LARGE-SCALE AND HIGH-FIDELITY WIRELESS NETWORK SIMULATION AND EMULATION

BY

YUHAO ZHENG

DISSERTATION

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

Urbana, Illinois

Doctoral Committee:

Professor David M. Nicol, Chair
Associate Professor Sam King
Associate Professor Jason Liu, Florida International University
Professor Klara Nahrstedt

Abstract

Large-scale networks, such as the Internet, cellular networks, play critical roles in today's life. The advancement of large-scale computer and communication networks heavily depends on the successful transformation from in-house research efforts to real productions. When studying these network systems, modeling is an important and useful approach as it allows studies not physically realizable. In the domain of wireless networks, simulations are usually used to study new or existing designs, as it is economically and technically expensive to implement those designs using real hardware. However, evaluation based on any methodologies other than actual measurements on actual networks raises questions of fidelity, due to necessary simplifications and assumptions. In this dissertation, we present a system for large-scale and high-fidelity wireless network simulation and emulation. Our system provides high-fidelity in three areas. 1) High physical layer fidelity: we use sophisticated radio propagation models such as ray-tracing, validated by an anechoic chamber. 2) High application functional fidelity: we allow unmodified and compiled application code to run inside our system, minimizing the modeling error of software behavior. 3) High application temporal fidelity: we provide virtual timestamps to the virtual machines rather than using the wall-clock time, making them perceive time as if they were running concurrently in real world. Besides high-fidelity, our system can handle large-scale network scenarios in reasonable speed, by making the simulation and emulation run in parallel across multiple machines. Application lookahead, the ability to predict future behavior of software, can help further improve speed and scalability by reducing the frequency of synchronization. However, it may affect fidelity as lookahead could be wrong due to software complexity and runtime uncertainty. We extensively study the impacts of lookahead on our system to provide guidelines of when lookahead may be used.

To my parents

Acknowledgments

I would like to express my sincere gratitude to my advisor Professor David Nicol, for his support, guidance, and trust throughout my Ph.D. study. I am greatly benefited from his foresight to research problems, and deeply influenced by his high standards to science. I witnessed his diligence and his becoming ITI Director as well as Franklin W. Woeltge Professor during my Ph.D. journey, which significantly encourages me to work harder and harder.

I also want express my appreciation to all the professors serving on my Ph.D. committee. Many thanks to Professor Klara Nahrstedt for her bringing broad knowledge and ideas from communication networks and multimedia systems to my thesis research. Professor Sam King guided me to state-of-the-art operating system and virtualization techniques, some of which directly contribute to my thesis research. Professor Jason Liu, my academic elder brother and undoubtedly an expert in parallel discrete-event simulation, not only provided many constructive advices to my thesis research but also shared some valuable memory of working with David to me.

Special thanks to my colleague and friend Dong Jin and Huaiyu Zhu, who make me feel I am not alone on my path especially when we worked on our paper at midnight. Sincere thanks to Professor Nitin Vaidya, who offered me a research assistantship and guided me on getting started with research. I also want to express my acknowledgement to Suhail Barot who provided technical supports for the anechoic chamber for more than a year, as well as to Tim Yardley who helped me set up many experiments in the lab. In addition, I would like to convey my memory of Professor Jennifer Hou, who gave profound guidance to my Ph.D. life when I first came to UIUC.

Finally I would like to thank my family for their endless love and support. I appreciate their understanding and patience to my Ph.D. study, which appears long to them.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, we introduce the motivations behind this thesis research, specify the research problems this thesis focuses on, identify the research contribution we have made, and present the outline of this thesis.

1.1 Motivations

Large-scale networks, such as the Internet, cellular networks, play critical roles in today's life. The advancement of large-scale computer and communication networks heavily depends on the successful transformation from in-house research efforts to real productions. Modeling is a very important and useful approach to study these network systems, as it allows studies not physically realizable [48]. Analytical models, when be found, are attractive for their theoretical simplicity as they make certain simplified assumptions that make the solution tractable. In many cases, unfortunately, a real system is so complex that either an analytical model cannot be found or its oversimplified assumptions make the result deviate from reality too much. Simulation models, on the other hand, are able to represent more complex systems with a large number of parameters. Being complementary to analytical models, simulations can also provide quantitative results for studying complex systems.

In the domain of wireless networks, simulations are usually used to study new or existing designs, as it is economically and technically expensive to implement those designs using real hardware. Some wireless network protocols are so complex that analytical models often have to make some simplified assumptions, and such assumptions can only represent limited cases in

reality. For example, Bianchi analytically studies the performance of IEEE 802.11 DCF [23] under the assumption of "single cell", i.e. all wireless nodes are within carrier sense range of each other. Despite this model is extremely accurate under its assumptions, many wireless networks in real world are not single cell. Although simulation models also need to make certain assumptions, such assumptions are much less restrictive compared with analytical models. In fact, analytical models often use simulation models for validations [23][70], indicating that the latter is more realistic.

Evaluation based on any methodology other than actual measurements on actual networks raises questions of fidelity, owing to necessary simplifications in representing behavior. Network emulations that involve real devices and live networks yield higher fidelity as they eliminate modeling error, despite that their scalability and flexibility are limited by hardware expense and what can be equipped in the lab. The combination of network simulation and emulation is promising to achieve the benefits of both — simulation can provide enormous background traffic at large-scale and high-speed, while emulation can model critical network devices and provide high fidelity. In this thesis, we present a system for large-scale and high-fidelity wireless network simulation and emulation. Our system is able to handle large-scale network scenarios up to thousands or millions of hosts, yet still provide sufficient level of fidelity according to the requirement of experiments. Our system can improve the creditability and/or the speed of research that relies on network simulation and emulation.

1.2 Research Objectives

This dissertation focuses on building and studying a large-scale and high-fidelity testbed for wireless network simulation and emulation. Our research efforts concentrate on the following four aspects.

High Physical Layer Fidelity

The radio propagation model (also known as the channel model) determines whether a wireless transmission can be reliably received by its intended receiver. All higher layers in the protocol stack rely on this model of the physical channel. Simplified channel models such as the free space model may not be suitable for a complicated scenario, e.g. an indoor with walls and obstacles, as the impact of obstacles on radio propagation is described only in a statistical way without reference to the placement of actual obstacles. To faithfully represent complex scenarios, our system supports sophisticated models like ray-tracing model [51] [72] [82] and the transmission line matrix (TLM) model [66] [67]. These models are based on numerical methods which describe the reaction between electromagnetic waves and objects. Furthermore, these channel models have been carefully validated by real experiments conducted inside the Illinois Wireless Wind Tunnel (iWWT) [78], which is built to minimize the impact of environment in wireless experiments. With the help of iWWT, we try to answer the question that how sensitive different models are to small changes in the reflection environment.

High Application Functional Fidelity

Besides the channel model (physical layer in the network protocol stack), modeling the behavior of software running over the network (application layer) may also bring considerable errors. Some network simulation tools like ns-2 [1] and J-Sim [73] use simple models to represent the applications, e.g. traffic source and traffic sink. However, these models may be too simple to give realistic behavior. Even if we want to implement a more complex application model ourselves, we have to translate the application logic to fit the programming interface defined by the target simulator, and such translation is unlikely to be lossless. An effective way to accurately capture the behavior of software is to actually run the software [15] [73] [77] [80], and our system supports this way. This is done by using virtualization techniques, which partition physical resources into different Virtual Environments (VEs) [4] [18] [79]. By running the compiled and unmodified application

code inside our system, we minimized the error caused by modeling software behavior, resulting in high functional fidelity. Through validation of application behavior, we try to find out how much error is introduced by our simulation/emulation testbed.

High Application Temporal Fidelity

Although running real application code in virtualized platforms greatly improves functional fidelity, such emulations typically virtualize execution, but not time. The software managed VEs takes their notion of time from the host system's clock, which means that time-stamped actions taken by virtual environments whose execution is multi-tasked on a host reflect the host's serialization. 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. In our system, we use a virtual time mechanism that gives virtualized applications the temporal appearance of running concurrently on different physical machines. By using virtual time, our system not only yields high functional fidelity but also provides high temporal fidelity. In this thesis, we analyze the error introduced by our virtual time system both analytically and experimentally, and we also study the speed-fidelity trade off by using scheduling timeslices of different length.

Good Scalability and Speed

There is a common as recognized tradeoff between behavioral accuracy and execution speed in simulation domain [58]. Normally, higher fidelity requires more computation, resulting in slower execution speed. As mobile computers become increasingly popular nowadays, the number of mobile station rises drastically, up to thousands or even millions of stations. While our system provides high fidelity, it needs to be reasonably fast when simulating large-scale networks. We seek for acceleration by making the simulation and emulation run in parallel across multiple machines. Specifically, the whole simulation and emulation progress is controlled by S3F [64], the next generation of Scalable Simulation Framework (SSF) [54]. From parallel discrete event simulations' (PDES) point of view [31], S3F belongs conservative synchronization, whose performance

highly depends on lookahead [61] [62]. We present an approach to predict application behaviors and provide application-level lookahead, in order to accelerate the parallel simulation and distributed emulation. We also extensively study the impact of application lookahead to simulation and emulation, in regard to both speed and fidelity.

1.3 Contributions

We made several contributions in this dissertation. The first contribution is the validation of radio channel models using iWWT, an electromagnetic anechoic chamber. Radio channel models, or the physical layer, are crucial to wireless network simulation. It determines whether a wireless transmission can be reliably received by its intended receiver or not, and all higher layers in the protocol stack rely on this. While simple channel models such as the free space model are well studied and validated, they may not be suitable for complicated scenarios involving a lot of obstacles. Sophisticated models, such as the ray-tracing model and transmission line matrix model, describe the reaction between electromagnetic waves and objects. However, they are often validated in an open environment which unavoidably contains noise and other interference. We study how sensitive different models are to small changes in the reflection environment, in which case even small noise and interference are undesired. In this thesis, we present validation results collected from an anechoic chamber, a well-controlled environment that eliminates noise and interference. We found that the errors eliminated by the anechoic chamber are small relative to errors introduce by model uncertainty. While seemingly a negative result, it is actually positive in a practical sense: future validations need not be attempted within such a chamber.

Our second contribution is the invention of timeslice-based virtual time system for OpenVZ network emulation. Emulations executing real code have high functional fidelity, but may not have high temporal fidelity because virtual machines usually use their host's clock. Our virtual time system greatly improves temporal fidelity in network emulations, by giving each virtual machine its own virtual clock. This frees emulation from real-time constrains, not only allowing emulations

to run slower than real time to preserve temporal fidelity when hardware is not real-time capable, but also allowing them to run faster than real time when resource is abundant. Although our implementation is based on OpenVZ OS-level virtualization, this approach may be extended to other virtualization platforms such as Xen. Unlike some other solutions that scale real time by a constant or variable ratio, our timeslice-based approach is less tied to real time as a virtual machine can be suspended arbitrarily long whenever simulation is under computation. This mechanism is closer to parallel discrete-event simulation (PDES), such that synchronization techniques from PDES may be borrowed. In addition, the tradeoff between speed and fidelity can be naturally explored by tuning the length of timeslice, the minimal scheduling granularity. In this thesis we present detailed design, implementation and evaluation of our virtual time system.

The third contribution is the integration of distributed virtual-time enabled emulation and S3F parallel simulator. By marrying the emulation framework with our next generation scalable network simulator, we can achieve the benefits of both simulation and emulation. We can use emulation to represent the execution of critical software, and use simulation to model an extensive ensemble of background computation and communication. The OpenVZ-based emulation runs distributedly across multiple machines, and this provides better scalability despite the fact that we can already run 300+ containers on a single machine. Our algorithmic contributions lay in the design and management of virtual time as it transitions from emulation, to simulation, and back. In particular, inescapable uncertainties in emulation behavior force us to explicitly set and reset timestamps so as to avoid either emulator or simulator having to deal with a packet arriving in its logical past. We provide analytic bounds and empirical evidence that the error introduced in resetting timestamps is small. We also demonstrate the usability of our testbed by validating behavior of different application categories and providing a case study using our testbed as an example. Our results show that our virtual time mechanism does not introduce additional error compared with the error introduced by OpenVZ itself.

The fourth contribution is the implementation of application-level lookahead, as well as studying its impacts to distributed emulation and parallel simulation. Lookahead is critical to the per-

formance of conservative parallel discrete-event simulation (PDES), and we find it also important to our simulation/emulation framework, as our timeslice-based virtual time system behaves in a PDES-like way. Application lookahead, the ability to predict future behaviors of software, may help reducing synchronization overhead for performance gain. However, such prediction is challenging due to software complexity and runtime uncertainty. In this thesis, we present an implementation of application lookahead by using artificial neural network (ANN) based time series prediction. We mainly focus on studying the impacts of application lookahead on our distributed simulation/emulation testbed, rather than how to improve lookahead accuracy using various approaches. We find that application lookahead can greatly reduce synchronization overhead and improve speed by up to 3X, but incorrect lookahead may affect fidelity to different degree depending on application categories.

1.4 Thesis Outline

The remainder of this dissertation is structured as follows.

Chapter 2 presents our work on high-fidelity radio channel models, validated by iWWT, an electromagnetic anechoic chamber. We start with an introduction to radio channel models in Section 2.1. Sections 2.2 introduces our anechoic chamber testbed and its repeatability, while Section 2.3 describes our experiment framework for validation. Section 2.4 defines the simple and advanced ray-tracing models. Section 2.5 reviews the transmission line matrix model, and its application to our experiments. Section 2.6 presents and analyzes the experimental results. Section 2.7 reviews related work, and Section 2.8 summarizes this chapter.

Chapter 3 presents our work on a virtual time system for network emulations based on OpenVZ. Section 3.1 gives an introduction by comparing network simulation and network emulation. Section 3.2 depicts the overall system architecture of our virtual time system, and Section 3.3 presents implementation details. Section 3.4 evaluates our system by giving extensive experimental results, while Section 3.5 studies the impacts of variable timeslice both analytically and experimentally.

Section 3.6 reviews related work, and Section 3.7 summarizes this chapter.

Chapter 4 presents our work on integration of the above virtual-time-enabled emulation to S3F parallel simulation. Section 4.1 starts with providing an overview to our S3F parallel discrete-event simulator. Section 4.2 describes the high-level system design architecture, and Section 4.3 illustrates the implementation details of the synchronization mechanism and VE controller of the system. Section 4.4 presents experimental results of application behavior validation to demonstrate the fidelity of our testbed. Section 4.5 shows a case study of our system for a DDoS attack security exercise in the AMI network. Section 4.6 reviews related work, and Section 4.7 summarizes this chapter.

Chapter 5 presents our work on application lookahead — the ability to predict future behavior of applications in order to reduce synchronization overhead and improve execution speed of simulation and emulation. Section 5.1 reviews the importance of lookahead in parallel discrete event simulation. Section 5.2 overviews the design of the distributed emulation version of our testbed with application lookahead support, and Section 5.3 presents the implementation of the application lookahead model. Section 5.4 studies the impact of application lookahead by analyzing extensive experimental results. Section 5.5 reviews related work, and Section 5.6 summarizes this chapter.

Finally, Chapter 6 summarizes the conclusions made in this dissertation and provides future research directions. Conclusion and future direction are presented in Section 6.1 and Section 6.2 respectively.

Chapter 2

Radio Channel Models

Wireless network simulation is used for research because of its simplicity and repeatability. While simple radio propagation models are evaluated quickly and are suitable for simple scenarios, sophisticated models can handle more complex environments and provide better accuracy. However, the cost of higher accuracy is slower execution speed. This chapter describes experiments that validate ray-tracing and transmission line matrix models of the radio channel, within each approach considering versions that differ in their attention to detail and computational cost. We conducted the experiments under highly controlled conditions, within an anechoic chamber. Our main conclusion is that the errors due to lack of knowledge about beam forms and antennae shape significantly outweigh errors that might have been introduced if the experiments had not been within the anechoic chamber. While seemingly negative, the implication is that for our problem domain and level of information about the wireless environment, complex means of radio isolation are not needed in validation studies.

2.1 Radio Channel Models

The research community has developed many protocols for wireless networking, often evaluating them via simulation. This often leaves open a question of fidelity, because analysts must make approximations in order to make the simulations tractable.

The radio propagation model (also known as the channel model) determines whether a wireless transmission can be reliably received by its intended receiver or not. All higher layers in the protocol stack rely on this model of the physical channel. Simplified channel models (e.g. that

used in ns-2 [1]) may not be suitable for a complicated scenario, e.g. an indoor with walls and obstacles, as the impact of obstacles on radio propagation is described only in a statistical way without reference to the placement of actual obstacles.

When details of the domain are known one may use sophisticated models, such as the method of ray-tracing [51] [72] [82] or the transmission line matrix (TLM) model [66] [67]. These models are based on numerical methods which describe the reaction between electromagnetic waves and objects. Although these kinds of techniques have large computational overhead and may be prohibitive for some complex scenarios [41], they yield better accuracy.

There is a tradeoff in simulation between computational cost and accuracy [58]. Complex models tend to have higher accuracy, but at the cost of a longer simulation run. It is particularly important to understand such tradeoffs, so that in a given context one is led to choose a model with the greatest chance of providing a level of accuracy needed, within the available computational budget. We are interested in exploring this tradeoff space, with an eye towards assessing how sensitive different models are to small changes in the reflection environment. Previous work [49] [51] [72] [82] on model validation was conducted in environments that admit to outside interference with the experiment. As the sensitivities we seek are potentially small, we conducted our experiments within the Illinois Wireless Wind Tunnel (iWWT) [78], which is built to minimize the impact of environment in wireless experiments. The iWWT is an electromagnetic anechoic chamber whose shielding prevents external radio sources from entering the chamber; and whose inner wall is lined with electromagnetically absorbing materials, which reflect minimal energy. The experimentalist has full control over placement of transmitters, receivers, and reflection obstacles. This gives us a well-controlled "free space" inside the chamber, which is ideal for conducting wireless network experiments.

This chapter presents study of different channel models, using the iWWT. The experimental framework is simple—there is a transmitter, a receiver, a barrier between them, and a reflective surface. By making controlled changes in the position of the reflective surface we can study the sensitivity of actual received signal strength to those changes, and how any model's predicted

received signal strength also changes. We consider two ray-tracing models; a simple one that considers path loss (the attenuation of an electromagnetic wave as it propagates) but not wave interference and reinforcement, and an advanced ray-tracing model that accounts for both path loss and the interaction between multiple paths. In addition we implemented the method of the transmission line matrix (TLM), a numerical method that models wave propagation through a discretized grid. Under TLM we consider different levels of discretization.

In this experimental framework and for both model types we explore the tradeoffs between computational cost and accuracy. We find that with well-chosen parameters, most models predict behavior within 2dB on average, that none of the models as configured do a particularly good job at predicting the response when the reflector is moved a few wavelengths, but do track the real data when changes occur at a larger scale. We find that within our experiments effects due to wave cancellation and reinforcement are noticeable, that TLM's accuracy is quite dependent on the discretization used. However, most important observation is that even within the highly controlled experimental framework we considered, lack of detailed knowledge about the radios and antennae used led to considerable unexplained (by our models) variation in the real data. This is good news for those who would validate wireless models—it appears that the errors removed by using an anechoic chamber in our context are dwarfed by other errors, which means that validation within a chamber is unnecessary.

2.2 Illinois Wireless Wind Tunnel

The Illinois Wireless Wind Tunnel (iWWT) [78], is an electromagnetic anechoic chamber, with the following two properties: 1) signal outside the chamber cannot interfere the devices inside the chamber, and 2) signal inside the chamber is absorbed by the wall and thus cannot be reflected. With these two properties, the wireless experiments inside the chamber are supposed to be repeatable. This section records several experimental results and studies the repeatability of the experiments inside the anechoic chamber.

Unlike using networks simulators in which all of the parameters are under control and the simulation is fully repeatable, several factors can affect the experiments inside the anechoic chamber, even though the chamber is a well-controlled environment.

- **Randomness of protocol**: 802.11a/b/g uses a random backoff mechanism to alleviate collisions. This brings uncertainty to the experiments. Although this should not affect the results over a sufficiently long period, no two experiments may be exactly the same due to the randomness of protocol.

- **Imprecise position of objects**: To guarantee the repeatability of experiments, the position of all the devices inside the chamber must be unchanged from different runs. However, once we move a device and try to put it back to its original position, there is usually some positioning error, no matter how small it is. Such small positioning error can sometimes dominate the experiments, which is shown later in this section.

This section mainly focuses on the above two factors that may affect the repeatability. We next present several experimental results to demonstrate the impact of these two factors.

Randomness of Protocol

To study the impact of protocol randomness, we conducted an experiment with the following deployment. There is only one wireless link in this scenario, with one transmitter and one receiver. Two laptops are deployed at two end points inside the chamber, and the topology is shown in Figure 2.1. Laptop #1 acts as the transmitter and laptop #2 acts as the receiver, and their antennas are in a straight line without obstacles between them.

In this experiment, we used a simple C program, which records the sending time and receiving time of each packet, in order to track all the packets.

The duration of the test is set to be 1 second, 3 seconds, and 10 seconds respectively. For each duration, there are 10 runs. During the experiment, the position of the two laptops is unchanged.

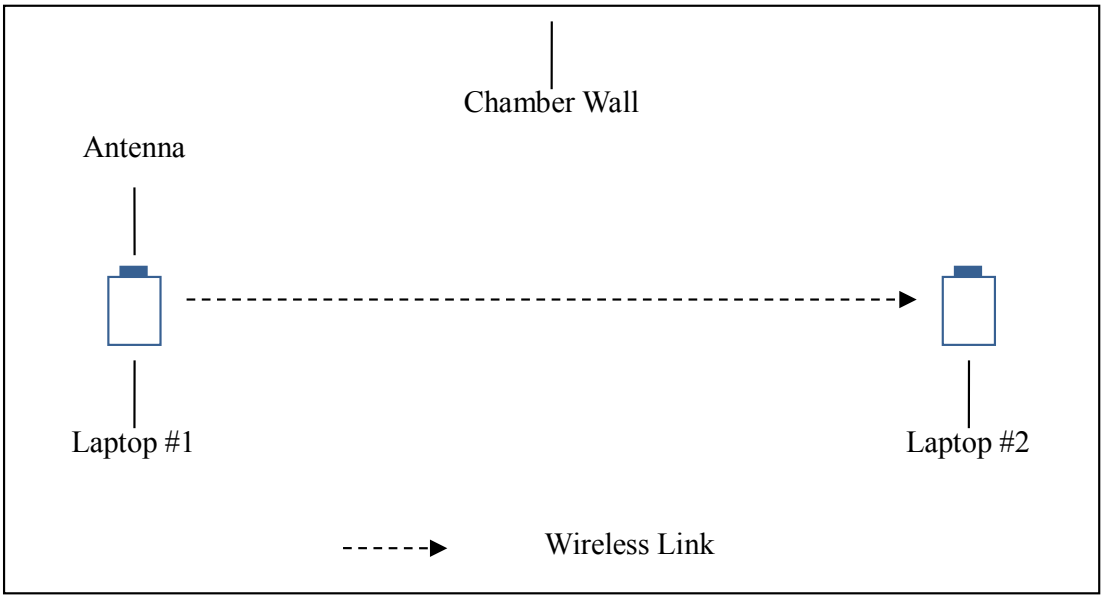


Figure 2.1: iWWT Repeatability Experiment — One-Link Scenario

Table 2.1 shows the results for 3 seconds case. In the table, *latency* refers to the time period between two consecutive packet receipts.

As we can see in Table 2.1, run #3 to #10 show consistent results regarding to throughput. However, run #1 and #2 show a much lower throughput. After careful analysis of the trace file, we found that the transmitter is not transmitting all the time. However, when it is transmitting, the instantaneous throughput is around 33Mbps, which is comparable to run #3 to #10. The best explanation for this we could find is given by the latest MadWifi document (v0.9.4) [2]: "Ad-hoc mode is broken; symptoms are intermittent operation". This perfectly matches the symptom that is happening in run #1 and #2.

Now let us focus on run #3 to #10. We can see that results like throughput and average latency is repeatable. However, individual packet latencies are not repeatable. This is unlike experiment using network simulators, in which we can make two experiments identical by setting the same random seed number. To justify the difficulty of repeating individual packet latencies, Figure 2.2 plots the latencies of run #9 and run #10. As we can see, although the average latencies are almost the same, the distributions are different.

Table 2.1: Repeatability Experiment Results — One-link Scenario

Run ID	Thrpt (Mbps)	Pkt Sent	Pkt Recv	PDR	Latency (sec)		
					Max	Avg	Min
1	10.85	2970	2970	100%	6.94E-2	1.03E-3	2.07E-4
2	11.84	3207	3123	97%	6.96E-2	9.47E-4	1.88E-4
3	32.99	8866	8866	100%	1.43E-3	3.40E-4	4.00E-5
4	33.03	8888	8888	100%	6.92E-3	3.40E-4	1.30E-5
5	32.99	8861	8861	100%	1.50E-3	3.40E-4	3.80E-5
6	32.96	8863	8863	100%	2.09E-3	3.40E-4	1.33E-4
7	33.15	8918	8918	100%	1.46E-3	3.38E-4	1.50E-5
8	33.07	8890	8890	100%	6.58E-3	3.39E-4	1.20E-5
9	33.02	8865	8865	100%	1.45E-3	3.40E-4	1.25E-4
10	33.09	8910	8910	100%	1.50E-3	3.39E-4	4.00E-5

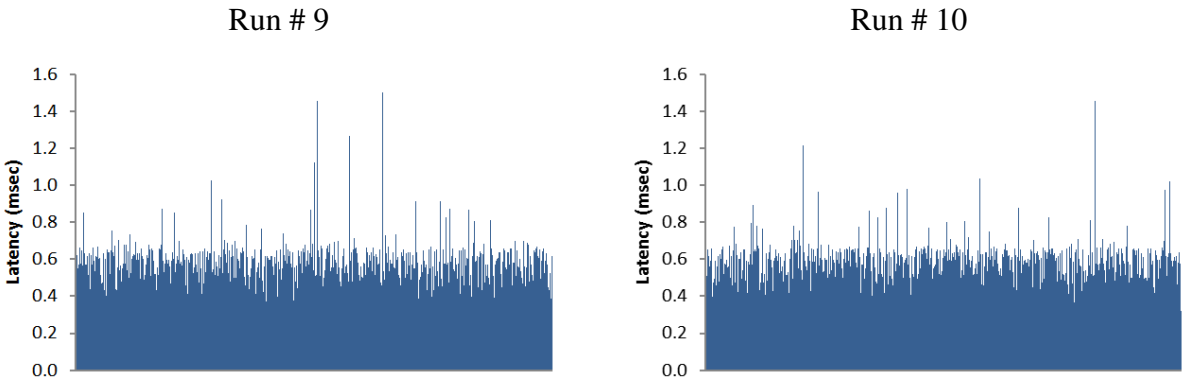


Figure 2.2: Latency Distribution on One-link Scenario

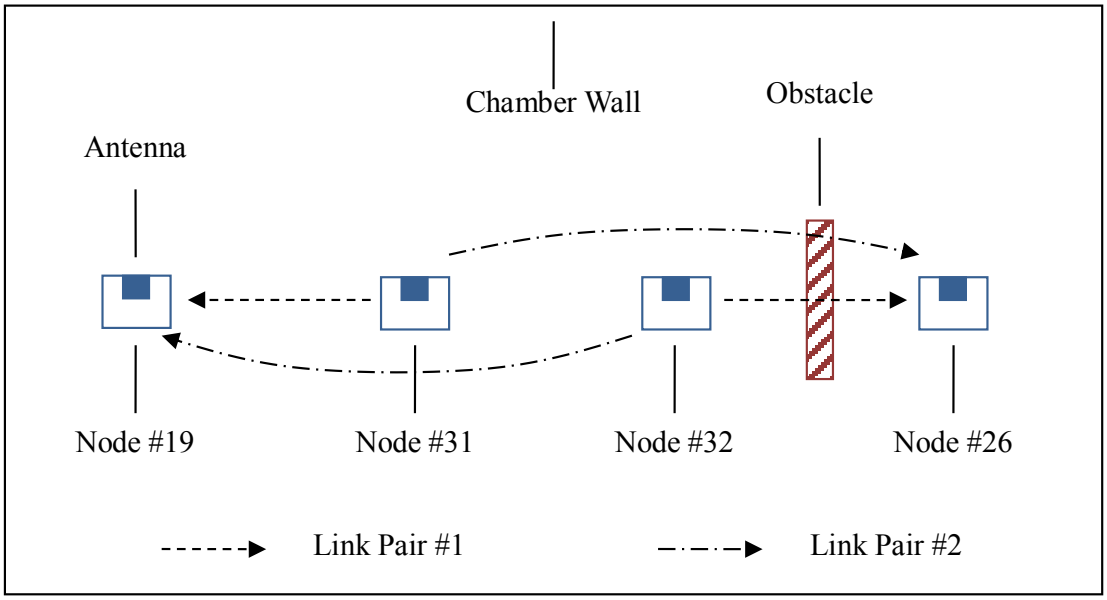


Figure 2.3: iWWT Repeatability Experiment — Straightline Placement

In summary, due to the randomness of the protocol, microscopic results are not repeatable in the chamber. However, macroscopic results are not affected by such randomness.

Imprecise Position of Objects

To study the impact of object position, we used the following topology, shown by Figure 2.3. To avoid the problem of intermittent operation of ad-hoc mode caused by MadWifi, we used Soekris net4521 boxes [6] instead of laptops. Each node contains one wireless card and uses the internal antenna of that card. In this topology, four nodes are deployed in a straight line inside the chamber. Node #31 and #32 are transmitters, and node #19 and #26 are receivers. With these four nodes, there are two possible link pairs: 1) 31→19 and 32→26, and 2) 31→26 and 32→19. The throughputs of these two link pairs are tested respectively. In addition, to figure out the impact of obstacle, one absorbing material may be placed between node #32 and node #26.

In this experiment, we used iperf [12] to measure UDP throughput of a link. The test time is set to be 1 second, 3 seconds, and 10 seconds respectively. Possible transmission rates of a link are 6Mbps and 54Mbps (802.11a). Each sub case contains 10 runs, while maximum, average, and

Table 2.2: Repeatability Experiment Results — Straightline Placement without Obstacle

	Link # 1 Thrpt (Mbps)			Link # 2 Thrpt (Mbps)		
	Max	Avg	Min	Max	Avg	Min
Link #1: 31→19 (6M) Link #2: 32→26 (6M)	3.08	2.97	2.87	3.76	3.65	3.52
Link #1: 31→26 (6M) Link #2: 32→19 (6M)	3.05	2.94	2.74	3.56	3.22	3.03
Link #1: 31→19 (54M) Link #2: 32→26 (6M)	7.34	7.05	6.80	4.41	4.36	4.25
Link #1: 31→26 (54M) Link #2: 32→19 (6M)	6.80	5.40	3.58	4.70	4.41	4.17
Link #1: 31→19 (54M) Link #2: 32→26 (54M)	19.20	18.91	18.10	16.40	15.99	15.50
Link #1: 31→26 (54M) Link #2: 32→19 (54M)	17.90	17.64	17.50	7.99	7.17	5.13

minimum throughputs of these 10 runs are recorded. Table 2.2 shows the results of 10-second case *without* obstacle.

