1-16-2013

Leveraging Symbiotic Relationships for Emulation of Computer Networks

Miguel A. Erazo
Florida International University, miguel.erazo@gmail.com

DOI: 10.25148/etd.FI13042205

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

LEVERAGING SYMBIOTIC RELATIONSHIPS FOR EMULATION OF COMPUTER NETWORKS

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Miguel A. Erazo

2013

To: Dean Amir Mirmiran
College of Engineering and Computing

This dissertation, written by Miguel A. Erazo, and entitled Leveraging Symbiotic Relationships for Emulation of Computer Networks, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this dissertation and recommend that it be approved.

Shu-Ching Chen

Deng Pan

Armando Barreto

Jason Liu, Major Professor

Date of Defense: January 16, 2013

The dissertation of Miguel A. Erazo is approved.

Dean Amir Mirmiran
College of Engineering and Computing

Dean Lakshmi N. Reddi
University Graduate School

Florida International University, 2013

DEDICATION

Antes que todo quiero dar gracias a Dios, que me dio espezanzas y fuerzas para cumplir este logro.

Quiero dedicar esta Disertación a mis padres Abel y Lourdes; sin cuyo amor, esfuerzo, y dedicación, jamás esto hubiera sido posible.

En la misma dimensión quiero también dedicar este logro a mi esposa Adriana. Su ayuda, constante paciencia, e incondicional amor han sido la luz que me guió y me dio esperanza estos últimos tiempos. Mi hijo Sebastián ha sido otra motivación para seguir siempre adelante. Los amo a ambos.

No sería justo no hacer agradecer a mis hermanos Ana y Alejandro, que me alegraron los días. Mi suegra Ana y don Lucho nos dieron valiosos consejos y estuvieron en los momentos más difíciles apoyándonos.

Quiero tambien hacer mención especial a mi amigo de toda la vida Michi Berdeja, que siempre estuvo conmigo en los buenos y malos días y también a toda LA NAUSEA.

Los amo a todos.

ACKNOWLEDGMENTS

I wish to thank my advisor, Dr. Jason Liu. Jason's commitment to excellence has been a source of inspiration for me. Thank you for always taking time from your busy schedule for guiding me in this whole process. These years have been tough but I have learnt a lot.

I also want thank the other members of my Ph.D. committee, Professors Shu-Ching Chen, Deng Pan, and Armando Barreto for making time to support my dissertation.

I want to thank my constant collaborators and friends, Nathanael Van Vorst, Ting Li, and Hao jiang. Our many hours of discussions were enjoyable and good sources of fruitful research ideas. I am grateful that I was able to make this journey with you. I want to thank Stephan Eidenbenz at Los Alamos National Labs for the opportunity to work with you. My internship at LANL was stimulating, enjoyable, something I will remember for years to come.

ABSTRACT OF THE DISSERTATION

LEVERAGING SYMBIOTIC RELATIONSHIPS FOR EMULATION OF COMPUTER NETWORKS

by

Miguel A. Erazo

Florida International University, 2013

Miami, Florida

Professor Jason Liu, Major Professor

The lack of analytical models that can accurately describe large-scale networked systems makes empirical experimentation indispensable for understanding complex behaviors. Research on network testbeds for testing network protocols and distributed services, including physical, emulated, and federated testbeds, has made steady progress. Although the success of these testbeds is undeniable, they fail to provide: 1) *scalability*, for handling large-scale networks with hundreds or thousands of hosts and routers organized in different scenarios, 2) *flexibility*, for testing new protocols or applications in diverse settings, and 3) *inter-operability*, for combining simulated and real network entities in experiments. This dissertation tackles these issues in three different dimensions.

First, we present SVEET, a system that enables inter-operability between real and simulated hosts. In order to increase the scalability of networks under study, SVEET enables time-dilated synchronization between real hosts and the discrete-event simulator. Realistic TCP congestion control algorithms are implemented in the simulator to allow seamless interactions between real and simulated hosts. SVEET is validated via extensive experiments and its capabilities are assessed through case studies involving real applications.

Second, we present PrimoGENI, a system that allows a distributed discrete-event simulator, running in real-time, to interact with real network entities in a federated environment. PrimoGENI greatly enhances the flexibility of network experiments, through which a great variety of network conditions can be reproduced to examine what-if questions. Furthermore, PrimoGENI performs resource management functions, on behalf of the user, for instantiating network experiments on shared infrastructures.

Finally, to further increase the scalability of network testbeds to handle large-scale high-capacity networks, we present a novel symbiotic simulation approach. We present SymbioSim, a testbed for large-scale network experimentation where a high-performance simulation system closely cooperates with an emulation system in a mutually beneficial way. On the one hand, the simulation system benefits from incorporating the traffic metadata from real applications in the emulation system to reproduce the realistic traffic conditions. On the other hand, the emulation system benefits from receiving the continuous updates from the simulation system to calibrate the traffic between real applications. Specific techniques that support the symbiotic approach include: 1) a model downscaling scheme that can significantly reduce the complexity of the large-scale simulation model, resulting in an efficient emulation system for modulating the high-capacity network traffic between real applications; 2) a queuing network model for the downscaled emulation system to accurately represent the network effects of the simulated traffic; and 3) techniques for reducing the synchronization overhead between the simulation and emulation systems.

TABLE OF CONTENTS

CHAPTER PAGE

LIST OF FIGURES

FIGURE PAGE

CHAPTER 1

INTRODUCTION

1.1 Motivation

Network researchers use three principal methods to evaluate protocols: live experiments, emulation, and simulation. Live experiments on existing research testbeds, such as PlanetLab [PACR02], provide protocol designers with realistic distributed environments and traffic conditions that resemble the target system on which the protocols are expected to be deployed. These testbeds, however, do not provide the level of scalability and flexibility (for dynamically changing the network model, and injecting events to change the course of the experiments), necessary for testing and benchmarking protocols and applications under diverse network conditions. Alternatively, emulation testbeds, such as Emulab [WLS+02] and ModelNet [VYW+02], provide an environment where real implementations of the test applications can run directly in a native operating environment while the systems mimic the network behavior between the application instances. The capability of the emulation testbeds, however, is constrained by the physical limitations of the testbeds, such as the scale of the underlying platform, the processing speed, as well as the bandwidth and latency of the supporting hardware. In contrast, network simulators, such as ns-2 [BEF+00], SSFNet [CNO99], and GTNetS [Ril03], offer complete control over the testing environment. In simulation, the network topologies and workloads are easy to configure, and events can be readily generated to test the protocols under various circumstances. Nevertheless, simulation lacks realism. The protocols are not easily implemented in simulation without significant effort and, above all, simulation models must be subjected to rigorous verification and validation tests, without which they may not necessarily reflect the behavior of the real network.

None of the aforementioned approaches simultaneously achieves large-scale, flexible, and realistic environments for testing new network protocols and applications. The importance of the above mentioned qualities for a network testbed is summarized in the following paragraphs:

- It is common that researchers ignore the possible differences between small-scale and large-scale networks. They conduct small-scale experiments and then extrapolate the results and make inferences regarding the effect and behavior of large-scale networks. There are cases, however, where **large-scale experiments capable of showing some interesting behavior are indispensable**. For example, scalability is critical for peer-to-peer applications to examine their "network effect", where the behavior of one user can be positively affected when another user joins the network [RFI02, KS94]. Another example is the study of Internet worms, where a large-scale model is needed to show the dynamics of the worm propagation [LYPN02]. In these cases, experimenters often favor simulation as it can achieve large-scale topologies and the results can be easily reproduced [GR05, SHC$^+$04].

- Live and emulation testbeds offer the advantage of providing the experimenters with the execution environment for applications and protocols under study. For example, Emulab [WLS$^+$02] provides the user with a set of OS images that can be selected for a specific host in the experiment. However, the user has access to only the edge of the network, i.e., the end hosts. Consequently, it is difficult for a researcher to conduct in-network measurements to gain complete understanding of the system's complex behavior. Also, some studies require complex event scheduling under various settings and scenarios [GR05], which would be impossible in live and emulation networks. In order to address those needs, **a highly flexible testbed**

capable of dynamically changing the network model, and injecting events to change the course of the experiments is needed.

- Most testbeds are closed to inter-operation with other heterogeneous testbeds. However, inter-operability can bring several advantages, including: 1) it can enlarge the size of the experiments that a user can instantiate; 2) in case inter-operability with an end user is enabled, then online monitoring and steering of the network experiments is possible, i.e., insertion of events at run time; and 3) improved performance can result from inter-operability [YKKY08]. Consequently, **interoperable testbeds are needed for large-scale, and flexible experiments**.

In this sense, *a platform capable of performing large-scale experiments over diverse network scenarios and network conditions, maintaining accurate representation of the target applications and network transactions, and supporting high-throughput high-capacity network traffic is ostensibly missing and much needed by networking research community.*

1.2 Research Approach

The primary goal of this dissertation is to provide large-scale, flexible, and inter-operable testbeds for network experimentation. We believe that combining a simulation and an emulation system, and enabling inter-operability with the end user and with the network entities, yields large-scale and flexible testbeds:

1. The simulation system runs the full-scale network models with detailed network topology and protocols for a close representation of the global-scale network traffic behavior. To simulate potentially massive-scale network system, we can adopt parallel simulation and advanced traffic modeling techniques. Furthermore, a number

of nodes in the simulated network will be selected as emulated hosts to run unmodified software in a real operating environment with the designated operating system, network stack, and software tool chain.

2. Flexibility is readily provided by the simulation system. Network topologies, traffic, and protocols under test can be changed easily by providing a new network model. Furthermore, complex and choreographed network conditions can be configured to commence at any point in the experiment.

3. Allowing inter-operability with an end user can yield increased flexibility since online monitoring is possible and the user could insert events at run time.

1.3 Main Contributions

We design and implement three frameworks that attempt to meet the goal in different dimensions. We thoroughly validate our approaches and the performance of the implementations. We summarize the contributions of this dissertation as follows:

1. **Scalable Virtualized Evaluation Environment [ELL09, EL10a]**

   Small-scale experiments are insufficient for a comprehensive study of new application or protocols. We propose an evaluation testbed, called SVEET, on which real applications or protocols can be accurately evaluated under diverse network configurations and workloads in large-scale network settings.

   - We apply immersive network simulation techniques, which allow the simulated network to interact with real implementations of network applications, protocols, and services.
   - We extend the real-time simulation capabilities of our network simulator such that the simulation time can be made to advance either in real time or proportional to real time.

- We use machine virtualization technologies to provide the exact execution environment of the target system. To enable network experiments with large capacity links, we apply the technique of time dilation on the virtual machines and make the discrete-event simulator to synchronize with the time-dilated virtual machines accordingly.
- We port thirteen TCP congestion control algorithms to our simulator and thoroughly test their accuracy and performance against the ns-2 [BEF$^+$00] simulator and also against the Linux real stack.
- We test the accuracy of SVEET comparing its output to that of physical testbeds achieving great resemblance . Furthermore, we test the performance of SVEET under time-dilated conditions and achieve an increased scalability in terms of throughput.

2. **Integrating Real-Time Network Simulation and Emulation in Federated Environments [EL10b, VVEL11, VVEL12]**

   Although live and emulation testbeds provide the necessary realism in prototyping and testing new network designs, simulation is more effective for studying complex behaviors of large-scale systems, which are both difficult to handle using realistic settings and intractable to close-form mathematical and analytical solutions. Our system, PrimoGENI, enables real-time network simulation in a federated environment by extending an existing network simulator to become part of the GENI [GEN] federation to support large-scale experiments involving physical, simulated and emulated network entities.

   - We design the PrimoGENI system as a layered system in order to insert it into the GENI federation.

- We design and implement the communication infrastructure that allows a user of PrimoGENI to automatically instantiate network experiments in the GENI federation through an integrated development environment, called Slingshot.
- We validate the accuracy of our testbed, and determine its performance limitations.
- We achieve a flexible emulation infrastructure that allows both remote client machines and local cluster nodes running virtual machines to seamlessly interoperate with the simulated network. We show the results of our preliminary validation and performance studies to demonstrate the capabilities and limitations of our approach

3. **Leveraging Symbiotic Relationship between Simulation and Emulation for Scalable Network Experimentation [EL13]**

   In order to allow high-fidelity high-performance large-scale experiments, we allow inter-operability between a simulation and an emulation system at metadata level. We design and implement SymbioSim, a system that leverages the symbiotic, i.e., cooperative, relationship between simulation and emulation and, in doing so, is capable of studying real network applications and protocols under diverse large-scale network conditions.

   - We use the symbiotic approach for network testbed, which consists of two subsystems: a simulation system that provides flexible models for representing large-scale network behaviors, and an emulation system that reproduces the distributed environment for running real applications.
   - We propose a model-downscaling technique that can significantly reduce the complexity of large-scale simulation models, resulting in an efficient emula-

tion system for modulating the high-capacity network traffic between the real applications.

- We propose a queuing network model for the downscaled emulation system to accurately represent the network effects of the simulated traffic.
- We propose techniques for reducing the synchronization overhead between the simulation and emulation systems, both for updating the global traffic on the simulated network from real applications, and for calibrating the emulation system and modulating real application's traffic under simulated network conditions.

1.4 Outline of the Dissertation

Chapter 2 describes the background and related work. Section 2.1 presents the definition and classification of symbiotic systems and describes network emulation testbeds based on symbiosis. Section 2.2 provides a description of the principal methods that can be used to test new protocols and applications. Finally, current network emulation testbeds are described in section 2.3.

Chapter 3 presents our approach to reproduce traffic inside the simulation supported by our realistic TCP implementation. Section 3.2 describes the incorporation of realistic TCP congestion control algorithms into the simulator. Section 3.3 describes the design, implementation, and evaluation of our time-dilated network experimentation system.

Chapter 4 presents PrimoGENI, a system that enables real-time network simulation in a federated environment. Section 4.1 motivates the use of federated systems. Sections 4.2 and 4.3 introduce and describe the architecture of PrimoGENI. Section 4.4 provides the workflow for a network experiment. The processes necessary to support the PrimoGENI functions are described in section 4.5. Validation and performance experiments are detailed in section 4.6.

Chapter 5 describes our symbiotic approach for achieving large-scale testbeds. Section 5.1 describes the symbiotic relationship that we propose. Section 5.2 provides a more in-detail description of SymbioSim's architecture, main sub-systems, and core techniques that yield high accuracy. Our novel approach for emulating network paths is described in section 5.3. Our novel topology downscaling scheme is presented in section 5.4. Section 5.5 provides techniques for reducing SymbioSim's overhead. In section 5.6, the practical implementations of SymbioSim are described. SymbioSim validation studies are presented in section 5.7.

Finally, chapter 6 concludes the dissertation and provides direction for future work.

CHAPTER 2
LITERATURE REVIEW

In this chapter, we describe the most relevant literature. First, we begin describing symbiotic systems in general; we later provide examples of symbiotic systems used for network experimentation. Second, we provide a discussion about current network experimentation testbeds. Finally, we describe some existing network emulation techniques that later serve as baseline for our approaches.

2.1 Symbiotic Systems

2.1.1 Definition and Classification

In Biology, *symbiosis* is defined as the association between two or more different organisms. Within that context, Lunceford at al. [FLPU02] define a symbiotic system as "*. . . one that interacts with the physical system in a mutually beneficial way. It is highly adaptive, in that the simulation system not only performs "what-if" experiments that are used to control the physical system, but also accepts and responds to data from the physical system. The physical system benefits from the optimized performance that is obtained from the analysis of simulation experiments. The simulation system benefits from the continuous supply of the latest data and the automatic validation of its simulation outputs. Such a definition implies continuous execution of the simulation and real time interaction with the physical system.*".

Aydt et al. [ATCL08] state that the above definition of symbiotic systems is narrow since it only refers to *mutualism*. In mutualism, all entities involved in the symbiosis benefit from each other, which is is implied in the previous definition. They defined symbiotic a simulation system as "*. . . A close association between a simulation system and a physical system, which is beneficial to at least one of them.*".

From the above definitions, it is clear how different symbiotic simulation systems are from classical discrete-event simulations. In classical simulations, there is no feedback control loop so the user has to input all parameters, trusting his intuition, and then run the simulation from beginning to end with no changes being possible to the network model at runtime. Two different types of symbiotic systems were identified in [ATCL08], which we describe in the following paragraphs.

Closed-loop symbiotic simulation systems are those where a control feedback is created to the physical system which affects it. In these systems, the simulator carries out what-if experiments in particular scenarios and performs decisions based on the results obtained. This subsystem is called *symbiotic simulation decision support system* (SSDSS). An external subsystem called *symbiotic simulation control system* (SSCS) is capable of implementing the decisions made by the SSDSS using actuators. The schematic view of these kind of systems is shown in Fig. 2.1.

In *open-loop symbiotic simulation systems*, no feedback is created to the physical system. Three types of systems can be categorized as belonging to this category: *symbiotic simulation forecasting system* (SSFS), *symbiotic simulation model validation system* (SSMVS), and *symbiotic simulation anomaly detection system* (SSADS). SSFS sytems predict future states of the physical system but do not interpret results in order to draw conclusions. SSMVS systems aim to determine a model which describes the behavior of the physical system subject to measurement. Finally, SSADS systems are used to detect anomalies either in the underlying simulation model or the physical system. These systems are depicted in Fig.2.2.

2.1.2 Existing Symbiotic Approaches for Computer Network Emulation

An earlier system, ROSENET [Gu07], implemented a close cooperation between an emulation system and a simulator. Gu et al. aimed to achieve *scalability*, *accuracy*, and

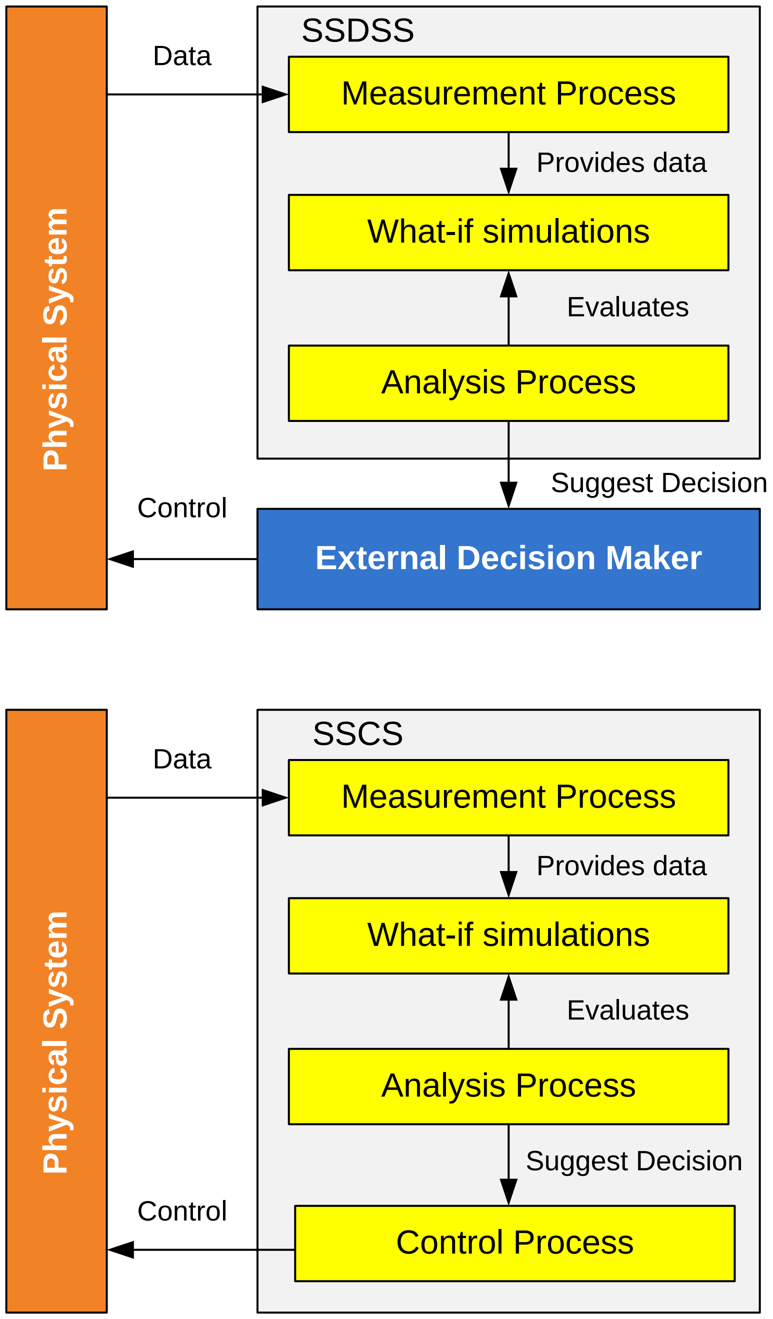


Figure 2.1: Closed-loop symbiotic simulation systems

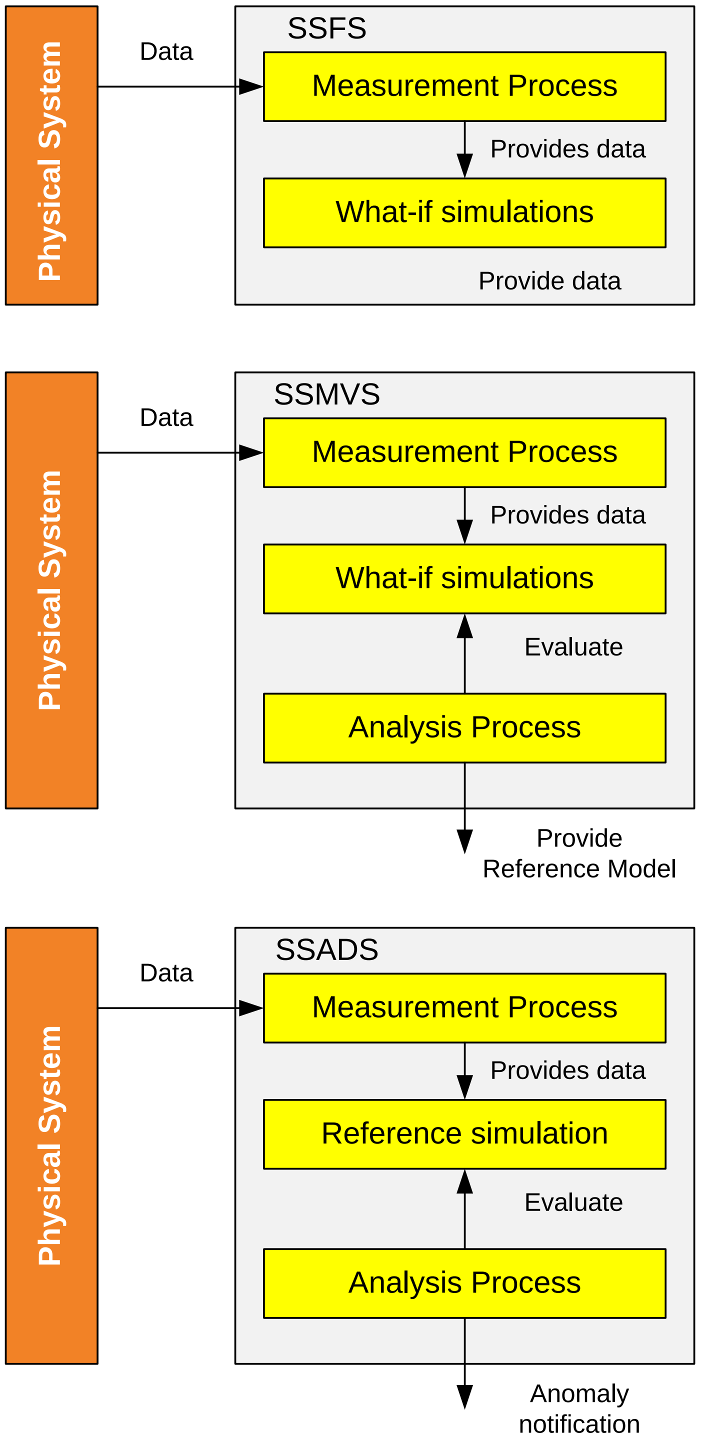


Figure 2.2: Open-loop symbiotic simulation systems

timeliness for simulation and emulation of computer systems by overcoming the disadvantages they identified in existing approaches for both clustered emulation systems and software emulation systems. For clustered-based emulation systems, e.g., Emulab, they identified the following limitations (we only list the most relevant):

1. *Scalability*. Cluster-based emulators are limited by the number of physical nodes in the cluster.
2. *Limited physical bandwidth inside the cluster*. This limits the total number of nodes supported in a simulation.
3. *Background traffic generation*. Systems either generate synthetic traffic that is not responsive or imitate the effects of cross-traffic on the link.
4. *Accuracy*. Simplified network models are often used in emulation clusters trading accuracy for increased scalability.

For software-based emulators, e.g. PRIME [Liu], ns-2 [BEF$^+$00], the following limitations were also identified (again, we only list the most relevant):

1. *Scalability*. The network topology of the parallel simulation based emulators is small.
2. *Timeliness*. To achieve timeliness, many simulators have to keep the network model small so that is executes faster than real time.
3. *Accessibility*. The infrastructure needed to run parallel simulations may not be locally available.

In ROSENET, close cooperation exists between a simulator and an emulation system. The simulator can run on a different physical location than the emulator and Gu et al. claim they can still provide accuracy, scale, and timeliness. The simulator continuously

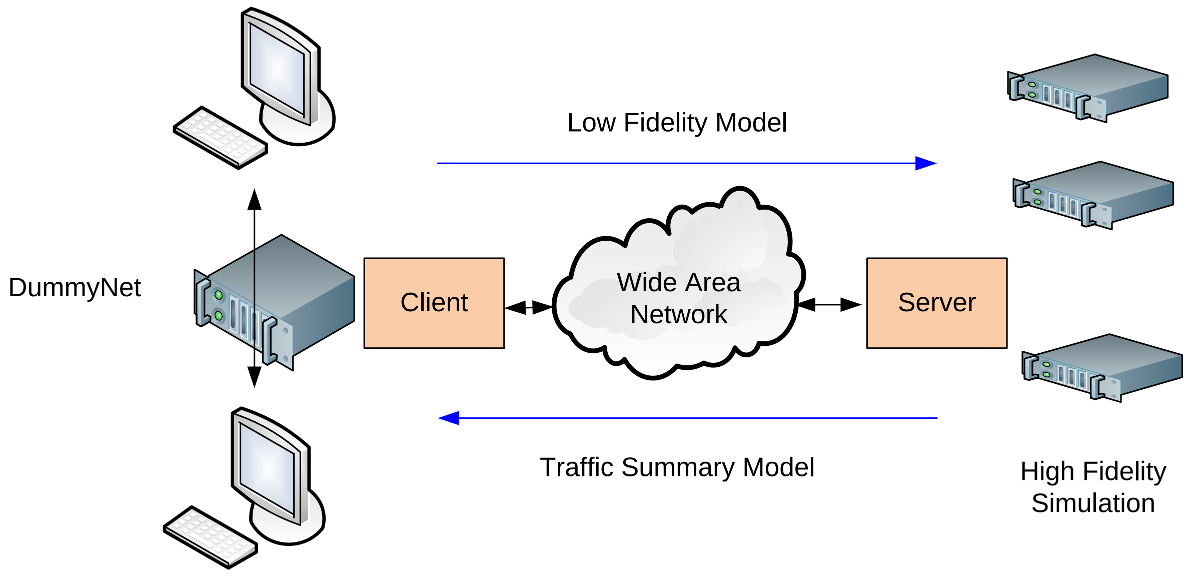


Figure 2.3: The ROSENET Approach

updates the model provided to the emulator, i.e., delay, jitter, and packet loss, so that the environment where real applications run is close to that expected according to the network model provided. In a symbiotic context, the emulated system benefits from the *Low Fidelity Model* provided by the simulator that calibrates the emulated system, realized by the use of Dummynet [Riz97]. At the same time, the simulator benefits from *Traffic Summary Model* provided by the emulated system, which is fed into the simulation. The ROSENET approach is shown in Fig.2.3.

ROSENET is based on the findings that the Internet traffic exhibits a constancy in timescales of minutes [ZD01]. This allows that a network model characterize the traffic dynamics between a time interval without loosing too much accuracy. The low fidelity model is sent at the end of each internal if it is detected that it is statiscally different from the previous one.

Gu et al. claim ROSENET achieves *accessibility*, since real world applications can run locally while the simulator can run on remote high performance facilities. Allowing users to run arbitrary network topologies and traffic loads is that ROSENET provides *flexibility*. *Scalability*, *accuracy*, and timeliness are claimed to be achieved through the use of a high-fidelity simulator.

2.2 Network Experimentation Testbeds

2.2.1 Simulation Testbeds

Three principal methods are used to test the new applications and protocols: simulation, live experiments, and emulation. Federated approaches such as GENI [GEN], put together all these technologies in a single experimentation platform.

Discrete-event network simulators, such as ns-2 [BEF$^+$00], SSFNet [CNO99], and GTNetS [Ril03], offer complete control over the testing environment. In simulation, the network models: topology, traffic and protocols are easy to configure, and events can be readily generated to test the protocols under various circumstances.

2.2.2 Emulation Testbeds

Emulation of computer networks can be considered as a middle ground approach between simulation and live networks(described later). Emulation subjects real applications, and protocols stacks to controllable and synthetic network topologies. In this section we describe the most salient emulation approaches.

ModelNet [VYW$^+$02] is an emulation facility where unmodified applications run on *edge nodes*. *Core nodes* emulate the target topology and route applications' traffic. Unlike other emulation testbeds, ModelNet targets the emulation of large-scale topologies. To that end, core nodes route packets through queues and links (pipes), which never copy data since packets are passed by reference. Shortest-path routes are pre-computed between each virtual edge node (VN). On entering the core network, ModelNet computes the set of routes that each packet has to traverse according to the source and destination of the packet. To achieve even larger experiments, ModelNet trades increased scalability for accuracy by 1) removing all interior nodes in the network, i.e., collapsing each path into a single pipe (*Distillation*), 2) increasing VN multiplexing into physical nodes and managing concurrency efficiently, 3) directly inserting dummy packets into the emulated topology instead of running edge applications to generate background traffic.

Often cataloged as an emulation approach, Emulab [WLS+02], an evolution of Netbed[WGN+02], is a software system composed of local (emulated) and geographically distributed nodes (wan) nodes. Virtual network topologies, specified with an ns-2 [BEF+00] script, are realized automatically using a subset of available nodes and are largely independent of their physical realization. The mechanism is simple: local nodes act as either hosts, routers, or wide-area links (using Dummynet) interconnected by switches which provide isolation and allow realization of arbitrary topologies. Simulated nodes, links, and traffic interact with real applications using ns-2's emulation facility called nse.

2.2.3 Live Experimentation

Different from emulation testbeds, whose main objective is to provide a controlled and flexible environment for network experimentation, live networks emerged later as a new class of environments to evaluate emerging applications. A salient example of these kind of testbeds is PlanetLab [PACR02].

PlanetLab aims to promote the introduction of disruptive technologies into the Internet through the use of *overlay networks*, i.e., networks built on top of another network. Its design principles are the following: 1) *slice-ability*: services should be able to run continuously in a slice; 2) *distributed control over resources*; 3) *unbundled management*; and 4) *application-centric interfaces*. In order to achieve slice-ability, each node has a *virtual machine monitor* (VMM) to allocate a slice of resources to an overlay network.

Distributed control means that even though a central authority provides credentials to researchers to create slices, each node may deny or accept such request based on local policy. Largely independent sub-services compose the testbed management. Finally, overlays support an existing and widely adopted programming interface.

2.2.4 Federated Approaches

GENI [GEN] is a set of network research infrastructure, which aims to be presented as a single collaborative and exploratory platform for implementing and testing new network designs and technologies. GENI is built on four premises: i) *programmability*: researchers shall be able to upload and execute software on GENI nodes deep inside the network hierarchy; ii) *resource sharing*: GENI shall be shared among multiple users and their experiments, possibly through virtualization; iii) *federation*: GENI shall support interoperability among various types of resources provided by different organizations; and iv) *slice-based experimentation*: each user shall be provided with a *slice*, i.e., a subset of resources of the GENI infrastructure, and network experiments shall be conducted independently on reserved resources within slices. The current GENI design consists of three main types of entities: *clearinghouses*, *aggregates*, and *principals*. A clearinghouse is the central location for management of GENI resources for experimenters and administrators. Specifically, it provides registry services for principals, slices and aggregates, and authentication services for accessing the resources. An aggregate represents a group of components encapsulating the GENI sharable resources (including computation, communication, measurement, and storage). When an experimenter from a research organization (i.e., a principal) decides to conduct a GENI experiment, she will negotiate with the clearinghouse and the associated aggregate managers through an elaborate resource discovery and allocation process. In response to the experimenter's request, each participating aggregate will provide a set of requested resources, which are called *slivers*. Jointly, these slivers form a *slice*, which is the environment where the experimenter conducts experiments, with the help of GENI experiment support services.

Conceived as a GENI Control Framework project, ProtoGENI [PRO] aims to integrate network systems that provide GENI key functionality. Although it initially relied heavily on Emulab infrastructure, it provides its own API to instantiate experiments (slices) within it. Our project, PrimoGENI [VVEL12] [VVEL11] [EL10b] uses the ProtoGENI control framework to manage, control and access the underlying resources.

2.3 Network Emulation Techniques

2.3.1 Topology Downscaling

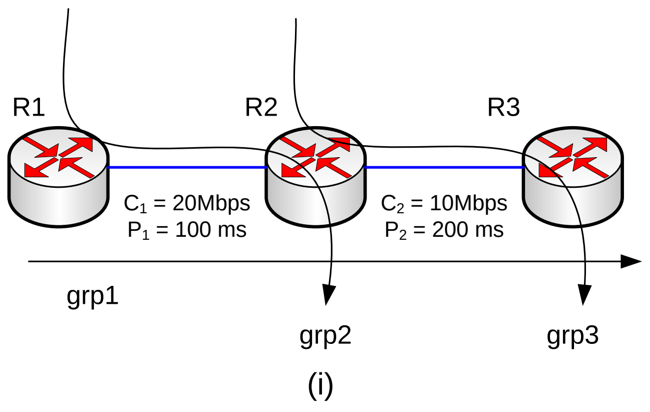
Topological downscaling is based in the intuition that only *bottleneck links* are the ones that introduce sizeable queuing delay and packet loss, while uncongested links may in some sense be transparent to the packets that traverse them. Then, if that is true, it may be possible to construct a smaller replica of the original including only congested links. In this sense, it may be possible that some flows that share some links in the original do not do so in the replica. This phenomena poses the question whether the replica captures all the correlations among flows. However, earlier studies concluded that only the congested links along the path of each flow introduce dependencies among flows and sizeable queuing delays [BTI$^+$02] [FML$^+$03] [PMF$^+$02] [FTD03]. Furthermore, in recent years it has been shown that links with capacities large enough to carry many flows without getting congested, are in a sense, transparent to the flows that pass across them [Eun03].

Supported by the findings listed above, is that Papadopoulos [PPG06] et al. proposed a performance preserving topological downscaling approach. In that work, they define a bottleneck link as that which changes the arrival process of the packets that traverse them. On the contrary, an uncongested link introduces no packet loss and its average queuing delay is much smaller than the end-to-end queuing delay, mainly due to propagation delay. It is the objective of the authors that this downscaled topology enables researchers to study the behavior of new applications in a smaller replica that preserves most important sizeable characteristics of the original.

The two methods proposed by Papadopoulos et al. operate on any given topology with **known traffic**. The first one, DSCALEd, accounts for the missing links by adding fixed delays to packets, while the second method, DSCALEs does so by adjusting the capacities and propagation delays of the replica. In the following paragraphs we summarize both proposed methods.

DSCALEd is the method proposed to downscale a network topology by retaining only the congested links. In this way, only those flows that traverse congested links are preserved in the replica while the others are ignored. Fixed delays are added to all flows in the replica, if necessary, so that the end-to-end delay for all of them is the same as in the original network. To illustrate their method, let's use Fig. 2.4. In Fig. 2.4(i), the link $R_1 - R_2$ is the uncongested link and as such it is not preserved in the downscaled topology shown in Fig. 2.4(ii). Consequently, the set of flows contained in grp2 is ignored in the replica. However, by ignoring the link $R_1 - R_2$, the end-to-end delay for flows in grp1 is being altered in the replica so a fixed delay is added to them equal to the propagation delay (P_1) plus the transmission delay ($packetSize(grp1)/C_1$).

DSCALEs is method which is aims to create a slower replica of the original one. Again, only the congested links are preserved together with the flows that traverse them. This method is based on a downscaling law: *If network flows are sampled with some factor and fed into a network replica whose link speeds are multiplied by and propagation delays by , then performance extrapolation is possible*. However, it is based on two assumptions: 1) Packet arrivals must be Poisson, and 2) End-to-end queuing delays are small in comparison to total end-to-end delays.



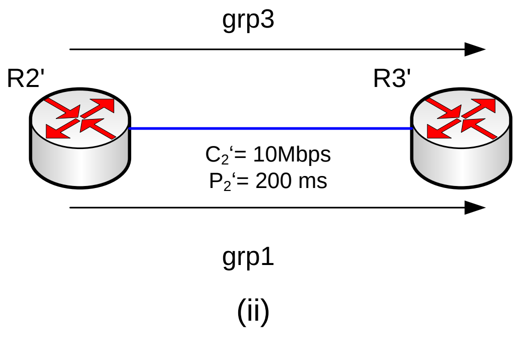


Figure 2.4: DSCALEd operation in a simple topology

2.3.2 Emulation of Internet Paths

Network emulators subject real applications to previously configured network conditions. They work by forwarding packets from applications under test through a set of queues that approximate the behavior of router queues. Salient network emulators include Dummynet [Riz97], ModelNet [VYW$^+$02], NIST Net [CS03], and Emulab[WLS$^+$02]. These emulators focus on emulating a *link* with all detail. However, most applications are concerned with the end-to-end characteristics of the *path* and not with faithfully emulating every link. These fact poses a need for a *path emulator*.

An approach to construct a path emulator is to have sequence of the above listed emulators so that each instance will emulate a link within a path. However, if the objective is to recreate Internet paths from the end-points, i.e., the hosts, obtaining the complete topology with bandwidths, delays, and queue sizes is very difficult if not impossible. Another alternative is to approximate the path as a single link with a given packet loss and delay. A salient survey [VV08], points out that many papers published in top venues [BPS06], [BBSW05], [GSS06], [PAK07], [TKAP06], use the approach of approximating a path with a single link. However, as pointed out in [SDRL09], this fails even the simplest tests. This is due to the fact that emulators model *link capacity* and not *availability of bandwidth*. Furthermore, this approach does not model interaction between flows, e.g., shared bottlenecks, and reactivity of background flows.

Sanaga et al. in [SDRL09], identified four basic principles for emulating Internet paths:

1. *Model capacity and available bandwidth separately*. It is not sufficient that emulators model *capacity*, also the available bandwidth (that will ultimately decide the data transfer rate) must be modeled and both must be modeled separately.

2. *Pick appropriate queue sizes*. From the end hosts, they propose mechanism to find the upper and lower bounds for queue sizes.

3. *Model the reactivity of background flows*. The reactivity of background flows is modeled as a function of the offered load, i.e., the foreground traffic.

4. *Model shared bottlenecks*. When modeling a set of paths from the end-hosts, it may be the case that these paths share bottleneck links. This effect must be modeled since it will affect the properties of the network seen by end hosts.

With the aforementioned four principles, Sanaga et al. [SDRL09] have five input parameters to their model that aims to model Internet paths:

1. *Base RTT*. This is propagation time that a packet would experiment from the source to the destination and back (approximated by the minimum observed RTT). The total RTT is given by the the base RTT plus the maximum delay that can be introduced by the queues in the bottleneck links:

   $$RTT_{max} = RTT_{base} + \frac{q_f}{C_f} + \frac{q_r}{C_r} \tag{2.1}$$

   In Eq. 2.1, q_f and q_r are the queue size in the forward and reverse direction respectively. C_f and C_r are the capacities (bandwidth) in the forward and reverse direction.

2. *Capacity and available bandwidth (ABW)*. From the end-hosts, the available bandwidth is visible and not capacity. However, if we use the available bandwidth instead of the capacity, the estimated total RTT may be much higher than the real one. Consequently they propose to model both parameters separately. In this way, if it is the objective to model a path with $C \geq ABW$, there must be background flows that partially fill the bottleneck link so that only the ABW is available to foreground

flows. They propose not to model background flows in detail, but instead focus on its *rate* and *reactivity*.

