LEVERAGING SYMBIOTIC RELATIONSHIPS FOR EMULATION OF COMPUTER NETWORKS

A proposal submitted in partial fulfillment of the requirements for the degree of DOCTOR OF PHILOSOPHY in COMPUTER SCIENCE by Miguel A. Erazo Villegas 2010
To: Dean Amir Mirmiran  
College of Engineering and Computing

This proposal, written by Miguel A. Erazo Villegas, 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 proposal and recommend that it be approved.

______________________________
Shu-Ching Chen

______________________________
Zhenyu Yang

______________________________
Armando Barreto

______________________________
Jason Liu, Major Professor

Date of Proposal:

The proposal of Miguel A. Erazo Villegas is approved.

______________________________
Dean Amir Mirmiran  
College of Engineering and Computing

______________________________
Dean Lakshmi N. Reddi  
University Graduate School

Florida International University, 2010
ABSTRACT
LEVERAGING SYMBIOTIC RELATIONSHIPS FOR EMULATION OF COMPUTER NETWORKS

by
Miguel A. Erazo Villegas
Florida International University, 2010
Miami, Florida
Professor Jason Liu, Major Professor
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>2 LITERATURE REVIEW</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Symbiotic systems</td>
<td>3</td>
</tr>
<tr>
<td>2.1.1 Definition and Classification</td>
<td>3</td>
</tr>
<tr>
<td>2.1.2 Existing Symbiotic Approaches for Computer Network Emulation</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Network Experimentation Testbeds</td>
<td>7</td>
</tr>
<tr>
<td>2.2.1 Simulation Testbeds</td>
<td>7</td>
</tr>
<tr>
<td>2.2.2 Emulation Testbeds</td>
<td>8</td>
</tr>
<tr>
<td>2.2.3 Live Experimentation</td>
<td>9</td>
</tr>
<tr>
<td>2.2.4 Federated Approaches</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Topology Downscaling</td>
<td>10</td>
</tr>
<tr>
<td>2.4 Emulation of Internet Paths</td>
<td>12</td>
</tr>
<tr>
<td>2.5 Traffic Generators</td>
<td>15</td>
</tr>
<tr>
<td>3 RESEARCH PROBLEM</td>
<td>17</td>
</tr>
<tr>
<td>3.1 Motivation</td>
<td>17</td>
</tr>
<tr>
<td>3.2 Problem Definition</td>
<td>18</td>
</tr>
<tr>
<td>4 ON ENABLING TRAFFIC REPRODUCTION INSIDE THE SIMULATOR</td>
<td>21</td>
</tr>
<tr>
<td>4.1 Incorporating TCP algorithms</td>
<td>21</td>
</tr>
<tr>
<td>4.1.1 TCP Congestion Window Trajectories</td>
<td>23</td>
</tr>
<tr>
<td>4.1.2 Throughput versus Packet Loss</td>
<td>24</td>
</tr>
<tr>
<td>4.1.3 Recreating TCP formats and packet exchanges</td>
<td>27</td>
</tr>
<tr>
<td>4.2 SVEET</td>
<td>27</td>
</tr>
<tr>
<td>4.2.1 Design Guidelines</td>
<td>29</td>
</tr>
<tr>
<td>4.2.2 Architecture</td>
<td>29</td>
</tr>
<tr>
<td>4.2.3 Time Dilation</td>
<td>30</td>
</tr>
<tr>
<td>4.2.4 Validation of Accuracy</td>
<td>32</td>
</tr>
<tr>
<td>4.2.5 Assessing time dilation effectiveness</td>
<td>33</td>
</tr>
<tr>
<td>4.2.6 Case Studies</td>
<td>35</td>
</tr>
<tr>
<td>4.3 Summary and Remaining work</td>
<td>42</td>
</tr>
</tbody>
</table>
# PrimoGENI: ON ENABLING SYMBIOTIC RELATIONSHIPS ON FEDERATED ENVIRONMENTS

5.1 Motivation .......................................................... 44
5.2 Introduction ......................................................... 45
5.3 Architecture .......................................................... 47
5.4 Experiment Workflow .............................................. 48
5.5 Meta-Controllers ...................................................... 51
5.6 Experiments .......................................................... 53
5.6.1 Validation Studies ............................................... 54
5.6.2 Performance Studies ............................................. 57
5.7 Summary and Remaining Work .................................... 59

# SymbioSim: LEVERAGING SYMBIOTIC RELATIONSHIPS TOWARDS SCALABLE NETWORK EMULATION EXPERIMENTS

6.1 Motivation for Symbiotic Systems ................................ 62
6.2 Background .......................................................... 63
6.2.1 Our Current Emulation Infrastructure ....................... 63
6.2.2 Symbiotic Relationships ......................................... 65
6.2.3 ROSENET .......................................................... 66
6.3 SymbioSim ............................................................. 70
6.3.1 SymbioSim Overview ............................................. 70
6.3.2 Fostering symbiotic relationships ............................. 72
6.3.3 Downscaling Topologies ......................................... 75
6.3.4 Limiting incoming traffic ...................................... 81
6.4 Implementation ....................................................... 83
6.4.1 Design Guidelines ............................................... 83
6.4.2 Subsystems ........................................................ 83
6.5 Validation ............................................................. 87
6.6 Conclusions and Future Work ..................................... 90

# SUMMARY ................................................................. 92

# BIBLIOGRAPHY .......................................................... 93
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>7</td>
</tr>
<tr>
<td>2.4</td>
<td>12</td>
</tr>
<tr>
<td>4.1</td>
<td>22</td>
</tr>
<tr>
<td>4.2</td>
<td>23</td>
</tr>
<tr>
<td>4.3</td>
<td>25</td>
</tr>
<tr>
<td>4.4</td>
<td>26</td>
</tr>
<tr>
<td>4.5</td>
<td>28</td>
</tr>
<tr>
<td>4.6</td>
<td>30</td>
</tr>
<tr>
<td>4.7</td>
<td>31</td>
</tr>
<tr>
<td>4.8</td>
<td>33</td>
</tr>
<tr>
<td>4.9</td>
<td>34</td>
</tr>
</tbody>
</table>
6.2 Our emulation infrastructure ........................................ 65

6.3 ROSENET test experimental setup .................................. 67

6.4 Shared bottlenecks not modeled ..................................... 68

6.5 Proposed closed-loop symbiotic simulation system ............. 71

6.6 Abstracting a subpath into a single link .......................... 78

6.7 Downscaling the network topology .................................. 80

6.8 Downscaling the network topology .................................. 82

6.9 Downscaling the network topology .................................. 84

6.10 Dumbbell topology used for validation. (a) Target network model with both simulated and real hosts. (b) Downscaled topology. (c) Instantiated topology on Emulab using a delay node ............... 86

6.11 Throughput achieved under random packet loss (10 Mb/s bottleneck link bandwidth) ........................................... 89

6.12 Throughput achieved under random packet loss (10 Mb/s bottleneck link bandwidth) ........................................... 90
**LIST OF ACRONYMS**

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ns-2</td>
<td>Network Simulator 2</td>
</tr>
<tr>
<td>SSANS</td>
<td>Symbiotic Simulation Anomaly Detection System</td>
</tr>
<tr>
<td>SSCS</td>
<td>Symbiotic Simulation Control System</td>
</tr>
<tr>
<td>SSDSS</td>
<td>Symbiotic Simulation Decision Support System</td>
</tr>
<tr>
<td>SSF</td>
<td>Scalable Simulation Framework</td>
</tr>
<tr>
<td>SSFC</td>
<td>Symbiotic Simulation Forecast System</td>
</tr>
<tr>
<td>SSMVS</td>
<td>Symbiotic Simulation Model Validation System</td>
</tr>
<tr>
<td>SVEET</td>
<td>Scalable Virtualized Evaluation Environment for TCP</td>
</tr>
<tr>
<td>phy-EDSS</td>
<td>Physical System External Decision Support System</td>
</tr>
<tr>
<td>sim-EDSS</td>
<td>Simulator External Decision Support System</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

The tremendous success of the Internet over the past decades is undeniable and can be attributed to many contrasting factors. Nevertheless, this very same success and the evolution point the Internet has reached prevent even the simplest changes and modifications to be implemented. A classic example is IPV6, a relative modest change which has proved quite difficult for ubiquitous deployment. Consequently, in order to break the Internet ossification, many research and development efforts have been launched in recent years to foster network innovations.

Research on network testbeds has made steady progress. Physical testbeds, such as PlanetLab [41], provide realistic network environments (including live traffic) for testing network protocols and distributed services. Emulation testbeds, such as Emulab [55] and ModelNet [49], allow more flexible and controlled network experiments. GENI [3] capitalizes on the success of these previous efforts and aims to provide an overarching technology that brings all different network testbeds together as a single collaborative and exploratory platform for implementing and testing new network designs and technologies.

Aligned with the aforementioned efforts, in this document we propose a symbiotic approach as the core concept for building a testbed platform for new protocols and applications. Specifically, in order to enable more realistic experiments, we ported the Linux TCP protocol from the network simulator(ns-2) [16] [17]. Also, we are currently building PrimoGENI [18], a platform that automatically realizes network experiments involving simulated and emulated entities over a ProtoGENI [5]/Emulab enabled platform. In this way, our realistic TCP implementation allows us to reproduce transport flows inside the simulator with high accuracy. Furthermore, PrimoGENI helps us instantiate emulation experiments, not an easy task considering the number of processes involved and the order in which they must be started. Having these components into place, we designed and we are currently build-
ing SymbioSim, a testbed platform based upon symbiotic relationships between a physical system and a high-performance simulator.

