real machine. All existing SDN emulation testbeds lack temporal fidelity. We have developed an emulation virtual time system in our prior work [2]. Freeing the emulation from real time enables us to run experiments slower or faster than real time. When resources are limited, we can always slow down experiments to ensure accuracy. On the other hand, experiments can be accelerated if system resources are sufficient. In this work, we extend the system to support OpenFlow-based SDN emulation with high functional fidelity and temporal fidelity.

While emulation testbeds offer fidelity, they are not suitable for large-scale network experiments. Simulation testbeds can provide scalability, but the accuracy degrades because of models' simplification and abstraction. Therefore, in prior work, we integrated a parallel network simulator with a virtual-machined-based emulation system based on virtual time [3]. When conducting network experiments, we can execute critical components in emulation and use simulation to provide a large-scale networking environment with background traffic. In this work, we also develop a framework to support OpenFlow-based SDN simulation, including models of an OpenFlow switch, controller, and protocol. The ns-3 network simulator also has an OpenFlow module [58] supporting both simulation and emulation, but, unlike our system, it does integrate virtual-time-based emulation and parallelized simulation. In addition, we have conducted extensive studies of application-level behaviors in our emulation system [60], and discovered that the errors introduced by the emulation system are bounded by one timeslice, which is the system execution unit (e.g., 100 μs). However, the processing delay within a gigabit switch is on the scale of one microsecond. If such a delay is critical to the SDN applications being tested, we can use simulation models instead, in which the time granularity is defined by the users to achieve the desired timing accuracy.

In this work, we extend our network testbed with the capability to emulate and simulate OpenFlow-based SDNs. The extension is based on close analysis of how SDN controllers typically behave, which led to organizational and synchronization optimizations that deal with problems that might otherwise greatly limit scalability and performance. The testbed provides the following benefits and is a useful tool for SDN-based research in general:

- Temporal fidelity through a virtual time system, especially with limited system and networking resources, e.g., it can emulate 10 hosts, each with a gigabyte Ethernet interface, on a single physical machine with only one one-gigabyte Ethernet network card
- More scalable network scenarios with simulated nodes
- Background traffic at different levels of detail among the simulated nodes
- Sophisticated underlying network environments (e.g., wireless communication CSMA/CA)

An OpenFlow controller typically connects with a large number of OpenFlow switches. Not only is the centralized controller a potential communication bottleneck in the real networks, but also it gives rise to performance drawbacks in simulating such networks in the context of parallel discrete-event simulation. The large number of switch-controller links could potentially reduce the length of the synchronization window in barrier-based global synchronization, and could also negatively impact the channel scanning type of local synchronization. We investigate techniques to improve the scalability of the simulated OpenFlow controllers through analysis of a list of basic controller applications in POX, which is a popular OpenFlow controller written in Python [61]. We classify the controller applications into active controllers and passive controllers based on whether the controllers proactively insert rules into the switches, or the rule insertions are triggered by the switches. We design an asynchronous synchronization algorithm for the passive controllers, and a two-level architecture for the active controllers for use in building scalable OpenFlow controller models.

The main contributions of this work are as follows: (1) We develop a network testbed for simulating and emulating OpenFlow-based SDNs. The testbed integrates a virtual-time embedded emulation system and a parallel network simulator to achieve both fidelity and scalability. (2) We explore and evaluate means to improve the performance of simulation of OpenFlow controllers in parallel discrete-event simulation, including an asynchronous synchronization algorithm for controllers that are "passive" (according to a definition given in Section 2.8.4), and a two-level architecture for controllers we classify as "active." (3) The two-level controller architecture is also a valuable reference for designing real scalable SDN controllers.

2.8.2 OpenFlow-Based Software-Defined Networks

In a traditional network architecture, the control plane and the data plane cooperate within a device via internal protocols. By contrast, in an SDN, the control plane is separated from the data plane, and the control logic is moved to an external controller. The controller monitors and manages all of the states in the network from a central vantage point. The controller talks to the data plane using the OpenFlow protocol [62], which defines the communication between the controller and the data planes of all the forwarding elements. The controller can set rules about the data-forwarding behaviors of each forwarding device through the OpenFlow protocol, including rules such as drop, forward, modify, or enqueue.

Each OpenFlow switch has a chain of flow tables, and each table stores a collection of flow entries. A *flow* is defined as the set of packets that match the given properties, e.g., a particular pair of source and destination MAC addresses. A flow entry defines the forwarding/routing rules. It consists of a bit pattern that indicates the flow properties, a list of actions, and a set of counters. Each flow entry states "execute this set of actions on all packets in this flow," e.g., forward this packet out of port A. Figure 2.18 shows the

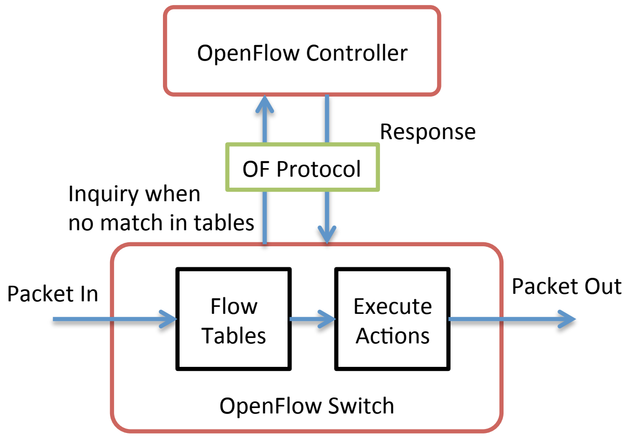


Figure 2.18: How an OpenFlow Switch Handles Incoming Packets

main components of an OpenFlow-based SDN and the procedures by which an OpenFlow switch handles an incoming packet. When a packet arrives at a switch, the switch searches for matched flow entries in the flow tables and executes the corresponding lists of actions. If no match is found for the packet, the packet is queued, and an inquiry event is sent to the OpenFlow controller. The controller responds with a new flow entry for handling that queued packet. Subsequent packets in the same flow will be handled by the switch without contacting the controller, and will be forwarded at the switch's full line rate.

Some benefits of applying SDN in large-scale and complex networks include the following:

- The need to configure network devices individually is eliminated.
- Policies are enforced consistently across the network infrastructures, including policies for access control, traffic engineering, quality of service, and security.
- Functionality of the network can be defined and modified after the network has been deployed.
- The addition of new features does not require change of the software on every switch, whose APIs are generally not publicly available.

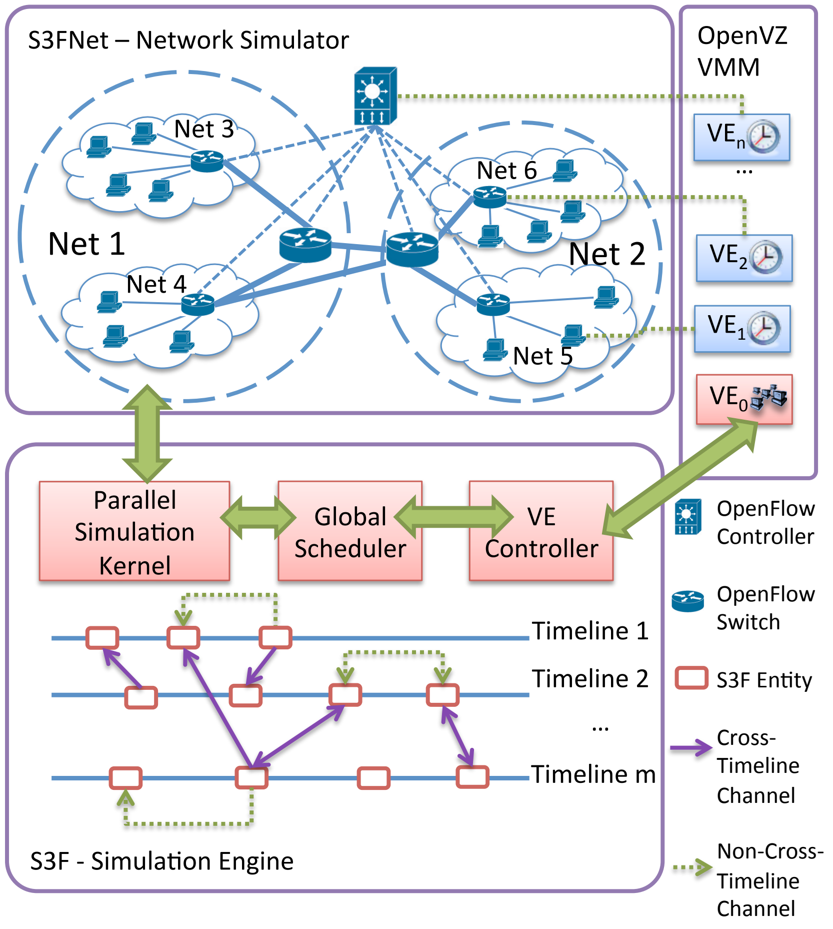


Figure 2.19: S3F System Architecture Design with OpenFlow Extension

2.8.3 OpenFlow Integration

The design of SDN separates the control plane from the data plane in network forwarding devices (switches, routers, gateways, or access points). Users can design their own logically centralized controllers to define the behaviors of those forwarding elements via a standardized API, such as OpenFlow, which provides flexibility to define and modify the functionalities of a network after the network has been physically deployed. Many large-scale networks, such as data centers, carriers, and campus networks, have been deployed with SDNs. Network simulation/emulation is often used to facilitate the testing and evaluation of large-scale network designs and applications running on them. Therefore, we extend our existing network testbed to support OpenFlow, in particular, simulation and/or emulation of network experiments that contains OpenFlow switches and controllers, which communicate via the OpenFlow protocol. The testbed architecture design is depicted in Figure 2.19.

To emulate OpenFlow-based networks, we can run unmodified OpenFlow switch and controller programs in the OpenVZ containers, and the network environment (such as wireless or wireline media) is simulated by S3FNet. Since the executables are run on the real network stacks within the containers, the prototype behaviors are close to the behaviors in real SDNs. Once an idea works on the testbed, it can be easily deployed to production networks. Other OpenFlow emulation testbeds, like MiniNet [54], have good functional fidelity too, but lack performance fidelity, especially with heavy loads. Imagine that one OpenFlow switch with ten fully loaded gigabit links is emulated on a commodity physical machine with only one physical gigabit network card. There is no guarantee that a switch ready to forward a packet will be scheduled promptly by the Linux scheduler in real time. Our system does not have such limitations, since the emulation system is virtual-time embedded. The experiments can run faster or slower depending on the workload. When load is high, performance fidelity is ensured by running the experiment slower than real time. On the other hand, when the load is low, we can reduce the execution time by quickly advancing the experiment.

However, experimental results show that errors introduced by the virtual time system at the application level are bounded by the size of one emulation timeslice. We may reduce the error bound by setting a smaller hardware interrupt interval [2]. Nevertheless, the interval cannot be arbitrarily small because of efficiency concerns and hardware limits. We typically use 100 μs as the smallest timeslice. If we want to simulate a gigabit switch whose processing delay is on the scale of a micro-second, a 100 μs error bound is simply too large. That motivated us to develop a simulated OpenFlow switch and an OpenFlow controller model; the simulation virtual time unit is defined by the users and can be arbitrarily small, typically a micro-second or nano-second.

Our OpenFlow simulation model has two components: the OpenFlow switch, and the OpenFlow controller. The switches and the controller communicate via the OpenFlow protocol [62], and the protocol library we use is the OpenFlow reference implementation at Stanford University [63], which has been used in many hardware-based and software-based SDN applications. The initial version of the switch and the controller models have been developed with reference to the ns-3 OpenFlow models [58]. Figure 2.20 depicts the model implementation details.

Our OpenFlow switch model can handle both simulated traffic and real traffic generated by the applications running in the containers. The switch model has multiple ports, and each port consists of a physical layer and a data link layer. Different physical and data link layer models allow us to simulate different types of network, such as wireless and wireline networks. A switch layer containing a chain of flow tables is located on top of the ports, as shown in Figure 2.20. It is responsible for matching flow entries in the tables and performing the corresponding predefined actions or sending an inquiry to the controller when no match is found in the tables. The controller model consists of a group of applications (e.g., learning switch, link discovery) as well as a list of connected OpenFlow switches. It is responsible for generating and modifying flow entries and sending them back to the switches.

Needless to say, running executables of OpenFlow components in the emulation mode has better functional fidelity than the simulation models do. We attempt to keep the high fidelity in the simulation models by using the original unmodified OpenFlow library, which has been used to design many real SDN applications. Also, the simulation models are not constrained by the 100 μs timeslice error bound in emulation. In addition, we can run experiments in the simulation mode with much larger network sizes. Finally, as a bonus effect of the OpenFlow design, we no longer have to preload a forwarding table at every network device, since where and how to forward the packets are decided on demand by the controller. Simulating a network with millions or even more network devices at the packet level is affordable in our system.

Figure 2.20 also shows the journey of a packet in our system. A packet is generated at the source end-host, either from the simulated application layer or from the real network application in a container. The packet is pushed down through the simulated network stacks of the host and then is popped up to the OpenFlow switch via the connected in-port. Depending on the emulation or simulation mode of the switch (the OpenVZEmu module in Figure 2.20 is used to handle the case where the entity is associated with a virtual-machine container), the packet is searched within the simulated flow tables or the real flow tables in the container, and a set of actions are executed when matches are found. Otherwise, a new flow event is di-

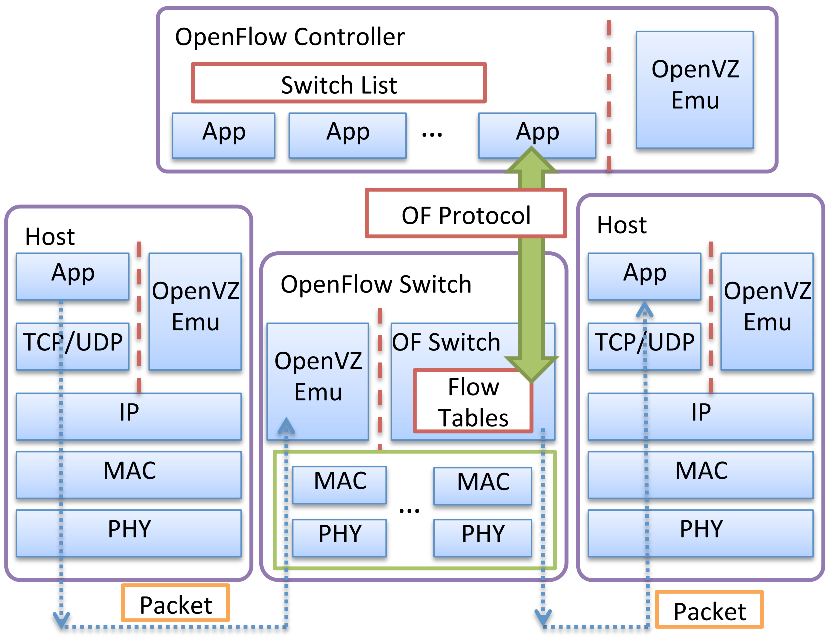


Figure 2.20: OpenFlow Implementation in S3F

rected to the controller, meaning either the simulated controller model or the real controller program in the container. The controller generates new flow entries and installs the flow entries onto the switches via the OpenFlow protocol. Afterward, the switch knows how to process the incoming packet (and subsequent additional packets of this type), and transmit it via the correct out-port. Eventually the packet is received by the application running on the destination end-host.

2.8.4 Scalability of Simulated OpenFlow Controllers

SDN-based network designs have multiple OpenFlow switches communicating with a single OpenFlow controller. However, many-to-one network topologies not only create communication bottlenecks at the controllers in real networks, but also negatively impact the performance of conservative synchronization of parallel discrete-event simulations. The conservative synchronization approaches in parallel discrete-event simulation generally fall into two categories: synchronous approaches based on barriers [64], [65], [66], and asynchronous approaches, in which a submodel's advance is a function of the advances of other submodels that might affect it [67]. The single-controller-to-many-switch architecture can be bad for both types of synchro-

Table 2.11: Classification of the Basic Network Applications in the POX OpenFlow Controller

Controller Application	**Description**	**Passive/ Active**	**State**	**Other Comments**
openflow. keep_alive	The application sends periodic echo requests to connected switches. Some switches will assume that an idle connection indicates a loss of connectivity to the controller and will disconnect after some period of silence, and this application is used to prevent that.	Active	Distributed	The request-sending period is a good source of lookahead.
forwarding. l2_learning	The application makes an OpenFlow switch act as a type of L2 learning switch. While the component learns L2 addresses, the flows it installs are exact matches on as many fields as possible. For example, different TCP connections will result in installation of different flows.	Passive	Distributed	
forwarding. l2_pairs	The application also makes an OpenFlow switch act like a type of L2 learning switch. However, it installs rules based purely on MAC addresses, probably the simplest way to do a learning switch correctly.	Passive	Distributed	
forwarding. l2_learning	The application is not quite a router. It uses L3 information to perform data switching similar to an L2 switch.	Passive	Distributed	
openflow. link_discovery	The application sends specially crafted messages out of OpenFlow switches to discover link status, e.g., to discover the network topology. Switches raise events to the controller when links go up or down.	Active	Shared among the connected switches	The link status query period is a good source of lookahead.
forwarding. l2_multi	This application can still be seen as a learning switch, but it learns where a MAC address is by looking up the topology of the entire network.	Passive	Network-wide	
openflow. spanning_tree	This application constructs a spanning tree based on the network topology, and then disables flooding on switch ports that are not on the tree.	Active	Network-wide	

nization.

We can view a network model as a direct graph: nodes are entities like hosts, switches, and routers; edges are the communication links among entities; and each link is weighted by a link delay. From S3F, the parallel simulation engine's viewpoint, the graph is further aggregated: a node represents a group of entities on the same timeline, and the simulation activity of all entities on a timeline is serialized; multiple links between timelines t_i and t_j are simplified into one link whose weight is the minimum (cross-timeline) delay between t_i and t_j.

A barrier-based synchronous approach is sensitive to the minimum incoming edge weight in the entire graph. If one of the OpenFlow switch-controller links has a very small link delay (e.g., a controller and a switch could be installed on the same physical machine in reality), even if there are few activities running on the link, the overall performance will be poor because of the small synchronization window. On the other hand, an asynchronous approach focuses on timeline interactions indicated by the topology, but is subject to significant overhead costs on timelines that are highly connected. The SDN-based architectures unfortunately always have a centralized controller with a large degree, which is not a desired property for asynchronous approaches either.

To improve the simulation performance with SDN architectures, we explore the properties of an OpenFlow controller with reference to a list of basic applications in POX, which is a widely used SDN controller written in Python [61]. The applications are summarized in Table 2.11. We have two key observations.

First, controllers can be classified as either passive or active. A controller is *passive* if the applications running on it never initiate communication to switches, but only passively respond to inquires from switches when no matches are found in switches' flow tables. The forwarding.l2_learning application is a good example of an application that runs on a passive controller. An *active* controller initiates communication to switches, e.g., detecting whether a switch or a link is working or broken. The openflow.link_discovery application is an example of an application that runs on an active controller.

Second, a controller is not simply a large single entity shared by all the connected switches. A controller actually has states that are shared by switches at different levels. Suppose there are N switches in a network, and m switches ($1 \leq m \leq N$) share a state.

- When $m = N$, the state is network-wide, i.e., the state is shared by all switches. For example, the openflow.spanning_tree application has a network-wide state, which is the global network topology.
- When $m = 1$, the state is distributed, i.e., no other switch shares the state with this switch. For example, the forwarding.l2_learning application has a distributed state, which is the individual learning table for each switch.
- When $1 < m < N$, the state is shared by a subset of switches of size m. For example, the openflow.link_discovery application has such a state, which is the link status shared among all the switches connected to that link.

Based on the aforementioned two observations, we revisit the controller design and investigate techniques to improve the performance of the simulation of OpenFlow-based SDNs with parallel discrete-event simulation. In particular, we design an efficient asynchronous algorithm for passive controllers; we also propose a two-level controller architecture for active controllers, and analyze performance improvement for three applications with different types of states. The proposed architecture is not only helpful with respect to simulation performance, but also a useful reference for designing scalable OpenFlow controller applications.

Passive Controller Design

A passive controller indicates that applications running on the controller do not proactively talk to switches, a feature we can use in designing a controller whose functionality is known to be passive. A new design is also motivated by another observation: when a switch receives an incoming packet, the switch can handle the packet without consulting the controller if a matched rule is found in the switch's flow table; and, for some applications, the number of times the controller must be consulted (e.g., first time the flow is seen, or when the flow expires) is far lower than the number of packets being processed locally. All the learning switch applications in the POX controller have this property.