3. *Queue size*. Again, from the endpoints it is very difficult to get the queue size at the bottleneck links. Instead, they propose to compute lower and upper bounds for the queue sizes at the bottleneck links. They model the queue size in terms of *size* and *time*, i.e., the maximum queueing delay they may introduce. The lower bound is given under the assumption that a router queue should be able to enable that all flows that traverse it achieve the ABW[Kes04]. In consequence, the lower bound for the queues is given by:

$$q > \sum_{f \varepsilon F} min(w_f, w_{max}) \tag{2.2}$$

In Eq. 2.2, w_f is the window size of a TCP flow f, roughly equal to the bandwidth-delay product (BDP). The maximum RTT that can be tolerated by a TCP flow before it is limited by itself, i.e., its congestion window, is given by [PFTK00]:

$$RTT_{max} = \frac{w_{max}}{ABW} \tag{2.3}$$

In steady state, Reno flows tend to reach a state in which the bottleneck queues are nearly full [PFTK00]. In consequence, those flows will experience a RTT near that given by Eq. 2.1 and for the emulation to be correct the RTT must be less than or equal to the maximum. Also setting the capacities to be the same in both directions we get:

$$q_f + q_r <= C * (\frac{w_{max}}{ABW_f} - RTT_{base}) \tag{2.4}$$

After all this considerations, the bottleneck is modeled with a queue that is within the lower and the upper bound which drains at a fixed rate in presence of constant bit-rate cross-traffic. In this way, the draining rate of the queue is the capacity and the difference between the capacity and the injection rate of cross traffic is the available bandwidth. Also, shared bottlenecks are modeled by first detecting them [KKS$^+$04], to then make that paths that share bottlenecks share the same bandwidth queue in the emulated topology. Finally, the reactivity of background traffic is modeled by finding analytical expressions for the available bandwidth (ABW) as a function of the offered load.

2.3.3 Network Traffic Generation

Traffic generation is important in several scenarios, including: queue management studies [LAJS03], capacity planning [MTS$^+$02], bandwidth estimation tools, network emulation [WLS$^+$02][WGN$^+$02][VYW$^+$02], and network simulation [BEF$^+$00].

A salient example of a traffic generator is Harpoon [SB04]. Harpoon is a traffic generator capable of recreating IP traffic flows, i.e., an IP/port pair using a specific transport protocol (TCP/UDP). Harpoon models background traffic starting with measured distributions of flow behavior on a target link. Harpoon is designed to match distributions from the underlying trace at a coarse granularity (minutes) and thus does not either extract or playback network characteristics. Harpoon has two components: *client threads* that make file transfer requests and *server threads* that transfer the requested files using either TCP or UDP.

Swing [VV09] is a traffic generator that aims at generating realistic traces that accurately represent the following charactestics of the original traffic: 1) packet interarrival rate and burstiness, 2) packet size distribution, 3) flow characteristics, and 4) destination IP address and port distributions. Venkatesh [VV09] et al. defined an structural model to capture interactions across multiple layers of the protocol stack. They divided the parameter space in four categories: 1) users, 2) sessions (network activity to carry out some high level task), 3) connections, and 4) network characteristics. Compared to Harpoon [SB04], Swing considers the characteristics of individual applications, thus enabling them to vary the mix of, for example, HTTP versus P2P traffic; thus providing more flexibility.

CHAPTER 3

ON ENABLING TRAFFIC REPRODUCTION IN SIMULATION

In this chapter, we present our performance evaluation testbed, on which real applications can be accurately evaluated under diverse network configurations and workloads in large-scale network settings. This testbed combines real-time simulation, emulation, machine and time virtualization techniques. We validate the testbed via extensive experiments and assess its capabilities through case studies involving real web services.

3.1 PRIME's OpenVPN-Based Emulation Infrastructure

The PRIME's [Liu] OpenVPN [OPEa]-based emulation infrastructure forwards the packets to the PRIME network simulator. From the simulator's point view, these packets seem to have been generated directly by the corresponding end-hosts on the virtual network, i.e., the topology being exercised. This emulation infrastructure is built upon OpenVPN, which has been customized to tunnel traffic between the virtual machines and the network simulator [LMVH07a].

Fig. 3.1 illustrates an example of the emulation infrastructure connecting the PRIME network simulator with two virtual machines—one running a web server sending HTTP traffic to the other virtual machine running a web client. To set up the experiment, the two virtual machines need to first establish separate VPN connections with a designated VPN server, which we call the emulation gateway. OpenVPN is an open-source VPN solution that uses TUN/TAP devices. The OpenVPN client on each virtual machine creates a virtual network interface (i.e., the `tun0` device), which is assigned with the same IP address as that of the corresponding end-host on the virtual network. The forwarding table of each virtual machine is automatically configured to forward traffic destined to the virtual network IP space via the VPN connection. In this case, data generated by the web server will be sent down to `tun0` via the TCP/IP stack and then given to the OpenVPN

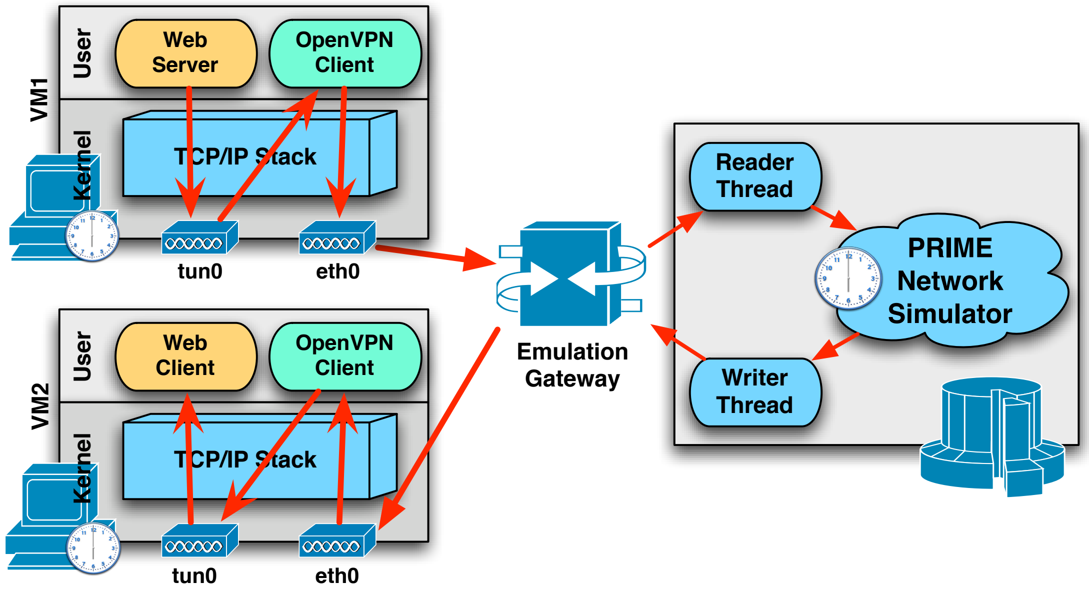


Figure 3.1: PRIME emulation infrastructure.

client. The OpenVPN client uses IP over UDP to transport packets to the OpenVPN server at the emulation gateway. Upon receiving the packets, the emulation gateway forwards the packets via a dedicated TCP connection to the simulator.

The reader thread at the simulator side receives the packets from the emulation gateway and then generates simulation events representing the packets sent from the corresponding end-hosts on the virtual network. PRIME simulates the network transactions as the packets being forwarded on the virtual network. Upon reaching their destinations, the packets are exported from the simulator and the writer thread sends the packets to the emulation gateway via another dedicated TCP connection. The OpenVPN server at the emulation gateway subsequently forwards the packets to the corresponding emulated hosts via the VPN connections. The OpenVPN client at the target virtual machine receives the packets and then writes them out to `tun0`. The web client receives these packets as if they arrived directly from a physical network.

3.2 Incorporating Linux TCP Algorithms

Simulation does not produce realistic results if the protocols used are partially implemented and thus do not behave as real ones. With the purpose of increasing our simulator's ability to produce realistic results, we port real Linux TCP algorithms into our simulator, PRIME [Liu]. We perform this task in two steps: in the first one we port the ns-2[BEF$^+$00] implementation to PRIME; in the second step, we preserve the original algorithms but augment the code in order to enable seamless communication between real and simulated hosts. We detail the implementation of TCP in the following paragraphs.

We follow the same design principle as that detailed in [WC06] for porting Linux TCP variants to PRIME. In fact, we reuse as many data structures as possible from the Linux TCP port to ns-2. The code structure is shown in Fig. 3.2. In PRIME, TCP protocols on each simulated host are organized as a list of protocol sessions, each represented as a `ProtocolSession` object. We created a protocol session that we called `LinuxTcpMaster`, to manage all active Linux TCP connections, and another protocol session called `LinuxTcpSimpleSocket`, to support a simple interface for applications to send or receive data over Linux TCP. Consequently, both are derived from the `ProtocolSession` class. A TCP connection is structured in the same fashion as in ns-2: we use `LinuxTcpAgent` to represent the TCP sender-side logic and `SinkTcpAgent` to represent the receiver-side logic. In this way, we achieve maximum reuse of the existing code from the Linux TCP implementation in ns-2. The congestion control mechanisms of the TCP variants are transplanted directly from the Linux TCP implementation. `ns-linux-util` is a set of facilities created by the ns-2 port as an interface between the Linux TCP functions and the ns-2 simulator.

We validate our implementation by performing a set of tests which are described in the following subsections.

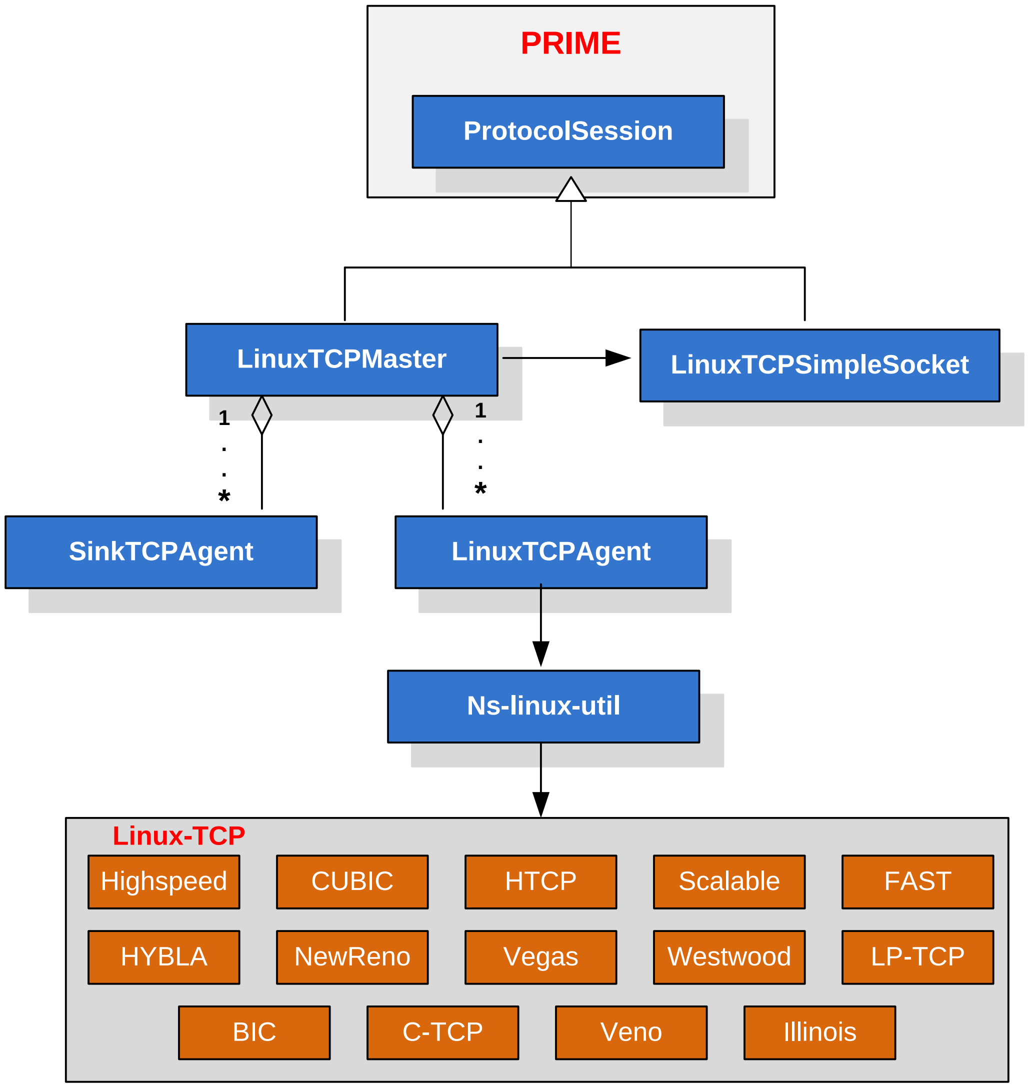


Figure 3.2: Linux TCP class structure

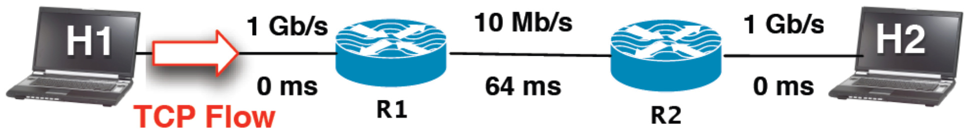


Figure 3.3: Simple topology for baseline validation

3.2.1 TCP Congestion Window Trajectories

Our first set of experiments aims to provide the baseline comparison between pure simulation results produced by PRIME, ns-2 (whose TCP implementation had already been benchmarked using a Dummynet [Riz97] testbed [WC06]), and those obtained from running PRIME with emulated hosts, i.e., hosts running on their real environment. We use a simple network with two end-hosts connected by two routers, as shown in Fig. 3.3, which is similar to the one used in a previous study [WC06]. The connection between the two routers forms a bottleneck link, configured with 10 Mb/s bandwidth and 64 ms delay. The network interfaces at both ends of the bottleneck link each has a drop-tail queue with a buffer size of around 66 KB (about 50 packets). The links connecting the routers and the end-hosts each has 1 Gb/s bandwidth and zero delay, respectively.

We conducted three tests for each TCP variant: the first with the ns-2 simulator, the second with the PRIME simulator (with emulation disabled), and the third with PRIME with emulation enabled (we call this system SVEET). Emulation was conducted on the same platform as was used in the experiments in the previous section. Both end-hosts were emulated in separate Xen [BDF^{+}03] domains (i.e., virtual machines) located on the same physical machine. Both PRIME and the simulation gateway were run on another machine, and the two machines are connected through a gigabit switch.

During each test, we directed one TCP flow from one end-host (H_1 to the other H_2) and measured the changes in the TCP congestion window size over time at the sender

(H_1). For both ns-2 and PRIME, we used a script to analyze the trace output from the simulators; for PRIME with emulation enabled, we used Web100 [WEB] to collect the congestion window size at the virtual machines. Fig. 3.4 shows the results. The results from ns-2 and PRIME match well, with only small differences that can be attributed to the differences between the two simulators in the calculation of transmission delays as packets traversing the routers. PRIME running with emulated hosts produced results similar to those from the simulators; the differences are typically more pronounced at the beginning of the data transfer; which resulted in a slight phase shift in the congestion window trajectory onward. One of the reasons for this small difference is the initial RTTs measured in each environment, which is slightly different on emulation mainly due to the emulation and virtualization infrastructure. However small, this difference causes VEGAS to behave differently on each trial, as its congestion avoidance mechanism relies very much on the initial RTTs. In Fig. 3.4, we show 10 separate congestion window trajectories for VEGAS predicted by emulation and compare them against the results from PRIME and ns-2. In any case, the results from these tests show conclusively that PRIME running on emulated environments can accurately represent the behavior of the TCP variants.

3.2.2 Throughput Versus Packet Loss

We use a second set of experiments to study the accuracy of our implementation in terms of delivered throughput. In these experiments, we reuse the same network scenario as in the previous set of experiments. The bandwidth of the bottleneck link is set to 10 Mbps. Random packet drops according to a specified probability are applied. We variy the packet drop probability between 10^{-6} and 10^{-1}, and measured the aggregate throughput of downloading a large data file over TCP for 100 seconds.

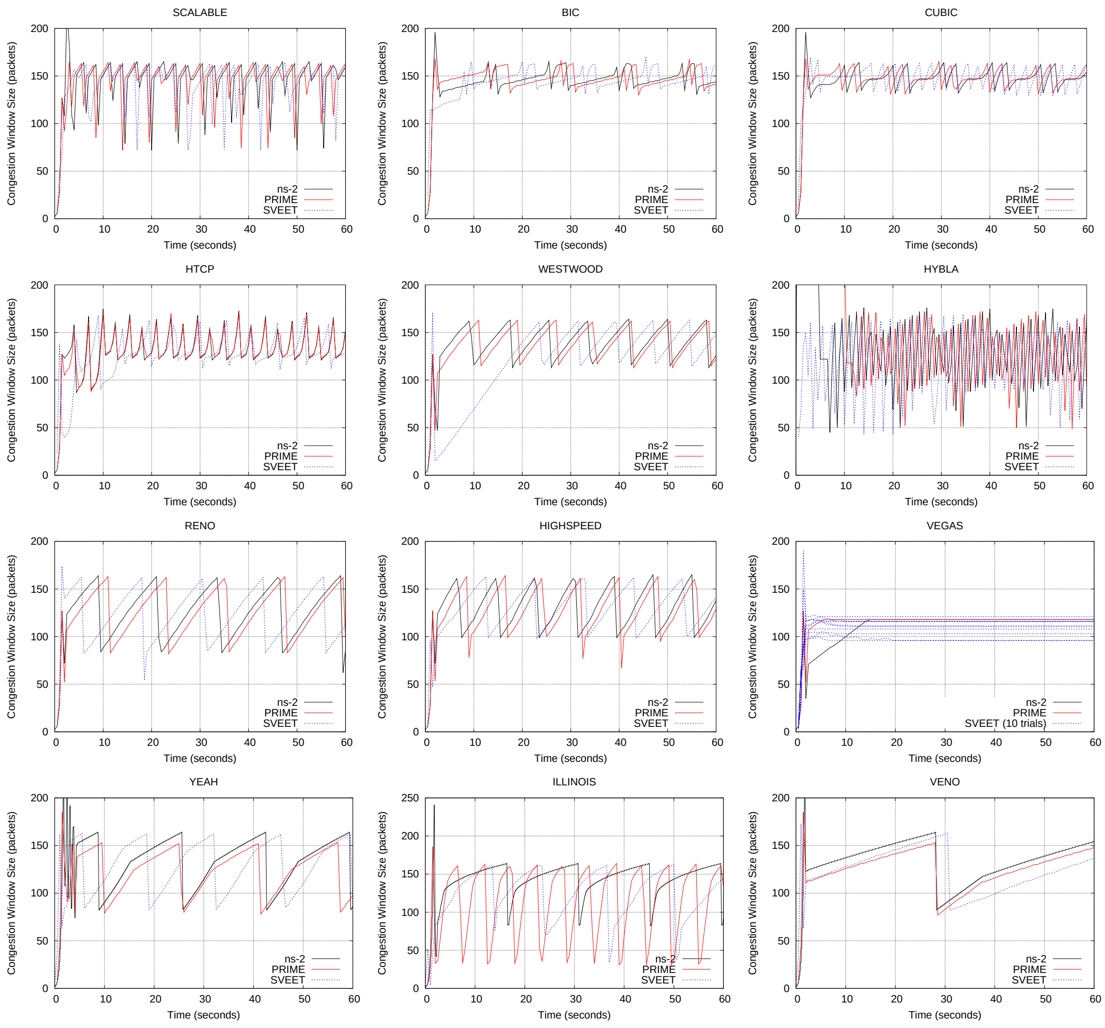


Figure 3.4: Congestion window trajectories of Linux TCP variants

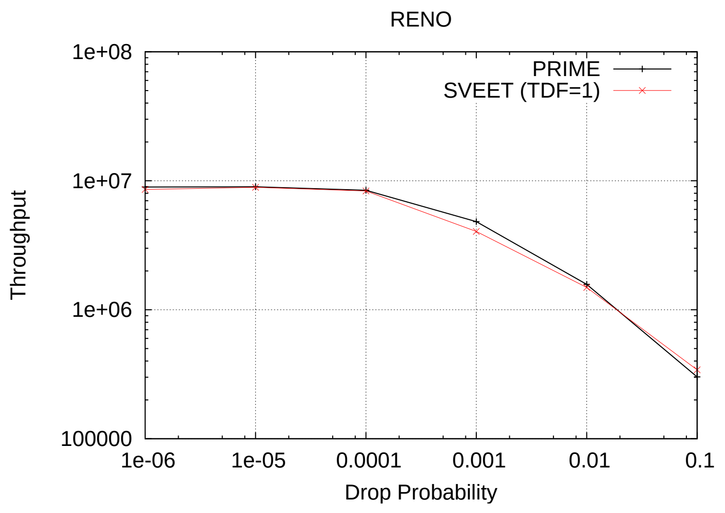
Fig. 3.5 shows the results both from PRIME, without emulation enabled, and PRIME with emulation for three TCP variants: TCP Reno, CUBIC, and Scalable TCP, which were selected for their drastically different congestion control behaviors. In all cases, very similar results were produced by both platforms.

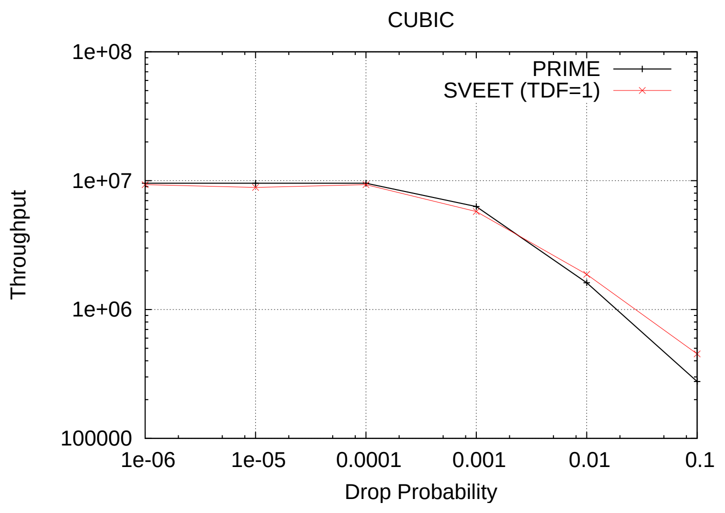
3.2.3 Recreating TCP Formats and Packet Exchanges

The results obtained so far show conclusively that our implementation of TCP behaves as the real ones available in Linux. However, this still fails to achieve seamless communication between emulated and simulated hosts because it was designed for simulation only and many of its features do not resemble that of real TCP:

- The packet header of TCP packets exchanged between simulated hosts using our implementation is much simplified and does not include all fields, e.g., *options*. Even worse, some simulator-specific fields are included in the packet.
- Sequence and acknowledgment numbers are packet-wise and not byte-wise.
- The sender and receiver windows are just integer variables but in a real implementation these are complex data structures.
- No state machine is included in the first implementation and therefore no handshake is being performed between end-hosts. The sender just starts to send packets to the intended receiver assuming that the connection has already beeen established and the receiver acknowledges the packets as soon as it gets them. Also, no host can stop or cancel a TCP session in presence of specific circumstances.
- No real payload is carried between end-hosts. A fake packet size number is included in the header to cheat the simulator and emulate the intended size.

In order to fix this issue, we augment the first implementation source code significantly [ELL09]. The class diagram is shown in 3.6.





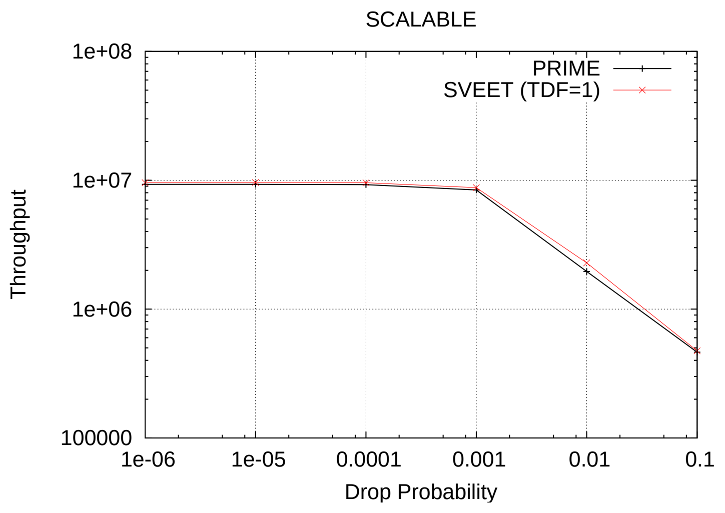


Figure 3.5: Throughput achieved under random packet loss

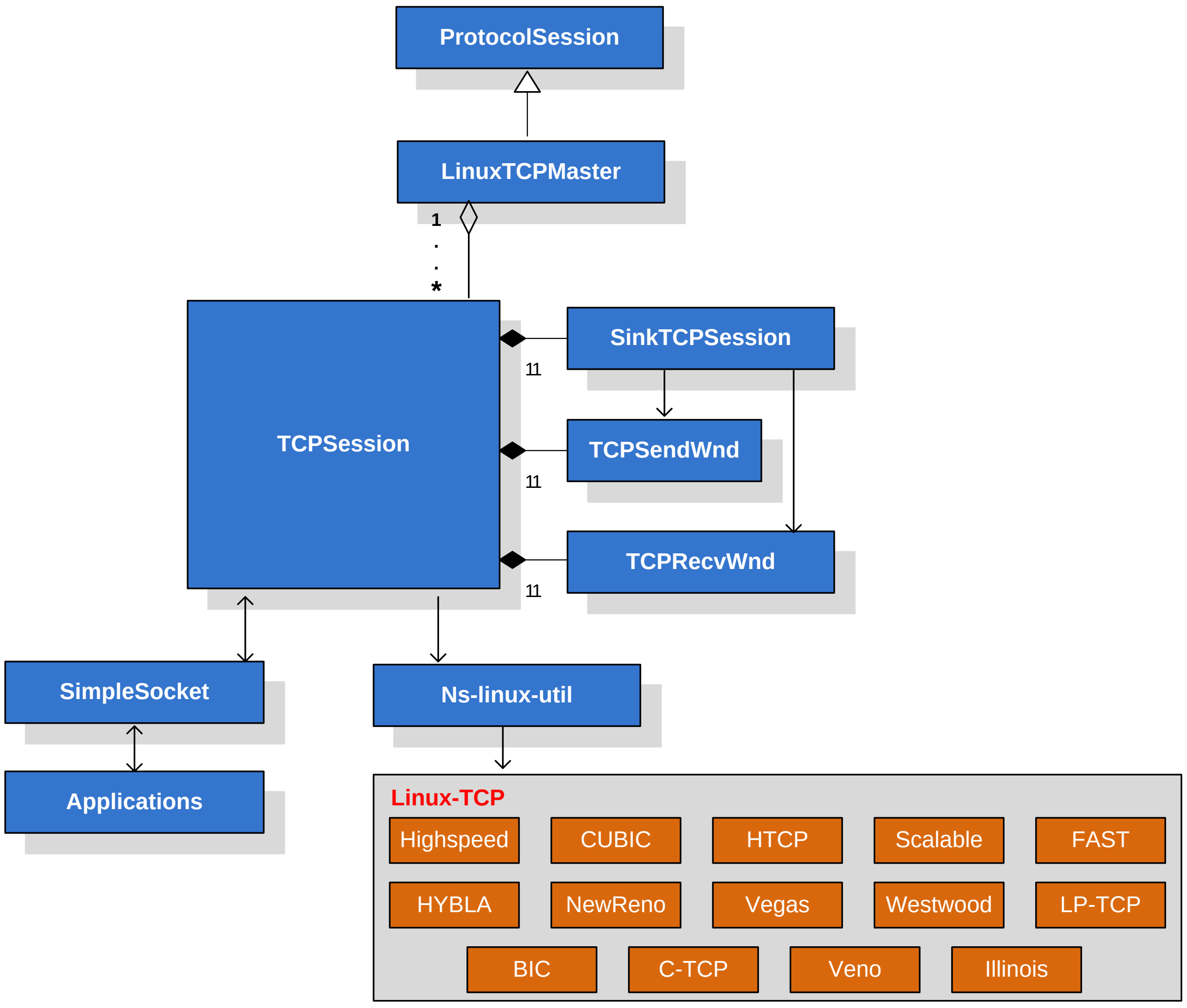


Figure 3.6: Class diagram for augmented TCP

In this new implementation, we performed some fundamental changes:

- To enable bi-directional communication, the old receiver part (that takes care of acknowledging the data packets) is now merged with the sender one. In this way, there is no distinction between sender and receiver.
- TCP's state machine was fully implemented.
- *TCPSndWnd* and *TCPSndWnd* are complex data structures that buffer real data.

We validate our extension work in [VVEL11].

3.3 SVEET

In this section, we present SVEET [ELL09], a performance evaluation testbed based on PRIME's OpenVPN emulation infrastructure, where real implementations of applications and protocols can be accurately evaluated under diverse network configurations and workloads from real applications in large-scale network settings enabled by the use of time dilation. SVEET uses our Linux TCP implementation and is used to conduct validation and testing.

3.3.1 Design Guidelines

We conceived SVEET to meet with the following characteristics:

- It must be able to generate reproducible results. Reproducibility is essential for protocol development; the users should be able to use the testbed and follow a set of procedures for regression testing, documenting, and benchmarking.
- It must be able to accommodate a diverse set of networking scenarios, ranging from small-scale topologies to large-scale configurations. Not only should the researcher

be able to use the testbed for inspecting the details of the protocol behavior in small, tightly controlled, choreographic conditions, but also be able to perform studies to assess large-scale impact, e.g., how much a TCP variant can affect and be affected by other network traffic in realistic large-scale network settings.

- It must be able to incorporate existing protocol implementations in real systems rather than develop its own version simply for testing purposes. The protocol development process is complicated and error-prone. Furthermore, maintaining a separate code base entails costly procedures for verification and validation.

3.3.2 Architecture

Fig. 3.7 provides a schematic view of the SVEET architecture. Distributed applications are executed directly on end-hosts configured as separate virtual machines (e.g., VM1 and VM2) with their own network stacks. We call these end-hosts emulated hosts. Traffic generated by the applications on these emulated hosts (emulated traffic) is captured by the virtual network interfaces (NICs), which forward the packets to the PRIME network simulator via the emulation infrastructure provided by PRIME. Once inside the network simulator, these packets are treated simply as simulation events. PRIME simulates packet forwarding according to the virtual network condition regardless whether the packets are simulated or emulated packets. If the packets reach a destination node that has been emulated, they are exported to the corresponding emulated host, again via PRIME's emulation infrastructure. The packets arrive at the virtual network interfaces of the emulated host as if received from a real network. In the following subsections, we present more details on the components of SVEET.

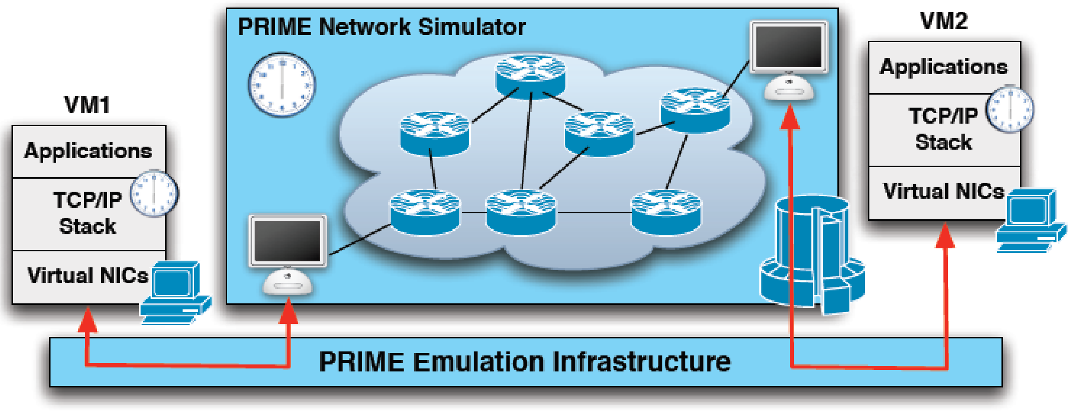


Figure 3.7: SVEET Architecture

3.3.3 Time Dilation

In order to test real applications running on virtual machines, plentiful bandwidth must be supported in the test network topology. There are two issues that could limit SVEET's capabilities of conducting experiments involving high-throughput network connections. First, high-bandwidth links transmit more packets per unit of time than slower links, which means more simulation events need to be processed in the network simulator and thus require more execution time. Second, high-capability links may cause more traffic to go through the emulation infrastructure situated between the virtual machines that generate the traffic and the network simulator that carries the traffic. The available bandwidth depends on the underlying network fabric, which could be substantially smaller than the bandwidth used in the test scenarios. In both cases, slowing down the progression of time both in the network simulator and the virtual machines enlarges the available bandwidth and the number of events that the simulator can process in a dilated second in order to satisfy the computation and communication demands of the test scenarios.

We adopt the time dilation technique by Gupta et al. in [GYM$^+$06]. This technology can uniformly slow the passage of time from the perspective of the guest operating system (XenoLinux). Time dilation can scale up the perceived I/O rate as well as the perceived processing power on the virtual machines. The ratio of the rate at which time passes in the physical world over the operating system's perception of time is called the *time dilation factor* (TDF). In Fig. 3.8, it is shown one system working on real-time on the left and another system working with TDF of 2, on the right. Each OS receives external events such as timer and device interrupts. Timer interrupts update the operating systems notion of time; in the system shown on the right, time dilation halves the frequency of delivered timer interrupts. Consequently, the dilated system appears to get network events at a higher rate that the one working on real-time.

Xen [BDF$^+$03] is chosen as the virtualization technique for our implementation. Xen is a high-performance open-source virtual machine solution. On most machines, Xen uses a technique called para-virtualization to achieve high performance by modifying the guest operating systems to obtain certain architectural features not provided by the host machines to support virtualization. SVEET currently supports Linux (a.k.a. XenoLinux under Xen) as the guest operating system. Specifically, SVEET was developed on Xen 3.0.4 and Linux kernel 2.6.16.33. This Linux kernel comes with all the TCP variants we want to include in our experiments. We also equipped the Linux kernel with Web100 [WEB], so that researchers can easily monitor and change TCP variables during the experiments.

In order to dilate the whole system, we set the TDF to be the same for all virtual machines and the network simulator at the start of the experiment in accordance with the maximum projected simulation event processing rate and emulation throughput.

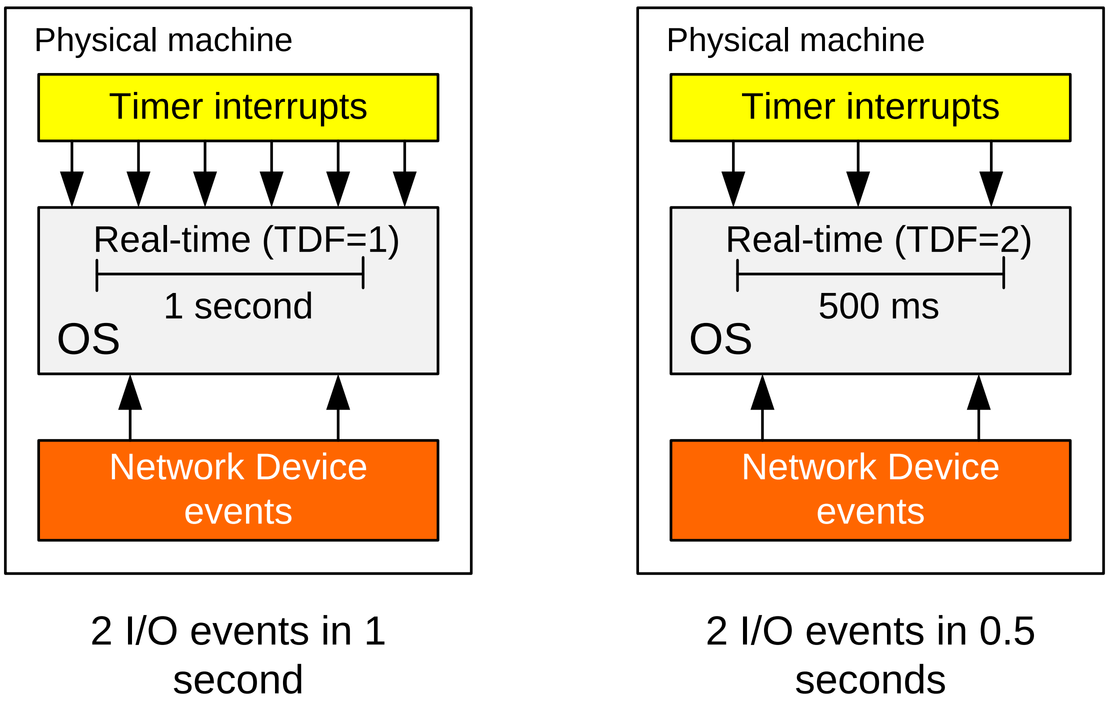


Figure 3.8: Comparison of a system operating in real-time (left) and a system running with a TDF of 2 (right).

3.3.4 Validation of Accuracy

This section is devoted to evaluate the accuracy of SVEET by examining the fairness between homogeneous TCP flows (i.e., using the same TCP variant). In order to further establish the accuracy of SVEET under time dilation, we perform this experiments using a TDF of 10. To that end, we create a dumbbell topology (similar to the one used in a recent TCP performance study by Li et al. [LLS07]) by adding another pair of end-hosts and attaching them separately to the two routers in our simple network model. We set the bandwidth of the bottleneck link to be 100 Mb/s and the delay to be 50 ms. At the start of each experiment, we select one of the end-hosts on the left to send data over TCP to one of the end-hosts on the right across the bottleneck link. After 50 seconds, the other end-host on the left establishes a separate TCP flow with the other end-host on the right. In this setting, all end-hosts are emulated and we set the TDF of the virtual machines and the simulator both to be 10. We measure the changes to the congestion window size over time at the senders of both flows.

Fig. 3.9 compares the results from SVEET and PRIME for TCP Reno, CUBIC, and Scalable TCP. In all cases, the emulation results match well with the corresponding simulation results. The slow convergence of Scalable TCP indicates that this protocol does not score well in intra-protocol fairness. This is mainly due to its aggressive congestion control mechanism, an multiplicative-increase and multiplicative-decrease (MIMD) algorithm. Such observations have been confirmed by earlier studies.

3.3.5 Assessing Time Dilation Effectiveness

In this section we evaluate SVEET's ability to deal with real traffic when running on a single physical machine [EL10a]. We choose the simple dumbbell network topology with a one-to-one mapping between the servers on the left side of the dumbbell and the clients

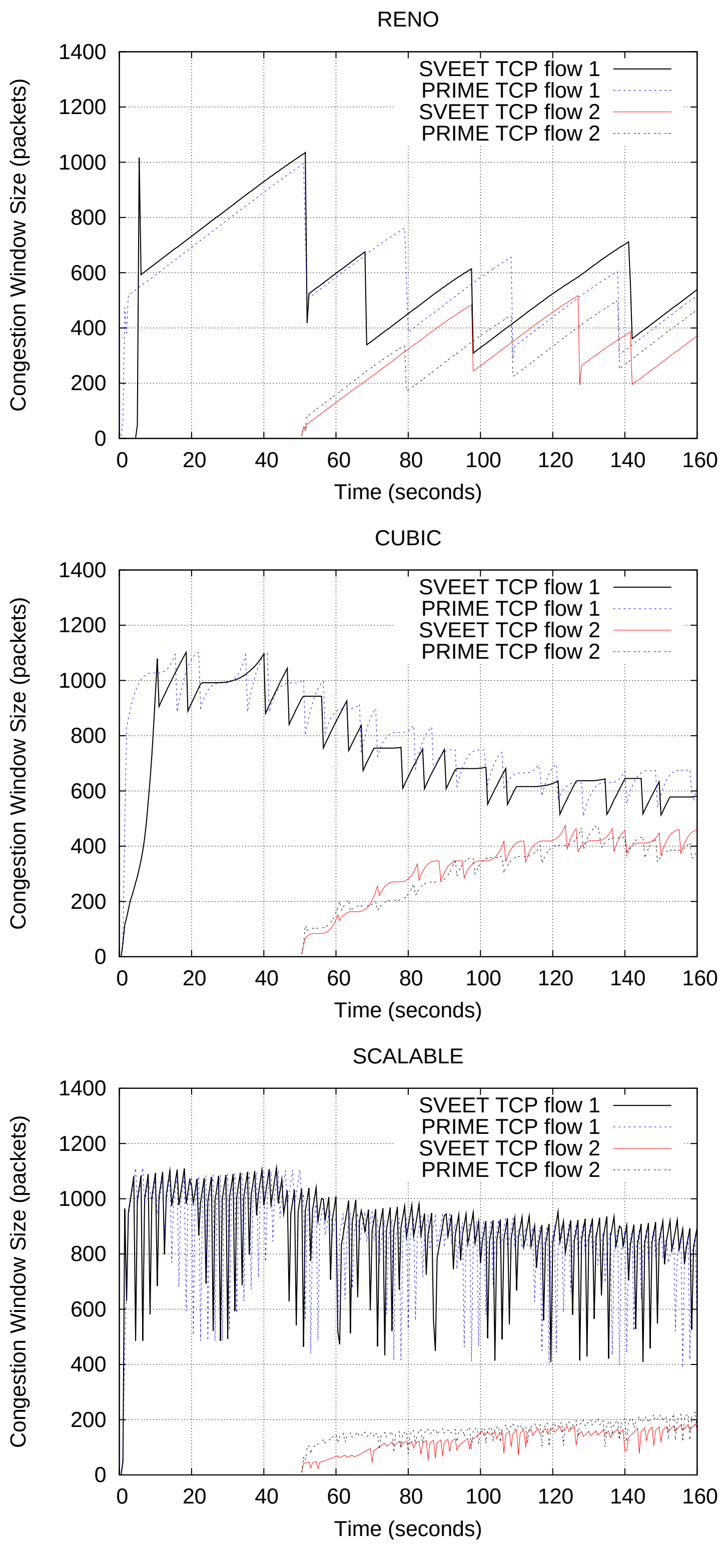


Figure 3.9: Congestion window sizes of two competing TCP flows

on the right side. The servers and clients are either simulated or emulated. We fix the ratio of emulated server-client pairs to simulated pairs to be 1:20. The clients and servers are connected through two simulated routers in the middle. In this case, the emulated and simulated traffic are multiplexed at the bottleneck link (middle link). We designate each emulated server to send a file to the corresponding emulated client using *iperf*. Likewise, we set each simulated server to transmit a file to the corresponding simulated client via FTP. We vary the number of servers and clients according to different traffic demands. We record the traffic intensity supported by the emulation infrastructure to see whether the system is still able to produce accurate results.