This document is organized as follows. In chapter 2, we describe the most salient related work. Chapter 3 introduces the research problem, the goal of these thesis, and the five concrete objectives to achieve our goal. In chapter 4, we present our realistic TCP implementation and our time-dilation enabled testbed that we built (SVEET); which uses our TCP protocol. PrimoGENI, our approach for increasing increased accessibility for emulation experiments is described in chapter 5. SymbioSim is presented in chapter 6 as an alternative approach to classical 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, in [25] a symbiotic system is defined 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. [7] 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 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 are symbiotic simulation systems 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 [7] which are described 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 systems 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 are used to detect anomalies either in the underlying simulation model or the physical system. These system are depicted in Fig. 2.2.
2.1.2 Existing Symbiotic Approaches for Computer Network Emulation

An earlier system ROSENET [27] implemented a close cooperation between an emulation system and a simulator. Their objective is to achieve scalability, accuracy, and 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 [35], ns-2 [13], 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 it 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 (which can run on a different physical location than the simulator) while providing accuracy, scale, and timeliness. The simulator continuously 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 *LowFidelityModel* provided by the simulator that calibrates the emulated system, realized by the use of Dummynet [44]. At the same time, the simulator benefits from *TrafficSummaryModel* 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 [58]. This allows that a network model characterize the traffic dynamics between a time internal without losing too much accuracy. The *LowFidelity* model is sent at the end of each internal if it is decided that it is statistically different from the previous one.
With the system briefly summarized above, ROSENET claims to achieve accessibility, since real world applications can be 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 achieved through the use of a high-fidelity simulator.

2.2 Network Experimentation Testbeds

2.2.1 Simulation Testbeds

Three principal methods can be used to test the TCP performance: simulation, live experiments, and emulation. Federated approaches such as GENI [3], put together all these technologies in a single experimentation platform.

Discrete-event network simulators, such as ns-2 [13], SSFNet [15], and GTNetS [43], 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), for network experimentation; which subjects real applications, and protocols stacks to controllable and synthetic network topologies. In this section we describe the most salient emulation approaches.

ModelNet [49] is an emulation facility where unmodified applications running on edge nodes route their traffic through core nodes; which emulate the target topology. 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 scale, 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 [55], an evolution of Netbed [54], is a software system composed of local (emulated) and geographically distributed nodes (wan) nodes. Virtual network topologies, specified with an ns [13] 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 ar-
bitrarry topologies. Simulated nodes, links, and traffic interacts with real applications using ns’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 [41].

PlanetLab’s aims to promote the introduction of disruptive technologies into the Internet through the use of overlay networks. 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 [3] is a set of network research infrastructure which aims to present this 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 (or 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 aims to integrate network systems that provide GENI key functionality. Although it initially relied heavily on Emulab infrastructure, it provided its own API to instantiate experiments (slices) within it. Our project, PrimoGENI uses the ProtoGENI control framework to manage, control and access the underlying resources.

2.3 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, \[8\], \[23\], \[40\], \[24\]. 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 \[19\].

Supported by the findings listed above, is that Papadopoulos et al. proposed a performance preserving topological downscaling approach in \[20\]. 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. operates 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 traversed 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 the propagation delay ($P_1$) plus the transmission delay ($packetSize(grp1)/C_1$).
Figure 2.4: (i) Original Network Topology. (ii) Downscaled topology using DSCALEd

DSCALEd 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.

2.4 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 [44], ModelNet [49], NIST Net [14], and Emulab[55]. These emulators focus on emulating a link with all detail. However, most applications are concerned with the end-to-end characteristics of the 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 [51], points out that many papers published in top venues [11], [12], [26], [42], [48], use the approach of approximating a path with a single link. However, as pointed out in [45], 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 [45], 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. [45] have 5 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 base RTT plus the maximum delay that can be introduced by the queues in the bottleneck links:

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

(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 queuing 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\[31\]. In consequence, the lower bound for the queues is given by:

$$q > \sum_{f \in F} \min(w_f, w_{\text{max}})$$

(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 [38]:

\[ RTT_{max} = \frac{w_{max}}{ABW} \]  \hspace{1cm} (2.3)

In steady state, Reno flows tend to reach a state in which the bottleneck queues are nearly full [38]. In consequence, those flows will experience a RTT near that given by equation 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:

\[ q_f + q_r \leq C * \left( \frac{w_{max}}{ABW_f} - RTT_{base} \right) \]  \hspace{1cm} (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 [32], 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.

2.5 Traffic Generators

Traffic generation is important in many scenarios, including: capacity planning [37], queue management studies [33], bandwidth estimation tools, network emulation [55][54][49], and network simulation [13][35].

A salient example of a traffic generator is Harpoon [46]. It 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 [52] is a traffic generator that aims at generating realistic traces that accurately represent the following characteristics 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 [52] 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 [46], 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

RESEARCH PROBLEM

This section introduces the research problem of this proposal. We start with the problem motivation, and then formulate the problem definition accordingly.

3.1 Motivation

Recent network research and engineering initiatives such as GENI [3] and Fire [2] are two examples of the tremendous effort being invested for providing the networking community testbed platforms for network research. In this sense, a platform for evaluating network protocols and applications in highly scalable, controllable, accessible settings is much needed by networking research community.

Networking researchers use three principal methods to evaluate protocols: live experiments, simulation, and emulation. Live experiments on existing research testbeds, such as PlanetLab [11] and VINI [10], provide protocol designers with realistic distributed environments and traffic conditions that resemble the target system on which the protocols are deployed. These testbeds, however, do not provide the level of reproducibility, controllability, and flexibility necessary for testing and benchmarking protocols and applications under diverse network conditions. In contrast, network simulators, such as ns-2 [13], SSFNet [15], and GTNetS [43], 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 stack.

Alternatively, emulation testbeds, such as Emulab [55] and ModelNet [49], allow flexible network experiments that directly involve real protocol implementations. However, in the
the case of Emulab, the scale of the experiments that can be realized is small since a whole
physical machine is dedicated to a either a host, router, or traffic generator. Also, not very
complicated network topologies can be achieved for the very same reason described above.
Finally, live networks such as PlanetLab achieve realism, but surrender repeatability and the
ability to modify or even monitor internal router behavior.

With the exception of possibly GENI and FIRE, none of the aforementioned approaches
achieves large-scale, controllable, and reproducible environments for testing new network
protocols and applications. Simulation offers most of this advantages but the accuracy of
the network protocols and applications implemented there is questionable unless they are
subjected to rigorous verification and given that most of the simulators cannot run on dis-
tributed fashion, the scalability is seriously compromised. On the other hand, although
emulation testbeds provide a high level of accessibility, they fail to achieve large scale sce-
narios.

In this dissertation proposal, we propose an orthogonal approach to GENI and FIRE, based on symbiotic relationships, to increase the scalability of emulation experiments. To
that end, we first implemented a realistic TCP implementation inside the simulator and
developed PrimoGENI as a platform for automatically instantiating emulation experiments
and deliver commands between symbiotic subsystems.

3.2 Problem Definition

Having introduced the motivation of this proposal, we now state the two main research
problems of this dissertation:

1. *Although many research efforts have been devoted to provide the networking research
community with a suitable testbed where new network protocols and applications can
be tested under realistic conditions, none of them succeeded at providing scalability in
accessible and controllable scenarios.*
2. In the context of computer networks, no system promotes and realizes a high cooperation between simulation and physical systems for scalable emulation of computer network.

To address the identified research problems, we have formulated the goal of this dissertation as follows:

The goal of this dissertation is to design, implement, and evaluate a new testbed for evaluation of new protocols and applications by promoting close cooperation (symbiosis) between a physical system, wherein real applications will run, and a high performance simulator, where simulated network elements are instantiated, aiming at achieving large-scale scenarios where real applications can be tested in an accessible and controllable way.

In order to achieve our goal, we must achieve the following milestones:

- **Assess the limits of our current emulation technology.** The precise limits at which we can inject traffic into our simulator are our baseline against which we will evaluate our new approach.

- **Implement a realistic TCP in our simulator.** A realistic transport protocol that mimics a real implementation is crucial to reproduce the traffic originated by real applications, using the real stack of TCP/IP protocols, inside the simulator.

- **Design and build a system to make emulation experiments more accessible to research community.** Manually instantiating large-scale emulation experiments can be cumbersome and time-consuming. Even more difficult is connecting with other testbeds to form a larger experiment involving different technologies. A tool to speed-up this process is needed.

- **Design, implement, and evaluate a system that truly promotes cooperation between a simulator and a physical system in a mutual beneficial way.** A direct consequence of such a system will be an increased amount of traffic between real applications that can
be handled by the simulator. Also, we expect to decrease the amount of computing resources needed.

