3-30-2012

Towards Simulation and Emulation of Large-Scale Computer Networks

Nathanael M. Van Vorst
Florida International University, n@vvc.im

DOI: 10.25148/etd.FI12050124

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

TOWARDS SIMULATION AND EMULATION OF LARGE-SCALE COMPUTER NETWORKS

A dissertation submitted in partial fulfillment of the

requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Nathanael Van Vorst

2012

To: Dean Amir Mirmiran
College of Engineering and Computing

This dissertation, written by Nathanael Van Vorst, and entitled Towards Simulation and Emulation of Large-Scale Computer Networks, having been approved in respect to style and intellectual content, is referred to you for judgment.

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

Raju Rangaswami

Ming Zhao

Chen Liu

Jason Liu, Major Professor

Date of Defense: March 30, 2012

The dissertation of Nathanael Van Vorst is approved.

Dean Amir Mirmiran
College of Engineering and Computing

Dean Lakshmi N. Reddi
University Graduate School

Florida International University, 2012

ACKNOWLEDGMENTS

I would first like to thank my husband Louis. Without his encouragement I would not have returned to school to finish my doctoral studies. His continued love and encouragement has provided me with the fortitude to complete this dissertation. Without him, none of this would have been possible. I am also grateful for the love, support and encouragement of my family and friends.

I wish to express my gratitude towards my adviser, Professor Jason Liu. Jason's commitment to excellence has been, and will continue to be, a source of inspiration for me. Thank you for always taking time from your busy schedule for me. Our numerous brain storming sessions and paper discussions were both enjoyable and edifying. I also wish thank Professors Raju Rangaswami, Ming Zhao, and Chen Liu, the other members of my Ph.D. committee, for making time to support my dissertation.

I want to thank my constant collaborators, Ting Li and Miguel Erazo. Our many hours of discussions were enjoyable as well as good sources of fruitful research ideas. I am grateful that I was able to make this journey with both you. I want to thank Ron Oldfield and Kevin Pedretti at Sandia National Labs for the opportunity to work with both of you. My internship at SNL was stimulating, enjoyable, and something I will remember for years to come.

I also want to thank Florida International University for generously providing me with a dissertation year fellowship. The fellowship provided me the opportunity to focus on my research and complete this dissertation.

ABSTRACT OF THE DISSERTATION

TOWARDS SIMULATION AND EMULATION OF LARGE-SCALE COMPUTER NETWORKS

by

Nathanael Van Vorst

Florida International University, 2012

Miami, Florida

Professor Jason Liu, Major Professor

Developing analytical models that can accurately describe behaviors of Internet-scale networks is difficult. This is due, in part, to the heterogeneous structure, immense size and rapidly changing properties of today's networks. The lack of analytical models makes large-scale network simulation an indispensable tool for studying immense networks. However, large-scale network simulation has not been commonly used to study networks of Internet-scale. This can be attributed to three factors: 1) current large-scale network simulators are geared towards simulation research and not network research, 2) the memory required to execute an Internet-scale model is exorbitant, and 3) large-scale network models are difficult to validate. This dissertation tackles each of these problems.

First, this work presents a method for automatically enabling real-time interaction, monitoring, and control of large-scale network models. Network researchers need tools that allow them to focus on creating realistic models and conducting experiments. However, this should not increase the complexity of developing a large-scale network simulator. This work presents a systematic approach to separating the concerns of running large-scale network models on parallel computers and the user facing concerns of configuring and interacting with large-scale network models.

Second, this work deals with reducing memory consumption of network models. As network models become larger, so does the amount of memory needed to simulate them.

This work presents a comprehensive approach to exploiting structural duplications in network models to dramatically reduce the memory required to execute large-scale network experiments.

Lastly, this work addresses the issue of validating large-scale simulations by integrating real protocols and applications into the simulation. With an emulation extension, a network simulator operating in real-time can run together with real-world distributed applications and services. As such, real-time network simulation not only alleviates the burden of developing separate models for applications in simulation, but as real systems are included in the network model, it also increases the confidence level of network simulation. This work presents a scalable and flexible framework to integrate real-world applications with real-time simulation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

This chapter presents motivations for our research, details the problems this research aims to address, lists the major contributions of our research, and outlines the remainder of this dissertation.

1.1 Motivation

As the size of the Internet increases, efficiently studying the behavior of the network becomes more and more challenging. Because of the complexity of network behavior, there are no analytical models available that can accurately describe the behavior of Internet-scale networks. This makes large-scale network simulation a necessary tool for studying immense networks. However, large-scale network simulation has not been commonly used to study networks the size of the Internet. The fact is, researchers often ignore the possible differences between small-scale and large-scale simulations. They extrapolate the results from small simulations and make inferences regarding effect and behavior without performing large-scale simulations. This dilemma is not because large-scale simulation can be replaced by small-scale simulation, but is primarily due to the lack of tools and models for large-scale study [RA02, FPP^{+}03]. Previous work has shown that in certain cases, only large-scale simulation is able to gain credible evidence. Some applications only show full potential on a large-scale. For example, scalability is critical for peer-to-peer applications since they exhibit ”network effect”, which means the behavior of one user is positively affected when another user joins and enlarges the network [RFI02, KS94]. Another example is worm study, which requires a large-scale model to show the propagation dynamics of the worm [LYPN02]. Furthermore, some applications of interest often focus on large-scale behaviors more than small-scale since the large-scale behaviors can be expected to be presented under a wide range of network conditions. For example, local fluctuations are known to be highly sensitive to network conditions such as the mix of applications, user population, and network topologies in high-speed network traffic modeling. Therefore, large-scale network simulation is indispensable for studying the behavior of today's immense network.

One cannot underestimate the importance of simulation validation. There have been efforts (e.g., [Fal99, BSU00]) in augmenting network simulation with emulation capabilities, which we call real-time network simulation [Liu08]. With the emulation extension, a network simulator operating in real time can run together with real-world distributed applications and services. These applications and services generate real packets, which are injected into the simulation system. Packet delays and losses are calculated as a function of the simulated network condition. To a certain extent, real-time network simulation not only alleviates the burden of developing separate models for applications in simulation, but also increases the confidence level of network simulation as real systems are included in the network model.

Executing a large-scale real-time network simulation is not enough. Researchers should be able to visualize not only the topology of the simulated network, but observe evolutions of states occurring within the simulator. Figure 1.1 depicts a network simulator that executes virtual networks using high performance computers and remote hardware while allowing researchers to visualize, scrutinize, and modify the virtual network. Some nodes within the virtual network run full-fledged network stacks, others run real applications, yet others are composed of analytical models. There are a number of challenges that arise in building a tool that operates at the scale of the Internet. The first challenge is building a tool which can be easily used by researchers to construct and execute large-scale network models. In order for large-scale simulation to become common place, we need to provide methods to configure and interact with large-scale network models without adding to the already complex task of developing a simulator capable of

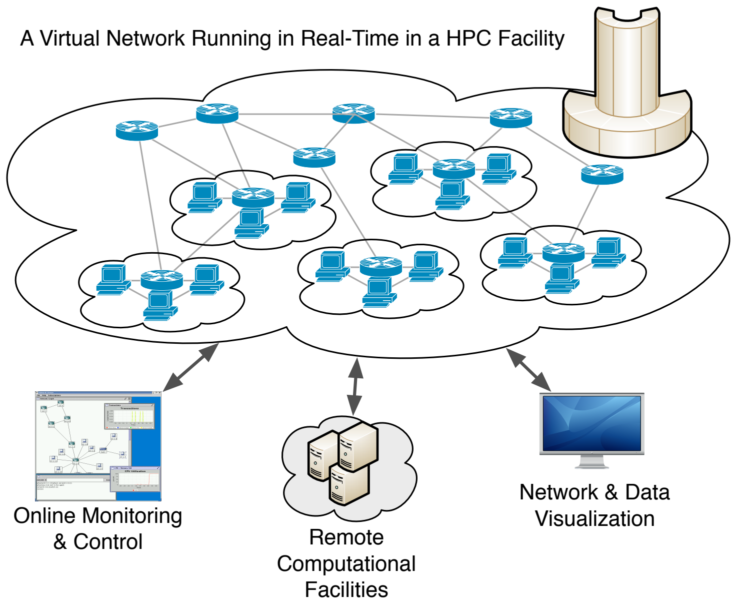

A Virtual Network Running in Real-Time in a HPC Facility
Online Monitoring & Control
Remote Computational Facilities
Network & Data Visualization

Figure 1.1: A large virtual network is simulated in real-time at a remote HPC facility along with remote computational resources. Researchers are visualizing and navigating the virtual network while monitoring and altering the state of the virtual network in real-time.

executing large-scale models. The second challenge is reducing the memory footprint of the network model. Maintaining state for hosts and routers running abstract protocols and applications while also managing the routing state for the entire virtual network is a huge burden. The integration of real protocols and applications exacerbates this burden because the real implementations consume even more memory than their abstract counterparts. The third challenge is managing potentially hundreds of thousands of routers and virtual hosts running real applications. Real applications produce and consume real network packets which must be managed by the network simulator. Efficiently transporting real and virtual packets in the virtual network quickly becomes problematic because of the amount of I/O that must be done.

In this dissertation, we consider the following three challenges in building a large-scale real-time network simulator:

- **Real-time Interaction with and Control of Large-Scale Experiments**: To date, large-scale simulators have focused on how to efficiently execute network models on parallel computing platforms and have largely ignored user facing concerns. Building network models of Internet-scale is a challenging endeavor and the lack of tools to assist in the development of such models exacerbates the difficulty. Our goal is provide a framework that can automatically provide users with easy to use and flexible tools for the configuration, control, and visualization of large-scale network models without increasing the complexity of constructing an efficient simulation engine to execute those models.

- **Reducing Memory Consumption**: As network models become larger, so does the amount of memory needed to simulate and emulate them. When constructing large-scale network models, it is common to reuse sub-structures within the model. Our goal is to exploit these structural duplications to reduce the amount of state that must be managed during model execution.

- **Supporting Large Numbers of Emulated Applications**: Packets in a purely virtual network take but a few bytes to represent kilobytes of virtual data. However, when dealing with packets that originated from real applications, the real-time simulator must preserve the full extent of the packet data. As such, the speed and efficiency of importing and exporting real-world network packets becomes paramount. Our goal is to build an emulation infrastructure capable of supporting many thousands of virtual applications in a single experiment.

Throughout this dissertation we aim to develop, implement and evaluate ideas to address these problems using our real-time network simulator.

1.2 Contributions

This research can be summarized with the following six contributions. Our first contribution is the development of a methodology which decomposes large-scale network models into an interactive model and an execution model in order to cleanly separate user facing concerns from the details of efficiently exploiting parallel platforms to execute large-scale network models. In order to implement model splitting, we created the technique of network scripting which automatically generates an execution and an interactive model from a single network model. Network scripting is a lightweight approach to resolve issues of configuration consistency and runtime synchronization that result from splitting a network model into the execution and interactive components.

Our second contribution is the conception of a model caching mechanism and a custom database engine which allows large-scale models to be created efficiently and persisted to disk. The model cache is the core component of the interactive model which allows users to modify, visualize, and interact with large-scale network models during their construction and execution.

Our third contribution is the development of a real-time monitoring and control framework. The monitoring and control framework is used to synchronize the interactive and execution models as well as the deployment and orchestration of large-scale simulation and emulation experiments.

Our fourth contribution is the development of our model replication strategy and the associated spherical routing scheme. Model replication and spherical routing can dramatically reduce the memory required to construct and execute large-scale network models.

Our fifth contribution is the development of a comprehensive, flexible, and performant emulation framework. Using our framework, researchers are able to create experiments that embed real-time applications running on virtual machines or real hosts connected via a virtual private network in addition to interacting with real networks with arbitrary topologies.

Our final contribution is the development of the PRIMEX simulator and the associated integrated development environment (IDE) called Slingshot. PRIMEX extends our existing parallel discrete event network simulator [Mod], with our interactive, control, and emulation frameworks. Slingshot and PRIMEX have been used to develop PrimoGENI [Pri10], a real-time simulation tool embedded within the GENI federation [GEN].

1.3 Outline

Chapter 2 presents relevant background material which is the starting point of this research. Section 2.1 gives an overview of existing network simulators and section 2.2 talks about current emulation techniques. Section 2.3 then discusses real-time network simulation.

Chapter 3 presents our efforts towards enabling online interaction with large-scale network simulations. Related work is presented in section 3.1. Section 3.2 contains an overview of our approach, called *model splitting*, which automatically generates interactive and execution models using a single network model in order to cleanly separate user facing concerns from concerns related to the simulator itself. Sections 3.3, 3.4, and 3.5 detail different aspects of model splitting and its implementation in our real-time simulator. The chapter ends with an evaluation of our system in section 3.6 and chapter summary in section 3.7.

Our work on reducing the memory complexity of network simulations is presented in chapter 4. Section 4.1 presents our overarching approach, *model replication*, which reduces the memory complexity of network models by exploiting structural duplications within a network topology. We then present our work in applying model replication in the minimization of the amount of memory required for routing in section 4.2. Section 4.3

details our method to further reduce the memory complexity of network models by sharing states between individual network models. We then evaluate our efforts of reducing the memory complexity of network models in section 4.4. The chapter is then concluded in section 4.5.

Chapter 5 is focused on realizing a flexible, efficient and scalable emulation infrastructure for our network simulator. Section 5.1 contains a discussion of related work. Our overall framework is then presented in section 5.2, followed by detailed discussions on the constituent components of our framework in sections 5.3, 5.4, and 5.5. A thorough evaluation of the emulation framework is then presented in section 5.6 and the chapter is concluded in section 5.7.

The dissertation is concluded in chapter 6 with a summary of our work and a discussion of future directions in which this research could be taken.

CHAPTER 2

BACKGROUND

There are three basic types of network testbeds: physical, simulation, and emulation. Physical testbeds can accurately capture detailed network transactions. However, they suffer a major drawback; their scale is inherently limited by physical resources. Emulation abstracts the networking hardware — typically using some form of virtualization — to overcome some of the scaling limitations of physical testbeds. The major distinction between simulation and emulation is that networking applications and protocols are purely virtual in simulation. Whereas in emulation, applications and protocols are real implementations. Real-time network simulation is somewhere between emulation and simulation. It aims to combine the fidelity and realism offered by network emulation with the flexibility, control, and scale of simulation.

The remainder of this section will review the current advances in network simulation, emulation, and real-time simulation on which this dissertation builds. Physical testbeds, such as PlanetLab [PACR02] and WAIL [BL03], are not covered here as they are mostly unrelated to this work. Section 2.1 covers basic simulation and discuss techniques to coordinate the execution of parallel discrete event simulators in section 2.1.1. A few parallel discrete event computer network simulator exemplars are then outlined in section 2.1.2. Section 2.2 covers the basics of computer network emulation. Sections 2.2.1, 2.2.2, and 2.2.3 discuss the major types of computer network emulators and describe salient examples of each. A discussion of real-time computer network simulation is then presented in section 2.3.

2.1 Simulation Environments

A simulation model can be viewed as an abstract representation that is used to mimic behaviors of a real-world process or system. A simulator executes simulation models by advancing time and observing how the state variables within the model evolve. We can classify simulation models by how the simulator advances time. There are two basic methods of advancing time in simulation: time-stepping and discrete-event. *Time-stepping* simulators, as the name implies, advance time in small increments over the course of the simulation. Time-stepping simulation models represent time as a continuous variable and are suitable for simulating systems of equations. For example, time-stepping simulation platforms such as Trilinos [HBH$^+$05, HBH$^+$03] have been used to model and predict global climate patterns [DNP94, ERS$^+$09] using extremely large systems of differential equations.

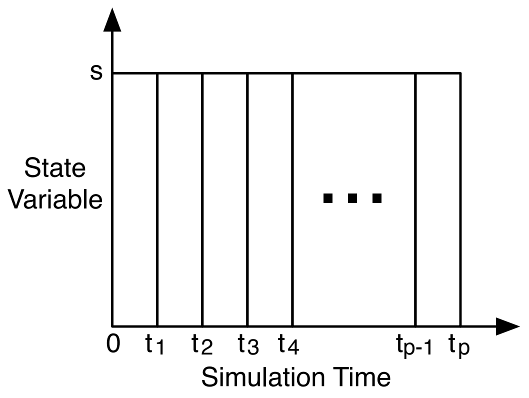
Event-driven simulators, also called *discrete event simulators*, are executed by processing a series of events and not by directly advancing time. Generally speaking, discrete event simulation models produce and consume events. As events are generated, they are stamped with the time at which they need to be processed. The simulator maintains all of the outstanding events in a data structure, typically called an *event list*, which allows the simulation to determine which event should be next consumed by the model. The current time in the system can be viewed as the minimum time-stamp in the event list.

There are three schools of thought when it comes to the development of discrete event simulators: event-oriented, process-oriented, and activity-scanning [Fuj01]. The *event-oriented approach* uses events as its basic modeling construct. Employing an event-oriented approach to model a system is akin to using assembly language to write a computer program: it is very low-level and consequently more efficient and difficult to use; especially for larger systems. The *process-oriented* approach describes a system as set of interacting processes and uses a *process* as the basic modeling construct. A process is used to encapsulate a portion of the system as a sub-model in the same way classes do in an object-oriented programming language. Typically, process-oriented simulation models are built on top of event-oriented simulators. A less common approach, *activity-scanning*, modifies the time-stepping approach by adding a predicate to all of the functions in the model. At each time step, the simulator will scan all functions within the model and execute those functions whose predicate evaluates to true.

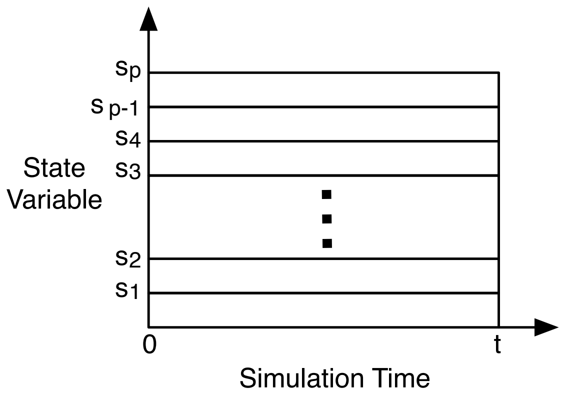
2.1.1 Parallel Discrete Event Simulation

Discrete event simulation models can be executed in parallel using a parallel discrete event simulator (PDES). Continuous simulation models can also be executed in parallel, however, the details of doing so are beyond the scope of this dissertation. Figure 2.1 depicts the two basic methods of decomposing a sequential discrete event simulation model into a parallel discrete event simulation model. To execute the model on p processors we need to divide the work into p chunks, one for each processor. We can do this division in either space (i.e. state variables) or time. If we decompose the model in time, each processor would execute the simulation for a given time period $[\mathsf{t}_{\mathsf{p}-1}, \mathsf{t}_{\mathsf{p}})$, as shown in figure 2.1(a). The key point with time-parallel decomposition is that we need to be able to predict all of the states at the time boundaries ($\{\mathsf{t}_1, \mathsf{t}_2, \dots, \mathsf{t}_{\mathsf{p}-1}, \mathsf{t}_{\mathsf{p}}\}$), and if our predictions are incorrect, we may have to recompute the state evolutions for the time interval. The advantage of this approach is that each processor can maintain its own event list without any need to synchronize with event lists on other processors.

We could also decompose the model in space. In this case, we separate the state space into p partitions and assign each partition to a processor which will simulation those states for the entire simulation. At each processor, we have a subsection of the original simulation model that consumes and produces events and a simulator to execute it. This approach alleviates the need to predict the values of state variables. However, we now have the problem of how to coordinate the execution of each subsection of the simulation model in parallel. In general, we can choose to either maintain a centralized event list, or a distributed event list. With a centralized event list, we need to schedule the execution



(a) Time-parallel decomposition.

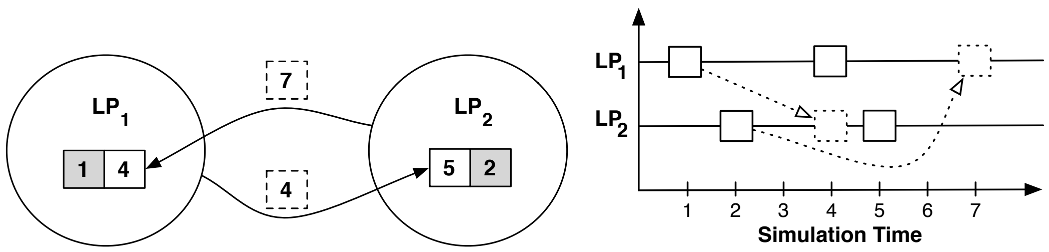


(b) Space-parallel decomposition.

Figure 2.1: (a) decomposes the model in time over p processors and (b) decomposes the model in space over p processors.

of each simulation instance in order to guarantee that any events that are produced by its associated model subsection are strictly in the future. Using a centralized event list causes many scalability concerns and the approach is not commonly used. The other approach is to distribute the event list across all of the simulation instances. In this case, we need protocols to synchronize the execution of each subsection of the model to ensure that the final result is the same as if the model were executed serially. Techniques to synchronize the execution of PDES have been extensively studied in the literature and are well covered by Fujimoto in [Fuj01]. Here, we only give a brief overview of techniques related to this dissertation.

Typically, PDES are realized using *logical processes*. A logical process is a logical grouping of processes which share a common event list and are executed using a single thread. Figure 2.2(a) depicts a simulation model employing the logical process approach. Each logical process has its own event list, executes independently, and each is able to schedule events in the other's event list. The logical processes communicate using standard facilities such Unix pipes or TCP. In this example, LP_1 is executing an event with a timestamp of 1 and LP_2 is executing an event with a timestamp of 2. As a result of LP_1



(a) Logical Processes (b) Timeline

Figure 2.2: (a) contains an process-oriented simulation and (b) contains example timelines for both processes.

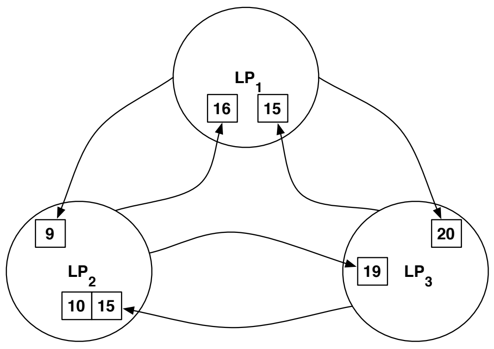
executing its event, it schedules an event in the event list of LP_2. This sequence of events is shown in figure 2.2(b). Recall that within a discrete event simulator, time advances by the simulator retrieving the next event in its event list with the earliest timestamp. After LP_2 finishes executing its current event, it will retrieve the next event to execute. If LP_2 is slow and LP_1 is fast, the new event will be scheduled in time. However, if LP_1 is slow and LP_2 is fast, the new event will be not scheduled in time and LP_2 will execute its events in the incorrect order. This is the heart of the synchronization problem for PDES frameworks.

Two classes of protocols have been developed to synchronize the execution of logical processes within parallel discrete event simulators: conservative and optimistic. Conservative protocols tackle the problem by requiring that a logical process should not execute an event until it can guarantee that no event could arrive from another logical process with an earlier timestamp. Optimistic protocols, on the other hand, assume that all the logical processes advance time more or less synchronously so they can simply execute the next event with the earliest timestamp. However, when a straggler event does arrive (i.e. an event with a timestamp in the simulated past), an optimistic logical process will have to "undo" the execution of any events with timestamps later than the straggler's. Details of both approaches are below.

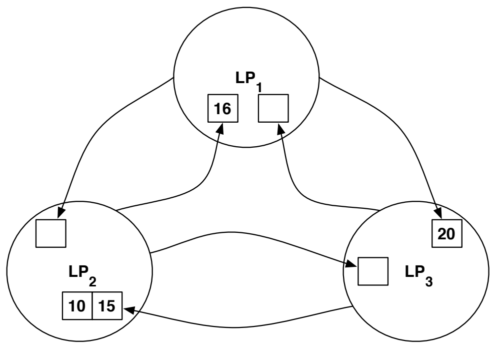
Conservative Synchronization

The classic conservative synchronization protocol was developed by Chandy, Misra, and Bryant in the late 1970s. The algorithm, referred to as CMB [CM79], places channels between all logical processes that will exchange events. When an event is sent over a channel, it is stored at the receiver end of the channel until it is processed. All of the events sent over a channel must have strictly increasing timestamps. The main logic loop of the logical process scans all of its incoming channels and processes the event with the lowest timestamp. Figure 2.3(a) shows three logical processes, each of which have events in all of their receiver queues. When a receiver queue at a channel is empty, the logical process must wait for an event to arrive before it can choose which event to process. The only way for a logical process to guarantee that it chooses the event with the lowest timestamp is to wait for an event to arrive at all of its queues. This immediately gives rise to the deadlock problem, as can be seen in figure 2.3(b). In figure 2.3(a), LP_1 will process the event with timestamp 15, LP_2 processes the event with timestamp 9, and LP_3 processes the event with timestamp 19. This produces the configuration in figure 2.3(b) where each logical process is waiting for another logical process before it can proceed, resulting in deadlock.

The CMB algorithm avoids deadlock using null messages. A *null message* is sent over a channel in leu of a real event as a pledge that no event with an earlier timestamp will be sent over that channel. After a logical process has processed an event, it may send zero or more events to other logical processes. In CMB's simplest form, a null message would be sent to any logical process that did not receive a real event. This approach guarantees that a logical process will not indefinitely wait for a message on any channel, thereby avoiding deadlock. The one drawback is that a non-trivial number of null messages must be sent. Since null messages are purely overhead, the efficiency of the simulation can be



(a) Logical Processes



(b) Deadlocked Logical Processes

Figure 2.3: (a) contains an example configuration of logical processes where the simulation can make progress and (b) shows the state after logical process consumes an event which causes the simulation to become deadlocked.

significantly decreased. There have been many extensions to the classic CMB algorithm to improve its efficiency. However, it is beyond the scope of this dissertation to discuss them. It is sufficient to understand the basic operation of the CMB algorithm which allows for synchronized execution of PDES using a conservative paradigm. A comprehensive treatment can be found in [Fuj01].

Optimistic Synchronization

The classic optimistic synchronization protocol, called TimeWarp, was proposed by Jefferson in the mid 1980s [Jef85]. In a TimeWarp paradigm, logical processes consume the event with the earliest timestamp in their event list in the hope that a *straggler event* — an event with an earlier timestamp — will not arrive at a later time. The core of TimeWarp is how to handle straggler events. When a straggler event is encountered, the logical process has to "un-process" any events that have a timestamp after the straggler's timestamp. This includes "un-processing" any events that may have been sent to other logical processes while consuming events which need to be "un-processed". To "un-process" the incorrectly executed events, the logical process needs to restore the values of every state variable to the value they held before processing any events that had a timestamp before that of the straggler. This is the problem of *state saving*. To "un-process" events sent to other logical processes, TimeWarp sends out *anti-events*, which annihilate the events they are associated with. We elaborate on state saving and anti-events below.

CMB maintains what is conceptually a single list at each logical process for its inbound events, and a single copy of each state variable. In TimeWarp, however, each logical process maintains three lists: one event list for inbound events, one event list for outbound events, and a list which stores the values of any state variables that were modified while processing an event. Additionally, once an event or state modification is added to one of the lists, they are never removed by the logical process — they might be needed for rollback if a straggler is encountered. For the moment, assume enough memory is available to maintain the three lists for each logical processing for the entire simulation. When a straggler is encountered, the logical process knows exactly which state variables were incorrectly modified, and which events were incorrectly sent to other logical processes. To rollback, the logical process needs only to copy the correct values to the effected state variables and send an anti-event to annihilate any event that was incorrectly sent.

When a logical process receives an anti-event there are three possibilities:

1. An event and it's associated anti-event could both end up in the inbound event list waiting to be processed. In this case, the logical process can just remove both events from the inbound queue.

2. The anti-event could conceivably arrive at the inbound event list before its associated event. In this case, the logical process can simply leave the anti-event in the inbound event list and wait for the event to show up, at which time both the event and anti-event can be removed from the inbound list.

3. The event associated with the anti-event may have already been processed by the logical process. In this case, the logical process needs to rollback the processing of the event.

It is not strictly necessary to store processed events, sent events, or state variable updates indefinitely. From a global perspective, there is an event which is unprocessed, partially processed, or in-flight, whose timestamp is the earliest in the simulation. The timestamp of that event is called the *global virtual time* (GVT). It is easy to show that the GVT never decreases, which means that no logical process can rollback to a time previous to the GVT. If a logical process knew the GVT, it could release any events from the inbound or outbound lists and any state variable updates with timestamps earlier than the GVT. This is commonly known as *fossil collection*. Efficient mechanisms to compute the GVT are well studied, but beyond the scope of this dissertation.

Figure 2.4 shows how TimeWarp handles the situation presented in figure 2.2(b). When LP_2 receives the event sent from LP_1 with a timestamp of 4, LP_2 has already processed an event with a timestamp of 5 and sent an event with a timestamp of 7 to LP_1. LP_2 needs to rollback its state variables and cancel the event sent to LP_1. As is clear from this example, GVT never decreases, but the progression of GVT may stall for long periods while rollbacks propagate through the network of logical processes.

There is a large and rich body of work improving and augmenting TimeWarp and other algorithms in a similar vein. For the purposes of this dissertation it is sufficient to understand the basic operation of the TimeWarp algorithm which allows for synchronized execution of PDES using an optimistic paradigm. A comprehensive treatment can be found in [Fuj01].

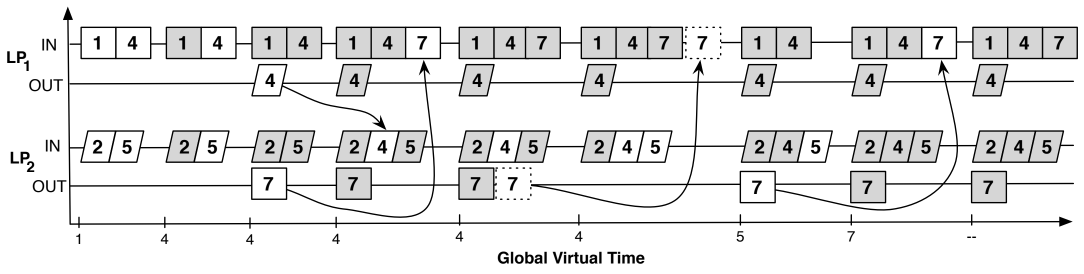


Figure 2.4: Example of how TimeWarp cancels events in response to encountering a straggler event.

2.1.2 Network Simulators

There have been numerous simulators built to model computer networks. Here we focus on a few simulators that are of general purpose, and have either gained widespread adoption, or have some defining characteristic related to this dissertation. We do not mention simulators which are largely purpose built; those that are meant to study a specific problem and not for use by the research community at large.

NS

The NS-2 [NS-a] simulator is currently the most popular simulator among network researchers. Its wide acceptance is largely due to its diverse set of different protocols and services at all protocol layers coupled with the ability to handle both wireless and wired networks. The primary technical contribution of NS-2 is its *split programming model* which allows researchers to construct their models using both a compiled language and a scripting language. Conceptually, developing models in a compiled language such as C is time consuming and difficult as compared to using a scripting language such a Python or OTCL. In NS-2, one is able use C++ to develop the core of the simulator and performance critical aspects of a specific network model. One can also use OTCL to craft the network topology and configuration, in addition to less performance critical aspects of the network model itself.

At the core of NS-2 is a sequential discrete event simulator which was not designed to handle large network models. NS-2's "tips for running large simulations" [NS-b] refers to large network models as those on the order of a thousand nodes – orders of magnitude less that what we consider to be a large. However, NS-2's ease of use and rich collection of protocols makes it invaluable for evaluating small to moderate-scale network models.

The NS-3 [NS-c] effort was started as the next evolution of NS-2. In theory, NS-3 addresses the shortcomings of NS-2; the most major being the lack of scalability and real-time simulation support. Balancing the tradeoffs between simplicity, usability, and scalability have proven to be difficult for NS-3. As such, NS-3 has, to date, failed to gain the widespread use and acceptance that its predecessor enjoys.

TeD

The Telecommunication Description Language (TeD) [POF98] is one of the earliest PDES targeted at computer networks. TeD is primarily modeled on the VSIC Hardware Description Language (VHDL). Like VHDL, it has a high-level language in which "components" (hosts, routers, network interfaces, queues, etc.) are configured and connected using event channels. In a low level language (C), developers can customize the behaviors of components. In TeD, developers can recursively compose components using other components. Models written in TeD are automatically transformed to be executed on the Georgia Tech TimeWarp simulator [DFP$^+$94].