The experiments were conducted on a Dell Optiplex 745 workstation with Intel Core 2 Duo 2.4 GHz processors and 2 GB of memory. The dumbbell network is designed in such a way that the maximum achievable traffic through the bottleneck link would be limited solely by the emulation infrastructure (which is achieved by setting a large bandwidth for the bottleneck link). More specifically, we set the latency and the bandwidth of the bottleneck link delay to be 50 ms and 1 Gbps. The branch links (between each router and the end hosts attached to the router) are set to be 1 ms and 1 Gbps. We set the TCP maximum congestion window size to be 30 packets and the TCP maximum segment size to be 960 bytes.

Figures 3.10 and 3.11 depict the aggregate and per-flow throughput on the bottleneck link as a function of the number of simulated nodes. We perform 10 runs for each configuration and plott the throughput with 95% confidence intervals resultant from 100 seconds of experimentation. As expected, as the traffic demand increased, the aggregate throughput increases linearly up to a certain limit. Beyond that limit, the emulation infrastructure can no longer sustain the traffic demand; TCP is scaled back leading to a reduced throughput. Fig. 3.10 clearly shows the achievable limits when run on a single machine with different TDFs. Fig. 3.11 shows that the average throughput per flow remains con-

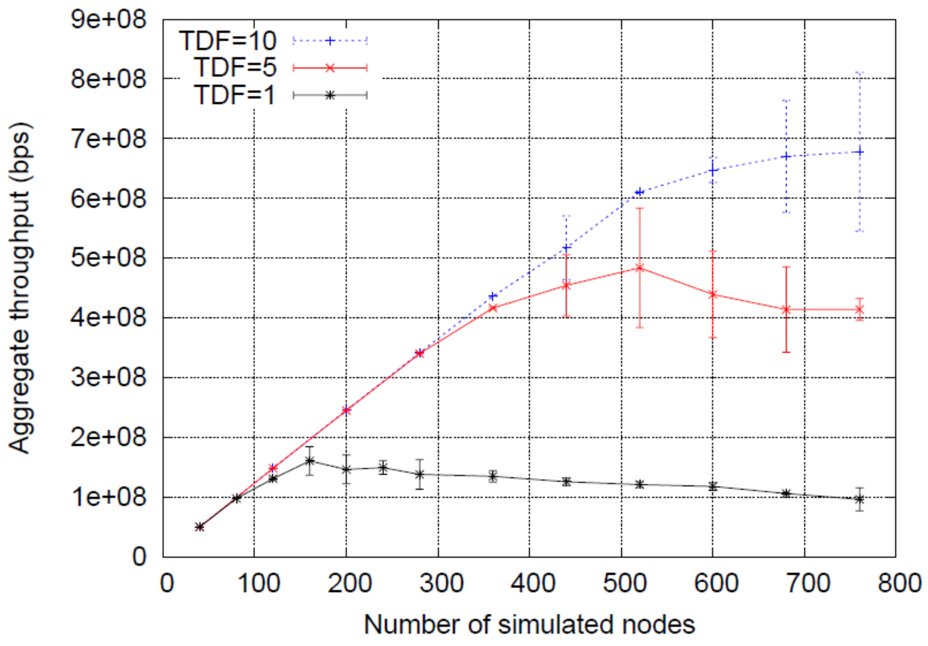


Figure 3.10: Aggregate throughput as a function of the number of simulated nodes for different TDFs

stant before degrading progressively as the traffic increased. We observe that the variance of the throughput measurement is more pronounced when the maximum throughput was achieved. Also, we observe that, although the throughput increases with higher TDFs, this increase is not linear. The maximum aggregate throughput is approximately 160, 480, and 670 Mbps, corresponding to TDFs of 1, 5, and 10. In fact, as we increase TDF, a smaller gain is obtained in terms of the achievable throughput.

3.3.6 Case Studies

In this section, we show the results of case studies that we conduct to show the usability of SVEET. Background traffic is incorporated to make the setting more realistic.

Background traffic is known to have a significant impact on the behavior of network applications and protocols. Floyd and Kohler [FK03] have been strongly advocating the use of better models for network research, including background traffic models, through

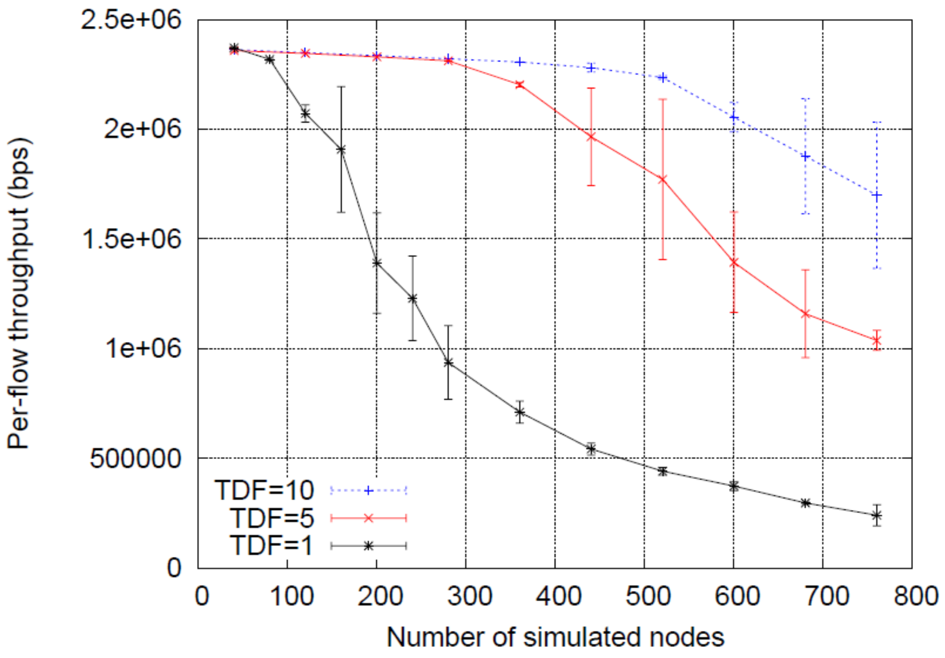


Figure 3.11: Average per-flow throughput as a function of the number of simulated nodes for different TDFs

careful examination of unrealistic assumptions in modeling and simulation studies. Ha et al. [HLRX07] conducted a systematic study of high-speed TCP protocols and demonstrated conclusively that the stability, fairness, and convergence speed of several TCP variants are clearly affected by the intensity and variability of background traffic. Recently, Vishwanath and Vahdat [VV08] investigated the impact of background traffic on distributed systems. They concluded that even small differences in the burstiness of background traffic can lead to drastic changes in the overall application behavior. In this section, we describe a set of case studies we conduct to assess the global effect of background traffic generated by the TCP variants on real applications.

Single Bottleneck

The experiments described in this section are conducted in the dumbbell topology (as shown in Fig. 3.12). Emulated traffic from real applications using a particular TCP vari-

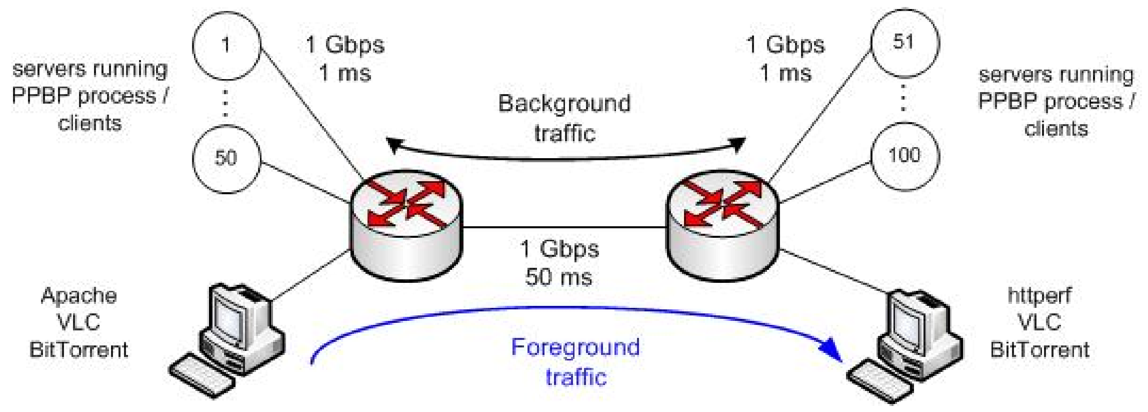


Figure 3.12: Single bottleneck topology

ant is the subject of this study. Emulated traffic competes for bandwidth with simulated background traffic, which is generated by 100 simulated nodes using the same TCP variant as the emulated traffic from real applications. A systematic study of the impact of background traffic on the performance of real applications, conducted by Vishwanath and Vahdat [VV08], was used as our guideline to configure the background traffic. Their study suggests that simple traffic models, such as constant bit rate (CBR) and Poisson arrival, cannot capture the complexity of real Internet traffic. Background traffic should be bidirectional and a good background traffic model is needed to capture traffic burstiness in a range of time scales. To represent the aggregate traffic behavior, we decided to use the *Poisson Pareto Burst Process* (PPBP), described by Zukerman et al. in [ZNA03]. PPBP is a process based on multiple overlapping bursts with Poisson arrivals and burst lengths following a heavy-tailed distribution. The major parameters of PPBP include the mean arrival rate (μ), the mean session duration (d), and the Hurst parameter ($\mathscr{H}$). For self-similar traffic that exhibits long-range dependencies (LRD), $0.5 < \mathscr{H} < 1$. We configured the background traffic corresponding to a light traffic load scenario (with $\mu = 1$) and a heavy traffic load scenario (with $\mu = 100$). We set $\mathscr{H} = 0.8$ and $d = 1$ second.

We place the servers on either side of the dumbbell topology, shown in Fig. 3.12, to create the bidirectional background traffic. For foreground traffic, we select three applications: web downloading, multimedia streaming, and peer-to-peer applications. For web downloading, we use httperf to measure the response time of downloading web objects of different size across the bottleneck link from the Apache server, subject to both light and heavy background traffic conditions. We vary the size of the web objects to be 10 KB, 100 KB, and 1 MB. Fig.3.13 depicts the empirical cumulative distribution function of the response time, defined as the time between the client's sending the request and finally receiving the entire object. We ran 30 independent trials for each TCP variant. The results show that, although the response time for small objects is almost indistinguishable among the TCP variants, with larger object sizes, certain TCP variants perform better than others. For multimedia streaming, we measure jitter (the difference in transit time between successive packets) as an indication of the perceived quality of a video stream. We used VLC from VideoLAN as the client playing an MPEG-4 movie streamed from an Apache server over the dumbbell network. To compute jitter, we capture packets at both server and client sides. We compute jitter from 100 seconds of video streaming for 15 independent trials for each TCP variant. Fig. 3.14 depicts the empirical cumulative distribution of jitter. CUBIC exhibits the best performance among the three TCP variants. For peer-to-peer applications, we measure the time for distributing a large data file. We use SVEET to evaluate the performance of BitTorrent. The test scenario consisted of one tracker and one seed, both running on the same emulated machine, and three peers, each on a different emulated host located on either side of the dumbbell. The peer-to-peer network is used to distribute a data file of 20 MB in size. We consider only the heavy traffic load condition for this experiment. The results, as shown in Fig. 3.15, clearly indicate that CUBIC outperforms Reno and Scalable TCP.

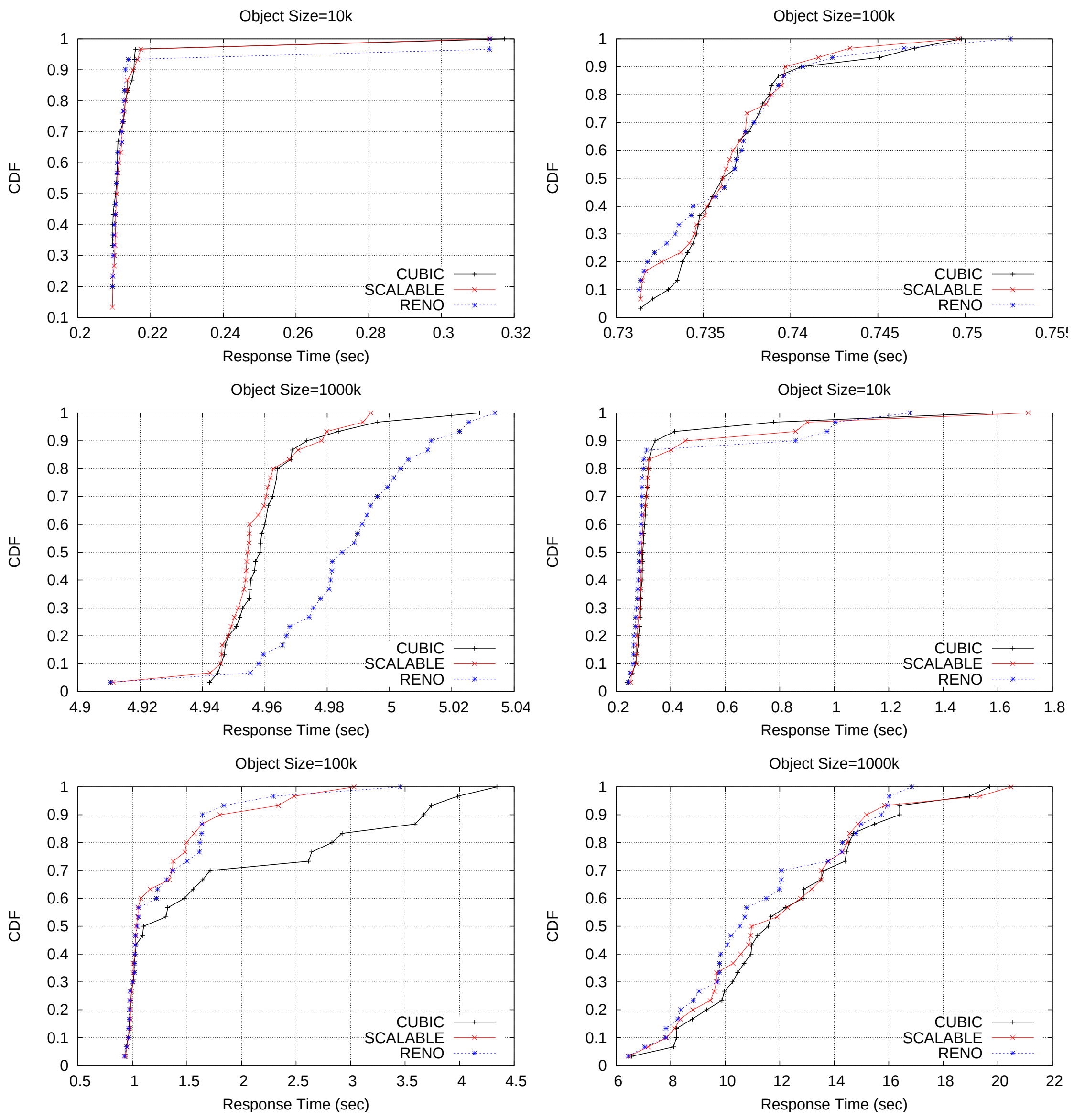


Figure 3.13: Response times for different object sizes

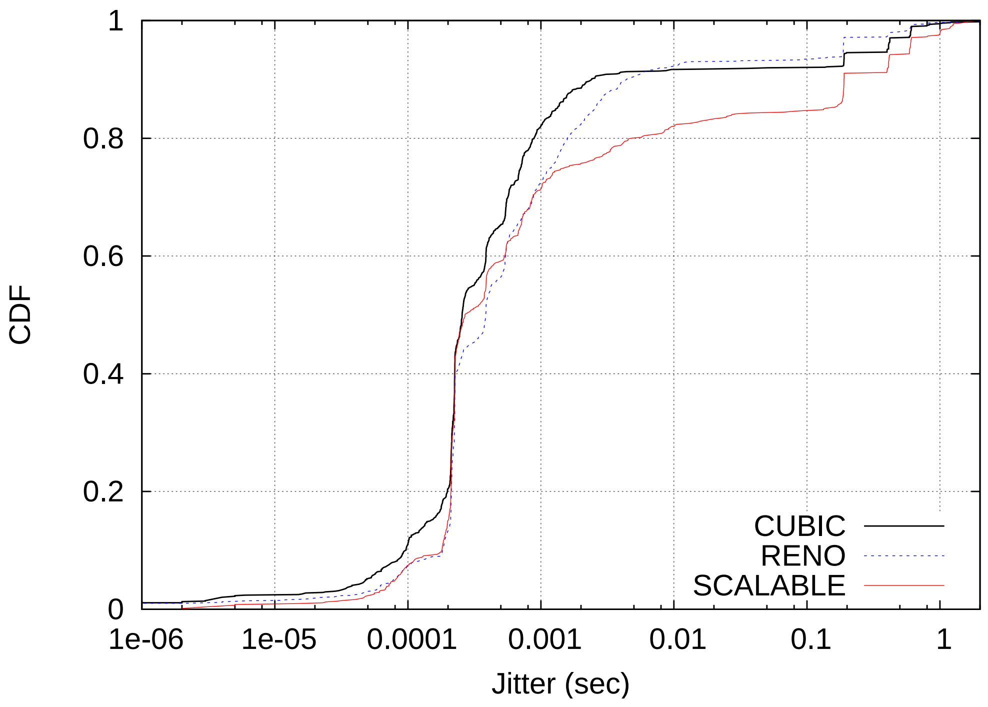


Figure 3.14: Jitter from video streaming

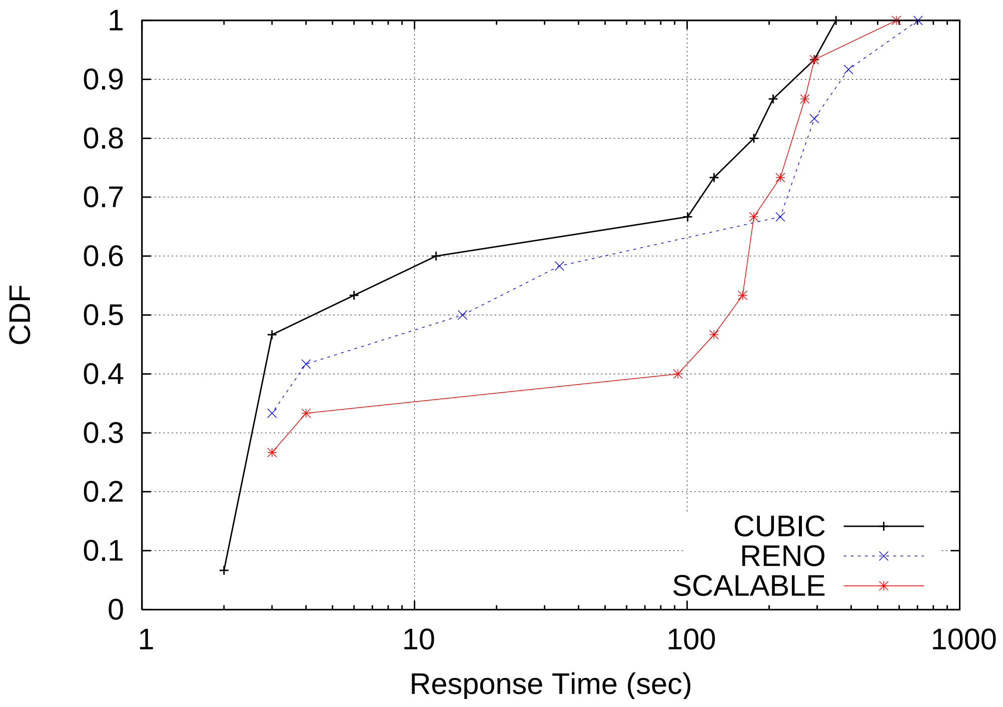


Figure 3.15: BitTorrent download time

Synthetic Topology

In order to show SVEET's capability of dealing with larger and more complex network scenarios, we conduct another experiment using a synthetic network topology, called the *campus network*. The network, consisting of 508 end-hosts and 30 routers, is a scaled-down version of the baseline network model that has been used for large-scale simulation studies. The network is shown in Fig. 3.16. It contains four subnets; within net2 and net3, there are 12 local area networks (LANs), each configured with a gateway router and 42 end-hosts. The LANs are 10 Mbps networks. For links connecting routers within net1 and net2, we set the bandwidth to be 100 Mbps and the link delay to be 10 ms. For other links connecting the routers, we set the bandwidth to be 1 Gbps and the link delay to be 10 ms. In this experiment, each end-host acts as an on-off traffic source: the node stays idle for a period of time, which is exponentially distributed with a mean of one second, before sending data using TCP to a randomly selected end-host in net1 for a duration, sampled from a Pareto distribution with the mean of one second. We enable time dilation and set TDF to 10 for both simulation and the virtual machines.

We place an Apache web server at one of the emulated end-host in net1 and selected another end-host in net2 as an emulated host running httperf to fetch objects from the web server. We use the same TCP variants for both simulated background traffic and emulated foreground web KB, 100 KB, and 1 MB in size. We collected measurements of 30 independent trials. Fig. 3.17 shows the empirical cumulative distributions of the response time. Results show that different TCP variants produced drastically different results. TCP Reno achieved the best response time among the three TCP variants. We speculate this is due to the protocol's better performance in terms of intra-protocol fairness; in this case, the foreground traffic could retain a larger share of the link bandwidths for downloading the objects. Surprisingly, SCALABLE seemed to perform better than CUBIC. The results

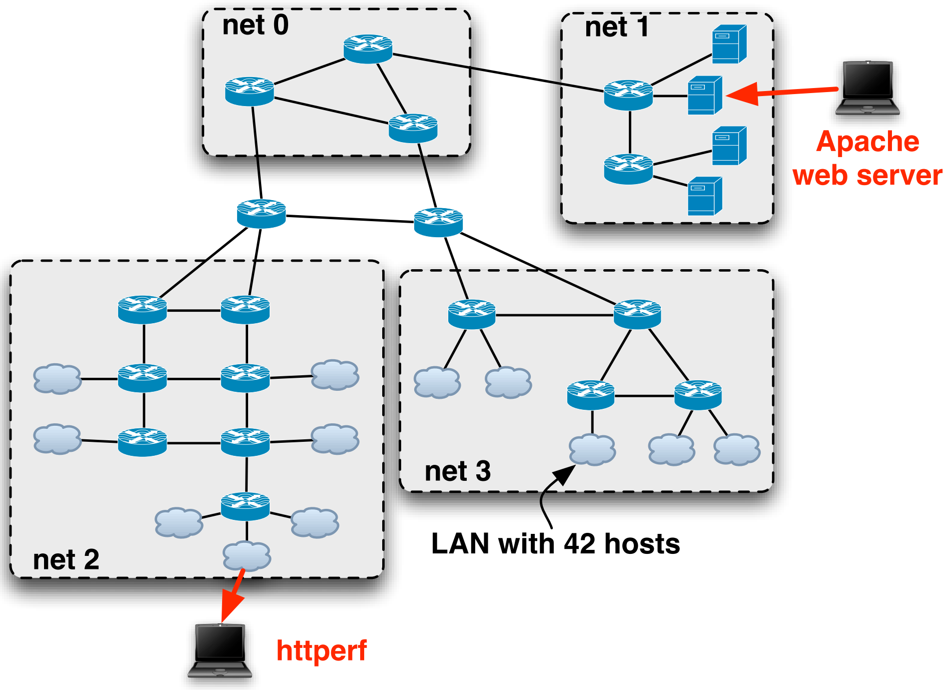


Figure 3.16: A campus network model

suggest that we need to further investigate the details of the background traffic used in this experiment, as well as its impact on the link utilization. Here we only use this example to show that SVEET can now enable us to begin studying large-scale TCP behaviors in fairly complex network settings.

3.4 Summary

In this chapter, we present SVEET, a system that enables network experiments involving emulated and simulated entities in dilated time. Results conclusively show that our system produces accurate results up to a given threshold depending on the TDF used. The maximum achievable limit by our emulation approach is found to be 160, 480, and 670 Mbps for TDF values of 1,5, and 10 respectively. Also, we demonstrate that the time dilation technology we adopted does not scale well as the value of TDF increases.

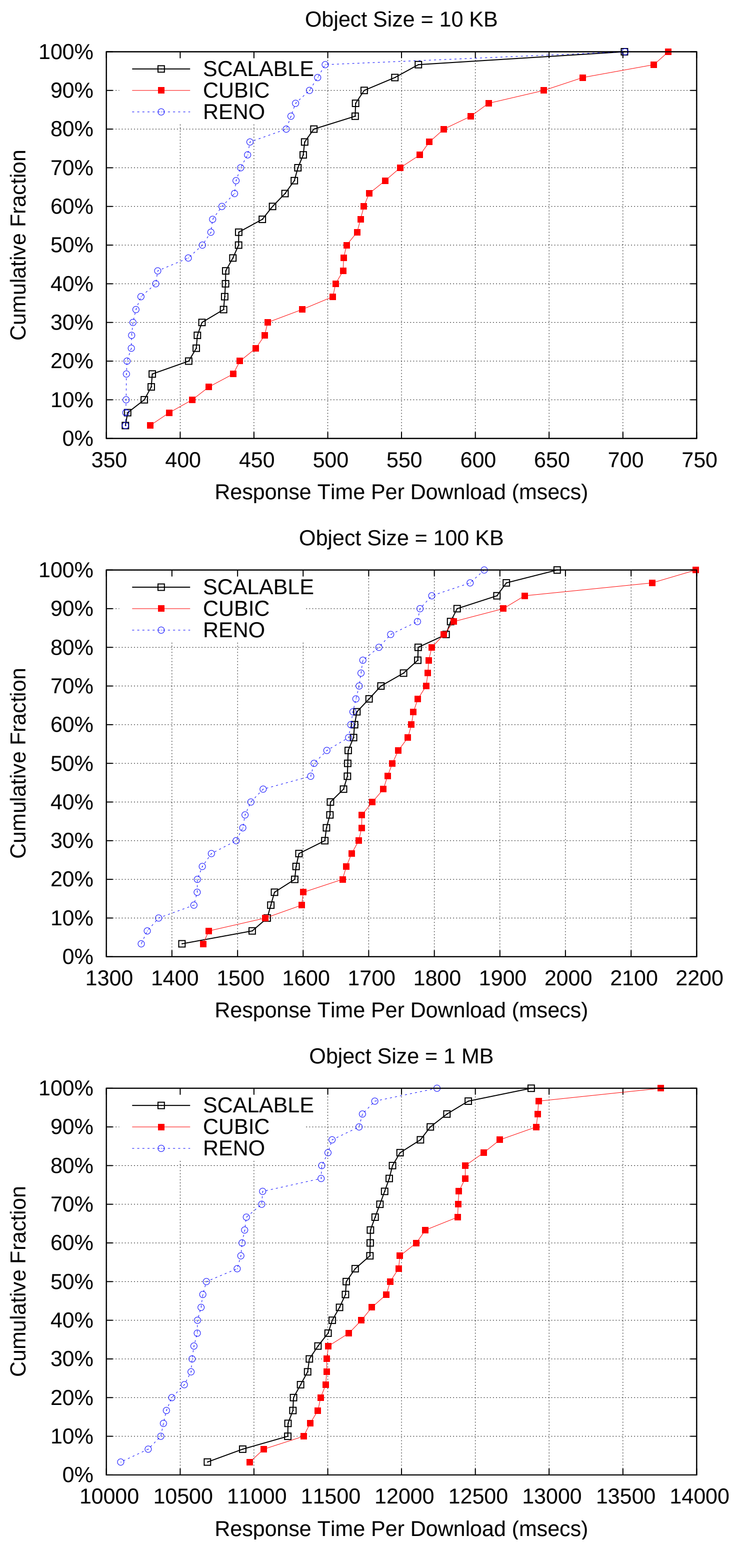


Figure 3.17: Response time under synthetic network model

Also, we describe our TCP implementation for our discrete-event simulator. Our TCP shows nearly identical results to those produced by the network simulator (ns-2); which was previously benchmarked against real implementations.

CHAPTER 4

ON ENABLING INTER-OPERATION BETWEEN SIMULATION AND EMULATION ON FEDERATED ENVIRONMENTS

In this chapter, we present PrimoGENI, a system that enables real-time network simulation by extending an existing network simulator to become part of a federation in order to support large-scale experiments involving physical, simulated and emulated network entities.

4.1 Motivation

Real-time simulation allows running a network model in real-time allowing interactions with real applications. It provides accurate results since it is able to capture detailed packet-level transactions. Flexibility is another of its advantages since events can be readily incorporated into the model to answer what-if questions.

Both simulation and real(physical) systems can benefit from each other in many different ways in order to achieve more faithful experiments and better testbeds. For instance, Yao et al. target network protocol configuration as a *black-box* optimization algorithm over the parameter state space [YKKY08]. A point in this space corresponds to a simulation run that evaluates the performance. The simulation imports the current network topology and traffic patterns from the real network so as to evaluate the utility function. Rosenet [Gu07] also fosters symbiotic relationships with a closed-loop symbiotic simulation system, where the physical system (in this case, the emulation system) benefits from the *low fidelity model* provided by the simulator while the simulator benefits from the the *traffic summary model* provided by the physical system.

In this chapter, we propose PrimoGENI, a system which aims at embedding real-time simulation into the GENI [GEN] *federated system*. GENI is a federated system composed of many aggregates. Doing so, enables PrimoGENI to establish symbiotic relationships with other real systems of different nature. Many types of symbiotic relationships are possible in this setup and each one poses different research and practical questions:

1. How to reserve and connect resources from multiple geographically distributed sites.
2. Once we have the computational resources in place, how to deploy experiments that enable interactions between real hosts and our simulator.

4.2 Introduction

Real-time simulation complements the GENI concept of federating global resources as a shared experimental network infrastructure. Our immersive large-scale network simulator, PRIMEX, supports experiments potentially with millions of simulated network entities (hosts, routers, and links) and thousands of emulated elements running unmodified network protocols and applications. PrimoGENI [VVEL11] and PRIMEX [VVEL12] are extended to fulfill all the required features of federated component managers in GENI:

- *Programmability*. Researchers should be able to download and run software into the allocated nodes for a slice in an experiment.
- *Resource Sharing*. Virtualization should allow a testbed to be shared simultaneously among users.
- *Federation*. Different component managers of the GENI federation are operated by different organizations.
- *Slice-based Experimentation*. GENI experiments are an interconnected set of reserved resources (*slices*) on platforms in diverse locations.

In order to interact with other GENI facilities, PrimoGENI functions as a GENI *aggregate* or *component manager*, so that experimenters can use a well-defined interface to remotely control and realize network experiments consisting of both physical, simulated and emulated network entities exchanging real network traffic.

PrimoGENI uses the ProtoGENI[PRO] control framework to manage, control and access the underlying resources. In PrimoGENI, we make distinction between *meta* and *virtual resources*. Meta resources include compute nodes and network connectivity between the compute nodes. We call these resources *meta resources* to distinguish them from the physical resources (also known as the substrate), since they could be virtual machines and virtual network tunnels. Meta resources are managed by and accessible within the ProtoGENI/Emulab suite. Virtual resources are elements of the virtual network instantiated by PRIMEX, which include simulated hosts, routers, links, protocols, and emulated hosts. We call these resources *virtual resources* as they represent the target (virtual) computing and network environment for the GENI experiments; they encompass both simulated network entities and emulated hosts; which are run on the virtual machines. PrimoGENI exports an aggregate interface as defined by the ProtoGENI control framework, and provides mechanisms for instantiating the virtual network onto the ProtoGENI/Emulab facilities as configured and allocated on behalf of the experimenter. In addition, PrimoGENI provides experiment support services to facilitate large-scale network experiments involving simulated and emulated components; these services include model configuration, resource specification, simulation deployment and execution, online monitoring and control, data collection, inspection, visualization and analysis.

4.3 Architecture

The PrimoGENI aggregate can be viewed as a layered system, as shown in Figure 4.1. At the lowest layer is the physical resources (*substrate*) layer; which is composed of cluster nodes, switches, and other resources that constitute the Emulab suite. These resources are made known to the clearinghouse(s) and can be queried by researchers during the

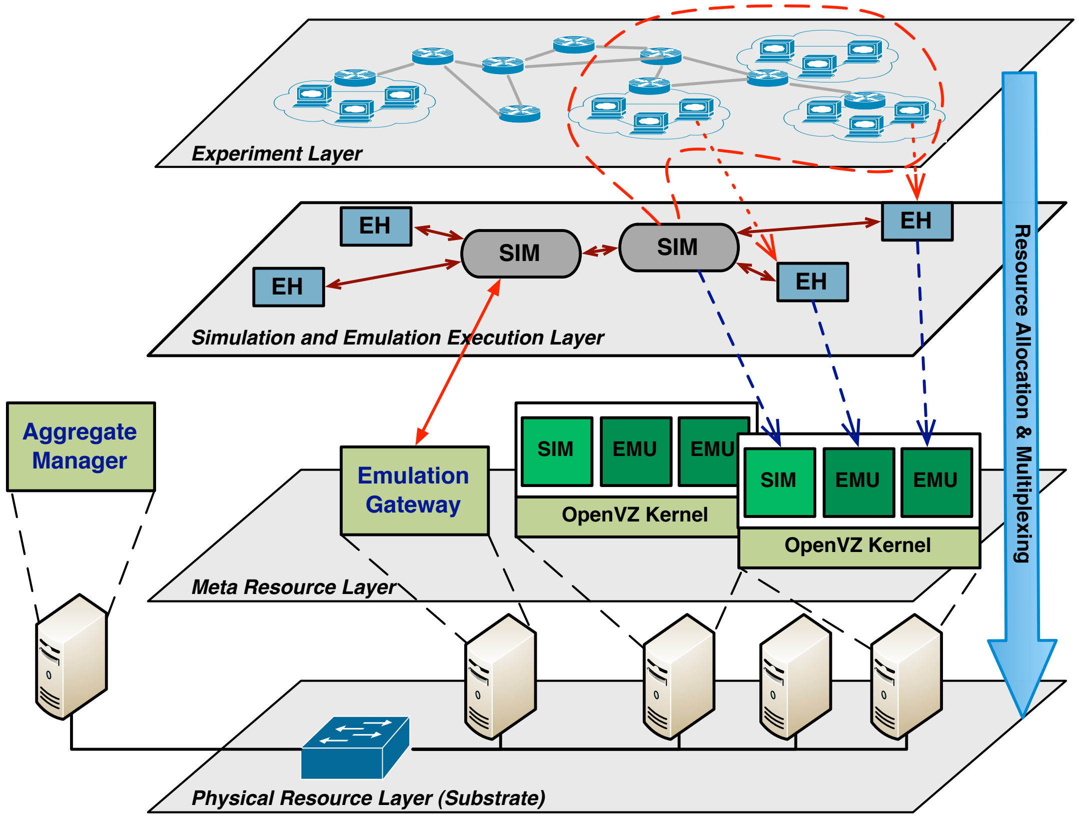


Figure 4.1: PrimoGENI architecture

resource discovery process. In addition, two servers are set up, the *aggregate manager* or *master* (for exporting an aggregate interface to researchers) and the *emulation gateway* (for communicating with other slivers on other aggregates), respectively. A meta resources layer is created upon resource assignment in a sliver. PrimoGENI uses the Emulab suite to allocate the meta resources (including a subset of cluster nodes, VLAN connectivity among the nodes, and possible GRE [GRE] channels created for communicating with resources on site). Each physical cluster node is viewed by PrimoGENI as an independent scaling unit loaded with an operating system image that supports virtual machines (e.g., OpenVZ). Multiple virtual machines are created on the same physical machine running a PRIME instance. In particular, the simulator runs on the host OS, and the emulated hosts are mapped to separate virtual machines so that they can run unmodified applications.

A simulation and emulation execution layer is created according to the virtual network specification of a sliver. The PRIMEX instances and the emulated hosts are mapped to the meta resources at the layer below. Currently, VM's inject packets to the simulator using OpenVPN[OPEa] and/or *emulation device drivers* [VVEL12], so that traffic generated by the emulated hosts is captured by the real-time simulator and conducted on the simulated network with appropriate delays and losses according to the specified network conditions. Each real-time simulator instance handles a sub-partition of the virtual network; they communicate through VLAN channels created by the Emulab suite at the meta resources layer. They also establish connections to the emulation gateway for traffic to and from slivers on other aggregates. Once the slivers are created and the slice is operational, researchers can conduct experiments on the experiment (logical) layer. Experimenters are able to log into individual emulated hosts, upload software, and launch it. Traffic between the emulated hosts will be conducted on the virtual network. Traffic originated from or destined to other physical network entities, e.g., traffic sent between slivers in different physical locations, will be redirected through the emulation gateway.

4.4 Experiment Workflow

The mechanics of creating an experiment (slice) start when a user of PrimoGENI specifies a network model using Java, Python, or XML. She submits this file to our user interface called *Slingshot* [VVEL12]. Slingshot, which has embedded the PrimoGENI functionality, performs the following steps on behalf of the user against a PrimoGENI-enabled Emulab/ProtoGENI site:

- It parses the file submitted by the user and then extracts valuable information such as the number of emulated hosts that have to be instantiated; which will be mapped later to a virtual container or virtual machine. Also, the number of physical machines needed to instantiate the intended model is extracted form this file.
- Using the information extracted from the previous step, it creates a file called *resource specification* (RSpec), which is a standard XML-like file used to describe the requested resources to a ProtoGENI site.
- Using the *certificate* provided by the user, which is specific to the ProtoGENI site where the slice will be instantiated, PrimoGENI requests the resources.
- Upon acceptance, ProtoGENI emits a *ticket* that specifies the resources that a component manager (CM) allocated (or promises to allocate) to a given slice.
- After the ticket is received, Slingshot redeems the ticket on the incumbent clearing house and then the slice is initiated.
- Alternatively, the user may instantiate the slice herself using the ProtoGENI provided tools like the *test scripts* [TES] or *flack*[FLA] and only provide Slingshot the *manifest* (a XML file describing the resources already provided to the user).