• *Instantiate a large-scale scenario that showcases the potential of symbiotic simulation.*

A successful instantiation of a large-scale experiment will be the final milestone to be achieved.
ON ENABLING TRAFFIC REPRODUCTION INSIDE THE SIMULATOR

As explained in detail in chapter 6, in order to enable symbiotic simulation and eliminate the bottleneck between the simulator and the physical system, we must be able to reproduce the real traffic originated from the real applications inside the simulator. For that purpose, a realistic TCP implementation must run inside the simulator. Specifically, we ported Linux TCP algorithms to our high-performance simulator. This also enables that both simulated and real applications to seamlessly interact with each other at transport-layer level.

We tested our implementation using SVEET [16], a system that enables communications between real and simulated applications which uses time dilation to increase emulation scalability. We describe SVEET in more details in this chapter.

4.1 Incorporating 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 ported real Linux TCP algorithms into our simulator, PRIME [35]. We performed this task in two steps: in the first one we ported the ns-2 implementation to PRIME; in the second step, we preserved the original algorithms but augmented the code in order to enable seamless communication between real and simulated hosts. We detail the implementation of TCP in the following paragraphs.

We followed the same design principle as in [53] for porting Linux TCP variants to PRIME. In fact, we reused as many data structures as possible from the Linux TCP port to ns-2 [13]. Fig. 4.1 shows the code structure. 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, LinuxTcpMaster, to manage all active Linux TCP
connections, and another protocol session, `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 was structured in the same fashion as in ns-2: we used `LinuxTcpAgent` to represent the TCP sender-side logic and `SinkTcpAgent` to represent the receiver-side logic. In this way we achieved maximum reuse of the existing code from the Linux TCP implementation in ns-2. The congestion control mechanisms of the TCP variants were 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 refurbished these facilities for them to run in PRIME.

Figure 4.1: Linux TCP class structure.
We validated our implementation by performing a set of tests which are described in the following subsections.

4.1.1 TCP Congestion Window Trajectories

Our first set of experiments aim to provide the baseline comparison between pure simulation results produced by PRIME, ns-2 (whose TCP implementation has already been benchmarked from a Dummynet [44] testbed on a previous work [53]), 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. 4.2 which is similar to the one used in a previous study [53]. 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. 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 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 [6] to collect the congestion window size at the virtual machines. Fig. 4.3 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 on 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. 4.3, 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.

4.1.2 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, but applied random packet drops according to a specified probability. We varied 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.

Fig. 4.4 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.
Figure 4.3: Congestion window trajectories of Linux TCP variants.
The results obtained so far show conclusively that our implementation of TCP behaves as the real ones available in Linux. However, this 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 been 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.

4.1.3 Recreating TCP formats and packet exchanges

Although TCP behavior of our implementation resembles that of real TCP, all the listed problems at the end of the previous section prevent simulated hosts to exchange packets with simulated hosts. In order to fix this problem, we augmented significantly this initial implementation published in [16]. The class diagram is shown in 4.5.

In this new implementation, we performed some fundamental changes:

- To enable bi-directional communication, the old sender 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.

- \textit{TCPSndWnd} and \textit{TCPSndWnd} are complex data structures that buffer real data.

We are currently in the process of evaluating our new implementation.

4.2 SVEET

In [16] we presented 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. SVEET uses our Linux TCP implementation and is used to conduct validation and testing.
Figure 4.5: New TCP class diagram.
4.2.1 Design Guidelines

We conceived SVEET to meet with the following requirements:

- 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.

4.2.2 Architecture

Fig. 4.6 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 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.

4.2.3 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 adopted the time dilation technique by Gupta et al. in [28]. This technology can uniformly slow the passage of time from the perspective of the guest operating system (Xeno-
Linux). Time dilation can scale up the perceived I/O rate as well as the perceived processing power on the virtual machines. The ratio between the rate at which time passes in the physical world to the operating systems perception of time is called the time dilation factor (TDF). On Fig. 4.7, 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 [9] was 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.

Figure 4.7: SVEET Architecture.
This Linux kernel comes with all the TCP variants which we would like to include in our experiments. We also equipped the Linux kernel with Web100, 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.

4.2.4 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 performed this experiments using a TDF of 10. To that end, we created a dumbbell topology (similar to the one used in a recent TCP performance study by Li et al.) 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 selected 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 established a separate TCP flow to 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 measured the changes to the congestion window size over time at the senders of both flows.

Fig. 4.8 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.
4.2.5 Assessing time dilation effectiveness

In this section we evaluate SVEET’s ability to deal with real traffic when running on a single physical machine. This evaluation work was presented in [17]. We chose the simple dumbbell network topology with a one-to-one mapping between the servers on the left side of the dumbbell and the clients on the right side. The servers and clients were either simulated or emulated. We fixed the ratio of emulated server-client pairs to simulated pairs to be 1:20. The clients and servers were connected through two simulated routers in the middle. In this case, the emulated and simulated traffic was multiplexed at the bottleneck link (middle link). We designated 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 varied the number of servers and clients according to different
traffic demands. We recorded the traffic intensity supported by the emulation infrastructure while the system were 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 was designed in such a way that the maximum achievable traffic through the bottleneck link would be limited solely by the emulation infrastructure (which was 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) were 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 4.9 and 4.10 depict the aggregate and perflow throughput on the bottleneck link as a function of the number of simulated nodes. We performed 10 runs for each configuration and plotted the throughput with 95% confidence intervals resultant from 100 seconds of experimentation. As expected, as the traffic demand increased, the aggregate throughput increased linearly up to a certain limit. Beyond that limit, the emulation infrastructure could no longer sustain the traffic demand; TCP was scaled back leading to a reduced throughput. Fig. 4.9 clearly shows the achievable limits when run on a single machine with
different TDFs. Fig. 4.10 shows that the average throughput per flow remained constant before degrading progressively as the traffic increased. We observe that the variance of the throughput measurement was more pronounced when the maximum throughput was achieved. Also, we observe that, although the throughput increased with higher TDFs, this increase is not linear. The maximum aggregate throughput was approximately 160, 480, and 670 Mbps, corresponding to TDFs of 1, 5, and 10. In fact, as we increased TDF, a smaller gain was obtained in terms of the achievable throughput.

4.2.6 Case Studies

In this section we show the results of case studies that we performed 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 [22] have been strongly advocating the use of better models for network research, including background traffic models, through careful examination of unrealistic assumptions in modeling and simulation studies. Ha et al. [29] 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 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 performed to assess the global effect of background traffic generated by the TCP variants on real applications.

Single Bottleneck

The experiments described in this section again used the dumbbell topology (as shown in Fig. 4.11). Emulated traffic from real applications using a particular TCP variant was the subject of this study. Emulated traffic competes for bandwidth with simulated background traffic, which was 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, 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 [59]. 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 ($\mu$), the mean session duration ($d$), and the Hurst parameter ($H$). For self-similar traffic that exhibits long-range dependencies (LRD), $0.5 < H < 1$. 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 $H = 0.8$ and $d = 1$ second.
Figure 4.11: Single bottleneck topology.

We placed the servers on either side of the dumbbell topology shown in Fig. 4.11 to create the bidirectional background traffic. For foreground traffic, we selected three applications: web downloading, multimedia streaming, and peer-to-peer applications. For web downloading, we used httpperf 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 varied the size of the web objects to be 10 KB, 100 KB, and 1 MB. Fig. 4.12 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 measured 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 captured packets at both server and client sides. We computed jitter from 100 seconds of video streaming for 15 independent trials for each TCP variant. Fig. 14 depicts the empirical cumulative distribution of jitter. CUBIC exhibits the best performance among the three TCP variants. For peer-to-peer applications, we measured
the time for distributing a large data file. We used 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 was used to distribute a data file of 20 MB in size. We considered only the heavy traffic load condition for this experiment. The results, as shown in Fig. 4.14, clearly indicate that CUBIC outperforms Reno and Scalable TCP.

Synthetic Topology

In order to show SVEET’s capability of dealing with larger and more complex network scenarios, we conducted 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. 4.15. 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 acted as an on-off traffic source: the node stayed 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 enabled time dilation and set TDF=10 for both simulation and the virtual machines.

We placed 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 used 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. 4.16 shows the empirical cumulative distributions of the response
Figure 4.12: Response time under light (top figures) and heavy (bottom figures) traffic conditions.
Figure 4.13: Jitter from video streaming.

Figure 4.14: BitTorrent download time.
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 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.
4.3 Summary and Remaining work

In this chapter we described our TCP implementation for our discrete-event simulator. Our TCP showed nearly identical results to those produced by the network simulator (ns-2); which was previously benchmarked against real implementations.

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

We have yet to achieve the following milestones:
• We are currently in the process of finalizing our TCP implementation that will enable seamless interaction between simulated and emulated hosts. We have to test our implementation in different scenarios including: 1) Bi-directional communication, 2) Scenario where one of the hosts finalized the connection prematurely.

• Streamlining the transmission of packets between simulated entities have to be implemented. That is, if multiple servers are configured in a simulation, all of them have to refer to the same position in memory where the payload has been created, by the first server which is contacted, and thus do not create their own buffer; which is very costly.
5.1 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 [57], network protocol configuration is formulated as a black-box optimization algorithm over the parameter state space. 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. Rosenet [27] also fosters symbiotic relationships. Gu achieved a closed-loop symbiotic simulation system where the physical system, i.e., the emulation system, benefits from the LowFidelityModel provided by the simulator while the simulator benefits from the the TrafficSummaryModel provided by the physical system.