As we can see from the table, the throughputs are not symmetric even though the two links are using the same data rate. This is due to the asymmetry of the two links. However, even though the two links are not symmetric, the throughput is expected to be repeatable. Table 2.2 indicates a fluctuation of throughput more than 20% in some cases (Link #1 of 4th row, and Link #2 of 6th row), and thus is considered to be not very repeatable. Such great fluctuations happen in link pair of 31→26 and 32→19, and with the 54Mbps link. The reason for this is that the four nodes are deployed in a straight line, so the node in the middle becomes an obstacle. For example, node #31 acts as an obstacle to link 32→19. When the link quality is poor and a high data rate is used, the throughput is not stable, resulting in a large fluctuation.

Table 2.3: Repeatability Experiment Results — Straightline Placement with Obstacle

(percentage comparison with Table 2.2)

	Link # 1 Thrpt (Mbps)			Link # 2 Thrpt (Mbps)		
	Max	Avg	Min	Max	Avg	Min
Link #1: 31→19 (6M) Link #2: 32→26 (6M)	3.47	3.18 (↑7.07%)	2.88	2.67	2.27 (↓37.8%)	1.89
Link #1: 31→26 (6M) Link #2: 32→19 (6M)	3.41	3.09 (↑5.10%)	2.74	2.38	2.07 (↓35.7%)	1.92
Link #1: 31→19 (54M) Link #2: 32→26 (6M)	6.89	6.45 (↓8.51%)	6.09	4.51	4.46 (↑2.29%)	4.41
Link #1: 31→26 (54M) Link #2: 32→19 (6M)	0.58	0.53 (↓90.2%)	0.44	5.06	5.03 (↑14.1%)	4.98
Link #1: 31→19 (54M) Link #2: 32→26 (54M)	17.20	17.08 (↓9.68%)	16.90	6.78	6.55 (↓59.0%)	6.36
Link #1: 31→26 (54M) Link #2: 32→19 (54M)	4.37	4.23 (↓76.0%)	4.08	1.22	0.99 (↓86.2%)	0.88

To study the impact of obstacles, we put an absorbing material between node #32 and node #26, while leaving the four wireless nodes unmoved. The same script is run again under this topology, and the results are shown in Table 2.3. As expected, some links experience a much lower throughput compared with Table 2.2, because of the added obstacle. Again, it demonstrates the previous explanation that when the link quality is poor, throughput fluctuates greatly, especially when using high data rate (54Mbps).

Next step, to find out the impact of imprecise object positions, we picked up every object (including the nodes and the obstacle) and tried to put it back to its original position. Then the same test script is run again. Table 2.4 shows the results of this.

Compared with Table 2.3, some results in Tables 2.4 are quite different. We believe that this is

Table 2.4: Repeatability Experiment Results — Straightline with Obstacle, Object Moved

(percentage comparison with Table 2.3)

	Link # 1 Thrpt (Mbps)			Link # 2 Thrpt (Mbps)		
	Max	Avg	Min	Max	Avg	Min
Link #1: 31→19 (6M) Link #2: 32→26 (6M)	3.68	3.27 (↑2.83%)	2.98	2.46	2.16 (↓4.85%)	1.70
Link #1: 31→26 (6M) Link #2: 32→19 (6M)	4.84	4.72 (↑52.8%)	4.61	0.73	0.68 (↓67.2%)	0.60
Link #1: 31→19 (54M) Link #2: 32→26 (6M)	8.21	7.14 (↑10.7%)	6.27	4.33	4.15 (↓6.95%)	4.06
Link #1: 31→26 (54M) Link #2: 32→19 (6M)	3.11	2.77 (↑423%)	2.46	2.29	2.20 (↓56.3%)	2.10
Link #1: 31→19 (54M) Link #2: 32→26 (54M)	17.00	16.85 (↓1.35%)	16.70	2.92	2.75 (↓58.0%)	2.62
Link #1: 31→26 (54M) Link #2: 32→19 (54M)	8.98	8.35 (↑97.4%)	7.37	2.69	2.35 (↑137%)	2.17

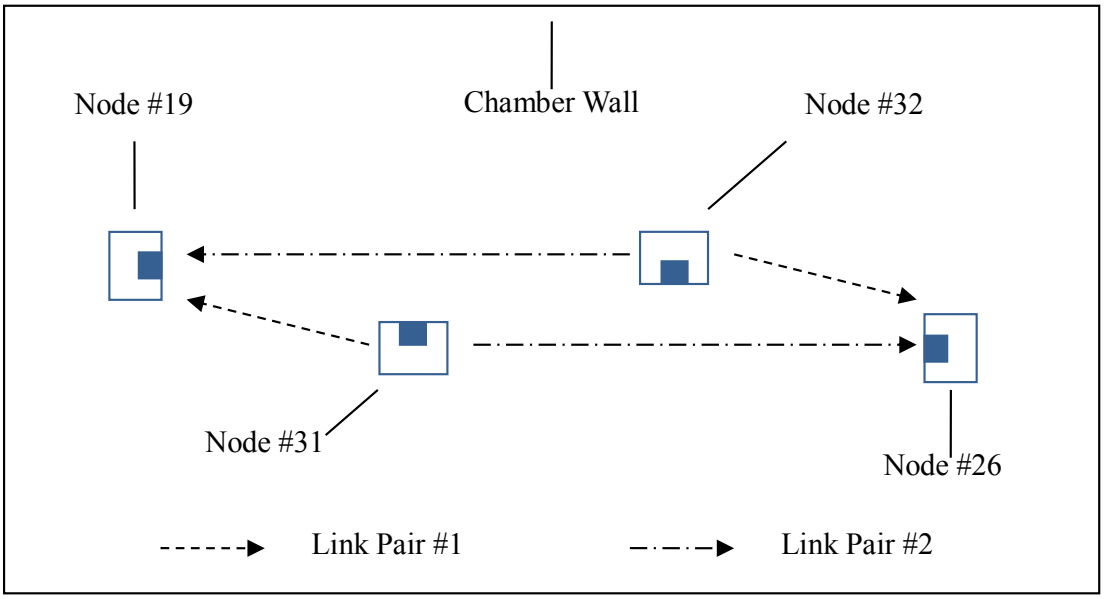


Figure 2.4: iWWT Repeatability Experiment — Diamond Placement

due to error in placement of objects. The explanation is as follow. When the link quality is poor (due to obstacle), a small position change can change the channel quality greatly, reflecting in a large change in throughput.

To verify the above explanation, we placed the four wireless nodes to a diamond shape, as shown in Figure 2.4. In this case, small change of position should not affect the channel quality too much, since the nodes themselves are not obstacles. The test results are shown in Table 2.5.

In Table 2.5, some links experience higher throughput compare with Table 2, because that there is no longer obstacles. As we can also see, the throughputs in Table 2.5 are more stable and have smaller fluctuation. To study the impact of each node's position change, we picked up only one node and put it back, and run the test script. Table 2.6 shows the results after moving node #19. It shows very little changes in average throughput. Then we picked up and put back node #26, node #32, and node #31 one by one. After each one move, we ran the test script again. The results show that none of these moves results in a large average throughput change.

In summary, we conclude that the experiment inside the chamber is repeatable if and only if small change of an object's position does not change the channel quality greatly.

Table 2.5: Repeatability Experiment Results — Diamond Placement

	Link # 1 Thrpt (Mbps)			Link # 2 Thrpt (Mbps)		
	Max	Avg	Min	Max	Avg	Min
Link #1: 31→19 (6M) Link #2: 32→26 (6M)	2.75	2.67	2.58	2.98	2.90	2.80
Link #1: 31→26 (6M) Link #2: 32→19 (6M)	2.67	2.60	2.50	2.70	2.60	2.51
Link #1: 31→19 (54M) Link #2: 32→26 (6M)	12.20	11.81	11.20	3.82	3.75	3.70
Link #1: 31→26 (54M) Link #2: 32→19 (6M)	13.30	13.01	12.70	3.41	3.37	3.32
Link #1: 31→19 (54M) Link #2: 32→26 (54M)	19.80	19.48	19.30	16.80	16.54	16.20
Link #1: 31→26 (54M) Link #2: 32→19 (54M)	19.60	19.25	18.90	17.30	16.95	16.50

Table 2.6: Repeatability Experiment Results — Diamond Placement, Object Moved

(percentage comparison with Table 2.5)

	Link # 1 Thrpt (Mbps)			Link # 2 Thrpt (Mbps)		
	Max	Avg	Min	Max	Avg	Min
Link #1: 31→19 (6M) Link #2: 32→26 (6M)	2.82	2.75 (↑3.00%)	2.71	2.88	2.83 (↓2.41%)	2.77
Link #1: 31→26 (6M) Link #2: 32→19 (6M)	2.69	2.62 (↑0.77%)	2.58	2.62	2.56 (↓1.54%)	2.47
Link #1: 31→19 (54M) Link #2: 32→26 (6M)	12.60	11.85 (↑0.34%)	10.90	3.88	3.74 (↓0.27%)	3.64
Link #1: 31→26 (54M) Link #2: 32→19 (6M)	13.20	12.96 (↓0.38%)	12.30	3.47	3.38 (↑0.30%)	3.33
Link #1: 31→19 (54M) Link #2: 32→26 (54M)	20.00	19.34 (↓0.72%)	18.80	17.30	16.64 (↑0.60%)	15.70
Link #1: 31→26 (54M) Link #2: 32→19 (54M)	19.50	19.23 (↓0.10%)	19.00	17.30	16.93 (↓0.12%)	16.30

In conclusion, due to the randomness of protocol, small-scale experimental results are not repeatable, but the large-scale ones are repeatable. In addition, small changes of object's position can affect performance. The degree to which the performance will be affected depends on how great the channel quality is changed due to the change of object position.

2.3 Experimental Framework

To validate radio channel models, Figure 2.5 shows our experiment setup inside the iWWT. Due to space limitations we can only put a limited number of objects inside, and these are constrained to lay on a narrow walkway along the chamber walls. We construct a simple scenario with one sender, and one receiver. We put an attenuator between two nodes to dampen the direct path, and put a reflector at the other side of the chamber, to create a secondary reflected path. All the objects inside the chamber are sufficiently far away from the chamber wall, in order to guarantee the inner anechoic wall can absorb energy as designed. Our experiments will move the reflector horizontally (along X axis) across the chamber (up to 10.0ft or 3.05m on either side of the center) at a fixed Y coordinate (10.5ft or 3.20m from the baseline between sender and receiver), and will also move it up to 1ft (304.8mm) in the Y dimension while centered in the X dimension.

Wireless Interface

We use Soekris Engineering net4521 [6] wireless boxes with wireless PC card adapters plugged in as wireless nodes inside the iWWT. The dimension of the wireless box is approximately 240mm $\times$ 150mm. We run our experiments under 802.11a, using frequency band 5.2 GHz, and the corresponding wavelength is around 57.7mm. This means that experiments which alter the reflector position in the Y dimension will change position in a range of under 5 wavelengths, whereas experiments that change the X dimension cover a range spanning almost 100 wavelengths.

We nominally treat a wireless node as a point, as we were unable to obtain any information about the radio's antenna size or orientation. We assume the antenna is omni-directional, i.e. it

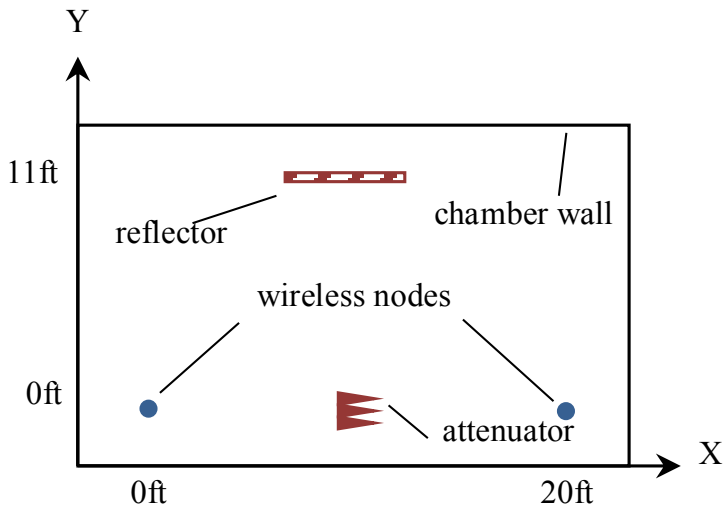


Figure 2.5: Channel Model Vadilation Experiment Setup inside iWWT

radiates power uniformly in the 2-D plane we consider. The received signal strength (RSS) is the field strength measured (and recorded) by the wireless device. We revisit these assumptions later in the chapter.

Attenuator

The inner wall of the iWWT is lined with absorbing foam panel, egg-crated foam impregnated with carbon to create an electromagnetically lossy material. We use one panel of this as an attenuator to dampen the line-of-sight signal paths. The shape of the absorbing panel is illustrated by Figure 2.5, with a 3×3 array of protruding cones.

This attenuator provides different path loss for different directions, depending on whether the transmitter is on the left, or the right, with the latter case giving the larger loss. Such asymmetry is due to the shape of the attenuator itself. When the wave is trying to penetrate the attenuator from right to left, it will be bounced back and forth by absorbing cones many times, with most of the energy being absorbed.

In our model, we treat path loss due to the attenuator as a fixed multiplier for the direct path. Since the attenuator is directional, we have different path loss factor k_{att} for different directions.

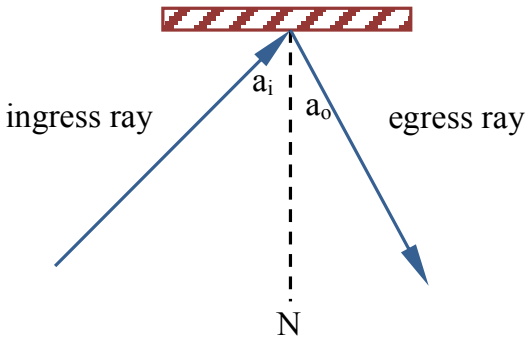


Figure 2.6: Reflection on a Surface

These factors are obtained from experimental data inside the chamber.

Reflector

In ray-tracing models, both the material and the size of the reflector matter. A larger reflector is likely to provide stronger reflection energy. We will discuss modeling the material and the size separately.

Reflector Material We experimented with reflection boards of different materials and different sizes, because the material and the size of reflector have significant impact on reflected energy. We tried four boards for reflection: (a) a 2ft $\times$ 1ft (0.61m $\times$ 0.30m) piece of acoustic ceiling board rescued from a dumpster, (b) the same board as (a) but with the board covered by aluminum foil, (c) a 4ft $\times$ 4ft (1.22m $\times$ 1.22m) piece of drywall, and (d) a 3ft $\times$ 2ft (0.91m $\times$ 0.61m) piece of copper board. The metallic surfaces (b) and (d) have higher reflection coefficients, while non-metallic materials (a) and (c) provide weaker reflections.

Consider reflection. For a single ingress ray, there can be multiple egress rays. The energy reflects along different egress directions depending on the material. As shown in Figure 2.6, N is the normal, and we denote the ingress angle and egress angle as a_i and a_o, respectively. For each

egress angle considered, the value by which the ingress ray's energy is scaled is calculated by

$$k_r = k_{ref} \times \big(\cos(a_i - a_o)\big)^p, \tag{2.1}$$

where k_{ref} is the reflection coefficient of the material, and p is material-dependent parameter that describes scattering [69]. Smaller p represents more diffusion, while infinite p means the material only provides perfect mirror reflection. The experiments we report used highly reflective material, and set $k_{ref} = 1$, and used $p = 1$.

Reflector Material Our domain model is 2-D, and so we represent the reflector as a straight line. We further approximate this straight line as a series of discrete points. Each discrete point on the reflector corresponds to the reflection of one ray, and so the more points we have for a reflector, we increase the potential for accuracy but also increase the computational cost.

2.4 Ray-Tracing Models

We will consider two ray-tracing models. The ray paths computed are the same in both models—from every point that represents the transmitter we direct rays to every point representing the reflector, and from every point representing the reflector we direct rays to all points representing the receiver.

One ray-tracing model we consider is simple, in that we only consider the path loss of a ray-path and ignore its delay. The signal strength at the receiver is obtained by summing the signal strengths contributed by all rays that reach the domain's representation of the receiver. While this technique fails to capture wave cancellation and reinforcement, it is of interest because of the potential for accelerating its execution using Graphical Processing Units [17]. Simple ray-tracing provides one data point in the accuracy/execution speed spectrum.

The difference between visible light and radio waves is wavelength, and the wavelength of 802.11a is several centimeters. When the wavelength is comparable to the dimension of objects,

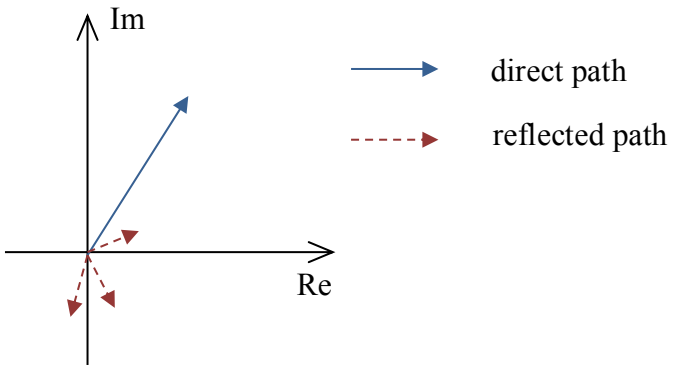


Figure 2.7: Vector Reinforcement/Cancellation Mechanism

reinforcement and cancellation of waves may be significant [50]. This phenomenon appears when there are multiple paths to the receiver, each with a different loss and delay. In our model we have multiple line-of-sight paths between sender and receiver, and also multiple reflected paths (because all rays from the sender hit all pointers representing the reflector, which then are directed to all points representing the receiver).

Cancellation

In the advanced ray-tracing model for wireless communication, a ray includes power (reflecting path loss) and delay (reflecting phase shift). The power delay of each ray at the receiver side is represented as a vector on the complex plane. The modulus of the vector represents power, while the angle represents the phase shift. When adding two rays at the receiver side, we do the complex addition. Figure 2.7 shows an example, in which the solid vector represents direct path, and dashed vectors represent reflected paths. As we can see, the reflected rays can be either constructive or destructive. The simple model can be viewed as a special case in which we use the magnitude of the vectors.

Scenario-Specific Optimization

The scenario-specific ray-tracing algorithm we used is given by Algorithm 2.1.

Algorithm 2.1 Scenario-Specific Optimized Ray-Tracing Algorithm

$d \leftarrow$ distance$(sender, receiver)$
$sum.mod \leftarrow$ pathloss$(d) \times k_{att}$
if simple-ray-tracing **then**
  $sum.arg \leftarrow 0$
else
  $sum.arg \leftarrow d/\lambda \times 2\pi$
end if

for all point i of $sender$ **do**
  for all point j of $reflector$ **do**
    for all point k of $receiver$ **do**
      $d \leftarrow$ distance$(i, j) +$ distance(j, k)
      $a_i \leftarrow$ angle(i, j)
      $a_o \leftarrow$ angle(j, k)
      $a.mod \leftarrow$ pathloss$(d) \times \big(\cos(a_i - a_o)\big)^p \times k_{ref}$
      if simple-ray-tracing **then**
        $a.arg \leftarrow 0$
      else
        $a.arg \leftarrow d/\lambda \times 2\pi - \pi$
      end if
      $sum \leftarrow sum + a/(n_{tx} \times n_{ref} \times n_{rx})$
    end for
  end for
end for

$rss \leftarrow 10 \times \log_{10}(sum.arg)$

For n_{tx} = n_{rx} = 100 (e.g., the sender and the receiver are each represented with 10 $\times$ 10 = 100 points) and n_{ref} = 100 (e.g, the reflector is discretized into 100 points), the run time of one scenario (which contains a series of sub-scenarios due to the movement of the reflector, as described in Section 0) is 24sec on our Lenovo T60 laptop (with Intel T7200 CPU and 2GB RAM).

2.5 Transmission Line Matrix Model

Overview

The transmission line matrix (TLM) model discretizes a domain into uniformly sized cells, and time into a quanta equal to the time it takes a wave to traverse a cell. It estimates the signal energy at a cell at time t as a function of the signal energies directed to it by adjacent cells at the time $t-1$. A simulation using TLM can be viewed as the repeated evaluation of a matrix-vector multiplication, where the vector holds all state variables, and one such multiplication advances simulation time by one time quanta.

In TLM, inhomogeneous space is modeled as discretized cells, and each cell is model as a homogeneous space [66]. A propagating wave can be simulated within this array of discrete event cells. A cell can change state in response to two types of events: (a) external events coming from adjacent cells, and (b) internal events when the cell is not at its equilibrium position. Different media are joined using reflection/transmission junctions that model reflection, transmission, and wave speed changes when a wave moves across a medium interface. A detailed description of the medium interface model can be found in [66].

Two simplifications are made in [66] to accelerate the execution. The first suggests that waves traveling in other types of materials (e.g., concrete or earth) can be discarded, e.g., radio waves do not transmit through materials. They present a simplified version of the junction that only propagates reflected waves implements this simplification. On the other hand, the second simplification restricts propagation calculations to the wave front by using two distinct cutoff thresholds: absolute threshold and relative threshold. Absolute threshold is a magnitude below which a grid point will not propagate any disturbance. Relative threshold is defined as a local cutoff threshold relative to the largest disturbance that has passed through the point. A cell's energy will propagate output to adjacent cells only if the output is greater than both the absolute threshold and the relative threshold.

The computational cost of TLM is determined by the number of active grid points. If the

propagation space is cluttered with objects that do not transmit radio signals, then the number of active grid points will be smaller as compared with the total number of grid points in the space. This is expected to significantly reduce the computational complexity with respect to other finite difference techniques.

Implementation Details

We applied the algorithm described in [66] to our domain and its experiments. The signal source is modeled as a continuous sinusoidal impulse, and the cell size is set to be λ/D, where λ is the wavelength of the radio wave, and D is a tunable parameter. To simulate the wave reinforcement and cancellation, we use larger D, e.g. $D = 8$, which means the wavelength is 8 cell widths. The attenuation is modeled according to Formula 10 in [66]. Figure 2.8 shows a snapshot of the wave propagation of the scenario described in Figure 2.5. Higher luminance of a grid point indicates larger displacement at this moment. The top-middle horizontal line is the reflector, and bottom-middle block is the attenuator. The sender is on the left and the receiver is on the right. As we can see, both the attenuator and the reflector behave as expected, waves reflect and are received, and some signal strength penetrates the attenuator (in this case the cones point at the receiver.)

We use different reflection coefficients for different reflector materials, and apply these to the junction model in [66]. Since the displacement of a grid point keeps changing over time while the wave is propagating, we use the following formula to calculate the average received power of a given point:

$$P = \sum \big(A(t)\big)^2/t, \tag{2.2}$$

where $A(t)$ is the displacement or amplitude of that grid point at time t. This is a discretized version of calculating the energy of a signal. We calculate the energy received at the receiver point, and convert it into decibel scale as the output of the TLM model.

The execution time of our TLM implementation is significant, even for this simple scenario. In particular, for $D = 8$, there are 1016×635 cells in the 2-D plane. It takes 6134sec to finish

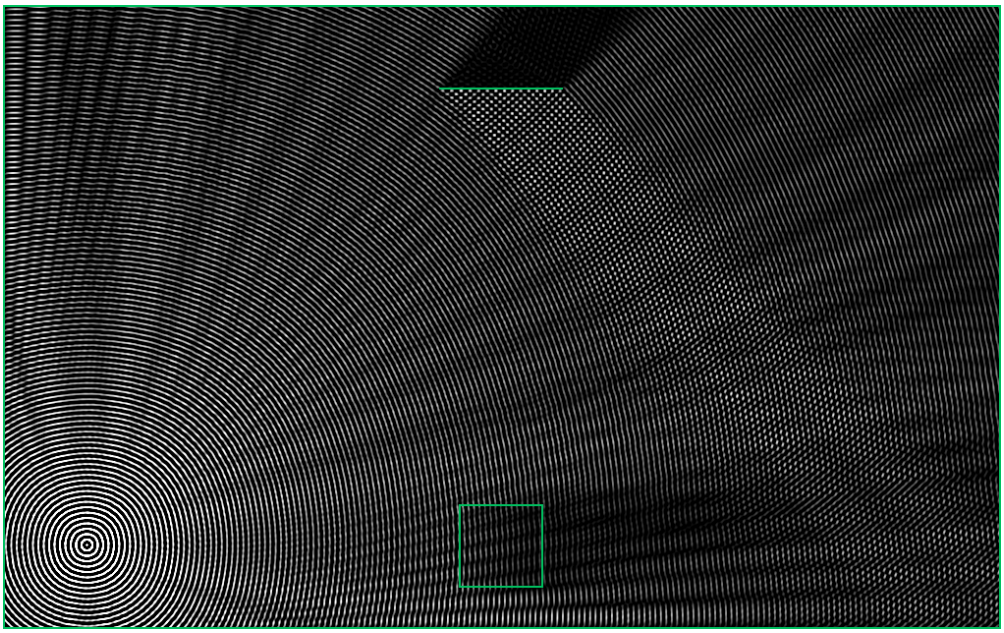


Figure 2.8: Snapshot of TLM Calculation (D=8)

a single scenario in (measured on the same Lenovo T60 laptop as mentioned in Section 2.4). The terminating condition ensures that the reflected wave reaches the receiver and that the average received power has converged.

2.6 Experimental Results

Next, we describe our experiments and how each of the wireless channel models compares with data collected inside the iWWT.

Recall the basic experimental framework of Figure 2.5. A basic experiment will position the reflector, and orient the sender-receiver either right-to-left (called Direction A) or left-to-right (called Direction B). The same radios are in the same places in all experiments, but the roles of sender and receiver are exchanged. Before an experiment the wireless signal strength is first calibrated. During the experiment 6000 packets (100 per second) are transmitted in succession, and the receiver records the received signal strength (RSS) for each, and the average is taken as the experimental result. The experiment is invalidated if there is significant variation in the RSS values (as the send-

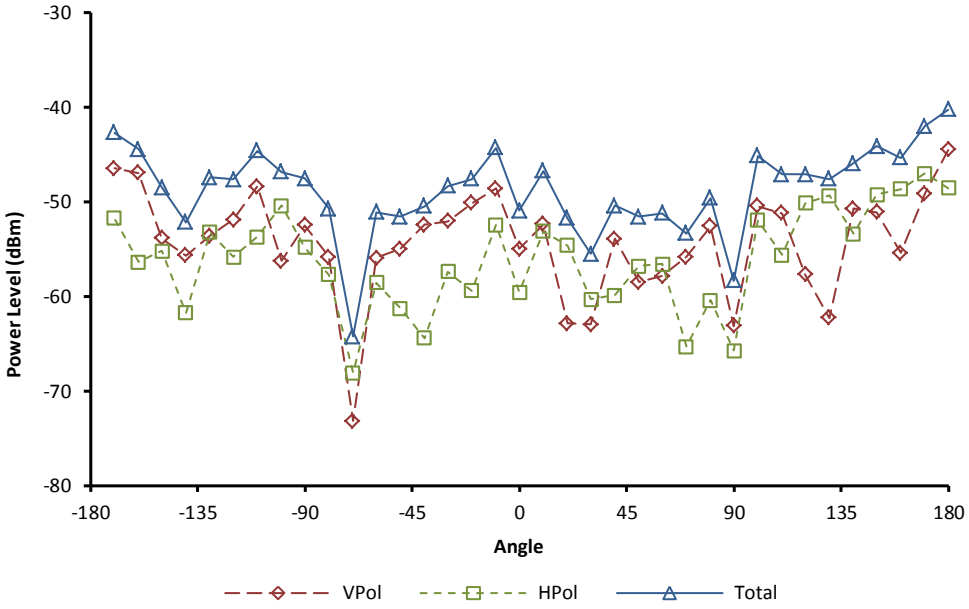


Figure 2.9: Beamform of a wireless node operating in 5.2 GHz

ing power would sometimes seemingly change in mid-run, for reasons unknown). Sensitivity to the attenuator and to the position of the reflector is determined by plotting the average received power as a function of these variables.

Before considering these sensitivity experiments, we first discuss the problem of modeling the antenna beamform.

Beamform of the Wireless Box

For better understanding the wireless boxes we use for experiments, we performed an experiment to measure its beamform profile. We placed the transmitter on a rotatable platform inside the anechoic chamber, and use a spectrum analyzer to measure signal strength at a fixed point while rotating the platform. The spectrum analyzer reports the signal strength of horizontal polar and vertical polar components respectively. We calculate the total signal strength by summing up both. The results are shown in Figure 2.9.

As we can see from Figure 2.9, the beamform of the wireless box is not very regular, varying as much as 10dB at small separation angles. Our experiments assume uniform transmission from

all angles. While this has the potential for significant error, note that in our experiments the only signals that matter are received come from 2 narrow bands (one towards the line-of-sight, one towards the reflector), and, unless one of those bands includes the large dip near -90 degrees, the variation within a band will be smaller. Still, we need to remember the beamform assumption when we consider the experimental results.

Analysis of Two-path Scenarios

We now show our results in two scenarios. Scenario 1 fixes the Y coordinate of the reflector at 10.5ft (3.20m) and moves it in the X dimension, while Scenario 2 centers the X coordinate and moves the reflector the distance of 1ft or 304.8mm in the Y dimension (from 10ft to 11ft according to Figure 2.5). These experiments modeled the wireless node as a single point, and both scenarios consider both Direction A (right-to-left) and Direction B (left-to-right). The results are shown in Figure 2.10 to Figure 2.13.

Several observations are important here. Although both radios are identical in make and model, the radio on the right consistently sends at a stronger strength—just compare the scale of the results on Direction A with those of Direction B. We find that this is caused by the radio; if we swap the position of the radios, the signal strengths swap as well. Doing wireless simulations, it might be important to include some uncertainty or variation in the basic sending strength of the radios. A second important point is that the models track the larger scale changes in the domain's X dimension than they do the Y dimension. We clearly see in Figure 2.10 and Figure 2.11 show a peak near the center that most models capture, but there is no clearly discernable pattern of models—or real data—in Figure 2.12 and Figure 2.13. Interestingly, in these latter two figures the advanced ray-tracing model and TLM have similar trend-lines, it is the real data that is at variance. As the models are based on similar simplifying assumptions about the radios, this suggests that tracking changes that are on the scale of a few wavelengths, those simplifying assumptions really matter.