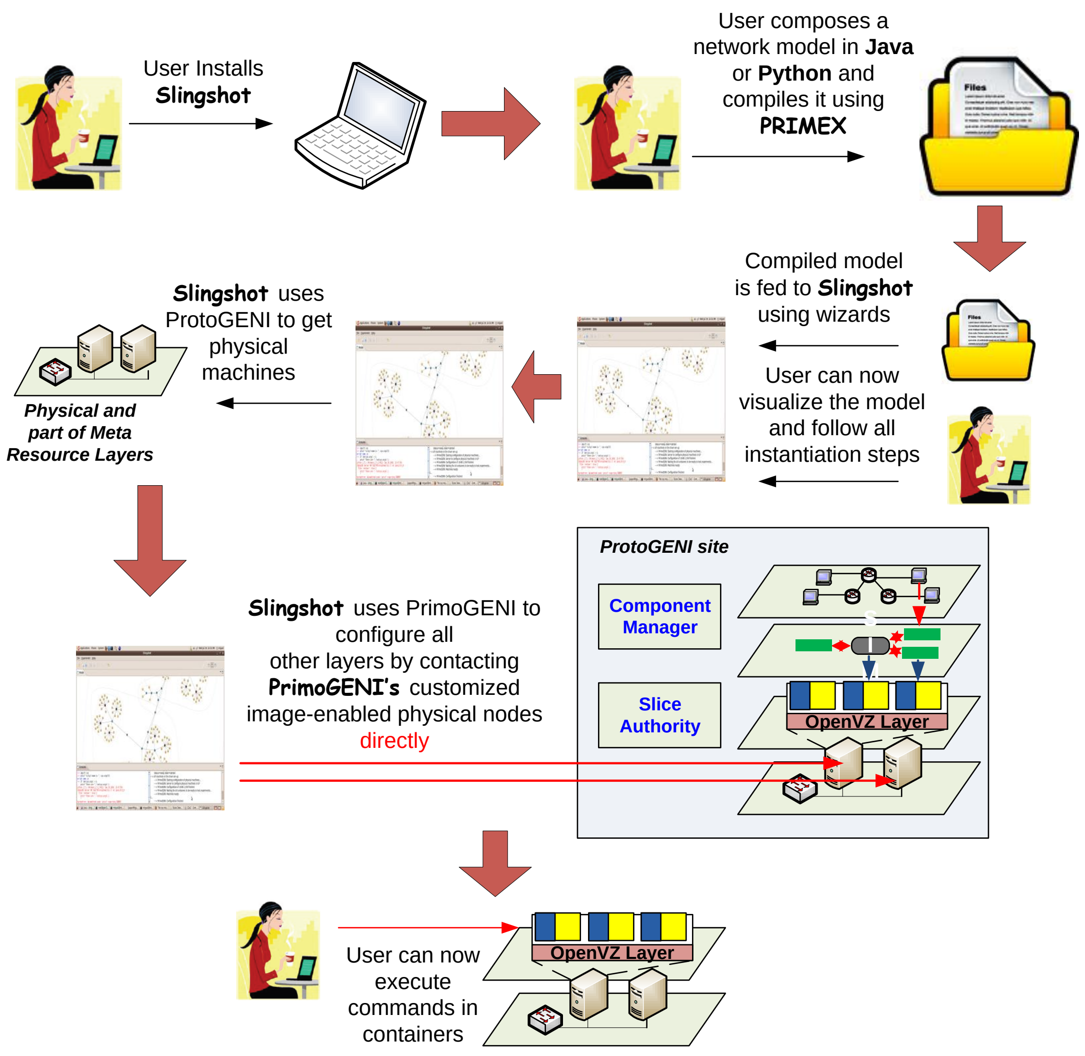

User Installs **Slingshot**
User composes a network model in **Java** or **Python** and compiles it using **PRIMEX**
Files
Compiled model is fed to **Slingshot** using wizards
User can now visualize the model and follow all instantiation steps
Slingshot uses ProtoGENI to get physical machines
Physical and part of Meta Resource Layers
ProtoGENI site
Component Manager
Slice Authority
OpenVZ Layer
Slingshot uses PrimoGENI to configure all other layers by contacting **PrimoGENI's** customized image-enabled physical nodes directly
User can now execute commands in containers
OpenVZ Layer

Figure 4.2: PrimoGENI's experiment workflow

- After the physical resource layer is created by ProtoGENI. Slingshot coordinates with experimental (physical) nodes to boot up a set of processes in an specific order, i.e., it creates the meta resource layer.
- At this point, the user may have to manually set up routes and tunnels to other slivers if necessary.
- Slingshot then starts the simulator and the experiment is ready to use at this point.

The whole experiment workflow is shown in Fig. 4.2. In the current setup, we use OpenVZ [OPEb] as the virtualization technique to host emulated hosts. To that end, we use RSpecs to specify the required number physical machines and experiment layer specific information that we use during ticket redemption. Four different network entities can be currently instantiated in physical nodes: simulator, emulation gateway, emulated nodes, and remote nodes (emulated nodes connected through the gateway). During ticket redemption, we can map one or more aforementioned entities to a physical machine.

4.5 Meta-Controllers

PrimoGENI uses the ProtoGENI control framework to allocate the necessary compute nodes to run experiments. After the compute nodes are booted from the customized OpenVZ image, we need a mechanism to configure the compute nodes, create the containers, set up the emulation infrastructure, start the simulation, and launch the applications inside the containers. One can use XML-RPC for this purpose. However, since the commands are originated from Slingshot, we created a mechanism that allows the compute nodes to be coordinated locally on the PrimoGENI cluster to achieve better efficiency. In this section, we describe a tiered command framework, which we use to set up a network experiment and orchestrate experiment execution.

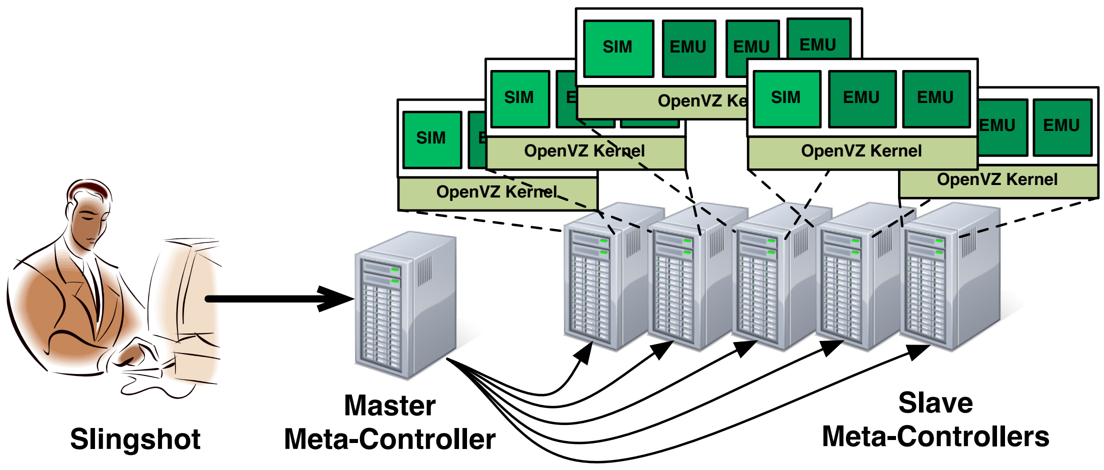


Figure 4.3: Meta-controllers for experiment setup

The command framework, as illustrated in Fig. 4.3, uses MINA [MIN], a Java framework for distributed applications. MINA provides both TCP and UDP-based data transport services with SSL/TLS support. It is designed with an event-driven asynchronous API for high-performance and high scalable network applications. Our implementation requires each compute node to run a meta-controller daemon process when first booted. When the meta-controller starts, it waits for an incoming connection. At this point the meta-controller takes no role, but after the connection is made from Slingshot, the meta-controller will become either a master or a slave. The difference between a master meta-controller and a slave meta-controller is that the master meta-controller is the only one directly connected to Slingshot. Slingshot chooses one compute node to be the master and establishes a connection to it. After Slingshot successfully connects to the master, Slingshot instructs it to take control over the remaining compute nodes. Then, the master establishes a connection to each of those compute nodes and instructs them to act as slaves. Commands sent from Slingshot, if they are not destined for the master, will be relayed to the corresponding slave meta-controller via the master.

After that, Slingshot sends commands to the master meta-controller, which properly distributes the commands to the slave meta-controllers, if needed. Each command specifies the target compute node or a virtual machine (hereon referred to as *(*container)) where the command is expected to run. A command can be either a blocking command or a nonblocking command. If it is a blocking command, the meta-controller waits until the the command finishes execution and sends back the result of the command (i.e., the exit status) to Slingshot. If the command is a nonblocking command, the meta-controller forks a separate process to handle the command and immediately responds to Slingshot. Our current implementation uses blocking commands to set up the containers and the emulation infrastructure, and uses nonblocking commands to start the simulator and the user applications within the containers.

To set up the experiments correctly, the meta-controllers on the compute nodes need to run a series of commands:

1. *Set up MPI*. The master meta-controller creates the machine file and instructs the slave meta-controllers to generate the necessary keys for MPI to enable SSH logins without using passwords.
2. *Create containers*. The meta-controller creates a container for each collocated emulated host on the compute node. In case the experimenter desires to save disk space and make all containers to share the same OS, the this step also includes the creation of union file systems for the containers [FUN].
3. *Set up the emulation infrastructure*.For collocated emulated hosts, this step includes installing the virtual Ethernet devices in the containers, creating and configuring the software bridges and TAP devices, and then connecting the network devices to the bridges [VVEL12]. For remote emulated hosts, this step includes setting up the OpenVPN server(s).

4. *Run the experiment*. The partitioned network model is distributed among slave compute nodes. The master meta-controller initiates the MPI run, which starts the simulator instances on each compute node with the partitioned model.
5. *Start applications within containers*. Individual commands are sent to the meta-controllers to run applications at the emulated hosts.
6. *Shut down the experiment*. At any time, one can shut down the experiment by terminating the simulator, stopping the containers, and removing the emulation infrastructure (such as the bridges and the TAP devices).

All these commands are issued from Slingshot automatically using the meta-controller command framework.

4.6 Experiments

In this section, we describe the set of experiments we performed to validate the accuracy of our testbed (in terms of throughout), and determine its performance limitations, in order to show the utility of our approach. The experiments described in this section are conducted on a prototype PrimoGENI cluster with eight Dell PowerEdge R210 rack-mount servers, each with dual quadcore Xeon 2.8 GHz processors and 8 GB memory. The servers are connected using a gigabit switch.

4.6.1 Validation Studies

We validate the accuracy of the testbed by comparing the TCP performance between emulation and simulation. We use TCP for validation because it is highly sensitive to the delay jitters and losses and therefore can magnify the errors introduced by the emulation infrastructure. Previously, we performed validation studies for the VPN-based emulation infrastructure [ELL09], which is designed for connecting applications running on remote machines. In this study, we focus only on the emulation infrastructure based on software bridges and TAP devices, which we use to connect the collocated emulated hosts with the simulator instances run on multiple compute nodes.

We first compare the TCP congestion window trajectories achieved by the real Linux TCP implementations on the OpenVZ containers against those from our simulation. We arbitrarily choose three congestion control algorithms −BIC, HIGHSPEED, and RENO− out of the 14 TCP variants we have implemented in the PRIME simulator [ELL09]. We use a dumbbell model for the experiments. The dumbbell model has two routers connected with a bottleneck link with 10 Mb/s bandwidth and 64 ms delay. We attach two hosts to the routers on either side using a link with 1 Gb/s bandwidth and negligible delay. The buffers in all network interfaces are set to be 64 KB. In the experiment, we direct a TCP flow from one host to the other that traverses the two routers and the bottleneck link. For each TCP algorithm we test three scenarios. In the first scenario, we perform pure simulation and use a simulated traffic generator. In the second and third scenario, we designate the two hosts as emulated hosts and use iperf to generate the TCP flow (and measure its performance). We run the emulation on one compute node for the second scenario, in which case the compute node runs simulator in container 0 and two other containers as the emulated hosts. In the third scenario, we use two compute nodes, each running one simulator instance and one emulated host. The emulated traffic in this case has to travel across the memory boundary between the two compute nodes in simulation (using MPI). For simulation, we use a script to analyze the trace output; for emulation, we sample the (/proc/net/tcp) file at regular intervals to extract the TCP congestion window size. Fig. 4.4 shows very similar TCP congestion window trajectories between simulation and emulation.

Next, we use a TCP fairness test to show whether our approach can correctly intermingle emulated and simulated packets. We use a similar dumbbell model; however, in

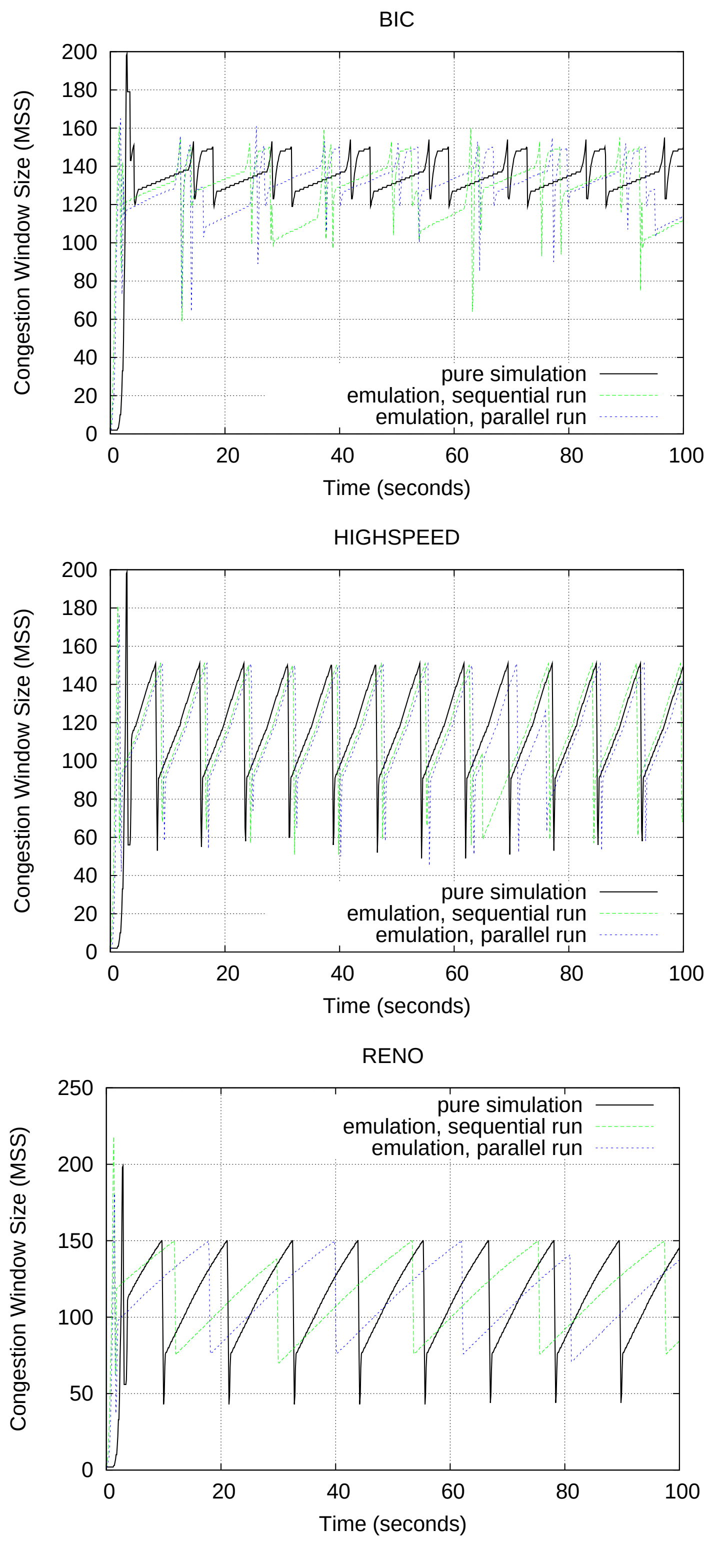


Figure 4.4: TCP congestion window trajectories for simulation and emulation

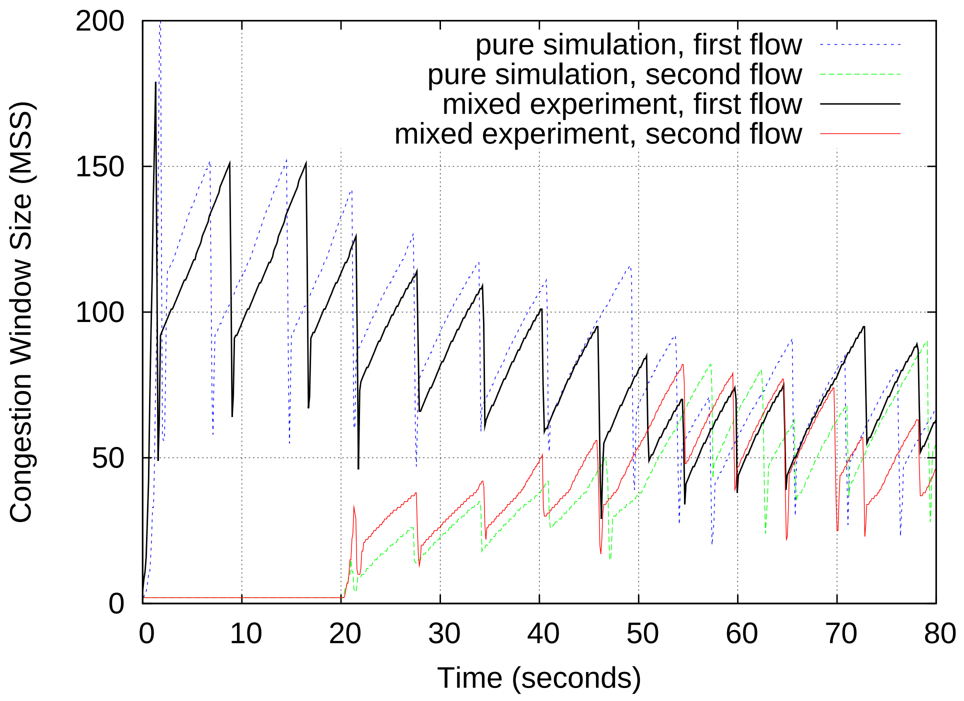


Figure 4.5: TCP fairness behavior for simulation and emulation

this time, we attach two hosts on either side of the routers. We generate two TCP flows in the same directionone for each of the two hosts on the left side to one of the two hosts on the right side. We select the TCP HIGHSPEED algorithm for both flows. We start one flow 20 seconds after the other flow. We compare two scenarios: in the first scenario we perform pure simulation; and in the second scenario, we designate the two end hosts of the first TCP flow as emulated hosts. The results are shown in Fig. 4.5. For both scenarios, we see that the congestion window size of the first TCP flow reduces when the second TCP flow starts; both flows eventually converge with a fair share of the bandwidth (at about 30 seconds after the second flow starts transmitting). Again, we see very similar results between simulation and emulation.

The emulation infrastructure inevitably puts a limit on the throughput of the emulated traffic. In the last validation experiment, we look into this limiting effect on the emulated TCP behavior. Again, we use the dumbbell model. To increase the TCP throughput, we reduce the delay of the bottleneck link of the dumbbell model to 1 millisecond. For the experiment, we vary the bandwidth of the bottleneck link from 10 Mb/s to 450 Mb/s with increments of 40 Mb/s. Like in the first experiment, we direct a TCP flow from one host to the other through the bottleneck link and we compare three scenarios: the first using pure simulation, the second with an emulated flow within one compute node, and the third with an emulated flow across two compute nodes. Fig. 4.6 shows the results. While the throughput increases almost linearly with the increased bandwidth for the simulated flow, the error becomes apparent for emulated traffic at high traffic intensity. The throughput for the emulated traffic is kept below roughly 250 Mb/s for sequential runs and 130 Mb/s for parallel runs. The reduced throughput for parallel runs is due to the communication overhead as the TCP traffic gets exposed to the additional delay between the parallel simulator instances. Since emulation accuracy heavily depends on the capacity of the emulation infrastructure. We look into the emulation performance in more detail in the next section.

4.6.2 Performance Studies

In the previous experiments we show that the emulation infrastructure determines an upper limit on the emulation traffic the system can support. Here we use a set of experiments to measure the capacity of the emulation infrastructure. We use the same dumbbell model (with a bottleneck link of 1 Gb/s bandwidth and 1 millisecond delay), and attach the same number of emulated hosts on each side of the dumbbell routers. We start a TCP flow (using iperf) for each pair of emulated hosts, one from each side of the dumbbell. So the total number of TCP flows is half the number of emulated hosts. We make same number of the TCP flows to go from left to right as those from right to left. We vary the number of emulated hosts in each experiment. We show the sum of the measured throughput (from iperf) for all emulated TCP flows in Fig. 4.7.

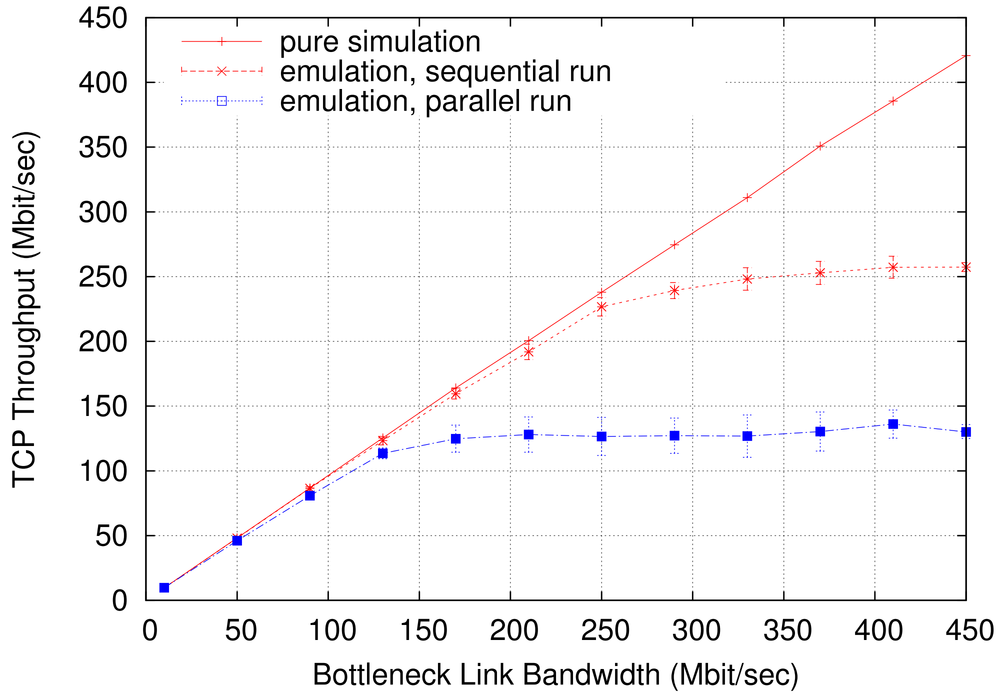


Figure 4.6: TCP throughput limited by emulation infrastructure

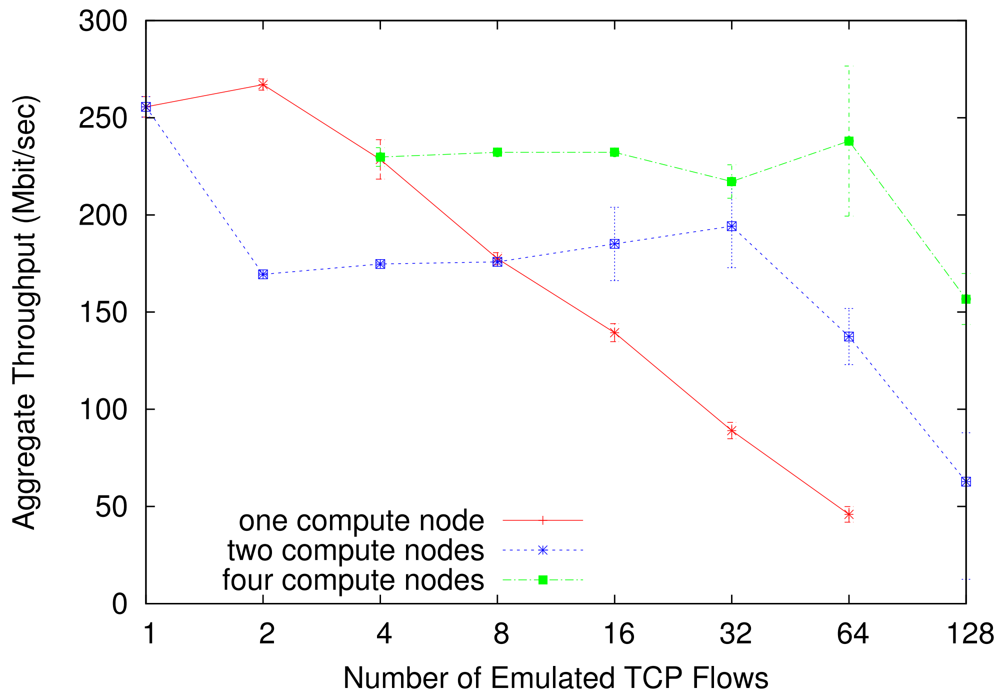


Figure 4.7: Aggregate TCP throughput vs. emulated TCP flow count

In the first experiment, we assign all emulated hosts to run on the same compute node. For one flow, the throughput reaches about 250 Mb/s, as we have observed in the previous experiment for sequential runs. The aggregate throughput increases slightly for two flows, but drops continuously as we increase the number of flows all the way to 64 flows (thats 128 VMs). The slight increase is probably due to TCPs opportunistic behavior that allows it achieve better channel utilization with more flows. We suspect that the drop in throughput is because the emulated hosts (OpenVZ containers) are competing for shared buffer space in the kernel network stack. With a smaller share when the number of containers gets bigger, the TCP performance degrades.

In the second experiment, we divide the network between two compute nodes (splitting the model along the bottleneck link). The throughput for one emulated flow in this case is around 130 Mb/s, like we have observed before. As we increase the number of flows, the aggregate throughput increases slightly until 32 flows, after which we start to see a significant drop. Running the simulation on two compute nodes results in more the event processing power, the simulator is thus capable of handling more emulated flows than the sequential case. In the third experiment, we extend the dumbbell model by placing four core routers in a ring and connecting them using the bottleneck links. We attach the same number of emulated hosts to each core router. Each TCP flow was sent from an emulated host attached to one router to another emulated host attached to the next router in the ring. In this case, we spread the number of emulated flows evenly among the emulated hosts. Comparing with the scenarios with two compute nodes, the aggregate throughput for the four-node case is higher. Again we think is is due to the higher processing power of the parallel simulator. The throughput starts to drop for 128 flows (thats 32 VMs per compute node) as they compete for the shared buffer space.

4.7 Summary

In this chapter we introduce PrimoGENI, a system conceived to enable real-time simulation for the GENI federation. It aims to enable automatic and efficient realization of large-scale real-time simulation experiments, involving interactions between real and simulated components. In particular we achieve:

1. PrimoGENI is fully integrated in the Utah ProtoGENI site. Users can instantiate experiments that include simulated, emulated, and real components in that facility.
2. Slingshot allows experiment creation, editing, instantiation, and display.
3. A network experiment with thousands of hosts between two geographically distant locations achieving full connectivity between real and simulated hosts. This experiment has tenths of simulated hosts downloading a web page from a real Apache web server while Slingshot showed the data exchanges; real hosts can download simple web pages from our simulated servers.

CHAPTER 5

ON ENABLING SYMBIOTIC RELATIONSHIPS TOWARDS SCALABLE NETWORK EXPERIMENTS

In this chapter we present SymbioSim, a system that leverages the symbiotic relationship between simulation and emulation and, in doing so, is capable of allowing studies of real network applications and protocols under diverse large-scale network conditions.

5.1 Introduction

The ability to conduct high-fidelity high-performance network experiments is crucial for studying future complex network systems. Existing network, simulation, and emulation testbeds, all have drawbacks, either in scalability (for handling large systems), flexibility (for dynamically changing the network model, and injecting events to change the course of the experiments), realism (for reproducing important system and network effects), or performance (for supporting high-throughput high-capacity data transport):

- *Physical testbeds*, such as PlanetLab [PACR02] and WAIL [BL03]), provide a realistic operational environment for testing network applications. They can directly test real applications *in-situ* with the needed operational realism and possibly even with live network traffic. However, physical testbeds lack flexibility in testing applications under network conditions other than the prescribed setup of the physical platforms. Also, physical testbeds are in general also limited in scale.
- *Simulation testbeds*, such as *ns-2* [BEF$^+$00] and OPNET [OPN], are effective at capturing overall design issues, answering what-if questions, and revealing complex system behaviors (such as multi-scale interactions, self-organizing characteristics, and emergent phenomena). Parallel simulation has also demonstrated its capability of dealing with large-scale detailed models by harnessing the collective power of parallel computing platforms. However, simulation often lacks a certain

level of realism, reproducing realistic network traffic and operational conditions in simulation remains elusive.

- *Emulation testbeds*, such as ModelNet [VYW$^+$02] and EmuLab [WLS$^+$02], provide a good balance between flexibility and accuracy, whereas real implementations of the test applications can run directly in a native operating environment while the systems mimic the network behavior between the application instances. The capability of the emulation testbeds, however, is constrained by the physical limitations of the testbeds, such as the scale of the underlying platform, the processing speed, the bandwidth and latency of the supporting hardware. Furthermore, significant effort is needed if complex events need to be inserted in an experiment.

A testbed capable of performing large-scale experiments over diverse network scenarios and network conditions, maintaining accurate representation of the target applications and network transactions, and supporting high-throughput high-capacity network traffic is needed. We already explored in chapter 3 the use of time dilation to achieve a large-scale settings. Also, in chapter 4 we achieved large-scale experiments by inserting a large-scale simulator in th GENI federation. Here, we explore a new dimension to achieve large-scale network experiments.

Both simulation and emulation provide good flexibility and reproducibility necessary for conducting experiments under various network conditions. However, we observe that their expected capabilities differ significantly. Simulations are typically used to construct high-level network scenarios with protocols that may not be fully developed (such as those at the physical and link layers); and a full-scale simulation is often desirable in order to obtain “the big picture”, especially when a complete understanding of the system’s complex behavior is absent. In general, simulation offers great flexibility and scalability, but cannot provide the detailed behavior of the network and the execution environment. In contrast, emulation offers the important capability of being able to execute code directly on a real system that accepts application data as input and produces detailed response as output. Furthermore, it provides the operational realism, but cannot represent all elements of a large-scale network experiment due to resource limitations.

To allow high-fidelity high-performance large-scale experiments, we propose to combine the advantages of both simulation and emulation systems by leveraging a symbiotic relationship between them. The simulation system runs large-scale network models with network topology and protocols for a close representation of the global-scale network traffic behavior. The emulation system runs real hosts where real applications and protocols run unmodified in a real operating environment with the designated operating system, network stack, and software tool chain.

Simulation and emulation form a symbiotic relationship so that both benefit from each other. The simulation system benefits from the emulation system by incorporating the behavior of real network traffic generated by the unmodified software running on the real hosts. The emulation system benefits from the simulation system as it receives up-to-date information of the global virtual network behavior and traffic conditions, and uses it to calibrate the emulation model. As a result, the proposed system allows experimenters to study and analyze applications by embedding them seamlessly in large-scale network settings.

5.2 SymbioSim

In this section we present *SymbioSim*, a system that leverages the symbiotic relationship between simulation and emulation and, in doing so, is capable of allowing studies of real network applications and protocols under diverse large-scale network conditions.

In SymbioSim, an experiment consists of a virtual network with a detailed specification of the network topology, potentially connecting a large number of hosts and routers, running various network protocols and distributed applications. While it is expected that most of these protocols and applications are simulated (either due to resource conservation, or for better flexibility), some of the hosts can be designated to run unmodified distributed applications and real implementations of network protocols on real machines. We name the unmodified applications as *emulated applications*; we name the real machines corresponding to the end-hosts on the virtual network that run the emulated applications as *emulated hosts*; and we name the network traffic between the emulated hosts as *emulated traffic* or *emulated flows*, to distinguish them from the simulated counterparts. Fig. 5.1 shows a virtual topology in which four hosts are designated to be emulated hosts, i.e., they are expected to run as individual physical or virtual machines. All the other end-hosts are simulated. During the experiment, the real applications at the emulated hosts may engage in communication over the simulated network; in this example, they establish three emulated flows, one between H1 and H2, the second between H2 and H3, and the third one between H2 and H4.

A schematic view of the SymbioSim architecture is shown in Fig. 5.2, which consists of a simulation and an emulation system in a closed loop (from hereon, we also refer to the emulation system as *physical system*). The simulation system runs the full-scale network model with a detailed specification of the network topology, traffic and protocols. The full-scale network model can be partitioned and mapped onto a high-end computing platform for parallel simulation in order to capture the detailed transactions of the large-scale network in real time. The emulation system consists of a set of physical or virtual machines, each representing an emulated host designated to run real applications; they are connected via a set of *pipes*, which are responsible for modulating the network traffic between the emulated hosts so that they experience the same network delays and losses as if the they were connected via the virtual network represented in full scale at the simulation system. The network topology that runs in the emulation system can be a *downscaled* version of the full-scale topology that runs in the simulation subsystem.

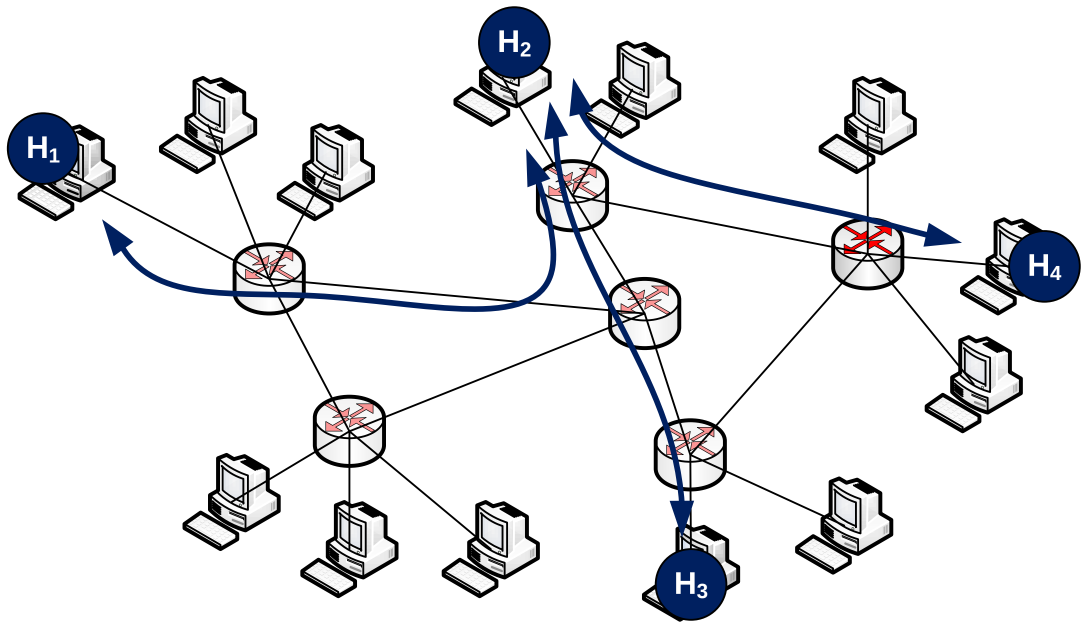


Figure 5.1: A network experiment consists of simulated and emulated components

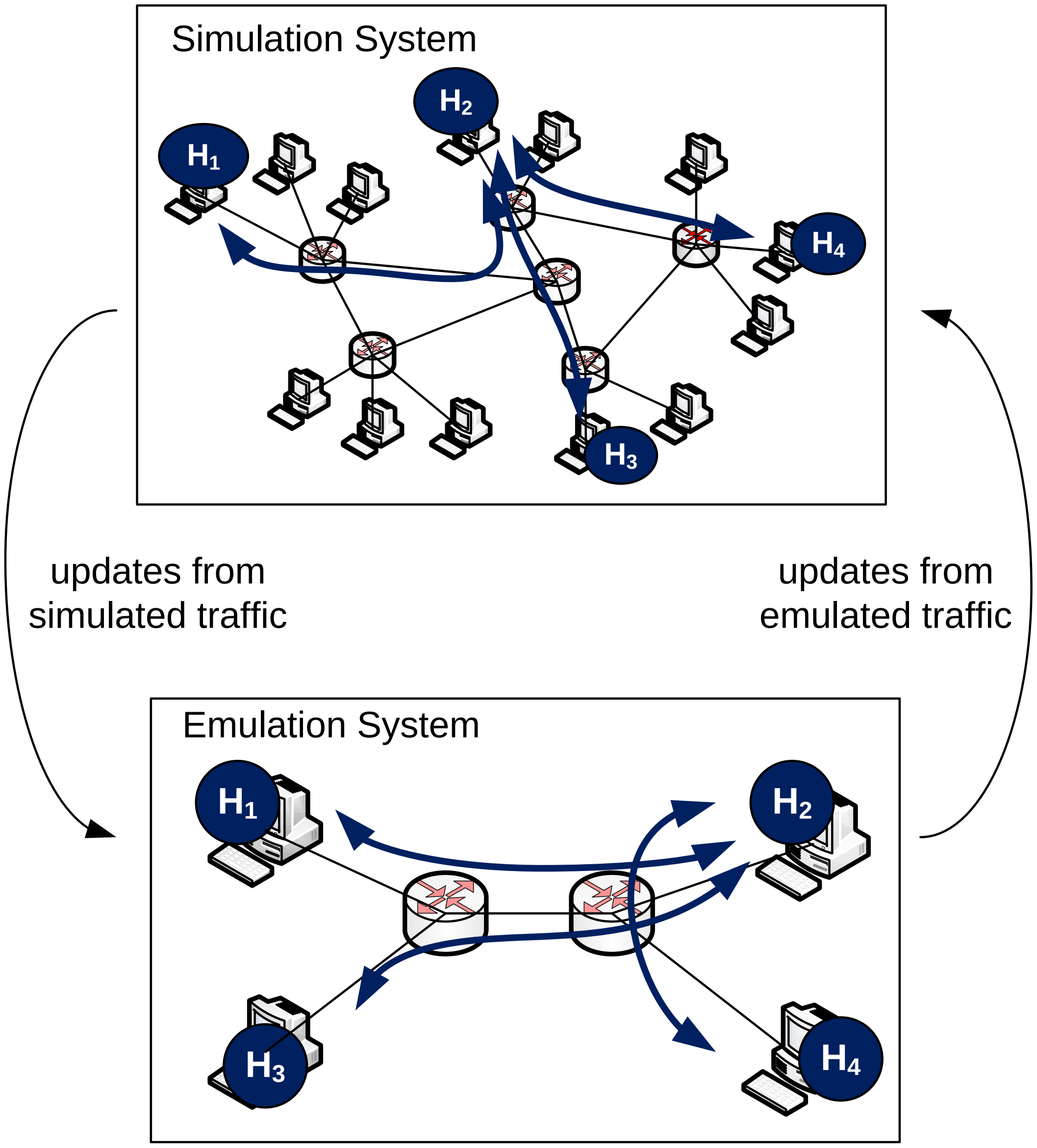


Figure 5.2: The SymbioSim architecture

The emulated traffic (i.e., the network traffic between emulated hosts) affects the simulated traffic (i.e., the traffic between simulated network entities), and vice versa, as they are mixed on the virtual network and compete for the bandwidth of common links. The simulation system only needs to periodically update the emulation system with the queuing effects of the simulated traffic on the emulated traffic at the corresponding pipes. Similarly, the emulation system only needs to periodically update the simulation system about the state of the emulated flows between the real applications, so that the simulation system can correctly reproduce the global traffic conditions on the virtual network.

SymbioSim is composed of the subsystems shown in Fig. 5.3 and described in the following paragraphs:

1. *Simulator*. The simulator performs what-if simulations in packet-level detail with background traffic generated by simulated hosts. It receives emulated traffic *meta-data* from the *Gateway* and regenerates the corresponding traffic in simulation using this data. Also, it exports network queues statistics (*raw updates* to the *Simulator - External Decision Support System* or *SimEDSS* hereafter. Raw updates include delay, throughput, and packet loss from each interface (modeled as a queue) in simulation.

2. *Simulator - External Decision Support System (SimEDSS)*. Gets *raw updates* coming from the simulator as input. In response, it computes and sends *updates* to calibrate the emulation system. In this context, *update* means that this module provides the emulation system with a set of parameters to calibrate the downscaled network topology running in it.

3. *Actuator*. Receives the *updates* from the SimEDSS and executes them in the emulation system, i.e., it changes the network model running in the emulation system.

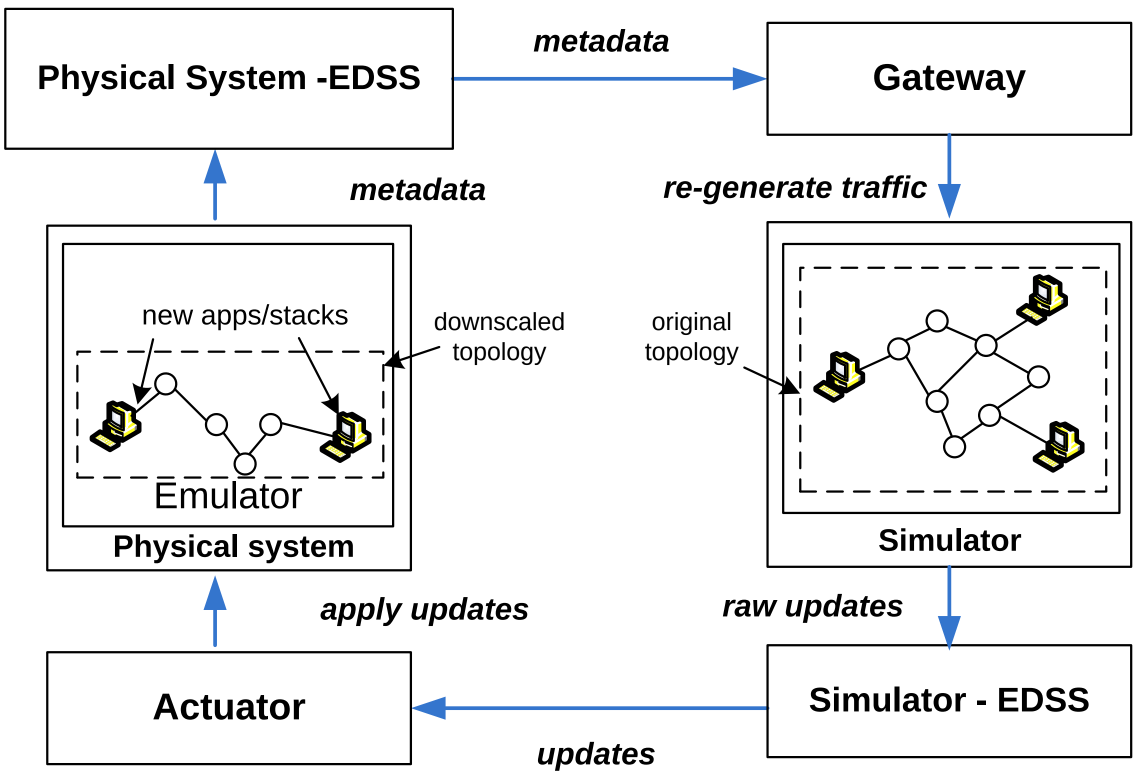


Figure 5.3: Proposed closed-loop symbiotic simulation system

4. *Emulation(Physical) System*. Real network topology where real applications run. It is calibrated in real-time by the *updates* sent from the simulator. This subsystem is composed of the network emulator, the real applications running on possibly virtual machines, and the supporting processes that allow updating the network model.
5. *Physical System - External Decision Support System (PhyEDSS)*. Receives the traffic measurements coming from emulated applications, and sends the corresponding *metadata* to the *Gateway*.
6. *Gateway*. Receives emulated traffic *metadata* from the PhyEDSS and injects it into the simulator.