Both symbiotic systems proposed above are fundamentally different. First, online-simulation proposed in [57] forms a closed loop between the simulator and the black-box while Rosenet does so between the simulator and the physical system. Also, the goal of the former is to achieve optimization while Rosenet’s goal is to decouple simulation and the physical so that they can run distant places.

In this chapter we propose PrimoGENI, a system which aims at embedding real-time simulation into a federated system, i.e. a system which is composed of many different experiments (slivers). 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 are we going to reserve resources from multiple distant sites and connect each other?

2. Once we have the computational resources in place, how are we going to deploy experiments that enable interactions between real hosts through our simulator?

3. The only guarantee that we provide is that is if PRIMEX is keeping up with real, then it is producing accurate results. When used in PrimoGENI, we currently do not provide any insight as to whether PRIMEX is running in real time. Furthermore, at design time we “accept” every model provided. However, profiling PRIMEX and examining the model we may know in advance whether the simulator can keep up with real time. Doing this is an open question.

In this chapter we present PrimoGENI, a system which works inside a federation and uses PRIMEX as the distributed simulation engine. PrimoGENI’s objective is to incorporate real-time simulation into the GENI federation and thus enable interactions between other experiments instantiated from other projects.

5.2 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 and PRIMEX are currently being extended to fulfill all the required features of a federated component manager to be included into the GENI environment:
• **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 (slice) 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\[18\] uses the ProtoGENI\[5\] 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 will provide
experiment support services to facilitate large-scale network experiments involving simulated and emulated components; these services will include model configuration, resource specification, simulation deployment and execution, online monitoring and control, data collection, inspection, visualization and analysis.

5.3 Architecture

The PrimoGENI aggregate can be viewed as a layered system, as shown in Figure 5.1. 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 will be made known to the clearinghouse(s) and can be queried by researchers during the resource discovery process. In addition, two servers are set up to run the aggregate manager (for exporting an aggregate interface to researchers and clearinghouses) 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 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 may be created on the same physical machine to run the PRIME instance and the emulated hosts, respectively. In particular, the simulator runs on a (privileged) virtual machine, and the emulated hosts will be 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[4], 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. She will be 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 will be redirected through the emulation gateway.

5.4 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. Then, Slingshot, which has embedded the PrimoGENI functionality, performs the following steps on behalf of the user against a PrimoGENI-enabled Emulab/ProtoGENI site:

- Parses the file submitted by the user and from there it 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 the 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.

- The Slice Authority (SA) resident in PrimoGENI’s CM and Utah’s Clearing House are quieried for slice creation permission.

- Upon acceptance, a ticket that specifies the resources that a component manager (CM) allocates (or promises to allocate) to a given slice is emmited.

- After the ticket is received, it is redeemed and then the slice is initiated.

- After part of the meta resources layer is created by Emulab. PrimoGENI cooordinates with experimental (physical) nodes the processes are instatiated in the correct order, i.e., the simulator must be instantiated first, then the emulation gateway, and finaly the emulated machines connect to the simulator using the gateway.

The whole experiment workflow is shown in Fig. 5.2. In the current setup, we use OpenVZ 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
User Installs Slingshot

User composes a network model in Java or Python and compiles it using PRIMEX.

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

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.

Figure 5.2: PrimoGENI’s experiment workflow
physical nodes: simulator, emulation gateway, emulated nodes, and remote nodes (emulated nodes connected through the gateway). During ticket redemption, we map one or more than one of the aforementioned entities to a physical machine specified.

5.5 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, which is the user interface experimenters use to configure, launch and control the experiments, we would like to have 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.

The command framework, as illustrated in Fig. 5.3, uses MINA [1], 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 scalability 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, 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 one connected with slingshot; commands sent from slingshot, if they are not destined for the master, will be relayed to the corresponding slave meta-controllers. After the compute nodes have been allocated for an experiment, slingshot will choose one compute node to be the master and establish a secure connection to it using
the experimenters key obtained from the aggregate manager. After slingshot successfully connects to master meta-controller, slingshot instructs it to take control over the remaining compute nodes the master establishes an secure connection to each of those compute nodes and instructs them to act as slaves.

After that, slingshot sends commands to the master metacontroller, which properly distributes the commands to the slave meta-controllers, if needed. Each command specifies the target compute node or one of its containers on the compute node, where the command is expected to run. A command can be either a blocking command or a non-blocking command. If it is a blocking command, the meta-controller will wait until the command finishes execution and sends back the result of the command (i.e., the exit status) to slingshot. If the command is a non-blocking 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 non-blocking 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:

Figure 5.3: Meta-controllers for experiment setup
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. This step also includes creating the union file systems for the containers.

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. For remote emulated hosts, this step includes setting up the OpenVPN server(s).

4. *Run the experiment.* The partitioned network model is distributed among the 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 install and 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).

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

5.6 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.

5.6.1 Validation Studies

We validate the accuracy of the test 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, 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 [16]. 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
Figure 5.4: Similar TCP congestion window trajectories between simulation and emulation 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. 5.4 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 this time, we attach two hosts on either side of the routers. We generate two TCP flows in the same direction—one 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
Figure 5.5: Similar TCP fairness behavior between simulation and emulation

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. 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
Figure 5.6: TCP throughput limited by emulation infrastructure
third with an emulated flow across two compute nodes. Fig. 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.

5.6.2 Performance Studies

In the previous experiment 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
Figure 5.7: Aggregate TCP throughput vs. emulated TCP flow count

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. 5.7.

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 (that’s 32 VMs per compute node) as they compete for the shared buffer space.

5.7 Summary and Remaining Work

In this chapter we have introduced 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. We published our very first initial implementation in [18] and subsequent improvements in [50]. Currently, we have launched PrimoGENI’s 1.0 version which enables automatic realization of experiments supporting one PRIMEX instance.

To fulfill our objectives we have yet to achieve:

- Based on the model inputted by the user, we need to be able to deploy an experiment that encompassed multiples sites (Emulab/ProtoGENI component managers). To that end, our system must be able to: 1) Process more than one certificate provided, 2)
Perform the instantiation in every site, 3) possibly establish tunnels between each site, 4) Perform other network activities to hook these sites, e.g., routing.

- VM placement is being performed in a naive way and there is currently no load-balancing scheme to do it. This functionality has to be provided.

- Secure data exchange between PrimoGENI and physical nodes, possibly using the same certificate provided by the user.

- A mechanism to inform the user whether the system is being able to keep up with real-time has to be provided so as to validate experiment results.

- We need to devise a mechanism so that users of PrimoGENI can update the software they need to run on the virtual machines in an easy way.

- The whole approach has to be validated and test cases provided.

- Our system should entirely run on Java so that it is platform independent. Currently, our version 1.0 only supports Linux.
SYMBIOSIM: LEVERAGING SYMBIOTIC RELATIONSHIPS TOWARDS SCALABLE NETWORK EMULATION EXPERIMENTS

Running network models in a discrete-event simulation environment outputs results that, to a great extension, resemble those that would be obtained in a real environment if the protocols used in the simulator behave as real ones. Given the fact that porting real protocols to the simulator is a difficult task that requires a large amount of validation, is that emulation comes into place. Simulation-based emulation systems (referred to as emulation systems in this document) are those in which live traffic originated in real applications is inserted into the simulator. Real protocols send packets using real TCP/IP stacks and the simulator somehow intercepts these to further treat them as pure simulation events to then subject them to losses and delays according to the network model inputted by the user. We tested the accuracy of our emulation infrastructure for TCP flows in diverse scenarios in [16] [17] [50] and the results conclusively showed that emulation outputs accurate results. A big advantage of these kind of systems is that events can be readily generated to test applications and protocols under varied scenarios.

However, running experiments in this way brings some disadvantages, including: 1) The emulation and the simulation infrastructure have to be co-located and in case they are not the round-trip delay may influence greatly the accuracy of emulation, 2) The rate at which the packets can be injected into the simulator is very limited [16] [17] [50].

As an alternative to emulation systems, in this chapter we present SymbioSim, an approach where a high performance simulator closely cooperates with a physical system, i.e., they compose a symbiotic system. SymbioSim symbiotic relationships promise to increase scalability and are orthogonal to GENIs [3] federation and we believe that they complement to each other. In fact, SymbioSim makes use of PrimoGENI [18] [50], a GENI aggregate which aims to integrate real-time simulation into GENI. The symbiotic relationships lever-
aged in SymbioSim allow the simulator to benefit from the traffic information injected into it from real applications running in the physical system. Consequently, no applications have to be ported into the simulator since they run on their native environment. On the other hand, having the simulator the traffic from real applications reproduced in it, plus the traffic from simulated applications, i.e., the background traffic, it can calibrate the physical system topology in such a way that real applications experiment the network conditions specified in the original network model (topology, traffic, and applications) inputted by the researcher. Also, as we will describe later in this chapter, this symbiotic relationship allows to decrease the amount of computational resources needed to realize the physical system.

The remainder of the chapter is organized as follows. First, the motivation that drove us to construct SymbioSim is discussed. After that, a review of the most relevant literature is performed. Next, our approach is described in detail followed by a description of the current implementation we have. Some preliminary results are discussed also. Finally, we summarize our approach and discuss the remaining work.

6.1 Motivation for Symbiotic Systems

The following are some of the reasons why symbiotic simulation systems are attractive to researchers:

1. DDDAS systems, described in chapter 2, promise more accurate analysis and prediction, more precise controls, and more reliable outcomes. Applications for this kind of systems are in diverse areas such as: tsunami simulation, air-traffic simulation, and medical imaging.