The center peak in the Scenario 1 results is explainable *in part*, by the fact that the average

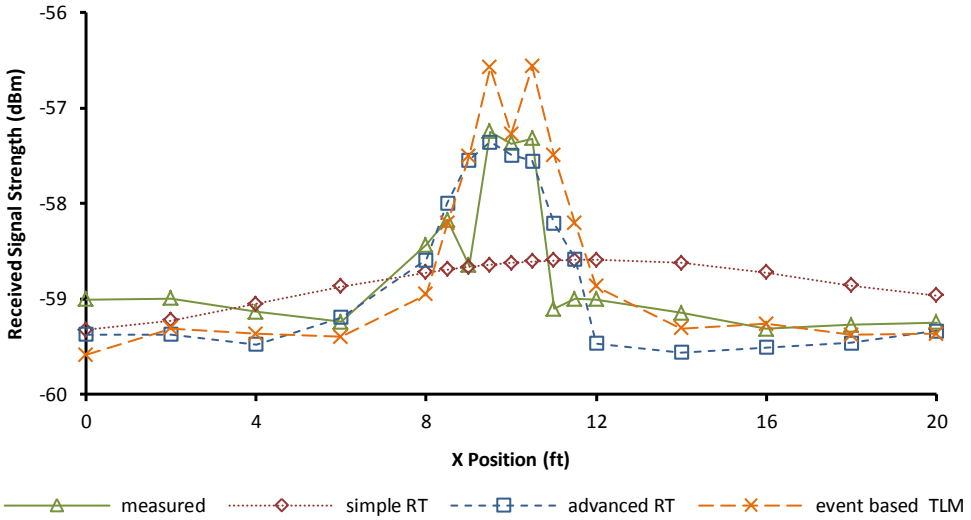


Figure 2.10: Scenario 1, Direction A

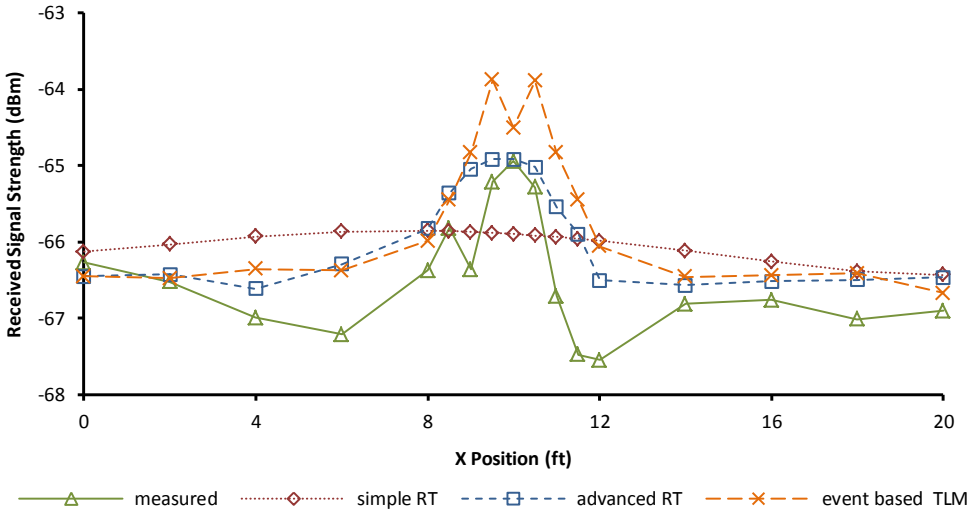


Figure 2.11: Scenario 1, Direction B

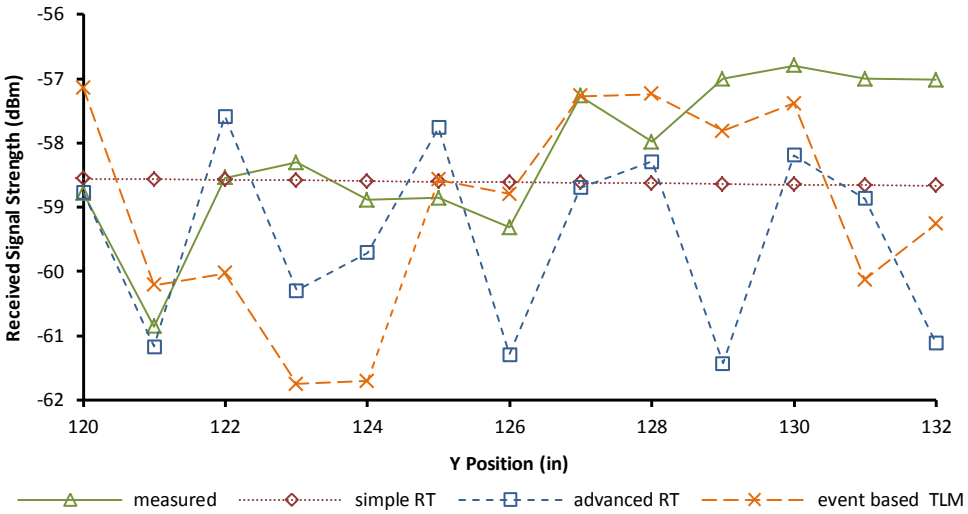


Figure 2.12: Scenario 2, Direction A

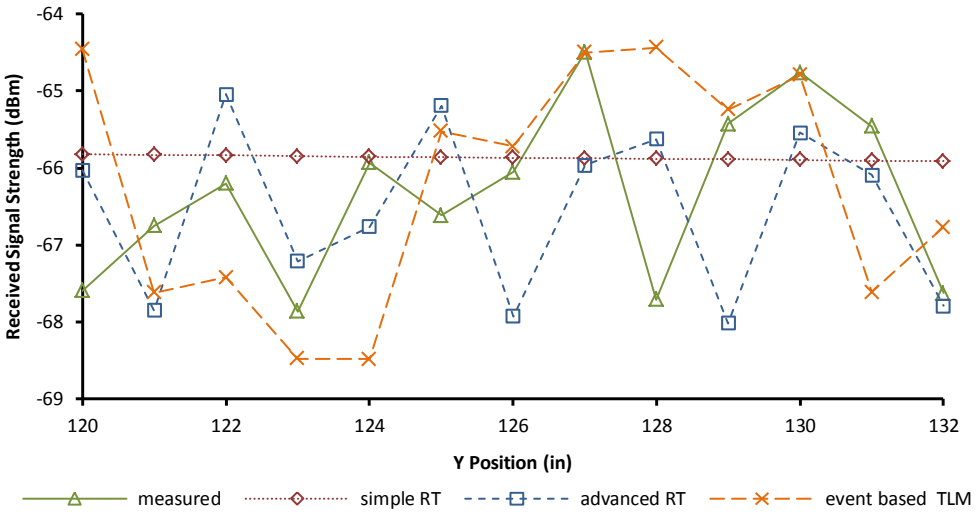


Figure 2.13: Scenario 2, Direction B

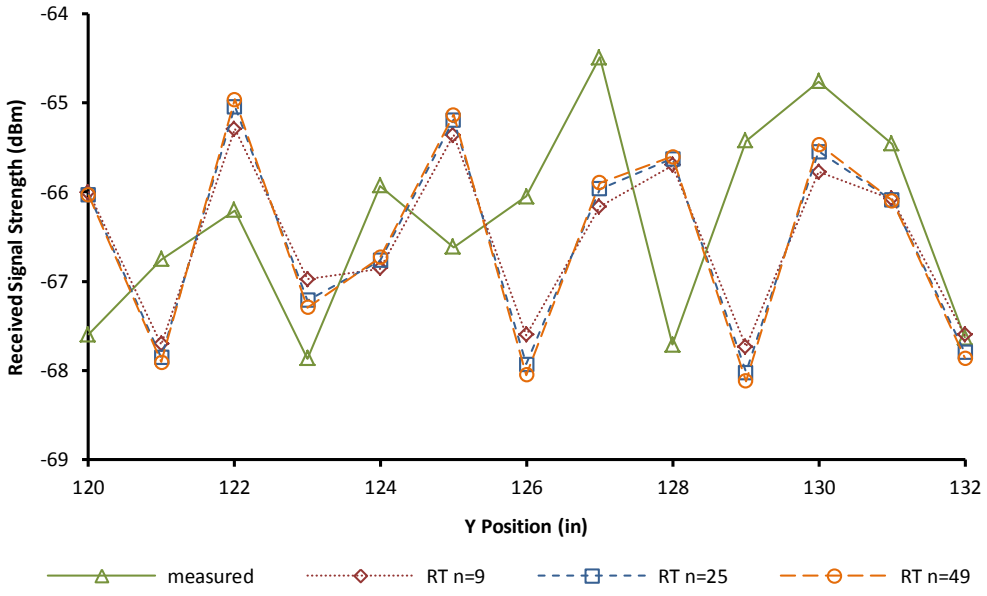


Figure 2.14: Different Resolutions of the Ray-Tracing Model

length of a ray from sender to receiver is shorter when the reflecting panel is centered than it is when the reflector is offset. However, this aspect is captured by the simple ray-tracing model, which indeed predicts larger values when centered, but not larger values that are so significantly larger. The height of the peak at the center and the trough (in the real data) are explainable by wave reinforcement and cancellation; advanced ray-tracing and TLM both capture the peak, but are rather weaker on capturing the troughs. It is also significant that variation in prediction among all these models is on the order of 2dB.

Resolution of the Ray-Tracing Model

As mentioned earlier, we discretize the reflector as a number of points. In general, the more points we have, the higher precision we can potentially achieve. Figure 2.14 shows a comparison of the ray-tracing models of different resolutions, in which n is the number of points we use to represent the reflector. Again, it is based on Scenario 2 Direction B earlier.

The figure shows that the three resolutions considered yield predictions that are nearly identical. This suggests that $n = 9$ is sufficient for the scenario we consider. For this specific scenario

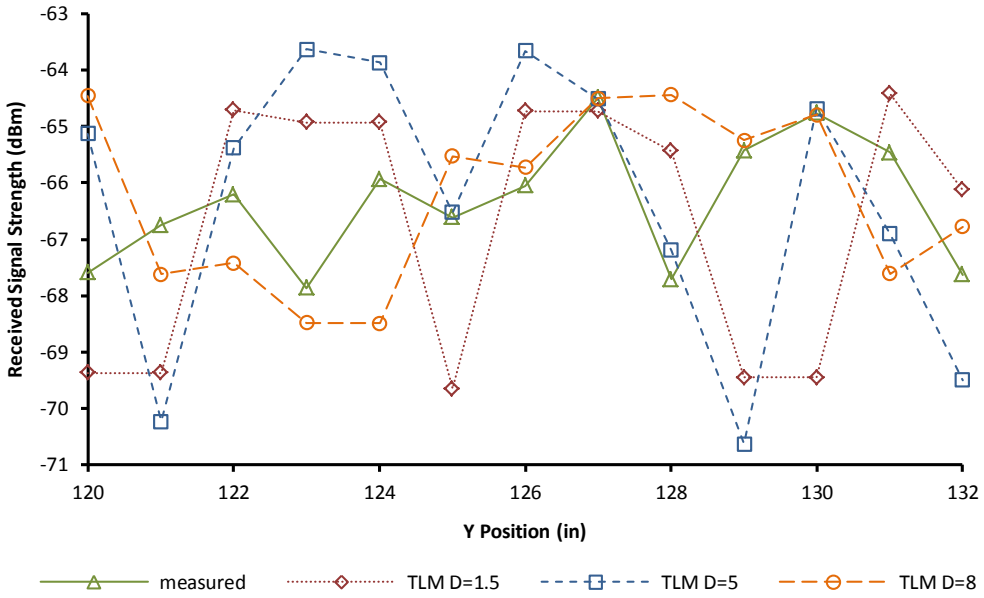


Figure 2.15: Different Resolutions of the TLM Mesh

execution time is approximately a linear function of the number of points on the reflector. The dependency of execution time on scenario features is a topic of future work.

Resolution of the TLM Mesh

We also study the impact of cell size used in the TLM model, varying cell width of /D by considering $D = 8$, $D = 5$, and $D = 1.5$. Figure 2.15 illustrates the various predictions, from Scenario 2 Direction B. Discretization $D = 8$ does the best job of tracking the real data, while the coarser ones have 4 and 5dB errors at some positions. Proponents of event-based TLM advocate using it with cell sizes that are significantly larger than the wave-length.

Figure 2.16 shows an example where a cell width is 5 wavelengths, looking at how its predictions behave while the reflector is moved in the X dimension (based on Scenario 1 Direction A earlier). We see that indeed the predictions track the change in path length due to the movement, although the error in its prediction is significantly larger than errors of the other models, as much as 10dB at some positions. To a first approximation execution time of one TLM time-step is proportional to the number of cells, which stronger weights the accuracy-tradeoff decision towards

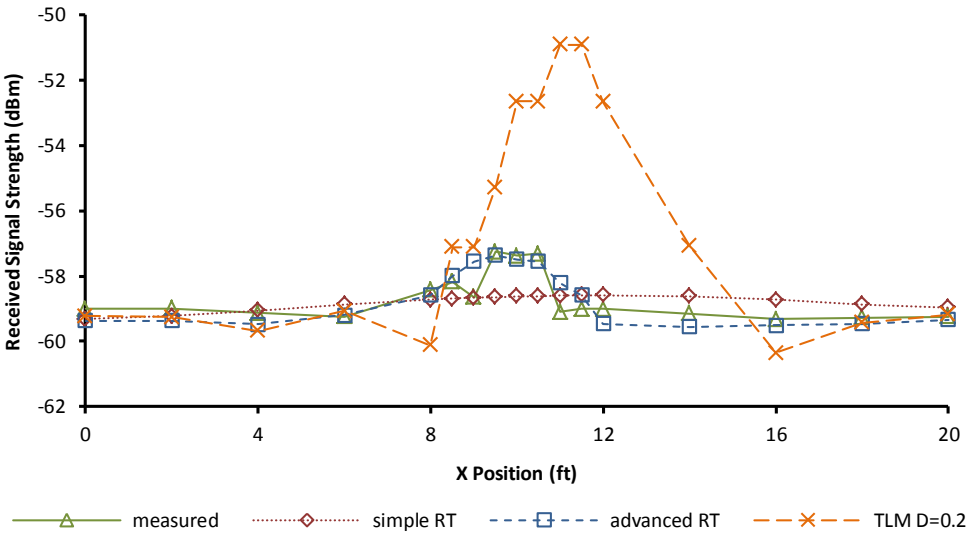


Figure 2.16: Coarse TLM mesh for Scenario 1, Direction A

accepting as much error as the application will admit.

2.7 Related Work

Questions of the accuracy versus execution speed tradeoff have long been of interest in wireless simulation. Studies of the tradeoff in the space of channel models include [37] [74]. One of the issues flagged in that work particularly is derivation of the threshold beyond which signal strength can be ignored.

Ray-tracing models of wireless channels has been explored for years, e.g. [51] [72] [82]. For example, Seidel et al. propose a ray-tracing channel model to predict propagation within buildings for Personal Communication System (PCS) in [72]. Their model predicts path loss based on a building blueprint representation as well as the locations of transmitters and receivers. In this model, they consider three kinds of ray paths: transmitted (penetrated), reflected, and diffracted ray paths. They model the power delay of each ray path separately on a 2-D plane, and then combine them at the receiver side. Finally, they compare the predicted propagation data with measured data, to validate their model, and achieve a 5dB accuracy. Yang et al. [82] also consider

a ray-tracing model to predict path loss for indoor environment. Their model is similar to [72] but performed it in a 3-D space; their validation studies yield errors on the order of 10dB, but their domain model is more complex.

The method of the transmission line matrix (TLM) numerically simulates the propagation of a radio wave through a discretized domain. While this method has been known for some time, it has received recent new interest in a so-called "event-driven" formulation [66] [67] wherein signal propagation through a cell is terminated in the solution once the radio energy in a cell becomes too weak. Kuruganti et al. [49] compare ray tracing, finite difference time domain (FDTD), and the event driven transmission line matrix models as tools for making site specific path loss predictions in cluttered environments whose objects are opaque to radio. Their experiments use a small indoor fixed domain, vary the placement of radios, and consider the response of three models. Prediction accuracy of TLM is observed to be on the order of 5dB.

An alternative approach to (slow) channel simulation in software is faster channel emulation in hardware, e.g., [14] [46]. With physical channel emulation, wireless signals from the senders are intercepted by the emulation controller. The controller does signal processing to emulate signal propagation, and finally sends the signal back to the receivers. Hardware acceleration adds another dimension to explore in the accuracy/fidelity space.

2.8 Chapter Summary

We validated ray-tracing and TLM radio channel models within an anechoic chamber that isolates the experiments from outside interference. For both model types we varied resolution (hence accuracy). We observed errors on the order of 2dB for all ray-tracing models, and TLM models when the cell width is a fraction of a wavelength. Errors in the TLM increase to 10dB on a coarser grid, but with significantly faster execution time. The scenarios possible within the anechoic chamber are so limited that no definitive conclusion about the relative speeds of ray-tracing and TLM can be drawn from our study. We do know the difference depends very much on the resolutions chosen

for both methods and on the features of the domain; future work will attempt to quantify these distinctions.

Both methods track changes in the placement of the reflector that are 10 or more wavelengths in magnitude. However experiments that moved the reflector fewer than 5 wavelengths yield real data whose variations we could not explain, and model predictions that did not track those variations particularly well. The models do assume uniform beamform, but data we collected show significant—albeit seemingly unstructured—variation. On embarking on this project we expected that the isolation of the anechoic chamber would allow us to predict received signal strength with great accuracy. What we found instead is that lack of detailed knowledge about the radios involved introduces uncertainty. While seemingly a negative result, it is actually positive in a practical sense: since the errors that anechoic chamber eliminates are small relative to errors introduce by model uncertainty, future validations need not be attempted within such a chamber. Due diligence in isolating an experiment from outside radio interference will suffice.

Chapter 3

Virtual Time System

Simulation and emulation are commonly used to study the behavior of communication networks, owing to the cost and complexity of exploring new ideas on actual networks. Emulations executing real code have high *functional* fidelity, but may not have high *temporal* fidelity because virtual machines usually use their host's clock. A host serializes the execution of multiple virtual machines, and time-stamps on their interactions reflect this serialization. In this chapter we improve temporal fidelity of the OS level virtualization system OpenVZ by giving each virtual machine its own virtual clock. The key idea is to slightly modify the OpenVZ and OpenVZ schedulers so as to measure the time used by virtual machines in computation (as the basis for virtual execution time) and have Linux return virtual times to virtual machines, but ordinary wall clock time to other processes. Our system simulates the functional and temporal behavior of the communication network between emulated processes, and controls advancement of virtual time throughout the system. We evaluate our system against a baseline of actual wireless network measurements, and observe high temporal accuracy. Moreover, we show that the implementation overhead of our system is as low as 3%. Our results show that it is possible to have a network simulator driven by real workloads that gives its emulated hosts temporal accuracy.

3.1 Virtual Time in Network Emulation

The research community has developed many techniques for studying diverse communication networks. Evaluation based on any methodology *other than* actual measurements on actual networks raises questions of fidelity, owing to necessarily simplifications in representing behavior. An effec-

tive way to accurately model the behavior of software is to actually run the software [15] [73] [77] [80], by virtualizing the computing platform, partitioning physical resources into different Virtual Environments (VEs), on which we can run unmodified application code [18] [79]. However, such emulations typically virtualize execution but not time. The software managing VEs takes its notion of time from the host system's clock, which means that time-stamped actions taken by virtual environments 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. 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.

In this chapter, we present a virtual time system that gives virtualized applications running under OpenVZ [4] the temporal appearance of running concurrently on different physical machines. This idea is not completely unique, related approaches have been developed for the Xen [24] [36] system. Xen and OpenVZ are very different, and so are the approaches for virtualizing time. Xen is a heavy-weight system whose VEs contain both operating system and application. Correspondingly Xen can simultaneously manage VEs running different operating systems. By contrast, all VEs under OpenVZ (called "containers" in OpenVZ parlance) use and share the host operating system. In Xen virtualization starts at the operating system whereas in OpenVZ virtualization starts at the application. There are tradeoffs of course, we are interested in OpenVZ because it scales better than Xen, as OpenVZ emulation can easily manage many more VEs than can Xen. We believe we are first to introduce virtual time to OpenVZ; by doing so we are able to construct large scale models that run real application code, with rather more temporal accuracy than would be enjoyed without our modifications.

We implement our virtual time system by slightly modifying the OpenVZ and Linux kernels. The OpenVZ modifications measure the time spent in bursts of execution, stop a container on any action that touches the network, and gives one container (the network emulator) control over the scheduling of all the other containers to ensure proper ordering of events in virtual time. Modifications to the Linux kernel are needed to trap interactions by containers with system calls related to

time, e.g., if a container calls `gettimeofday()`, the system should return the container's virtual time rather than the kernel's wall clock time — but calls by processes other than OpenVZ's ought to see the kernel's unmodified clock time.

Our time virtualization is not exact. However, comparison with experiments that use real time-stamped data measured on a wireless network reveal temporal errors on the order of 1ms — which is not large for this application. We also measure the overhead of our system's instrumentation and find it to be as low as 3%. In addition, our method is more efficient than the time virtualization proposed for Xen [36]. That technique simply scales real time by a constant factor, and gives each VM a constant sized slice of virtualized time, regardless of whether any application activity is happening. Necessarily, Xen VEs virtualized in time this way can only advance more slowly in virtual time than the real-time clock advances. Our approach is less tied to real time, and in principle can actually advance in virtual time faster than the real-time clock, depending on the number of containers and their applications.

3.2 System Architecture

We begin by providing an introduction of OpenVZ system, and then explain the architecture of our system.

Overview of OpenVZ

OpenVZ provides container-based virtualization for Linux [4]. It enables multiple isolated execution environments (called Virtual Environments, VEs, or containers) within a single OS kernel. It provides better performance and scalability as compared with other virtualization technologies. Figure 3.1 shows the architecture of OpenVZ. A virtual environment looks like a separate physical machine. It has its own process tree starting from the `init` process, its own file system, users and groups, network interfaces with IP addresses, etc. Multiple VEs coexist within a single physical machine, and they not only share the physical resources but also share the same OS kernel. All

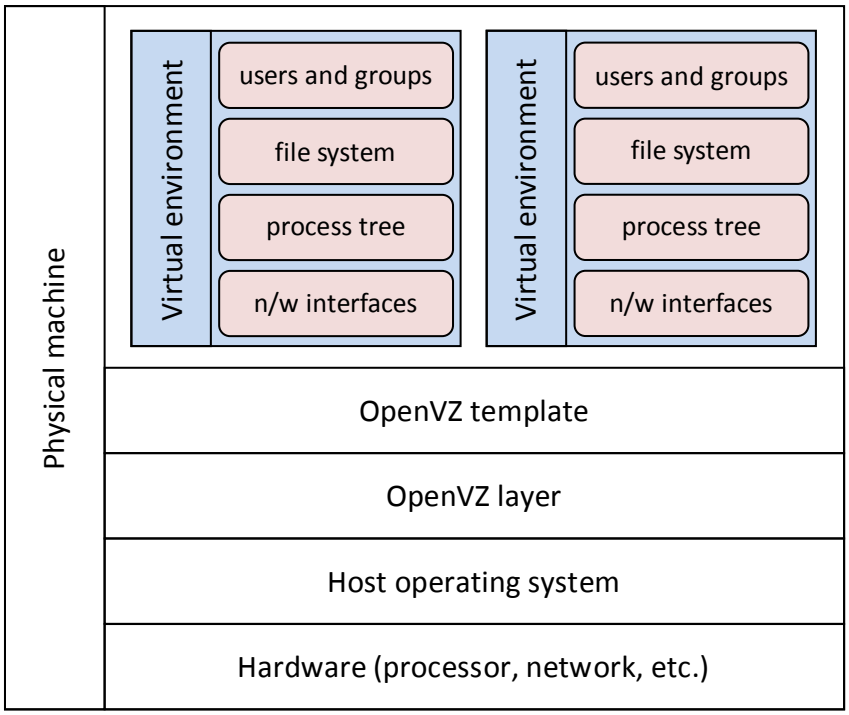


Figure 3.1: Architecture of OpenVZ

VEs have to use the same version of the same kernel.

A VE is different from a real OS. A VE uses fewer resources. For example, a newly created Ubuntu VE can have fewer than 10 processes. A VE has limited function compare with a real machine, e.g., it is prohibited from loading or unloading kernel modules inside a VE. Finally, the Linux host operating system provides all kernel services to every VE; the operating system is shared.

OpenVZ offers two types of virtual network interfaces to the VEs, one called a virtual network device (or `venet` in OpenVZ parlance) and the other called a virtual Ethernet device (or `veth` in OpenVZ parlance) [4]. A virtual network device has lower overhead, but with limited functionality, serving simply as a point-to-point connection between a VE and the host OS. It does not have a MAC address, has no ARP protocol support, no bridge support, and no possibility to assign an IP address inside the VE. By contrast, a virtual Ethernet device has slightly higher (but still very low) overhead, but it behaves like an Ethernet device. A virtual Ethernet device consists of a pair of

network devices in the Linux system, one inside the VE and one in the host OS. Such two devices are connected via Ethernet tunnel: a packet goes into one device will come out from the other side.

The OpenVZ CPU scheduler has two levels. The first level scheduler determines which VE to give timeslice to, according the VE's CPU priority and limit settings. The second level scheduler is a standard Linux scheduler, which decides which process within a VE to run, according to the process priorities.

System Designs

The architecture of our OpenVZ-based emulation system is illustrated by Figure 3.2. For a given experiment a number of VEs are created, each of which represents a physical machine in the scenario being emulated. Applications that run natively on Linux run in VEs without any modification. The sequencing of applications run on different VEs is controlled by the Simulation/Control application, which runs on host OS (or `VE0` in OpenVZ parlance). Sim/Control communicates to the OpenVZ layer to control VE execution so as to maintain temporal fidelity. For instance, a VE that is blocked on a socket read ought to be released when data arrives on that socket. Sim/Control knows when the data arrives, and so knows when to signal OpenVZ that the blocked VE may run again.

Under unmodified OpenVZ all VEs share the *wallclock* of the host computer (accessed through the shared host operating system). In our virtual time system, each VE has its own *virtual clock* (denoted as `vclk` in Figure 3.2), while the host OS still uses the wallclock (denoted as `wclk`). Different virtual clocks advance separately, but all of them are controlled by the network simulator via the virtual time kernel module (`V/T module` in the figure).

Sim/Control captures packets sent by VEs and delivers them to destination VEs at the proper time ("proper time" being a function of what happens as the network is simulated to carry those packets). Sim/Control also controls the virtual times of VEs, advancing their virtual times as a function of their execution, but also blocking a VE from running, in order to prevent causal violation. For example, if a packet should arrive at a VE at virtual time t, but the virtual time of

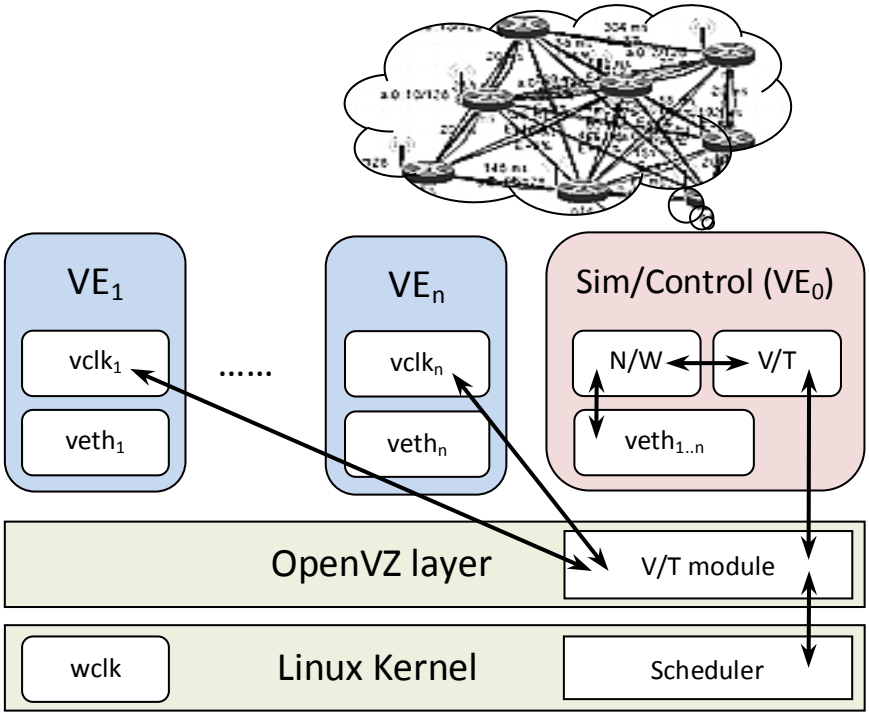


Figure 3.2: Architecture of Network Emulation with Virtual Time

that VE is already $t + 1$, a causal violation occurs because the application has missed the packet and may behave differently than expected. Sim/Control is responsible for stopping this VE at virtual time t, until the packet arrives. This is accomplished by modifying the scheduler, as we will describe in Section 3.3.

Sim/Control consists of two cooperating subsystems: 1) network subsystem (denoted as `N/W` in Figure 3.2) and 2) virtual time subsystem (denoted as `V/T`). For instance, when a packet sent by VE1 to VE2, it is captured by Sim/Control, which has to know the virtual sending timestamp of that packet in order to know when it entered the network. After the simulator determines the virtual arrival time of the packet at VE2, the simulator must ensure that VE2 has advanced far enough in simulation time to receive that packet, or that VE2 is blocked waiting for a packet, and so needs to be released to run.

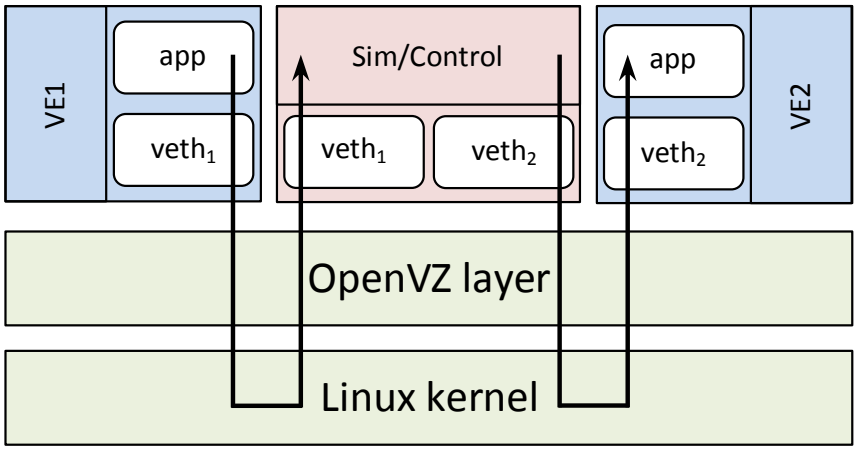


Figure 3.3: Packet Traverse Route in Emulation

Sim/Control

The Sim/Control process captures VE packets, simulates their travel within the imagined network, and delivers them to their destinations. Packet capture is accomplished using the OpenVZ virtual Ethernet device (`veth`). When an application within a VE sends a packet via its `veth` interface, the packet appears at `veth` in the host OS due to the virtual Ethernet tunneling. Sim/Control monitors all `veth` interfaces to capture all packets sent by all VEs. Similarly, when it wants to deliver a packet to a VE, it just simply sends the packet to the corresponding `veth` interface. The packet tunnels to the VE's corresponding `veth` interface, where the application receives it. The packet travel route is shown in Figure 3.3.