The effectiveness of the proposed symbiotic system lies in its ability to efficiently and accurately modulate emulated traffic in accordance with the simulated large-scale network conditions. Furthermore, due to the interdependency of the simulation and emulation systems, the two systems must be effectively synchronized. In the subsequent sections, we address these specific problems:

- In section 5.3, we present our novel approach to emulate a network path, i.e., accurately represent the network effect of the simulated traffic on the network paths between the emulated hosts.
- In section 5.4, we present a novel topology downscaling technique conceived to decrease the amount of computational resources needed in the emulation system and reduce the synchronization overhead between the simulation and emulation systems. The downscaled emulation model is expected to be functionally equivalent to the full-scale simulation model in terms of conducting traffic between the emulated hosts.
- In section 5.5, we aim at reducing the overhead for the emulation system to update the simulation system, and present a technique for capturing the state of the emulated flows. Rather than measuring and transferring the emulated traffic statistics at individual network devices, we collect the network demand due to the real applications at the transport layer. The emulated traffic is then reproduced using the same implementation of the transport-layer protocols in simulation. We also present in section 5.5 a technique for reducing the computational overhead for capturing network interfaces' statistics in the simulation system.

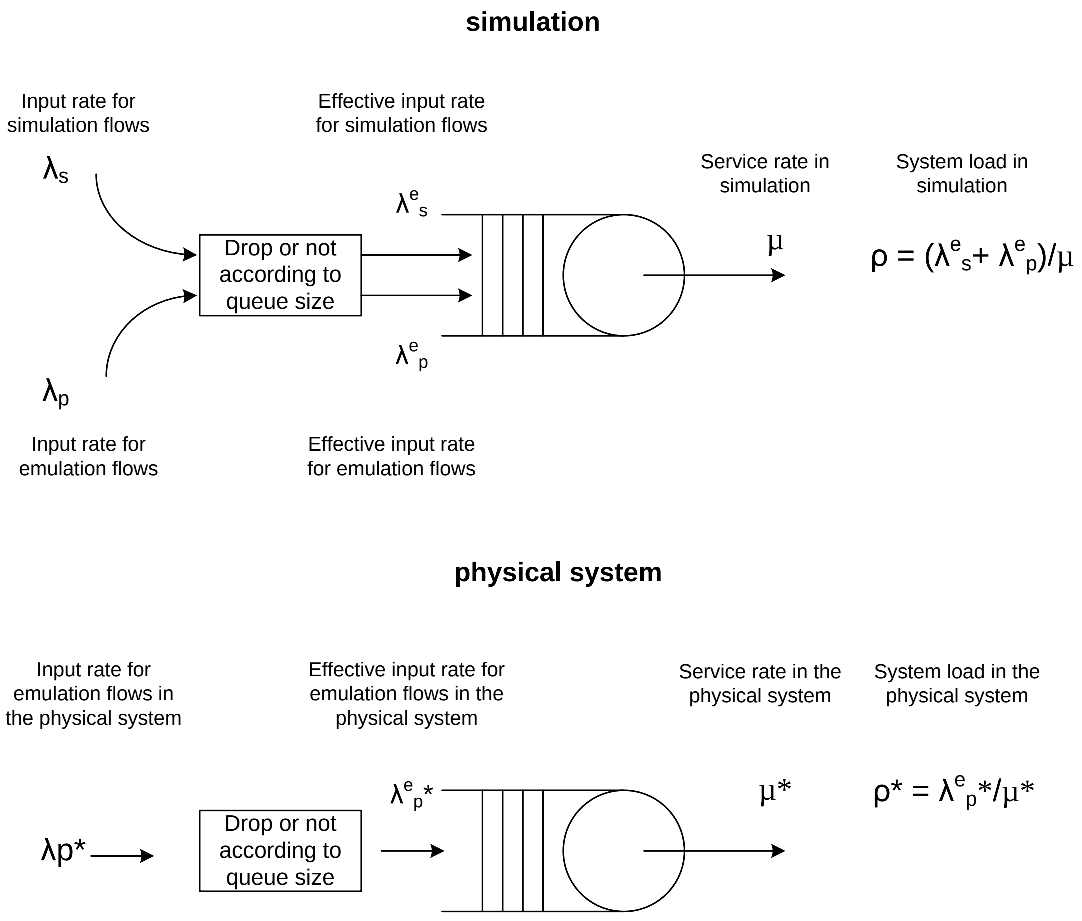
5.3 Emulating a Network Path

The effectiveness of SymbioSim depends on how accurately it represents the network effect of the simulated traffic on the network paths between the emulated hosts. We refer to this action as *emulating a network path*.

Many approaches have been used in the past to model and emulate network paths. The most intuitive is to use multiple instances of a link emulator, e.g., Dummynet [Riz97], thus creating a series of links (modeled as queues). However, emulating a network path in this way requires to have accurate values for queue size, and background traffic for each link for every time instant, which is very difficult. Another approach, proposed by Sanaga et al. [SDRL09], consists on approximating a path by modeling the capacity and available bandwidth separately. In order to mimic the portion of the bandwidth that is being used in a path, Sanaga et al. emulate a particular level of traffic using non-responsive, constant-bit-rate traffic. Even though this approach gives a good approximation for modeling a path when no detailed information about the characteristics of traffic is available (an excellent approach for modeling Internet paths), it fails to provide accurate results. Modeling a level of traffic using non-responsive traffic is obviously unfair for TCP flows.

We here propose to emulate a network path using a M/D/1 queuing model. We adopt the M/D/1 queue model because of its simplicity and easy to manipulate closed forms. Network paths in the virtual topology in simulation are emulated as queues (pipes) in the emulation system. The service rate, loss, and delay of these queues are calibrated every time interval T to offer the traffic of emulated applications the same delay, loss, and throughput as those experimented by emulated traffic in simulation. In this way, the details of background traffic do not have to be known since we model only the *effects* of it. Also, our approach does not entail that we emulate a particular level of traffic. To increase the accuracy of our model, we use real time adjustments that make it accurate for any kind of traffic of any intensity.

Fig. 5.4 depicts simulation and emulation systems showing the parameters we consider for emulating a network path. In our model, we distinguish between simulated and emulated flows and model them separately. We begin at imposing that emulated traffic in simulation experiments the same delay as emulated traffic in the emulation system.



simulation
Input rate for simulation flows
Effective input rate for simulation flows
Service rate in simulation
System load in simulation
λ_s
λ^e_s
Drop or not according to queue size
μ
$\rho = (\lambda^e_s + \lambda^e_p)/\mu$
λ^e_p
λ_p
Input rate for emulation flows
Effective input rate for emulation flows
physical system
Input rate for emulation flows in the physical system
Effective input rate for emulation flows in the physical system
Service rate in the physical system
System load in the physical system
λp^*
Drop or not according to queue size
$\lambda^e_p{}^*$
μ^*
$\rho^* = \lambda^e_p{}^*/\mu^*$

Figure 5.4: Modeling both systems

The queue length (L) in the M/D/1 model is a function of the load in the system (ρ) and is given by the expression shown below:

$$L = \rho + \frac{\rho^2}{2(1-\rho)} \tag{5.1}$$

Little's law states that the average time a packet spends in the system (W) is a function of the queue length and the arrival rate (λ). It is given by:

$$W = \frac{L}{\lambda} \tag{5.2}$$

Then, we force that both systems impose the same delay in their packets using Eq. 5.2:

$$\frac{L_1}{\lambda_s^e + \lambda_p^e} = \frac{L_2}{\lambda_p^{e*}} \tag{5.3}$$

Replacing L_1 and L_2 by their respective expressions using Eq. 5.2, we get,

$$\frac{\rho_1 + \frac{\rho_1^2}{2(1-\rho_1)}}{\lambda_s^e + \lambda_p^e} = \frac{\rho_2 + \frac{\rho_2^2}{2(1-\rho_2)}}{\lambda_p^{e*}} \tag{5.4}$$

Eq. 5.4 is useful if we estimate the value of W_1 (the delay in simulation) from L_1. However, in simulation we can *measure* W_1 precisely and avoid estimation errors. Again, forcing both systems to have the same waiting time in the system,

$$W_1 = \frac{\rho_2 + \frac{\rho_2^2}{2(1-\rho_2)}}{\lambda_p^{e*}} \tag{5.5}$$

Solving Eq. 5.5 for the load in the emulation system (ρ_2) we get the following expression:

$$\rho_2 = 1 + W_1 * \lambda_p^{e*} - \sqrt{1 + W_1^2 * (\lambda_p^{e*})^2} \tag{5.6}$$

In SymbioSim, we aim at having $\lambda_p^e = \lambda_p^{e*}$ since we precisely measure the packet loss for emulated flows in simulation and impose the same rate in real traffic. Hence, we have,

$$\rho_2 = 1 + W_1 * \lambda_p^e - \sqrt{1 + W_1^2 * (\lambda_p^e)^2} \tag{5.7}$$

In order to get the service rate in the emulation system from our queuing model (sr_Q) we use:

$$\rho = \frac{\lambda}{\mu} \tag{5.8}$$

Then,

$$sr_Q = \frac{\lambda_p^e}{1 + W_1 * \lambda_p^e - \sqrt{1 + W_1^2 * (\lambda_p^e)^2}} \tag{5.9}$$

Using Eq. 5.9, we calibrate, in real time, the emulation system's queues from measurements in the simulator every period *T*. This service rate, guarantees that both systems impose the *same delay* to emulated traffic in both simulation and emulation systems, when the input's inter-arrival times are exponentially distributed. We show empirically in forthcoming sections that this model provides a good approximation even when the input's inter-arrival times are not exponentially distributed. We also impose, when getting equation 5.7, that emulated traffic in both simulation and emulation systems experiment same packet drop rate. Then, if the same number of packets in both systems are ready for being transmitted and both systems impose the same waiting time, then it is also expected the throughput to be the same in both systems, thus achieving our goal.

The above presented model can be readily extended to emulate a whole path in simulation as a single queue in the physical system. To that end, W_1 in equation 5.9 is the delay of the whole path and λ_p^e is the throughput that emulated traffic is achieving in the bottleneck link of that path. For the packet drop in the whole path we use the following expression:

$$P_{path}(loss) = \sum_{i=1}^{n} p_i \prod_{j=1}^{i-1} (1 - p_j) \tag{5.10}$$

In the above formula, $P_{path}(loss)$ is the configured drop probability in the path's corresponding pipe in the emulation system and p_i is the drop probability in simulated link i that composes the path.

5.3.1 Queuing Model Improvement

When the congestion level in simulation is low and thus the queuing delay is close to zero for a given link or path being emulated, the model presented in the previous section provides good results. Nevertheless, our model fails to provide accurate results in presence of congestion in the link being modeled. The reasons are: 1) The model assumes that the physical system can reach steady state within a period T, and 2) The model does not consider the current queue level in the emulation system.

The error comes in two forms: the delay in simulation (W_1) is greater or less than that delay in the emulation system (W_2). Empirical evidence suggests that the delay in the physical system tends to be less than that observed in simulation when the path being emulated has a congested link. In order to overcome this mismatch, we *synchronize* the queuing delay in the emulation system to be the same as current one in the simulator. In this case, we say that the emulation system *inherits* the queue size from the simulator. To mimic that, we set that at the end of every period T, all packets in the corresponding pipe, in the emulation system, to have an extra delay given by:

$$extra\ delay\ in\ emulation\ system(t) = W_1(t) - W_2(t) \tag{5.11}$$

Equation 5.11 states that the *extra delay* to be set in the emulation system's corresponding pipe, at time t, is a function of the difference between the delay in simulation at time t and the delay in the emulation system at the same instant of time. Doing this corrects the effects not considered using the original model.

As a consequence of the delay adjustment given by equation 5.11, there are times when W_2 is greater than W_1. In this case, the queue in the emulation system has more packets than the corresponding path has in all its queues in simulation, thus producing more delay. Consequently, we "get rid" of the excess number of packets in the emulation system. First, we estimate the number of packets that cause the delay difference using the service rate at time t in the emulation system ($sr(t)$):

$$N_{excess} = sr(t) * (W_2(t) - W_1(t)) \tag{5.12}$$

Then, we compute the extra service rate in the emulation system (sr_{extra}) that we should have to "get rid of", i.e., transmit, N_{excess} in a time interval T:

$$sr_{extra} = \frac{N_{excess}}{T} = \frac{sr(t) * (W_2(t) - W_1(t))}{T} \tag{5.13}$$

Then, we configure the service rate in the emulation system as the sum of the service rate computed from the queuing model using equation 5.9 (sr_Q) and the extra service rate:

$$\mu^* = sr_Q + sr_{extra} \tag{5.14}$$

In this way, we constantly "synchronize" the emulation system to simulation every period T. On the one hand, when no congestion exists in the path being emulated, the service rate outputted from equation 5.9 is good enough and small adjustments are expected from equations 5.13 and 5.11. On the other hand, when paths being emulated are congested, equations 5.13 and 5.11 provide the adjustments we need to faithfully emulate network paths.

5.4 Topology Downscaling

One major disadvantage of network emulators is scalability: every host in the topology is directly mapped to a physical host and every link to a pair of link emulator instances, e.g., Dummynet [Riz97]. This fact severely limits the amount of network entities that can be used in an experiment.

Within the symbiotic scenario described in previous sections, if we used an emulator such as Emulab to provide the underlying infrastructure to host the emulation system, we would be using closely as many hosts as those used if the network topology were instantiated directly in Emulab, without SymbioSim. Even worse, the simulator and the symbiotic sub-systems need processing capability that would increase even more the number of physical nodes needed. In order to overcome this problem, and therefore increase to larger sizes, we propose a novel approach for topology downscaling.

Papadopoulos et al. proposed DSCALEd [PPG06], a method that aims at downscaling a network topology by retaining only the congested links and compensating for the ignored links adding delays. The big issue of DSCALEd, is that the bottleneck has to be known in advance in order to downscale the topology, a luxury that we cannot allow if new protocols or applications are evaluated.

Inspired by previous efforts, we here propose a new downscaling method that we call *DSCALE^R*, which stands for *downscale in real time*. The actual downscaling of the original topology indeed happens before the experiment starts, but the core of it is based on real time calibration of the downscaled model. *DSCALE^R* is based on the fact that simulated hosts are not instantiated in the emulation system and thus the links that carry only simulated traffic can be extracted off the original topology thus yielding the *downscaled topology*, i.e., the topology instantiated in the emulation system. *DSCALE^R* lays upon the following two assumptions:

1. Routing does not change during the simulation. Thus, we currently do not support routing experiments.

2. Traffic main characteristics change little within each within a specific time interval [ZD01]. In [ZD01], it is concluded that for packet loss, delay and throughput, one can generally rely on constancy on at least the time scale of minutes.

DSCALE^R consists of three components: 1) the first takes the original topology as input and yields a *pruned topology*, which contains only those links that may be traversed by emulated flows, 2) the second produces the *downscaled topology* from the output of the previous step, and 3) based on the approach described in the previous section, queues in the downscaled topology are calibrated in real time to reflect present network conditions in simulation.

In order to get the pruned topology, a simple experiment is setup where are all emulated hosts in simulation send a packet to all other emulated hosts in the virtual topology. Hereon we regard packets originated from and destined to virtual hosts as *virtual packets*. All links which have been traversed by these packets are included into the pruned topology. The output is the graph G_p in the form of an adjacency matrix (A_p).

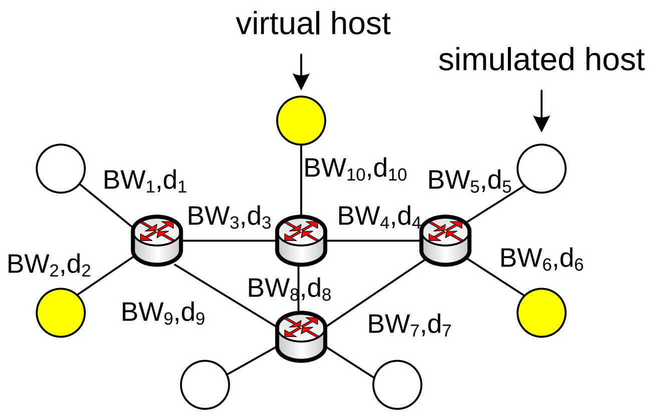
The second component of *DSCALE*^{R} further downscales the pruned topology. Any of the remaining links can be a bottleneck link and thus cannot be extracted off the pruned topology at first instance. However, we can still reduce a set of links (a path) in simulation into a single queue to be instantiated in the emulation system if, starting from the first vertex that composes this path, all other vertices (hosts or routers) have only one child, i.e., all connect to another of degree less than or equal to two. We can picture this path as having all links forming an straight line. On the contrary, we cannot abstract a set of links attached to routers of degree greater than two because we would be *artificially changing* the original topology and consequently some TCP/UDP flows would end up sharing a link in the downscaled topology when they actually do not do so in the original model. Of course, this would dramatically change the results obtained from the downscaled network.

Our algorithm gets the pruned topology in the form of a graph G_p, composed of the set of vertices V_p and edges E_p, as input and outputs the final downscaled topology to be instantiated in the emulation system (G_d). Our algorithm uses a combination of BFS and DFS to visit all vertices at most once. DFS is used instead of BFS whenever a vertex with a degree of two is found; which indicates that a set of edges(links) will be downscaled to a single one. After all the adjacent edges to vertices of degree two are discovered via DFS, only one edge is added to V_d (the set of vertices in the downscaled topology). If an edge is attached to a vertex of degree greater than two, then this edge is added directly to V_d.

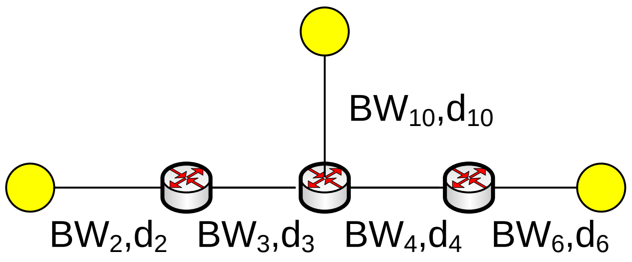
Fig. 5.5 shows an example of how the first two components of $DSCALE_R$ operate over a specific topology. Fig. 5.5i) shows the original topology, with a label for each link's bandwidth and delay. On Fig. 5.5ii), the pruned topology is shown containing only those links necessary for emulated hosts to send packets to each other. The downscaled topology is shown in Fig. 5.5iii). Notice how BW_2 and BW_3 were downscaled to BW_1' and d_2 and d_3 were to d_1'. At run time, $DSCALE^R$'s real time component computes μ^*, additional delay (if any), and measures packet loss for each link represented in the emulation system and passes the computed values to the Actuator. The Actuator composes commands to impose the current simulation traffic state using the passed values.

5.5 Reducing Synchronization Overhead

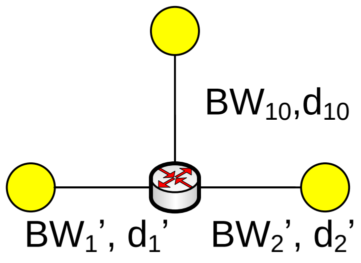
In this section, we present two techniques for reducing the overhead originated by the symbiotic relationship imposed at both simulation and emulation systems.



(a) Original topology



(b) Pruned topology with only those links used by virtual hosts to communicate between each other



(c) Downscaled topology with some links abstracted as a single link.

Figure 5.5: Downscaling the network topology

5.5.1 Limiting Incoming Traffic

Our proposed scheme is based in that the traffic from real (emulated), distributed applications can be regenerated in the simulator so that the simulator mingles emulated and simulated traffic and calibrate the emulation system. For this to happen, the traffic coming from real applications has to be injected into the simulator in real time. Previous approaches, such as nse [Fal99], the ns-2 emulator [BEF$^+$00], and PRIME [LMVH07b], insert every packet generated from real applications into the simulator. As a direct consequence, while the output generated is accurate (see [ELL09] or [EL10a] for comparison between simulation and real systems), the scalability is seriously compromised by the maximum rate at which packets can be inserted into simulation.

Given that ns-2 runs on a single node, the scale of the network topology and the amount of traffic that can be simulated in real time is very limited. PRIME with its distributed simulation capabilities, still suffers from reduced rate between real applications and the simulator [EL10a].

SymbioSim addresses this issue by not transferring entire packets generated by emulated applications to the simulator. Instead, we capture only an integer that corresponds to the amount of bytes that are transmitted from the applications to the transport layer in the emulation system. Then, this metadata (hereon referred to as *appBytes*) is sent to the simulator. Inside the simulator, we have one simulated host for each real application in the emulation system. These simulated hosts, via the symbiotic infrastructure (see sec. 5.6 for implementation details), receive appBytes. Then, they send appBytes number of bytes to the transport layer beneath them on behalf of real applications. Upon receiving a transmission request, our realistic simulated TCP implementation transmits appBytes using the congestion control algorithm that matches that used by the corresponding emulated

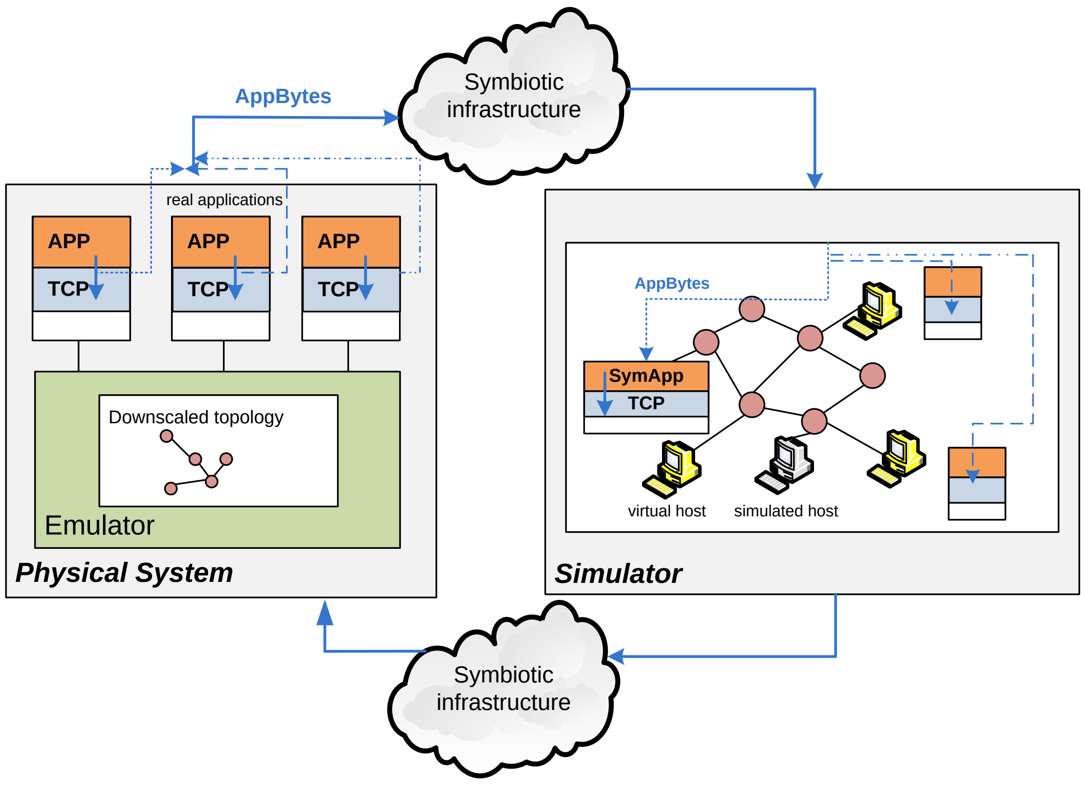


Figure 5.6: Limiting incoming traffic

application in the emulation system. In this way, we can scale better by series never sending payload data between the emulation system and the simulator.

We depict emulated hosts in simulation in Fig. 5.6. These hosts have in their stack a *symbiotic application*, which upon receiving the amount of bytes sent from emulated applications to TCP (appBytes), generate the corresponding traffic. For this approach to work accurately, our simulated TCP implementation has to be able to behave very similarly to the TCP algorithm running in the emulation system. We ported Linux TCP and tested its behavior in previous papers [ELL09][EL10a].

5.5.2 Generating Updates

Calibrating the emulation system entails computing λ_p^e as input to equation 5.9 for every link in the original topology running in the simulator, every period of time T. Then, the simulator can update the emulation system with the service rates values μ^* for each queue. Doing this adds some overhead to the simulator thus decreasing the number of events that can be executed per unit of time and consequently preventing our platform to scale to larger sizes. Going into the details, computing λ_p^e involves counting the number of bytes originated from virtual hosts for every interface inside the model. This brute force approach is of order $O(L)$; where L is the number of links in the network model which have a corresponding queue in the emulation system.

In order to decrease the number of computations, we use sampling. We follow a similar technique to that proposed in [CPZ03] and [CPZ02]. The brute force approach entails that during every period of time T we count the number of virtual packets that traverse every link i (*m*). Then the throughput attributed to virtual hosts (λ_p^e) in link i is $\sum_{j=1}^{m} X_j$; where X is a random variable denoting the packet size. Instead of capturing the size of every packet that traverses each link, our objective is to sample n out of those m packets so that the estimated throughput would be given by $\hat{\lambda_p^e} = \frac{m}{n}\sum_{j=1}^{n} X_j$. Our target is to bound the error of $\hat{\lambda_p^e}$:

$$P\left[|\frac{\hat{\lambda_p^e} - \lambda_p^e}{\lambda_p^e}| > \varepsilon \right] \leq \eta \qquad (5.15)$$

Equation 5.15 enforces that the probability that the relative error be greater than *epsilon* to be less than η. Using the central limit theorem for random samples [BL96], Choi et al. found an expression for the number of samples required to meet the boundaries imposed by equation 5.15:

$$n \geq n^* = \left(\frac{\Phi^{-1}(1-\frac{\eta}{2})}{\varepsilon} \frac{\rho}{\mu} \right)^2 \tag{5.16}$$

In the above equation, ρ and μ are the standard deviation and the mean of the packet size seen at that interface. Also, Φ is the cumulative distribution function (c.d.f.) of the standard normal distribution ($N(0,1)$). The optimal sampling probability is given by:

$$p^* = \frac{n^*}{m} \tag{5.17}$$

From the above equations, the sampling probability must be at least p^* in order to achieve a predefined accuracy determined by the pair (ε, η). However, we still have to know the values for ρ and μ for the packet size corresponding to a period of time T in order to compute n^* using equation 5.16. For each period of time T we measure ρ and μ an and keep these values for each period. Then, we use autoregression methods for estimating the future ρ and μ values so that we can compute from the above equations the optimal sampling probability for a bound error.

We conduct a simple experiment in which we measure the overhead of running the brute force approach and the sampling-autoregression one for computing the total simulation traffic at each interface. A simple topology with one host at each end connected via a configurable number of routers are used for this purpose. As the number of routers between the two hosts grows in size, the wall-clock time needed by the simulator for completing a large file transfer is measured and plotted in the first plot of Fig. 5.7. Each point corresponds to the average of 20 runs with its corresponding 95% error bars. As expected, as the size grows, the method that uses sampling and autoregression methods outperforms the brute force approach. Furthermore, the second plot of Fig. 5.7 shows the number of events achievable as the number pf routers grows for the simulator with no symbiotic code, the brute force, and the sampling-autoregression method. Again, for roughly more than 40 routers, the samling-autoregression method is computationally less intensive that the brute force method.

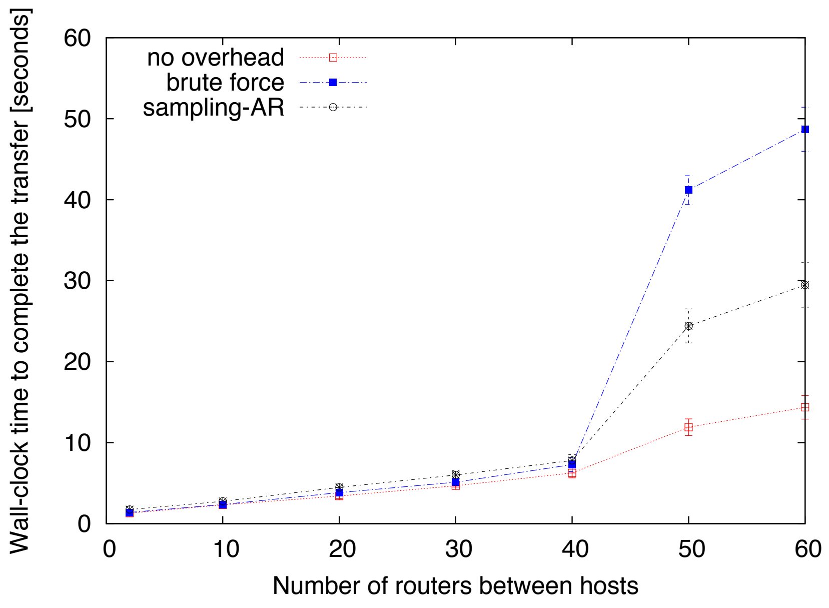
In order to test our approach to correctly estimate λ_p^e we set up a simple experiment using two hosts connected through a router. The sender increases its transmission rate at 5, 15, and 25 seconds after the experiment starts as depicted in Fig. 5.8. The error boundary for the first plot of Fig. 5.8 is 50%, 25% for the second, and 10% for the last plot. Notice that as the error boundary decreases the better is our approach to correctly estimate λ_p^e, which is indicated by the AR curve better matching the one produced using the brute-force approach. Of course, the more error we tolerate the less packets we have to sample and consequently the more events we can process per unit of time. Finally, we show in Fig. 5.9 how the *average sampling* probability decreases as ε increases. For example, if we are in position to tolerate a relative error in estimation of 10%, then we would only have to sample approximately 50% of the packets.

5.6 Implementation

In this section, we describe our fully simulated and physical implementations of SymbioSim. The simulated version is needed because it provides great flexibility for rapidly changing the setups. Using the real implementation, we can test real applications.

5.6.1 Simulated SymbioSim

We implemented a reduced version of SymbioSim fully in simulation for validation and rapid prototyping purposes. This version has a *simulator* sub-system composed of a sender and a receiver connected through a series of queues. The simulated emulation system has a single queue which is calibrated every period of time T. The simulated SimEDSS collects all the necessary statistics from the simulator sub-system every pe-



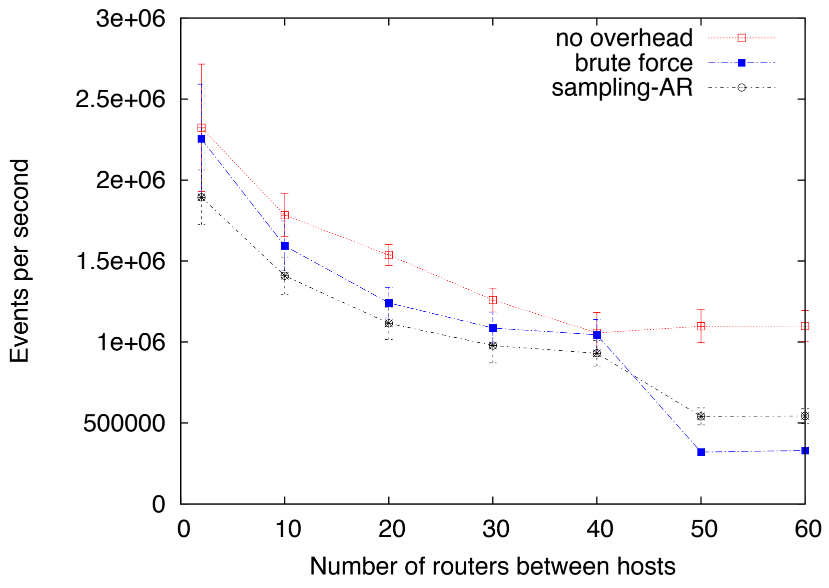


Figure 5.7: Performance comparison between the system without extensions, brute-force approach, and sampling-autoregression approaches

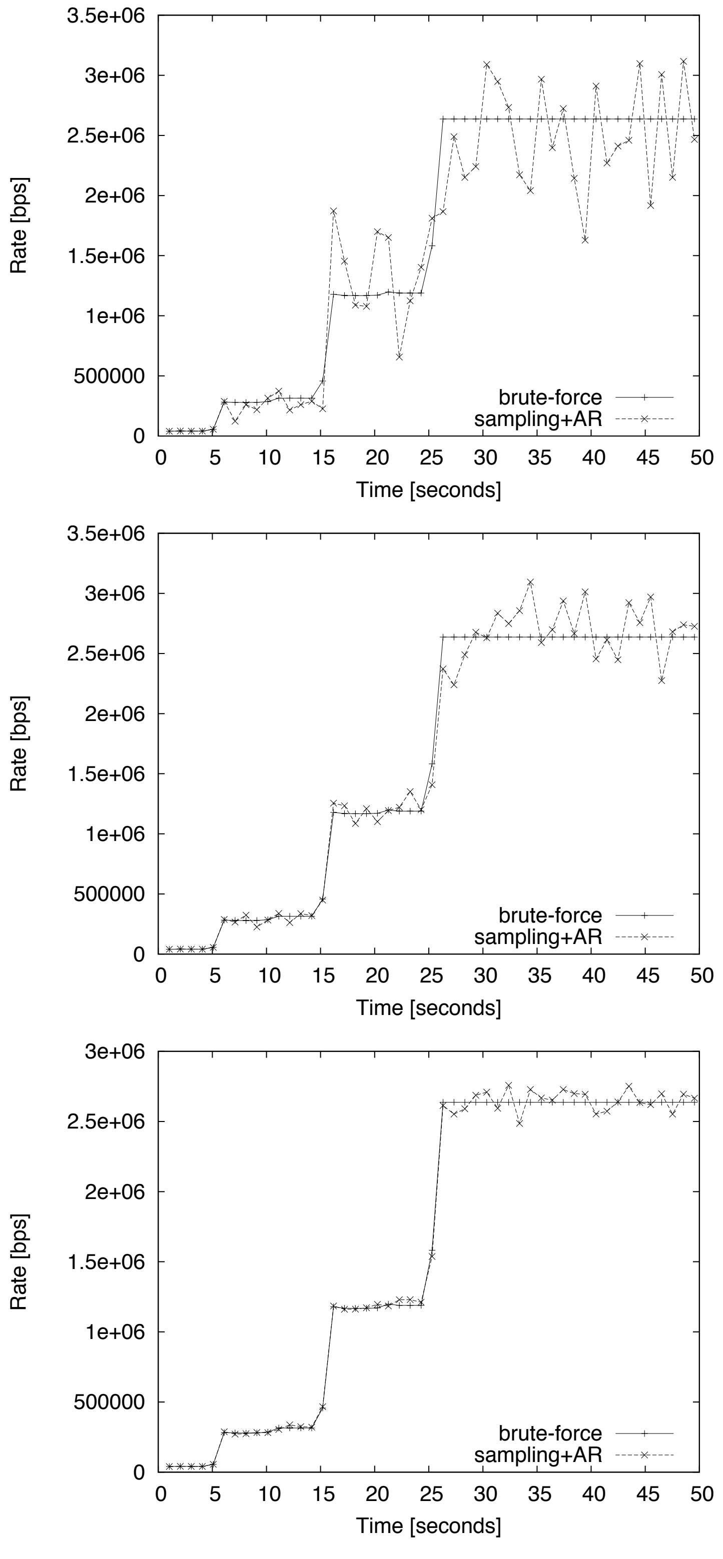


Figure 5.8: Transmission rate computed for various error boundaries

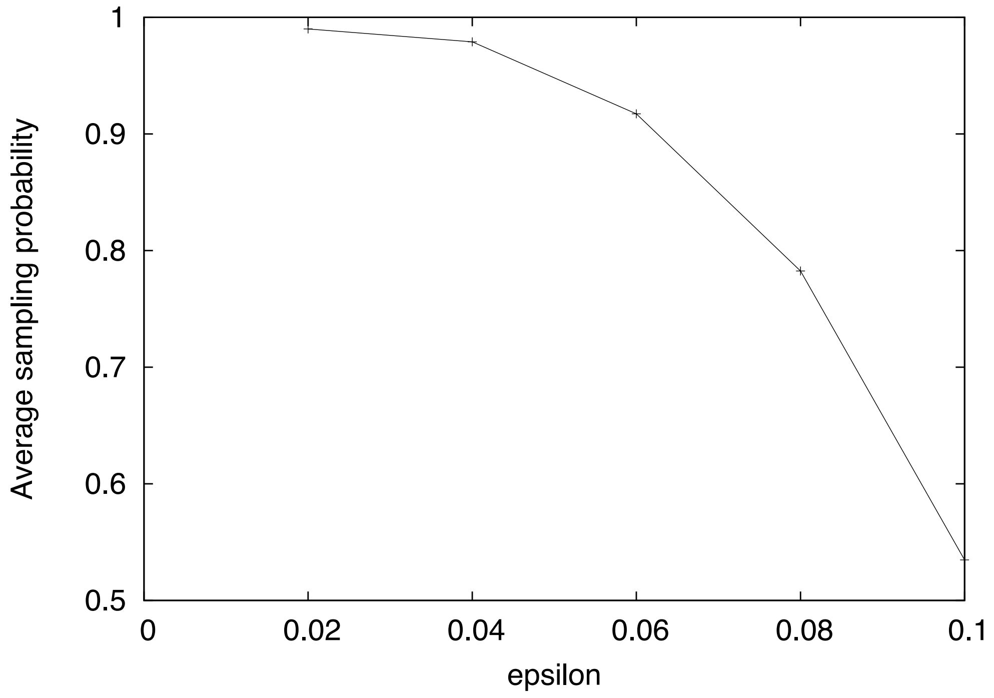


Figure 5.9: Sampling probability as as function of ε

riod of time T, and updates the simulated emulation system with the computed service rates, extra delay, and measured drop rates. To compute the service and drop rates, the SimEDSS collects statistics for a configurable interval of time WIN.

Both the simulator and the simulated emulation sub-systems can be configured to have constant, and exponentially distributed inter-arrival times. Furthermore, our system can replay traces obtained from using TCPDUMP [TCP] over a network interface.

5.6.2 Prototype System

The current current SymbioSim implementation is depicted in Fig. 5.10. As shown, the SimEDSS and the PRIME network simulator may be co-located, which improves the timeliness of the arrival of the *statistics* exported from the simulator. PRIME exports these statistics every configurable period of time without any pre-processing. A set of

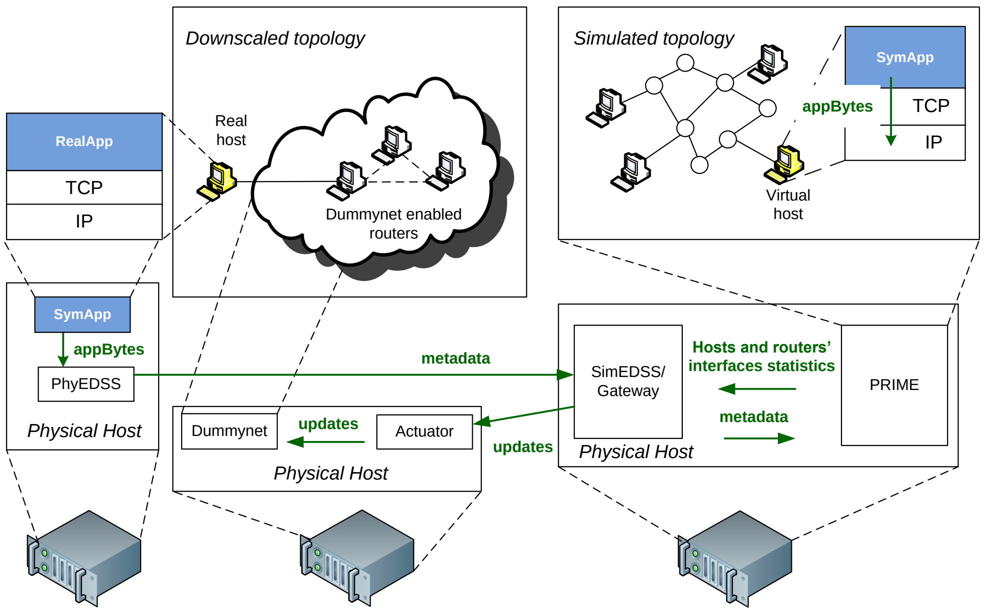


Figure 5.10: SymbioSim's current implementation

statistics is sent per each interface in the original topology which has a corresponding one in the downscaled model.