Therefore, our idea is that for a passive controller, switches are free to advance their model states without constraint, until the switches have to communicate with the controller. If multiple switches share a state, then the controller needs to refrain from answering the inquiring switch until all its cross-timeline dependent switches have advanced to the time of the inquiring switch. The algorithm is presented as follows:

Let A be the set of applications running in the OpenFlow controller, and R be the set of OpenFlow switches in a network model. We define $TL(r)$ to be the timeline to which switch r is aligned. For each $a \in A$ and $r \in R$, we define $f(r, a)$ to be the subset of OpenFlow switches that share at least one state with switch r for application a. For example, $f(r_1, a_1) = \{r_2, r_3\}$ means that switches r_1, r_2, r_3 are dependent on (sharing states with) application a_1. Causality is ensured only if the controller responds to the inquiry from switch r_1 with timestamp t_1, after all the dependent cross-timeline switches r_i, i.e., $r_i \in f(r_1, a)$ and $TL(r_i) \neq TL(r_1)$, have advanced their times to at least t_1. For an application with network-wide states, $f(r, a) = R - \{r\}$; for a fully distributed application, $f(r, a) = \phi$.

Algorithm 3 is designed for efficient synchronization among switches and fast simulation advancement with correct causality in the case of a passive controller. The algorithm is divided into two parts: one at the controller side and another at the switch side. Since the controller cannot actively affect a switch's state, it is safe for a switch to advance independent of the controller until a switch-controller event happens (e.g., a packet is received that has no match in the switch's flow tables). The delays between the controller and the switches thus do not affect the global synchronization. The causality check is performed at the controller, since it has the global dependency information of all the connected switches. Upon receiving an OpenFlow inquiry, the controller is responsible for making sure no response will be sent back to the switch (thus, the switch will not advance its clock further) until all its dependent switches have caught up with it in simulation time. In addition, this design does **not** require that the controller be modeled as an S3F entity, which means that the controller does not have to align with any timeline. All the interactions can be done through function calls instead of via message passing through S3F channels. This design works only for a passive controller and can greatly reduce the communication overhead between the controller and switches. As a result, a passive controller is not a bottleneck in conservatively synchronized parallel simulation, as low latency to switches and high fan-out might otherwise cause it to be.

Figure 2.21 illustrates our synchronization algorithm with an example. The steps are as follows:

1. Switch r_1 sends an inquiry to the controller with $t_1(now)$.
2. The controller gets the current times of all dependent cross-timeline switches, e.g., $t_2(now)$, $t_3(now)$.
3. Because $t_2(now) < t_1(now)$, the controller schedules a timing report event at $t_1(now)$ on timeline 2; no event is scheduled on timeline 3 since $t_3(now) > t_1(now)$.
4. At $t_1(now)$ on Timeline 2, r_2 signals the controller.
5. The controller generates rule(s) based on the up-to-date states and sends a response back to r_1.

The algorithm works for all passive controllers, whether the application states are distributed (e.g., forwarding.l2_learning) or network-wide (e.g., forwarding.l2_multi). The state property plays an important role in the active controller design, which will be discussed in Section 2.8.4. However, the performance of passive controllers does benefit from use of distributed states as well as the smaller number of cross-timeline dependent switches.

Active Controller Design

Active OpenFlow controllers proactively send events to the connected OpenFlow switches, and those events can potentially affect the states of switches. Therefore, the switches do not have the freedom to advance the model states like those switches that connect to passive controllers, but are subject to the minimum link latency between the controller and the switches. However, the question we have is: are the assumptions about connectivity in SDNs overly pessimistic? For example, can any timeline generate an event at any instant that might affect every other timeline?

Algorithm 3 Synchronization Algorithm with Passive Controller

```
Controller Side
/* Upon receiving an OpenFlow inquiry from switch $r_i$ with timestamp $t_i$ */
for each application $a_j$ related to the inquiry do
  for each switch $r_k \in f(r_i, a_j)$ AND $TL(r_k) \neq TL(r_i)$ do
    get the current simulation time, $t_k$, of $r_k$
    if $t_k < t_i$ then
      schedule a timing report event at time $t_i$ on the timeline of $r_k$
      increase $dcts[i]$ by 1
      /* $dcts[i]$ is the counter of unresolved dependent cross-timeline
      switches for switch $r_i$ */
    end if
  end for
end for

pthread_mutex_lock()
while $dcts[i] > 0$ do
  pthread_cond_wait()
end while
pthread_mutex_unlock()

process the inquiry (i.e., generate rules to handle packets)
send an OpenFlow response to switch $r_i$

Switch Side
/* Upon receiving a packet at an ingress port */
check flow tables
if found matched rule(s) then
  process the packet accordingly
else
  send an OpenFlow inquiry to the controller
end if

/* On reception of an OpenFLOW response */
store the rule(s) in the local flow table(s)
process the packet accordingly

/* Scheduled timing report event for switch $r_i$ fires */
pthread_mutex_lock()
decrease $dcts[i]$ by 1
if $dcts[i] = 0$ then
  pthread_cond_signal()
end if
pthread_mutex_unlock()
```

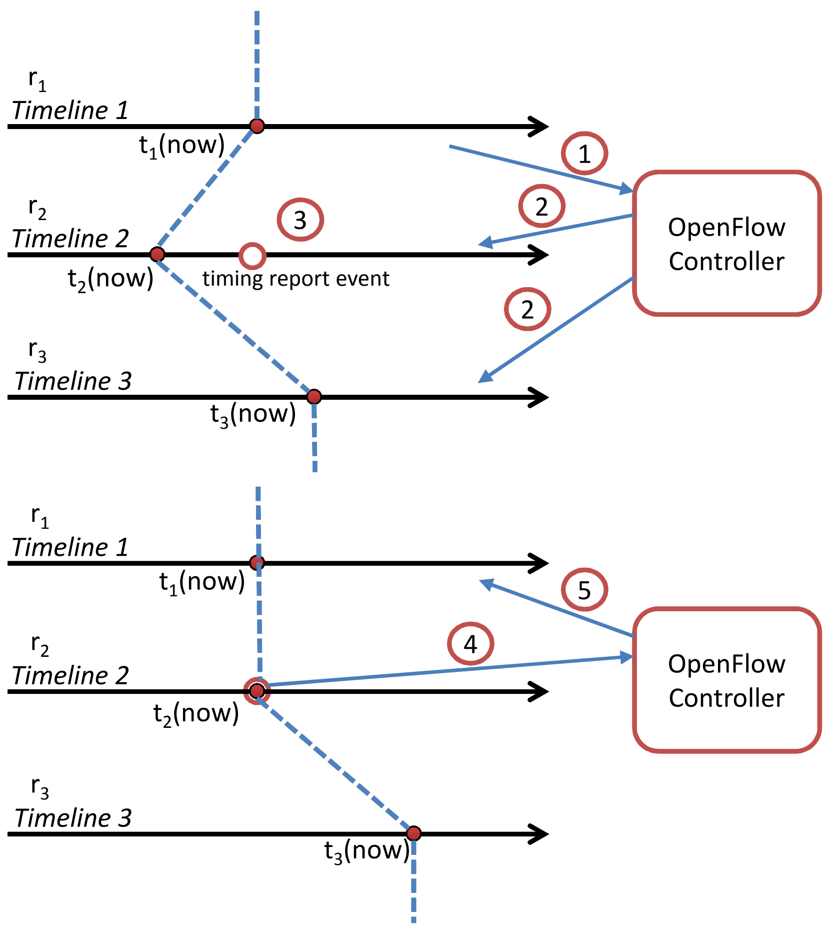


Figure 2.21: Sample Communication Patterns between the OpenFlow Switch and the Passive OpenFlow Controller Using Our Synchronization Algorithm

We make the following observations about the active controller applications. First, not all controllers have only network-wide states. Some have fully distributed states, e.g., a switch's on/off state (openflow.keep_alive application); some have states that are shared among a number of switches, e.g., a link's on/off state is shared among switches connected to the same link (openflow.link_discovery application). Second, not all events will result in global state changes, and quite often a large number of events are handled locally, and only influence switches' local states. For instance, only when a communication link fails, or when a link is recovered from failure or when some new switches join or leave the network, a link status change event is generated for the openflow.link_discovery application, so that it updates its global view; during the remaining (most of the) time, the network-wide state, i.e., the global topology, remains unchanged. Therefore, for such applications, instead of having a centralized controller frequently send messages to all the connected switches and wait for responses, we can first determine which events affect network-wide states and which do not, and then off-load those events, which only cause local state changes, towards the switch side. Thus, we relieve the pressure at the controller.

Based on our observations of active controller applications, we propose a two-level active controller architecture as depicted in Figure 2.22. Local states are separated from network-wide states in a controller, and the controller is divided into a top-controller and a number of local controllers. The top controller communicates only with the local controllers, not with the switches, to handle events that potentially affect the network-wide states. The local controllers run applications that can function using the local states in switches, e.g., the local policy enforcer, or link discovery. There are no links among local controllers. With the two-level controller design, we aim to improve the overall scalability, especially at the top controller, as follows:

- The top controller has a smaller degree, which is good for local synchronization approaches.

- Fewer messages are expected to occur among local controllers and the top controller for many applications, since the heavy communication is kept between the switches and local controllers. That is good for local synchronization approaches as well.

- If we align the switches that share the same local controller to the same timeline, the local controllers actually do not have to be modeled as an entity. Message passing through channels is not needed, as function calls are sufficient.

Conversion of a controller into a two-level architecture requires that modelers carefully analyze the states in the controller applications. That process is not only useful in creating a scalable simulation model, but also helpful in designing a high-performance real SDN controller, because it offloads local-events processing to local resources. In the remainder of this section, we will inspect three active controller applications with our two-level controller design: (1) openflow.keep_alive with fully distributed states, (2) open-

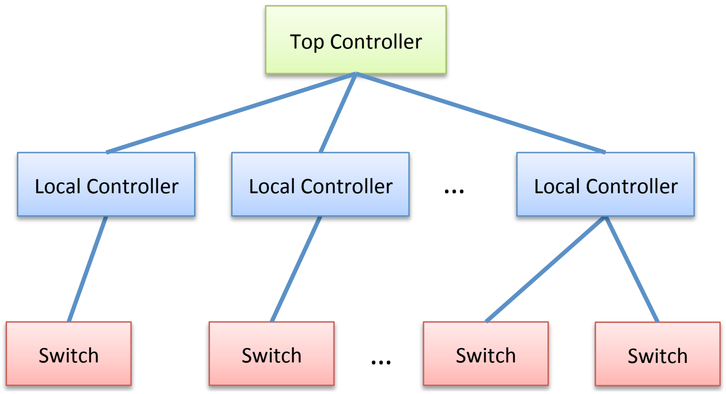


Figure 2.22: Two-Level Controller Architecture

flow.link_discovery with shared states, and (3) openflow.spanning_tree with network-wide states.

The **openflow.keep_alive** application is designed to send periodic echo requests to all connected switches to prevent loss of connections, since some switch firmware by default treats an idle link as a link failure. Meanwhile, the controller also maintains a complete list of the running switches in the network. Since the on/off state of a switch is independent of other switches, the openflow.keep_alive is an example of applications with fully distributed states. In the two-level controller design, one local controller is created for each switch. Let us now compute the total number of simulation events (e.g., message-sending events, message-receiving events, and time-out events) within one echo request period with the single controller and the two-level controller.

Assume that the network has n switches, and let p be the probability that a switch fails during one request period. In the single-controller case, the controller needs one timer event to enable the echo request sending process, and then sends one request to each switch. If a switch is working, the request is received at it, and it replies with a response; finally, the controller receives the response. If a switch fails, a timeout event occurs at the controller. Therefore, for the single-controller case, the total number of events within one period is $M_1 = 1 + n + (1-p) * n * (1+1+1) + p * n = 1 + 4n - 2pn$. In the two-level controller case, we have n local controllers, and each local controller fires one timer event to start the request-sending process. Since the communication with the local controllers and the switches is modeled by function calls instead of by message passing, no event occurs as long as a switch is working. If a switch fails, the local controller reports the state change to the global controller, and the global controller receives the report. Therefore, for the two-level controller case, the total number of events within one period is $M_2 = n + p * n * (1 + 1) = n + 2pn$.

Assume $n \gg 1$; when the fail rate of a switch, p, is 0.75, the single-controller and the two-level controller have approximately the same total number of events. If $p < 0.75$, the two-level controller has fewer events to simulate. If p is close to 0, which means that switches rarely fail during experiments, $M_2 \approx n$ events, and $M_1 \approx 4n$ events. Also, let us count the total number of events processed at the top controller within one period. The single controller has the total number of events $C_1 = 1 + n + n * (1 - p) + n * p = 1 + 2n$, and the two-level controller has the total number of events $C_2 = np$. The latter is always smaller than the former, and thus the two-level controller for the openflow.keep_alive application results in a more scalable controller. Actually, it is possible to remove the top controller as long as all the applications running in the controller are fully distributed, like openflow.keep_alive. If so, we can prevent a single global controller from becoming a potential system bottleneck.

The **openflow.link_discovery** application periodically requests messages from switches in order to discover the link status, and switches raise events to the controller when links go up or down. Potentially, this application could be used to discover the network topology. The states of this application are the links' on/off states shared among all the switches connecting to the link. With the two-level controller design, we assign one local controller to each switch, and let the top controller manage the shared states.

Let us calculate the total number of simulation events within one link discovery period for both cases; the switch-switch and switch-host interactions are not counted, since they are the same for both cases. Assume that the network has n switches, and that the probability that a switch has at least one link state change is q. For the single-controller case, the controller fires one timer event to start the link discovery process, and then sends each switch a link discovery request. Upon receipt of the request at every switch, the switch sends back a response if at least one of its links has a state change. Finally, the response is received at the controller to update the global topology. Therefore, the total number of events within one period is $M_1 = 1 + n + n + q * (n + n) = 1 + 2n + 2qn$. For the two-level controller case, n local controllers fire one timer event each to start the link discovery process locally. Since the interaction between the local controller and the switch is modeled by function calls instead of by message passing, no event is generated if no link state change is detected. If a link change is discovered by the switch, the local controller then reports the change to the global controller, and the global controller receives the report. Therefore, the total number of events within one period for the two-level controller is $M_2 = n + q * n(1 + 1) = n + 2qn$. We can see that there are $2n$ fewer events than in the single-controller scenario. Let us count the number of events processed at the top controller within one period. The single controller has $1+n+qn$ events, while the two-level controller has qn events. Therefore, the two-level controller always has better scalability, especially when the network has a relatively steady topology. The two-level controller is also a good reference design for the real link discovery application in the SDN controller. The number of packets being processed at the top controller is greatly reduced to achieve good scalability.

Good lookahead can improve performance in simulating networks with either the single controller or the two-level controller. We notice that the link discovery request is sent to the switch periodically. The period is often quite long compared with the time to transmit a packet, and the period can serve as a good source of lookahead. In addition, another optimization we could do is to model the top controller as a passive controller since the requests are now initiated at the local controllers.

The **openflow.spanning_tree** application creates a spanning tree based on the network topology, and then it disables flooding on switch ports that are not on the tree. The application has only one state, the global network topology. The state is a network-wide state. Therefore, it is hard to convert this application to a two-level controller. However, the openflow.spanning_tree application still has invariants that we could utilize to improve the system performance.

S3F synchronizes its timelines at two levels. The inner level uses the barrier-synchronization and the channel-scanning-synchronization for parallel discrete-event simulation. At the outer level, timelines are left to run during an epoch, which could terminate 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. A spanning tree algorithm is often running at the beginning of an experiment and then is triggered by topology changes or a global timer. In the scenarios in which a network topology does not change frequently, it is reasonable to assume that the topology is invariant for a certain minimum length of time, and we can use that time length as the length of an epoch. Between two epochs, a spanning tree is recomputed. States (particular spanning trees) created by those computations are otherwise taken to be constant when the simulation is running.

We have studied three active applications in the POX controller. Through careful application state analysis, we can often convert applications with local states into two-level controller architectures for simulation performance gain. The analysis not only helps modelers to create scalable SDN network models, but also helps in the design of scalable real SDN controllers. Many SDN applications can be far more complicated than the basic applications in POX. They could be a combination of passive and active applications, with both distributed and network-wide states, and the states may change dynamically with time and network conditions. Even so, it is still useful to divide a complex controller into small building-block applications, classify them according to the state type as well as passiveness/activeness, and then apply the corresponding performance optimization techniques.

2.8.5 Evaluation

We first study the performance improvement with the asynchronous synchronization algorithm for the passive controllers. We created a network model with 16 timelines. The backbone of the network model consisted of 32 OpenFlow switches, and each switch connected 10 hosts. Half of the hosts ran client applications, and the other half ran server applications. File transfers between clients and servers were on UDP. The minimum communication link delay among switches and hosts was set to be 1 ms. During the experiments, each client randomly chose a server from across the entire network, and started to download a 10 KB file. Once the file transfer was complete, the client picked another server and started the downloading process again. All the OpenFlow switches connected to an OpenFlow controller, and the controller ran the openflow.learning_multi application, which is essentially a learning switch, but learns where a MAC address is by looking up the topology of the entire network. In the first case, we modeled the controller as a normal S3F entity, and it connected to switches through channels. In the second case, we used the passive controller design described in Section 2.8.4. Since all switches shared the global network topology, they were dependent on one another. We ran the experiments on a machine with 16 2GHz-processors and 64 GB memory.

In the first set of experiments, we varied the link delay between the switches and the controller with values of 1 μs, 10 μs, 100 μs and 1 ms. Each experiment was executed for 100 seconds in simulation time for both cases. We observed 12,810 completed file transfers for both cases. Figure 2.23 shows the event-processing rates for both cases. In the first case, the traditional barrier-type global synchronization was used. Since the minimal controller-switch link delay was set to be no larger than other link delays within the network, that delay dominated the synchronization window calculation with the global synchronization algorithm in S3F. Therefore, the performance degraded as the minimum link latency decreased because of more frequent synchronization overheads. In the second case, synchronization was actually two-level: the global synchronization was used to simulate activities among switches and hosts, and within a synchronization window, the asynchronous synchronization algorithm was used to simulate interactions between the switches and the passive controller. Therefore, the controller-switch delay did not affect the global synchronization window size. For the passive controller case, the event-processing rate remained almost unchanged, and the performance was two to three times better than in the first case, when the controller-switch delay was small. The performance was still better in the second case even when the controller-switch link delay was set to 1 ms (equal to the minimal delay of the other links). The reason is that the interactions between the passive controller and the switches were through function calls instead of event-passings with S3F channels, which we explored further in the next set of experiments.

In the second set of experiments, we increased the number of OpenFlow switches connected to the controller, and the size of the network was thus increased proportionally. Every client application performed the same actions as in the first experiment set. The model still used 16 timelines, and each

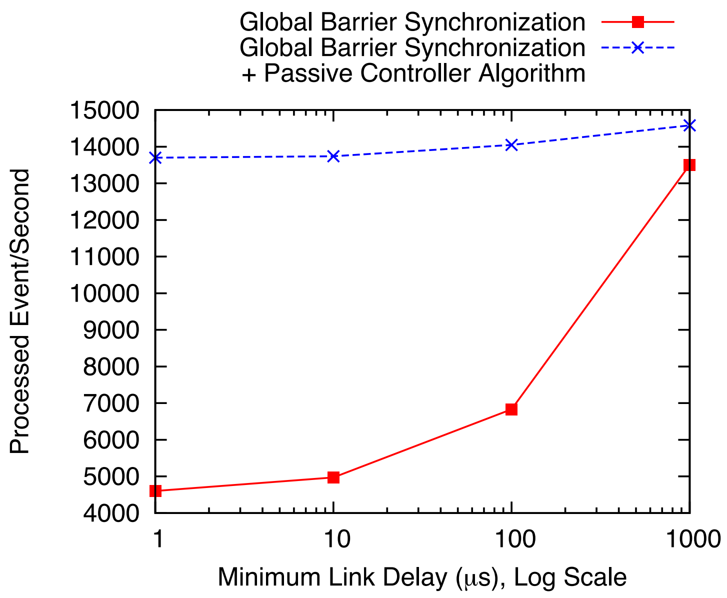


Figure 2.23: Performance Improvement with the Passive Controller Asynchronous Synchronization Algorithm: Minimal Controller-Switch Link Latency

experiment was simulated for 100 seconds with the two types of controller. Let λ be the event-processing rate of a model using the passive controller algorithm over the event-processing rate of the same model with the S3F entity controller. The λ indicates the performance improvement of asynchronous synchronization designed for passive controllers. As shown in Figure 2.24, λ grows as the number of switches increases. Since each switch needs a larger flow table as the size of the network increases, switches have more interactions with the controller. As a result, the performance gain of replacing the message-passing mechanism with function calls for the controller-switch interactions is more prominent. In addition, the modeled controller application has a network-wide state, resulting in a fully dependent switch list in the passive controller. More performance improvement is expected for modeling controller applications with distributed states.