Sim/Control needs to cooperate with the virtual time subsystem when VEs are either sending packets or receiving packets. For example, in real system, blocking socket sends (e.g., `sendto()`) are returned after the packets have been taken care of the underlying OS. Correspondingly, in emulation, the process should perceive comparable amount of elapsed time after the call returns. This is done by trapping those system calls, suspending the VE, have Sim/Control figure out the time at which the packet is taken care, and then return control to the VE, at the corresponding virtual time. Similarly, when an application is blocked waiting to receive a packet, it should be unblocked at the virtual time of the packet's arrival. The comparison between real network operations and emulated

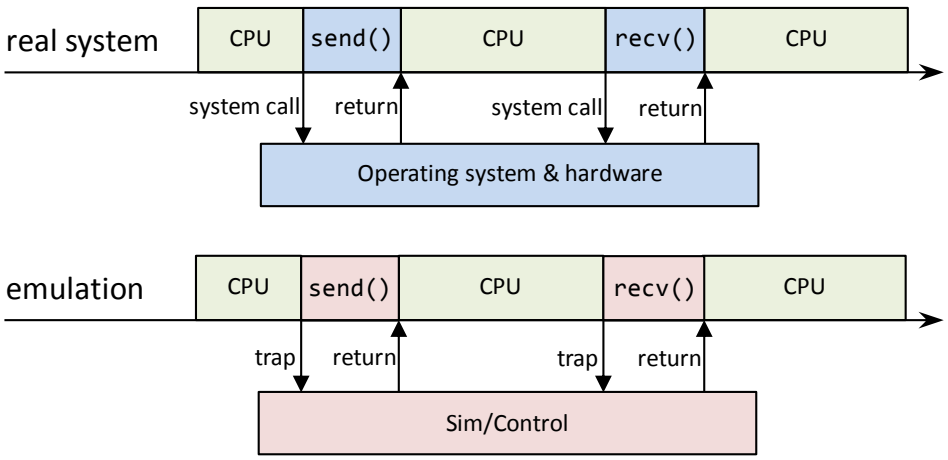


Figure 3.4: Real Network Operations vs. Simulated Network Operations

ones is shown in Figure 4. As long as the processes perceive comparable elapsed time after system calls are returned and the network simulation gives high enough fidelity for the system measures of interest, this approach is viable. Our approach to integrating the network simulation allows us to include any one of a number of physical layer models. Detailed technical issues are discussed in Section 3.3.

Virtual Time System

The responsibility of the virtual time subsystem includes advancing virtual clocks of VEs and controlling the execution of VEs. From the operating system's point of view, a process can either have CPU resources and be running, or be blocked and waiting for I/O [75] (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 our system, the virtual time of a VE advances in two ways. At the point a VE becomes suspended, the elapsed wallclock time during its execution burst is added to the virtual clock. This is shown in Figure 3.5.

The situation is different when the application within a VE interacts with the I/O system. Our modified OpenVZ kernel traps the I/O calls, Sim/Control determines the I/O delay and adds that to the VE's virtual clock, and then returns the I/O request to unblock the process. As shown in

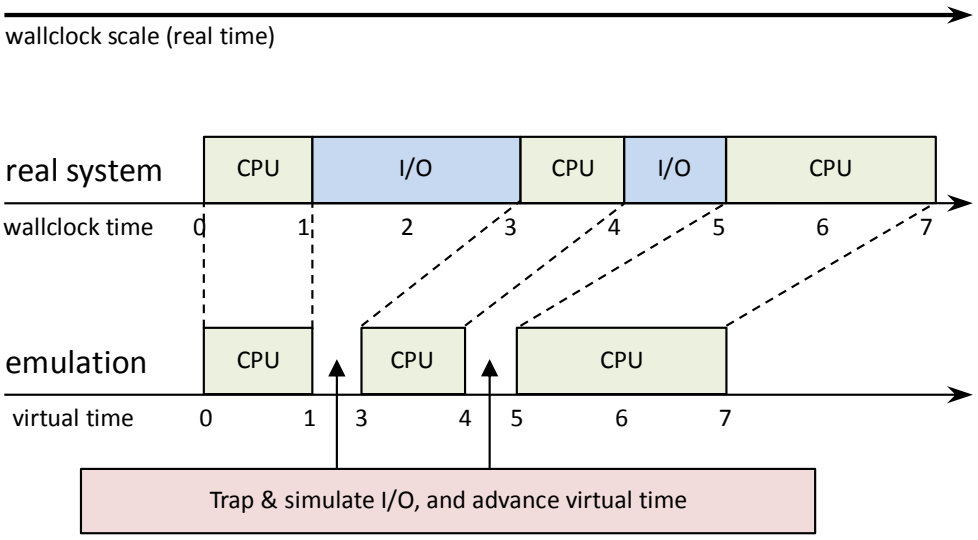


Figure 3.5: Wall Clock Time Advancement vs. Virtual Time Advancement

Figure 3.5, when I/O delay is accurately simulated, the virtual clock will have the same value as wall clock, and therefore the application perceives the same elapsed time. However, the real elapsed time depends on the time spent on emulating such I/O, which depends on the model and the communication load.

It is sometimes necessary to block a running VE in order to prevent casual violation. An example is when an application queries for incoming I/Os, e.g. a non-blocking socket call using `select()` [21]. Even though there may be no pending packets at that wallclock moment, it is possible still for a packet to be delivered with a virtual arrival time that is no greater than the virtual time t of the `select()` call, because the virtual clock the sending VE may be less than t. Therefore, when an application makes a non-blocking socket receive call at virtual time t, our system suspends it until we can ensure no packets can arrive with time-stamps less than or equal to t. On a blocking call we need to ensure that the right packet unblocks the VE, and so the same logic applies — before the VE is released at time t, we ensure that any packet with time stamp t or small has already been delivered. Implementation details are given in Section 3.3.

3.3 Implementation

We next present implementation details of our virtual time system, and discuss some related issues. As shown in Figure 3.2, virtual time management needs kernel support, therefore modification to OpenVZ kernel is necessary. We try to keep the modifications simple. The kernel implements only some primitive operations, while Sim/Control calls these operations to control sequencing of VE execution. Sim/Control runs at user level on the host OS (`VE0`), and communicates with the kernel through new system calls we have implemented. We have chosen system calls to be the communication channel between user space and kernel because of its high efficiency. We placed Sim/Control in user space in order to keep the kernel safe, but the frequent communication between user and kernel raises the question of overheads. Section 3.4 discusses our measurement of this, found in our experiments to be small.

Modification of OpenVZ Kernel

OpenVZ Scheduler Scheduling VEs properly ensures the correctness of the emulation. To support this we modified the OpenVZ scheduler so that Sim/Control governs execution of the VEs. By making system calls to the kernel, Sim/Control explicitly gives timeslices to certain VEs; a VE may run only through this mechanism.

The typical scheduling sequence of emulation is shown in Figure 3.6, showing how Sim/Control and VEs take turns running. We refer Sim/Control timeslice together with all the subsequent VE timeslices before the next Sim/Control timeslice as one *emulation cycle*. At the beginning of a cycle, all VEs are blocked. In its timeslice Sim/Control pushes all due events to VEs (such as packet deliveries, details in Section 3.2), and then decides which VEs can have timeslices and notifies the kernel that they may run. Causal constraints may leave any given VE blocked, but Sim/Control will always advance far enough in virtual time so that at least one VE is scheduled to run in the emulation cycle. VE executions within an emulation cycle are logically independent of each other, and so their execution order does not matter.

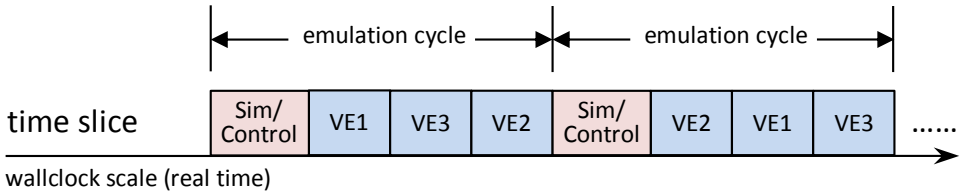


Figure 3.6: Scheduling of VEs and Sim/Control (Example of 3 VEs)

An executing VE is suspended when either it uses up the timeslice or when it makes a system call that is trapped (typically one that interacts with the communication system). Such system calls includes network send and receive calls, as discussed in Section 3.2. Once a VE makes such special calls, it is blocked immediately and cannot run any more within this emulation cycle. After a VE stops, the actual time it executed will be added to its virtual clock.

This discussion summarizes ***Rule #1****: A VE can run only after Sim/Control has released it, and will then suspend when either its timeslice is consumed, or it executes a trapped system call that interacts with the I/O subsystem.*

Trap Network Related System Calls We first discuss socket send calls, both blocking and non-blocking. As pointed out by Section 3.2, blocking socket sends should be returned after a virtual time equivalent to the time required to transmit the packet. In unmodified OpenVZ, such system calls are returned almost immediately, as the virtual Ethernet device handles packets at an extremely high speed. However, the time elapsed in real systems depends on the underlying physical layer. It can be in the order of microseconds for some gigabit Ethernet [44], but can be as large as several milliseconds for some wireless networks. We modified OpenVZ to give Sim/Control the responsibility of returning those system calls. The VE is suspended at the point of the call, and its current virtual time is updated. Since the packet was presented to the virtual Ethernet interface it tunnels almost immediately to be reflected at its corresponding interface in Sim/Control. Therefore at the beginning of the next emulation cycle Sim/Control observes the packet and marks its send time as the virtual time of its suspended source. Once the packet departure has been fully simulated (and this may take some number of emulation cycles, depending on the network model

and the traffic load on the simulator), Sim/Control will know to release the suspended sender. We likewise suspend a VE that makes a non-blocking send, but just to obtain the packet's send time. In this case Sim/Control immediately releases the sender to run again in the following emulation cycle.

Now consider socket receive calls. As discussed in Section 3.2, an application that makes a socket receive call (either blocking or not) is suspended *before* it looks at the receive buffer in order to ensure that the state of the receive buffer is correct for that virtual time. The VE must wait at least until the next cycle, at which point Sim/Control will either release it to run, or not, depending on whether there is a threat of violating causality constraints. Once the VE is released to run again it looks at the receive buffer and responds to the previous socket receive call with normal semantics (possibly blocking again immediately, if it is a blocking receive that finds no packet in the buffer).

This discussion summarizes ***Rule #2****: a VE is always suspended upon making a network related system call.*

Other Kernel Modifications Other kernel modifications are also necessary. This includes trapping `gettimeofday()` system calls and returning virtual times to the application, and basing kernel timers on virtual time. The implementations are entirely straightforward and need no further comment.

VE Scheduling Control

Sim/Control runs in user-space in the host OS. With the support of the kernel, it only needs to maintain a simple logic to control VE execution. The algorithm used is described in Algorithm 3.1, and it is a simple variation of a conservative parallel discrete event simulation (PDES) synchronization method [34] [61]. Sim/Control maintains its own virtual clock `current_time`. Conceptually, in every timeslice of an emulation cycle, Sim/Control does the following one by one: (1) buffers packets sent by VEs during the last cycle (line 10-15), (2) simulates the network and pushes due events to VEs (line 17-23), (3) decides which VEs can run in the next cycle (line 25-30), (4) ad-

vances virtual clocks if no VEs can run (line 32-36), and finally (5) yields the processor to let VEs run (line 39-40). In step (3), an event is considered "due" if its virtual time is no greater than the virtual time of the VE to which it is pushed. We will comment more on this in our section on error analysis.

As shown in Algorithm 3.1, the Sim/Control calls the network simulator program `nw_sim` to simulate the network (line 17 and 33); this function call needs some explanation. The first parameter is the current status of the network (stored in the VE data structure), a status that changes as the simulation executes. The second and third parameters indicate the desired start time and end time of simulation. The fourth parameter is a flag, explained later. The outcomes of `nw_sim` are the events to be returned to VEs (both finished transmissions and packet receptions) within the desired time interval, and they are stored back into the network status VE. With this information, the Sim/Control knows which system calls to return and which packets to deliver within a single timeslice (line 20-21).

The fourth parameter of `nw_sim` interface is a flag which tells the simulator whether to stop when such return event *first occurs*. When the flag of `nw_sim` is set to false, network is simulated for the given time interval (e.g. line 17) and returns the virtual end-time of its execution period. When the flag is set to true, the network simulator finds the exact time of the next return event (e.g. line 33), stops and returns that time. When the emulator detects that no VEs can run, it advances its current virtual clock to the point when next event happens. Such event can be either a packet transmission or a kernel timer expiration, whichever happens first (line 32-33).

Finally, there is a tunable parameter δ in line 26 used to control how tightly virtual clocks of different VE are synchronized. In the following section we show how an application running multiple threads in a VE can have different behavior in our system that it does in a real system. The smaller δ is, the smaller the potential error of that difference can be. Our experiments use a value of 1msec, which is the minimum possible value in our current platform.

Algorithm 3.1 Logic for Controlling VE Execution

```
1:  for all VE_i do
2:      init_ve_status(VE_i)
3:  end for
4:  current_time ← 0
5:
6:  while true do
7:      wait_until_all_ves_stop()
8:      last_time ← current_time
9:      current_time ← min_i(VE_i.time)
10:     for all VE_i do
11:         for all packet p sent by VE_i do
12:             p.send_time ← VE_i.time
13:             buffer_packet(VE_i, p)
14:         end for
15:     end for
16:
17:     nw_sim(VE, last_time, current_time, false)
18:     while true do
19:         for all VE_i do
20:             return_due_send_calls(VE_i)
21:             deliver_due_packets(VE_i)
22:             fire_due_timers(VE_i)
23:         end for
24:
25:         for all VE_i do
26:             VE_i.can_run ← VE_i.ready and VE_i.time < current_time + δ
27:         end for
28:         if at_least_one_VE_can_run then
29:             break
30:         end if
31:
32:         next_time ← min_j(virtual_timer_j.exp)
33:         current_time ← nw_sim(VE, current_time, next_time, true)
34:         for all VE_i do
35:             VE_i.time ← max(VE_i.time, current_time)
36:         end for
37:     end while
38:
39:     release_VEs()
40:     sched_yield()
41: end while
```

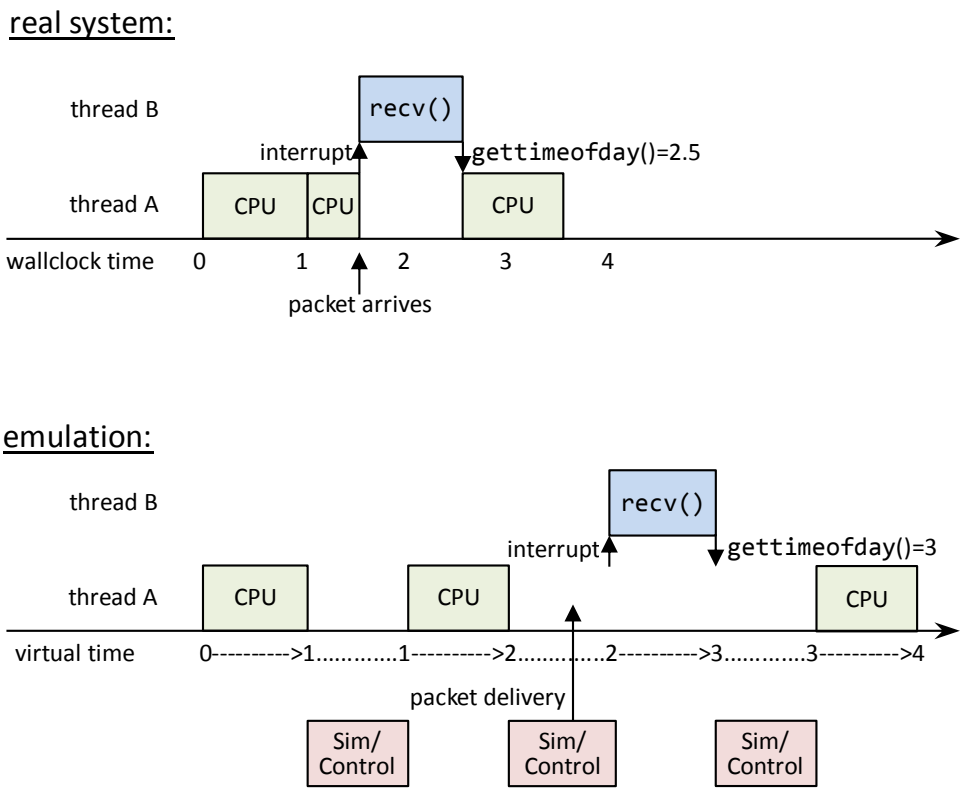


Figure 3.7: Error in Virtual Time of Packet Delivery

Error Analysis

According to Algorithm 3.1, an event is delivered to a VE *no earlier* than it should be, in the sense that if an event time is t, the VE notices it at a time u that is at least as large as t. Since we suspend a VE at all points where it interacts with the network, if the VE is single-threaded it will be insensitive to the difference between t and u because its behavior is not affected by the arrival at t. This is not the case however if an application is multi-threaded, as we show below.

Consider an application which consists of two threads. Thread A does CPU intensive computation, while Thread B repeatedly receives packets using blocking socket call `recvfrom()`, and calls `gettimeofday()` immediately after it receives a packet. In a Linux system, when there is an incoming packet, Thread B will be immediately be run, pre-empting Thread A. As a result, this application can get the exact time of packet arrival. This is illustrated in Figure 3.7.

Now consider the same application running on our virtual time system. Whenever Thread B makes the socket receive call, the *whole application* — both threads — is blocked by Rule #2. After Sim/Control releases the application to run again, Thread A will take off. However, once the application gets the CPU, it uses the entire timeslice because Thread A keeps doing computation. Meanwhile Sim/Control is waiting its turn and so packets are *not* delivered to wake up Thread B until that turn comes around, after the actual delivery time. In this case, the error in that delivery time can be *at most* as large as the timeslice the application is given to. The comparison is shown in Figure 3.7.

We summarize the above case as an instance of an *interrupt-like* event. The key problem are situations where in the real system a process is interrupted immediately, whereas in the emulation the interrupting event is not recognized until Sim/Control gets a timeslice. We can reduce this error by reducing the length of the timeslice, but this of course increases the overhead by increasing context switching [75]. The tradeoff between behavioral accuracy and execution speed is a common tradeoff in simulation [58]. Section 3.5 presents detailed analysis and results of variable timeslice.

3.4 Evaluation

This section provides our experimental results that demonstrate the performance of our virtual time system.

Experimental Framework

In order to validate the emulated results, we compare them with that we obtained in Section 2.2 [88] within the Illinois Wireless Wind Tunnel (iWWT). The iWWT [78] is built to minimize the impact of environment in wireless experiments. It is an electromagnetic anechoic chamber whose shielding prevents external radio sources from entering the chamber; and whose inner wall is lined with electromagnetically absorbing materials, which reflect minimal energy. This gives us a simu-

lated "free space" inside the chamber, which is ideal for conducting wireless network experiments.

We run the same application as we used in Section 2.2 [88] [85] within our emulator. We notice the hardware difference between the machine on which we run our emulator, and the devices we used to collected data inside the chamber. Specifically, our emulator is running on a Lenovo T60 laptop with Intel T7200 CPU and 2GB RAM (if not otherwise specified), while we used Soekris Engineering net4521 [6] with mini-PCI wireless adapters plugged in as wireless nodes inside the iWWT. Due to the difference in processors, applications running on the Soekris box should observe longer elapsed times than the Lenovo laptop. However, this should not bring large error into the results, as the applications we run are I/O bound ones, i.e. the time spend on I/O is dominant while the time spend on CPU is negligible.

In the experiment inside the chamber, the wireless nodes were operating on 802.11a. Correspondingly, we have an 802.11a physical layer model in our Sim/Control, which predicts packet losses and delay. Our 802.11a model implements CSMA/CA, and it uses bit-error-rate model to stochastically sample lost packets.

Validating Bandwidth, Delay and Jitter — One-link Scenario

We start with the simplest scenario with two wireless nodes and one wireless link. Packets are sent to the receiver as fast as possible using 54Mbps data rate under 802.11a, and the receiver records the timestamp of each received packet. The comparison between real trace and emulated results is shown in Figure 3.8 and Figure 3.9. We study the delay rather than throughput because the sender is sending packets of constant size, and therefore accurate delay implies accurate throughput, but not vice versa. The experiment actually persists for 10sec in real time, but we only show 20 packets for conciseness.

As shown in Figure 3.8, the emulated result (the 1-flow line) under our virtual time system is almost identical to the real trace, with the error within 1msec. The only difference is due to random retransmissions, which are caused by low SINR. In 802.11a, when the receiver cannot correctly decode the data frame, it will not send an ACK frame. In this case, the sender will

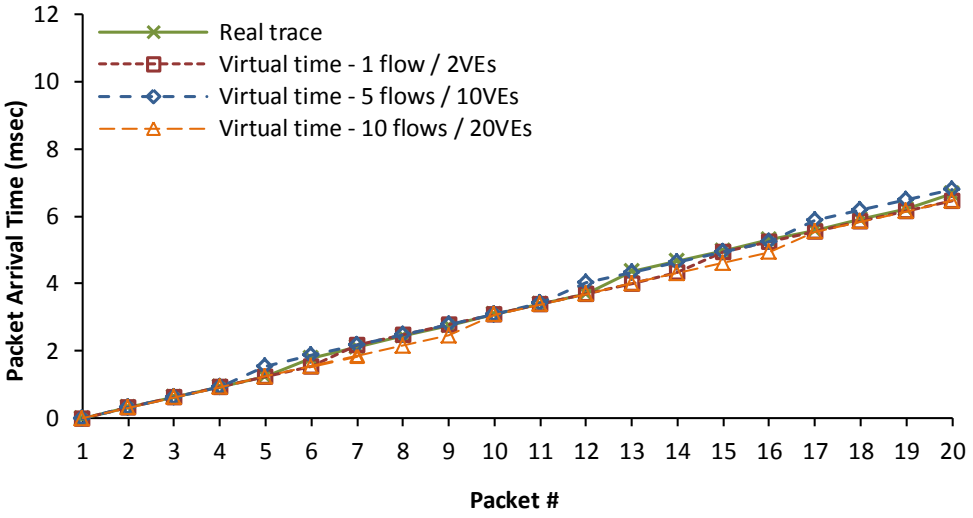


Figure 3.8: Packet Arrival Time, One-link Scenario, with Virtual Time

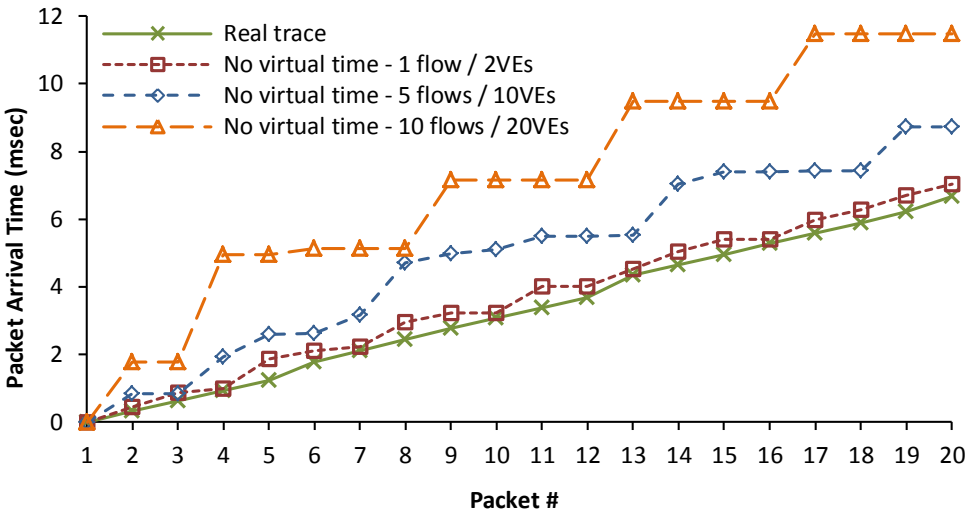


Figure 3.9: Packet Arrival Time, One-link Scenario, without Virtual Time

have to retransmit that data frame, and this will approximately double the transmission time of this single packet compare with no retransmission. From the figure, we are able to tell when a retransmission happens by examining inter-packet arrival time. As retransmission is a random event, it is reasonable that our 802.11a model cannot predict retransmission for the exact packet as real trace. In general, the model predicts comparable retransmission rate.

To demonstrate the accuracy of our system under different loads, the previous one-link scenario is replicated several times, with each replication emulated simultaneously. However, replicated links are only used to saturate the emulator, so they are independent and will not interfere with each other. This represents the scenario in which we have multiple non-interfering wireless links; each flow has the same behavior as a single flow (modulo differences in random number seeds.) In Figure 3.8, the 5-flow line and the 10-flow line show the results of a single flow out of the 5 or 10 simultaneous ones. Regardless of the number of flows, the behavior of each flow is identical to 1-flow case, as expected.

For comparison, we run the same experiment under the system without virtual time, and the results are shown in Figure 3.9. The 1-flow result is has only slightly larger error than that with virtual time, being caused by scheduling delay. This occurs when a VE cannot get the CPU and execute exactly on time, including (1) when the network emulator should deliver a packet to destination VE, and (2) when the destination VE should process an incoming packet. In fact, such scheduling delay also exists in virtual time implementation, but VEs will not perceive delay because their virtual clocks do not advance meanwhile. The error of not having virtual time here is as small as 1msec, but it will scale up when the number of VEs increases, as more VEs may introduce longer delay. As shown by the 5-flow line in Figure 3.9, the error can be as large as several milliseconds. Worse still, when the offer load is too high (e.g. the 10-flow line) for the emulator to process in real time, the error accumulates and becomes larger and larger. This is occurs because the emulator cannot run fast enough to catch up with real time.

Validating Bandwidth, Delay and Jitter — Two-link Scenario

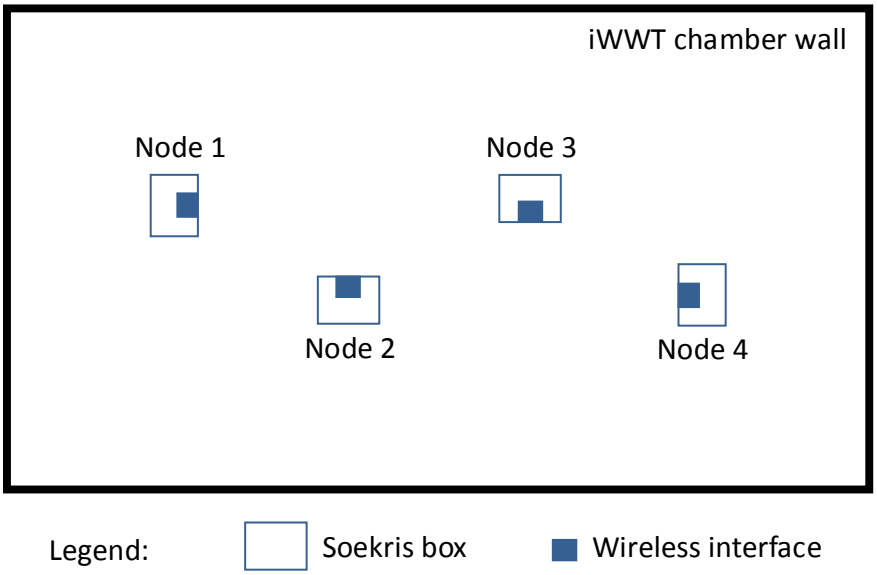
We next validate the scenario with four wireless nodes and two conflicting wireless links. Space limitations inside the iWWT prohibit study of more complex scenarios. Figure 3.10 shows the experimental setup. Although we only have four wireless nodes, we have six different combinations by changing source/destination pair and data rates. For example, Scenario 3 and Scenario 4 are those of heterogeneous data rate.

We use the `iperf` link test application [12] to test both links, and we are interested in both bandwidth and jitter (delay variation). We configure the emulator to simulate the above scenario. However, by examining the real data, we found that the link with Node 2 as sender usually has a higher throughput, regardless of its receiver. As analyzed in Section 2.2, we conclude this is due to the hardware, although the four wireless adapters are of the same manufacture and same model. We capture such hardware characteristic by increasing the antenna gain of Node 2 in our network simulator. Higher antenna gain results in higher SINR, lower bit error rate (BER), lower retransmission rate, and finally higher throughput.

The results are shown in Figure 3.11 and Figure 3.12. The former shows comparison of throughput and the latter shows that of jitter. We observe a very accurate emulated throughput, with error less than 3% in most cases. On the other hand, the jitter obtained from emulated platform has a larger error, which is within 10% in most cases but which sometimes is as large as 20%. By comparing timestamps generated by the simulator and that perceived by the application, we found such error is due to the inaccuracy of the 802.11a model, *not* the virtual time system itself. As discussed before, accurate delay implies accurate throughput but not vice versa. The results here demonstrate that jitter is more difficult to model than throughput.