For single links emulated by a single queue, the SimEDSS computes a command and dispatches it immediately. Instead, when a whole path is emulated as a single queue, the SimEDSS gathers statistics for all links that compose the path, and then computes and update. After an update is computed, the SimEDSS dispatches it to the corresponding *delay node*, i.e., a physical node that runs Dummynet instances to emulate links. Currently, we use Emulab to host our emulated system because of the easiness for instantiating network topologies with routers and end-hosts inter-connected via Dummynet. Every Dummynet instance emulates a queue in our emulated system that is calibrated in real time by the SimEDSS. Emulab may map many links (a pair of queues or *pipes*) to a single physical node. Consequently, the SimEDSS has to be aware of the IP address of the target delay node for a corresponding pipe (identified by a *pipe id*). When the actuator receives the update, it extracts the $< pipe\ id, extra\ delay, service\ rate, drop\ probability >$ and constructs a *command* that is immediately executed.

The PhyEDSS is in essence a TCP server bootstrapped at each physical machine hosting real applications. It listens for all *appBytes* updates coming from real hosts. Currently, we change the source code of applications so that every time they send a chunk of bytes to TCP, they also do so to the PhyEDSS. The PhyEDSS sends this metadata to the Gateway, which inserts these values to the simulator. Upon receiving the metadata, the simulator has appropriate virtual hosts re-generate traffic accordingly.

5.7 Validation

In this section, we present the results of our experiments to assess the accuracy of our SymbioSim system.

5.7.1 Single Link Validation

The first set of experiments aims to examine whether the M/D/1 model can be used as a valid approximation for general situations, especially when the packet arrival of the input flows is not Poisson. To that end, we validate our model with the packet inter-arrival time to be exponentially distributed, a combination of constant and exponentially distributed, and also from the real packet traces. We test whether the model can reproduce the same packet delays in the downscaled system as those in the original system. Since we only focus on validation of the queuing model in this study, we experiment with the simulated system (described in the previous section), in which case we can conveniently explore different parameter settings, and stay clear from potential system-related artifacts in the real implementation.

The setup for the experiments is show in Fig. 5.11. To simulate the original system, we designate two flows at a network interface: one as "simulated flow" and the other as "emulated flow", each with an independent input process for the packet arrival rate λ_s and λ_p, respectively. We measure at each second the drop probability p, the effective arrival rate of emulated flow λ_p^e, and the average packet delay of both flows W_1, at the network interface. The measurements are collected in a trace file and later used by a subsequent simulation of the downscaled system. To simulate the downscaled system, we include only the "emulated flow" using the same input process as that in the original system. The downscaled system implements the queuing model as described in previous section.

Poisson Arrivals

We first set the packet inter-arrival time of both simulated and emulated flows to be exponentially distributed. We set the bandwidth of the network interface to be 10 Mb/s and the queue length to be 1.5 MB. We also fix the packet size to be 1500 bytes in this study (we use variable packet sizes when experimenting with real traces later). We examine the accuracy of the system by comparing the packet delays between the original and the downscaled systems. We vary the aggregate arrival rate of both simulated and emulated flows to be 10%, 50%, and 90% of the bandwidth for different service utilization levels, and we vary the proportion of the emulated flow to be 20%, 50%, and 80%, respectively.

Figures 5.12, 5.13, and 5.14 show the average packet delays measured at each second during the experiment for the low, mid, and high service utilization scenarios. In all cases, both systems match quite well. The difference is around 10 microseconds on average for 10% utilization, 40 microseconds for 50% utilization, and well under one millisecond for 90%.

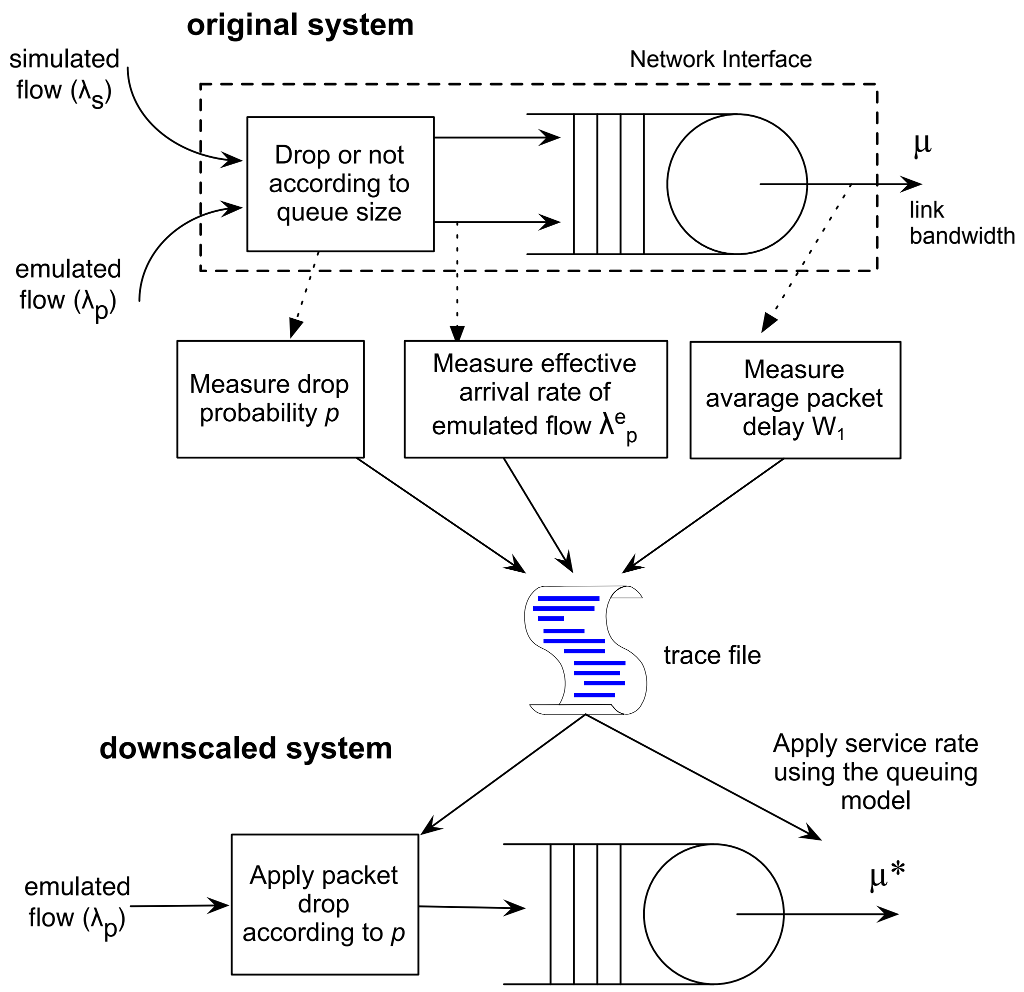

original system
simulated flow (λ_s)
emulated flow (λ_p)
Network Interface
Drop or not according to queue size
μ
link bandwidth
Measure drop probability *p*
Measure effective arrival rate of emulated flow λ^e_p
Measure avarage packet delay W_1
trace file
downscaled system
Apply service rate using the queuing model
emulated flow (λ_p)
Apply packet drop according to *p*
μ^*

Figure 5.11: Experiment setup for single link validation

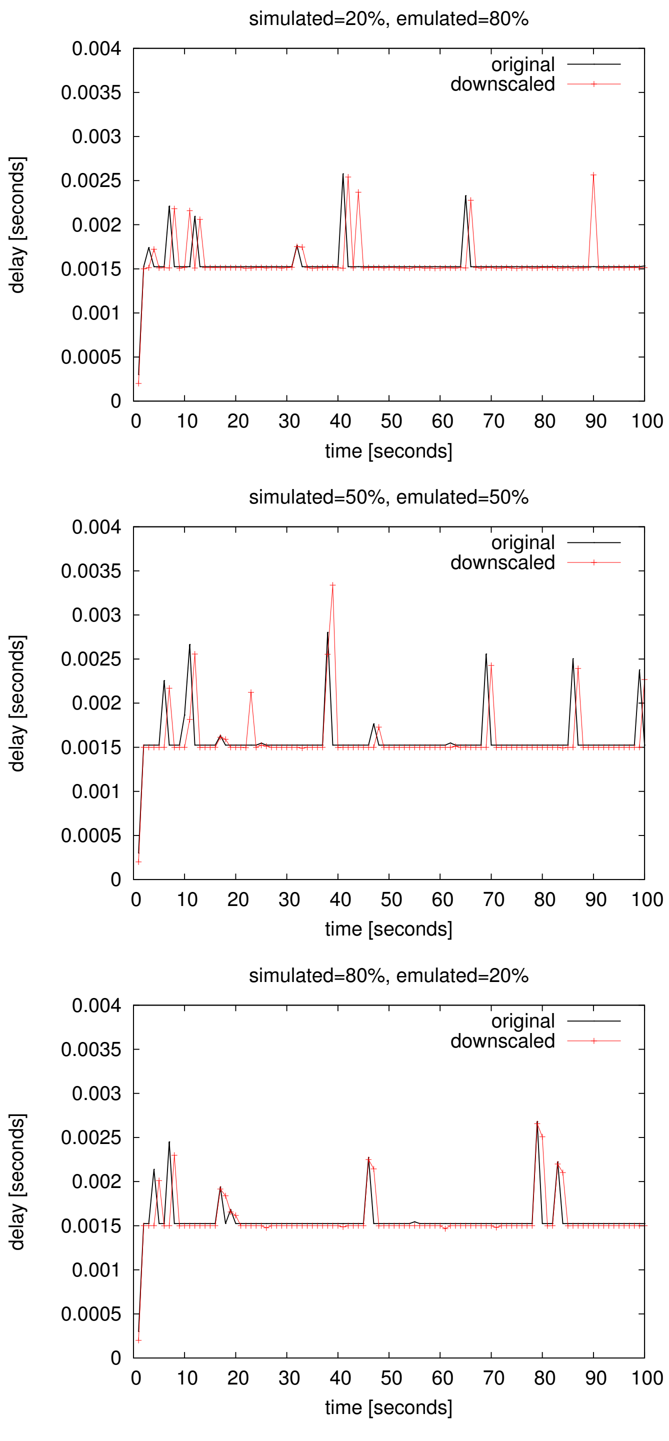


Figure 5.12: Comparing packet delays for Poisson arrivals under 10% utilization

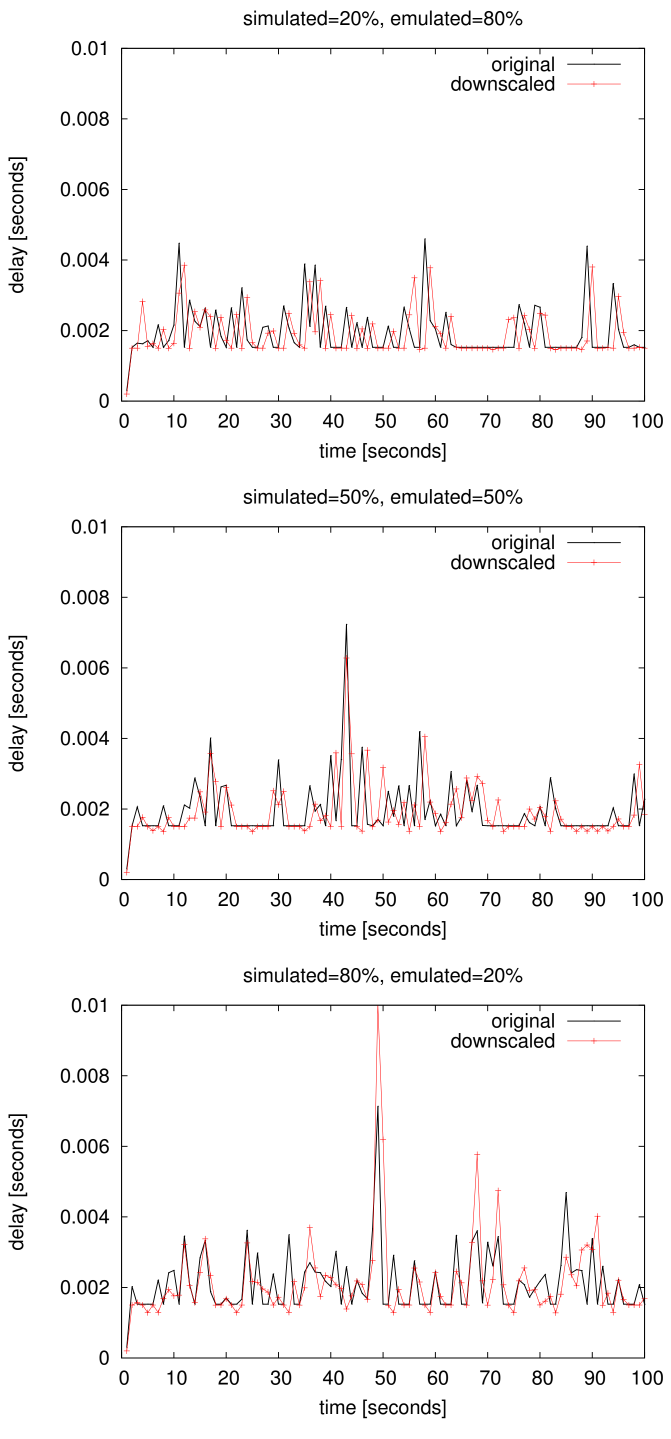
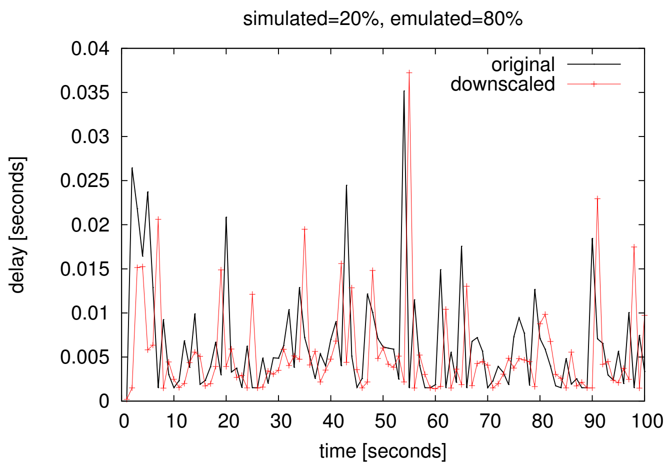
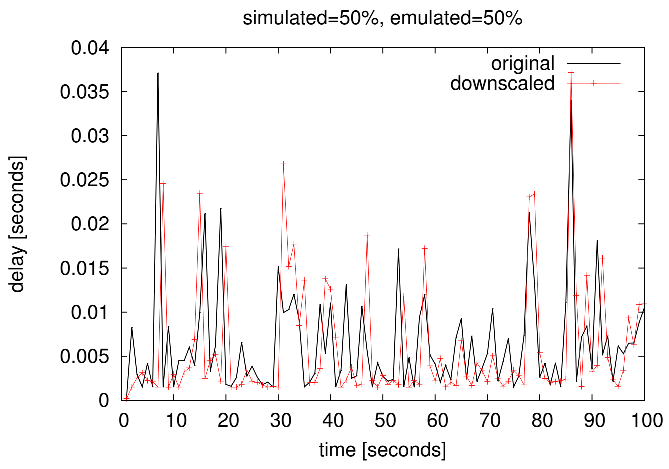


Figure 5.13: Comparing packet delays for Poisson arrivals under 50% utilization





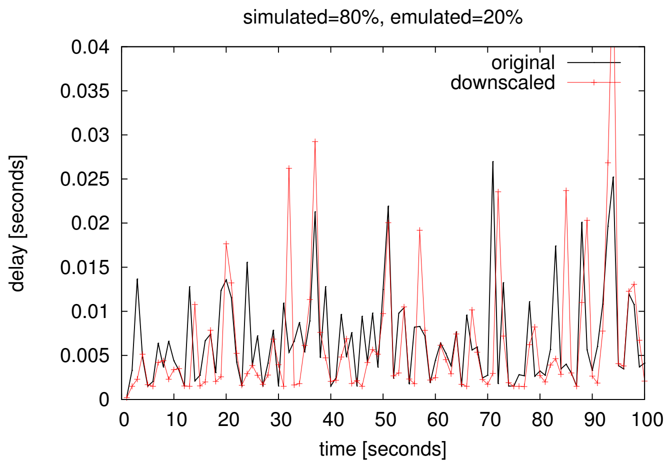


Figure 5.14: Comparing packet delays for Poisson arrivals under 90% utilization

Mixed Poisson and Constant Arrivals

Next, we use different arrival processes for simulated and emulated flows. We set the packet inter-arrival time of the emulated flow to be constant and that of the simulated flow to be exponentially distributed, and vice versa. We use the same experiment setting as that in the previous experiment. Again, we vary the aggregate arrival rate of both simulated and emulated flows to be 10%, 50%, and 90% of the total bandwidth, and the proportion of the emulated flow to be 20%, 50%, and 80%, respectively.

Figures 5.15, 5.16, and 5.17 show the average packet delays for different service utilizations when the packet inter-arrival time of the emulated flow is constant and that of the simulated flow is exponentially distributed. In all cases, both systems match quite well. The difference is under one millisecond.

Real Packet Traces

The simulator is able to read packet traces generated by TCPDUMP so that it can reproduce similar traffic conditions as in the real system. We add the functions to either dilate or contract packet inter-arrival times by a constant factor in order to artificially adjust the traffic intensity as needed.

The first trace is obtained from one of our local servers during a relative idle period and has a low traffic intensity. Figure 5.18 shows the results comparing the original and downscaled systems when the bandwidth of the original system is 1Mbps, the queue size is 150000 bytes, the simulated flow is replayed from the traces, and the emulated traffic is exponentially distributed and of a rate of 0.5Mbps. Fig. 5.18 shows the delay for the emulated flow in both systems for factor values of 0.1, 1, 10, 100, and 1000. Fig. 5.18 shows that when the dilation factor is very small, more traffic is generated in a shorter time interval, thus producing a bottleneck very early in the experiment. As the factor

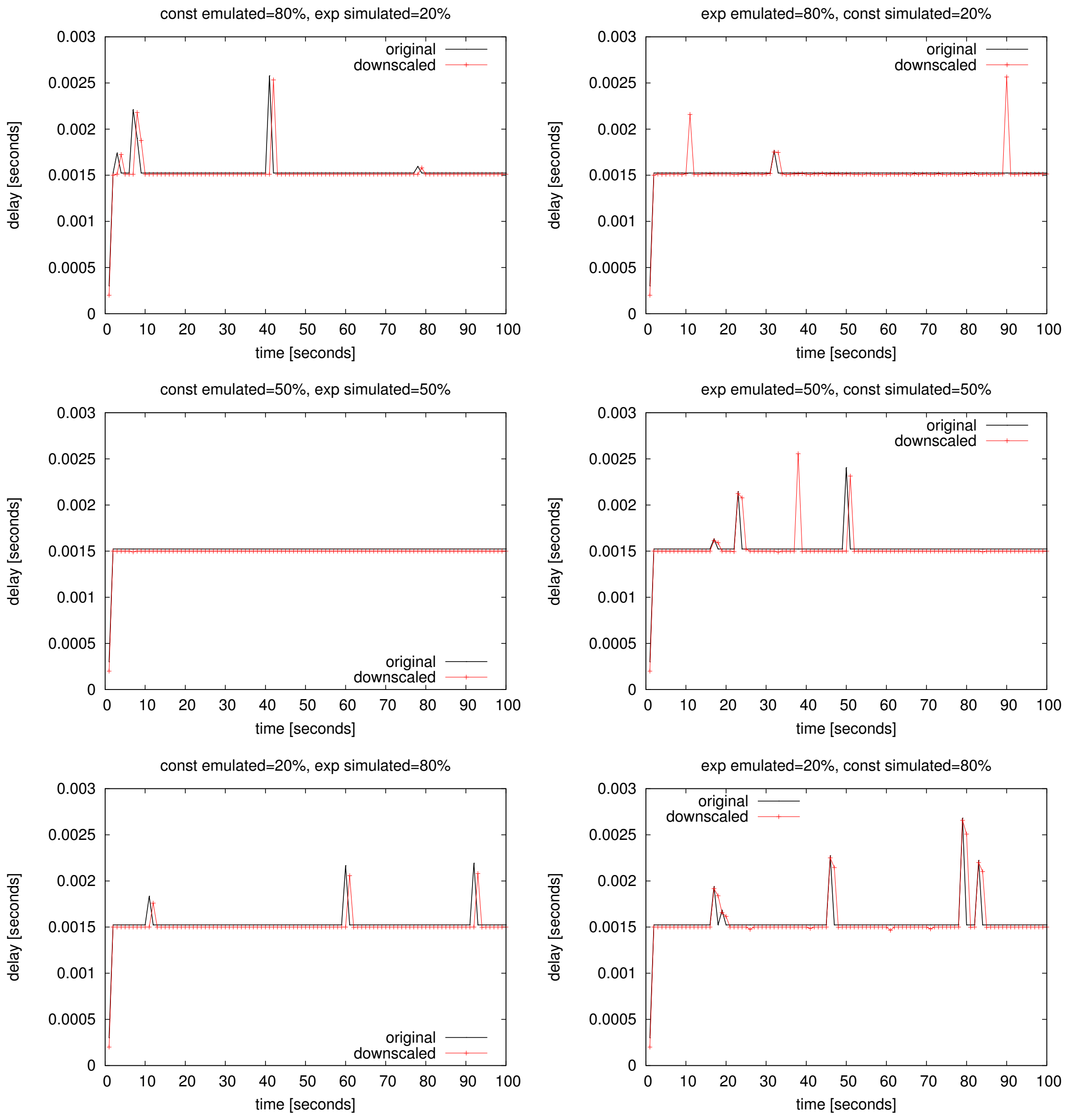


Figure 5.15: Comparing packet delays for mixed constant and exponential inter-arrival times for simulated and emulated flows under 10% utilization

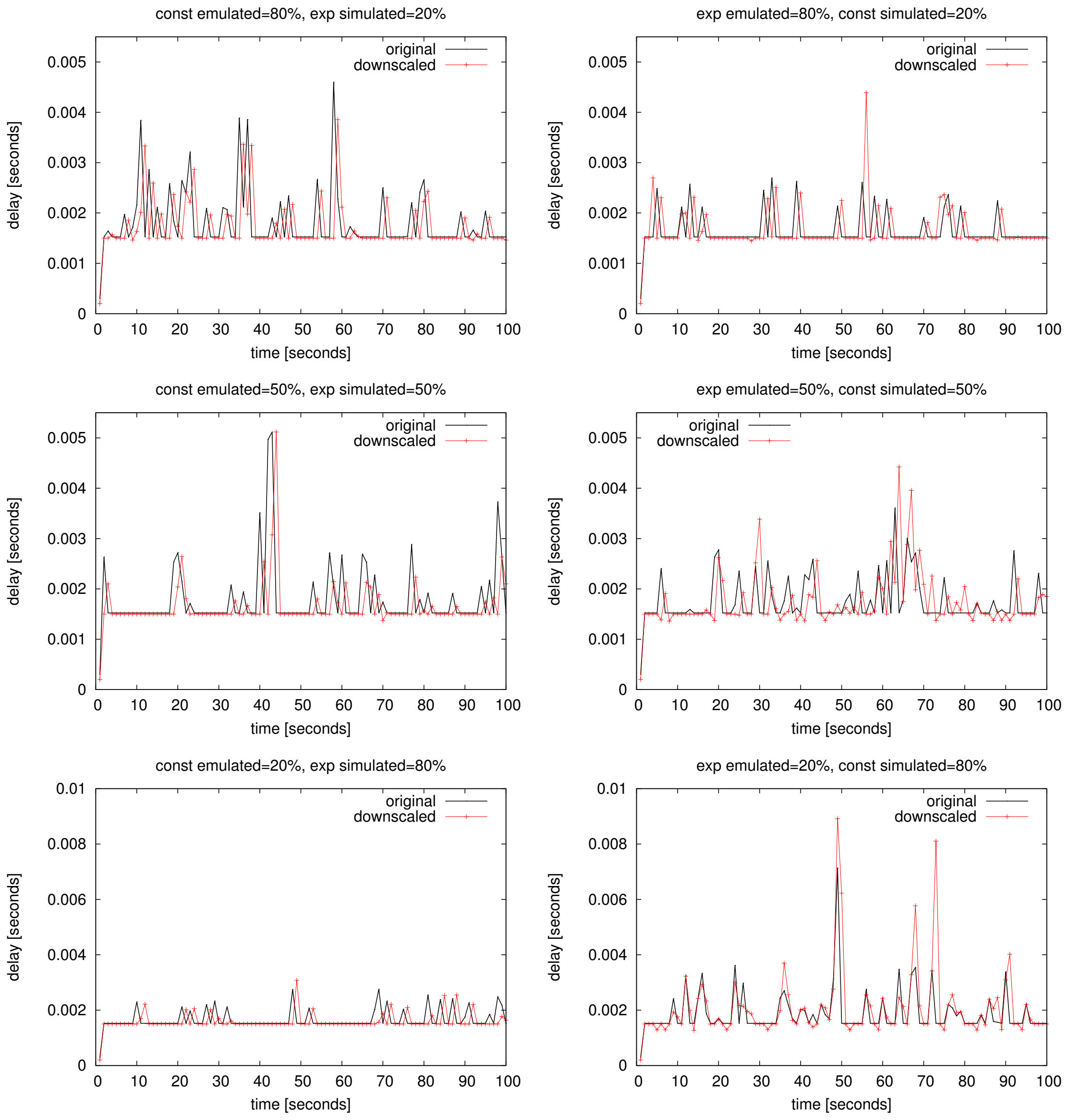


Figure 5.16: Comparing packet delays for mixed constant and exponential inter-arrival times for simulated and emulated flows under 50% utilization

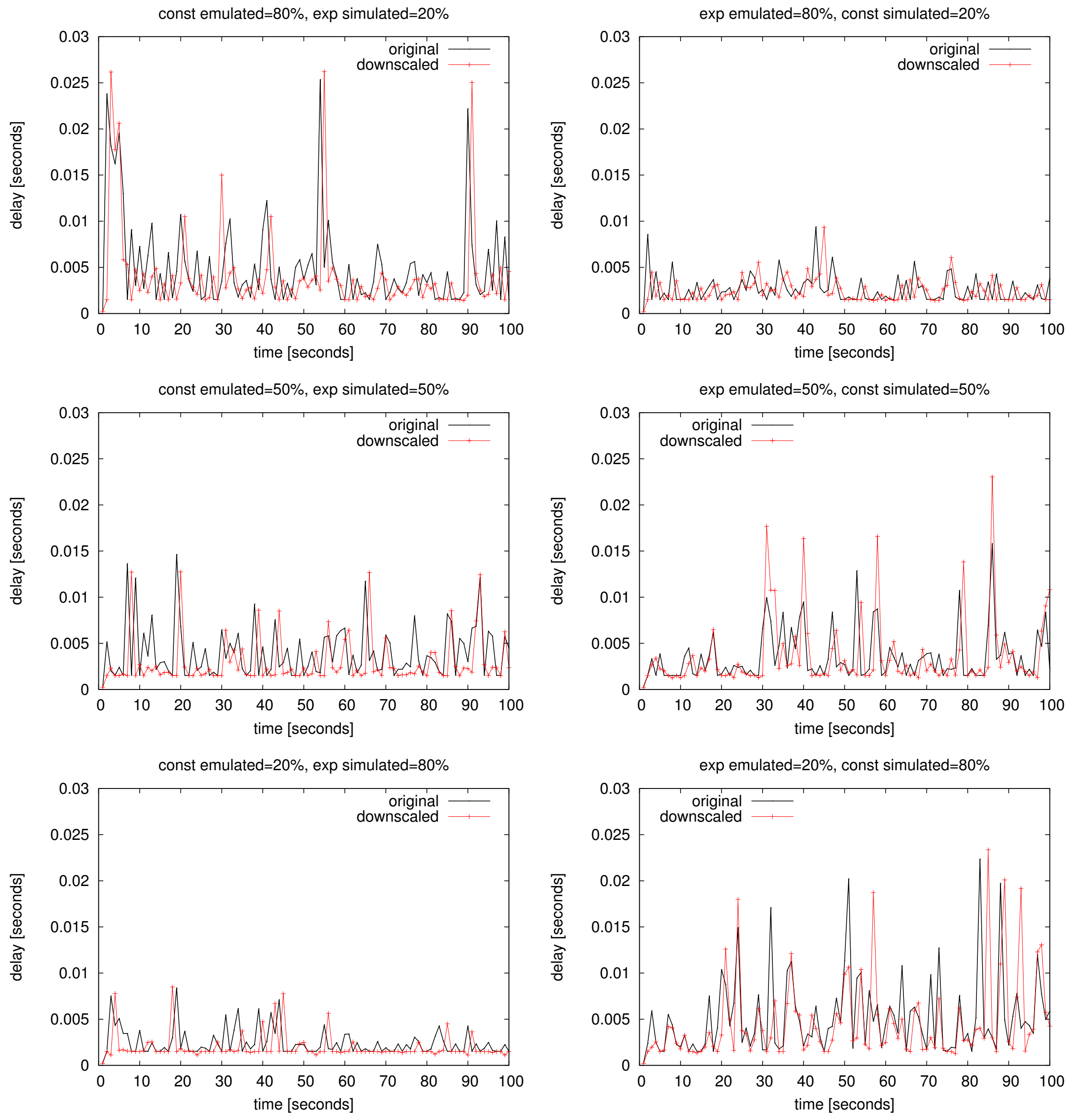


Figure 5.17: Comparing packet delays for mixed constant and exponential inter-arrival times for simulated and emulated flows under 90% utilization

increases, the bottleneck moves to a later time in the experiment until it totally disappears for a value of 1000. For all values, our model produces very good match between the delay obtained in the original and the downscaled system.

A second round of experiments is performed keeping the previous setting but changing the bandwidth to 10Mbps. Fig. 5.19 shows the delay for the emulated flow in both systems for factor values of 0.1, 1, 10, 100, and 1000. Again, we achieve a good match between the delay obtained in the original and the downscaled system.

We also gathered a real trace from the *The CAIDA Anonymized Internet Traces 2011 Dataset* [CAI], corresponding to a OC192 link (9953Mbps) link. The trace consists of twenty million packet headers and payloads, which we parsed to obtain a flat file containing the time the packet is seen and its size. We replay the CAIDA traces as simulated traffic; the emulation traffic is exponentially distributed.

Replaying the traces as given at a service rate of 10Gbps does not produce a significant queuing delay, thus yielding minimal variations in delay. In consequence, we created two different alternative scenarios to test our approach: for the first one, we dilated the trace by a factor of 30; for the second, we dilated the trace by a factor of 100. For both scenarios, we used a queue of 8300 packets, and made the emulated traffic rate to be 50% of the service rate; the service rate for the first scenario is 1Gbps and for the second is 100Mbps.

Fig. 5.20 shows the delay and the received packets history for the emulated flow in both the original and the downscaled subsystems for the first scenario. No congestion exists for this scenario and thus no packet loss; most of the delay is propagation delay(3ms). In this non-congested scenario, our model achieves great accuracy in terms of delay and received packets history.

Fig. 5.21 shows the delay, drop, and the received packets history for the emulated flow in both the original and the downscaled subsystems for the second scenario. As

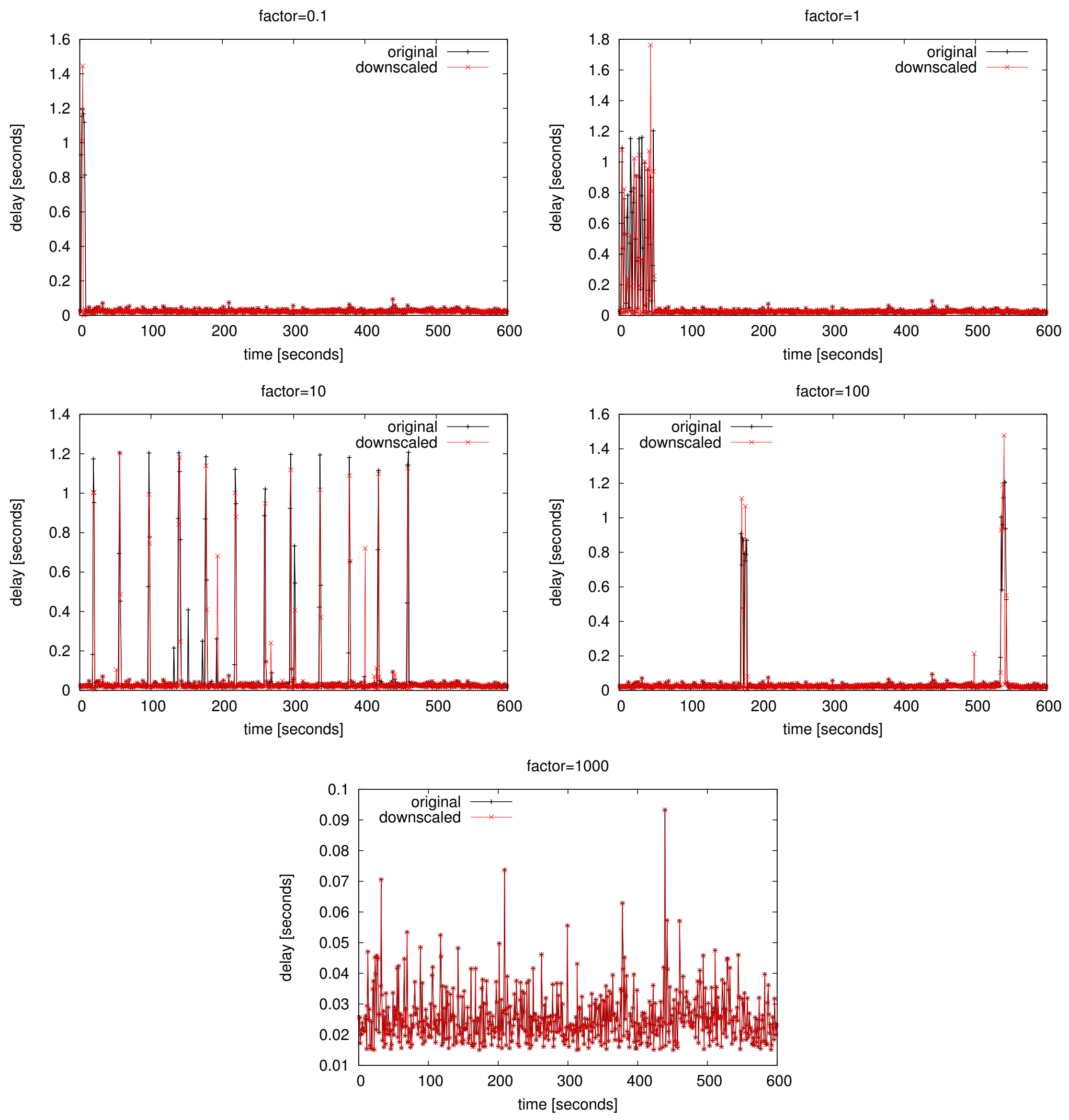


Figure 5.18: Comparing packet delays using low-intensity trace with different dilation factors for 1Mbps bandwidth

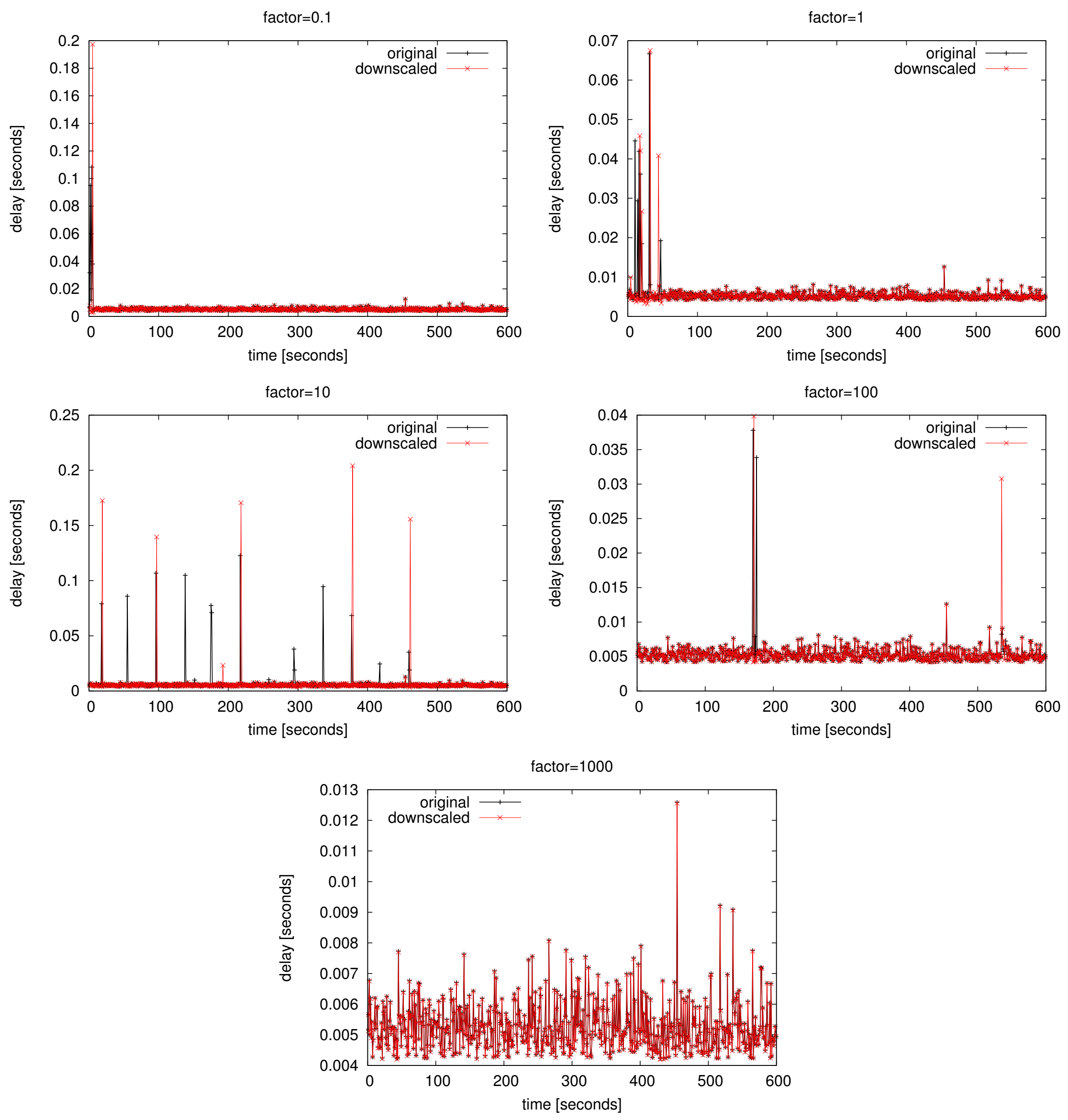
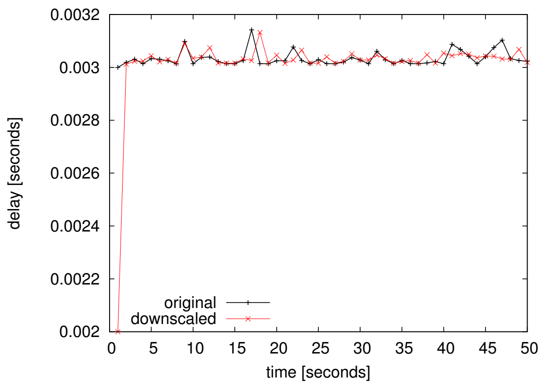


Figure 5.19: Comparing packet delays using low-intensity trace with different dilation factors for 10Mbps bandwidth



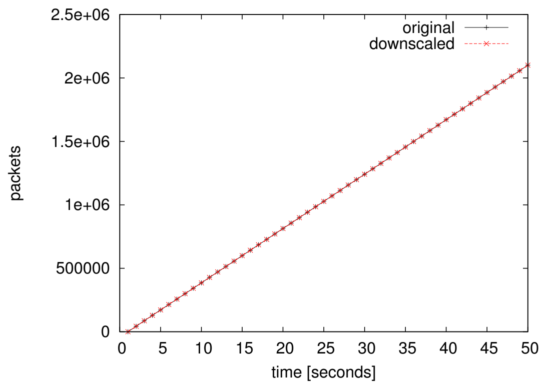


Figure 5.20: Delay and received packets history for the first scenario with CAIDA traces

observed, this is a highly congested scenario characterized by a total occupation of the queue between seconds 5 and 33 (Fig. 5.21 a)). Fig. 5.21b) shows that packets are only lost in this saturation stage. Finally, Fig. 5.21c) shows the history of received packets. For all three metrics, our model provides great accuracy.

Exploring Various Network Settings

In order to quantitatively evaluate the effectiveness of our model to emulate a single link, with and without congestion, we run a batch of experiments for different bandwidth and buffer sizes, and compare the average packet delay, the number of packets dropped, and the number of received packets, between the original system and the downscaled system. For the experiment, we set the link bandwidth to be either 1, 10, or 100 Mb/s, and the buffer size to be 100, 500, or 1000 packets. For each bandwidth and buffer size pair, we independently set the packet arrival rate for the emulated and simulated flows to be 12, 25, 50, or 75% of the link bandwidth, and consequently we conduct a total of 16 experiments. Each experiment lasts 100 seconds, and both simulated and emulated flows start from time zero.