We also investigated the performance improvement with the two-level architecture for active controllers. The openflow. keep_alive application was developed with the two designs as discussed in Section 2.8.4: (1) a single centralized controller, and (2) a two-level controller in which the local controllers are responsible for querying switch states and reporting to the top controller when switch failures are discovered. The network model had 80 OpenFlow

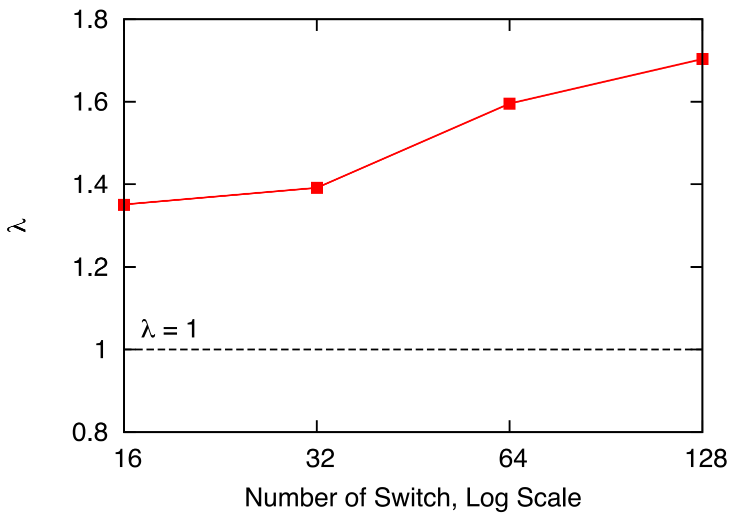


Figure 2.24: Performance Improvement with the Passive Controller Asynchronous Synchronization Algorithm: Number of OpenFlow Switches in the Network

switches with a minimal link delay of 1 ms, running with 16 timelines. The switch state inquiry period was set to be 1 second. Let us define the performance indicator ρ to be the event-processing rate of a model using the two-level controller architecture over the event-processing rate of the same model using the single controller. We varied the switch failure rate, and the results are shown in Figure 2.25.

The model with the two-level controller performed better than the model with the single controller under all switch failure rates. The reasons are that (1) switches and their local controller were in the same timeline, and the switch state inquiry activities were well-parallelized; and (2) the switch state responses were sent back to the top controller only when switch failures were detected, resulting in less data communication with the top controller. Simulating a network with smaller switch failure rates had better performance, e.g., 19 times faster with a zero failure rate, and 5 times faster with a 10% failure rate, as shown in Figure 2.25.

2.9 Chapter Summary

In this chapter, we present a network testing system that integrates an OpenVZ-based network emulation system [8] into the S3F simulation frame-

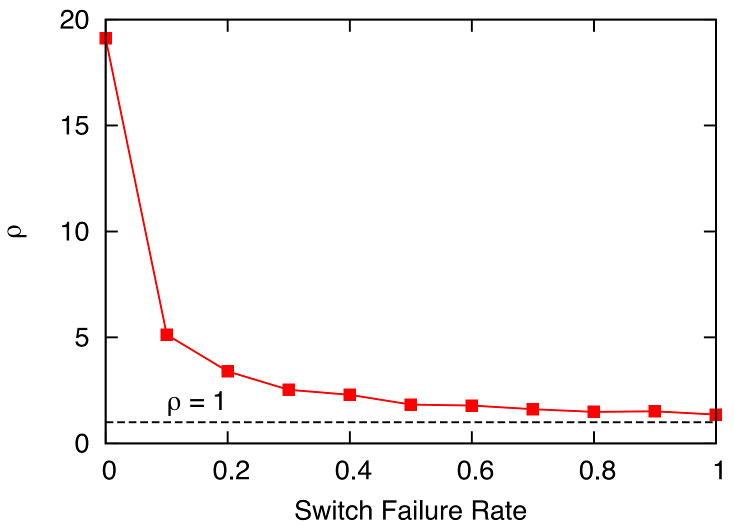


Figure 2.25: Performance Improvement for the openflow.keep_alive Application with a Two-Level Controller Architecture

work [1]. 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 facilitates 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 [2].

We also test and evaluate the application behaviors of the testbed. The test cases are designed to cover both network-intensive application and CPU-intensive applications over a group of commonly-used protocols. We found the minimal temporal error (100 μs) introduced by the emulation does not introduce additional behavioral errors in all the tested applications than the known errors that are bounded by the scale of a timeslice. We also observe that the errors caused by the native OpenVZ kernel is larger than the ones caused by the virtual time system, especially for TCP-based applications. Overall, the application-level validation study suggests that our emulation/simulation testbed can deliver behaviors with temporal errors no greater than those induced simply by using OpenVZ virtualization.

In addition, we present a few models developed in the testbed, including background traffic models and SDN models.

Background traffic models are developed for simulating network traffic flow at a coarse scale with the fundamental assumption that the network switches manage queueing using FCFS scheduling and/or fair queueing scheduling. The design of the model, such as the way multiple input flows share the bandwidth of an output port, the independent behavior among output ports and aggregation of flows having same input-output-port pair, are validated from the data collected on the real switches using an unique traffic measurement testbed. The background traffic models run extremely fast relative to packet-based flow simulation and the impact on foreground flow is still accurate enough for our purposes.

We have extended our network testbed to support OpenFlow-based SDNs simulation and emulation. Users can conduct SDN-based network experiments with real OpenFlow switch and controller programs in virtual-time-embedded emulations for high functional and temporal fidelity, or with OpenFlow switch and controller simulation models for scalability, or with both. We also explore ways to improve the scalability of the centralized simulated controllers, including an asynchronous synchronization algorithm for passive controllers and a two-level architecture design for active controllers, which also serves as a useful guideline for real SDN controller design.

The current system requires both OpenVZ and S3F to reside on the same shared memory multiprocessor. One of our ongoing works is to separate those to support multiple machines running virtual machine managers (i.e. distributed emulation), 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. Regarding the SDN simulation and emulation models, we plan to evaluate the network-level and application-level behaviors for different SDN-based applications under various network scenarios (e.g., long delay, or lossy link). We would also like to conduct a performance comparison with ground truth data collected from a real physical SDN testbed as well as data collected from other testbeds, like MiniNet. In addition, we plan to utilize the testbed to design and evaluate SDN applications, especially in the context of the smart grid; an example would be design of efficient quality-of-service mechanisms in substation routers, where all types of smart grid traffic aggregate.

CHAPTER 3

NETWORK SIMULATION-BASED EVALUATION OF SMART GRID APPLICATIONS

We have built a large-scale, high-fidelity network testbed by marrying our S3F parallel simulator [1] with our virtualization-based emulation framework [2] to enhance the successful transition from in-house research efforts to real smart grid productions. The testbed uses emulation to represent the execution of real critical software, and simulation to model an extensive ensemble of background computation and communication. We have utilized the testbed to study various smart grid applications. Examples include a DDoS attack using C12.22 trace service in AMI network (Section 3.1), an event buffer flooding attack in DNP3-controlled SCADA systems [68] (Section 3.2), and demand response control algorithms in a hierarchical transactive control network [69] as part of the Pacific Northwest smart grid demonstration project [70] (Section 3.3).

3.1 AMI Network: A DDoS Attack Using C12.22 Trace Service

3.1.1 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 3.1 illustrates this attack.

3.1.2 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 3.2. The meters

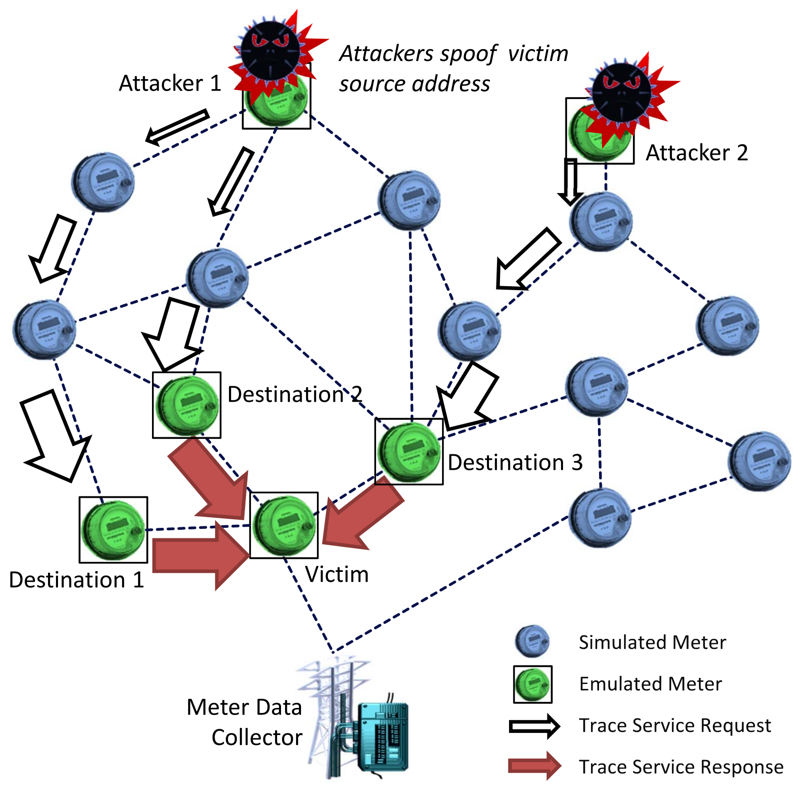


Figure 3.1: C12.12 Trace Service DDoS Attack in AMI Network

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 3.2-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 chokepoint 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

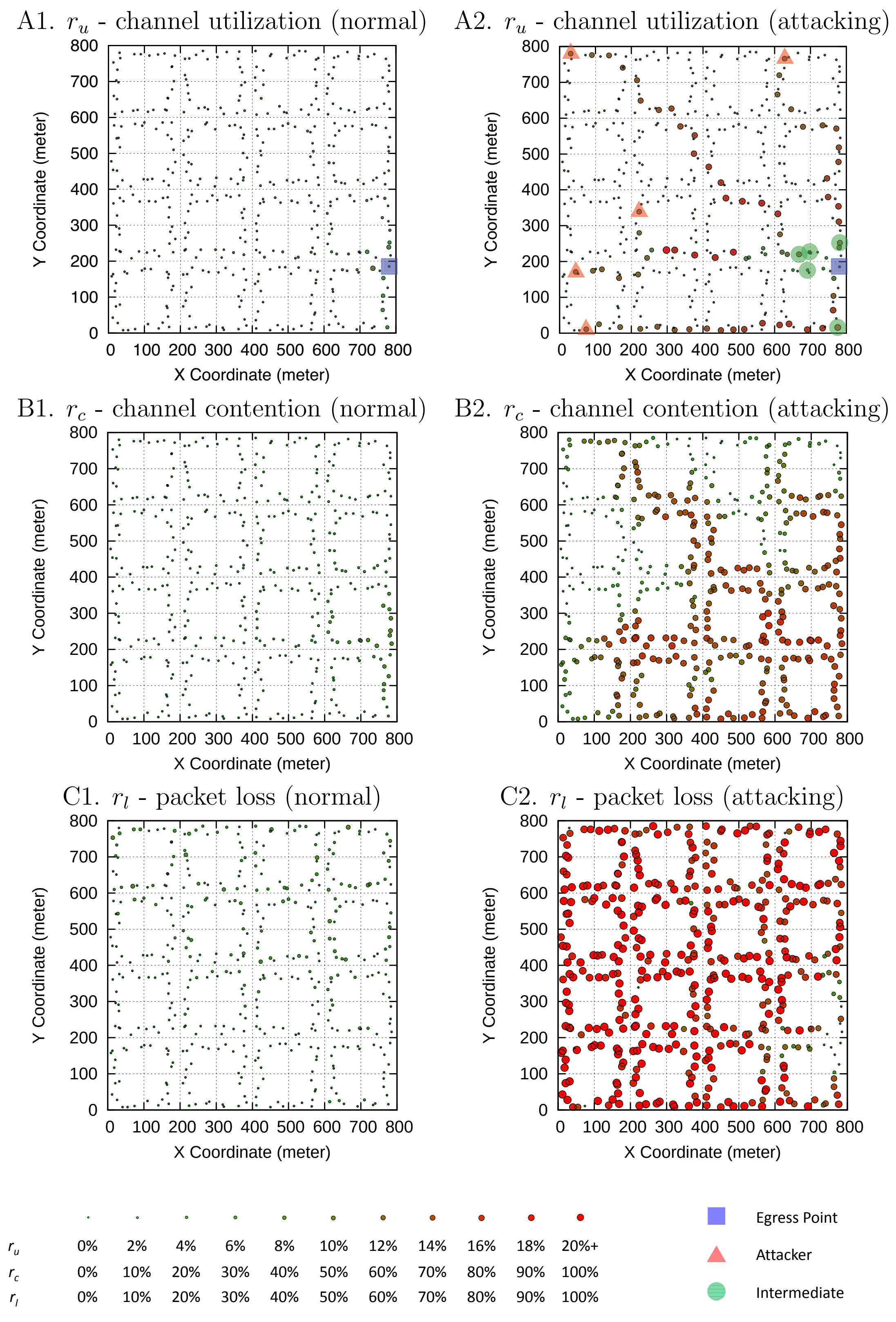


Figure 3.2: Experimental Results of DDoS Attacks in AMI Networks Using C12.22 Trace Service

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 data is collected in a 100 second window and the results are shown in Figure 3.2 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 3.2-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 3.2-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 that forward the traffic but also jams the channels of meters around them. Figure 3.2-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 3.2-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 3.2-B2.

In the third set of experiments, we compared results in Figure 3.2-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 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.

3.1.3 Scalability

How large a model might this system be capable of simulating? To see, we increase the size of the model by adding the number of neighborhoods (hence number of simulated meters and links, and length of communication paths). There is one egress point in each test case, to which every meter periodically sends data traffic. Each experiment uses 32 timelines and runs for 1 hour in virtual time. Table 3.1 summarizes the experimental results. We are able to simulate a system as large as 64 by 64 neighborhood, with 100,000+ meters, 600,000+ links and 437 billion of events. As the model size increases, the event rate remains around 6M $\sim$ 7M event/second. The results indicate that the performance of our testbed scales.

Table 3.1: Simulation Scalability Test Results Using AMI DDoS Test Cases

Block	#Host	#Link	Simulation Time (s)	#Events (M)	Event Rate (M Event/s)
2 X 2	112	392	13	62.3	4.78
4 X 4	448	1,953	69	509.9	7.34
8 X 8	1,792	8,691	658	4,414.0	6.70
16 X 16	7,168	36,501	3,958	25,636.3	6.48
32 X 32	28,672	149,702	16,491	107,855.7	6.54
64 X 64	114,688	604,566	63,180	437,452.4	6.92

3.2 SCADA Network: An Event Buffer Flooding Attack in DNP3 Controlled SCADA Systems

The Distributed Network Protocol v3.0 (DNP3) protocol is widely used in Supervisory control and data acquisition (SCADA) systems (particularly electrical power) as a means of communicating observed sensor state information back to a control center. Typical architectures using DNP3 have a two-level hierarchy, where a specialized data aggregator device receives observed state from devices within a local region, and the control center collects the aggregated state from the data aggregator. The DNP3 communication between control center and data aggregator is asynchronous with the DNP3 communication between data aggregator and relays; this leads to the possibility of completely filling a data aggregator's buffer of pending events, when a relay is compromised or spoofed and sends overly many (false) events to the data aggregator. We investigate the attack by implementing the attack using real SCADA system hardware and software, and show the existence and effectiveness of the attack. A Discrete-Time Markov Chain (DTMC) model is developed for understanding conditions under which the attack is successful and effective. The DTMC model is validated by a simulation model and data collected on real SCADA testbed.

3.2.1 Overview

SCADA systems are used to control and monitor critical infrastructure processes including electrical power, water and gas systems. As such, SCADA systems are critical to our daily lives. The United States is currently conducting a major upgrade of its electrical system, making the grid "smarter", but in doing so adding more vulnerabilities. We have seen the consequence when large areas lose power for an extended period of time [71], [72], [73]; the obvious threat is that attackers harm the grid infrastructure through largely electronic means.

We are interested in a vulnerability that arises within the communication infrastructure of the grid. The DNP3 is the most widely used SCADA network communication protocol in North America (approximately 75%) [74]. Designed to provide interoperability and as an open standard to device manufactures, DNP3 has no notion of security, and most DNP3 devices lack identity authentication, data encryption and access control. Although some enhanced versions of DNP3, such as DNP3 Secure Authentication [75] or DNPSec [76], have been developed but yet still under evaluation phase, the majority of DNP3-controlled devices in SCADA networks are currently working with little protection.

Most existing works on DNP3 security scrutinize potential security risks inherent in the DNP3 protocol specifications. A taxonomy of attacks across all layers of the DNP3 protocol has been summarized to show the extent of vulnerability of the protocol [77]. The attack we identified in this work is against the vendor implementation as well as the underlying communication structure. An attacker on the network can simply send many data events to a device that temporarily buffers SCADA data before they are retrieved by a control station. The attack fills an event buffer so as to prohibit the buffering of critical alerts from legitimate devices, negatively impacting the control station's situational awareness. The simple attack works effectively because (1) many commercial DNP3 data aggregators implement shared event buffers and (2) the communication between a control center and data aggregator is asynchronous with the communication between the data aggregator and relays. In addition, many proof-based DoS defense techniques, such as client-puzzle and public/private key, may not work appropriately because SCADA networks are generally resource-limited and have strong real-time requirements.

In a nutshell, the main contributions of this work are: (1) we identify a simple but very effective flooding attack in DNP3-controlled SCADA networks. We prove the existence and effectiveness of the attack using commercial power grid equipment in our lab; (2) we develop a DTMC model for analyzing the effectiveness of the attack as a function of various behavioral parameters. The analytical model has been validated by the data from the real testbed and a simulation model created in Möbius [78]; and (3) we suggest some countermeasures against this type of attacks.

3.2.2 Introduction of Distributed Network Protocol v3.0

The DNP3 protocol carries control and data communication among SCADA system components. It is a master-slave based protocol, where a master issues control commands to a slave and a slave collects data that is returned to the master. Typically a utility has a central control station for managing and monitoring its portion of the grid. The control station acts as a top-level DNP3 master, gathering data from substations, displaying the data in a human-readable formation, and making control decisions. A *data aggregator* located in a remote substation serves both as a DNP3 master to control and collect data from monitoring devices, **and** serves as a DNP3 slave to transmit (on demand) all of the data it has collected back to the control station. Figure 3.3 depicts the typical two-level architecture. DNP3 devices were widely used on serial links in old days, and many of them are still in use. Newer DNP3-controlled networks use TCP/IP-based connections where the DNP3 message is embedded as a payload of the underlying layer's packet. As a result, DNP3 can take advantage of Internet technology and to conduct economical data collection and control between widely separated devices. Our work focuses only on the DNP3 over TCP communication.

The data collected at the DNP3 slave is classified as being one of *binary data, analog data* or *counter data*. Binary data is used to monitor two-state devices, e.g. a circuit breaker is closed or tripped; analog data carry information like voltage and current on a power line. Counters are useful for reporting incremental values such as electricity usage in kilowatt hours. Data is transmitted to a master via two modes: polling and unsolicited response. In polling mode, a master periodically asks all the connected slaves for data, typically in a round robin fashion. Polling mode can be further divided into integrity polling and event polling. An integrity poll simply collects all static data with its present values. An event poll only collects DNP3 *events* that flag important changes, e.g. when a binary data value changes from an on to an off state or when an analog value changes by more than its configured threshold. In unsolicited response mode, a slave spontaneously sends DNP3 events to its master. A DNP3 master usually issues an integrity poll at start-up and then primarily uses event polling, with periodic refreshes with an integrity poll. The period of integrity polling (e.g. hourly) is generally much longer than the period of event polling (e.g. a few seconds).

A DNP3 slave that is configured to use *unsolicited response mode* may deliver data to a DNP3 master without being polled. This is useful for reporting state changes where a reaction is time-critical. The attack we have identified exploits the unsolicited response mode.

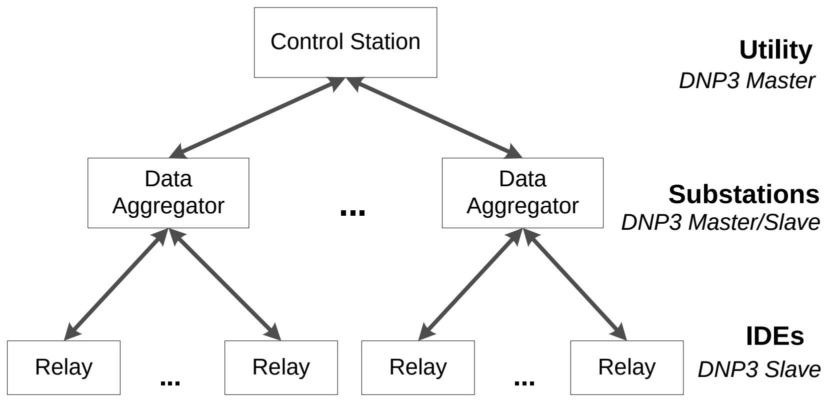


Figure 3.3: A Typical Two-Level Architecture of a DNP3-Controlled SCADA Network

3.2.3 Threat Model

The buffer flooding attacks assume the ability to access the substation network through some entry points, such as the utility's enterprise network or even the Internet. Although the flooding targets are the data aggregators within a substation, the attacks do not assume the ability to compromise a data aggregator. In order to flood the data aggregator's event buffer, the attackers must establish a connection with the data aggregator as a legitimated relay, which can be achieved by either spoofing a normal relay or compromising a victim relay.

No authentication is currently supported in DNP3 protocol to prevent the attackers from spoofing the relays. The attackers can suppress a normal relay by redirecting the victim relay's traffic to itself with techniques such as ARP spoofing and then spoof the victim relay to re-establishing a new connection with the data aggregator. The attackers can also act as a secret middle man between the victim relay and the data aggregator and aggressively replay unsolicited response events captured from the victim relay to exhaust the buffer resource.