Emulation Runtime and Scalability

We tested the execution speed of our system, by modifying the number of simultaneous network flows. We reuse the previous one-link scenario, in which the sender transmits packets to the re-



Scenario ID	Link 1		Link 2	
	src→dst	data rate	src→dst	data rate
1	2→1	6Mbps	3→4	6Mbps
2	2→4	6Mbps	3→1	6Mbps
3	2→1	54Mbps	3→4	6Mbps
4	2→4	54Mbps	3→1	6Mbps
5	2→1	54Mbps	3→4	54Mbps
6	2→4	54Mbps	3→1	54Mbps

Figure 3.10: Two-link Experiment Setup Inside the iWWT

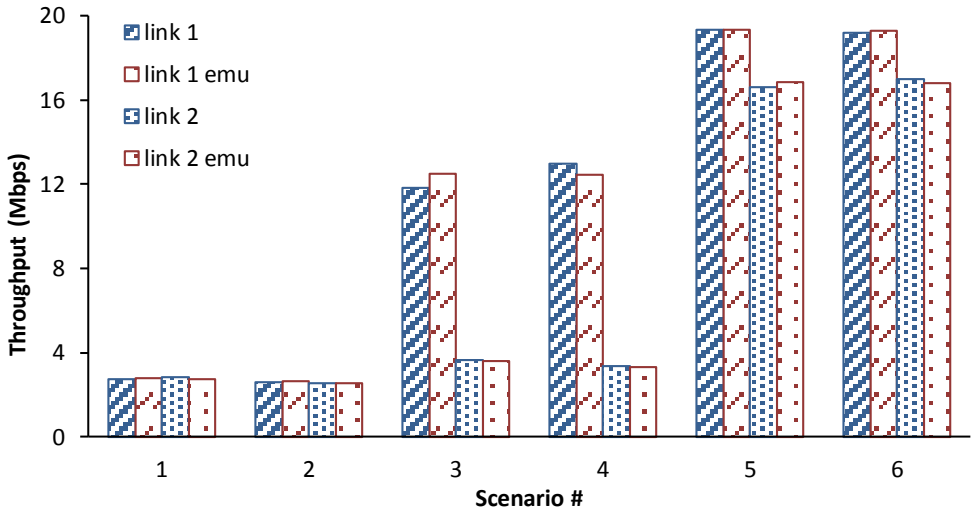


Figure 3.11: Two-link Scenarios — Throughput

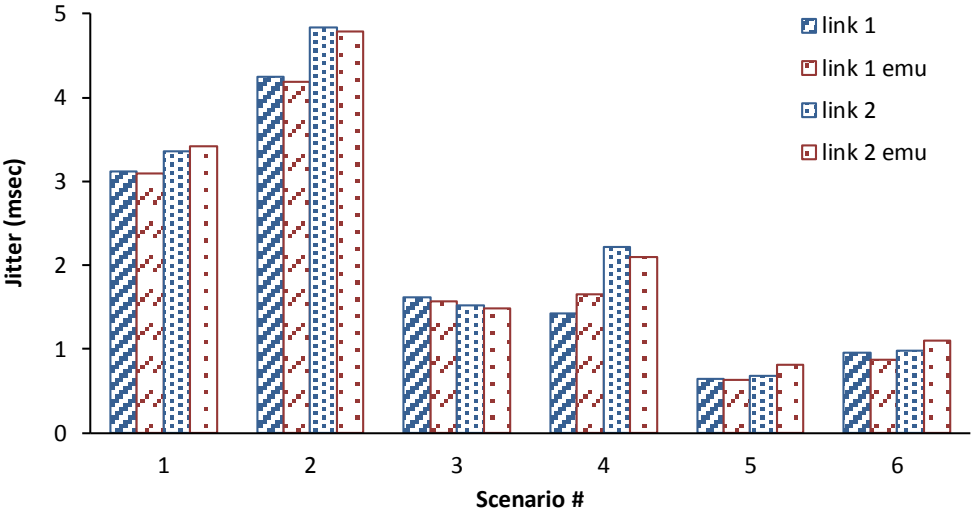


Figure 3.12: Two-link Scenarios — Jitter

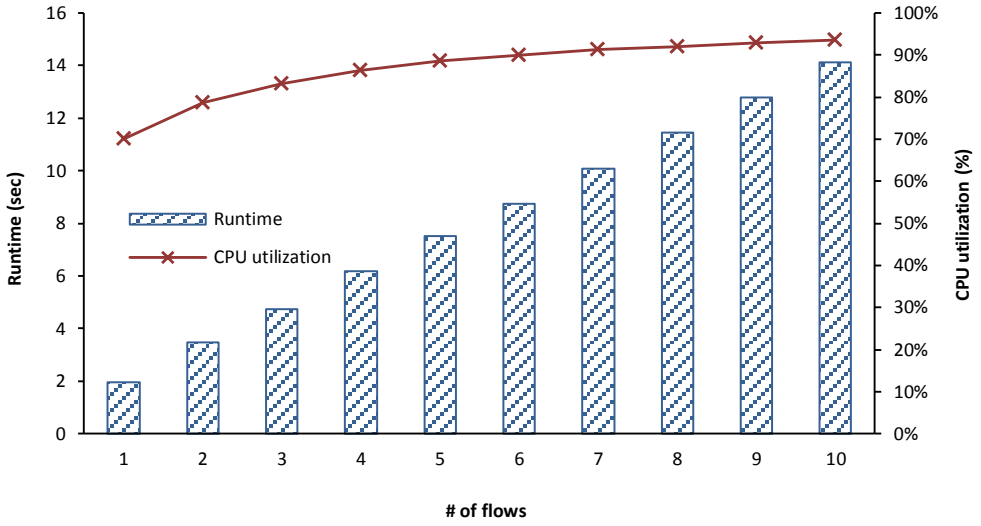


Figure 3.13: Emulation Runtime and CPU Utilization

ceiver for 10sec using 54Mbps data rate. We replicate this link by several times, but only in order to saturate the emulator.

The emulation runtime under different loads is shown in Figure 3.13. When there is only one link, the runtime is only less than 2sec in real time. Compared with 10sec elapsed in virtual time, emulation runs faster because in this case the emulated I/O is faster than real I/O. As the number of flows increases, the runtime increases linearly as well. When there are more than 7 flows, the emulation runs slower than real time because of large volume of traffic. In addition, we plot the CPU usage percentage, and we find that our system can achieve high CPU usage when there is enough load. It does not achieve 100% CPU usage because our Lenovo laptop is using dual-core CPU. As explained in Section 3.3, before the emulator process starts its timeslice, it will wait until all VEs stop running. Different VEs can run on different CPUs simultaneously, but the emulator process runs on only one CPU, and meanwhile the other CPU is idle. As shown in Figure 3.13, increased number of flows results in higher CPU utilization. This is because as the number of flows increases and so does the number of VEs, the fraction of time during which the CPUs are saturated by VE execution also increases.

To demonstrate the scalability of our system, we tested our system on a Dell PowerEdge 2900

server with dual Intel Xeon E5405 and 16GB RAM. We rerun the above scenario but with 160 flows and 320 VEs, which finishes in 307sec (compared with <2 sec for 1 flow). The system presented in this chapter can only run on a single machine, but with its distributed version (Chapter 5), we will be able to emulate more network nodes. Nevertheless, we found that implementation with OpenVZ yields high VE density (320 VEs per physical machine), thanks to the light weight of OpenVZ.

Implementation Overhead

We are concerned about the implementation overhead of our system, and we measure it in the following way. We slightly modify the user-level emulator application, making it run in real time instead of virtual time so that it can run on original Linux platform. When running on original Linux, we use the `snull` virtual Ethernet tunnel module [26] to replace the virtual Ethernet devices created OpenVZ, so that both `iperf` and the emulator can be configured in the same way.

We reuse the scenario of the previous subsection, but we only use 5 flows so that our Lenovo laptop is capable to run in real time. When running in real time, either on original Linux or on original OpenVZ, the runtime of emulation is exactly 10sec. Instead of using the elapsed time of wallclock, we use the total CPU time for comparison. For instant, if the average CPU utilization is 60% during the 10-second period, the total CPU time is 6sec.

The comparison among Linux, OpenVZ, and our virtual time system is shown in Table 3.1. For the scenario we consider, we obverse 2% overhead of OpenVZ compared with original Linux. This matches the claim on OpenVZ project website, which says OS-level virtualization only has 1%-3% overhead [4]. In addition, we observe another 3% overhead of implementing virtual time, based on the original OpenVZ system. We find that such overhead is due to both frequent system calls and context switches. Implementing the emulator in kernel space might help on reducing such overhead, but we prefer to keep the kernel safe and simple. The observed 3% overhead is low, and considering the performance of OS-level virtualization, we conclude that our system is highly scalable.

Table 3.1: Implementation Overhead

	Original Linux	Original OpenVZ	Virtual Time System
Total CPU Time	6.345 sec	6.478 sec	6.654 sec
Time in %	100.0%	102.1%	104.9%

Finally, we notice that such implementation overhead on OpenVZ is competitively low compared with other virtualization techniques. For instance, QEMU without a kernel acceleration module is 2X to 4X slower [20]. We are currently developing a virtual time system for QEMU as well, in order to support time virtualization to applications running on a larger number of operating systems.

3.5 Variable Timeslice

Our virtual time system can reduce the temporal error by using smaller timeslices. But as explained above, 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 250µs, rather than 1ms. Actually, with `HZ`=4000, any timeslice length of $n\times$250µs 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 [16], 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

$$\rho = \frac{TS - T_{CS} - k \times T_{int}}{TS} = 1 - T_{CS} \times \frac{HZ}{k} - HZ \times T_{int} \tag{3.1}$$

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 $1 - HZ \times T_{int}$. 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 $HZ = \frac{1}{E}$. We next provide experimental validation which shows that the above formula fits our measurements on real hardware quite well.

We change the scheduler timeslice in our OpenVZ implementation, and measure the maximum network traffic process rate in real time. Such process rate reflects the speed of the system: the higher the rate, the faster the emulation runs. The result is shown in Figure 3.14. We observe a 45% overhead when we reduce the timeslice from 1ms to 30µs.

On the other hand, as shown in Figure 3.15, we also find the analytical result we get previously fits the experimental results roughly, with the parameters being properly chosen (T_{int} = 6µs, T_{CS} = 8µs, and let $throughput$ = $\rho \times$970Mbps). As we can see from the analytical results, overhead increases convexity as TS gets small. When the timeslice is already small, any further reducing it will cause significant overhead, e.g. 60µs to 30µs. However, as we have seen already, for a given error constraint we can choose a timeslice that maximizes efficiency subject to that constraint.

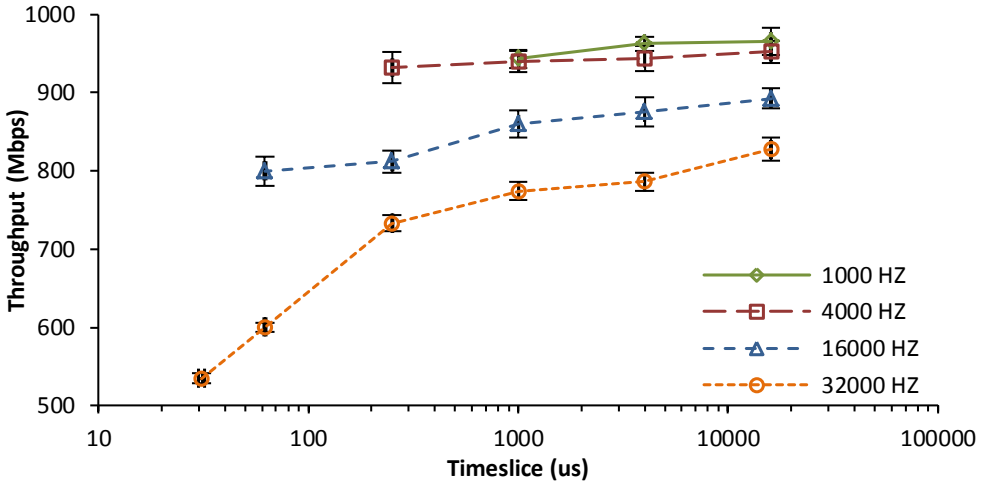


Figure 3.14: Emulation Speed under Various Timeslice — Experimental Results

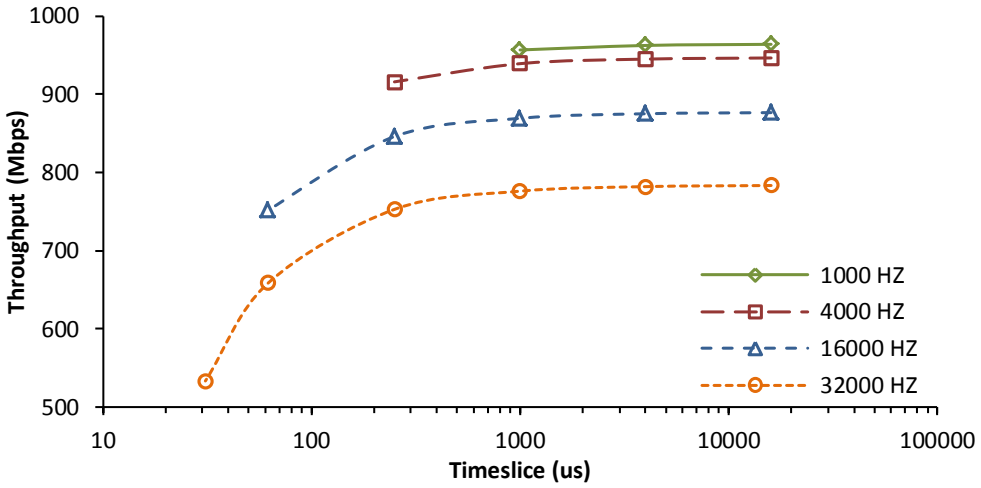


Figure 3.15: Emulation Speed under Various Timeslice — Analytical Results

3.6 Related Work

Related work falls into the following three categories: 1) network simulation and emulation, 2) virtualization technique and 3) virtual time systems. They are discussed one by one as follows.

Network Simulation and Emulation

Network simulation and network emulation are two common techniques to validate new or existing networking designs. Simulation tools, such as ns-2 [1], ns-3 [3], J-Sim [73], and OPNET [5] typically run on one or more computers, and abstract the system and protocols into simulation models in order to predict user-concerned performance metrics. As network simulation does not involve real devices and live networks, it generally cannot capture device or hardware related characteristics.

In contrast, network emulations such as PlanetLab [25], ModelNet [77], and Emulab [80] either involve dedicated testbed or connection to real networks. Emulation promises a more realistic alternative to simulation, but is limited by hardware capacity, as these emulations need to run in real time, because the network runs in real time. Some systems combine or support both simulation and emulation, such as CORE [15], ns-2 [1], J-Sim [73], and ns-3 [3]. Our system is most similar to CORE (which also uses OpenVZ), as both of them run unmodified code and emulate the network protocol stack through virtualization, and simulate the links that connect them together. However, CORE has no notion of virtual time.

Virtualization Technique

Virtualization divides the resources of a computer into multiple separated Virtual Environments (VEs). Virtualization has become increasingly popular as computing hardware is now capable enough of driving multiple VEs concurrently, while providing acceptable performance to each. There are different levels of virtualization: 1) virtual machines such as VMware [79] and QEMU [20], 2) paravirtualization such as Xen [18] and UML [7], and 3) Operating System (OS) level

virtualization such as OpenVZ [4] and Virtuozzo [8]. Virtual machine offers the greatest flexibility, but with the highest level of overhead, as it virtualizes hardware, e.g., disks. Paravirtualization is faster as it does not virtualize hardware, but every VE has its own full blown operating system. OS level virtualization is the lightest weight technique among these [68], utilizing the same operating system kernel (and kernel state) for every VE. The problem domain we are building this system to support involves numerous lightweight applications, and so our focus is on the most scalable of these approaches. The potential for lightweight virtualization was demonstrated by Sandia National Lab who demonstrated a one million VM run on the Thunderbird Cluster, with 250 VMs each physical server [56]. While the virtualization techniques used are similar to those of the OpenVZ system we have modified, the Sandia system has neither a network simulator between communicating VMs, nor a virtual time mechanism such as we propose.

Virtual Time System

Recent efforts have been made to improve temporal accuracy using Xen paravirtualization. DieCast [36], VAN [24] and SVEET [30] modify the Xen hypervisor to translate real time into a slowed down virtual time, running at a slower but constant rate, and they call such mechanism time dilation. At a sufficiently coarse time-scale this makes it appear as though VEs are running concurrently. Other Xen-based implementations like Time Jails [35] enable dynamic hardware allocation in order to achieve higher utilization. Our approach also tries to maximize hardware utilization and keep emulation runtime short. Unlike the mechanism of time dilation, we try to advance virtual clock as fast as possible, regardless it is faster or slower than real time.

Our approach also bears similarity to that of the LAPSE [28] system. 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, 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 (LAPSE simulates only the switching network of the Intel Paragon).

3.7 Chapter Summary

We have implemented a virtual time system which allows unmodified application to run on different virtual environments (VEs). Although multiple VEs coexist on a single physical machine, they perceive virtual time as if they were running independently and concurrently. Unlike previous work based on Xen paravirtualization, our implementation is based on OpenVZ OS-level virtualization, which offers better performance and scalability (at the price of less flexibility). In addition, our system can achieve high utilization of physical resources, making emulation runtime as short as possible. Our implementation has only 3% overhead compared with OpenVZ, and 5% compared with native Linux. This indicates that our system is efficient and scalable.

Through evaluation, we found the accuracy of virtual time can be within 1ms, at least if an accurate network simulator is used. It might be indeed the simulator that introduces the error, rather than the virtual time system itself. If not from the simulator, the temporal error can be further reduced by having smaller scheduler timeslices, but at the cost of slower execution speed. This is the common tradeoff between behavioral accuracy and execution speed in simulation domain. We observe a 45% slower execution speed when we reduce the time slice from 1ms to 30µs.

Chapter 4

Integration with S3F Simulator

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

4.1 Overview and Motivation

The advancement of large-scale computer and communication networks, such as Internet, power grid control networks, heavily depends on the successful transformation from in-house research efforts to real productions. To enhance this transformation, research has created various network testbeds that use emulation, or simulation, for conducting medium to large scale experiments. The emulation testbeds coordinate real physical devices and provide a configurable environment to

conduct live experiments, but for networking are constrained by budget and what can be equipped in a lab. This limits scalability and flexibility. On the other hand, network simulation provides better scalability and much more flexibility, but degrades fidelity owing to the sort of model abstraction and simplification necessary to achieve scale. Furthermore, development of simulation models can be labor-intensive.

Our work in studying security in the smart grid's Advanced Metering Infrastructure (AMI) is one of the motivators of this chapter. We need to study behavior of software and networking in a system with many meters, connected locally through wireless networks, and through wire-line networks to utilities. We need to study how particular software behaves under cyber-attack, and how the nature of a distributed denial of service (DDoS) attack affects delivery of that attack to victims, and how it impacts the overall network behavior. We use emulation technology to run real software stacks that run in meters, and simulation technology to model wireless and wirelined networks, as well as models of meters that contribute to the network traffic load but are not otherwise particular objects of study. We combine here two prior efforts. We use a version of OpenVZ modified to operate in virtual time [86] with a new parallel network simulator, S3F [64], which was inspired by SSF [59], and RINSE [52].

OpenVZ allows one to run real applications under a real OS and pass messages between simulated and emulated hosts. Users can plug in a real smart meter program rather than be forced to create a simulation model of one. OpenVZ operates in virtual time, not wallclock time, thereby increasing temporal fidelity [86] (unlike most other emulation systems). Freeing the emulation from the real-time clock permits one to run experiments either faster than real time, or slower, depending on the inherent simulation workload. We use S3F for simulating large network scenarios, as it provides sophisticated networking layer protocols and the ability to simulate many many devices such as routers, switches, and hosts creating and receiving background traffic. S3F therefore provides scalability. Coordination of activity between OpenVZ's emulation and S3F's simulation is handled by new extensions to S3F, described in this chapter. The current system can run 300+ OpenVZ virtual machines and simulate millions of devices on a single multi-core server.

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

4.2 Design

System Design Architecture

Figure 4.1 depicts the system design architecture of our system, which integrates the OpenVZ network emulation into a S3F-based network simulator on a single physical machine. The system is capable of running large-scale and high-fidelity network experiments with both emulated and simulated nodes.

Background: SSF and S3F The Scalable Simulation Framework (SSF) is an API developed to support modular construction of simulation models, in such a way that potential parallelism can be easily identified and exploited. Following ten years of use, we created a second generation API named S3F [64]. In both SSF and S3F, a simulation is composed of interactions among a number of *entity* objects. Entities interact by passing *event*s through *channel* endpoints they own. Channel endpoints are described by *InChannel*s and *OutChannel*s depending on the message direction. Each entity is aligned to a *timeline*, which hosts an event list and is responsible for advancing all entities aligned to it. Interactions between co-aligned entities need no synchronization other than this event-list. Multiple timelines may run simultaneously to exploit parallelism, but they have to be carefully synchronized to guarantee global causality. The synchronization mechanism is built around explicitly expressed delays across channels whose end-points reside on entities that are not aligned. We call these *cross-timeline channels*. The synchronization algorithm creates synchronization windows, within which all timelines are safe to advance without being affected

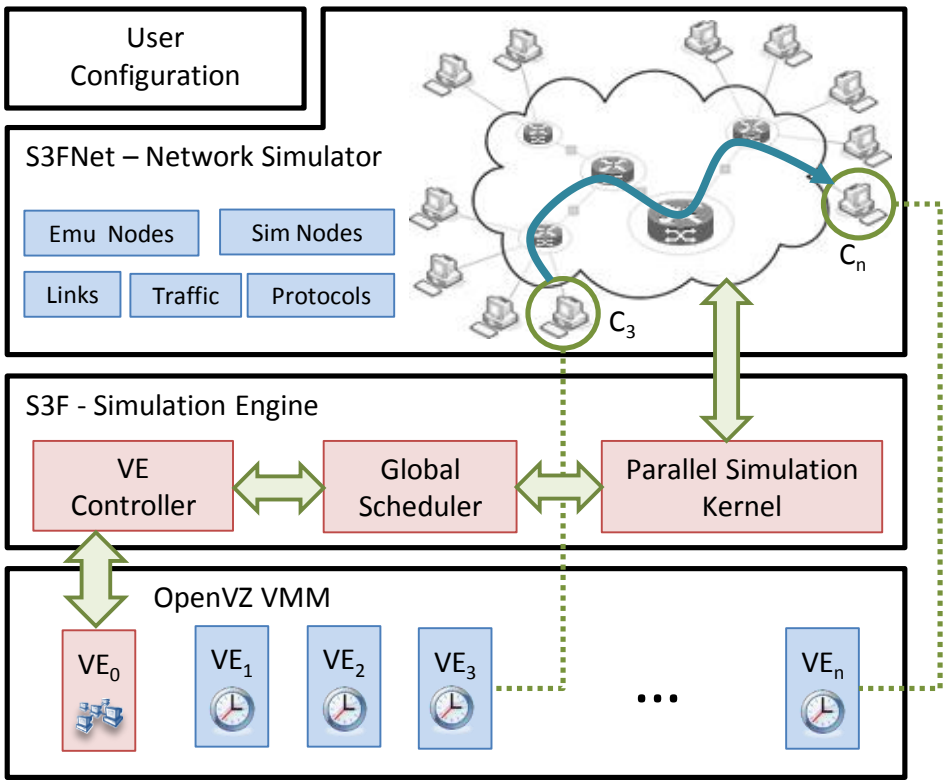


Figure 4.1: System Design Architecture

by other timelines. More details about S3F are in [64].

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

Structurally, every VE in the OpenVZ model is represented in the S3FNet model as a host within the modeled network. Within S3FNet traffic that is generated by a VE emerges from its proxy host inside S3FNet, and, when directed to another VE, is delivered to the recipient's proxy host. The synchronization mechanism needs to know the distinction though between an emulated host (VE-host) or a virtual host (non-VE host), as shown in Figure 4.1. However, the type of host should make no difference to the simulated passing and receipt of network traffic. The global scheduler we added in S3F is designed for coordinating safe and efficient advancement of the two systems and to make the emulation integration nearly transparent to S3FNet.

S3F synchronizes its timelines at two levels. At a coarse level, timelines are left to run during an *epoch*, which terminates either after a specified length of simulation time, or when the global state meets some specified conditions. Between epochs S3F allows a modeler to do computa-

tions that affect the global simulation state, without concern for interference by timelines. Good examples of use include periodically recalculating of path loss delays in a wireless simulator, or periodic updating of forwarding tables within routers. States created by these computations are otherwise taken to be constant when the simulation is running. Within an epoch, timelines synchronize with each other using barrier synchronization, each of which establishes the length of the next *synchronization window* during which timelines may execute concurrently. Synchronization between emulation and simulation is managed by the global scheduler at the end of a synchronization window, when all timelines are blocked. Here it is that events and control information pass between OpenVZ and S3F, using S3F's global scheduler and *VE controller*. Details about these interactions will be discussed in Section 4.3.

OpenVZ Emulation and VE Controller OpenVZ is an OS level virtualization technology, which enables multiple isolated execution 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. More details are in Chapter 3.

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 [86], 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 4.1, S3F controls all emulation hosts through VE controller, which is responsible for controlling all emulation VEs according to S3F's command, as well as providing necessary communications between S3F and VEs. More details are provided in Section 4.3.

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 [31], 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.

Simulation/Emulation Coordination

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

A packet bound for a VE proxy host transits the network model, reaches the proxy host, and

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

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

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

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 $t_0 + \alpha \times T$, 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 $t_0 + \alpha \times T$.

In the OpenVZ system, the unit of scheduling (minimum execution time) is a timeslice. We currently set timeslice length TS=100µs, but TS is tunable according to Section 3.5 [87]. 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 $t_s = t_0 + \alpha \times \min\{x, T\}$, 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 $\alpha \times T$ 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 $\alpha \times T_\epsilon$, 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 $t_0 + \alpha \times x$ by no greater than $\alpha \times TS$, 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 $t_d + \alpha \times TS$.*

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 $t_0 + \alpha \times x$ versus an earlier arrival at time $t_0 + \alpha \times TS$.

4.3 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.

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 [64]. The larger the synchronization window size is, the less frequent a simulator needs to stop for global synchronization, and so achieve better performance. In terms of pure network simulation, the channel mapping can be used to model links among host network interfaces, and the latencies across the channels can be constructed by the packet transfer time and the link propagation delay.

However, integration with the OpenVZ-based emulation brings new features and constraints to the existing synchronization mechanism. Firstly, emulation and simulation never operate concurrently, therefore two clocks actually exist in the system: the current simulation time and the current emulation time; there exist also two types of synchronization window: the *emulation synchronization window (ESW)* and the *simulation synchronization window (SSW)*. The system first computes an ESW and runs the emulation for that long, and then injects packets created during that window into the simulator, at the end of the ESW. The new emulation events contribute to the computation of SSW for the next simulation cycle. Both ESW and SSW are calculated at S3F. Secondly, our system design ensures that the simulation can never run ahead of the current emulation time. Thirdly, once the OpenVZ emulation starts to run, it has to run for at least one timeslice [86], during which no simulation work can interrupt any VE. This system level constraint affects the granularity of the system. Finally, the OpenVZ system introduces opportunities for offering real application specific lookahead for increasing the size of ESW.

The notations used in this section are listed in Table 4.1, and the scheduling mechanism used in the global scheduler is described in Algorithm 4.1. 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.

Equation 4.1 below illustrates how ESW is calculated:

Table 4.1: Notation Descriptions

Notation	Description
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 4.2
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 $e \in E_{sim}$

Algorithm 4.1 Global Scheduler

```
1: while true do
2:    if t_sim = t_emu then
3:       compute ESW
4:       run OpenVZ emulation for ESW (Algorithm 4.2)
5:       inject packets to simulation
6:    else
7:       compute SSW
8:       run S3F simulation (all timelines) for SSW
9:    end if
10: end while
```

$$ESW = \max\left\{\alpha \times TS, \min_{timeline\ i}\{P_i\} - t_{emu}\right\} \tag{4.1}$$

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:

$$P_i = \min\left\{\left[\min\left(n_i^{sim}, \min_{entity\ e}\{l_{i,e}\}\right) + B_i\right], n_i^{emu}\right\} \tag{4.2}$$

and B_i is the minimum channel delay from timeline i:

$$B_i = \min_{outchannel\ j}\left\{w_{i,j} + \min_{inchannel\ k}\{r_{i,j,k}\}\right\} \tag{4.3}$$

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 a VE from running too far ahead and bypassing a potential packet delivery event.

Equation 4.4 below illustrates how SSW is calculated:

$$SSW = \min\left\{t_{emu}, \min_{timeline\ i}\{Q_i\}\right\} - t_{sim} \tag{4.4}$$

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:

$$Q_i = n_i + C_i \tag{4.5}$$

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

$$C_i = \min_{outchannel\ j} \left\{ w_{i,j} + \min_{inchannel\ x} \{s_{i,j,x}\} \right\} \tag{4.6}$$

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 $\alpha \times TS$, the simulation has to run multiple synchronization windows to catch up to the emulation. On the other hand, when SSW is larger than $\alpha \times TS$, the emulation can run multiple time slices in one emulation cycle. Figure 4.2 and Figure 4.3 illustrate the behavior of the system in the two cases respectively.

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

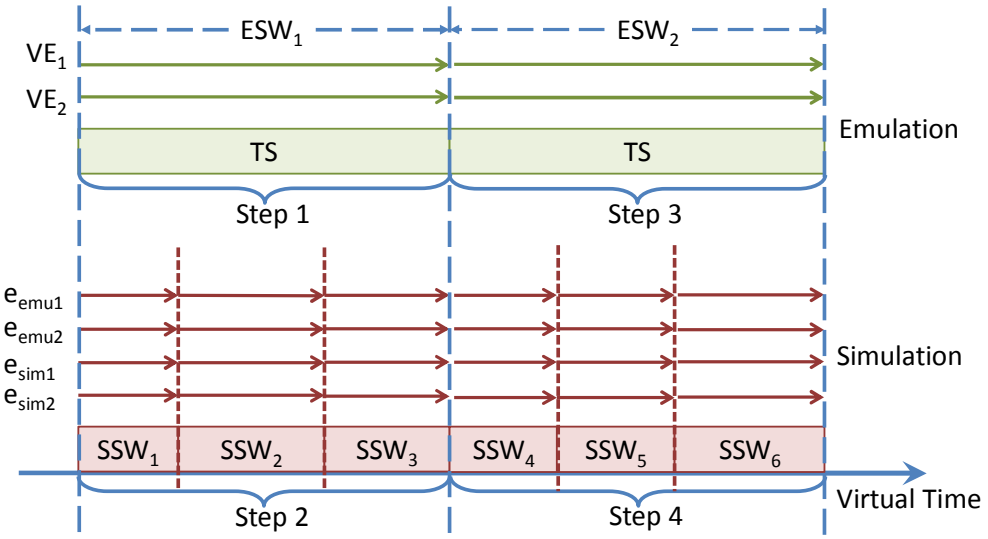


Figure 4.2: Global Synchronization, Emulation Timeslice $\geq$ Simulation Sync Window

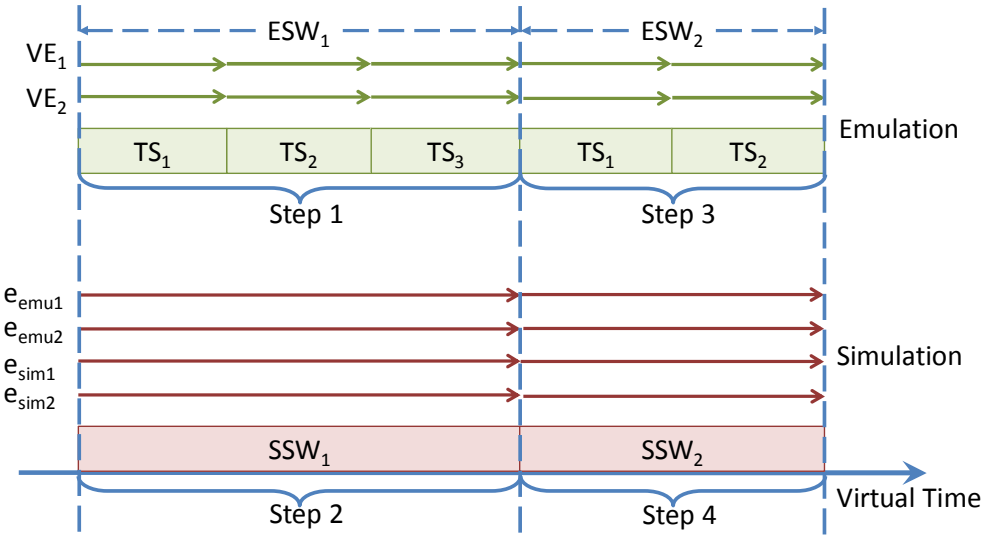


Figure 4.3: Global Synchronization, Emulation Timeslice $<$ Simulation Sync Window

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 [86].