2. Online simulation, described in chapter 2, offers the opportunity to increase the performance of the physical system. The simulator using the data provided by the physical system in an experiment computes a better ‘point’ within the parameter space so that
the next iteration is run using this point until the optimal one, where a utility function is maximized or minimized, is found.

3. The high performance packet-level simulator can run on a different physical location than the physical system. This brings an opportunity for researchers to run real application locally, where they have full control of their machines while running simulation on remote high-performance facilities, e.g., a grid environment. Fig 6.1 depicts this scenario.

In this scenario, the environment presented to the real applications running locally is similar to that they would experiment if connected directly to the high performance simulator.

That is possible because the SSDSS is continuously updating the network model running on the emulator (the physical system) with the results generated by inputting the measurements from the physical system plus those generated in the simulator itself and directly visible in the physical system, i.e., background traffic.

4. The simulator can be, to some extent, independent of the physical system and it is the simulator the one that will update and largely manage the physical system. This brings the potential of steering any physical system as long as data and measurements can be extracted from it and actions can be applied to it.

6.2 Background

6.2.1 Our Current Emulation Infrastructure

Our current emulation infrastructure [30] captures the packets coming from real applications and forwards the packets to PRIME [35] (our network simulator); which treats them as simulation events. From the simulator’s point view, these packets seem to have been generated directly by the corresponding end-hosts on the virtual network. This particular emulation
infrastructure, used in [16][17][18], is built upon OpenVPN, which has been customized to tunnel traffic between the virtual machines and the network simulator.

Fig.6.2 illustrates our emulation infrastructure connecting PRIME with two virtual machines. To set up an experiment, the two virtual machines first establish separate VPN connections with the designated VPN server, which we call the *emulation gateway*. The OpenVPN client on each virtual machine creates a virtual network interface (tun0), which is assigned 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’s IP space via the VPN connection. In this case, data generated by applications will be sent to tun0 down the TCP/IP stack and then given to the OpenVPN client. Upon receiving the packets, the emulation gateway forwards the packets via a dedicated connection to the simulator.

The reader thread at the simulator receives the packets from the emulation gateway and then generates simulation events representing the packets being sent from the correspond-
Figure 6.2: Our emulation infrastructure

ing end-hosts on the virtual network. PRIME simulates packet forwarding 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. 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. In this case, applications will receive these packets as if directly transmitted from a physical network.

6.2.2 Symbiotic Relationships

We identified two types of symbiotic relationships between a simulator and the physical system, which according to the ultimate objective they pursue can be classified as:

1. *Calibration*. The simulator benefits from the *data* provided by the physical system since this is using real applications and protocol stacks. At the same time, the physical system benefits from the *model* provided by the simulator which *calibrates* the physical system.
2. **Optimization.** The simulator gets *performance measurements* from the physical system after an iteration of a simulation has been completed, using a given set of configuration parameters for the real applications, protocols, and network. Then, it computes the next ‘point’ within the parameter space that maximizes a given predefined function, and sets the configuration parameters according to this point. This action is performed several times until an optimal or close to optimal point has been found for the specified function. This symbiotic relationship may also involve Calibration.

SymbioSim is based on the **calibration** symbiotic relationship. It is our objective to present the real applications and protocols, running on the physical system, an environment that preserves key network characteristics compared to those specified originally in the network model while having the simulator and the physical system running on possibly different physical locations. In the following paragraphs we summarize the disadvantages of current approaches to later describe our proposed system.

### 6.2.3 ROSENET

The closest system to ours is ROSENET [27]. From a symbiotic simulation perspective, they achieved a closed-loop symbiotic simulation system where the physical system, i.e., the emulation system, benefits from the **LowFidelityModel** provided by the simulator while the simulator benefits from the the **TrafficSummaryModel** provided by the physical system. They evaluated their system using the configuration shown in Fig. 6.3.

Gu et al. succeeded at calibrating the emulation system (a NIST Net box with two hosts attached) with the network conditions present in the simulator; which indicates that the emulation system is being updated correctly at an appropriate interval set on the simulator. They tested their system and obtained metrics from it from the point of view of the real applications, specifically they measured: 1) *end-to-end delay*, 2) *packet loss*, and 3) the *effect of the sending rate* on the accuracy of the emulation. Their results clearly indicate
that their approach creates an environment, i.e., loss and delay, where real applications experiment network conditions that resembles, to a great extent, to those that would be expected if the applications sent each and every packet to the simulator (a simulation-based emulation system).

Nevertheless, we have identified the following disadvantages in the ROSENET approach:

1. Scalability. In case an experimenter decides to have more than a pair of emulated nodes, multiple instances of the NIST Net [14] emulator have to be provided, one for each pair of hosts. Clearly, this is not a scalable approach given that each instance runs in a single computer, as tested in [27].

2. Accuracy. Using an instance of a network emulator, a link emulator in this case, creates another effect: shared bottlenecks are not modeled in the emulator. That is the case because each pair hosts shares a NIST Net box, independent of all others.

Thus, in the case a pair of emulated hosts (real hosts attached to the NIST Net box) saturate a link, i.e., create a bottleneck link, that is shared by other hosts, these other hosts do not experience this saturation. It is only after a time interval has elapsed that the high performance simulator updates each instance of the emulator so that all
incumbent hosts experience the bottleneck link. This fact leads to a high inaccuracy in achieved throughput and loss. Fig. 6.4 depicts these issues.

With the objective of tackling the problem of accuracy and replace the periodic updates, Gu et al. designed two adaptive model update algorithms [56]: 1) Prediction-error, and 2) Live-traffic-measurement. The former launches a model update whenever the simulators computes that the error between the current obtained results and the old model provided to the client exceeds a given threshold. The latter is implemented in the client and is based on monitoring the round-trip-time delay between the client and the server. Nevertheless, the prediction-error based model update suffers from another problem: high delay. This is because the live-traffic-measurement model update algorithm does not trigger new traffic updates; which would allow the simulator to quickly compute the current network conditions. Therefore, the simulator has to wait until a new traffic update comes from the simulation topology, compute the prediction error and then launch the update. If we consider that an update period could be in the range of minutes, then we can easily realize that still the accuracy problem exists.
In the worst case, the update delay would be given by the following expression:

\[ ModelUpdateTime_{MAX} = TrafficUpdatePeriod + \]

\[ TimeforComputingErrorPrediction + ClientServerPropagationTime \quad (6.1) \]

If a dramatic network condition occurs in the emulation facility which lasts much less than a traffic update period, the server may not notice it since the TrafficSummary-Model would not reflect it for having lasted so little and having been averaged with the normal conditions.

We state that two factors originate for the disadvantages detailed above:

1. **All the intelligence resides in the server.** Except for the computation of the traffic summary model, all other functions are performed in the server. As a consequence, only the server can update the model at run time. Nevertheless, both the foreground traffic, i.e. the traffic exchanged between the real applications, and the background traffic can change the global network conditions. Therefore, both sides of the system, i.e., the client and the server should be able to update each other.

2. **The conditions applied to the client are static during an update interval.** During an update interval, the real application’s traffic cannot change in any way the network conditions they experiment. For instance, let’s suppose that real applications start to send a lot of data between them using TCP. Since TCP is an adaptive algorithm, its congestion window will grow continuously until a packet loss occurs. In steady state, their transfer rate, if RENO is used, will be dictated by the following expression [38]:

\[ B(p) = \frac{1}{RTT} \sqrt{\frac{3}{2bp}} + o\left( \frac{1}{\sqrt{p}} \right) \quad (6.2) \]
For the first flow, the transfer rate will be accurate, however, for the subsequent ones that will not be the case because they still see the same round-trip-time ($RTT$ in equation 6.2) and packet loss probability ($p$ in equation 6.2) as if they had all the bandwidth available to them; which is obviously not the case because the RTT will be higher due to queuing delay and, if severe congestion occurs, packet loss probability will also change.

6.3 SymbioSim

6.3.1 SymbioSim Overview

We propose a system that is based on the Calibration symbiotic relationship aimed at overcoming the aforementioned issues and propose new techniques for: 1) Decreasing the amount of computational resources needed, and 2) Overcoming the bottleneck between the physical system and the simulator.

Within the symbiotic nomenclature, SymbioSim is a closed-loop symbiotic simulation system, specifically a SSDSS (see sec. 2.1 for a more detailed explanation). In SymbioSim, there is not a clear client-server relationship, instead both have a balanced amount of computation to perform and both can update each other in presence of determined conditions. Following our objective, we want to preserve key network properties, i.e., (shared bottlenecks, model capacity(bandwidth) and available bandwidth separately, and original RTTs) so that to offer real applications running on the physical system a close-to-exact environment to that specified in the network model. However, to that end we do not intend to reproduce the same environment we have in the simulation side because this would replicate functionality and waste precious computational resources. Instead, we downscale the topology that will be instantiated on the physical system following key insights described in some detail in sec. 6.3.3 and published in [20]. However, we do not assume any 'known bottleneck' to construct the downscaled topology from the original one. A high-level schematic view of the proposed system is shown in Fig. 6.5.
The identified modules in our system are depicted in Fig. 6.5 and summarized below:

1. **Simulator**. The simulator performs what-if simulations in packet-level detail with perhaps background traffic generated by simulated hosts. It receives the TrafficModel from the Simulator-Gateway. Also, it exports SimulationResults as the product of simulation to the Simulator - External Decision Support System. We will later clarify what this SimulationResults include.