OMNeT++

OMNet++ [VH08, And] is a popular multi-purpose simulation environment capable of simulating communication networks, multiprocessors, and many other distributed systems. OMNet++ is designed to be extremely modular and handle large-scale models. The simulator's flexibility is derived from its modularity, a well designed API, and an extensible integrated development environment. Users are able to create reusable *compound modules* which encapsulate network components (such as an applications, interfaces, queues, etc). In a similar vein as TeD, OMNeT++ allows researchers to wire many compound modules together using a separate scripting language, called NED. OMNeT++ has been extended to allow the embedding of real applications into experiments using real-time network simulation techniques. Model developers can also create experiments which can be executed on parallel machines.

OPNET

OPNET modeler is the flagship product of OPNET Technologies Inc [OPN]. OPNET is primarily a commercial product which can be freely used by qualifying groups. The selection of ready-to-use protocols and applications within OPNET rivals that of NS-2. In many ways, OPNET is much like OMNeT++. Just like OMNeT++, users can create "modules" in OPNET that can be recursively wired together to create large network experiments. The main differences between OPNET and OMNeT++ are that OPNET is written in C while OMNeT++ is written in C++, and because OPNET is commercial, much of the core simulation code is proprietary and not publicly available.

SSFNet

SSFNet [NLLY03] is a PDES built using the Scalable Simulation Framework (SSF) [Jam]. SSF has both Java and C++ implementations. The Java version of SSF is embedded within SSFNet itself while the C++ implementation is available as stand alone package called DaSSF [LN]. DaSSF uses a logical process world view and is specifically designed to execute large-scale simulation models in both shared and distributed memory parallel computers. SSFNet extends SSF to create a PDES specifically designed for modeling computer networks.

The key driver of SSFNet is the scale of the network models it aims to execute. Large network models will have substantial computational demands, hence DaSSF's support for parallel execution on both shared and distributed memory parallel machines. DaSSF was shown to execute over one million events per second using just fourteen processors [CNO99].

SSFNet also addresses the complexity of configuring a network topology and traffic patterns for large network models [CLL+99] using the Domain Modeling Language (DML). SSFNet uses DML to separate a network model into a network topology, a traffic pattern, a network configuration, and model logic. The logic of the protocols and applications are written in Java or C++ and compiled into simulator. The topology and traffic patterns along with their configuration are specified using DML, and loaded by the simulator when the network model is executed. This separation is critical to supporting very large network models. For example, it allows complex tasks such as partitioning the model to run on a parallel computers to be outside the simulation environment. This diverges from other contemporary PDES frameworks such as GTNets [Ril03] or PDNS [FPP+03].

PDNS

Parallel/Distributed NS (PDNS) [FPP$^+$03] extends the venerable NS-2 simulator with PDES functionality. To do this, PDNS creates many instances of the NS-2 simulator and treats each one as a logical process. The idea is compelling. As previously stated, NS-2 is widely adopted and supports a rich collection of protocols. However, specifying large network topologies and actually executing them on a large parallel machine proves to be an incredibly arduous task. NS-2's OCTL configuration language was not designed to support PDES models and assumes that each NS-2 instance has a complete view of the entire network; however, in PDNS each NS-2 instance has a partial view of the network. PDNS modified the configuration language to support this partial view. However, the modifications result in the user having to manually partition the network topology for execution on parallel computers. Partitioning a large model is a difficult task [Nic98], and doing this by hand only increases the complexity. However, with enough effort and time devoted to constructing the model, PDNS has been shown to process over 106 million packets per second using 1,536 processors [FPP$^+$03].

GTNetS

The GTNetS [Ril03] simulator is written in C++, and was designed to allow researchers to easily create large-scale experiments. The design of GTNetS closely matches the design of real network protocol stacks and networking hardware. The end result is that researchers are able to easily understand how to extend GTNetS and use it to create experiments. In order to support parallel execution of network models, GTNetS uses ghost nodes [RJFA04]. In the ghost node approach, each simulation instance contains the entire network topology. Each simulation instance maintains a complete representation of the nodes it is executing, and a minimalistic representation for nodes that are executed by other simulation instances (i.e. ghost nodes). GTNetS has been shown to execute over 5 million packets per second using 128 processors [Ril03].

ROSSNet

ROSSNet [BYC+03] extends Rensselaer's Optimistic Simulation System (ROSS) [CBP00]. ROSS is an extremely modular simulation kernel that is capable of achieving event rates as high as 1,250,000 events per second [CBP00]. ROSSNet, much like SSFNet, uses an external configuration language to describe the topology, traffic, and configuration of a network model. Instead of DML, ROSSNet uses XML with a custom data schema. ROSSNet has two unique characteristics. It uses an optimistic simulator at its core, and it adds an additional abstraction to the definition of a network experiment called an *experiment design* [Jai91]. The basic premise behind experiment design is that running an experiment has a non-negligible cost so we need to minimize the number of experiments that are preformed while maximizing how well the system being studied is characterized. ROSSNet uses concepts from the experiment design in their *design of experiments* tool which allows researchers to efficiently explore the parameter space of their model.

2.2 Computer Network Emulation

Network emulation testbeds focus on allowing real applications to interact with an *emulated network*. There are three basic types of network emulators: link-centric, node-centric, and network-centric. *Link-centric* emulators focus on providing a "virtual link". The link calculates, in software, packet delays and drops in order to emulate network behavior. *Node-centric* network emulators focus on providing a virtual environment in which real applications run. As network packets leave virtual environments, they are intercepted by the emulation system and are subjected to delays and losses. *Network-*

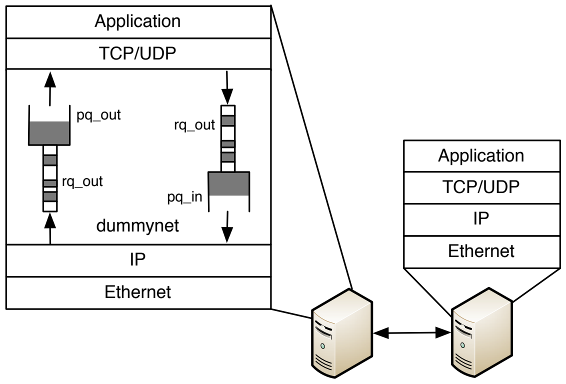


Figure 2.5: Dummynet Architecture.

centric emulators focus on virtualizing the network and can be seen as a blend of the link-centric emulators, node-centric emulators, and physical testbeds. Both node-centric and link-centric emulators focus on reproducing behaviors of a single link or connection between applications, whereas network-centric emulators aim to reproduce network-scale behaviors. In the remainder of this section, we describe the different classes of network emulators in more detail.

2.2.1 Link-Centric Emulation

Link-centric emulators create a virtual link which can shape the traffic which passes through it. The virtual link is represented using finite queues that impose bandwidth constraints and delays. Dummynet [Riz97] is a popular and prototypical link-centric emulator. The basic functioning of dummynet is depicted in figure 2.5. The virtual link is modeled as two unidirectional links, one for inbound packets and one for outbound packets. Each unidirectional link is then modeled with two queues (rq and pq), a maximum bandwidth (B), a maximum size of rq (k), a propagation delay (pd), and a queuing policy. The operation of dummynet is as follows:

1. As packets arrive from the upper or lower protocol sessions, they are inserted into the appropriate rq using the chosen queuing policy. The typical queuing policy is a

drop-tail FIFO queue. However, more complex policies such as RED [FJ93] can be used. Dummynet can also randomly reorder packets at this stage.

2. Packets in rq are moved to pq at a maximum rate of B. pq uses a FIFO policy and will never drop packets.

3. Packets are stored in the pq for pd seconds at which time they are dequeued and sent to the next protocol layer. Dummynet can introduce random losses at this stage to model lossy mediums such as a wireless channel.

Users are able to emulate a wide variety of link/network conditions by changing the parameters within dummynet. The main drawback here is that one can only test a single application per dummynet instance. Because dummynet modifies the operating system kernel, you would need a separate protocol stack for each application to be tested. There are obvious scalability issues with the link-centric approach.

2.2.2 Node-Centric Emulation

Node-Centric emulation focus on virtualizing the operating system kernel. The goal is to create a protocol development and test environment that can be run in user-space and not the operating system kernel. The creation of a realistic protocol stack in user space becomes the next hurdle. NCTUns [WCH$^+$03] employs TUN/TAP devices to intercept packets from the operating system and then passes them to a custom protocol stack in user space which functions like dummynet. ENTRAPID [HSK99] and NIST Net [CS03] take a different approach and provide "virtualized" operating systems with their associated protocol stacks in user space. The benefit here is that you can develop protocols using the actual operating system kernel code instead of a custom one. In general, node-centric emulators are a more scalable alternative to link-centric emulators.

2.2.3 Network-Centric Emulation

Testbeds such as Emulab [WLS$^+$02, GRL05] and DETER [BBK$^+$06] opt for a different approach to address the scalability issues of link-centric emulators. They assume there is a large testbed with many compute nodes connected with a programable high-speed switch. Some of the nodes in the testbed will run unmodified operating systems and host test applications (*application nodes*). Other compute nodes will use dummynet to shape the traffic (*delay nodes*). They then use VLANs to connect the application and delay nodes to create a small test network. It is assumed that institutions would have to share the testbed to reduce costs. The main research problem in Emulab-like environments is the mapping of multiple current experiments onto the testbed in order to maximize resource utilization.

ModelNet [VYW$^+$02] takes a slightly different approach. Modelnet, like Emulab, assumes there is a large testbed with nodes connected with a programable switch. Some of the compute nodes are used as *edge nodes* which run test applications and the remaining compute nodes are used to emulate the core of the network. To model the network's core, Modelnet extends dummynet so that many network paths can be emulated in parallel. The operating systems on the edge nodes are modified to capture traffic from the test applications and move it through the emulated network core. In a similar fashion, the edge nodes send traffic from the emulated network core back to the applications. The main research challenge here is how to extend dummynet to emulate the network core.

2.3 Real-Time Computer Network Simulation

Real-time network simulation aims to combine emulation and large-scale network simulation to create an accurate, scalable, and flexible networking testbed. One of the major drawbacks of simulation is the difficulty of validation. Developing realistic and scalable simulation models can be difficult and error prone. Allowing existing protocols and applications to be integrated into simulation helps to validate the realism of the simulation because real implementations — not abstract models— are used. A major drawback of emulation is the lack of background traffic. Background traffic has been shown to significantly effect the end-to-end behavior of the (foreground) applications under test [SDRL09].

One of the major drawbacks of both physical and emulation testbeds is a lack of true scalability. Even with extreme amounts of virtualization, the scale of emulation and physical testbeds are ultimately limited by physical resources. Simulation provides a trade-off between scalability and realism (or accuracy). Protocols, applications, and packets are abstract objects and routing a packet across dozens of routers can be done with just a few computations. In an emulation testbed however, real applications must exchange real packets, all of which take resources.

The benefits of combining simulation and emulation testbeds are clear, and a number of real-time network simulators have been developed. NSE [Fal99], IP-TNE [BSU00], MaSSF [LXC03], RINSE [LLN$^+$05], and ROSENET [GF09] are good examples. In the remainder of this section, we will describe these real-time network simulators in more detail.

NSE

NSE [Fal99] is an extension to the NS-2 simulator. NSE modified the event scheduler in NS-2 so that it can operate in real-time. NSE uses standard TUN/TAP devices to intercept packets and inject them into the NS-2 simulator. NSE does nothing to address the scalability of NS-2. As a result, NSE can only operate with small networks.

IP-TNE

IP-TNE [BSU00] extends IP-TN, a PDES, with the ability to process real packets within the simulation. IP-TNE allows real hosts to route packets through a virtual network. Instead of using dummynet to create a delay node, IP-TNE allows the delay node to be a complex network. This reduces the burden on researchers because they no longer have to abstract the network as a link and estimate parameters for dummynet. Instead, they can directly describe the network model they wish to evaluate.

In addition to acting as a complex delay node, IP-TNE allows real hosts to interact with simulated hosts using ICMP and UDP. IP-TNE lacks full-blown TCP implementations which would be required for simulated hosts to interact with real hosts.

DaSSF based Real-time Network Simulators

Both MaSSF [LXC03] and RINSE [LLN+05], extend the DaSSF [LN] and
SSFNet [NLLY03] simulators. MaSSF adds extensions to emulate grid computing environments. MaSSF integrates the authors pre-existing MicroGrid emulator [SLJ+00]. MicroGrid uses virtual machines and a virtualized grid software to create an emulated grid environment in which to execute test applications. MaSSF modifies MicroGrid by adding a custom *wrapped socket interface*. The wrapped socket interface is used by applications within the virtual machines to communicate with each other [LXC04]. Traffic that is transported over the wrapped sockets is injected into the MaSSF real-time network simulator. MaSSF provides no functionality to allow applications to interact with simulated routers or hosts. Its primary use is to evaluate how a computing grid's design impacts the performance of grid applications.

The Real-time Immersive Network Simulation Environment (RINSE) extends DaSSF with the ability to exchange traffic with real applications. RINSE expects an instance of the simulator to be run next to each application and uses packet filters [MJ92] to intercept traffic generated by the application. Traffic is injected back into the operating system using a raw socket. The operating system will then forward the traffic to the application as if it originated from a network interface. RINSE has three notable contributions: a detailed host model, a real-time scheduler, and multi-resolution traffic modeling. The detailed host model is used to model application and user behaviors on simulated hosts. The real-time scheduler ensures that simulated and real packets are correctly processed in the real-time network simulation. In order to reduce the computational demand of simulating large numbers of TCP sessions, RINSE integrates a fluid model of TCP into the simulator. Fluid models of TCP have been shown to operate orders of magnitude faster than their detailed counterparts.

ROSENET

Recently, ROSENET [GF09] has integrated symbiotic emulation into a real-time simulator. In ROSENET, a large-scale network simulator is run disjoint from an emulated network. The emulated network runs real applications and protocols, and is a scaled down version of some portion of the network which is being simulated. As both the simulation and emulation progress, they are synchronized. Measurements from the emulation are used to tune the simulation, and in a similar fashion, results from the simulators are used to adjust the emulation. The ROSENET approach is interesting in that the emulation system is used more like an abstract model. This diverges from most other real-time network simulators. Ordinarily, emulation is used to increase the credibility of a real-time simulation since traffic from real protocols and applications are directly mixed with their virtual counterparts.

CHAPTER 3

INTERACTIVE NETWORK SIMULATION VIA MODEL SPLITTING

Simulation is useful for studying large complex networks, in particular, the emergent behaviors and large-scale phenomena, which are difficult to scale down due to the interaction and inter-dependencies between components at different layers of the system that cannot be easily abstracted away. And yet, large-scale network simulation is not used commonly today. This can be attributed to the lack of easy-to-use large-scale network simulators and realistic network models that can withstand the test of time.

There is an important distinction between developing a large-scale network simulator and a large-scale simulation experiment. The former copes with the computational issue of running parallel simulation on high-end computing platforms, while the latter deals with the challenges of creating topologies and traffic conditions resembling the target network. In between, different players assume different roles.

A *parallel simulation engine* provides a method for decomposing large-scale models into an interconnected graph of logical processes – each capable of processing simulation events concurrently on today's parallel and distributed platforms. Large-scale network simulation produces a huge number of simulation events, thus providing ample opportunities for parallelism. A *network simulation framework* extends the parallel simulation engine with domain-specific models, including common network components (such as subnetworks, routers, hosts, links, and network interfaces), and basic programming constructs (such as sending and receiving messages and scheduling timers). Based on these network components and constructs, *network models* can be developed for specific network protocols, services, and applications. Most existing network simulators offer a good selection of common protocols at different layers of the network stack, such as Ethernet, TCP/IP, BGP, OSPF, DNS, etc. *Network experiments* consist of a detailed specification of network topology and traffic for the specific network scenarios for testing applications.

Depending on the goal of the simulation, a network experiment typically needs to reproduce the network conditions observed on the real network at some level.

To date, several large-scale network simulators have demonstrated good performance and scalability with various degrees of success. They are confirmed cases of good parallel simulation practice. For example, TeD [POF98] adopts a time warp simulation engine for running large-scale network models. PDNS [RFA99] uses a federated approach to parallelize a popular sequential network simulator. SSFNet [CLL$^+$99] features a programming interface designed for large complex models using conservative synchronization protocols. GTNetS [Ril03] is also a parallel network simulator with a good set of network protocols. ROSSNet [YBB$^+$03] uses the technique of reverse computation for compact and light-weight optimistic parallel simulation of large network models.

All the above large-scale network simulators focus primarily on the performance aspect of the simulators, and all of them failed, to some degree, to capture the attention of the network research community at large, which nevertheless is in great need of veritable large-scale network simulation tools. The failure can be partially attributed to the difficulties in using a large-scale network simulator to conduct experiments. We believe this can be attributed to a poor separation of concerns. A network researcher is more interested in the development and use of realistic, verifiable large-scale network models and network experiments, while a simulator developer is more concerned with methods for developing the parallel simulation engine and the network simulation framework, which can effectively project large-scale network models onto parallel platforms to achieve good parallel performance. This separation of concerns inevitably brings complexities both in the description of the network models and the execution of the models on parallel platforms.

Our work begins with the notion that a large-scale network simulator shall be both easy to use and able to efficiently execute on parallel platforms. In particular, the simulator must provide a user-friendly interface allowing users to easily focus on developing complex network models and conducting large-scale network experiments, with little concern about model execution issues. Such user-friendly interfaces need to provide a complete end-to-end workflow for network experimentation, from the initial development and maintenance of network models, through the configuration and execution of the network experiments, to the visualization and analyses of the experiment results.

We introduce the method of *model splitting*, which naturally divides the system into an interactive model and an execution model. The *interactive model* is an iconic, succinct representation of the target network configurations; it is intended to be small so it can fit in a client machine and possibly work in conjunction with a well-designed user interface. The interactive model is used to address the user facing aspects of the system: it allows the user to easily specify the network models, visualize the network configurations, scrutinize the model parameters, inspect or modify the simulation state, and allow for online monitoring and steering of the network experiments. In contrast, the *execution model* is a full-fledged representation of the large-scale network models for the target high-end computing platforms. The execution model needs to include partitioning and mapping information for the network models to be deployed and run on the parallel platforms. The execution model also needs to capture the detailed evolution of the simulation state. In addition, certain simulation state needs to be tracked or recorded during the experiment for online or postmortem data collection and analysis.

To maintain consistency between the interactive model and the execution model, we introduce a process called *network scripting*, which allows the system to automatically create the interactive model and the execution model from a single specification of the network configuration. To facilitate real-time high-capacity interaction between the interactive model and the execution model, we introduce a streamlined *monitoring and control mechanism*, that can modify the runtime state of the simulation and filter state updates between the execution model and the interactive model in support of interactive experimentation and real-time visualization. To reduce the model size, we introduce the technique of *model replication*, which allows the sharing of data for representing the network topologies with identical structures, the state of network entities and the behavior of network protocols with identical functions. To ensure efficient in-memory processing of the interactive model, we present a *model caching scheme* on top of a database management system supporting data persistence of large-scale network models that reside out of core.

We implemented the model splitting approach in PRIMEX, which is a large-scale network simulator with a real-time extension that allows parallel and distributed simulation of large-scale networks in real time so that the simulator can interact with real network applications and real network traffic. We created an integrated development environment that interfaces with the interactive model to allow the users to construct and inspect network models via a graphical user interface, through an interactive python console, or using scripts written in XML, Java, or Python. The interactive model is persisted to a database so that the users can later reuse the model – or simply a part of it – to construct other models. The execution model, equipped with the information of the network configuration and the target execution environment, is run at the parallel machines. During the experiment, the user can use the graphical user interface to visualize the state of the network and interactively control the network experiment by reconfiguring the network model.

The remainder of this chapter is organized as follows. In section 3.1 we outline related work and in section 3.2 we describe the model splitting technique and identify major issues which arise from the method. Then in sections 3.3 to 3.5 we present our solutions to address issues related to model splitting in detail. We then wrap up the chapter with an evaluation in section 3.6 and a conclusion in section 3.7.

3.1 Related Work

Existing network simulators have partially addressed the complexities of configuring network models. A common approach is to use a description language to describe the configuration of the network, e.g., the Domain Modeling Language (DML) used by SSFNet [CLL$^+$99], and the NEtwork Description language (NED) used by OMNeT++ and INET [And]. Using a separate language can simplify the specification of network experiments, including the structure of the networks and the parameters for individual components. However, in doing so, one must manually ensure the consistency between the description language and the model implementation.

The Telecommunication Description Language (TeD) [POF98], as one of the earliest efforts of parallel network simulation, takes a different approach. Similar to VHDL, TeD provides a modular description of the hierarchical structure of the network model together with the definition of detailed behaviors of the components. The problem with this approach is that the user is forced to deal with the complexities associated with the detailed model implementation in the same model description language [PNL96].

Yet another method is to adopt a split programming approach, where the simulation code is developed in a compiled language and the logic of building and configuring the network models is written in a scripting language (such as Tcl). This is the approach employed by NS-2 [BEF$^+$00] and GTNeTS [Ril03]. The split programming approach certainly simplifies the specification of the network and provides automatic consistency between the scripting language and the model implementation. However, the two need to be on the same machine—the simulator cannot handle situations where the specification, configuration, and visualization is done interactively on a machine separate from where the model is run.

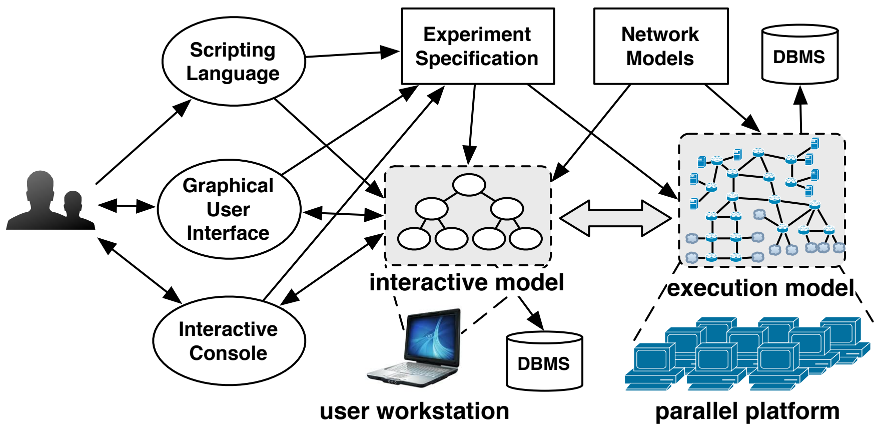


Figure 3.1: A schematic view of model splitting.

3.2 Model Splitting

In this section we present the overall architecture of model splitting and describe the associated problems and solutions.

3.2.1 Architecture

The basic premise behind model splitting is that a user's interaction with a model should be explicitly separate from how a model is executed. Figure 3.1 shows a schematic view of a simulation system that employs the model splitting technique. With the two separate models—the interactive model that runs on the user workstation and the execution model that runs on the parallel platform—we are able to address the separation of concerns. The interactive model should focus on model configuration, experiment specification, interaction and visualization, by supporting user facing tools such as the scripting languages, the graphical user interface, and interactive consoles. In comparison, the execution model should focus on execution issues, such as parallelization, synchronization, communication, and data collection on parallel platforms.

Both models are derived from the network models implemented for various network protocols and applications, and the experiment specification with detailed network topology and traffic description. However, they intend to address different concerns. More specifically, the interactive model has four important design considerations:

1. The interactive model shall be constructed and configured easily, through a graphical user interface or via a scripting language, or both.
2. The interactive model needs to be persistent across different experiments, for repeatability, for debugging, and for model reuse.
3. The interactive model needs to maintain a small memory footprint as it is expected to interact with the user on a user workstation (the front end) with fairly limited memory.
4. The interactive model can be geographically separated from the execution model: the user configures and visualizes network experiments at the front end on a user workstation remotely connected to the back end, which is the parallel computing cluster running the simulation.

In contrast, there are five key design considerations for the execution model.

1. The execution model shall be able to capture the network operations at sufficient levels of detail that satisfy the fidelity requirements of the simulation study.
2. The execution model must be able to run efficiently at the back end on the parallel platform. The model needs to be partitioned and mapped onto the parallel machines to minimize the synchronization overhead.
3. The execution model needs to conserve memory; this is especially important when dealing with extremely large-scale network models.

4. The execution model needs to interoperate with the interactive model running at the front end (at the user workstation), for network monitoring and visualization, user interaction and dynamic reconfiguration.
5. The execution model may generate a huge volume of simulation results, which needs to be collected efficiently for postmortem analysis.

3.2.2 Problems and Solutions

Splitting the network model and experiment specification into the interactive and execution models can provide the desired separation of concerns; however, it also brings several potential problems. The first problem is that we need to maintain consistency between the two models. On the one hand, the interactive model and the execution model should be based on the same implementation of the network models. That is, they should contain the same definition of network entities and protocols with the same state variables (with the same data types and ranges). Inconsistencies often arise at the development stage when the network models are constantly being changed. On the other hand, the two models should be derived from the same experiment specification. They should follow the same network configuration, with the same number of instances of network entities and protocols, and the same configuration parameters. Inconsistencies may happen when the user interactively creates and configures the network experiment.

To solve this problem, we introduce the method of *network scripting*. Network scripting is a technique of automatically generating the interactive and execution models using annotations embedded in the implementation of the network models. These annotations provide the meta-information of the state variables and the hierarchical structure of the network model. For example, the annotations can describe whether the state variables are configurable by the user, and if so, what are their data types, what are the ranges of values they can take, and what are the default values. The annotations can also describe, for example, the minimum and maximum number of sessions that can be instantiated for a protocol on a host or router. The meta-information is used to generate the schema for the scripting language, the interactive console, and the graphical user interface, which are then used to create and interact with network experiments. The interactive model is also embedded with a database management system to deal with large-scale models that must be handled out of core. The meta-information is used by the execution model to prepare for the full-fledged network model and enable the mechanism for interoperating with the interactive model (and the user) during the simulation.

The second potential problem associated with model splitting is that the two models need to be synchronized during runtime. That is, the state update information needs be exchanged between the two models: modification to the state of the interactive model needs to be sent to the execution model in a timely fashion, and vice versa. For large-scale networks, there can be considerable amount of information exchanged between the two models causing significant overhead. Furthermore, since the interactive model and the execution model can be geographically distributed with relatively constrained connections in-between (i.e., with large latencies and low bandwidths), the update information may not be delivered without significant delay. To overcome this problem, we propose a *real-time monitoring and control framework* to automatically and efficiently synchronize the states of the interactive and execution models. This is achieved by reducing the volume of update information and dynamically adjusting the synchronization intervals in response to user input.

The third problem with model splitting is about the memory consumption. The execution model is engineered to run on parallel machines to cope with the computational demands of large-scale network experiments. The interactive model, however, runs on a user workstation, which does not have sufficient resources to handle large-scale experiments in the same way as the execution model. We need to conserve memory for both models, particularly for the interactive model. We propose a two-pronged approach to solve this problem. First, we use *model replication* to minimize the memory footprint of large-scale network models. Model replication is a technique for constructing large network topologies using light-weight copies of identical sub-structures. This method is especially effective at conserving the memory required to conduct routing on the simulated network. Second, for the interactive model, we use an efficient *model caching algorithm* on top of a database management system for handling experiments that cannot be contained in the main memory. We extend network scripting to embed functionalities within the interactive model which allow for transparent paging of the interactive model to and from the database.

Network scripting is presented in section 3.3 and our framework for real-time interaction and control is described in section 3.4. Our model caching technique is explained in section 3.5. A detailed discussion of model replication is postponed until chapter 4.

3.3 Network Scripting

Network scripting is a technique of embedding simple annotations within the implementation of network models to help create a consistent representation of the target network both in the form of the interactive model (for users to create, configure, visualize, and control network experiments), and the execution model (for efficient execution of the network experiments on parallel platforms). Using the annotations, network scripting also automatically creates the mechanisms to synchronize the runtime state of the models during the experiment execution.

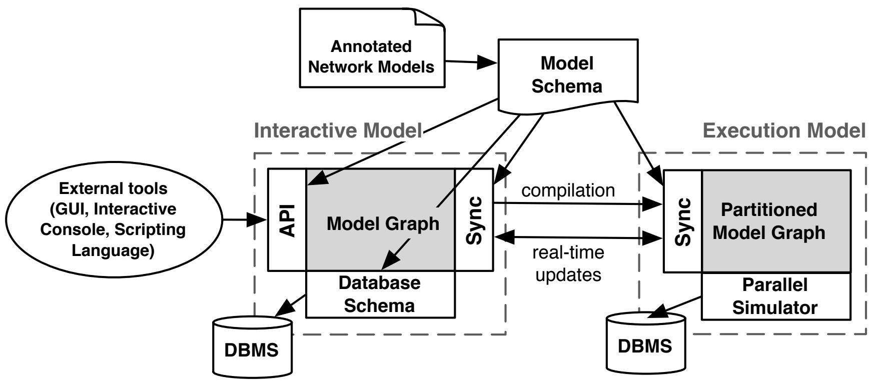


Figure 3.2: Network scripting for consistent interactive and execution models.

3.3.1 Annotations

Figure 3.2 illustrates the procedure and the functioning components of network scripting. The developer annotates the implementation of network models. All basic network elements, such as subnetworks, routers, hosts, links, interfaces, and protocols, are represented as *model nodes*, each containing the state of the corresponding network element and defining the logic and behavior of the network element. A network model is organized as a hierarchy of model nodes: a network may contain subnetworks, routers, hosts, and links; a router or host may contain one or more network interfaces and a stack of protocols.

With network scripting, the model nodes shall be embedded with annotations to indicate the relationship of the model nodes within the hierarchy and the access types of the state variables. More specifically, the model nodes shall contain *structure annotations*, which are used to ensure the model nodes can be instantiated properly in relation to one another. For example, a network can have routers as its children, but not the other way around. In this case, a *child-type* annotation is used to specify whether a model node can be composed of specific types of model nodes as its children. The annotations can also specify the minimum and maximum number of instances allowed for a specific type of model node. For example, a router requires that it contain at least one network interface, but at most one IP protocol.

There are four basic types for *state annotations*: configurable, controllable, mutable, and sharable. A *configurable* state annotation is used to mark a variable that can be set by the interactive model at the configuration stage, in which case the value needs to be propagated to the execution model before setting up the experiment. Each configurable state variable can also be defined with a default value: if a configurable state variable is not specified by the user, the simulator will use the default value. A *controllable* state annotation is used to mark a variable that can be modified by the interactive model during the experiment and the value needs to be propagated to the execution model in real time. Both configurable and controllable variables can also specify their minimum and maximum values. A *mutable* state annotation indicates that when the variable is modified by the execution model during the experiment, its value needs be propagated to the interactive model in real time. The mutable and controllable variables are essentially used for online monitoring and steering of network experiments. A *sharable* state annotation indicates that the variable can be shared among the replicated instances of the model node, in which case only one copy of the state variable needs to be maintained in the simulation. In chapter 4, we give a detailed explanation of our model replication technique, which is designed to reduce the memory consumption of large network models.

The annotated network model is parsed to generate the *model schema*, which contains the meta-information about the state variables of the model nodes and the structural relationship of the the model nodes. The model schema is then used to generate the interactive model and the execution model.

The interactive model maintains a lightweight representation of network model instances, which we call the *model graph*. The model schema is used to generate three major components of the interactive model. It generates an *application programming in-*

```
class NIC: public ModelNode {
  annotations {
    shared configurable
    long bitrate {
      default=1e8;
      min=1e4;
    }
    shared configurable
    long maxqlen {
      default=65536;
      min=1024;
    }
    controllable configurable
    double dropprob {
      default=0;
      min=0; max=1;
    }
    mutable long inbytes = 0;
    mutable long outbytes = 0;
    child_type<NICQueue>
    queue { min=1; max=1; }
  };
  Timer sendtimer;
  ...
  void receive(Packet* pkt);
  void send(Packet* pkt);
};
```

(a) Annotated model node.

```
class NIC extends ModelNode {
  NIC(ModelNode* parent) {
    super(parent);
  }

  long getInitBitrate() {..}
  void setInitBitrate(long v) {..}

  long getInitMaxqlen() {...}
  void setInitMaxqlen(long v) {..}

  double getInitDropprob() {..}
  void setInitDropprob(double v) {..}
  double getDropprob() {..}
  void setDropprob(double v) {..}

  long getInbytes() {..}
  void getOutbytes(long v) {..}

  List<NICQueue> getQueue() {..}
  NICQueue addDropTailQueue() {..}
  NICQueue addREDQueue() {..}
}
```

(b) Generated interactive model node.

```
class NIC: public ModelNode {
  class shared: public
  ModelNode::shared {
    Configurable<long> bitrate;
    Configurable<long> maxqlen;
    shared() :
      bitrate(1e8),
      maxqlen(65536) {}
  };
  Controllable<double> dropprob;
  Mutable<long> inbytes;
  Mutable<long> outbytes;
  ChildList<NICQueue> queue;
  Timer sendtimer;
  ...
  NIC() :
    ModelNode(new shared()):
    dropprob(0), inbytes(0),
    outbytes(0) {}
  NIC(shared* s) :
    ModelNode(s), dropprob(0),
    inbytes(0), outbytes(0) {}
  ...
  void receive(Packet* pkt);
  void send(Packet* pkt);
};
```

(c) Generated execution model node.