The buffer flooding attack can also be launched from compromised relays. The reality is that the security of many commercial relays is only provided by having each relay require a password. Once the single password is captured, the relay is fully compromised. Unfortunately, bad password practices have always been observed in substation-level networks. Many operators do not change the default password for the sake of convenience. The magic words "otter tail" was definitely listed at the top of an attacker's dictionary, because it was used by a major relay manufacturer as a default password and surprisingly was observed to remain unchanged over many SCADA systems. Furthermore, most relays do not have a limit on the number of log in attempts, which could easily make a typical automated password cracker software effective.

3.2.4 The Vulnerability

A data aggregator serves as a DNP3 master to relays and as a DNP3 slave to the control station; one can think of it as having a master module and a slave module. The master module queries relays and stores received events into the slave module event memory. The data aggregator responds to queries from the control station by reading out portions of its slave module event memory. The vulnerability arises because the aggregator's polling of relays is performed asynchronously with the control station's queries to it. The slave memory is therefore a buffer, filled by responses from relays and emptied by a control station query.

Two types of event buffers are commonly used in commercial DNP3 slave devices: *sequence of event* and *most recent event*. The former simply stores all received data in the event buffer. Every new event occupies new buffer space; if the buffer is full then the event is discarded. This type of buffer is useful for various applications including grid state estimation and trend analysis. By contrast, a most recent event buffer reserves space for each individual data point that the aggregator might acquire. When an event arrives, all the buffer locations associated with data points it carries are overwritten, regardless of whether their current values have first been read out by a control station query.

The potential vulnerability of interest arises with sequence of event buffers, because it is feed by all slaves from which the data aggregator acquires data. The attack has a compromised DNP3 slave (or an attacker on the network successfully pretending to be a DNP3 slave) send so many unsolicited events that the buffer is filled, and events from uncompromised slaves are lost until the buffer is emptied by a query from the control station.

3.2.5 Experiments on Data Aggregator

Buffering Mechanism Experiments

The DNP3 specification describes the general guidelines on event buffer semantics and leaves the implementation to vendors [79]. The vendor's implementation is generally not publicly available. Therefore, in order to verify the existence of this buffer flooding attack, we need to first conduct experiments on a real data aggregator to understand its buffering mechanism.

The test data aggregator supports the three data types (binary, analog, and counters) mentioned in Section 3.2.2. Each data type has an independent buffer. To understand how each buffer works, we connected the device with relay A and relay B as two DNP3 slaves, and configured one host as a DNP3 master that plays the role of a control station. Initially, we set the size of every buffer to 5, and cleared all the buffers in the data aggregator by issuing sufficient integrity polls from the control station. Let A_i and $B_i (i = 1, 2, ...)$ be the unsolicited response event sent from relay A and relay B to the data aggregator respectively. Each event contains the same one data point with a different value. Figure 3.4 is the time sequence diagram showing the experimental results for all three data types.

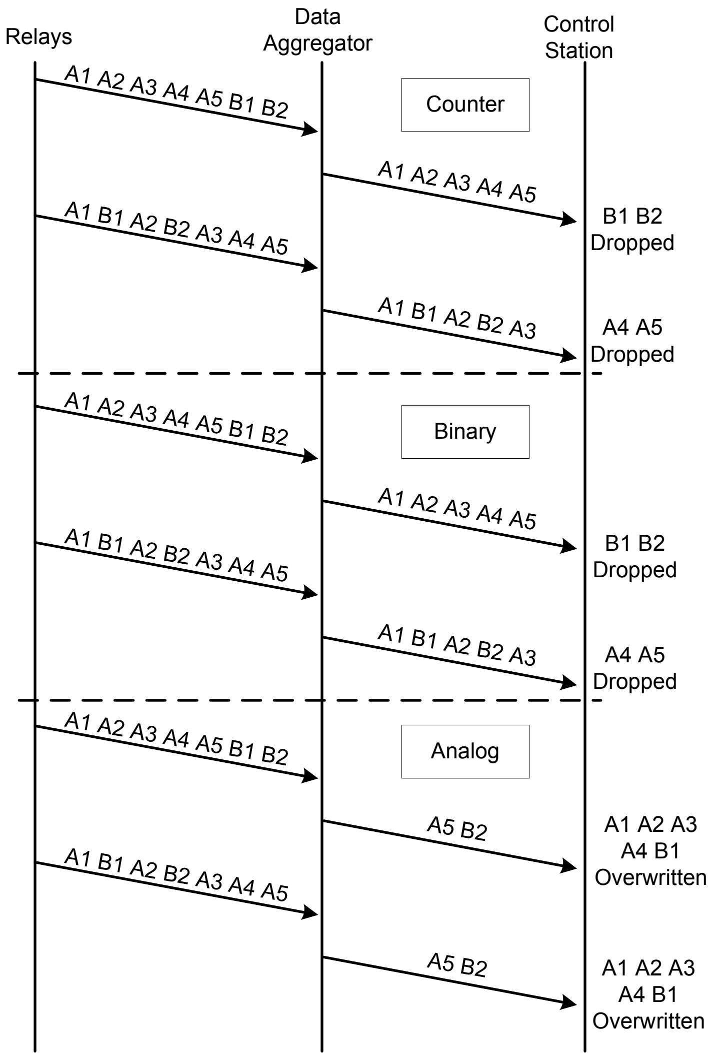


Figure 3.4: Time Sequence Diagram: Revealing Data Aggregator's Buffering Mechanism, Buffer Size = 5

The experimental results indicate the following:

- Buffers of all three data types have the first come first serve (FCFS) scheduling mechanism.

- The counter event and binary event buffers use "sequence of event" mode, and thus are vulnerable to the buffer flooding attack. Once the buffer was full, any incoming events were dropped, and the event buffer overflow indicator bit in the head of the DNP3 message was observed to be set to true.

- The analog event buffer uses "most recent event" mode; once the same data point was received more than once before being read out, its storage location was overwritten. Analog event buffers are immune to flooding, because an attacker's flooding affects only the buffer space allocated for the attacker's device.

Buffer Flooding Experiments

The next experiment launches buffer flooding attacks. The data aggregator serves as the DNP3 master to two relays, and as a DNP3 slave to a control station. The data aggregator polls the relays every 10 seconds. In addition the relays also send unsolicited response events to the data aggregator. Assume one relay is captured or spoofed by the attacker and it can generate many unsolicited response events and stop responding to polling requests. The unsolicited response event traffic from the attacker relay is injected with a constant inter-event time (which we will also refer to as "constant bit rate"). A normal relay always provides three events in response to a polling request, and also injects unsolicited response event traffic with an exponentially distributed inter-event time, with rate parameter three events per 10 seconds. All the traffic contains only counter events. Each event takes a value from a sequence number (continually incremented) to facilitate us identifying which events are lost (by looking for gaps in the reported sequence numbers). For these experiments we left the counter buffer at its default size of 50 events. The control station periodically polls the data aggregator every 10 seconds.

The attacker sending rate is chosen from 1 event/sec to 20 event/sec; each experiment generates 100,000 attack events. Figure 3.5 shows the fraction of dropped events for the normal relay's polling and unsolicited response events, under various attacker sending rates. Both types of events start to be lost when the attack rate is 5 event/s, because the buffer fills within one polling interval. The drop fraction increases as the attacker sending rate increases, and is nearly 80% at an attack rate of 20 event/sec. The sending rate can be no larger than network bandwidth / packet size. For example, with a 10 Mb/s Ethernet connection and 100-byte packet (which contains four DNP3 counter events), an attacker might send up to 50,000 counter events per second. From this we see that the buffer can be flooded and cause significant loss of real events under attacks whose rates are far smaller than the network line rate. Of course, the control station will realize that events have been lost (because of a status bit in the DNP3 response), and a burst of unusual unsolicited events could easily be noticed if a sniffer was watching traffic (which is actually very unusual in real DNP3 contexts). The flooding attack would be most effective if launched in coordination with other attacks (perhaps even physical attacks), denying the control station's situational awareness of the state of the substation.

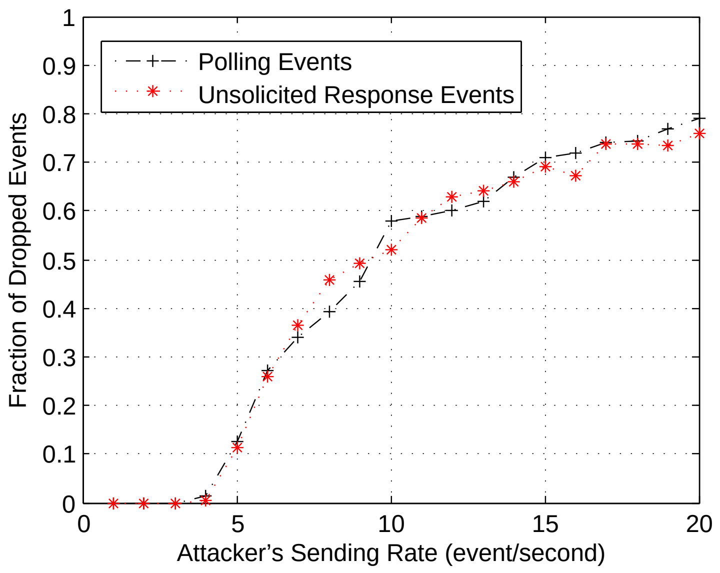


Figure 3.5: Fraction of Dropped Events from the Normal Relay on the Real Testbed

3.2.6 Modeling and Analysis

Analytical Model

We developed a DTMC analytical model for investigating this buffer flooding attack. The time-step is the control station polling interval length. The DTMC state is the buffer size at the instant a control station poll request arrives. Figure 3.6 depicts the data aggregator's event buffer as a queueing system. The system has three inputs: the unsolicited response events from the attacker relay, polling events and unsolicited response events from the normal relay. The shared buffer with finite size will drop any incoming events once it gets full. The output is triggered by control station's periodic polling request. Figure 3.7 illustrates event arrivals within a control station's polling interval. Here we assume that the control station and the data aggregator are configured to have the same polling interval.

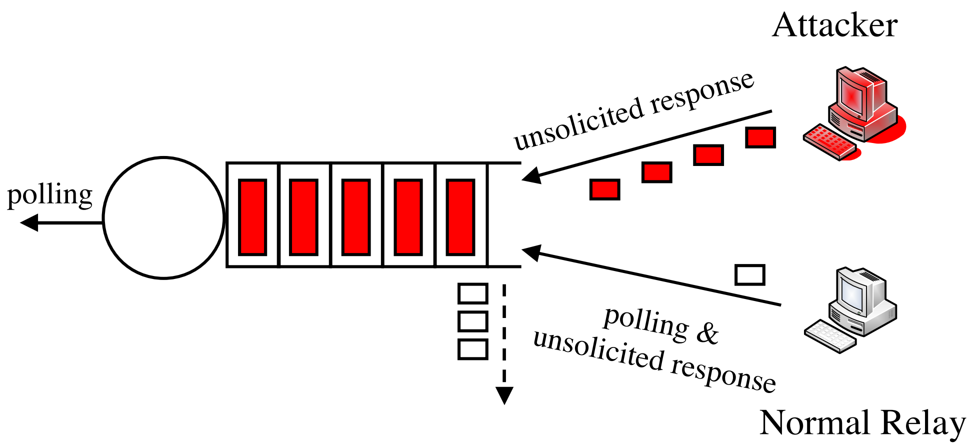
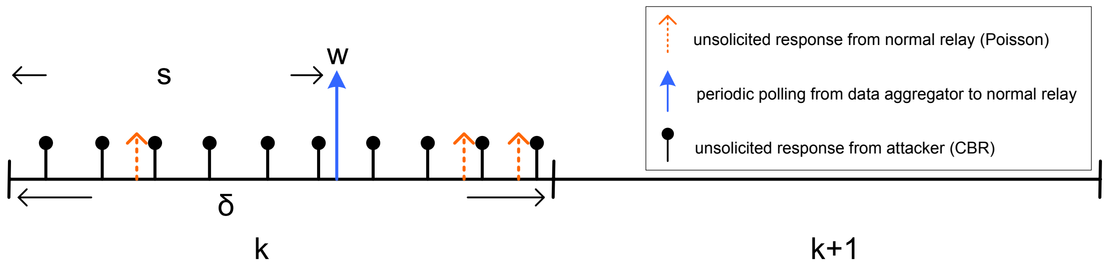


Figure 3.6: Queueing Diagram of the Data Aggregator's Event Buffer



unsolicited response from normal relay (Poisson)
periodic polling from data aggregator to normal relay
unsolicited response from attacker (CBR)
s
w
δ
k
k+1

Figure 3.7: Timing Diagram of Event Arrivals

The parameters of the analytical model are summarized as follows:

b	event buffer size
m	max number of events transmitting to control station from data aggregator per control station poll
δ	control station's constant polling interval
r	attacker's unsolicited response event sending rate, events arrive in constant bit rate
λ	mean arrival rate of unsolicited response events from normal relay, event arrival follows a Poisson distribution
w	number of events collected from normal relay per data aggregator's polling
S	normalized time within time-step at which bulk arrivals from normal relay poll arrive
k	time slot index, the time is slotted by the control station's polling interval
$Q(k)$	number of events in the buffer at the beginning of k^{th} time slot
$A(k)$	total number of arriving events during k^{th} time slot
$N(k)$	number of unsolicited response events from normal relay during k^{th} time slot
$D(k)$	number of departing events polled by the control station at the end of k^{th} time slot

The queueing system can be described by

$$Q(k+1) = [min(Q(k) + A(k), b) - D(k)]^{+} \tag{3.1}$$

The system can therefore be modeled as a DTMC, in which the time is discretized by the control station's polling interval. Let $Q(k)$ be the state of the Markov chain, $Q(k) \in 0, 1, 2...b - m$. The state transition probability is derived by

$$P(Q(k+1)=j|Q(k)=i) = \begin{cases} P(i+A(k) \leq m) & \text{if } j=0 \\ \\ Pr(i+A(k) \geq b) & \text{if } j=b-m \\ \\ Pr(i+A(k)-m=j) & \text{otherwise} \end{cases} \tag{3.2}$$

$$\begin{aligned} P(A(k) = r\delta + w + N(k)) &= P(N(k)=n) \\ &= \frac{(\lambda\delta)^n e^{-\lambda\delta}}{n!}, \text{ where } n \in 0,1,2... \end{aligned} \tag{3.3}$$

The DTMC is time-homogeneous. Let $\Pi = (\pi_0, \pi_1, ..., \pi_{b-m})$ denote the state occupancy probability vector in steady state, where π_i is probability that the DTMC is in state i in steady state.

$$\begin{cases} \sum_{i=0}^{b-m} \pi_i = 1 \\ \Pi = \Pi P \end{cases} \tag{3.4}$$

Let L_i be the total number of dropped events per time slot in state i, i.e. there are i events in the buffer at the beginning of the time slot.

$$L_i = ((A-(b-i))^+ \tag{3.5}$$

where the distribution of A is specified in Equation (3.3), and the dependence on k is removed from the notation as we are interested in the asymptotic behavior.

The average number of dropped events per time slot is computed as

$$E(L) = \sum_{i=0}^{b} \pi_i E[(A-(b-i))^+] \tag{3.6}$$

The ratio of expected dropped events of all types to expected events in a time slot is

$$\rho = \frac{E(L)}{E(A)} = \frac{E(L)}{r\delta + \lambda\delta + w} \tag{3.7}$$

a value which by Jensen's inequality [80] is a lower bound on the expected fraction of all events that are dropped.

The ρ bounds the overall fraction of dropped events (including attacker events); of more interest is the fraction of events dropped from the normal relay. Define T_f to be the time required (from beginning of a time slot) for the buffer to fill in a time slot.

$$P_i(T_f = t|S = s) =$$

$$\begin{cases} P(N_f = b - i - rt) & \text{if } 0 \le t < s \\ \sum_{j=0}^{w} P(N_f = b - i - \lfloor rs \rfloor - j) & \text{if } t = s \\ P(N_f = b - i - rt - w) & \text{if } s < t \le \delta \end{cases}$$

$$= \begin{cases} \frac{(\lambda t)^{b-rt-i} e^{-\lambda t}}{(b-rt-i)!} & \text{if } 0 \le t < s \\ \sum_{j=0}^{w} \frac{(\lambda s)^{b-\lfloor rs \rfloor - j - i} e^{-\lambda s}}{(b-\lfloor rs \rfloor - i - j)!} & \text{if } t = s \\ \frac{(\lambda t)^{b-rt-w-i} e^{-\lambda t}}{(b-rt-w-i)!} & \text{if } s < t \le \delta \end{cases} \tag{3.8}$$

where N_f is the random number of unsolicited response events from normal relay within T_f time; these events are not dropped. Time t is defined as $t \in \left\{\frac{b-i-z}{r}, \text{ where } z = 0, 1, 2...\right\} \cup \{s\}$ and $0 \le t \le \delta$.

The average number of dropped unsolicited response events and polling events from normal relay given T_f can be computed respectively as

$$E(L_i^{ur}|T_f = t, S = s) = E(L_i^{ur}|T_f = t) = (\delta - t)\lambda \tag{3.9}$$

$$E(L_i^{poll}|T_f = t, S = s) =$$

$$\begin{cases} w & \text{if } 0 \le t < s \\ \sum_{j=0}^{w} (w - j) P(N_f = b - i - \lfloor rs \rfloor - j) & \text{if } t = s \\ 0 & \text{if } s < t \le \delta \end{cases} \tag{3.10}$$

The average number of dropped unsolicited response events and polling events from normal relay within a time slot can be derived respectively:

$$E(L^{ur}) = \sum_{i=0}^{b-m} \pi_i \int_{s=0}^{\delta} f(s) \sum_t \bar{P}_i(T_f = t|S = s) E(L_i^{ur}|T_f = t, S = s) ds \tag{3.11}$$

$$E(L^{poll}) = \sum_{i=0}^{b-m} \pi_i \int_{s=0}^{\delta} f(s) \sum_t \bar{P}_i(T_f = t|S = s) E(L_i^{poll}|T_f = t, S = s) ds \tag{3.12}$$

where $P_i(T_f = t|S = s)$ is normalized by

$$\bar{P}_i(T_f = t|S = s) = \frac{P_i(T_f = t|S = s)}{\sum_t P_i(T_f = t|S = s)} \tag{3.13}$$

Thus, a lower bound on the expected fraction of lost normal unsolicited response events is

$$\rho^{ur} = \frac{E(L^{ur})}{\lambda\delta} \tag{3.14}$$

while the exact expected fraction of lost normal polling events is

$$\rho^{poll} = \frac{E(L^{poll})}{w} \tag{3.15}$$

The ρ^{poll} is exact because w is constant in this model.

Simulation Model

We also built a stochastic activity network (SAN) [81] simulation model with respect to the real testbed setup in Möbius v2.3.1. Möbius was first introduced in [82], with the goal of providing a flexible, extensible, and efficient framework for implementing algorithms to model and solve discrete-event systems. SAN, which is a stochastic extension to Petri net [83], is a high-level modeling formalism supported in Möbius. SANs consist of four primitive objects: places, activities, input gates, and output gates. Activities (thick vertical lines graphically) represent actions of the modeled system that take some specified amount of time to complete. Places (circles graphically) represent the state of the modeled system. Input gates (triangles graphically) are used to control the enabling of activities, and output gates (triangle with its flat side connected to an activity) are used to change the state of the system when an activity completes. In Möbius, we can also define reward variables that measure information about the modeled system.

Figure 3.8 shows the core design of the event buffer attack model. The place “EventBuffer” models the shared finite event buffer in a data aggregator. The event buffer queues events from three data sources, which are modeled as three activities: attacker relay’s constant bit rate traffic, normal relay’s Poisson arrival traffic and normal relay’s constant polling traffic, of which two are deterministic processes and one is an exponential process. The places “UR_Drop” and “Polling_Drop” are used to keep track of the number of dropped unsolicited response events and polling events from normal relay respectively. The fraction of dropped events are, for both types, set to be steady-state reward variables for simulation study.