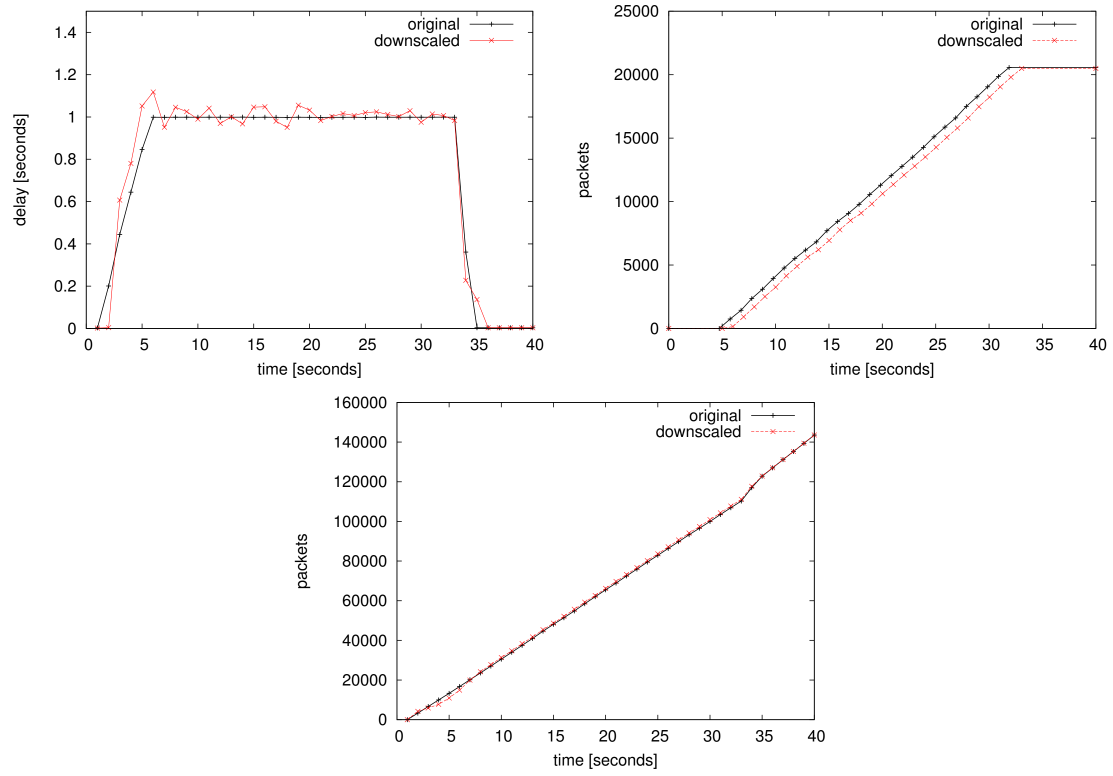


Figure 5.21: Delay and received packets history for the second scenario with CAIDA traces

Table 5.1: Errors for different link bandwidth and buffer size combinations

Bandwidth	Buffer Size	Packet Delays	Packet Losses	Received Packets
1 Mb/s	100 pkts	8 *ms* (7.96%)	2.06 (0.22%)	3.25 (0.07%)
1 Mb/s	500 pkts	8 *ms* (7.63%)	5.93 (0.56%)	7.12 (0.16%)
1 Mb/s	1000 pkts	7 *ms* (7.57%)	2.93 (0.49%)	3.00 (0.07%)
10 Mb/s	100 pkts	2 *ms* (8.83%)	18.56 (0.13%)	21.37 (0.04%)
10 Mb/s	500 pkts	2 *ms* (7.93%)	34.18 (0.29%)	32.12 (0.07%)
10 Mb/s	1000 pkts	2 *ms* (7.78%)	24.43 (0.22%)	21.00 (0.04%)
100 Mb/s	100 pkts	0.3 *ms* (5.04%)	141.50 (0.62%)	56.75 (0.01%)
100 Mb/s	500 pkts	0.4 *ms* (3.90%)	183.12 (0.17%)	98.56 (0.02%)
100 Mb/s	1000 pkts	0.4 *ms* (3.67%)	209.00 (0.19%)	113.12 (0.02%)

Table 5.1 shows the average errors for packet delays, packet losses, and throughput across the 16 experiments using different packet arrival rates for the emulated and simulated flows for each combination of link bandwidth and buffer size.

The average percentage error for the number of dropped packets and the number of received packets is very small, indicating a near perfect match between the two systems. For emulating a link with a bandwidth of 100 Mb/s, the error in the packet delay is less than one millisecond. It goes up to 2 milliseconds for a 10 Mb/s link, and 8 milliseconds for a 1 Mb/s link, which is mainly due to the slower packet transmission time. The results show conclusively that our queuing model provides a very good approximation for emulating a link of different bandwidth and buffer size, regardless of the congestion level.

Effect of Queuing Model Improvement

In section 5.3.1, we describe our improvement to the simple M/D/1 queuing model to deal with the link congestion more accurately. In the previous sections, we show that our model is able to accurately represent the queuing behavior whether the link is congested or not. In this section, we examine the effect of this improvement in more detail.

In the experiment, we create an artifical scenario, by fixing the Poisson arrival rate of the simulated and emulated flows to be 30 packets/s and 10 packets/s, respectively, and changing the link bandwidth from 60 packets/s (for no congestion), to 40 packets/s (for mild congestion), and then to 20 packets/s (for heavy congestion). In Figure 5.22, we compare the results with and without the improved queuing model. In Figure 5.23, we switch the arrival rate of the simulated and emulated flows and again compare the results with and without the improved queuing model. In both cases, we can observe the dramatic improvement of the model in terms of accurracy.

5.7.2 Multiple Links Validation

In the previous section we assess the accuracy of our queuing model for emulating a single link. In this section we examine the model for emulating a network path with multiple links. Again, we use with the simulated system as in the previous section, so that we can conveniently explore different parameter settings.

The setup for the experiments is shown in Figure 5.24. In the original system, the network path consists of three links each represented using a network queue with distinct bandwidth. We designate an "emulated flow" to traverse all three links. We also designate three "simulated flows", each going through a separate network queue. All simulated and emulated flows have independent input source with potentially different packet arrival rate. Similar to the setup for the single link validation, we measure at each second the drop probability and the effective arrival rate of the emulated flow at each network queue. We also calculate the average packet delay of all packets going through the system. These measurements are collected and stored in a trace file to be used by the subsequent simulation of the downscaled system. For the downscaled system, we include the emulated flow as in the original system and apply the queuing model using data stored in the trace file.

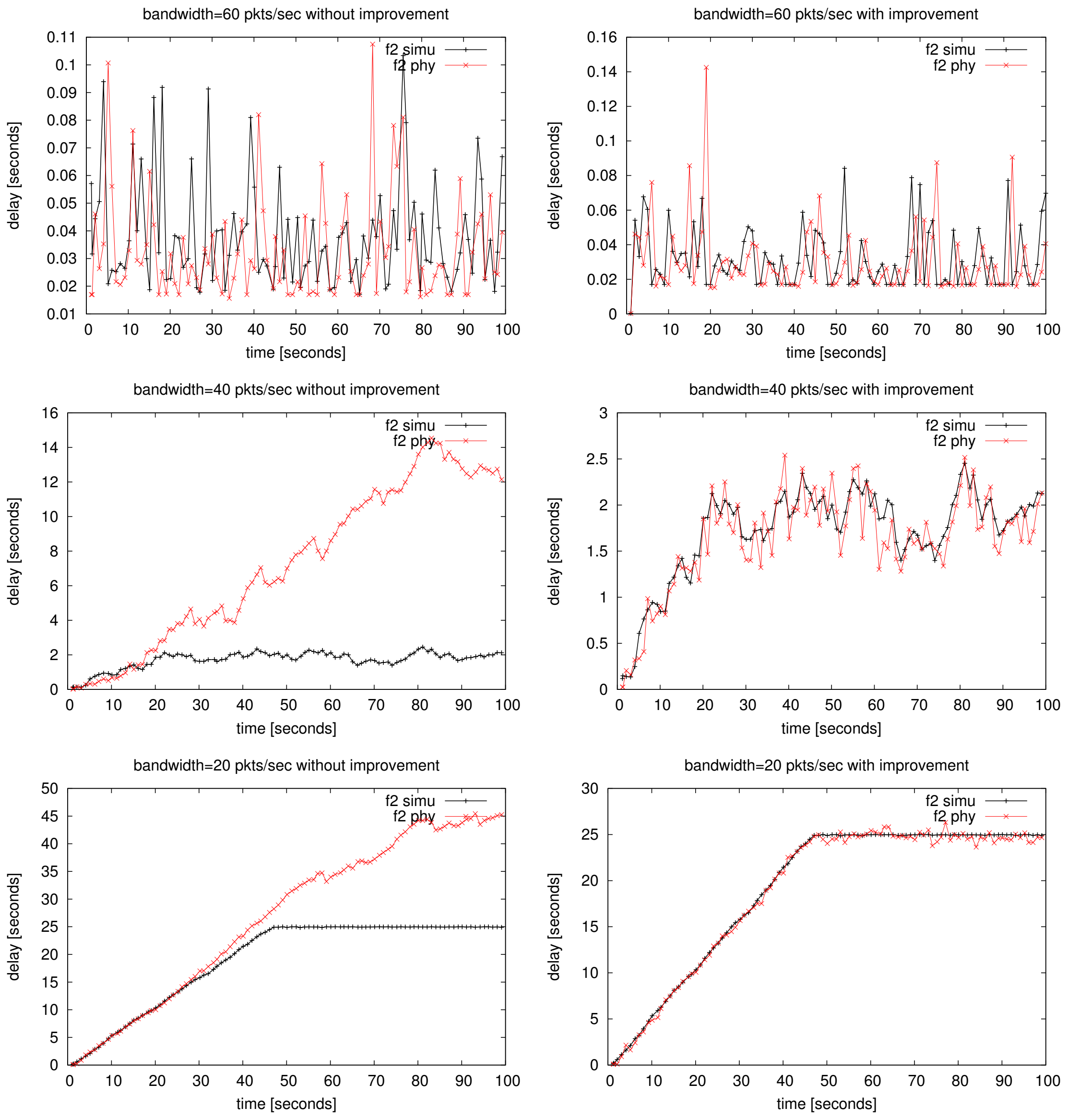


Figure 5.22: Model improvement shown with higher emulated flow arrival rate

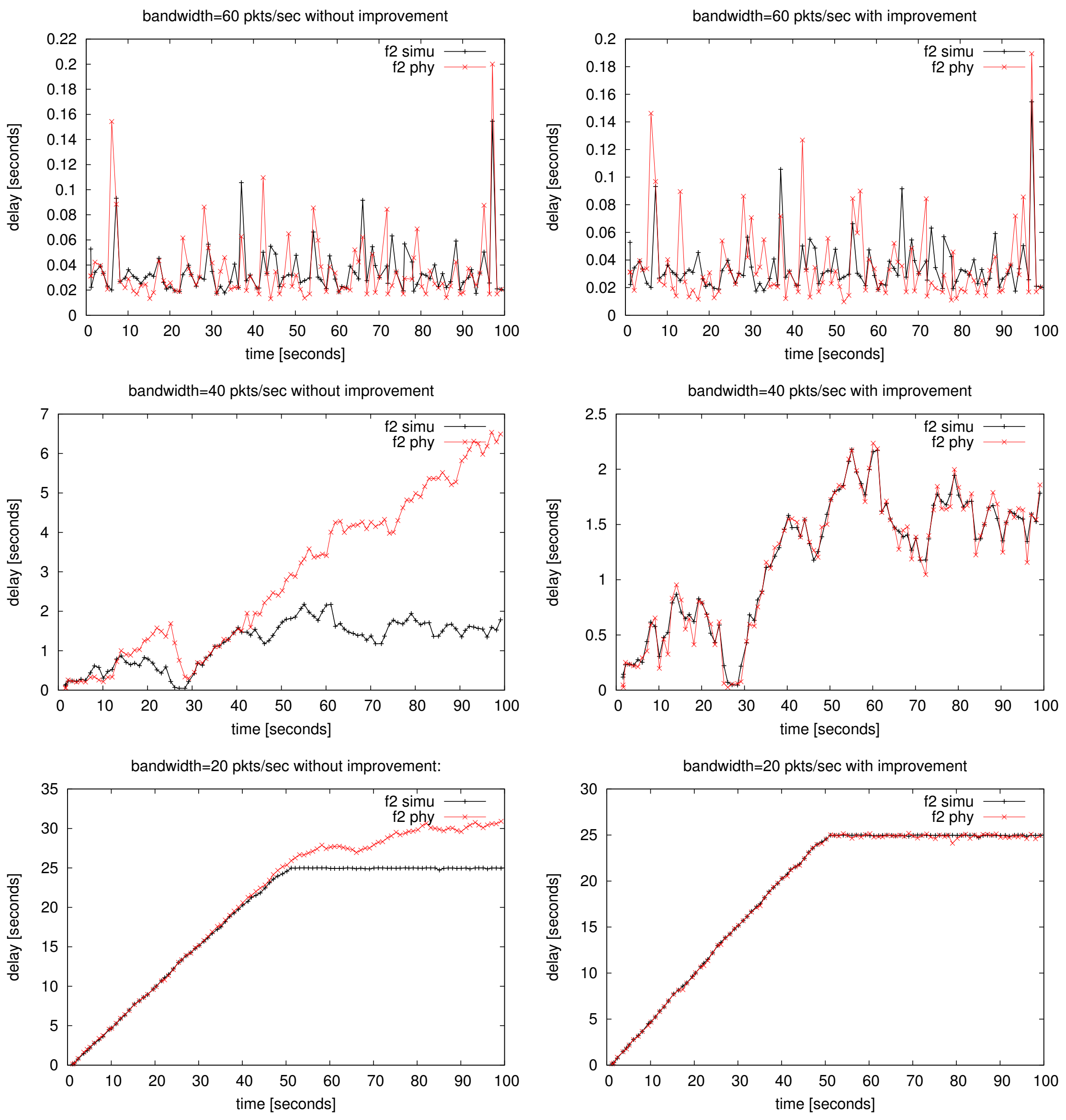


Figure 5.23: Model improvement shown with higher simulated flow arrival rate

Staggered Arrivals

In the first experiment, we set the bandwidth to be 1, 10, and 100 Mb/s, respectively for the three links. The buffer size of all network queues is set to be 200 packets. We fix the packet size to be 1.5 KB. All flows use Poisson arrivals. We set the arrival rate of the emulated flow to be 50 packets/s (i.e., 600 Kb/s); we start the flow at the beginning of the experiment and let it persist throughout the whole experiment. We set the arrival rate of the simulated flows to be 600 Kb/s, 9.6 Mb/s, and 99.6 Mb/s. We start the third simulated flow at the beginning of the experiment and let it last for 40 seconds. The second simulated flow starts at 30 seconds and ends at 50 seconds. The first simulated flow starts at 60 seconds and ends at 80 seconds. In this way, we created three congestion points in the network paths. The experiment runs for 120 seconds in total.

Figures 5.25, 5.26, and 5.27 show the average packet delay, the cumulative packet loss, and the throughput of the emulated flow over time. We can see clearly from the delay plot that congestion happens between 30 and 50 seconds, and between 60 and 100 seconds. It takes time to clear up a congestion: the figure shows that the average delay waits until 106 seconds to come back to normal. Since the first link is the slowest link, as expected, it results in more significant packet delay during the congestion (between time 60 and 106), which is caused by the longer packet transmission time at the first queue. The packet loss is also significant between 60 and 100 seconds; this is because the emulated and simulated flows are mixed up with the same proportion at the first queue and therefore would share the loss equally, while at the other two queues the simulated flow dominates the emulated flow. In all cases, the original system and the downscaled system match very well. Our queuing model works correctly for multiple links.

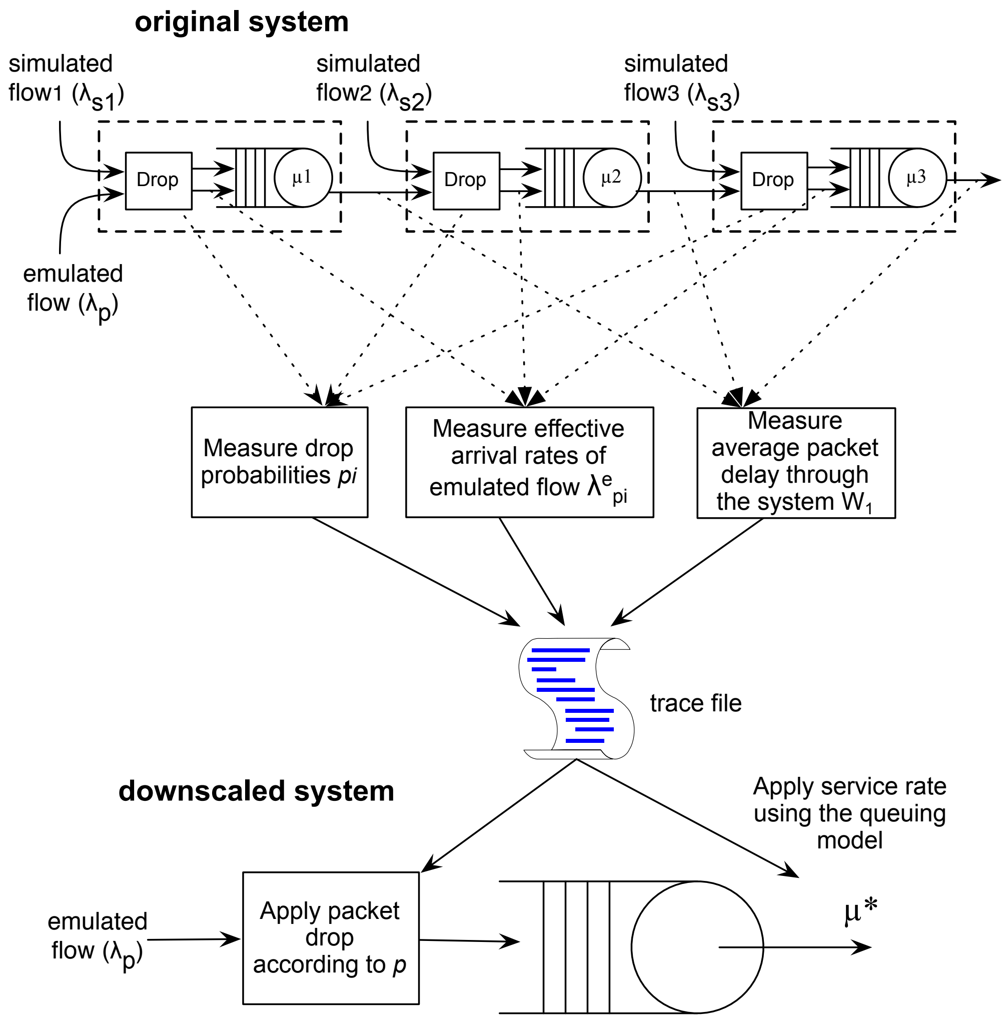

original system
simulated flow1 (λ_{s1})
simulated flow2 (λ_{s2})
simulated flow3 (λ_{s3})
Drop
μ1
Drop
μ2
Drop
μ3
emulated flow (λ_p)
Measure drop probabilities *pi*
Measure effective arrival rates of emulated flow λ^e_{pi}
Measure average packet delay through the system W_1
trace file
downscaled system
Apply service rate using the queuing model
emulated flow (λ_p)
Apply packet drop according to *p*
μ*

Figure 5.24: Experiment setup for multiple links validation

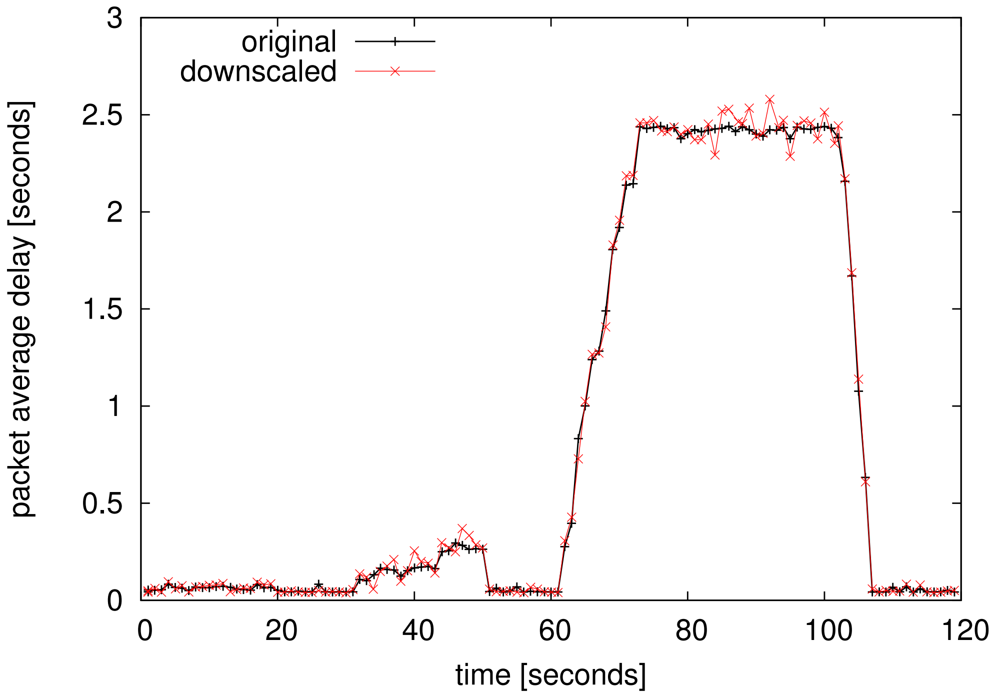


Figure 5.25: Average packet delay of the emulated flow

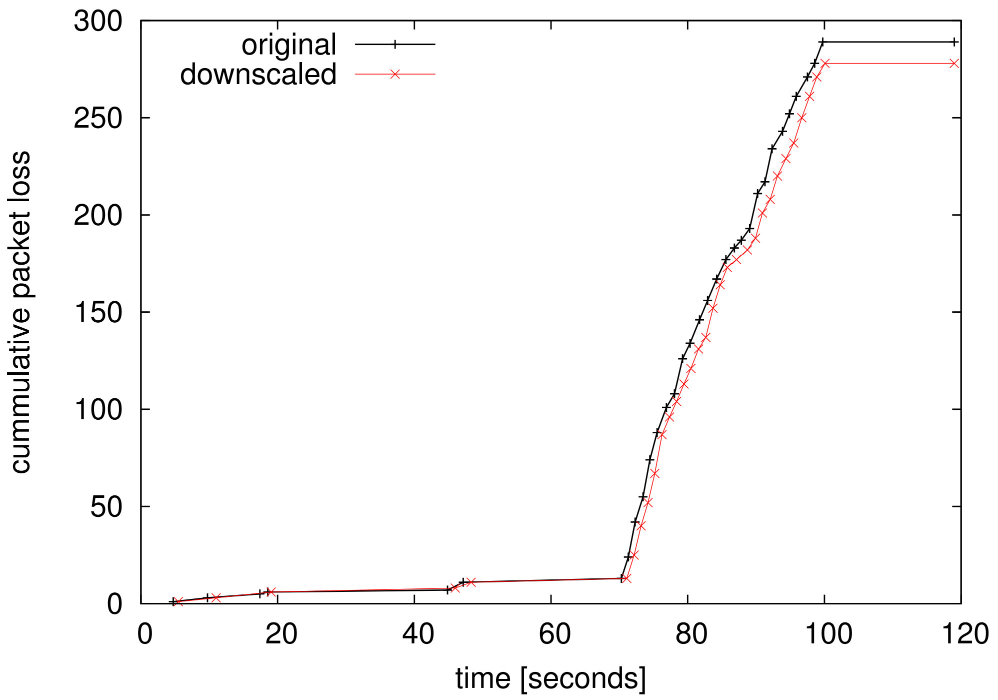


Figure 5.26: Cumulative packet loss of the emulated flow

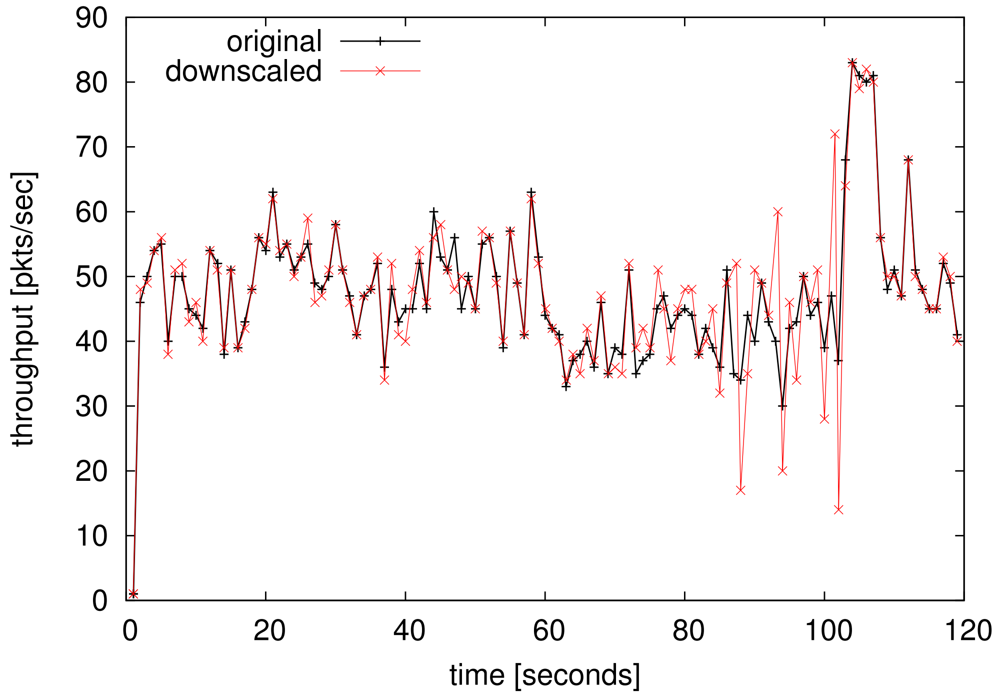


Figure 5.27: Throughput of the emulated flow

Exploring Various Network Settings

Like before, in order to quantitatively evaluate the effectiveness of our queuing model to emulate multiple links, we run a batch of experiments using different bandwidths for the links. To assess accuracy, we again compare the average packet delay, the number of dropped packets, and the number of received packets, between the original system and the downscaled system.

For this experiment, we independently set the bandwidth of the three links to be either 10 or 100 Mb/s. For each of the eight permutations of the link bandwidths setting, we conduct six separate experiments, each by setting the packet arrival rate of the simulated flows or the emulated flow to be 12, 25, 50, 100, 125, or 150% of the link bandwidth at the flow's entry point. For example, for the experiment where the link bandwidth of the first queue (μ_1) is 10 Mb/s, and the bandwidth of both the second and third queue (μ_2

Table 5.2: Errors for different link bandwidth settings

μ_1	μ_2	μ_3	Packet Delays	Packet Losses	Throughput
1 Mb/s	1 Mb/s	1 Mb/s	1.72%	0.30%	0.28%
1 Mb/s	1 Mb/s	10 Mb/s	1.96%	0.59%	0.28%
1 Mb/s	10 Mb/s	1 Mb/s	1.91%	2.47%	0.21%
1 Mb/s	10 Mb/s	10 Mb/s	0.85%	0.40%	0.15%
10 Mb/s	1 Mb/s	1 Mb/s	0.54%	0.32%	0.99%
10 Mb/s	1 Mb/s	10 Mb/s	0.55%	0.18%	0.79%
10 Mb/s	10 Mb/s	1 Mb/s	0.19%	0.37%	0.97%
10 Mb/s	10 Mb/s	10 Mb/s	1.8%	0.23%	0.10%

and μ_3) is 100 Mb/s, and the arrival rate of all the flows is 50%, the packet arrival rate of the simulated flow (λ_p) and the first simulated flow (λ_{s1}) would both be 5 Mb/s, and the packet arrival rate of the second and the third simulated flow (λ_{s2} and λ_{s3}) would be 50 Mb/s. In this experiment, we run the simulation for 500 seconds, and we set all flows to start at time zero and run for the entire duration of the experiment.

Table 5.2 shows the differences between the original and the downscaled systems, for packet delays, packet losses, and throughput across the six experiments for each of the eight permutations of the link bandwidths setting. The errors are all below 2%, indicating that our queuing model can correctly emulate the network path with multiple queues in this case.

5.7.3 Real System Validation

We implemented a prototype of SymbioSim extending our PRIME network simulator and we have it currently running in ProtoGENI [PRO]/Emulab [WLS$^+$02]. We use ProtoGENI for hosting our simulator and the SIMEDSS because it gives us the possibility of owning an experiment for a long period of time, thus saving a lot of time for installation. Emulab is easier to handle when delay nodes are needed so it is used for hosting the Actuator, the PhyEDSS, and the emulation (physical) system (including the real hosts).

In this section, we use the SymbioSim prototype to validate the accuracy of our approach when using real applications and protocols. To that end, we compare the delay and throughput produced by a real application running in SymbioSim's emulation system against those produced by a simulated application, of similar characteristics as the real application, running entirely in simulation. We conduct this test using the parking lot topology shown in Fig. 5.28. In Fig. 5.28a), we show the network topology used for both the purely simulated experiment and for the one using SymbioSim. When using SymbioSim, all hosts are *simulated hosts* except hosts 1 and 2, which are *emulated hosts*. Fig. 5.28b) shows the downscaled network topology, used in the SymbioSim experiment, which only includes real hosts 1 and 2 connected through a link, i.e., two pipes in the emulation system. Finally, Fig. 5.28c) shows the real implementation of the downscaled topology which includes a Dummynet instance to emulate a link.

In simulation, F1 is a TCP flow generated by a HTTP application that transfers a large object. In SimbioSim, we use a home crafted Java application that also transfers a large object to generate F1. We make our HTTP application to resemble the Java one by tuning it to use the same TCP congestion window and the same TCP congestion control algorithm (Reno). At the beginning of each experiment, we initiate the transfer between hosts 1 and 2; this transfer lasts for the whole experiment lifetime. At second 10, other 5 TCP flows, depicted as F2 in Fig. 5.28, start transferring 0.5MB each. At second 30, other 5 TCP flows (depicted as F3) start transferring 2MB each. Before these finish, other 10 TCP flows (composing F4) start transferring 1MB each. Flows F3 and F4 overlap and create two simultaneous points of congestion. Finally, at second 50, 5 TCP flows (depicted as F5) start transferring 1MB each as shown in Fig. 5.28.

Fig. 5.29 shows the results. The first plot of Fig. 5.29 shows the RTT for both the purely simulated experiment and SymbioSim. For the former, we extracted the RTT from the trace files outputted by our simulated TCP version. For the latter, we used *ping* in

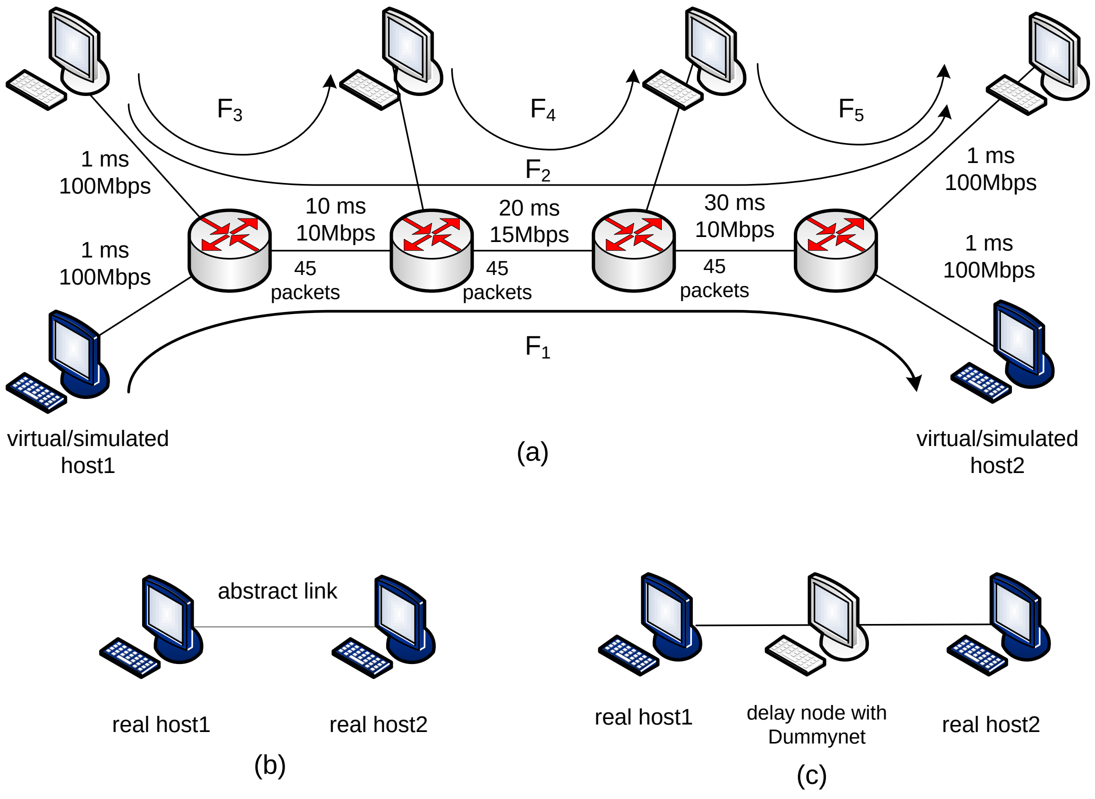
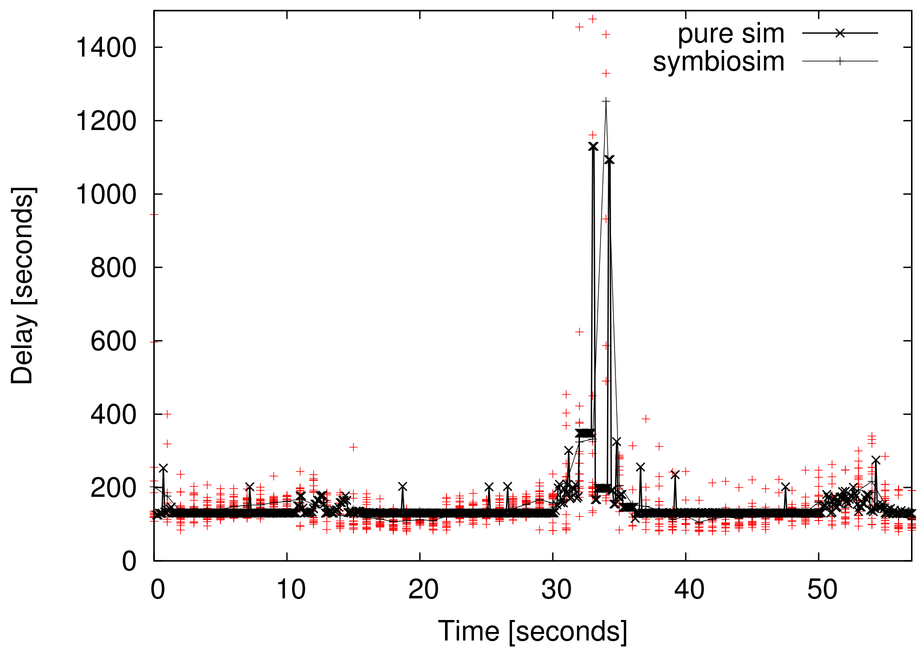


Figure 5.28: Model and experimental setup for baseline validation



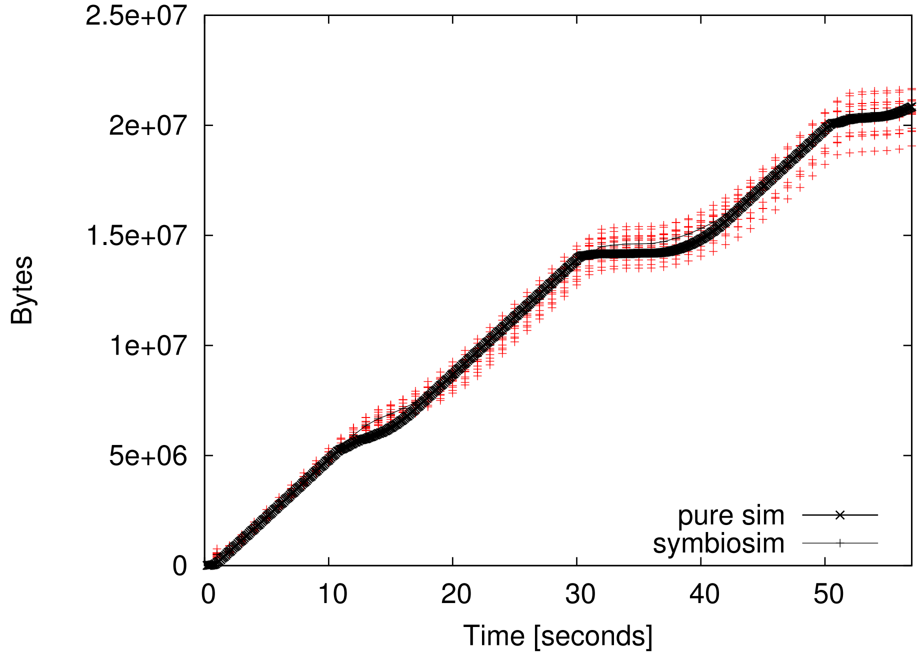


Figure 5.29: Results for prototype validation

parallel with the TCP transfer (generated by our Java application) to get the RTT. We plot both the average RTT from SymbioSim for fifteen runs and all the cloud of points corresponding to raw results from the emulation system. As shown there, SymbioSim outputs a delay curve that very closely follows that from the purely simulated experiment. The average delay from simulation is 148ms and that obtained from SymbioSim is 152ms, achieving an error of around 3%. As expected, the curve from SymbioSim is smoother since we update the physical system every 1 second. Observe that for all four congestion points we created, SymbioSim reacts accordingly, thus yielding the appropriate delay. If the delay and packet loss rate imposed by SymbioSim to the real flow are very similar to those experimented by the corresponding flow in simulation, then the sequence number history must match as well. The second plot of Fig. 5.29 shows the average sequence number history for the real TCP flow as well as the cloud of points that originate it for fifteen runs. Also, the corresponding one for simulation is depicted. As observed, the sequence number history curve from our simulated TCP is almost indistinguishable from the average sequence number history curve (corresponding to the real TCP flows originated by the Java application). The maximum sequence number seen at the end of the experiment in simulation is 20,843,961 against 20,748,458 in SymbioSim, yielding an error of around 0.5%. Some runs yield less or more final throughput depending on the exact instant *appBytes* values are injected into the simulator.

Confirming the results obtained in our SymbioSim's purely simulated version, the results shown above confirm that our model is valid for correctly emulating a network path. Also show, that our prototype reasonably and timely imposes the updates into the physical system.

5.8 Test Case

Previous approaches advocate for a class of protocols where multi-path routing is enabled in the Internet to take advantage of path diversity [HSH^{+}06, RWH09]. Using congestion feedback signals from the routers, Han et al. [HSH^{+}06] presented a transport protocol implemented at the sources to split the flow between each source-destination pair. In this section, we use SymbioSim to test a simple home-crafted multipath protocol inspired by the above mentioned research studies [HSH^{+}06, RWH09]. Here we do not aim at conducting a thorough study of a full-fledged protocol; instead, we aim at exposing SymbioSim as an approach to test new applications and protocols by leveraging the symbiotic cooperation between real systems and simulation.

The simple protocol we aim to test is based on source routing, i.e., the end host chooses its routing paths using the periodic updates it receives from dedicated routers. These special routers would provide multi-path service to subscribed end-hosts. The periodic updates may include network path drop rate, available bandwidth, or RTT. Using these updates, the end-host can decide what approach to use in order to maximize its target utility function.