Figure 3.3: An example of network scripting.

terface (API), which contains the definition of the *interactive* model nodes. An interactive model node is a memory-efficient representation of the corresponding model node defined in the annotated network model, which contains only the configurable, controllable and mutable state variables. The API provides a common framework for all external tools, such as the graphical user interface, the interactive console, and the scripting language, which are used to configure and interact with the execution model visually or programmatically.

The model schema generates the *database schema*, which is used to marshall model nodes to and from an external database. The network model is persisted to the database so that one can repeat the network experiment or reuse any element of the network model. Persisting to a database also allows us to handle large-scale network models out of core.

The model schema also generates the *synchronization mechanism.* After the user finishes the configuration of the network model and the specification of the target runtime environment, this information needs to be "compiled" and sent to the execution model. Also, preprocessing needs to be done at this stage, such as the automatic IP address assignment for the network interfaces, the pre-calculation of static routing tables, and the partitioning of the network model for the target parallel platform. During the execution of the experiment, the interactive model can update the controllable variables, which need to be propagated to the execution model in real time; similarly, the mutable variables can be updated by the execution model and need to be propagated to the interactive model in real time. In section 3.4 we describe the real-time interactions between the two models in more detail.

The execution model maintains a full-fledged representation of network model instances, which we call the *partitioned model graph*. The execution model includes both annotated and unannotated states of all model nodes already partitioned to run on the parallel machines. For the annotated states, the execution model uses the model schema to create a variable map so that each state can be individually retrieved and modified. The execution model also provides a special treatment for sharable state variables: it groups all sharable states within a model node together so that the entire collection can be shared among replicated mode nodes. We discuss model replication in more detail in chapter 4.

The model schema generates the *synchronization mechanism* for the execution model, so that it can import information about the network configuration and the preprocessed results from the interactive model. The execution model also needs to receive real-time updates of controllable variables from the interactive model, and send real-time updates of both controllable and mutable variables to the interactive model. The partitioned model is executed by the parallel simulator on the parallel platform, which generates simulation results and trace data to be stored in a distributed database or a parallel file system.

3.3.2 Implementation

We implemented network scripting in our PRIMEX simulator. We annotated the network model implemented in C++. Figure 3.3(a) shows an example of an annotated model node, which represents a network interface. In the `annotations` block, we defined three configurable variables: `bitrate` is the bandwidth of the network interface, `maxqlen` is the maximum queue length, and `dropprob` is the packet drop probability. Two of them are also marked as shared. We also defined two mutable variables (`inbytes` and `outbytes`) for collecting the statistics on the number of bytes received and sent by the network interface during simulation. A network interface is required to have one and only one child of type `NICQueue`—either a drop-tail queue or a RED queue depending on the configuration. There are also unannotated variables defined within the model node. Here we show only `sendtimer`, which is used for scheduling packet departure events.

The annotated network model is parsed to generate the interactive model and the execution model. Figure 3.3(b) shows the corresponding interactive model node, which is implemented in Java. There is also an XML schema (not shown) generated from the annotated network model. For each annotated variable in a model node, the interactive model provides the corresponding getter and setter methods. If the variable is configurable (such as `bitrate`, `maxqlen`, and `dropprob`), we define `getInit` and `setInit` methods so that the user can check and modify the variable during model configuration. If the variable is controllable (such as `dropprob`), we define both getter and setter methods so that the user can query and modify the value of the variable during the runtime. If the variable is mutable (such as `inbytes` and `outbytes`), we only need to provide the getter method, since the execution model will update this variable in real time.

In the example, the network interface has one child type, `NICQueue`; we define a `getQueues` method to obtain a list of network queues (although only one is allowed according to the model schema). For each model node type derived from `NICQueue`, we create a separate `add` method. The model schema identified only two derived model node types: `DroptailQueue` and `REDQueue`; that's why there are two `add` methods.

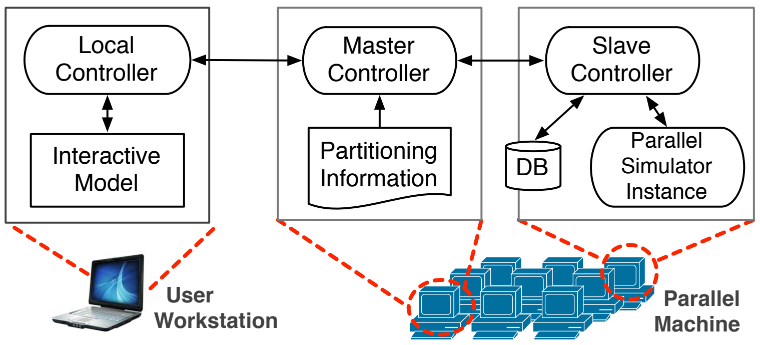


Figure 3.4: The monitoring and control framework.

We implemented a graphical user interface (using Prefuse [Hee04]) for the users to visualize and interact with the network model. We also implemented an interactive console in Python, which uses a python interpreter written in Java [Jyt]. PRIMEX allows the user to specify the network model in XML, Python, or Java. All these tools are developed using the above API generated from the model schema.

Figure 3.3(c) shows the definition of the corresponding execution model node for the network interface in C++. All sharable variables (`bitrate` and `maxqlen`) are put together and defined in an inner class, called `shared`. The default constructor of the model node creates the shared region when a new model node is created; however, if the new model instance is “replicated”, the second constructor is used so that the region is shared among the replicated instances. The configurable, controllable, and mutable variables are all wrapped using templates (`Configurable`, `Controllable`, and `Mutable`) for a uniform treatment of all data types when synchronizing with the interactive model. In addition, for controllable and mutable variables, the templates also track modifications so that the changes can be propagated to the interactive model during runtime.

3.4 Real-time Interactions

To support real-time interaction with network experiments, the interactive and execution models must synchronize their runtime state by exchanging state updates. In this section, we describe the design and implementation of an online monitoring and control framework for this purpose.

The monitoring and control framework, shown in figure 3.4, consists of: 1) a local controller, collocated with the interactive model at the user workstation; 2) a master controller, chosen to run at one of the parallel machines; and 3) the slave controllers, each running together with the corresponding parallel simulator instance on the parallel platform. The master and slave controllers are collocated with the execution model. When the execution model modifies the controllable or mutable states, the simulator instance generates state updates and sends them to the slave controller collocated on the same parallel machine. The slave controller then forwards the state updates to the master controller, which sends them to the local controller at the user workstation. The local controller eventually updates the interactive model. In the opposite direction, when controllable states are modified in the interactive model, which generates the state updates and forwards them to the master controller. The master controller, with the help of the partitioning information, sends the state updates to the corresponding slave controller, which modifies the execution model accordingly.

We implemented two mechanisms for collecting data from the execution model. One is a *monitor*, which is associated with a state variable. A monitor can output data, periodically (with a predefined interval), on-demand (whenever the state changes), or both (by exporting the data at most once for a given period if it is changed). The other mechanism is an *aggregator*, which is associated with a collection of state variables. An aggregator is used to produce the aggregate statistics from a collection of state variables, and can thus reduce the amount of data that needs to be collected. Note that, in addition to supporting real-time interaction, these mechanisms can also be used to support data collection for postmortem analysis, by storing the data in a collocated database or using a parallel file system.

The interval at which the execution and interactive models exchange state updates determines the responsiveness and timeliness of the interaction. Ideally, the interval should be kept small. However, the connection between the interactive and execution models may be constrained, making it infeasible to send updates at a high frequency. We propose two techniques to reduce the volume of state updates. The first method defines an *area of interest*, which is the set of interactive model nodes that a user is currently viewing or interacting with. A user (or program) cannot access the runtime state of a model node beyond the area of interest. The interactive model keeps track of the set of model nodes that are currently viewed by the user and informs the execution model to only send the state updates within the area of interest.

The second method uses a *rate limiting* mechanism, which dynamically adjusts the interval at which state updates are sent. As a user interacts with an experiment, the area of interest may grow to be very large, and in extreme circumstances it may include the entire execution model. This could cause a significant backlog of state updates to accumulate on the path from the execution model to the interactive model. To prevent the backlog, we can dynamically adjust the interval of the state updates so that the rate stays below the capacity of the system. This would ensure that the updates can be delivered in a timely manner, however, at a lower rate. We implemented the rate limiting mechanism in the simulator to control the time interval between sending batches of updates for all mode nodes within the area of interest.

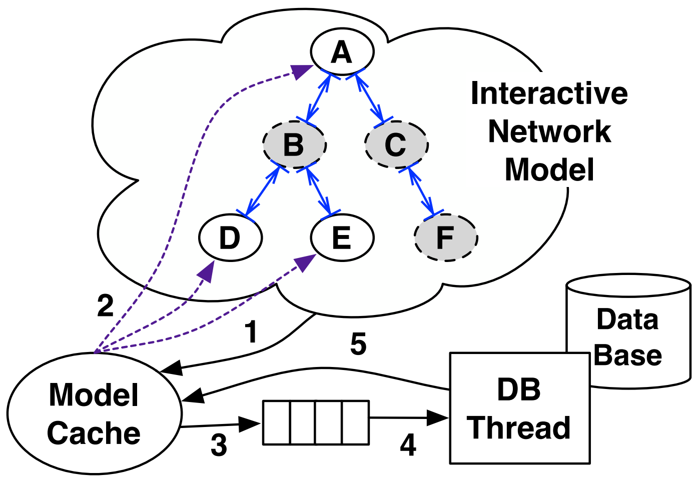


Figure 3.5: The model caching mechanism.

3.5 Model Caching

The interactive model needs to operate on large network models potentially too big to fit in the main memory. We present a model caching technique for out-of-core processing, which transparently marshals the interactive model nodes to and from a database as they are created and modified.

3.5.1 Design

Figure 3.5 shows a basic design of the model caching mechanism, which we explain below. Network scripting automatically creates an interactive model node for each annotated network model node. All interactive model nodes extend a base class that includes the functionalities for out-of-core processing. The user runs the external tools, such as the graphical user interface, in separate threads to create, modify or delete the model nodes. The interactive model registers these events with the model cache, which is also run as a separate thread (step 1 in the figure).

The model cache maintains a working set of model nodes that have been recently accessed or modified. The model nodes that are not in the working set must be persisted to the database. To do this, we use an advanced Java referencing scheme, in which a model node refers to its parent and children using *soft references*, as opposed to *strong references* typically used in Java. The main difference between strong and soft reference is that softly referenced objects can be reclaimed by the Java Virtual Machine (JVM) during the garbage collection. The model cache maintains a strong reference to all nodes within the working set to ensure that they cannot reclaimed by the JVM (step 2). When new model nodes are added to the working set, and if the cache is full, the model cache needs to evict existing model nodes by removing the strong references to them and by sending requests to the queue to have the model nodes written back to the database (step 3). When the interactive model accesses a model node, it checks to ensure the node is in memory, and if it is not, the model cache will send a request to load the node from the database and it will wait for the completion of the requested operation (also, step 3). We designate a separate thread, called the DB thread, to handle the requests from the model cache and perform the corresponding database operations (step 4). When the DB thread finishes loading a model node, it will unblock the waiting model cache thread (step 5).

3.5.2 Caching Policy

An important aspect in the design of the model caching mechanism is the choice of the cache eviction policy. The eviction policy needs to take advantage of the access locality. We study the typical access pattern of the interactive model via an experiment. We constructed a synthetic network connecting 216 campus networks [Nic] in a four tier topology. The model contains a total of 425,369 models nodes including all the hosts, routers, network interfaces, links, and subnetworks.

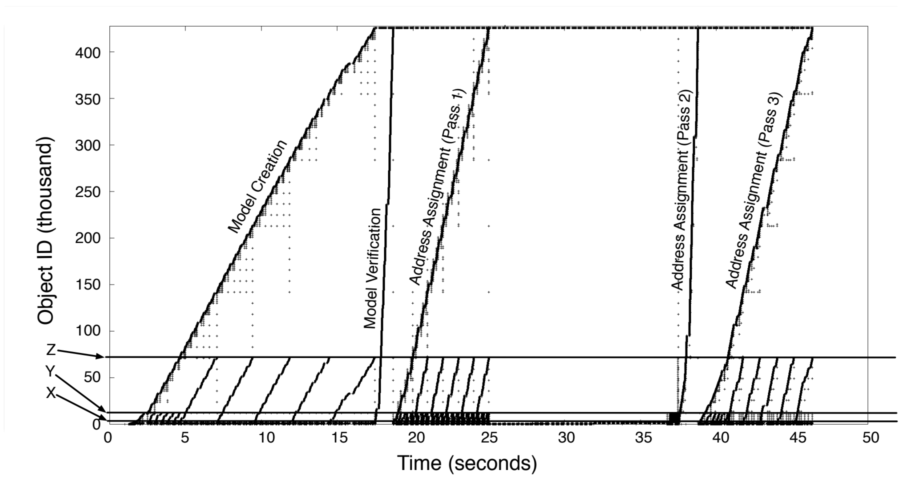


Figure 3.6: The access pattern for creating a campus network.

Figure 3.6 shows the results. The y-axis is the id of the model nodes being accessed. The diagonal lines indicate the model nodes are accessed in sequence at different preprocessing stages. For example, the first diagonal line shows that the model nodes (425,369 of them) are created one by one. The network topology has multiple levels and replications are applied at each level. At the bottom level are the campus networks, each with around 2K model nodes (denoted as X in the figure); the next to the bottom level has around 12K model nodes (Y); and the level above it has around 72K model nodes (Z). The figure shows that lower level model nodes are more frequently accessed because they are replicated by the other model nodes and they contain the sharable state variables.

The campus network topology and its replications account for nearly all of the locality seen in our models, as was the case in this example. Since the model nodes are accessed mostly sequentially, we choose to use a modified FIFO eviction policy. We select the victim to be the earliest created model node that is not a network and has not been replicated. This allows the model cache to give priority to networks and model nodes that have been replicated in memory.

3.5.3 Database

Another important aspect in the design of model caching is the choice of the database and the supporting framework for persisting the interactive network model. We first implemented the interactive model using the Java persistence architecture (JPA), which is a standard programming framework for managing relational data in applications. We found, however, that the JPA incurs significant overhead when dealing with large datasets; as a result, it was unable to handle network experiments with more than a few hundred thousand model nodes. We went on implementing a custom persistence framework so that we could have full control over the overhead involved in persisting and retrieving objects to and from the database.

We also implemented two storage engines for the database. One uses the standard JDBC interface to store models in the Apache Derby database [Apaa]. Derby is a Java database with a stable and relatively fast implementation. It can be easily embedded within a Java application. The other one is a custom database based on files. The custom database takes a "write once" approach: once a model node is persisted on disk, it cannot be updated in place—a modification will cause the entire model node to be written out to replace the original one. This approach can benefit from large sequential writes, which are critical to good I/O performance, but at the cost of wasted disk space. Fortunately, the wasted space can be reclaimed offline.

3.6 Evaluation

3.6.1 Control and Monitoring Framework

To evaluate the effectiveness of using the area of interest and the rate limiting scheme, we conducted an experiment with a synthetic network model with 12K model nodes running on a Linux cluster. We placed the local controller, the master controller, and the slave

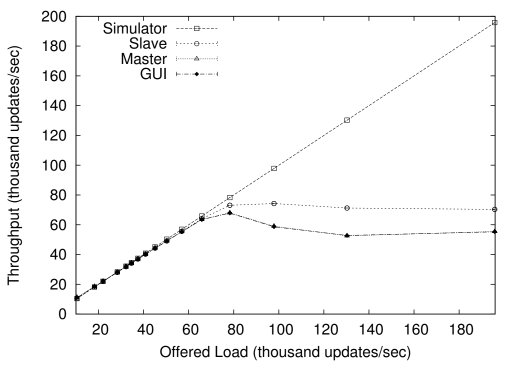


Figure 3.7: Throughput measurement at different locations.

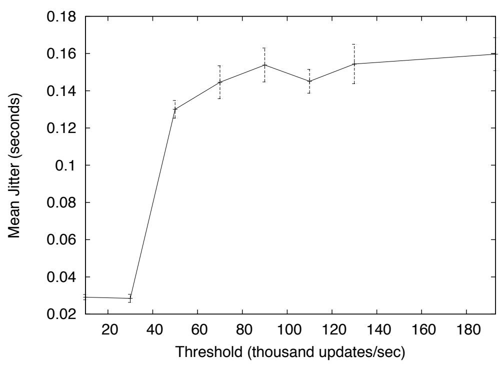


Figure 3.8: Jitter measured at the interactive model.

controller on three separate machines connected by a gigabit switch and measured the throughput at the simulator and the three controllers. First, we fixed the area of interest to encompass 75% of the model and changed the update frequency to produce different loads. Figure 3.7 shows the throughput at different measurement locations, averaged over ten separate trials (the confidence intervals are too small to be seen on the figure). Although the simulator can output updates at a high rate, the throughput peaks at around 74K updates per second at the slave controller, and around 68K updates per second at the master and local controllers.

This experiment shows the throughput limitations of the monitoring and control framework implementation. If the offered load is higher than the achievable throughput, the system may become congested, which would cause further delays for the updates to reach the interactive model. We conducted another experiment, this time by changing the area of interest at the interactive model to oscillate between 25% and 75% of the model every 5 seconds. We also enabled the rate limiting mechanism to control the rate at the simulator to stay below a given threshold. We varied the threshold in the experiment and measured the mean jitter, calculated as the difference in the arrival time between the successive batches of updates. Figure 3.8 shows the results. We observe that the jitter increases dra-

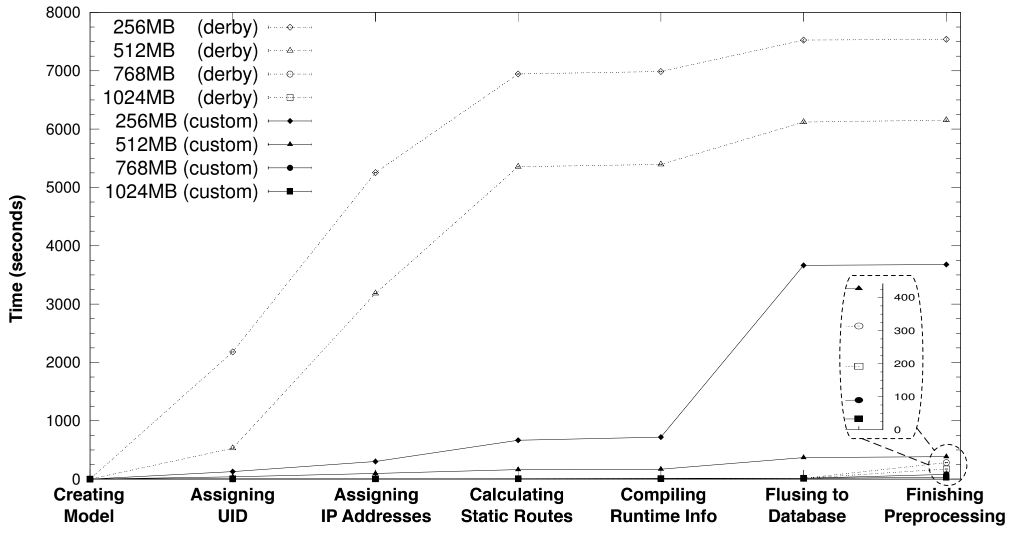


Figure 3.9: Performance of different database storage engines.

matically between 30K and 50K updates per second, which is before the peak throughput. This could be attributed to the lack of rate limiting within the batch. The experiment nevertheless demonstrates that, with the proper threshold, the rate limiting mechanism can deliver the updates with less jitter.

3.6.2 Model Caching

We conducted an experiment to compare the performance of Derby and our custom database. We used the same network model as in the previous experiment. We varied JVM's memory size from 256 MB to 1024 MB, and measured the time to create and preprocess the interactive model. Figure 3.9 shows the cumulative time averaged among ten trials (the confidence intervals are too small to be seen). The x-axis shows the preprocessing stages in order. With 1024 MB of memory the entire model fits in memory, and at 768 MB about 90% of the model is in memory. Both databases spent the majority of the time flushing the model to disk. Our custom database is about 5 times faster than Derby.

With less memory (512 MB), the performance difference between the two databases becomes significant. Derby must preform a lot of small-size random writes for the objects that are modified and flushed, while the custom database benefits from the "write once" approach, which sequentializes the writes. The results show a 15x performance improvement. For 256 MB, the time it takes to flush to the database increases dramatically for our custom database. This is because of the significant increase in the number of interleaved read and write operations. The custom database is optimized for writes, but not reads; the interleaved read and write operations caused the I/O cost to soar.

3.7 Conclusion

Model splitting divides a network model into an interactive model and an execution model in order to separate the user-facing concerns, such as visualization and interaction, from the concerns for the efficient execution of network models on parallel computers. The consistency problem between the two models is addressed using network scripting, which allows us to extract the model schema and automatically generate the interactive and execution models using simple annotations. The interoperation between the interactive and execution models is accomplished by a real-time monitoring and control framework with data filtering and rate limiting mechanisms to cope with a constrained connection between the two models. Finally, we presented a modeling cache scheme which utilizes an external database to enable processing of large-scale interactive models.

CHAPTER 4

REDUCING MEMORY CONSUMPTION

Memory consumption is a critical problem for large-scale network simulations. The enormous memory footprint needed for maintaining routing tables can severely obdurate scalability. Reducing the space requirement of routing tables is key to minimizing the space complexity of large-scale simulations. Previous work on reducing memory consumption in network simulation has primarily focused on reducing routing state. Routing state, however, is not the only consumer of memory in large-scale network models. We feel that a more systematic approach to the problem should be taken.

We propose a new technique called *model replication* which exploits replicated substructures within network models to dramatically reduce the space complexity of network models. One can group the memory consumption within a network simulator into the following three categories:

- *Structural:* This type of information deals with the topology of the network model. For example, the data needed to keep track of which hosts are in which networks, which interfaces are attached to which hosts, and how to route between hosts, would all be considered structural information. We further sub-classify structural memory into *routing state* and *model structure.* Routing state is the information needed to make forwarding decisions in the network, and model structure is everything else.
- *Instance State:* This type of information deals with parameters of specific elements in the network model. The memory required to store interface and link bandwidths, delays, host addresses, and other state that is associated with specific entities within the model, would be considered instance state. We can further sub-classify instance state as *sharable* and *internal*. Sharable instance state is state that is common to many entities within a network model and changes relatively infrequently. Information such as the link bandwidth or type of TCP protocol that is run on a host

would be considered sharable. Information such as the IP of an interface or the packets within an output buffer are examples of internal state; each instance must have its own mutable copy of the state.

- *Ephemeral:* In packet oriented network simulations, applications exchange data using discrete packets. Ephemeral state refers to short lived objects within the simulator such as network packets and TCP sessions states. The proportion of ephemeral state is largely dictated by the specific traffic patterns within a network model.

By exploiting replications within the network model, one can reduce memory the complexity of structural and instance state. Ephemeral state is primary related to the specific traffic patterns within the network model. Modeling traffic patterns in beyond the scope of this dissertation and as such, we focus our attention to reducing the memory complexity of structural and instance state of network models. Section 4.1 presents our *model replication* technique. We first argue that one can build network models using replicated sub-structures in section 4.1.1 and then in section 4.1.2 we briefly describe *spherical routing*, a novel routing scheme which reduces the space complexity of routing state. Section 4.2 presents the design of spherical routing. We then discuss our work on reducing the amount of memory used to store both instance state and model structure in section 4.3. The chapter is then wrapped up with an evaluation in section 4.4 and a conclusion in section 4.5.

4.1 Model Replication

Model replication is designed to allow users to create memory-efficient network models by sharing both structural and instance state among the model nodes that are replicated from the same base structure. The topology of the network model is represented as an augmented tree that encodes the replications. Figure 4.1 shows an example of a small

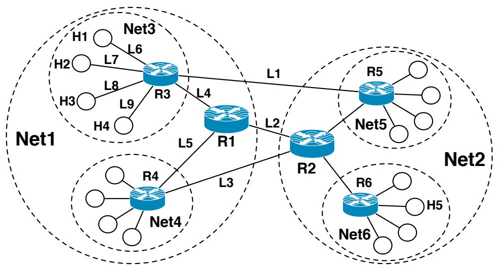


Figure 4.1: An example network model.

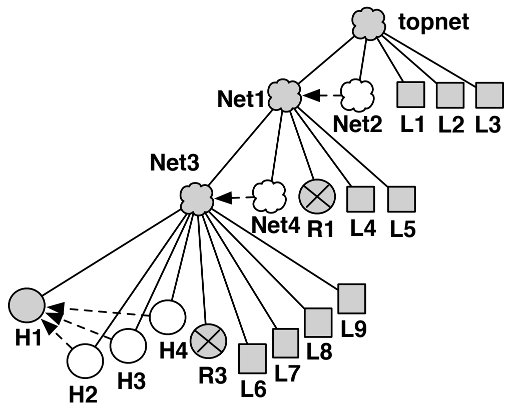


Figure 4.2: Internal model structure configured with routing spheres.

topology and figure 4.2 contains its corresponding internal representation. The top-level network contains two identical subnetworks, Net1 and Net2, each also containing two identical subnetworks. In the internal representation, Net2 makes a reference to Net1, which indicates that both share the same sub-structure and use the same set of sharable variables. Similarly, Net4 makes a reference to Net3, and H2, H3 and H4 all make a reference to H1. At the preprocessing step, the tree will be expanded; however, all replicated model nodes still maintain only one copy of sharable variables. The example network contains six subnetworks, six routers, twenty links, and sixteen hosts in the two subnetworks at the top level. Eventually, only one set of sharable variables is needed for each of the two subnetworks (Net1 and Net3), the two routers (R1 and R3), the links (L1 - L9), and the one host (H1).

4.1.1 Building Network Models Using Replicated Sub-Structures

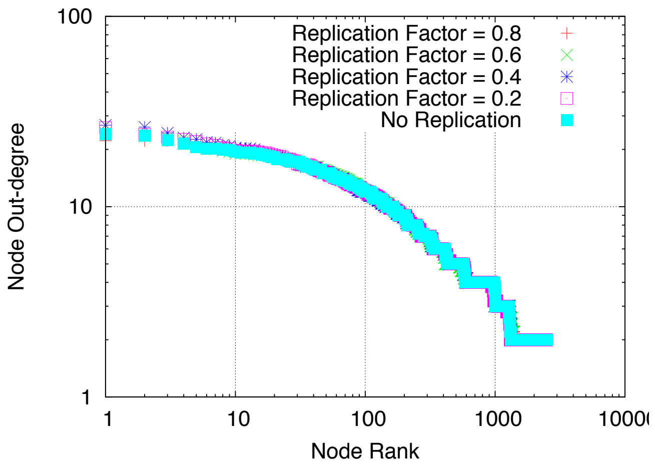
Network generators, such as BRITE [MLMB01] and Orbis [MHK$^+$07], are often used to construct large-scale network models. These network generators make use of metrics and trends extracted from measurements of real networks. Having the topologies used in simulation match the characteristics of real networks is critical in accurately representing the behavior of network applications and protocols [PF97]. We argue that realistic network models can be built with replicated network structures while maintaining the same network characteristics. In this section we explore the effects of composing network models using structural replications.

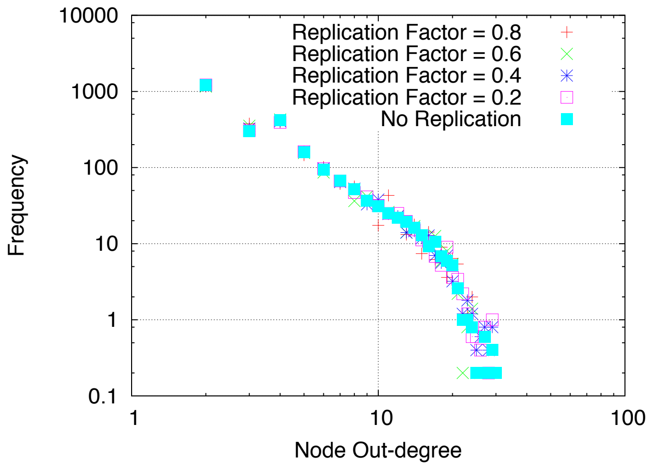
The intuition that network models can be built using replicated structures is bolstered by Leskovec's recent work in building realistic network models using Kronecker graphs [LCK$^+$10]. The basic idea is that, starting from a small graph (called the initiator) which captures some essential qualities of a target network, one can recursively build a large network mathematically through an operator (called the Kronecker product). The resulting network graph shares the same network structure of the initiator at all different levels. It has been shown that with a properly chosen initiator, the resulting network graph can preserve all the fundamental network characteristics of the Internet. This includes static graph patterns like heavy-tail distributions for in-degree, out-degree, eigenvalues and eigenvectors, and temporal evolution patterns such as the densification power law (i.e., the network diameter shrinks as the network grows).

For this study, network graphs generated by BRITE [MLMB01] were used. BRITE can produce hierarchical network models using a top-down method. The top-down method first generates a network topology for the autonomous systems (ASes). Then for each AS, BRITE generates a router-level topology. Finally, BRITE generates the overall network topology by merging the AS-level and router-level topologies and connecting the routers belonging to different ASes. To introduce structural replication, BRITE was modified to generate the router-level topologies according to a given replication factor α. α is defined as the fraction of the ASes having the same router-level topologies as other ASes. For example, if one generates 100 ASes and the replication factor is 0.2, 20 ASes will have router-level topologies that are exact replicas of those randomly chosen from the other 80 ASes.

In the experiment, the modified BRITE was used to generate network topologies with the replication factor varying from 0 (no replication) to 0.8. Figure 4.3 plots the statistics of generated network topologies that contain 50 ASes, each with 50 routers. The first plot in figure 4.3 is a log-log plot of the node's out-degree versus its rank in the sequence of decreasing out-degrees for all nodes. The second is a log-log plot of the frequency versus the out-degree, and the third is a log-log plot of the number of nodes within a given number of hops. For each level of replication, 20 different topologies are generated, and the plots only show the average results. The standard deviation is not shown here because it would be indistinguishable in the plots. The same tests were run on topologies of up to 100 ASes each with 2000 routers. The results (not shown) are very similar to those here. Overall, little difference is seen among the results from different replication levels.

The characteristic path lengths and cluster coefficients of the generated network graphs are also calculated. The characteristic path lengths include the mean and standard deviation of the shortest-path lengths among all nodes. The cluster coefficients capture the level of connectivity among all the nodes. For any given node, the node and its neighbors form a cluster from which a cluster coefficient can be calculated, by dividing the number of connections among all the nodes in the cluster by $N(N-1)$ (assuming N is the size of the cluster). The mean and standard deviation among all the nodes in the network graph are then calculated. The results are shown in table 4.1. Like before, the replication factor does not have any obvious effect on these values. All the results above strongly suggest that one can use structural replication to build realistic large-scale network models.





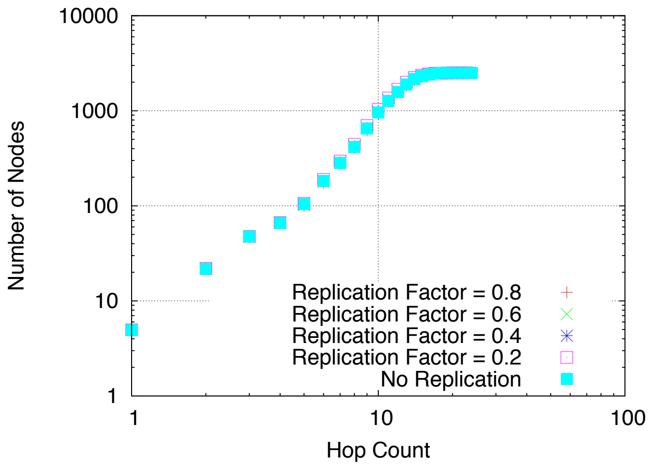


Figure 4.3: Network characteristics of BRITE-generated graphs with varying replication factors.

α	**Path avg**	**Path std**	**Cluster avg**	**Cluster std**
0.0	11.28	0.34	$7.76E-5$	$2.33E-6$
0.2	11.02	0.28	$7.91E-5$	$6.40E-6$
0.4	11.3	0.24	$7.76E-5$	$2.06E-6$
0.6	11.21	0.23	$8.05E-5$	$6.66E-6$
0.8	11.41	0.42	$7.56E-5$	$3.39E-6$

Table 4.1: Characteristic path lengths and cluster coefficients.