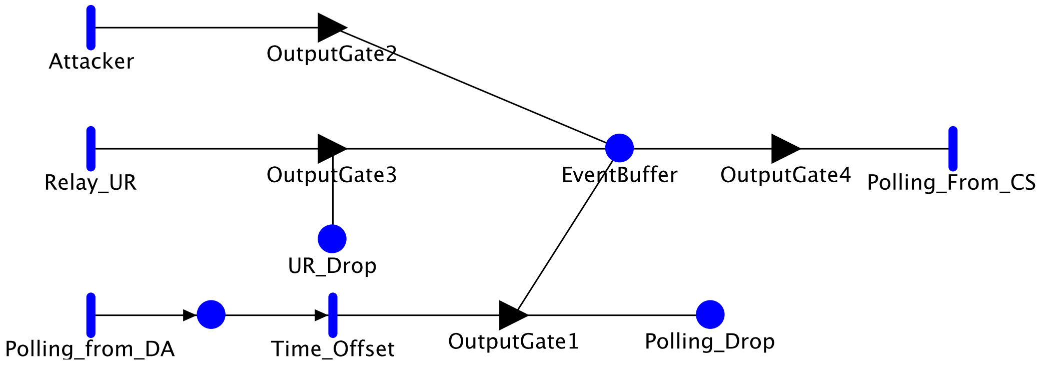


Figure 3.8: SAN Model of a DNP3-Controlled Data Aggregator’s Event Buffer in Möbius

Model Validation

Both real testbed data and the simulation model are used to validate the analytical model. All the parameters of the analytical model and the simulation model are taken from the real testbed: $b = 50, m = 50, \lambda = 0.3$ event/second, $w = 3$ event/second, $\delta = 10$ seconds. Recall that S is the fraction of time between successive control station polls that elapses before the data aggregator poll delivers a bulk arrival to the buffer. We empirically determined the probability distribution of S from testbed data based on 10,000 samples and plot the empirical CDF of S in Figure 3.9. It is clear that S can be modeled as a uniform distributed random variable between 0 to 10. With all the parameters in analytical model and simulation model aligned well with

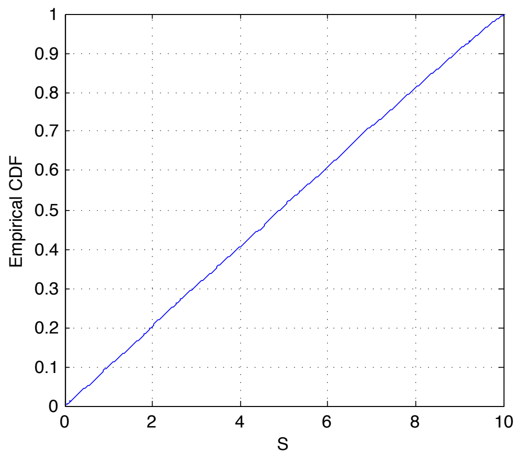


Figure 3.9: CDF of S: Time Difference between Control Station's Poll and Data Aggregator's Poll

real testbed setup, we vary the attacker sending rate from 1 event/second to 20 event/second with 1 event/second increment, and statistically compute the mean fraction of dropped events for both unsolicited response events and polling events from the normal relay. For all the reward variables in the Möbius model, the confidence level is set to 0.99 and relative confidence is set to 0.1, which means that results will not be satisfied until the confidence interval is within 10% of the mean estimate 99% of the time. For every experiment of the Möbius model, we conducted 10 independent runs with a different random seed. For each experiment, the minimum number of runs is 10,000 and maximum number of runs is 100,000. During all the experiments, the reward variables in the Möbius model are able to converge within the maximum number of runs. The degree of closeness of two sets of data are measured by the relative error. The relative error is defined as $\frac{|\hat{y}-y|}{y}$, where y is the baseline data and $\hat{y}$ are the data points to compare with the baseline data.

Figure 3.10 plots our estimates of the fraction of dropped events. The real data curve plots empirically observed fractions, the simulation model curve plots statistical estimates of the true observed fractions, and the analytic model plots the analytic upper bound on the true observed fractions. For the Möbius model, the results from the 10 independent runs have little variance and are extremely close to the testbed observations. The relative errors are also listed in Table 3.2. It can be seen that the analytic estimates for both unsolicited response and polling events match those of the simulation model with very small relative error. The analytical model and simulation model also match well with the real testbed data. Therefore, the analytical model is validated and can be used for quantifying how the attacker's sending rate blocks legitimate traffic on the test data aggregator; furthermore, the simulation model can provide an accurate and flexible environment for exploring the model's parameter space for investigating the buffer flooding attack.

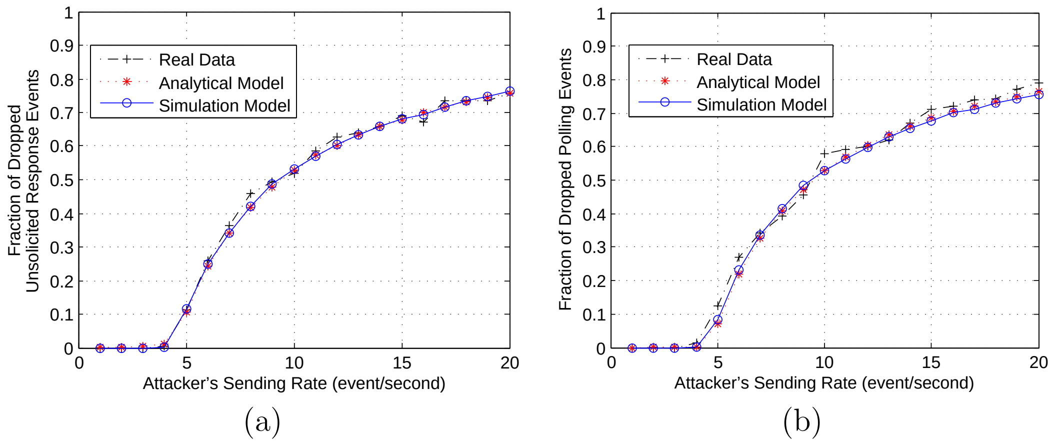


Figure 3.10: Estimated Fraction of Dropped (a) Unsolicited Response Events and (b) Polling Events from Normal Relay, Experimental Results from Real Testbed, Analytical Model and Simulation Model

Table 3.2: Relative Error of the Estimated Fraction of Dropped (a) Unsolicited Response Events and (b) Polling Events from the Normal Relay

$\hat{y}$	y	Relative Error of Drop Fraction			
		(a) UR Events		(b) Polling Events	
		mean	std	mean	std
Analytical	Real	0.0245	0.0252	0.0535	0.0998
Simulation	Real	0.0206	0.0221	0.0494	0.0754
Analytical	Simulation	0.0056	0.0081	0.0105	0.0133

We observed that the test data aggregator simply sends everything inside the buffer in response to a control station's poll. If the number of events in the buffer is large, they will be fragmented into multiple DNP3 data packets that are resembled at the destination. Therefore, the real testbed has the constraint that $b = m$ and the corresponding DTMC model has only one state. However, it is recommended that in the second-level DNP3 slave, such as the data aggregator in this case, the maximum number of items returned per poll be configurable in order to avoid overwhelming the network link [79]. Since the feature has been supported in many commercial data aggregators as well as by the Triangle Microworks' DNP3 Test Harness [84], it is necessary to evaluate whether the analytical model correctly captures the attacker's effect on the data aggregator when $b > m$. The simulation model is used as a baseline to validate the analytical model. Let $m = 30$ and $b = 50$, now the DTMC model has 21 states. While keeping the rest parameters with the same values, we ran the same set of experiments on both the analytical model and the simulation model, and plot the unsolicited response events and polling events drop fractions in Figure 3.11(a) and 3.11(b) respectively. The drop fractions derived from the Möbius model are again the average of 10 independent runs with little variance. The relative error of the unsolicited response event drop fraction has mean of 0.0080 with standard deviation 0.0080, and the relative error of the polling event drop fraction has mean of 0.0066 with standard deviation of 0.0050. The extremely small relative error indicates that the DTMC model can efficiently compute the drop fraction of legitimate traffic as accurate as the simulation model.

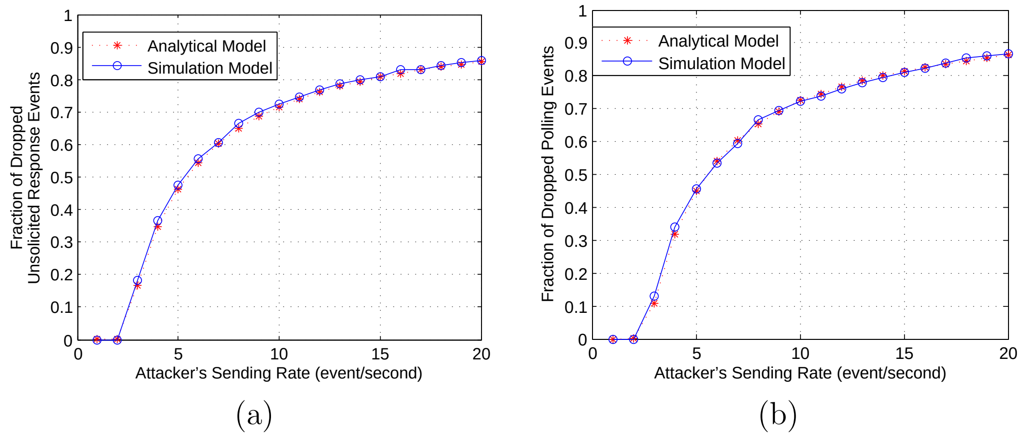


Figure 3.11: Estimated Fraction of Dropped (a) Unsolicited Response Events and (b) Polling Events from Normal Relay, with b = 50, m = 30

Model Analysis

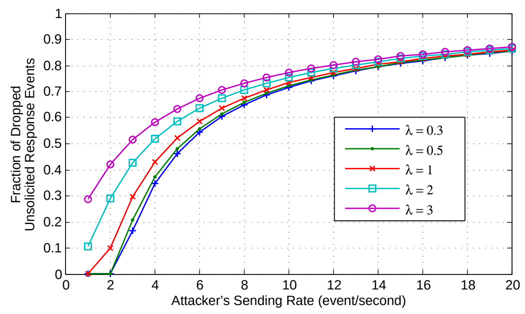
We then explore the impact on the drop fraction of key model parameters λ, w, S and m. The idea is to vary only one selected parameter for every set of experiments, and again measure the relationship between the attack sending rate and the fraction of dropped events. The baseline parameters are chosen as follows: $b = 50$, $m = 30$, $\delta = 10$, $\lambda = 0.3$, $w = 3$, and S is uniformly distributed between 0 and 10. Figure 3.12 displays the plots of drop fractions versus attacking rate for every selected parameter.

The λ is the mean arrival rate of unsolicited response events from normal relay. Figure 3.12 (a1) and (a2) show that all the lines with different λ values tend to converge as the attacker sending rate increases. Once the attacker sending rate is greater than 10 event per second, which is easy to achieve, λ has a small impact on both types of dropped events.

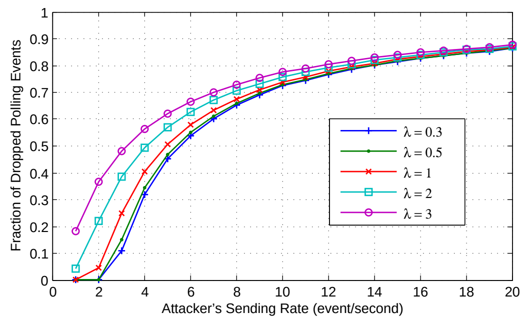
The w is the number of events collected from the normal relay in response to a data aggregator's poll. Similar to the impact of λ, the lines tend to converge as the attacking rate increases and thus w also has a small impact on both types of event drop fractions, especially on the unsolicited response events.

The S is the time offset between neighboring control station's poll and data aggregator's poll. The variation we noted earlier was taken over successive experiments. Under the assumption that polling intervals of both the control station and the data aggregator are constant, in any given experiment S will be constant. We vary it here to see what impact a given constant S may have. It has little impact on the unsolicited response events. Within a polling interval, the number of attacking events is much more than the number of the normal relay's polling events, therefore when the polling events arrive has minimum impact on the drop fraction of the unsolicited response events from the normal relay. However, the value of S greatly affects the fraction of polling events that are dropped. If the polling events arrive right after the previous control station's poll, there is always space in the buffer to hold them. On the other hand, if the polling events arrive just before the next control station's poll, the buffer has almost surely been filled up by the attacking events.

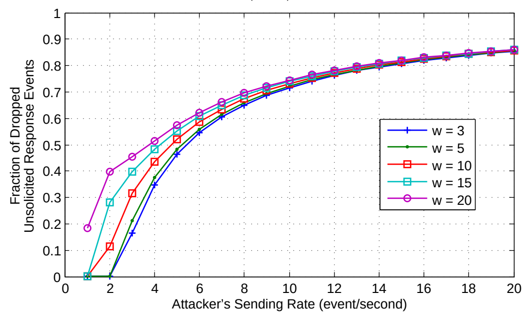
The S varies in real scenarios because of the uncontrollable variance in the clocks that DNP3 masters use for issuing periodic polling requests. One enhancement could be developing rules on the data aggregator to generate polling requests to all the connected relays right after a control stations's poll (use multicast if supported), the polling events from normal relay can possibly enter the data aggregator's buffer before the attacking events overflow the



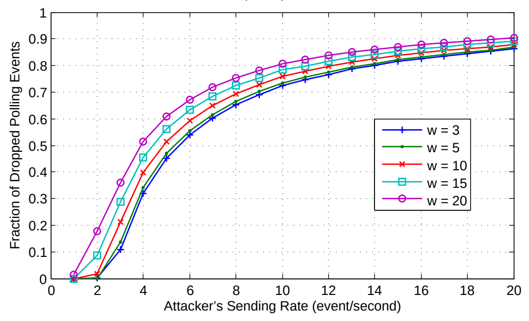
(a1)



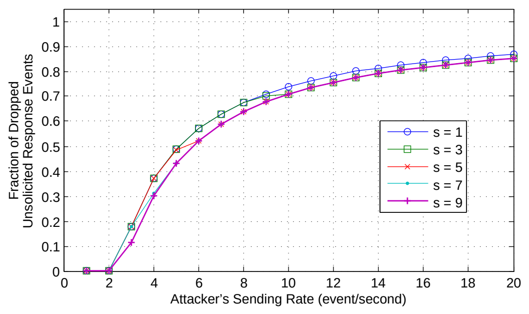
(a2)



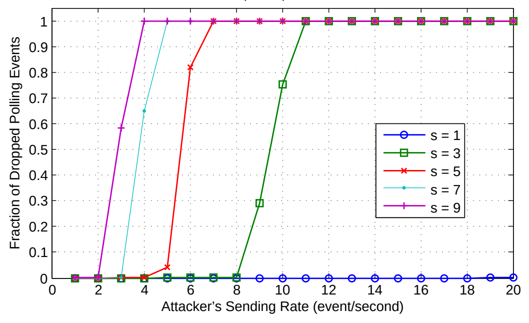
(b1)



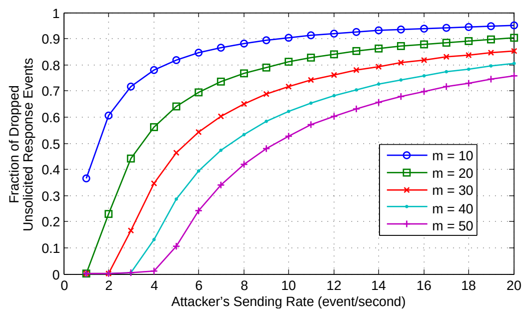
(b2)



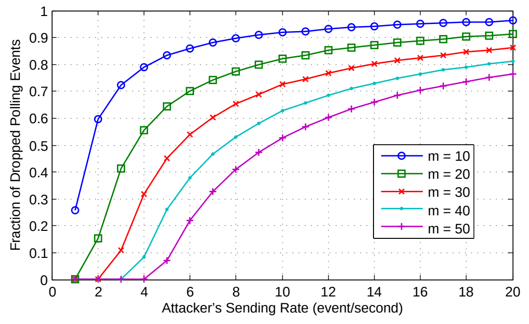
(c1)



(c2)



(d1)



(d2)

Figure 3.12: Model Analysis: Fraction of Dropped Unsolicited Response/Polling Events vs. Attacking Sending Rate with Varying (a) λ (b) w (c) S (d) m

buffer and minimize the fraction of dropped packets.

The m is the maximum number of events transmitted to control station in response to a control station poll. Larger m essentially means a larger service rate, and results in more available buffer space at the beginning of each time slot. Therefore, the fractions of dropped events of both types are reduced as shown in Figure 3.12 (d1) and (d2). However, increasing m is generally not a good solution, because the control station actually wastes even more resources including processing power and communication bandwidth to serve the attacking events. As a result, the attacker's impact effectively propagates to the communication between the control station and the data aggregator.

3.2.7 Countermeasures

The key reason that the buffer flooding attack works is that buffer space is shared among sources, and use of the buffer follows a first-come-first-serve rule. The fraction of service that a data flow receives is always proportional to its input rate with FCFS policy when the buffer is congested. Therefore a high load flow like those of the attacker relay's unsolicited response events, can occupy most of the bandwidth, and influence the low load flows, such as the unsolicited response events and polling events from the normal relay. Another class of scheduling policies is designed with the goal of providing fair queueing [44], such as round robin (RR), weighted round robin (WRR) [45], weighted fair queueing [85] and virtual clock [46]. Applied in this context, the fair queueing scheduling policies aim to ensure that every input flow has reserved buffer space, and the additional buffer space will be equally distributed among flows that need more. Therefore, a reasonable defense against the buffer flooding attack is to allocate space in a shared event buffer according to a fair queueing policy. Round robin based scheduling could be a good choice due to the low time complexity O(1) and the low implementation cost [47].

As specified in the DNP3 protocol standard, every DNP3 slave's application response header contains a two-octet internal indications (IIN) field [79]. The bits in these two octets indicate certain states and error conditions within the slave. The third bit of the second octet indicates that an event buffer overflow condition exists in the DNP3 slave and at least one unconfirmed event was lost because the event buffers did not have enough room to store the information. The overflow condition continues to hold until the slave has available event buffer. It provides a means for the DNP3 master to detect whenever a buffer overflow occurs, however, the action recommended by the DNP3 user group, and in fact many vendors implemented in their products, is to issue an integrity poll in order to re-establish the current state of all data in the slave device [86]. However, the action is not sufficient to protect the device from the flooding attack discussed in this work. The integrity poll is passively issued upon receiving a response from DNP3 slave, and therefore it can only delay the time that next buffer overflow occurs. In addition, an integrity poll simply asks for all the static data rather than changed events, therefore generating many integrity polls could potentially overwhelm the network link between data aggregator and control station, and as a result, unintentionally wasting bandwidth and processing resources. One improvement could be applying rule-based policies to limit or filter the attacking traffic. For example, if relay A causes three successive sets of the event buffer overflow indication bit, the data aggregator will filter any data traffic whose DNP3 source address is of relay A. The rule will continue to take effect if the upcoming traffic from relay A exceeds a configured threshold. In addition, if the data aggregator's scheduling algorithm involves computation of weight, such as weighted round robin and weighted fair queueing, we could associate the event buffer overflow indication with an extremely small weight, and therefore minimizes amount of the attacking traffic entering the event buffer.

Lack of authentication in the DNP3 protocol enables attackers to spoof normal relays. Researchers are actively working on various forms of crypto-based solutions to establish strong authentication in the SCADA environment, such as studying the practicality of various forms of key management [87], examining the practicality of using puzzle-based identification techniques to prevent DOS attack in a large-scale network [88], or evaluating enhanced DNP3 protocols like DNP3 Secure Authentication [75] or DNPSec [76].

3.2.8 Related Work

DNP3 was designed without concern for security because SCADA networks were physically isolated with other networks at that time. However, with the growing of smart grid technologies, dependences of critical infrastructures on interconnected physical and cyber-based control systems grow, and so do vulnerabilities. The buffer flooding attack discussed in this work targets data aggregators, and results in the loss of awareness in the control center. Detailed attacks against DNP3 specifications across all three layers were also proposed and classified into 28 generic attacks and 91 specific instances [77]. The impact of those attacks could result in loss of confidentiality, loss of awareness and even loss of control. A survey of SCADA-related attacks was conducted in [89], covering techniques of attack trees, fault trees, and risk analysis specific to critical infrastructures. The buffer flooding attack overwhelms the limited buffer resources in data aggregators, and thus it belongs to the class of DoS attacks. DoS attack and defense mechanisms in the Internet have been studied and classified in [90]. The real-time constraints and limited resources of the SCADA network makes the defense of such DoS attack even hard. Much research has also been done on realistic cyber attack vectors [91], [92], [93] and security gaps [94], [95], [96] specific to SCADA networks.

Investigation of attack vectors and security gaps will result in remediation techniques that can provide protection. Research has been done on countermeasures specific to DNP3 attacks, including data set security [97], SCADA-specific intrusion detection/prevention systems with sophisticated DNP3 rules [98], [99], and encapsulating DNP3 in another secure protocol such as SSL/TLS or IPSec [100]. Design guidances for authentication protocols based on extensive studies of the DNP3 Secure Authentication was proposed in [101]. Clearly, the smart grid technologies will bring much more attacks to the existing and new SCADA networks, therefore both independent researchers and government officials have formed workgroups [102] to investigate and offer their advice [103].

3.3 Demand Response: Evaluation of Network Design and Hierarchical Transactive Control Mechanisms

Smart-grid technologies seek to enhance the electrical grid's efficiency by introducing bi-directional communication and dynamic adaptive control between energy suppliers and consumers. We develop a large-scale network simulation model for evaluating such a hierarchical transactive control system that is part of our work on the Pacific Northwest Smart Grid Demonstration Project. The transactive control system communicates local supply conditions using *incentive* signals and load adjustment responses using *feedback* signals in a distributed fashion in order to match the consumer-desired load to the utility-desired supply scenario. We study the system protocol and a dynamic control mechanism that is implied by the design under a reasonable collection of models that capture load variation, stochastic signal losses, consumer fatigue to demand response and certain "stickiness" criterion on the control signals that arise out of physical constraints. Our results indicate that the control mechanism can perform adequately in adjusting the aggregate supply-demand mismatch, and is robust to steady transactive signal losses.