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 4.2.

Algorithm 4.2 VE Controller

1: $barrier = t_{emu} + ESW$
2: **for all** VE_i **do**
3: $VE_i.stop \leftarrow barrier - \alpha \times TS/2 - VE_i.offset$
4: $VE_i.done \leftarrow$ false
5: **while** $VE_i.done \leftarrow$ false **do**
6: deliver due packets to VE_i
7: give a timeslice to VE_i
8: ($VE_i.clock$ keeps advancing while VE_i is running)
9: wait until VE_i stops
10: collect sent packets from VE_i
11: **if** VE_i is idle (has no runnable processes) **then**
12: $VE_i.offset \leftarrow 0$
13: $VE_i.clock \leftarrow \min(VE_i.nextPacket, VE_i.stop)$
14: **end if**
15: **if** $VE_i.clock \geq VE_i.stop$ **then**
16: $VE_i.offset \leftarrow VE_i.offset + (VE_i.clock - barrier)$
17: $VE_i.clock \leftarrow barrier$
18: $VE_i.done \leftarrow$ true
19: **end if**
20: **end while**
21: calculate emulation lookahead for VE_i
22: **end for**
23: $t_{emu} \leftarrow barrier$

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 4.2 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. In this chapter, we demonstrate the promise of emulation lookahead; estimating it and tolerating errors in it is presented in Chapter 5.

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

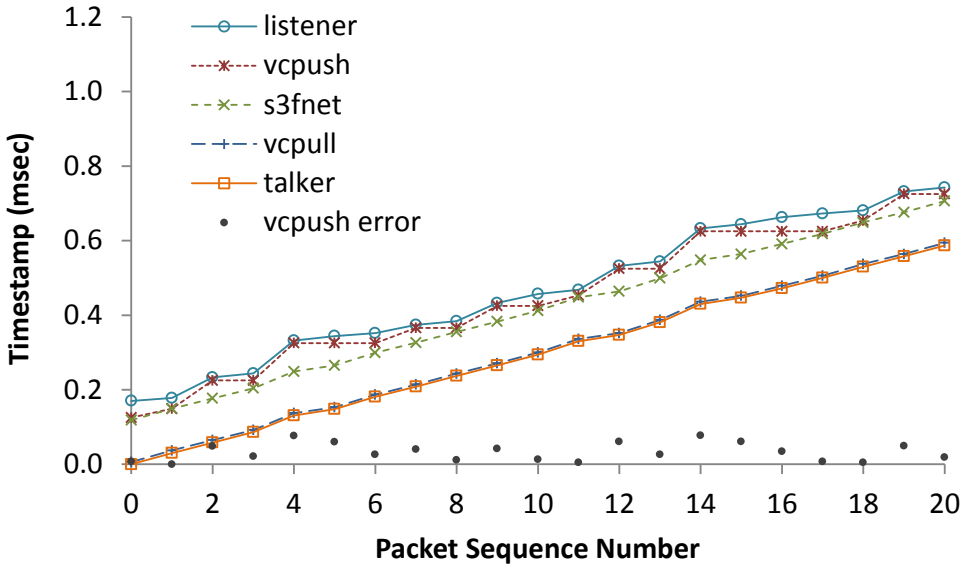


Figure 4.4: Timestamps during Packets Traverse Route

- $vcpush$: 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 4.4 for the 400 Mb/s case. The x-axis indexes the packet, the y-axis shows the times associated with each packet. Slopes decrease with increasing sending rate because inter-packet arrival times decrease.

Although we plot only one sending rate, the behavior of the error in each case is very close, and in this case is bounded by 100 µs—the length of a timeslice. Likewise, the effect of communication latency is the same in each plot, and can be seen in Figure 4.4. As explained in Section 4.2, 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 $\alpha \times TS$. 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 $\alpha \times TS$ 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 (Section 3.5 [87]).

4.4 Validation of Application Behavior

Experiment Setup

Our testbed provides both functional and temporal fidelity, by embedding the virtual machines in virtual time [86]. 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 section 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

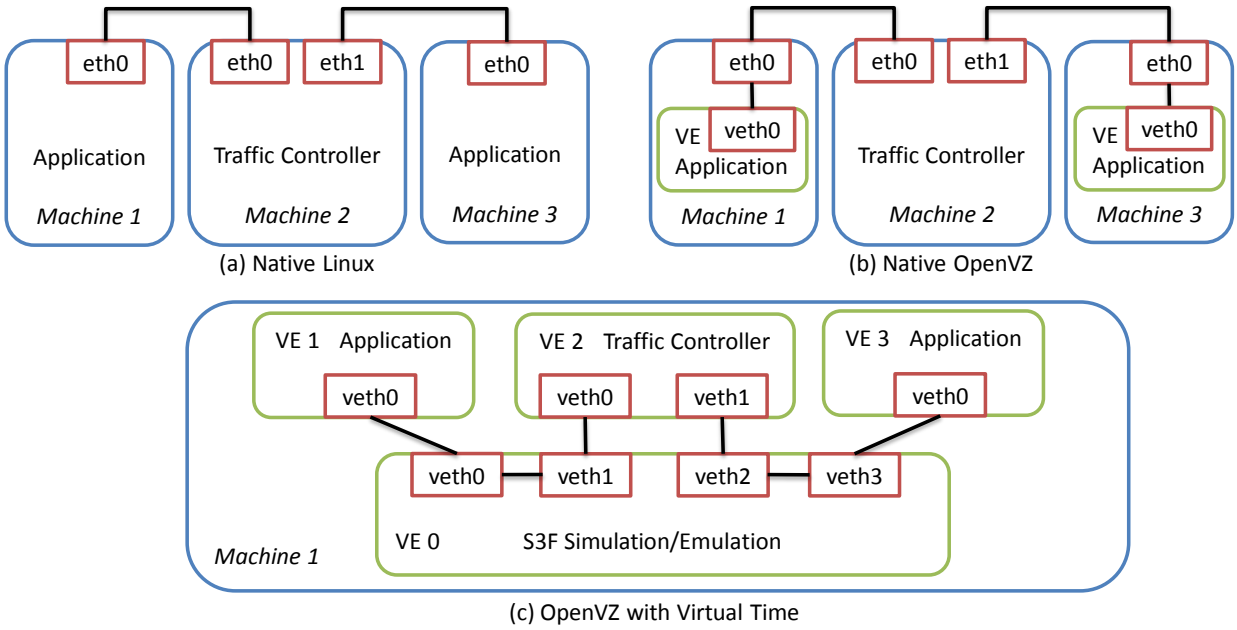


Figure 4.5: Three Testbeds Setups for Validation Experiments

and errors introduced by our testbed.

Figure 4.5 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 4.5(a) consists only the physical hosts, which serves as the ground truth data collector. The second platform, as shown in Figure 4.5(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 4.5(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 context for measuring the application-level network metrics.

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 4.5.

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 4.2.

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. Comparison

Table 4.2: Ping Results

Network Condition		Result								
Loss	Delay	Loss (%)			RTT avg (ms)			RTT mdev		
(%)	(ms)	1	2	3	1	2	3	1	2	3
0%	1	0	0	0	2.24	2.36	2.67	0.03	0.02	0.04
0%	10	0	0	0	20.2	20.3	20.6	0.04	0.05	0.05
0%	100	0	0	0	200	200	200	0.00	0.00	0.00
20%	1	30	30	30	2.24	2.36	2.69	0.04	0.02	0.01
20%	10	30	30	30	20.2	20.3	20.6	0.00	0.00	0.05
20%	100	30	30	30	200	200	200	0.00	0.00	0.00
50%	1	80	80	80	2.23	2.33	2.68	0.01	0.02	0.01
50%	10	80	80	80	20.2	20.3	20.6	0.06	0.00	0.06
50%	100	80	80	80	200	200	200	0.00	0.00	0.00

between Testbed 2 and Testbed 3 shows the timeslice error explained in Section 3.3. 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 [12] 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 4.3 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.

Table 4.3: Iperf UDP Results

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

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 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 4.4.

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

Table 4.4: Iperf TCP Results

Network Condition			Result		
Loss (%)	Delay (ms)	BW (Mb/s)	Throughput (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

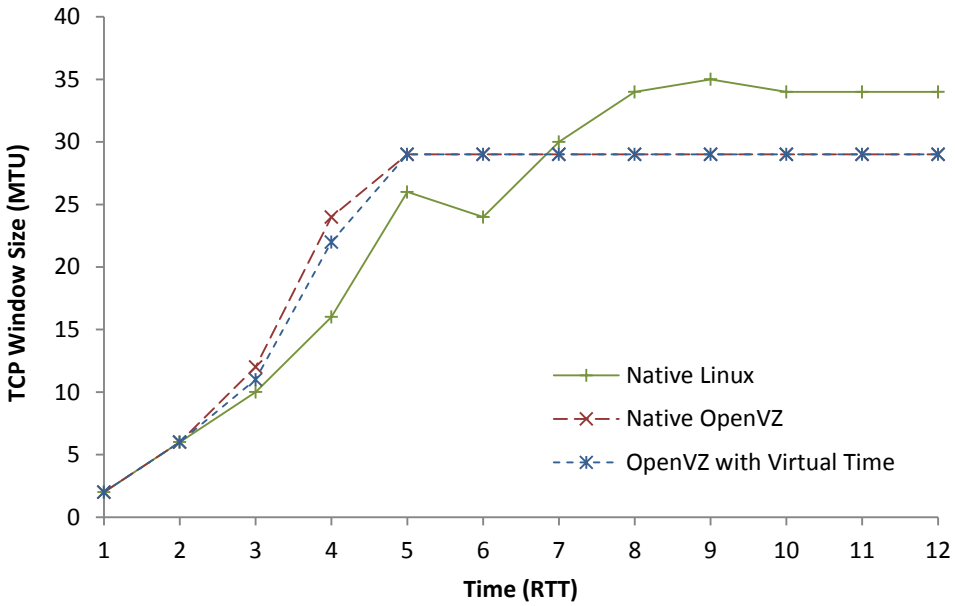


Figure 4.6: TCP Window Size of Three Platforms

2 yielding a significantly larger throughput! Investigation revealed that OpenVZ configuration allows for control of certain TCP buffer sizes, and that were set to be larger 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 4.6 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.

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 4.5.

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 establish 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.

Table 4.5: FTP Results

File Size (MB)	Network Condition		Result								
	Loss (%)	Delay (ms)	Throughput (KB/s)			Transfer Time (s)			Initiate Time (s)		
			1	2	3	1	2	3	1	2	3
10	0%	1	1140	1140	1140	8.8	8.8	8.8	0.1	0.1	0.1
10	0%	10	1130	1140	1130	8.8	8.8	8.8	0.3	0.2	0.3
1	0%	100	180	186	185	5.7	5.5	5.5	2.4	2.4	2.5
1	0%	1000	20.4	18.3	18.9	50.2	56.0	54.2	24.1	24.1	23.9
1	1%	100	79.5	96.1	108	12.9	10.7	9.5	2.4	2.5	2.4
1	2%	100	41.6	42.4	41.4	24.6	24.2	24.7	2.4	2.3	2.5
1	5%	100	29.6	29.1	29.3	34.6	35.2	34.9	2.3	2.5	2.4

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 [11] and a text-based web browser, named lynx [13], 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 4.6.

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.

Table 4.6: HTTP Results

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

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 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.

CPU-intensive Applications

For CPU-intensive applications, we implemented the Client Puzzle protocol [47], 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

Table 4.7: Puzzle Results

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

correctly solve the puzzle. In particular, a puzzle is essentially a hash inversion problem, which currently has no efficient algorithms to solve but using brute force search. Table 4.7 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, and 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 [45],

making the CPU computation time correct in long term. Without such mechanism, application runtime is less accurate and less repeatable.

4.5 Case Study: DDoS Attack in AMI Network

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

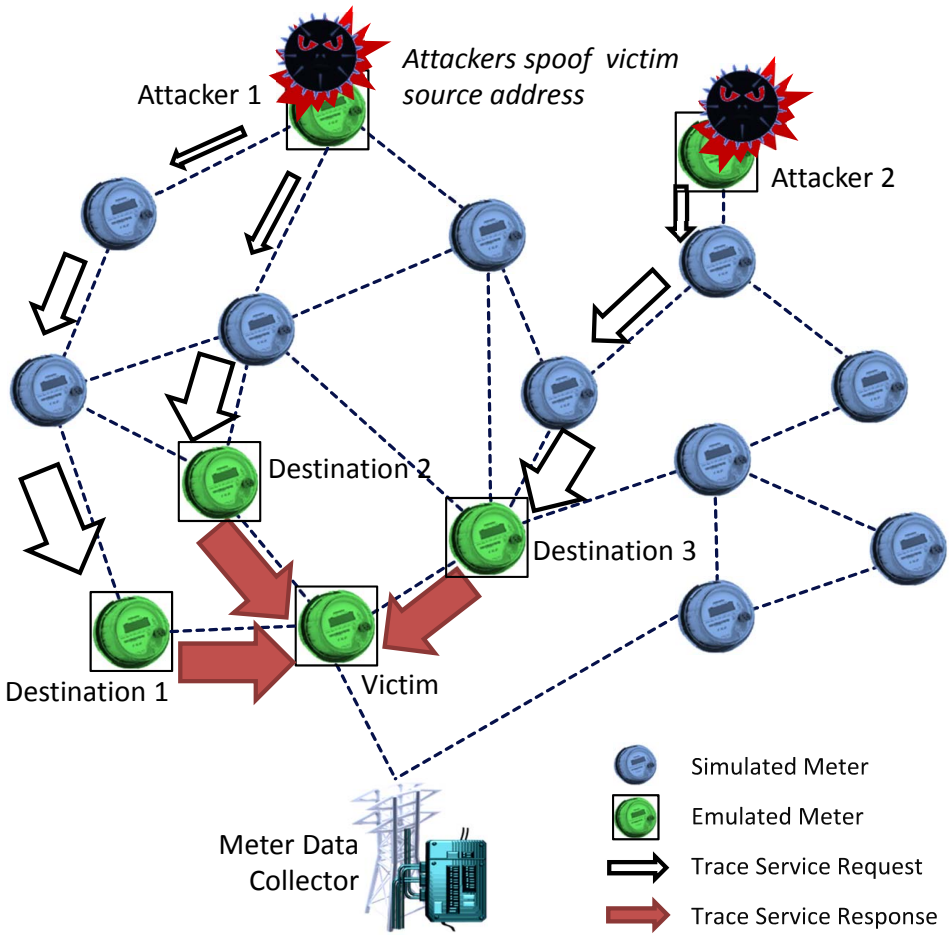


Figure 4.7: C12.12 Trace Service DDoS Attack in AMI Network

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 4.7 illustrates this attack.

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 4.8. The meters responsible for parsing and processing C12.22 packets are emulated by applications in VEs using real OS protocol stack. This set includes five attacking meters that generate trace service requests, five meters to which the traces are directed (each attacker targets its own), and one victim device, whose source address is spoofed by the attackers. The rest meters and the underlying communication network (802.15.4 ZigBee wireless network) with 1 Mb/s bandwidth are modeled and simulated by S3F. The radio channel path-loss model is the simple $\frac{1}{d^2}$ line-of-sight model. More sophisticated models can be introduced as needed.

Figure 4.8-A1 and A2 illustrate key meters in the experiment. The egress point is seen on the lower right edge. All the meters send routine traffic to that device, around 100-byte packet per 10 seconds. Attacking this choke-point maximizes the impact of the DDoS attack, and thus we set all the five attacks choose this point as the victim, and choose one of its close neighbors as the destination of the trace service request (and hence, reflection point). The figures mark out the location of the attackers, and the locations of their trace request destinations. Each attacker sends a trace service packet every 0.05 seconds (200 times faster than a normal meter) and each intermediate meter will add additional 20 bytes into the payload. Experimenting on the testbed shows that attackers initializing a few large-size packets rather than many small-size packets can

improve attacking efficiency due to eliminating frequent back-off times, therefore the trace service packet size is set to 500 bytes. Also learning from the experimental results, arranging the attacker meters in such a way that each of them covers a long (around 15 to 30 hops in this scenario) and spacial-separated route to the victim's surrounding meters could effectively render the entire network useless. More details will be covered soon in the result analysis.

We investigate and evaluate the impact of the DDoS attack by the following three metrics from each meter's viewpoint. The experiments data are collected in a 100 second window and the results are shown in Figure 4.8 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 4.8-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 Figure 4.8-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 4.8-B1), they tend to block out each other. Therefore, an efficient strategy requires the attackers to smartly select routes covering the entire network, especially the area around the victim, with minimum overlaps.

The AMI network uses ZigBee wireless as communication model which means the attacking traffic does not only affect the meters who forwards the traffic but also jams the channels of meters around them. Figure 4.8-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.

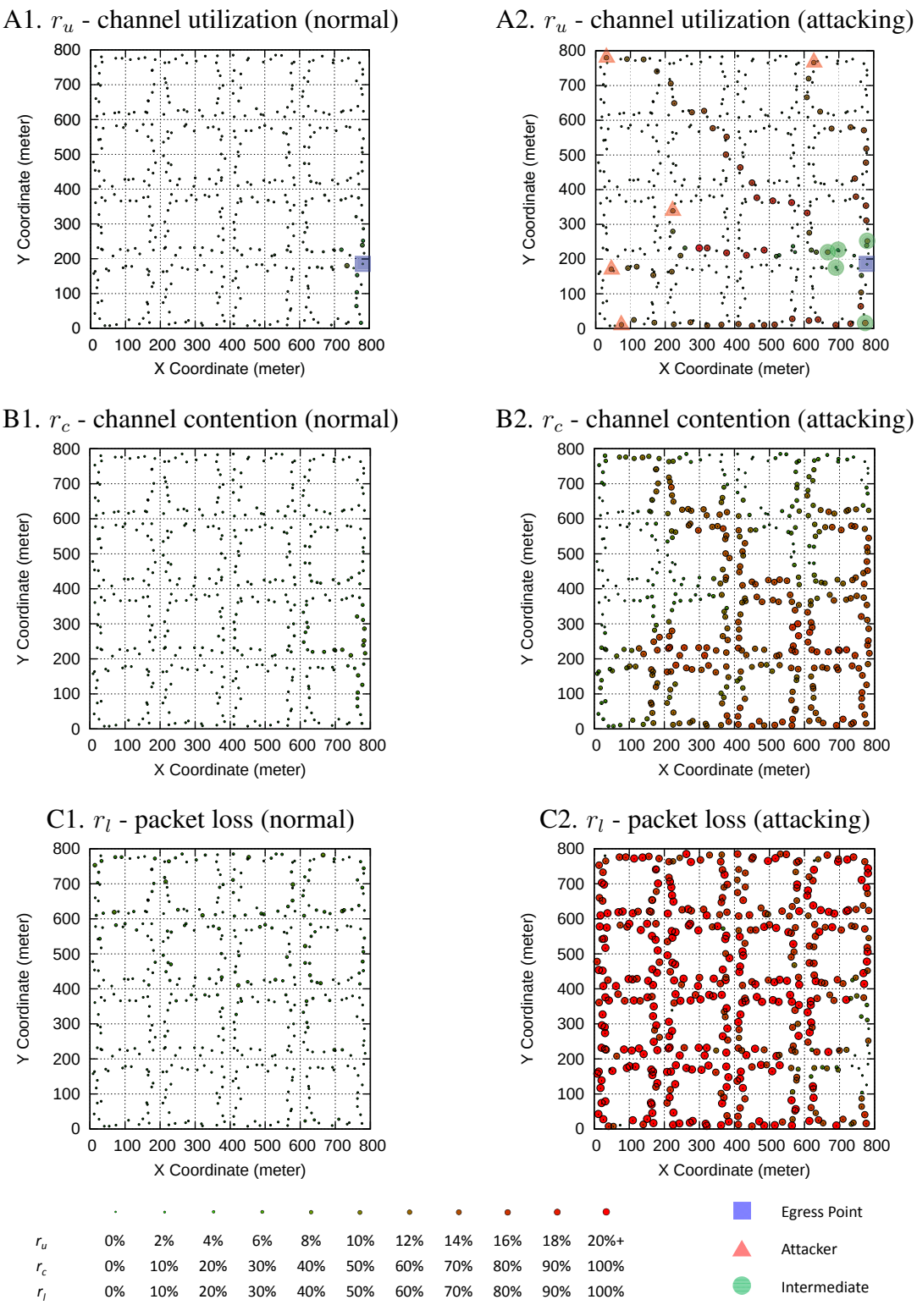


Figure 4.8: Experimental Results of the C12.22 Trace Service DDoS Attack

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 4.8-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 4.8-B2.

In the third set of experiments, we compared results in Figure 4.8-C1 and Figure 4.8-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.

Scalability

A simulator's performance scales if the ratio of simulation time to wallclock time increases only linearly as the model size increases. For our model we simultaneously increased the number of neighborhoods (hence numbers of meters and length of trace path), and the number of attackers

and destinations. We have run the system on an 8-core machine, with 2GHz processor each and 16 GB memory. We have simulated a system as large as 32×32 neighborhood, with 100 attacker/intermediate pairs and 10 egress points (hence 210 VEs) and 28672 simulated nodes, and noted the desired scaling property. We also noted, and are correcting, abnormally large memory use for forwarding tables. We should be able to simulate models two or three orders of magnitude larger with these optimizations.

4.6 Related Work

Network Simulation and Emulation

Network simulation and emulation are commonly used techniques to test and evaluate networking designs. Representative network simulators include ns-2 [1], ns-3 [3], SSFNet [27], GTNetS [9], and QualNet [10]. These network simulators generally cannot capture device or hardware characteristics because they do not involve real devices and live networks. On the other hand, the set of commonly used emulation testbeds include EmuLab [80], ModelNet [77], PlanetLab [25], DETER [22], VINI [19], X-Bone [76], and VIOLIN [43]. These emulators present more realistic alternatives to simulators because they combine real physical devices with emulation, but are limited by hardware capacity as they need to run in real time.

Some systems combine both simulation and emulation. One such example is CORE [15]. Recent work by Zheng et al. [86] is similar to CORE in that both of them use OpenVZ to run unmodified code and emulate the network protocol stack through virtualization, and simulate the links that connect them together. A difference is that CORE has no notion of virtual time, while [86] implemented it in their work.

Virtual Time

Recent efforts have been made to improve temporal accuracy in para-virtualization. DieCast [36] and VAN [24] modify the Xen hypervisor to translate real time into a slowed down virtual time,

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

4.7 Chapter Summary

In this chapter we present a system that integrates an OpenVZ-based network emulation system [86] into the S3F simulation framework [64]. The emulation allows native Linux applications to run inside the system; and the emulation is based on virtual time, which both provides temporal fidelity and facilitate the integration with the simulation system. We design and study the global synchronization and VE controlling mechanism in the system. Through analysis and experiment, we show that the virtual time error is bounded as a function of timeslice length, which itself is tunable at the cost of different execution speed [87]. We examine this error empirically, noting that it is much smaller than the guaranteed error bound. We then apply this platform to a case study of a DDoS attack within an Advanced Metering Infrastructure (AMI). Our experiments demonstrate the utility of the approach on an important modeling problem.

The current system requires both OpenVZ and S3F to reside on the same shared memory multiprocessor. The next chapter includes separating these to support distributed setup accross multiple machines. We are also interested in means of estimating lookahead from within the emulation and providing it to the simulation, to accelerate performance.

Chapter 5

Application Lookahead

Large-scale and high-fidelity testbeds play critical roles in analyzing large-scale networks such as data centers, cellular networks, and smart grid control networks. Previous chapter combines parallel simulation and virtual-time-integrated emulation, such that it offers both functional and temporal fidelity to the critical software execution in large scale network settings. To achieve better scalability, we have developed a distributed emulation system. However, we find synchronization overhead increases linearly as the number of machines increases. Application lookahead, the ability to predict future behaviors of software, may help reducing overhead for performance gain. In this chapter, we study the impacts of application lookahead on our distributed emulation testbed. We find that application lookahead can greatly reduce synchronization overhead and improve speed by up to 3 times in our system, but incorrect lookahead may affect application fidelity to different degree depending on application categories.

5.1 Overview of Lookahead

Today's quality of life is highly dependent on the successful operations of many large-scale networks, such as the Internet, cellular networks, and enterprise networks. It is essential to evaluate applications and protocols across development phases (e.g., design, implementation, testing, and verification) using testing systems, especially those with the capability to conduct large-scale network experiments. Simulation testbeds are widely used because of scalability and flexibility. However, the fidelity of simulation models is always being challenged because of model simplification and abstraction. Integrating emulation well complements a simulation testbed by offering high

fidelity, since real programs are executed on real operating systems instead of executing models to advance experiments. Therefore, we developed such a network testbed in our prior work by marrying a parallel simulator with a virtualization-based emulator [45]. The testbed runs on a single shared-memory server, and is capable to emulate 300+ virtual environments (VEs) concurrently thanks to the lightweight OS-level virtualization technologies.

To enable experiments with larger number of emulated nodes, we have extended our testbed to support distributed emulation. Good scalability implies not only the ability to emulate more nodes, but also to emulate them efficiently. Since emulated nodes now reside on different physical machines, the synchronization overhead can dramatically increase comparing with the shared-memory system. Hence, we investigate techniques to extract application lookahead from the historical application-level behaviors to reduce the synchronization overhead, and thus achieve better scalability. Lookahead is critical to the performance of conservative parallel discrete-event simulation, and our results indicate that it is also important to emulation, e.g. emulation with application lookahead is up to 3X faster in some of our experiment setups. Loosely speaking, simulation lookahead is the lower bound of the time that a logical process will not affect other logical processes. The application lookahead we investigated in this work is defined as the lower bound of the time that any emulated nodes will not affect any simulated entities. The application lookahead is calculated by a neural-network-based model based on observed historical data. It is difficult for our predication model to produce the exact lookahead. The lookahead may be too small, which means there is still room for performance improvement; or even worse, the lookahead may be too large, which has negatively impact on fidelity. In addition, the computational overhead of computing the lookahead is not negligible. Therefore, we have performed extensive studies on the impact of application lookahead, i.e. speed and fidelity, with various network scenarios. We find that application lookahead can greatly reduce synchronization frequency and overhead in some setups, but it may affect application fidelity to different degree depending on application categories. The results serve as guidelines for users of our testbed: in which conditions one ought to consider application lookahead to speed up their experiments, and in which conditions one should not.

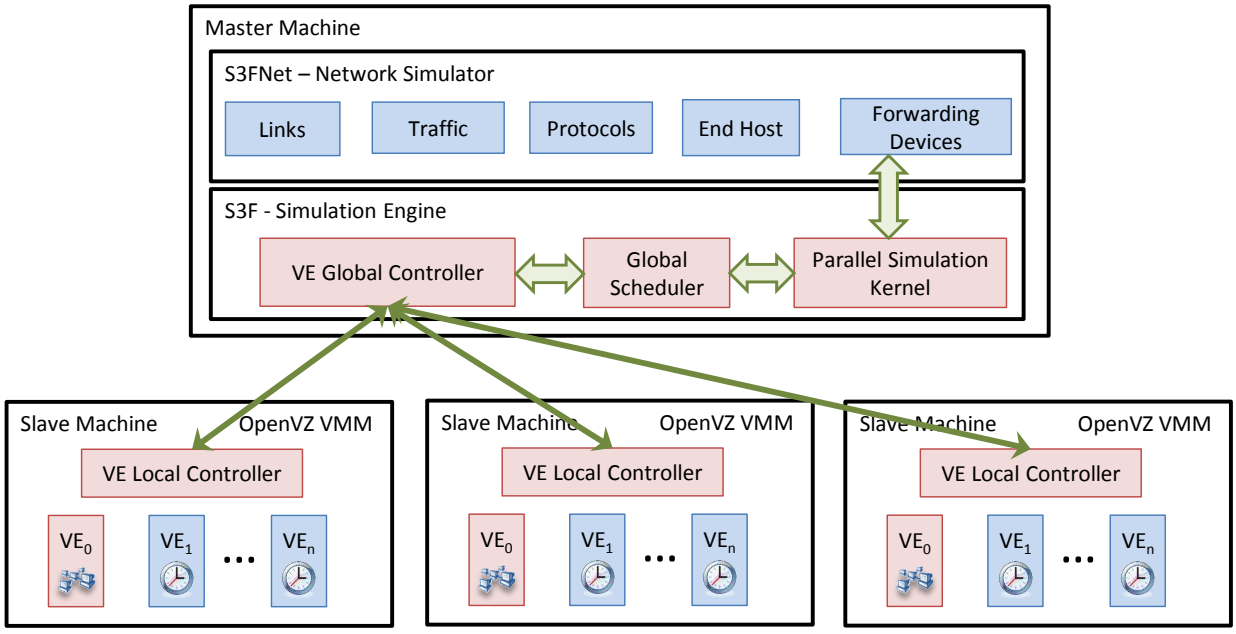


Figure 5.1: Network Testbed Architecture with Distributed Emulation Support

5.2 Distributed Emulation Design

System Architecture

Our testbed consists of three major components, an OpenVZ-based network emulator embedded in virtual time, a network simulator, and a parallel simulation engine responsible for coordinating operations of other two components. We can run 300+ emulated hosts on a single physical machine because of the lightweight OS-level virtualization. To further increase the scale of the experiments the testbed can conduct, we have developed the distributed emulation capability in our tesbed.

Figure 5.1 depicts the system architecture to support distributed emulation. A master server machine is connected to multiple slave machines via TCP/IP over gigabit Ethernet links. Each slave manages a group of local containers running on the same physical machine, and independently advances program states during its emulation window. The master has the knowledge on global network topology. It is responsible to coordinate all slave machines through a global synchronization algorithm, to manage cross-slave events, and to perform the simulation experiments.

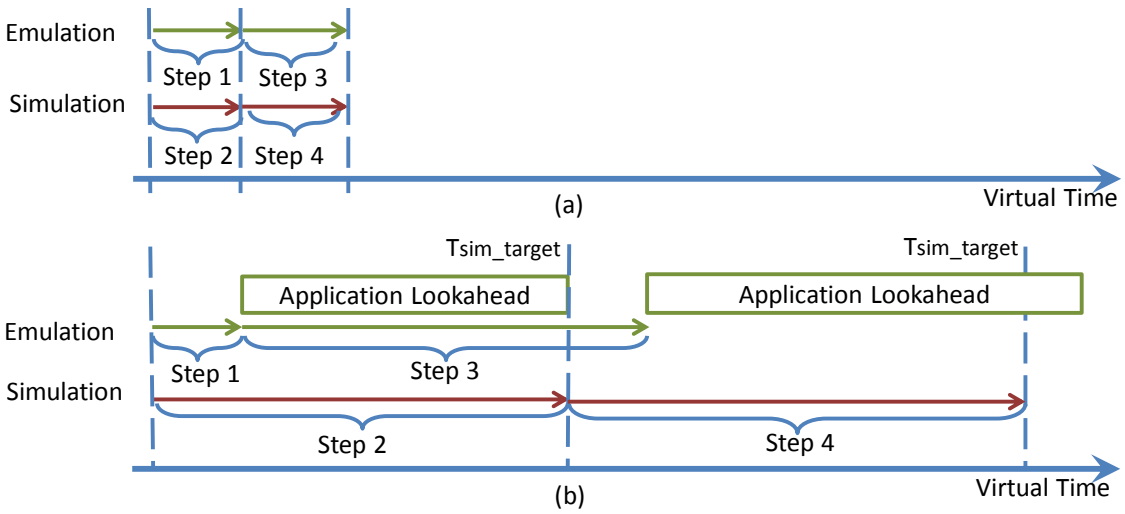


Figure 5.2: Experiments advancement (a) without app lookahead (b) with app lookahead

Three types of information are exchanged between the slaves and the master: control commands, network data packets, and application lookahead.