4.1.2 Spherical Routing

Spherical routing is a novel static routing scheme for large-scale network simulations. Spherical routing pre-calculates the forwarding tables and conducts routing within so-called *routing spheres*, each with a user-specified routing strategy. A routing sphere is defined internally as a network graph (with vertices and edges) on which a single routing strategy is applied. A routing strategy can either be based on shortest paths, which is commonly used for intra-domain routing (such as OSPF and RIP), or based on routing policies dictated by the peering relationships between autonomous systems (as used by BGP for inter-domain routing). Using spherical routing, a network can be viewed as a hierarchy of routing spheres: a routing sphere of a sub-network is enclosed by the routing sphere of its parent network; the routing sphere of the sub-network is represented as a vertex in the network graph of the parent network's routing sphere. Within each sphere, a static forwarding table is calculated according to its routing strategy before simulation. During simulation, the simulator conducts packet forwarding according to the forwarding table and the location of the routing sphere with respect to its parent network's routing sphere.

The hierarchy of the routing spheres is defined by the modeler according to the network structure. The space complexity of the model can be reduced (1) by subdividing the network into multiple routing spheres so that the size of the forwarding tables can be reduced, and (2) by identifying routing spheres with identical network graphs. If two routing spheres have the same network structure, we can reduce the memory consumption by allowing the spheres to be replicated to share the same forwarding table. In general, spherical routing allows the modeler to control the trade-offs between modeling accuracy and space complexity.

4.1.3 Related Work

Most network simulators implement common routing protocols. Some even do so with the capabilities of performing large-scale routing experiments so as to capture network dynamics in great detail. For example, Griffin and Premore [GP01] studied the convergence behavior of inter-domain routing using a full-fledged BGP implementation in SSFNet [CNO99]. Bauer et al. [BYCK06] studied the stability and dynamics between OSPF and BGP using ROSSNet [YBB$^+$03].

There are also simulators tailored specially for inter-domain routing. BGP++ [DR03] implements BGP in the NS-2 simulator [BEF$^+$00] by porting from a public domain routing software called Zebra [Kun]. Follow-up work on BGP++ [DR04] proposed optimizations to reduce the space complexity. One technique is to change Zebra's memory recycle scheme to provide an architecture to enable the multiple information bases within a BGP router to share the same memory. C-BGP [QU05] only simulates the BGP decision process based on router configurations and network topology, but ignores timers and packet exchange between peers. The trade-off is, the simulator can reproduce the routing behavior algorithmically in a cost effective manner. However, the simulation cannot represent detailed routing effect from network dynamics.

In the above cases, the simulators can support large-scale network scenarios, and some even benefit from parallel and distributed simulation, the cost of running detailed routing protocols (both in time and space) is considerably high. It is certainly undesirable if the primary focus of the simulation is not on routing. In this case, routing should be considered a service to effectively forward packets within the simulation. Our work falls into this category.

There are two methods commonly used for implementing routing services in simulation: pre-calculation and on-demand. Given a network model, pre-calculation prepares the forwarding information off-line and the simulator queries this information to forward packets during run-time. Alternatively, the routes can be calculated on demand when a packet forwarding decision must be made at a router during simulation.

Global shortest-path routing flattens the network model to perform shortest path calculations. The calculations only consider reachability and do not consider routing policies for inter-domain routing. This is the default routing strategy used in the NS-2 simulator. For a network with N nodes, a flat shortest-path routing requires $\mathsf{O}(\mathsf{N}^2)$ memory space. To reduce the memory consumption, NS-2 allows hierarchical shortest-path routing. For simplicity, suppose a network is divided into K clusters ($\mathsf{K} << \mathsf{N}$), and each cluster is represented as node in the network graph. Further, each cluster is divided into K sub-clusters and therefore each cluster can also be represented as a graph of K nodes. If one continues in this fashion until all graph nodes are either routers or end hosts, the space complexity for the forwarding tables is $\mathsf{O}(\mathsf{KN})$.

To further reduce the cost, Huang and Heidemann [HH01] proposed algorithmic routing, which maps a given network topology to a k-ary tree and reassigns the node addresses to match the newly imposed tree. This pre-assigned order allows simple next hop computation without maintaining a route table. The space complexity is reduced to $\mathsf{O}(\mathsf{N})$, which comes from maintaining the address mapping between the original topology and the corresponding k-ary tree. Hiromori et al. [HYY$^+$03] proposed combining the algorithmic routing and the global shortest-path algorithm to reduce the amount of path inflation caused in the original algorithmic routing approach. One method is to find the near-optimal shortest-path tree by comparing the result of algorithmic routing with the result of the global shortest-path routing algorithm. Another method is to maintain a variable number of shortest-path trees for a given topology, and at run time choose the best path from the trees.

NIx-vector routing [RAF00] is an on-demand routing scheme. Routes are calculated on-demand via breadth-first search. The result is a sequence of NIC indices, which are compacted and stored in the packet for it to traverse through the network from source to destination (like source routing). Liljenstam and Nicol [LN04] proposed an on-demand algorithm for modeling BGP routing which relies on Gao's work on the peering relationships between autonomous systems [Gao01]. For on-demand routing, since it is unnecessary to store all forwarding tables, we see a huge memory savings. If the simulated traffic is only between a relatively small number of network nodes, the additional time used for computing the routes is justifiable given the substantial space savings. However, the performance degradation can be significant if the condition is not satisfied.

The aforementioned approaches all suffer from a problem. The calculated routes may differ from the true routes — sometimes significantly — such as in the case of algorithmic routing. Of course, the amount of difference depends on the network model, and the significance of the difference depends on the goal of the simulation. Spherical routing allows hierarchical routing with specific routing strategies (either shortest-path or policy based) within individual routing spheres. Further, spherical routing allows the modeler to define the boundary of the routing spheres to balance between accuracy and memory consumption. Spherical routing also uses a combination of pre-calculation and on-demand routing strategies to balance between computing cost and memory consumption.

4.2 Design of Spherical Routing

This section focuses on the design of the spherical routing algorithm. Section 4.2.1 describes how the algorithm works through a simple example. Section 4.2.2 introduces the addressing scheme and forwarding table structure which allows spherical routing to efficiently conduct routing within spheres and effectively share forwarding tables. We then present our BGP routing policy for spherical routing in section 4.2.3. Section 4.2.4 details how packets are forwarding within spherical routing and section 4.2.5 presents subtle issues related to how routes are calculated and packets are forwarded.

4.2.1 An Example

We use an example to show how spherical routing works. The user specifies a network model in a hierarchical fashion, where networks serve as containers for routers, hosts, links and sub-networks. Figure 4.4 shows a network consisting of two sub-networks with the same network structure (`Net1` and `Net2`), connected by three links (`L1`, `L2` and `L3`). In this case, the simulator simply marks `Net2` as a replica of `Net1` in its internal tree representation of the network (as shown on the right of figure 4.5). `Net1` consists of two sub-networks (`Net3` and `Net4`), one router (`R1`), and two links (`L4` and `L5`). Again, `Net4` is simply a replica of `Net3`. The latter consists of four identical hosts (`H1` to `H4`), one router (`R3`), and four links (`L6` to `L9`). `H2`, `H3` and `H4` are replicas of `H1`. Hosts and routers also serve as containers for network interface cards (NICs); we ignore them in the figure for simplicity. `Net1`, `R1`, `H1`, and `L1`, are examples of names one can use to identify network entities in the model. In simulation, all network entities (networks, routers, hosts, links, and interfaces) are assigned unique integer identifiers that capture the hierarchical structure of the model (we describe the hierarchical addressing scheme in section 4.2.2).

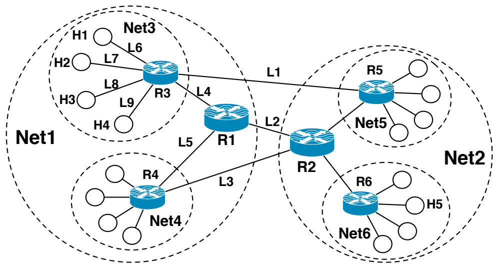


Figure 4.4: An example network model.

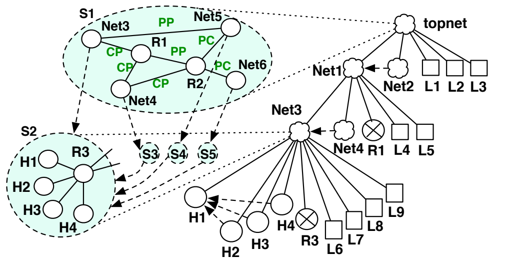


Figure 4.5: Internal model structure configured with routing spheres.

For each network in the model, the user can assign a routing sphere. A sub-network can also inherit the routing sphere of its parent network. For network entities within a network, the routing sphere is called the *owning routing sphere*. Each routing sphere must specify a routing strategy based on either shortest paths or routing policies. In the example, the top-level network (named `topnet`) is assigned a routing sphere for policy-based inter-domain routing. `Net1` and `Net2` are not specified with routing spheres, so they inherit the routing sphere of `topnet`, and their network graphs are flattened into the network graph of the owning routing sphere. `Net3` through `Net6` each have their own routing sphere specified for shortest-path intra-domain routing, and the routing spheres for `Net4` though `Net6` are actually replicas of `Net3`. If a sub-network is assigned a routing sphere, the sub-network is represented as a "super-node" in the network graph of the routing sphere for the parent network. The routing sphere for the parent network is called the *parent routing sphere*; the routing sphere for the sub-network is called the *child routing sphere*. Like routers, a super-node can have multiple network interfaces. Unlike routers, the distance between the interfaces on the same super-node is not always zero, since the interfaces may actually belong to different hosts or routers. In this case, the distance should be calculated based on the network topology and the associated routing strategy of the child routing sphere. The example defines five routing spheres, `S1` through

S5, as shown on the left of figure 4.5. Because of replication, there are actually only two routing spheres in this model. Note that topnet's routing sphere, S1, uses policy-based routing. The user needs to classify all the links in its network graph as either customer-provider (we mark them as either cp or pc depending on whether the left end-point is a customer or a provider), peer-peer (pp), or sibling-sibling (ss). This classification reflects the BGP peering relationships between the autonomous systems [Gao01].

Once a network model has been specified, the static forwarding table for each routing sphere is calculated based on its topology and associated routing strategy. The resulting forwarding tables are stored together with the network model. R-trees [Gut84] are used for a compact representation of the forwarding table entries and fast look-up during packet forwarding (the details of the forwarding table data structure are described in section 4.2.2). The forwarding tables are loaded during model instantiation when the simulation starts, and the simulator creates the data structure for all network entities. For a replica, it creates only a shallow copy of the data structure to save space. All replicated network entities share the same network structure as well as all static configurations (such as the maximum TCP window size). Each replica still needs to maintain its own run-time state (such as the TCP congestion window size). In the case of replicated routing spheres, they all share the same static forwarding table. However, they maintain their own copy of the run-time state (such as a cache used to improve the forwarding table look-up time).

Now, suppose a packet is sent from H1 to H5. At H1, the owning routing sphere, S2, determines that the destination is outside of this sphere and therefore consults its parent routing sphere, S1, for the cost of sending the packet out from its external interfaces to the destination. S1 has two external interfaces connected to L1 and L4 respectively. S1 finds that the destination is within Net6. Both Net3 (the source) and Net6 (the destination) are represented as super-nodes in the routing sphere S1. Using its forwarding table, S1 determines the cost from the interface at Net3 connected to L1 to Net6 is infinite (since there is no valley-free path), and the cost from the interface connected to L4 is 3 (hops). This information is used by S2 to settle a routing path from H1 to the external interface connected to L4 by consulting its own forwarding table. The idea of NIx-vector routing [RAZ01] is extended to work within a routing sphere. To forward the packet, three NIx-vectors are created as the packet enters the three routing spheres, S3, S1, and S5. The previous steps only apply to the first packet in the flow of packets from H1 to H5. The NIx-vectors and the result of the up-call from S3 to S1 are cached so that subsequent packets in the flow can be forwarded almost immediately.

4.2.2 Addressing Scheme and Forwarding Table Structure

All network entities (networks, routers, hosts, links, and interfaces) in simulation are uniquely identified using an integer. The simulator provides name services, which can translate from network entity names or IP addresses to unique identifiers (UIDs), and vice versa. Although using integers is efficient, UIDs must be able to capture the hierarchical structure of the model. As mentioned earlier, networks serve as containers for hosts, routers, links, and other networks. Similarly, hosts and routers are containers for network interfaces. A network model is represented as a tree (such as the one shown on the right of figure 4.5). If a network entity A is contained in another network entity B, A is a descendant of B (and B is an ancestor of A) in the tree representation. For example, in the network model shown in figure 4.5, router R1 and link H1 are both contained in network Net1. Given the UID of a network entity, one should be able to determine whether the network entity is contained within another network entity in an efficient manner. Further, since a forwarding table is shared among routing spheres with the same network structure (i.e., with the same network topology), one should be able to use relative identifiers (RIDs) in the forwarding table. That is, one should be able to translate from a network entity's RID to UID, and vice versa, in an efficient manner.

The RID of a network entity A is defined with respect to its ancestor B, denoted as $\mathsf{R}_{\mathsf{A}}^{\mathsf{B}}$, to be the rank of A in the post-order traversal (starting from 1) of the sub-tree rooted at B, as if all replicated nodes were expanded. Considering the network model in figure 4.5, $\mathsf{R}_{\mathsf{R3}}^{\mathsf{Net1}} = 5$ and $\mathsf{R}_{\mathsf{R1}}^{\mathsf{Net1}} = 21$. Note that, for simplicity, this example does not account for the interfaces contained within hosts and routers. The UID of a network entity A, denoted as U_{A}, is defined to be the RID of A with respect to the top-level network, i.e., $\mathsf{U}_{\mathsf{A}} = \mathsf{R}_{\mathsf{A}}^{\mathsf{topnet}}$. In practice, each network entity (A for instance) in the tree representation is simply labeled with an offset related to its parent (O_{A}) and its tree size (S_{A}). For example, $\mathsf{O}_{\mathsf{Net3}} = 0$, $\mathsf{S}_{\mathsf{Net3}} = 10$, $\mathsf{O}_{\mathsf{R1}} = 20$, and $\mathsf{S}_{\mathsf{R1}} = 1$. It is easy to see that:

$$\mathsf{R}_{\mathsf{A}}^{\mathsf{B}} = \mathsf{O}_{\mathsf{A}} + \mathsf{S}_{\mathsf{A}} + \sum_{\mathsf{a} \in \gamma_{\mathsf{A}}^{\mathsf{B}}} \mathsf{O}_{\mathsf{a}} \tag{4.1}$$

where $\gamma_{\mathsf{A}}^{\mathsf{B}}$ is the list of ancestors of A who are descendants of B. For example, $\gamma_{\mathsf{R3}}^{\mathsf{Net1}} = \{\mathsf{Net3}\}$, and $\mathsf{R}_{\mathsf{R1}}^{\mathsf{Net1}} = \mathsf{O}_{\mathsf{R3}} + \mathsf{S}_{\mathsf{R3}} + \mathsf{O}_{\mathsf{Net3}} = 4+1+0 = 5$. In this way, one need not expand the replicated nodes in the tree representation in order to calculate the RIDs. Suppose C is an ancestor of B, and B is an ancestor of A, it can easily be shown that:

$$\mathsf{R}_{\mathsf{A}}^{\mathsf{C}} = \mathsf{R}_{\mathsf{B}}^{\mathsf{C}} + \mathsf{R}_{\mathsf{A}}^{\mathsf{B}} - \mathsf{S}_{\mathsf{B}} \tag{4.2}$$

Equation 4.2 is used to convert between UID and RID.

The forwarding tables use RIDs to encode addresses within routing spheres. As discussed above, one can convert the RID of a node with respect to its owning routing sphere to UID, and vice versa. The forwarding table is a map from the source and destination RIDs to the next hop RID and associated cost. After all the forwarding table entries are determined, the following procedure is applied to compress the forwarding table. The entries are first sorted by the next hop RID, then by the cost, then by the source RID, and finally by the the destination RID. A linear scan of the sorted entries is then made where entries with the same next hop RID and cost are merged if the source and/or destination RIDs form a contiguous range. The forwarding table entries take the form $\langle[\mathsf{src}_{\mathsf{min}},\mathsf{src}_{\mathsf{max}}],[\mathsf{dst}_{\mathsf{min}},\mathsf{dst}_{\mathsf{max}}]\rangle \rightarrow \langle\mathsf{next\_hop},\mathsf{cost}\rangle$. This compression technique is particularly effective for routing spheres representing local area networks where all the hosts in the network use a common gateway to reach outside of the network.

To obtain the next hop cost, the forwarding tables are loaded at the start of the simulation and queried during packet forwarding using the source and destination RIDs. R-trees [Gut84] are used to store the compressed forwarding table entries with intervals for fast look-up. R-trees are similar to B-trees, but are used for spatial queries. The data structure splits the space hierarchically using nested bounding rectangles. The search is done recursively starting from the root of the R-tree searching for the child node that spatially overlaps the search rectangle. In our case, the search rectangle is a single point with the source and destination RIDs as its coordinates.

4.2.3 BGP Policy-Based Routing

After the user specifies the network model, we can pre-calculate the forwarding tables for the routing spheres. As mentioned earlier, spherical routing is for static routing, and therefore is not meant for studying the detailed routing dynamics (such as BGP convergence and stability). Rather, it should be used as a simulation service for realistic packet forwarding to study other aspects of large-scale networked systems or applications (such as caching policies for content distribution networks). For shortest-path routing, the calculation can be done using traditional shortest-path algorithms (such as Dijkstra's algorithm). In this section, we describe an algorithm for simplified BGP policy-based routing.

For common global-scale network simulations, BGP is expected to be applied at the routing sphere of the top-level network, where different subnetworks are treated as autonomous systems (ASes) and defined as individual routing spheres. For BGP, the relationships between the ASes play a critical role in determining the forwarding paths, in which case graph connectivity no longer implies reachability. Previously, Gao classifies the AS relationships into three categories: customer-provider, peer-peer, and sibling-sibling [Gao01]. By studying the common practice of BGP policy settings, it follows that legitimate BGP routes should be *valley-free paths*, which consist of an uphill segment and a downhill segment. The uphill segment is composed of zero or more customer-provider or sibling-sibling links and is optionally followed by a single peer-to-peer link. The downhill segment is comprised of zero or more provider-customer or sibling-sibling links. This observation was later confirmed through measurement studies that shortest valley-free paths are commonly selected as the forwarding paths on Internet [MQWZ05]. Our algorithm simplifies the shortest valley-free path calculation based on Gao's classification of AS relationships to derive the static forwarding tables.

The definition of a valley-free path can be expressed as a regular expression:

$$(\mathsf{cp} \cup \mathsf{ss})^*(\mathsf{pp} \cup \varepsilon)(\mathsf{pc} \cup \mathsf{ss})^* \tag{4.3}$$

where cp stands for a customer-to-provider link, pc is provider-to-customer, ss is sibling-to-sibling, and pp is peer-to-peer. The corresponding deterministic finite automaton (DFA) that recognizes this regular expression is a 5-tuple: $\mathsf{M} = (\mathsf{Q}, \Sigma, \delta, \mathsf{q}_0, \mathsf{F})$. The DFA has two states: $\mathsf{Q} = \{\mathsf{s}_0, \mathsf{s}_1\}$, and four input symbols: $\Sigma = \{\mathsf{pc}, \mathsf{cp}, \mathsf{pp}, \mathsf{ss}\}$. The transition function, $\delta : \mathsf{Q} \times \Sigma \to \mathsf{Q}$, is defined as follows:

$$\delta(\mathsf{q}, \mathsf{i}) = \begin{cases} \mathsf{s}_0, & \text{if } \mathsf{q} = \mathsf{s}_0 \wedge \mathsf{i} \in \{\mathsf{cp}, \mathsf{ss}\} \\ \mathsf{s}_1, & \text{if } \mathsf{q} = \mathsf{s}_0 \wedge \mathsf{i} \in \{\mathsf{pp}, \mathsf{pc}\} \\ \mathsf{s}_1, & \text{if } \mathsf{q} = \mathsf{s}_1 \wedge \mathsf{i} \in \{\mathsf{pc}, \mathsf{ss}\} \end{cases} \tag{4.4}$$

The DFA has a start state: $\mathsf{q}_0 = \mathsf{s}_0$, and two accept states: $\mathsf{F} = \{\mathsf{s}_0, \mathsf{s}_1\}$.

Our algorithm simplifies Gao's idea in calculating the shortest valley-free paths [Gao01]. The input to our algorithm is the network graph of a routing sphere (e.g., S1 in figure 4.5). The graph, $\mathsf{G} = (\mathsf{V}, \mathsf{E})$, contains vertices representing ASes as routers or super-nodes for

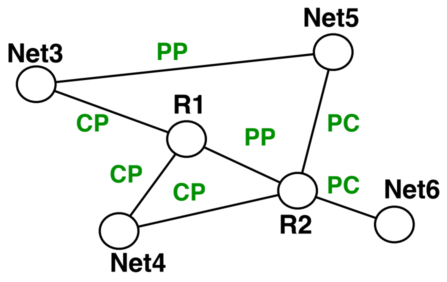


Figure 4.6: An example network model with BGP annotations.

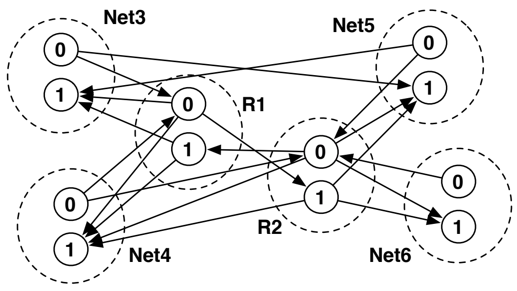


Figure 4.7: Corresponding BGP graph for calculating shortest valley-free paths.

child routing spheres. The links between the vertices are labelled with the proper AS relationships (customer-provider, peer-peer, or sibling-sibling). Our method is to construct another network graph G' based on G, so that running the shortest-paths on G' will result in the shortest valley-free paths on G.

We construct G' as follows. For each vertex $v \in V$, we add two vertices v_0 and v_1 in G' (corresponding to the two states in DFA). For each link $(u, v) \in E$, we add several links to G' in accordance with the AS relationships between the vertices. If the link (u, v) is a customer-provider link, assuming u is v's provider, we add (v_0, u_0), (u_0, v_1), and (u_1, v_1) to G'. If (u, v) is a peer-peer link, we add (u_0, v_1) and (v_0, u_1) to G'. If (u, v) is a sibling-sibling link, we add (u_0, v_0), (u_1, v_1), (v_0, u_0), and (u_1, v_1) links to G'. Figure 4.7 shows the BGP graph constructed corresponding to the network model in figure 4.6 (which is sphere S1 in figure 4.5).

G' contains links between $u \in \{u_0, u_1\}$ and $v \in \{v_0, v_1\}$, if and only if (u, v) is in G. That is, G' cannot contain paths that do not exist in G. Furthermore, since the links between $u \in \{u_0, u_1\}$ and $v \in \{v_0, v_1\}$ are only added if they obey the state transition rules as specified in the DFA, any path between $u \in \{u_0, u_1\}$ and $v \in \{v_0, v_1\}$ must be valley-free. Therefore, we can simply run the shortest-path algorithm on G' to find all of the shortest valley-free paths in G.

4.2.4 Packet Forwarding

In spherical routing, every sphere maintains its own forwarding table and can conduct forwarding locally. For packets with destinations outside of the routing sphere, the routing sphere needs to consult with its parent routing sphere to determine the outbound interfaces to send the packets out. The routing sphere is treated as a super-node in its parent routing sphere, which applies the same packet forwarding decision process. Thus, we can establish a recursion.

We first make a few definitions. Let S be the source node. It can be a host or a router, which is either the traffic source or an immediate node along the path through which a packet is forwarded to the destination. We denote Ω_S to be the owning routing sphere of S, and Ψ_S to be the set of network interfaces that S possesses. In our algorithm, S can also be a routing sphere that contains the source node (recall that a routing sphere is treated as a super-node in its parent routing sphere). In this case, Ω_S is S's parent routing sphere and Ψ_S is the set of interfaces of hosts and routers within routing sphere S that connect to the outside of S. When needed, we call these interfaces the *edge interfaces* of the routing sphere to distinguish them from the interfaces of a host or router.

For a given a routing sphere X, suppose i and j are network entities defined within X, the distance between i and j, denoted by $\omega(\mathsf{R}_\mathsf{i}^\mathsf{X}, \mathsf{R}_\mathsf{j}^\mathsf{X})$, can be determined by querying the pre-calculated forwarding table for X using the interfaces' RIDs, $\mathsf{R}_\mathsf{i}^\mathsf{X}$ and $\mathsf{R}_\mathsf{j}^\mathsf{X}$. The RIDs can be calculated using equation (4.1). The UIDs of all network entities contained within the routing sphere X shall range between $\mathsf{U}_\mathsf{X}^\mathsf{L} = \mathsf{R}_\mathsf{X}^\mathsf{topnet} - \mathsf{S}_\mathsf{X} + 1$ and $\mathsf{U}_\mathsf{X} = \mathsf{R}_\mathsf{X}^\mathsf{topnet}$. Note that the same range can still apply if we consider X as an individual router or host.

Algorithm 1 : **packet_forwarding**

```
Require: $S$, the node that conducts packet forwarding, and $T$, the UID of the destination node
1: if $U^L_{\Omega_S} \leq T \leq U_{\Omega_S}$ then
2:     find node $Z \in \Omega_S$ such that $U^L_Z \leq T \leq U_Z$
3:     for all $i \in \Psi_S$ do
4:         $d_i \Leftarrow \min_{j \in \Psi_Z}\{\omega(R_i^{\Omega_S}, R_j^{\Omega_S})\}$
5: else
6:     $\{d'_j, \forall j \in \Psi_{\Omega_S}\} \Leftarrow$ packet_forwarding($\Omega_S$, $T$)
7:     for all $i \in \Psi_S$ do
8:         $d_i \Leftarrow \min_{j \in \Psi_{\Omega_S}}\{d'_j + \omega(R_i^{\Omega_S}, R_j^{\Omega_S})\}$
9: return $\{d_i, \forall i \in \Psi_S\}$
```

Algorithm 1 shows the main logic behind the packet forwarding decisions. The algorithm takes two arguments as input: S is the source node (in the beginning it's the host or router that conducts the packet forwarding), and T is the UID of the packet destination. The algorithm is recursive and eventually returns a set of distances to the destination, one for each of S's interfaces: $\{d_i, \forall i \in \Psi_S\}$. A packet shall be forwarded out from the interface with the least distance to destination: $\arg\min_{i \in \Psi_S} d_i, \forall i \in \Psi_S$. The algorithm first determines whether the destination is within S's owning or parent routing sphere Ω_S (line 1). If so, the algorithm finds (within Ω_S) the node or the child routing sphere, denoted as Z, that contains the destination (line 2). For each of S's interfaces, $i \in \Psi_S$, the algorithm calculates the minimal distance d_i to all interfaces of Z (lines 3 and 4). If the destination is outside of Ω_S, the algorithm makes a recursive call which returns the set of distances to the destination from Ω_S's interfaces: $\{d'_j, \forall j \in \Psi_{\Omega_S}\}$ (line 6). Then, for each of S's interfaces, $i \in \Psi_S$, the algorithm calculates the minimal distance d_i to all edge interfaces of Ω_S (lines 7 and 8).

The cost of running the algorithm depends on the size of the forwarding tables and the location of the destination in relation to the source. It is common for a network session to send a large number of packets from the same source to the same destination. As long as the forwarding path does not change, those packets should be forwarded along the same path. In order to improve the performance of packet forwarding, we cache the paths within the routing spheres using NIx-vectors [RAZ01]. A NIx-vector is a compact representation using a sequence of network interface indices to indicate a routing path from the source to the destination (similar to source routing). During simulation, the NIx-vector is included as part of the packet header and at each hop that the packet traverses, the NIx-vector is shifted left by the number of bits needed to represent all network interfaces at the hop to retrieve the index of outgoing interface to forward the packet.

We actually provide two caches. We use the first cache to store the target local interface. If the destination, T, is within the owning/parent routing sphere, Ω_S, we insert a map entry (after line 4) from (S,T) to the target local interface's RID, $R_j^{\Omega_S}$, where $j = \arg\min_{j \in \Psi_Z}\{\omega(R_i^{\Omega_S}, R_j^{\Omega_S}), \forall i \in \Psi_S\}$. If the destination is outside of Ω_S, we add an entry from (S,T) to $R_j^{\Omega_S}$ (after line 8), where $j = \arg\min_{j \in \Psi_{\Omega_S}}\{d_j' + \omega(R_i^{\Omega_S}, R_j^{\Omega_S}), \forall i \in \Psi_S\}$. The second cache stores a map from the source node, S, and the target local interface's RID, $R_j^{\Omega_S}$, to the NIx-vector. After the first packet populates these caches, subsequent packets can use the same NIx-vector from the cache.

4.2.5 Discussions

In this section we discuss some subtle and yet important issues regarding the algorithm design.

Child routing spheres are treated as super-nodes in the parent routing sphere. The difference between a super-node and a regular node that represents a host or router is that the distance between the edge interfaces of a super-node can be greater than zero, since they may belong to different hosts or routers within the child routing sphere. For shortest-path calculations, we must consider the cost for traversing a node in the graph. Consequently, we need to calculate the shortest paths between interfaces rather than nodes. Also, we need to calculate the shortest paths from bottom up; that is, we settle the forwarding tables for the child routing spheres before their parents, since the parent routing sphere may require the distances between the edge interfaces of a child routing sphere. Note that for BGP policy-based routing, the distances between the edge interfaces of a child routing sphere are not needed in calculating the shortest valley-free paths, because BGP path is only a list of ASes (not routers).

The algorithm for packet forwarding returns the distance to destination node via all interfaces of the source node. In this way we can make a recursive procedure: if the destination node is outside of the routing sphere containing the source node, we walk up the routing sphere hierarchy until we find the routing sphere that contains the destination node. However, we do not walk down the hierarchy for the destination node. That is, once we find the routing sphere that contains the destination node, we do not make a recursive call into the destination routing sphere (i.e., node Z at line 2 of Algorithm 1), even if the destination node is not an immediate child of the destination routing sphere. Our decision is based on three reasons. First, this method can bring a significant cost reduction. Not walking down the routing sphere hierarchy can cut the average number of spheres that need to be traversed by half. Second, for parallel and distributed simulation, walking down the destination routing sphere implies that each simulation instance must maintain the forwarding tables of all routing spheres. This is certainly not a scalable solution. Third, global shortest path is not realistic for routing. It is common for the top-level sphere of a network model to adopt an inter-domain policy-based routing and treat the second-level spheres as autonomous systems. In this case, one does not need to walk up to the top-level sphere. This implies "hot potato routing": between ASes with multiple peering locations, it is common practice (based on the normal peering agreements) for an AS to pass traffic off to another AS via the nearest peering location. In this case, the distances from the edge interfaces to the destination node are all treated as equal; no recursion into the top-level sphere is necessary.

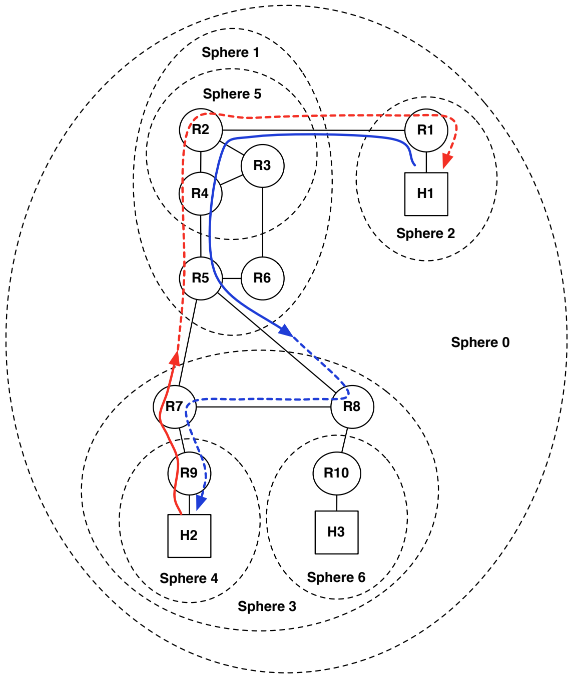


Figure 4.8: An example of asymmetric routes.