3.3.1 Transactive Control for Demand Response Management

With the advent of the smart grid, the infrastructure for energy supply generation and transmission is experiencing a transition from the current centralized system to a decentralized one. The ability to access and act upon real-time information on supply availability and prices supported by the demand offers unique opportunities to improve the overall efficiency of the grid in terms of both long-term supply-demand management as well as near-term dispatching of diverse generation facilities to meet current demand. The responsiveness and flexibility envisioned for the smart grid provides additional advantages in facing the significant new challenges ([104, 105, 106]) of integrating distributed and intermittent generation capability such as small-scale generators and renewable energy sources (wind, solar, etc.) at a scale unmatched by current grid technology. This is critical to renewable energy technologies playing a more prominent important role in the portfolio mix of electricity generation.

We have embarked on a comprehensive smart grid demonstration project in the Pacific Northwest involving 60,000 customers from 12 utilities across five states, covering the end-to-end electrical system from generation to consumption, built around a substantial infrastructure of deployed smart meters: [107] describes our effort in greater detail. This unique smart grid demonstration spans the electrical system from generation to transmission and distribution to ultimately end-user load. The demonstration will test more than 20 types of responsive smart grid assets applied to six specific regional and utility-level operational objectives at 14 unique distribution sites operated by 12 utilities. All end-use classes will be well represented including residential, commercial, and industrial customers.

In transactive control, responsive demand assets bid into and become controlled by a single, shared, price-like incentive signal called the Transactive Incentive Signal (TIS) in our project. The TIS may be, in turn, influenced by many local and regional operational objectives of the electric power grid. Responsive assets include at the household level appliances such as thermostats, water heaters and clothes dryers, and at the commercial level assets with higher demand characteristics such as HVAC systems, distributed diesel generators, a gas turbine and municipal water pumps. Appropriately formulated control algorithms automatically generate bids and offers from these responsive demand assets based on user preferences and the degree to which the assets preferences (e.g., room temperature or water level) had been satisfied. The returned load estimate signal is called the Transactive Feedback Signal (TFS) in our system. Energy management systems, both commercially available and specially engineered devices and collectively called *smart meters*, generate and communicate back the asset's response as well as take local actions upon the value signals and bids by adapting demand accordingly. This diversity of system components will be coordinated and controlled via the Internet from the project control center at the Pacific Northwest National Laboratory. The transactive signaling is supported by a communications backbone built in partnership with IBM using their Internet-Scale Control System (iCS).

We develop a network simulation model that captures the relevant aspects of the system protocol and illustrates whether, and how well, one of the central goals of this demonstration project is being met: how can transactive control be used to manage the distribution problem of peak demand,

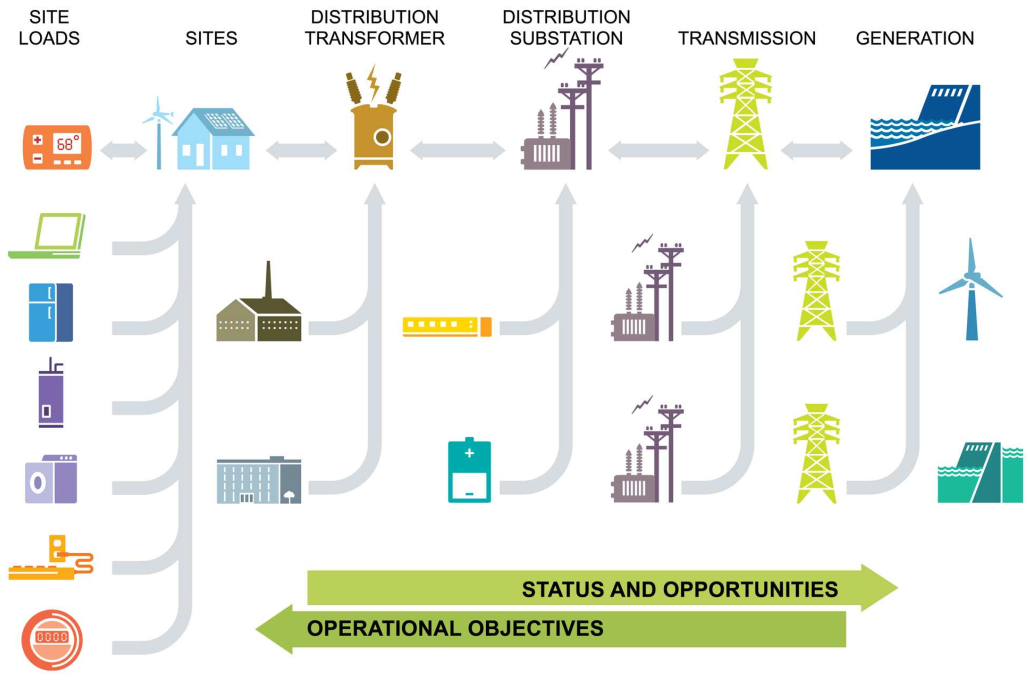


Figure 3.13: Hierarchical Architecture of the Smart Grid

and improve system efficiency and reliability at such a large scale. Figure 3.13 illustrates a schematic of the proposed hierarchical structure. In this figure, the incentive value signals flow downstream toward the left, while the corresponding demand signals flow upstream toward the right. Note that responsive assets (respectively, value signal calculations) do not occur only at the extreme downstream (respectively, upstream) locations in the figure. Indeed, just as every node in the hierarchy can interject the degree of meeting its own operational objectives, responsive assets can reside quite far upstream, even at the transmission nodes in form of flow control devices, resource dispatch practices, and voltage control devices. At each node in this hierarchy a demand signal is aggregated from its children nodes, and a price (or value signal) is calculated using information obtained from its parent nodes.

Our primary measure of performance of a control algorithm measures the aggregate (weighed) mismatch between the utility's desired level of load and the consumer's aggregate desired level. We study a distribution network with a top-down tree structure where multiple end-users ultimately draw their power from a single distributor. Our model attempts to capture a number of relevant features of real distribution systems and electricity consumption:

- The characteristics of the communication protocol used by the demonstration are faithfully modeled.
- We provide a simplified version of the control algorithm for all the various types of nodes in the model.
- The simplest control algorithm sends no information through the signals and models the *no-smart-grid* case.
- Demand response is often observed to be a nonlinear function of incentives. We study both a simple linear model as well as a nonlinear model.
- Users express fatigue in responding to repeated signals to reduce demand. We explore a specific fatigue-model that changes the forecast demand of the user in periods subsequent to ones where load is shed in response to incentive signals.
- Electrical system dynamics place a heavy penalty on the system when loads exhibit high volatility. We study control algorithms under a setting where the transactive signals for a specific future time period are required to be within pre-specified bounds in subsequent signals.
- The local area network connecting households to the smart grid is expected to have some reliability challenges, and our model captures signal losses and studies the impact of such losses.

Combinations of these various factors complicate the control problem of managing the utility's demand load. We study each addition in turn and illustrate the value of or limitation placed by each. We emphasize here that this list is by no means complete, examples of which include end-users who draw from two or more distributors, end-users who collude with each other etc.

We design and implement the transactive control algorithm model and the underlying timing and network protocols with all the previously identified features in the S3FNet discrete-event network simulator based on the second generation of the scalable simulation framework (SSF) [1], which enables us to evaluate various control models in large scale up to 100,000+ transactive nodes and links. The simulation results show that with reasonable settings of the fatigue experienced by consumers and the stickiness of the transactive signals, the elasticity control models are able to properly adjust the aggregated consumer-desired load to the utility-desired load. In addition, the control algorithm is quite robust to steady signal loss.

3.3.2 Transactive Control System Design and Modeling

System Overview

Figure 3.14 depicts a hierarchical two-level power grid control system. At the consumer level, thousands of smart meters form the advanced metering infrastructure (AMI) network, in which various wireless technologies are deployed. The meters can be arranged to form tree, star or mesh topologies depending on their geographical distribution. We primarily investigate the tree structure for this demonstration project and plan to extend the work to other topologies in the future. The meter data are aggregated at the AMI gateways, which communicate directly to the substations. Each substation monitors the load of its portion of the grid in real time, and communicate with the utility head-end to ensure the situation awareness of the grid. The utility head-end usually resides in the enterprise network and is equipped with sufficient computing resources and storage. The head-end overviews the load of its portion of the grid, perform computational-intensive tasks such as demand forecast, state estimation and pricing control, and making system-wide management decisions.

One key feature of the transactive control system is the bidirectional control communication, and the smart meters at the consumer end are capable to make local decisions. A transactive nodes is defined as a physical point within an electrical connectivity map of the system. The utility, substations and consumers are all considered as the transactive nodes. Each transactive node can send and receive transactive signals. The consumers receive incentive signals from the utility containing the cost information (e.g. a lower price for future load reduction), and generate a corresponding feedback signal containing the demand change information to be sent back upstream. In the substation-level nodes, components of the incentive and feedback signals must be additively combined into one incentive and one feedback signal.

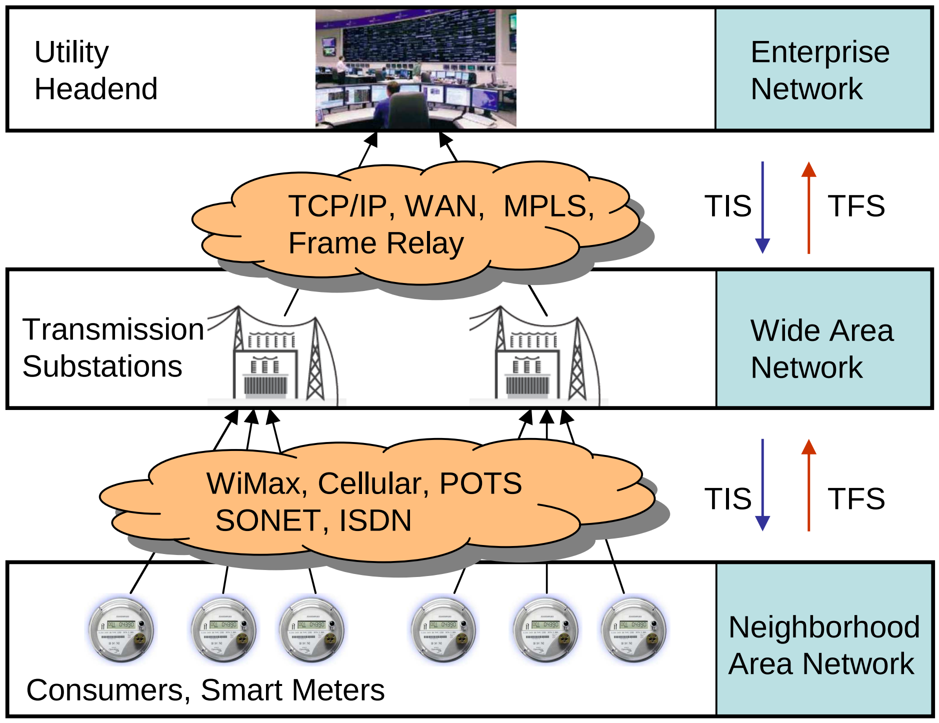


Figure 3.14: Two-Level System Architecture

The substations are assumed not to modify the incentive and feedback signals based on the local responsive assets in this work and we plan to integrate sophisticated local decision making algorithms at substations in the future.

Transactive Node Control Algorithm

Notations

A summary for all the notations used in the proposed models is provided as follows, to aid in reference.

N	Total number of transactive nodes at bottom level
k	Index of time interval
i	Index of consumer
D^u	Utility-desired load
D^c	Total consumer-desired load
ϵ^c	Elasticity of the individual consumer

ϵ^u	Integrated elasticity at the utility/substations
C	Cost signal (in TIS) generated at the utility
ΔC	Change of cost at the utility
C_0	Initial cost set by the utility
c	Cost received at the individual consumer
Δc	Change of cost observed at the consumer
Δd	Change of demand (in TFS) generated at the consumer in response to the received TIS from the utility
Δd^{max}	Maximum load change at the individual consumer
ΔD^{max}	Maximum load change at the utility

Assumptions

A1. The same TIS is passed to all the consumers from the utility, i.e. $c_i(k) = C(k)$.

A2. At each time interval, the consumer's elasticity is subjected to a small disturbance with constant variance. We assume

$$\epsilon_i^c(k) = \epsilon_i^c(0)\,(1 + z(k))\,,\ \text{where } z(k) \sim N(0, \sigma^2) \tag{3.16}$$

A3. The utility and the substations update the integrated elasticity using the load adjustment function F, which is the same as that at the consumer level f, i.e.

$$\begin{aligned} F(\epsilon^u(k), \Delta C(k)) &= f(\epsilon^u(k), \Delta C(k)) \\ &= \textstyle\sum_{i=1}^{N} f(\epsilon_i^c(k), \Delta c_i(k)) \end{aligned} \tag{3.17}$$

and the integrated elasticity for deciding the next TIS is updated by

$$\epsilon^u(k) = f^{-1}\left(\sum_{i=1}^{N} f(\epsilon_i^c(k), \Delta c_i(k)), \Delta C(k)\right) \tag{3.18}$$

A4. The upper limits for the load reducible of all consumers are approximately equal, i.e.

$$\Delta d_i^{max} = \frac{1}{N}\Delta D^{max} \tag{3.19}$$

Linear Load Adjustment Model

The linear load adjustment function is defined as

$$f(\epsilon^c, \Delta c) = -\epsilon^c \Delta c \tag{3.20}$$

At time interval k, the consumers receive the TIS $C(k)$, and the TFS sent to substations is

$$\Delta d_i(k) = -\epsilon_i^c(k)[C(k) - C_0] \tag{3.21}$$

After collecting all the available TFS for the time interval k, the utility/substations update $\Delta D(k) = \sum_{i=1}^{N} \Delta d_i(k)$, and then calculates the integrated elasticity using Equation (3.18),

$$\begin{aligned} \epsilon^u(k) &= f^{-1}\left(\sum_{i=1}^{N} \Delta d_i(k), \Delta C(k)\right) \\ &= -\frac{\Delta D(k)}{\Delta C(k)} = -\frac{D^u(k) - D^c(k)}{C(k) - C_0} \end{aligned} \tag{3.22}$$

After receiving the updated information on consumer load profile, the utility calculates the TIS sent to consumers in the next time interval as

$$C(k+1) = -\frac{D^u(k+1) - D^c(k+1)}{\epsilon^u(k)} + C_0 \tag{3.23}$$

Nonlinear Load Adjustment Model

It is more realistic to consider the situation that the consumers are more reluctant to reduce load with the continuous same amount of price change. Such demand response behavior leads to concavity of demand response function [108]. Consider the load adjustment function defined by

$$\begin{aligned} f(\epsilon_i^c(k), \Delta c_i(k)) = &-I\{\Delta c_i(k) > 0\} g(\epsilon_i^c(k), \Delta c_i(k)) \\ &+I\{\Delta c_i(k) < 0\} g(\epsilon_i^c(k), \Delta c_i(k)) \end{aligned} \tag{3.24}$$

where

$$g(\epsilon_i^c(k), \Delta c_i(k)) = \Delta d_i^{max}(k) - \frac{1}{\epsilon_i^c(k)|\Delta c_i(k)| + \dfrac{1}{\Delta d_i^{max}(k)}} \tag{3.25}$$

and $I\{\cdot\}$ is the indicator function. Here we assume that $\Delta D^{max}(k)$ equals $\alpha D^c(k)$ with $\alpha \in (0,1)$ representing the overall maximum percentage load reducible. Thus by A4, we have

$$\Delta d_i^{max}(k) = \frac{1}{N}\alpha D^c(k) \tag{3.26}$$

The TIS/TFS update procedure is the same as that described for the linear case.

Sticky Price

In the transactive control system, TIS/TFS can only offer information up to a certain limited amount of time in the future, which bounds the change in both TIS and TFS due to the relative short-period view of the entire system. In addition, there is always a lower bound of the price that the utility is willing to offer. The price is highly dependent on the time-varying spot market price. In the current model, we simplified these constraints by adding an upper bound for $\Delta C(k)$ as ΔC^{max}.

Fatigue Model

In reality, consumers more or less tend to be less sensitive to price change in consecutive several time intervals. The load reduced earlier will eventually shift to a later period time and vice versa. For example, a customer stopped charging the electrical vehicle for avoiding peak hour electricity cost will have to charge the car at some point. Therefore, a consumer's elasticity is negatively correlated across the time horizon. Such response fatigue phenomenon has been investigated in [109], [110]. We integrate a consumer fatigue model into the elasticity model by adjusting the total consumer-desired load curve according to previous TFS at each time interval as

$$D'^c(k) = D^c(k) + \beta \sum_{i=1}^{N} \Delta d_i(k-1) + \beta^2 \sum_{i=1}^{N} \Delta d_i(k-2) \tag{3.27}$$

with the decaying factor $\beta \in (0,1)$.

Distributed Timing Protocol

Time is an important component of transactive control. Diverse smart grid resources can have different time characteristics such as the time limitation due to operational control of physical equipment or market operation, the duty cycle of a physical asset, and the time constraints of a communication network in part of the system. Therefore, it is essential to ensure applications with different time constraints to be interoperable under the transactive control framework.

Figure 3.15 shows the proposed timing protocol of the transactive control system for the demonstration project. It is flexible to choose variable update time intervals and the demonstration project has elected to use a constant update time interval 5 minutes in length. Each time interval is further divided into three sections, and each section performs the corresponding task based on the latest received transactive signals. This highly distributed protocol enables the intelligence to be shifted from the centralized head-end down to thousands of smart devices in the grid. Since the TIS and TFS that interact with each other along a particular transactive control pathway are time-aware and time-sensitive, adopting event-driven and unsynchronized data communication is one of the architectural decisions. Whenever a new event arrives, the transactive node immediately processes the event and if necessary propagates the event to the destination for keeping the freshness of the local transactive signals and minimizing the end-to-end delay, which ensures the efficiency of the control algorithm running on top of the timing protocol.

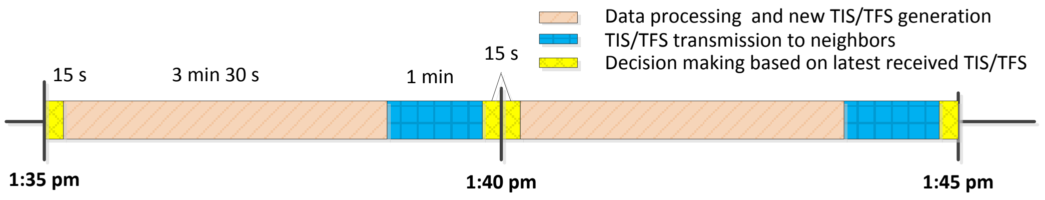

Data processing and new TIS/TFS generation
TIS/TFS transmission to neighbors
Decision making based on latest received TIS/TFS
15 s
3 min 30 s
1 min
15 s
1:35 pm
1:40 pm
1:45 pm

Figure 3.15: Distributed Timing Protocol

3.3.3 Simulation Results

The simulation experiments are designed to investigate the various control algorithms of the transactive nodes in a small town level network. All the test cases are constructed with the two-level tree topology, in which a utility node is connected to 10 substations with 1 Mb/s bandwidth and each substation is connected to 100 consumers with 10 Kb/s bandwidth. The initial elasticity, $\epsilon_i^c(0)$, for all the consumers are uniformly distributed between [0.1, 0.4]. All the experiments are running for 24 hours in simulation time. The utility-desired load represents a typical winter weekday load collected from PNW Demonstration Program [70].

To qualitatively evaluate all the proposed models, we define the goodness of measurement ϕ by aggregating the difference between the utility-desired load and total realized load from consumers through the transactive control system, and weighted by the percentage of utility-desired load for each time interval, i.e.

$$\phi = 1 - \frac{\lambda}{\lambda'} \tag{3.28}$$

where

$$\lambda = \textstyle\sum_k [(D^u(k) - D^c(k) - \sum_{i=1}^N d_i(k))^2 \frac{D^u(k)}{D^u_m(k)}] \tag{3.29}$$

$$\lambda' = \textstyle\sum_k [(D^u(k) - D^c(k))^2 \frac{D^u(k)}{\tilde{D}^u(k)}] \tag{3.30}$$

where $\tilde{D}^u(k) = \sum_k D^u(k)$.

Linear Load Adjustment Model

Figure 3.16 shows the load adjustment with the linear load adjustment model. The realized load includes the total demand change reported through TFS by all the consumers. Provided that no signal loss occurs, the consumer-desired load is nearly perfectly adjusted to the utility-desired load for every time interval regardless of peak hours or non-peak hours ($\phi = 0.998$). This is because the same cost change information is passed down to all the consumers and the same linear load adjustment function is applied at both the utility and the consumers. The elasticity at the utility is ideally equivalent to the sum of the elasticity of all the consumers, and thus can be accurately derived by Equation (3.18).