Global Synchronization Algorithm

Our systems consists of two subsystems: emulation and simulation, and we design a global synchronization algorithm to integrate the two systems based on virtual time (Chapter 4 [45]). In this chapter, we revisit the synchronization algorithm to incorporate the application lookahead. Figure 5.2 describes how a network experiment advances in our testbed with application lookahead disabled and enabled. Emulation and simulation always execute their cycles alternatively. Without application lookahead, emulation always runs ahead of simulation, while with application lookahead, emulation and simulation appears like a racing game, one may run ahead of the other at any cycle. Application lookahead is explored to improve scalability, especially for the distributed version. Three factors determine a high quality application lookahead:

- large application lookahead on average, which implies small synchronization overhead,
- small overhead for computing the lookahead, and

- high accuracy of the predicted lookahead.

The three factors in many cases are unsurprisingly contradictory to one another. We investigate the performance gain and the fidelity loss with various network scenarios, and results are presented in Section 5.4. The results serves as guidelines on the types of application that are beneficial for users to turn on the application lookahead functionality in our testbed.

The notations used in this section are listed in Table 5.1, and the high-level global synchronization algorithm for the distributed testbed is described in Algorithm 5.1. Let us define t_{emu} be the current emulation time (OpenVZ virtual time), t_{sim} be the current simulation time, ESW be the emulation synchronization window (the length of the next emulation advancement), and SSW be the simulation synchronization window (the length of the next simulation advancement). t_{sim_target} is the time where the simulator hands the control over to the network emulator, and this is the farthest time simulation can advance to its best knowledge.

Algorithm 5.1 Global Scheduler

```
1:  while true do
2:    if t_sim = t_sim_target then
3:      compute ESW
4:      run emulation for ESW
5:      inject packets to simulation
6:      t_sim_target ← t_emu + min_{VE_e}{lookahead_e}
7:    else
8:      compute SSW
9:      run simulation for SSW
10:     t' ← the first executed deliver-to-VE event in previous SSW
11:     t_sim_target ← min{t_sim_target, max{t', t_emu}}
12:   end if
13: end while
```

Equation 5.1 below illustrates how ESW is calculated:

$$ESW = \max\left\{\alpha \times TS, \min_{timeline\ i}\{P_i\} - t_{emu}\right\} \tag{5.1}$$

Table 5.1: Notation Descriptions

Notation	Description
t_{emu}	current emulation time: OpenVZ virtual time
t_{sim}	current simulation time
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
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
$l_{i,e}$	application lookahead of entity e in timeline i, computed by VE Controller in every ESW; $l_{i,e} = +\infty$ if $e \in E_{sim}$

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:

$$P_i = \min\left\{\left[\min\left(n_i^{sim}, \min_{entity\ e}\{l_{i,e}\}\right) + B_i\right], n_i^{emu}\right\} \tag{5.2}$$

and B_i is the minimum channel delay from timeline i:

$$B_i = \min_{outchannel\ j}\left\{w_{i,j} + \min_{inchannel\ k}\{r_{i,j,k}\}\right\} \tag{5.3}$$

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. Since emulation will advance beyond simulation, the packet is not available to emulation until simulation catches up and finishes processing that event. The P_i calculation prevents a VE from running too far ahead and bypassing a potential packet delivery event.

Equation 5.4 below illustrates how SSW is calculated:

$$SSW = \min\left\{t_{sim\_target}, \min_{timeline\ i}\{Q_i\}, \min_{timeline\ i}\{R_i\}\right\} - t_{sim} \tag{5.4}$$

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:

$$Q_i = n_i + C_i \tag{5.5}$$

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

$$C_i = \min_{outchannel\ j}\left\{w_{i,j} + \min_{inchannel\ x}\{s_{i,j,x}\}\right\} \tag{5.6}$$

and R_i is the lower bound of the time that an event of timeline i can potentially affect a VE-proxy entity in the simulation system, for the global scheduler to decide the next SSW_l:

$$R_i = \max\{n_i + B_i, t_{emu}\} \tag{5.7}$$

Compared with the algorithm specified in Chapter 4, we revise the algorithm to better exploit lookahead. Specifically, when lookahead does not present, simulation may not run beyond emulation due to lack of knowledge of emulation. However, with application lookahead and assuming it is correct, simulation can calculate the lower bound of time before which no events will be generated from emulation side, so that it can safely advance beyond emulation for this period. Consequently, the next ESW is also enlarged according to Equation 5.2, as simulation's running beyond also pushes next event n_i^{sim} and n_i^{emu} forward. One main bottleneck is that we currently do not have conditional lookahead, i.e. the ability to predict application behavior after a particular packet receipt, therefore we have to consider the worst case: the application may immediately send a packet in response to a recent packet receipt (Algorithm 5.1 line 11). Future work of predicting conditional lookahead may help further enlarge ESW and reduce overhead.

5.3 Implementation of Application Lookahead

We next present implementation details about how application lookahead is predicted, as well as analysis of lookahead error. In this chapter we mainly focus on the impacts of application lookahead, and we leave how to predict lookahead using other various approaches as future work.

Lookahead Using Time Series Forecasting

As discussed in Section 5.2, application lookahead is the ability to predict future behavior of applications. When it comes to network emulation, we are particularly interested in knowing the time of next packet sent from an application. Since application behaviors are fully determined

by their executables and runtime environments, one possible way to predict application lookahead is based on binary code. However, this approach is challenging not only due to the complexity of executable code, but also because runtime environment brings uncertainty, e.g. multi-thread scheduling controlled by operating system. While network emulation running native application code improves functional fidelity, it also makes exact lookahead more difficult.

Rather than trying to analyze the application code itself, we present an approach based on applications' packet inputs and outputs. Packets sent and received by an application can be viewed as a time series, and therefore the lookahead problem can be converted to a time series forecasting problem. To form a time series, all the packets sent or received by an application are sorted by timestamps, so that each packet naturally becomes observation vector y:

$$y = (ts, a_1, a_2, \ldots, a_x) \tag{5.8}$$

where ts is the timestamp relative to the previous packet, and a_i is the i-th attribute of this packet. Packet attributes may contain packet size or other application specific fields. They are to help indicate packet content and are crucial to forecasting accuracy. We found that attributes containing packet size (positive value for sent packets and negative value for received packets), TCP `SYN` flag, and TCP `FIN` flag perform well for the applications we have tested in this chapter. For more complex applications, using more attributes may improve accuracy but at the cost of slower lookahead computation speed.

We use artificial neural networks (ANNs) to solve the time series forecasting problem [83]. Time series forecasting typically assumes there is an underlying relationship between future values and past observations. Formally, a future value is a function (either known or unknown) of past observations:

$$y_{t+1} = f(y_t, y_{t-1}, \ldots, y_{t-w+1}) \tag{5.9}$$

where y_t is the t-th observation and w is the forecasting window size. As ANNs are universal function approximators [38] [53], it can be trained to perform the f function mapping using his-

torical data. Since we are only interested in the duration within which an application will not send a packet, the ANN is only trained to predict the timestamp field ts of observation vector y_{t+1}, ignoring all other attribute fields a_i. In case that the next observation in historical data y_{t+1} is a received packet, using timestamp ts of this packet as training target does not violate the semantic of application lookahead — a period within which an application will not send a packet. In fact, ts is the maximum lookahead amount given this training data, as the y_{t+1} packet receipt may suppress other packet sends.

ANN models for forecasting are application dependent, as different applications behave differently. To train an ANN for a given application, we run the application under its typical setup and collect the sent and received packet trace. For better accuracy and generalization, the trace had better be comprehensive to cover most the functionalities of the application. Consequently, when collecting trace for training, we run the application under different setup, e.g. with different parameters, and under different network conditions.

Lookahead Error Handling and Analysis

As previously defined, application lookahead is defined as a predict amount of time in future, within which an application will not send a packet. Ideally, application lookahead should be the exact time of the next sent packet given that the application does not receive a packet before that. Unfortunately, due to application complexity and runtime uncertainty, lookahead may not be exact. On one hand, an actual packet send may occur later than the lookahead. According to our scheduling algorithm presented in Section 5.2, this underestimated lookahead may result in an unnecessary synchronization, i.e. a synchronization that does nothing. However, this only has impact on speed but not fidelity, as packets will be delivered to destination at the same time as that without lookahead.

On the other hand, an application *may* send a packet before the predicted lookahead. In this case, it may result in some emulation hosts' advancing too far ahead without noticing the presence of a potential received packet. Since our virtual time system used in emulation system is not exact

and may introduce error [86], it naturally allows packet arrival with a timestamp in the past and will try delivering late packets to the destination containers at the earliest possible time. However, an event with a timestamp in the past is strictly prohibited in conservative parallel discrete-event simulation. To fix this, when a packet sending event is presented to the simulation system, if this event has a timestamp smaller than the current simulation time, its timestamp is set to the current simulation time.

When a lookahead error happens, i.e. an application *may* send a packet before its lookahead. it may increase the one-way delay of a packet sending from one emulation host to another. Previous chapter shows that temporal error up to a timeslice (100 µs) does not introduce additional error to application behaviors (Section 4.4 [84]). However, incorrect lookahead may introduce additional temporal error beyond that, and this may have impacts to fidelity. Section 5.4 present experimental results demonstrating the impact of incorrectly lookahead to application fidelity, but we analyze such impacts to different application categories as follows.

Different applications have different sensitivity to the increased one-way packet delays caused by incorrect lookahead. We classify applications into the following three categories according to how much impact incorrect lookahead may introduce.

- **Local impact**: some applications are relatively insensitive to lookahead error. An increased one-way delay of a single packet brings minimal impact to the application functionality. Such error does not cumulate and thus will not affect later functionality. Typical instances of this category are those applications with long-term one-way traffic, such as transferring large files using FTP protocol. As long as the lookahead error is not large enough to trigger a functionality change (e.g. a FTP retransmission due to timeout), when an increased delay occurs on a single packet, later functionality is performed as if the lookahead error never occurs. In this case, lookahead error creates a *jitter* in application behavior.

- **Global impact**: some other applications are sensitive to lookahead error. An increase delay on a single packet may cumulate and postpone all later functionality. Many communicating

protocols behave in this way: when an application receives a packet, it performs some action and sends a reply packet, and the application on the other end will not advance until it receives this reply. In fact, FTP protocol may belong to this category if the file size is very small and most time is spent in control messages. However, as long as the increased delay is not long enough to trigger a timeout, application still perform the same subsequent actions, only at a later time. In this case, lookahead error creates a *skew* in application behavior.

- **Drastic impact**: finally, some applications are extremely sensitive to network delays. Any small error in delays may lead to either different application behavior or incorrect application output. Typical examples are those applications mainly based on network delay, e.g. ICMP ping message that measures round trip time. In Section 5.4, we provide an example by implementing an end-to-end available bandwidth measurement approach proposed by [42]. This approach mainly relies on measuring one-way delay at different sending rate, and therefore could be extremely sensitive to any lookahead error.

We provide the above framework to estimate applications' sensitivity to incorrect lookahead. Note that this classification is vague, i.e. some applications may locate around the boundary between two categories and have the characteristics of both. Detailed experiment results are presented in Section 5.4.

5.4 Evaluation

We evaluate the impacts of application lookahead in a distributed setup. The overall architecture is shown in Figure 5.1. The master machine is a server with 32 logical processors and 64GB memory. The master machine is connected with 4 slave machines via a gigabit switch. Each slave machine is a commodity laptop with 2 processors and 2GB memory. This setup demonstrates our distributed design can use relatively low-profile machines to achieve scalable distributed emulation.

Next we present our results showing the impacts of application lookahead to speed and fidelity

respectively. For experiment with application lookahead, as stated in Section 5.3, ANNs models are application dependent. In our experiments, we pre-trained ANNs for different applications, and the training time is not counted into the run time. Training an ANN is usually time-consuming, but a well-trained ANN is reusable until there is a functionality change in the application. In addition, we let the forecasting window size w defined in Section 5.3 be 10.

Lookahead Impacts on Speed

We first investigate application lookahead in different network setups. We start with using 2 slave machines with 50 VEs on each. Our setup contains 50 VE pairs, i.e. each VE is only connected to another one via a link. These 50 VE pairs run independently and are used to create enough workload for our testbed. We vary the link bandwidth from 100Kbit/s to 10Mbit/s, and vary the link propagation delay from 10µs to 1ms. On each VE pair, an iperf [12] client/server pair is running, sending UDP traffic at constant 100 Kbps. The results are shown in Table 5.2 and 5.3.

We find that lookahead may improve speed only for those high bandwidth and short delay setups, while it slightly slows down for those setups with low bandwidth and/or long delay. This can be explained by the average emulation synchronization windows (ESW) between the master machine and those slave machines. In low bandwidth and long delay setups (e.g. bw=100K, delay=1ms), due to the minimum Ethernet packet size, the minimum transfer delay from a VE to another is long (detailed referring to synchronization algorithm in Section 5.2). Large minimum transfer delays result in large ESWs, and hence low synchronization frequency/overhead even if lookahead is not presented. In this case, lookahead only brings limited benefits: it can still slightly increase ESW. However, the computation overhead introduced by the neural network overwhelms the benefits lookahead may bring, resulting in a slower execution speed in all.

We next test study different sending rate of applications, by varying sending rate from 100Kbps to 10Mbps. We use the above mentioned high bandwidth (10 Mbps) and low latency (10 µs) network setup to maximize the benefits lookahead may bring, and we still use the 2 slave machines with 50 VEs each setup. The result is shown in Table 5.4.

Table 5.2: Impacts of Lookahead to Speed — Various Network Scenario, No Lookahead

Abbreviations:
Diff — speed difference between without/with lookahead
Sync Win — size of emulation synchronization windows (ESW)
Ovhd — percentage time spent in synchronization
Avg LA — average lookahead amount over all VEs
Avg Min LA — average minimal lookahead amount
vtime — simulation time / virtual time
bw — link bandwidth, in bits per second
delay — link propagation delay

Setup	Speed (vtime/sec)	Diff (%)	Sync Win (vtime µs)	Ovhd (%)	Avg LA (vtime µs)	Avg Min LA (vtime µs)
bw=100K, delay=10µs	0.539	n/a	1226	6%	n/a	n/a
bw=100K, delay=100µs	0.540	n/a	1260	6%	n/a	n/a
bw=100K, delay=1ms	0.542	n/a	1278	6%	n/a	n/a
bw=1M, delay=10µs	0.421	n/a	420	15%	n/a	n/a
bw=1M, delay=100µs	0.451	n/a	468	14%	n/a	n/a
bw=1M, delay=1ms	0.503	n/a	809	9%	n/a	n/a
bw=10M, delay=10µs	0.207	n/a	100	31%	n/a	n/a
bw=10M, delay=100µs	0.243	n/a	141	26%	n/a	n/a
bw=10M, delay=1ms	0.459	n/a	498	14%	n/a	n/a

Table 5.3: Impacts of Lookahead to Speed — Various Network Scenario, With Lookahead

Abbreviations:
Diff — speed difference between without/with lookahead
Sync Win — size of emulation synchronization windows (ESW)
Ovhd — percentage time spent in synchronization
Avg LA — average lookahead amount over all VEs
Avg Min LA — average minimal lookahead amount
vtime — simulation time / virtual time
bw — link bandwidth, in bits per second
delay — link propagation delay

Setup	Speed (vtime/sec)	Diff (%)	Sync Win (vtime µs)	Ovhd (%)	Avg LA (vtime µs)	Avg Min LA (vtime µs)
bw=100K, delay=10µs	0.504	-6%	1316	6%	58500	1966
bw=100K, delay=100µs	0.504	-7%	1314	6%	58500	1982
bw=100K, delay=1ms	0.504	-7%	1315	6%	58500	1976
bw=1M, delay=10µs	0.472	+12%	965	8%	58500	1901
bw=1M, delay=100µs	0.483	+7%	992	7%	58500	2035
bw=1M, delay=1ms	0.486	-3%	1011	7%	58500	1944
bw=10M, delay=10µs	0.483	+133%	982	7%	58500	1920
bw=10M, delay=100µs	0.486	+100%	969	8%	58500	1965
bw=10M, delay=1ms	0.486	+6%	933	8%	58500	1980

Table 5.4: Impacts of Lookahead to Speed — Various Application Sending Rate

Abbreviations:
Diff — speed difference between without/with lookahead
Sync Win — size of emulation synchronization windows (ESW)
Ovhd — percentage time spent in synchronization
Avg LA — average lookahead amount over all VEs
Avg Min LA — average minimal lookahead amount
vtime — simulation time / virtual time
sending rate — application sending rate

(a) No Lookahead

Setup	Speed (vtime/sec)	Diff (%)	Sync Win (vtime µs)	Ovhd (%)	Avg LA (vtime µs)	Avg Min LA (vtime µs)
sending rate=100Kbps	0.207	n/a	100	31%	n/a	n/a
sending rate=1Mbps	0.202	n/a	100	31%	n/a	n/a
sending rate=10Mbps	0.119	n/a	100	18%	n/a	n/a

(b) With Lookahead

Setup	Speed (vtime/sec)	Diff (%)	Sync Win (vtime µs)	Ovhd (%)	Avg LA (vtime µs)	Avg Min LA (vtime µs)
sending rate=100Kbps	0.483	+133%	982	7%	58500	1920
sending rate=1Mbps	0.231	+14%	164	21%	6600	371
sending rate=10Mbps	0.111	-7%	100	17%	3700	55

We find lookahead improves speed only when the sending rate is low (100Kbps and 1Mbps), but slightly slows down when applications are sending traffic very frequently (10Mbps). This is intuitive because lower sending rate implies longer pause after each packet send, therefore we can get decent lookahead amount and increase the size of ESW. However, when an application is sending traffic at a very fast rate, the lookahead amount we can obtain is small. In addition, the overhead to predict lookahead is increased due to the increased traffic volume, since our approach to predict lookahead is based on historic traffic of applications.

Lastly, we stick with a scenario to which application lookahead could be potentially beneficial. The network has high bandwidth (10 Mbps) and low latency (10 µs), which makes simulation-emulation synchronization windows small when lookahead is not presented. On the other hand, the applications are sending traffic very infrequently — UDP traffic at 100 Kbps — which means there are lots of unnecessary synchronizations and this can be reduce by application lookahead. We vary the number of slave machines as well as the number of virtual environments (VEs) per slave. Still, in each setup the VEs are paired, and different link pairs are independent and are just to increase simulation/emulation load. As all the packets have to pass through the simulator within the master machine, it does not matter whether two VEs of a link pair are on the same slave or not. Baseline results without lookahead are shown in Table 5.5, and the ones with lookahead is given in Table 5.6.

We observed small synchronization windows (100 µs) when lookahead is not presented, resulting in a high synchronization overhead (more than 50% when VE density is low, e.g. 20 VEs/slave). As VE density increases, the synchronization overhead decreases due to improved synchronization efficiency per slave machines. But the overhead still increases when the number of slaves grows, and this is harmful to scalability. On the other hand, when application lookahead is given, simulation-emulation synchronization windows are greatly enlarged, resulting in lower overhead and higher execution speed (2X to 3X faster). As we can see, application lookahead brings performance gain.

We also notice the synchronization window decreases when the total number of VEs increases.

Table 5.5: Impacts of Lookahead to Speed — Various Scale, No Lookahead

Abbreviations:
Diff — speed difference between without/with lookahead
Sync Win — size of emulation synchronization windows (ESW)
Ovhd — percentage time spent in synchronization
Avg LA — average lookahead amount over all VEs
Avg Min LA — average minimal lookahead amount
vtime — simulation time / virtual time

Setup	Speed (vtime/sec)	Diff (%)	Sync Win (vtime µs)	Ovhd (%)	Avg LA (vtime µs)	Avg Min LA (vtime µs)
20 VEs/slave × 1 slave	0.445	n/a	100	51%	n/a	n/a
20 VEs/slave × 2 slaves	0.398	n/a	100	53%	n/a	n/a
20 VEs/slave × 4 slaves	0.350	n/a	100	56%	n/a	n/a
50 VEs/slave × 1 slave	0.234	n/a	100	30%	n/a	n/a
50 VEs/slave × 2 slaves	0.207	n/a	100	31%	n/a	n/a
50 VEs/slave × 4 slaves	0.193	n/a	100	32%	n/a	n/a
100 VEs/slave × 1 slave	0.138	n/a	100	17%	n/a	n/a
100 VEs/slave × 2 slaves	0.111	n/a	100	19%	n/a	n/a
100 VEs/slave × 4 slaves	0.102	n/a	100	20%	n/a	n/a

Table 5.6: Impacts of Lookahead to Speed — Various Scale, With Lookahead

Abbreviations:
Diff — speed difference between without/with lookahead
Sync Win — size of emulation synchronization windows (ESW)
Ovhd — percentage time spent in synchronization
Avg LA — average lookahead amount over all VEs
Avg Min LA — average minimal lookahead amount
vtime — simulation time / virtual time

Setup	Speed (vtime/sec)	Diff (%)	Sync Win (vtime µs)	Ovhd (%)	Avg LA (vtime µs)	Avg Min LA (vtime µs)
20 VEs/slave × 1 slave	1.302	+193%	3577	5%	58500	5282
20 VEs/slave × 2 slaves	1.226	+208%	2075	7%	58500	3424
20 VEs/slave × 4 slaves	1.088	+211%	1192	14%	58500	2388
50 VEs/slave × 1 slave	0.534	+128%	1711	4%	58500	3068
50 VEs/slave × 2 slaves	0.483	+133%	982	7%	58500	1920
50 VEs/slave × 4 slaves	0.403	+109%	525	12%	58500	1289
100 VEs/slave × 1 slave	0.272	+97%	983	4%	58500	1892
100 VEs/slave × 2 slaves	0.214	+93%	497	6%	58500	1252
100 VEs/slave × 4 slaves	0.169	+66%	297	11%	58500	744

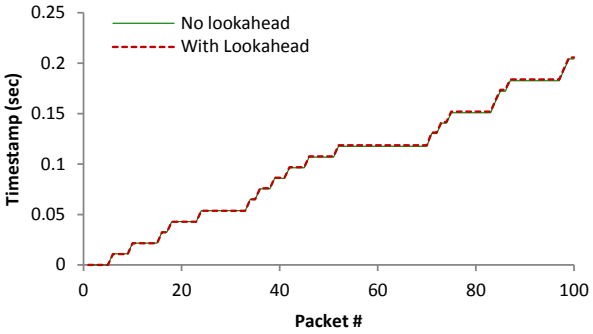
This is due to the synchronization mechanism we use (Section 5.2). Every time emulation is about to run, the global scheduler computes an ESW for all VEs to advance. This approach is a barrier-based synchronous approach, whose performance is sensitive is to minimal latency and minimal lookahead [65]. As seen from Table 5.6, the average lookahead is independent to the number of VEs, since all the applications are performing the same functionality. However, the average minimum lookahead decreases very quickly as the size grows, because different link pairs are independent and are running out of sync. We conclude that the reduced window size is the nature of barrier-based synchronization. Changing to asynchronous approach may improve performance for this scenario as the node degree is small, but this remains future effort.

Lookahead Impacts on Fidelity

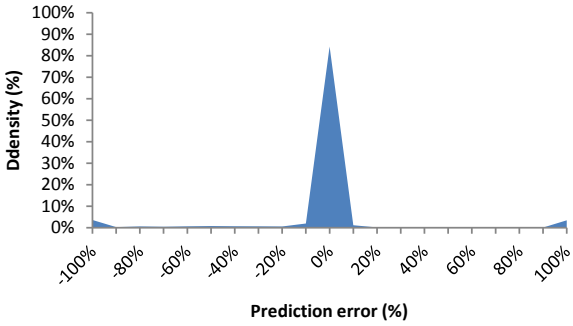
To investigate the impact of application lookahead to fidelity, we use one slave machine with 2 VEs, maximizing the impact of incorrect lookahead. As explained in previous subsection, increased number of VEs will result in smaller synchronization window, which reduces the likelihood of a VE's advancing too far ahead due to incorrect lookahead and causing an increased one-way packet delay. Detailed analysis refers to Section 5.3. The experiment setup we use is this subsection is 2 VEs connected by a link with 10 Mbps bandwidth and 10 μs latency. Next we present the impacts of lookahead to applications' fidelity, by discussing the three categories in Section 5.3 respectively.

No Impact We start with an application category whose behavior is very predictable. We use a traffic generator sending controllable one-way UDP traffic. The traffic source is sending packets whose size is exponential distribution with mean of 10 Kbyte, and the inter-packet arrival time is fixed at 10 ms. The results is shown in Figure 5.3, and we observe very high prediction accuracy and almost identical behavior after lookahead is enabled. This is because the constant inter-packet arrival time is easy to capture. Although the packet size is random, it is still easy to capture because most packets are larger than Ethernet MTU and will be fragmented, so that the last

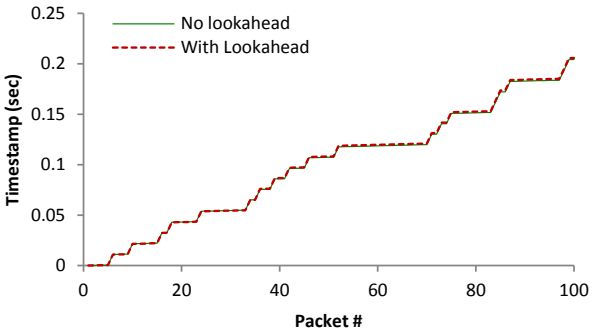
(a) Source Sent Packet Trace



(b) Source Lookahead Error Distribution



(c) Sink Received Packet Trace



(d) Sink Lookahead Error Distribution

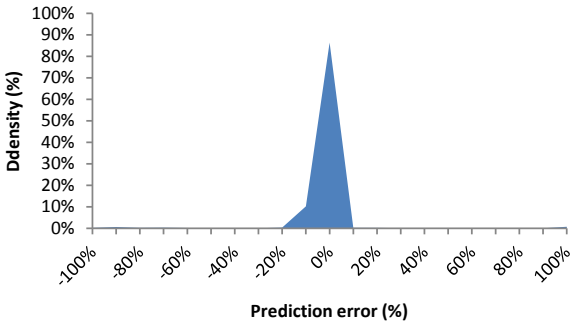


Figure 5.3: Impacts of Lookahead to Fidelity — No Impact

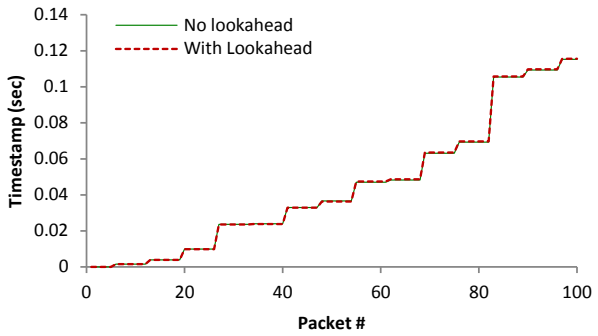
packet in fragmentation is usually not full. In general, this variable packet size fixed inter-packet arrival time is relatively easy to predict for the artificial neural network. This also represents those applications whose behavior is very predictable, e.g. deterministic and/or periodic functionality.

Local Impact To model the application category which only suffers local impact from incorrect lookahead, we use a traffic generator sending controllable one-way UDP traffic. The traffic source is sending 10-Kbyte constant size packets, and the inter-packet arrival time is exponential distribution with mean of 10 ms. We use the same seed for random number generator to ensure repeatability, but we use different seeds when training the ANNs to ensure generalization. The results are shown in Figure 5.4. We only plot the first 100 packets for conciseness.

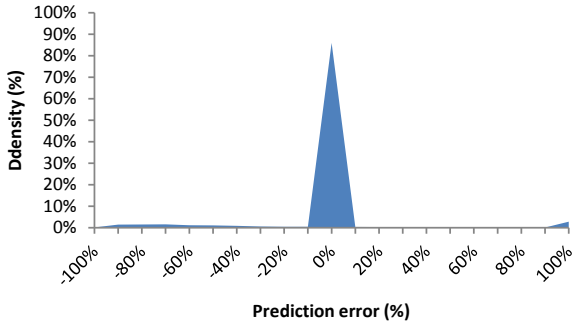
Because of the one-way traffic and the sink application does not have any feedback to the source, the source application is performing exactly the same behavior regardless of incorrect lookahead, as shown in Figure 5.4(a). On the other hand, when an incorrect lookahead occurs, i.e. the source application sends a packet before its predicted lookahead, and because the network delay is small, the sink application may advance too far ahead without noticing the packet. In this case, the packet arrival time will be late due to the error lookahead, as shown in Figure 5.4(c), near packet #10 and #65. However, such error is not propagating due to the one-way characteristic, and the subsequent packets may arrive at the right time. Application level statistics show lookahead does not affect overall throughput (0.976 Mbps without lookahead, and 0.975 Mbps with lookahead), but it does increase jitter (102 μs vs. 466 μs).

We also plot the lookahead prediction error in Figure 5.4(b)(d). A negative error means the actual next packet send is later than the lookahead (no harm for fidelity but may slow down execution speed), while a positive error indicates a packet is send before the lookahead (may affect fidelity). Since the traffic source is sending 10-Kbyte packets, it will be fragmented into a sequence of packets due to the limitation of Ethernet MTU. Most of time when emulation reaches the end of an ESW, the source VE is in the middle of sending those packet sequence and the next send packet will occur almost immediately. The ANN is able to capture this and predict correct lookahead

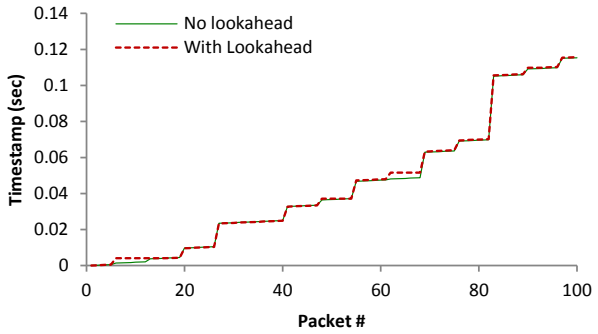
(a) Source Sent Packet Trace



(b) Source Lookahead Error Distribution



(c) Sink Received Packet Trace



(d) Sink Lookahead Error Distribution

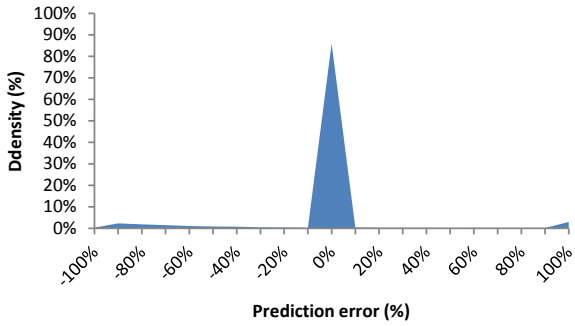


Figure 5.4: Impacts of Lookahead to Fidelity — Local Impact

(nearly zero lookahead in this case).