2. **Simulator - External Decision Support System (S-EDSS)**. Gets the SimulationResults coming from the simulator as input. In response, it sends all this data in order to update the physical system. In this context, update means that this module provides the physical system a set of parameters to calibrate its less-detailed network model.

3. **Emulator-Gateway**. Receives the updates coming from the simulator and applies them to the physical system, i.e., it changes the network model running in the physical system.
4. **Physical System.** The system subject to study that benefits from the *updates* originated from the simulator. This subsystem is composed of the emulator, the real applications running on virtual machines, and also the interface that allows *Calibration*.

5. **Emulator - External Decision Support System (E-EDSS).** Gets the traffic measurements coming from real application, generates a *TrafficModel*, and sends it to the Simulator-Gateway.

6. **Simulator-Gateway.** Receives the *TrafficModel* from the E-EDSS and applies it to the simulator.

The key elements of our system are detailed in the following subsections. We start by describing how we construct our downscaled topology which will settle the basis for the rest of the system. Then, we describe the physical system in detail. After that, the *updates* that will synchronize both the physical system and the simulator are discussed. Finally, we describe the modifications that have to be performed in the gateways to achieve our objective.

6.3.2 Fostering symbiotic relationships

The cornerstone of our approach is the symbiotic relationship that we propose between a simulator and a physical system. It is a symbiotic relationship (mutualistic to be more precise) because both benefit from each other. On one hand, the simulator benefits from the traffic introduced from real applications that provide it with a *global view* of the whole simulation. On the other hand, the physical system benefits from the continuous *calibration* originated from the simulator. As a consequence of this relationship, we will be able to downscale the physical system as we detail in sec. 6.3.3.

In the above described scenario, the simulator runs the complete target network model (topology, protocols, and traffic from both real and simulated applications). The *emulated traffic*, i.e. traffic originated in real applications, is re-generated in the simulator from data
gathered from the physical system and injected into our simulator via our symbiotic infrastructure (described later in more detail). Simulated applications that re-generate the emulated traffic inside the simulator are hereon called virtual applications and there exists a corresponding virtual application per each real application in the physical system. Emulated traffic in mingled inside the simulator with the simulated traffic (traffic originated from simulated applications) and both are subjected to the conditions specified in the network model. Having this global view of the running model allows the simulator to calibrate the physical system so that real applications, running in the physical system, are exposed to same network conditions as they would if they were attached to simulation-base emulators [36][21]. The schematic view of this symbiotic system is depicted in fig. 6.5.

In fig. 6.5 the two main subsystems, the simulator and the physical system, run on (possibly distant) computer clusters. Both the physical system’s EDSS (phy-EDSS) and the simulator’s EDSS (sim-EDSS) receive data from their respective subsystems and forward them to the gateway and actuator respectively. The actuator will apply the changes, computed in the sim-EDSS and originated from the simulator, to the physical system. The gateway receives traffic abstractions (explained in subsequent sections) sent from the physical system and passed that to the simulator; which will generate traffic according to the traffic abstractions received.

The first parameter that is exported from the simulation in order to calibrate the physical system is delay. Not only do we export propagation delay, but also queuing delay. This ensures that real applications are exposed to the same RTT as their virtual applications counterparts. Equation 6.3 shows that the delay in link $i$ in the physical system ($D^P_i$) is the result of adding both the propagation delay in link $i$ in simulation ($Dp^S_i$) and the queuing delay in link $i$ in simulation ($Dq^S_i$). From hereon, we denote a variable in the simulator with an ‘$S$’ in the superscript. Likewise, a variable in the physical system is denoted with a ‘$P$’ in the superscript. Values specified on the model are denoted by a ‘$M$’. The second
superscript (if present) denotes whether this variable refers to simulated or emulated (real) hosts and the index that identifies this host (if present).

\[ D_i^P = Dp_i^S + Dq_i^S \]  \hspace{1cm} (6.3)

The bandwidth of each link of a router or a host in the simulator is either used by real applications (via the virtual applications), simulated applications or idle. The summation of all compose the capacity of a link:

\[ C_i^M = AB_i^S + T_{i,E}^S + T_{i,S}^S \]  \hspace{1cm} (6.4)

In the equation shown above, \( C_i^M \) is the capacity of link \( i \). \( AB_i^S \) is the available bandwidth in link \( i \). \( T_{i,E}^S \) and \( T_{i,S}^S \) are the throughput achieved in simulation by the real and simulated applications respectively.

Real applications running in the physical system should achieve same throughput as corresponding virtual applications. Also, if there is available bandwidth in a link, real applications should be able to achieve it. Consequently, we set/calibrate the bandwidth of each link in the physical system \( (BW_i^P) \) to the summation of the available bandwidth of that link and the throughput being achieved by real applications.

\[ BW_i^P = AB_i^S + T_{i,E}^S \]  \hspace{1cm} (6.5)

Substituing equation 6.5 into 6.4 we obtain the parameters that must be measured in the simulator to calibrate the physical system.

\[ BW_i^P = C_i^M - T_{i,S}^S \]  \hspace{1cm} (6.6)

The capacity is specified in the model so we must only measure the throughput achieved by simulated flows that traverse link \( i \) \( (T_{i,x}^S) \).
\[
BW_i^P = C_i^M - \sum_{x \in U_i} T_i^{S,S}
\] 

(6.7)

In equation 6.7, \(U_i\) is the union of all the simulated flows (UDP or TCP) that traverse link \(i\).

\[
U_i = \{f_{x,i} | f_{x,i} \text{ is flow } x \text{ that traverses link } i\}
\] 

(6.8)

From the two above equations, we measure the throughput achieved by all simulated flows at every link inside the simulator so as to calibrate links in the physical system with \(BW_i^P\).

Similar approach was followed in [45] but here we do not model the background traffic with a constant bit-rate traffic source, because we run simulated applications that can generate traffic using our TCP implementation (ported from Linux and tested in previous work [16][17]).

6.3.3 Downscaling Topologies

Network emulators such as Emulab [55], offer a testbed where protocols and applications can be exercised. Users are given root privileges on the machines they are assigned. Furthermore, users can create their own disk images and thus customize the kernel according to their needs. In consequence, potentially any real application can run in such platform. However, one major disadvantage of such testbeds is scalability: every host in the virtual topology is directly mapped to a physical host and so is every link. This severely limits the amount of instances than can be exercised in a networking experiment.

Within the symbiotic scenario described in previous sections, if we used an emulator such as Emulab to provide the underlying infrastructure to host our physical system, we would be using closely as many hosts as those used if the network topology were instantiated directly. Furthermore, the simulation and the symbiotic systems need processing capability
that would increase even more the number of physical nodes needed. In order to overcome this problem, here we propose a technique to downscale the physical network as means to lower needed computation resources.

Described in section 2, DScaLeD is a method proposed to downscale a network topology by retaining only the congested links and compensating for the ignored links adding delays. We here adopt the core of that method to downscale the topology in the physical system in order to lower the number of physical nodes to realize the full topology. Nevertheless, the disadvantage of DScaLeD is that the bottleneck has to be known in advance in order to downscale the topology, a luxury that we cannot allow us if we are going to evaluate new protocols or applications. In that sense, we propose here a method that modifies the original DScaLeD and achieve downscaling. We call our algorithm DScaLeD^R (real-time DScaLeD), an approach which consists of two algorithms: 1) The first aims at downscaling the original topology, and 2) The second allows a discrete-event simulator running on real-time to calibrate the downscaled topology (physical system).

Based on the findings described in seq. 6.3.3, we now list the assumptions upon which our topology downscaling approach (DScaLeD^R), and our system in general lay upon:

1. Only bottleneck links introduce sizeable queuing delay and packet loss, while uncongested links may in some sense be transparent to the packets that traverse them [19], [20].

2. There is only one bottleneck link on a path in a given direction [30].

3. Traffic characteristics change little within each within a specific time interval [58]. In [58], it is concluded that for packet loss, delay and throughput, one can generally rely on constancy on at least the time scale of minutes.

4. Routing does not change during the simulation.
The core of \textit{DCS克莱d} is the algorithm to downscale the topology. We now detail this algorithm; which aims to achieve a downscaled topology \textbf{without knowing the traffic in advance} as it was necessary in \cite{20}:

1. \textit{Obtain the original topology in the form of a adjacency-matrix.}

2. \textit{Get rid of simulated access links.} In our platform, emulated hosts, i.e., real hosts running on the physical system, cannot communicate with simulated hosts as in \cite{36} because not every packet is sent between real applications and the simulator (see next section for more details). Therefore, in the physical system, all access links that attach simulated hosts to the topology are not needed and can be removed from the topology. However, this will not lead to a loss of accuracy, since the simulator will continuously \textit{calibrate} the physical system.

3. \textit{Obtain routing information.} From the network model specified, we get the routing information. Then, we establish imaginary flows between all remaining hosts, i.e., the real hosts, and eliminate all links that are not traversed by any flow between real applications.

4. \textit{Identification of uncongested links.} Uncongested links are identified exploring just the network topology and these are removed from the downscaled topology \cite{39}. This step is based on the fact that links with a capacity that is larger than the sum of the capacities of all the links which are attached to it cannot be bottleneck links.