There are two important consequences of not walking down the hierarchy of routing spheres while selecting the specific routes for packets. First, paths can be inflated: Once we reach the the topmost routing sphere, we select an entry point to descend into the child routing spheres without actually descending into them. We make this choice by choosing the entry point which has the shortest path to the source, however, this entry point may not have the shortest path to the destination. Asymmetric routes are the second consequence. In figure 4.8, we have a scenario where the route between node H_1 and H_2 is inflated by one hop (compared to global shortest path) and is different than the route from H_2 to H_1. In this case, the routes between H_1 and H_2 are asymmetric. As this example shows, asymmetric routes are also caused by our algorithm not descending into child routing spheres before selecting a route. On the way from H_1 to H_2 (the blue line) we chose to leave $Sphere_1$ using the left interface of R_5, which causes the route to be inflated as we enter Sphere_3 through R_8. The right interface is chosen arbitrarily because the path from either interface of R_5 are equidistant to H_1. On the reverse path from H_2 to H_1 (the red line), we select R_7's interface to exit Sphere_3, which yields the shortest path from H_2 to H_1. A previous measurement based study of routes within the global Internet found that asymmetric routes are somewhat common [Pax96]. The study found that asymmetric routes account for nearly 20% of the observed routes. In many cases, the asymmetries could be attributed to so called "hot potato routing", which further bolsters our decision to not descend into child spheres. Given this, we feel that any potential negative effects of slightly inflated routes or asymmetric routes are easily outweighed by the performance improvement to online route calculations and increased scalability of spherical routing.

4.3 Sharing Instance State & Model Structure

As previously described in section 3.3, we use network scripting to generate *model nodes* using annotations embedded in the original network models. Model replication is primarily concerned with state variables which are annotated as *shared*. There are three important details about how the model node classes are generated:

1. All states which are annotated as sharable are grouped into an internal class called *shared*, and all other variables are placed into a separate class called *internal* (see figure 3.3).
2. The generated model node contains two member variables, pointers to their shared and internal state classes, and functions to implement the network model's functionality. The result is that all model nodes are the same size, irrespective of type.
3. The model nodes are generated using a class hierarchy. In the end there are three class hierarchies. One for the model nodes, one for the shared state classes, and one for the internal state structures.

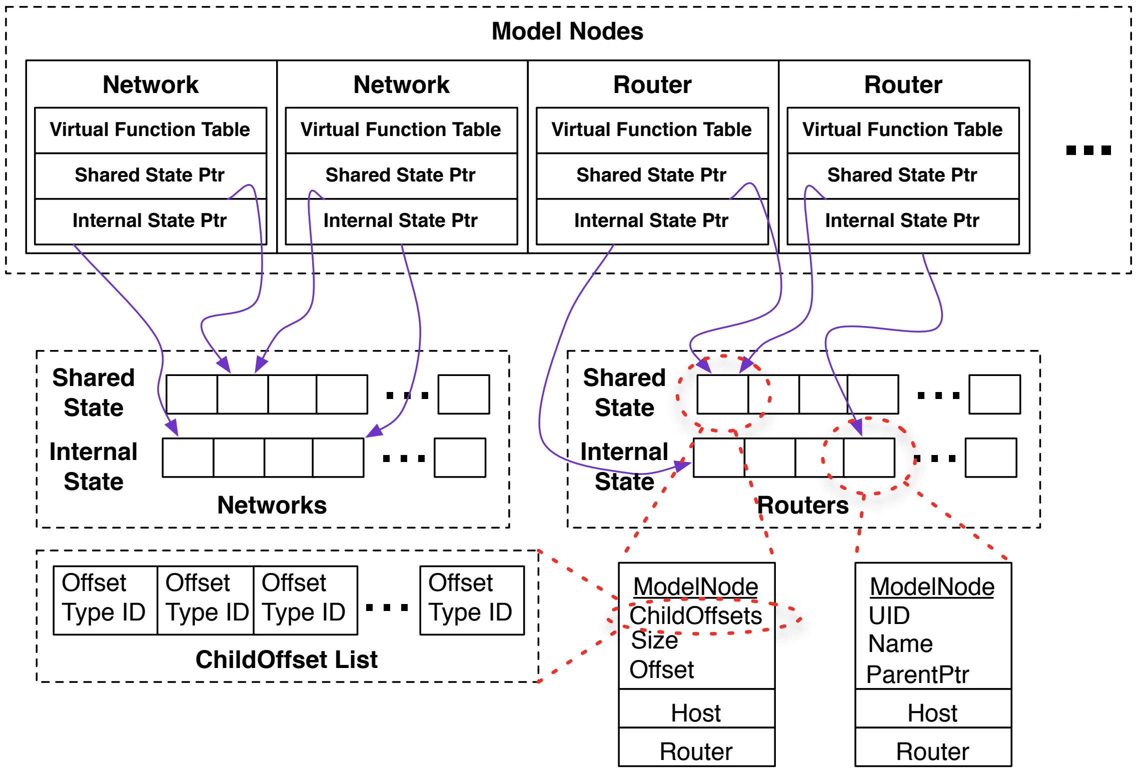


Figure 4.9: Internal memory layout of the execution model.

In section 4.3.1 we present details on how the simulator's memory layout exploits structural replications, and in section 4.3.2 we discuss the limits of how much memory can be saved by sharing model structure and sharable instance state.

4.3.1 Design

Figure 4.9 shows the basic memory layout of model nodes within the network simulator. All of the model nodes are stored in an array – we can do this since all nodes types are the same size and extend a common base class). Nodes are organized within the array based on their UID (see section 4.2.2 for details on how UIDs are assigned. For each type of model node, we also create an array of shared and internal states – we must make different arrays for each type because the size of the shared and internal state classes depend on the model node's type. During initialization, we assign each model node an instance of it's associated internal state. Replicated model nodes are given a pointer to the same shared state structures.

In addition to pointers to their shared and internal state classes, each node must also have a mechanism to retrieve its parent and child nodes. This could have been avoided by assigning UIDs in a k-nary fashion (UIDs are not). However, in order to assign UIDs in a k-nary fashion, we would have had to assign K to be the largest number of children found in the model – which could be large. Assuming we could cap the number of children at sixty four; we would exhaust an unsigned sixty four bit integer after just eleven levels in the model. As seen in figure 4.5, eleven levels is not much considering each host and router will also expand into two, three, or more levels, depending on how complex the model is. Clearly, a k-nary approach would not support large network models.

For ease and simplicity, each model node stores a pointer to their parent in their internal state class. However, in order to share the structural information we store indirect references to the child nodes using our `ChildOffsetList` structure which can be shared between replicated nodes. Using the offset from the `ChildOffsetList` and the `UID`, `size`, and `offset` of the node, we are able to calculate the UID of each child and use that as an index into our model node array using equations 4.1 and 4.2. The `Type ID` is used to provide type-safe access and type-specific child iterators for use by network model developers to implement the model's functionality.

4.3.2 Limits

$$size(\mathsf{E},\mathsf{x}) = \begin{cases} \mathsf{S_x} + \mathsf{I_x} + \mathsf{O} + \mathsf{C} & \text{if } \mathsf{E} \text{ is not replicated,} \\ \mathsf{I_x} + \mathsf{O} & \text{if } \mathsf{E} \text{ is replicated,} \end{cases} \tag{4.5}$$

The amount of memory used by a model instance (E) of type x can be calculated using equation 4.5, where:

- $\mathsf{S_x}$ is the size of the shared state class for model node of type x, not including the memory cost of the child index array.
- $\mathsf{I_x}$ is the size of the internal state class for model node of type x.
- O is the overhead of maintaining the object, the pointers to the internal and shared state classes, and the virtual function table.
- C is the cost of maintaining the child-offset list in order to access the children. For simplicity, we estimate the cost to be $e^{\frac{\log \mathsf{N}}{\log \mathsf{H}}}$ where N is the number of nodes in the model and H is maximum depth of the model.

Given the number of entities of each type ($\mathsf{N_x}$) and the number of each type which are replicated ($\mathsf{N_x^{rep}}$), one can estimate the amount of memory ($\mathsf{M_{total}}$) needed to store the entire network model using the following (where T is the set of all model node types):

$$\mathsf{M_{total}} = \sum_{\mathsf{x} \in \mathsf{T}} \left[\mathsf{N_x}\,(\mathsf{I_x} + \mathsf{O}) + (\mathsf{N_x} - \mathsf{N_x^{rep}})\,(\mathsf{S_x} + \mathsf{C})\right] \tag{4.6}$$

From equation 4.6 it follows that the maximum amount of savings (i.e. $\sum_{\mathsf{x}\in\mathsf{T}} \mathsf{N_x} \approx \sum_{\mathsf{x}\in\mathsf{T}} \mathsf{N_x^{rep}}$) that can be obtained using replications can be calculated using:

$$\mathsf{Percent_{saved}} \approx \sum_{\mathsf{x}\in\mathsf{T}} \frac{N_x}{\left(1 - \frac{I_x + O}{\mathsf{S_x} + \mathsf{C}}\right)|T|} \tag{4.7}$$

Equation 4.7 shows that the cost of storing *model structure* (i.e. C) can be nearly eliminated, and the amount of space needed to store *instance state* depends on two factors: the proportion of sharable and internal state for each entity type, and the proportion of entity types within a specific network model. This is expected. To maintain realism, entities need the ability to act independently. This can only be done if the entity is allowed to keep some states that are specific to itself (i.e. internal state).

4.4 Evaluation

We evaluate spherical routing in section 4.4.1. We then evaluate spherical routing's scalability in section 4.4.2 and validate our BGP algorithm in section 4.4.3. We then evaluate how well we exploit structural replications to reduce instance state and model structure in section 4.4.4.

4.4.1 Reducing Routing State

In this section we present several experiments to evaluate spherical routing in terms of memory and run time.

Memory Consumption

Spherical routing can reduce memory consumption by allowing users to compose network models using structural replications. Our first experiment examines the memory reduction using models generated using our modified BRITE with different replication factors, α, as described in section 4.1.1. Table 4.2 shows the memory consumption of the forwarding tables, both uncompressed and compressed for range queries using R-trees (in columns 2 and 4, respectively). As expected, the ratio of memory reduction (R%) is almost proportional to $(1-\alpha)$ in both cases. The compression ratio (CR) varies depending on the network topology, and in this case stays about 60%.

Spherical routing also allows one to adjust the trade-off between accuracy and memory consumption by selecting routing spheres differently. Our second experiment examines this trade-off using a simple campus network model [Nic]. The model has a backbone connecting four subnetworks, two of which contain smaller subnetworks, as seen in figure 4.10. In total, the network has 508 hosts and 30 routers in 17 subnetworks. We study the effect of different routing sphere configurations by placing different routing spheres at

α	**Uncomp.**	**R%**	**Comp.**	**R%**	**CR**
0.0	7.05 MB	100%	4.18 MB	100%	59.3%
0.2	5.75 MB	81.6%	3.48 MB	83.3%	60.5%
0.4	4.49 MB	63.7%	2.75 MB	65.8%	61.4%
0.6	2.87 MB	40.7%	1.72 MB	41.2%	60.0%
0.8	1.29 MB	18.3%	0.70 MB	16.8%	54.4%

Table 4.2: Forwarding Table Sizes for Different Replication Factors.

	Uncomp.	**Comp.**	**CR**	**Path Lengths**
ALG1	16 KB	-	-	8.15 ± 2.99
SP1	9.5 MB	857 KB	9.0 %	7.39 ± 2.73
SP4	4.6 MB	475 KB	10.2 %	7.71 ± 3.06
SP17	72 KB	28 KB	38.9 %	7.66 ± 2.98

Table 4.3: Forwarding Table Sizes (in KB) and Path Lengths.

different subnetworks. As a baseline, we set up one routing sphere for the entire campus network and use either algorithmic routing (ALG1) or shortest-paths (SP1) as the routing strategy. We also include a setup with 4 routing spheres for the subnetworks connected by the backbone (SP4) and another with 17 routing spheres (SP17). Table 4.3 shows the forwarding table sizes and the path lengths for different sphere placements. With 17 routing spheres, the compressed forwarding tables take only 28 KB, a reduction over two orders of magnitude compared to the global shortest paths (9.5 MB). The memory consumption for SP17 is comparable to that of algorithmic routing (16 KB).

To investigate the accuracy of different routing strategies, we use a simple metric to show how the routes are distributed over the network. We count the number of unique routes that cross each link in the model and then plot the distribution by sorting the links by the number of routes. Figure 4.11 clearly shows that algorithmic routing concentrates the routes to fewer links than does the global shortest-path routing (about 80% of the links being removed by algorithmic routing since they have no routing paths through them).

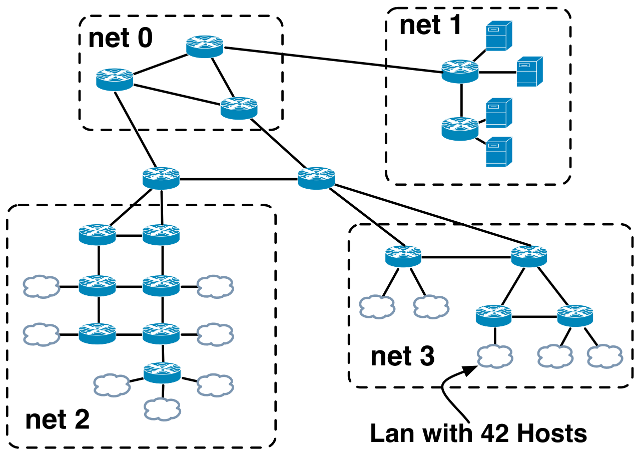


Figure 4.10: Campus model.

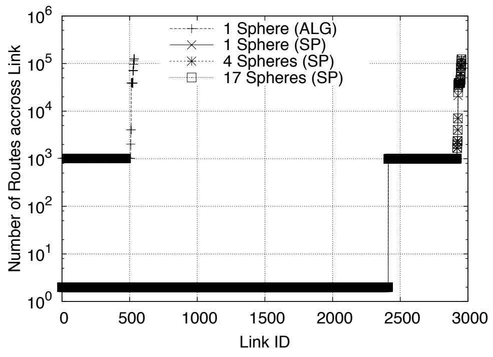


Figure 4.11: Distribution of routes.

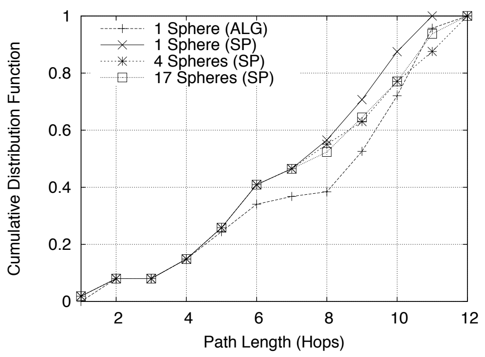


Figure 4.12: Path length inflation.

Subdivision of routing spheres, on the other hand, yields a nearly identical distribution as the global shortest paths. We also compute the distribution of path lengths for the different strategies. Figure 4.12 shows that algorithmic routing produces a very different distribution, while using different number of routing spheres only slightly inflates the path lengths (at most by one hop for longer paths). This is caused by the algorithm's cost-saving measure of not walking down the destination routing sphere (as discussed in section 4.2.5).

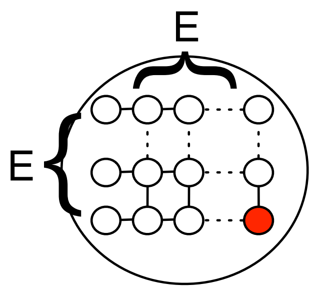


Figure 4.13: Model used to evaluate cost vs. sphere size.

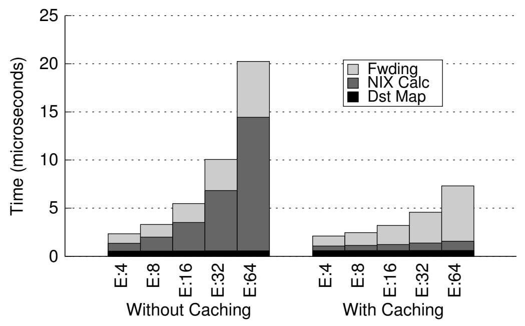


Figure 4.14: Cost vs. sphere size.

Run Time

Our next experiment examines the packet forwarding time in spherical routing. The time associated with spherical routing consists of: (1) the time to map a destination node in a remote sphere to a node local to the current sphere, (2) the time to create the NIx-vector, and (3) the time to forward the packet at each network interface (either processing the Nix-vector or looking up the forwarding table). As such, the total cost of forwarding a packet from the source to the destination is determined by four major factors. The first factor is the path length within each sphere: a longer path will cost more to build the NIx-vector and process it during packet forwarding. The second factor is the depth of the routing sphere in the hierarchy: spheres at lower levels will spend more time to make an up call during remote destination mapping. The third factor is the number of spheres that a packet needs to traverse before reaching its destination: more transit spheres will add to the cost of mapping remote destinations to local nodes. The last factor is the number of edge interfaces at a routing sphere: the algorithm needs to select the shortest among the shortest paths from each edge interface at the current routing sphere to each edge interface at the destination routing sphere.

To estimate the overhead we construct a network model where we can easily configure the routing path length. We created the network model in figure 4.13, which contains a single routing sphere consisting of hosts placed on a rectangular grid. We designate a host in one corner of the grid (the red node) to send 100 pings to each host in the grid (the white nodes). We vary the number of edge interfaces (E), and the path length, by changing the dimension of the rectangular grid. We run the experiment on a Linux workstation with a 2.3 GHz Intel Core2 Duo processor and 2 GB of RAM. We preform this experiment with and without caching and present the median of 50 trails. When caching is enabled, we make sure that the cache is large enough to store all entries without eviction (at least 64 entries for this experiment). Figure 4.14 shows a breakdown of the average cost (per sphere) of sending a packet from source to destination. The cost of destination mapping remains constant with and without caching, and with varying number of edge interfaces. This is simply because we have just one routing sphere here. The cost of forwarding a packet includes both the time for creating the NIx-vector and the time for extracting the next hop from the NIx-vector at each hop in the path. The cost increases as we increase the path length; the cost is similar with and without caching. The cost for calculating the NIx-vector is reduced drastically when caching is enabled. This is expected. Without caching, the algorithm has to determine the shortest path for each packet at the source; the cost is proportional to the number of forwarding table lookups, or the path length, which is determined by the dimension of the rectangular grid. In case of caching, the cost stays very low and increases only slightly for bigger cache as we increase the dimension of the rectangular grid (more destinations).

We construct two additional network models where we can control the height of the hierarchy of the routing spheres and the number of transit spheres. In one experiment we vary the height of the hierarchy (H) by arranging the rectangular grids vertically, each being a routing sphere. The test model is shown in figure 4.15. We designate a host at the bottom routing sphere (the red host) to send 100 pings to each host at the top routing sphere (the green hosts). The results are shown in figure 4.16. When caching is disabled,

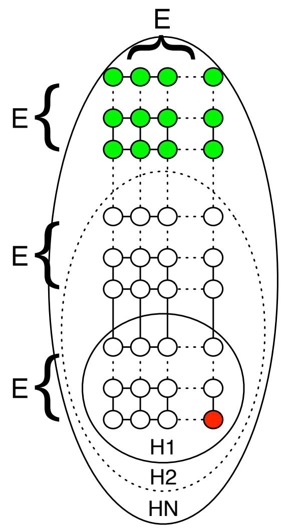


Figure 4.15: Model used to evaluate cost vs. sphere depth.

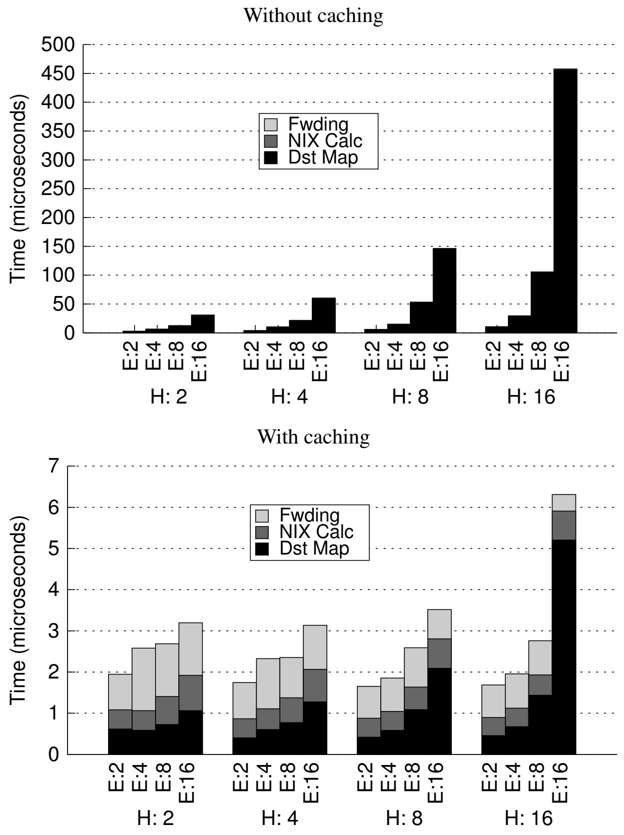


Figure 4.16: Cost vs. sphere depth.

the total cost is dominated by the cost of destination mapping. The cost is quadratic in the number of edge interfaces because the algorithm needs to calculate the shortest among the shortest paths from every edge interface at the source routing sphere to every edge interface at the destination routing sphere. The cost also increases linearly with H since the calculation needs to be carried out at each routing sphere in the hierarchy. Caching is most effective in this case. The overall cost reduces significantly by almost two orders of magnitude. As we increase H, the cost of destination mapping increases due to the increasing cache lookup cost (which is slightly super-linear to E) and the increasing number of cache lookups (which is related to H).

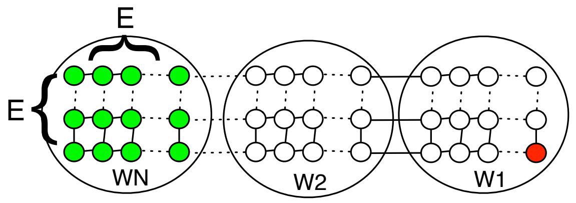


Figure 4.17: Model used to evaluate cost vs. transit spheres.

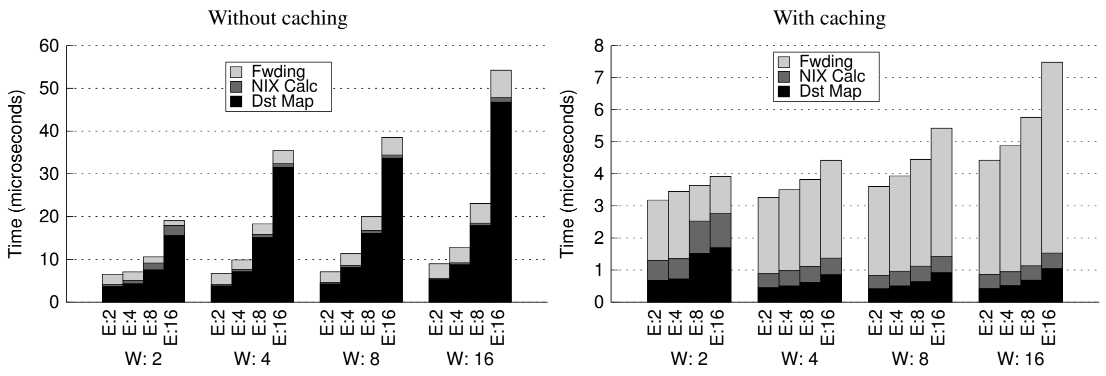


Figure 4.18: Cost vs. transit spheres.

In another experiment, we vary the number of transit spheres (W) by placing the rectangular grids horizontally. Again, each rectangular gird is a routing sphere, as shown in figure 4.17. We designate a host at the left-most routing sphere (the red host) to send 100 pings to each host at the right-most routing sphere (the green hosts). The results are shown in figure 4.18. When caching is disabled, the total cost is again dominated by the cost of destination mapping, which is insensitive to the number of transit spheres, W. Caching in this case can reduce the forwarding time by an order of magnitude. The cost of destination mapping increases due to the increasing cache lookup time as we increase the number of destination hosts (which is related to E). For both experiments, with caching, the average cost for forwarding a packet at each hop (for an average path length of 32.8 hops) is approximately 1.44 μs.

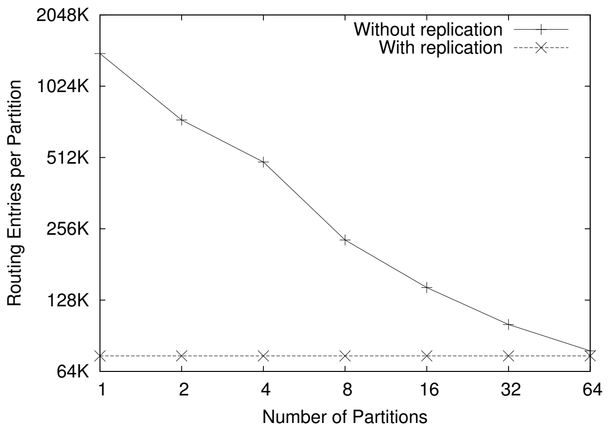


Figure 4.19: Scalability of spherical routing.

4.4.2 Large-Scale Model

We extend one of the tier-1 ISP topologies provided by Rocketfuel [SMWA04] to build a large network model to show the overall effectiveness of spherical routing. We choose the AT&T backbone network, which contains 640 routers and 1,382 links. We use the METIS [Geo] graph partitioner to partition the backbone network into 16 clusters similar in size. Each cluster is assigned a routing sphere. We then attach 1,920 campus networks (each having 17 routing spheres) evenly distributed among the clusters, resulting in a network model with a total of 1,033,600 hosts and routers in 328,337 routing spheres organized in 6 levels. For spherical routing, the total memory consumed by the forwarding tables measures only about 7.4 MB. For global and hierarchical shortest-path routing, the memory consumption can be derived from the number of all possible source-destination pairs and the size of each forwarding table entry. The hierarchical shortest-path routing calculates the forwarding tables independently for the ISP backbone and the campus networks. The global and hierarchical shortest-path routing strategies would need approximately 12 TB and 6.2 GB, respectively. Even with the 9% compression (from the previous experiments), the forwarding tables could still dominate the memory consumption.

We then partitioned the network model (using METIS) to execute on a different number of compute nodes (from 1 to 64) and measured the number of routing entries needed at each compute node (with compression). Figure 4.19 shows the number of route entries per partition with and without using replication. When replication is enabled, each partition contains only one copy of the campus' routing table in addition to the subsection of the backbone network it is responsible for. As a result, the number of entries per partition is nearly constant. If we disable replication, each campus keeps its own forwarding table. Since the campuses are divided evenly among the compute nodes, the total size of the forward tables decreases as we increase the number of partitions. This experiment shows that spherical routing can spread the memory burden of maintaining routing state in distributed simulations in a scalable fashion whether or not routing spheres are replicated.

We randomly select 500 pairs of hosts such that each pair had hosts from different routing spheres. We then direct one host from each pair to send 100 pings to the other host and measure the time taken to forward the pings across all the routing spheres. From 50 trials we obtain the average time to forward a packet at each hop to be 91.8 μs without caching and 2.4 μs with caching (the cache size being 1024 entries). This shows spherical routing can yield good performance on large realistic topologies.

4.4.3 BGP validation

We provided a simplified BGP routing algorithm in section 4.2.3, which determines the forwarding paths based on the policy relationships between autonomous systems (ASes). Network models that treat BGP routing as if it were plain shortest path routing by not considering AS relationships will experience inaccuracies such as shorter paths, larger path diversity and lower traffic load than in reality [DKVR07]. Here, we show that by using our BGP routing algorithm, these inaccuracies can be avoided. For simplicity, we borrow the AS topology used by Dimitropoulos et al [DKVR07] to validate our algorithm.

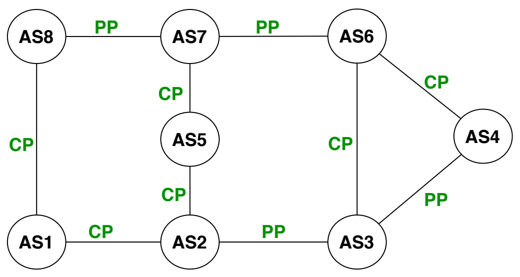


Figure 4.20: An AS topology annotated with AS relationships.

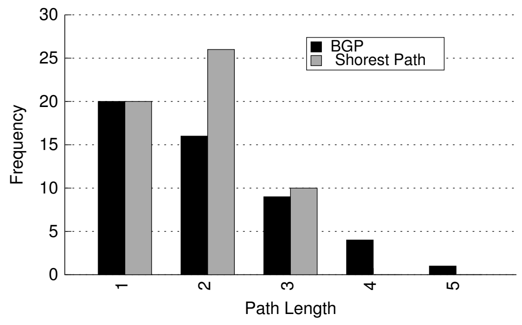


Figure 4.21: The histogram of all paths with AS relationships enabled or disabled.

The topology, shown in figure 4.20, is a portion of an AS topology that is annotated with AS relationships that were inferred using real-word measurements [DKH+05].

We verified that all of the routes produced using our simplified BGP algorithm were valley-free. We then compared our BGP routes to routes produced using standard shortest path. A histogram of path lengths produced by the two routing strategies is shown in figure 4.21. We can see that considering the AS relationships will produce routes that are longer than plain shortest path. In this case, a shortest path policy produces paths with an average of 1.75 hops while a BGP routing policy produces paths with an average of 2.0 hops. More specifically, we found that 23% of the paths produced by BGP are longer than those produced by shortest path.

An even more important observation is that any node which is connected in a network topology will also be able to reach that node when using a shortest path routing policy. With BGP, however, connectivity does not imply reachability. In this small topology, 9% of all pairs had no valid paths between them. For example, there are no valid paths from *AS*6 to *AS*8 when considering AS relationships, even though they are clearly connected in the topology.

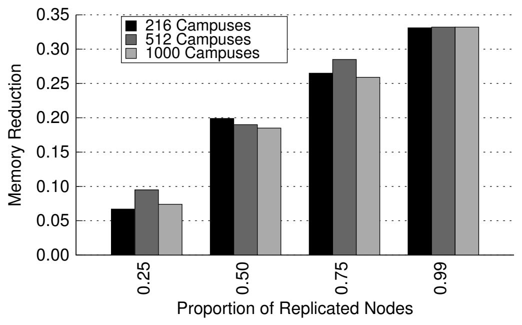


Figure 4.22: Memory reduction from replication.

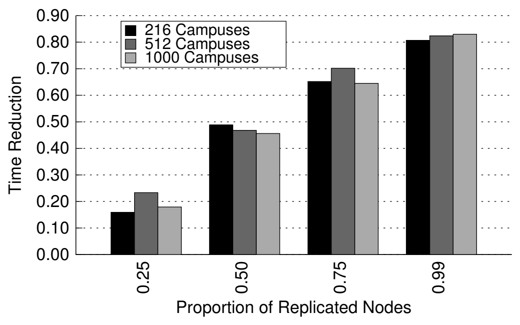


Figure 4.23: Preprocessing time reduction from replication.

4.4.4 Reducing Instance State

To evaluate the effectiveness of model replication, we conducted an experiment using a synthetic network. We varied the proportion of replicated sub-structures in the model and measured the memory needed to instantiate the execution model. At the base level, we used the campus network, which consists of 508 hosts and 30 routers in 17 nested subnetworks [Nic]. We examined three networks with 216, 512, and 1,000 campuses. We varied the proportion of replicated campuses from 25% to 99%. The results are shown in figure 4.22.

The three network models performed consistently. The memory savings due to model replication reached as much as 33%. A detailed analysis of the model's memory usage reveals that the amount of sharable state stays around 38%. The 5% difference is due to the overhead of maintaining the replications, which include the memory for the virtual function tables, pointers to the shared state, and some auxiliary data structures to allow the model nodes to be shared among replications.

Model replication can save memory for both interactive and execution models (see chapter 3 for details on interactive and execution models). For the interactive model, it can also improve the preprocessing time. This can be attributed to three factors. First, model replication reduces memory for storing the interactive model and therefore demands less I/O when the interactive model is persisted to the database (see section 3.5 for details on out-of-core processing). Second, putting the sharable variables together improves locality, which can be exploited by an intelligent caching technique (again, section 3.5). Finally, preprocessing is needed by the interactive model to compile and send the information about the network configuration and the runtime environment to the execution model. The preprocessing time is reduced for replicated models because it can reuse the results from replicated model nodes.

We conducted another experiment using the same network models and measured the reduction in the preprocessing time of the interactive model. Figure 4.23 shows the results. We observe significant reduction in the preprocessing time, as much as 83%, with 99% replications.

4.5 Conclusion

Model replication addresses the problem of the large memory requirement for executing large-scale network models. The method takes advantage of two important observations. One observation is that, by partitioning the network model into smaller subnetworks on which one can conduct localized packet forwarding, the memory space for maintaining the forwarding tables can be reduced significantly. The other observation is that large-scale network models can be built using sharable model fragments, in which case we can significantly reduce the memory consumption by allowing structural replications and sharing of the forwarding tables and states.