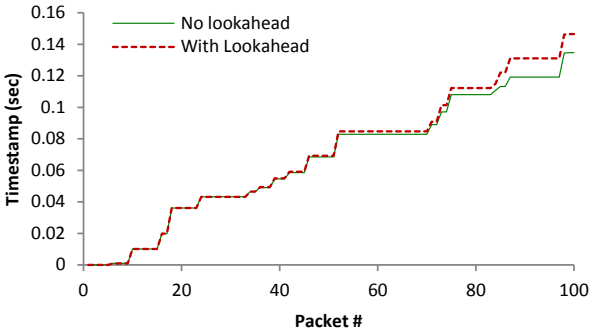
Global Impact To model the application category which may suffer global impact from incorrect lookahead, we change the traffic generator from open-loop to closed-loop. Specifically, the source application must wait for a reply to proceed for each sent packet. We also change the packet size to exponential distribution with mean of 10 Kbyte, and set the packet arrival time to exponential distribution with 10-ms mean. The result are shown in Figure 5.5.

In Figure 5.5, we can see that the error introduce by incorrect lookahead is cumulating, despite that the different is small at the beginning. This is due to the above-mentioned closed-loop characteristic, in which the error is cumulative and any late packet arrival will defer all subsequent actions. We also observe a higher prediction error, as the source application is sending smaller packets which are usually not fragmented. The ANN is unable to exactly predict the random distribution inter-packet arrival time. The application level statistics show lookahead shows lookahead results in a lower throughput (0.890 Mbps vs. 0.834 Mbps) as well as a larger jitter (0.878 ms vs. 1.584 ms).

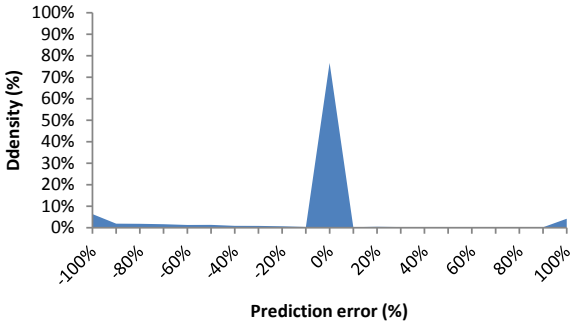
We provide another example of global impact by changing the packet size to exponential distribution with mean of 1 Kbyte, and setting the packet arrival time to exponential distribution with 1-ms mean. Compared with the previous setup, this one has tighter coupling between the client and server. The result is shown in Figure 5.6, and we can observer a clear skew. Again, the application level statistics show lookahead shows lookahead results in a lower throughput (0.792 Mbps vs. 0.720 Mbps) as well as a larger jitter (87 µs vs. 185 µs).

When it comes to the prediction error showing in Figure 5.6 (b)(d), we see larger prediction error due to the smaller packet size. Packet size that is smaller than Ethernet MTU makes lookahead prediction more difficult, because the characteristic of packet fragmentation no longer exists. Interestingly, we notice the most ANN forecasting errors are negative. We try to analyze why ANN behaves in this way, and find this is due to the ANN input normalization and its training objective function. Since ANN works best when input and output have small absolute values, we perform

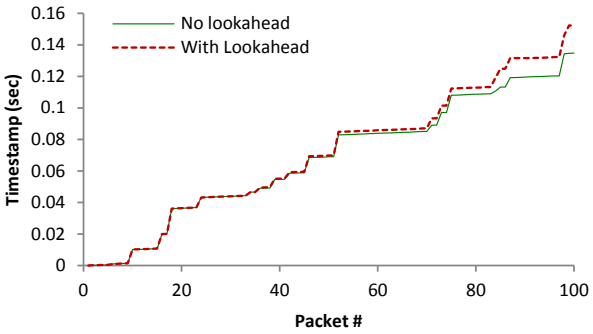
(a) Source Sent Packet Trace



(b) Source Lookahead Error Distribution



(c) Sink Received Packet Trace



(d) Sink Lookahead Error Distribution

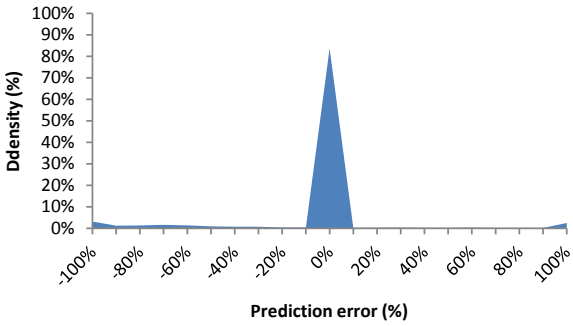
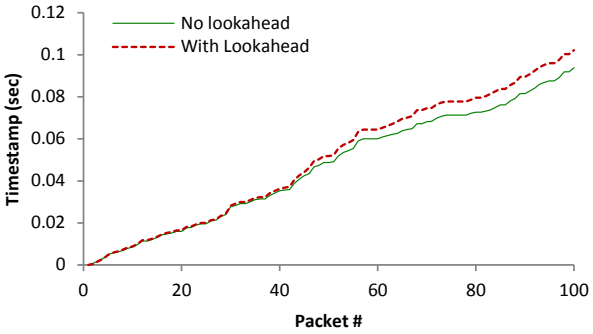
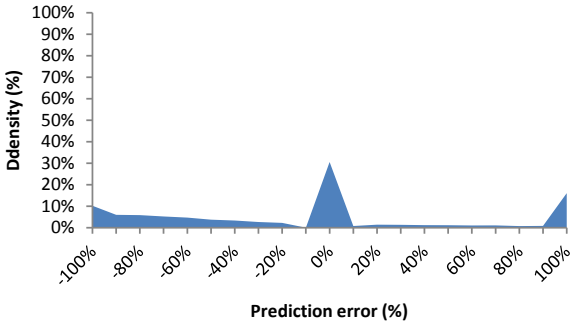


Figure 5.5: Impacts of Lookahead to Fidelity — Global Impact #1

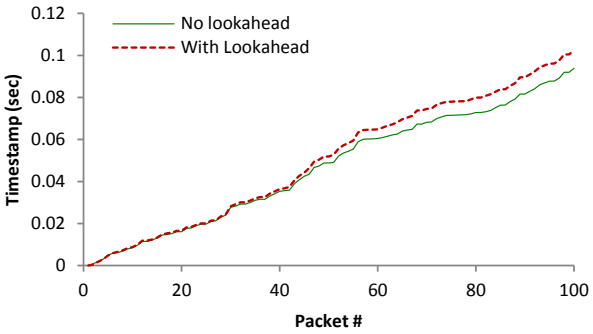
(a) Source Sent Packet Trace



(b) Source Lookahead Error Distribution



(c) Sink Received Packet Trace



(d) Sink Lookahead Error Distribution

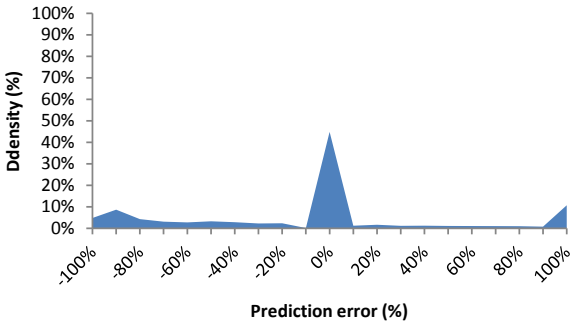


Figure 5.6: Impacts of Lookahead to Fidelity — Global Impact #2

Table 5.7: Results of End-to-End Available Bandwidth Measurement

	Average Output (Mbps)	Std Dev (Mbps)
No Lookahead	9.93	0.14
With Lookahead	3.84	1.48

a logarithmic operation on those large input values such as timestamp and packet size. Under the mean square error (MSE) ANN training objective function, the ANN tends to converge to an output value slightly smaller than the mean of exponential distribution. We find this approach is good for conservative parallel discrete-event simulation, as negative lookahead error is for safety concerns and does not violate the sematic of lookahead.

Drastic Impact Finally, to model the application category which is extremely sensitive to lookahead errors, we implement the end-to-end available bandwidth measure approach proposed in [42], and we slightly tune the algorithm to make it more sensitive one-way delay variations. For the setup with or without lookahead, we run the application for 10 times, and the results are shown in Table 5.7. When lookahead is not presented, the application correctly measures the available bandwidth (10 Mbps), and the results are quite stable given the small standard deviation. However, when lookahead is introduce, the application produces wrong answer and with a very large variation. That is because lookahead affect one-way delay, on which the application behavior highly depends.

In summary, we find the degree to which application lookahead may affect fidelity depends on applications' sensitivity of network delay. For those applications that can tolerate slight changed delay, lookahead may bring performance gain while not affecting fidelity too much. For those applications that are extremely sensitive to network delay, lookahead is not suitable regardless the potential performance gain.

5.5 Related Work

Lookahead in Parallel Discrete-Event Simulation

Lookahead has been extensively studied in conservative parallel discrete-event simulation, because good lookahead significantly improves the simulation performance by increasing the length of time by which logical processes can independently advance. Lookahead is defined as the ability to predict what will occur and what will not occur in simulated future [34]. A lookahead is loosely defined as the minimum time that a logical process will not affect event lists of other logical processes. Dimensions of lookahead based on the knowledges extracted from model characteristics and simulated applications are classified in [62]. A good lookahead reduces synchronization overhead and hence results in performance gain. Extensive researches have been performed to show the importance of simulation lookahead [32] [33] [71]. Sample applications of exploiting good simulation lookahead include simulations of stochastic queuing networks [60], continuous-time Markov chains [63], and wireless ad-hoc networks [55]. The difference with our history-based emulation lookahead is that the emulation lookahead is not absolutely correct (an event may occur before the lookahead). The quality of the emulation lookahead is dependent on how accurate our lookahead model can predict application-level behaviors.

Artificial Neural Networks

Inspired by biological systems, artificial neural networks (ANNs) are mathematical models that emulate biological neural networks [38] [53]. An ANN consists of a group of nodes (called neurons, or perceptrons), and these nodes are interconnected and perform functions in parallel. As the number of nodes increases, the capability of the network increases dramatically. For example, multi-layer perceptrons (MLPs) are universal function approximators, and it can approximate any continuous function at any desired accuracy [40]. ANNs are data-driven, and they can learn and generalize from past experience, in the way that they can correctly infer some unseen part from the training data. Due to these natures, ANNs have been widely used in many domains [81], including

forecasting [83]. Forecasting models usually assume there is an underlying relationship between the future and the past, and such relationship is usually unknown and complicated. ANNs are powerful tools for learning such underlying relationship, making them very capable of forecasting. Our work also uses ANNs for forecasting, but we have demanding accuracy requirements as network delays are in the order of milliseconds or even microseconds. The ANN-based forecasting only takes previously packet trace as input, and is occasionally inadequate to capture those uncertainties created by application internal states.

Time Synchronization Protocols

Clock synchronization is critical in distributed systems, especially for those time-sensitive applications. Historically, network nodes use protocols such as the Network Time Protocol (NTP) [57] for time synchronization. NTP synchronizes clocks based on a series of round-trip time (RTT) measurements, and its accuracy depends on not only the consistency of RTT but also the symmetry of link delay. Later standards such as the IEEE 1588 Precision Time Protocol (PTP) [29] and SynUTC [39] achieve higher precision by utilizing hardware assistance, despite that they still rely on the same assumptions of underlying networks. In particular, PTP achieves clock accuracy in the sub-microsecond range [29]. Our system also requires some sort of time synchronization to preserve causality, but it differs from the clock synchronization of distributed systems in several ways. Firstly, we synchronize virtual time rather than wallclock time. The virtual clocks do not advance at constant speed due to the timeslice mechanism, yet read clocks advance continuously at constant speed, which is the assumption of NTP and PTP. Secondly, while NTP and PTP try to minimize the clock difference, our system does not require strict synchronization of virtual clocks. Virtual clocks may run slightly out of sync without violating causality, depending on the transfer delay of the underlying simulated network.

5.6 Chapter Summary

We extend our network simulation/emulation testbed to support distributed emulation. We find synchronization overhead increases as the system size grows, and application lookahead may help reduce such overhead and achieve better scalability. We proposed an approach of application lookahead based on time series forecasting using artificial neural network. Through experiments, we find application lookahead can enlarge simulation-emulation synchronization windows and improve speed up to 3 times. However, the downside is that it may affect fidelity as lookahead could be wrong, and the degree to which lookahead affect fidelity highly depends on application behaviors. We conclude that lookahead is suitable for those applications that can tolerate small changes in network delay, in which case lookahead can improve speed without affect fidelity too much.

In this chapter we mainly focus on studying the impacts of application lookahead, while investigation about how to provide better lookahead using other approaches (e.g. application code analysis) is left as future work. Another possible future effort is to implement asynchronous synchronization between simulation and emulation, as it may overcome the reduced synchronization window size caused by scale increases.

Chapter 6

Conclusions and Future Directions

6.1 Summary of Thesis Research

Modeling is a very important and useful approach to study large-scale network systems, as it allows studies not physically realizable. In the domain of wireless networks, simulations are usually used to study new or existing designs, as it is economically and technically expensive to implement those designs using real hardware. In this dissertation, we are aiming to build and study a system for large-scale and high-fidelity wireless network simulation and emulation. To achieve this goal, our efforts focus on the following four areas. 1) We use sophisticated radio propagation models such as ray-tracing model and transmission line matrix model, validated by an anechoic chamber. We found that the errors that anechoic chamber eliminates are small relative to errors introduce by model uncertainty, indicating that future validations need not be attempted within such a chamber. 2) We present a timeslice-based virtual time system that achieves high functional fidelity (can run unmodified application executables) and high temporal fidelity (applications perceive time almost in the same way they do in real world). Our variable timeslice mechanism also allows users to achieve their desired temporal accuracy at fastest possible speed. 3) We integrate the virtual time system with our S3F parallel discrete-event simulator, and allow emulation to run distributedly across multiple machines. This can achieve the advantages of both simulation and emulation: emulation can be used to represent the execution of critical software, while simulation is used to model an extensive ensemble of background computation and communication. Through validation of application behavior and case study, we demonstrate that our testbed is high-fidelity, scalable, and easy to use. 4) We implement application lookahead, the ability to predict future behavior of

applications, in order to speed up simulation and emulation by reducing synchronization overhead. We find that application lookahead can improve speed by up to 3X, but incorrect lookahead due to software complexity and runtime uncertainty may affect fidelity to different degree depending on application categories.

6.2 Future Directions

To further extend the work of this dissertation, several future directions are worth considering.

Virtual Time System

The first future direction is to improve the virtual time system by providing better support for OS diversity, as well as modeling different resource type to enhance fidelity. Our current implementation is based on OpenVZ OS-level virtualization technique. Although it is attractive for its lightweight and scalability (can run 300+ VEs on a single machine), an OpenVZ container is not a fully functional operating system. On one hand, some operations such as inserting kernel modules are prohibited within a container, since all the containers have to share the same (Linux) kernel. On the other hand, our results of validating application behavior show that OpenVZ virtualization introduces small error, and we believe this is due to OpenVZ implementation itself. By migrating our virtual time system to other virtualization platforms such as Xen and QEMU, not only we can support more operating systems including Microsoft Windows, but also it could potentially eliminate the small error introduce by OpenVZ.

Besides OS diversity, one may also want to have accurate models for more resource type, such as disk. Our current virtual time system focuses on accurately modeling both network delay and CPU time. However, we do not model disk read/write behaviors, nor do we support multiprocessors to the containers. Having accurate models for disk and multiprocessors may be crucial to some applications such as cloud computing ones.

Simulation/Emulation Synchronization

The second future direction is to implement and study different synchronization approaches between the simulation subsystem and emulation subsystem. Our current implementation is a barrier-based one, i.e. the synchronous approach in conservative parallel discrete-event simulation's terminology. Although the synchronous approach is straightforward to implement, its performance is sensitive to the minimum latency and minimum lookahead. In particular, we find that the benefit brought by application lookahead degrades as the total number of emulation hosts increases. In our barrier-based synchronization approach, it is indeed the minimal application lookahead from all containers that takes effect, despite how large the average lookahead amount is. Changing to other synchronization approaches such as asynchronous approach or composite synchronization may improve performance when the node degree is small, and this remains future research effort.

On the other hand, our original design makes simulation and emulation take turns to run. This keeps our synchronization algorithm simple, and works well on the single-machine setup. However, when it comes to distributed setup, either the master machine (simulation) or the slave machines (emulation) are idle at any given time, and therefore computation resource is not fully utilized. Future efforts may design a synchronization mechanism that allows simulation and emulation to run concurrently whenever possible. This may maximize parallelism and achieve high execution speed, but requires more careful synchronizations.

Application Lookahead

A third future direction is to investigate how to better provide application lookahead, by both enhancing accuracy and reducing overhead. We find prediction application future behavior challenging due to the complexity of software itself and runtime uncertainty such as multi-thread scheduling. In this thesis we present an approach to provide application lookahead using time series forecasting based on artificial neural network (ANN). We mainly focus on studying the impacts of application lookahead on simulation/emulation in regard to both speed and fidelity, in

order to provide motivations to future studies of application lookahead. Since lookahead is crucial to the performance of conservative parallel discrete-event simulation, it is worth future effort to investigate how to compute application lookahead both accurately and efficiently.

In regard to the ANN-based model, studying how the ANN model behaves in various network conditions and application setups is important. In particular, we would like to investigate the sensitivity of ANN model to network packet loss, to see how well the model can capture complex realistic scenarios. In addition, approaches based on executable code analysis are promising as the behavior of applications is defined by their code, but how to quickly provide lookahead using this requires future effort. Finally, conditional lookahead, the ability of predict what will happen in the software after an input event such as packet receipt, may help further reduce unnecessary synchronizations. Currently, since we have no idea how a container will response to a packet receipt, we have to consider the worst case — it may send a packet immediately. However this is sometimes not the case and will cause extra synchronization overhead.

References

[1] "The network simulator (ns-2)." http://www.isi.edu/nsnam/ns, 2007.

[2] "The MadWifi project." http://madwifi-project.org/, 2009.

[3] "The ns-3 project." http://www.nsnam.org/, 2010.

[4] "OpenVZ: a container-based virtualization for linux." http://wiki.openvz.org/Main_Page, 2010.

[5] "OPNET modeler: scalable network simulation." http://www.opnet.com/solutions/network_rd/modeler.html, 2010.

[6] "Soekris engineering net4521." http://www.soekris.com/net4521.htm, 2010.

[7] "The User-Mode linux kernel." http://user-mode-linux.sourceforge.net/, 2010.

[8] "Virtuozzo containers." http://www.parallels.com/products/pvc46/, 2010.

[9] "GTNetS." http://www.ece.gatech.edu/research/labs/MANIACS/GTNetS, 2011.

[10] "Scalable network technologies." http://scalable-networks.com, 2011.

[11] "Apache: HTTP server project." http://httpd.apache.org/, 2012.

[12] "Iperf network performance testing tool." http://iperf.sourceforge.net/, 2012.

[13] "Lynx: a text browser for the World Wide Web." http://lynx.browser.org/, 2012.

[14] D. Aguayo, J. Bicket, S. Biswas, G. Judd, and R. Morris, "Link-level measurements from an 802.11 b mesh network," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, 2004.

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

[16] J. Andrews, "Linux: Running at 10,000 hz." http://kerneltrap.org/node/1766, 2003.

[17] S. Bai and D. M. Nicol, "Acceleration of wireless channel simulation using gpus," in *Wireless Conference (EW), 2010 European*, IEEE, 2010.

[18] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, 2003.

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

[20] F. Bellard, "QEMU, a fast and portable dynamic translator," in *USENIX*, 2005.

[21] C. Benvenuti, *Understanding Linux Network Internals*. O'Reilly Media, 2005.

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

[23] G. Bianchi, "Performance analysis of the IEEE 802.11 distributed coordination function," *Selected Areas in Communications, IEEE Journal on*, vol. 18, no. 3, 2000.

[24] P. K. Biswas, C. Serban, A. Poylisher, J. Lee, S.-C. Mau, R. Chadha, C.-Y. Chiang, R. Orlando, and K. Jakubowski, "An integrated testbed for virtual ad hoc networks," in *Testbeds and Research Infrastructures for the Development of Networks & Communities and Workshops, 2009. TridentCom 2009. 5th International Conference on*, IEEE, 2009.

[25] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: an overlay testbed for broad-coverage services," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 3, 2003.

[26] J. Corbet, A. Rubini, and G. Kroah-Hartman, *Linux Device Drivers, Thrid Edition*. O'Reilly Media, 2005.

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

[28] P. M. Dickens, P. Heidelberger, and D. M. Nicol, "A distributed memory LAPSE: Parallel simulation of message-passing programs," in *ACM SIGSIM Simulation Digest*, ACM, 1994.

[29] J. Eidson and K. Lee, "IEEE 1588 standard for a precision clock synchronization protocol for networked measurement and control systems," in *Sensors for Industry Conference, 2002. 2nd ISA/IEEE*, IEEE, 2002.

[30] M. A. Erazo, Y. Li, and J. Liu, "SVEET! a scalable virtualized evaluation environment for tcp," in *Testbeds and Research Infrastructures for the Development of Networks & Communities and Workshops, 2009. TridentCom 2009. 5th International Conference on*, IEEE, 2009.

[31] R. Fujimoto, "Parallel discrete event simulation," in *Proceedings of the 21st conference on Winter simulation*, ACM, 1989.

[32] R. M. Fujimoto, "Performance measurements of distributed simulation strategies," tech. rep., DTIC Document, 1987.

[33] R. M. Fujimoto, "Lookahead in parallel discrete event simulation," tech. rep., DTIC Document, 1988.

[34] R. M. Fujimoto, "Parallel discrete event simulation," *Communications of the ACM*, vol. 33, no. 10, 1990.

[35] A. Grau, S. Maier, K. Herrmann, and K. Rothermel, "Time jails: A hybrid approach to scalable network emulation," in *Principles of Advanced and Distributed Simulation, 2008. PADS'08. 22nd Workshop on*, IEEE, 2008.

[36] D. Gupta, K. V. Vishwanath, M. McNett, A. Vahdat, K. Yocum, A. Snoeren, and G. M. Voelker, "DieCast: Testing distributed systems with an accurate scale model," *ACM Transactions on Computer Systems (TOCS)*, vol. 29, no. 2, 2011.

[37] E. B. Hamida, G. Chelius, and J. M. Gorce, "Impact of the physical layer modeling on the accuracy and scalability of wireless network simulation," *Simulation*, vol. 85, no. 9, 2009.

[38] J. A. Hertz, A. S. Krogh, and R. G. Palmer, *Introduction to the theory of neural computation*, vol. 1. Westview press, 1991.

[39] R. Holler, M. Horauer, G. Gridling, N. Kero, U. Schmid, and K. Schossmaier, "SynUTC-high precision time synchronization over ethernet networks," *CERN EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH-REPORTS-CERN*, no. 3, 2002.

[40] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, 1989.

[41] M. F. Iskander and Z. Yun, "Propagation prediction models for wireless communication systems," *Microwave Theory and Techniques, IEEE Transactions on*, vol. 50, no. 3, 2002.

[42] M. Jain and C. Dovrolis, "End-to-end available bandwidth: Measurement methodology, dynamics, and relation with TCP throughput," in *ACM SIGCOMM Computer Communication Review*, ACM, 2002.

[43] X. Jiang and D. Xu, "VIOLIN: Virtual internetworking on overlay infrastructure," *Parallel and Distributed Processing and Applications*, 2005.

[44] D. Jin, D. M. Nicol, and M. Caesar, "Efficient gigabit ethernet switch models for large-scale simulation," in *Principles of Advanced and Distributed Simulation (PADS), 2010 IEEE Workshop on*, IEEE, 2010.

[45] D. Jin, Y. Zheng, H. Zhu, D. M. Nicol, and L. Winterrowd, "Virtual time integration of emulation and parallel simulation," in *Proceedings of the 2012 ACM/IEEE/SCS 26th Workshop on Principles of Advanced and Distributed Simulation*, IEEE Computer Society, 2012.

[46] G. Judd and P. Steenkiste, "Characterizing 802.11 wireless link behavior," *Wireless Networks*, vol. 16, no. 1, 2010.

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

[48] W. D. Kelton and A. M. Law, *Simulation modeling and analysis*. McGraw Hill Boston, MA, 2000.

[49] P. T. Kuruganti and J. Nutaro, "A comparative study of wireless propagation simulation methodologies: Ray tracing, FDTD, and event based TLM," in *Proc. Huntsville Simulation Conference*, 2006.

[50] D. I. Laurensen, "Indoor radio channel propagation modelling by ray tracing techniques," Ph.D. dissertation, University of Edinburgh, 1994.

[51] G. Liang and H. L. Bertoni, "A new approach to 3-d ray tracing for propagation prediction in cities," *Antennas and Propagation, IEEE Transactions on*, vol. 46, no. 6, 1998.

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

[53] R. Lippmann, "An introduction to computing with neural nets," *ASSP Magazine, IEEE*, vol. 4, no. 2, 1987.

[54] J. Liu, "Parallel real-time immersive modelling environment (PRIME), scalable simulation framework (SSF), user's manual. colorado school of mines department of mathematical and computer sciences (2006)," 2006.

[55] J. Liu and D. M. Nicol, "Lookahead revisited in wireless network simulations," in *Proceedings of the sixteenth workshop on Parallel and distributed simulation*, IEEE Computer Society, 2002.

[56] J. Mayo, R. Minnich, D. Rudish, and R. Armstrong, "Approaches for scalable modeling and emulation of cyber systems: LDRD final report," tech. rep., Sandia report, SAND2009-6068, Sandia National Lab, 2009.

[57] D. Mills, "Network time protocol (version 3) specification, implementation and analysis," 1992.

[58] D. Nicol, "Tradeoffs between model abstraction, execution speed, and behavioral accuracy," in *European Modeling and Simulation Symposium*, 2006.

[59] D. Nicol, J. Liu, M. Liljenstam, and G. Yan, "Simulation of large scale networks using SSF," in *Proceedings of the 2003 Winter Simulation Conference, 2003.*, IEEE, 2003.

[60] D. M. Nicol, *Parallel discrete-event simulation of FCFS stochastic queueing networks*, vol. 23. ACM, 1988.

[61] D. M. Nicol, "The cost of conservative synchronization in parallel discrete event simulations," *Journal of the ACM (JACM)*, vol. 40, no. 2, 1993.

[62] D. M. Nicol, "Principles of conservative parallel simulation," in *Proceedings of the 28th conference on Winter simulation*, IEEE Computer Society, 1996.

[63] D. M. Nicol and P. Heidelberger, "A comparative study of parallel algorithms for simulating continuous time markov chains," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 5, no. 4, 1995.

[64] D. M. Nicol, D. Jin, and Y. Zheng, "S3F: The scalable simulation framework revisited," in *Proceedings of the Winter Simulation Conference*, Winter Simulation Conference, 2011.

[65] D. M. Nicol and J. Liu, "Composite synchronization in parallel discrete-event simulation," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 13, no. 5, 2002.

[66] J. Nutaro, "A discrete event method for wave simulation," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 16, no. 2, 2006.

[67] J. Nutaro, P. T. Kuruganti, R. Jammalamadaka, T. Tinoco, and V. Protopopescu, "An event driven, simplified TLM method for predicting path-loss in cluttered environments," *Antennas and Propagation, IEEE Transactions on*, vol. 56, no. 1, 2008.

[68] P. Padala, X. Zhu, Z. Wang, S. Singhal, K. G. Shin, *et al.*, "Performance evaluation of virtualization technologies for server consolidation," tech. rep., HP Labs Tec. Report, 2007.

[69] B. T. Phong, "Illumination for computer generated pictures," *Communications of the ACM*, vol. 18, no. 6, 1975.

[70] L. Qiu, Y. Zhang, F. Wang, M. K. Han, and R. Mahajan, "A general model of wireless interference," in *Proceedings of the 13th annual ACM international conference on Mobile computing and networking*, ACM, 2007.

[71] D. A. Reed, A. D. Malony, and B. D. McCredie, "Parallel discrete event simulation using shared memory," *IEEE Transactions on Software Engineering*, vol. 14, no. 4, 1988.

[72] S. Y. Seidel and T. S. Rappaport, "Site-specific propagation prediction for wireless in-building personal communication system design," *Vehicular Technology, IEEE Transactions on*, vol. 43, no. 4, 1994.

[73] A. Sobeih, J. C. Hou, L.-C. Kung, N. Li, H. Zhang, W.-P. Chen, H.-Y. Tyan, and H. Lim, "J-Sim: a simulation and emulation environment for wireless sensor networks," *Wireless Communications, IEEE*, vol. 13, no. 4, 2006.

[74] M. Takai, R. Bagrodia, K. Tang, and M. Gerla, "Efficient wireless network simulations with detailed propagation models," *Wireless Networks*, vol. 7, no. 3, 2001.

[75] A. Tanenbaum, *Modern Operating Systems*. Prentice Hall, 2007.

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

[77] 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, 2002.

[78] N. H. Vaidya, J. Bernhard, V. Veeravalli, P. Kumar, and R. Iyer, "Illinois wireless wind tunnel: a testbed for experimental evaluation of wireless networks," in *Proceedings of the 2005 ACM SIGCOMM workshop on Experimental approaches to wireless network design and analysis*, ACM, 2005.

[79] B. Walters, "VMware virtual platform," *Linux journal*, vol. 1999, no. 63es, 1999.

[80] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, 2002.

[81] B. Widrow, D. E. Rumelhart, and M. A. Lehr, "Neural networks: Applications in industry, business and science," *Communications of the ACM*, vol. 37, no. 3, 1994.

[82] C.-F. Yang, B.-C. Wu, and C.-J. Ko, "A ray-tracing method for modeling indoor wave propagation and penetration," *Antennas and Propagation, IEEE Transactions on*, vol. 46, no. 6, 1998.

[83] G. Zhang, B. E. Patuwo, and M. Y. Hu, "Forecasting with artificial neural networks: The state of the art," *International journal of forecasting*, vol. 14, no. 1, 1998.

[84] 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*, Winter Simulation Conference, 2012.

[85] Y. Zheng and D. M. Nicol, "Validation of radio channel models using an anechoic chamber," in *Proceedings of the 2010 IEEE Workshop on Principles of Advanced and Distributed Simulation*, IEEE Computer Society, 2010.

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

[87] 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, 2012.

[88] Y. Zheng and N. Vaidya, "Repeatability of illinois wireless wind tunnel," tech. rep., University of Illinois at Urbana-Champaign, 2008.