5. \textit{Abstract a path into a link.} At this point, all remaining links are potential shared bottlenecks and as such they should be kept in the downscaled topology. Note that here we do not restrict the exchange of packets between any emulated hosts and we indeed are not aware of which hosts will establish communication between themselves, that is competence of the application. However, we can still abstract subpaths into a single link applying algorithm \cite{6.1} carefully and do not eliminate a potential bottleneck.
In our approach, any link on a path can be a bottleneck. Thus, whenever we represent a subpath as a single link by applying algorithm 6.1 to this subpath, we make the remaining single link, at runtime, to 'behave differently' according to the network conditions. In other words, we change the bandwidth and delay of this link to depend on what link in the subpath it represents (the *tight link*). To this end, we can follow two directions:

(a) We record the *lowest bandwidth* among all the links that are going to be simplified as one. In this case, at runtime, we change the bandwidth and the one-way delay of the link accordingly.

(b) We record the *highest bandwidth* among all the links that are going to be simplified as one and configure this single link accordingly. In this case, at runtime, depending on the exact link that experiments a bottleneck within the subpath being abstracted, a non-responsive UDP traffic in injected in this link to emulate the capacity of the link with the bottleneck.

The algorithm we devised for abstracting subpaths into single links follows the first aforementioned option and is shown in alg. 6.1.

Fig. 6.6 depicts an example of replacing a subpath with a single link. Fig. 6.7 depicts a whole process of downscaling a topology from the original. Equation 6.7 is still valid after
Algorithm 6.1 subpath abstraction algorithm

1: for each \( s \in V(G) \) AND \( v \) is and end host do
2: \( u \leftarrow s \)
3: for each \( v \in \text{Adj}[u] \) such that Color[\( v \)] \( \neq \) BLACK do
4: \( \text{bandwidth} \leftarrow \infty \)
5: \( \text{delay} \leftarrow 0 \)
6: \( \text{subpath} \leftarrow \emptyset \)
7: \( \text{artificialDelay} \leftarrow 0 \)
8: \( \text{found} \leftarrow 0 \)
9: while \( \text{found} = 0 \) do
10: \( \text{subpath} \leftarrow \text{subpath} \cup (u, v) \)
11: if \( \text{bandwidth}((u, v)) < \text{bandwidth} \) then
12: \( \text{bandwidth} \leftarrow \text{bandwidth}((u, v)) \)
13: \( \text{delay} \leftarrow \text{delay}[(u, v)] \)
14: end if
15: \( \text{artificialDelay} \leftarrow \text{artificialDelay} + \text{Delay}[u, v] \) color(\( v \)) \( \leftarrow \) BLACK
16: if \( \text{Outdegree}[v] > 2 \) then
17: \( \text{found} = 1 \)
18: else
19: \( u \leftarrow v \)
20: \( v \leftarrow \text{Adj}[u] \text{suchthatAdj}[u]! = v \)
21: end if
22: end while
23: if |\( \text{subpath} \)\| > 1 then
24: Create anewlink
25: \( \text{Bandwidth}[\text{link}] \leftarrow \text{bandwidth} \)
26: \( \text{Delay}[\text{link}] \leftarrow \text{delay} \)
27: \( \text{artificialDelay} \leftarrow \text{artificialDelay} - \text{delay} \)
28: \( \text{Delay}[s] \leftarrow \text{artificialDelay} \)
29: Replace subpath with link
30: end if
31: end for
32: end for
Original topology

No access links that attach a simulated host

Obtain routing information and eliminate all unused links

Identification of uncongested links by inspection

Abstract subpaths as single links

Figure 6.7: Downscaling the network topology
downscaling but now we do not map each and every link in the simulator to a link in the physical system. Instead, we keep a database of all the links abstracted as a single link in the set called \textit{paths} and apply algorithm 6.2 to all of them.

\begin{algorithm}
\textbf{Algorithm 6.2} Real time calibration
\begin{algorithmic}
\For {each \textit{p} \in \textit{Paths}}
\State $MinBW \leftarrow \infty$
\For {each link \textit{i} \in \textit{p}}
\If {$AB_{i}^S + T_{i}^{S,E} < MinBW$}
\State $MinBW \leftarrow AB_{i}^P$
\EndIf
\State $BW_{i}^P \leftarrow MinBW$
\EndFor
\EndFor
\State Update all $BW_{i}^P$
\end{algorithmic}
\end{algorithm}

In algorithm 6.2 we assume that the bottleneck link is the tight link [17].

6.3.4 Limiting incoming traffic

Our proposed scheme is based in that the traffic from real, distributed applications can be routed through the simulator so that this mingles emulated and simulated traffic and can finally calibrate the physical 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 [21], the ns-2 emulator [13], and PRIME [36], insert every packet generated from real applications into the simulator. As a direct consequence, while the output generated is accurate (see [16] or [17] for comparison between simulation and real systems), the scalability is seriously compromised.

Given that the 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 throughout rate between real applications and the simulator [17].
In this document we propose not to transfer every packet between the real applications. Instead, we capture the amount of bytes that are transmitted from the applications to the transport layer, and send over these values (referred to as *appBytes* from hereon) to the simulator. Inside the simulator, we have one virtual simulated host for each real application. *Virtual hosts*, via the symbiotic infrastructure (see sec. 6.4), receive appBytes. Then, they send appBytes to the transport layer beneath them on behalf of real applications. Then, our realistic TCP implementation will transmit appBytes according to the congestion control algorithm configured and the network traffic conditions existent. In this way, we can scale better by never sending payload data between the physical system and the simulator, i.e., sending only appBytes (an scalar value) which can represent hundreds of packets (after being process by TCP or UDP).

Fig. 6.8 depicts virtual hosts, in the simulator. All virtual hosts have in their stacks a *symbiotic application*; which upon receiving the amount of bytes sent from the real ap-
 applications to TCP (appBytes), generate 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 Linux. We ported Linux TCP and tested its behavior in previous papers [16][17].

6.4 Implementation

6.4.1 Design Guidelines

The design principles upon which we build our implementations are the following:

- We tried to make the simulator as independent of the the emulator as possible. In the future, this will allow us to use any network emulator as the testbed to host the physical system.

- We extended the PrimoGENI [18] [50] system to realize our symbiotic message delivering infrastructure.

- The PRIME network simulator was extended to support symbiotic operations.

6.4.2 Subsystems

We implemented most of the functionality of the proposed symbiotic simulation system; which is composed of three main subsystems: 1) simulator, 2) physical system, 3) and symbiotic infrastructure.

The symbiotic infrastructure backbone to deliver commands between the simulator and the physical system, is composed of PrimoGENI’s monitor and slaves. The monitor receives commands from PrimoGENI’s slingshot and distributes them to the target slave. In turn, slaves execute these commands and return the status to slingshot via the monitor. We use PrimoGENI to boot the monitor and slave processes in ProtoGENI [5]/Emulab [55] platforms, to then use these processes to deliver symbiotic messages embedded inside commands. To this end, PrimoGENI requests a specific disk image to be loaded in the physical nodes.
that will host master and slaves. One physical none is needed for the master and one for each node where the simulation will run (in case of a distributed simulation). After this image is loaded, PrimoGENI’s *slingshot* starts up master and slave processes to set the symbiotic infrastructure ready.

The simulator subsystem comprises the PRIME\[35\] discrete-event high-performance simulator and the Sim-EDSS. PRIME was extended with a *symbiotic application* (SymApp) that is instantiated in each simulated virtual host inside PRIME. SymApp does not have to mimic any real application, it merely receives values (appBytes) from the the corresponding real application. In response, SymApp forwards these values to our realistic simulated TCP. It is important to highlight that real packets’ payloads are never carried inside the simulator; which saves precious resources. It is at the physical system that real applications exchange data carried in packets’ payloads (where plenty of resources are enabled). Upon receiving appBytes, our TCP generates segments of maximum segment size (MSS) that matches that of Linux. This traffic, generated in first instance, by the real applications in the physical system, is inter-mingled with simulated traffic (traffic generated utterly within the simulator)
inside PRIME. Once IP packets are on its way towards their destination inside the simulator, we monitor the queuing delay these originate in routers and hosts. To that end, we extended PRIME to monitor at each interface of a router or a host, both the queuing delay and the throughput achieved by simulated applications ($T_{i}^{S,S}$). Currently, we tag packets generated by simulated applications to measure $T_{i}^{S,S}$.

The Sim-EDSS is a java process that uses Mina [1] as the core technology for performing asynchronous input/output over TCP, i.e. sending the commands. It is started manually after the symbiotic infrastructure is up and running, connects to the slave running in the same physical machine, and starts listening for updates from the simulator. Specifically, the Sim-EDSS receives the ($T_{i}^{S,S}$) from the bottleneck links as well as the queueing delay from for all links from PRIME. Upon receiving all this information, this process, ships updates out to the PrimoGENI’s master in the form of a command.