Spherical routing provides the flexibility for modelers to construct network models judiciously using structural replications and subdivision of routing spheres to balance between model accuracy and space complexity. It also allows different routing spheres to adopt their own static routing strategies. We implemented three routing strategies: shortest-path routing, simplified BGP policy-based routing, and algorithmic routing (which is based on shortest-path trees). Spherical routing is also amenable to parallel and distributed simulation, in which case most routing spheres associated with the network partitions can be assigned to different processors for parallel processing; only the routing spheres at higher levels in the hierarchy need to be shared, and spherical routing can safely replicate them across the processors.

CHAPTER 5

REAL-TIME SIMULATION & EMULATION OF LARGE-SCALE MODELS

Building an emulation infrastructure that is able to cope with large-scale network models presents a number of challenges. First, real applications require far more computational resources than purely simulated applications. Second, transporting real packets within the virtual network requires the system to perform a significant amount of I/O. If the system cannot keep up with the CPU or I/O demands of an application, the system will exhibit unrealistic artifacts causing the realism of the experiment to suffer. Our emulation framework aims to address the following issues:

- *Scalability*: The framework should provide scalable interaction and effortless integration with real applications. A real-time simulation infrastructure should be provided for the network simulator to run on large parallel machines. This enables dynamic interaction with a large number of real applications.

- *Flexibility:* Applications that require special hardware or have strict resource constraints need the ability to run on dedicated hardware which is remote from the simulator.

- *Realism:* Application behaviors that are observed within the emulation framework should accurately reflect behaviors that would be seen in native environments.

The remainder of the chapter is organized as follows. Section 5.1 discusses related work. The current emulation framework and how it addresses the needs for scalability, flexibility, and realism are outlined in sections 5.2, 5.3, 5.4, and 5.5. Section 5.6 presents an experimental evaluation of our emulation infrastructure. The chapter is then concluded in section 5.7.

5.1 Related Work

Network testbeds are commonly used for prototyping, evaluating, and analyzing new network designs and services. *Physical testbeds*, such as WAIL [BL03] and PlanetLab [PACR02], provide an iconic version of the network for experimental studies, sometimes even with live traffic. They provide a realistic testing environment for network applications, but with limited user control. Further, being shared facilities, they are constantly overloaded due to heavy use, which can severely affect their availability and accuracy [SPBP06].

An *emulation testbed* can be built on a variety of computing platforms, including dedicated compute clusters, such as ModelNet [VYW$^+$02] and EmuLab [WLS$^+$02], distributed platforms (such as VINI [BFH$^+$06]), and special programmable devices, such as ONL [DKP$^+$06] and ORBIT [RSO$^+$05].

While most emulation testbeds provide basic traffic "shaping" capabilities, *simulation testbeds* can generally achieve better flexibility and controllability. For example, ns-2 [NS-a] features a rich collection of network algorithms and protocols that can be selected to model a myriad of network environments, wired or wireless. Simulation can also be scaled up to handle large-scale networks through parallelization, e.g., SSFNet [Ren], GTNeTS [Ril03] and ROSSNet [YBB$^+$03]. It would be otherwise difficult and costly to build a large-scale physical or emulation testbed.

Real-time simulation is the technique of running network simulation in real time, and thus can interact with real implementations of network applications [Liu08]. Most existing real-time simulators, e.g., [Fal99, BSU00, LC04, ZJTB04], are based on existing network simulators extended with emulation capabilities. PRIMEX is a discrete-event simulator designed to run on parallel and distributed platforms and handle large-scale network models. This work builds up the Parallel Real-Time Immersive Modeling Environment (PRIMEX) for real-time simulation. PRIMEX provides several important functions:

- PRIMEX provides models for 14 TCP variants (mostly ported from the Linux TCP implementation). They have been validated carefully through extensive simulation and emulation studies [ELL09].
- PRIMEX uses multi-scale modeling for large-scale simulation. PRIMEX implements a fluid traffic model, which has been integrated with simulated and emulated network traffic. The hybrid traffic model can achieve a speedup of more than three orders of magnitude over the traditional packet-oriented simulation [LL08].

5.2 Emulation Framework

The emulation infrastructure needs to support high-throughput low-latency data communication between the emulated hosts and the simulation instances.

There are two kinds of emulated hosts: collocated and remote. *Collocated emulated hosts* run as virtual machines on the same compute node as the simulator instance that simulates the subnetwork containing the corresponding virtual hosts. In our approach we assume the compute nodes run OpenVZ. OpenVZ can support a large number of collocated emulated hosts and therefore allows for network emulation at larger scales.

Remote emulated hosts are either physical compute nodes collocated with the real-time simulator (not VMs), or machines that are not part of the compute cluster and have the potential for different geographic locations. Remote emulated hosts can run applications which cannot otherwise be run on a virtual machine due to either stringent resource requirements or system compatibility concerns. For example, an emulated host may require a special operating system (or version), or need specialized hardware that is not available on the compute node.

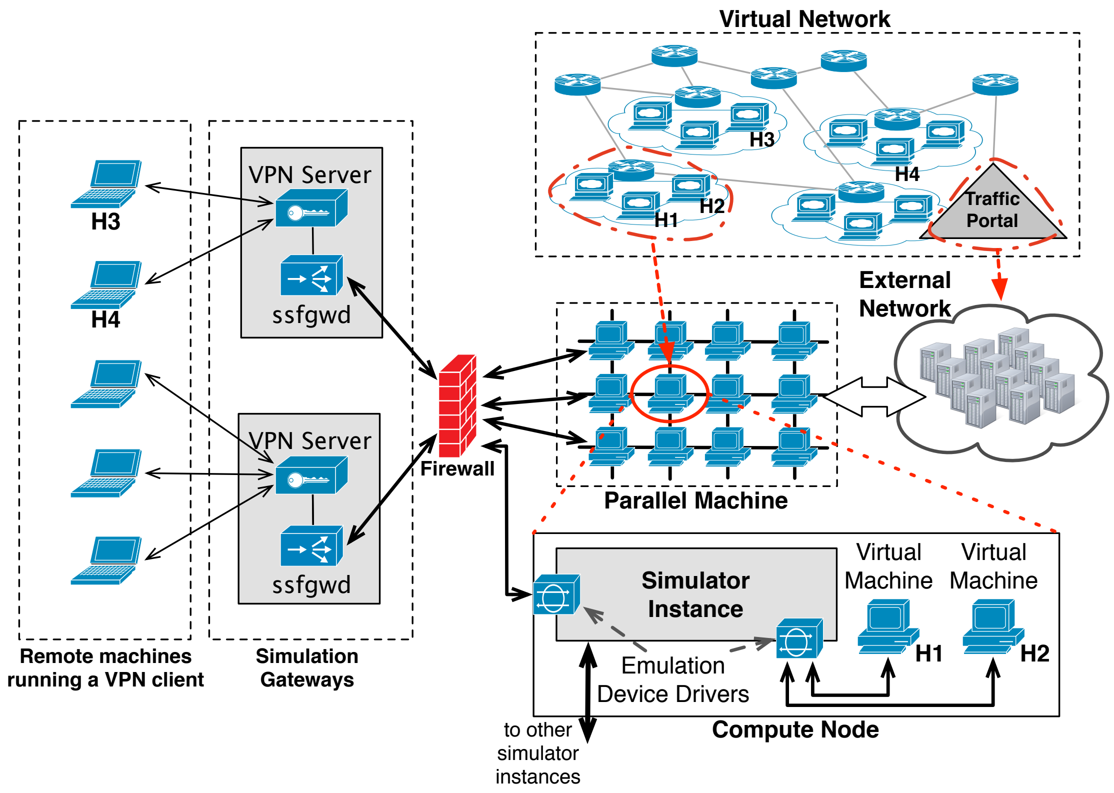


Figure 5.1: A virtual network is run on parallel machines and emulated hosts are run on remote machines and collocated with simulator instances.

In addition to emulated hosts, our architecture provides *traffic portals* to exchange traffic with real networks. Traffic portals allow hosts and routers outside the virtual network to interact with simulated and emulated traffic within the virtual network.

Figure 5.1 contains a high level view of the emulation infrastructure which consists of three major components: the virtual network interfaces, the emulation device drivers, and the interconnection mechanism.

The *virtual network interfaces (VNICs)* are installed at the emulated hosts, and treated as regular network devices, through which applications can send or receive network packets. Packets sent to a VNIC are transported to the simulator instance that handles the corresponding virtual host. The simulator subsequently simulates the packets traversing the virtual network as if they were originated from the corresponding network interface of the virtual host. When packets arrive at a network interface of an emulated virtual host in simulation, they are sent to the corresponding VNIC at the emulated host, so that its applications can receive the packets.

The *emulation device drivers (EDDs)* are collocated with the simulator instances running on the compute nodes. They are software components used by the real-time simulator to import real network packets sent from VNICs at the emulated hosts or traffic portals. Conversely, EDDs also export simulated packets and send them to the emulated hosts or traffic portals. Our emulation infrastructure supports different types of EDDs for remote emulated hosts, collocated emulated hosts, and traffic portals. PRIMEX provides functions designed specifically for interactive simulations, which include handling real-time events and performing conversions between fully formed network packets and simulation events. Further details of emulation device drivers are presented in section 5.3.

The *interconnection mechanism* connects VNICs on emulated hosts and EDDs with the simulator instances. Its design depends on whether it is dealing with traffic portals, remote or collocated emulated hosts. For remote emulated hosts, we use an OpenVPN-based infrastructure where clients are run at the remote emulated hosts, each corresponding to a single VNIC. One or more compute nodes are designated to run modified OpenVPN servers, which forward IP packets to and from the EDDs collocated with the simulator instances. Detailed information about this interconnection mechanism can be found in section 5.3.1. The interconnection mechanisms for collocated emulated hosts and external networks are described in section 5.3.2 and section 5.3.3, respectively.

In section 5.4 we present relevant details of the design of the compute nodes which will host remote and collocated emulated hosts as well as traffic portals. In section 5.5 we present a *meta-controller* framework which automates the deployment and configuration of large emulation experiments.

5.3 The Real-Time Network Simulator and Emulation Device Drivers

A virtual network is partitioned among compute nodes to be simulated in parallel. Each compute node is composed of one or more parallel processors. Each processor is assigned a simulation process for event processing. Additionally, if the simulation process manages virtual nodes which are emulated, the process will spawn an EDD for each type of emulated host. The EDD(s) are responsible for both exporting and importing packets.

EDDs are composed of two essential functions: importing and exporting packets. EDDs come in two basic types: blocking and polling. Blocking drivers must use threads to execute the import and export functions because their underlying transport requires that the processes wait for I/O operations to complete. In other words, the underlying transport is synchronous. In contrast, polling devices do not require separate threads to execute the import and export functions. Instead, polling devices schedule themselves to "poll" for new packets. Exactly how often a polling devices reschedules itself is implementation specific. When exporting packets, polling devices use the context of the virtual node which requests the packet to be exported to perform the necessary I/O to export the packet.

The EDDs use established SSF functions designed specifically to support emulation, including both exporting simulation events and importing real network packets (see [LLN$^+$05] for more details). Each emulated virtual node in the simulation contains an emulation session that intercepts packets at the link layer (i.e. ethernet). When a virtual node's network interface receives a simulated packet, it checks whether it has an active emulation session. If it has an active emulation session, the session is queried to determine if the associated application runtime environment is available to accept packets. If so, the interface hands the simulated packet to the emulation session which will route the packet to the correct EDD. The EDD translates the packet into a fully formed IP packet and exports it to the application runtime environment. If the EDD is exporting the packet

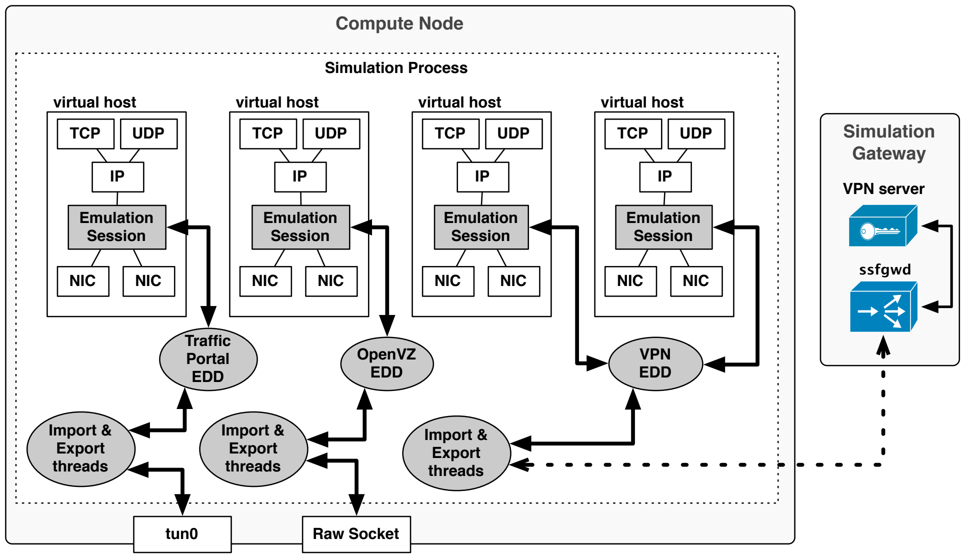


Figure 5.2: Connecting the Simulator with local and remote environments using different emulation device drivers (*EDD*s).

to a remote environment, it will use a simulation gateway to accomplish this. Otherwise, the driver will interact with the host OS to export and route the packet to the correct application. On the reverse path, when a EDD receives an IP packet, it translates the packet to a simulation event and presents it to its associated simulation process. That process forwards the event to the emulation session on the corresponding virtual interface, which pushes the packet down the simulated protocol stack. This procedure is illustrated in figure 5.2.

5.3.1 Remote Emulated Hosts

Emulation device drivers that connect to remote emulated hosts are composed of two main components: (1) the local import/export mechanisms and (2) the simulation gateway.

The simulation gateway resides between the network simulator and the client applications running on distributed machines. An OpenVPN server is set up at each simulation gateway managing incoming connections from the client machines. There can be multiple simulation gateways for balancing the traffic load, or for differentiated services. For example, each simulated gateway could offer a different quality of service guarantee for its connected clients. At each gateway, traffic to and from the client applications are handled by OpenVPN.

In addition to running the OpenVPN server that manages the OpenVPN clients, another process is created at the simulation gateway (`ssfgwd`) to handle traffic to and from the real-time network simulator. The `ssfgwd` daemon is a process responsible for shunting IP packets between the simulator's emulation device driver and the OpenVPN server. It maintains a separate TCP connection with the export and import threads of each EDD that has connected to the gateway. Packets to be inserted into the real-time simulator are sent from `ssfgwd` via a dedicated TCP connection to the import thread associated with the simulation process on the parallel machines. Packets exported from the real-time simulator are sent from the export thread via another TCP connection to `ssfgwd`. In either case, it is important to have the TCP connections initiated from the simulator, since the simulator is expected to run on high-performance hardware like supercomputers, which are normally situated behind firewalls and/or cannot be determined *a priori*.

When an EDD connects to a simulation gateway, it sends the IP addresses of all virtual hosts that it is responsible for. The `ssfgwd` process records this list of IP addresses and creates a mapping from each IP address to the corresponding TCP connection to the import thread that sends these addresses. Later, `ssfgwd` uses this mapping to forward traffic from the client machines to the correct EDD.

In order to support emulation of hosts with multiple network interfaces, the VPN server was modified to spawn the `ssfgwd` process directly and communicate with it using standard Unix pipes. An IP packet received by the VPN server from one of its clients is preceded with the client's IP address before it is sent to `ssfgwd` via the pipe. The IP address is used by `ssfgwd` to deliver the packet to the corresponding simulation process.

A mapping from the IP address to the simulation process that contains the client's virtual host is established immediately after the TCP connection is made by the import thread. Once the TCP connection is located, the packet is sent via that connection to the designated EDD. The packet is subsequently inserted into the protocol stack of the virtual node that carries the same IP address of the client machine that initiates this packet. In such a way, the packet appears as if it were generated directly by the virtual host. Similarly, when an IP packet emanating from a virtual host is sent to the simulation gateway through the TCP connection established by the export thread, the packet is preceded with an IP address identifying the virtual interface that exports the packet. The `ssfgwd` process sends both the IP address and the packet via a pipe to the OpenVPN server, which forwards the packet to the corresponding OpenVPN client with the IP address that preceded the packet. In such a way, the packet arrives at the client machine as if it received the packet directly from a physical network.

The VPN Connections

Our scheme supports the use of multiple simulation gateways to alleviate the traffic load placed otherwise on a single gateway. Such load balancing decisions can be made either statically or dynamically. All virtual nodes in the virtual network can be partitioned and assigned to different simulation gateways at configuration time. The entire emulation traffic is therefore divided among the simulation gateways and a better throughput is anticipated. Alternatively, a client may dynamically choose among a set of simulation gateways at connection time. For example, the IP Virtual Server (IPVS) can be used to implement a rather sophisticated load balancing scheme at a front-end server, which subsequently redirects services to a cluster of servers at the back-end [Lin]. OpenVPN's simple load-balancing scheme, where clients can choose randomly to connect to a server from a set of simulation gateways at the time of connection, was adopted in our imple-

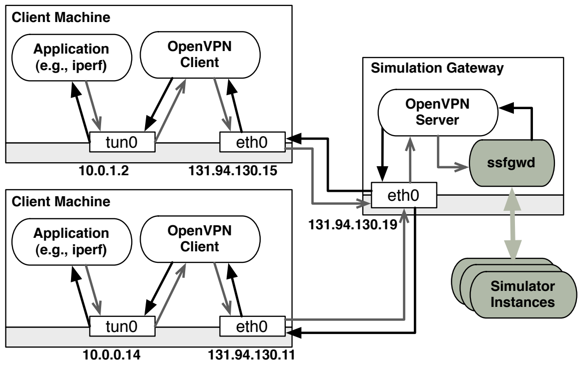


Figure 5.3: OpenVPN Uses Virtual Network Interfaces for Tunneling Application Traffic.

mentation. Additionally, mechanisms to allow a client to connect to a different gateway were implemented. This modification registers clients with the simulator, so traffic from the simulator will be routed to the client machine through the gateway with which the client is currently engaged.

OpenVPN was chosen to allow applications to dynamically connect to the simulation gateway and to emulate network interfaces in the application runtime environments. OpenVPN was chosen because it is publicly available and runs on most operating systems. OpenVPN uses the TUN/TAP interface provided by the operating system. As shown in figure 5.3, each OpenVPN instance creates a virtual network device (named `tun0` in this case). The virtual network device has two end-points: one as a regular network interface with an assigned IP address, and the other with a file interface through which one can apply read and write operations. The virtual network interface is assigned an IP address that matches the IP address of the emulated network interface in simulation. An entry to the kernel forwarding table is also added to direct all traffic targeted at the virtual IP address space to be sent through the virtual network device. In this way, the client machine will assume the proper identity, as applications running on this machine can transparently forward traffic to the simulated network. IP packets sent to the virtual network interface are read by OpenVPN through the file interface. Subsequently, OpenVPN applies proper data

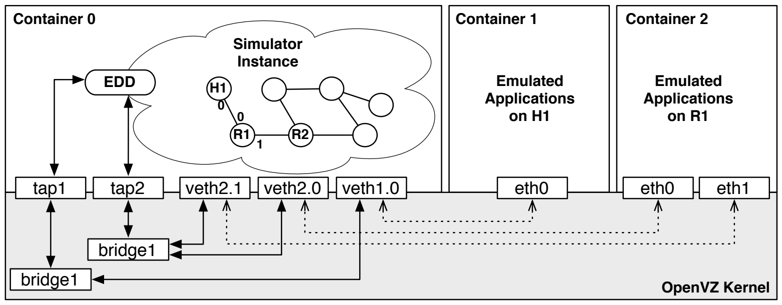


Figure 5.4: An EDD that employs a TAP driver to exchange packets with collocated application runtime environments.

compression and encryption to the packets before sending them via UDP to the OpenVPN server running at the simulation gateway. In the opposite direction, UDP packets received from the remote OpenVPN server are decrypted and decompressed before the packets are written out the virtual network interface. Thus, the packets arrive at the application as if they come directly from the virtual network interface.

5.3.2 Collocated Emulated Hosts

This section describes an interconnection mechanism designed for collocated emulated hosts, which use Linux bridges and TAP devices. Figure 5.4 depicts our design. For each collocated emulated host, we create a software bridge and a TAP device connected to the bridge. Multiple bridges are used to segregate the traffic of each emulated host and ensure the emulated traffic is routed properly. In particular, we do not allow collocated emulated hosts to communicate with each other directly without going through the simulator. There are two possible alternatives. One could use VLANs to separate the traffic. However, processing the VLAN headers may introduce additional overhead. One could also use ebtables, which is a mechanism for filtering traffic passing through a Linux bridge [Art].

In the implementation, we use the TAP device for the EDD collocated at the simulator instance (in container 0) to send and receive packets to and from the VNICs[1]. The EDD also acts as an ARP proxy responding to the ARP requests from the emulated hosts, so as to direct packets originated from the VNICs to the associated TAP device.

The collocated emulated hosts are running as OpenVZ containers. We use a virtual ethernet device [Opeb] for each VNIC on the emulated hosts. The virtual ethernet device is an ethernet-like device, which actually consists of two ethernet interfaces—one in container 0 (the privileged container) and the other in the container where the emulated host is. The two interfaces are connected to each other, so that a packet sending to one interface will appear at the other interface. We connect the interface at container 0 to the bridge which corresponds to the emulated host.

In order to explain how this mechanism works we will follow the path of a ping packet from H1 via R1 and R2 as shown in figure 5.4. Suppose H1 and R1 are emulated in container 1 and 2, respectively. Before container 1 can send a packet to R1, assuming its ARP cache is currently empty, it sends an ARP request out from eth0 asking for the MAC address of R1's network interface eth0. The EDD responds with the MAC address of tap1, which is the TAP device connected to bridge1, the software bridge designated for container 1. Subsequently, container 1 is able to forward the ICMP packet to the EDD, which injects the packet into the simulator (by inserting an event that represents the packet sent out from network interface 0 of the simulated host H1). Once the simulation packet gets to R1 on the simulated network, the packet is exported from the simulator and sent by the EDD to eth0 in container 2 through tap2 and bridge2. Similarly, before container 2 forwards the packet onward to R2, it sends an ARP request out from eth1. The EDD responds with the MAC address of tap2, which is the TAP device connected to bridge2,

[1]The latest TAP device implementation prohibits writing IP packets to the kernel, possibly for security reasons. We choose to use a raw socket instead for the EDD to send IP packets.

the software bridge designated for container 2. In this way, the packet can find its way to the simulator, which forwards it on the virtual network.

5.3.3 Traffic Portals

A simulated network interface can be attached to an external network using a *traffic portal*. Traffic portals are an interconnection mechanism designed to allow physical networks to exchange traffic with the simulated network. Figure 5.5 shows an example where we attach two physical networks to the simulated network: the `13.0.0.0/8` network is attached to interface 1 of R1 and the `14.0.0.0/8` network is attached to interface 2 of R3. In this example, each traffic portal manages the connection to a single physical network; in general, a traffic portal is capable of managing multiple external networks with the only restriction being that each external network can only be attached to a single traffic portal.

The traffic portals are mapped to network interfaces on the compute node running the virtual network. In the previous example, interface 1 of R1 is mapped to eth1 and interface 2 of R3 is mapped to eth2 on the compute node. The EDD uses raw sockets to read/write raw ethernet packets from/to the interfaces. Additionally, the EDD runs the ARP protocol and responds to ARP requests for any IP addresses that are within the simulated network or other attached external networks. For example, the ARP on eth1 will respond to ARP requests for any IP in the simulated network and in the `14.0.0.0/8` network. For all other address the EDD will remain silent.

In order to explain the traffic portal mechanism, we will follow the path of a ping packet from 13.1.0.30 via R1, R2 and R3 to 14.1.0.20. Before 13.1.0.30 can send a packet to 14.1.0.20, assuming its ARP cache is currently empty, it sends an ARP request for the MAC address of 14.1.0.20. The EDD responds with the MAC address of eth1, which is the network interface mapped to the traffic portal for the `13.0.0.0/8` network.

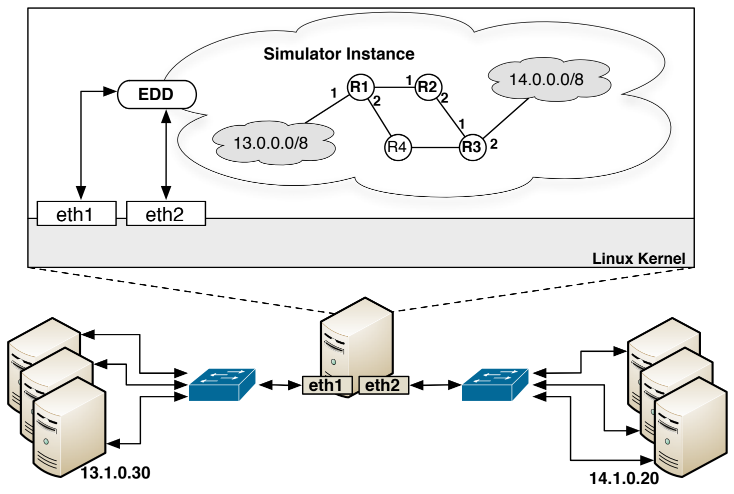


Figure 5.5: Traffic portal example

Subsequently, the EDD will receive the ICMP packet, which is injected into the simulator (by inserting an event that represents a packet received on network interface 1 of the simulated router R1). The packet is routed through the virtual network until it arrives at R3. The packet is then exported from the simulator and sent by the EDD via eth2 and routed by the real switch to 14.1.0.20. Similarly, before 14.1.0.20 can respond to the ICMP ECHO request, it must send an ARP for 13.1.0.30. The EDD responds with the MAC address of eth2. In this way, real packets can be routed back through the virtual network.

In the previous example we have directly attached eth1 and eth2 to separate switches of the corresponding physical networks. This is but one of many configurations that could be used. One could also attach eth1 and eth2 to a single switch. However, the interfaces must segregate the traffic of the two physical networks; otherwise the underlying hardware could bypass the simulator and route the traffic around the simulated network. We can simply use VLAN tags to segregate traffic from the two networks in this case. Furthermore, the structure of the external network could be arbitrarily complex. For the sake of clarity we have kept this example simple.

5.4 Compute Node Configuration

Virtual machines (*VMs*) come in many different forms, but can generally be categorized into system level virtualization and OS level virtualization. System level VMs allow for entirely distinct operating systems, each with their own kernels, to be run along side the host OS, thereby sharing the underly hardware. System level VMs are typically expensive in terms of resources since the different OSes cannot share resources or services, and the hypervisor [SN05] must intervene to broker access to shared devices like network interface cards. OS level VMs can partition the systems into a set of distinct *containers* that each appear to be stand-alone machines. The major difference is that OS level VMs typically share a common kernel. As such, OS level VMs are very efficient in comparison to system level VMs because they can share resources between the *containers*.

There are a number of different OS level VM solutions available, each one providing a unique set of features. Application runtime environments must provide some level of memory, CPU, and network isolation. Memory and CPU isolation are needed to limit the interactions of applications from different virtual nodes that are emulated on the physical hardware. Network isolation is necessary to capture traffic from the application runtime environments and to inject traffic back into the application environments. Without the network isolation, one would have to employ to the techniques to capture and inject traffic used in MaSSF [LXC03].

We have adopted OpenVZ [Opea] to run the virtual machines for the emulated hosts where unmodified network applications can run. OpenVZ is an OS-level VM solution that meets our minimum requirements in terms of providing necessary separation of CPU, memory, and network resources among the VMs. OpenVZ compartmentalizes the system resources inside the so-called containers; applications can run within these containers as if they were running on dedicated systems. Each container has its own set of processes, file system, and manages its own set of users (including root), and network stack (including network interfaces and routing/forwarding tables). Using OpenVZ, we are able to perform network experiments with a large number of emulated hosts.

We provide an OS template with which we preload each guest container. The OS template consists of a root file system and common network applications that may be run on a container (such as iperf and tcpdump). Additional applications can be added to individual containers using the meta-controller framework, which we describe in section 5.5.

Each container maintains its own file system. The amount of storage space needed by the file system at each container may range from tens to hundreds of megabytes, depending on the applications the experimenter wishes to run. Consequently, network experiments with a large number of emulated hosts could require a lot of disk space. To solve this problem, we choose to use a union file system [Ste]. A union file system consists of two parts: a read-only file system (RF), which is common to all containers, and a writable file system (WF), which is specific to a container. A copy-on-write policy is used: when a file in RF needs to be modified, the file will be first copied to WF. To prepare the file system for each emulated host, we union the default OS-template with an initially empty writable base file system. In this way we need only to store changes that later occur for each container as opposed to storing the entire file system.

Figure 5.6 depicts the two possible configuration scenarios of a compute node. If there are collocated emulated applications, then there will be one simulator instance on the compute node run in container zero. The emulated applications would then be run in separate containers, one for each virtual node. If the compute node does not host collocated emulated applications, then there will be a simulator instance for each processor on the compute node. This second case applies even if the simulation instance contains a remote application environment or traffic portals.

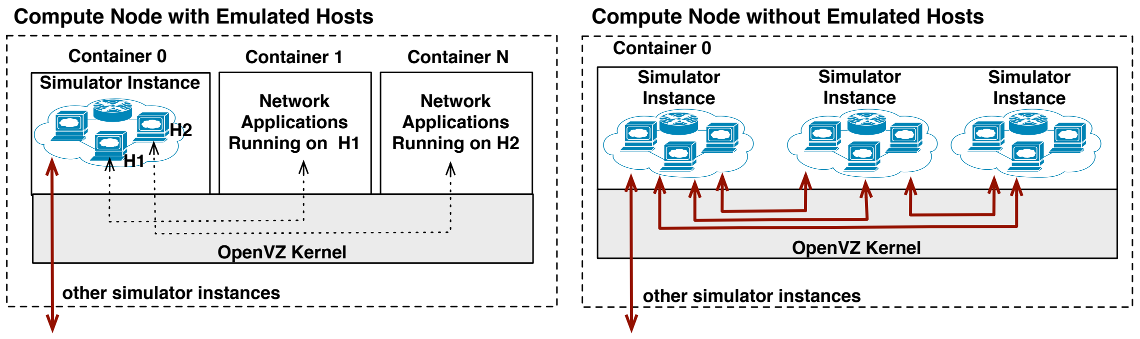

Compute Node with Emulated Hosts
Container 0
Container 1
Container N
Simulator Instance
Network Applications Running on H1
Network Applications Running on H2
H2
H1
OpenVZ Kernel
other simulator instances
Compute Node without Emulated Hosts
Container 0
Simulator Instance
Simulator Instance
Simulator Instance
OpenVZ Kernel
other simulator instances

Figure 5.6: Compute Node configurations with and without emulated hosts. Simulation instances use MPI to communicate, and the emulated hosts communicate with their col-located simulation instance via emulation device drivers.

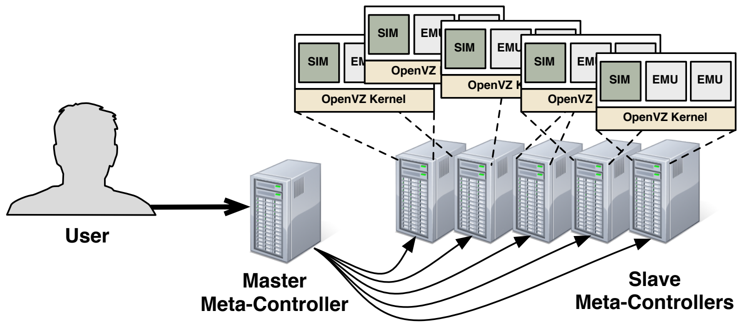


Figure 5.7: Meta-controllers

5.5 Meta-Controllers

Manually configuring each compute node within a large experiment proves to be a difficult and burdensome task. We developed a *tiered* command framework, which automates the configuration of the compute nodes used for the experiment, and orchestrates experiment execution.

The command framework, as illustrated in figure 5.7, uses MINA, a Java framework for distributed applications [Apab]. 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 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 master meta-controller acts as a central point to receive commands which are then distributed to the slave controllers which will execute them. In order to launch an emulation experiment, a user will start meta-controllers on all utilized compute nodes, and send the experiment to a controller chosen to be the master.