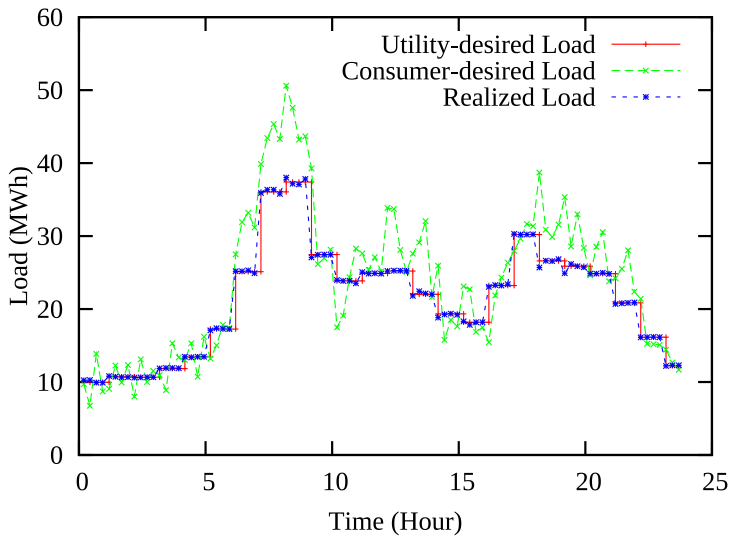


Figure 3.16: Linear Elasticity Model - Utility Load Adjustment

Nonlinear Load Adjustment Model

We then run the same test case with the basic nonlinear load adjustment model with various ΔD^{max}. Figure 3.17 shows the results with $\alpha = 10\%$, 20%, and 30% respectively. Less load adjustment is observed with the same price for the nonlinear case than the linear case. This is because the nonlinear model uses a strictly concave load adjustment function. The figure also briefly shows the impact of the sticky TFS. As ΔD^{max} increases, ϕ significantly improves (0.438 for $\alpha = 10\%$, 0.819 for $\alpha = 20\%$ and 0.961 for $\alpha = 30\%$) and the realized load is observed to shift away from the consumer-desired load to the utility-desired load.

Sticky Price

The cost that a utility is willing to offer is bounded by many facts including spot market price, accuracy of the demand estimation and consumers' feedback. Therefore, we integrate an upper bound of the cost change (ΔC^{max}) in the nonlinear load adjustment model. This set of experiments explores the impact of the ΔC^{max} to the system. Figure 3.18 plots the goodness of measurement under various ΔC^{max} and Figure 3.19 shows the one day's cost with $\Delta C^{max} = 5, 10, 20$ cents/kWh respectively. We can see from Figure 3.18 that as ΔC^{max} grows, the improvement tends to slow down. A small ΔC^{max}

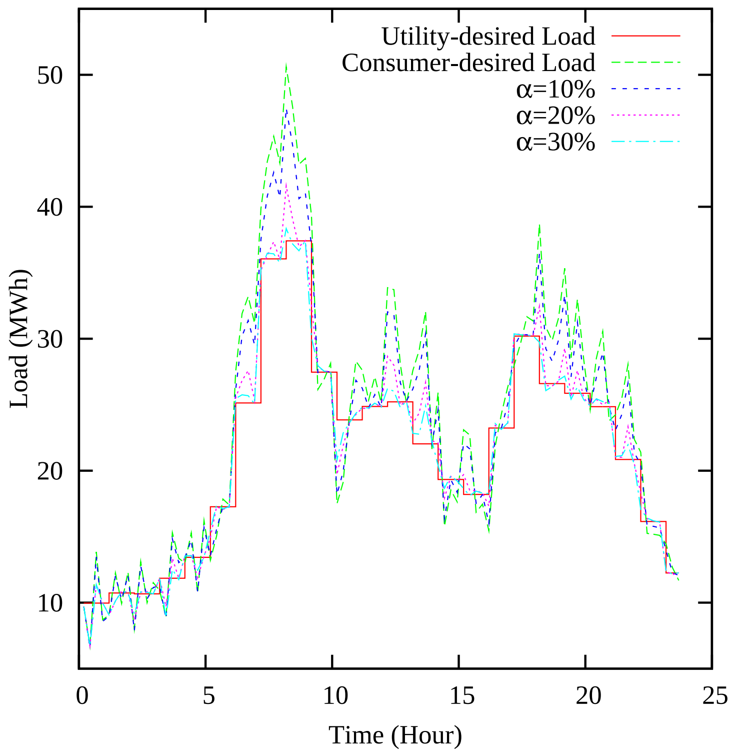


Figure 3.17: Nonlinear Load Adjustment Model - Utility Load Adjustment with Various Maximum Load Change ($\Delta D^{max}(k)$)

limits the system's capacity to offer enough incentive to the consumers and thus less load adjustment is conducted at the consumer side. As shown in Figure 3.19, when ΔC^{max} is small, the offered cost is more likely to hit the maximum indicating that the cost offered is not sufficient of the desired load change. On the other hand, when ΔC^{max} is already large enough to cover most consumer's elasticity, simply offering more price change has less and less impact. First, there is still the portion of consumers whose electricity usage is insensitive to cost change. Second, people have demand bottom-lines. Even those people with high elasticity are not willing to sacrifice certain activities merely for cheap electricity cost. Therefore, it is important for the utility to pick up an optimal ΔC^{max} to ensure the efficiency of the transactive control system.

Fatigue Model

In the next step, we integrate the fatigue model described in Section 3.3.2 into the nonlinear load adjustment model and study its impact on the whole

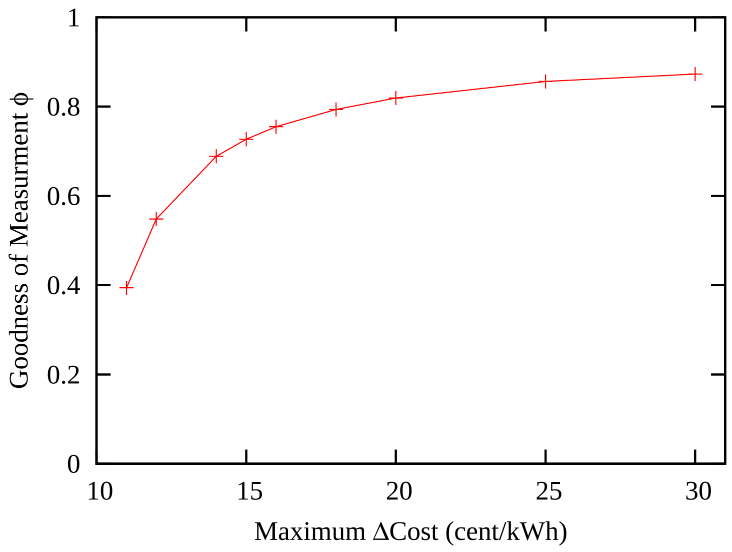


Figure 3.18: Nonlinear Load Adjustment Model with ΔC^{max} - Goodness of Measurement

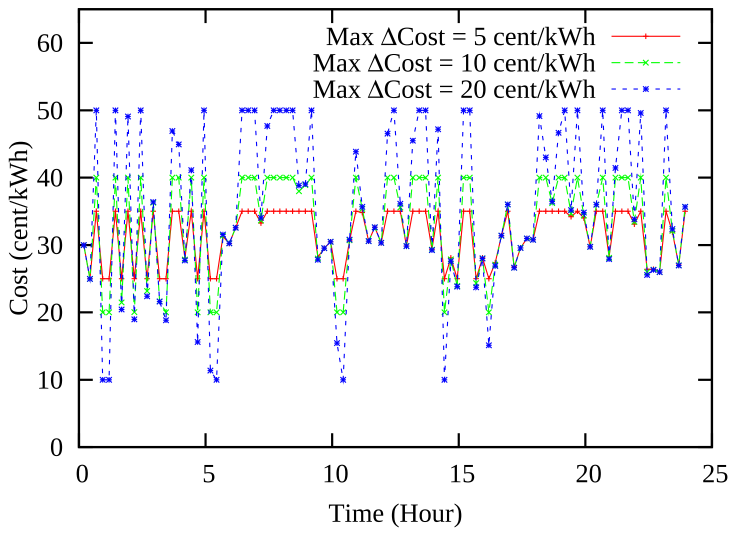


Figure 3.19: Nonlinear Load Adjustment Model with ΔC^{max} - Cost

system. ΔC^{max} is set to 20 cent/kWh and ΔC^{max} is set to $20\%\tilde{D}(k)$. Figure 3.20 compares the load adjustment between the model without fatigue and the model with 40% fatigue. With the fatigue model, the consumer-desired load increases in most time intervals. It is because some portion of the early adjusted load is shifted back to the current time interval. The utility has to deal with the extra the load change, which is hard to accurately estimate in practice, when making decision of the cost for the next interval. Therefore, the realized load generated with the consumer fatigue model gets farther

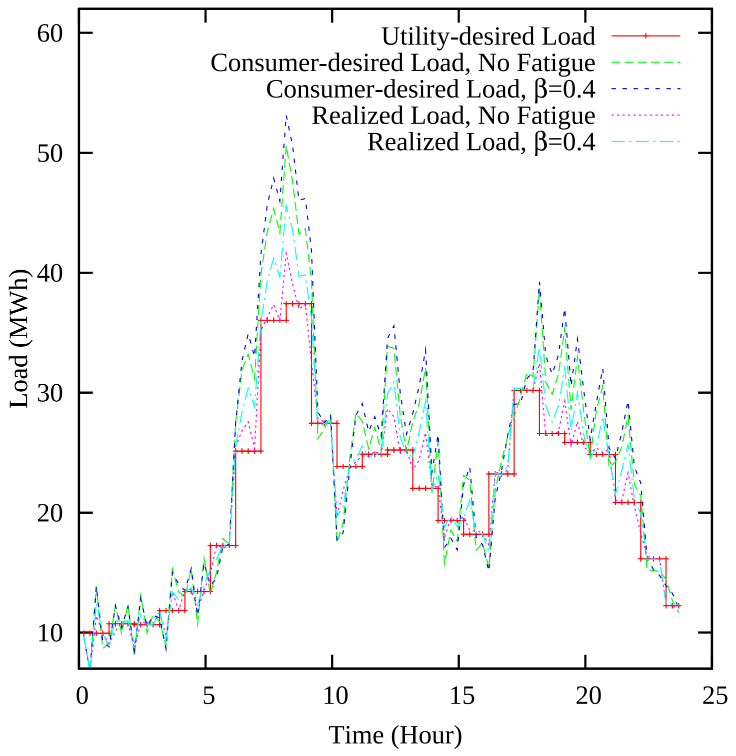


Figure 3.20: Nonlinear Load Adjustment with Fatigue Model - Utility Load Adjustment

away from the utility-desired load than the realized load without fatigue model, especially during the peak hours when the desired demand change is relatively large.

Transactive Signal Loss

There are two major sources of packet loss in the computer network: buffer overflow in the intermediate forwarding devices, such as switches and routers, as a result of limited buffer size and network congestion; and the lossy links interconnecting various network components. The majority of the lost packets in the wire-line networks are caused by the buffer overflow, while lossy channel or link failures occur more frequently in the wireless networks due to noise, interference, and channel fading. The smart grid is a large-scale complex network consisting a wide variety of communication technologies. When transactive signals are communicated between the utility and the consumers, they may travel through different types of networks and numerous networking devices, and losses can occur anywhere. Since signal losses are

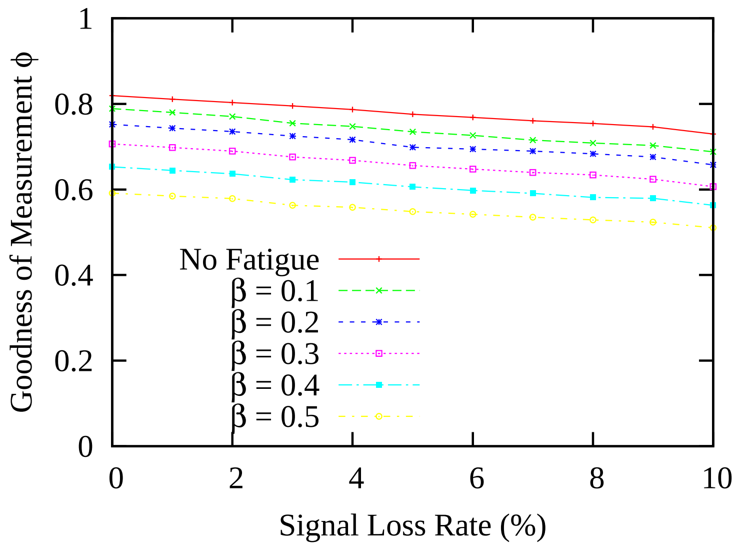


Figure 3.21: Nonlinear Load Adjustment Model with Signal Loss and Fatigue Model - Goodness of Measurement

unavoidable, and it is good to design the control algorithm to be robust to some reasonable amount of signal losses.

We implement a dropping packet feature in every node and every link, so that modelers can specify the amount of traffic to be randomly dropped during simulation run time. In this set of experiments, we specify the drop rate from 1% to 10% on both directions to simulate both TIS and TFS losses. ΔC^{max} is fixed to 20 cent/kWh and ΔC^{max} is fixed to $20\%\tilde{D}(k)$. Figure 3.21 shows the goodness of measurement for various combinations of loss rate and fatigue level. We can see that loss slightly reduces the goodness of measurement at all fatigue level. Given the loss rate as high as 10%, the goodness of measurement is reduced by less than 0.1. Since less TFS are collected at the utility, there will be a derivation of the consumer desired load in utility's perspective from that actually occurred. However, this derivation is automatically captured by the way that the utility updates the integrated elasticity $\epsilon^u(k)$ in each time interval. By providing an updated price with consideration of the signal loss, the system can still efficiently bring down the load to the utility-desired level. Figure 3.22 depicts the impact on the cost. Given that 20 cent/kWh is the base price, the price produced with the 10% signal loss case has larger changes than the no loss case. Therefore, our algorithm is quite robust to steady signal loss with the side effect of bigger jump in price.

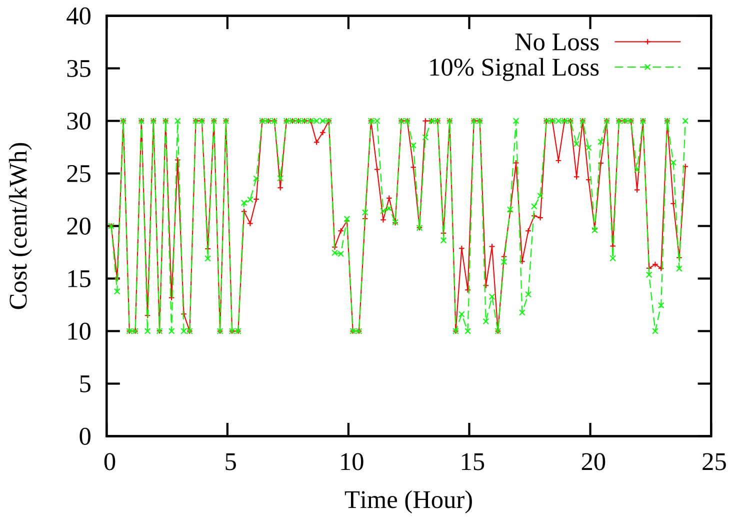


Figure 3.22: Nonlinear Load Adjustment Model with Signal Loss - Cost

3.3.4 Related Work

Research in the area of demand response and real time electricity pricing has been quite active. Customized electricity pricing agreements between an utility and its key customers are proposed in [111], using a what-if discrete event simulation procedure. The experiments of residential customer response to real time pricing in Anaheim area, where the real time pricing was only restricted to critical peak hours, is described in [112]. The methods to classify customers into two demand response programs, i.e. day ahead and real time programs and to help the supply side to build a more efficient contract portfolio is investigated in [113]. Analytical results for multi-period demand response program with customized real time pricing mechanism are provided in [114]. In particular, the stability issues with real-time pricing are investigated in [115].

In the field of transactive control mechanisms for the smart grid, a recent implementation and results of a smart grid field demonstration in Washington and Oregon are presented in [70], wherein two-way communication signals are used to coordinate load profile and price signals. The generalization and standardization of the transactive control approach is described in [116]. A comprehensive smart grid demonstration project in the Pacific Northwest is described in [107] and it demonstrates how transactive control can be used to manage distributed generation and demand response. A five-minute slotted communication protocol for power scheduling at home area network using real-time price provided by the smart meters is investigated in [117]. Another important type of the transactive control network for the smart grid is the wireless mesh network. The optimization of the geographical routing in the wireless mesh networks, and the potential usage of such information for optimizing the distribution and collection of transactive signals are presented in [118].

3.4 Chapter Summary

This chapter shows how our network testbed can be used for security and performance evaluation of applications in critically important infrastructures. It provides the capability and flexibility to set up a testing scenario with real emulated hosts modeling complicated applications and large-scale simulated network environment. Three applications are explored and reported in this chapter.

First, 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. This scenario requires detailed functional behavior of some meters (those ones directly involved in the attack), but only routing and background traffic behavior from the others. Therefore, we emulated a few meters with the rest of the meters and the communication network being simulated. The whole experiment has been done on a single multi-core machine, and this makes the experiment economic and flexible to set up with different network topologies.

Second, we investigate a buffer flooding attack on DNP3-controlled data aggregators. The attacker spoofs or captures a normal relay, and floods the connected data aggregator with unsolicited response events as if they are coming from the victim relay. The goal is to overload the shared event buffer in the data aggregator so that events from other normal relays will be dropped upon arriving to a full buffer. The attack has been implemented on a real data aggregator. Also a DTMC model and a simulation model have been developed for analyzing the behavior of such attacks. Results have shown the simple flooding attack can be very effective, and strong authentication is definitely required toward securing the DNP3-controlled SCADA networks.

Finally, we develop a transactive control network simulation framework in our testbed, and evaluate several transactive control algorithms models within this framework. Specifically, we study a two-level hierarchical architecture of power distribution. The simulation framework and control algorithms can be extended to more complex multi-level tree topologies. We have investigated impact of customer-varying pricing in the context of demand response in [114], and plan to integrate the idea with detailed customer elasticity models into this simulation framework. Another avenue of future research includes the case where utilities are purchasing electricity from various generation sources and must dynamically choose the schedule of power to be drawn from each. Also interesting is the case where multiple utilities can provide services to the same group of consumers, which impose strong constraints on the offered dynamic price. We plan to create detailed models of dynamic pricing and investigate its impact on this transactive control network.

CHAPTER 4

CONCLUSIONS AND FUTURE DIRECTIONS

4.1 Summary of Thesis Research

There is emerging awareness of the need for security in the communication networks of the power grid, and the government is giving considerable attention to securing the power grid. However, the large size of the network and the fast evolution of technology often make it infeasible to conduct evaluation experiments on real-device testing systems. Network simulation and emulation help to address this concern, as it is possible to create network simulation/emulation models as large as regional or even national power grids. With the goal of building a large-scale network testbed to facilitate research on the smart grid, we developed a parallel network simulator named S3F/S3FNet, and an OpenVZ-based network emulation system whose network transactions are timestamped in virtual time. We then integrated the two systems to create our network testbed, in which emulation is used for executing native programs to ensure fidelity, and simulation is used to model a large-scale communication environment and background traffic. We have utilized the testbed to create several large-scale simulation/emulation models for evaluating various smart grid applications including a DDoS attack in an AMI network, an event buffer flooding attack in a DNP3-controlled SCADA network, and demand response control algorithms in a hierarchical transactive control network.

4.2 Future Directions

Our long-term goal is to conduct research in cyber-security, networks, and simulation and modeling in order to build more secure, resilient, and safe computing and communication networks. Some future directions are highlighted in the following section.

4.2.1 A Virtual Power System Testbed for Security Evaluation and Decision Support

It is crucial that cyber-security be brought to the cyber-infrastructure that controls the electric power grid. There are many decisions to be made concerning whether to protect, what to protect, how much to protect, and what to do when some intrusion is suspected. To assist decision-makers in answering such questions, we plan to extend our existing network simulation/emulation testbed to create a cyber-physical system testbed by seamlessly integrating the following components:

- real power system hardware and software, such as PMUs, relays, data aggregators, and control stations, and other testbed containing real equipment, such as the smart meter testbed in the TCIPG lab
- a simulator of power generation and distribution, such as PowerWorld [119] and Real Time Power System Simulation (RTDS) [120]

Through a combination of simulation and emulation, the testbed seamlessly will integrate virtual and real components, allowing for evaluation of the effectiveness of security technology in a realistic setting, and enabling decision-makers to observe the consequences of potential decisions in a safe virtual environment. It will help answer questions like: What sort of security architecture best balances competing concerns of availability, safety, and security? What impact does any given security technology have on the operators, and on a system's ability to keep up with real-time monitoring requirements? How does one balance the cost of implementing and maintaining security measures against the risk of not doing so? The following are some examples of the potential usage of the testbed:

- Training and human-in-the-loop event analysis
- Assisting in decision-making of response mechanisms against both attacks and accidents

- Analysis of incremental deployment, and evaluation of whether or not to adopt certain technologies

4.2.2 Software-Defined Networking-Based Architecture Design for Traffic Management in the Smart Grid

A large number of applications with various requirements (e.g., delay, loss, and jitter) are running concurrently in the smart grid, and the diversity of the applications keeps increasing as more intelligence is pushed into this huge, complex network. It is very challenging to efficiently manage all kinds of traffic and ensure the appropriate quality of service (QoS) across multiple smart grid applications to meet all the different timing requirements, which range from 10 ms to 1 second (e.g., critical information, such as control commands must be delivered on time). A proper location for investigating traffic management is the substation routers, where all kinds of traffic (AMI, SCADA, PMU, enterprise data, etc.) are aggregated and then communicated to the control station(s) through the core network. The work is challenging because (1) the huge size of the network slows down the deployment process of any new ideas; and (2) the growing number of new features integrated into the smart grid would require frequent revisits of many existing designs. Therefore, we would like to investigate a software-defined networking-based architecture design for easy deployment and efficient traffic management in the large-scale and rapidly changing smart grid.