Emulab [55] was chosen as the network emulator because it provides a suitable means to configure the links that connect physical machines where applications run: Dummynet [44]. Furthermore, PrimoGENI is designed to run on top of ProtoGENI [5]; which runs on top of Emulab. We use Emulab’s web interface to instantiate an experiment to realize the downscaled topology (see section 6.3.3 for more details) that interconnects real applications. Once the physical machines are up, we start the Phy-EDSS manually; which will in turn connect to the master. Also, the actuator is started in Emulab’s delay nodes, i.e., nodes that run Dummynet to emulate a link. Before the experiment starts, all Phy-EDSS processes listen for updates from applications specifying bytes sent to TCP. These updates, containing appBytes, are sent to the master in the form of commands and will finally reach the simulator. The control loop is completed when the simulator, via the symbiotic infrastructure, calibrates the physical system by changing the available bandwidth and the delay in the Dummynet nodes.

The current system is shown in Fig. 6.9. In Fig. 6.9 it is depicted the three types of messages that traverse or system: Primo-GENI, symbiotic, and data. PrimoGENI messages
are used for initial setup of the symbiotic infrastructure subsystems. Symbiotic messages, i.e., appBytes and updates ($T_{i,S}^{S,S}$ and queueing delays) are carried within commands between the simulation and the physical system. The monitor receives raw data and executes algorithm 6.2 to then send updates to the actuator. The aforementioned messages, are used for testbed maintenance.

In the future, the whole experiment instantiation will be fully automatized using ProtoGENI’s XML-RPC API and ProtoGENI’s Slingshot [18] [50]. In the current implementation, we get the number of bytes sent from real applications to TCP by modifying the applications and monitoring the number of bytes received by the socket and sending them to the Phy-EDSS. In the future, we expect to instrument the physical systems with the Web100 [6] TCP instrumentation. Also, we have yet to incorporate into the software that instantiates the experiments, the topology downscaling algorithm and request an experiment from ProtoGENI/Emulab accordingly. Also, we have to incorporate an efficient algorithm to compute the available bandwidth in the bottleneck link to later report this to the symbiotic infrastructure. Currently, we report this value from all interfaces and determine the tight link in each possible path.
6.5 Validation

The first round of experiments aims to validate our current symbiotic infrastructure. Fig. 6.10 depicts both the simulated and the corresponding downscaled topology that we use for this purpose. Fig. 6.10a shows the target topology that comprises four hosts: two simulated and two real. The central link has a bandwidth of 5/10 Mbps and a delay of 100ms while the side links have a bandwidth of 100Mbps and a propagation delay of 1ms. Fig. 6.10b depicts the downscaled topology after running algorithm 6.1 in the target model. In the downscaled topology only one link is necessary to link both real hosts that run iPerf between them. As explained in 6.3.3, we only retain bottleneck links in the downscaled topology. At runtime, our system will calibrate this link with the queuing delay and the available bandwidth for real applications. Fig. 6.10c shows how we instantiate the downscaled topology using Emulab. Notice that the DummyNet link emulator is placed between the real applications to emulate a network link. Our system will use DummyNet commands to configure the pipes that emulate the network link.

The first round of experiments aims to validate the basic functionality of our experiments: the delivery of symbiotic messages between the simulator and physical system. If the symbiotic messages (appBytes and updates) carry accurate data and these are timely delivered, real applications, running on the physical system, should be exposed to the same environment (delay and available bandwidth) as that if the whole original target network model was instantiated in the emulator (real implementation). Having a simulated application that behaves very similarly to iPerf and having validated our simulated TCP against Linux TCP [16][17], pure simulation experiments will generate close-to-identical results to those from a real implementation. Therefore, it is sufficient to compare the results obtained from the real applications running within our symbiotic framework (physical system) to those obtained from simulation. Furthermore, not only should the physical system environment resemble that of a pure simulation but also the simulation running within the symbiotic framework should also output same results.
In accordance with the arguments discussed in the previous paragraph, we obtain the results from: 1) pure simulation, 2) the simulator subsystem using our proposed symbiotic approach, and 3) the physical system using our symbiotic approach; and compare them all in terms of 1) congestion window trajectories, 2) sequence number history, and 3) RTT experimented by applications.

The network traffic for our first set of experiments comprises two flows, the first one starts at second 0 and the second start 20 seconds later. For pure simulation, both are generated by simulated hosts running our simulated application whose behavior resembles that of iperf. For the the symbiotic scenario, the first flow is also generated by simulated host while the second is established by iperf running in two physical machines, see Fig. 6.10c. In simulation, we obtain the congestion window, sequence numbers history, and RTT from TCP traces generated by PRIME, one for each flow. In the physical system, we use tcpdump to capture packet exchanges and from there we obtain sequence number history. Also, from the actuator, we get the delay being applied at a specific time instant to the single link present in the physical system to later compute RTT.

Fig.6.11 shows the results obtained from running the model described above with limited congestion window size equal to the default value in Linux. For the value of one-way delay (OWD) of 100ms, the default maximum congestion window size is not enough to achieve the whole available bandwidth. In consequence, the congestion window size for both flow 1 and 2 remain constant after the slow start in both pure simulation and the simulation subsystem, see Fig.6.11a). We did not monitor the congestion window in the physical system because sampling the congestion window from the Linux proc filesystem consumes valuable resources that change the expected behavior.

Notice the similarity between corresponding flows in both scenarios. For the same network model, Fig.6.11b) shows the sequence number time series. Observe how the five curves have the same slope; which indicates that both flows in both scenarios are achieving same throughput. Finally, Fig.6.11c) shows the RTT seen by applications. Although there is a
Figure 6.11: Throughput achieved under random packet loss (10 Mb/s bottleneck link bandwidth).

clear difference in the same of the curves, this is less than 5ms and is not big enough to affect the behavior of the iperf. From these results we can conclude that the environment seen from iperf is very similar to that is would experiment in a real environment (simulation was previously validated against real implementations).

Fig. ?? shows the results obtained from running the model described above with unlimited congestion window size for flow 2. Thus, we explore here if the similarity remains in case the full bandwidth is utilized in the bottleneck link. Also, we evaluate in this experiment if the real flow (flow 2) and the simulated one (flow 1) share bandwidth correctly. Once again, the OWD is set to 100ms in the central link. Fig.6.12a) clearly shows that the congestion window decrease for flow 1 after flow 2 starts. Also, observe the similarity between the congestion window trajectories of flows 1 and 2 achieved in the simulator (first two curves) and those achieved in the symbiotic environment. Nevertheless, there is a small offset in the physical sysytem as to when flow 2 starts. This is attributed to the fact that we start the real flow manually since in the current implementation we do not have a global event scheduler for the physical system. Again, the sequence number time series is almost indistinguishable between pure simulation and our symbiotic approach as shown in Fig. ??b). Finally, we have the RTT curves that will dictate the achieved throughput by TCP flows. Observe how the shapes of all four curves is very similar to each other; which is expected given that both flows traverse same bottleneck link in both simulation and the symbiotic environment. However, we observe that the delay in the simulator subsystem is large when the flow starts
Figure 6.12: Throughput achieved under random packet loss (10 Mb/s bottleneck link bandwidth).

to later stabilize and show the expected behavior. We are investigating the nature of this behavior.

From the above 4 experiments, 2 purely simulated and 2 using our symbiotic framework we can strongly conclude that our approach, in accordance to our initial objectives, exposes real applications to the same enviroment (delay and bandwidth) as that that would exist if the whole, and not the downscaled, topology has been instantiated. Product of our symbiotic infrastructure that updates the physical system constantly, iperf could achieve very similar throughput to that of pure simulation; which is clearly seen in the sequence number time series curves.

6.6 Conclusions and Future Work

In this chapter we presented SymbioSim, our approach towards achieving more scalable network emulation experiments using the concept of symbiotic simulation. We compared the output of our system and that of pure simulation in terms of congestion window trajectories, sequence number histories, and RTTs and our results show hat our prototype implementation is delivering the correct data (updates and AppBytes) in a timely manner.

Still a lot of work to do though:

1. We have to devise a frequency to intelligently sample packets in the interfaces so as not to inspect every packet. Thus, saving computation resources.
2. Currently, our system runs on real time. Consequently, if the simulator cannot keep up with all the computation to perform, the experiment is not valid. To prevent this we need to optimize some existing data structures in our system. More importantly, we plan to modify the environment within which the real applications run so that the system runs at simulation time.

3. Many of the processes to instantiate an experiment need to be automatized.
CHAPTER 7

SUMMARY

In this document we stated the goal and decomposed it into many objectives that must be accomplished. We achieved the following milestones:

- *Assess the limits of our current emulation technology.* The precise limits for our current emulation infrastructure were assessed in chapter 4 for different time dilation factors. When PRIME runs on real time the limit was found to around 160Mbps.

- *Implement a realistic TCP in our simulator.* A realistic transport protocol was ported from the ns-2. Its behavior was tested in terms of congestion window trajectories, and fairness, in both small and middle-size topologies.

Still we have a lot of work to do:

- *Design and build a system to make emulation experiments more accessible to research community.* PrimoGENI is currently under construction. Our biggest challenge is how to connect multiple PrimoGENI experiments. Furthermore, we need to connect to PrimoGENI experiments other networks instantiated from other component managers.

- *Design, implement, and evaluate a system that truly promotes cooperation between a simulator and a physical system in a mutual beneficial way.* We already have a prototype implementation of SymbioSim. Results have shown that

- *Instantiate a large-scale scenario that showcases the potential of symbiotic simulation.* A successful instantiation of a large-scale experiment will be the final milestone to be achieved.
BIBLIOGRAPHY