The master controller will then issue commands to all the slaves in order to configure the compute nodes to run the simulation and corresponding emulated hosts or traffic portals. 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 nonblocking command. If it is a blocking command, the meta-controller will wait until the command finishes execution, and returns the result of the command (i.e., the exit status) to the user. If the command is a nonblocking command, the meta-controller forks a separate process to handle the command and immediately responds to the user. Our current implementation uses blocking commands to instantiate the containers and the emulation infrastructure, and uses nonblocking commands to start the simulator and the experimental applications within the containers. To configure the experiments correctly, the meta-controllers on the compute nodes need to run a series of commands:

1. *Set up MPI.* The master meta-controller creates the machine file and instructs the slave meta-controllers to generate the necessary keys for MPI to enable SSH logins without using passwords.

2. *Create containers.* The meta-controller creates a container for each collocated emulated host on the compute node. This step also includes creation of union file systems for the containers.

3. *Initialize the emulation infrastructure.* This step includes installing and configuring necessary network devices and other software components (such as software bridges and TAP devices) in containers, and on the physical host. 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 experiment 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 the experiment down by terminating the simulator, stopping the containers, and removing the emulation infrastructure.

The meta-controller framework allows users to easily script entire emulation experiments. Users can script the deployment and configuration of the compute nodes, as well as schedule real-applications to generate traffic and interact with the virtual network during the experiment's execution. The cleanup of the compute nodes and final shutdown of the experiment can also be scripted. The meta-controller framework is an extension of the real-time monitoring and control framework presented in section 3.4.

5.6 Evaluation

The experiments described in this section are conducted on a Linux cluster with eight Dell PowerEdge R210 rack-mount servers, each with dual quad-core 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 testbed by comparing the TCP performance between emulation and simulation. TCP is used in these experiments because it is highly sensitive to delay, jitter, and losses, and can therefore magnify the errors introduced by the emulation infrastructure. We choose to use our simulated TCP protocols has a baseline in these experiments because the low-level behaviors of real or emulated TCP protocols will vary slightly for each run. We can do this because our simulated TCP protocols have previously been validated to produce realistic behavior [ELL09].

Remote Emulated Hosts

The TCP congestion window trajectories achieved by the real Linux TCP implementations on the OpenVPN clients are compared against those from our simulation. We arbitrarily choose three congestion control algorithms: BIC, HIGHSPEED, and RENO; out of the 14 TCP variants implemented in the PRIMEX simulator. We use a dumbbell network model for the experiments. The dumbbell model has two routers connected by 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 experiments, 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 the three scenar-

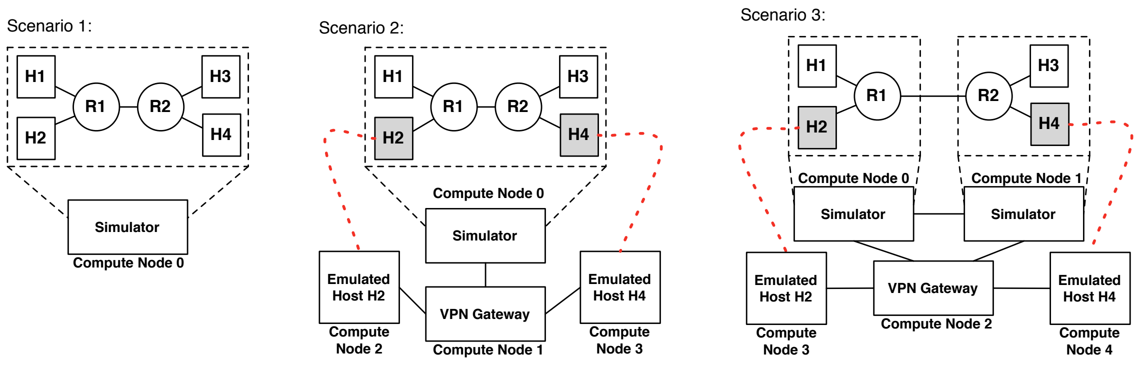


Figure 5.8: Dumbbell experiment scenarios for remote emulation.

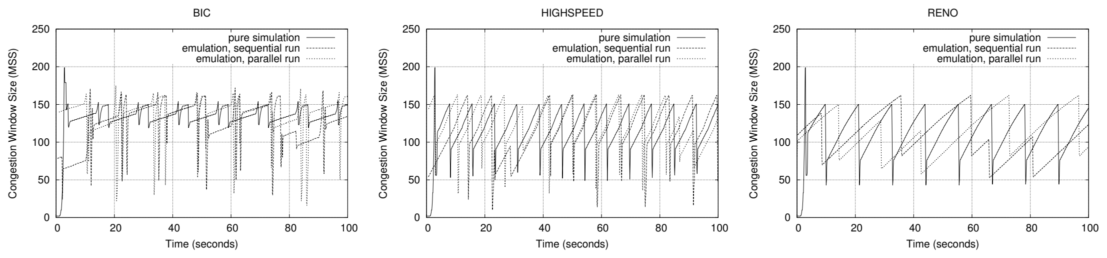


Figure 5.9: TCP congestion window trajectories for remote emulated hosts.

ios depicted in in figure 5.8. In the first scenario, we perform pure simulation and use a simulated traffic generator. The pure simulation scenario (S1) is compared against scenarios two (S2) and three(S3). In remaining two scenarios, we designate the two hosts as remote emulated hosts. In order to collect the TCP trajectories from the remote emulated hosts, we sample the (/proc/net/tcp) file at regular intervals to extract the TCP congestion window size. For simulation, we use a script to analyze the trace output.

Figure 5.9 shows an example of the congestion windows trajectories (CWTs) we observed. The trajectories for RENO and HIGHSPEED match simulation well for all three scenarios. BIC's trajectories do not match as well because BIC is more sensitive to jitter in the delay which our OpenVPN infrastructure can introduce. However, for all three algorithms, the congestion window peaks around 150 segments, which matches well with simulation.

We computed the following three metrics to compare congestion window trajectories:

- We calculated the normalized Manhattan distance ($\mathsf{d}_\mathsf{m}^{\mathsf{X,Y}}$) between the simulated trajectory (scenario one) and emulated trajectories (scenarios two and three). We calculated the normalized Manhattan distance as

$$\mathsf{d}_\mathsf{m}^{\mathsf{X,Y}} = \frac{1}{\mathsf{n}} \sum_{\mathsf{i}=0}^{\mathsf{n}} \left| \mathsf{X_i} - \mathsf{Y_i} \right| \tag{5.1}$$

- We calculated the normalized Euclidean distance ($\mathsf{d}_\mathsf{e}^{\mathsf{X,Y}}$) between the simulated trajectory (scenario one) and emulated trajectories (scenarios two and three). We calculated the normalized Euclidean distance as

$$\mathsf{d}_\mathsf{e}^{\mathsf{X,Y}} = \frac{1}{\mathsf{n}} \sqrt{\sum_{\mathsf{i}=0}^{\mathsf{n}} \left(\mathsf{X_i} - \mathsf{Y_i} \right)^2} \tag{5.2}$$

- We calculated the normalized dynamic time warp distance between the simulated trajectory (scenario one) and emulated trajectories (scenarios two and three). Dynamic time warp [BC94, HDSM05] automatically aligns the two congestion window trajectories in order to minimize the the $\mathsf{DTWError}$:

$$\mathsf{DTWError} = \sum_{\mathsf{k}=0}^{|\mathsf{W}|} \left| \mathsf{X_i} - \mathsf{Y_j} \right| \text{ where } \mathsf{W_k} = (i, j) \tag{5.3}$$

  $\mathsf{W_k}$ is the alignment of the two traces which minimizes $\mathsf{DTWError}$. We used the dynamic time warp package provided by Giorgino [Gio09] to compute $\mathsf{W_k}$ and the normalized dynamic time warp distance ($\mathsf{d}_\mathsf{w}^{\mathsf{X,Y}}$).

The results are shown in table 5.1. The $\mathsf{d}_\mathsf{m}^{\mathsf{X,Y}}$ and $\mathsf{d}_\mathsf{e}^{\mathsf{X,Y}}$ metrics indicate very different behavior. On one hand, $\mathsf{d}_\mathsf{m}^{\mathsf{X,Y}}$ indicates that both the parallel (S3) and sequential (S2) traces for all TCP variants differ by more than 20 packets in comparison to the simulated trace. The difference is so large because $\mathsf{d}_\mathsf{m}^{\mathsf{X,Y}}$ is sensitive to local noise and the traces are not perfectly aligned. On the other hand, $\mathsf{d}_\mathsf{e}^{\mathsf{X,Y}}$ shows that the traces differ by single packet. The difference is small because $\mathsf{d}_\mathsf{e}^{\mathsf{X,Y}}$ dampens local noise. Neither $\mathsf{d}_\mathsf{m}^{\mathsf{X,Y}}$ nor $\mathsf{d}_\mathsf{e}^{\mathsf{X,Y}}$ are good metrics for similarity of the congestion traces.

	$d_m^{S1,S2}$	$d_m^{S1,S3}$	$d_e^{S1,S2}$	$d_e^{S1,S3}$	$d_w^{S1,S3}$	$d_w^{S1,S3}$
BIC	22.29	18.38	0.97	0.95	5.25	5.10
HIGHSPEED	29.79	32.79	1.14	1.29	4.58	4.20
RENO	31.54	31.73	1.23	1.24	7.97	7.57

Table 5.1: Error metrics for remotely emulated congestion window trajectories.

Dynamic time warp aligns the traces before computing the differences. The result is that $d_w^{X,Y}$ shows the traces differ by about 5 packets on average. This matches a visual inspection of figure 5.9. The alignments that were chosen by dynamic time warp are shown in figure 5.13. In general, the alignments track the behavior of the two curves very well.

Collocated Emulated Hosts

We use the same three TCP variants and dumbbell model as in the remote emulation experiments. For each TCP algorithm we test the three scenarios depicted in figure 5.11. In the first scenario, we perform pure simulation and use a simulated traffic generator. The pure simulation scenario is compared against scenarios two and three. In the remaining two scenarios, we designate the two hosts as collocated emulated hosts.

Figure 5.12 shows an example of the congestion windows trajectories (CWTs) we observed. The trajectories for all three variants match simulation well for all three scenarios. We also observe the congestion window for all three variants peaking around 150 segments, the same as in the remote emulation experiments. We also computed the normalized Manhattan, Euclidean, and dynamic time warp distances as we did for remotely emulated hosts. The results are shown in table 5.2. Again, the $d_m^{X,Y}$ and $d_e^{X,Y}$ metrics indicate very different behavior. $d_m^{X,Y}$ indicates that both the parallel (S3) and sequential (S2) traces for all TCP variants differ by more than 15 packets in comparison to the simulated trace. $d_e^{X,Y}$ shows that the traces differ by single packet. Visually, we can see that the con-

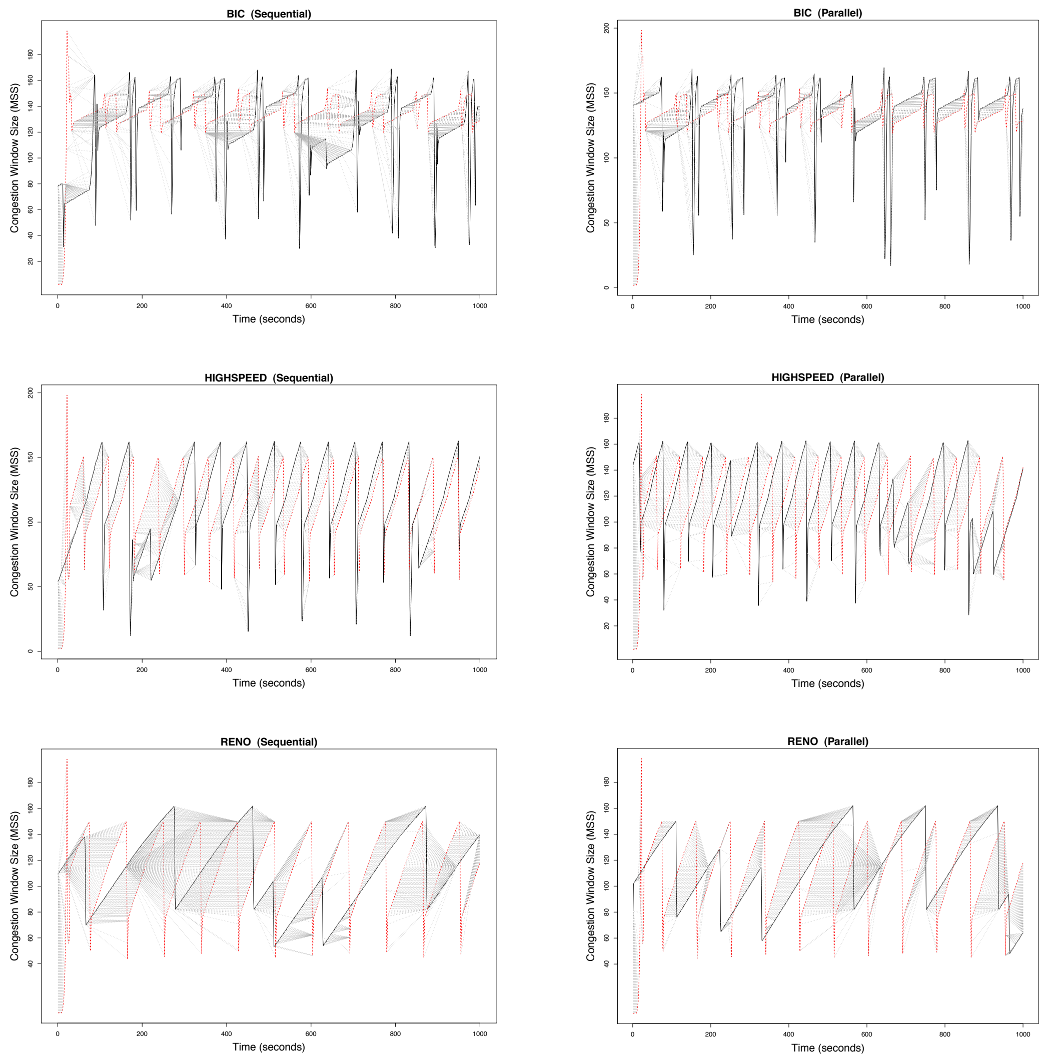


Figure 5.10: Congestion window trajectory distance mappings for remote emulated hosts.

gestion window trajectories for collocated emulated hosts match the simulated trajectory better than the trajectories of remotely emulated hosts, yet neither $d_m^{X,Y}$ nor $d_e^{X,Y}$ indicate this. Dynamic time warp, however, indicates that the collocated traces are more similar to the simulated traces than those of remotely emulated hosts. Again, $d_w^{X,Y}$ is the preferred metric. The alignments that were chosen by dynamic time warp are shown in figure 5.13. In general, the alignments track the behavior of the two curves very well.

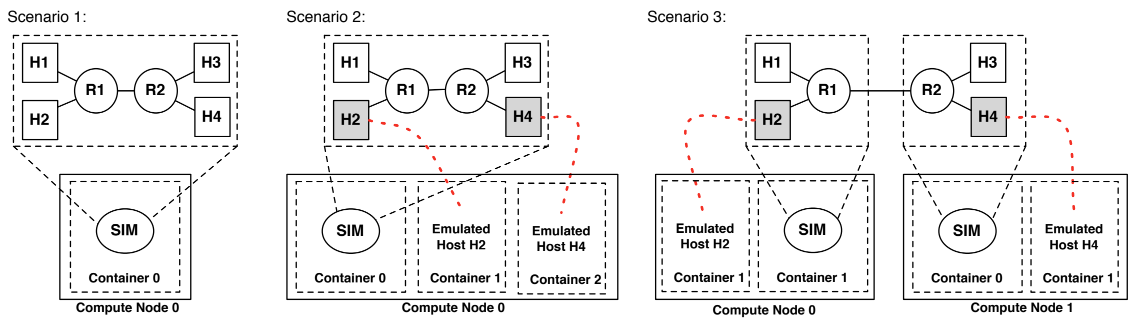


Figure 5.11: Dumbbell experiment scenarios for collocated emulation.

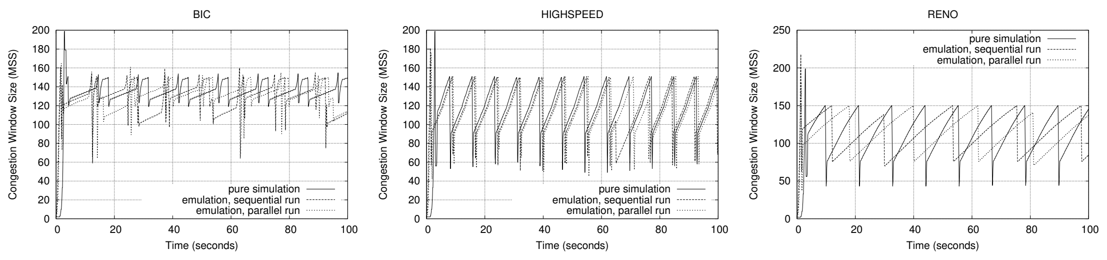


Figure 5.12: TCP congestion window trajectories for collocated emulated hosts.

	$d_m^{S1,S2}$	$d_m^{S1,S3}$	$d_e^{S1,S2}$	$d_e^{S1,S3}$	$d_w^{S1,S3}$	$d_w^{S1,S3}$
BIC	15.02	14.19	0.68	0.63	2.25	1.54
HIGHSPEED	15.60	17.97	0.82	0.85	1.36	1.18
RENO	27.47	28.54	1.11	1.12	4.81	4.86

Table 5.2: Error metrics for collocated emulated congestion window trajectories.

Next, we use a TCP fairness test to show whether our approach can correctly intermingle emulated and simulated packets. Using scenario two from figure 5.11, 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 seconds after the first flow. We compare the first and the second scenario. The results are shown in figure 5.14. In both cases, we see that the congestion window size of the first TCP flow reduces when the second TCP flow starts;

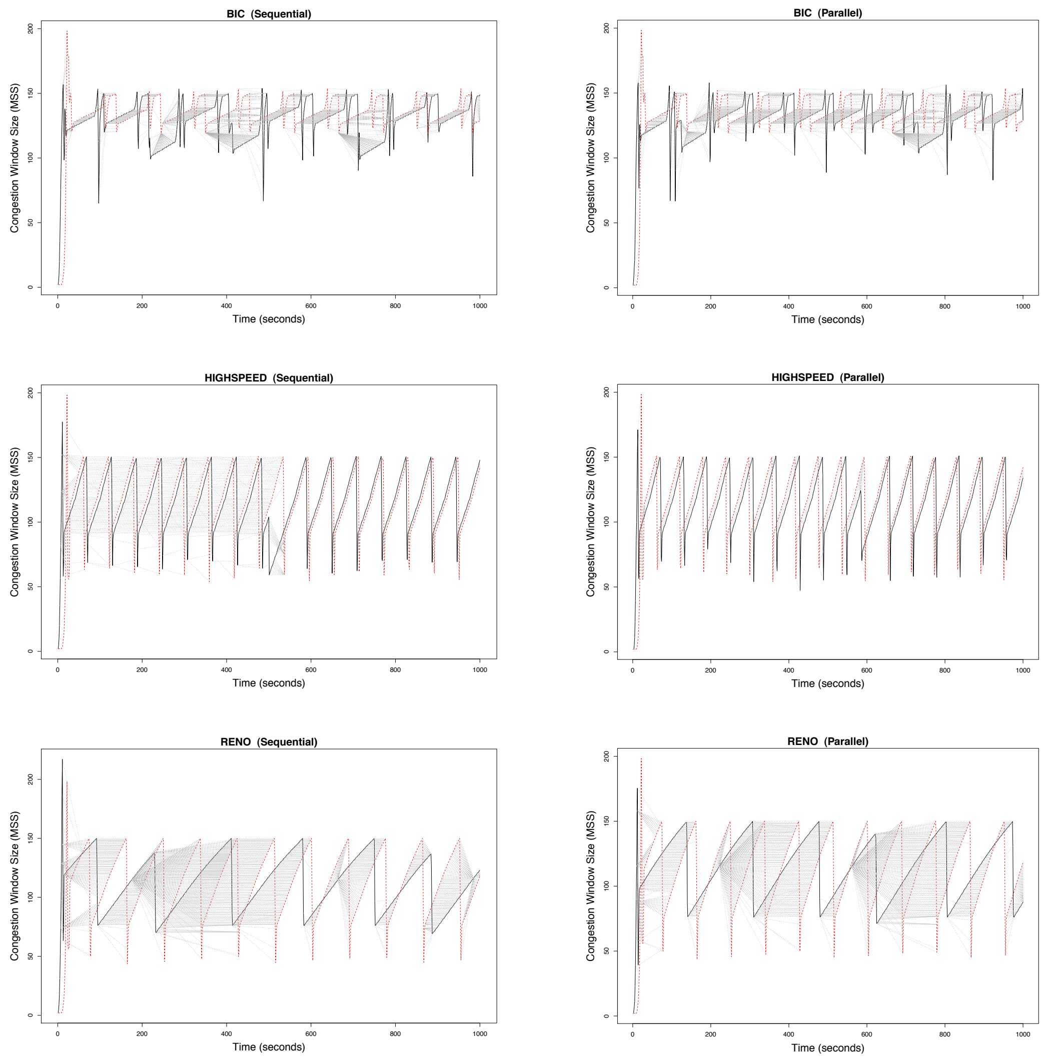


Figure 5.13: Congestion window trajectory distance mappings for collocated emulated hosts.

both flows eventually converge with a fair share of the bandwidth (at about 30 seconds after the second flow starts transmitting). Again, we see similar results between simulation and emulation.

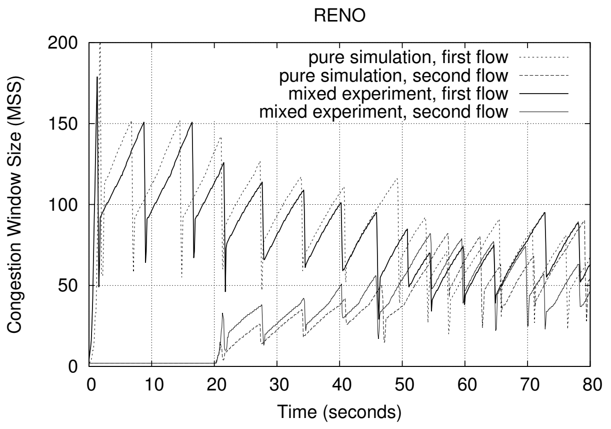


Figure 5.14: TCP fairness

Traffic Portals

In this experiment, we investigate whether the traffic portal can correctly intermingle real and simulated flows. We evaluated three scenarios shown in figure 5.15. In the first scenario (called "simulated"), we simulate the entire model. In the second scenario (called "mixed"), we simulate all of the routers, as well as hosts S2, C2, S3 and C3. We use two real hosts, S1 and C1, and connect them to the simulator via the traffic portals. In the third scenario (called "physical"), we instantiated the whole network topology on Utah EmuLab [WLS+02].

During the experiment, we first start Flow1 at time zero, which consists of a single TCP flow from S1 to C1. We then start Flow2 10 seconds later, which consists of two separate TCP sessions from S2 to C2. 10 seconds after Flow2, we start Flow3, which also consists of two TCP sessions from S3 to C3. Each TCP session in Flow2 and Flow3 transfers 100 MB. The TCP session in Flow1 transfers an object large enough that it does not complete during the experiment.

In figure 5.16 we plot the TCP sequence numbers over time for Flow1. We conducted 25 runs per data point and plot the average with the 95% confidence intervals. The results match fairly well in the first 10 seconds, and after 50 seconds. Between 10 and 50 seconds,

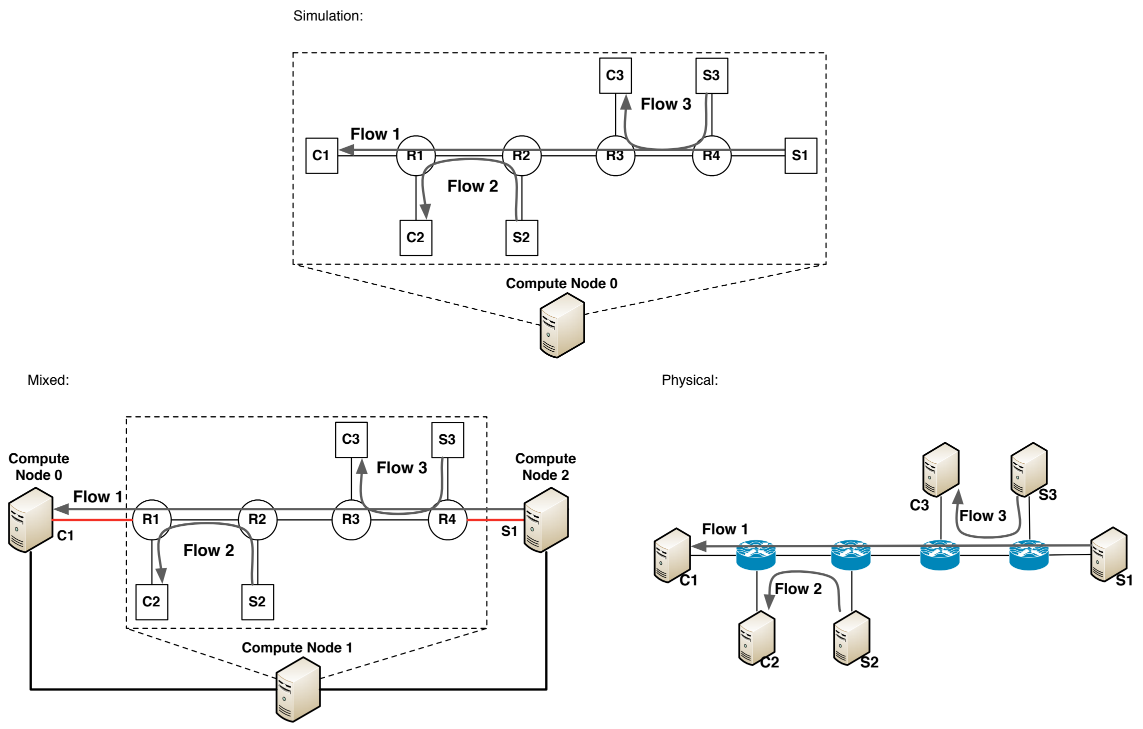


Figure 5.15: Cross-traffic scenarios.

when Flow1 interacts with the other two flows, we observe some difference in throughput achieved for the mixed scenario. We attribute this to the additional delay imposed by the I/O system that we discuss in the next section.

5.6.2 Performance Studies

The emulation infrastructure inevitably puts a limit on the throughput of the emulated traffic. In this section we explore the limitations of the different types of EDDs.

Remote Emulated Hosts

For this experiment, we use same dumbbell model as in the validation studies. However, to increase the TCP throughput, we reduce the delay of the bottleneck link of the dumbbell

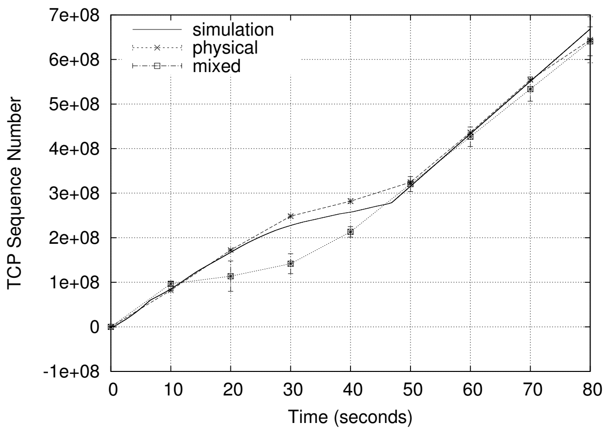


Figure 5.16: TCP sequence numbers for Flow 1

model to 1 millisecond. We vary the bandwidth of the bottleneck link from 10 Mb/s to 450 Mb/s with increments of 40 Mb/s. As in the previous experiment, we direct a TCP flow from one host to the other through the bottleneck link and we compare the three scenarios in figure 5.8.

Figure 5.17 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 200 Mb/s for sequential runs (scenario 2) and 130 Mb/s for parallel runs (scenario 3). 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 (over MPI).

Collocated Emulated Hosts

Again, for this experiment, we use same dumbbell model as in the validation studies, but we reduce the delay of the bottleneck link of the dumbbell model to 1 millisecond. We vary the bandwidth of the bottleneck link from 10 Mb/s to 450 Mb/s with increments of

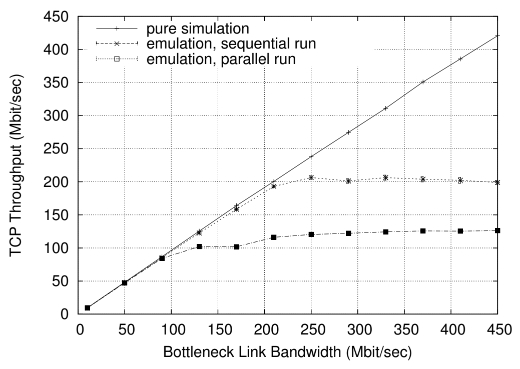


Figure 5.17: TCP throughput for remote hosts

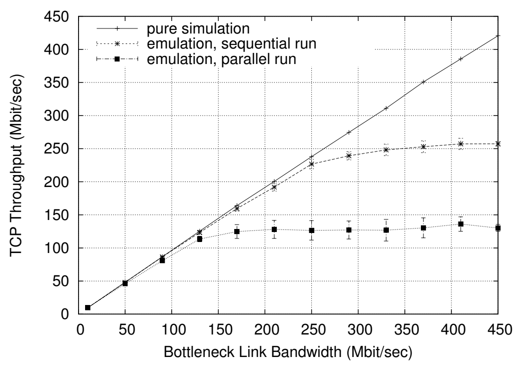


Figure 5.18: TCP throughput for collocated hosts

40 Mb/s and direct a TCP flow from one host to the other through the bottleneck link and we compare the three scenarios in figure 5.11.

Figure 5.18 shows the results. Again, the throughput increases almost linearly with the increased bandwidth for the simulated flow and at high traffic intensity the error becomes apparent for emulated traffic. The throughput for the emulated traffic is kept below roughly 250 Mb/s for sequential runs (scenario 2) and 130 Mb/s for parallel runs (scenario 3). Sequential runs see an additional 50 Mb/s of throughput over remote emulation because of the reduced delay. Again, 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 (over MPI).

We were surprised to find the throughput of the sequential run was limited to roughly 250 Mb/s. We conducted preliminary evaluations using UDP and saw a throughput closer to 500 Mb/s. After an extensive investigation, we discovered that the I/O system intermittently imposed significant delays on packets when the packet rate was low. This would have a significant impact on TCP during its starting phase, and cause TCP to throttle the throughput.

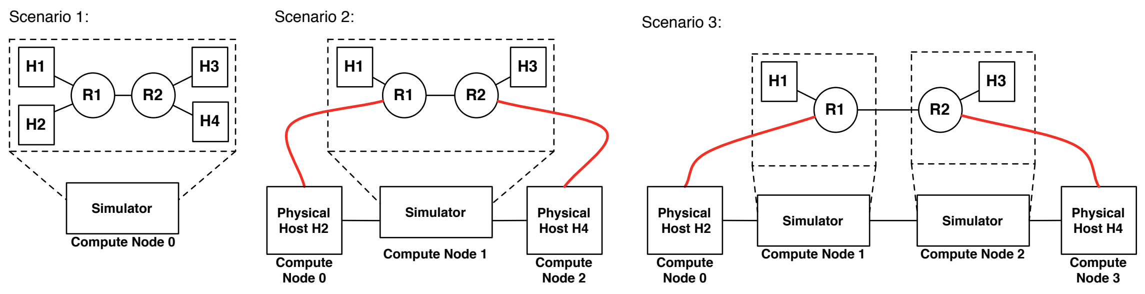


Figure 5.19: Dumbbell experiment scenarios using traffic portals.

Traffic Portals

We also take a closer look at the capacity of the traffic portal. We use the same dumbbell model as in the previous section. However, this time we test the three scenarios in figure 5.19. Both H2 and H4 are physical hosts, and we run the simulator either sequentially on one compute node, or in parallel on two compute nodes. We direct either one or ten TCP flows between the two physical hosts through the traffic portal. We again compare the results with those from the pure simulation case.

Figure 5.20 shows the results. While the throughput increases linearly with the increased bandwidth for the simulated flow, it becomes apparent that the traffic with 10 TCP flows crossing through the traffic portal is capped at roughly 500 Mb/s for sequential runs and 180Mb/s for parallel runs. Again, the reduced throughput for parallel runs is due to the communication overhead imposed by the simulator as the TCP traffic goes across the boundary of the parallel simulator instances. This indicates there is room for future improvement.