An SDN design decouples the data plane and the control plane of a switch or a router. The logically centralized controller can directly configure the packet-handling mechanisms in the underlying forwarding devices (e.g., drop, forward, modify, or enqueue). The benefits of applying SDN in the context of the smart grid include the following:

- The need to individually configure network devices is eliminated.
- Policies are enforced consistently across the network infrastructures, including policies for access control, traffic engineering, quality of service, and security.
- Functionality of the network can be defined and modified after it has been deployed.

- Products evolve at software speeds, rather than at standards-body speed.

We will build and evaluate SDN-based designs using our testbed for efficient traffic management on the routers in the substations, the core network, and the control stations in the smart grid. A new paradigm of QoS for smart grid traffic will be defined, including specific rules to classify traffic into flows according to various application requirements and operational constraints. Queuing algorithms to handle traffic from the separate flows will be investigated with respect to the specific traffic patterns observed under realistic operational configurations. In addition, an SDN-based design will offer the opportunity to conduct global optimization in the centralized controller for convenience, a feature that is not available in traditional smart grid communication networks.

REFERENCES

[1] D. M. Nicol, D. Jin, and Y. Zheng, "S3F: The scalable simulation framework revisited," in *Proceedings of the 2011 Winter Simulation Conference (WSC)*, Phoenix, AZ, December 2011, pp. 3283–3294.

[2] Y. Zheng, D. M. Nicol, D. Jin, and N. Tanaka, "A virtual time system for virtualization-based network emulations and simulations," *Journal of Simulation*, vol. 6, no. 3, pp. 205–213, August 2012. [Online]. Available: http://dx.doi.org/10.1057/jos.2012.12

[3] D. Jin, Y. Zheng, H. Zhu, D. Nicol, and L. Winterrowd, "Virtual time integration of emulation and parallel simulation," in *Proceedings of the 2012 Workshop on Principles of Advanced and Distributed Simulation (PADS)*, Zhangjiajie, China, July 2012, pp. 120–130.

[4] "TCIPG: Trustworthy Cyber Infrastructure for the Power Grid," http://tcipg.org/, Accessed 2011.

[5] D. Jin, D. Nicol, and M. Caesar, "Efficient gigabit ethernet switch models for large-scale simulation," in *Proceedings of the 2010 Workshop on Principles of Advanced and Distributed Simulation (PADS)*, May 2010, pp. 1 –10.

[6] D. Jin and D. Nicol, "Fast simulation of background traffic through fair queueing networks," in *Proceedings of the 2010 Winter Simulation Conference (WSC)*, Baltimore, MD, December 2010, pp. 2935–2946.

[7] D. Jin and D. M. Nicol, "Parallel simulation of software defined networks," in *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, Montreal, Quebec, Canada, May 2013, pp. 91–102.

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

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

[10] 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 the 19th Workshop on Principles of Advanced and Distributed Simulation*, 2005, pp. 119–128.

[11] P. R. Barford and L. H. Landweber, "Bench-style network research in an internet instance laboratory," in *ITCom 2002: The Convergence of Information Technologies and Communications*. International Society for Optics and Photonics, 2002, pp. 175–183.

[12] "PlanetLab," http://www.planet-lab.org, Accessed 2011.

[13] "Emulab," http://www.emulab.net, Accessed 2011.

[14] 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.

[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 (TRIDENTCOM)*. 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 *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. ACM, 2006, pp. 3–14.

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

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

[19] J. DeHart, F. Kuhns, J. Parwatikar, J. Turner, C. Wiseman, and K. Wong, "The open network laboratory," in *ACM SIGCSE Bulletin*, vol. 38, no. 1. ACM, 2006, pp. 107–111.

[20] D. Raychaudhuri, I. Seskar, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu, and M. Singh, "Overview of the orbit radio grid testbed for evaluation of next-generation wireless network protocols," in *Wireless Communications and Networking Conference, 2005 IEEE*, vol. 3. IEEE, 2005, pp. 1664–1669.

[21] G. Judd and P. Steenkiste, "Repeatable and realistic wireless experimentation through physical emulation," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 1, pp. 63–68, 2004.

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

[23] "GTNetS," http://www.ece.gatech.edu/research/labs/MANIACS/GTNetS, Accessed 2010.

[24] G. R. Yaun, D. Bauer, H. L. Bhutada, C. D. Carothers, M. Yuksel, and S. Kalyanaraman, "Large-scale network simulation techniques: Examples of TCP and OSPF models," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 3, pp. 27–41, 2003.

[25] 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.

[26] "The ns-3 Project," http://www.nsnam.org, Accessed 2012.

[27] "Omnet," http://www.omnetpp.org, Accessed 2008.

[28] "J-sim," http://www.j-sim.org/, Accessed 2013.

[29] OPNET, "Network modeling and simulation environment," http://www.opnet.com/, Accessed 2008.

[30] "Scalable Network Technologies," http://scalable-networks.com, Accessed 2009.

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

[32] 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 (NSDI)*. Berkeley, CA, USA: USENIX Association, 2008, pp. 407–422.

[33] 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," *International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*, pp. 1–10, 2009.

[34] P. M. Dickens, P. Heidelberger, and D. M. Nicol, “A distributed memory LAPSE: Parallel simulation of message-passing programs,” in *Proceedings of the 8th Workshop on Parallel and Distributed Simulation.* New York, NY, USA: ACM, 1994, pp. 32–38.

[35] “OpenVZ: A container-based virtualization for Linux,” http://wiki.openvz.org, Accessed 2011.

[36] R. M. Fujimoto, “Parallel discrete event simulation,” in *Proceedings of the 21st Conference on Winter Simulation.* Washington, D.C., USA: ACM, 1989, pp. 19–28.

[37] R. M. Fujimoto, “Parallel discrete event simulation,” *Communications of the ACM*, vol. 33, no. 10, pp. 30–53, Oct. 1990.

[38] “Linux: Running at 10,000 HZ,” http://kerneltrap.org/node/1766, Accessed 2011.

[39] Iperf, http://iperf.sourceforge.net/, 2012.

[40] Apache, “Apache: HTTP server project,” http://httpd.apache.org/, 2012.

[41] Lynx, “Lynx: A text browser for the World Wide Web,” http://lynx.browser.org/, 2012.

[42] A. Juels and J. Brainard, “Client puzzles: A cryptographic countermeasure against connection depletion attacks,” in *NDSS*, 1999.

[43] D. Nicol and G. Yan, “High-performance simulation of low-resolution network flows,” *Simulation*, vol. 82, no. 1, p. 21, 2006.

[44] D. Stiliadis and A. Varma, “Design and analysis of frame-based fair queueing: A new traffic scheduling algorithm for packet-switched networks,” *SIGMETRICS Performance Evaluation Review*, vol. 24, no. 1, pp. 104–115, 1996.

[45] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis, “Weighted round-robin cell multiplexing in a general-purpose ATM switch chip,” *IEEE Journal on Selected Areas in Communications*, vol. 9, no. 8, pp. 1265–1279, 1991.

[46] L. Zhang, “VirtualClock: A new traffic control algorithm for packet-switched networks,” *ACM Transactions on Computer Systems (TOCS)*, vol. 9, no. 2, p. 124, 1991.

[47] G. Chuanxiong, "SRR: An O(1) time complexity packet scheduler for flows in multi-service packet networks," in *SIGCOMM '01: Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, 2001, pp. 211–222.

[48] G. Covington, G. Gibb, J. Lockwood, and N. McKeown, "A packet generator on the NetFPGA platform," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2009.

[49] G. Watson, N. McKeown, and M. Casado, "NetFPGA: A tool for network research and education," in *Workshop on Architecture Research using FPGA Platforms*, 2006.

[50] "ENDACE DAG network monitoring cards," http://www.endace.com, Accessed 2009.

[51] M. Zukerman, T. Neame, and R. Addie, "Internet traffic modeling and future technology implications," in *22nd Annual Joint Conference of the IEEE Computer and Communications Societies*, 2003.

[52] Google, "Inter-Datacenter WAN with centralized TE using SDN and OpenFlow," https://www.opennetworking.org/images/stories/downloads/misc/googlesdn.pdf, Accessed 2012.

[53] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[54] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks.* ACM, 2010, p. 19.

[55] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies.* ACM, 2012, pp. 253–264.

[56] "OFTest, a Python based OpenFlow switch test framework," http://www.openflow.org/wk/index.php/OFTestTutorial, Accessed 2012.

[57] R. Sherwood, "OFlops," http://www.openflow.org/wk/index.php/Oflops, Accessed 2012.

[58] "ns-3 OpenFlow switch support," http://www.nsnam.org/docs/release/3.13/models/html/openflow-switch.html, Accessed 2012.

[59] “Open vSwitch,” http://openvswitch.org/, Accessed 2012.

[60] Y. Zheng, D. Jin, and D. M. Nicol, “Validation of application behavior on a virtual time integrated network emulation testbed,” in *Proceedings of the Winter Simulation Conference*, 2012, p. 246.

[61] “POX,” http://www.noxrepo.org/pox/about-pox/, Accessed 2012.

[62] “Openflow switch specification version 1.1.0,” http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf, 2011.

[63] “OpenFlow Switching Reference System,” http://www.openflow.org/wp/downloads/, Accessed 2012.

[64] R. Ayani, “A parallel simulation scheme based on distances between objects,” *Royal Institute of Technology, Department of Telecommunication Systems-Computer Systems*, 1988.

[65] B. Lubachevsky, “Efficient distributed event-driven simulations of multiple-loop networks,” *Communications of the ACM*, vol. 32, no. 1, pp. 111–123, 1989.

[66] D. Nicol, “The cost of conservative synchronization in parallel discrete event simulations,” *Journal of the ACM (JACM)*, vol. 40, no. 2, pp. 304–333, 1993.

[67] K. Chandy and J. Misra, “Distributed simulation: A case study in design and verification of distributed programs,” *IEEE Transactions on Software Engineering*, no. 5, pp. 440–452, 1979.

[68] D. Jin, D. M. Nicol, and G. Yan, “An event buffer flooding attack in DNP3 controlled SCADA systems,” in *Proceedings of the 2011 Winter Simulation Conference (WSC)*, Phoenix, AZ, December 2011, pp. 2614–2626.

[69] D. Jin, X. Zhang, and S. Ghosh, “Simulation models for evaluation of network design and hierarchical transactive control mechanisms in smart grids,” in *Proceedings of the 2012 Conference on Innovative Smart Grid Technologies (ISGT)*, January 2012, pp. 1–8.

[70] D. Hammerstrom, R. Ambrosio, J. Brous, T. Carlon, D. Chassin, J. DeSteese, R. Guttromson, G. Horst, O. Jarvegren, R. Kajfasz et al., “Pacific northwest gridwise testbed demonstration projects,” *Part I. Olympic Peninsula Project*, 2007.

[71] G. Andersson, P. Donalek, R. Farmer, N. Hatziargyriou, I. Kamwa, P. Kundur, N. Martins, J. Paserba, P. Pourbeik, J. Sanchez-Gasca et al., “Causes of the 2003 major grid blackouts in North America and

Europe, and recommended means to improve system dynamic performance," *IEEE Transactions on Power Systems*, vol. 20, no. 4, pp. 1922–1928, 2005.

[72] "Pacific Northwest National Laboratory (PNNL), looking back at the August 2003 blackout," http://eioc.pnl.gov/research/2003blackout.stm, Accessed 2010.

[73] S. Corsi and C. Sabelli, "General blackout in Italy Sunday September 28, 2003, h. 03: 28: 00," in *IEEE Power Engineering Society General Meeting*, 2005, pp. 1691–1702.

[74] "Electric Power Research Insititute, DNP security development, evaluation and testing project opportunity," http://mydocs.epri.com/docs/public/000000000001016988.pdf, 2008.

[75] "DNP Users Group, DNP3 specification, secure authentication, supplement to volume 2," http://www.dnp.org/Modules/Library/Document.aspx, March Accessed 2010.

[76] M. Majdalawieh, F. Parisi-Presicce, and D. Wijesekera, "DNPSec: Distributed network protocol version 3 (DNP3) security framework," *Advances in Computer, Information, and Systems Sciences, and Engineering*, pp. 227–234, 2006.

[77] S. East, J. Butts, M. Papa, and S. Shenoi, "A taxonomy of attacks on the DNP3 protocol," *Critical Infrastructure Protection III*, pp. 67–81, 2009.

[78] Möbius Team, *The Möbius Manual www.mobius.illinois.edu*, University of Illinois at Urbana-Champaign, Urbana, IL, 2010.

[79] "DNP3 specification application layer volume 2 part 1," http://www.dnp.org/DNP3Downloads/Forms/AllItems.aspx, Accessed 2007.

[80] S. Ross, *Stochastic Processes*. Wiley New York, 1996.

[81] J. Meyer, A. Movaghar, and W. Sanders, "Stochastic activity networks: Structure, behavior, and application," in *International Workshop on Timed Petri Nets*. IEEE Computer Society, 1985, pp. 106–115.

[82] W. Sanders, "Integrated frameworks for multi-level and multi-formalism modeling," in *8th International Workshop on Petri Nets and Performance Models*. IEEE, 2002, pp. 2–9.

[83] F. Bause and P. Kritzinger, *Stochastic Petri Nets*. Vieweg, 2002.

[84] Triangle Microworks, "DNP3 communication protocol test harness," http://www.trianglemicroworks.com/, Accessed 2009.

[85] A. Demers, S. Keshav, and S. Shenker, “Analysis and simulation of a fair queueing algorithm,” in *Symposium Proceedings on Communications Architectures and Protocols.* ACM, 1989, pp. 1–12.

[86] “DNP3 specification application layer volume 2 part 2,” http://www.dnp.org/DNP3Downloads/Forms/AllItems.aspx, Accessed 2007.

[87] L. Piètre-Cambacédès and P. Sitbon, “Cryptographic key management for SCADA systems-issues and perspectives,” in *International Conference on Information Security and Assurance.* IEEE, 2008, pp. 156–161.

[88] C. Bowen III, T. Buennemeyer, and R. Thomas, “A plan for SCADA security employing best practices and client puzzles to deter DoS attacks,” *Working Together: R&D Partnerships in Homeland Security*, 2005.

[89] P. Ralston, J. Graham, and J. Hieb, “Cyber security risk assessment for SCADA and DCS networks,” *ISA Transactions*, vol. 46, no. 4, pp. 583–594, 2007.

[90] J. Mirkovic and P. Reiher, “A taxonomy of DDoS attack and DDoS defense mechanisms,” *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 2, pp. 39–53, 2004.

[91] E. Bompard, C. Gao, R. Napoli, A. Russo, M. Masera, and A. Stefanini, “Risk assessment of malicious attacks against power systems,” *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, vol. 39, no. 5, pp. 1074–1085, 2009.

[92] J. Fernandez and A. Fernandez, “SCADA systems: Vulnerabilities and remediation,” *Journal of Computing Sciences in Colleges*, vol. 20, no. 4, pp. 160–168, 2005.

[93] V. Igure, S. Laughter, and R. Williams, “Security issues in SCADA networks,” *Computers and Security*, vol. 25, no. 7, pp. 498–506, 2006.

[94] S. Patel and Y. Yu, “Analysis of SCADA security models,” *International Management Review*, vol. 3, no. 2, 2007.

[95] A. Faruk, “Testing and exploring vulnerabilities of the applications implementing DNP3 protocol,” M.S. thesis, Kungliga Tekniska högskolan, 2008.

[96] S. Hong and S. Lee, “Challenges and pespectives in security measures for the SCADA system,” in *Proceedings of 5th Myongji-Tsinghua University Joint Seminar on Prototection and Automation*, 2008.

[97] T. Mander, R. Cheung, and F. Nabhani, "Power system DNP3 data object security using data sets," *Computers & Security*, vol. 29, no. 4, pp. 487–500, 2010.

[98] D. Rrushi and U. di Milano, "SCADA intrusion prevention system," in *Proceedings of 1st CI2RCO Critical Information Infrastructure Protection Conference*, 2006.

[99] "Digital Bond, DNP3 IDS Signatures," http://www.digitalbond.com/index.php/research/scada-idsips/ids-signatures/dnp3-ids-signatures-2/, Accessed 2010.

[100] J. Graham and S. Patel, "Security considerations in SCADA communication protocols," Intelligent Systems Research Laboratory, Tech. Rep. TR-ISRL-04-01, 2004.

[101] H. Khurana, R. Bobba, T. Yardley, P. Agarwal, and E. Heine, "Design principles for power grid cyber-infrastructure authentication protocols," in *43th Annual Hawaii International Conference on System Sciences*, 2010.

[102] U.S. National Institute of Standards and Technology (NIST), "Smart grid interoperability standards project," http://www.nist.gov/smartgrid/, 2010.

[103] "NIST IR 7628 Guidelines for Smart Grid Cyber Security," http://csrc.nist.gov/publications/PubsNISTIRs.html, August Accessed 2010.

[104] K. Cheung, X. Wang, B. Chiu, Y. Xiao, and R. Rios-Zalapa, "Generation dispatch in a smart grid environment," in *Innovative Smart Grid Technologies (ISGT), 2010.* IEEE, 2010, pp. 1–6.

[105] H. Li, Y. Li, and Z. Li, "A multiperiod energy acquisition model for a distribution company with distributed generation and interruptible load," *IEEE Transactions on Power Systems*, vol. 22, no. 2, pp. 588–596, 2007.

[106] S. Ghosh, D. Iancu, D. Katz-Rogozhnikov, D. Phan, and M. Squillante, "Power generation management under time-varying power and demand conditions," in *Power and Energy Society General Meeting, 2011 IEEE.* IEEE, 2011, pp. 1–7.

[107] P. Huang, J. Kalagnanam, R. Natarajan, M. Sharma, R. Ambrosio, D. Hammerstrom, and R. Melton, "Analytics and transactive control design for the pacific northwest smart grid demonstration project," in *First IEEE International Conference on Smart Grid Communications (SmartGridComm).* IEEE, 2010, pp. 449–454.

[108] J. Roos and I. Lane, "Industrial power demand response analysis for one-part real-time pricing," *IEEE Transactions on Power Systems*, vol. 13, no. 1, pp. 159–164, 1998.

[109] R. Boisvert, P. Cappers, B. Neenan, and B. Scott, "Industrial and commercial customer response to real time electricity prices," *Unpublished Manuscript at Neenan Associates, Syracuse, NY*, 2003.

[110] C. Su and D. Kirschen, "Quantifying the effect of demand response on electricity markets," *IEEE Transactions on Power Systems*, vol. 24, no. 3, pp. 1199–1207, 2009.

[111] J. Roos and C. Kern, "Modelling customer demand response to dynamic price signals using artificial intelligence," in *8th International Conference on Metering and Tariffs for Energy Supply.* IET, 1996, pp. 213–217.

[112] F. A. Wolak, "Residential customer response to real-time pricing: The anaheim critical peak pricing experiment," http://escholarship.org/uc/item/3td3n1x1, UC Berkeley: Center for the Study of Energy Markets, 2007.

[113] S. Valero, M. Ortiz, C. Senabre, C. Alvarez, F. Franco, and A. Gabaldon, "Methods for customer and demand response policies selection in new electricity markets," *Generation, Transmission and Distribution, IET*, vol. 1, no. 1, pp. 104–110, 2007.

[114] S. Ghosh, J. Kalagnanam, D. Katz, M. Squillante, X. Zhang, and E. Feinberg, "Incentive design for lowest cost aggregate energy demand reduction," in *Proceedings of First IEEE International Conference on Smart Grid Communications (SmartGridComm)*, 2010, pp. 519–524.

[115] M. Roozbehani, M. Dahleh, and S. Mitter, "Dynamic pricing and stabilization of supply and demand in modern electric power grids," in *Proceeding of First IEEE International Conference on Smart Grid Communications (SmartGridComm)*, 2010, pp. 543–548.

[116] D. Hammerstrom, T. Oliver, R. Melton, and R. Ambrosio, "Standardization of a hierarchical transactive control system," *Grid Interop*, vol. 9.

[117] S. Magazine, "Smart (in-home) power scheduling for demand response on the smart grid," *2011 IEEE PES Innovative Smart Grid Technologies (ISGT)*, pp. 1–7, 2011.

[118] B. Lichtensteiger, B. Bjelajac, C. Müller, and C. Wietfeld, "RF mesh systems for smart metering: System architecture and performance,"

in *Proceeding of First IEEE International Conference on Smart Grid Communications (SmartGridComm)*, 2010, pp. 379–384.

[119] “Powerworld simulator, the visual approach to electric power systems,” http://www.powerworld.com/, Accessed 2012.

[120] “Real Time Digital Simulator (RTDS) Technologies,” http://www.rtds.com/, Accessed 2012.