The network topology we use for testing is shown in Fig. 5.30. In that topology, we place one special router the sender is subscribed to and we mark it as the *special router*, which sends periodic updates reporting the average packet loss probability and available bandwidth for each network path to the sender. Each alternative path is depicted as a cloud in Fig. 5.30, which represents the set of network links that compose a path from the special router to the *aggregation router*. Our simple protocol is implemented at the application layer on top of the TCP protocol. As such, the sender establishes independent TCP flows with the receiver. For all our tests, at any given time, two TCP flows are active

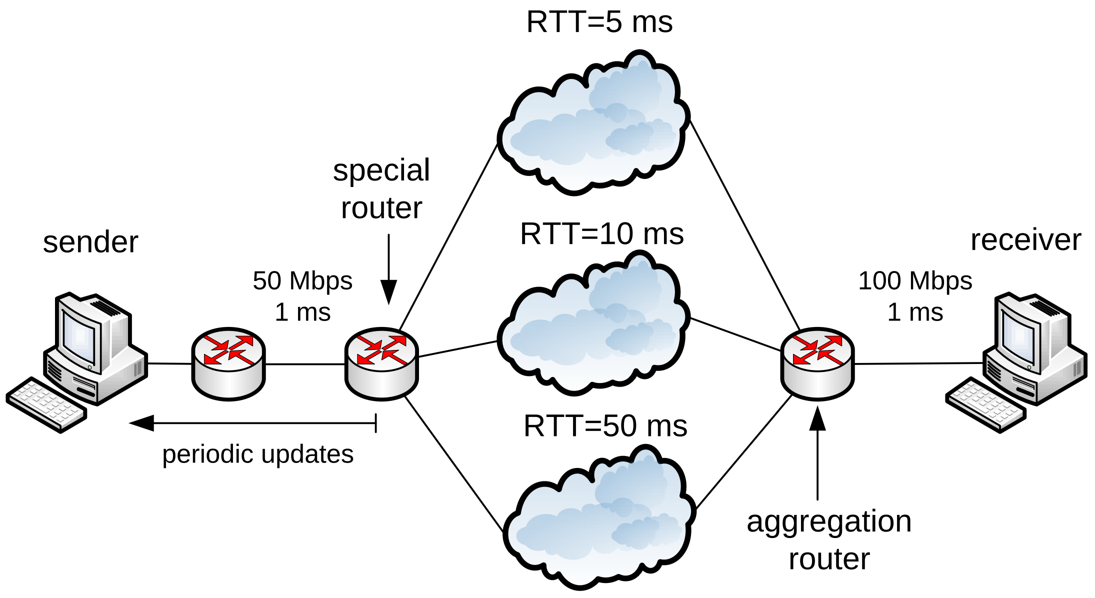


Figure 5.30: Topology for test case

between the sender and the receiver, each using a different network path. The specific approach implemented at the sender makes it switch one network path for another.

According to Fig. 5.30, the link that connects the receiver to the topology cannot be congested because the sender can only send at a maximum speed of 50 Mbps, dictated by the capacity of its access links. In consequence, this link is in some sense transparent to the TCP flows that traverse it [PPG06]. Therefore, we can we extract off that link from the topology to be implemented. Also, since the TCP flows do not share any more links from the special router up to the aggregation router, the network effects resemble those obtained from a topology where the flows would be established against three different receivers as shown in Fig. 5.31. This fact allows us to model the data exchanges using three different receivers as destinations and thus avoid injecting special information in the packets from the sender to the receiver that would be read by the special router to use an specific network path. Also, this fact saves us from providing specific logic at the special router. In a nutshell, we aim to model and test the logic at the sender to choose

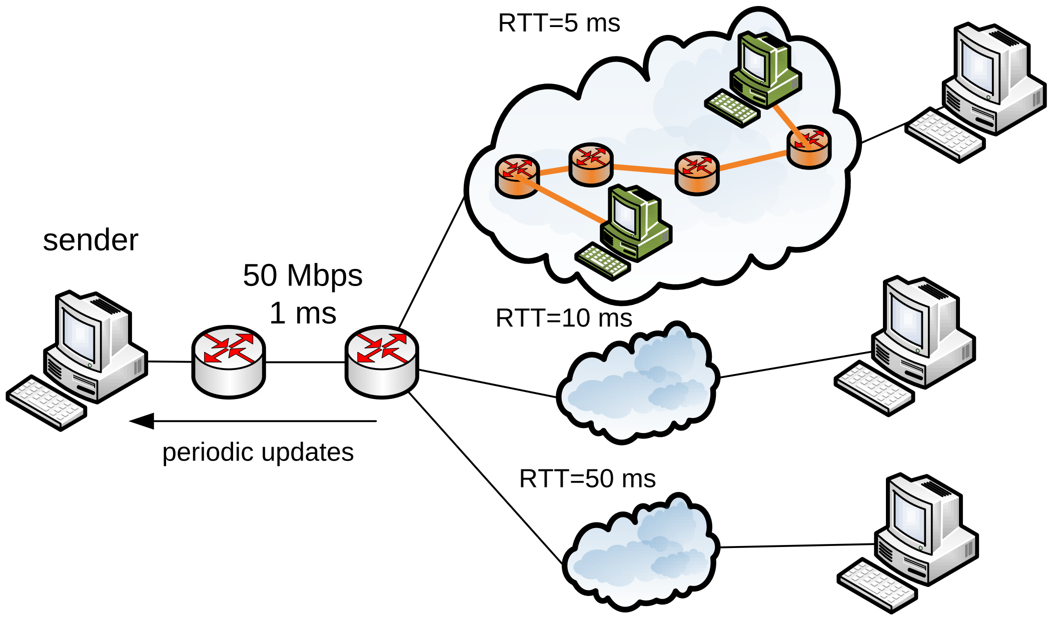


Figure 5.31: Approximation topology for test case

network paths and not the control plane data exchanges. Also, Fig. 5.30 shows that each path from the special router to the aggregation router, depicted previously as a cloud, is modeled as three network links, which can be downscaled to a single link in the physical system implementation. Finally, we place one simulated host at each end of each cloud to generate background traffic.

Fig. 5.31 can be downscaled using the approach proposed in section 5.4. The resulting topology is shown in Fig. 5.32.

At the implementation side, the sender is a Java application which runs three threads to send data to the receiver, one for each network path. Also, the sender listens on a UDP port for updates from the corresponding delay node. Upon receiving updates, the sender updates a shared data structure shared by all threads and the logic determines which pair of paths is used. Without SymbioSim, it would be difficult for a router to measure a precise value for drop packet rate or available in a network path. Also, SymbioSim is used to impose background traffic using simulated flows.

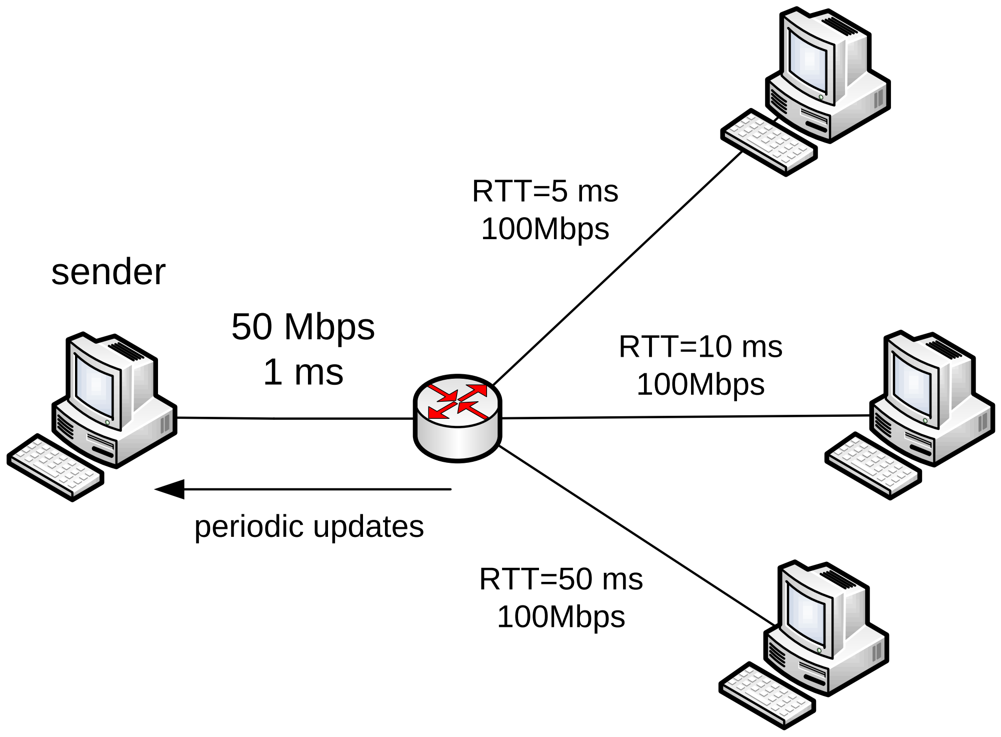


Figure 5.32: Downscaled topology for physical system

We test three different strategies for using multipath routing. The first approach is the simplest one and in a real implementation would impose the least computational burden in the participating routers, and least number of control packets exchanged between routers and end-hosts. In this case, the sender subscribes with the special router so that the latter routes the sender's packets through two different paths, which remain static during the whole experiment. In this way, aside from the initial packet exchange, no further packet exchanges are necessary. Although this approach has the potential of increasing the throughput when using just one flow (when having an spare network path when one of them is experimenting congestion), it has the drawback that when the two network paths used are congested, the throughput decays until at least one them becomes uncongested. At the beginning of the experiment, the sender establishes two independent TCP flows with the receiver. At second 20, ten TCP background flows start at each network path previously used, the average throughout history is depicted in Fig. 5.33 together with the 95% confidence interval corresponding to 20 trials. The throughput history was obtained from SymbioSim TCP traces files, so no other measurement tools were needed in the physical system. SymbioSim provides a set of parameters that may be customized by the user for specific needs. Linux default configuration including Reno as the TCP congestion control are used. From Fig. 5.33, it can be noted that before the first loss, TCP believes it can send packets at a higher rate than the configured bandwidth thus producing the peak at second at second 3. After that, TCP flows suffer the first loss, decrease the throughput and then recover and adjust. After second 20, the throughput decreases continuously since the sender is not able to recover from the loses given that both paths are congested, and the sender cannot find other paths during the experiment.

The second strategy makes effective use of the updates and uses the drop rate as the criteria to choose between network paths. When the experiment starts, the sender chooses the two paths with the lowest RTTs (see Fig. 5.30). Again, from second 20, these two net-

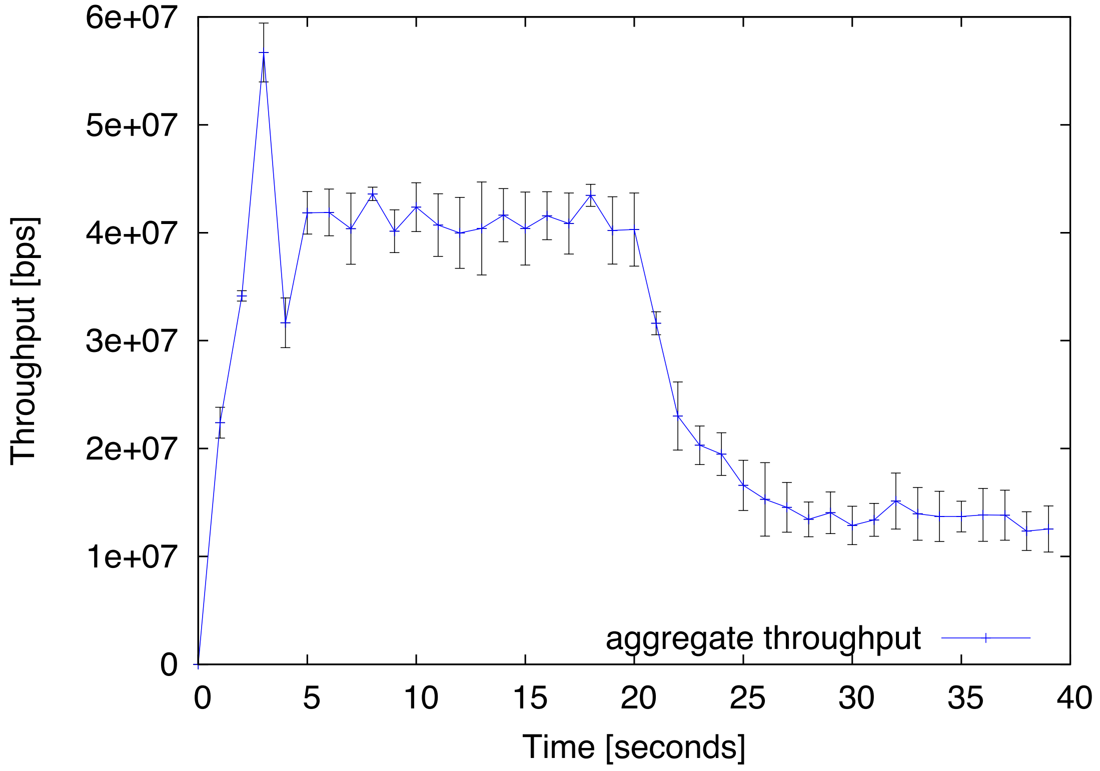


Figure 5.33: Throughput history for first approach

work paths start dropping packets because background flows start to run in the simulation system, with 10 TCP flows in each path. These simulated TCP flows have a large congestion window and thus can impose heavy packet drops over the network links they traverse. There is a substantial throughput decay at second 21 because the sender started sending packets using the third path and TCP needs some RTTs to reach its optimal transmission rate. After that, the throughout goes up to almost the previous levels. This is not possible however, because the sender needs to use one congested path among the two available. Another factor that causes the throughput not to reach its optimal is the multiple switches from one path to another after second 21. This happens because after initial loss, TCP adapts and even if the available bandwidth is low, the packet loss may be zero until TCP suffers another loss, thus deceiving the sender that the path is not congested when actually it is. We tested our second argument for decreased throughput, i.e., multiple switches, and thus correlate the number of switches with the average throughput obtained in a single run and obtained a weak correlation coefficient of -0.24; which indicates that indeed as the number of switches increases, the average throughput decreases.

Finally, the third strategy uses available bandwidth as the criteria to choose paths. It uses the two paths with higher available bandwidth. The setup is similar to the other two cases previously described and the results are shown in Fig. 5.35. Now, we can observe a big improvement from the second approach that uses path drop probability to determine the paths to use. In this case, the sender does not switch paths as in the second approach and thus achieves better throughput. In a real implementation, this approach could be the hardest the implement and the one that demands more control packets exchanges between special routers and end hosts and the trade-offs would have to be carefully evaluated.

From the above test case, we show that SymbioSim may be used to thoroughly evaluate new protocols and applications. Not only SymbioSim provides the network model where new designs can be tested, with potentially detailed and complex background traf-

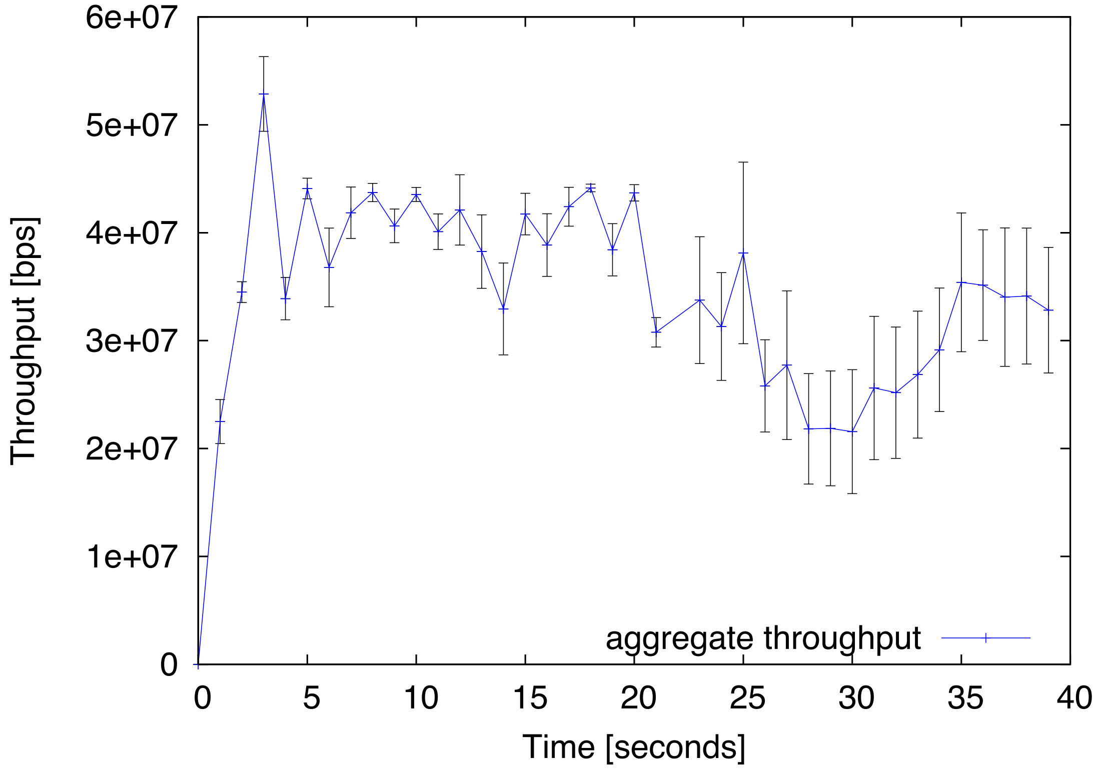


Figure 5.34: Throughput history for second approach

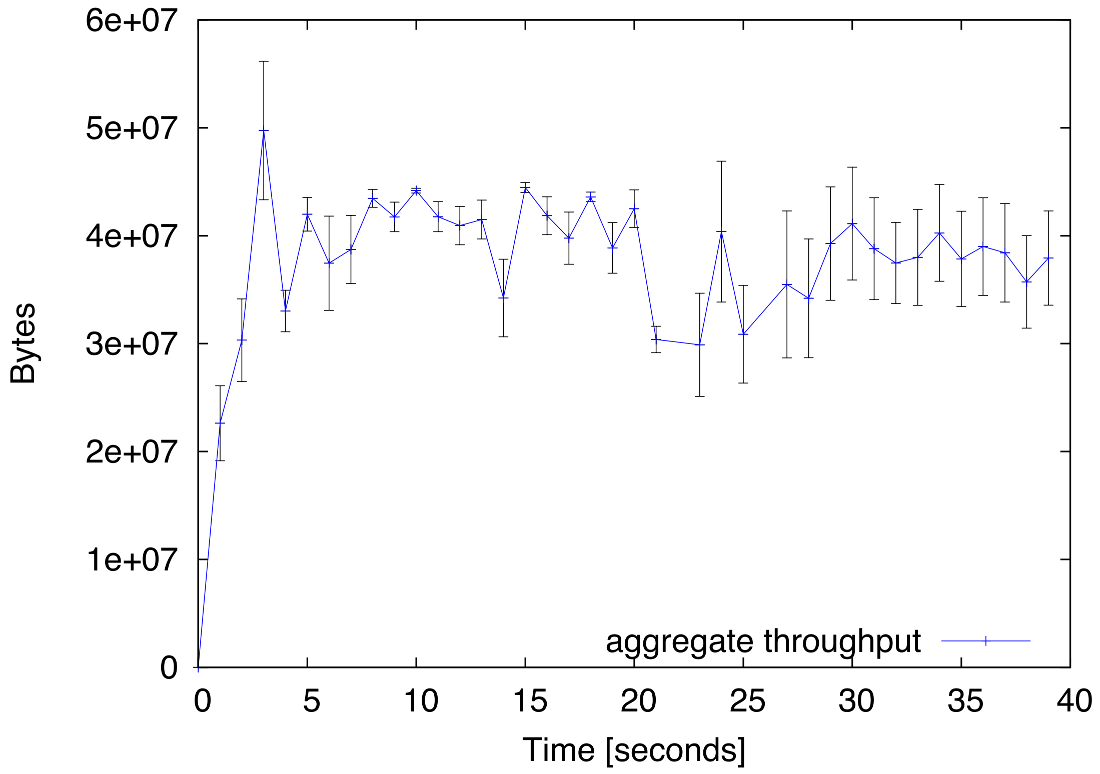


Figure 5.35: Throughput history for third approach

fic, but also provides means to measure and thus evaluate the performance of such new designs.

5.9 Summary and Conclusions

In this chapter we present SymbioSim, our approach towards achieving more scalable network emulation experiments using the symbiotic simulation paradigm.

We describe our closed-form model, and extension, that guarantees that both the original and the downscaled models impose same delay and packet loss to emulated flows. Aiming towards reducing the number of computational resources needed, we present our approach for downscaling the original model and allow real-time calibrations ($DSCALEd^R$). A dynamic sampling approach is also described and tested which aims at reducing the overhead of running $DSCALEd^R$. We describe how we scale better than current approaches by never sending payloads between the physical system and the simulator. We extensively validate our model in presence of different traffic intensities and conclude that our model model behaves correctly even if the inputs are not exponentially distributed. Finally, we validate our prototype implementation in Emulab. We compare the output of our system to that of pure simulation in terms of sequence number histories, and RTTs. Our results confirm the validity of our model and the correctness of the implementation.

SymbioSim's capabilities are used to rapidly prototype and test a simple multi-path protocol. The fact that the model in simulation can be easily and quickly changed without affecting the topology used in the physical system, plus the easiness for imposing background traffic with specific characteristics, and also the number of statistics and measurements that are exported from the simulation system, make SymbioSim a suitable tool for testing new protocols and applications.

CHAPTER 6

CONCLUSIONS

This chapter presents a brief summary of this dissertation and future directions the research could be taken.

6.1 Summary

The focus of this dissertation is on improving large-scale network simulation and emulation. Specifically, we:

1. Identify and quantify the limitations of current software-based emulators. First, we measure the maximum rate at which packets can be inserted into the simulator and determine that it is around 200Mbps for our simulator. Second, we realize that large-scale network emulation experiments are severely bounded by the available hardware constraints. In order to address these issues we do the following:

    - We propose a mechanism to reproduce traffic inside the simulator from traffic matadata. In this way, the payload of packets and most of the header never gets inserted into the simulator thus overcoming its inherent limitations measured previously.
    - We propose a novel approach for downscaling networks, which does not need knowing the bottleneck link in advance as previous approaches. In consequence, only a shorter version of the original target network topology needs to be instantiated in the physical network.

2. Expose the drawbacks of current approaches for modeling a network path, i.e., using a sequence of link emulators and filling a link with unresponsive traffic. In response:

- We propose a novel way for emulating a network path based on a M/D/1 queuing model. We test our approach in diverse scenarios for many types of inputs, with and without congestion, and conclude that it provides a very good approximation yielding small errors in terms of delay, drop, and throughput.
- We incorporate this approach into our SymbioSim simulator. SymbioSim is a system based on the symbiotic simulation paradigm that allows larger network experiments than those possible with current software-based emulators.

3. Address the lack of a mechanism to instantiate network experiments, with flexible and configurable network conditions, in large-scale settings. In particular we:
   - Design, and develop the PrimoGENI system as part of the GENI project. This system is composed of a high performance distributed simulator, a front end that we call Slingshot, and a customized OS image installed in GENI sites.
   - Instantiate network experiments, involving simulated and real entities, encompassing geographically distant locations. In order to achieve seamless communication between simulated and real hosts, and vice versa, we port sixteen Linux TCP congestion control algorithms to our simulator and extend them to handle real packets.
4. Enhance the scalability of software-based emulators by increasing the number of events that a discrete-event simulator can process per second. In particular, we:
   - We design, implement, and test SVEET, a performance evaluation testbed where real implementations of applications and protocols can be accurately evaluated under diverse network configurations and workloads from real applications in large-scale network settings enabled by the use of time dilation.

6.2 Future Directions

The research presented in this dissertation can be extended in the following directions:

1. Although the accuracy and usefulness of time dilation was tested for TDF factors less than 10, the improvement was shown not to be linear. PRIME, specifically SVEET, could achieve much greater network experiments from a more scalable, linear, time dilation approach.

2. The accuracy of our TCP implementation has been validated thoroughly in different scenarios. However, the implementation is very detailed, including complex data structures for bookkeeping, thus preventing the simulator to scale to larger sizes. A systematic profiling of the code, under diverse circumstances, may lead to identification of bottlenecks and eventually to a highly optimized implementation. Furthermore, if scalability if the ultimate goal, a more radical approach may be needed to decrease the complexity of the implementation without sacrificing accuracy.

3. In our research group, we observed that when dealing with thousands of simultaneous TCP flows, some unfair behavior is observed favoring those flows that started earlier. We have not gathered enough evidence to claim whether this behavior is the expected. A large-scale real testbed (with a set of scripts and processes in place) is needed to determine whether this phenomena is an artifact. In that sense, a bold move is needed to validate simulated TCP fairness behavior in large-scale settings.

4. We currently have two ways of injecting real packets into PRIME, which may be further optimized, thus benefiting the PrimoGENI and PRIME systems.

5. Currently, we use DummyNet to emulate a link. Extending its code to further understand PRIME demands regarding drop rate, and delay, would allow SymbioSim to achieve even more accurate and tight results. Furthermore, DummyNet could

be extended to include code to output several necessary statistics that may help experimenters, e.g., delay.

6. The real testbed controlled by SymbioSim's simulation system is currently closed to interaction with other testbeds not calibrated by the instance of SymbioSim's simulation system. A new approach that accomplishes this task would greatly increase the scale of experiments, thus insertion of SymbioSim in a federation, such as GENI, would be possible. Also, interconnecting testbeds calibrated by different simulators would ne possible.

BIBLIOGRAPHY

[ATCL08] Heiko Aydt, Stephen John Turner, Wentong Cai, and Malcolm Yoke Hean Low. Symbiotic simulation systems: An extended definition motivated by symbiosis in biology. In *PADS '08: Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation*, pages 109–116, Washington, DC, USA, 2008. IEEE Computer Society.

[BBSW05] Josep M. Blanquer, Antoni Batchelli, Klaus Schauser, and Rich Wolski. Quorum: flexible quality of service for internet services. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 159–174, Berkeley, CA, USA, 2005. USENIX Association.

[BDF$^+$03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, 2003.

[BEF$^+$00] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in network simulation. *IEEE Computer*, 33(5):59–67, 2000.

[BL96] D.A. Berry and B.W. Lindgren. Statistics: theory and methods. 1996.

[BL03] Paul Barford and Larry Landweber. Bench-style network research in an Internet instance laboratory. *ACM SIGCOMM Computer Communication Review*, 33(3):21–26, 2003.

[BPS06] Ashwin Bharambe, Jeffrey Pang, and Srinivasan Seshan. Colyseus: a distributed architecture for online multiplayer games. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 12–12, Berkeley, CA, USA, 2006. USENIX Association.

[BTI$^+$02] Chadi Barakat, Patrick Thiran, Gianluca Iannaccone, Christophe Diot, and Philippe Owezarski. A flow-based model for internet backbone traffic. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, IMW '02, pages 35–47, New York, NY, USA, 2002. ACM.

[CAI] Apache mina. `http://www.caida.org/data/passive/passive_2011_dataset.xml`.

[CNO99] James Cowie, David Nicol, and Andy Ogielski. Modeling the global Internet. *Computing in Science and Engineering*, 1(1):42–50, January 1999.

[CPZ02] Baek-Young Choi, Jaesung Park, and Zhi-Li Zhang. Adaptive random sampling for load change detection. *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 272–273, 2002.

[CPZ03] Baek-Young Choi, Jaesung Park, and Zhi-Li Zhang. Adaptive random sampling for traffic load measurement. *Communications, 2003. ICC '03. IEEE International Conference on*, 3:1552 – 1556 vol.3, may 2003.

[CS03] Mark Carson and Darrin Santay. Nist net: a linux-based network emulation tool. *SIGCOMM Comput. Commun. Rev.*, 33:111–126, July 2003.

[EL10a] Miguel A. Erazo and Jason Liu. A model-driven emulation approach to large-scale tcp performance evaluation. In *International Journal of Communication Networks and Distributed Systems (IJCNDS)*, pages 130–150. Inderscience, 2010.

[EL10b] Miguel A. Erazo and Jason Liu. On enabling real-time large-scale network simulation in geni: the primogeni approach. In *SIMUTools '10: Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, pages 1–2. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2010.

[EL13] Miguel A. Erazo and Jason Liu. Leveraging symbiotic relationship between simulation and emulation for scalable network experimentation. *to appear in Principles of Advanced and Distributed Simulation (PADS)*, 2013.

[ELL09] Miguel A. Erazo, Yue Li, and Jason Liu. Sveet! a scalable virtualized evaluation environment for tcp. In *Testbeds and Research Infrastructures for the Development of Networks and Communities, International Conference on*, volume 0, pages 1–10. IEEE Computer Society, 2009.

[Eun03] N.B. Eun, D.Y.; Shroff. Simplification of network analysis in large-bandwidth systems. In *INFOCOM*, 2003.

[Fal99] Kevin Fall. Network emulation in the vint/ns simulator. In *Proceedings of the fourth IEEE Symposium on Computers and Communications*, pages 244–250, 1999.

[FK03] Sally Floyd and Eddie Kohler. Internet research needs better models. *Computer Communication Review*, 33(1):29–34, 2003.

[FLA] Flack. `http://www.protogeni.net/trac/protogeni/wiki/Flack`.

[FLPU02] R. Fujimoto, D. Lunceford, E. Page, and A. M. Uhrmacher. Grand challenges for modeling and simulation. In *Technical Report No. 350, Schloss Dagstuhl*, 2002.

[FML$^+$03] C. Fraleigh, S. Moon, B. Lyles, C. Cotton, M. Khan, D. Moll, R. Rockell, T. Seely, and S.C. Diot. Packet-level traffic measurements from the sprint ip backbone. *Network, IEEE*, 17(6):6 – 16, nov.-dec. 2003.

[FTD03] C. Fraleigh, F. Tobagi, and C. Diot. Provisioning ip backbone networks to support latency sensitive traffic. In *INFOCOM*, 2003.

[FUN] Funionfs. `http://funionfs.apiou.org/?lng=en`.

[GEN] Global environment for network innovations. `http://www.geni.net`.

[GR05] C. Gkantsidis and P.R. Rodriguez. Network coding for large scale content distribution. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, 2005.

[GRE] Generic routing encapsulation (gre). `http://tools.ietf.org/html/rfc2784.html`.

[GSS06] P. Brighten Godfrey, Scott Shenker, and Ion Stoica. Minimizing churn in distributed systems. *SIGCOMM Comput. Commun. Rev.*, 36:147–158, August 2006.

[Gu07] Yan Gu. Rosenet: A remote server-based network emulation system. In *Georgia Institute of Technology - PhD Dissertation*, 2007.

[GYM$^+$06] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex Snoeren, Amin Vahdat, and Geoffrey Voelker. To infinity and beyond: time-warped network emulation. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI'06)*, 2006.

[HLRX07] Sangtae Ha, Long Le, Injong Rhee, and Lisong Xu. Impact of background traffic on performance of high-speed TCP variant protocols. *Computer Networks*, 51(7):1748–1762, 2007.

[HSH+06] Huaizhong Han, Srinivas Shakkottai, C. V. Hollot, R. Srikant, and Don Towsley. Multi-path tcp: a joint congestion control and routing scheme to exploit path diversity in the internet. *IEEE/ACM Trans. Netw.*, 14(6):1260–1271, December 2006.

[Kes04] Isaac Keslassy. Sizing router buffers. In *in Proceedings of ACM SIGCOMM*, pages 281–292, 2004.

[KKS+04] Min Sik Kim, Taekhyun Kim, YongJune Shin, Simon S. Lam, and Edward J. Powers. A wavelet-based approach to detect shared congestion. *SIGCOMM Comput. Commun. Rev.*, 34:293–306, August 2004.

[KS94] M. Katz and C. Shapiro. Systems competition and network effects. *Journal of Economic Perspectives*, 8(2):93–115, 1994.

[LAJS03] Long Le, Jay Aikat, Kevin Jeffay, and F. Donelson Smith. The effects of active queue management on web performance. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '03, pages 265–276, New York, NY, USA, 2003. ACM.

[Liu] Jason Liu. The prime research. http://www.cis.fiu.edu/prime/.

[LLS07] Yee-Ting Li, Douglas Leith, and Robert N. Shorten. Experimental evaluation of TCP protocols for high-speed networks. *IEEE/ACM Transactions on Networking*, 15(5):1109–1122, 2007.

[LMVH07a] Jason Liu, Scott Mann, Nathanael Van Vorst, and Keith Hellman. An open and scalable emulation infrastructure for large-scale real-time network simulations. In *Proceedings of IEEE INFOCOM MiniSymposium*, pages 2476–2480, 2007.

[LMVH07b] Jason Liu, Scott Mann, Nathanael Van Vorst, and Keith Hellman. An open and scalable emulation infrastructure for large-scale real-time network simulations. In *In Proceedings of IEEE INFOCOM MiniSymposium*, 2007.

[LYPN02] M. Liljenstam, Y. Yuan, BJ Premore, and D. Nicol. A mixed abstraction level simulation model of large-scale internet worm infestations. In *Proceedings of the 10th Annual International Symposium on Modeling, Analysis and Simulation on Computer and Telecommunication Systems (MASCOTS)*, 2002.

[MIN] The caida anonymized internet traces 2011 dataset. `http://mina.apache.org/`.

[MTS$^+$02] A. Medina, N. Taft, K. Salamatian, S. Bhattacharyya, and C. Diot. Traffic matrix estimation: existing techniques and new directions. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '02, pages 161–174, New York, NY, USA, 2002. ACM.

[OPEa] Open source vpn. `http://www.openvpn.net/`.

[OPEb] Openvz. `http://wiki.openvz.org/`.

[OPN] Opnet: Application and network performance. http://www.opnet.com/.

[PACR02] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A blueprint for introducing disruptive technology into the Internet. In *Proceedings of the 1st Workshop on Hot Topics in Networking (HotNets-I)*, 2002.

[PAK07] Himabindu Pucha, David G. Andersen, and Michael Kaminsky. Exploiting similarity for multi-source downloads using file handprints. *CiteSeerX - Scientific Literature Digital Library and Search Engine [http://citeseerx.ist.psu.edu/oai2] (United States)*, 2007.

[PFTK00] Jitendra Padhye, Victor Firoiu, Donald F. Towsley, and James F. Kurose. Modeling tcp reno performance: a simple model and its empirical validation. *IEEE/ACM Trans. Netw.*, 8:133–145, April 2000.

[PMF$^+$02] K. Papagiannaki, S. Moon, C. Fraleigh, P. Thiran, F. Tobagi, and C. Diot. Analysis of measured single-hop delay from an operational backbone network. In *INFOCOM*, 2002.

[PPG06] F. Papadopoulos, K. Psounis, and R. Govindan. Performance preserving topological downscaling of internet-like networks. In *IEEE Journal on Selected Areas in Communications*, volume 24, 2006.

[PRO] Protogeni. http://www.protogeni.net/trac/protogeni.

[RFI02] Matei Ripeanu, Ian Foster, and Adriana Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing special issue on Peer-to-Peer Networking*, 6(1):50–57, 2002.

[Ril03] George F. Riley. The Georgia Tech network simulator. In *Proceedings of the ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research (MoMeTools'03)*, pages 5–12, 2003.

[Riz97] Luigi Rizzo. Dummynet: a simple approach to the evaulation of network protocols. *ACM SIGCOMM Computer Communication Review*, 27(1):31–41, 1997.

[RWH09] Costin Raiciu, Damon Wischik, and Mark Handley. Practical congestion control for multipath transport protocols. *University College of London Technical Report*, 2009.

[SB04] Joel Sommers and Paul Barford. Self-configuring network traffic generation. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, IMC '04, pages 68–81, New York, NY, USA, 2004. ACM.

[SDRL09] Pramod Sanaga, Jonathon Duerig, Robert Ricci, and Jay Lepreau. Modeling and emulation of internet paths. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 199–212, Berkeley, CA, USA, 2009. USENIX Association.

[SHC$^+$04] Victor Shnayder, Mark Hempstead, Bor-rong Chen, Geoff Werner Allen, and Matt Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, 2004.

[TCP] Tcpdump and libpcap. `http://www.tcpdump.org/`.

[TES] Protogeni test scripts. `http://www.protogeni.net/trac/protogeni/wiki/TestScripts`.

[TKAP06] Niraj Tolia, Michael Kaminsky, David G. Andersen, and Swapnil Patil. An architecture for internet data transfer. In *Proceedings of the 3rd conference* *on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.

[VV08] Kashi Venkatesh Vishwanath and Amin Vahdat. Evaluating distributed systems: Does background traffic matter. In *Proceedings of the 2008 USENIX Technical Conference*, pages 227–240, 2008.

[VV09] K.V. Vishwanath and A. Vahdat. Symbiotic simulation systems: An extended definition motivated by symbiosis in biology. In *Networking, IEEE/ACM Transactions on*, 2009.

[VVEL11] Nathanael Van Vorst, Miguel A. Erazo, and Jason Liu. Primogeni: Integrating real-time network simulation and emulation in geni. In *Principles of Advanced and Distributed Simulation (PADS), 2011 IEEE Workshop on*, pages 1–9. IEEE, 2011.

[VVEL12] Nathanael Van Vorst, Miguel A. Erazo, and Jason Liu. Primogeni for hybrid network simulation and emulation experiments in geni. In *Journal of Simulation*, pages 179–192. Nature Publishing Group, 2012.

[VYW$^+$02] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. *SIGOPS Oper. Syst. Rev.*, 36(SI):271–284, 2002.

[WC06] David X. Wei and Pei Cao. NS-2 TCP-Linux: An NS-2 TCP implementation with congestion control algorithms from Linux. In *Proceedings of the 2nd International Workshop on NS-2 (WNS2)*, 2006.

[WEB] The web100 project. http://www.web100.org/.

[WGN$^+$02] Brian White, Shashi Guruprasad, Mac Newbold, Jay Lepreau, Leigh Stoller, Robert Ricci, Chad Barb, Mike Hibler, and Abhijeet Joglekar. Netbed: an integrated experimental environment. *SIGCOMM Comput. Commun. Rev.*, 32(3):27–27, 2002.

[WLS$^+$02] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.

[YKKY08] Tao Ye, Hema T. Kaur, Shivkumar Kalyanaraman, and Murat Yuksel. Large-scale network parameter configuration using an on-line simulation framework. *IEEE/ACM Trans. Netw.*, 16(4):777–790, 2008.

[ZD01] Yin Zhang and Nick Duffield. On the constancy of internet path properties. In *IMW '01: Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 197–211, New York, NY, USA, 2001. ACM.

[ZNA03] Moshe Zukerman, Timothy D. Neame, and Ronald G. Addie. Internet traffic modeling and future technology implications. In *INFOCOM*, 2003.

VITA

MIGUEL A. ERAZO

Born, La Paz, Bolivia

2003	B.Sc., Electrical Engineering Universidad Mayor de San Andres La Paz, Bolivia
2006	M.S., Computer Engineering University of Puerto Rico at Mayaguez Mayaguez, Puerto Rico
2012	Ph.D., Computer Science Florida International University Miami, Florida, US

PUBLICATIONS AND PRESENTATIONS

Miguel A. Erazo and Jason Liu. Leveraging symbiotic relationship between simulation and emulation for scalable network experimentation. to appear in Principles of Advanced and Distributed Simulation (PADS), 2013.

Miguel A. Erazo, Ting Li, Jason Liu, and Stephan Eidenbenz. Toward comprehensive and accurate simulation performance prediction of parallel file systems. In Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on, pages 1-12. IEEE, 2012.

Nathanael Van Vorst, **Miguel A. Erazo**, and Jason Liu. Primogeni for hybrid network simulation and emulation experiments in geni. In Journal of Simulation, pages 179-192. Nature Publishing Group, 2012.

Nathanael Van Vorst, **Miguel A. Erazo**, and Jason Liu. Primogeni: Integrating real-time network simulation and emulation in geni. In Principles of Advanced and Distributed Simulation (PADS), 2011 IEEE Workshop on, pages 1-9. IEEE, 2011.

Miguel A. Erazo and Jason Liu. On enabling real-time large-scale network simulation in geni: the primogeni approach. In SIMUTools 10: Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques, pages 1-2. ICST (Institute for Computer Sciences, Social- Informatics and Telecommunications Engineering), 2010.

Miguel A. Erazo and Jason Liu. A model-driven emulation approach to large-scale tcp performance evaluation. In International Journal of Communication Networks and Distributed Systems (IJCNDS), pages 130-150. Inderscience, 2010.

Miguel A. Erazo and Roberto Pereira. On profiling the energy consumption of distributed simulations: A case study. In Proceedings of the 2010 IEEE/ACM Intl Conference on Green Computing and Communications & Intl Conference on Cyber, Physical and Social Computing, pages 133-138. IEEE Computer Society, 2010.

Miguel A. Erazo, Yi Qian, Kejie Lu, and Domingo Rodriguez. Enabling periodic monitoring applications in wireless sensor networks: Design and analysis of an efficient mac protocol. In Handbook on Sensor Networks, Edited by Yang Xiao, Hui Chen, and Frank H. Li, pages 747-764. World Scientific Publishing Co., 2010.

Miguel A. Erazo, Yue Li, and Jason Liu. Sveet! a scalable virtualized evaluation environment for tcp. In Testbeds and Research Infrastructures for the Development of Networks and Communities, International Conference on, volume 0, pages 1-10. IEEE Computer Society, 2009.

Miguel A Erazo, Yi Qian, Kejie Lu, and Domingo Rodriguez. Analysis and design of a mac protocol for wireless sensor networks with periodic monitoring applications. In Military Communications Conference, 2007. MILCOM 2007. IEEE, pages 1-7. IEEE, 2007.

Miguel A. Erazo and Yi Qian. Sea-mac: A simple energy aware mac protocol for wireless sensor networks for environmental monitoring applications. InWireless Pervasive Computing, 2007. ISWPC07. 2nd International Symposium on, pages 115-122. IEEE, 2007.