In the case of 1 TCP flow, the throughput is significantly lower. We discovered that this is due to the same timing issue at the I/O system that we mentioned earlier. The sporadic delay imposed on the packets caused TCP to throttle its congestion window size. As a result, we also found significant variance in the throughput measurement for the

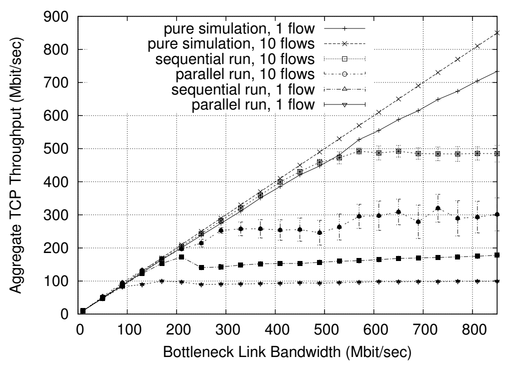


Figure 5.20: Cross-traffic TCP throughput

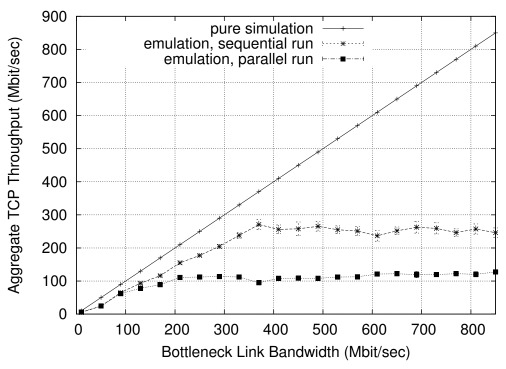


Figure 5.21: Interactive TCP throughput

sequential runs. This behavior is not obvious for the parallel runs because the throughput is limited by MPI communication overhead.

In another experiment, we evaluated the ability for an external physical host to interact with a simulated host using TCP. The physical host acted as a client to simultaneously download 10 files of significant size from a simulated host. PRIMEX provides full TCP interoperability. The results are shown in figure 5.21. We measured the aggregate throughput of all TCP flows and found the throughput is limited to about 250 Mb/s for sequential runs and about 120 Mb/s for parallel runs. This is mainly due to the simulator falling behind when generating large amounts of traffic. This indicates further room for improvement.

Emulation Capacity

In the previous experiments, we determined that the emulation infrastructure is placing 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 as we increase the number of TCP flows, while increasing the number of emulated hosts we run.

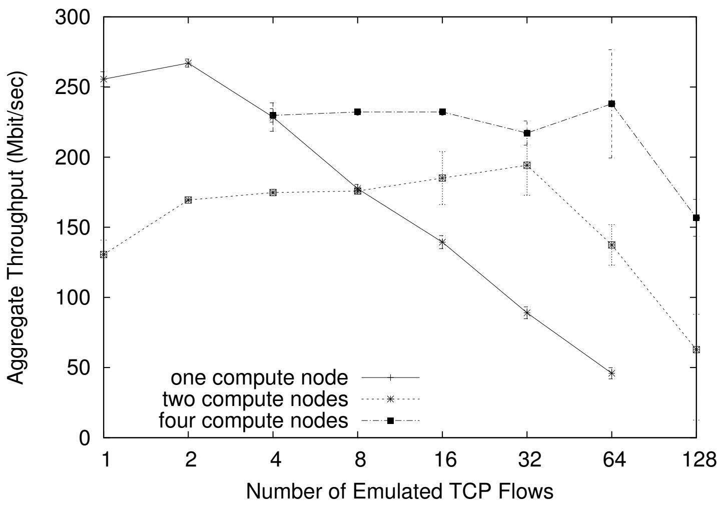


Figure 5.22: Emulation capacity

Here we use the same dumbbell model (with a bottleneck link of 1 Gb/s bandwidth and 1 millisecond delay), and attach the same number of emulated hosts on each side of the dumbbell routers. We start a TCP flow (using iperf) for each pair of emulated hosts, one from each side of the dumbbell. So the total number of TCP flows is half the number of emulated hosts. We create the same number of the TCP flows from left to right as those from right to left. The measured throughput (from iperf) is shown for all emulated TCP flows in figure 5.22. The curves shown in figure 5.22 correspond to three experiments, described as follows.

In the first experiment, all emulated hosts are run on the same compute node. For one flow, the throughput reaches about 250 Mb/s. This is was also 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 (128 VMs). The slight increase is probably due to TCP's opportunistic behavior that allows it achieve better channel utilization with more flows. We suspect the emulated hosts (OpenVZ containers) are competing for shared buffer space in the kernel network stack, causing the drop in throughput. As more emulated hosts are placed on the same compute node, each container gets a smaller share of the available buffer space thereby degrading TCP's performance.

In the second experiment, we divide the network between two compute nodes (splitting the model along the bottleneck link). As observed previously, the throughput for one emulated flow in this case is around 130 Mb/s. As we increase the number of flows, the aggregate throughput slightly increases until we reach 32 flows. After 32 flows we start to see a significant drop in the aggregate throughput. Running the simulation on two compute nodes results in more event processing power; the simulator is capable of handling more emulated flows than in the sequential case.

In the third experiment, we extend the dumbbell model by placing four core routers in a ring and connecting them using bottleneck links. We attach the same number of emulated hosts to each core router. We direct TCP flows from an emulated host attached to one router to an emulated host attached to the next router in the ring. This allows us to spread the number of emulated flows evenly among the emulated hosts. The aggregate throughput for the four-node case is higher than the scenarios using two compute nodes. Again, we think this is due to the higher processing power of the parallel simulator. The throughput starts to drop at 128 flows (that's 32 VMs per compute node) resulting from increased contention for the shared buffer space.

5.7 Conclusion

A real-time network simulation and emulation framework that is both scalable and flexible has been presented. Both the scalability and flexibility are derived from the ability to emulate applications in both remote and collocated environments. The framework provides the necessary tools for experimenters to construct and configure the network models, allocate, deploy and run the experiments, and collect the experiment results. The resources may consist of the compute nodes in the cluster and possibly remote machines.

Each compute node is treated as a scaling unit, and can be configured to run a parallel simulator instance together with a set of virtual machines for emulated hosts. The latter provides a realistic operating environment for testing real implementations of network applications and services. The PRIMEX network simulator uses distributed simulation techniques to synchronize the simulator instances running within the scaling units. The virtual machines are connected with the simulator instances using a flexible emulation infrastructure.

Our validation experiments show that our infrastructure can produce accurate emulation results when the emulated traffic does not exceed the capacity of the emulation infrastructure. Our performance studies show that the throughput of the emulated traffic is dependent upon the number of virtual machines running on each scaling unit.

CHAPTER 6
CONCLUSIONS

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

6.1 Summary

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

1. A poor separation of concerns has lead to complex and unwieldy tools that are not designed for performing large-scale network experiments, but for exploring the details related to implementing a large-scale simulator itself. The end result is that using large-scale network simulators is still not a common practice in the network research community. In order to address this we did the following:

    - We developed the technique of *model splitting* which cleanly divides a network model into an interactive model and an execution model. Within the interactive model, we can focus on user facing concerns and in the execution model, we can focus on implementing an efficient and performant parallel simulator.
    - We developed *network scripting* which generates the interactive and execution models from a single network model using simple annotations.
    - We developed our *model caching* method in order to operate on large network models out-of-core by persisting the interactive model to a database.
    - We developed an interactive and control framework which allows for geographically separated execution and interactive models to be synchronized in real-time.

2. Large network models come with an enormous memory demand which can obturate the scalability of their execution. In order to reduce the memory complexity of large-scale network models we did the following:

    - We developed our *model replication* technique which allows large network models to be constructed in a recursive manner using duplicated sub-structures. We implemented our model replication technique in three different configuration languages (Java, Python, and XML).
    - We preformed an empirical study to determine whether network models constructed with large proportions of identical sub-structures exhibited different graph metrics than network models that were constructed using no identical sub-structures. Our study found that models with a large percentage of identical sub-structures were virtually indistinguishable from those with no duplicate sub-structures.
    - We developed a method to exploit model replication and reduce the memory required to store, instantiate, and execute network models. The core of this was the development of *spherical routing* which provides a flexible and realistic framework for routing within large-scale network models. Spherical routing is able to reduce the memory complexity of routing state by several orders of magnitude while preserving a high-degree of realism.

3. Large network models are hard to validate. We can improve the realism of network models by integrating network traffic from real applications and external networks into the simulation. To this end, we developed a flexible and scalable emulation framework.

- Our framework is able to exchange real network data with remote applications using OpenVPN and subject the real traffic to the conditions of the virtual network.
- Our framework is able to exchange real network data with applications running in collocated virtual machines and subject the real traffic to the conditions of the virtual network.
- Our framework is able to interact with external networks, subjecting all traffic that is routed through our framework to the conditions of the virtual network.
- Our interactive and control framework was extended to automate the deployment and configuration of large emulation experiments.

4. We implemented model splitting and model replication in our parallel real-time network simulator, called PRIMEX. We also embedded our emulation framework and interactive and control framework in PRIMEX. PRIMEX is the basis of the PrimoGENI project which embeds network simulation into the GENI federation of network testbeds [GEN]. Additionally, we developed an integrated development environment called *Slingshot* using PRIMEX's interactive model. Slingshot allows network researchers the ability to visualize, inspect, modify, persist, and reuse network models. Slingshot also automates the process of deploying, configuring, controlling, and executing network models involving real, emulated, and simulated components.

6.2 Future Directions

The research presented in this dissertation can be extended in at least three directions. The first direction would be to improve and extend spherical routing. Spherical routing is able to produce huge memory gains because it uses statically computed routing tables.

Requiring all spheres to be static reduces the scenarios which spherical routing can be used to study. Spherical routing could be extended in the following two ways:

1. In spherical routing, the routing within each sphere is mostly independent from the routing in other spheres. Spherical routing could be extended to support real routing protocols within a subset of routing spheres. Embedding a real routing protocol within a sphere could be done by embedding a simulated version of the protocol. To embed a real protocol using emulation or direct execution would be another approach. Using either method would prove very useful in studying how local routing protocols interact and are effected by routing in the Internet.
2. Currently within spherical routing, if a particular network interface or router goes offline, then all routes that use the failed router or interface will simply drop the packets. In other words, we do not reroute around network failures. This is due to the fact that spherical routing only maintains a single forwarding table for each routing sphere. In reality, a routing protocol would adapt to the failure and route around it. Spherical routing could be extended to support link, interface, or router failures by re-calculating shortest path routes online and dynamically switching between statically calculated and dynamically calculated forwarding tables.

Another direction would to be improve the emulation capacity of PRIMEX. Currently, we use standard interfaces to interact with collocated emulated hosts or external networks. The following could extend our emulation framework to support improved emulation capacity:

1. Using custom hardware such as FPGAs to offload the processing of real data. With this type of approach, the FPGA would interact with the virtual machine or external networks and signal the simulator using a light-weight API. In such a scheme, real packets in the network simulator only include the IP headers. When packets are

dropped, forwarded, or when IP header fields are changed, the simulator would communicate those changes to the FPGA.

2. Supporting emulation via directly executed virtual machines. By extending and combining the approach taken by the Denali isolation kernel [WSG02] and Weaves runtime framework [Var04] we could compile test applications with all their associated operating system dependencies into a single executable that could be embedded into the simulator. This would give us many of the benefits of running a full-blown virtual machine but with more control and less overhead.

3. Improving the real-time event scheduler within the simulation framework (SSF). The event scheduler currently uses a ”best effort” approach. There are two immediate problems with this. First, there is no warning or indication that the system on which the simulator is running is oversubscribed. It is not until the user observes artifacts from the oversubscription during execution of the simulation that they are notified. Second, when resources are in short supply, the scheduler has a bias for simulation events (or, with a simple modification, a bias for emulation events). We could extend and improve the scheduler by applying quality of service algorithms in order to guarantee a minimum level of service for emulation and simulation events.

The last direction would be to devise an algorithm which could automatically transform a network model that was not explicitly built using model replication into a network model which is equivalent with the exception of taking full advantage of model replication. Such a tool would be useful in determining how often real networks exhibit structural duplications. It would also allow users to automatically optimize their network model in terms of memory use and further reduce the burden of developing large-scale network experiments.

BIBLIOGRAPHY

[And] András Varga. OMNeT++ Network Simulation Framework. `http://www.omnetpp.org/`.

[Apaa] Apache Project. Derby Database. `http://http://db.apache.org/derby/`.

[Apab] Apache Project. Multipurpose Infrastructure for Network Applications (MINA). `http://mina.apache.org/`.

[Art] Art in Algorithms. Ethernet Bridge Tables (ebtables). `http://ebtables.sourceforge.net`.

[BBK$^+$06] Terry Benzel, Bob Braden, Dongho Kim, Clifford Neuman, Anthony Joseph, Keith Sklower, Ron Ostrenga, and Stephen Schwab. Experience with DETER: A testbed for security research. In *Proceedings of the 2nd IEEE Conference on testbeds and Research Infrastructures for the Development of Networks and Communities (TRIDENTCOM'06)*, 2006.

[BC94] D. J. Berndt and J. Clifford. Using Dynamic Time Warping to Find Patterns in Time Series. In *Proceedings of KDD-94: AAAI Workshop on Knowledge Discovery in Databases*, pages 359–370, Seattle, Washington, 1994.

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

[BFH$^+$06] Andy Bavier, Nick Feamster, Mark Huang, Larry Peterson, and Jennifer Rexford. In vini veritas: realistic and controlled network experimentation. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 3–14, New York, NY, USA, 2006. ACM.

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

[BSU00] Russell Bradford, Rob Simmonds, and Brian Unger. A parallel discrete event ip network emulator. In *MASCOTS '00: Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer* *and Telecommunication Systems*, page 315, Washington, DC, USA, 2000. IEEE Computer Society.

[BYC+03] D. Bauer, G. Yaun, C.D. Carothers, M. Yuksel, and S. Kalyanaraman. Ross.net: optimistic parallel simulation framework for large-scale internet models. In *Simulation Conference, 2003. Proceedings of the 2003 Winter*, volume 1, pages 703 – 711 Vol.1, dec. 2003.

[BYCK06] David Bauer, Murat Yuksel, Christopher Carothers, and Shivkumar Kalyanaraman. A case study in understanding OSPF and BGP interactions using efficient experiment design. In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation (PADS)*, pages 158–165, 2006.

[CBP00] Christopher D. Carothers, David Bauer, and Shawn Pearce. ROSS: a high-performance, low memory, modular time warp system. In *Proceedings of the 14th Workshop on Parallel and Distributed Simulation (PADS'00)*, pages 53–60, May 2000.

[CLL+99] James Cowie, Hongbo Liu, Jason Liu, David Nicol, and Andy Ogielski. Towards realistic million-node internet simulations. *International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[CM79] K.M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, 5(5):440–452, 1979.

[CNO99] James H. Cowie, David M. Nicol, and Andy T. Ogielski. Modeling the global internet. *Computing in Science and Engineering*, pages 42–50, 1999.

[CS03] Mark Carson and Darrin Santay. NIST Net: a Linux-based network emulation tool. *SIGCOMM Computer Communication Review*, 33(3):111–126, 2003.

[DFP+94] Samir Das, Richard Fujimoto, Kiran Panesar, Don Allison, and Maria Hybinette. Gtw: A time warp system for shared memory multiprocessors. In *Proceedings of the 26th conference on Winter Simulation*, WSC '94, pages 1332–1339, San Diego, CA, USA, 1994. Society for Computer Simulation International.

[DKH+05] X. Dimitropoulos, D. Krioukov, B. Huffaker, KC Claffy, and G. Riley. Inferring as relationships: Dead end or lively beginning? In *Proceedings of Springer Workshop on Experimental Algorithms (WEA)*, 2005.

[DKP+06] John DeHart, Fred Kuhns, Jyoti Parwatikar, Jonathan Turner, Charlie Wiseman, and Ken Wong. The open network laboratory. *ACM SIGCSE Bulletin*, 38(1):107–111, 2006.

[DKVR07] X. Dimitropoulos, D. Krioukov, A. Vahdat, and G. Riley. Graph annotations in modeling complex network topologies. 2007.

[DNP94] C. Daly, R. P. Neilson, and D. L. Phillips. A Statistical Topographic Model for Mapping Climatological Precipitation over Mountainous Terrain. *Journal of Applied Meteorology*, 33:140–158, 1994.

[DR03] X. Dimitropoulos and G. Riley. Creating realistic bgp models. *Symposium on Modeling, Analysis and Simulation on Computer and Telecommunication Systems (MASCOTS)*, 2003.

[DR04] X. Dimitropoulos and G. Riley. Large-scale simulation models of bgp. In *Proceedings of the 12th Annual International Symposium on Modeling, Analysis and Simulation on Computer and Telecommunication Systems (MASCOTS)*, pages 287–294, 2004.

[ELL09] Miguel Erazo, Yue Li, and Jason Liu. SVEET! A scalable virtualized evaluation environment for TCP. In *Proceedings of the 5th International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom'09)*, April 2009.

[ERS+09] Katherine J. Evans, Damian W. Rouson, Andrew G. Salinger, Mark A. Taylor, Wilbert Weijer, and James B. White, Iii. A scalable and adaptable solution framework within components of the community climate system model. In *Proceedings of the 9th International Conference on Computational Science*, ICCS 2009, pages 332–341, Berlin, Heidelberg, 2009. Springer-Verlag.

[Fal99] Kevin Fall. Network emulation in the Vint/NS simulator. In *Proceedings of the 4th IEEE Symposium on Computers and Communications (ISCC'99)*, pages 244–250, July 1999.

[FJ93] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, 1993.

[FPP+03] R. M. Fujimoto, K. Perumalla, A. Park, H. Wu, M. H. Ammar, and G. F. Riley. Large-scale network simulation: How big? how fast? In *Proceed-*

ings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation on Computer and Telecommunication Systems (MASCOTS), 2003.

[Fuj01] Richard M. Fujimoto. Parallel simulation: parallel and distributed simulation systems. In *WSC '01: Proceedings of the 33nd conference on Winter simulation*, pages 147–157, Washington, DC, USA, 2001. IEEE Computer Society.

[Gao01] L. Gao. On inferring autonomous system relationships in the internet. *IEEE/ACM Transactions on Networking*, 9(6):733–745, 2001.

[GEN] GENI Project Office. The Global Environment for Network Innovations (GENI). `http://www.geni.net`.

[Geo] George Karypis Lab. The METIS Graph Partitioner. `http://www.cs.umn.edu/~metis/`.

[GF09] Yan Gu and Richard Fujimoto. Performance evaluation of the rosenet network emulation system. *Simulation*, 85(5):319–333, 2009.

[Gio09] Toni Giorgino. Computing and visualizing dynamic time warping alignments in r: The dtw package. *Journal of Statistical Software*, 31(7):1–24, 8 2009.

[GP01] Timothy G. Griffin and Brian J. Premore. An experimental analysis of bgp convergence time. In *International Conference on Network Protocols (ICNP)*, pages 53–61, 2001.

[GRL05] Shashi Guruprasad, Robert Ricci, and Jay Lepreau. Integrated network experimentation using simulation and emulation. In *Proceedings of the 1st International Conference on Testbeds and Research Infrastructures for the DEvelopment of NeTworks and COMmunities (TRIDENTCOM'05)*, pages 204–212, February 2005.

[Gut84] Antomn Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[HBH$^+$03] Michael Heroux, Roscoe Bartlett, Vicki Howle Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric

Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An overview of trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[HBH$^+$05] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[HDSM05] Matthias Hauswirth, Amer Diwan, Peter F. Sweeney, and Michael C. Mozer. Automating vertical profiling. In *OOPSLA*, pages 281–296, 2005.

[Hee04] Jeffrey Heer. prefuse: a software framework for interactive information visualization. Master's thesis, University of California, Berkeley, 2004.

[HH01] X. W. Huang and J. Heidemann. Minimizing routing state for light-weight network simulation. *9th International Symposium on Modeling, Analysis and Simulation on Computer and Telecommunication Systems (MASCOTS)*, pages 108–116, 2001.

[HSK99] X. W. Huang, R. Sharma, and S. Keshav. The ENTRAPID protocol development environment. In *Proceedings of the IEEE INFOCOM 1999*, pages 1107–1115, March 1999.

[HYY$^+$03] A. Hiromori, H. Yamaguchi, K. Yasumoto, T. Higashino, and K. Taniguchi. Reducing the size of routing tables for large-scale network simulation. In *Proceedings of the 17th Workshop on Parallel and Distributed Simulation*, 2003.

[Jai91] Raj Jain. *The Art of Computer Systems Performance Analysis: techniques for experimental design, measurement, simulation, and modeling*. Wiley, 1991.

[Jam] James H. Cowie. Scalable Simulation Framework API Reference Manual. `http://www.ssfnet.org/SSFdocs/ssfapiManual.pdf`.

[Jef85] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, 1985.

[Jyt] Jython Project. Jython: Python for the Java Platform. `http://www.jython.org/`.

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

[Kun] Kunihiro Ishiguro. GNU Zebra. `http://www.gnu.org/software/zebra/`.

[LC04] Xin Liu and Andrew A. Chien. Realistic large-scale online network simulation. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 31, Washington, DC, USA, 2004. IEEE Computer Society.

[LCK$^+$10] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani. Kronecker graphs: an approach to modeling networks. *Journal of Machine Learning Research (JMLR)*, 11:985–1042, Feb 2010.

[Lin] Linux Virtual Server Project. IP Virtual Server (IPVS). `http://kb.linuxvirtualserver.org/wiki/IPVS`.

[Liu08] Jason Liu. A primer for real-time simulation of large-scale networks. In *ANSS-41 '08: Proceedings of the 41st Annual Simulation Symposium (anss-41 2008)*, pages 85–94, Washington, DC, USA, 2008. IEEE Computer Society.

[LL08] Jason Liu and Yue Li. On the performance of a hybrid network traffic model. *Simulation Modelling Practice and Theory*, 16(6):656–669, 2008.

[LLN$^+$05] Michael Liljenstam, Jason Liu, David M. Nicol, Yougu Yuan, Guanhua Yan, and Chris Grier. RINSE: the real-time interactive network simulation environment for network security exercises. In *Proceedings of the 19th Workshop on Parallel and Distributed Simulation (PADS'05)*, pages 119–128, June 2005.

[LN] Jason Liu and David M. Nicol. Dartmouth Scalable Simulation Framework (DaSSF). `http://www.cis.fiu.edu/~liux/research/projects/dassf/index.html`.

[LN04] M. Liljenstam and D. Nicol. On-demand computation of policy based routes for large-scale network simulation. In *Proceedings of the 2004 Winter Simulation Conference*, pages 215–223, 2004.

[LXC03] Xin Liu, Huaxia Xia, and Andrew A. Chien. Network emulation tools for modeling grid behavior. In *Proceedings of 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid'03)*, May 2003.

[LXC04] Xin Liu, Huaxia Xia, and Andrew A. Chien. Validating and scaling the micrgrid: A scientific instrument for grid dynamics. *J. Grid Comput.*, 2(2):141–161, 2004.

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

[MHK$^+$07] Priya Mahadevan, Calvin Hubble, Dmitri Krioukov, Bradley Huffaker, and Amin Vahdat. Orbis: rescaling degree correlations to generate annotated Internet topologies. *ACM SIGCOMM Computer Communication Review*, 37(4):325–336, 2007.

[MJ92] Steven Mccanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. pages 259–269, 1992.

[MLMB01] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. Brite: An approach to universal topology generation. In *Proceedings of the 9th Annual International Symposium on Modeling, Analysis and Simulation on Computer and Telecommunication Systems (MASCOTS)*, 2001.

[Mod] Modeling and Networking Research Group. PRIME Project. `http://www.primessf.net/bin/view/Public/PRIMEProject`.

[MQWZ05] Z. M. Mao, L. Qiu, J. Wang, and Y. Zhang. On as-level path inference. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 339–349, 2005.

[Nic] David Nicol. DARPA NMS baseline network topology. `http://www.ssfnet.org/Exchange/gallery/baseline/index.html`.

[Nic98] David M. Nicol. Scalability, locality, partitioning and synchronization pdes. In *PADS '98: Proceedings of the twelfth workshop on Parallel and distributed simulation*, pages 5–11, Washington, DC, USA, 1998. IEEE Computer Society.

[NLLY03] David M. Nicol, Jason Liu, Michael Liljenstam, and Guanhua Yan. Simulation of large scale networks i: simulation of large-scale networks using ssf. In *WSC '03: Proceedings of the 35th conference on Winter simulation*, pages 650–657. Winter Simulation Conference, 2003.

[NS-a] NS-2 Project. ns-2. `http://nsnam.isi.edu/nsnam/index.php/Main_Page`.

[NS-b] NS-2 Project. Tips and Statistical Data for Running Large Simulations in NS. `http://www.isi.edu/nsnam/ns/ns-largesim.html`.

[NS-c] NS-3 Project. ns-3. `http://www.nsnam.org/index.html`.

[Opea] OpenVZ Project. Linux Containers. `http://openvz.org`.

[Opeb] OpenVZ Project. OpenVZ Virtual Ethernet Device. `http://wiki.openvz.org/Veth`.

[OPN] `http://www.opnet.org`.

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

[Pax96] Vern Paxson. End-to-end routing behavior in the internet. In *Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '96, pages 25–38, New York, NY, USA, 1996. ACM.

[PF97] V. Paxson and S. Floyd. Why we don't know how to simulate the internet. In *Proceedings of the 1997 Winter Simulation Conference (WSC)*, 1997.

[PNL96] B. J. Premore, D. M. Nicol, and X. Liu. A critique of the telecommunications descripting language (ted). Technical Report PCS-TR96-299, Department of Computer Science, Dartmouth College, 1996.

[POF98] Kalyan Perumalla, Andy Ogielski, and Richard Fujimoto. TeD – a language for modeling telecommunication networks. *ACM SIGMETRICS Performance Evaluation Review*, 25(4):4–11, March 1998.

[Pri10] http://groups.geni.net/geni/wiki/PrimoGENI, 2010.

[QU05] Bruno Quoitin and Steve Uhlig. Modeling the routing of an autonomous system with C-BGP. *IEEE Network*, 19(6), 2005.

[RA02] George F. Riley and Mostafa H. Aremar. Simulating large networks - how big is big enough? In *Conference on Grand Challenges for Modeling and Sim.*, 2002.

[RAF00] George F. Riley, Mostafa H. Ammar, and Richard Fujimoto. Stateless routing in network simulations. In *Proceedings of the 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 524–531, 2000.

[RAZ01] George F. Riley, Mostafa H. Ammar, and Ellen W. Zegura. Efficient routing using nix-vectors. In *Proceedings of IEEE Workshop on High Performance Switching and Routing (HPSR)*, 2001.

[Ren] Renesys. SSF Research Network. http://www.ssfnet.org/.

[RFA99] George F. Riley, Richard Fujimoto, and Mostafa H. Ammar. A generic framework for parallelization of network simulations. In *Proceedings of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'99)*, pages 128–135, 1999.

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

[Ril03] George F. Riley. The georgia tech network simulator. In *MoMeTools '03: Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, pages 5–12, New York, NY, USA, 2003. ACM.

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

[RJFA04] George F. Riley, Talal M. Jaafar, Richard M. Fujimoto, and Mostafa H. Ammar. Space-parallel network simulations using ghosts. *Parallel and Distributed Simulation, Workshop on*, 0:170–177, 2004.

[RSO$^+$05] D. Raychaudhuri, I. Seskar, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu, and M. Singh. Overview of the ORBIT radio grid testbed for evaluation of next-generation wireless network protocols. In *Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC 2005)*, 2005.

[SDRL09] Pramod Sanaga, Jonathon Duerig, Robert Ricci, and Jay Lepreau. Modeling and emulation of internet paths. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*, NSDI'09, pages 199–212, Berkeley, CA, USA, 2009. USENIX Association.

[SLJ$^+$00] H. J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, and A. Chien. The micrgrid: a scientific tool for modeling computational grids. In *Proceedings of IEEE Supercomputing (SC 2000*, pages 4–10, 2000.

[SMWA04] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring ISP topologies with Rocketfuel. *IEEE/ACM Transactions on Networking*, 12(1):2–16, 2004.

[SN05] J.E. Smith and Ravi Nair. The architecture of virtual machines. *Computer*, 38(5):32 – 38, may. 2005.

[SPBP06] Neil Spring, Larry Peterson, Andy Bavier, and Vivek Pai. Using PlanetLab for network research: myths, realities, and best practices. *ACM SIGOPS Operating Systems Review*, 40(1):17–24, 2006.

[Ste] Stephane Apiou. Union filesystem for FUSE (FunionFS). `http://funionfs.apiou.org/`.

[Var04] Srinidhi Varadarajan. The weaves runtime framework. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 197, april 2004.

[VH08] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Simutools '08: Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, pages 1–10, ICST, Brussels, Belgium,

Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[VYW$^+$02] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeff Chase, and David Becker. Scalability and accuracy in a large scale network emulator. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, 2002.

[WCH$^+$03] S. Y. Wang, C. L. Chou, C. H. Huang, C. C. Hwang, Z. M. Yang, C. C. Chiou, and C. C. Lin. The design and implementation of the nctuns 1.0 network simulator. *Comput. Netw.*, 42(2):175–197, 2003.

[WLS$^+$02] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, pages 255–270, 2002.

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: A scalable isolation kernel. In *Proceedings of the Tenth ACM SIGOPS European Workshop*, 2002.

[YBB$^+$03] Garrett R. Yaun, David Bauer, Harshad L. Bhutada, Christopher D. Carothers, Murat Yuksel, and Shivkumar Kalyanaraman. Large-scale network simulation techniques: examples of TCP and OSPF models. *ACM SIGCOMM Computer Communication Review*, 33(3):27–41, 2003.

[ZJTB04] Junlan Zhou, Zhengrong Ji, Mineo Takai, and Rajive Bagrodia. MAYA: integrating hybrid network modeling to the physical world. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 14(2):149–169, 2004.

VITA

Nathanael Van Vorst

March 14, 1981	Born, Denver, Colorado
1999–2003	B.S., Electrical Engineering & Computer Science Colorado State University Fort Collins, Colorado
2005–2007	M.S, Computer Science Colorado School of Mines Golden, Colorado
2008–2012	Doctoral Candidate Florida International University Miami, Florida

PUBLICATIONS

J. Liu, S. Mann, N. Van Vorst, and K. Hellman, *An open and scalable emulation infrastructure for large-scale real-time network simulations.* INFOCOM 2007 MiniSymposium, Anchorage, Alaska, May 6-12, 2007.

J. Liu, Y. Li, N. Van Vorst, S. Mann, and K. Hellman, *A real-time network simulation infrastructure based on OpenVPN.* Journal of System Software (2009), pg 473-485.

N. Van Vorst, K. Pedretti, and R. Oldfield, *Super-Scale Real-Time Network Simulation on the Cray XT.* CSRI Summer Proceedings 2010. SAND Report, pg 309-320.

N. Van Vorst, T. Li, and J. Liu, *How Low Can You Go? Spherical Routing for Scalable Network Simulations.* The 19th Annual Meeting of the IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, Raffles Hotel, Singapore, July 25-27, 2011.

N. Van Vorst, M. Erazo, and J. Liu. *PrimoGENI: Integrating Real-Time Network Simulation and Emulation in GENI.* 25th Workshop on Principles of Advanced and Distributed Simulation, Nice, France, June 14-17, 2011.

T. Li, N. Van Vorst, R. Rong, J. Liu. *Simulation Studies of OpenFlow-Based In-Network Caching Strategies.* SpringSim (CNS) 2012, Orlando, FL, March 26-29, 2012.

N. Van Vorst, J. Liu, *Realizing Large-Scale Interactive Network Simulation via Model Splitting*. 26th Workshop on Principles of Advanced and Distributed Simulation, Zhangjiajie, China, July 15-19, 2012.

N. Van Vorst, M. Erazo, and J. Liu. *PrimoGENI for Hybrid Network Simulation and Emulation Experiments in GENI*. Journal of Simulation. To appear.

T. Li, N. Van Vorst, J. Liu. *A Fast Rate-Based TCP Traffic Model*. Submitted.

PRESENTATIONS

N. Van Vorst and M. Erazo. *PrimoGENI Demonstration*. 7th GENI Engineering Conference, Durham, NC, March 16-18, 2010.

J. Liu, N. Van Vorst and M. Erazo. *PrimoGENI Tutorial*. 10th GENI Engineering Conference, San Juan, Puerto Rico, March 15-17, 2011.

N. Van Vorst. *PrimoGENI Poster*. 11th GENI Engineering Conference, Denver, CO, July 26-28, 2011.

N. Van Vorst. *Graduate Student Seminar*. School of Computing and Information Sciences, Florida International Univerisity, Miami, FL, Feburary 9, 2012.

J. Liu, N. Van Vorst and M. Erazo. *PrimoGENI Tutorial and Plenary Demonstration*. 13th GENI Engineering Conference, Los Angeles, CA, March 13-15, 2012.